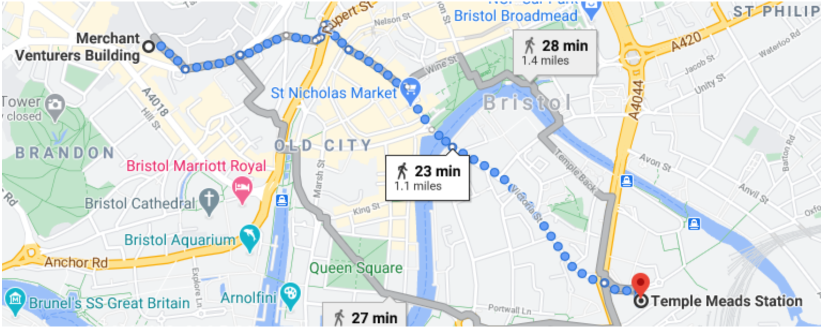
Dijkstra's algorithm
COMS20010 (Algorithms II)

John Lapinskas, University of Bristol

Distances in real networks are weighted!

We often model road networks as graphs: junctions and destinations are vertices, roads are edges, one-way roads are directed edges.

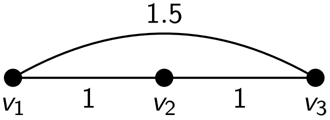


But when we want to find a "shortest path" in this graph, we don't care about the number of edges, we care about the **physical distance**.

(We may also want to weight by e.g. elevation changes or current traffic.)

Weighted graphs

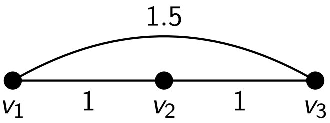
A **weighted graph** is a pair (G, w), where G is a graph and $w \colon E(G) \to \mathbb{R}$ is a **weight function**. This could represent distances, costs, times, etc.



$$w(v_1, v_2) = 1$$
$$w(v_2, v_3) = 1$$
$$w(v_1, v_3) = 1.5$$

Weighted graphs

A **weighted graph** is a pair (G, w), where G is a graph and $w\colon E(G) \to \mathbb{R}$ is a **weight function**. This could represent distances, costs, times, etc.



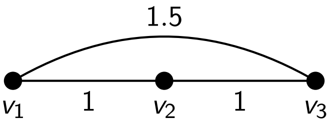
$$w(v_1, v_2) = 1$$
$$w(v_2, v_3) = 1$$
$$w(v_1, v_3) = 1.5$$

The **length** of a path/walk $P = x_1 \dots x_t$ is the total weight of P's edges:

$$\mathrm{length}(P) = \sum_{i=1}^{t-1} w(x_i, x_{i+1}).$$

Weighted graphs

A **weighted graph** is a pair (G, w), where G is a graph and $w\colon E(G) \to \mathbb{R}$ is a **weight function**. This could represent distances, costs, times, etc.



$$w(v_1, v_2) = 1$$
$$w(v_2, v_3) = 1$$
$$w(v_1, v_3) = 1.5$$

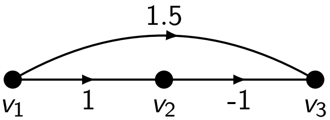
The **length** of a path/walk $P = x_1 \dots x_t$ is the total weight of P's edges:

$$\mathrm{length}(P) = \sum_{i=1}^{t-1} w(x_i, x_{i+1}).$$

The **distance** from x to y is the shortest length of any path/walk from x to y, or ∞ if they are in different components. E.g. $d(v_1, v_3) = 1.5$.

Negative-weight edges

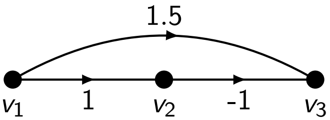
For some applications, it can make sense to allow edges to have **negative weight**. (E.g. costs versus profits...) This can be counterintuitive!



Here, $d(v_1, v_3) =$

Negative-weight edges

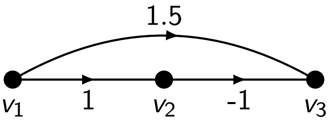
For some applications, it can make sense to allow edges to have **negative weight**. (E.g. costs versus profits...) This can be counterintuitive!



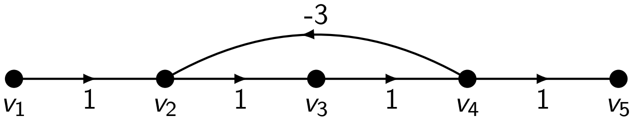
Here, $d(v_1, v_3) = \mathbf{0}$, since $v_1 v_2 v_3$ has cost $w(v_1, v_2) + w(v_2, v_3) = 0$.

Negative-weight edges

For some applications, it can make sense to allow edges to have **negative weight**. (E.g. costs versus profits...) This can be counterintuitive!

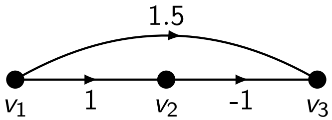


Here, $d(v_1, v_3) = \mathbf{0}$, since $v_1 v_2 v_3$ has cost $w(v_1, v_2) + w(v_2, v_3) = 0$.

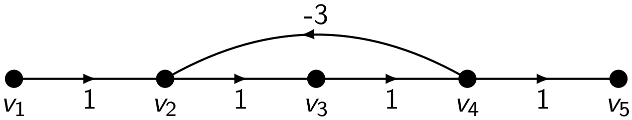


Negative-weight edges

For some applications, it can make sense to allow edges to have **negative weight**. (E.g. costs versus profits...) This can be counterintuitive!



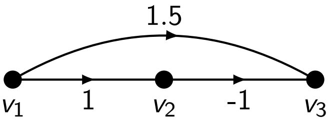
Here, $d(v_1, v_3) = \mathbf{0}$, since $v_1 v_2 v_3$ has cost $w(v_1, v_2) + w(v_2, v_3) = 0$.



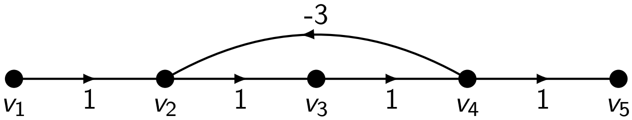
Here, "distance" doesn't even make sense — there are walks from v_1 to v_5 with **arbitrarily low** length. E.g. $\mathrm{length}(v_1 v_2 v_3 v_4 v_5) = 4$.

Negative-weight edges

For some applications, it can make sense to allow edges to have **negative weight**. (E.g. costs versus profits...) This can be counterintuitive!



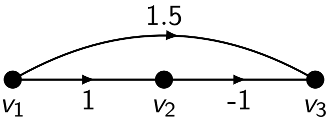
Here, $d(v_1, v_3) = \mathbf{0}$, since $v_1 v_2 v_3$ has cost $w(v_1, v_2) + w(v_2, v_3) = 0$.



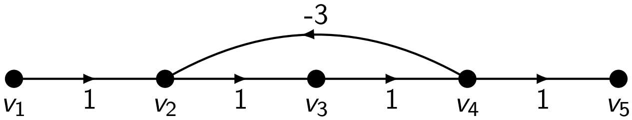
Here, "distance" doesn't even make sense — there are walks from v_1 to v_5 with **arbitrarily low** length. E.g. $\mathrm{length}(v_1 v_2 v_3 v_4 \boldsymbol{v_2 v_3 v_4} v_5) = \mathbf{3}$.

Negative-weight edges

For some applications, it can make sense to allow edges to have **negative weight**. (E.g. costs versus profits...) This can be counterintuitive!



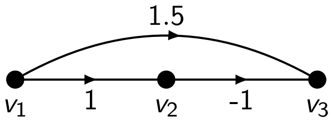
Here, $d(v_1, v_3) = \mathbf{0}$, since $v_1 v_2 v_3$ has cost $w(v_1, v_2) + w(v_2, v_3) = 0$.



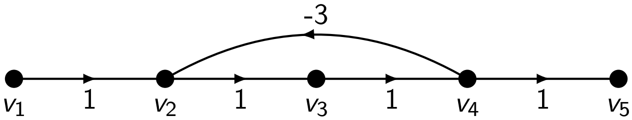
Here, "distance" doesn't even make sense — there are walks from v_1 to v_5 with **arbitrarily low** length. E.g. $\mathrm{length}(v_1 v_2 v_3 v_4 v_2 v_3 v_4 \boldsymbol{v_2 v_3 v_4} v_5) = \mathbf{2}$...

Negative-weight edges

For some applications, it can make sense to allow edges to have **negative weight**. (E.g. costs versus profits...) This can be counterintuitive!



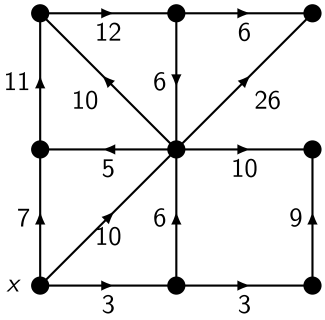
Here, $d(v_1, v_3) = \mathbf{0}$, since $v_1 v_2 v_3$ has cost $w(v_1, v_2) + w(v_2, v_3) = 0$.



Here, "distance" doesn't even make sense — there are walks from v_1 to v_5 with **arbitrarily low** length. E.g. $\mathrm{length}(v_1 v_2 v_3 v_4 v_2 v_3 v_4 \boldsymbol{v_2 v_3 v_4} v_5) = \mathbf{2}$...

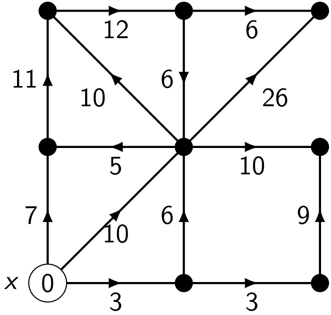
This lecture, we ignore negative weights. (This is also faster!)

Dijkstra's algorithm: The idea



Think of breadth-first search as water flooding a set of pipes, starting from x... and now allow the pipes to have **different lengths**.

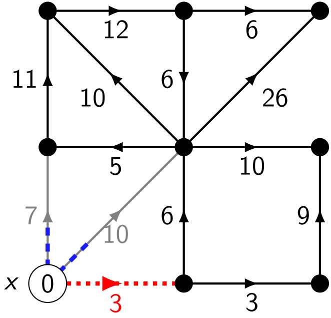
When the water first reaches a vertex v, you know $d(x, v)$ and a shortest path from x to v.



Think of breadth-first search as water flooding a set of pipes, starting from x... and now allow the pipes to have **different lengths**.

When the water first reaches a vertex v, you know $d(x, v)$ and a shortest path from x to v.

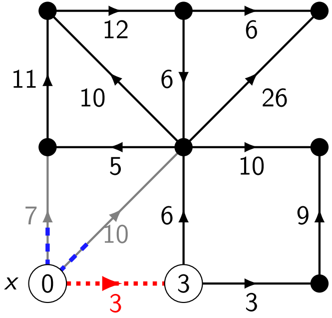
Dijkstra's algorithm: The idea



Think of breadth-first search as water flooding a set of pipes, starting from x... and now allow the pipes to have **different lengths**.

When the water first reaches a vertex v, you know $d(x, v)$ and a shortest path from x to v.

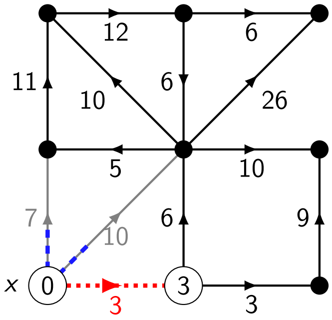
Dijkstra's algorithm: The idea



Think of breadth-first search as water flooding a set of pipes, starting from x... and now allow the pipes to have **different lengths**.

When the water first reaches a vertex v, you know $d(x,v)$ and a shortest path from x to v.

Dijkstra's algorithm: The idea



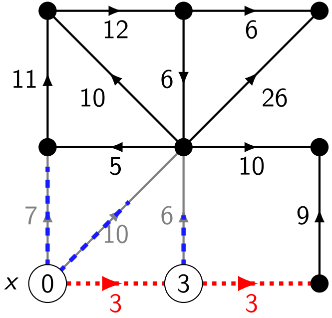
Think of breadth-first search as water flooding a set of pipes, starting from x... and now allow the pipes to have **different lengths**.

When the water first reaches a vertex v, you know $d(x, v)$ and a shortest path from x to v.

At each stage, pick an edge (u, v) with $d(x, u)$ known and $d(x, v)$ unknown that minimises $d(x, u) + \mathrm{length}(u, v)$. (Break ties arbitrarily.)

Then set $d(x, v) = d(x, u) + \mathrm{length}(u, v)$.

Dijkstra's algorithm: The idea



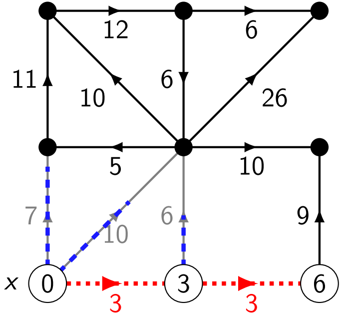
Think of breadth-first search as water flooding a set of pipes, starting from x... and now allow the pipes to have **different lengths**.

When the water first reaches a vertex v, you know $d(x, v)$ and a shortest path from x to v.

At each stage, pick an edge (u, v) with $d(x, u)$ known and $d(x, v)$ unknown that minimises $d(x, u) + \mathrm{length}(u, v)$. (Break ties arbitrarily.)

Then set $d(x, v) = d(x, u) + \mathrm{length}(u, v)$.

Dijkstra's algorithm: The idea

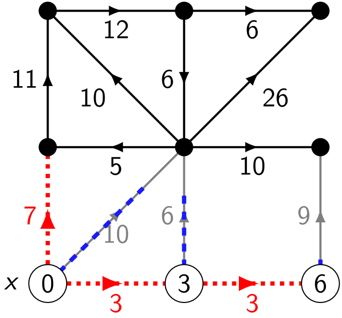


Think of breadth-first search as water flooding a set of pipes, starting from x... and now allow the pipes to have **different lengths**.

When the water first reaches a vertex v, you know $d(x, v)$ and a shortest path from x to v.

At each stage, pick an edge (u, v) with $d(x, u)$ known and $d(x, v)$ unknown that minimises $d(x, u) + \mathrm{length}(u, v)$. (Break ties arbitrarily.)

Then set $d(x, v) = d(x, u) + \mathrm{length}(u, v)$.



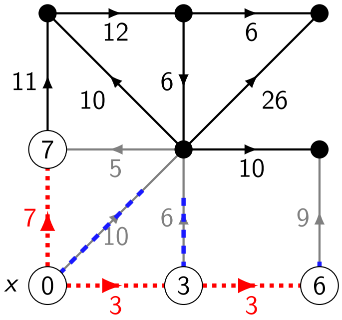
Think of breadth-first search as water flooding a set of pipes, starting from x... and now allow the pipes to have **different lengths**.

When the water first reaches a vertex v, you know $d(x, v)$ and a shortest path from x to v.

At each stage, pick an edge (u, v) with $d(x, u)$ known and $d(x, v)$ unknown that minimises $d(x, u) + \mathrm{length}(u, v)$. (Break ties arbitrarily.)

Then set $d(x, v) = d(x, u) + \mathrm{length}(u, v)$.

Dijkstra's algorithm: The idea

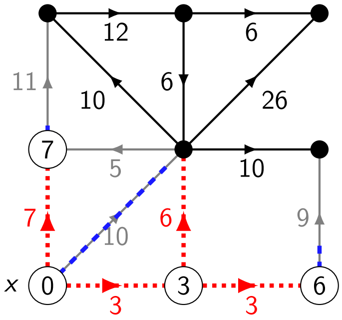


Think of breadth-first search as water flooding a set of pipes, starting from x... and now allow the pipes to have **different lengths**.

When the water first reaches a vertex v, you know $d(x, v)$ and a shortest path from x to v.

At each stage, pick an edge (u, v) with $d(x, u)$ known and $d(x, v)$ unknown that minimises $d(x, u) + \mathrm{length}(u, v)$. (Break ties arbitrarily.)

Then set $d(x, v) = d(x, u) + \mathrm{length}(u, v)$.



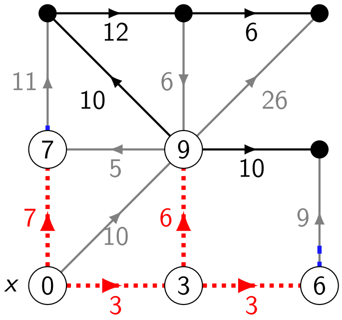
Think of breadth-first search as water flooding a set of pipes, starting from x... and now allow the pipes to have **different lengths**.

When the water first reaches a vertex v, you know $d(x, v)$ and a shortest path from x to v.

At each stage, pick an edge (u, v) with $d(x, u)$ known and $d(x, v)$ unknown that minimises $d(x, u) + \mathrm{length}(u, v)$. (Break ties arbitrarily.)

Then set $d(x, v) = d(x, u) + \mathrm{length}(u, v)$.

Dijkstra's algorithm: The idea



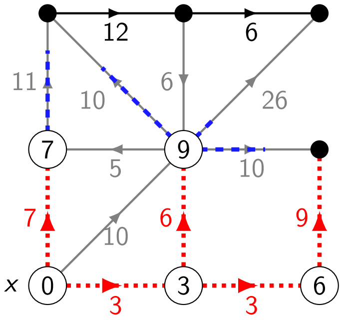
Think of breadth-first search as water flooding a set of pipes, starting from x... and now allow the pipes to have **different lengths**.

When the water first reaches a vertex v, you know $d(x, v)$ and a shortest path from x to v.

At each stage, pick an edge (u, v) with $d(x, u)$ known and $d(x, v)$ unknown that minimises $d(x, u) + \mathrm{length}(u, v)$. (Break ties arbitrarily.)

Then set $d(x, v) = d(x, u) + \mathrm{length}(u, v)$.

Dijkstra's algorithm: The idea



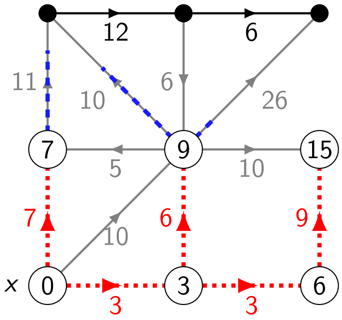
Think of breadth-first search as water flooding a set of pipes, starting from x... and now allow the pipes to have **different lengths**.

When the water first reaches a vertex v, you know $d(x, v)$ and a shortest path from x to v.

At each stage, pick an edge (u, v) with $d(x, u)$ known and $d(x, v)$ unknown that minimises $d(x, u) + \mathrm{length}(u, v)$. (Break ties arbitrarily.)

Then set $d(x, v) = d(x, u) + \mathrm{length}(u, v)$.

Dijkstra's algorithm: The idea

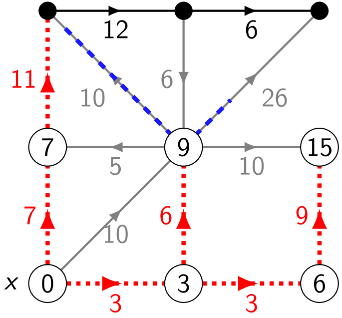


Think of breadth-first search as water flooding a set of pipes, starting from x... and now allow the pipes to have **different lengths**.

When the water first reaches a vertex v, you know $d(x, v)$ and a shortest path from x to v.

At each stage, pick an edge (u, v) with $d(x, u)$ known and $d(x, v)$ unknown that minimises $d(x, u) + \mathrm{length}(u, v)$. (Break ties arbitrarily.)

Then set $d(x, v) = d(x, u) + \mathrm{length}(u, v)$.



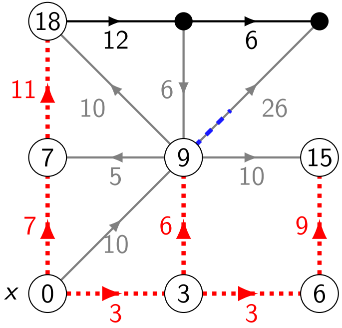
Think of breadth-first search as water flooding a set of pipes, starting from x... and now allow the pipes to have **different lengths**.

When the water first reaches a vertex v, you know $d(x, v)$ and a shortest path from x to v.

At each stage, pick an edge (u, v) with $d(x, u)$ known and $d(x, v)$ unknown that minimises $d(x, u) + \mathrm{length}(u, v)$. (Break ties arbitrarily.)

Then set $d(x, v) = d(x, u) + \mathrm{length}(u, v)$.

Dijkstra's algorithm: The idea



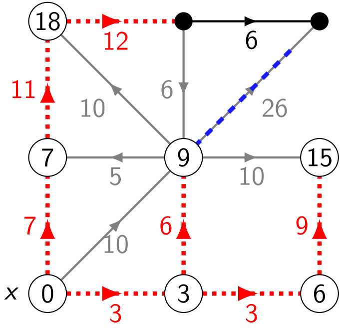
Think of breadth-first search as water flooding a set of pipes, starting from x... and now allow the pipes to have **different lengths**.

When the water first reaches a vertex v, you know $d(x, v)$ and a shortest path from x to v.

At each stage, pick an edge (u, v) with $d(x, u)$ known and $d(x, v)$ unknown that minimises $d(x, u) + \mathrm{length}(u, v)$. (Break ties arbitrarily.)

Then set $d(x, v) = d(x, u) + \mathrm{length}(u, v)$.

Dijkstra's algorithm: The idea



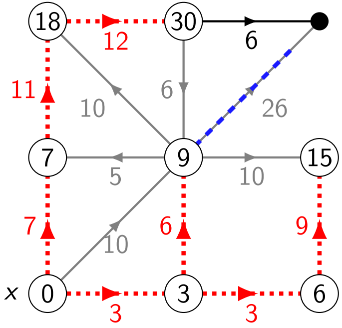
Think of breadth-first search as water flooding a set of pipes, starting from x... and now allow the pipes to have **different lengths**.

When the water first reaches a vertex v, you know $d(x, v)$ and a shortest path from x to v.

At each stage, pick an edge (u, v) with $d(x, u)$ known and $d(x, v)$ unknown that minimises $d(x, u) + \mathrm{length}(u, v)$. (Break ties arbitrarily.)

Then set $d(x, v) = d(x, u) + \mathrm{length}(u, v)$.

Dijkstra's algorithm: The idea



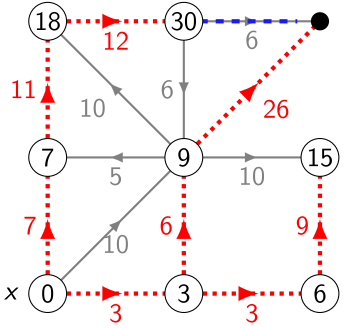
Think of breadth-first search as water flooding a set of pipes, starting from x... and now allow the pipes to have **different lengths**.

When the water first reaches a vertex v, you know $d(x, v)$ and a shortest path from x to v.

At each stage, pick an edge (u, v) with $d(x, u)$ known and $d(x, v)$ unknown that minimises $d(x, u) + \mathrm{length}(u, v)$. (Break ties arbitrarily.)

Then set $d(x, v) = d(x, u) + \mathrm{length}(u, v)$.

Dijkstra's algorithm: The idea

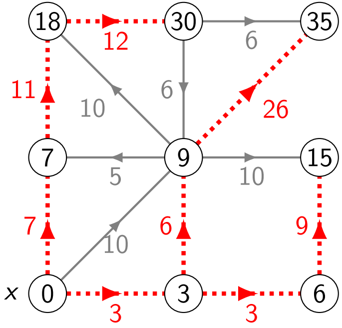


Think of breadth-first search as water flooding a set of pipes, starting from x... and now allow the pipes to have **different lengths**.

When the water first reaches a vertex v, you know $d(x, v)$ and a shortest path from x to v.

At each stage, pick an edge (u, v) with $d(x, u)$ known and $d(x, v)$ unknown that minimises $d(x, u) + \mathrm{length}(u, v)$. (Break ties arbitrarily.)

Then set $d(x, v) = d(x, u) + \mathrm{length}(u, v)$.



Think of breadth-first search as water flooding a set of pipes, starting from x... and now allow the pipes to have **different lengths**.

When the water first reaches a vertex v, you know $d(x, v)$ and a shortest path from x to v.

At each stage, pick an edge (u, v) with $d(x, u)$ known and $d(x, v)$ unknown that minimises $d(x, u) + \mathrm{length}(u, v)$. (Break ties arbitrarily.)

Then set $d(x, v) = d(x, u) + \mathrm{length}(u, v)$.

Dijkstra's algorithm: Correctness

For input graph G, vertex x: At each stage, pick an edge (u, v) with $d(x, u)$ known and $d(x, v)$ unknown that minimises $d(x, u) + \mathrm{length}(u, v)$. Then set $d(x, v) = d(x, u) + \mathrm{length}(u, v)$.

Dijkstra's algorithm: Correctness

For input graph G, vertex x: At each stage, pick an edge (u, v) with $d(x, u)$ known and $d(x, v)$ unknown that minimises $d(x, u) + \mathrm{length}(u, v)$. Then set $d(x, v) = d(x, u) + \mathrm{length}(u, v)$.

Claim: Dijkstra's algorithm calculates distances correctly.

Dijkstra's algorithm: Correctness

For input graph G, vertex x: At each stage, pick an edge (u, v) with $d(x, u)$ known and $d(x, v)$ unknown that minimises $d(x, u) + \mathrm{length}(u, v)$. Then set $d(x, v) = d(x, u) + \mathrm{length}(u, v)$.

Claim: Dijkstra's algorithm calculates distances correctly.

Proof: By induction on the number of vertices u with $d(x, u)$ set.

Dijkstra's algorithm: Correctness

For input graph G, vertex x: At each stage, pick an edge (u, v) with $d(x, u)$ known and $d(x, v)$ unknown that minimises $d(x, u) + \mathrm{length}(u, v)$. Then set $d(x, v) = d(x, u) + \mathrm{length}(u, v)$.

Claim: Dijkstra's algorithm calculates distances correctly.

Proof: By induction on the number of vertices u with $d(x, u)$ set.

Base case: We have $d(x, x) = 0$. ✓

Dijkstra's algorithm: Correctness

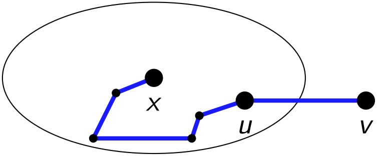
For input graph G, vertex x: At each stage, pick an edge (u, v) with $d(x, u)$ known and $d(x, v)$ unknown that minimises $d(x, u) + \mathrm{length}(u, v)$. Then set $d(x, v) = d(x, u) + \mathrm{length}(u, v)$.

Claim: Dijkstra's algorithm calculates distances correctly.

Proof: By induction on the number of vertices u with $d(x, u)$ set.

Base case: We have $d(x, x) = 0$. ✓

Inductive step: Suppose we know $d(x, u)$ for all $u \in X$, for some set X.
Say Dijkstra's algorithm picks an edge (u, v) with $u \in X$.



Dijkstra's algorithm: Correctness

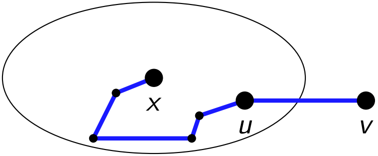
For input graph G, vertex x: At each stage, pick an edge (u, v) with $d(x, u)$ known and $d(x, v)$ unknown that minimises $d(x, u) + \mathrm{length}(u, v)$. Then set $d(x, v) = d(x, u) + \mathrm{length}(u, v)$.

Claim: Dijkstra's algorithm calculates distances correctly.

Proof: By induction on the number of vertices u with $d(x, u)$ set.

Base case: We have $d(x, x) = 0$. ✓

Inductive step: Suppose we know $d(x, u)$ for all $u \in X$, for some set X. Say Dijkstra's algorithm picks an edge (u, v) with $u \in X$.



We can append (u, v) to any path from x to u, so we have

$$d(x, v) \leq d(x, u) + \mathrm{length}(u, v).$$

Dijkstra's algorithm: Correctness

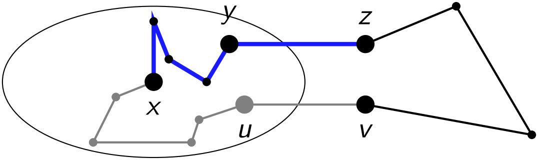
For input graph G, vertex x: At each stage, pick an edge (u, v) with $d(x, u)$ known and $d(x, v)$ unknown that minimises $d(x, u) + \mathrm{length}(u, v)$. Then set $d(x, v) = d(x, u) + \mathrm{length}(u, v)$.

Claim: Dijkstra's algorithm calculates distances correctly.

Proof: By induction on the number of vertices u with $d(x, u)$ set.

Base case: We have $d(x, x) = 0$. ✓

Inductive step: Suppose we know $d(x, u)$ for all $u \in X$, for some set X.
Say Dijkstra's algorithm picks an edge (u, v) with $u \in X$.



Also, any path P from x to v has to leave X on some edge (y, z).

Dijkstra's algorithm: Correctness

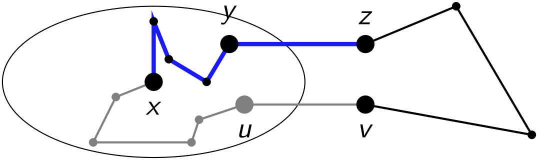
For input graph G, vertex x: At each stage, pick an edge (u, v) with $d(x, u)$ known and $d(x, v)$ unknown that minimises $d(x, u) + \mathrm{length}(u, v)$. Then set $d(x, v) = d(x, u) + \mathrm{length}(u, v)$.

Claim: Dijkstra's algorithm calculates distances correctly.

Proof: By induction on the number of vertices u with $d(x, u)$ set.

Base case: We have $d(x, x) = 0$. ✓

Inductive step: Suppose we know $d(x, u)$ for all $u \in X$, for some set X.
Say Dijkstra's algorithm picks an edge (u, v) with $u \in X$.



Also, any path P from x to v has to leave X on some edge (y, z). Hence P has length at least $d(x, y) + \mathrm{length}(y, z)$. So from the way we picked (u, v), we have $d(x, v) \geq d(x, u) + \mathrm{length}(u, v)$. □

Priority queues

We need a **priority queue** (see COMS10007) to implement this efficiently.

Not like a normal queue: each element has a **priority**, and the "first" element is the one with the **lowest** priority (breaking ties **arbitrarily**).

Relevant operations:

- `StartQueue`(n) returns a new priority queue of maximum length n.
- `Insert`(x, p) inserts a new element x with priority p.
- `Extract`$()$ removes and returns the lowest-priority element.
- `ChangeKey`(x, p) updates the priority of x to p.

`StartQueue` takes $O(n)$ time, all other operations take $O(\log n)$ time.

Priority queues

We need a **priority queue** (see COMS10007) to implement this efficiently.

Not like a normal queue: each element has a **priority**, and the “first” element is the one with the **lowest** priority (breaking ties **arbitrarily**).

Relevant operations:

- `StartQueue`(n) returns a new priority queue of maximum length n.
- `Insert`(x, p) inserts a new element x with priority p.
- `Extract`() removes and returns the lowest-priority element.
- `ChangeKey`(x, p) updates the priority of x to p.

`StartQueue` takes $O(n)$ time, all other operations take $O(\log n)$ time.

```
StartQueue(5);
```

Priority queues

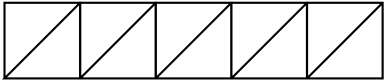
We need a **priority queue** (see COMS10007) to implement this efficiently.

Not like a normal queue: each element has a **priority**, and the "first" element is the one with the **lowest** priority (breaking ties **arbitrarily**).

Relevant operations:

- `StartQueue`(n) returns a new priority queue of maximum length n.
- `Insert`(x, p) inserts a new element x with priority p.
- `Extract`$()$ removes and returns the lowest-priority element.
- `ChangeKey`(x, p) updates the priority of x to p.

`StartQueue` takes $O(n)$ time, all other operations take $O(\log n)$ time.



`StartQueue(5);`

Priority queues

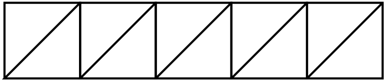
We need a **priority queue** (see COMS10007) to implement this efficiently.

Not like a normal queue: each element has a **priority**, and the "first" element is the one with the **lowest** priority (breaking ties **arbitrarily**).

Relevant operations:

- `StartQueue`(n) returns a new priority queue of maximum length n.
- `Insert`(x, p) inserts a new element x with priority p.
- `Extract`$()$ removes and returns the lowest-priority element.
- `ChangeKey`(x, p) updates the priority of x to p.

`StartQueue` takes $O(n)$ time, all other operations take $O(\log n)$ time.



`StartQueue`(5); `Insert`$(v_1, 0)$;

Priority queues

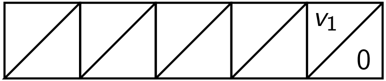
We need a **priority queue** (see COMS10007) to implement this efficiently.

Not like a normal queue: each element has a **priority**, and the "first" element is the one with the **lowest** priority (breaking ties **arbitrarily**).

Relevant operations:

- `StartQueue`(n) returns a new priority queue of maximum length n.
- `Insert`(x, p) inserts a new element x with priority p.
- `Extract`$()$ removes and returns the lowest-priority element.
- `ChangeKey`(x, p) updates the priority of x to p.

`StartQueue` takes $O(n)$ time, all other operations take $O(\log n)$ time.



`StartQueue`(5); `Insert`$(v_1, 0)$;

Priority queues

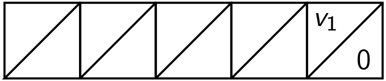
We need a **priority queue** (see COMS10007) to implement this efficiently.

Not like a normal queue: each element has a **priority**, and the "first" element is the one with the **lowest** priority (breaking ties **arbitrarily**).

Relevant operations:

- `StartQueue`(n) returns a new priority queue of maximum length n.
- `Insert`(x, p) inserts a new element x with priority p.
- `Extract`$()$ removes and returns the lowest-priority element.
- `ChangeKey`(x, p) updates the priority of x to p.

`StartQueue` takes $O(n)$ time, all other operations take $O(\log n)$ time.



`StartQueue`(5); `Insert`$(v_1, 0)$; `Insert`$(v_2, 4)$;

Priority queues

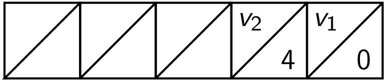
We need a **priority queue** (see COMS10007) to implement this efficiently.

Not like a normal queue: each element has a **priority**, and the "first" element is the one with the **lowest** priority (breaking ties **arbitrarily**).

Relevant operations:

- `StartQueue`(n) returns a new priority queue of maximum length n.
- `Insert`(x, p) inserts a new element x with priority p.
- `Extract`$()$ removes and returns the lowest-priority element.
- `ChangeKey`(x, p) updates the priority of x to p.

`StartQueue` takes $O(n)$ time, all other operations take $O(\log n)$ time.



`StartQueue`(5); `Insert`$(v_1, 0)$; `Insert`$(v_2, 4)$;

Priority queues

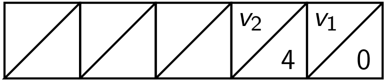
We need a **priority queue** (see COMS10007) to implement this efficiently.

Not like a normal queue: each element has a **priority**, and the "first" element is the one with the **lowest** priority (breaking ties **arbitrarily**).

Relevant operations:

- `StartQueue`(n) returns a new priority queue of maximum length n.
- `Insert`(x, p) inserts a new element x with priority p.
- `Extract`$()$ removes and returns the lowest-priority element.
- `ChangeKey`(x, p) updates the priority of x to p.

`StartQueue` takes $O(n)$ time, all other operations take $O(\log n)$ time.



`Insert`$(v_3, 2)$;

Priority queues

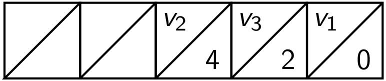
We need a **priority queue** (see COMS10007) to implement this efficiently.

Not like a normal queue: each element has a **priority**, and the "first" element is the one with the **lowest** priority (breaking ties **arbitrarily**).

Relevant operations:

- `StartQueue`(n) returns a new priority queue of maximum length n.
- `Insert`(x, p) inserts a new element x with priority p.
- `Extract`() removes and returns the lowest-priority element.
- `ChangeKey`(x, p) updates the priority of x to p.

`StartQueue` takes $O(n)$ time, all other operations take $O(\log n)$ time.



`Insert`($v_3, 2$);

Priority queues

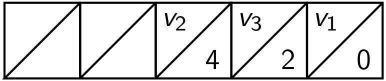
We need a **priority queue** (see COMS10007) to implement this efficiently.

Not like a normal queue: each element has a **priority**, and the "first" element is the one with the **lowest** priority (breaking ties **arbitrarily**).

Relevant operations:

- `StartQueue`(n) returns a new priority queue of maximum length n.
- `Insert`(x, p) inserts a new element x with priority p.
- `Extract`$()$ removes and returns the lowest-priority element.
- `ChangeKey`(x, p) updates the priority of x to p.

`StartQueue` takes $O(n)$ time, all other operations take $O(\log n)$ time.



`Insert`$(v_3, 2)$; `Insert`$(v_5, 3)$;

Priority queues

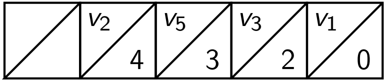
We need a **priority queue** (see COMS10007) to implement this efficiently.

Not like a normal queue: each element has a **priority**, and the "first" element is the one with the **lowest** priority (breaking ties **arbitrarily**).

Relevant operations:

- `StartQueue`(n) returns a new priority queue of maximum length n.
- `Insert`(x, p) inserts a new element x with priority p.
- `Extract`$()$ removes and returns the lowest-priority element.
- `ChangeKey`(x, p) updates the priority of x to p.

`StartQueue` takes $O(n)$ time, all other operations take $O(\log n)$ time.



`Insert`$(v_3, 2)$; `Insert`$(v_5, 3)$;

Priority queues

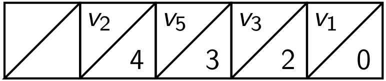
We need a **priority queue** (see COMS10007) to implement this efficiently.

Not like a normal queue: each element has a **priority**, and the "first" element is the one with the **lowest** priority (breaking ties **arbitrarily**).

Relevant operations:

- `StartQueue`(n) returns a new priority queue of maximum length n.
- `Insert`(x, p) inserts a new element x with priority p.
- `Extract`$()$ removes and returns the lowest-priority element.
- `ChangeKey`(x, p) updates the priority of x to p.

`StartQueue` takes $O(n)$ time, all other operations take $O(\log n)$ time.



`Insert`$(v_3, 2)$; `Insert`$(v_5, 3)$; `Extract`$()$;

Priority queues

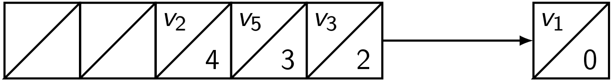
We need a **priority queue** (see COMS10007) to implement this efficiently.

Not like a normal queue: each element has a **priority**, and the "first" element is the one with the **lowest** priority (breaking ties **arbitrarily**).

Relevant operations:

- `StartQueue`(n) returns a new priority queue of maximum length n.
- `Insert`(x, p) inserts a new element x with priority p.
- `Extract`$()$ removes and returns the lowest-priority element.
- `ChangeKey`(x, p) updates the priority of x to p.

`StartQueue` takes $O(n)$ time, all other operations take $O(\log n)$ time.



`Insert`$(v_3, 2)$; `Insert`$(v_5, 3)$; `Extract`$()$;

Priority queues

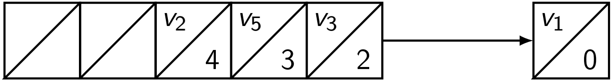
We need a **priority queue** (see COMS10007) to implement this efficiently.

Not like a normal queue: each element has a **priority**, and the "first" element is the one with the **lowest** priority (breaking ties **arbitrarily**).

Relevant operations:

- `StartQueue`(n) returns a new priority queue of maximum length n.
- `Insert`(x, p) inserts a new element x with priority p.
- `Extract`() removes and returns the lowest-priority element.
- `ChangeKey`(x, p) updates the priority of x to p.

`StartQueue` takes $O(n)$ time, all other operations take $O(\log n)$ time.



`Insert`(v_4, 20);

Priority queues

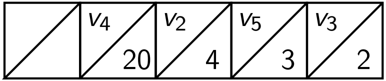
We need a **priority queue** (see COMS10007) to implement this efficiently.

Not like a normal queue: each element has a **priority**, and the "first" element is the one with the **lowest** priority (breaking ties **arbitrarily**).

Relevant operations:

- `StartQueue`(n) returns a new priority queue of maximum length n.
- `Insert`(x, p) inserts a new element x with priority p.
- `Extract`() removes and returns the lowest-priority element.
- `ChangeKey`(x, p) updates the priority of x to p.

`StartQueue` takes $O(n)$ time, all other operations take $O(\log n)$ time.



`Insert`(v_4, 20);

Priority queues

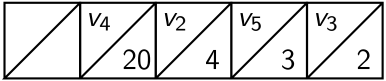
We need a **priority queue** (see COMS10007) to implement this efficiently.

Not like a normal queue: each element has a **priority**, and the "first" element is the one with the **lowest** priority (breaking ties **arbitrarily**).

Relevant operations:

- `StartQueue`(n) returns a new priority queue of maximum length n.
- `Insert`(x, p) inserts a new element x with priority p.
- `Extract`() removes and returns the lowest-priority element.
- `ChangeKey`(x, p) updates the priority of x to p.

`StartQueue` takes $O(n)$ time, all other operations take $O(\log n)$ time.



`Insert`(v_4, 20); `ChangeKey`(v_4, 1);

Priority queues

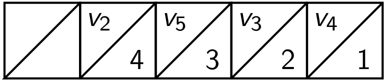
We need a **priority queue** (see COMS10007) to implement this efficiently.

Not like a normal queue: each element has a **priority**, and the “first” element is the one with the **lowest** priority (breaking ties **arbitrarily**).

Relevant operations:

- `StartQueue`(n) returns a new priority queue of maximum length n.
- `Insert`(x, p) inserts a new element x with priority p.
- `Extract`$()$ removes and returns the lowest-priority element.
- `ChangeKey`(x, p) updates the priority of x to p.

`StartQueue` takes $O(n)$ time, all other operations take $O(\log n)$ time.



`Insert`$(v_4, 20)$; `ChangeKey`$(v_4, 1)$;

Priority queues

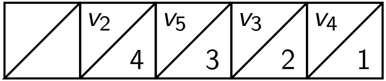
We need a **priority queue** (see COMS10007) to implement this efficiently.

Not like a normal queue: each element has a **priority**, and the "first" element is the one with the **lowest** priority (breaking ties **arbitrarily**).

Relevant operations:

- `StartQueue`(n) returns a new priority queue of maximum length n.
- `Insert`(x, p) inserts a new element x with priority p.
- `Extract`$()$ removes and returns the lowest-priority element.
- `ChangeKey`(x, p) updates the priority of x to p.

`StartQueue` takes $O(n)$ time, all other operations take $O(\log n)$ time.



`Insert`$(v_4, 20)$; `ChangeKey`$(v_4, 1)$; `Extract`$()$;

Priority queues

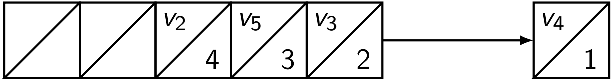
We need a **priority queue** (see COMS10007) to implement this efficiently.

Not like a normal queue: each element has a **priority**, and the "first" element is the one with the **lowest** priority (breaking ties **arbitrarily**).

Relevant operations:

- $\texttt{StartQueue}(n)$ returns a new priority queue of maximum length n.
- $\texttt{Insert}(x, p)$ inserts a new element x with priority p.
- $\texttt{Extract}()$ removes and returns the lowest-priority element.
- $\texttt{ChangeKey}(x, p)$ updates the priority of x to p.

$\texttt{StartQueue}$ takes $O(n)$ time, all other operations take $O(\log n)$ time.



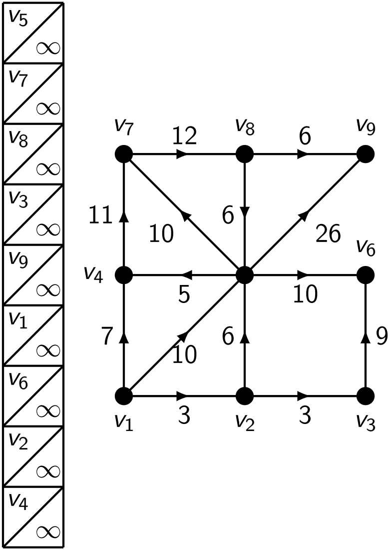
$\texttt{Insert}(v_4, 20); \texttt{ChangeKey}(v_4, 1); \texttt{Extract}();$

Dijkstra's algorithm: Implementation

Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.
Output : $d(v, y)$ for all $y \in V$.

Number the vertices of G as $v = v_1, \dots, v_n$.
`queue` $\leftarrow$ `StartQueue`(n).
foreach $i = 1$ *to* n **do**
    `dist`$[i] \leftarrow \infty$ and call `queue.Insert`(v_i, ∞).
Call `queue.ChangeKey`$(v_1, 0)$.
do
    $v_i \leftarrow$ `queue.Extract`$()$.
    foreach $(v_i, v_j) \in E$ **do**
        `dist`$[j] \leftarrow \min\{$`dist`$[j],$ `dist`$[i] + w(i, j)\}$.
        Call `queue.ChangeKey`$(v_j,$ `dist`$[j])$,
while `queue` *is not empty*
Return `dist`.

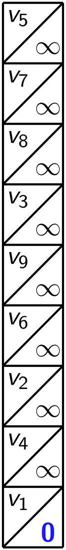


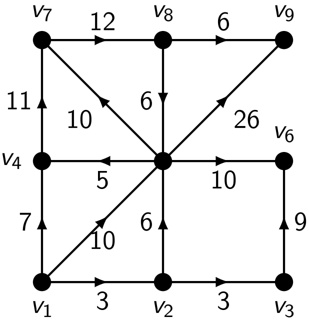
Dijkstra's algorithm: Implementation

Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.
Output : $d(v, y)$ for all $y \in V$.

```
Number the vertices of G as v = v_1, ..., v_n.
queue ← StartQueue(n).
foreach i = 1 to n do
    dist[i] ← ∞ and call queue.Insert(v_i, ∞).
Call queue.ChangeKey(v_1, 0).
do
    v_i ← queue.Extract().
    foreach (v_i, v_j) ∈ E do
        dist[j] ← min{dist[j], dist[i] + w(i, j)}.
        Call queue.ChangeKey(v_j, dist[j]),
while queue is not empty
Return dist.
```

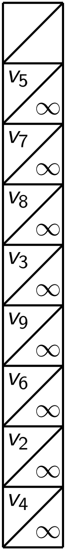


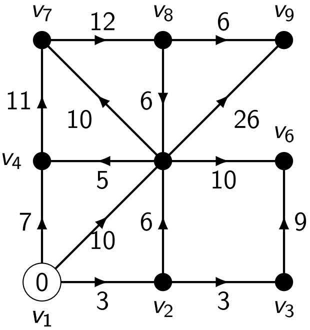


Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.
Output : $d(v, y)$ for all $y \in V$.

Number the vertices of G as $v = v_1, \dots, v_n$.
queue $\leftarrow$ `StartQueue`(n).
foreach $i = 1$ *to* n **do**
  `dist`$[i] \leftarrow \infty$ and call `queue.Insert`(v_i, ∞).
Call `queue.ChangeKey`$(v_1, 0)$.
do
  $v_i \leftarrow$ `queue.Extract()`.
  foreach $(v_i, v_j) \in E$ **do**
    `dist`$[j] \leftarrow \min\{$`dist`$[j]$, `dist`$[i] + w(i, j)\}$.
    Call `queue.ChangeKey`$(v_j,$ `dist`$[j])$,
while queue *is not empty*
Return `dist`.



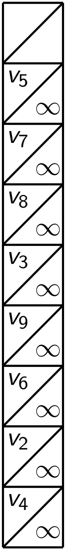


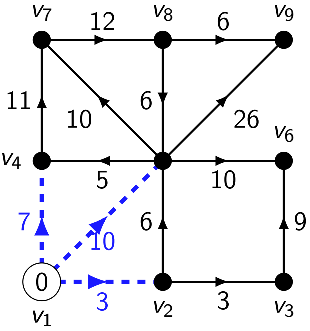
Dijkstra's algorithm: Implementation

Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.
Output : $d(v, y)$ for all $y \in V$.

```
Number the vertices of G as v = v_1, ..., v_n.
queue ← StartQueue(n).
foreach i = 1 to n do
    dist[i] ← ∞ and call queue.Insert(v_i, ∞).
Call queue.ChangeKey(v_1, 0).
do
    v_i ← queue.Extract().
    foreach (v_i, v_j) ∈ E do
        dist[j] ← min{dist[j], dist[i] + w(i, j)}.
        Call queue.ChangeKey(v_j, dist[j]),
while queue is not empty
Return dist.
```

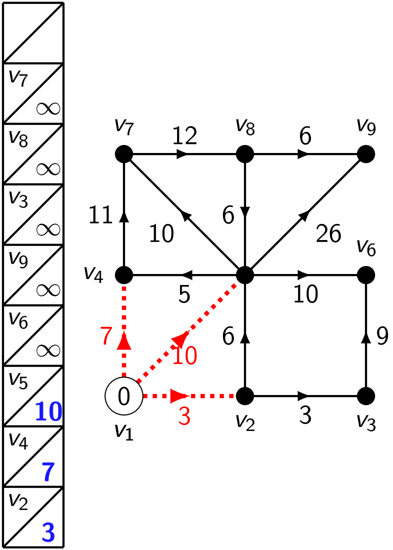




Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.
Output : $d(v, y)$ for all $y \in V$.

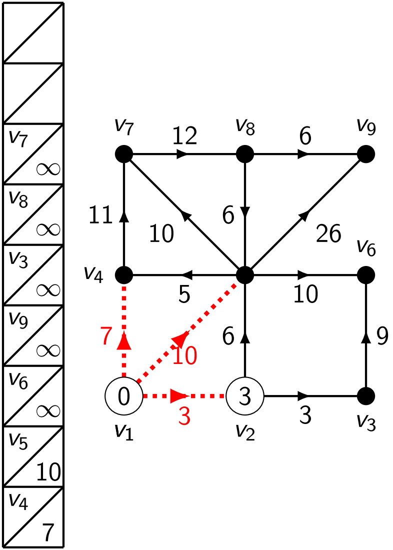
Number the vertices of G as $v = v_1, \dots, v_n$.
queue $\leftarrow$ `StartQueue`(n).
foreach $i = 1$ *to* n **do**
  `dist`$[i] \leftarrow \infty$ and call `queue.Insert`(v_i, ∞).
Call `queue.ChangeKey`$(v_1, 0)$.
do
  $v_i \leftarrow$ `queue.Extract()`.
  foreach $(v_i, v_j) \in E$ **do**
    `dist`$[j] \leftarrow \min\{$`dist`$[j]$, `dist`$[i] + w(i, j)\}$.
    Call `queue.ChangeKey`$(v_j,$ `dist`$[j])$,
while queue *is not empty*
Return `dist`.



Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.
Output : $d(v, y)$ for all $y \in V$.

Number the vertices of G as $v = v_1, \dots, v_n$.
queue $\leftarrow$ `StartQueue`(n).
foreach $i = 1$ *to* n **do**
    `dist`$[i] \leftarrow \infty$ and call `queue.Insert`(v_i, ∞).
Call `queue.ChangeKey`$(v_1, 0)$.
do
    $v_i \leftarrow$ `queue.Extract`$()$.
    foreach $(v_i, v_j) \in E$ **do**
        `dist`$[j] \leftarrow \min\{$`dist`$[j],$ `dist`$[i] + w(i, j)\}$.
        Call `queue.ChangeKey`$(v_j,$ `dist`$[j])$,
while queue *is not empty*
Return `dist`.

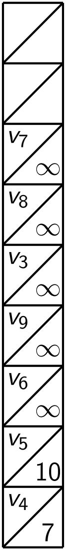


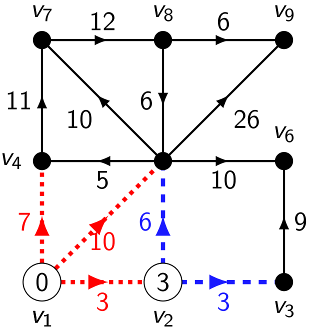
Dijkstra's algorithm: Implementation

Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.
Output : $d(v, y)$ for all $y \in V$.

Number the vertices of G as $v = v_1, \ldots, v_n$.
queue $\leftarrow$ `StartQueue`(n).
foreach $i = 1$ *to* n **do**
    `dist`$[i] \leftarrow \infty$ and call `queue.Insert`(v_i, ∞).
Call `queue.ChangeKey`$(v_1, 0)$.
do
    $v_i \leftarrow$ `queue.Extract()`.
    foreach $(v_i, v_j) \in E$ **do**
        `dist`$[j] \leftarrow \min\{$`dist`$[j]$, `dist`$[i] + w(i, j)\}$.
        Call `queue.ChangeKey`$(v_j$, `dist`$[j])$,
while queue *is not empty*
Return `dist`.



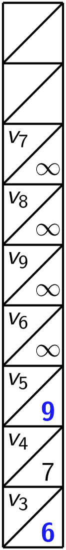


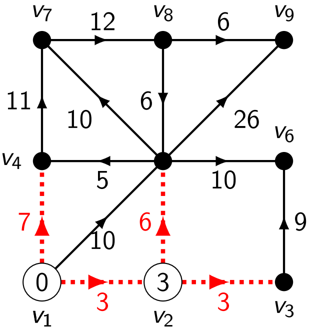
Dijkstra's algorithm: Implementation

Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.
Output : $d(v, y)$ for all $y \in V$.

```
Number the vertices of G as v = v_1, ..., v_n.
queue ← StartQueue(n).
foreach i = 1 to n do
    dist[i] ← ∞ and call queue.Insert(v_i, ∞).
Call queue.ChangeKey(v_1, 0).
do
    v_i ← queue.Extract().
    foreach (v_i, v_j) ∈ E do
        dist[j] ← min{dist[j], dist[i] + w(i, j)}.
        Call queue.ChangeKey(v_j, dist[j]),
while queue is not empty
Return dist.
```



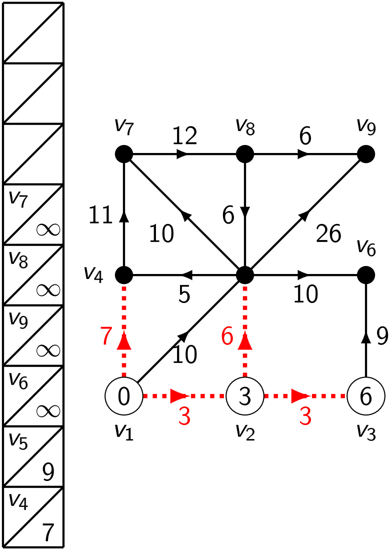


Dijkstra's algorithm: Implementation

Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.
Output : $d(v, y)$ for all $y \in V$.

Number the vertices of G as $v = v_1, \dots, v_n$.
queue $\leftarrow$ `StartQueue`(n).
foreach $i = 1$ *to* n **do**
  `dist`$[i] \leftarrow \infty$ and call `queue.Insert`(v_i, ∞).
Call `queue.ChangeKey`$(v_1, 0)$.
do
  $v_i \leftarrow$ `queue.Extract()`.
  foreach $(v_i, v_j) \in E$ **do**
    `dist`$[j] \leftarrow \min\{$`dist`$[j]$, `dist`$[i] + w(i, j)\}$.
    Call `queue.ChangeKey`$(v_j$, `dist`$[j])$,
while queue *is not empty*
Return `dist`.

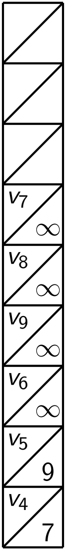


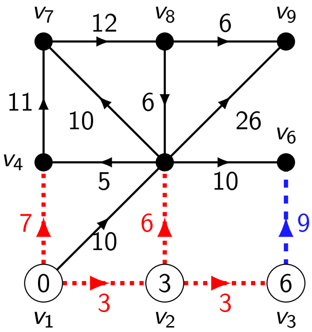
Dijkstra's algorithm: Implementation

Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.
Output : $d(v, y)$ for all $y \in V$.

Number the vertices of G as $v = v_1, \ldots, v_n$.
queue $\leftarrow$ `StartQueue`(n).
foreach $i = 1$ *to* n **do**
    `dist`$[i] \leftarrow \infty$ and call `queue.Insert`(v_i, ∞).
Call `queue.ChangeKey`$(v_1, 0)$.
do
    $v_i \leftarrow$ `queue.Extract`().
    foreach $(v_i, v_j) \in E$ **do**
        `dist`$[j] \leftarrow \min\{$`dist`$[j]$, `dist`$[i] + w(i, j)\}$.
        Call `queue.ChangeKey`$(v_j,$ `dist`$[j])$,
while queue *is not empty*
Return `dist`.



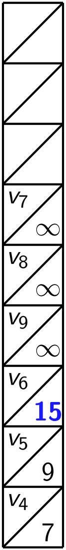


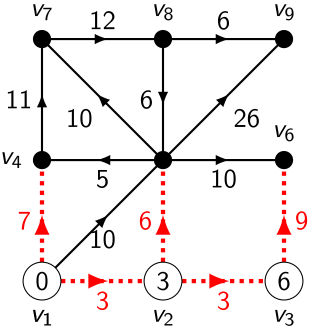
Dijkstra's algorithm: Implementation

Algorithm: Dijkstra

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.
Output : $d(v, y)$ for all $y \in V$.

Number the vertices of G as $v = v_1, \ldots, v_n$.
queue $\leftarrow$ `StartQueue`(n).
foreach $i = 1$ *to* n **do**
    `dist`$[i] \leftarrow \infty$ and call `queue.Insert`(v_i, ∞).
Call `queue.ChangeKey`$(v_1, 0)$.
do
    $v_i \leftarrow$ `queue.Extract()`.
    foreach $(v_i, v_j) \in E$ **do**
        `dist`$[j] \leftarrow \min\{$`dist`$[j]$, `dist`$[i] + w(i, j)\}$.
        Call `queue.ChangeKey`$(v_j,$ `dist`$[j])$,
while queue *is not empty*
Return `dist`.

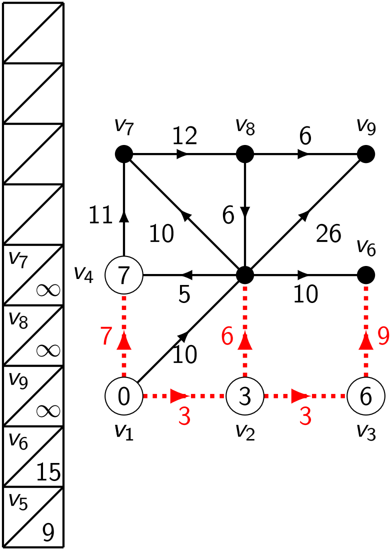




Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.
Output : $d(v, y)$ for all $y \in V$.

Number the vertices of G as $v = v_1, \ldots, v_n$.
queue $\leftarrow$ `StartQueue`(n).
foreach $i = 1$ *to* n **do**
    `dist`$[i] \leftarrow \infty$ and call `queue.Insert`(v_i, ∞).
Call `queue.ChangeKey`$(v_1, 0)$.
do
    $v_i \leftarrow$ `queue.Extract()`.
    foreach $(v_i, v_j) \in E$ **do**
        `dist`$[j] \leftarrow \min\{$`dist`$[j]$, `dist`$[i] + w(i, j)\}$.
        Call `queue.ChangeKey`$(v_j$, `dist`$[j])$,
while queue *is not empty*
Return `dist`.

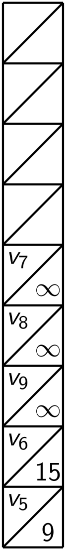


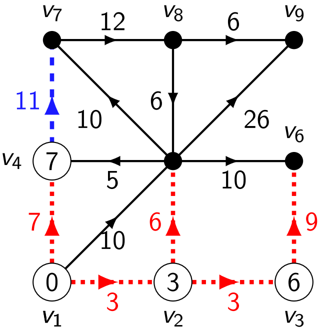
Dijkstra's algorithm: Implementation

Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.
Output : $d(v, y)$ for all $y \in V$.

Number the vertices of G as $v = v_1, \ldots, v_n$.
queue $\leftarrow$ `StartQueue`(n).
foreach $i = 1$ *to* n **do**
    `dist`$[i] \leftarrow \infty$ and call `queue.Insert`(v_i, ∞).
Call `queue.ChangeKey`$(v_1, 0)$.
do
    $v_i \leftarrow$ `queue.Extract`$()$.
    foreach $(v_i, v_j) \in E$ **do**
        `dist`$[j] \leftarrow \min\{$`dist`$[j]$, `dist`$[i] + w(i, j)\}$.
        Call `queue.ChangeKey`$(v_j,$ `dist`$[j])$,
while queue *is not empty*
Return `dist`.



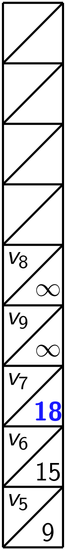


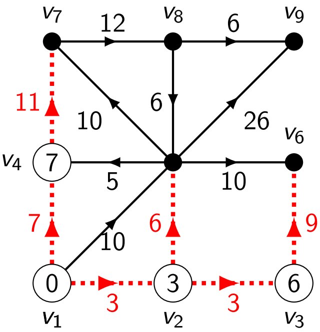
Dijkstra's algorithm: Implementation

Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.
Output : $d(v, y)$ for all $y \in V$.

```
Number the vertices of G as v = v_1, ..., v_n.
queue ← StartQueue(n).
foreach i = 1 to n do
    dist[i] ← ∞ and call queue.Insert(v_i, ∞).
Call queue.ChangeKey(v_1, 0).
do
    v_i ← queue.Extract().
    foreach (v_i, v_j) ∈ E do
        dist[j] ← min{dist[j], dist[i] + w(i, j)}.
        Call queue.ChangeKey(v_j, dist[j]),
while queue is not empty
Return dist.
```

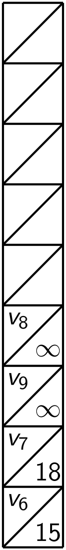


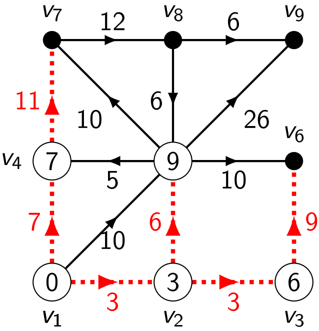


Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.
Output : $d(v, y)$ for all $y \in V$.

Number the vertices of G as $v = v_1, \dots, v_n$.
queue $\leftarrow$ `StartQueue`(n).
foreach $i = 1$ *to* n **do**
  `dist`$[i] \leftarrow \infty$ and call `queue.Insert`(v_i, ∞).
Call `queue.ChangeKey`$(v_1, 0)$.
do
  $v_i \leftarrow$ `queue.Extract`$()$.
  foreach $(v_i, v_j) \in E$ **do**
    `dist`$[j] \leftarrow \min\{$`dist`$[j],$ `dist`$[i] + w(i, j)\}$.
    Call `queue.ChangeKey`$(v_j,$ `dist`$[j])$,
while queue *is not empty*
Return `dist`.



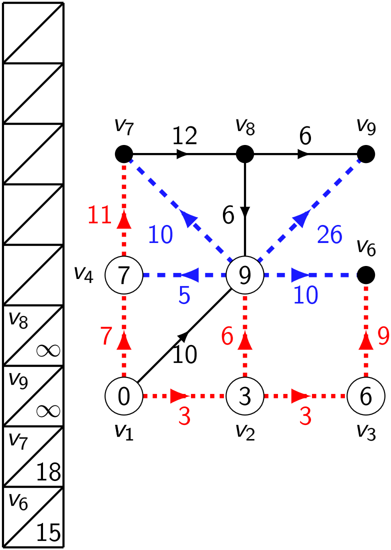


Dijkstra's algorithm: Implementation

Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.
Output : $d(v, y)$ for all $y \in V$.

Number the vertices of G as $v = v_1, \dots, v_n$.
queue $\leftarrow$ `StartQueue`(n).
foreach $i = 1$ *to* n **do**
  `dist`$[i] \leftarrow \infty$ and call `queue.Insert`(v_i, ∞).
Call `queue.ChangeKey`$(v_1, 0)$.
do
  $v_i \leftarrow$ `queue.Extract()`.
  foreach $(v_i, v_j) \in E$ **do**
    `dist`$[j] \leftarrow \min\{$`dist`$[j]$, `dist`$[i] + w(i, j)\}$.
    Call `queue.ChangeKey`$(v_j,$ `dist`$[j])$,
while queue *is not empty*
Return `dist`.

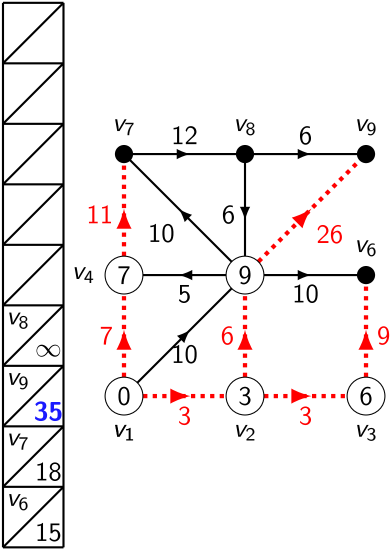


Dijkstra's algorithm: Implementation

Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.
Output : $d(v, y)$ for all $y \in V$.

Number the vertices of G as $v = v_1, \dots, v_n$.
queue $\leftarrow$ $\texttt{StartQueue}(n)$.
foreach $i = 1$ *to* n **do**
  $\texttt{dist}[i] \leftarrow \infty$ and call $\texttt{queue.Insert}(v_i, \infty)$.
Call $\texttt{queue.ChangeKey}(v_1, 0)$.
do
  $v_i \leftarrow \texttt{queue.Extract}()$.
  foreach $(v_i, v_j) \in E$ **do**
    $\texttt{dist}[j] \leftarrow \min\{\texttt{dist}[j], \texttt{dist}[i] + w(i, j)\}$.
    Call $\texttt{queue.ChangeKey}(v_j, \texttt{dist}[j])$,
while queue *is not empty*
Return dist.

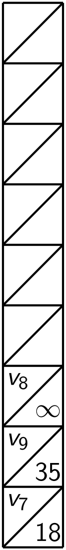


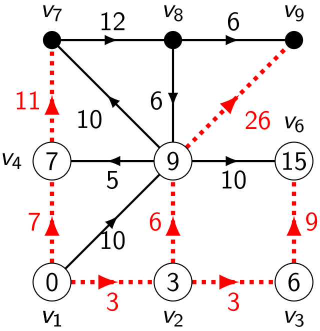
Dijkstra's algorithm: Implementation

Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.
Output : $d(v, y)$ for all $y \in V$.

Number the vertices of G as $v = v_1, \dots, v_n$.
queue $\leftarrow$ `StartQueue`(n).
foreach $i = 1$ *to* n **do**
  `dist`$[i] \leftarrow \infty$ and call `queue.Insert`(v_i, ∞).
Call `queue.ChangeKey`$(v_1, 0)$.
do
  $v_i \leftarrow$ `queue.Extract()`.
  foreach $(v_i, v_j) \in E$ **do**
    `dist`$[j] \leftarrow \min\{$`dist`$[j],$ `dist`$[i] + w(i, j)\}$.
    Call `queue.ChangeKey`$(v_j,$ `dist`$[j])$,
while queue *is not empty*
Return `dist`.



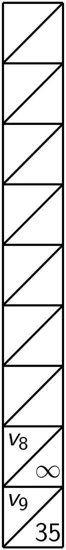


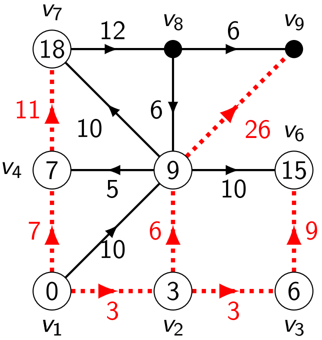
Dijkstra's algorithm: Implementation

Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.
Output : $d(v, y)$ for all $y \in V$.

Number the vertices of G as $v = v_1, \dots, v_n$.
queue $\leftarrow$ `StartQueue`(n).
foreach $i = 1$ *to* n **do**
  `dist`$[i] \leftarrow \infty$ and call `queue.Insert`(v_i, ∞).
Call `queue.ChangeKey`$(v_1, 0)$.
do
  $v_i \leftarrow$ `queue.Extract`$()$.
  foreach $(v_i, v_j) \in E$ **do**
    `dist`$[j] \leftarrow \min\{$`dist`$[j]$, `dist`$[i] + w(i,j)\}$.
    Call `queue.ChangeKey`$(v_j,$ `dist`$[j])$,
while queue *is not empty*
Return `dist`.



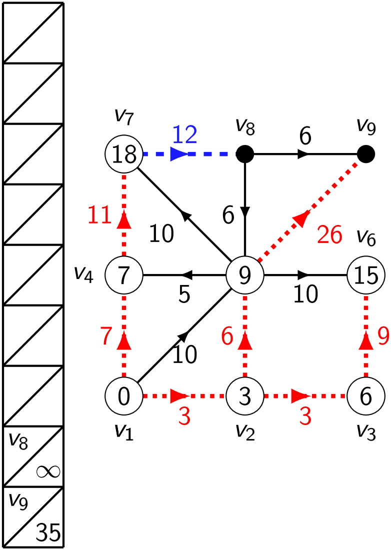


Dijkstra's algorithm: Implementation

Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.
Output : $d(v, y)$ for all $y \in V$.

Number the vertices of G as $v = v_1, \ldots, v_n$.
queue $\leftarrow$ `StartQueue`(n).
foreach $i = 1$ *to* n **do**
  `dist`$[i] \leftarrow \infty$ and call `queue.Insert`(v_i, ∞).
Call `queue.ChangeKey`$(v_1, 0)$.
do
  $v_i \leftarrow$ `queue.Extract()`.
  foreach $(v_i, v_j) \in E$ **do**
    `dist`$[j] \leftarrow \min\{$`dist`$[j],$ `dist`$[i] + w(i,j)\}$.
    Call `queue.ChangeKey`$(v_j,$ `dist`$[j])$,
while queue *is not empty*
Return `dist`.

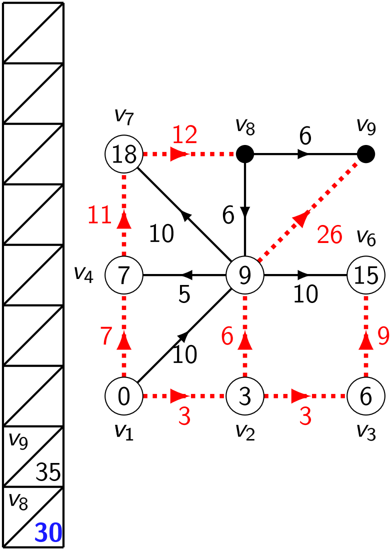


Dijkstra's algorithm: Implementation

Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.
Output : $d(v, y)$ for all $y \in V$.

Number the vertices of G as $v = v_1, \dots, v_n$.
queue $\leftarrow$ `StartQueue`(n).
foreach $i = 1$ *to* n **do**
  `dist`$[i] \leftarrow \infty$ and call `queue.Insert`(v_i, ∞).
Call `queue.ChangeKey`$(v_1, 0)$.
do
  $v_i \leftarrow$ `queue.Extract`$()$.
  foreach $(v_i, v_j) \in E$ **do**
    `dist`$[j] \leftarrow \min\{$`dist`$[j]$, `dist`$[i] + w(i, j)\}$.
    Call `queue.ChangeKey`$(v_j$, `dist`$[j])$,
while queue *is not empty*
Return `dist`.

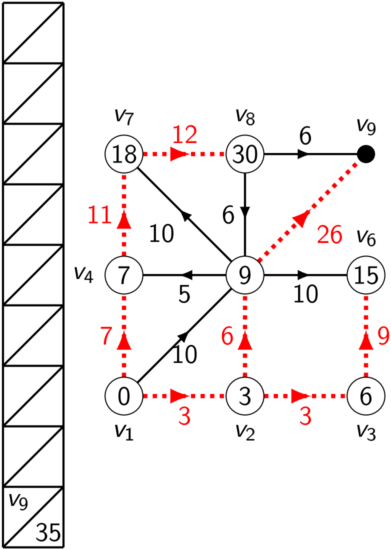


Dijkstra's algorithm: Implementation

Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V,E), w)$, $v \in V$.
Output : $d(v,y)$ for all $y \in V$.

Number the vertices of G as $v = v_1, \dots, v_n$.
queue $\leftarrow$ `StartQueue`(n).
foreach $i = 1$ *to* n **do**
    `dist`$[i] \leftarrow \infty$ and call `queue.Insert`(v_i, ∞).
Call `queue.ChangeKey`$(v_1, 0)$.
do
    $v_i \leftarrow$ `queue.Extract()`.
    foreach $(v_i, v_j) \in E$ **do**
        `dist`$[j] \leftarrow \min\{$`dist`$[j]$, `dist`$[i] + w(i,j)\}$.
        Call `queue.ChangeKey`$(v_j,$ `dist`$[j])$,
while queue *is not empty*
Return `dist`.

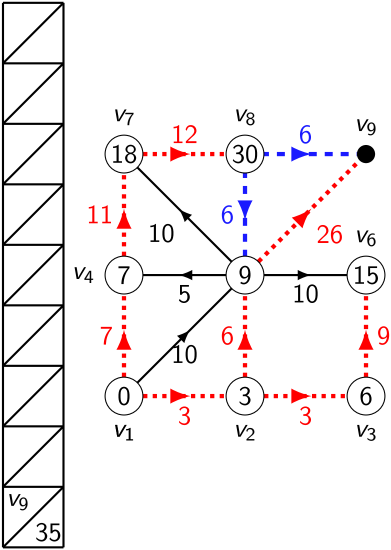


Dijkstra's algorithm: Implementation

Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.
Output : $d(v, y)$ for all $y \in V$.

Number the vertices of G as $v = v_1, \dots, v_n$.
queue $\leftarrow$ `StartQueue`(n).
foreach $i = 1$ *to* n **do**
    `dist`$[i] \leftarrow \infty$ and call `queue.Insert`(v_i, ∞).
Call `queue.ChangeKey`$(v_1, 0)$.
do
    $v_i \leftarrow$ `queue.Extract()`.
    foreach $(v_i, v_j) \in E$ **do**
        `dist`$[j] \leftarrow \min\{$`dist`$[j]$, `dist`$[i] + w(i, j)\}$.
        Call `queue.ChangeKey`$(v_j$, `dist`$[j])$,
while queue *is not empty*
Return `dist`.

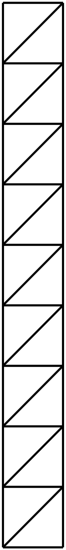


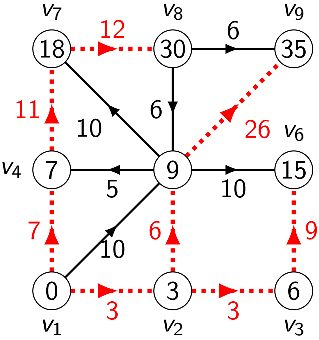
Dijkstra's algorithm: Implementation

Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.
Output : $d(v, y)$ for all $y \in V$.

Number the vertices of G as $v = v_1, \dots, v_n$.
queue $\leftarrow$ `StartQueue`(n).
foreach $i = 1$ *to* n **do**
  `dist`$[i] \leftarrow \infty$ and call `queue.Insert`(v_i, ∞).
Call `queue.ChangeKey`$(v_1, 0)$.
do
  $v_i \leftarrow$ `queue.Extract()`.
  foreach $(v_i, v_j) \in E$ **do**
    `dist`$[j] \leftarrow \min\{$`dist`$[j]$, `dist`$[i] + w(i, j)\}$.
    Call `queue.ChangeKey`$(v_j$, `dist`$[j])$,
while queue *is not empty*
Return `dist`.





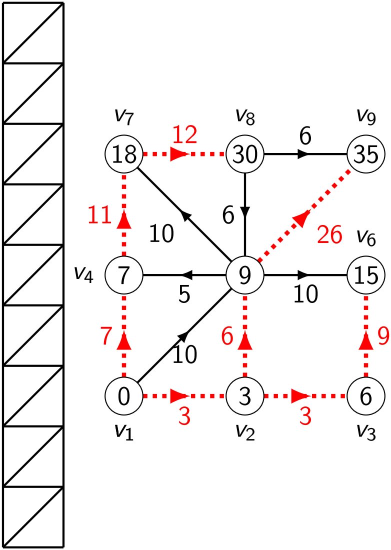
Dijkstra's algorithm: Implementation

Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.
Output : $d(v, y)$ for all $y \in V$.

Number the vertices of G as $v = v_1, \ldots, v_n$.
queue $\leftarrow$ `StartQueue`(n).
foreach $i = 1$ *to* n **do**
  `dist`$[i] \leftarrow \infty$ and call `queue.Insert`(v_i, ∞).
Call `queue.ChangeKey`$(v_1, 0)$.
do
  $v_i \leftarrow$ `queue.Extract()`.
  foreach $(v_i, v_j) \in E$ **do**
    `dist`$[j] \leftarrow \min\{$`dist`$[j]$, `dist`$[i] + w(i, j)\}$.
    Call `queue.ChangeKey`$(v_j,$ `dist`$[j])$,
while queue *is not empty*
Return `dist`.

Invariant: `dist`$[j]$ is the minimum value of $d(v_1, v_i) + w(v_i, v_j)$ over all v_i's whose distances are finalised, as in mathematical version. ✓



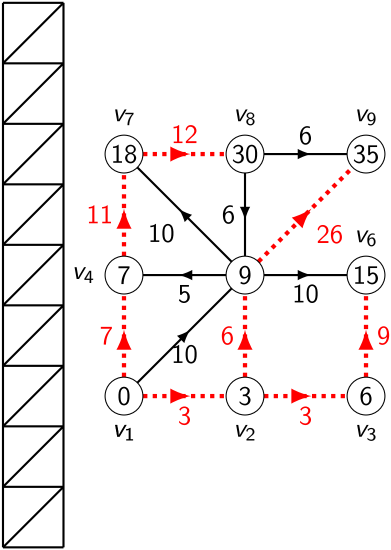
Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.
Output : $d(v, y)$ for all $y \in V$.

Number the vertices of G as $v = v_1, \dots, v_n$.
queue $\leftarrow$ `StartQueue`(n).
foreach $i = 1$ *to* n **do**
    `dist`$[i] \leftarrow \infty$ and call `queue.Insert`(v_i, ∞).
Call `queue.ChangeKey`$(v_1, 0)$.
do
    $v_i \leftarrow$ `queue.Extract()`.
    foreach $(v_i, v_j) \in E$ **do**
        `dist`$[j] \leftarrow \min\{$`dist`$[j],$ `dist`$[i] + w(i, j)\}$.
        Call `queue.ChangeKey`$(v_j,$ `dist`$[j])$,
while queue *is not empty*
Return `dist`.

Invariant: `dist`$[j]$ is the minimum value of $d(v_1, v_i) + w(v_i, v_j)$ over all v_i's whose distances are finalised, as in mathematical version. ✓

We can recover shortest paths by storing and returning the dotted red edges.



Dijkstra's algorithm: Time analysis

Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.
Output : $d(v, y)$ for all $y \in V$.

Number the vertices of G as $v = v_1, \dots, v_n$.
queue $\leftarrow$ `StartQueue`(n).
foreach $i = 1$ *to* n **do**
    `dist`$[i] \leftarrow \infty$ and call `queue.Insert`(v_i, ∞).
Call `queue.ChangeKey`$(v_1, 0)$.
do
    `vert` $\leftarrow$ `queue.Extract`$()$, say `vert` $= v_i$.
    foreach $(v_i, v_j) \in E$ **do**
        `dist`$[j] \leftarrow \min\{$`dist`$[j]$, `dist`$[i] + w(i, j)\}$.
        Call `queue.ChangeKey`$(v_j,$ `dist`$[j])$,
while queue *is not empty*
Return `dist`.

Dijkstra's algorithm: Time analysis

Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.
Output : $d(v, y)$ for all $y \in V$.

Number the vertices of G as $v = v_1, \dots, v_n$.
queue $\leftarrow$ `StartQueue`(n).
foreach $i = 1$ *to* n **do**
  `dist`$[i] \leftarrow \infty$ and call `queue.Insert`(v_i, ∞).
Call `queue.ChangeKey`$(v_1, 0)$.
do
  `vert` $\leftarrow$ `queue.Extract()`, say `vert` $= v_i$.
  foreach $(v_i, v_j) \in E$ **do**
    `dist`$[j] \leftarrow \min\{$`dist`$[j]$, `dist`$[i] + w(i, j)\}$.
    Call `queue.ChangeKey`$(v_j,$ `dist`$[j])$,
while queue *is not empty*
Return `dist`.

We perform $O(|V|)$ `Insert` operations and `Extract` operations, and $O(|E|)$ `ChangeKey` operations, for a total of $O((|V| + |E|) \log |V|)$ time when G is given in adjacency list form.

Dijkstra's algorithm: Time analysis

Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.
Output : $d(v, y)$ for all $y \in V$.

Number the vertices of G as $v = v_1, \dots, v_n$.
queue $\leftarrow$ `StartQueue`(n).
foreach $i = 1$ *to* n **do**
  `dist`$[i] \leftarrow \infty$ and call `queue.Insert`(v_i, ∞).
Call `queue.ChangeKey`$(v_1, 0)$.
do
  `vert` $\leftarrow$ `queue.Extract()`, say `vert` $= v_i$.
  foreach $(v_i, v_j) \in E$ **do**
    `dist`$[j] \leftarrow \min\{$`dist`$[j]$, `dist`$[i] + w(i, j)\}$.
    Call `queue.ChangeKey`$(v_j,$ `dist`$[j])$,
while queue *is not empty*
Return `dist`.

We perform $O(|V|)$ `Insert` operations and `Extract` operations, and $O(|E|)$ `ChangeKey` operations, for a total of $O((|V| + |E|) \log |V|)$ time when G is given in adjacency list form.

We could drop this to $O(|V| \log |V| + |E|)$ time by using a **Fibonacci heap** as a priority queue...

Dijkstra's algorithm: Time analysis

Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.
Output : $d(v, y)$ for all $y \in V$.

Number the vertices of G as $v = v_1, \ldots, v_n$.
queue $\leftarrow$ `StartQueue`(n).
foreach $i = 1$ *to* n **do**
    `dist`$[i] \leftarrow \infty$ and call `queue.Insert`(v_i, ∞).
Call `queue.ChangeKey`$(v_1, 0)$.
do
    `vert` $\leftarrow$ `queue.Extract()`, say `vert` $= v_i$.
    foreach $(v_i, v_j) \in E$ **do**
        `dist`$[j] \leftarrow \min\{$`dist`$[j]$, `dist`$[i] + w(i, j)\}$.
        Call `queue.ChangeKey`$(v_j,$ `dist`$[j])$,
while queue *is not empty*
Return `dist`.

We perform $O(|V|)$ `Insert` operations and `Extract` operations, and $O(|E|)$ `ChangeKey` operations, for a total of $O((|V| + |E|) \log |V|)$ time when G is given in adjacency list form.

We could drop this to $O(|V| \log |V| + |E|)$ time by using a **Fibonacci heap** as a priority queue... But Fibonacci heaps have *awful* constants, and generally $\log |V| \lessapprox 50$, so let's not!