

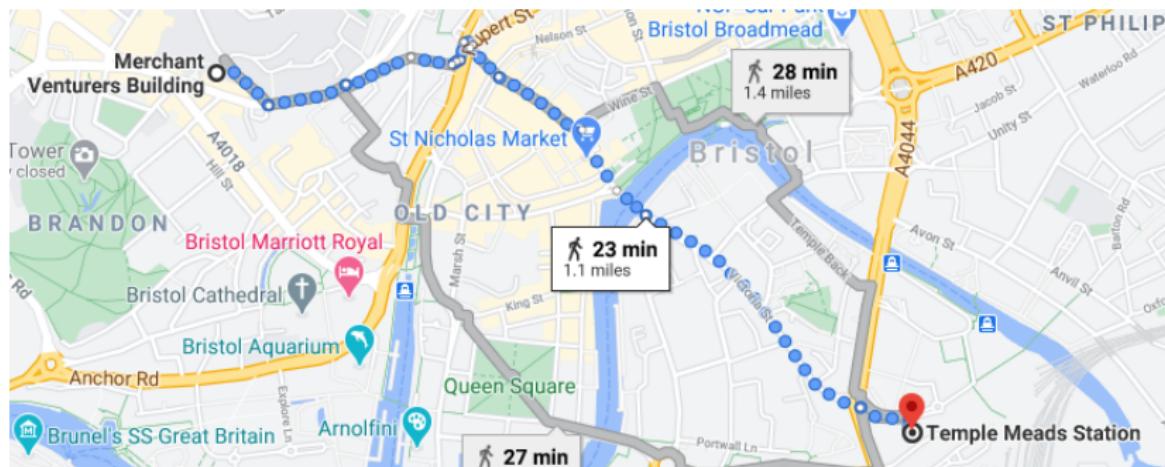
Dijkstra's algorithm

COMS20017 (Algorithms and Data)

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.

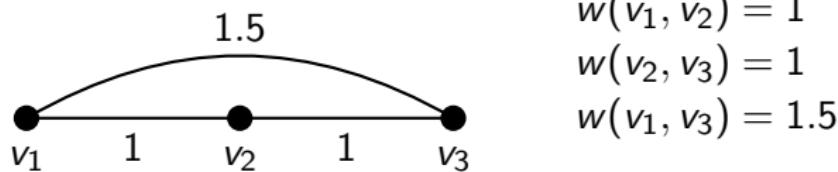


But when we want to find a “shortest path” in this graph, we don’t care about the number of edges, we care about the **physical distance**.

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

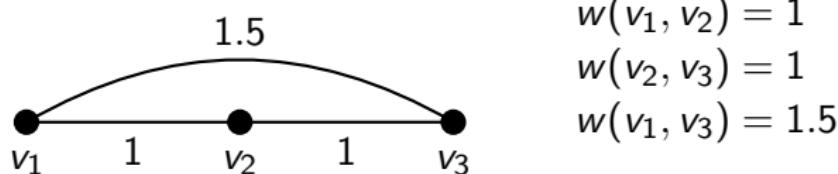
Weighted graphs

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



Weighted graphs

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

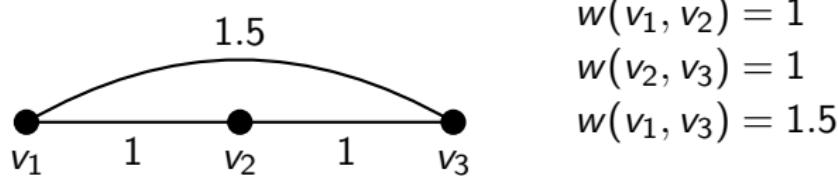


The **length** of a path/walk $P = x_1 \dots x_t$ is the total weight of P 's edges:

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

Weighted graphs

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



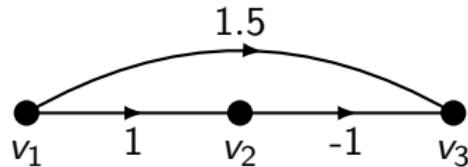
The **length** of a path/walk $P = x_1 \dots x_t$ is the total weight of P 's edges:

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

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

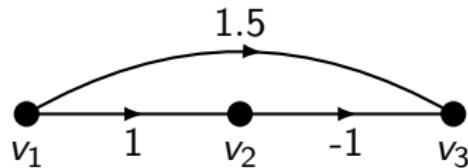
For some applications, it can make sense to allow edges to have **negative weight**. (E.g. costs versus profits...) This can be counterintuitive!



Here, $d(v_1, v_3) =$

Negative-weight edges

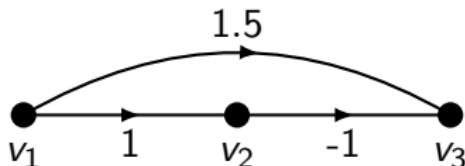
For some applications, it can make sense to allow edges to have **negative weight**. (E.g. costs versus profits...) This can be counterintuitive!



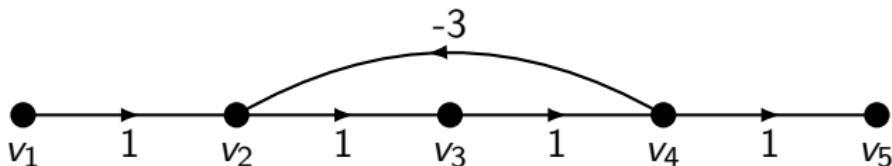
Here, $d(v_1, v_3) = 0$, since $v_1 v_2 v_3$ has cost $w(v_1, v_2) + w(v_2, v_3) = 0$.

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!

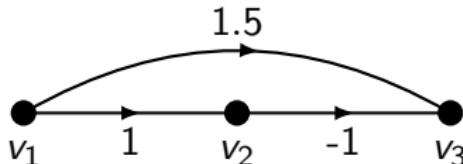


Here, $d(v_1, v_3) = 0$, since $v_1 v_2 v_3$ has cost $w(v_1, v_2) + w(v_2, v_3) = 0$.

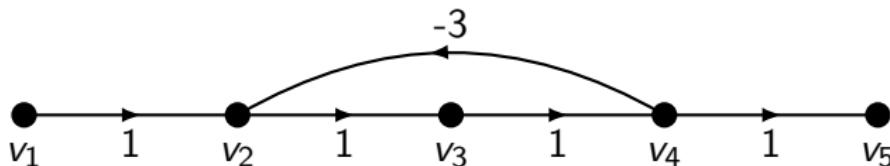


Negative-weight edges

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



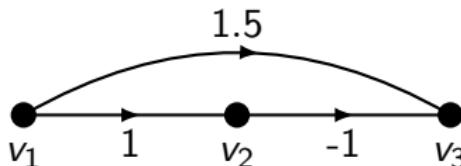
Here, $d(v_1, v_3) = 0$, since $v_1 v_2 v_3$ has cost $w(v_1, v_2) + w(v_2, v_3) = 0$.



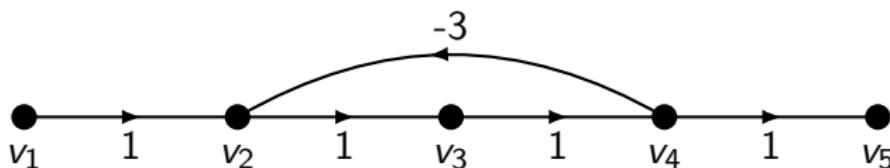
Here, “distance” doesn’t even make sense — there are walks from v_1 to v_5 with **arbitrarily low** length. E.g. $\text{length}(v_1 v_2 v_3 v_4 v_5) = 4$.

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!



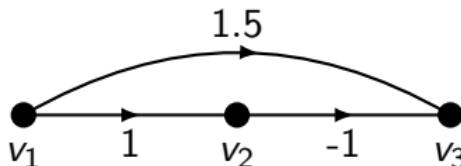
Here, $d(v_1, v_3) = 0$, since $v_1 v_2 v_3$ has cost $w(v_1, v_2) + w(v_2, v_3) = 0$.



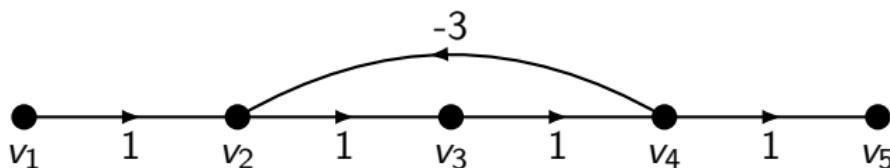
Here, “distance” doesn’t even make sense — there are walks from v_1 to v_5 with **arbitrarily low** length. E.g. $\text{length}(v_1 v_2 v_3 v_4 v_2 v_3 v_4 v_5) = 3$.

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!



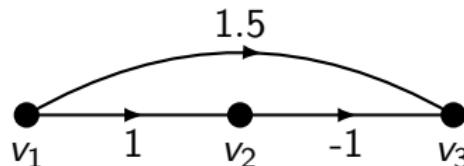
Here, $d(v_1, v_3) = 0$, since $v_1 v_2 v_3$ has cost $w(v_1, v_2) + w(v_2, v_3) = 0$.



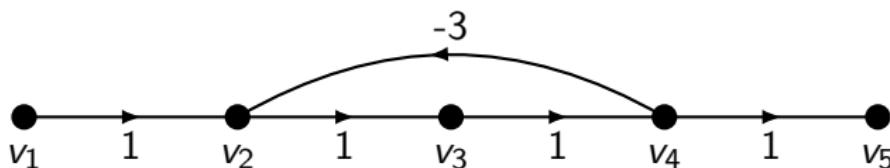
Here, “distance” doesn’t even make sense — there are walks from v_1 to v_5 with **arbitrarily low** length. E.g. $\text{length}(v_1 v_2 v_3 v_4 v_2 v_3 v_4 \color{blue}{v_2 v_3 v_4} v_5) = 2\dots$

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!



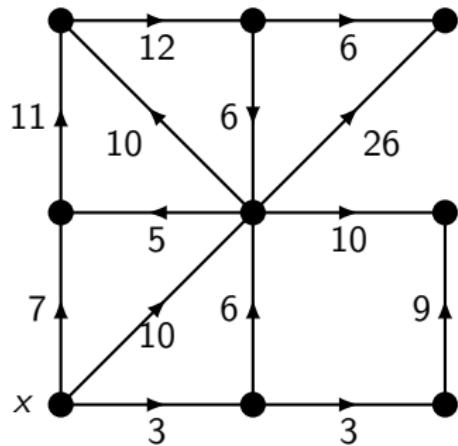
Here, $d(v_1, v_3) = 0$, since $v_1 v_2 v_3$ has cost $w(v_1, v_2) + w(v_2, v_3) = 0$.



Here, “distance” doesn’t even make sense — there are walks from v_1 to v_5 with **arbitrarily low** length. E.g. $\text{length}(v_1 v_2 v_3 v_4 v_2 v_3 v_4 \color{blue}{v_2 v_3 v_4} v_5) = 2\dots$

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

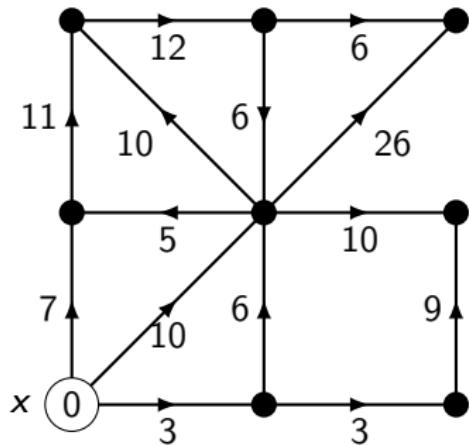
Dijkstra's algorithm: The idea



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 .

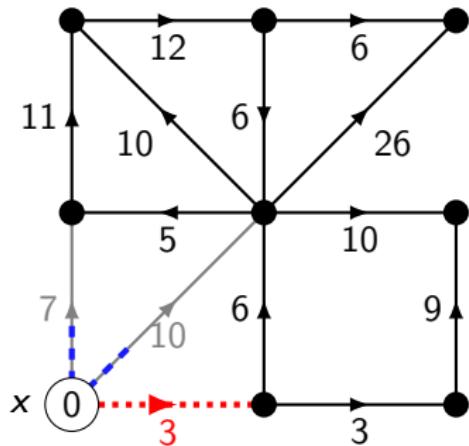
Dijkstra's algorithm: The idea



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 .

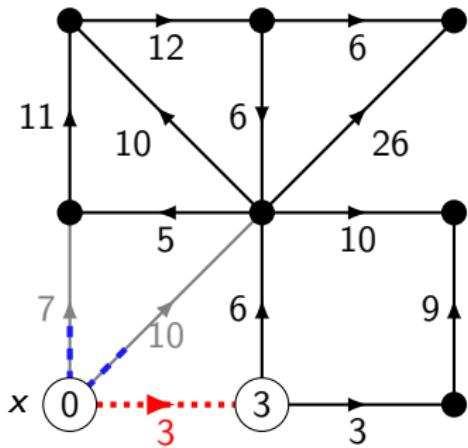
Dijkstra's algorithm: The idea



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 .

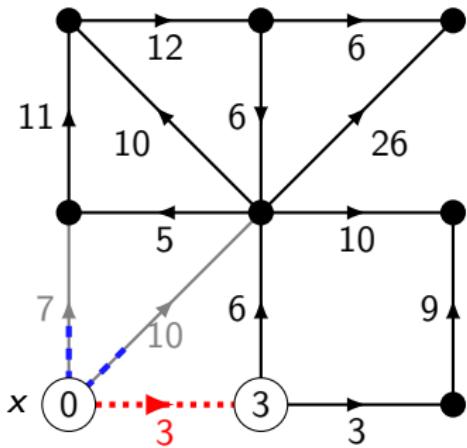
Dijkstra's algorithm: The idea



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 .

Dijkstra's algorithm: The idea



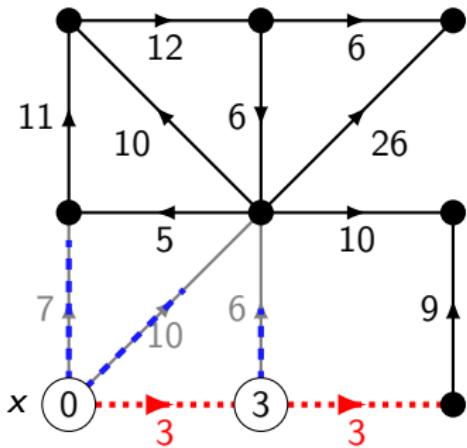
Think of breadth-first search as water flooding a set of pipes, starting from x ... and now allow the pipes to have **different lengths**.

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 $d(x, u) + \text{length}(u, v)$. (Break ties arbitrarily.)

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

Dijkstra's algorithm: The idea



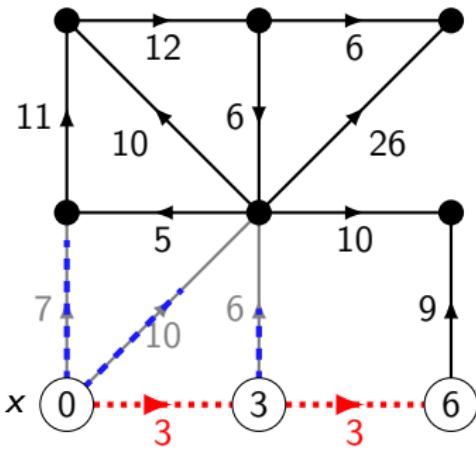
Think of breadth-first search as water flooding a set of pipes, starting from x ... and now allow the pipes to have **different lengths**.

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 $d(x, u) + \text{length}(u, v)$. (Break ties arbitrarily.)

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

Dijkstra's algorithm: The idea



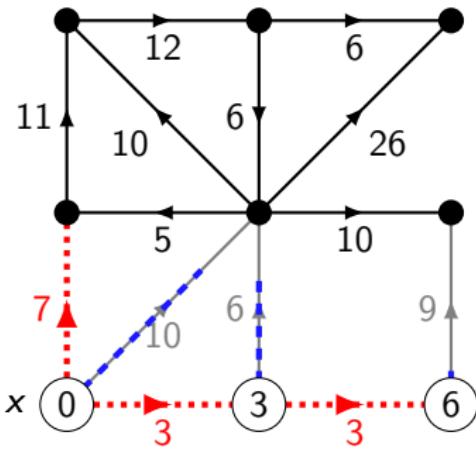
Think of breadth-first search as water flooding a set of pipes, starting from x ... and now allow the pipes to have **different lengths**.

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 $d(x, u) + \text{length}(u, v)$. (Break ties arbitrarily.)

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

Dijkstra's algorithm: The idea



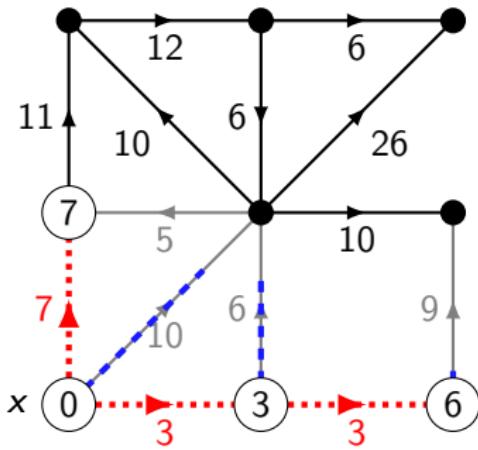
Think of breadth-first search as water flooding a set of pipes, starting from x ... and now allow the pipes to have **different lengths**.

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 $d(x, u) + \text{length}(u, v)$. (Break ties arbitrarily.)

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

Dijkstra's algorithm: The idea



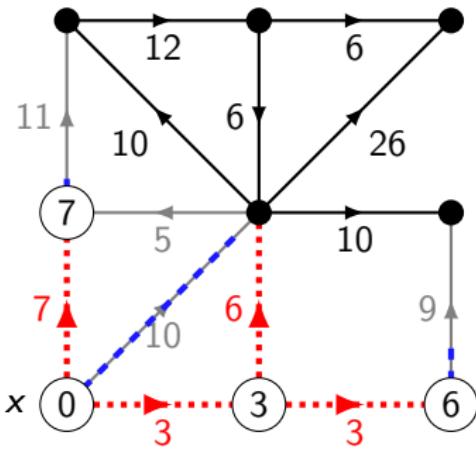
Think of breadth-first search as water flooding a set of pipes, starting from x ... and now allow the pipes to have **different lengths**.

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 $d(x, u) + \text{length}(u, v)$. (Break ties arbitrarily.)

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

Dijkstra's algorithm: The idea



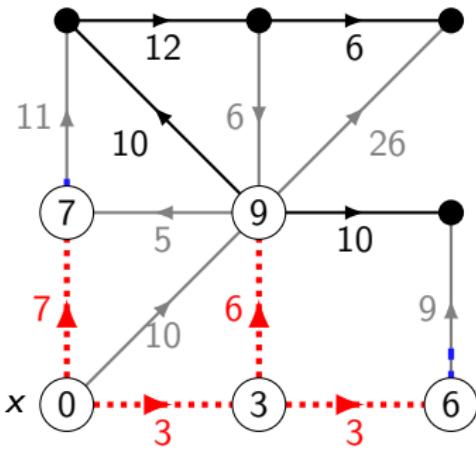
Think of breadth-first search as water flooding a set of pipes, starting from x ... and now allow the pipes to have **different lengths**.

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 $d(x, u) + \text{length}(u, v)$. (Break ties arbitrarily.)

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

Dijkstra's algorithm: The idea



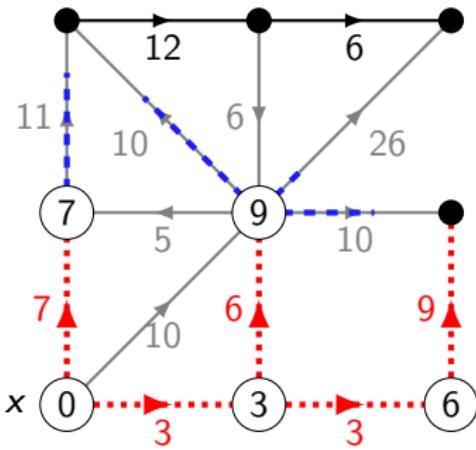
Think of breadth-first search as water flooding a set of pipes, starting from x ... and now allow the pipes to have **different lengths**.

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 $d(x, u) + \text{length}(u, v)$. (Break ties arbitrarily.)

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

Dijkstra's algorithm: The idea



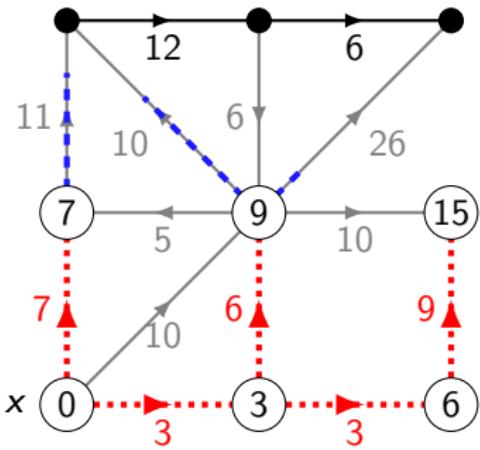
Think of breadth-first search as water flooding a set of pipes, starting from x ... and now allow the pipes to have **different lengths**.

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 $d(x, u) + \text{length}(u, v)$. (Break ties arbitrarily.)

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

Dijkstra's algorithm: The idea



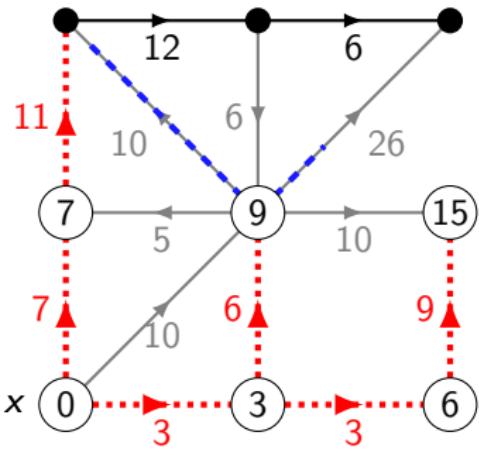
Think of breadth-first search as water flooding a set of pipes, starting from x ... and now allow the pipes to have **different lengths**.

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 $d(x, u) + \text{length}(u, v)$. (Break ties arbitrarily.)

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

Dijkstra's algorithm: The idea



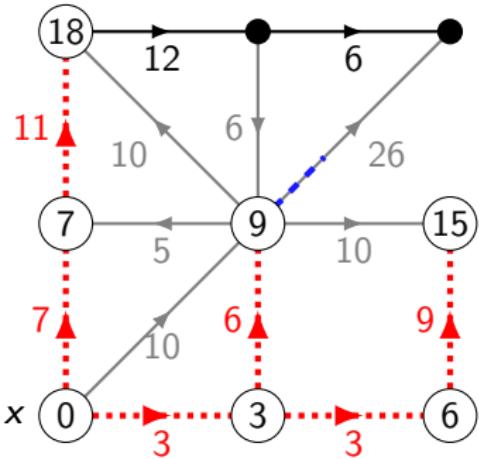
Think of breadth-first search as water flooding a set of pipes, starting from x ... and now allow the pipes to have **different lengths**.

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 $d(x, u) + \text{length}(u, v)$. (Break ties arbitrarily.)

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

Dijkstra's algorithm: The idea



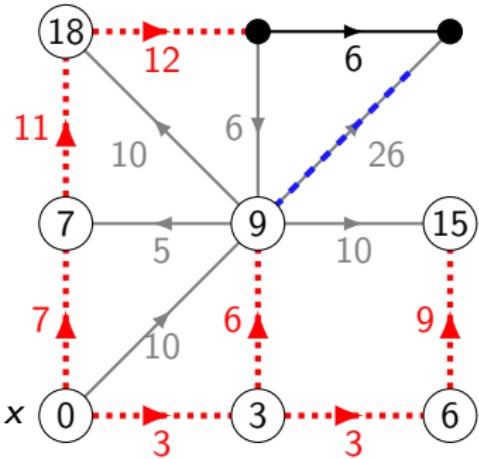
Think of breadth-first search as water flooding a set of pipes, starting from x ... and now allow the pipes to have **different lengths**.

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 $d(x, u) + \text{length}(u, v)$. (Break ties arbitrarily.)

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

Dijkstra's algorithm: The idea



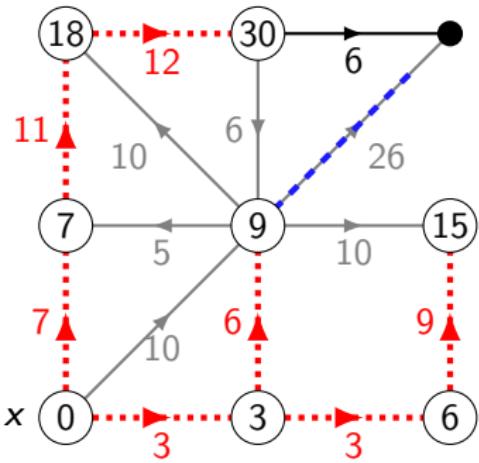
Think of breadth-first search as water flooding a set of pipes, starting from x ... and now allow the pipes to have **different lengths**.

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 $d(x, u) + \text{length}(u, v)$. (Break ties arbitrarily.)

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

Dijkstra's algorithm: The idea



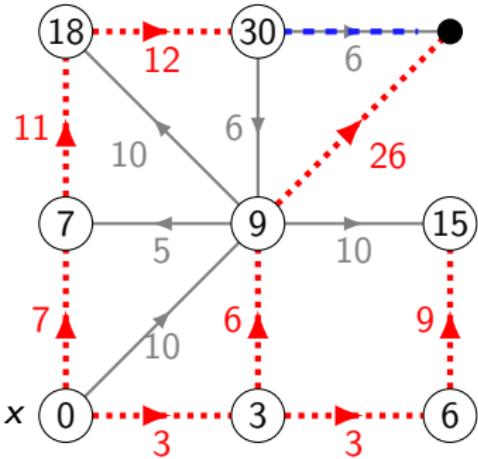
Think of breadth-first search as water flooding a set of pipes, starting from x ... and now allow the pipes to have **different lengths**.

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 $d(x, u) + \text{length}(u, v)$. (Break ties arbitrarily.)

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

Dijkstra's algorithm: The idea



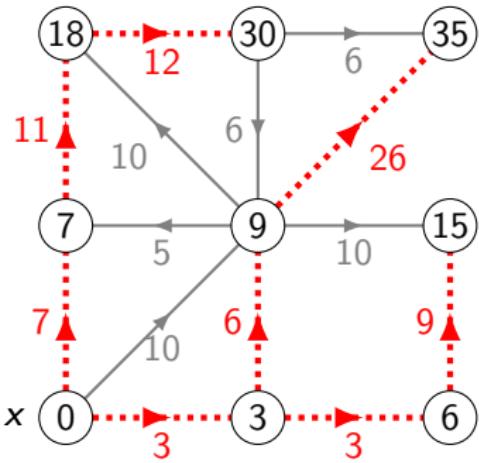
Think of breadth-first search as water flooding a set of pipes, starting from x ... and now allow the pipes to have **different lengths**.

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 $d(x, u) + \text{length}(u, v)$. (Break ties arbitrarily.)

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

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 $d(x, u) + \text{length}(u, v)$. (Break ties arbitrarily.)

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

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

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

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

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

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

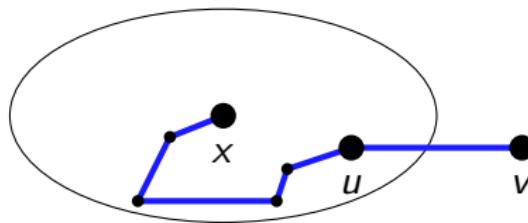
For input graph G , vertex x : At each stage, pick an edge (u, v) with $d(x, u)$ known and $d(x, v)$ unknown that minimises $d(x, u) + \text{length}(u, v)$. Then set $d(x, v) = d(x, u) + \text{length}(u, v)$.

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

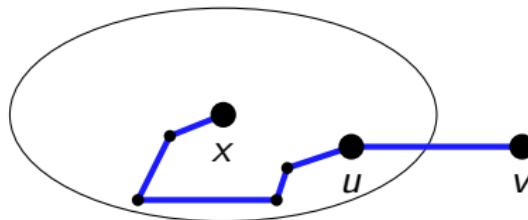
For input graph G , vertex x : At each stage, pick an edge (u, v) with $d(x, u)$ known and $d(x, v)$ unknown that minimises $d(x, u) + \text{length}(u, v)$. Then set $d(x, v) = d(x, u) + \text{length}(u, v)$.

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

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

Dijkstra's algorithm: Correctness

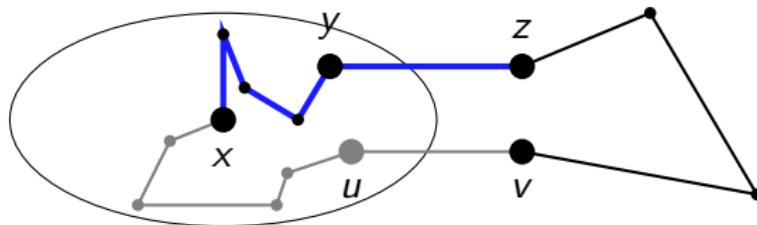
For input graph G , vertex x : At each stage, pick an edge (u, v) with $d(x, u)$ known and $d(x, v)$ unknown that minimises $d(x, u) + \text{length}(u, v)$. Then set $d(x, v) = d(x, u) + \text{length}(u, v)$.

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

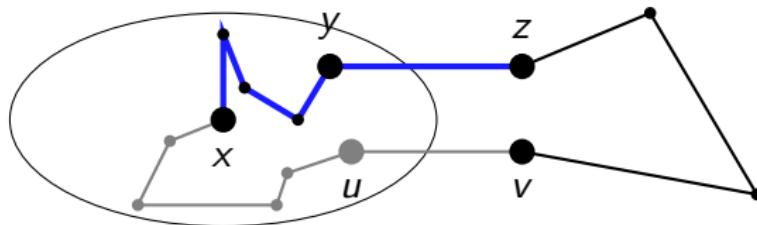
For input graph G , vertex x : At each stage, pick an edge (u, v) with $d(x, u)$ known and $d(x, v)$ unknown that minimises $d(x, u) + \text{length}(u, v)$. Then set $d(x, v) = d(x, u) + \text{length}(u, v)$.

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 $d(x, y) + \text{length}(y, z)$. So from the way we picked (u, v) , we have $d(x, v) \geq d(x, u) + \text{length}(u, v)$. □

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

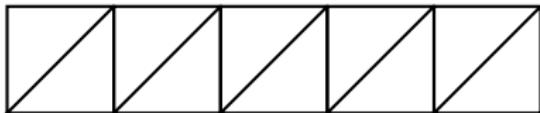
We need a **priority queue** (see COMS10007) to implement this efficiently.

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

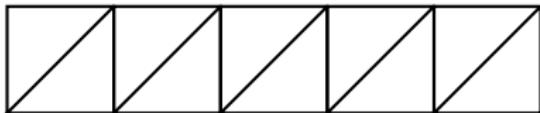
We need a **priority queue** (see COMS10007) to implement this efficiently.

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:

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

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



$\text{StartQueue}(5); \text{Insert}(v_1, 0);$

Priority queues

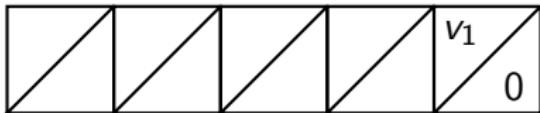
We need a **priority queue** (see COMS10007) to implement this efficiently.

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:

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

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



$\text{StartQueue}(5); \text{Insert}(v_1, 0);$

Priority queues

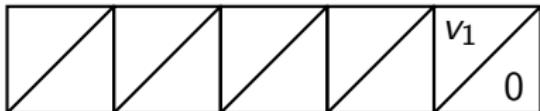
We need a **priority queue** (see COMS10007) to implement this efficiently.

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:

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

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



$\text{StartQueue}(5); \text{Insert}(v_1, 0); \text{Insert}(v_2, 4);$

Priority queues

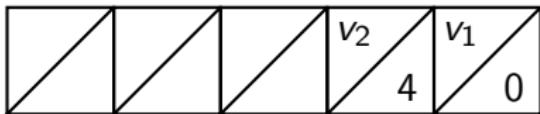
We need a **priority queue** (see COMS10007) to implement this efficiently.

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:

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

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



$\text{StartQueue}(5); \text{Insert}(v_1, 0); \text{Insert}(v_2, 4);$

Priority queues

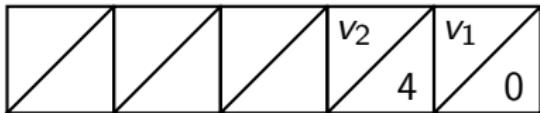
We need a **priority queue** (see COMS10007) to implement this efficiently.

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:

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

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



$\text{Insert}(v_3, 2);$

Priority queues

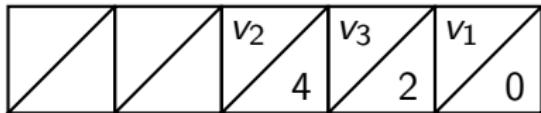
We need a **priority queue** (see COMS10007) to implement this efficiently.

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:

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

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



$\text{Insert}(v_3, 2);$

Priority queues

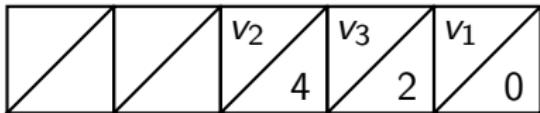
We need a **priority queue** (see COMS10007) to implement this efficiently.

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:

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

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



$\text{Insert}(v_3, 2); \text{Insert}(v_5, 3);$

Priority queues

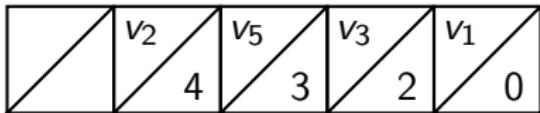
We need a **priority queue** (see COMS10007) to implement this efficiently.

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:

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

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



$\text{Insert}(v_3, 2); \text{Insert}(v_5, 3);$

Priority queues

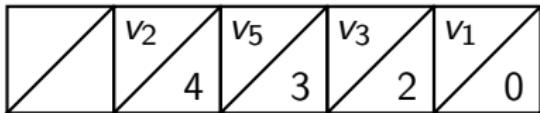
We need a **priority queue** (see COMS10007) to implement this efficiently.

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:

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

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



$\text{Insert}(v_3, 2); \text{Insert}(v_5, 3); \text{Extract}();$

Priority queues

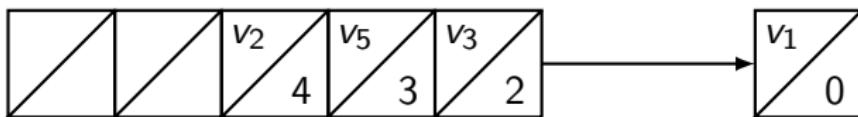
We need a **priority queue** (see COMS10007) to implement this efficiently.

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:

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

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



$\text{Insert}(v_3, 2); \text{Insert}(v_5, 3); \text{Extract}();$

Priority queues

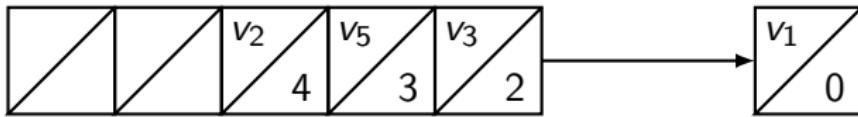
We need a **priority queue** (see COMS10007) to implement this efficiently.

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:

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

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



$\text{Insert}(v_4, 20);$

Priority queues

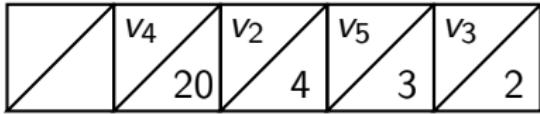
We need a **priority queue** (see COMS10007) to implement this efficiently.

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:

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

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



$\text{Insert}(v_4, 20);$

Priority queues

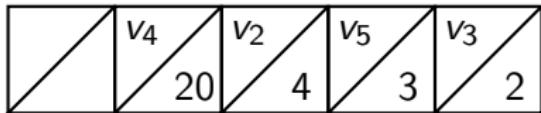
We need a **priority queue** (see COMS10007) to implement this efficiently.

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:

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

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



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

Priority queues

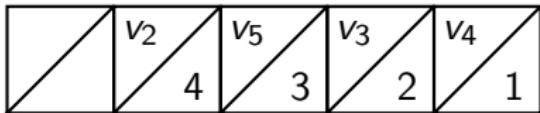
We need a **priority queue** (see COMS10007) to implement this efficiently.

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:

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

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



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

Priority queues

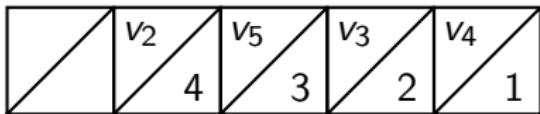
We need a **priority queue** (see COMS10007) to implement this efficiently.

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:

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

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



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

Priority queues

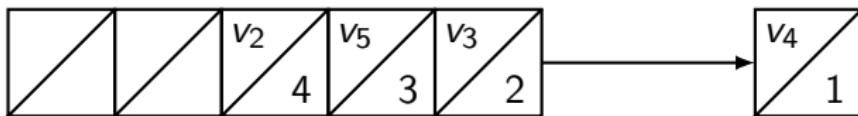
We need a **priority queue** (see COMS10007) to implement this efficiently.

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:

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

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



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

Dijkstra's algorithm: Implementation

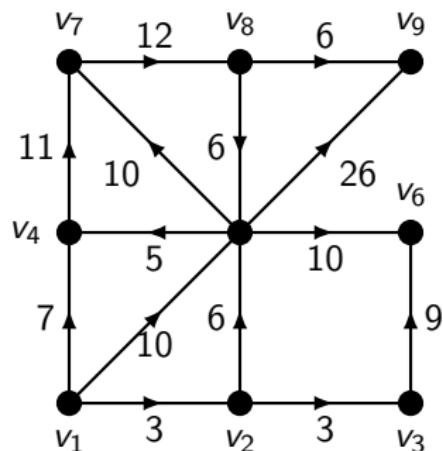
Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.

Output : $d(v, y)$ for all $y \in V$.

- 1 Number the vertices of G as $v = v_1, \dots, v_n$.
- 2 queue \leftarrow StartQueue(n).
- 3 **foreach** $i = 1$ to n **do**
- 4 $\text{dist}[i] \leftarrow \infty$ and call queue.Insert(v_i, ∞).
- 5 Call queue.ChangeKey($v_1, 0$).
- 6 **do**
- 7 $v_i \leftarrow \text{queue}.\text{Extract}()$.
- 8 **foreach** $(v_i, v_j) \in E$ **do**
- 9 $\text{dist}[j] \leftarrow \min\{\text{dist}[j], \text{dist}[i] + w(i, j)\}$.
- 10 Call queue.ChangeKey($v_j, \text{dist}[j]$),
- 11 **while** queue is not empty
- 12 Return dist.

v_5	∞
v_7	∞
v_8	∞
v_3	∞
v_9	∞
v_1	∞
v_6	∞
v_2	∞
v_4	∞



Dijkstra's algorithm: Implementation

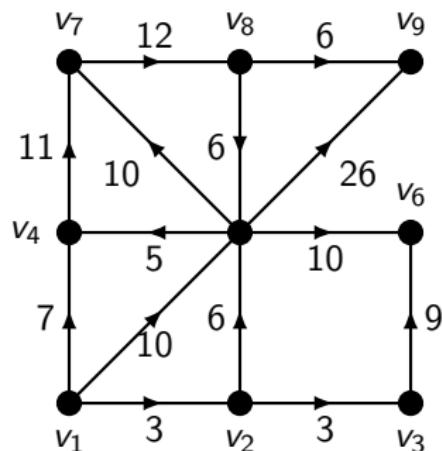
Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.

Output : $d(v, y)$ for all $y \in V$.

```
1 Number the vertices of  $G$  as  $v = v_1, \dots, v_n$ .  
2 queue  $\leftarrow$  StartQueue( $n$ ).  
3 foreach  $i = 1$  to  $n$  do  
4    $\text{dist}[i] \leftarrow \infty$  and call queue.Insert( $v_i, \infty$ ).  
5 Call queue.ChangeKey( $v_1, 0$ ).  
6 do  
7    $v_i \leftarrow \text{queue}.\text{Extract}()$ .  
8   foreach  $(v_i, v_j) \in E$  do  
9      $\text{dist}[j] \leftarrow \min\{\text{dist}[j], \text{dist}[i] + w(i, j)\}$ .  
10    Call queue.ChangeKey( $v_j, \text{dist}[j]$ ).  
11 while queue is not empty  
12 Return dist.
```

v_5	∞
v_7	∞
v_8	∞
v_3	∞
v_9	∞
v_6	∞
v_2	∞
v_4	∞
v_1	0



Dijkstra's algorithm: Implementation

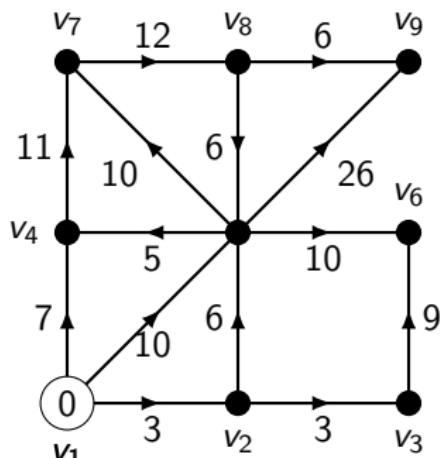
Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.

Output : $d(v, y)$ for all $y \in V$.

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

v_5	∞
v_7	∞
v_8	∞
v_3	∞
v_9	∞
v_6	∞
v_2	∞
v_4	∞



Dijkstra's algorithm: Implementation

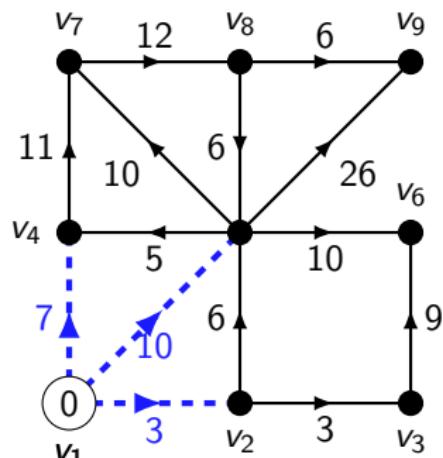
Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.

Output : $d(v, y)$ for all $y \in V$.

```
1 Number the vertices of  $G$  as  $v = v_1, \dots, v_n$ .  
2 queue  $\leftarrow$  StartQueue( $n$ ).  
3 foreach  $i = 1$  to  $n$  do  
4    $\text{dist}[i] \leftarrow \infty$  and call queue.Insert( $v_i, \infty$ ).  
5 Call queue.ChangeKey( $v_1, 0$ ).  
6 do  
7    $v_i \leftarrow \text{queue}.\text{Extract}()$ .  
8   foreach  $(v_i, v_j) \in E$  do  
9      $\text{dist}[j] \leftarrow \min\{\text{dist}[j], \text{dist}[i] + w(i, j)\}$ .  
10    Call queue.ChangeKey( $v_j, \text{dist}[j]$ ).  
11 while queue is not empty  
12 Return dist.
```

v_5	∞
v_7	∞
v_8	∞
v_3	∞
v_9	∞
v_6	∞
v_2	∞
v_4	∞



Dijkstra's algorithm: Implementation

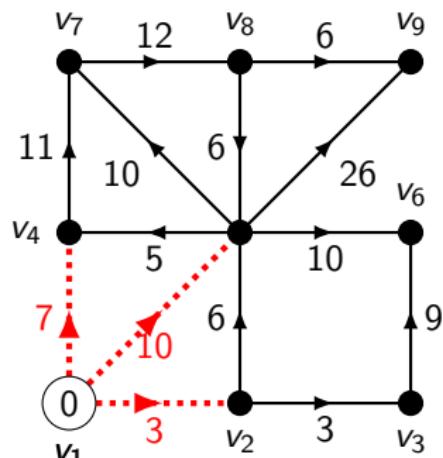
Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.

Output : $d(v, y)$ for all $y \in V$.

```
1 Number the vertices of  $G$  as  $v = v_1, \dots, v_n$ .
2 queue  $\leftarrow$  StartQueue( $n$ ).
3 foreach  $i = 1$  to  $n$  do
4    $\mid$   $dist[i] \leftarrow \infty$  and call queue.Insert( $v_i, \infty$ ).
5 Call queue.ChangeKey( $v_1, 0$ ).
6 do
7    $v_i \leftarrow$  queue.Extract().
8   foreach  $(v_i, v_j) \in E$  do
9      $\mid$   $dist[j] \leftarrow \min\{dist[j], dist[i] + w(i, j)\}$ .
10    Call queue.ChangeKey( $v_j, dist[j]$ ).
11 while queue is not empty
12 Return dist.
```

v_7	∞
v_8	∞
v_3	∞
v_9	∞
v_6	∞
v_5	10
v_4	7
v_2	3



Dijkstra's algorithm: Implementation

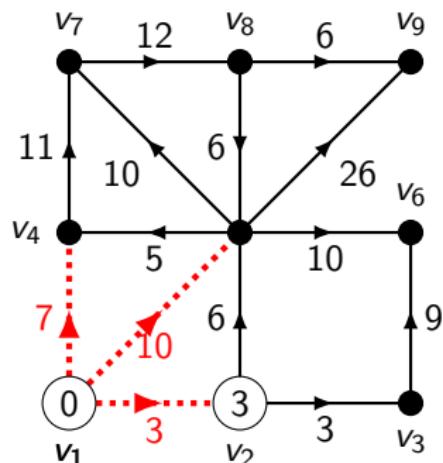
Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.

Output : $d(v, y)$ for all $y \in V$.

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

v_7	12		
v_8	∞		
v_3	∞		
v_9	∞		
v_6	∞		
v_5	10		
v_4	7	10	6



Dijkstra's algorithm: Implementation

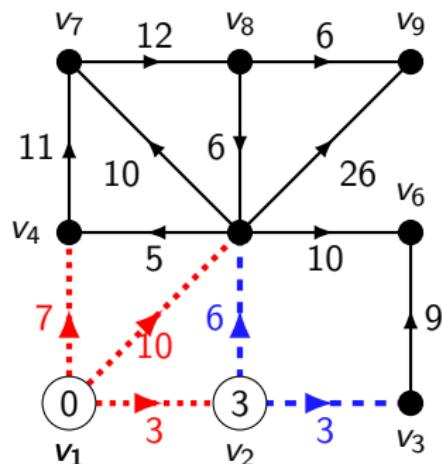
Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.

Output : $d(v, y)$ for all $y \in V$.

```
1 Number the vertices of  $G$  as  $v = v_1, \dots, v_n$ .  
2 queue  $\leftarrow$  StartQueue( $n$ ).  
3 foreach  $i = 1$  to  $n$  do  
4    $\text{dist}[i] \leftarrow \infty$  and call queue.Insert( $v_i, \infty$ ).  
5 Call queue.ChangeKey( $v_1, 0$ ).  
6 do  
7    $v_i \leftarrow \text{queue}.\text{Extract}()$ .  
8   foreach  $(v_i, v_j) \in E$  do  
9      $\text{dist}[j] \leftarrow \min\{\text{dist}[j], \text{dist}[i] + w(i, j)\}$ .  
10    Call queue.ChangeKey( $v_j, \text{dist}[j]$ ).  
11 while queue is not empty  
12 Return dist.
```

v_7	∞
v_8	∞
v_3	∞
v_9	∞
v_6	∞
v_5	10
v_4	7



Dijkstra's algorithm: Implementation

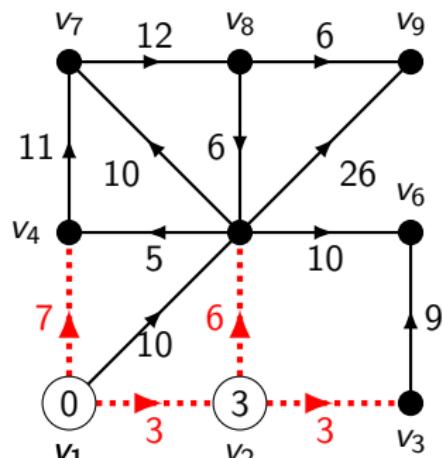
Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.

Output : $d(v, y)$ for all $y \in V$.

```
1 Number the vertices of  $G$  as  $v = v_1, \dots, v_n$ .
2 queue  $\leftarrow$  StartQueue( $n$ ).
3 foreach  $i = 1$  to  $n$  do
4    $\downarrow$   $dist[i] \leftarrow \infty$  and call queue.Insert( $v_i, \infty$ ).
5 Call queue.ChangeKey( $v_1, 0$ ).
6 do
7    $v_i \leftarrow$  queue.Extract().
8   foreach  $(v_i, v_j) \in E$  do
9      $dist[j] \leftarrow \min\{dist[j], dist[i] + w(i, j)\}$ .
10    Call queue.ChangeKey( $v_j, dist[j]$ ).
11 while queue is not empty
12 Return dist.
```

v_7	∞
v_8	∞
v_9	∞
v_6	∞
v_5	9
v_4	7
v_3	6



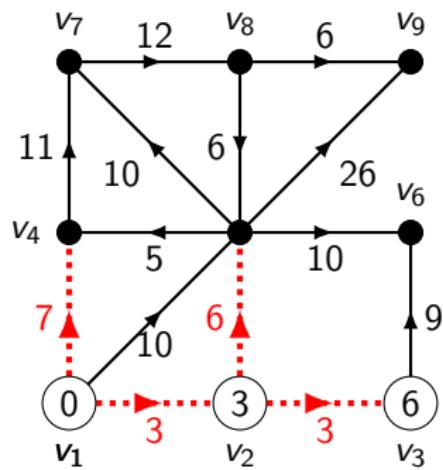
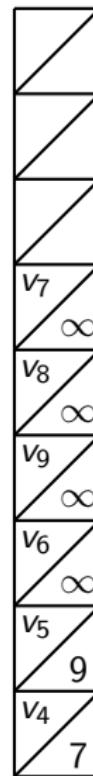
Dijkstra's algorithm: Implementation

Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.

Output : $d(v, y)$ for all $y \in V$.

```
1 Number the vertices of  $G$  as  $v = v_1, \dots, v_n$ .
2 queue  $\leftarrow$  StartQueue( $n$ ).
3 foreach  $i = 1$  to  $n$  do
4    $\downarrow$   $dist[i] \leftarrow \infty$  and call queue.Insert( $v_i, \infty$ ).
5 Call queue.ChangeKey( $v_1, 0$ ).
6 do
7    $v_i \leftarrow$  queue.Extract().
8   foreach  $(v_i, v_j) \in E$  do
9      $dist[j] \leftarrow \min\{dist[j], dist[i] + w(i, j)\}$ .
10    Call queue.ChangeKey( $v_j, dist[j]$ ).
11 while queue is not empty
12 Return dist.
```



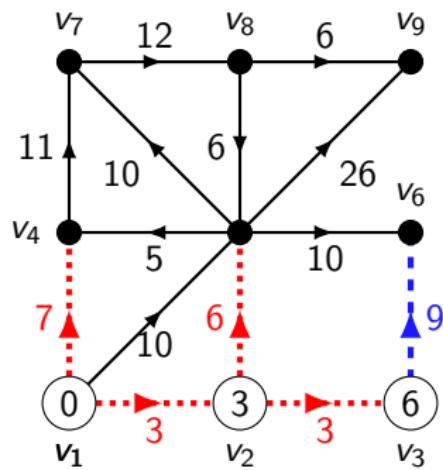
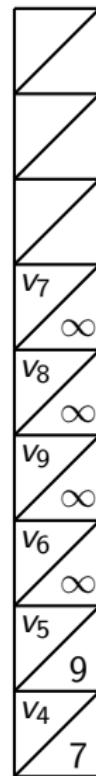
Dijkstra's algorithm: Implementation

Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.

Output : $d(v, y)$ for all $y \in V$.

```
1 Number the vertices of  $G$  as  $v = v_1, \dots, v_n$ .
2 queue  $\leftarrow$  StartQueue( $n$ ).
3 foreach  $i = 1$  to  $n$  do
4    $\downarrow$   $dist[i] \leftarrow \infty$  and call queue.Insert( $v_i, \infty$ ).
5 Call queue.ChangeKey( $v_1, 0$ ).
6 do
7    $v_i \leftarrow$  queue.Extract().
8   foreach  $(v_i, v_j) \in E$  do
9      $dist[j] \leftarrow \min\{dist[j], dist[i] + w(i, j)\}$ .
10    Call queue.ChangeKey( $v_j, dist[j]$ ).
11 while queue is not empty
12 Return dist.
```



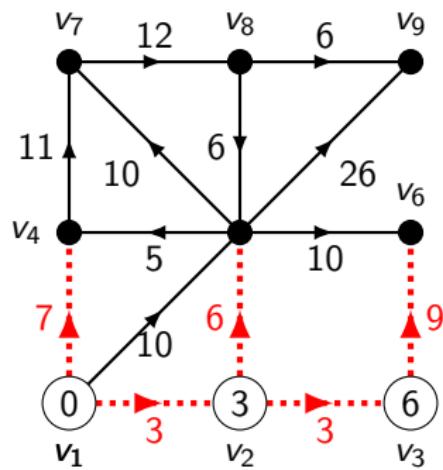
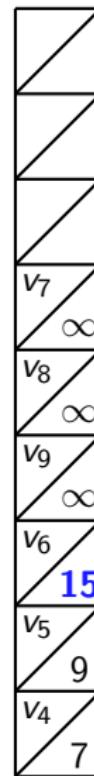
Dijkstra's algorithm: Implementation

Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.

Output : $d(v, y)$ for all $y \in V$.

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



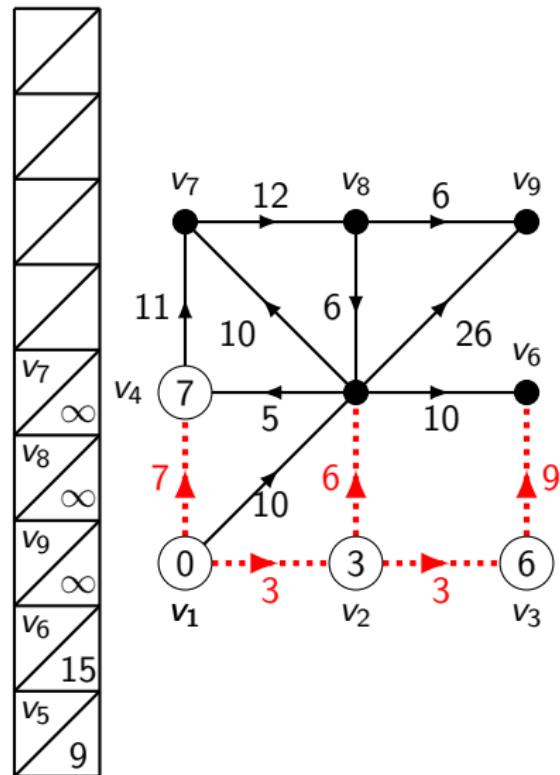
Dijkstra's algorithm: Implementation

Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.

Output : $d(v, y)$ for all $y \in V$.

```
1 Number the vertices of  $G$  as  $v = v_1, \dots, v_n$ .
2 queue  $\leftarrow$  StartQueue( $n$ ).
3 foreach  $i = 1$  to  $n$  do
4    $\downarrow$   $dist[i] \leftarrow \infty$  and call queue.Insert( $v_i, \infty$ ).
5 Call queue.ChangeKey( $v_1, 0$ ).
6 do
7    $v_i \leftarrow$  queue.Extract().
8   foreach  $(v_i, v_j) \in E$  do
9      $dist[j] \leftarrow \min\{dist[j], dist[i] + w(i, j)\}$ .
10    Call queue.ChangeKey( $v_j, dist[j]$ ).
11 while queue is not empty
12 Return dist.
```



Dijkstra's algorithm: Implementation

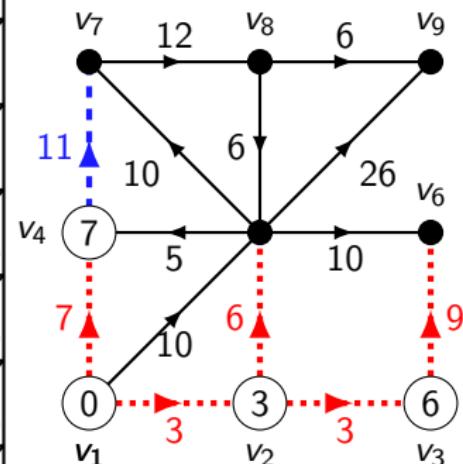
Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.

Output : $d(v, y)$ for all $y \in V$.

```
1 Number the vertices of  $G$  as  $v = v_1, \dots, v_n$ .  
2 queue  $\leftarrow$  StartQueue( $n$ ).  
3 foreach  $i = 1$  to  $n$  do  
4    $\text{dist}[i] \leftarrow \infty$  and call queue.Insert( $v_i, \infty$ ).  
5 Call queue.ChangeKey( $v_1, 0$ ).  
6 do  
7    $v_i \leftarrow \text{queue}.\text{Extract}()$ .  
8   foreach  $(v_i, v_j) \in E$  do  
9      $\text{dist}[j] \leftarrow \min\{\text{dist}[j], \text{dist}[i] + w(i, j)\}$ .  
10    Call queue.ChangeKey( $v_j, \text{dist}[j]$ ).  
11 while queue is not empty  
12 Return dist.
```

v_7	∞
v_8	∞
v_9	∞
v_6	15
v_5	9



Dijkstra's algorithm: Implementation

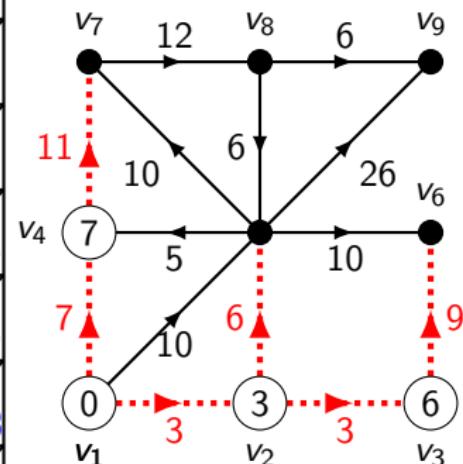
Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.

Output : $d(v, y)$ for all $y \in V$.

```
1 Number the vertices of  $G$  as  $v = v_1, \dots, v_n$ .
2 queue  $\leftarrow$  StartQueue( $n$ ).
3 foreach  $i = 1$  to  $n$  do
4    $\downarrow$   $dist[i] \leftarrow \infty$  and call queue.Insert( $v_i, \infty$ ).
5 Call queue.ChangeKey( $v_1, 0$ ).
6 do
7    $v_i \leftarrow$  queue.Extract().
8   foreach  $(v_i, v_j) \in E$  do
9      $dist[j] \leftarrow \min\{dist[j], dist[i] + w(i, j)\}$ .
10    Call queue.ChangeKey( $v_j, dist[j]$ ).
11 while queue is not empty
12 Return dist.
```

v_7	
v_8	∞
v_9	∞
v_7	18
v_6	15
v_5	9



Dijkstra's algorithm: Implementation

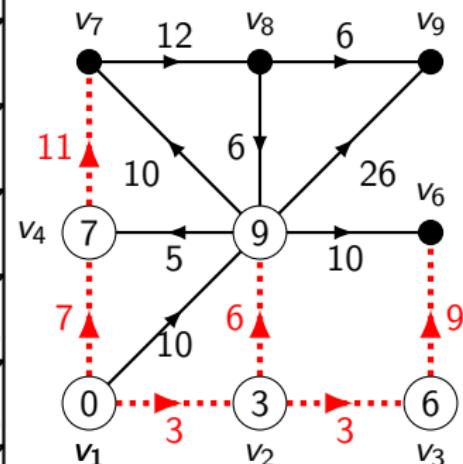
Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.

Output : $d(v, y)$ for all $y \in V$.

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

v_7	∞
v_8	∞
v_9	∞
v_7	18
v_6	15



Dijkstra's algorithm: Implementation

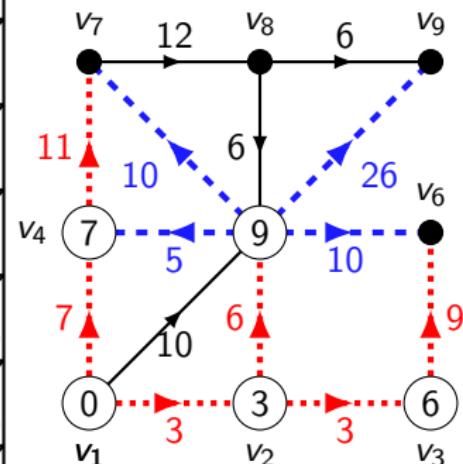
Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.

Output : $d(v, y)$ for all $y \in V$.

```
1 Number the vertices of  $G$  as  $v = v_1, \dots, v_n$ .
2 queue  $\leftarrow$  StartQueue( $n$ ).
3 foreach  $i = 1$  to  $n$  do
4    $\downarrow$   $dist[i] \leftarrow \infty$  and call queue.Insert( $v_i, \infty$ ).
5 Call queue.ChangeKey( $v_1, 0$ ).
6 do
7    $v_i \leftarrow$  queue.Extract().
8   foreach  $(v_i, v_j) \in E$  do
9      $\downarrow$   $dist[j] \leftarrow \min\{dist[j], dist[i] + w(i, j)\}$ .
10    Call queue.ChangeKey( $v_j, dist[j]$ ).
11 while queue is not empty
12 Return dist.
```

v_7	∞
v_8	∞
v_9	∞
v_7	18
v_6	15



Dijkstra's algorithm: Implementation

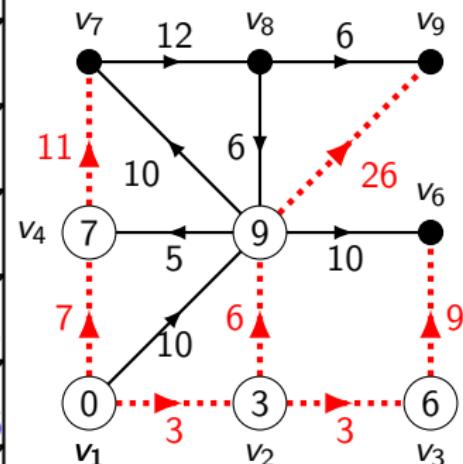
Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.

Output : $d(v, y)$ for all $y \in V$.

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

v_7	
v_8	∞
v_9	35
v_7	18
v_6	15



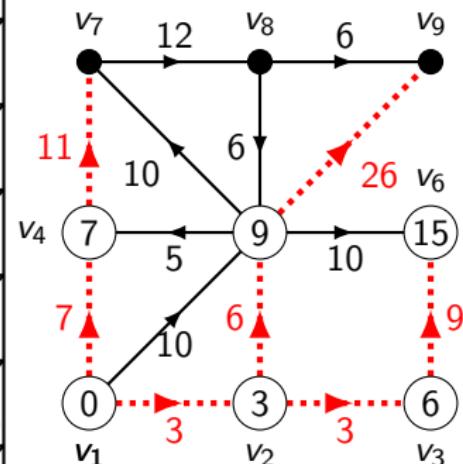
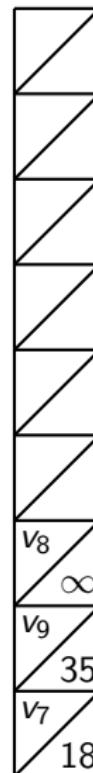
Dijkstra's algorithm: Implementation

Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.

Output : $d(v, y)$ for all $y \in V$.

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



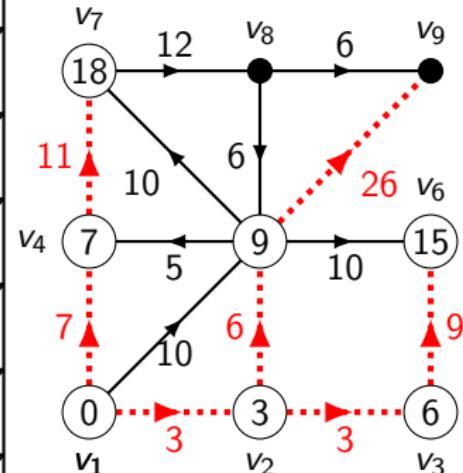
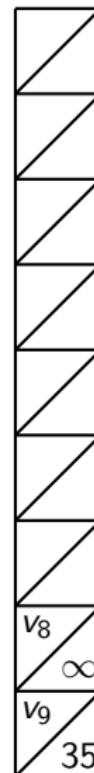
Dijkstra's algorithm: Implementation

Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.

Output : $d(v, y)$ for all $y \in V$.

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



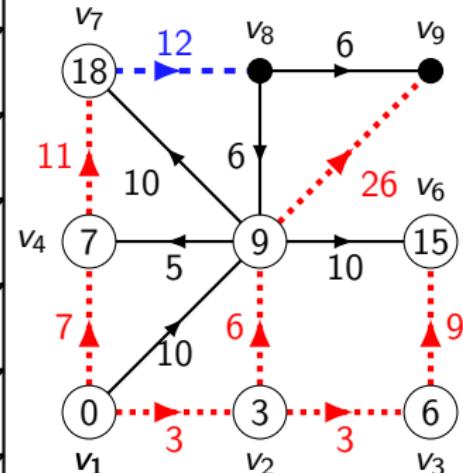
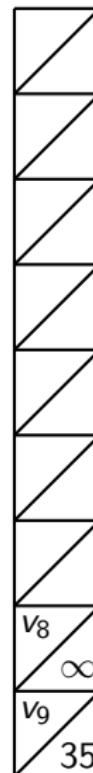
Dijkstra's algorithm: Implementation

Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.

Output : $d(v, y)$ for all $y \in V$.

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



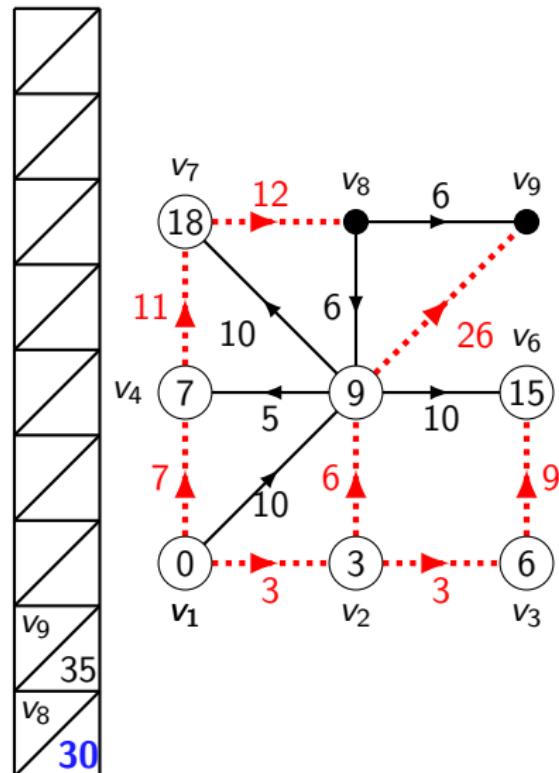
Dijkstra's algorithm: Implementation

Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.

Output : $d(v, y)$ for all $y \in V$.

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



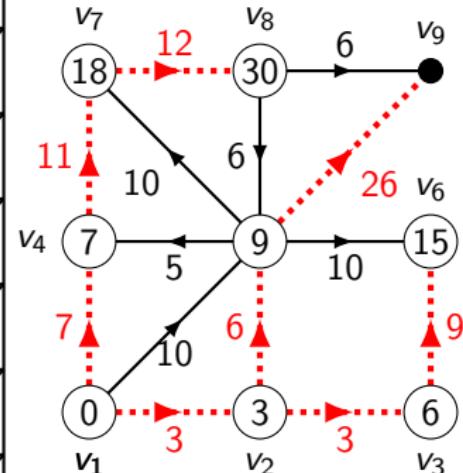
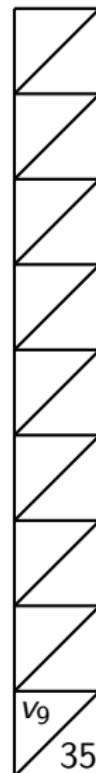
Dijkstra's algorithm: Implementation

Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.

Output : $d(v, y)$ for all $y \in V$.

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



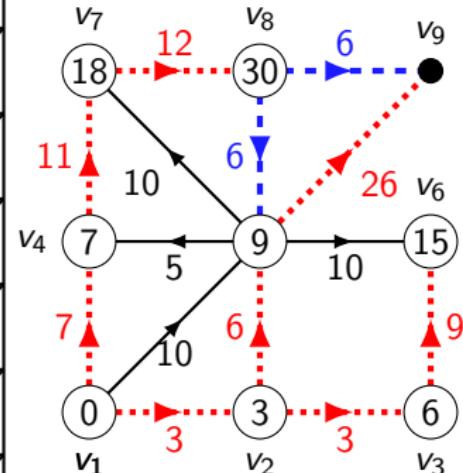
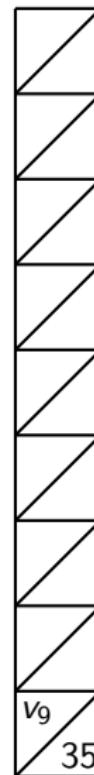
Dijkstra's algorithm: Implementation

Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.

Output : $d(v, y)$ for all $y \in V$.

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



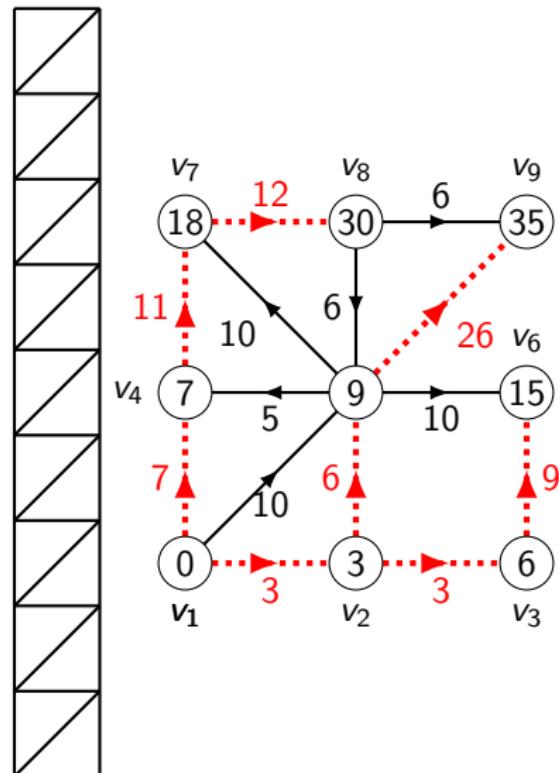
Dijkstra's algorithm: Implementation

Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.

Output : $d(v, y)$ for all $y \in V$.

```
1 Number the vertices of  $G$  as  $v = v_1, \dots, v_n$ .
2 queue  $\leftarrow$  StartQueue( $n$ ).
3 foreach  $i = 1$  to  $n$  do
4    $\quad$   $dist[i] \leftarrow \infty$  and call queue.Insert( $v_i, \infty$ ).
5 Call queue.ChangeKey( $v_1, 0$ ).
6 do
7    $v_i \leftarrow$  queue.Extract().
8   foreach  $(v_i, v_j) \in E$  do
9      $\quad$   $dist[j] \leftarrow \min\{dist[j], dist[i] + w(i, j)\}$ .
10    Call queue.ChangeKey( $v_j, dist[j]$ ),
11 while queue is not empty
12 Return dist.
```



Dijkstra's algorithm: Implementation

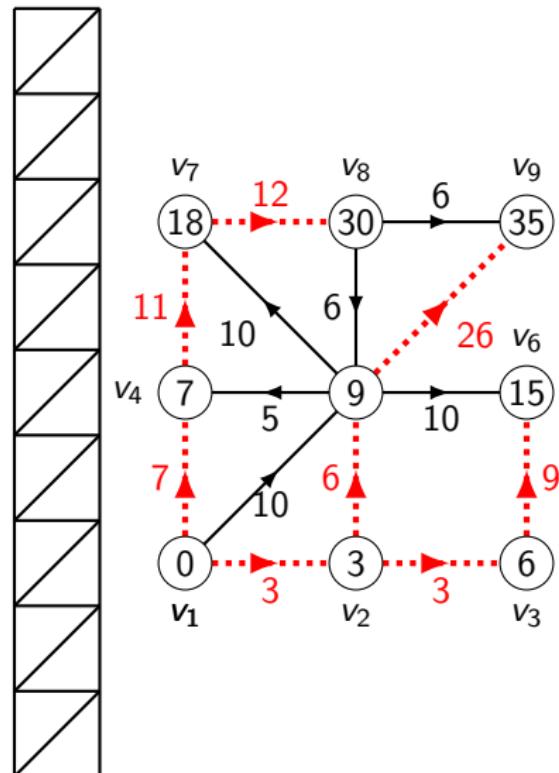
Algorithm: DIJKSTRA

Input : Weighted graph $G = ((V, E), w)$, $v \in V$.

Output : $d(v, y)$ for all $y \in V$.

```
1 Number the vertices of  $G$  as  $v = v_1, \dots, v_n$ .
2 queue  $\leftarrow$  StartQueue( $n$ ).
3 foreach  $i = 1$  to  $n$  do
4    $dist[i] \leftarrow \infty$  and call queue.Insert( $v_i, \infty$ ).
5 Call queue.ChangeKey( $v_1, 0$ ).
6 do
7    $v_i \leftarrow$  queue.Extract().
8   foreach  $(v_i, v_j) \in E$  do
9      $dist[j] \leftarrow \min\{dist[j], dist[i] + w(i, j)\}$ .
10    Call queue.ChangeKey( $v_j, dist[j]$ ).
11 while queue is not empty
12 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. ✓



Dijkstra's algorithm: Implementation

Algorithm: DIJKSTRA

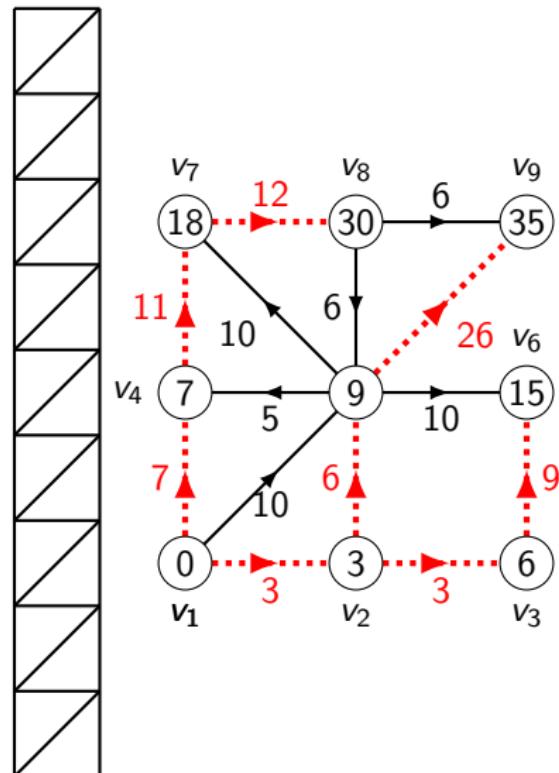
Input : Weighted graph $G = ((V, E), w)$, $v \in V$.

Output : $d(v, y)$ for all $y \in V$.

- 1 Number the vertices of G as $v = v_1, \dots, v_n$.
- 2 queue \leftarrow StartQueue(n).
- 3 **foreach** $i = 1$ to n **do**
- 4 $dist[i] \leftarrow \infty$ and call queue.Insert(v_i, ∞).
- 5 Call queue.ChangeKey($v_1, 0$).
- 6 **do**
- 7 $v_i \leftarrow$ queue.Extract().
- 8 **foreach** $(v_i, v_j) \in E$ **do**
- 9 $dist[j] \leftarrow \min\{dist[j], dist[i] + w(i, j)\}$.
- 10 Call queue.ChangeKey($v_j, dist[j]$),
- 11 **while** queue is not empty
- 12 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 $y \in V$.

- 1 Number the vertices of G as $v = v_1, \dots, v_n$.
 - 2 $\text{queue} \leftarrow \text{StartQueue}(n)$.
 - 3 **foreach** $i = 1$ to n **do**
 - 4 $\text{dist}[i] \leftarrow \infty$ and call $\text{queue.Insert}(v_i, \infty)$.
 - 5 Call $\text{queue.ChangeKey}(v_1, 0)$.
 - 6 **do**
 - 7 $\text{vert} \leftarrow \text{queue.Extract}()$, say $\text{vert} = v_i$.
 - 8 **foreach** $(v_i, v_j) \in E$ **do**
 - 9 $\text{dist}[j] \leftarrow \min\{\text{dist}[j], \text{dist}[i] + w(i, j)\}$.
 - 10 Call $\text{queue.ChangeKey}(v_j, \text{dist}[j])$,
 - 11 **while** queue is not empty
 - 12 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 $y \in V$.

- 1 Number the vertices of G as $v = v_1, \dots, v_n$.
 - 2 $\text{queue} \leftarrow \text{StartQueue}(n)$.
 - 3 **foreach** $i = 1$ to n **do**
 - 4 $\text{dist}[i] \leftarrow \infty$ and call $\text{queue.Insert}(v_i, \infty)$.
 - 5 Call $\text{queue.ChangeKey}(v_1, 0)$.
 - 6 **do**
 - 7 $\text{vert} \leftarrow \text{queue.Extract}$ (), say $\text{vert} = v_i$.
 - 8 **foreach** $(v_i, v_j) \in E$ **do**
 - 9 $\text{dist}[j] \leftarrow \min\{\text{dist}[j], \text{dist}[i] + w(i, j)\}$.
 - 10 Call $\text{queue.ChangeKey}(v_j, \text{dist}[j])$,
 - 11 **while** queue is not empty
 - 12 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 $y \in V$.

- 1 Number the vertices of G as $v = v_1, \dots, v_n$.
 - 2 $\text{queue} \leftarrow \text{StartQueue}(n)$.
 - 3 **foreach** $i = 1$ to n **do**
 - 4 $\text{dist}[i] \leftarrow \infty$ and call $\text{queue.Insert}(v_i, \infty)$.
 - 5 Call $\text{queue.ChangeKey}(v_1, 0)$.
 - 6 **do**
 - 7 $\text{vert} \leftarrow \text{queue.Extract}$ (), say $\text{vert} = v_i$.
 - 8 **foreach** $(v_i, v_j) \in E$ **do**
 - 9 $\text{dist}[j] \leftarrow \min\{\text{dist}[j], \text{dist}[i] + w(i, j)\}$.
 - 10 Call $\text{queue.ChangeKey}(v_j, \text{dist}[j])$,
 - 11 **while** queue is not empty
 - 12 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 $y \in V$.

- 1 Number the vertices of G as $v = v_1, \dots, v_n$.
 - 2 $\text{queue} \leftarrow \text{StartQueue}(n)$.
 - 3 **foreach** $i = 1$ to n **do**
 - 4 $\text{dist}[i] \leftarrow \infty$ and call $\text{queue.Insert}(v_i, \infty)$.
 - 5 Call $\text{queue.ChangeKey}(v_1, 0)$.
 - 6 **do**
 - 7 $\text{vert} \leftarrow \text{queue.Extract}$ (), say $\text{vert} = v_i$.
 - 8 **foreach** $(v_i, v_j) \in E$ **do**
 - 9 $\text{dist}[j] \leftarrow \min\{\text{dist}[j], \text{dist}[i] + w(i, j)\}$.
 - 10 Call $\text{queue.ChangeKey}(v_j, \text{dist}[j])$,
 - 11 **while** queue is not empty
 - 12 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 $\log |V| \lesssim 50$, so let's not!