WS 2021/22

Algorithms

Chapter 4.4
Dijkstra's algorithm

Martin Dietzfelbinger

February 2022

Section 4.4: Dijkstra's Algorithm

Please read the introductory remarks in the book (pages 108–110).

Here we use slightly different notation, but in principle it's exactly the same algorithm.

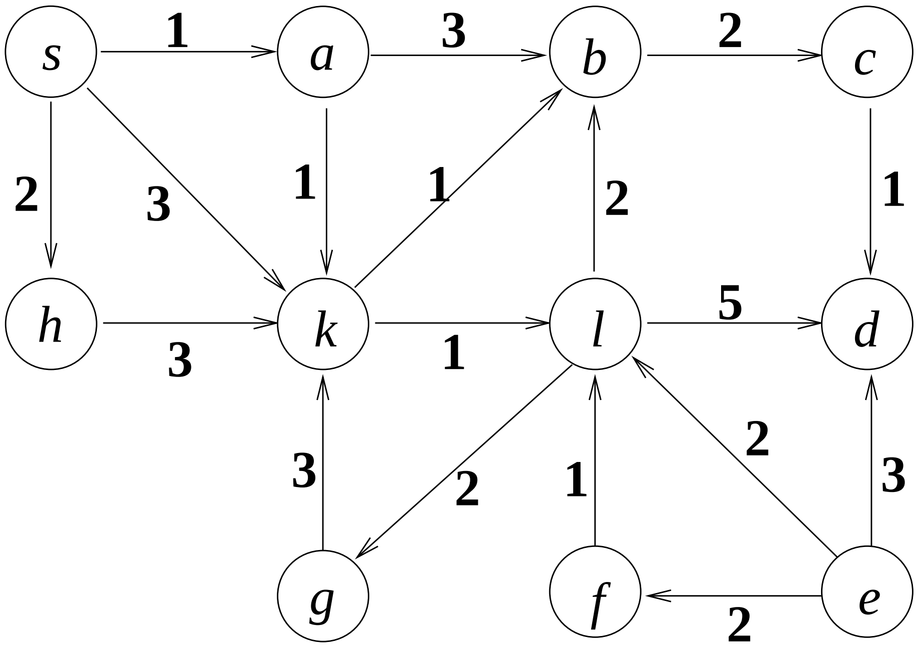
Shortest paths with one start node: Dijkstra's algorithm

Definition 4.4.1

1. A *weighted digraph* is a triple $G = (V, E, c)$,
   where (V, E) is a digraph and $c\colon E \to \mathbb{R}$ is a function.
   $c(v, w)$ can be interpreted as "cost" or "length" or "weight" of edge (v, w).
2. A directed walk $p = (v_0, v_1, \ldots, v_k)$ in G has **cost**/**length**

$$c(p) = \sum_{i=1}^{k} c(v_{i-1}, v_i).$$

Example: Nonnegative edge weights.



$$c((s,a,b,c)) = 6, \quad c((s,a,k,b,c)) = 5, \quad d(s,c) = 5;$$
$$d(s,s) = 0;$$
$$d(s,e) = d(s,f) = \infty.$$

3. The **(directed) distance** of nodes $v, w \in V$ is

$$d(v,w) := \text{min}\{c(p) \mid p \text{ walk from } v \text{ to } w\}$$

(= ∞ if there is no such walk;
$= -\infty$ if there are walks from v to w with negative costs of arbitrarily large absolute value.)

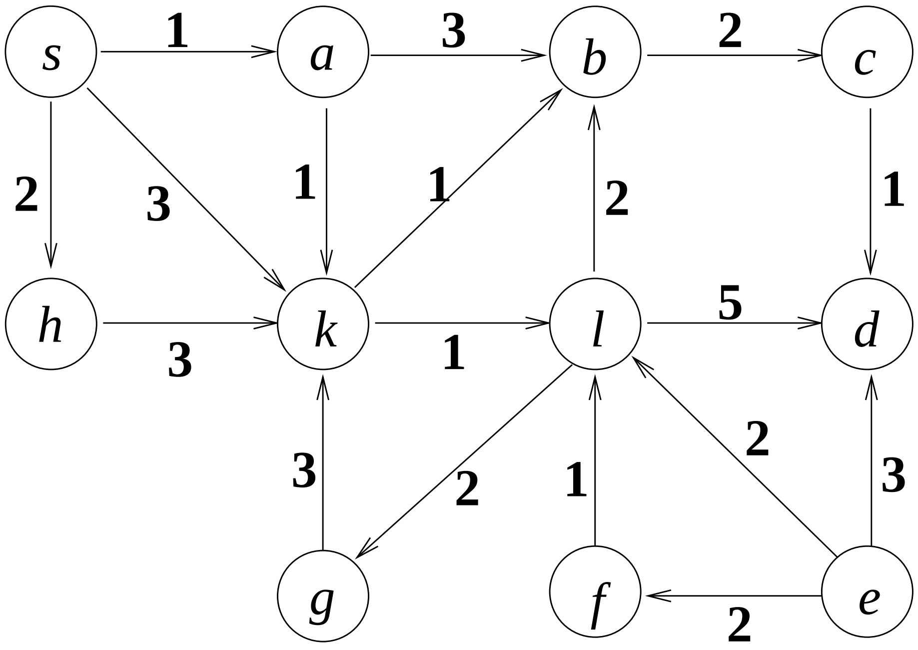
Obvious: $d(v,v) \le 0$. (Walk with no edge.)

Remark

All edge weights are $\ge 0 \Rightarrow$
$d(v,w) =$ minimal length of a **path** from v to w.

(One can take an arbitrary walk from v to w and cut out cycles without increasing the cost.)

Example: Nonnegative edge weights.



$c((s,a,b,c)) = 6, \quad c((s,a,k,b,c)) = 5, \quad d(s,c) = 5;$
$d(s,s) = 0;$
$d(s,e) = d(s,f) = \infty.$

Dijkstra's* **Algorithm** solves the problem

"Single-Source-Shortest-Paths"

(Shortest paths from one starting node)

Given: Weighted digraph $G = (V, E, c)$ with **nonnegative edge lengths** and start node $s \in V$.

Task: For each $v \in V$ find distance $d(s, v)$ and in case $d(s, v) < \infty$ find a path from s to v of length $d(s, v)$.

* Pronunciation: "dike-stra".
Edsger W. Dijkstra (1930–2002), Dutch computer scientist, pioneer of the "science of programming".)

(Sparkling) Idea:

An edge (v, w) is thought as a "one-way fuse" of length $c(v, w)$.

Spark advances along a fuse with constant speed 1 [m/s] (or [km/h] or [miles/h]).

At time $t_0 = 0$ we ignite node s.

All fuses that correspond to edges (s, v) start burning.

The next interesting event:
At time $t = \min\{c(s, v) \mid (s, v) \in E\}$ the spark reaches (at least) one other node v.

We say: When the spark reaches v, all fuses that belong to edges (v, w) start to burn, without delay.

When a spark reaches v later, on another edge/fuse, nothing happens with v.

Numbers in nodes: When does the fire reach v, *by current information*?
A **green** edge into a node v that has not yet been reached indicates from which direction v will be *first reached*. (These edges must be watched. This information can change.)

"Intuitively clear": The fire reaches v exactly at time $d(s, v)$.

Namely: The time span $[0, d(s, v)]$ is exactly the time in which the fire can walk along a shortest path from s to v, not faster, not slower.

Unfortunate: One cannot really carry out this algorithm. Afterwards the network is reduced to ashes. So we simulate! This gives an algorithm.

Observation: Only the n points in time $d(s, v)$, $v \in V$, are interesting. (If the fire reaches v again after $d(s, v)$, nothing happens.)

Thus our algorithms has at most $n = |V|$ rounds.

In between the sparks walk along the fuses/edges without anything happening.
(In the book they say one can go to sleep and has to wake up only when an alarm rings.)

W.l.o.g.: $V = \{1, \ldots, n\}$.

The algorithm works in up to n rounds, one for each *reachable* node v.

Data structure:

V is split in two disjoint sets S and $V - S$.

(Nodes in S: already reached; nodes in $V - S$: not yet reached.)

Initialization: $S \leftarrow \emptyset$.

Array $\texttt{dist}[1..n]$ stores times.

For $v \in S$: $\texttt{dist}[v] = d(s,v)$. (Time when fire has reached v.)

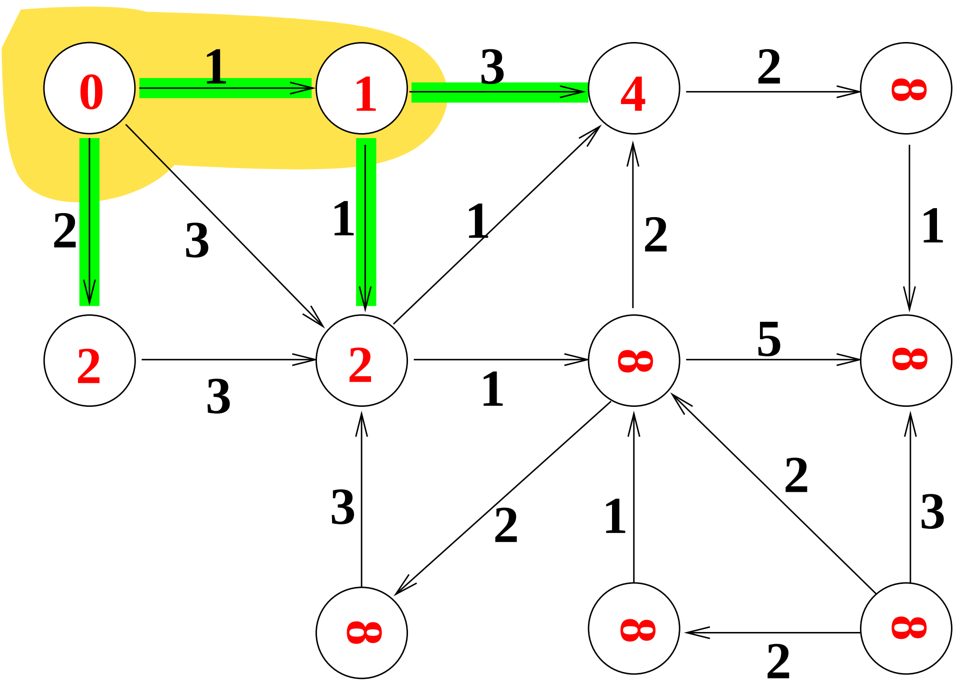
For $v \notin S$: $\texttt{dist}[v] =$ the point in time when fire will reach v
according to the information currently available
$= \min\{\texttt{dist}[w] + c(w,v) \mid w \in S, (w,v) \in E\}$.

(If there is no edge (w,v) with $w \in S$, we have $\texttt{dist}[v] = \infty$.)

Initialization: $\texttt{dist}[s] \leftarrow 0$.

For all $v \neq s$: $\texttt{dist}[v] \leftarrow \infty$.

Example: **Orange:** S.



Red numbers in the nodes are the `dist[·]` values.

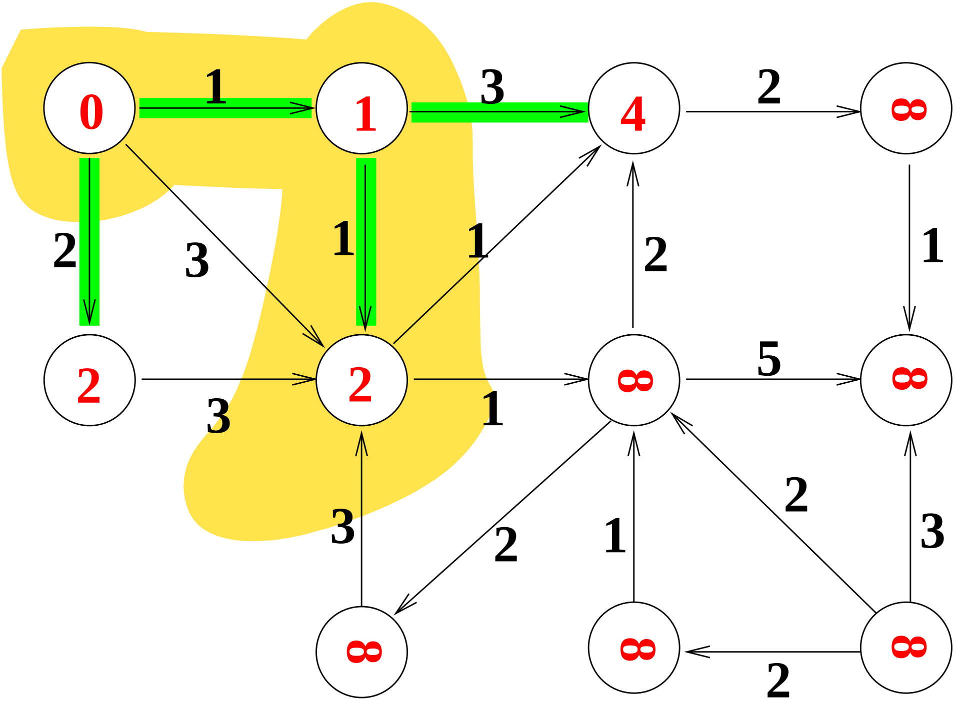
Round:

Find some node $u \in V - S$ that minimizes `dist`$[v]$, $v \in V - S$ (need not be unique).

Add u to S. (Edges out of u start burning.)

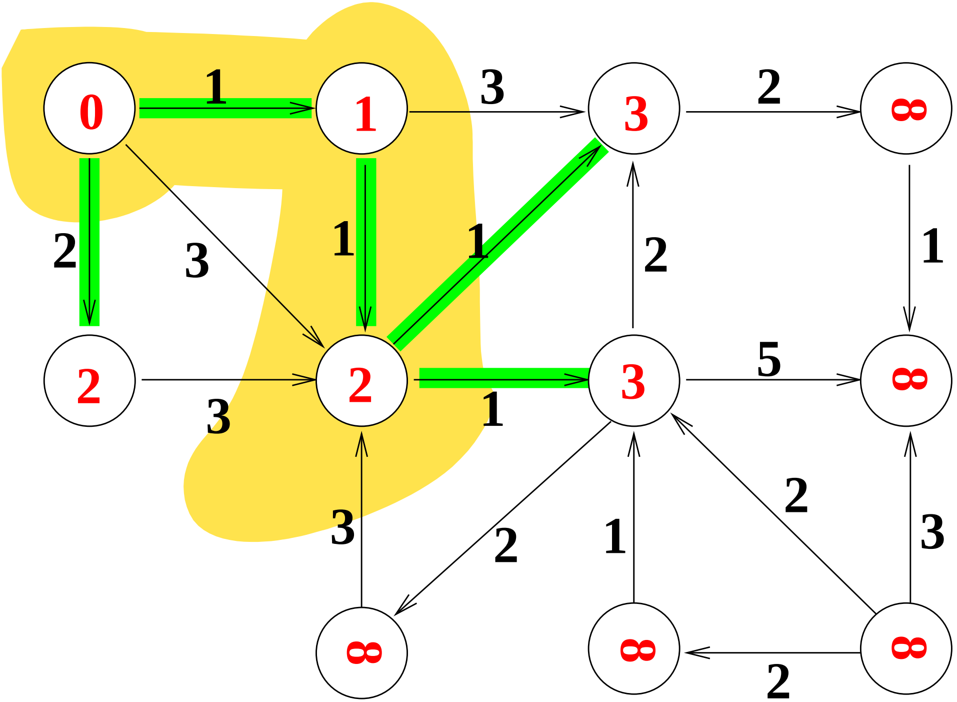
The current value `dist`$[u]$ is "frozen" and will not change anymore.

Example:



What else has to be done?

Example:



What else has to be done?
On edge (u, v) could a node $v \in V - S$ be reached that was not reachable before, or $v \in V - S$ can be reached faster.

For such nodes v we must check if $\texttt{dist}[u] + c(u,v) < \texttt{dist}[v]$, and if so, update the dist value:

$$\texttt{dist}[v] \leftarrow \min\{\texttt{dist}[v], \texttt{dist}[u] + c(u,v)\}.$$

(The spark now also walks along the edge (u,v).)

If $\texttt{dist}[v]$ switches from ∞ to some finite value, we say that v is **found** in this round.

The algorithm as described so far calculates the **lengths** of the shortest paths from s to all other nodes (the *times* at which the fire reaches the nodes, or ∞ if unreachable).

Algorithm DijkstraDistances(G, s) // rough version

Input: weighted digraph $G = (V, E, c)$ with $c(e) \geq 0$, $V = \{1, \ldots, n\}$, start node s

Output: lengths of the shortest paths from $\mathtt{s}$ to the nodes in G

Data structure: Array $\mathtt{dist}[1..n]$

(1) $\mathtt{S} \leftarrow \emptyset$; // (1)–(3): Initialization

(2) $\mathtt{dist[s]} \leftarrow 0$;

(3) **for** $\mathtt{v} \in V - \{\mathtt{s}\}$ **do** $\mathtt{dist[v]} \leftarrow \infty$;

(4) **while** $\exists u \in V - S$: $\mathtt{dist}[u] < \infty$ **do** // a round, in which u is “scanned”

(5) $\quad$ $\mathtt{u} \leftarrow$ one such node u with minimal $\mathtt{dist}[u]$;

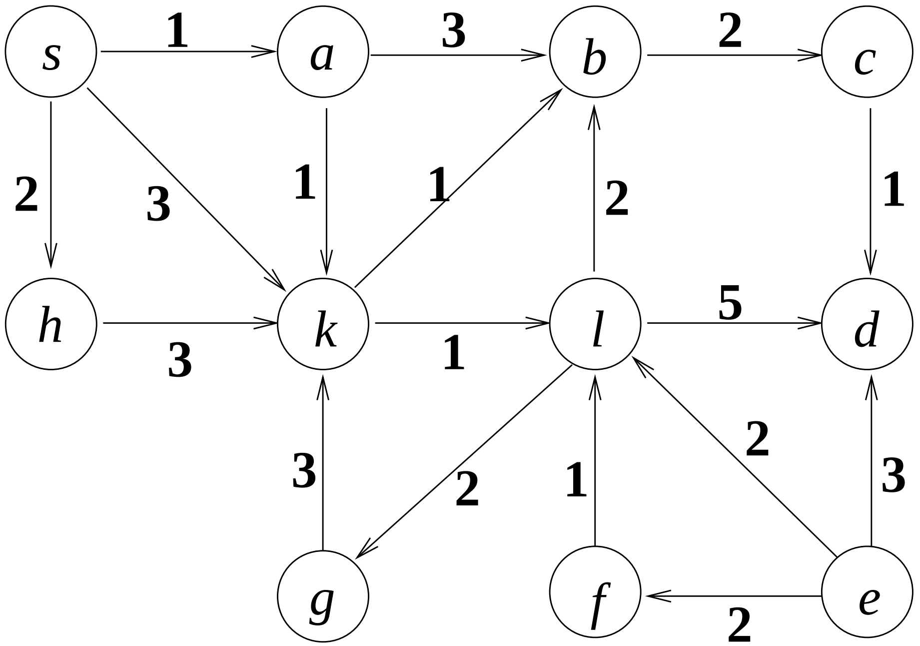
(6) $\quad$ $\mathtt{S} \leftarrow \mathtt{S} \cup \{\mathtt{u}\}$;

(7) $\quad$ **for** $\mathtt{v} \in V - S$ with $(\mathtt{u}, \mathtt{v}) \in E$ **do**

(8) $\quad\quad$ $\mathtt{dist[v]} \leftarrow \min\{\mathtt{dist[v]}, \mathtt{dist[u]} + c(\mathtt{u}, \mathtt{v})\}$;

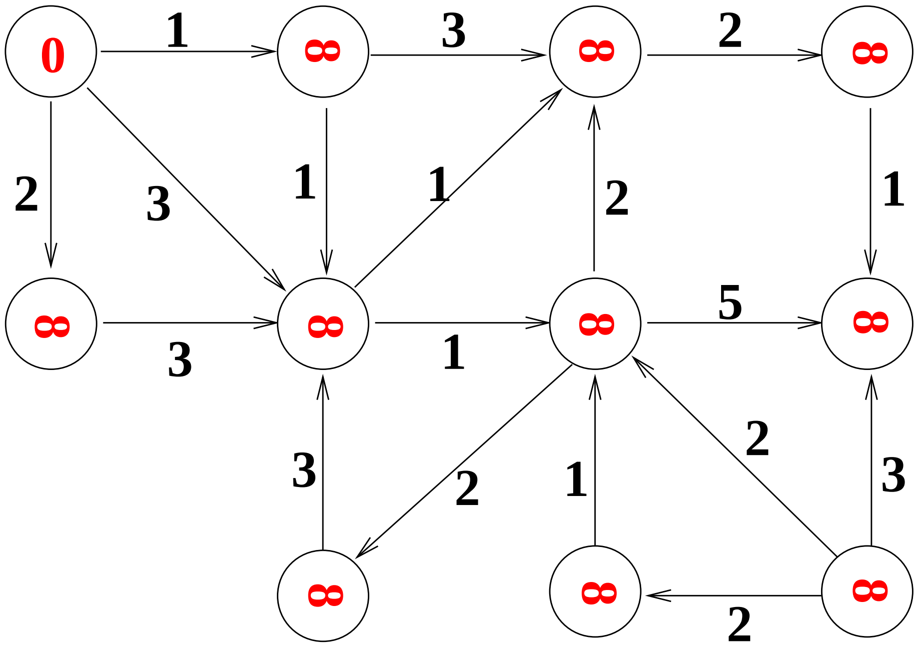
(9) **return** $\mathtt{dist}[1..n]$.

Dijkstra's algorithm, rough version, in action



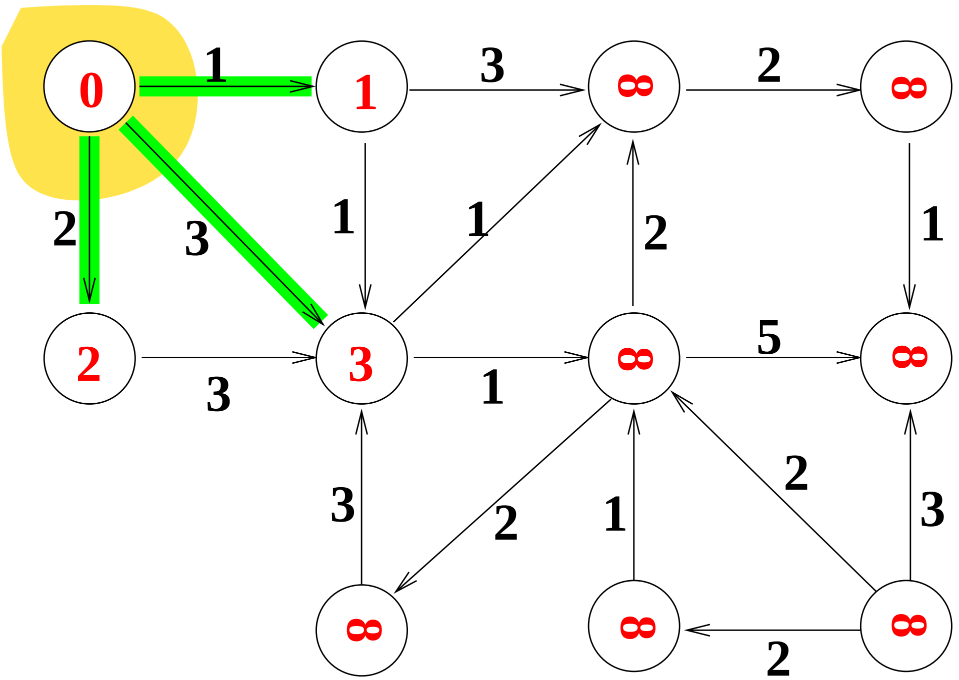
The input digraph.

Dijkstra's algorithm, rough version, in action



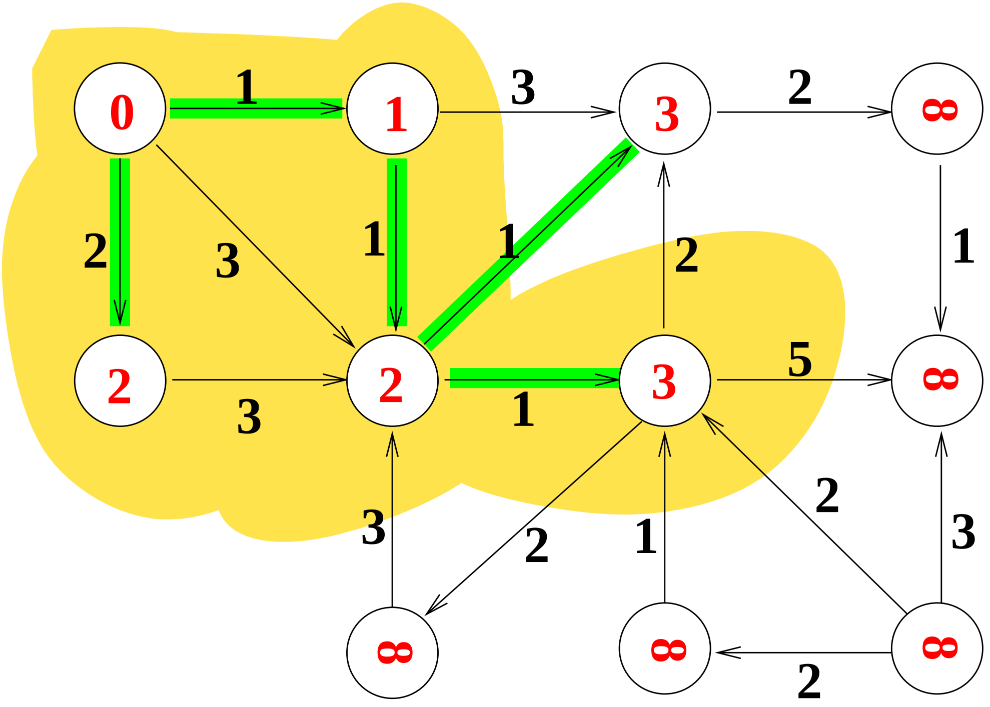
After initialization.

Dijkstra's algorithm, rough version, in action



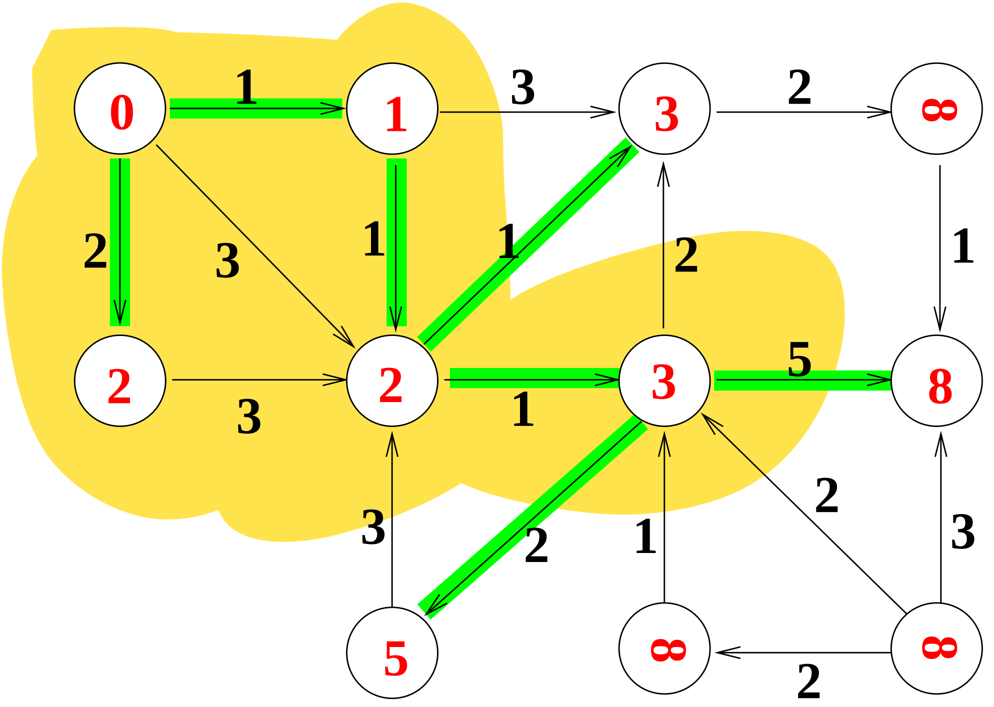
Scanning $u = s$. Nodes a, h, k are found.

Dijkstra's algorithm, rough version, in action



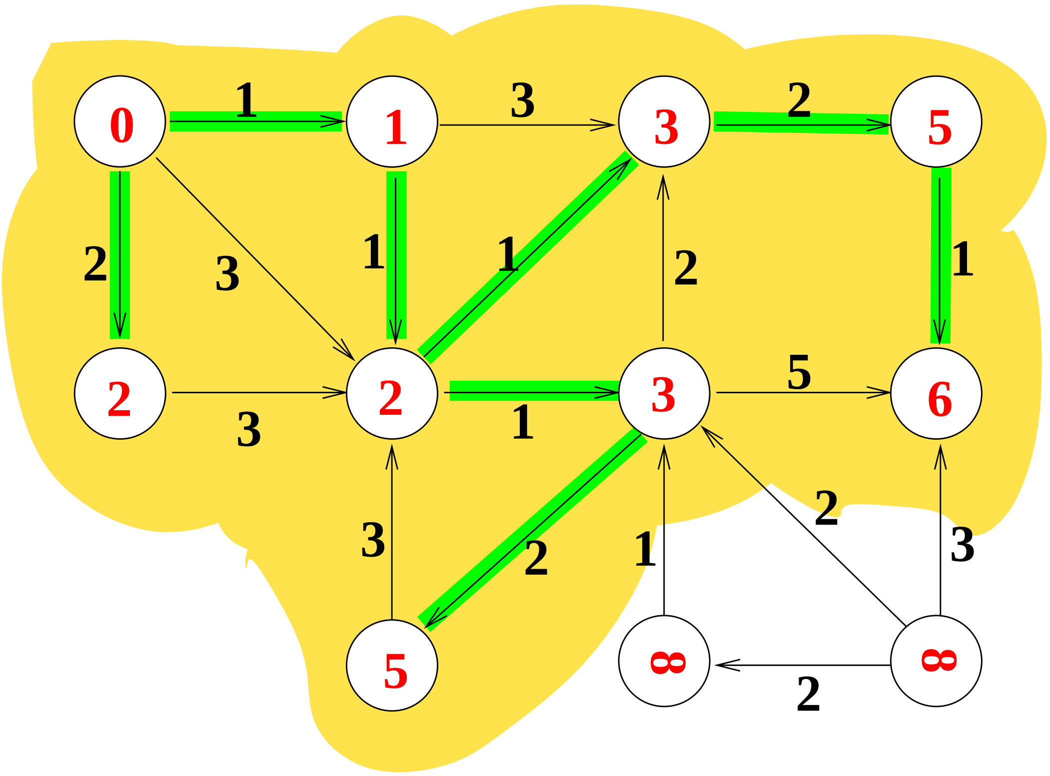
$u = l$ is chosen.

Dijkstra's algorithm, rough version, in action



Scanning $u = l$. Two new nodes are found.

Dijkstra's algorithm, rough version, in action



All $v \in V - S$ satisfy `dist`$[v] = \infty$: **algorithm ends**.

Lemma 4.4.2 Algorithm **DijkstraDistances** outputs, in `dist`$[1..n]$, the value $d(s,v)$, for all $v \in V$.

Proof:

If node u is considered in lines (5)–(8), we say that u is **scanned**.
(Actually, the *edges out of* u are scanned.)

If `dist`$[v]$ is set to a value $< \infty$ for the first time, in line (2) or (8), we say that v is **found**.

We first deal with the simple case of the unreachable nodes.

By a simple induction, as in BFS, one can show that every node v that can be reached from s on a path (or walk) will be found at some time and will be scanned at some later time.

Nodes v not reachable from s will never be found, and they keep the value `dist`$[v] = \infty$.

Now we show the following invariants, which are valid at the end of each round.

(I1) $\forall v \in V\colon$ $\texttt{dist}[v] < \infty$
$\Rightarrow$ there is a path from s to v of length at most $\texttt{dist}[v]$.

(I2) $\forall v \in S\colon$ $\texttt{dist}[v] = d(s, v)$.

Proof of (I1) by induction over rounds:

After initialization we have $\texttt{dist}[v] < \infty$ only for $v = s$, and there is a path from s to s of length 0.

Now consider a round in which u is scanned, and a node v. If $\texttt{dist}[v]$ does not change in the round, there is nothing to show. So assume $\texttt{dist}[v]$ changes. This implies $v \in V - S$.

$\texttt{dist}[v]$ is changed to the new value $\texttt{dist}[u] + c(u, v)$.

By induction hypothesis there is a path p_u from s to u of length at most $\texttt{dist}[u]$. By extending p_u by edge (u, v) we obtain a path from s to v of length at most $\texttt{dist}[v]$.

Proof of (I2) by induction over rounds:

Basis: At the beginning S is empty. In the first round s is put into S, and there is a path of length $\texttt{dist}[s] = 0$ from s to s (the path with no edge). On the other hand we have $d(s, s) = 0$, since all edges have nonnegative weight, and walks with cycles do not help.

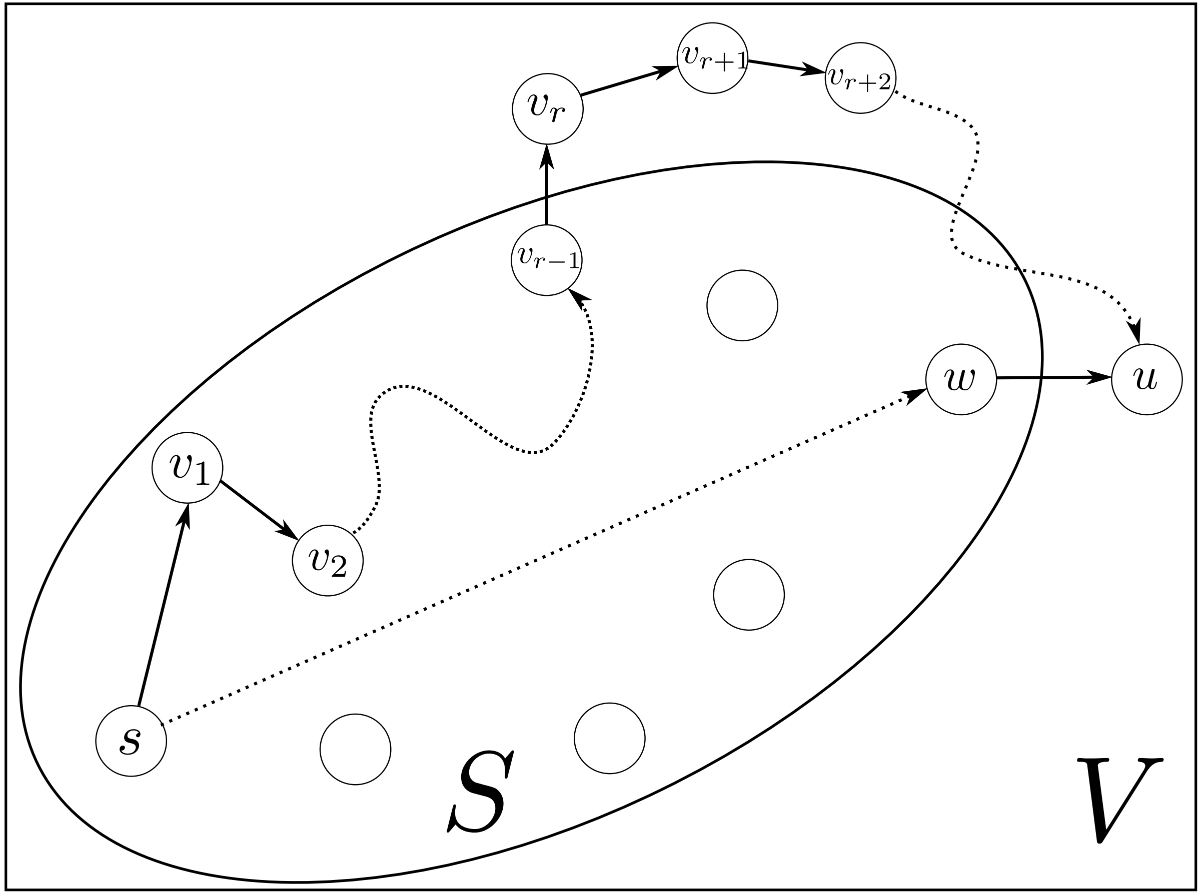
Induction step: Consider a round in which $u \neq s$ is scanned.

By (I1) there is a path from s to u of length at most $\texttt{dist}[u]$.

We must show:
There is no path from s to u shorter than $\texttt{dist}[u]$.

Let $p = (s = v_0, v_1, \ldots, v_t = u)$ be an arbitrary path from s to u.

p starts in $v_0 = s \in S$ and ends in $v_t = u \in V - S$, hence there is some r such that $s = v_0, v_1, \ldots, v_{r-1} \in S$ and $v_r \notin S$.



v_{r+1}
v_{r+2}
v_r
v_{r-1}
w
u
v_1
v_2
s
S
V

We consider the initial segment $p_{r-1} = (v_0, \ldots, v_{r-1})$ of p.

By the induction hypothesis for $v_{r-1} \in S$ we have $d(s, v_{r-1}) = \texttt{dist}[v_{r-1}]$.

By the definition of the distance function we have $d(s, v_{r-1}) \le c(p_{r-1})$.

So: $\texttt{dist}[v_{r-1}] + c(v_{r-1}, v_r) \le c(p_r)$, for the initial segment $p_r = (v_0, \ldots, v_r)$ of p.

By the general assumption all **edge weights** are **nonnegative**.
In particular: $c(p) \ge c(p_r)$.

Thus:

$(*) \quad c(p) \ge \texttt{dist}[v_{r-1}] + c(v_{r-1}, v_r)$.

Furthermore:

$$(**) \quad \texttt{dist}[v_{r-1}] + c(v_{r-1}, v_r) \geq \texttt{dist}[v_r].$$

Why is this so? We go back to the round in which v_{r-1} was scanned. In that round the algorithm compared $\texttt{dist}[v_{r-1}] + c(v_{r-1}, v_r)$ and $\texttt{dist}[v_r]$, and $(**)$ is enforced. Afterwards $\texttt{dist}[v_r]$ may change, but it can only decrease.

Finally we observe:

$$(***) \quad \texttt{dist}[v_r] \geq \texttt{dist}[u].$$

This is because the algorithm chooses a node in $V - S$ with minimal $\texttt{dist[.]}$-value as u, and v_r is qualified for the competition.

Combining $(*)$, $(**)$, and $(***)$ gives $c(p) \geq \texttt{dist}[u]$. $\square$

Actually, we do not only want to calculate distances $d(s, v)$, but also find **shortest paths** from s to all other nodes.

Idea: For each node v we record the edge (w, v) on which "the spark" has reached node v.

If we start in v and walk back step by step according to this **"predecessor"** information we get a shortest path.

Technically, for each node $v \notin S$ that has been found take down $w = p(v) \in S$ with $(w, v) \in E$ and $\mathtt{dist}[v] = \mathtt{dist}[w] + c(w, v)$. Whenever $\mathtt{dist}[v]$ is decreased, update $p(v)$.

Data structure: $\mathtt{p}[1\,..\,n]$.

We modify the algorithm as follows:

(2+). . . $\texttt{p[s]} \leftarrow -2$; // the root s is a special case: this value never changes

(3+) **for** $\texttt{v} \in V - \{\texttt{s}\}$ **do** . . . $\texttt{p[v]} \leftarrow -1$; // "undefined"

Update in later rounds:

(7) **for** $\texttt{v} \in V - S$ with $(u, \texttt{v}) \in E$ **do**
(8a) $\texttt{dd} \leftarrow \texttt{dist}[u] + c(u, \texttt{v})$;
(8b) **if** $\texttt{dd} < \texttt{dist[v]}$ **then**
(8c) $\texttt{dist[v]} \leftarrow \texttt{dd}$;
(8d) $\texttt{p[v]} \leftarrow u$;

The operation in lines (7)–(8d) is known as **update**`(u)` or **relax**`(u)`.

Algorithm DijkstraTree(G, s)

Input: weighted digraph $G = (V, E, c)$ with $c(e) \geq 0$, $V = \{1, \ldots, n\}$, start node s

Output: length $d(s, v)$ of the shortest paths, predecessor $p(v)$ on shortest path

(1) $\texttt{S} \leftarrow \emptyset$;

(2+) $\texttt{dist[s]} \leftarrow 0$; $\texttt{p[s]} \leftarrow -2$;

(3+) **for** $\texttt{v} \in V - \{\texttt{s}\}$ **do** $\texttt{dist[v]} \leftarrow \infty$; $\texttt{p[v]} \leftarrow -1$;

(4) **while** $\exists u \in V - S$: $\texttt{dist}[u] < \infty$ **do**

(5) $\quad$ $\texttt{u} \leftarrow$ one such node u that minimizes $\texttt{dist}[u]$;

(6) $\quad$ $\texttt{S} \leftarrow \texttt{S} \cup \{\texttt{u}\}$;

(7) $\quad$ **for** $\texttt{v} \in V - S$ with $(\texttt{u}, \texttt{v}) \in E$ **do**

(8a) $\quad\quad$ $\texttt{dd} \leftarrow \texttt{dist[u]} + c(\texttt{u}, \texttt{v})$;

(8b) $\quad\quad$ **if** $\texttt{dd} < \texttt{dist[v]}$ **then**

(8c) $\quad\quad\quad$ $\texttt{dist[v]} \leftarrow \texttt{dd}$;

(8d) $\quad\quad\quad$ $\texttt{p[v]} \leftarrow \texttt{u}$;

(9+) **return** $\texttt{dist}[1\,..\,n]$ and $\texttt{p}[1\,..\,n]$.

Since exactly the reachable nodes are found we have that $\texttt{p}[v] \neq -1$ ("undefined") holds at the end if and only if $\texttt{dist}[v] < \infty$.

$\texttt{p}[v] = -2$ is true only for $v = s$.

Definition A path $(s = v_0, v_1, \ldots, v_t)$ is called an ***S*-path** if all nodes excepting maybe v_t are in S.

Claim: In addition to (I1) and (I2) Dijkstra's algorithm maintains the following invariants:

(I3) If $v \in S$ then $\texttt{p}[v] \neq -1$ and if in addition $v \neq s$ then $\texttt{p}[v]$ is the second-to-last node on a path from s to v that runs completely in S and has length $d(s, v)$.

(I4) If $v \notin S$ and $\texttt{dist}[v] < \infty$ then: $\texttt{p}[v] \in S$ and $\texttt{p}[v]$ the last S-node on an S-path from s to v of shortest length $\texttt{dist}[v]$.

Proof of (I3) and (I4): Induction on rounds. (Omitted.)

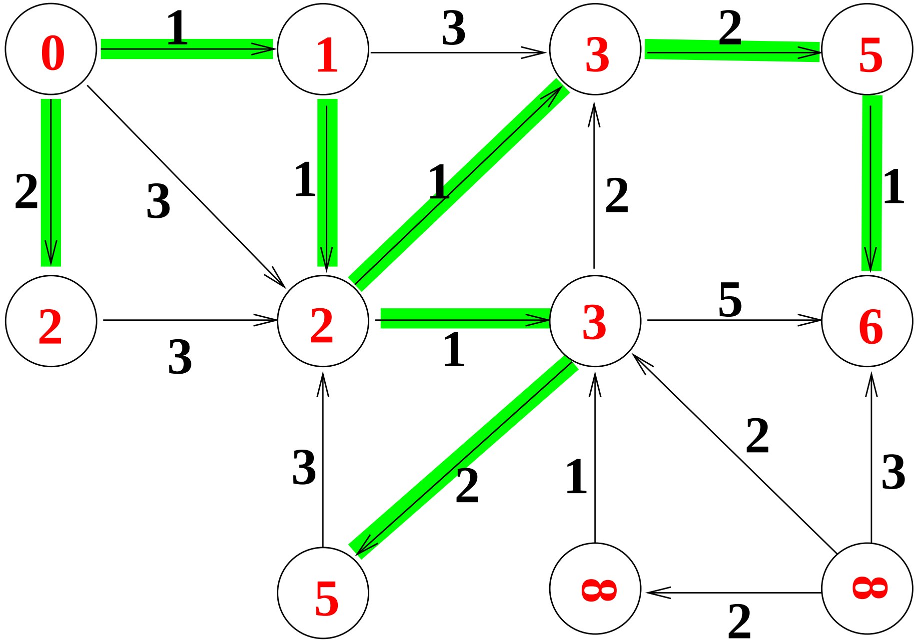
Result:

When Dijkstra's algorithms stops, iteratively following the $\mathtt{p}[v]$-pointers starting from w until s is reached will give a shortest path from s to w (in opposite direction).

The $\mathtt{p}[v]$-pointers cannot form a cycle, since in each round a new node u is attached to S, and the edge $(\mathtt{p}[u], u)$ is fixed forever, the edges $(\mathtt{p}[v], v)$ form a **tree** with root s, the so-called

shortest path tree.

Example: A shortest-path tree.



Implementation details:

In a round, how do we **efficiently** find u with smallest value `dist`$[u]$?

Very simple solution: In each round scan the `dist`-array to find the node v with minimum value `dist`$[v]$ among nodes in $V - S$.

Then each of the up to n rounds takes time $\Theta(n)$, and the total running time of Dijkstra's algorithm will be $\Theta(n^2)$, quadratic.

For "dense" graphs, meaning graphs with a number of edges close to n^2, this is acceptable and actually not bad. If, however, we have a graph with $|E| \ll |V|^2$, quadratic running time is not good. One can do better.

Efficient alternative: A clever data structure.

We maintain the set $v \in V - \mathtt{S}$ with values ("keys") $\mathtt{dist}[v] < \infty$ in a

$$\boxed{\textbf{priority queue PQ}.}$$

A priority queue (for graph nodes) can be imagined to be a (variable) set of pairs (v, k), $v \in V$, $k \in \mathbb{R}_0^+$, with the following operations (i.e., methods):

- **init**(): Initializes PQ to the empty set.
- **insert**(v, k): Inserts a node $v \in V$ plus a key $k \in \mathbb{R}_0^+$.
- **extractMin**: Remove from PQ a pair (u, k) with minimum k, and return node u.
- **isempty** returns *true* if PQ is empty and *false* if PQ is not empty.
- **decreaseKey**(v, ℓ): assumes that (v, k) is in PQ, with $\ell < k$. (Otherwise illegal use of this operation.) Replace (v, k) by (v, ℓ) in PQ.

Fact: One can implement a priority queue for graph nodes in such a way that

- **init**$()$ takes time $O(n)$.
- **insert**(v, k), **extractMin**, **decreaseKey**(v, ℓ) take time $O(\log n)$.
- **isempty** takes time $O(1)$.

The name of an implementation with these properties is „binary heap".

It is described in Section 4.5 in the book.

DijkstraFullWithPQ(G, s)

Input: weighted digraph $G = (V, E, c)$ with $c(e) \geq 0$, $V = \{1, \ldots, n\}$, start node s;
Output: length $d(s, v)$ of shortest paths, predecessor nodes $p(v)$, for nodes v reachable from s
auxiliary data structures: **PQ**: a priority queue for nodes; `p`, `dist`: as before; `inS`$[1..n]$: boolean

for `v` **from** 1 **to** n **do**
    `dist[v]` $\leftarrow \infty$; `inS[v]` $\leftarrow$ *false*; `p[v]` $\leftarrow -1$;
PQ.init(); // set up empty priority queue
`dist[`s`]` $\leftarrow 0$; `p[`s`]` $\leftarrow -2$; **PQ.insert(**s**)**;
while not PQ.isempty do
    `u` $\leftarrow$ **PQ.extractMin()**; `inS[u]` $\leftarrow$ *true*; // now "scan" `u`
    for node `v` with $($`u`$,$ `v`$) \in E$ **and not** `inS[v]` **do**
        `dd` $\leftarrow$ `dist[u]` $+ c($`u`$,$ `v`$)$;
        if `p[v]` ≥ 0 **and** `dd` $<$ `dist[v]` **then**
            PQ.decreaseKey(`v`,`dd`**)**; `p[v]` $\leftarrow$ `u`; `dist[v]` $\leftarrow$ `dd`;
        if `p[v]` $= -1$ **then** // `v` not found before
            `dist[v]` $\leftarrow$ `dd`; `p[v]` $\leftarrow$ `u`; **PQ.insert(**`v`**)**;// "find" `v`
Output: `dist[`$1..n$`]` and `p[`$1..n$`]`.

Cost of Dijkstra's algorithm, with PQ realized as binary heap:

Maximum number of entries in PQ: $n-1$.

Initialization: $O(n)$.

Let V' = set of reachable nodes, $n' = |V'| \le m+1$.

There are $\le n'$ executions of the **while** loop (1 execution = "scanning" a node).

One loop execution costs time $O(1)$ for loop organization plus $O(\log n)$ for **extractMin** plus the time for looking at the edges out of u (in u). Each edge causes cost $O(\log n)$ (for **insert** or for **decreaseKey**). The number of such edges is $\mathrm{outdeg}(u)$, so the total time for scanning u is $O((1+\mathrm{outdeg}(u))\log n)$.

Initialization plus summing over all reachable u gives time bound $O(n+(n'+m)\log n) = O(n+m\log n)$.

Theorem 4.4.3 **Dijkstra's algorithm** with a priority queue, implemented as a binary heap, finds shortest paths from start node s in a weighted digraph with nonnegative edge weights in time $O(n+m\log n)$.

Variants of heaps with running times, Box at the end of 4.4.3

With simple scan to find minimum `dist`-value: $O(n^2)$.

With binary heap as priority queue: $O(n + m \log n)$.

With "Fibonacci heap" (which offers cheaper decreaseKey operations): $O(m + n \log n)$.

With "d-ary heaps", $d = m/n$: $O(m \cdot \frac{\log n}{\log(m/n)})$.
This is $O(n \log n)$ for $m = O(n)$ and it is $O(m)$ for $m = n^{1+\varepsilon}$, for any constant $\varepsilon > 0$. This is almost as good as Fibonacci-Heaps.