Dijkstra’s shortest paths algorithm

Vassos Hadzilacos

Shown below is pseudocode for Dijkstra’s algorithm. The input is a directed graph \( G = (V, E) \) with non-negative edge weights \( \text{wt}(u, v) \) for every edge \( (u, v) \in E \), and a distinguished node \( s \), called the source (or start) node. The algorithm computes, for each node \( u \in V \), the weight of a minimum-weight path from \( s \) to \( u \). (It can be easily modified to compute, for each node \( u \), the predecessor of \( u \) in a minimum-weight path from \( s \) to \( u \), in addition to the weight of such a path.)

1. \( R := \emptyset \)
2. \( d(s) := 0 \)
3. for each \( v \in V - \{s\} \) do \( d(v) := \infty \)
4. while \( R \neq V \) do
5. \hspace{1em} let \( u \) be a node not in \( R \) with minimum \( d \)-value (i.e., \( u \in V - R \) and \( \forall u' \in V - R, d(u) \leq d(u') \))
6. \hspace{1em} \( R := R \cup \{u\} \)
7. \hspace{1em} for each \( v \in V \) such that \( (u, v) \in E \) do
8. \hspace{2em} if \( d(u) + \text{wt}(u, v) < d(v) \) then \( d(v) := d(u) + \text{wt}(u, v) \)

Intuitively Dijkstra’s algorithm works as follows. It maintains a set \( R \) (the “explored” part of the graph, consisting of nodes to which it has determined the weight of shortest paths). We say that an \( s \to u \) path is an \( R \)-path (to \( u \)) if every node on the path except (possibly) \( u \) is in the set \( R \). The algorithm also maintains, for every node \( u \), a label \( d(u) \), which is the minimum weight of \( R \)-paths to \( u \). The algorithm starts with an empty \( R \), and it greedily expands the set \( R \) with the node \( u \) that is not presently in \( R \) and has minimum \( d \)-value. The algorithm then updates the \( d \)-values of the nodes adjacent to \( u \) to account for the fact that there may now exist shorter \( R \)-paths to these nodes, going through \( u \). When \( R \) contains all nodes, any \( s \to u \) path is an \( R \)-path to \( u \) and so \( d(u) \) is the minimum weight of \( s \to u \) paths, which is what we want to compute.

We now prove the correctness of the algorithm. \( X_i \) denotes the value of a variable \( X \) at the end of iteration \( i \) of the while loop. For every node \( u \), we define \( \delta(u) \) to be the minimum weight of \( s \to u \) paths (\( \infty \) if there is no \( s \to u \) path). The first claim states that the \( d \)-value of every node does not increase in time.

**Claim 1** For every node \( v \) and iterations \( i, j \), if \( i \leq j \) then \( d_i(v) \geq d_j(v) \).

**Proof.** After being initialized (in line 1 or 2), the value of \( d(v) \) is changed only in line 8, where it is obviously assigned a smaller value than before.

**Claim 2** If node \( u \) is added to \( R \) in iteration \( i \) the value of \( d(u) \) does not change in iteration \( i \).

**Proof.** Suppose for contradiction that \( u \) is added to \( R \) in iteration \( i \) and \( d(u) \) changes in iteration \( i \). Thus, by the algorithm, \( (u, u) \) is an edge and \( d_{i-1}(u) + \text{wt}(u, u) < d_{i-1}(u) \) (where \( d_0(u) \) — i.e., if \( i = 1 \) — refers to the initial value of \( d(u) \)). But then \( \text{wt}(u, u) < 0 \), contradicting the assumption that weight of every edge is non-negative.
Claim 3. For every node \( v \) and iteration \( i \), if \( d_i(v) = k \neq \infty \) then there is an \( R_i \)-path to \( v \) of weight \( k \).

Proof. By induction on the iteration number \( i \geq 0 \). For the basis \( i = 0 \), i.e., just before we start the while loop, we have \( R_0 = \emptyset \), \( d_0(s) = 0 \), and \( d_0(u) = \infty \) for every node \( u \neq s \). It is true that there is an \( s \rightarrow s \) \( R_0 \)-path of weight 0.

For the induction step, let \( i > 0 \) and suppose the claim holds at the end of iteration \( i - 1 \). Let \( u \) be the node added to \( R \) in iteration \( i \) and consider any node \( v \). If \( d(v) \) does not change in iteration \( i \), the claim holds after iteration \( i \) by induction hypothesis. If \( d(v) \) changes in iteration \( i \) and since (by Claim 2) \( d_{i-1}(u) = d_i(u) \), by the algorithm \( d_i(v) = d_{i-1}(u) + \text{wt}(u, v) \) and \( d_{i-1}(u) \neq \infty \). By the induction hypothesis, there is an \( R_{i-1} \)-path to \( u \) of weight \( d_{i-1}(u) \), say path \( p \). Since \( R_i = R_{i-1} \cup \{ u \} \), path \( p \) followed by the edge \((u, v)\) is an \( R_i \)-path to \( v \), whose weight is \( \text{wt}(p) + \text{wt}(u, v) = d_{i-1}(u) + \text{wt}(u, v) = d_i(u) + \text{wt}(u, v) \).

So, the claim holds after iteration \( i \).

Claim 4. For every node \( u \), if \( u \) is added to \( R \) in iteration \( i \) and \( d_i(u) = \infty \) then there is no \( s \rightarrow u \) path.

Proof. Suppose, for contradiction, that some node \( u \) is added to \( R \) in iteration \( i \) and \( d_i(u) = \infty \) but there is an \( s \rightarrow u \) path. Without loss of generality, assume that \( i \) is the earliest iteration in which this happens. Since \( s \) is added to \( R \) in iteration 1 and \( d_1(s) \neq \infty \), \( i > 1 \). So \( s \in R_{i-1} \) and \( u \notin R_{i-1} \) (since \( u \) is added to \( R \) in iteration \( i \)). Thus, there is an edge \((x, y)\) on the \( s \rightarrow u \) path with \( x \in R_{i-1} \) and \( y \notin R_{i-1} \). Let \( j < i \) be the iteration in which \( x \) was added to \( R \); by the definition of \( i \), \( d_j(x) \neq \infty \) and so by the algorithm \( d_j(y) \neq \infty \). By Claim 1, \( d_{i-1}(y) \neq \infty \). The facts that (a) \( y \notin R_{i-1} \) and (b) \( d_{i-1}(y) \neq \infty \) contradict that in iteration \( i \) the algorithm added to \( R \) the node \( u \) with \( d_{i-1}(u) = \infty \).

Claim 5. For every node \( u \) and every iteration \( i \geq 1 \), if \( u \) is added to \( R \) in iteration \( i \), then \( d_i(u) = \delta(u) \).

Proof. By complete induction on \( i \). Suppose \( u \) is the node added to \( R \) in iteration \( i \), and suppose the claim holds for all nodes added to \( R \) before iteration \( i \).

If \( i = 1 \), the claim holds since (by the initialization in lines 2–3) the node added to \( R \) in iteration 1 is \( s \) and \( d_1(s) = 0 = \delta(s) \).

If \( i > 1 \), the claim holds by Claim 4 if \( d_i(u) = \infty \). So, suppose \( d_i(u) \neq \infty \). Then by Claim 3 there is an \( R_{i-1} \)-path of weight \( d_i(u) \) from \( s \) to \( u \); therefore \( d_i(u) \geq \delta(u) \). We will now show that \( d_i(u) \leq \delta(u) \), proving that \( d_i(u) = \delta(u) \), as wanted.

Since there is an \( R_{i-1} \)-path to \( u \), there is also a minimum weight \( s \rightarrow u \) path, say \( p \) (refer to Figure 1).

![Figure 1](image_url)

**Figure 1**
Since $u$ is added to $R$ in iteration $i$, $u \notin R_{i-1}$ (recall that $i > 1$, so iteration $i-1$ exists, and $R_{i-1}$ is well defined). Since $s \in R_{i-1}$ and $u \notin R_{i-1}$, $p$ contains an edge $(x, y)$ such that $x \in R_{i-1}$ and $y \notin R_{i-1}$ (it is possible that $y = u$). Let $j$ be the iteration which $x$ was added to $R$, so $j \leq i - 1$. Since $p$ is a minimum-weight $s \rightarrow u$ path, the prefix $p_x$ of $p$ up to node $x$ is a minimum-weight $s \rightarrow x$ path, i.e., $\text{wt}(p_x) = \delta(x)$. We have:

$$d_i(u) = d_{i-1}(u)$$

by Claim 2

$$\leq d_{i-1}(y)$$

by definition of $u$, since $u, y \notin R_{i-1}$

$$\leq d_j(y)$$

by Claim 1, since $j \leq i - 1$

$$\leq d_j(x) + \text{wt}(x, y)$$

by Claim 2 and line 8, since $x$ is added to $R$ in iteration $j$

$$= \delta(x) + \text{wt}(x, y)$$

by the induction hypothesis, since $j < i$

$$= \text{wt}(p_x) + \text{wt}(x, y)$$

since $\text{wt}(p_x) = \delta(x)$, as argued above

$$\leq \text{wt}(p)$$

since all edges have non-negative weight

$$= \delta(u)$$

by definition of $p$

So, $d_i(u) \leq \delta(u)$, as needed to complete the proof that $d_i(u) = \delta(u)$.

The algorithm terminates, since one node is added to $R$ in each iteration. The next theorem states that when the algorithm terminates, it has computed the weight of a minimum-weight $s \rightarrow u$ path, for every node $u$.

Theorem 6 When the algorithm terminates, for every node $u$, $d(u) = \delta(u)$.

Proof. By Claim 5, when $u$ is added to $R$, $d(u) = \delta(u)$. By Claim 1, $d(u)$ cannot later be assigned a larger value, and by Claim 3 it cannot later be assigned a smaller value. So when the algorithm terminates, $d(u) = \delta(u)$.

Running time of Dijkstra’s algorithm. Let $n$ be the number of nodes and $m$ be the number of edges in the graph. The running time of Dijkstra’s algorithm depends on the data structure used to store $d(u)$.

In the simplest implementation, we store $d$ in an array of $n$ elements, one per node, in no particular order. The initialization of $R$ and $d$ takes $O(n)$ time. The while loop is executed $n - 1$ times, because initially $R$ has one node, we add one node to it in each iteration, and the loop ends when all $n$ nodes are in $R$. Each iteration of the loop takes $O(n)$ time (to find the minimum element in array $d$ of a node that is not in $R$, and to update the relevant entries of $d$). So the loop in total takes $O(n^2)$ time. This implementation then takes $O(n) + O(n^2) = O(n^2)$ time.

We can also use a heap to store the $d$-values of the nodes that are not in $R$. Thus we can find a node not in $R$ with the minimum $d$-value by performing an EXTRACTMIN operation, which takes $O(\log n)$ time; and we can update the value of $d$ for a node by performing a CHANGEKEY operation, which also takes $O(\log n)$ time. We perform $n$ EXTRACTMIN operations, one in each iteration of the while loop. We perform at most $m$ CHANGEKEY operations: at most once for each edge $(u, v)$, in the iteration of the while loop in which $u$ is added to $R$. In the initialization we must also perform a BUILD_HEAP operation, to create the initial heap; this takes $O(n)$ time. Thus, the total time required to process all these operations is $O(n) + O(m \log n) = O((m + n) \log n)$. If we assume that there is a path from $s$ to each node, then $m \geq n - 1$, and so the above expression simplifies to $O(m \log n)$.

If the graph is “dense”, i.e., it has (roughly) an edge between every two nodes, then $m = \Theta(n^2)$, and in that case the simple array implementation is actually faster! However, in practice often the graph is “sparse” — typically, each node has a constant or perhaps a logarithmic number of neighbours, so $m = \Theta(n)$ or $m = \Theta(n \log n)$. In this case, the heap implementation of Dijkstra’s algorithm is substantially faster.