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 shortest path from $s$ to $u$. (It can be easily modified to compute, for each node $u$, the predecessor of $u$ in a shortest path from $s$ to $u$, in addition to the weight of such a path.)

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

To discuss the algorithm and its correctness it will be useful to define some terminology. If $R$ is a set of nodes that includes the start node $s$, we say that a path from $s$ to some node $u$ is an $R$-path if every node on the path except (possibly) $u$ is in the set $R$.

Roughly speaking, Dijkstra’s algorithm works as follows. At each stage, it maintains a set of nodes $R$ (the “explored region” of the graph) that includes $s$, and the algorithm has computed a number $d(u)$ for every node $u$ (not only the nodes in $R$). The set $R$ and the numbers $d(u)$ satisfy the following property:

\[\text{for each } u \in V, \ d(u) \text{ is the weight of a shortest } R \text{-path from } s \text{ to } u,\]

\[\text{or } \infty, \text{ if there is no } R \text{-path from } s \text{ to } u.\]  

\[\text{(*)}\]

If this holds at the end of each iteration of the loop then, when the loop terminates, $R = V$ (by the loop exit condition) and so any path is an $R$-path. Therefore, property (*) implies that, when the algorithm ends, $d(u)$ is the weight of a shortest path from $s$ to $u$ — which is what we want to compute.

In each iteration of the loop in lines 6–10, Dijkstra’s algorithm greedily expands the set of nodes $R$ by a node $v$ that has minimum $d$-value among all nodes that are not in $R$ (lines 7–8). Having expanded $R$, however, we may have also expanded the set of $R$-paths: In particular, $R$-paths may now go through node $v$ before reaching their destination. Consequently, the $d$-values of some nodes $u$ may decrease, because now there is a shorter $R$-path from $s$ to $u$ than before, namely, an $R$-path that goes through $v$. To ensure that property (*) remains true, we must recompute the $d$-values of nodes, to account for the addition of node $v$ to $R$.

This is done in lines 9–10. For every edge $(v,u)$ out of the node $v$ we just added to $R$, we compare $d(u)$ (which is the weight of a shortest $R$-path from $s$ to $u$ that does not go through $v$) to $d(v) + \text{wt}(v,u)$ (which is the weight of a new $R$-path from $s$ to $u$, that goes through $v$). If the latter is smaller, we have discovered a better $R$-path from $s$ to $u$ (namely one that goes to $v$ and then to $u$), and we update $d(u)$ accordingly, by setting it to $d(v) + \text{wt}(v,u)$.
The careful reader may wonder whether this rule for updating the \(d\)-values is sufficient. After all, the addition of \(v\) to \(R\) does not only create new \(R\)-paths from \(s\) to \(u\) in which \(v\) appears only once as the predecessor of \(u\). It may also create new \(R\)-paths in which \(v\) appears several times or in which \(v\) appears only once but the next node on the \(R\)-path is not \(u\). We will show (see Fact 1) that for the purposes of updating the \(d\)-values of nodes it suffices to consider only those new \(R\)-paths in which \(v\) appears only once and is the predecessor of \(u\).

**Correctness of Dijkstra’s algorithm.** As we just noted, to prove the correctness of the algorithm it suffices to establish that \((*)\) is an invariant of the while loop. To do so, we must strengthen the invariant.

Let \(R_i\) and \(d_i\) denote, respectively, the set in variable \(R\) and the array in variable \(d\) at the end of the \(i\)-th iteration of the while loop, where “the end of the 0-th iteration” is the point in the program just before the loop starts. We prove by induction on \(i \geq 0\), that if iteration \(i\) exists then

(a) \(s \in R_i\),
(b) for each \(u \in V\), \(d_i(u)\) is the weight of a shortest \(R_i\)-path from \(s\) to \(u\) (or \(\infty\), if there is no \(R_i\)-path from \(s\) to \(u\)), and
(c) for each \(u, u' \in V\), if \(u \in R_i\) and \(u' \notin R_i\) then \(d_i(u) \leq d_i(u')\).

Note that (b) is simply \((*)\); the other two properties are “crutches” for the induction.

The base case, \(i = 0\), is obvious from the initialisation in lines 1–5. For the induction step we assume that (a)–(c) hold, and that iteration \(i + 1\) exists. We will prove that the corresponding properties also hold at the end of iteration \(i + 1\). That is, we will prove that

(a') \(s \in R_{i+1}\),
(b') for each \(u \in V\), \(d_{i+1}(u)\) is the weight of a shortest \(R_{i+1}\)-path from \(s\) to \(u\) (or \(\infty\), if there is no \(R_{i+1}\)-path from \(s\) to \(u\)), and
(c') for each \(u, u' \in V\), if \(u \in R_{i+1}\) and \(u' \notin R_{i+1}\) then \(d_{i+1}(u) \leq d_{i+1}(u')\).

**Proof that (a') holds.** By the induction hypothesis (a), \(s \in R_i\). By line 8, \(R_{i+1} \supset R_i\), so \(s \in R_{i+1}\).

**Proof that (b') holds.** By line 8, \(R_{i+1} = R_i \cup \{v\}\), where \(v\) is a node not in \(R_i\) such that \(d_i(v) \leq d_i(v')\) for every node \(v' \notin V - R_i\).

First suppose that there is no \(R_{i+1}\)-path from \(s\) to \(u\). A fortiori, there is no \(R_i\)-path from \(s\) to \(u\). By the induction hypothesis (b), \(d_i(u) = \infty\). Since there is no \(R_{i+1}\)-path from \(s\) to \(u\), either there is no \(R_i\)-path from \(s\) to \(v\) — in which case, by the induction hypothesis (b), \(d_i(v) = \infty\) — or there is no edge \((v, u)\) in \(E\). Either way, by lines 9–10, \(d_{i+1}(u) = d_i(u)\). Therefore, \(d_{i+1}(u) = \infty\), as wanted.

Next suppose that there is an \(R_{i+1}\)-path from \(s\) to \(u\). We first establish the following:

**Fact 1** There is a shortest \(R_{i+1}\)-path from \(s\) to \(u\) in which \(v\) occurs at most once, and if \(v\) occurs at all then it is the predecessor of \(u\).

**Proof.** Let \(P\) be a shortest \(R_{i+1}\)-path from \(s\) to \(u\) with the fewest occurrences of \(v\). If \(v\) occurs zero times in \(P\), the lemma holds. If \(v\) occurs at least twice in \(P\), then \(P = P_1P_2P_3\), where \(P_1\) is a path from \(s\) to \(v\), \(P_2\) is a non-empty path from \(v\) to \(v\) (i.e., a cycle), and \(P_3\) is a path from \(v\) to \(u\). Therefore, \(P' = P_1P_3\) is an \(R_{i+1}\)-path from \(s\) to \(u\) with \(\text{wt}(P') \leq \text{wt}(P)\) and \(P'\) has fewer occurrences of \(v\) than \(P\), contradicting the definition of \(P\).

Therefore, \(v\) occurs at most once in \(P\). We will now prove that \(v\) is the predecessor of \(u\) in \(P\), as wanted. Suppose, for contradiction, that \(v\) is not the predecessor of \(u\) in \(P\). Let \(v'\) be the successor of \(v\) on \(P\), \(P_1\) be the prefix of \(P\) from \(s\) to \(v\), and \(P_2\) be the suffix of \(P\) from \(v'\) to \(u\). So, \(\text{wt}(P) = \text{wt}(P_1) + \text{wt}(v, v') + \text{wt}(P_2)\).
Note that \( v' \neq u \) (since \( v' \) is the successor of \( v \) in \( P \) and \( u \) is not). Since \( P \) is an \( R_{i+1} \)-path with at most one occurrence of \( v \), it is also the case that \( v' \neq v \). Since every node in \( P \) other than \( u \) is in \( R_{i+1} \), \( v' \in R_{i+1} \). Since every node in \( R_{i+1} \) other than \( v \) is in \( R_i \), \( v' \in R_i \). Since \( v' \in R_i \) and \( v \notin R_i \), by the induction hypothesis (c),

\[
d_i(v') \leq d_i(v) \tag{1}
\]

Let \( P'_1 \) be a shortest \( R_i \)-path from \( s \) to \( v' \). By the induction hypothesis (b), \( d_i(v') = \text{wt}(P'_1) \). Also, \( d_i(v) \leq \text{wt}(P_1) \) (since \( P_1 \) is an \( R_i \)-path from \( s \) to \( v \)). Thus, by (1),

\[
\text{wt}(P'_1) \leq \text{wt}(P_1) \tag{2}
\]

Let \( P' \) be the path consisting of \( P'_1 \) followed by \( P_2 \). So, \( P' \) is an \( R_i \)-path (and therefore an \( R_{i+1} \)-path) from \( s \) to \( u \) that contains fewer occurrences of \( v \) than \( P \). Furthermore,

\[
\begin{align*}
\text{wt}(P') &= \text{wt}(P'_1) + \text{wt}(P_2) \\
&\leq \text{wt}(P_1) + \text{wt}(P_2) \quad \text{[since } P' = P'_1P_2] \\
&\leq \text{wt}(P_1) + \text{wt}(v, v') + \text{wt}(P_2) \quad \text{[since } \text{wt}(v, v') \geq 0] \\
&= \text{wt}(P) 
\end{align*}
\]

So, \( P' \) is an \( R_{i+1} \)-path from \( s \) to \( u \) with no greater weight and fewer occurrences of \( v \) than \( P \), contradicting the definition of \( P \). So, \( v \) is the predecessor of \( u \) in \( P \).

We now return to the proof of (b'). Let \( u \) be any node, and let \( P \) be a shortest \( R_{i+1} \)-path from \( s \) to \( u \). We must prove that \( d_{i+1}(u) = \text{wt}(P) \). By the induction hypothesis (b), \( d_i(u) \) is the weight of some (in fact, a shortest) \( R_i \)-path from \( s \) to \( u \), which is a fortiori an \( R_{i+1} \)-path from \( s \) to \( u \). Therefore, \( d_i(u) \geq \text{wt}(P) \). There are two cases.

**Case 1.** \( d_i(u) = \text{wt}(P) \). First, we prove that in this case \( d_{i+1}(u) = d_i(u) \). If the edge \( (v, u) \) does not exist then by lines 9–10, \( d(u) \) does not change, and so \( d_{i+1}(u) = d_i(u) \). If the edge \( (v, u) \) exists, then any \( R_{i+1} \)-path from \( s \) to \( v \) followed by the edge \( (v, u) \) is an \( R_{i+1} \)-path from \( s \) to \( u \), and since \( P \) is a shortest \( R_{i+1} \)-path from \( s \) to \( u \), \( \text{wt}(P) \leq d_i(v) + \text{wt}(v, u) \). Therefore, \( d_i(u) = \text{wt}(P) \leq d_i(v) + \text{wt}(v, u) \). So by line 10, \( d(u) \) does not change, and again \( d_{i+1}(u) = d_i(u) \). By the hypothesis of the case, \( d_i(u) = \text{wt}(P) \), so \( d_{i+1}(u) = \text{wt}(P) \), as wanted.

**Case 2.** \( d_i(u) > \text{wt}(P) \). In this case \( P \) is an \( R_{i+1} \)-path that is not an \( R_i \)-path and it must therefore contain the node \( v \). By Fact 1, we can assume without loss of generality that \( v \) occurs only once and is the predecessor of \( u \) in \( P \). Let \( P_1 \) be the prefix of \( P \) from \( s \) to \( v \); clearly, \( P_1 \) is a shortest \( R_i \)-path from \( s \) to \( v \). We have,

\[
\begin{align*}
\text{wt}(P) &= \text{wt}(P_1) + \text{wt}(v, u) \\
&= d_i(v) + \text{wt}(v, u) \quad \text{[by the induction hypothesis (b)]} 
\end{align*}
\]

So, by the hypothesis of the case and (3), \( d_i(u) > d_i(v) + \text{wt}(v, u) \) and so, by line 10, \( d_{i+1}(u) = d_i(v) + \text{wt}(v, u) \). Then, by (3), \( d_{i+1}(u) = \text{wt}(P) \), as wanted.

**Proof that (c') holds.** We must prove that for all \( u, u' \in V \), if \( u \in R_{i+1} \) and \( u' \notin R_{i+1} \), \( d_{i+1}(u) \leq d_{i+1}(u') \). By the induction hypothesis (c') and the definition of \( v \) (as a node not in \( R_i \) with minimum \( d_i \)-value — see line 7), we have that for each node \( u \in R_{i+1} = R_i \cup \{v\} \) and each node \( u' \notin R_{i+1} \),

\[
d_i(u) \leq d_i(v) \leq d_i(u'). \tag{4}
\]
By (b), (b)', and the fact that $R_{i+1} \supset R_i$, we have that for any node $u$,

$$d_{i+1}(u) \leq d_i(u).$$  \hfill (5)

Also, by lines 9–10, for each node $u'$, and in particular for each node $u' \notin R_{i+1}$, either (i) $d_{i+1}(u') = d_i(u')$ or (ii) $d_{i+1}(u') = d_i(v) + \text{wt}(v, u')$.

Let $u, u'$ be any nodes such that $u \in R_{i+1}$ and $u' \notin R_{i+1}$. In case (i) we have,

$$d_{i+1}(u) \leq d_i(u) \quad \text{[by (5)]}$$
$$\leq d_i(u') \quad \text{[by (4)]}$$
$$= d_{i+1}(u') \quad \text{[by the hypothesis of case (i)]}$$

In case (ii) we have,

$$d_{i+1}(u) \leq d_i(u) \quad \text{[by (5)]}$$
$$\leq d_i(v) \quad \text{[by (4)]}$$
$$\leq d_i(v) + \text{wt}(v, u') \quad \text{[since every edge has non-negative weight]}$$
$$= d_{i+1}(u') \quad \text{[by the hypothesis of case (ii)]}$$

So, in both cases we have $d_{i+1}(u) \leq d_{i+1}(u')$, as wanted.

**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 $\Theta(n)$ 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 $d$ 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. 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 $n - 1$ `ExtractMin` operations, one in each iteration of the while loop. We perform at most $m$ `ChangeKey` operations: at most once for each edge $(v, u)$, in the iteration of the while loop in which $v$ is added to $R$. In the initialisation we must also perform a `BuildHeap` operation, to create the initial heap; this takes $O(n)$ time. Thus, the total time required to process all these operations is $O(n) + O(n \log n) + O(m \log n) = O((m + n) \log n)$. 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.