1 Shortest paths in graphs

Let $G = (V, E)$ be a directed graph with $|V| = n$ vertices, $|E| = m$ arcs, and arc weights $c : E \to \mathbb{R}$. A *path* in G from $u \in V$ to $v \in V$ is a series $(x_0, x_1), (x_1, x_2), (x_2, x_3), \ldots, (x_{r-1}, x_r)$ of arcs in E with $x_0 = u$ and $x_r = v$. A path is called *simple*, if no vertex appears more than once. The *length* of a path is the sum of the weights of its arcs, i.e. $\sum_{i=1}^{r} c(x_{i-1}, x_i)$. The number of arcs contained in a path is called its *arc length*.

Many applications call for finding *shortest* paths in a graph, e.g. routing of communication network connections, or choosing a route in navigation systems. The following cases can be distinguished:

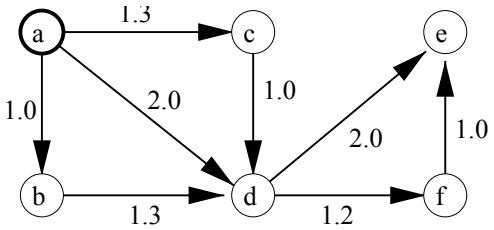
- *single-pair*: given u and v, compute a shortest path from u to v
- *single-source*: given a start vertex s, compute the shortest paths from s to all other vertices
- *single-sink*: given a target vertex t, compute the shortest path from all other vertices to t
- *all-pairs*: compute the shortest paths of all pairs of vertices

Until now, we know of no substantially better algorithm for the single-pair problem than for the single-source problem. This is why, in the following, we focus on the single-source problem. (The single-sink problem can be solved by using the single-source problem and reversing the direction of all the arcs. The all-pairs problem can be solved by solving n single-source problems.)

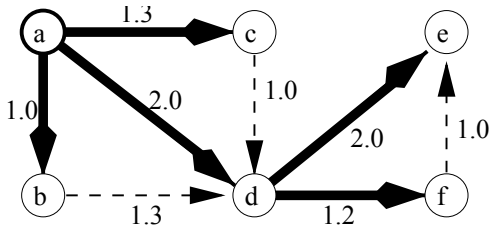
Let s be the start vertex of the single-source problem. We assume that all other vertices can be reached from s (vertices which are not reachable by a path cannot have a shortest path) and that there is no cycle of negative weight (otherwise a path can be made arbitrarily short by repeatedly traversing this cycle; given a graph with negative cycles, finding *simple* shortest paths in that graph is an $\mathcal{NP}$-hard problem).

If G contains no cycles of negative length, there always exist *simple* shortest paths: every non-simple path contains a cycle which can be left out without increasing the length of the path. The length of the a shortest path from a start vertex to another vertex v is also called the *distance* of v.

Furthermore, `from`$[v]$ for all vertices $v \neq s$ denotes the last vertex before v on a shortest path from s to v. Hence, the shortest paths from s to all other vertices put together result in a *shortest paths tree* with root s and every vertex $v \neq s$ being a child of `from`$[v]$.



(a) Input graph



(b) Shortest paths tree

2 Dijkstra's algorithm

Dijkstra's algorithm solves the single-source problem in a directed graph with *non-negative* arc weights. The main idea is to find the shortest paths ordered by increasing lengths. To do so, the algorithm calculates a value $\texttt{dist}[v]$ for every vertex v, which denotes the length of a shortest path from the start vertex s to v, and a vertex $\texttt{from}[v]$ (except for $v = s$), which denotes the last vertex before v on a shortest path from s. (While implementing the algorithm, it also makes sense to remember the arc $(\texttt{from}[v], v)$.) At the end, the shortest paths can be easily constructed using the `from` values.

The algorithm (implicitly) manages a subset V' of the vertices V with already *processed* vertices, for which the shortest paths are already computed, i.e. the values `dist` and `from` values are already correct and will not be changed anymore. At the initialization we set $V' = \{s\}$ and $\texttt{dist}[s] = 0$. In every step one additional vertex v is processed and inserted into V'. The vertex v is chosen among those which are not already processed and is the one which minimizes the value

$$\min_{u \in V', (u,v) \in E} \{\texttt{dist}[u] + c(u, v)\},$$

i.e. it is the closest (regarding the length of the path) vertex to V' which can be reached by an arc (u, v). Then we set $\texttt{dist}[v] = \texttt{dist}[u] + c(u, v)$ and $\texttt{from}[v] = u$, and the vertex v is processed. After $n - 1$ steps the algorithm is done.

The correctness of the algorithm is easy to prove by induction. After $k < n$ steps of the algorithm, it has to be shown that V' consists of k vertices with already calculated shortest distances to s.

For the efficient implementation of Dijkstra's algorithm we have to think about how we choose the vertex v for processing in the next step. We use a priority queue, which contains all vertices of $V \setminus V'$ which can be reached from V' over only one arc. The priority of a vertex v in the priority queue is equal to $\min_{u \in V', (u,v) \in E} \{\texttt{dist}[u] + c(u, v)\}$. Using a DeleteMin operation the next vertex v can be found. Additionally, for every vertex v in the priority queue we remember the vertex $u \in V'$, which has been the last vertex before v in the shortest path to v up to now. This vertex provides us with the correct value for $\texttt{from}[v]$ when v becomes the minimum of the priority queue.

At the beginning, when $V' = \{s\}$ holds, the priority queue is initialized by inserting all neighbors v of s with priority $c(s, v)$ (i.e. vertices which are on the other ends of arcs leaving s). As soon as a vertex v is deleted from the priority queue by a DeleteMin operation we may have to insert some neighbors of v into the priority queue or decrease their priorities. If a neighbor u of v was not in the priority queue before the operation, it is inserted into it with priority $\texttt{dist}[v] + c(v, u)$. If a neighbor u of v was in the priority queue with priority greater than $\texttt{dist}[v] + c(v, u)$, its priority is decreased to that value (DecreasePriority operation). In both cases, we remember that the last vertex before u in a present shortest path to u is v and store this in $\texttt{from}[u]$.

Altogether, this implementation of Dijkstra's algorithm is very similar to that of Prim's algorithm for finding a minimum cost spanning tree. The main difference is the mapping of the priorities for the vertices in the priority queue. Even the running is similar to that of Prim's algorithm: it takes $n-1$ steps, in each of those steps a DeleteMin

operation is executed and for all neighbors of the current vertex an INSERT or DECREASEPRIORITY operation might be executed. Using Fibonacci heaps, the algorithm has a worst case running time of $O(|V| \log |V| + |E|)$.

3 Shortest paths in general graphs

Let $G = (V, E)$ be a directed graph with $|V| = n$ vertices, $|E| = m$ arcs, and arc weights $c : E \to \mathbb{R}$. We assume that a start vertex $s \in V$ is chosen, and that all other vertices can be reached from s. (It then follows that G contains at least $n - 1$ arcs, i.e. $n = O(m)$ holds.) If $c(e) > 0$ holds for all $e \in E$, Dijkstra's algorithm can solve the single-source problem, i.e. computing the shortest paths from s to all other vertices in time $O(m + n \log n)$. If negative weights occur, Dijkstra's algorithm can fail (see Figure 1), and the problem has to be solved in another way. In the following, we describe the Bellman-Ford algorithm.

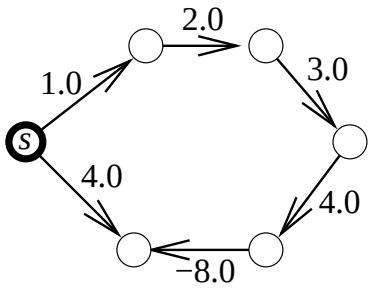


Figure 1: Counterexample for Dijkstra's algorithm

Instead of computing the shortest paths ordered by increasing length, we consider the paths ordered by increasing number of arcs. The resulting algorithm works in phases and after phase i all shortest paths with length $\leq i$ are computed. If G does not contain any cycles of negative length, there exist simple shortest paths and the algorithm terminates after n phases at the latest.

Again, we store two values for every vertex v: `dist`$[v]$ (the current distance of v to the start vertex) and `from`$[v]$ (the last vertex before v on a shortest path from s to v). At the end of the computation these values represent the shortest paths from s to all other vertices. Furthermore, we use a queue Q of vertices to which we found a shorter path and whose neighbors must again be tested on whether there exist a shorter path to them as well.

Initially, we set `from`$[v]$ = `NULL` for all $v \in V$, `dist`$[s] = 0$ and `dist`$[v] = \infty$ for all $v \neq s$. In addition we insert the start vertex s into the queue. Then we repeat the following steps:

1. get the first vertex v from the queue
2. for every outgoing arc $e = (v, w)$ of v, check if `dist`$[v] + c(v, w) <$ `dist`$[w]$ (i.e. if there is a shorter path to w via v); if yes, set `dist`$[w] =$ `dist`$[v] + c(v, w)$ and `from`$[w] = v$ and insert w into the queue if it is not there already

These steps are grouped into phases as follows: Let $v_1, \ldots, v_k$ be the vertices included in the queue at the end of the previous phase (or at the beginning of the algorithm, respectively). Then the next phase of the algorithm consists of the following k steps, where one of these vertices is processed in each of these steps. The phase ends when the vertex v_k is processed. The first vertex to be inserted into the queue after v_k, is then the first one to be processed in the next phase. Hence, one phase represents the processing of all vertices which were in the queue at the beginning of that phase.

The running time per phase is at most $O(m+n)$, because every vertex is processed at most once in each phase, and for every vertex only its outgoing arcs are considered. Because $n = O(m)$ holds, the running time per phase can be written as $O(m)$.

If the queue still contains vertices after n phases, the algorithm terminates and notifies the user of the existance of a negative cycle in G. Hence, the overall running time of the algorithm is $O(mn)$.

Theorem 1 *If G does not contain any negative cycles, the Bellman-Ford algorithm correctly computes the shortest paths from s to all other vertices.*

Proof: It is obvious that the `dist` values computed by the algorithm can never be smaller than the actual distances. Thus, we only have to show that the `dist` values coincide with the actual values at the end of the computation. We prove the following statement by induction over i:

($*$) At the end of phase i, the algorithm has computed the correct shortest paths to at least those vertices, to which there exists a shortest path with $\leq i$ arcs.

For $i = 1$, ($*$) is obviously correct. Let ($*$) be correct for $i = 1, 2, \ldots, k-1$. Let w be a vertex in V, to which there is a shortest path with k arcs, but none with less than k arcs. Let w' be the last vertex before w on such a path. Then there is a shortest path from s to w' with $k-1$ arcs. Hence, at the beginning of phase k, the distance of w' is correct. At the time when the correct distance was already computed, w' was either already in the queue or it was inserted. In any case, w' (and therefor its outgoing arc (w', w) to w) was or will be processed again after that time, in phase k at the latest. Thus, at that time, w will have computed its distance correctly and with that, its corresponding shortest path. □

If G does not contain a negative cycle, all shortest paths are computed by the time phase $n-1$ ends, and no vertices are inserted into the queue in phase n. If there are still vertices in the queue after n phases, then G must contain a negative cycle.

Historical remark: The idea to work with temporary `dist` values and to update the value `dist`$[w]$ while considering an arc (v, w) with `dist`$[v] + c(v, w) <$ `dist`$[w]$, originates from L. R. Ford in the 1950s. The idea to keep the vertices waiting to be processed in a queue, and to consider all outgoing arcs of the current vertex is on account of R. E. Bellman and E. F. Moore, which thought of this idea independently of one another. The resulting algorithm is often cited as the Bellman-Ford algorithm, and occasionally as Moore-Ford algorithm.

4 Detection of cycle of negative length

The Bellman-Ford algorithm knows about the existance of negative cycles when the queue still contains vertices after n phases. In this case, we would like to display such a negative cycle.

4.1 Detection after n phases

The following theorem shows how to effectively identify a negative cycle after n phases.

Theorem 2 *For $k = 1, \ldots, n$, and a vertex w in the queue after k phases, the chain $w, \texttt{from}[w], \texttt{from}[\texttt{from}[w]] = \texttt{from}^{(2)}[w], \ldots$ can be backtracked at least k times without ever discovering a* `NULL` *value. In other words: $\texttt{from}^{(k)}[w] \neq \texttt{NULL}$.*

Proof: Each vertex, which was inserted into the queue after one of the n phases, has a defined `from` value unequal to `NULL`. We show the claim of the theorem with induction over k. For $k = 1$, it is obviously true. Let w be one of those vertices in the queue after k phases. Thus, there must be a vertex w', which was in the queue after $k-1$ phases, and whose outgoing arc (w', w) is the cause of w to be inserted into the queue again. Either $\texttt{from}[w] = w'$ still holds at the end of phase k, then we know (induction hypothesis) that the `from` chain can be backtracked $k-1$ times starting at w'; or afterwards an even shorter path to w' was found, then $\texttt{from}[w] = w''$ holds for a vertex w'' which was in queue after phase $k-1$. And then again the `from` chain can be backtracked k times starting at w. □

Now let w be a vertex in the queue after n phases. Because the `from` chain can be backtracked n times starting at w, but the $n+1$ values $\texttt{from}[w]$, $\texttt{from}^{(2)}[w]$, …, $\texttt{from}^{(n)}[w]$ cannot all be different, this chain must contain a cycle. It is easy to see that this cycle must have a negative length. Otherwise the algorithm would not have inserted the arc which closed the cycle.

If the queue still contains vertices after n phases, a negative cycle can be found easily by getting an arbitrary vertex w from the queue and backtracking its `from` chain until a vertex w' appears for the second time in that chain. The corresponding arcs from the first to the second appearance of w' form a cycle of negative length.

4.2 Early detection

If the input graph contains a negative cycle, this cycle may occur after only a few phases in the `from` chain of some vertex. In this case the computation can be terminated instantly and does not have to cover the whole n phases. To do so, every time we find a shorter path to w traversing an arc (v, w) and want to set the value $\texttt{dist}[w] = \texttt{dist}[v] + c(v, w)$, we check if w is contained in the `from` chain of v. If yes, then the corresponding arcs in the `from` chain from w to v followed by the arc (v, w) form a negative cycle. But because the `from` chain may contain up to n vertices, this search may take $O(n)$ time in the worst case every time. The worst case running time of the modified Bellman-Ford algorithm will then be $O(n^2 m)$.

Another method for an early detection of negative cycles does not check if the vertex w is contained in the `from` chain of v, but does check if v is a successor of w in the current shortest paths tree. This method requires additional data structures to efficiently compute the successors of a vertex in the shortest paths tree, but – with a clever implementation – yields a worst case running time of $O(nm)$.

Literatur

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1st edition, 1990.