Bellman-Ford - finding shortest paths with negative weights - Algorithms for Competitive Programming

Source: https://cp-algorithms.com/graph/bellman_ford.html

Last update: September 10, 2025 Translated From: e-maxx.ru Bellman-Ford Algorithm ¶ Single source shortest path with negative weight edges Suppose that we are given a weighted directed graph G with n vertices and m edges, and some specified vertex v . You want to find the length of shortest paths from vertex v to every other vertex. Unlike the Dijkstra algorithm, this algorithm can also be applied to graphs containing negative weight edges . However, if the graph contains a negative cycle, then, clearly, the shortest path to some vertices may not exist (due to the fact that the weight of the shortest path must be equal to minus infinity); however, this algorithm can be modified to signal the presence of a cycle of negative weight, or even deduce this cycle. The algorithm bears the name of two American scientists: Richard Bellman and Lester Ford. Ford actually invented this algorithm in 1956 during the study of another mathematical problem, which eventually reduced to a subproblem of finding the shortest paths in the graph, and Ford gave an outline of the algorithm to solve this problem. Bellman in 1958 published an article devoted specifically to the problem of finding the shortest path, and in this article he clearly formulated the algorithm in the form in which it is known to us now. Description of the algorithm ¶ Let us assume that the graph contains no negative weight cycle. The case of presence of a negative weight cycle will be discussed below in a separate section. We will create an array of distances $d[0 \ldots n-1]$, which after execution of the algorithm will contain the answer to the problem. In the beginning we fill it as follows: $d[v] = 0$, and all other elements $d[ ]$ equal to infinity ∞ . The algorithm consists of several phases. Each phase scans through all edges of the graph, and the algorithm tries to produce relaxation along each edge (a,b) having weight c . Relaxation along the edges is an attempt to improve the value $d[b]$ using value $d[a] + c$. In fact, it means that we are trying to improve the answer for this vertex using edge (a,b) and current answer for vertex a . It is claimed that $n-1$ phases of the algorithm are sufficient to correctly calculate the lengths of all shortest paths in the graph (again, we believe that the cycles of negative weight do not exist). For unreachable vertices the distance $d[ ]$ will remain equal to infinity ∞ . Implementation ¶ Unlike many other graph algorithms, for Bellman-Ford algorithm, it is more convenient to represent the graph using a single list of all edges (instead of n lists of edges - edges from each vertex). We start the implementation with a structure $\rm edge$ for representing the edges. The input to the algorithm are numbers n , m , list e of edges and the starting vertex v . All the vertices are numbered 0 to $n - 1$. The simplest implementation ¶ The constant $\rm INF$ denotes the number "infinity" — it should be selected in such a way that it is greater than all possible path lengths. struct Edge { int a , b , cost ; }; int n , m , v ; vector < Edge > edges ; const int INF = 1000000000 ; void solve () { vector < int > d (n , INF); d [v] = 0 ; for (int i = 0 ; i < n - 1 ; ++ i) for (Edge e : edges) if (d [e . a] < INF) d [e . b] = min (d [e . b], d [e . a] + e . cost); // display d, for example, on the screen } The check if (d[e.a] < INF) is needed only if the graph contains negative weight edges: no such verification would result in relaxation from the vertices to which paths have not yet found, and incorrect distance, of the type $\infty - 1$, $\infty - 2$ etc. would appear. A better implementation ¶ This algorithm can be somewhat speeded up: often we already get the answer in a few phases and no useful work is done in remaining phases, just a waste visiting all edges. So, let's keep the flag, to tell whether something changed in the current phase or not, and if any phase, nothing changed, the algorithm can be stopped. (This optimization does not improve the asymptotic behavior, i.e., some graphs will still need all $n-1$ phases, but significantly accelerates the behavior of the algorithm "on an average", i.e., on random graphs.) With this optimization, it is generally unnecessary to restrict manually the number of phases of the algorithm to $n-1$ — the algorithm will stop after the desired number of phases. void solve () { vector < int > d (n , INF); d [v] = 0 ; for (;;) { bool any = false ; for (Edge e : edges) if (d [e . a] < INF) if (d [e . b] > d [e . a] + e . cost) { d [e . b] = d [e . a] + e . cost ; any = true ; } if (! any) break ; } // display d, for example, on the screen } Retrieving Path ¶ Let us now consider how to modify the algorithm so that it not only finds the length of shortest paths, but also allows to reconstruct the shortest paths. For that, let's create another array $p[0 \ldots n-1]$, where for each vertex we store its "predecessor", i.e. the penultimate vertex in the shortest path leading to it. In fact, the shortest path to any vertex a is a shortest path to some vertex $p[a]$, to which we added a at the end of the path. Note that the algorithm works on the same logic: it assumes that the shortest distance to one vertex is already calculated, and, tries to improve the shortest distance to other vertices from that vertex. Therefore, at the time of improvement we just need to remember $p[ ]$, i.e, the vertex from which this improvement has occurred. Following is an implementation of the Bellman-Ford with the retrieval of shortest path to a given node t : void solve () { vector < int > d (n , INF); d [v] = 0 ; vector < int > p (n , -1); for (;;) { bool any = false ; for (Edge e : edges) if (d [e . a] < INF) if (d [e . b] > d [e . a] + e . cost) { d [e . b] = d [e . a] + e . cost ; p [e . b] = e . a ; any = true ; } if (! any) break ; } if (d [t] == INF) cout << "No path from " << v << " to " << t << "." ; else { vector < int > path ; for (int cur = t ; cur != -1 ; cur = p [cur]) path . push_back (cur); reverse (path . begin (), path . end ()); cout << "Path from " << v << " to " << t << ": " ; for (int u : path) cout << u << ' ' ; } } Here starting from the vertex t , we go through the predecessors till we reach starting vertex with no predecessor, and store all the vertices in the path in the list $\rm path$. This list is a shortest path from v to t , but in reverse order, so we call $\rm reverse()$ function over $\rm path$ and then output the path. The proof of the algorithm ¶ First, note that for all unreachable vertices u the algorithm will work correctly, the label $d[u]$ will remain equal to infinity (because the algorithm Bellman-Ford will find some way to all reachable vertices from the start vertex v , and relaxation for all other remaining vertices will never happen). Let us now prove the following assertion: After the execution of i_{th} phase, the Bellman-Ford algorithm correctly finds all shortest paths whose number of edges does not exceed i . In other words, for any vertex a let us denote the k number of edges in the shortest path to it (if there are several such paths, you can take any). According to this statement, the algorithm guarantees that after k_{th} phase the shortest path for vertex a will be found. Proof : Consider an arbitrary vertex a to which there is a path from the starting vertex v , and consider a shortest path to it $(p_0=v, p_1, \ldots, p_k=a)$. Before the first phase, the shortest path to the vertex $p_0 = v$ was found correctly. During the first phase, the edge (p_0,p_1) has been checked by the algorithm, and therefore, the distance to the vertex p_1 was correctly calculated after the first phase. Repeating this statement k times, we see that after k_{th} phase the distance to the vertex $p_k = a$ gets calculated correctly, which we wanted to prove. The last thing to notice is that any shortest path cannot have more than $n - 1$ edges. Therefore, the algorithm sufficiently goes up to the $(n-1)_{th}$ phase. After that, it is guaranteed that no relaxation will improve the distance to some vertex. The case of a negative cycle ¶ Everywhere above we considered that there is no negative cycle in the graph (precisely, we are interested in a negative cycle that is reachable from the starting vertex v , and, for an unreachable cycles nothing in the above algorithm changes). In the presence of a negative cycle(s), there are further complications associated with the fact that distances to all vertices in this cycle, as well as the distances to the vertices reachable from this cycle is not defined — they should be equal to minus infinity $(- \infty)$. It is easy to see that the Bellman-Ford algorithm can endlessly do the relaxation among all vertices of this cycle and the vertices reachable from it. Therefore, if you do not limit the number of phases to $n - 1$, the algorithm will run indefinitely, constantly improving the distance from these vertices. Hence we obtain the criterion for presence of a cycle of negative weights reachable for source vertex v : after $(n-1)_{th}$ phase, if we run algorithm for one more phase, and it performs at least one more relaxation, then the graph contains a negative weight cycle that is reachable from v ; otherwise, such a cycle does not exist. Moreover, if such a cycle is found, the Bellman-Ford algorithm can be modified so that it retrieves this cycle as a sequence of vertices contained in it. For this, it is sufficient to remember the last vertex x for which there was a relaxation in n_{th} phase. This vertex will either lie on a negative weight cycle, or is reachable from it. To get the vertices that are guaranteed to lie on a negative cycle, starting from the vertex x , pass through to the predecessors n times. In this way, we will get to the vertex y , which is guaranteed to lie on a negative cycle. We have to go from this vertex, through the predecessors, until we get back to the same vertex y (and it will happen, because relaxation in a negative weight cycle occur in a circular manner). Implementation: ¶ void solve () { vector < int > d (n , INF); d [v] = 0 ; vector < int > p (n , -1); int x ; for (int i = 0 ; i < n ; ++ i) { x = -1 ; for (Edge e : edges) if (d [e . a] < INF) if (d [e . b] > d [e . a] + e . cost) { d [e . b] = max (- INF , d [e . a] + e . cost); p [e . b] = e . a ; x = e . b ; } } if (x == -1) cout << "No negative cycle from " << v ; else { int y = x ; for (int i = 0 ; i < n ; ++ i) y = p [y]; vector < int > path ; for (int cur = y ;; cur = p [cur]) { path . push_back (cur); if (cur == y && path . size () > 1) break ; } reverse (path . begin (), path . end ()); cout << "Negative cycle: " ; for (int u : path) cout << u << ' ' ; } } Due to the presence of a negative cycle, for n iterations of the algorithm, the distances may go far in the negative range (to negative numbers of the order of $-n m W$, where W is the maximum absolute value of any weight in the graph). Hence in the code, we adopted additional measures against the integer overflow as follows: d [e . b] = max (- INF , d [e . a] + e . cost); The above implementation looks for a negative cycle reachable from some starting vertex v ; however, the algorithm can be modified to just look for any negative cycle in the graph. For this we need to put all the distance $d[i]$ to zero and not infinity — as if we are looking for the shortest path from all vertices simultaneously; the validity of the detection of a negative cycle is not affected. For more on this topic — see separate article, Finding a negative cycle in the graph . Shortest Path Faster Algorithm (SPFA) ¶ SPFA is a improvement of the Bellman-Ford algorithm which takes advantage of the fact that not all attempts at relaxation will work. The main idea is to create a queue containing only the vertices that were relaxed but that still could further relax their neighbors. And whenever you can relax some neighbor, you should put him in the queue. This algorithm can also be used to detect negative cycles as the Bellman-Ford. The worst case of this algorithm is equal to the $O(n m)$ of the Bellman-Ford, but in practice it works much faster and some people claim that it works even in $O(m)$ on average . However be careful, because this algorithm is deterministic and it is easy to create counterexamples that make the algorithm run in $O(n m)$. There are some care to be taken in the implementation, such as the fact that the algorithm continues forever if there is a negative cycle. To avoid this, it is possible to create a counter that stores how many times a vertex has been relaxed and stop the algorithm as soon as some vertex got relaxed for the n -th time. Note, also there is no reason to put a vertex in the queue if it is already in. const int INF = 1000000000 ; vector < vector < pair < int , int >>> adj ; bool spfa (int s , vector < int >& d) { int n = adj . size (); d . assign (n , INF); vector < int > cnt (n , 0); vector < bool > inqueue (n , false); queue < int > q ; d [s] = 0 ; q . push (s); inqueue [s] = true ; while (! q . empty ()) { int v = q . front (); q . pop (); inqueue [v] = false ; for (auto edge : adj [v]) { int to = edge . first ; int len = edge . second ; if (d [v] + len < d [to]) { d [to] = d [v] + len ; if (! inqueue [to]) { q . push (to); inqueue [to] = true ; cnt [to] ++ ; if (cnt [to] > n) return false ; // negative cycle } } } } return true ; } Related problems in online judges ¶ A list of tasks that can be solved using the Bellman-Ford algorithm: E-OLYMP #1453 "Ford-Bellman" [difficulty: low] UVA #423 "MPI Maelstrom" [difficulty: low] UVA #534 "Frogger" [difficulty: medium] UVA #10099 "The Tourist Guide" [difficulty: medium] UVA #515 "King" [difficulty: medium] UVA 12519 - The Farnsworth Parabox See also the problem list in the article Finding the negative cycle in a graph . * CSES - High Score * CSES - Cycle Finding Contributors: bimalkant-lauhny (53.11%) wangwillian0 (17.95%) jakobkogler (16.12%) likecs (3.66%) adamant-pwn (2.57%) RodionGork (2.56%) aryamanjain-scifin (0.73%) aleksmish (0.37%) joaomarcosth9 (0.37%) Aman-Codes (0.37%) orendon (0.37%) DiegoHDMGZ (0.37%) pirateksh (0.37%) sachinpatel9135 (0.37%) prprprpony (0.37%) Morass (0.37%)