WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
العربية
Български
Català
Čeština
Deutsch
Español
فارسی
Français
한국어
Հայերեն
Bahasa Indonesia
Italiano
עברית
ქართული
Latviešu
Nederlands
日本語
Polski
Português
Русский
Српски / srpski
ไทย
Українська
Tiếng Việt
中文
Edit links

Bellman–Ford algorithm

From Wikipedia, the free encyclopedia

The **Bellman–Ford algorithm** is an algorithm that computes shortest paths from a single source vertex to all of the other vertices in a weighted digraph.[1] It is slower than Dijkstra's algorithm for the same problem, but more versatile, as it is capable of handling graphs in which some of the edge weights are negative numbers. The algorithm is named after two of its developers, Richard Bellman and Lester Ford, Jr., who published it in 1958 and 1956, respectively; however, Edward F. Moore also published the same algorithm in 1957, and for this reason it is also sometimes called the **Bellman–Ford–Moore algorithm**.[1]

Negative edge weights are found in various applications of graphs, hence the usefulness of this algorithm.[2] If a graph contains a "negative cycle" (i.e. a cycle whose edges sum to a negative value) that is reachable from the source, then there is no *cheapest* path: any path can be made cheaper by one more walk around the negative cycle. In such a case, the Bellman–Ford algorithm can detect negative cycles and report their existence. [3][1]

Bellman–Ford algorithm

Class	Single-source shortest path problem (for weighted directed graphs)
Data structure	Graph
Worst case performance	$O(\|V\|\|E\|)$
Worst case space complexity	$O(\|V\|)$

Graph and tree search algorithms

α–β · A* · B* · Backtracking · Beam · **Bellman–Ford** · Best-first · Bidirectional · Borůvka · Branch & bound · BFS · British Museum · D* · DFS · Depth-limited · Dijkstra · Edmonds · Floyd–Warshall · Fringe search · Hill climbing · IDA* · Iterative deepening · Johnson · Jump point · Kruskal · Lexicographic BFS · Prim · SMA*

Listings

Graph algorithms · *Search algorithms* · *List of graph algorithms*

Related topics

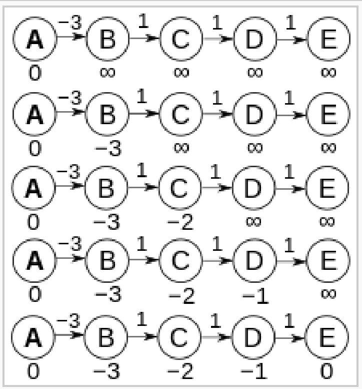
Dynamic programming · Graph traversal · Tree traversal · Search games

v · t · e

Contents [hide]

Algorithm [edit]

Like Dijkstra's Algorithm, Bellman–Ford is based on the principle of relaxation, in which an approximation to the correct distance is gradually replaced by more accurate values until eventually reaching the optimum solution. In both algorithms, the approximate distance to each vertex is always an overestimate of the true distance, and is replaced by the minimum of its old value with the length of a newly found path. However, Dijkstra's algorithm greedily selects the minimum-weight node that has not yet been processed, and performs this relaxation process on all of its outgoing edges; by contrast, the Bellman–Ford algorithm simply relaxes *all* the edges, and does this $|V|-1$ times, where $|V|$ is the number of vertices in the graph. In each of these repetitions, the number of vertices with correctly calculated distances grows, from which it follows that eventually all vertices will have their correct distances. This method allows the Bellman–Ford algorithm to be applied to a wider class of inputs than Dijkstra.



In this example graph, assuming that A is the source and edges are processed in the worst order, from right to left, it requires the full |V|−1 or 4 iterations for the distance estimates to converge. Conversely, if the edges are processed in the best order, from left to right, the algorithm converges in a single iteration.

Bellman–Ford runs in $O(|V|\cdot|E|)$ time, where $|V|$ and $|E|$ are the number of vertices and edges respectively.

```
function BellmanFord(list vertices, list edges, vertex
source)
```

```
    ::distance[],predecessor[]

    // This implementation takes in a graph, represented as
    // lists of vertices and edges, and fills two arrays
    // (distance and predecessor) with shortest-path
    // (less cost/distance/metric) information

    // Step 1: initialize graph
    for each vertex v in vertices:
        if v is source then distance[v] := 0
        else distance[v] := inf
        predecessor[v] := null

    // Step 2: relax edges repeatedly
    for i from 1 to size(vertices)-1:
        for each edge (u, v) in Graph with weight w in edges:
            if distance[u] + w < distance[v]:
                distance[v] := distance[u] + w
                predecessor[v] := u

    // Step 3: check for negative-weight cycles
    for each edge (u, v) in Graph with weight w in edges:
        if distance[u] + w < distance[v]:
            error "Graph contains a negative-weight cycle"
    return distance[], predecessor[]
```

Simply put, the algorithm initializes the distance to the source to 0 and all other nodes to infinity. Then for all edges, if the distance to the destination can be shortened by taking the edge, the distance is updated to the new lower value. At each iteration i that the edges are scanned, the algorithm finds all shortest paths of at most length i edges. Since the longest possible path without a cycle can be $|V|-1$ edges, the edges must be scanned $|V|-1$ times to ensure the shortest path has been found for all nodes. A final scan of all the edges is performed and if any distance is updated, then a path of length $|V|$ edges has been found which can only occur if at least one negative cycle exists in the graph.

Proof of correctness [edit]

The correctness of the algorithm can be shown by induction. The precise statement shown by induction is:

Lemma. After *i* repetitions of *for* loop:

- If Distance(*u*) is not infinity, it is equal to the length of some path from *s* to *u*;
- If there is a path from *s* to *u* with at most *i* edges, then Distance(u) is at most the length of the shortest path from *s* to *u* with at most *i* edges.

Proof. For the base case of induction, consider `i=0` and the moment before *for* loop is executed for the first time. Then, for the source vertex, `source.distance = 0`, which is correct. For other vertices *u*, `u.distance = infinity`, which is also correct because there is no path from *source* to *u* with 0 edges.

For the inductive case, we first prove the first part. Consider a moment when a vertex's distance is updated by `v.distance := u.distance + uv.weight`. By inductive assumption, `u.distance` is the length of some path from *source* to *u*. Then `u.distance + uv.weight` is the length of the path from *source* to *v* that follows the path from *source* to *u* and then goes to *v*.

For the second part, consider the shortest path from *source* to *u* with at most *i* edges. Let *v* be the last vertex before *u* on this path. Then, the part of the path from *source* to *v* is the shortest path from *source* to *v* with at most *i-1* edges. By inductive assumption, `v.distance` after *i*−1 iterations is at most the length of this path. Therefore, `uv.weight + v.distance` is at most the length of the path from *s* to *u*. In the i^{th} iteration, `u.distance` gets compared with `uv.weight + v.distance`, and is set equal to it if `uv.weight + v.distance` was smaller. Therefore, after *i* iteration, `u.distance` is at most the length of the shortest path from *source* to *u* that uses at most *i* edges.

If there are no negative-weight cycles, then every shortest path visits each vertex at most once, so at step 3 no further improvements can be made. Conversely, suppose no improvement can be made. Then for any cycle with vertices *v*[0], ..., *v*[*k*−1],

`v[i].distance <= v[(i-1) mod k].distance + v[(i-1) mod k]v[i].weight`

Summing around the cycle, the *v*[*i*].distance terms and the *v*[*i*−1 (mod *k*)] distance terms cancel, leaving

```
0 <= sum from 1 to k of v[i-1 (mod k)]v[i].weight
```

I.e., every cycle has nonnegative weight.

Finding negative cycles [edit]

When the algorithm is used to find shortest paths, the existence of negative cycles is a problem, preventing the algorithm from finding a correct answer. However, since it terminates upon finding a negative cycle, the Bellman–Ford algorithm can be used for applications in which this is the target to be sought - for example in cycle-cancelling techniques in network flow analysis.[1]

Applications in routing [edit]

A distributed variant of the Bellman–Ford algorithm is used in distance-vector routing protocols, for example the Routing Information Protocol (RIP). The algorithm is distributed because it involves a number of nodes (routers) within an Autonomous system, a collection of IP networks typically owned by an ISP. It consists of the following steps:

1. Each node calculates the distances between itself and all other nodes within the AS and stores this information as a table.
2. Each node sends its table to all neighboring nodes.
3. When a node receives distance tables from its neighbors, it calculates the shortest routes to all other nodes and updates its own table to reflect any changes.

The main disadvantages of the Bellman–Ford algorithm in this setting are as follows:

- It does not scale well.
- Changes in network topology are not reflected quickly since updates are spread node-by-node.
- Count to infinity if link or node failures render a node unreachable from some set of other nodes, those nodes may spend forever gradually increasing their estimates of the distance to it, and in the meantime there may be routing loops.

Improvements [edit]

The Bellman–Ford algorithm may be improved in practice (although not in the worst case) by the observation that, if an iteration of the main loop of the algorithm terminates without making any changes, the algorithm can be immediately terminated, as subsequent iterations will not make any more changes. With this early termination condition, the main loop may in some cases use many fewer than $|V| - 1$ iterations, even though the worst case of the algorithm remains unchanged.

Yen (1970) described two more improvements to the Bellman–Ford algorithm for a graph without negative-weight cycles; again, while making the algorithm faster in practice, they do not change its O(|V|*|E|) worst case time bound. His first improvement reduces the number of relaxation steps that need to be performed within each iteration of the algorithm. If a vertex *v* has a distance value that has not changed since the last time the edges out of *v* were relaxed, then there is no need to relax the edges out of *v* a second time. In this way, as the number of vertices with correct distance values grows, the number whose outgoing edges that need to be relaxed in each iteration shrinks, leading to a constant-factor savings in time for dense graphs.

Yen's second improvement first assigns some arbitrary linear order on all vertices and then partitions the set of all edges into two subsets. The first subset, E_f, contains all edges (v_i, v_j) such that $i < j$; the second, E_b, contains edges (v_i, v_j) such that $i > j$. Each vertex is visited in the order $v_1, v_2, ..., v_{|V|}$, relaxing each outgoing edge from that vertex in E_f. Each vertex is then visited in the order $v_{|V|}, v_{|V|-1}, ..., v_1$, relaxing each outgoing edge from that vertex in E_b. Each iteration of the main loop of the algorithm, after the first one, adds at least two edges to the set of edges whose relaxed distances match the correct shortest path distances: one from E_f and one from E_b. This modification reduces the worst-case number of iterations of the main loop of the algorithm from $|V| - 1$ to $|V|/2$.[4][5]

Another improvement, by Bannister & Eppstein (2012), replaces the arbitrary linear order of the vertices used in Yen's second improvement by a random permutation. This change makes the worst case for Yen's improvement (in which the edges of a shortest path strictly alternate between the two subsets E_f and E_b) very unlikely to happen. With a randomly permuted vertex ordering, the expected number of iterations needed in the main loop is at most $|V|/3$.[5]

Notes [edit]

1. ^ *a b c d* Bang-Jensen & Gutin (2000)

2. ^ Sedgewick (2002).
3. ^ Kleinberg & Tardos (2006).
4. ^ Cormen et al., 2nd ed., Problem 24-1, pp. 614–615.
5. ^ *a* *b* See Sedgewick's web exercises for *Algorithms*, 4th ed., exercises 5 and 11 (retrieved 2013-01-30).

References [edit]

Original sources [edit]

- Bellman, Richard (1958). "On a routing problem". *Quarterly of Applied Mathematics* **16**: 87–90. MR 0102435.
- Ford Jr., Lester R. (August 14, 1956). *Network Flow Theory*. Paper P-923. Santa Monica, California: RAND Corporation.
- Moore, Edward F. (1959). *The shortest path through a maze*. Proc. Internat. Sympos. Switching Theory 1957, Part II. Cambridge, Mass.: Harvard Univ. Press. pp. 285–292. MR 0114710.
- Yen, Jin Y. (1970). "An algorithm for finding shortest routes from all source nodes to a given destination in general networks". *Quarterly of Applied Mathematics* **27**: 526–530. MR 0253822.
- Bannister, M. J.; Eppstein, D. (2012). *Randomized speedup of the Bellman–Ford algorithm* (PDF). Analytic Algorithmics and Combinatorics (ANALCO12), Kyoto, Japan. pp. 41–47. arXiv:1111.5414.

Secondary sources [edit]

- Bang-Jensen, Jørgen; Gutin, Gregory (2000). "Section 2.3.4: The Bellman-Ford-Moore algorithm". *Digraphs: Theory, Algorithms and Applications* (First ed.). ISBN 978-1-84800-997-4.
- Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L.. *Introduction to Algorithms*. MIT Press and McGraw-Hill., Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 24.1: The Bellman–Ford algorithm, pp. 588–592. Problem 24-1, pp. 614–615. Third Edition. MIT Press, 2009. ISBN 978-0-262-53305-8. Section 24.1: The Bellman–Ford algorithm, pp. 651–655.
- Heineman, George T.; Pollice, Gary; Selkow, Stanley (2008). "Chapter 6: Graph Algorithms". *Algorithms in a Nutshell*. O'Reilly Media. pp. 160–164. ISBN 978-0-596-51624-6.
- Kleinberg, Jon; Tardos, Éva (2006). *Algorithm Design*. New York: Pearson Education, Inc.
- Sedgewick, Robert (2002). "Section 21.7: Negative Edge Weights". *Algorithms in Java* (3rd ed.). ISBN 0-201-36121-3.

Categories: Graph algorithms | Polynomial-time problems | Dynamic programming