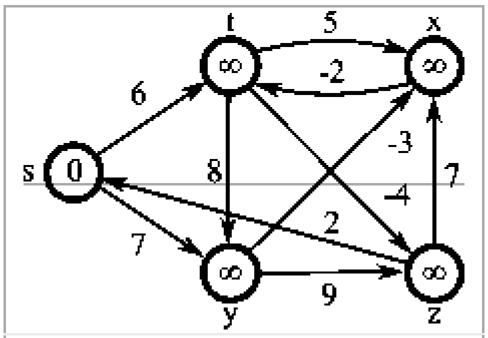
WIKIPEDIA
The Free Encyclopedia
WIKIPEDIA

Bellman–Ford algorithm

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 was first proposed by Alfonso Shimbel (1955), but is instead named after Richard Bellman and Lester Ford Jr., who published it in 1958 and 1956, respectively.[2] Edward F. Moore also published a variation of the algorithm in 1959, 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.[3] 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 that has a point on the negative cycle can be made cheaper by one more walk around the negative cycle. In such a case, the Bellman–Ford algorithm can detect and report the negative cycle.[1][4]

Bellman–Ford algorithm



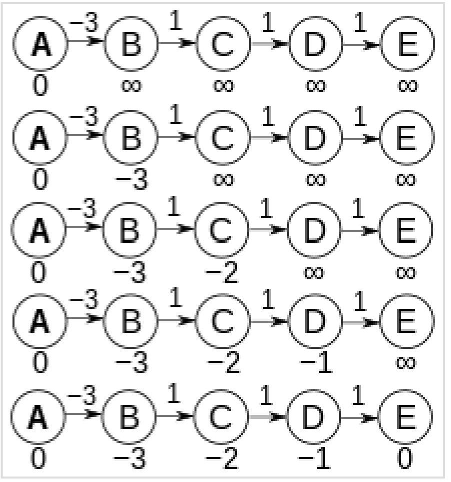
Class	Single-source shortest path problem (for weighted directed graphs)
Data structure	Graph
Worst-case performance	$\Theta(\|V\|\|E\|)$
Best-case performance	$\Theta(\|E\|)$
Worst-case space complexity	$\Theta(\|V\|)$

Algorithm

Like Dijkstra's algorithm, Bellman–Ford proceeds by relaxation, in which approximations to the correct distance are replaced by better ones until they eventually reach the 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 and the length of a newly found path. However, Dijkstra's algorithm uses a priority queue to greedily select the closest vertex 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. The intermediate answers depend on the order of edges relaxed, but the final answer remains the same.

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) is

    // This implementation takes in a graph, represented as
    // lists of vertices (represented as integers [0..n-1]) and edges,
    // and fills two arrays (distance and predecessor) holding
    // the shortest path from the source to each vertex

    distance := list of size n
    predecessor := list of size n

    // Step 1: initialize graph
    for each vertex v in vertices do
        // Initialize the distance to all vertices to infinity
        distance[v] := inf
        // And having a null predecessor
        predecessor[v] := null
    // The distance from the source to itself is, of course, zero
    distance[source] := 0

    // Step 2: relax edges repeatedly
    repeat |V|−1 times:
        for each edge (u, v) with weight w in edges do
            if distance[u] + w < distance[v] then
                distance[v] := distance[u] + w
                predecessor[v] := u

    // Step 3: check for negative-weight cycles
    for each edge (u, v) with weight w in edges do
        if distance[u] + w < distance[v] then
            predecessor[v] := u
            // A negative cycle exists; find a vertex on the cycle
            visited := list of size n initialized with false
            visited[v] := true
            while not visited[u] do
                visited[u] := true
                u := predecessor[u]
            // u is a vertex in a negative cycle, find the cycle itself
            ncycle := [u]
            v := predecessor[u]
            while v != u do
                ncycle := concatenate([v], ncycle)
                v := predecessor[v]
            error "Graph contains a negative-weight cycle", ncycle
    return distance, predecessor
```



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.

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.

The core of the algorithm is a loop that scans across all edges at every loop. For every $i \leq |V|-1$, at the end of the i-th iteration, from any vertex *v*, following the predecessor trail recorded in *predecessor* yields a path that has a total weight that is at most *distance[v]*, and further, *distance[v]* is a lower bound to the length of any path from source to *v* that uses at most *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.

The edge (u, v) that is found in step 3 must be reachable from a negative cycle, but it isn't necessarily part of the cycle itself, which is why it's necessary to follow the path of predecessors backwards until a cycle is detected. The above pseudo-code uses a boolean array (`visited`) to find a vertex on the cycle, but any cycle finding algorithm can be used to find a vertex on the cycle.

A common improvement when implementing the algorithm is to return early when an iteration of step 2 fails to relax any edges, which implies all shortest paths have been found, and therefore there are no negative cycles. In that case, the complexity of the algorithm is reduced from $O(|V| \cdot |E|)$ to $O(l \cdot |E|)$ where l is the maximum length of a shortest path in the graph.

Proof of correctness

The correctness of the algorithm can be shown by induction:

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*; and
- 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 a shortest path *P* (there may be more than one) from *source* to *v* with at most *i* edges. Let *u* be the last vertex before *v* on this path. Then, the part of the path from *source* to *u* is a shortest path from *source* to *u* with at most *i-1* edges, since if it were not, then there must be some strictly shorter path from *source* to *u* with at most *i-1* edges, and we could then append the edge *uv* to this path to obtain a path with at most *i* edges that is strictly shorter than *P*—a contradiction. By inductive assumption, `u.distance` after *i*−1 iterations is at most the length of this path from *source* to *u*. Therefore, `uv.weight + u.distance` is at most the length of *P*. In the i^{th} iteration, `v.distance` gets compared with `uv.weight + u.distance`, and is set equal to it if `uv.weight + u.distance` is smaller. Therefore, after *i* iterations, `v.distance` is at most the length of *P*, i.e., the length of the shortest path from *source* to *v* 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 and *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

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

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 (AS), 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

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. The following improvements all maintain the $O(|V| \cdot |E|)$ worst-case time complexity.

A variation of the Bellman–Ford algorithm known as Shortest Path Faster Algorithm, first described by Moore (1959), 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 (1970) described another improvement to the Bellman–Ford algorithm. His 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 $v_1, v_2, \ldots, v_{|V|}$, relaxing each outgoing edge from that vertex in E_f. Each vertex is then visited in the order $v_{|V|}, v_{|V|-1}, \ldots, v_1$, 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 $|V|/2$.[5][6]

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 $|V|/3$.[6]

Notes

1. Bang-Jensen & Gutin (2000)
2. Schrijver (2005)
3. Sedgewick (2002).
4. Kleinberg & Tardos (2006).
5. Cormen et al., 2nd ed., Problem 24-1, pp. 614–615.
6. See Sedgewick's web exercises (http://algs4.cs.princeton.edu/44sp/) for *Algorithms*, 4th ed., exercises 5 and 12 (retrieved 2013-01-30).

References

Original sources

- Shimbel, A. (1955). *Structure in communication nets*. Proceedings of the Symposium on Information Networks. New York, New York: Polytechnic Press of the Polytechnic Institute of Brooklyn. pp. 199–203.
- Bellman, Richard (1958). "On a routing problem" (https://doi.org/10.1090%2Fqam%2F102435). *Quarterly of Applied Mathematics*. **16**: 87–90. doi:10.1090/qam/102435 (https://doi.org/10.1090%2Fqam%2F102435). MR 0102435 (https://mathscinet.ams.org/mathscinet-getitem?mr=0102435).
- Ford, Lester R. Jr. (August 14, 1956). *Network Flow Theory* (http://www.rand.org/pubs/papers/P923.html). 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, Massachusetts: Harvard Univ. Press. pp. 285–292. MR 0114710 (https://mathscinet.ams.org/mathscinet-getitem?mr=0114710).
- Yen, Jin Y. (1970). "An algorithm for finding shortest routes from all source nodes to a given destination in general networks" (https://doi.org/10.1090%2Fqam%2F253822). *Quarterly of Applied Mathematics*. **27** (4): 526–530. doi:10.1090/qam/253822 (https://doi.org/10.1090%2Fqam%2F253822). MR 0253822 (https://mathscinet.ams.org/mathscinet-getitem?mr=0253822).
- Bannister, M. J.; Eppstein, D. (2012). *Randomized speedup of the Bellman–Ford algorithm*. Analytic Algorithmics and Combinatorics (ANALCO12), Kyoto, Japan. pp. 41–47. arXiv:1111.5414 (https://arxiv.org/abs/1111.5414). doi:10.1137/1.9781611973020.6 (https://doi.org/10.1137%2F1.9781611973020.6).

Secondary sources

- Ford, L. R. Jr.; Fulkerson, D. R. (1962). "A shortest chain algorithm". *Flows in Networks*. Princeton University Press. pp. 130–134.
- Bang-Jensen, Jørgen; Gutin, Gregory (2000). "Section 2.3.4: The Bellman-Ford-Moore algorithm". *Digraphs: Theory, Algorithms and Applications* (http://www.cs.rhul.ac.uk/books/dbook/) (First ed.). Springer. ISBN 978-1-84800-997-4.
- Schrijver, Alexander (2005). "On the history of combinatorial optimization (till 1960)" (http://homepages.cwi.nl/~lex/files/histco.pdf) (PDF). *Handbook of Discrete Optimization*. Elsevier: 1–68.
- 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* (https://web.archive.org/web/20080531142256/http://safari.oreilly.com/0201361213/ch21lev1sec7) (3rd ed.). ISBN 0-201-36121-3. Archived from the original (http://safari.oreilly.com/0201361213/ch21lev1sec7) on 2008-05-31. Retrieved 2007-05-28.