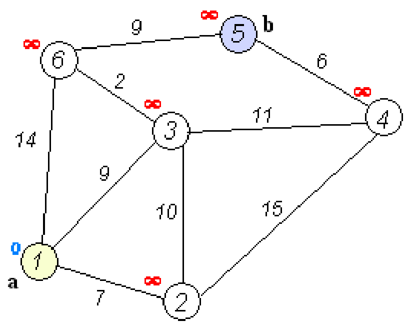
Contents

Dijkstra's algorithm

Dijkstra's algorithm



Dijkstra's algorithm. It picks the unvisited vertex with the lowest-distance, calculates the distance through it to each unvisited neighbor, and updates the neighbor's distance if smaller. Mark visited (set to red) when done with neighbors.

Class	Search algorithm
Data structure	Graph
Worst case performance	$O(\lvert E\rvert + \lvert V\rvert \log \lvert V\rvert)$

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
- Kruskal
- Johnson
- Lexicographic BFS
- Prim

• SMA*   • Uniform-cost
Listings
• *Graph algorithms*   • *Search algorithms*   • *List of graph algorithms*
Related topics
• Dynamic programming   • Graph traversal   • Tree traversal   • Search games

Dijkstra's algorithm, conceived by Dutch computer scientist Edsger Dijkstra in 1956 and published in 1959, is a graph search algorithm that solves the single-source shortest path problem for a graph with non-negative edge path costs, producing a shortest path tree. This algorithm is often used in routing and as a subroutine in other graph algorithms.

For a given source vertex (node) in the graph, the algorithm finds the path with lowest cost (i.e. the shortest path) between that vertex and every other vertex. It can also be used for finding costs of shortest paths from a single vertex to a single destination vertex by stopping the algorithm once the shortest path to the destination vertex has been determined. For example, if the vertices of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road, Dijkstra's algorithm can be used to find the shortest route between one city and all other cities. As a result, the shortest path first is widely used in network routing protocols, most notably IS-IS and OSPF (Open Shortest Path First).

Dijkstra's original algorithm does not use a min-priority queue and runs in $O(|V|^2)$ (where $|V|$ is the number of vertices). The idea of this algorithm is also given in (Leyzorek et al. 1957). The implementation based on a min-priority queue implemented by a Fibonacci heap and running in $O(|E| + |V| \log |V|)$ (where $|E|$ is the number of edges) is due to (Fredman & Tarjan 1984). This is asymptotically the fastest known single-source shortest-path algorithm for arbitrary directed graphs with unbounded non-negative weights.

Algorithm

Let the node at which we are starting be called the **initial node**. Let the **distance of node *Y*** be the distance from the **initial node** to *Y*. Dijkstra's algorithm will assign some initial distance values and will try to improve them step by step.

1. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.
2. Mark all nodes unvisited. Set the initial node as current. Create a set of the unvisited nodes called the *unvisited set* consisting of all the nodes.
3. For the current node, consider all of its unvisited neighbors and calculate their *tentative* distances. For example, if the current node *A* is marked with a distance of 6, and the edge connecting it with a neighbor *B* has length 2, then the distance to *B* (through *A*) will be 6 + 2 = 8. If this distance is less than the previously recorded tentative distance of *B*, then overwrite that distance. Even though a neighbor has been examined, it is not marked as "visited" at this time, and it remains in the *unvisited set*.
4. When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the *unvisited set*. A visited node will never be checked again.
5. If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the *unvisited set* is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes), then stop. The algorithm has finished.
6. Select the unvisited node that is marked with the smallest tentative distance, and set it as the new "current node" then go back to step 3.

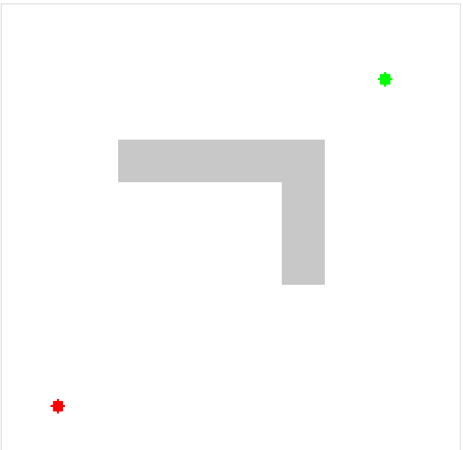


Illustration of Dijkstra's algorithm search for finding path from a start node (lower left, red) to a goal node (upper right, green) in a robot motion planning problem. Open nodes represent the "tentative" set. Filled nodes are visited ones, with color representing the distance: the greener, the farther. Nodes in all the different directions are explored uniformly, appearing as a more-or-less circular wavefront as Dijkstra's algorithm uses a heuristic identically equal to 0.

Description

__Note:__ For ease of understanding, this discussion uses the terms __intersection__, __road__ and __map__ — however, formally these terms are __vertex__, __edge__ and __graph__, respectively.

Suppose you would like to find the shortest path between two intersections on a city map, a starting point and a destination. The order is conceptually simple: to start, mark the distance to every intersection on the map with infinity. This is done not to imply there is an infinite distance, but to note that that intersection has not yet been *visited*; some variants of this method simply leave the intersection unlabeled. Now, at each iteration, select a *current* intersection. For the first iteration the current intersection will be the starting point and the distance to it (the intersection's label) will be zero. For subsequent iterations (after the first) the current intersection will be the closest unvisited intersection to the starting point—this will be easy to find.

From the current intersection, update the distance to every unvisited intersection that is directly connected to it. This is done by determining the sum of the distance between an unvisited intersection and the value of the current intersection, and relabeling the unvisited intersection with this value if it is less than its current value. In effect, the intersection is relabeled if the path to it through the current intersection is shorter than the previously known paths. To facilitate shortest path identification, in pencil, mark the road with an arrow pointing to the relabeled intersection if you label/relabel it, and erase all others pointing to it. After you have updated the distances to each neighboring intersection, mark the current intersection as *visited* and select the unvisited intersection with lowest distance (from the starting point) – or lowest label—as the current intersection. Nodes marked as visited are labeled with the shortest path from the starting point to it and will not be revisited or returned to.

Continue this process of updating the neighboring intersections with the shortest distances, then marking the current intersection as visited and moving onto the closest unvisited intersection until you have marked the destination as visited. Once you have marked the destination as visited (as is the case with any visited intersection) you have determined the shortest path to it, from the starting point, and can trace your way back, following the arrows in reverse.

Of note is the fact that this algorithm makes no attempt to direct "exploration" towards the destination as one might expect. Rather, the sole consideration in determining the next "current" intersection is its distance from the starting point. This algorithm therefore "expands outward" from the starting point, iteratively considering every node that is closer in terms of shortest path distance until it reaches the destination. When understood in this way, it is clear how the algorithm necessarily finds the shortest path, however it may also reveal one of the algorithm's weaknesses: its relative slowness in some topologies.

Pseudocode

In the following algorithm, the code `u := vertex in Q with smallest dist[]`, searches for the vertex `u` in the vertex set `Q` that has the least `dist[u]` value. That vertex is removed from the set `Q` and returned to the user. `dist_between(u, v)` calculates the length between the two neighbor-nodes `u` and `v`. The variable `alt` on lines 20 & 22 is the length of the path from the root node to the neighbor node `v` if it were to go through `u`. If this path is shorter than the current shortest path recorded for `v`, that current path is replaced with this `alt` path. The `previous` array is populated with a pointer to the "next-hop" node on the source graph to get the shortest route to the source.

```
1  function Dijkstra(Graph, source):
 2      for each vertex v in Graph:                             // Initializations
 3          dist[v]  := infinity ;                               // Unknown distance function from
 4                                                              // source to v
 5          previous[v]  := undefined ;                          // Previous node in optimal path
 6      end for                                                 // from source
 7
 8      dist[source]  := 0 ;                                     // Distance from source to source
 9      Q := the set of all nodes in Graph ;                    // All nodes in the graph are
10                                                              // unoptimized – thus are in Q
11      while Q is not empty:                                   // The main loop
12          u := vertex in Q with smallest distance in dist[] ;    // Source node in first case
13          remove u from Q ;
14          if dist[u] = infinity:
15              break ;                                         // all remaining vertices are
16          end if                                              // inaccessible from source
17
18          for each neighbor v of u:                           // where v has not yet been
19                                                              // removed from Q.
20              alt := dist[u] + dist_between(u, v) ;
21              if alt < dist[v]:                               // Relax (u,v,a)
22                  dist[v]  := alt ;
23                  previous[v]  := u ;
24                  decrease-key v in Q;                        // Reorder v in the Queue
```

```
25              end if
26          end for
27      end while
28      return dist;
29  endfunction
```

If we are only interested in a shortest path between vertices `source` and `target`, we can terminate the search at line 13 if `u = target`. Now we can read the shortest path from `source` to `target` by reverse iteration:

```
S := empty sequence
u := target
while previous[u] is defined:                               // Construct the shortest path with a stack S
    insert u at the beginning of S                          // Push the vertex into the stack
    u := previous[u]                                        // Traverse from target to source
end while ;
```

Now sequence `S` is the list of vertices constituting one of the shortest paths from `source` to `target`, or the empty sequence if no path exists.

A more general problem would be to find all the shortest paths between `source` and `target` (there might be several different ones of the same length). Then instead of storing only a single node in each entry of `previous[]` we would store all nodes satisfying the relaxation condition. For example, if both `r` and `source` connect to `target` and both of them lie on different shortest paths through `target` (because the edge cost is the same in both cases), then we would add both `r` and `source` to `previous[target]`. When the algorithm completes, `previous[]` data structure will actually describe a graph that is a subset of the original graph with some edges removed. Its key property will be that if the algorithm was run with some starting node, then every path from that node to any other node in the new graph will be the shortest path between those nodes in the original graph, and all paths of that length from the original graph will be present in the new graph. Then to actually find all these shortest paths between two given nodes we would use a path finding algorithm on the new graph, such as depth-first search.

Running time

An upper bound of the running time of Dijkstra's algorithm on a graph with edges E and vertices V can be expressed as a function of $|E|$ and $|V|$ using big-O notation.

For any implementation of vertex set Q the running time is in $O(|E| \cdot dk_Q + |V| \cdot em_Q)$, where dk_Q and em_Q are times needed to perform decrease key and extract minimum operations in set Q, respectively.
The simplest implementation of the Dijkstra's algorithm stores vertices of set Q in an ordinary linked list or array, and extract minimum from Q is simply a linear search through all vertices in Q. In this case, the running time is $O(|E| + |V|^2) = O(|V|^2)$.
For sparse graphs, that is, graphs with far fewer than $O(|V|^2)$ edges, Dijkstra's algorithm can be implemented more efficiently by storing the graph in the form of adjacency lists and using a self-balancing binary search tree, binary heap, pairing heap, or Fibonacci heap as a priority queue to implement extracting minimum efficiently. With a self-balancing binary search tree or binary heap, the algorithm requires $\Theta((|E| + |V|) \log |V|)$ time (which is dominated by $\Theta(|E| \log |V|)$, assuming the graph is connected). To avoid O(|V|) look-up in decrease-key step on a vanilla binary heap, it is necessary to maintain a supplementary index mapping each vertex to the heap's index (and keep it up to date as priority queue Q changes), making it take only $O(\log |V|)$ time instead. The Fibonacci heap improves this to $O(|E| + |V| \log |V|)$.
Note that for directed acyclic graphs, it is possible to find shortest paths from a given starting vertex in linear time, by processing the vertices in a topological order, and calculating the path length for each vertex to be the minimum

length obtained via any of its incoming edges.[1]

Related problems and algorithms

The functionality of Dijkstra's original algorithm can be extended with a variety of modifications. For example, sometimes it is desirable to present solutions which are less than mathematically optimal. To obtain a ranked list of less-than-optimal solutions, the optimal solution is first calculated. A single edge appearing in the optimal solution is removed from the graph, and the optimum solution to this new graph is calculated. Each edge of the original solution is suppressed in turn and a new shortest-path calculated. The secondary solutions are then ranked and presented after the first optimal solution.

Dijkstra's algorithm is usually the working principle behind link-state routing protocols, OSPF and IS-IS being the most common ones.

Unlike Dijkstra's algorithm, the Bellman–Ford algorithm can be used on graphs with negative edge weights, as long as the graph contains no negative cycle reachable from the source vertex *s*. The presence of such cycles means there is no shortest path, since the total weight becomes lower each time the cycle is traversed.

The A* algorithm is a generalization of Dijkstra's algorithm that cuts down on the size of the subgraph that must be explored, if additional information is available that provides a lower bound on the "distance" to the target. This approach can be viewed from the perspective of linear programming: there is a natural linear program for computing shortest paths, and solutions to its dual linear program are feasible if and only if they form a consistent heuristic (speaking roughly, since the sign conventions differ from place to place in the literature). This feasible dual / consistent heuristic defines a non-negative reduced cost and A* is essentially running Dijkstra's algorithm with these reduced costs. If the dual satisfies the weaker condition of admissibility, then A* is instead more akin to the Bellman–Ford algorithm.

The process that underlies Dijkstra's algorithm is similar to the greedy process used in Prim's algorithm. Prim's purpose is to find a minimum spanning tree that connects all nodes in the graph; Dijkstra is concerned with only two nodes. Prim's does not evaluate the total weight of the path from the starting node, only the individual path.

Breadth-first search can be viewed as a special-case of Dijkstra's algorithm on unweighted graphs, where the priority queue degenerates into a FIFO queue.

Dynamic programming perspective

From a dynamic programming point of view, Dijkstra's algorithm is a successive approximation scheme that solves the dynamic programming functional equation for the shortest path problem by the **Reaching** method.[2]

In fact, Dijkstra's explanation of the logic behind the algorithm, namely

> **Problem 2.** Find the path of minimum total length between two given nodes P and Q .
>
> We use the fact that, if R is a node on the minimal path from P to Q , knowledge of the latter implies the knowledge of the minimal path from P to R .

is a paraphrasing of Bellman's famous Principle of Optimality in the context of the shortest path problem.

Notes

[1] http://www.boost.org/doc/libs/1_44_0/libs/graph/doc/dag_shortest_paths.html

[2] Online version of the paper with interactive computational modules. (http://www.ifors.ms.unimelb.edu.au/tutorial/dijkstra_new/index.html)

References

- Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs" (http://www-m3.ma.tum.de/twiki/pub/MN0506/WebHome/dijkstra.pdf). *Numerische Mathematik* **1**: 269–271. doi: 10.1007/BF01386390 (http://dx.doi.org/10.1007/BF01386390).
- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). "Section 24.3: Dijkstra's algorithm". *Introduction to Algorithms* (Second ed.). MIT Press and McGraw–Hill. pp. 595–601. ISBN 0-262-03293-7.
- Fredman, Michael Lawrence; Tarjan, Robert E. (1984). "Fibonacci heaps and their uses in improved network optimization algorithms" (http://www.computer.org/portal/web/csdl/doi/10.1109/SFCS.1984.715934). 25th Annual Symposium on Foundations of Computer Science. IEEE. pp. 338–346. doi: 10.1109/SFCS.1984.715934 (http://dx.doi.org/10.1109/SFCS.1984.715934).
- Fredman, Michael Lawrence; Tarjan, Robert E. (1987). "Fibonacci heaps and their uses in improved network optimization algorithms" (http://portal.acm.org/citation.cfm?id=28874). *Journal of the Association for Computing Machinery* **34** (3): 596–615. doi: 10.1145/28869.28874 (http://dx.doi.org/10.1145/28869.28874).
- Zhan, F. Benjamin; Noon, Charles E. (February 1998). "Shortest Path Algorithms: An Evaluation Using Real Road Networks". *Transportation Science* **32** (1): 65–73. doi: 10.1287/trsc.32.1.65 (http://dx.doi.org/10.1287/trsc.32.1.65).
- Leyzorek, M.; Gray, R. S.; Johnson, A. A.; Ladew, W. C.; Meaker, Jr., S. R.; Petry, R. M.; Seitz, R. N. (1957). *Investigation of Model Techniques — First Annual Report — 6 June 1956 — 1 July 1957 — A Study of Model Techniques for Communication Systems*. Cleveland, Ohio: Case Institute of Technology.
- Knuth, D.E. (1977). "A Generalization of Dijkstra's Algorithm". *Information Processing Letters* **6** (1): 1–5.

External links

- C/C++
  - Dijkstra's Algorithm in C++ (https://github.com/xtaci/algorithms/blob/master/include/dijkstra.h)
  - Implementation in Boost C++ library (http://www.boost.org/doc/libs/1_43_0/libs/graph/doc/dijkstra_shortest_paths.html)
  - Dijkstra's Algorithm in C Programming Language (http://www.rawbytes.com/dijkstras-algorithm-in-c/)
- Java
  - Applet by Carla Laffra of Pace University (http://www.dgp.toronto.edu/people/JamesStewart/270/9798s/Laffra/DijkstraApplet.html)
  - Visualization of Dijkstra's Algorithm (http://students.ceid.upatras.gr/~papagel/english/java_docs/minDijk.htm)
  - Shortest Path Problem: Dijkstra's Algorithm (http://www-b2.is.tokushima-u.ac.jp/~ikeda/suuri/dijkstra/Dijkstra.shtml)
  - Dijkstra's Algorithm Applet (http://www.unf.edu/~wkloster/foundations/DijkstraApplet/DijkstraApplet.htm)
  - Open Source Java Graph package with implementation of Dijkstra's Algorithm (http://code.google.com/p/annas/)
  - A Java library for path finding with Dijkstra's Algorithm and example Applet (http://www.stackframe.com/software/PathFinder)

    - Dijkstra's algorithm as bidirectional version in Java (https://github.com/graphhopper/graphhopper/tree/90879ad05c4dfedf0390d44525065f727b043357/core/src/main/java/com/graphhopper/routing)
- C#/.Net
    - Dijkstra's Algorithm in C# (http://www.codeproject.com/KB/recipes/ShortestPathCalculation.aspx)
    - Fast Priority Queue Implementation of Dijkstra's Algorithm in C# (http://www.codeproject.com/KB/recipes/FastHeapDijkstra.aspx)
    - QuickGraph, Graph Data Structures and Algorithms for .NET (http://quickgraph.codeplex.com/)
- Dijkstra's Algorithm Simulation (http://optlab-server.sce.carleton.ca/POAnimations2007/DijkstrasAlgo.html)
- Oral history interview with Edsger W. Dijkstra (http://purl.umn.edu/107247), Charles Babbage Institute University of Minnesota, Minneapolis.
- Animation of Dijkstra's algorithm (http://www.cs.sunysb.edu/~skiena/combinatorica/animations/dijkstra.html)
- Haskell implementation of Dijkstra's Algorithm (http://bonsaicode.wordpress.com/2011/01/04/programming-praxis-dijkstra’s-algorithm/) on Bonsai code
- Implementation in T-SQL (http://hansolav.net/sql/graphs.html)
- A MATLAB program for Dijkstra's algorithm (http://www.mathworks.com/matlabcentral/fileexchange/20025-advanced-dijkstras-minimum-path-algorithm)
- Step through Dijkstra's Algorithm in an online JavaScript Debugger (http://www.turb0js.com/a/Dijkstra's_Algorithm)

Bellman–Ford algorithm

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
- Kruskal
- Johnson
- Lexicographic BFS
- Prim
- SMA*
- Uniform-cost

Listings

- *Graph algorithms*
- *Search algorithms*
- *List of graph algorithms*

Related topics

- Dynamic programming
- Graph traversal
- Tree traversal
- Search games

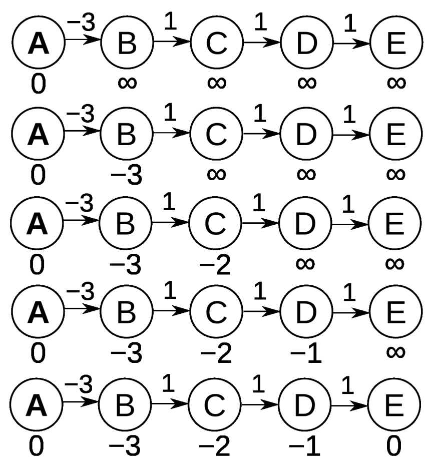
The **Bellman–Ford algorithm** is an algorithm that computes shortest paths from a single source vertex to all of the other vertices in a weighted digraph. It is slower than Dijkstra's algorithm for the same problem, but more versatile, as it is capable of handling graphs in which some of the edge weights are negative numbers. The algorithm is usually 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**.

Negative edge weights are found in various applications of graphs, hence the usefulness of this algorithm.[1] If a graph contains a "negative cycle", i.e., a cycle whose edges sum to a negative value, then there is no *cheapest* path, because any path can be made cheaper by one more walk through the negative cycle. In such a case, the Bellman–Ford algorithm can detect negative cycles and report their existence, but it cannot produce a correct "shortest path" answer if a negative cycle is reachable from the source.[2]

Algorithm

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; in 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.

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



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.

```
procedure BellmanFord(list vertices, list edges, vertex source)
   // This implementation takes in a graph, represented as lists of vertices and edges,
   // and fills two arrays (distance and predecessor) with shortest-path information

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

```
// Step 2: relax edges repeatedly
for i from 1 to size(vertices)-1:
    for each edge (u, v) 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) with weight w in edges:
    if distance[u] + w < distance[v]:
        error "Graph contains a negative-weight cycle"
```

Proof of correctness

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

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

- 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* cycle 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 cycles 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} cycle, `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* cycles, `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

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.

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, 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.

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 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 $|V|/2$.[3]

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 $|V|/3$.[]

Notes

[1] Sedgewick (2002).
[2] Kleinberg & Tardos (2006).
[3] Cormen et al., 2nd ed., Problem 24-1, pp. 614–615.

References

Original sources

- Bellman, Richard (1958). "On a routing problem". *Quarterly of Applied Mathematics* **16**: 87–90. MR 0102435 (http://www.ams.org/mathscinet-getitem?mr=0102435).
- Ford Jr., Lester R. (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, Mass.: Harvard Univ. Press. pp. 285–292. MR 0114710 (http://www.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". *Quarterly of Applied Mathematics* **27**: 526–530. MR 0253822 (http://www.ams.org/mathscinet-getitem?mr=0253822).
- Bannister, M. J.; Eppstein, D. (2012). "Randomized speedup of the Bellman–Ford algorithm" (http://siam.omnibooksonline.com/2012ANALCO/data/papers/005.pdf). *Analytic Algorithmics and Combinatorics (ANALCO12), Kyoto, Japan*. pp. 41–47. arXiv: 1111.5414 (http://arxiv.org/abs/1111.5414).

Secondary sources

- Bang-Jensen, Jørgen; Gutin, Gregory (2000). "Section 2.3.4: The Bellman-Ford-Moore algorithm" (http://www.cs.rhul.ac.uk/books/dbook/). *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" (http://safari.oreilly.com/0201361213/ch21lev1sec7). *Algorithms in Java* (3rd ed.). ISBN 0-201-36121-3.

External links

- C++ code example (https://github.com/xtaci/algorithms/blob/master/include/bellman_ford.h)
- Open Source Java Graph package with Bellman-Ford Algorithms (http://code.google.com/p/annas/)

Floyd–Warshall algorithm

Floyd–Warshall algorithm

Class	All-pairs shortest path problem (for weighted graphs)
Data structure	Graph
Worst case performance	$O(\|V\|^3)$
Best case performance	$\Omega(\|V\|^3)$
Worst case space complexity	$\Theta(\|V\|^2)$

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
- Kruskal
- Johnson
- Lexicographic BFS
- Prim
- SMA*
- Uniform-cost

Listings

- *Graph algorithms*
- *Search algorithms*
- *List of graph algorithms*

Related topics

- Dynamic programming
- Graph traversal
- Tree traversal
- Search games

In computer science, the **Floyd–Warshall algorithm** (also known as **Floyd's algorithm**, **Roy–Warshall algorithm**, **Roy–Floyd algorithm**, or the **WFI algorithm**) is a graph analysis algorithm for finding shortest paths in a weighted graph with positive or negative edge weights (but with no negative cycles, see below) and also for finding transitive closure of a relation R. A single execution of the algorithm will find the lengths (summed weights) of the shortest paths between *all* pairs of vertices, though it does not return details of the paths themselves. The algorithm is an example of dynamic programming. It was published in its currently recognized form by Robert Floyd in 1962. However, it is essentially the same as algorithms previously published by Bernard Roy in 1959 and also by Stephen Warshall in 1962 for finding the transitive closure of a graph. The modern formulation of Warshall's algorithm as three nested for-loops was first described by Peter Ingerman, also in 1962.

Algorithm

The Floyd–Warshall algorithm compares all possible paths through the graph between each pair of vertices. It is able to do this with only $\Theta(|V|^3)$ comparisons in a graph. This is remarkable considering that there may be up to $\Omega(|V|^2)$ edges in the graph, and every combination of edges is tested. It does so by incrementally improving an estimate on the shortest path between two vertices, until the estimate is optimal.

Consider a graph G with vertices V numbered 1 through N. Further consider a function shortestPath(i, j, k) that returns the shortest possible path from i to j using vertices only from the set $\{1,2,...,k\}$ as intermediate points along the way. Now, given this function, our goal is to find the shortest path from each i to each j using only vertices 1 to $k + 1$.

For each of these pairs of vertices, the true shortest path could be either (1) a path that only uses vertices in the set $\{1, ..., k\}$ or (2) a path that goes from i to $k + 1$ and then from $k + 1$ to j. We know that the best path from i to j that only uses vertices 1 through k is defined by shortestPath(i, j, k), and it is clear that if there were a better path from i to $k + 1$ to j, then the length of this path would be the concatenation of the shortest path from i to $k + 1$ (using vertices in $\{1, ..., k\}$) and the shortest path from $k + 1$ to j (also using vertices in $\{1, ..., k\}$).

If $w(i,j)$ is the weight of the edge between vertices i and j, we can define shortestPath(i, j, k + 1) in terms of the following recursive formula: the base case is

$$\mathrm{shortestPath}(i,j,0) = w(i,j)$$

and the recursive case is

$$\mathrm{shortestPath}(i,j,k{+}1) = \min(\mathrm{shortestPath}(i,j,k),\ \mathrm{shortestPath}(i,k{+}1,k){+}\mathrm{shortestPath}(k{+}1,j,k))$$

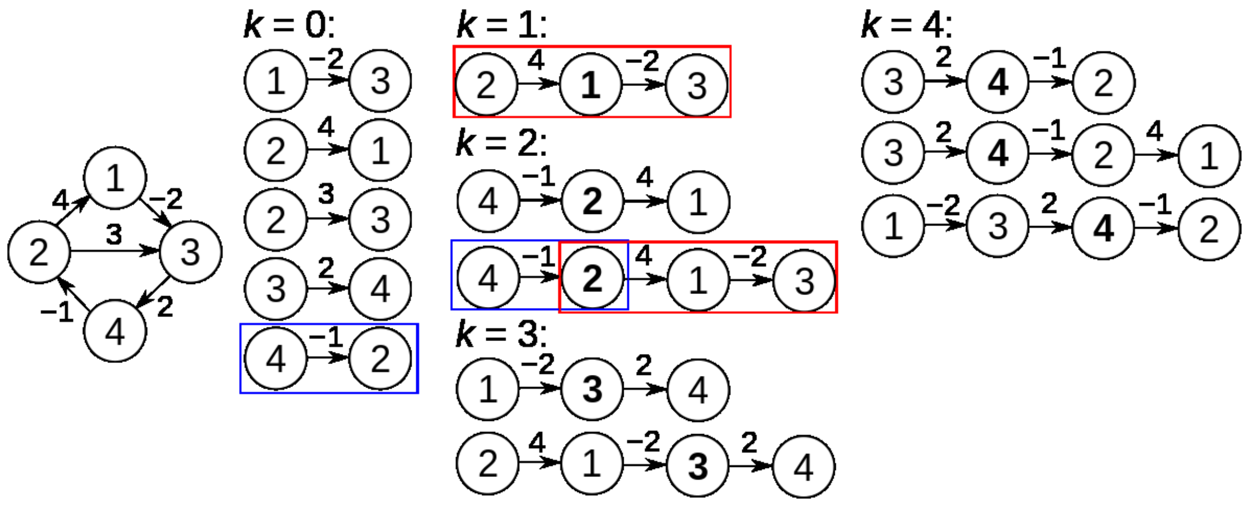
This formula is the heart of the Floyd–Warshall algorithm. The algorithm works by first computing shortestPath(i, j, k) for all (i, j) pairs for k = 1, then k = 2, etc. This process continues until k = n, and we have found the shortest path for all (i, j) pairs using any intermediate vertices. Pseudocode for this basic version follows:

```
let dist be a |V| × |V| array of minimum distances initialized to ∞ (infinity)
for each vertex v
   dist[v][v] ← 0
for each edge (u,v)
   dist[u][v] ← w(u,v)  // the weight of the edge (u,v)
for k from 1 to |V|
   for i from 1 to |V|
      for j from 1 to |V|
         if dist[i][k] + dist[k][j] < dist[i][j] then
```

```
            dist[i][j] ← dist[i][k] + dist[k][j]
```

Example

The algorithm above is executed on the graph on the left below:



Prior to the first iteration of the outer loop, labelled *k*=0 above, the only known paths correspond to the single edges in the graph. At k=1, paths that go through the vertex 1 are found: in particular, the path 2→1→3 is found, replacing the path 2→3 which has fewer edges but is longer. At k=2, paths going through the vertices {1,2} are found. The red and blue boxes show how the path 4→2→1→3 is assembled from the two known paths 4→2 and 2→1→3 encountered in previous iterations, with 2 in the intersection. The path 4→2→3 is not considered, because 2→1→3 is the shortest path encountered so far from 2 to 3. At k=3, paths going through the vertices {1,2,3} are found. Finally, at k=4, all shortest paths are found.

Behavior with negative cycles

A negative cycle is a cycle whose edges sum to a negative value. There is no shortest path between any pair of vertices i, j which form part of a negative cycle, because path-lengths from i to j can be arbitrarily small (negative). For numerically meaningful output, the Floyd–Warshall algorithm assumes that there are no negative cycles. Nevertheless, if there are negative cycles, the Floyd–Warshall algorithm can be used to detect them. The intuition is as follows:

- The Floyd–Warshall algorithm iteratively revises path lengths between all pairs of vertices (i, j), including where $i = j$;
- Initially, the length of the path (i,i) is zero;
- A path $\{(i,k), (k,i)\}$ can only improve upon this if it has length less than zero, i.e. denotes a negative cycle;
- Thus, after the algorithm, (i,i) will be negative if there exists a negative-length path from *i* back to *i*.

Hence, to detect negative cycles using the Floyd–Warshall algorithm, one can inspect the diagonal of the path matrix, and the presence of a negative number indicates that the graph contains at least one negative cycle. Obviously, in an undirected graph a negative edge creates a negative cycle (i.e., a closed walk) involving its incident vertices.

Path reconstruction

The Floyd–Warshall algorithm typically only provides the lengths of the paths between all pairs of vertices. With simple modifications, it is possible to create a method to reconstruct the actual path between any two endpoint vertices. While one may be inclined to store the actual path from each vertex to each other vertex, this is not necessary, and in fact, is very costly in terms of memory. For each vertex, one need only store the information about the highest index intermediate vertex one must pass through if one wishes to arrive at any given vertex. Therefore, information to reconstruct all paths can be stored in a single |V| × |V| matrix *next* where next[*i*][*j*] represents the highest index vertex one must travel through if one intends to take the shortest path from *i* to *j*.

To implement this, when a new shortest path is found between two vertices, the matrix containing the paths is updated. The *next* matrix is updated along with the matrix of minimum distances *dist*, so at completion both tables are complete and accurate, and any entries which are infinite in the *dist* table will be null in the *next* table. The path from *i* to *j* is the path from *i* to next[*i*][*j*], followed by the path from next[*i*][*j*] to *j*. These two shorter paths are determined recursively. This modified algorithm runs with the same time and space complexity as the unmodified algorithm.

```
let dist be a |V| × |V| array of minimum distances initialized to ∞ (infinity)
let next be a |V| × |V| array of vertex indices initialized to null

procedure FloydWarshallWithPathReconstruction ()
   for each vertex v
      dist[v][v] ← 0
   for each edge (u,v)
      dist[u][v] ← w(u,v)  // the weight of the edge (u,v)
   for k from 1 to |V|
      for i from 1 to |V|
         for j from 1 to |V|
            if dist[i][k] + dist[k][j] < dist[i][j] then
               dist[i][j] ← dist[i][k] + dist[k][j]
               next[i][j] ← k

function Path (i, j)
   if dist[i][j] = ∞ then
     return "no path"
   var intermediate ← next[i][j]
   if intermediate = null then
     return " "   // the direct edge from i to j gives the shortest path
   else
     return Path(i, intermediate) + intermediate + Path(intermediate, j)
```

Analysis

Let n be |V|, the number of vertices. To find all n^2 of shortestPath(i,j,k) (for all i and j) from those of shortestPath(i,j,k−1) requires $2n^2$ operations. Since we begin with shortestPath(i,j,0) = edgeCost(i,j) and compute the sequence of n matrices shortestPath(i,j,1), shortestPath(i,j,2), …, shortestPath(i,j,n), the total number of operations used is $n \cdot 2n^2 = 2n^3$. Therefore, the complexity of the algorithm is $\Theta(n^3)$.

Applications and generalizations

The Floyd–Warshall algorithm can be used to solve the following problems, among others:

- Shortest paths in directed graphs (Floyd's algorithm).
- Transitive closure of directed graphs (Warshall's algorithm). In Warshall's original formulation of the algorithm, the graph is unweighted and represented by a Boolean adjacency matrix. Then the addition operation is replaced by logical conjunction (AND) and the minimum operation by logical disjunction (OR).
- Finding a regular expression denoting the regular language accepted by a finite automaton (Kleene's algorithm)
- Inversion of real matrices (Gauss–Jordan algorithm)
- Optimal routing. In this application one is interested in finding the path with the maximum flow between two vertices. This means that, rather than taking minima as in the pseudocode above, one instead takes maxima. The edge weights represent fixed constraints on flow. Path weights represent bottlenecks; so the addition operation above is replaced by the minimum operation.
- Testing whether an undirected graph is bipartite.
- Fast computation of Pathfinder networks.
- Widest paths/Maximum bandwidth paths

Implementations

Implementations are available for many programming languages.

- For C++, in the boost::graph [1] library
- For C#, at QuickGraph [2]
- For Java, in the Apache Commons Graph [3] library
- For JavaScript, at Turb0JS [4]
- For MATLAB, in the Matlab_bgl [5] package
- For Perl, in the Graph [6] module
- For PHP, on page [7] and PL/pgSQL, on page [8] at Microshell
- For Python, in the NetworkX library
- For R, in package e1017 [9]
- For Ruby, in script [10]

References

[1] http://www.boost.org/libs/graph/doc/
[2] http://www.codeplex.com/quickgraph
[3] http://commons.apache.org/sandbox/commons-graph/
[4] http://www.turb0js.com/a/Floyd%E2%80%93Warshall_algorithm
[5] http://www.mathworks.com/matlabcentral/fileexchange/10922
[6] https://metacpan.org/module/Graph
[7] http://www.microshell.com/programming/computing-degrees-of-separation-in-social-networking/2/
[8] http://www.microshell.com/programming/floyd-warshal-algorithm-in-postgresql-plpgsql/3/
[9] http://cran.r-project.org/web/packages/e1071/index.html
[10] https://github.com/chollier/ruby-floyd

- Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L. (1990). *Introduction to Algorithms* (1st ed.). MIT Press and McGraw-Hill. ISBN 0-262-03141-8.
  - Section 26.2, "The Floyd–Warshall algorithm", pp. 558–565;
  - Section 26.4, "A general framework for solving path problems in directed graphs", pp. 570–576.
- Floyd, Robert W. (June 1962). "Algorithm 97: Shortest Path". *Communications of the ACM* **5** (6): 345. doi: 10.1145/367766.368168 (http://dx.doi.org/10.1145/367766.368168).
- Ingerman, Peter Z. (November 1962). "Algorithm 141: Path Matrix". *Template:Communications of the ACM* **5** (11): 556. doi: 10.1145/368996.369016 (http://dx.doi.org/10.1145/368996.369016).
- Kleene, S. C. (1956). "Representation of events in nerve nets and finite automata". In C. E. Shannon and J. McCarthy. *Automata Studies*. Princeton University Press. pp. 3–42.
- Warshall, Stephen (January 1962). "A theorem on Boolean matrices". *Journal of the ACM* **9** (1): 11–12. doi: 10.1145/321105.321107 (http://dx.doi.org/10.1145/321105.321107).
- Kenneth H. Rosen (2003). *Discrete Mathematics and Its Applications, 5th Edition*. Addison Wesley. ISBN 0-07-119881-4 (ISE). Unknown parameter `|ISBN status=` ignored (help)
- Roy, Bernard (1959). "Transitivité et connexité.". *C. R. Acad. Sci. Paris* **249**: 216–218.

External links

- Interactive animation of the Floyd–Warshall algorithm (http://www.pms.informatik.uni-muenchen.de/lehre/compgeometry/Gosper/shortest_path/shortest_path.html#visualization)
- The Floyd–Warshall algorithm in C#, as part of QuickGraph (http://quickgraph.codeplex.com/)
- Visualization of Floyd's algorithm (http://students.ceid.upatras.gr/~papagel/english/java_docs/allmin.htm)

Article Sources and Contributors

Image Sources, Licenses and Contributors

License