WIKIPEDIA
The Free Encyclopedia
WIKIPEDIA

Floyd–Warshall algorithm

In computer science, the **Floyd–Warshall algorithm** (also known as **Floyd's algorithm**, the **Roy–Warshall algorithm**, the **Roy–Floyd algorithm**, or the **WFI algorithm**) is an algorithm for finding shortest paths in a directed weighted graph with positive or negative edge weights (but with no negative cycles).[1][2] A single execution of the algorithm will find the lengths (summed weights) of shortest paths between all pairs of vertices. Although it does not return details of the paths themselves, it is possible to reconstruct the paths with simple modifications to the algorithm. Versions of the algorithm can also be used for finding the transitive closure of a relation R, or (in connection with the Schulze voting system) widest paths between all pairs of vertices in a weighted graph.

Floyd–Warshall algorithm

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

History and naming

The Floyd–Warshall algorithm is an example of dynamic programming, and was published in its currently recognized form by Robert Floyd in 1962.[3] However, it is essentially the same as algorithms previously published by Bernard Roy in 1959[4] and also by Stephen Warshall in 1962[5] for finding the transitive closure of a graph,[6] and is closely related to Kleene's algorithm (published in 1956) for converting a deterministic finite automaton into a regular expression.[7] The modern formulation of the algorithm as three nested for-loops was first described by Peter Ingerman, also in 1962.[8]

Algorithm

The Floyd–Warshall algorithm compares many possible paths through the graph between each pair of vertices. It is guaranteed to find all shortest paths and is able to do this with $\Theta(|V|^3)$ comparisons in a graph, even though there may be $\Theta(|V|^2)$ edges in the graph. 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 $\mathrm{shortestPath}(i,j,k)$ that returns the length of the shortest possible path (if one exists) from i to j using vertices only from the set $\{1,2,\ldots,k\}$ as intermediate points along the way. Now, given this function, our goal is to find the length of the shortest path from each i to each j using *any* vertex in $\{1,2,\ldots,N\}$. By definition, this is the value $\mathrm{shortestPath}(i,j,N)$, which we will find recursively.

Observe that $\mathrm{shortestPath}(i,j,k)$ must be less than or equal to $\mathrm{shortestPath}(i,j,k-1)$: we have *more* flexibility if we are allowed to use the vertex k. If $\mathrm{shortestPath}(i,j,k)$ is in fact less than $\mathrm{shortestPath}(i,j,k-1)$, then there must be a path from i to j using the vertices $\{1,2,\ldots,k\}$ that is shorter than any such path that does not use the vertex k. Since there are no negative cycles this path can be decomposed as:

(1) a path from i to k that uses the vertices $\{1,2,\ldots,k-1\}$, followed by

(2) a path from k to j that uses the vertices $\{1,2,\ldots,k-1\}$.

And of course, these must be the *shortest* such paths, otherwise we could further decrease the length. In other words, we have arrived at the recursive formula:

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

Meanwhile, the base case is given by

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

where $w(i,j)$ denotes the weight of the edge from i to j if one exists and ∞ (infinity) otherwise.

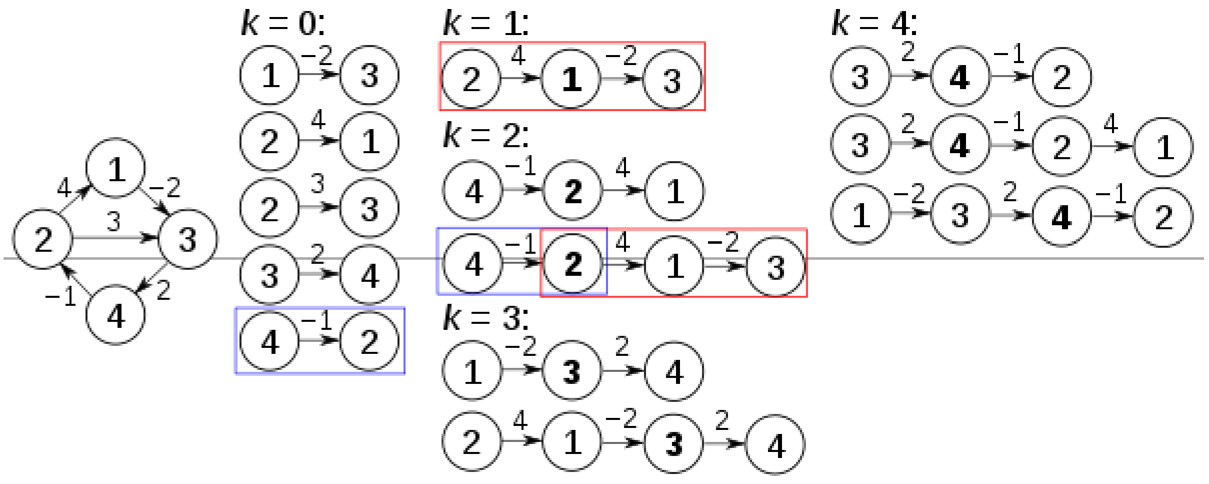
These formulas are the heart of the Floyd–Warshall algorithm. The algorithm works by first computing $\mathrm{shortestPath}(i,j,k)$ for all (i,j) pairs for $k=0$, then $k=1$, then $k=2$, and so on. This process continues until $k=N$, and we have found the shortest path for all (i,j) pairs using any intermediate vertices. Pseudocode for this basic version follows.

Pseudocode

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

Example

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



Prior to the first recursion of the outer loop, labeled $k=0$ 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 (in terms of weight). 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.

The distance matrix at each iteration of k, with the updated distances in **bold**, will be:

$k = 0$		j			
		1	2	3	4
i	1	0	∞	−2	∞
	2	4	0	3	∞
	3	∞	∞	0	2
	4	∞	−1	∞	0

$k = 1$		j			
		1	2	3	4
i	1	0	∞	−2	∞
	2	4	0	**2**	∞
	3	∞	∞	0	2
	4	∞	−1	∞	0

$k = 2$		j			
		1	2	3	4
i	1	0	∞	−2	∞
	2	4	0	2	∞
	3	∞	∞	0	2
	4	**3**	−1	**1**	0

$k = 3$		j			
		1	2	3	4
i	1	0	∞	−2	**0**
	2	4	0	2	**4**
	3	∞	∞	0	2
	4	3	−1	1	0

$k = 4$		j			
		1	2	3	4
i	1	0	**−1**	−2	0
	2	4	0	2	4
	3	**5**	**1**	0	2
	4	3	−1	1	0

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 $[i, k, \ldots, i]$ 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.[9] During the execution of the algorithm, if there is a negative cycle, exponentially large numbers can appear, as large as $\Omega(\cdot 6^{n-1} w_{max})$, where w_{max} is the largest absolute value of a negative edge in the graph. To avoid overflow/underflow problems one should check for negative numbers on the diagonal of the path matrix within the inner for loop of the algorithm.[10] Obviously, in an undirected graph a negative edge creates a negative cycle (i.e., a closed walk) involving its incident vertices. Considering all edges of the above example graph as undirected, e.g. the vertex sequence 4 – 2 – 4 is a cycle with weight sum −2.

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. Instead, the shortest-path tree can be calculated for each node in $\Theta(|E|)$ time using $\Theta(|V|)$ memory to store each tree which allows us to efficiently reconstruct a path from any two connected vertices.

Pseudocode

Source:[11]

```
let dist be a $|V| \times |V|$ array of minimum distances initialized to ∞ (infinity)
let prev be a $|V| \times |V|$ array of vertex indices initialized to null

procedure FloydWarshallWithPathReconstruction() is
    for each edge (u, v) do
        dist[u][v] ← w(u, v)  // The weight of the edge (u, v)
        prev[u][v] ← u
    for each vertex v do
        dist[v][v] ← 0
        prev[v][v] ← v
    for k from 1 to |V| do // standard Floyd-Warshall implementation
        for i from 1 to |V|
            for j from 1 to |V|
                if dist[i][j] > dist[i][k] + dist[k][j] then
                    dist[i][j] ← dist[i][k] + dist[k][j]
                    prev[i][j] ← prev[k][j]
```

```
procedure Path(u, v)
    if prev[u][v] = null then
        return []
    path ← [v]
    while $u \neq v$
        v ← prev[u][v]
        path.prepend(v)
    return path
```

Time analysis

Let n be $|V|$, the number of vertices. To find all n^2 of $\mathrm{shortestPath}(i,j,k)$ (for all i and j) from those of $\mathrm{shortestPath}(i,j,k-1)$ requires $2n^2$ operations. Since we begin with $\mathrm{shortestPath}(i,j,0) = \mathrm{edgeCost}(i,j)$ and compute the sequence of n matrices $\mathrm{shortestPath}(i,j,1)$, $\mathrm{shortestPath}(i,j,2)$, $\ldots$, $\mathrm{shortestPath}(i,j,n)$, 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, a closely related generalization of the Floyd–Warshall algorithm)[12]
- Inversion of real matrices (Gauss–Jordan algorithm) [13]
- 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.

- Fast computation of Pathfinder networks.
- Widest paths/Maximum bandwidth paths
- Computing canonical form of difference bound matrices (DBMs)
- Computing the similarity between graphs
- Transitive closure in AND/OR/threshold graphs.[14]

Implementations

Implementations are available for many programming languages.

- For C++, in the boost::graph (http://www.boost.org/libs/graph/doc/) library
- For C#, at QuickGraph (http://www.codeplex.com/quickgraph)
- For C#, at QuickGraphPCL (https://www.nuget.org/packages/QuickGraphPCL/3.6.61114.2) (A fork of QuickGraph with better compatibility with projects using Portable Class Libraries.)
- For Java, in the Apache Commons Graph (http://commons.apache.org/sandbox/commons-graph/) library
- For JavaScript, in the Cytoscape library
- For Julia, in the Graphs.jl (https://docs.juliahub.com/Graphs/VJ6vx/1.7.0/algorithms/shortestpaths/#Graphs.floyd_warshall_shortest_paths-Union{Tuple{AbstractGraph{U}},%20Tuple{T},%20Tuple{U},%20Tuple{AbstractGraph{U},%20AbstractMatrix{T}}}%20where%20{U%3C:Integer,%20T%3C:Real}) package
- For MATLAB, in the Matlab_bgl (http://www.mathworks.com/matlabcentral/fileexchange/10922) package
- For Perl, in the Graph (https://metacpan.org/module/Graph) module
- For Python, in the SciPy library (module scipy.sparse.csgraph (http://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csgraph.floyd_warshall.html#scipy.sparse.csgraph.floyd_warshall)) or NetworkX library
- For R, in packages e1071 (https://cran.r-project.org/web/packages/e1071/index.html) and Rfast (https://cran.r-project.org/web/packages/Rfast/index.html)

Comparison with other shortest path algorithms

For graphs with non-negative edge weights, Dijkstra's algorithm can be used to find all shortest paths from a *single* vertex with running time $\Theta(|E|+|V|\log|V|)$. Thus, running Dijkstra starting at *each* vertex takes time $\Theta(|E||V|+|V|^2\log|V|)$. Since $|E|=O(|V|^2)$, this yields a worst-case running time of repeated Dijkstra of $O(|V|^3)$. While this matches the asymptotic worst-case running time of the Floyd-Warshall algorithm, the constants involved matter quite a lot. When a graph is dense (i.e., $|E|\approx|V|^2$), the Floyd-Warshall algorithm tends to perform better in practice. When the graph is sparse (i.e., $|E|$ is significantly smaller than $|V|^2$), Dijkstra tends to dominate.

For sparse graphs with negative edges but no negative cycles, Johnson's algorithm can be used, with the same asymptotic running time as the repeated Dijkstra approach.

There are also known algorithms using fast matrix multiplication to speed up all-pairs shortest path computation in dense graphs, but these typically make extra assumptions on the edge weights (such as requiring them to be small integers).[15][16] In addition, because of the high constant factors in their running time, they would only provide a speedup over the Floyd–Warshall algorithm for very large graphs.

References

1. 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. See in particular Section 26.2, "The Floyd–Warshall algorithm", pp. 558–565 and Section 26.4, "A general framework for solving path problems in directed graphs", pp. 570–576.
2. Kenneth H. Rosen (2003). *Discrete Mathematics and Its Applications, 5th Edition*. Addison Wesley. ISBN 978-0-07-119881-3.

3. Floyd, Robert W. (June 1962). "Algorithm 97: Shortest Path" (https://doi.org/10.1145%2F367766.368168). *Communications of the ACM*. **5** (6): 345. doi:10.1145/367766.368168 (https://doi.org/10.1145%2F367766.368168). S2CID 2003382 (https://api.semanticscholar.org/CorpusID:2003382).
4. Roy, Bernard (1959). "Transitivité et connexité" (https://gallica.bnf.fr/ark:/12148/bpt6k3201c/f222.image). *C. R. Acad. Sci. Paris* (in French). **249**: 216–218.
5. Warshall, Stephen (January 1962). "A theorem on Boolean matrices" (https://doi.org/10.1145%2F321105.321107). *Journal of the ACM*. **9** (1): 11–12. doi:10.1145/321105.321107 (https://doi.org/10.1145%2F321105.321107). S2CID 33763989 (https://api.semanticscholar.org/CorpusID:33763989).
6. Weisstein, Eric W. "Floyd-Warshall Algorithm" (https://mathworld.wolfram.com/Floyd-WarshallAlgorithm.html). *MathWorld*.
7. Kleene, S. C. (1956). "Representation of events in nerve nets and finite automata". In C. E. Shannon and J. McCarthy (ed.). *Automata Studies*. Princeton University Press. pp. 3–42.
8. Ingerman, Peter Z. (November 1962). "Algorithm 141: Path Matrix" (https://doi.org/10.1145%2F368996.369016). *Communications of the ACM*. **5** (11): 556. doi:10.1145/368996.369016 (https://doi.org/10.1145%2F368996.369016). S2CID 29010500 (https://api.semanticscholar.org/CorpusID:29010500).
9. Hochbaum, Dorit (2014). "Section 8.9: Floyd-Warshall algorithm for all pairs shortest paths" (http://www.ieor.berkeley.edu/~hochbaum/files/ieor266-2014.pdf) (PDF). *Lecture Notes for IEOR 266: Graph Algorithms and Network Flows*. Department of Industrial Engineering and Operations Research, University of California, Berkeley.
10. Stefan Hougardy (April 2010). "The Floyd–Warshall algorithm on graphs with negative cycles". *Information Processing Letters*. **110** (8–9): 279–281. doi:10.1016/j.ipl.2010.02.001 (https://doi.org/10.1016%2Fj.ipl.2010.02.001).
11. "Free Algorithms Book" (https://books.goalkicker.com/AlgorithmsBook/).
12. Gross, Jonathan L.; Yellen, Jay (2003), *Handbook of Graph Theory* (https://books.google.com/books?id=mKkIGIea_BkC&pg=PA65), Discrete Mathematics and Its Applications, CRC Press, p. 65, ISBN 9780203490204.
13. Penaloza, Rafael. "Algebraic Structures for Transitive Closure". CiteSeerX 10.1.1.71.7650 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.71.7650). `{{cite journal}}`: Cite journal requires `|journal=` (help)
14. Gillies, Donald (1993). Scheduling Tasks with AND/OR precedence contraints (PhD Thesis, Appendix B) (http://www.ece.ubc.ca/~gillies/download/Donald_W_Gillies_PhD_1993_Scheduling_With_AND_OR_Precedence.pdf) (PDF) (Report).
15. Zwick, Uri (May 2002), "All pairs shortest paths using bridging sets and rectangular matrix multiplication", *Journal of the ACM*, **49** (3): 289–317, arXiv:cs/0008011 (https://arxiv.org/abs/cs/0008011), doi:10.1145/567112.567114 (https://doi.org/10.1145%2F567112.567114), S2CID 1065901 (https://api.semanticscholar.org/CorpusID:1065901).
16. Chan, Timothy M. (January 2010), "More algorithms for all-pairs shortest paths in weighted graphs", *SIAM Journal on Computing*, **39** (5): 2075–2089, CiteSeerX 10.1.1.153.6864 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.153.6864), doi:10.1137/08071990x (https://doi.org/10.1137%2F08071990x).

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)
- Interactive animation of the Floyd–Warshall algorithm (Technical University of Munich) (https://algorithms.discrete.ma.tum.de/graph-algorithms/spp-floyd-warshall/index_en.html)

-