# **Floyd–Warshall algorithm**

In [computer science](https://en.wikipedia.org/wiki/Computer_science), the **Floyd–Warshall algorithm** is an [algorithm](https://en.wikipedia.org/wiki/Algorithm) for finding [shortest paths](https://en.wikipedia.org/wiki/Shortest_path_problem) in a [weighted graph](https://en.wikipedia.org/wiki/Weighted_graph) with positive or negative edge weights (but with no negative cycles). A single execution of the algorithm will find the lengths (summed weights) of the 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](https://en.wikipedia.org/wiki/Transitive_closure) of a relation R {\displaystyle R} R, or (in connection with the [Schulze voting system](https://en.wikipedia.org/wiki/Schulze_method)) [widest paths](https://en.wikipedia.org/wiki/Widest_path_problem) between all pairs of vertices in a weighted graph.

**Algorithm**

The Floyd–Warshall algorithm compares all possible paths through the graph between each pair of vertices. It is able to do this with Θ ( | V | 3 ) {\displaystyle \Theta (|V|^{3})} O(|V|)3 comparisons in a graph. This is remarkable considering that there may be up to Ω ( | V | 2 ) {\displaystyle \Omega (|V|^{2})} Ω(|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 {\displaystyle G} G with vertices V {\displaystyle V} V numbered 1 through  N {\displaystyle N} N. Further consider a function shortestPath(i,j,k) s h o r t e s t P a t h ( i , j , k ) {\displaystyle \mathrm {shortestPath} (i,j,k)} shortestthat returns the shortest possible path from i i {\displaystyle i} iito j j {\displaystyle j} using vertices only from the set {1,2,…,k} { 1 , 2 , … , k } {\displaystyle \{1,2,\ldots ,k\}} {1,2,….as intermediate points along the way. Now, given this function, our goal is to find the shortest path from each i i {\displaystyle i} to each j j {\displaystyle j} jusing only vertices in {1,2,…,k+1}.{ 1 , 2 , … , k + 1 } {\displaystyle \{1,2,\ldots ,k+1\}} {1,2,…,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 {1,2,…,k+1}.{ 1 , … , k } {\displaystyle \{1,\ldots ,k\}} {1ds

or

(2) a path that goes from i i {\displaystyle i} ito k+1 k + 1 {\displaystyle k+1} and then from k+1 k + 1 {\displaystyle k+1} to jj {\displaystyle j}.

We know that the best path from i i {\displaystyle i} to j j {\displaystyle j} that only uses vertices 1 through k k {\displaystyle k} is defined by s h o r t e s t P a t h ( i , j , k ) {\displaystyle \mathrm {shortestPath} (i,j,k)} , and it is clear that if there were a better path from i i {\displaystyle i} ito k+1 k + 1 {\displaystyle k+1} to j j {\displaystyle j}, then the length of this path would be the concatenation of the shortest path from i i {\displaystyle i} to k+1 k + 1 {\displaystyle k+1} (using vertices in {1,…,k{ 1 , … , k } {\displaystyle \{1,\ldots ,k\}}}) and the shortest path from {k+1} { k + 1 } {\displaystyle \{k+1\}} to j j {\displaystyle j} (also using vertices in {1,…,k} { 1 , … , k } {\displaystyle \{1,\ldots ,k\}}).

If w(i,j) w ( i , j ) {\displaystyle w(i,j)} is the weight of the edge between vertices i i {\displaystyle i} and jj {\displaystyle j}, we can define shortestPath(I,j,k+1) s h o r t e s t P a t h ( i , j , k + 1 ) {\displaystyle \mathrm {shortestPath} (i,j,k+1)} in terms of the following [recursive](https://en.wikipedia.org/wiki/Recursion) formula: the base case is

shortestPath(i,j,k+1) = w(i,j)

s h o r t e s t P a t h ( i , j , 0 ) = w ( i , j ) {\displaystyle \mathrm {shortestPath} (i,j,0)=w(i,j)} and the recursive case is

shortestPath(i,j,k+1) = min(shortestPath(i,j,k+1) , shortest(i,k+1,k) + shortestPath(k+1,j,k))

s h o r t e s t P a t h ( i , j , k + 1 ) = {\displaystyle \mathrm {shortestPath} (i,j,k+1)=} m i n ( s h o r t e s t P a t h ( i , j , k ) , {\displaystyle \mathrm {min} {\Big (}\mathrm {shortestPath} (i,j,k),} s h o r t e s t P a t h ( i , k + 1 , k ) + s h o r t e s t P a t h ( k + 1 , j , k ) ) {\displaystyle \mathrm {shortestPath} (i,k+1,k)+\mathrm {shortestPath} (k+1,j,k){\Big )}}

This formula is the heart of the Floyd–Warshall algorithm. The algorithm works by first computing shortestPath(i,j,k) s h o r t e s t P a t h ( i , j , k ) {\displaystyle \mathrm {shortestPath} (i,j,k)} for all (i,j) ( i , j ) {\displaystyle (i,j)}pairs for k=1k = 1 {\displaystyle k=1}, then k=2k = 2 {\displaystyle k=2}, etc. This process continues until k=Nk = N {\displaystyle k=N}, and we have found the shortest path for all (i,j) ( i , j ) {\displaystyle (i,j)}pairs using any intermediate vertices. Pseudocode for this basic version follows:

1 **let** dist be a |V| × |V| array of minimum distances initialized to ∞ (infinity)

2 **for each** vertex *v*

3 dist[*v*][*v*] ← 0

4 **for each** edge (*u*,*v*)

5 dist[*u*][*v*] ← w(*u*,*v*) *// the weight of the edge (*u*,*v*)*

6 **for** *k* **from** 1 **to** |V|

7 **for** *i* **from** 1 **to** |V|

8 **for** *j* **from** 1 **to** |V|

9 **if** dist[*i*][*j*] > dist[*i*][*k*] + dist[*k*][*j*]

10 dist[*i*][*j*] ← dist[*i*][*k*] + dist[*k*][*j*]

11 **end if**

## 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](https://en.wikipedia.org/wiki/Shortest-path_tree) can be calculated for each node in O(|E|) Θ ( | E | ) {\displaystyle \Theta (|E|)} time using O(|V|) Θ ( | V | ) {\displaystyle \Theta (|V|)} memory to store each tree which allows us to efficiently reconstruct a path from any two connected vertices.

**let** dist be a |V|X|V|

|

V

|

×

|

V

|

{\displaystyle |V|\times |V|}

array of minimum distances initialized to

∞

{\displaystyle \infty }

(infinity)

**let** next be a

|

V

|

×

|

V

|

{\displaystyle |V|\times |V|}

|V|X|V| array of vertex indices initialized to **null**

**procedure** *FloydWarshallWithPathReconstruction* ()

**for each** edge (u,v)

dist[u][v] ← w(u,v) *// the weight of the edge (u,v)*

next[u][v] ← v

**for** k **from** 1 **to** |V| *// 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]

next[i][j] ← next[i][k]

**procedure** Path(u, v)

**if** next[u][v] = null **then**

**return** []

path = [u]

**while u ≠ v**

u ← next[u][v]

path.append(u)

**return** path

**Analysis**

Let n n {\displaystyle n} be |V|| V | {\displaystyle |V|}, the number of vertices. To find all n2 n 2 {\displaystyle n^{2}} of shortestPath(i,j,k) s h o r t e s t P a t h ( i , j , k ) {\displaystyle \mathrm {shortestPath} (i,j,k)} (for all i i {\displaystyle i} and jj {\displaystyle j}) from those of shortestPath(i,j,k-1) s h o r t e s t P a t h ( i , j , k − 1 ) {\displaystyle \mathrm {shortestPath} (i,j,k-1)} requires 2n2 2 n 2 {\displaystyle 2n^{2}} operations. Since we begin with shortestPath(i,j,0) s h o r t e s t P a t h ( i , j , 0 ) = e d g e C o s t ( i , j ) {\displaystyle \mathrm {shortestPath} (i,j,0)=\mathrm {edgeCost} (i,j)} and compute the sequence of n n {\displaystyle n} matrices shortestPath(i,j,1) s h o r t e s t P a t h ( i , j , 1 ) {\displaystyle \mathrm {shortestPath} (i,j,1)}, shortestPath(i,j,2)s h o r t e s t P a t h ( i , j , 2 ) {\displaystyle \mathrm {shortestPath} (i,j,2)}, …… {\displaystyle \ldots }, shortestPath(i,j,n)s h o r t e s t P a t h ( i , j , n ) {\displaystyle \mathrm {shortestPath} (i,j,n)}, the total number of operations used is n.2n2 n ⋅ 2 n 2 = 2 n 3 {\displaystyle n\cdot 2n^{2}=2n^{3}}. Therefore, the [complexity](https://en.wikipedia.org/wiki/Computational_complexity_theory) of the algorithm is O(n3).