All Pairs Shortest Paths

Dr. Bibhudatta Sahoo

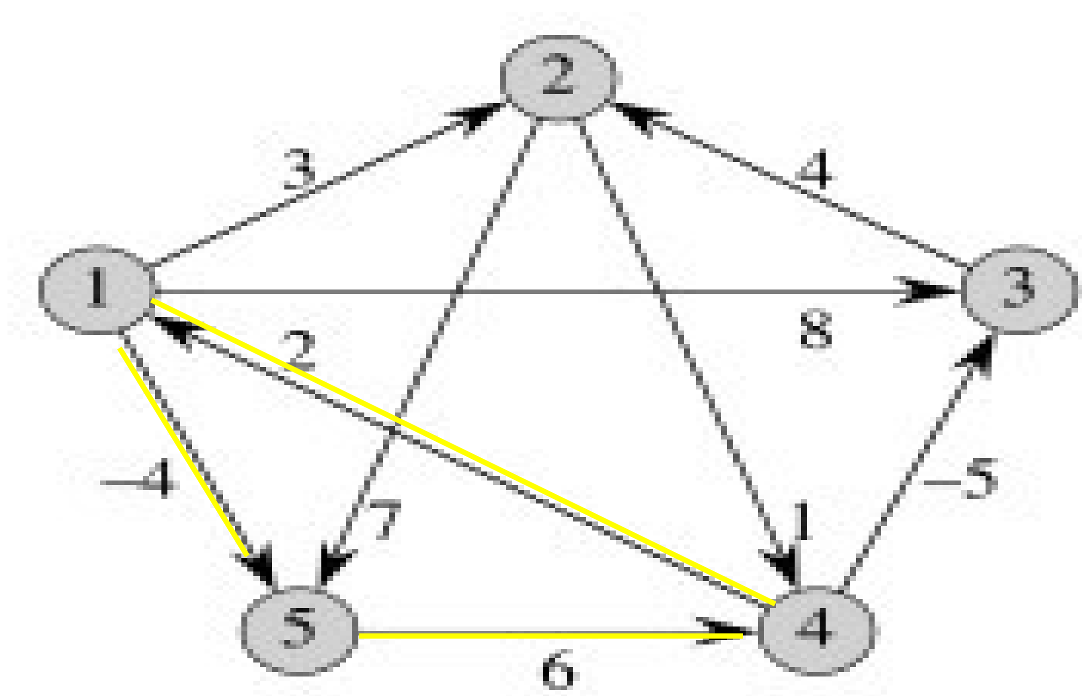
Communication & Computing Group

Department of CSE, NIT Rourkela

Email: bdsahu@nitrkl.ac.in, 9937324437, 2462358

The all pair shortest path

- The **all pair shortest path** algorithm is also known as **Floyd-Warshall algorithm** is used to find **all pair shortest path** problem from a given weighted graph.



Floyd's algorithm

- In computer science, the **Floyd–Warshall algorithm** (also known as **Floyd's algorithm**, **Roy–Warshall algorithm**, **Roy–Floyd algorithm**, or the **WFI algorithm**)
- This is a graph analysis algorithm for finding shortest paths in a weighted graph with **positive** or **negative** edge weights (but with **no negative cycles**).
- This algorithm can also be used for finding transitive closure of a relation .
- 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.

All-Pairs Shortest Paths Problem:

- Given a weighted, directed graph represented in its weight matrix form $W[1..n][1..n]$, where n = the number of nodes, and $W[i][j]$ = the edge weight of edge (i, j).
- The problem is find a shortest path between every pair of the nodes.
- Find the "shortest path" from a to b (where the length of the path is the sum of the edge weights on the path).
- Perhaps we should call this the minimum weight path!
- given : directed graph $G = (V, E)$,

  weight function $\omega : E \rightarrow R$, $|V| = n$
- goal : create an $n \times n$ matrix $D = (d_{ij})$ of shortest path distances

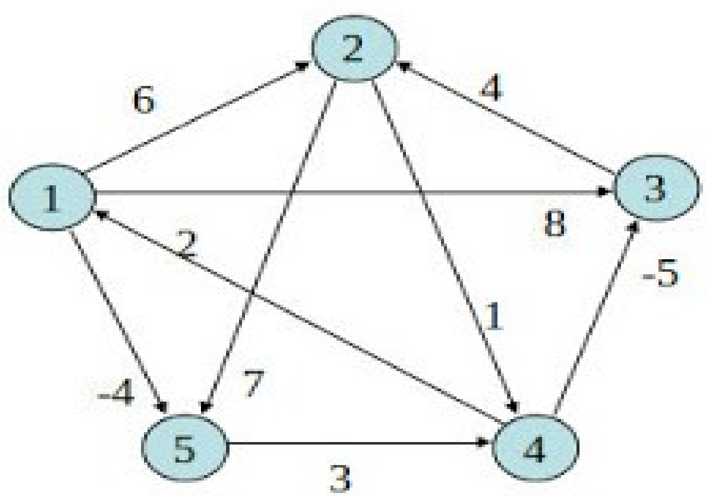
  i.e., $d_{ij} = \delta(v_i, v_j)$

Adjacency Matrix Representation of Graphs

- n x n matrix $W = (\omega_{ij})$ of edge weights :

$$\omega_{ij} = \begin{cases} \omega(v_i, v_j) & \text{if } (v_i, v_j) \in E \\ \infty & \text{if } (v_i, v_j) \notin E \end{cases}$$

- assume $\omega_{ii} = 0$ for all $v_i \in V$, because no neg-weight cycle $\Rightarrow$ shortest path to itself has no edge, i.e., $\delta(v_i, v_i) = 0$



	1	2	3	4	5
1	0	6	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	3	0

Why not Greedy algorithm?

- Start at **a**, and greedily construct a path that goes to ***w*** by adding vertices that are closest to the current endpoint, until you reach **b**.
- Problem: it doesn't work correctly! Sometimes you can't reach **b** at all, and would have to backtrack… and sometimes you get a path that doesn't have the minimum weight.

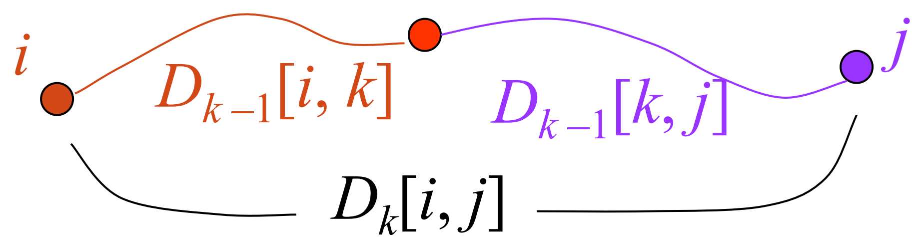
Designing a DP solution

- How are the sub-problems defined?
- Where is the answer stored?
- How are the boundary values computed?
- How do we compute each entry from other entries?
- What is the order in which we fill in the matrix?

All-Pairs Shortest Paths Problem:

- We first note that the principle of optimality applies:

- If node k is on a shortest path from node i to node j, then the subpath from i to k, and the subpath from k to j, are also shortest paths for the corresponding end nodes.



- Therefore, the problem of finding shortest paths for all pairs of nodes becomes developing a strategy to compute these shortest paths in a systematic fashion.

Floyd's Algorithm

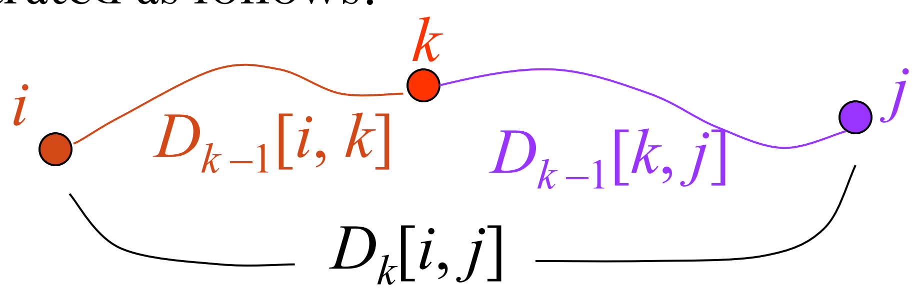
Define the notation $D_k[i, j]$, $1 \le i, j \le n$, and $0 \le k \le n$, that stands for the shortest distance (via a shortest path) from node i to node j, passing through nodes whose number (label) is $\le k$. Thus, when $k = 0$, we have

$$\boldsymbol{D_0[i, j] = W[i][j]} = \text{the edge weight from node } i \text{ to node } j$$

This is because no nodes are numbered ≤ 0 (the nodes are numbered 1 through n). In general, when $k \ge 1$,

$$\boldsymbol{D_k[i, j] = \min(D_{k-1}[i, j], D_{k-1}[i, k] + D_{k-1}[k, j])}$$

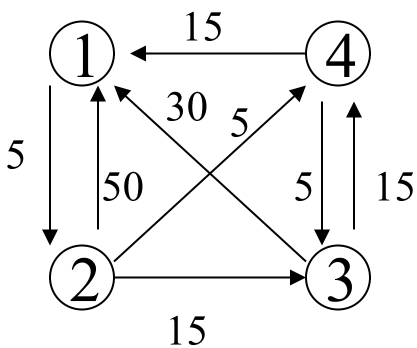
The reason for this recurrence is that when computing $D_k[i, j]$, this shortest path either doesn't go through node k, or it passes through node k exactly once. The former case yields the value $D_{k-1}[i, j]$; the latter case can be illustrated as follows:



Example 1:All pairs shortest paths

We demonstrate Floyd's algorithm for computing $D_k[i, j]$ for $k = 0$ through $k = 4$, for the following weighted directed graph:

$$\boldsymbol{D_k[i,j] = \min(D_{k-1}[i,j], D_{k-1}[i,k] + D_{k-1}[k,j])}$$



$$D_1 = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix} \qquad D_2 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

reduced from ∞ because the path (3,1,2) going thru node 1 is possible in D_1

$$D_0 = W = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{pmatrix}$$

$$D_3 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 45 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix} \qquad D_4 = \begin{pmatrix} 0 & 5 & 15 & 10 \\ 20 & 0 & 10 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

Implementation of Floyd's Algorithm:

Input: The weight matrix $W[1..n][1..n]$ for a weighted directed graph, nodes are labeled 1 through n.
Output: The shortest distances between all pairs of the nodes, expressed in an $n \times n$ matrix.

Algorithm:

Create a matrix D and initialize it to W.

for $k = 1$ to n do
for $i = 1$ to n do
for $j = 1$ to n do

$D[i][j] = \min(D[i][j], D[i][k] + D[k][j])$

Note that one single matrix D is used to store D_{k-1} and D_k, i.e., updating from D_{k-1} to D_k is done immediately.

This causes no problems because in the kth iteration, the value of $D_k[i, k]$ should be the same as it was in $D_{k-1}[i, k]$; similarly for the value of $D_k[k, j]$.

The time complexity of the above algorithm is $O(n^3)$ because of the triple-nested loop; the space complexity is $O(n^2)$ because only one matrix is used.

Function to compute lengths of shortest paths:

Algorithm Allpaths(W, A, n)

Input: *The weight matrix W[1..n][1..n] for a weighted directed graph, nodes are labeled 1 through n.*

Output: *The matrix A[1..n][1..n] shortest distances between all pairs of the nodes, expressed in an n × n matrix.*

1. for i = 1 to n do
2. for j = 1 to n do
3. $A[i][j] = W[i][j]$; // Copy cost to A
4. for k = 1 to n do
5. for i = 1 to n do
6. for j = 1 to n do
7. $A[i][j] = \mathbf{min}(A[i][j],\ D[i][k] + A[k][j])$;
8. }

Floyd–Warshall dynamic programming algorithm

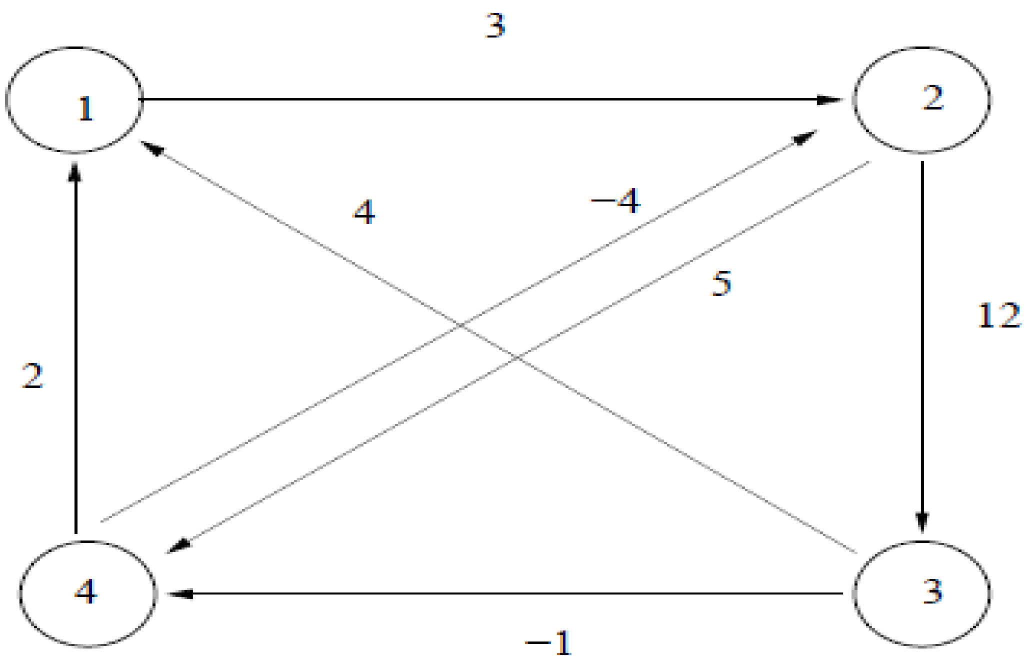
- Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex i to vertex j for which all intermediate vertices are in the set $\{1, 2, \ldots, k\}$.
- When $k = 0$, a path from vertex i to vertex j with no intermediate vertex numbered higher than 0 has no intermediate vertices at all, hence $d_{ij}^{(0)} = w_{ij}$.

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \, , \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 \, . \end{cases}$$

```
Floyd-Warshall(W)
  n ← rows[W]
  D^(0) ← W
  for k ← 1 to n
        do for i ← 1 to n
              do for j ← 1 to n
                    do d_ij^(k) ← min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
  return D^(n)
```

Running time $O(V^3)$

Example 2: All pairs shortest paths



$$D^0 = \begin{pmatrix} 0 & 3 & 0 & 0 \\ 0 & 0 & 12 & 5 \\ 4 & 0 & 0 & -1 \\ 2 & -4 & 0 & 0 \end{pmatrix} D^1 = \begin{pmatrix} 0 & 3 & 0 & 0 \\ 0 & 0 & 12 & 5 \\ 4 & 7 & 0 & -1 \\ 2 & -4 & 0 & 0 \end{pmatrix} D^2 = \begin{pmatrix} 0 & 3 & 15 & 8 \\ 0 & 0 & 12 & 5 \\ 4 & 7 & 0 & -1 \\ 2 & -4 & 8 & 0 \end{pmatrix}$$

$$D^3 = \begin{pmatrix} 0 & 3 & 15 & 8 \\ 16 & 0 & 12 & 5 \\ 4 & 7 & 0 & -1 \\ 2 & -4 & 8 & 0 \end{pmatrix} D^4 = \begin{pmatrix} 0 & 3 & 15 & 8 \\ 7 & 0 & 12 & 5 \\ 1 & -5 & 0 & -1 \\ 2 & -4 & 8 & 0 \end{pmatrix}$$

The all pairs shortest paths :Horowitz, Sahni, Rajasekaran

Let $G = (V, E)$ be a directed graph with n vertices. Let *cost* be a cost adjacency matrix for G such that $cost(i,i) = 0$, $1 \leq i \leq n$. Then $cost(i,j)$ is the length (or cost) of edge $\langle i,j \rangle$ if $\langle i,j \rangle \in E(G)$ and $cost(i,j) = \infty$ if $i \neq j$ and $\langle i,j \rangle \notin E(G)$. The *all-pairs shortest-path problem* is to determine a matrix A such that $A(i,j)$ is the length of a shortest path from i to j.

$$A(i,j) = \min \left\{ \min_{1 \leq k \leq n} \{A^{k-1}(i,k) + A^{k-1}(k,j)\}, cost(i,j) \right\} \qquad (5.7)$$

Clearly, $A^0(i,j) = cost(i,j)$, $1 \leq i \leq n$, $1 \leq j \leq n$. We can obtain a recurrence for $A^k(i,j)$ using an argument similar to that used before. A shortest path from i to j going through no vertex higher than k either goes through vertex k or it does not. If it does, $A^k(i,j) = A^{k-1}(i,k) + A^{k-1}(k,j)$. If it does not, then no intermediate vertex has index greater than $k-1$. Hence $A^k(i,j) = A^{k-1}(i,j)$. Combining, we get

The all pairs shortest paths :Horowitz, Sahni,

$$A^k(i,j) = \min\ \{A^{k-1}(i,j),\ A^{k-1}(i,k) + A^{k-1}(k,j)\},\quad k \geq 1 \qquad (5.8)$$

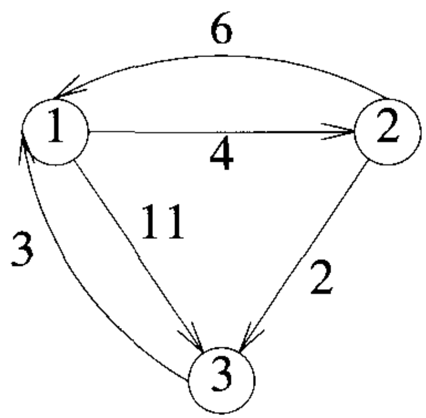
Recurrence (5.8) can be solved for A^n by first computing A^1, then A^2, then A^3, and so on. Since there is no vertex in G with index greater than n, $A(i,j) = A^n(i,j)$. Function AllPaths computes $A^n(i,j)$. The computation is done inplace so the superscript on A is not needed. The reason this computation can be carried out in-place is that $A^k(i,k) = A^{k-1}(i,k)$ and $A^k(k,j) = A^{k-1}(k,j)$. Hence, when A^k is formed, the kth column and row do not change. Consequently, when $A^k(i,j)$ is computed in line 11 of Algorithm 5.3, $A(i,k) = A^{k-1}(i,k) = A^k(i,k)$ and $A(k,j) = A^{k-1}(k,j) = A^k(k,j)$. So, the old values on which the new values are based do not change on this iteration.

Algorithm AllPaths

```
Algorithm AllPaths(cost, A, n)
// cost[1 : n, 1 : n] is the cost adjacency matrix of a graph with
// n vertices; A[i, j] is the cost of a shortest path from vertex
// i to vertex j. cost[i, i] = 0.0, for 1 ≤ i ≤ n.
{
    for i := 1 to n do
        for j := 1 to n do
            A[i, j] := cost[i, j]; // Copy cost into A.
    for k := 1 to n do
        for i := 1 to n do
            for j := 1 to n do
                A[i, j] := min(A[i, j], A[i, k] + A[k, j]);
}
```

Algorithm 5.3 Function to compute lengths of shortest paths

Example 3: All pairs shortest paths



(a) Example digraph

A^0	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

(b) A^0

A^1	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

(c) A^1

A^2	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

(d) A^2

A^3	1	2	3
1	0	4	6
2	5	0	2
3	3	7	0

(e) A^3

Thanks for Your Attention!

Exercises

Exercises

1. (a) Does the recurrence (5.8) hold for the graph of Figure 5.7? Why?

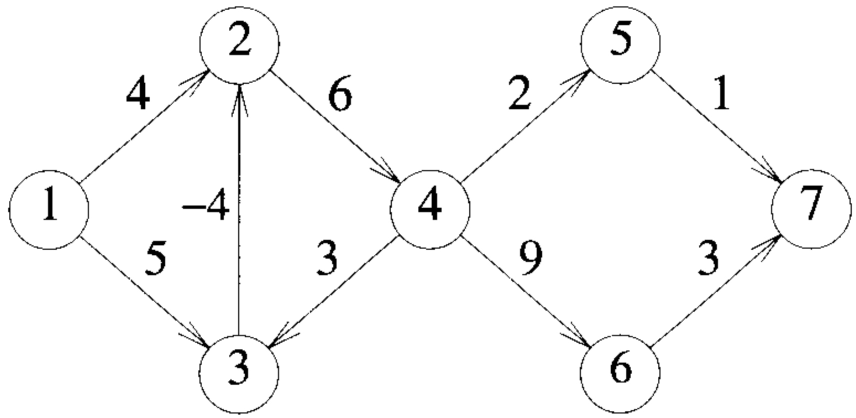


Figure 5.7 Graph for Exercise 1

   (b) Why does Equation 5.8 not hold for graphs with cycles of negative length?

2. Modify the function AllPaths so that a shortest path is output for each pair of vertices (i, j). What are the time and space complexities of the new algorithm?

Exercises

3. Let A be the adjacency matrix of a directed graph G. Define the transitive closure A^+ of A to be a matrix with the property $A^+(i,j) = 1$ iff G has a directed path, containing at least one edge, from vertex i to vertex j. $A^+(i,j) = 0$ otherwise. The reflexive transitive closure A^* is a matrix with the property $A^*(i,j) = 1$ iff G has a path, containing zero or more edges, from i to j. $A^*(i,j) = 0$ otherwise.

   (a) Obtain A^+ and A^* for the directed graph of Figure 5.8.

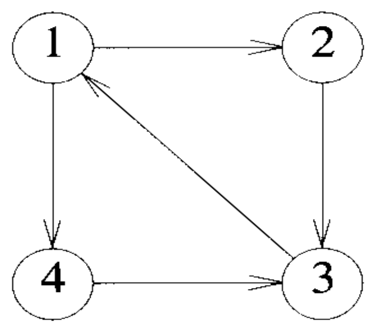


Figure 5.8 Graph for Exercise 3

   (b) Let $A^k(i,j) = 1$ iff there is a path with zero or more edges from i to j going through no vertex of index greater than k. Define A^0 in terms of the adjacency matrix A.

Exercises

(c) Obtain a recurrence between A^k and A^{k-1} similar to (5.8). Use the logical operators **or** and **and** rather than **min** and $+$.

(d) Write an algorithm, using the recurrence of part (c), to find A^*. Your algorithm can use only $O(n^2)$ space. What is its time complexity?

(e) Show that $A^+ = A \times A^*$, where matrix multiplication is defined as $A^+(i,j) = \vee_{k=1}^{n}(A(i,k) \wedge A^*(k,j))$. The operation $\vee$ is the logical **or** operation, and $\wedge$ the logical **and** operation. Hence A^+ may be computed from A^*.

All Pairs Shortest Paths (APSP)

- **all edge weights are nonnegative** : use **Dijkstra's algorithm**
  - PQ = linear array : O (V^3 + VE) = O ($\mathbf{V^3}$)
  - PQ = binary heap : O (V^2lgV + EVlogV) = O ($\mathbf{V^3 logV}$) for dense graphs
    - **better only for sparse graphs**
  - PQ = fibonacci heap : O (V^2logV + EV) = O ($\mathbf{V^3}$) for dense graphs
    - **better only for sparse graphs**
- **negative edge weights : use Bellman-Ford algorithm**
  - O ($\mathbf{V^2E}$) = O ($\mathbf{V^4}$) on dense graphs

Dijkstra's algorithm

- **Dijkstra's algorithm**, conceived by 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.

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.
- 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.
- Edward F. Moore also published the same algorithm in 1957, and for this reason it is also sometimes called the **Bellman–Ford–Moore algorithm**.[1]