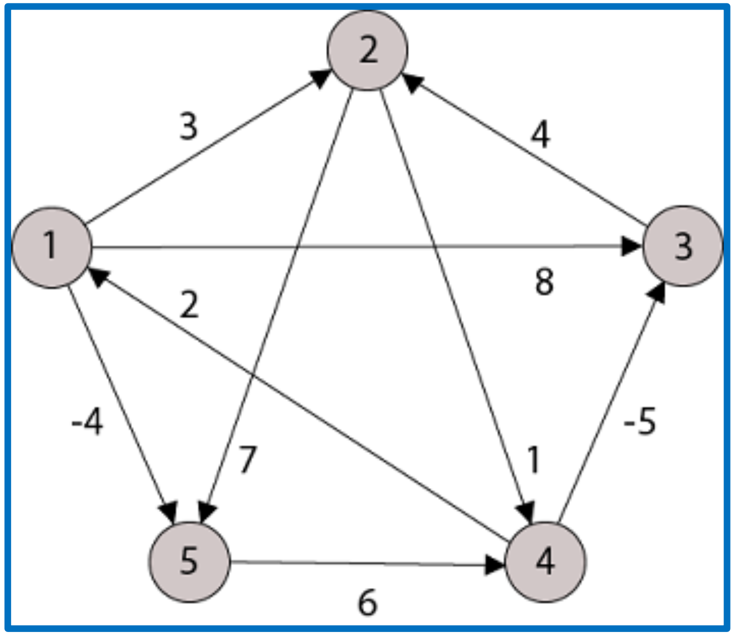
All-Pairs Shortest Path In A Graph



Partha P Chakrabarti
Indian Institute of Technology Kharagpur

Approaches to All-Pair Shortest Paths

Problem: Given a weighted directed Graph G = (V, E), find the shortest (cost) path between all pairs of vertices in G.

Case 1: For Directed Acyclic Graphs (DAGs), the recursive algorithm discussed earlier can be extended by computing the all-pair paths at every node during the recursion.

Best first search

Case 2: For Graphs with positive edge costs, we can adapt the single source algorithm to continue to find the shortest path from s to all nodes (continue till OrQ is empty). We now repeat that for all nodes as source nodes.

Case 3: For Graphs which may have negative edges but no negative edge cycles. We will discuss two methods, namely, Matrix Multiplication based method and the Floyd-Warshall Algorithm

Case 4: For graphs which may also have negative edge cycles, we will discuss the Bellman Ford Algorithm

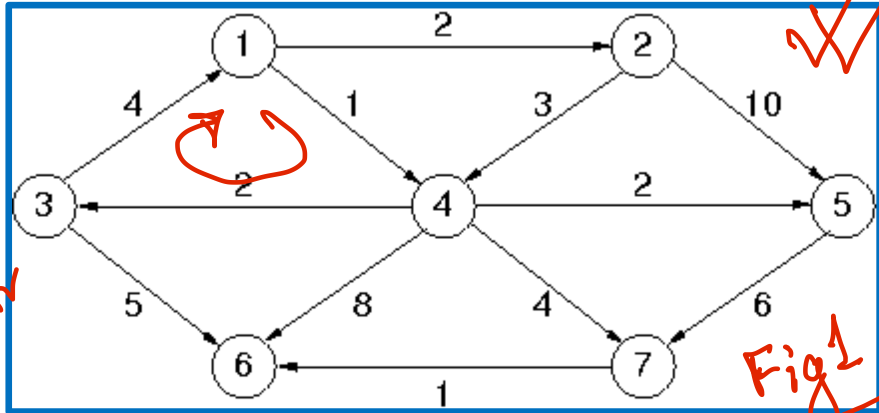
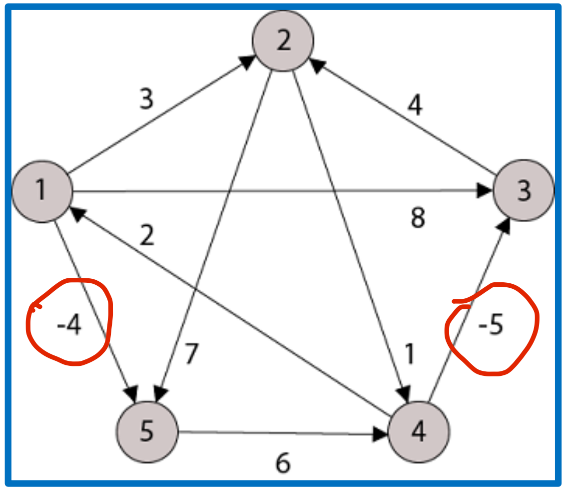
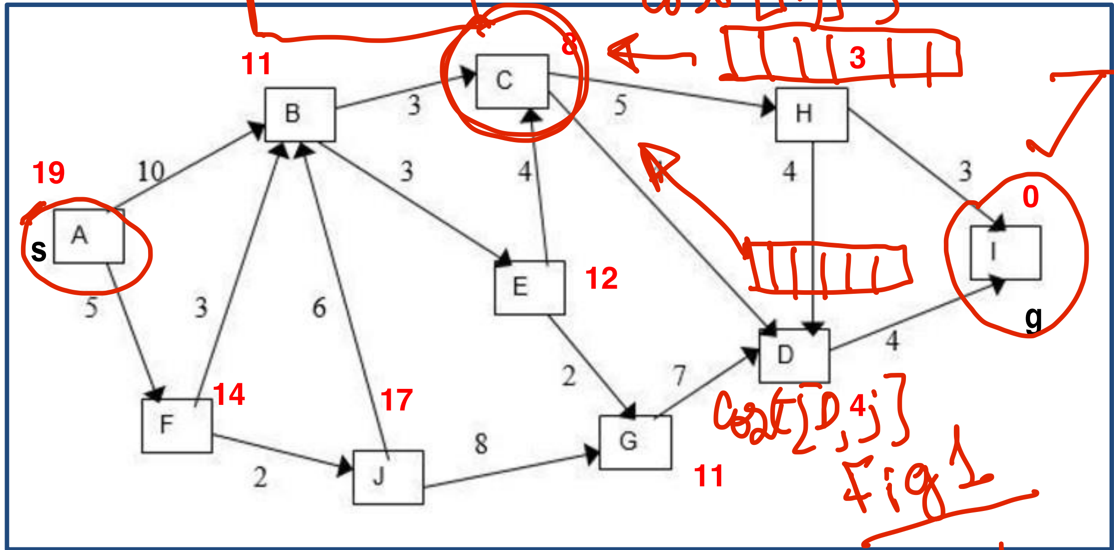


Fig 1



Modifying Shortest Cost Path Algorithm for DAGs



visited [i] indicates if node i is visited. / initially 0 /

cost[i] = cost of path from i to g initially infinity

succ(i) = {set of nodes to which node i is connected}

DFSP(node,g) {

  local variable value = ∞;

  visited[node] = 1;

  ~~if (node == g) { cost[node] = 0; return 0};~~

  for each n in succ(node) do {

    if (visited [n] == 0) DFSP(n);

    value = min (value, (cost[n] + C[node,n]))

  }

  cost[node] = value;

  return cost[node];

}

value[i] = ∞

value[node] = 0

Time Complexity O(|V| + |E|)   O(|V| * (|V| + |E|))

Will not work for Graphs which have cycles.

Works for negative edge cost DAGs.

Can be adapted to all pairs shortest paths for DAGs (Exercise).

Cost[i, j] = cost of SP from i to j

(initialization) = ∞ if i ≠ j, 0 if i = j

value[j] = min (value[n], cost[~~node~~,n,j] + C[node,n]

cost [node, j] = value[j]

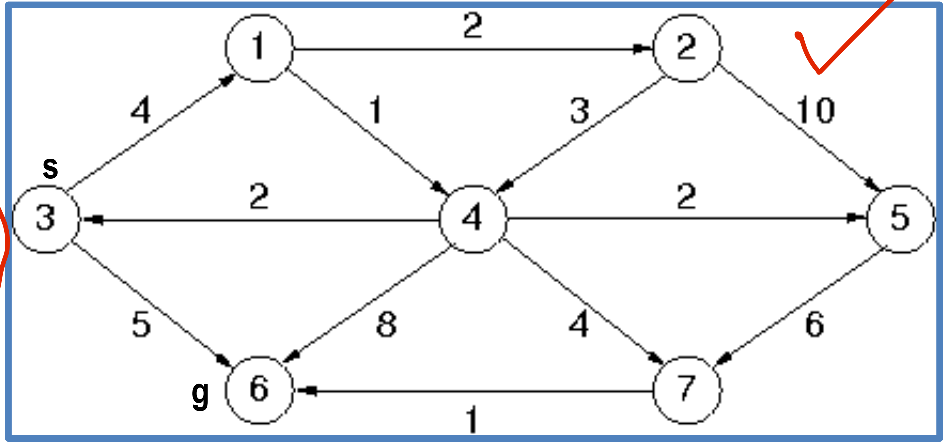
Modifying the Best First Search Algorithm

G = (V,E) / Assume positive edge costs/
visited[i] all initialized to 0
cost[j] cost from s to j, all initialized to ∞
Ordered Queue OrQ initially {}

```
BFSW(s,g) {
cost [s] = 0; OrQ = {s};
While OrQ != NULL {
    j = Remove_Min (OrQ); visited[j] = 1;
    ~~if (j == g) terminate with solution cost[j];~~
  For each k in succ (j) {
  If (visited[k] == 0) {
        if (cost[k] > (cost[j] + C[j,k])) {
            cost[k] = cost[j] + C[j,k];
            Insert_Reorder(OrQ,k);}
        }
  } }
~~If OrQ is empty terminate ("No Solution");~~
} / This method is  called Dijkstra's Algorithm /
```

$O(|V| * (|E| \log |V|))$

cost REVISE



	Queue OrQ with node costs	Node Removed
1	{1[0]}	1 [0]
2	{4[1], 2[2]}	4 [1]
3	{2[2], 3[3], 5[3], 7[5], 6[9]}	2 [2]
4	{3[3], 5[3], 7[5], 6[9]}	3 [3]
5	{5[3], 7[5], 6[8]}	5 [3]
6	{7[5],6[8])	7[5]
7	{6[6]}	6 [6]

Whenever a node is removed from OrQ, the best cost path to that node has been obtained. (Detailed proof is left as exercise)

Complexity is O(|E| log |E|); that is, O(|E| log |V|) using MinHeap or Balanced Tree. May also be implemented by an array in $O(|V|^2 + |E|)$

Bellman Ford Algorithm

visited [i] indicates if node i is visited. / initially 0 /
cost[i] = cost of path from s to i, initially infinity
succ(i) = {set of nodes to which node i is connected}
Parent[i] are parent pointers of shortest path, initialized to NULL

```
Bellman Ford(s) {
cost[s] = 0;
For i = 1 to |V| - 1 {
For each edge (n,k) in E {
          if (cost[k] > (cost[n] + C[n,k])) {
                  cost[k] = cost[n] + C[n,k];
                  Parent[k] =n };
                  } }
For each edge (n,k) in E {
      if (cost[k] > cost[n] + C[n,k]) return ("Negative Cycle")
      }
 return("Success")
}
```

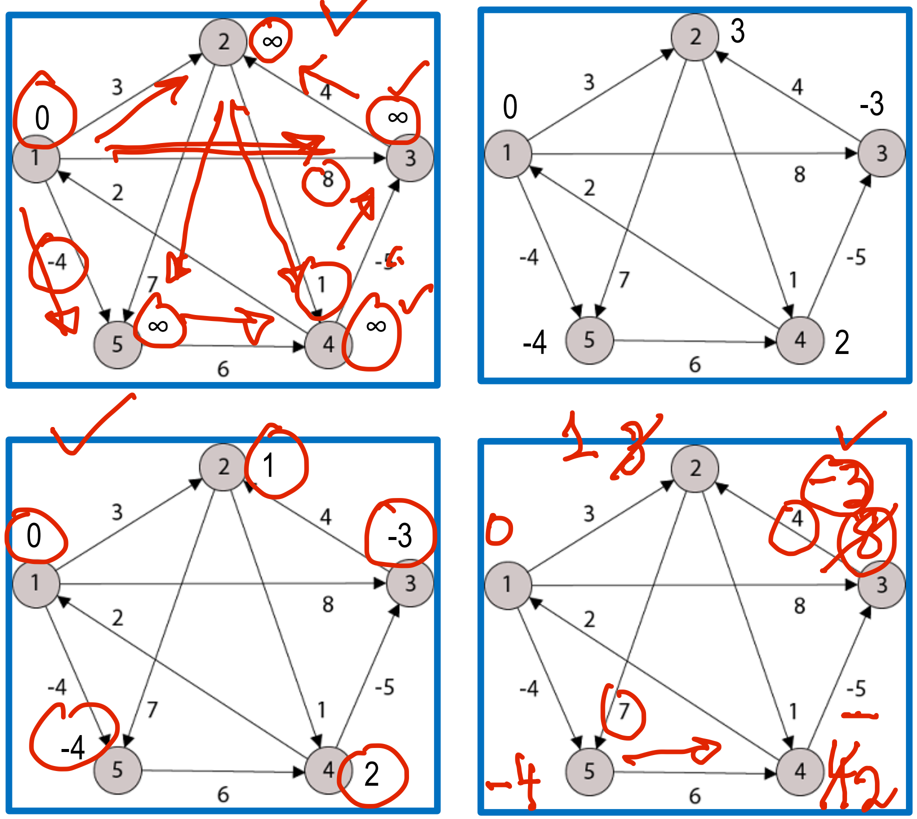
cost[n] + C[n,k] → K cost[k]

n

cost revision step

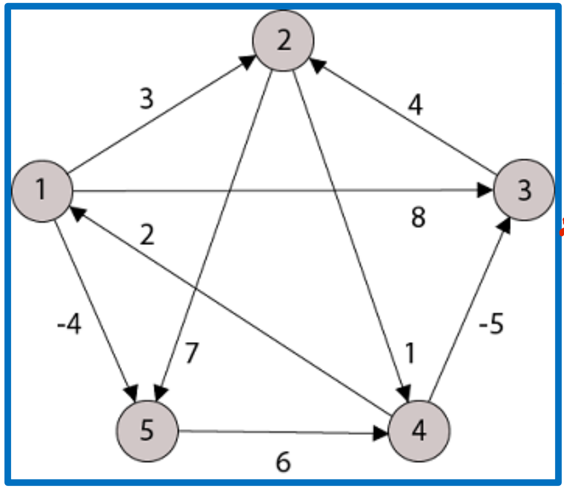
Time Complexity O(|E|*|V|) from s to all other nodes.
Works for negative edge cost graphs with negative edge loops.
For all-pairs, we run for each node as start node to get an $\boxed{O(|E| * |V|^2)}$ Algorithm.

s = Node 1



Example taken from the book "Introduction to Algorithms" by Cormen, Leiserson, Rivest and Stein

Matrix Multiplication Based Method



D[i,j,p]

= cost of SP from i to j of length AT MOST p

D[i,j,k-1] + C[j,j]

RECURSIVE DEFINITION:

D[i,j,0] = 0 (if i = j), ∞ (if i != j)

D[i,j,k] = min { D[i,j,k-1], min { D[i,m,k-1] + C[m,j]}},
for all m in |V|

which is the same as: min { D[i,m,k-1] + C[m,j]}
since C[j,j] = 0 for all j;

∀m in V

Final Solution is D[i,j,n-1] where n = |V|

Analyzing the Recursive Definition we choose a Dynamic Programming Strategy using two 2-dimensional arrays D[n,n] for Memoization:

Top Down Recursive Scheme:

D[i,j,k] = done (1)
not done (-1)

Bottom-up Iterative Scheme:

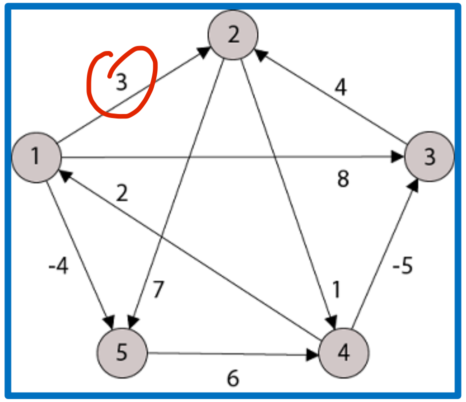
→ k = 1 to |V| - 1

→ (i,j)

→ min ∀m ∈ V

Time Complexity O(|V|^4) time

Matrix Multiplication Based Method: Example



Example taken from the book "Introduction to Algorithms" by Cormen, Leiserson, Rivest and Stein

D[0]

0	∞	∞	∞	∞
∞	0	∞	∞	∞
∞	∞	0	∞	∞
∞	∞	∞	0	∞
∞	∞	∞	∞	0

D[1]

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

D[2]

0	3	8	2	-4
3	0	-4	1	7
∞	4	0	5	11
2	-1	-5	0	-2
8	∞	1	6	0

D[3]

0	3	-3	2	-4
3	0	-4	1	-1
7	4	0	5	11
2	-1	-5	0	-2
8	5	1	6	0

D[4]

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

RECURSIVE DEFINITION:

D[i,j,0] = 0 (if i = j), ∞ (if i != j)

D[i,j,k] = min { D[i,j,k-1], min { D[i,m,k-1] + C[m,j]}},
for all m in |V|

which is the same as: min { D[i,m,k-1] + C[m,j]}
since C[j,j] = 0;

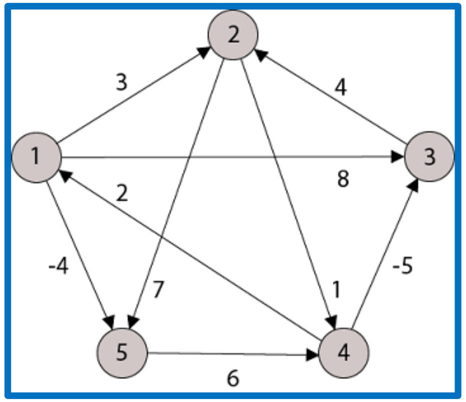
Final Solution is D[i,j,n-1] where n = |V|

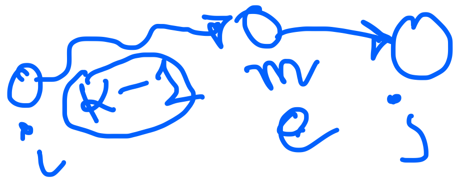
D[2,1,2] ∝
= D[2,1,1] + C[1,1] ∝
0 D[2,2,1] + C[2,1] ∝
0 D[2,3,1] + C[3,1] ∝
1 D[2,4,1] + C[4,1] 2
7 D[2,5,1] + C[5,1] ∝

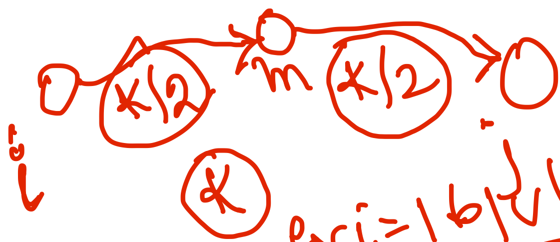
Detection of -ve cost cycle?

$\boxed{3}$

Improved Matrix Multiplication Based Method







for i=1 to |V|

RECURSIVE DEFINITION:

D[i,j,1] = 0 if (i==j)

= C[i,j] if (i!=j)

D[i,j,2k] = min { D[i,m,k] + D[m,j,k] },

for all m in |V|

Final Solution is D[i,j,n-1] where n = |V|

Analyzing the Recursive Definition we choose a Dynamic Programming Strategy using Two 2-dimensional arrays D[n,n] for Memoization:

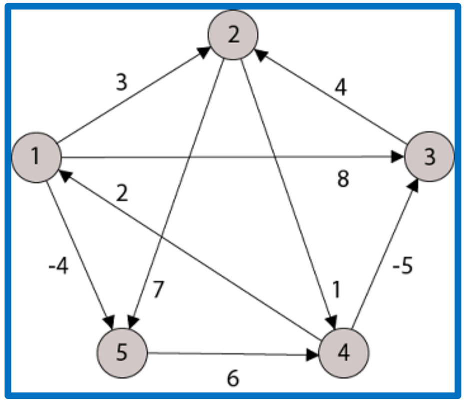
Top Down Recursive Scheme:

Bottom-up Iterative Scheme:

for k = 1 to log|V|

Time Complexity O(|V|^3 log |V|) time

Improved Matrix Multiplication Based Method: Example



Example taken from the book “Introduction to Algorithms” by Cormen, Leiserson, Rivest and Stein

D[1]

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

D[2]

0	3	8	2	-4
3	0	-4	1	7
∞	4	0	5	11
2	-1	-5	0	-2
8	∞	1	6	0

D[4]

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

RECURSIVE DEFINITION:

D[i,j,1] = 0 if (i==j)

= C[i,j] if (i!=j)

D[i,j,2k] = min { D[i,m,k] + D[m,j,k] }, for all m in |V|

Final Solution is D[i,j,n-1] where n = |V|

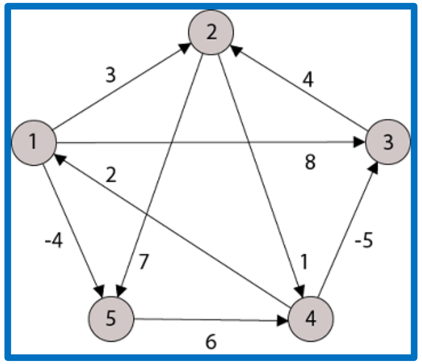
~~D[16]~~

~~D[256]~~

D[1] , D[2] , D[4] , D[6]

D[8] , log|V|

Floyd Warshall Algorithm



V = {1, 2, 3, ... , n}

F[i, j, k] = cost of SP from i, to j through nodes {1 to k}

i {1,...k-1} j

i {1,.....k-1} k {1,...k-1} j

RECURSIVE DEFINITION:

F[i,j,0] = 0 if (i==j), and = C[i,j] otherwise

F[i,j,k] = min { F[i,j,k-1], F[i,k,k-1] + F[k,j,k-1]}

Final Solution is F[i,j,n] where n = |V|

k

Analyzing the Recursive Definition we choose a Dynamic Programming Strategy using Two 2-dimensional arrays D[n,n] for Memoization:

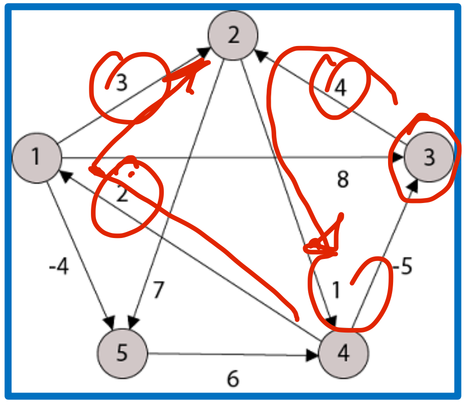
Top Down Recursive Scheme:

Bottom-up Iterative Scheme:

F[i, j, k] = MIN {F[i, j, k-1], F[i, k, k-1] + F[k, j, k-1]}

Time Complexity O(|V|^3) time

Floyd Warshall Algorithm: Example



Example taken from the book "Introduction to Algorithms" by Cormen, Leiserson, Rivest and Stein

F[0]

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

F[1]

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

F[2]

0	3	8	4	-4
∞	0	∞	1	7
∞	4	0	5	11
2	5	-5	0	-2
∞	∞	∞	6	0

F[3]

0	3	8	4	-4
∞	0	∞	1	7
∞	4	0	5	11
2	-1	-5	0	-2
∞	∞	∞	6	0

F[4]

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

F[5]

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

RECURSIVE DEFINITION:

F[i,j,0] = 0 if (i==j) and = C[i,j] otherwise

F[i,j,k] = min { F[i,j,k-1], F[i,k,k-1] + F[k,j,k-1]}

Final Solution is F[i,j,n] where n = |V|

$O(|V|^3)$

F[i,j,k] = cost of min cost path thru nodes {1,... k}

Summary: All-Pair Shortest Paths

Case 1: For Directed Acyclic Graphs (DAGs), the Recursive DFS Algorithm discussed earlier can easily be extended by computing the all-pair paths at every node. $O(|V|^2 + |V|*|E|)$

Case 2: For Graphs with positive edge costs, we can adapt the single source Best First Search (Dijkstra's) Algorithm to continue to find the shortest path from s to all nodes (Continue till OrQ is empty). We repeat that for all nodes as source nodes. $O(|V|*(|E| \log |V|))$

Case 3: For Graphs which may have negative edges but no negative edge cycles. We discussed two methods, namely,

Matrix Multiplication Based $O(|V|^3 \log |V|)$ and

Floyd-Warshall Algorithm $O(|V|^3)$

Case 4: For graphs which may also have negative edge cycles, we discussed the Bellman Ford Algorithm $O(|E| * |V|^2)$

Thank you