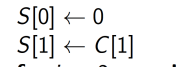
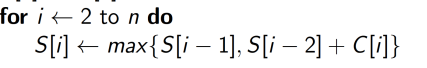
Use the dynamic programming algorithm to solve the instance of the coin row problem : **20,50,20,5,10,20,5** ?

Solution:

We would fill the table according to the operation

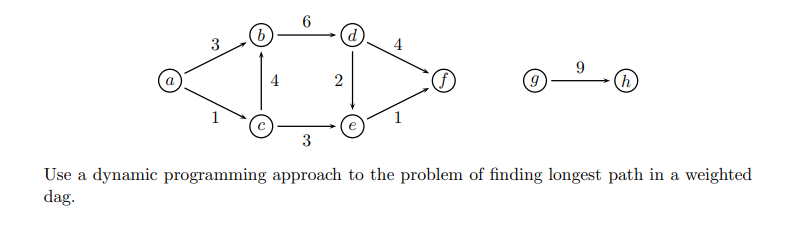




|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Coin ,i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Value C[i] | - | 20 | 50 | 20 | 5 | 10 | 20 | 5 |
| S[i] | S[0]=0 | S[1] = C[1] = 20 | S[2]= max {S[2-1], S[2-2]+C[2]}= max{S[1],S[0]+C[2]} = max{20,0+50}=max{20,50}=50 | S[3]= max{S[3-1],S[3-2]+C[3]}=max{S[2],S[1]+C[3]}  = max{S[2],S[1]+C[3]} = max{50,20+20}=max{50,40}=50 | max{S[3],S[2]+C[4]}=  max{50,50+5}=55 | Max{55,50+10}  = 60 | Max{60,55+20}=  75 | Max{75,60+5}= Max{75,65}=75. |

So we have maximum value for the given knapsack to be equal to **75.** So the optimal selection would be 20+50+5 = At index 2, 4, 6 coins of values 50,5,20. .

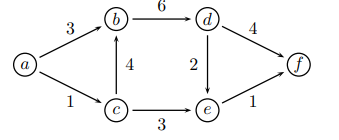
Q2. Consider the problem of finding the length of a “longest path” in a weighted, not necessarily connected dag. We assume that all of the weights are positive and that a longest path is a path whose edge weights add up to the maximal possible value. For example, for the following graph, the longest path is of length 15:



Solution:

DAG is an acyclic graph with **NO CYCLES**.

Given graph

has no directed cycles in it so it is a DAG.

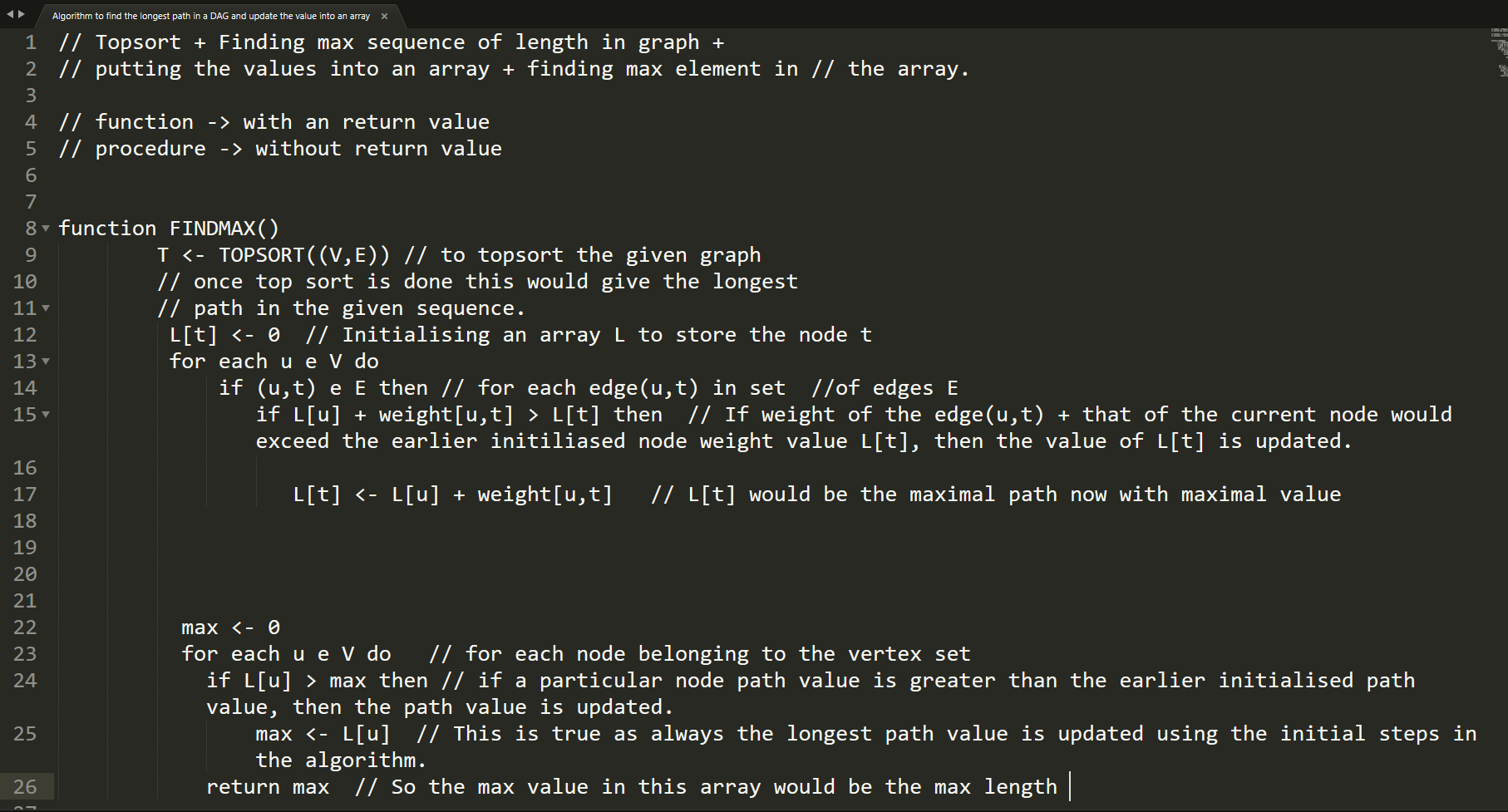
So we have to use the dynamic programming approach to find the DAG in the graph.

This is easy if we would process the nodes in a topologically sorted order.

* For each node t we would want to find its longest distance from a source and store these distances in an array L. For each of the node t , we would want to calculate



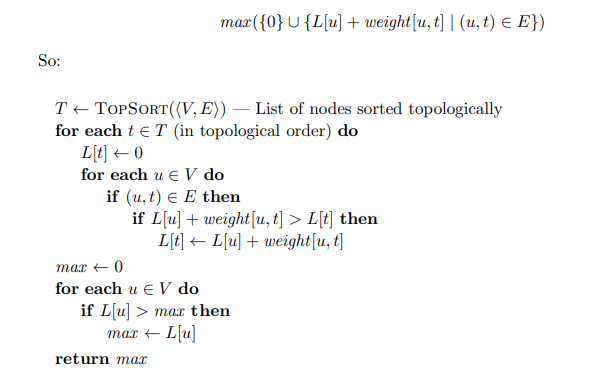
* In the given algorithm we would initial top sort the given graph, and after top sort for each of edge we would compare its neighbors with it and see if the weight would correspond to maximal value. And then all of these values are put in an array, after which a linear approach is taken to find the max element in the array.



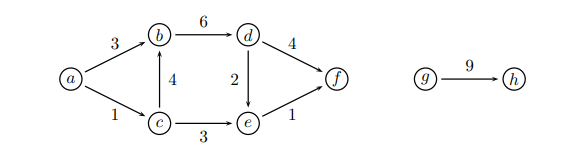
Extra notes:

Dynamic programming is an algorithm design technique that is sometimes applicable when we want to solve a recurrence relation and the recursion involves overlapping instances.

* The primary goal of dynamic programming is to find the optimal solution: The one with lowest cost, or highest profit or subject to some constraints.
* In both contexts it refers to simplifying a complicated problem by breaking it down into simpler sub-problems in a [recursive](https://en.wikipedia.org/wiki/Recursion) manner.
* In computer science, if a problem can be solved optimally by breaking it into sub-problems and then recursively finding the optimal solutions to the sub-problems, then it is said to have optimal substructure.



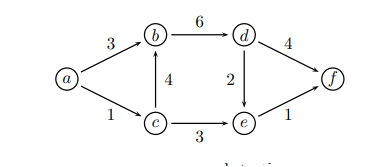
Tracing the above algorithm:



Doing TOPSORT for the above graph would give us:

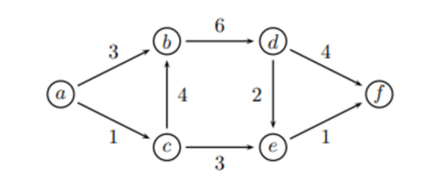
**TO do TOPSORT we would have the direct approach, remove all nodes with no incoming edges and its associated edges.**

* Initially remove “g” as it has no incoming edges and remove all of the edges going out from it.



**Sequence : g**

* **Remove “h” and add it to the sequence.**

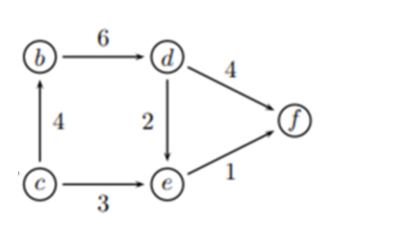


**Sequence : g,h**

**Remove “a” and add it to the sequence.**

**Sequence : g,h,a**

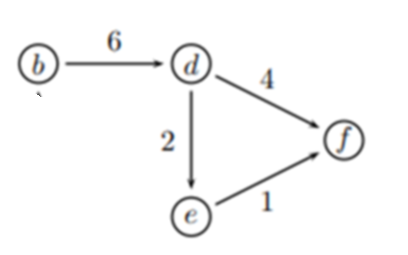
**Resulting graph:**



* Remove c and add it to the sequence. (No incoming edge)

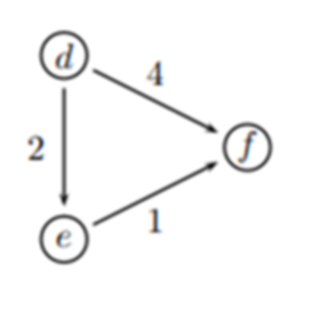
Sequence: g,h,a,c

Resulting graph:

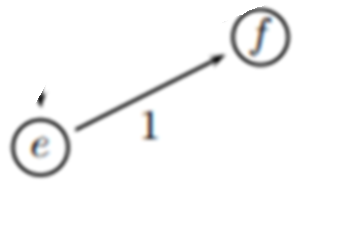


* Remove b with no incoming edges and add it to the sequence.

Sequence: g,h,a,c,b



* Remove d and add it to the sequence (NO INCOMING EDGES).



Sequence : g,h,a,c,b,d

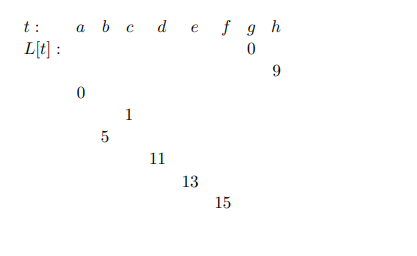
.

* Remove e and add it to the sequence (NO INCOMING EDGES).

Sequence : g,h,a,c,b,d,e

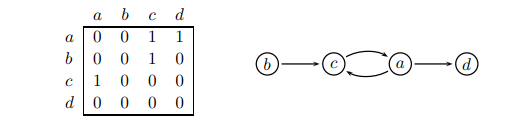


* Remove f and add it to the sequence (NO INCOMING EDGES).
* **Sequence : g,h,a,c,b,d,e,f**



**So the max distance from a to h is 1+4+6+4 = 11+4 = 15 .**

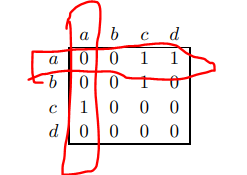
**Q. Work through the Warshall’s algorithm to find the transitive closure of the binary relation given by this table.**



**Solution:**

**Step 1**

**Initially when k = a.**



A -> adjacency matrix.

A[c,c] = 1

i values A(i,k) = c

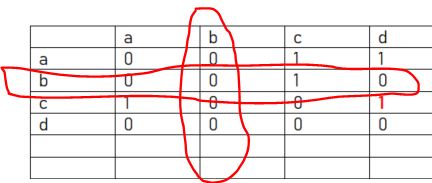
j values A(a,j) = c,d -> using **a** as the **STEPPING STONE**.

A[c,d] = 1 (A path is possible from c to d using a as the stepping stone)

We would then update the corresponding A matrix as above corresponding to **A[c,d]** in the matrix above.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | a | b | c | d |  |  |  |
| a | 0 | 0 | 1 | 1 |  |  |  |
| b | 0 | 0 | 1 | 0 |  |  |  |
| c | 1 | 0 | 0 | **1** |  |  |  |
| d | 0 | 0 | 0 | 0 |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

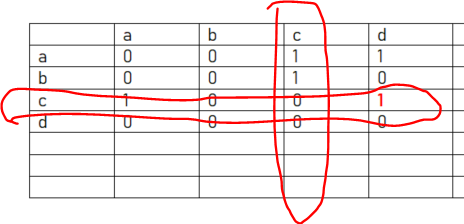
**When the value of k = b.**



No change as we have zeroes throughout for i values corresponding to A(i,k).

Step 3

When k = c



i values A(i,k) = a,b

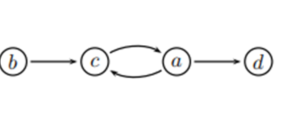
j values A(k,j) = a,d

A[a,a] = 1

A[a,d] = already 1

A[b,a] = 1 (traversal from b to a using c as an intermediate node) – IT IS POSSIBLE

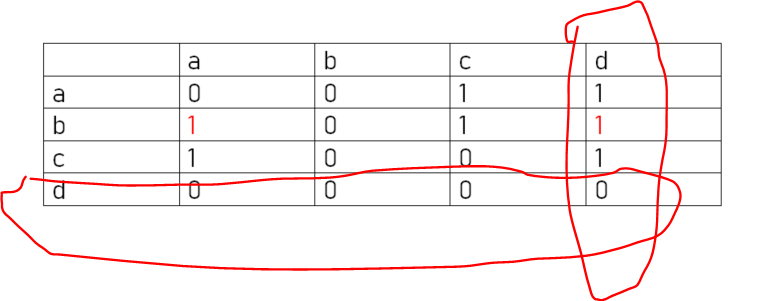
A[b,d] = 1 (traversal from b to d using c as an intermediate node) – IT IS POSSIBLE



We would then update the table according to the above values then.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | a | b | c | d |
| a | 0 | 0 | 1 | 1 |
| b | 1 | 0 | 1 | 1 |
| c | 1 | 0 | 0 | 1 |
| d | 0 | 0 | 0 | 0 |

When k = d

 **Zero throughout when k = d. So then we would not update the matrix accordingly.**

**Final adjacency matrix to compute the Transitive closure using Warshall Algorithm:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | a | b | c | d |
| a | 1 | 0 | 1 | 1 |
| b | 1 | 0 | 1 | 1 |
| c | 1 | 0 | 1 | 1 |
| d | 0 | 0 | 0 | 0 |

**Corrections: A[a,a] = 1 (From step 3 when k =c), A[c,c]=1 (From step1 when k =a).**

**An example of Floyd’s Algorithm: All pairs shortest Paths:**

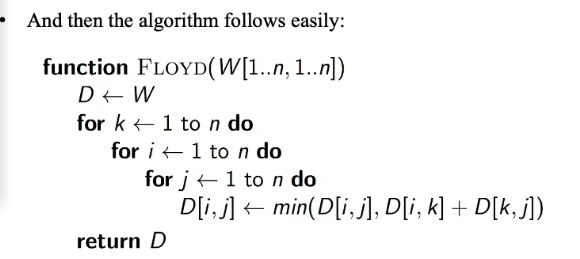
**Here we it is the Distance Matrix, so we have to consider the weight of each edge while considering the path or edge from one node to the another.**

**And we would update the Distance or weight values in the matrix (Distance Matrix).**

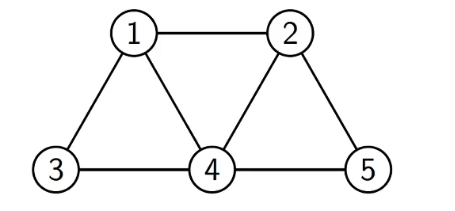
**The Distance Matrix for an Undirected Graph would be Symmetric.**

* Already updated distances need not be updated again.

The Floyd’s algorithm is given to be as:



**Algorithm in action :**



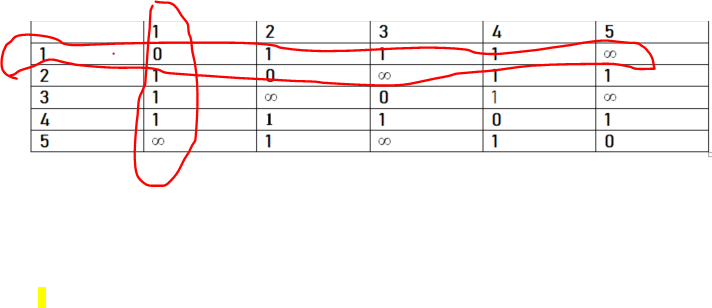
The Distance matrix for the above graph is given to be as:

This is the initial distance matrix (for the unweighted graph and is symmetric)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **1** | **2** | **3** | **4** | **5** |
| **1** | **0** | **1** | **1** | **1** | **∞** |
| **2** | **1** | **0** | **∞** | **1** | **1** |
| **3** | **1** | **∞** | **0** | 1 | **∞** |
| **4** | **1** | **1** | **1** | **0** | **1** |
| **5** | **∞** | **1** | **∞** | **1** | **0** |

**REMEMBER HERE WE WOULD ALWAYS UPDATE THE DISTANCE AND THE WEIGHT OF THE EDGE UNLIKE ADJACENCY MATRIX.**

Now we would consider when k = 1



D[k,i] = 2,3,4

D[j,k] = 2,3,4

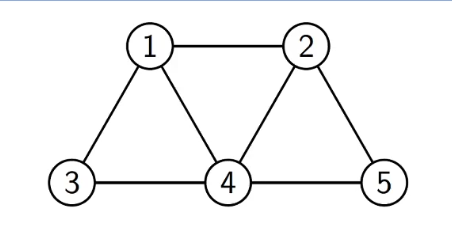
We have the combinations as D[2,2],D[2,3],D[2,4],D[3,2],D[3,3],D[3,4],D[4,2],D[4,3],D[4,4]

Now D[2,2],D[3,3],D[4,4] is set to be ZERO already as it’s an undirected graph.

Now D[2,4], D[3,4],D[4,2],D[4,3] is already set to 1. So we don’t have to consider that as well.

We have to consider only D[2,3],D[3,2] .

Now we consider D[2,3] and D[3,2] using k=1 as the PIVOT.



Now we see if it would be possible to traverse from 2 to 3 using 1 as the PIVOT. And it is possible to traverse from 2 to 3 using 1 as the PIVOT with a distance or a weight of 1 + 1 = 2.

And similarly as it is an undirected graph it is possible to travel from 3 to as well, using a distance of 1+1 = 2.

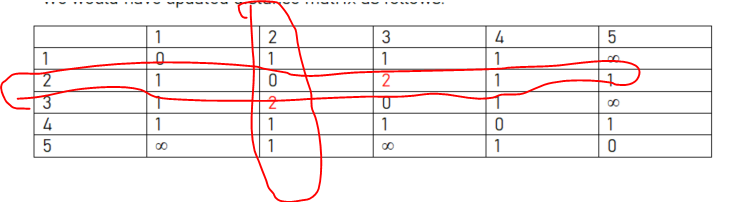
So D[2,3] = 2

D[3,2] = 2

We would have updated distance matrix as follows:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 |
| 1 | 0 | 1 | 1 | 1 | ∞ |
| 2 | 1 | 0 | 2 | 1 | 1 |
| 3 | 1 | 2 | 0 | 1 | ∞ |
| 4 | 1 | 1 | 1 | 0 | 1 |
| 5 | ∞ | 1 | ∞ | 1 | 0 |

**.Now when k =2,**

D[i,k] = 1,3,4,5

D[k,j] = 1,3,4,5

Similarly as above we can avoid {(1,1),(2,2),(3,3),(4,4),(5,5)} as all of the values would be 0.

We have the other combinations as (1,3), (1,4),(1,5),(3,1),(3,4),(3,5),(4,1),(4,3),(4,5),(5,1),(5,3),(5,4)

So ALREADY 1 values would be (1,4),(3,1),(3,4),(4,3),(4,1),(1,3).

∞ = {(3,5),(5,3), (1,5),(5,1)}

The distance from 3 to 5 = 1+1+1 = 3 (3 to 1 to 2 to 5).

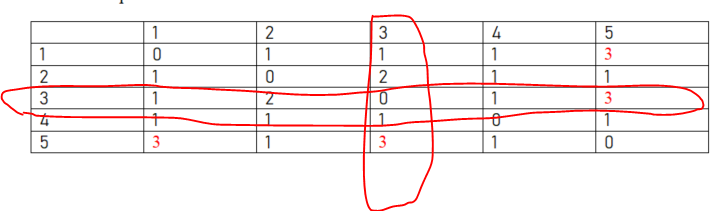
The distance from 5 to 3 (through 1) = 1+1+1= 3 (3 to 1 to 2 to 5).

The distance from (1, 5) and (5, 1) would be equal to = 1+1 = 2 (1 to 2 to 5)

So would update the distance matrix as follows:

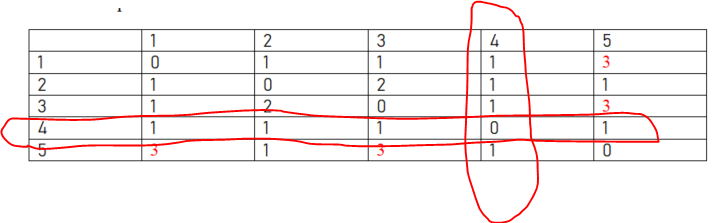
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 |
| 1 | 0 | 1 | 1 | 1 | 3 |
| 2 | 1 | 0 | 2 | 1 | 1 |
| 3 | 1 | 2 | 0 | 1 | 3 |
| 4 | 1 | 1 | 1 | 0 | 1 |
| 5 | 3 | 1 | 3 | 1 | 0 |

When k = 3



No major change happens.

When k = 4,



D[i,k] = 1,2,3,5

D[k,j] = 1,2,3,5

We can ignore {(1,1), (2,2), (3,3), (4,4), (5,5)} as they all would be 0 (In an undirected graph with a distance matrix that is symmetric. )

The other combinations would be {(1,2) , 1,3, 1,5, 2,1,2,3,2,4,2,5,3,1,3,2,3,5,5,1,5,2,5,3