Design and Analysis of Algorithms (BCS-28)

[B Tech IIIrd Year, Vth Sem, Session: 2020-21]

Prof. Rakesh Kumar
Department of Computer Science and Engineering
MMM University of Technology Gorakhpur-273010
Email: rkcs@mmmut.ac.in, rkiitr@gmail.com

UNIT II

Greedy Algorithm

- "Greedy Method finds out of many options, but you have to choose the best option.“
- Greedy algorithms build a solution part by part, choosing the next part in such a way, that it gives an immediate benefit. This approach never reconsiders the choices taken previously. This approach is mainly used to solve optimization problems.
- In this approach/method we focus on the first stage and decide the output, don't think about the future.

- **Areas of Application**
  - Greedy approach is used to solve many problems, such as
  - Finding the shortest path between two vertices using Dijkstra’s algorithm.
  - Finding the minimal spanning tree in a graph using Prim’s /Kruskal’s algorithm, etc.

Greedy Algorithm: Properties

- "A greedy algorithm works if a problem exhibits the following two properties:
  - **Greedy Choice Property:** A globally optimal solution can be reached at by creating a locally optimal solution. In other words, an optimal solution can be obtained by creating "greedy" choices.
  - **Optimal substructure:** Optimal solutions contain optimal subsolutions. In other words, answers to subproblems of an optimal solution are optimal.

- **Steps for achieving a Greedy Algorithm are:**
  - **Feasible:** Here we check whether it satisfies all possible constraints or not, to obtain at least one solution to our problems.
  - **Local Optimal Choice:** In this, the choice should be the optimum which is selected from the currently available
  - **Unalterable:** Once the decision is made, at any subsequence step that option is not altered.

Components of Greedy Algorithm

- Greedy algorithms have the following five components −
  - **A candidate set** − A solution is created from this set.
  - **A selection function** − Used to choose the best candidate to be added to the solution.
  - **A feasibility function** − Used to determine whether a candidate can be used to contribute to the solution.
  - **An objective function** − Used to assign a value to a solution or a partial solution.
  - **A solution function** − Used to indicate whether a complete solution has been reached.

- **Example:**
  - machine scheduling
  - Fractional Knapsack Problem
  - Minimum Spanning Tree
  - Huffman Code
  - Job Sequencing
  - Activity Selection Problem

Knapsack Problem

- Given a set of items, each with a weight and a value, determine a subset of items to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.
- The knapsack problem is in combinatorial optimization problem. It appears as a subproblem in many, more complex mathematical models of real-world problems.

- **Applications**
  - In many cases of resource allocation along with some constraint, the problem can be derived in a similar way of Knapsack problem. Following is a set of example.
  - Finding the least wasteful way to cut raw materials
  - portfolio optimization
  - Cutting stock problems

Knapsack Problem: Problem Scenario

- **Problem Scenario**
  - A thief is robbing a store and can carry a maximal weight of W into his knapsack. There are n items available in the store and weight of i^{th} item is w_i and its profit is p_i. What items should the thief take?
  - In this context, the items should be selected in such a way that the thief will carry those items for which he will gain maximum profit. Hence, the objective of the thief is to maximize the profit.
- Based on the nature of the items, Knapsack problems are categorized as
  - Fractional Knapsack
  - Knapsack

Fractional Knapsack: Generalized Solution to the problem

- In this case, items can be broken into smaller pieces, hence the thief can select fractions of items.
- According to the problem statement,

  There are **n** items in the store

  Weight of **i**th item $wi>0$

  Profit for **i**th item $pi>0$

  and

  Capacity of the Knapsack is **W**
- In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction $\boldsymbol{x}_i$ of **i**th item.

$0 \leqslant xi \leqslant 1$

Fractional Knapsack: Generalized Solution to the problem

The **i**th item contributes the weight *xi*.*wi*

to the total weight in the knapsack and profit *xi*.*pi*to the total profit.

Hence, the objective of this algorithm is to $maximize \sum_{n=1}^{n}(x_i . pi)$

subject to constraint, $\sum_{n=1}^{n}(x_i . wi) \leqslant W$

It is clear that an optimal solution must fill the knapsack exactly, otherwise we could add a fraction of one of the remaining items and increase the overall profit.

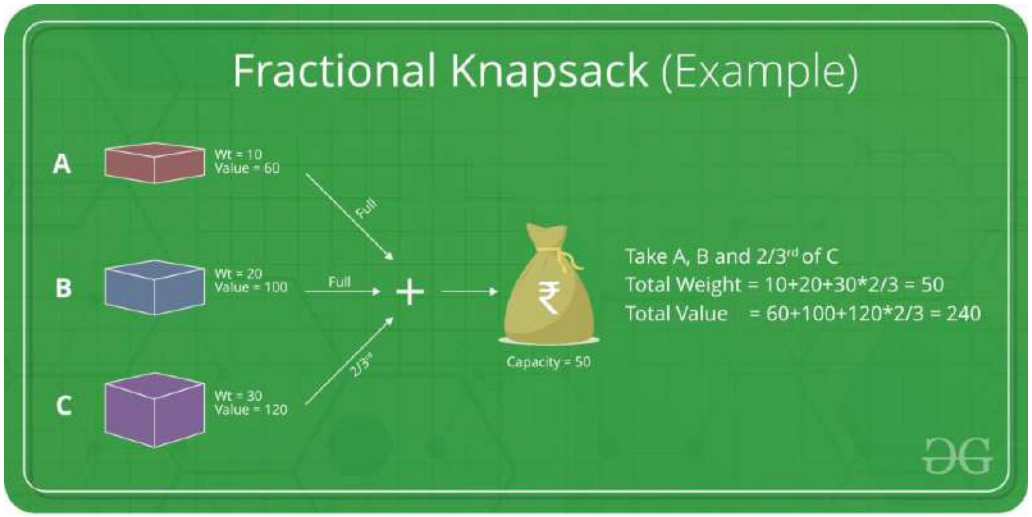
Thus, an optimal solution can be obtained by $\sum_{n=1}^{n}(x_i . wi) = W$

In this context, first we need to sort those items according to the value of *pi/wi*, so that *pi*+1/*wi*+1 ≤ *pi/wi* . Here, ***x*** is an array to store the fraction of items.

- **Analysis**
  - If the provided items are already sorted into a decreasing order of **pi/wi** then the whileloop takes a time in ***O(n)***; Therefore, the total time including the sort is in ***O(n logn)***.

Fractional Knapsack

- Fractions of items can be taken rather than having to make binary (0-1) choices for each item.
- **Steps to solve the Fractional Problem:**
  - Compute the value per pound for each item.
  - Obeying a Greedy Strategy, we take as possible of the item with the highest value per pound.
  - If the supply of that element is exhausted and we can still carry more, we take as much as possible of the element with the next value per pound.
  - Sorting, the items by value per pound, the greedy algorithm run in O (n log n) time.



Fractional Knapsack : Method

- Fractions of items can be taken rather than having to make binary (0-1) choices for each item.
- Fractional Knapsack Problem can be solvable by greedy strategy whereas 0 - 1 problem is not.
- **Steps to solve the Fractional Problem:**
  - Compute the value per pound for each item.
  - Obeying a Greedy Strategy, we take as possible of the item with the highest value per pound.
  - If the supply of that element is exhausted and we can still carry more, we take as much as possible of the element with the next value per pound.
  - Sorting, the items by value per pound, the greedy algorithm run in O (n log n) time.

Fractional Knapsack : Ist Example

Let us consider that the capacity of the knapsack ***W* = 60** and the list of provided items are shown in the following table −

Item	A	B	C	D
Profit	280	100	120	120
Weight	40	10	20	24
Ratio $(\frac{p_i}{w_i})$	7	10	6	5

As the provided items are not sorted based on **piwi**. After sorting, the items are as shown in the following table.

Item	B	A	C	D
Profit	100	280	120	120
Weight	10	40	20	24
Ratio $(\frac{p_i}{w_i})$	10	7	6	5

Fractional Knapsack : Example (Contd…)

- **Solution**

After sorting all the items according to *pi/wi*.

First all of ***B*** is chosen as weight of ***B*** is less than the capacity of the knapsack. Next, item ***A*** is chosen, as the available capacity of the knapsack is greater than the weight of ***A***. Now, ***C*** is chosen as the next item. However, the whole item cannot be chosen as the remaining capacity of the knapsack is less than the weight of ***C***.

Hence, fraction of ***C*** (i.e. (60 − 50)/20) is chosen.

Now, the capacity of the Knapsack is equal to the selected items. Hence, no more item can be selected.

The total weight of the selected items is **10 + 40 + 20 * (10/20) = 60**

And the total profit is **100 + 280 + 120 * (10/20) = 380 + 60 = 440**

This is the optimal solution. We cannot gain more profit selecting any different combination of items.

Fractional Knapsack : 2nd Example

(Contd…)

- **Example:** Consider 5 items along their respective weights and values: -

I = (I_1,I_2,I_3,I_4,I_5)

w = (5, 10, 20, 30, 40)

v = (30, 20, 100, 90,160)

The capacity of knapsack W = 60

Now fill the knapsack according to the decreasing value of p_i.

First, we choose the item I_i whose weight is 5.

Then choose item I_3 whose weight is 20. Now,the total weight of knapsack is 20 + 5 = 25

Now the next item is I_5, and its weight is 40, but we want only 35, so we chose the fractional part of it

$$\text{i.e., } 5 \times \frac{5}{5} + 20 \times \frac{20}{20} + 40 \times \frac{35}{40}$$

Weight = 5 + 20 + 35 = 60

Maximum Value:-

$$30 \times \frac{5}{5} + 100 \times \frac{20}{20} + 160 \times \frac{35}{40}$$

= 30 + 100 + 140 = 270 (Minimum Cost)

Fractional Knapsack : Example (Contd…)

- **Solution:**

ITEM	w_i	v_i
I_1	5	30
I_2	10	20
I_3	20	100
I_4	30	90
I_5	40	160

Fractional Knapsack : Example

(Contd…)

- Taking value per weight ratio i.e. $p_i = \frac{v_i}{w_i}$

ITEM	w_i	v_i	$P_i = \frac{v_i}{w_i}$
I_1	5	30	6.0
I_2	10	20	2.0
I_3	20	100	5.0
I_4	30	90	3.0
I_5	40	160	4.0

Fractional Knapsack : Example (Contd…)

- Taking value per weight ratio i.e. $p_i = \frac{v_i}{w_i}$

ITEM	w_i	v_i	$P_i = \frac{v_i}{w_i}$
I_1	5	30	6.0
I_2	10	20	2.0
I_3	20	100	5.0
I_4	30	90	3.0
I_5	40	160	4.0

Fractional Knapsack : Example (Contd…)

- **Now, arrange the value of p_i in decreasing order.**

ITEM	w_i	v_i	$p_i = \frac{v_i}{w_i}$
I_1	5	30	6.0
I_3	20	100	5.0
I_5	40	160	4.0
I_4	30	90	3.0
I_2	10	20	2.0

Job Sequencing with deadline

- **Problem Statement**

  In job sequencing problem, the objective is to find a sequence of jobs, which is completed within their deadlines and gives maximum profit.

- **Solution**

  Let us consider, a set of ***n*** given jobs which are associated with deadlines and profit is earned, if a job is completed by its deadline. These jobs need to be ordered in such a way that there is maximum profit.

  It may happen that all of the given jobs may not be completed within their deadlines.

  Assume, deadline of $\mathbf{i^{th}}$ job $\boldsymbol{J_i}$ is $\boldsymbol{d_i}$ and the profit received from this job is $\boldsymbol{p_i}$. Hence, the optimal solution of this algorithm is a feasible solution with maximum profit.

  Thus, $D(i)>0$ for $1 \leqslant i \leqslant n$.

  Initially, these jobs are ordered according to profit, i.e. $p1 \geqslant p2 \geqslant p3 \geqslant ... \geqslant pn$.

Job Sequencing with deadline Algorithm

```
Algorithm: Job-Sequencing-With-Deadline (D, J, n, k)
D(0) := J(0) := 0
k := 1
J(1) := 1   // means first job is selected
for i = 2 … n do
   r := k
   while D(J(r)) > D(i) and D(J(r)) ≠ r do
      r := r - 1
   if D(J(r)) ≤ D(i) and D(i) > r then
      for l = k … r + 1 by -1 do
         J(l + 1) := J(l)
         J(r + 1) := i
         k := k + 1
```

Job Sequencing with deadline

- **Analysis**
  - In this algorithm, we are using two loops, one is within another. Hence, the complexity of this algorithm is $O(n^2)$
- **Example**
  - Let us consider a set of given jobs as shown in the following table. We have to find a sequence of jobs, which will be completed within their deadlines and will give maximum profit. Each job is associated with a deadline and profit.

Job	J_1	J_2	J_3	J_4	J_5
Deadline	2	1	3	2	1
Profit	60	100	20	40	20

Job Sequencing with deadline

- **Solution**
  - To solve this problem, the given jobs are sorted according to their profit in a descending order. Hence, after sorting, the jobs are ordered as shown in the following table.

Job	J_2	J_1	J_4	J_3	J_5
Deadline	1	2	2	3	1
Profit	100	60	40	20	20

- From this set of jobs, first we select J_2, as it can be completed within its deadline and contributes maximum profit.
- Next, J_1 is selected as it gives more profit compared to J_4.
- In the next clock, J_4 cannot be selected as its deadline is over, hence J_3 is selected as it executes within its deadline.
- The job J_5 is discarded as it cannot be executed within its deadline.
- Thus, the solution is the sequence of jobs (J_2, J_1, J_3), which are being executed within their deadline and gives maximum profit.
- Total profit of this sequence is **100 + 60 + 20 = 180**.

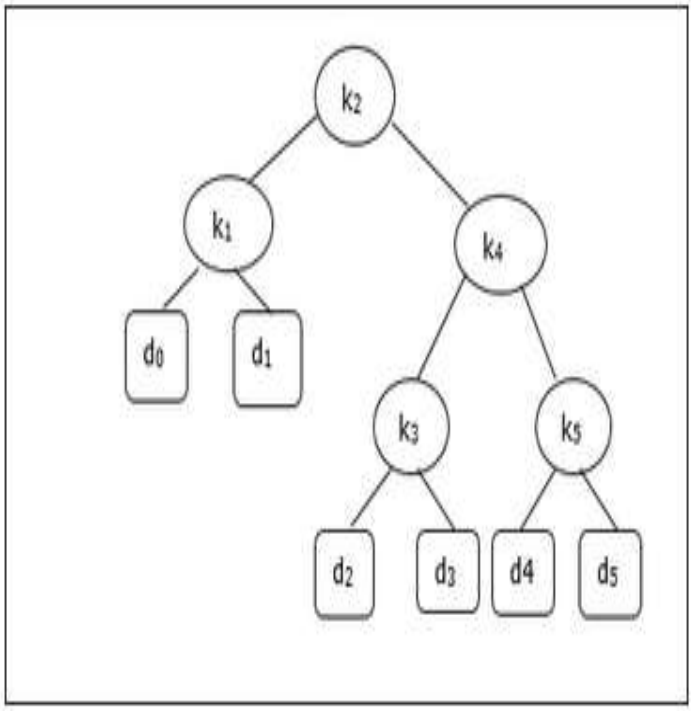
Optimal Cost Binary Search Tree

- A Binary Search Tree (BST) is a tree where the key values are stored in the internal nodes. The external nodes are null nodes. The keys are ordered lexicographically, i.e. for each internal node all the keys in the left sub-tree are less than the keys in the node, and all the keys in the right sub-tree are greater.
- Search time of an element in a BST is ***O(n)***, whereas in a Balanced-BST search time is ***O(log n)***. Again the search time can be improved in Optimal Cost Binary Search Tree, placing the most frequently used data in the root and closer to the root element, while placing the least frequently used data near leaves and in leaves.
- Here, the Optimal Binary Search Tree Algorithm is presented. First, we build a BST from a set of provided **n** number of distinct keys $< k_1, k_2, k_3, \ldots k_n >$. Here we assume, the probability of accessing a key K_i is p_i. Some dummy keys ($d_0, d_1, d_2, \ldots d_n$) are added as some searches may be performed for the values which are not present in the Key set ***K***. We assume, for each dummy key d_i probability of access is q_i.

- **Analysis**
  - The algorithm requires **O (n^3)** time, since three nested **for** loops are used. Each of these loops takes on at most **n** values.

Optimal Cost Binary Search Tree: Example

- Considering the following tree, the cost is 2.80, though this is not an optimal result.



Node	Depth	Probability	Contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			2.80

Optimal Cost Binary Search Tree: Example

- To get an optimal solution, using the algorithm discussed in this chapter, the following tables are generated.
- In the following tables, column index is **i** and row index is **j**.

e	1	2	3	4	5	6
5	2.75	2.00	1.30	0.90	0.50	0.10
4	1.75	1.20	0.60	0.30	0.05	
3	1.25	0.70	0.25	0.05		
2	0.90	0.40	0.05			
1	0.45	0.10				
0	0.05					

w	1	2	3	4	5	6
5	1.00	0.80	0.60	0.50	0.35	0.10
4	0.70	0.50	0.30	0.20	0.05	
3	0.55	0.35	0.15	0.05		
2	0.45	0.25	0.05			
1	0.30	0.10				
0	0.05					

Optimal Cost Binary Search Tree: Example

root	1	2	3	4	5
5	2	4	5	5	5
4	2	2	4	4	
3	2	2	3		
2	1	2			
1	1				

- From these tables, the optimal tree can be formed.

Travelling Salesman Problem

- **Problem Statement**
  - A traveler needs to visit all the cities from a list, where distances between all the cities are known and each city should be visited just once. What is the shortest possible route that he visits each city exactly once and returns to the origin city?
- **Solution**
  - Travelling salesman problem is the most notorious computational problem. We can use brute-force approach to evaluate every possible tour and select the best one. For **n** number of vertices in a graph, there are **(*n* - 1)!** number of possibilities.
  - Instead of brute-force using dynamic programming approach, the solution can be obtained in lesser time, though there is no polynomial time algorithm.
  - Let us consider a graph ***G = (V, E)***, where ***V*** is a set of cities and ***E*** is a set of weighted edges. An edge ***e(u, v)*** represents that vertices ***u*** and ***v*** are connected. Distance between vertex ***u*** and ***v*** is ***d(u, v)***, which should be non-negative.

Travelling Salesman Problem (Contd…)

- Suppose we have started at city ***1*** and after visiting some cities now we are in city ***j***. Hence, this is a partial tour. We certainly need to know ***j***, since this will determine which cities are most convenient to visit next. We also need to know all the cities visited so far, so that we don't repeat any of them. Hence, this is an appropriate sub-problem.
- For a subset of cities ***S ∈ {1, 2, 3, … , n}*** that includes ***1***, and ***j ∈ S***, let ***C(S, j)*** be the length of the shortest path visiting each node in **S** exactly once, starting at ***1*** and ending at ***j***.
- When $|S| > 1$, we define ***C(S, 1)*** = ∝ since the path cannot start and end at **1**.
- Now, let express **C(S, j)** in terms of smaller sub-problems. We need to start at ***1*** and end at **j**. We should select the next city in such a way that
- $C(S,j) = minC(S-\{j\},i)+d(i,j)$ *where* $i \in S$ *and* $i \neq j$
- $c(S,j) = minC(s-\{j\},i)+d(i,j)$ *where* $i \in S$ *and* $i \neq j$

Travelling Salesman Problem (Contd…)

- **Algorithm: Traveling-Salesman-Problem**

  C ({1}, 1) = 0

  for s = 2 to n do

  for all subsets S Є {1, 2, 3, … , n} of size s and containing 1 C (S, 1) = ∞

  for all j Є S and j ≠ 1 C (S, j) = min {C (S – {j}, i) + d(i, j) for i Є S and i ≠ j}

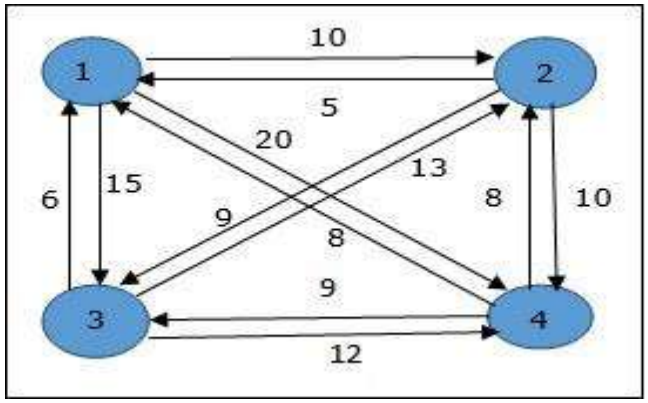
  Return minj C ({1, 2, 3, …, n}, j) + d(j, i)

- **Analysis**
  - There are at the most 2^*n*.*n*
  - sub-problems and each one takes linear time to solve. Therefore, the total running time is *O*(2^*n*.*n*^2).

Travelling Salesman Problem (Contd…)

- **Example**

  In the following example, we will illustrate the steps to solve the travelling salesman problem.



From the above graph, the following table is prepared.

	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

TSP Solution

- **S = Φ**
- $Cost(2,\Phi,1)=d(2,1)=5Cost(2,\Phi,1)=d(2,1)=5$
- $Cost(3,\Phi,1)=d(3,1)=6Cost(3,\Phi,1)=d(3,1)=6$
- $Cost(4,\Phi,1)=d(4,1)=8Cost(4,\Phi,1)=d(4,1)=8$
- **S = 1**
- $Cost(i,s)=min\{Cost(j,s–(j))+d[i,j]\}Cost(i,s)=min\{Cost(j,s)−(j))+d[i,j]\}$
- $Cost(2,\{3\},1)=d[2,3]+Cost(3,\Phi,1)=9+6=15cost(2,\{3\},1)=d[2,3]+cost(3,\Phi,1)=9+6=15$
- $Cost(2,\{4\},1)=d[2,4]+Cost(4,\Phi,1)=10+8=18cost(2,\{4\},1)=d[2,4]+cost(4,\Phi,1)=10+8=18$
- $Cost(3,\{2\},1)=d[3,2]+Cost(2,\Phi,1)=13+5=18cost(3,\{2\},1)=d[3,2]+cost(2,\Phi,1)=13+5=18$
- $Cost(3,\{4\},1)=d[3,4]+Cost(4,\Phi,1)=12+8=20cost(3,\{4\},1)=d[3,4]+cost(4,\Phi,1)=12+8=20$
- $Cost(4,\{3\},1)=d[4,3]+Cost(3,\Phi,1)=9+6=15cost(4,\{3\},1)=d[4,3]+cost(3,\Phi,1)=9+6=15$
- $Cost(4,\{2\},1)=d[4,2]+Cost(2,\Phi,1)=8+5=13cost(4,\{2\},1)=d[4,2]+cost(2,\Phi,1)=8+5=13$

TSP Solution

S = 2

$$Cost(2,\{3,4\},1) = \begin{cases} d[2,3] + Cost(3,\{4\},1) = 9 + 20 = 29 \\ d[2,4] + Cost(4,\{3\},1) = 10 + 15 = 25 = 25 Cost(2,\{3,4\},1) \\ \{d[2,3] + cost(3,\{4\},1) = 9 + 20 = 29 d[2,4] + Cost(4,\{3\},1) = 10 + 15 = 25 \\ = 25 \end{cases}$$

$$Cost(3,\{2,4\},1)$$

$$= \begin{cases} d[3,2] + Cost(2,\{4\},1) = 13 + 18 = 31 \\ d[3,4] + Cost(4,\{2\},1) = 12 + 13 = 25 = 25 Cost(3,\{2,4\},1) \\ \{d[3,2] + cost(2,\{4\},1) = 13 + 18 = 31 d[3,4] + Cost(4,\{2\},1) = 12 + 13 = 25 \end{cases} = 25$$

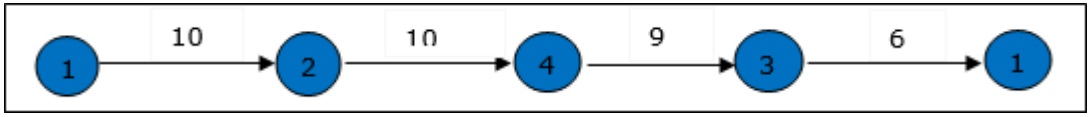
$$Cost(4,\{2,3\},1) = \begin{cases} d[4,2] + Cost(2,\{3\},1) = 8 + 15 = 23 \\ d[4,3] + Cost(3,\{2\},1) = 9 + 18 = 27 = 23 Cost(4,\{2,3\},1) \\ \{d[4,2] + cost(2,\{3\},1) = 8 + 15 = 23 d[4,3] + Cost(3,\{2\},1) = 9 + 18 = 27 \\ = 23 \end{cases}$$

S = 3

$$Cost(1,\{2,3,4\},1) = \begin{cases} d[1,2] + Cost(2,\{3,4\},1) = 10 + 25 = 35 \\ d[1,3] + Cost(3,\{2,4\},1) = 15 + 25 = 40 \\ d[1,4] + Cost(4,\{2,3\},1) = 20 + 23 = 43 = 35 cost(1,\{2,3,4\}),1) \\ d[1,2] + cost(2,\{3,4\},1) = 10 + 25 = 35 \\ d[1,3] + cost(3,\{2,4\},1) = 15 + 25 = 40 \\ d[1,4] + cost(4,\{2,3\},1) = 20 + 23 = 43 = 35 \end{cases}$$

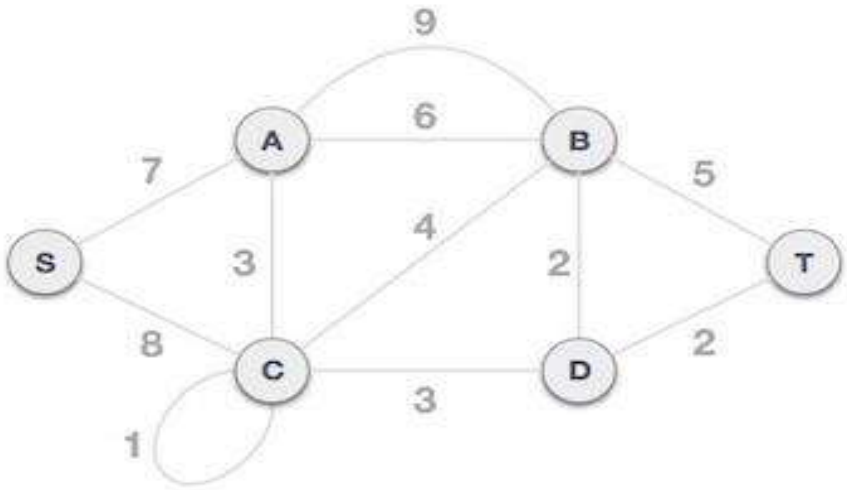
TSP Solution

- The minimum cost path is 35.
- Start from cost **{1, {2, 3, 4}, 1}**, we get the minimum value for **d [1, 2]**. When **s = 3**, select the path from 1 to 2 (cost is 10) then go backwards. When **s = 2**, we get the minimum value for **d [4, 2]**. Select the path from 2 to 4 (cost is 10) then go backwards.
- When **s = 1**, we get the minimum value for **d [4, 3]**. Selecting path 4 to 3 (cost is 9), then we shall go to then go to **s = Φ** step. We get the minimum value for **d [3, 1]** (cost is 6).



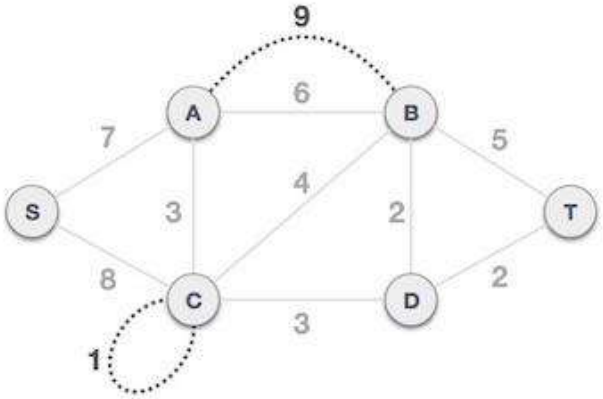
Kruskal’s Spanning Tree

- Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.
- To understand Kruskal's algorithm let us consider the following example −

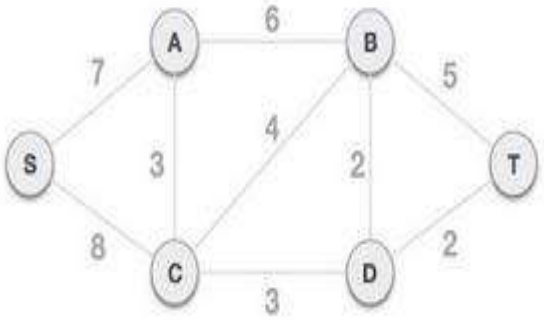


Kruskal's Spanning Tree (Contd…)

- **Step 1 - Remove all loops and Parallel Edges**
- Remove all loops and parallel edges from the given graph.



- In case of parallel edges, keep the one which has the least cost associated and remove all others.

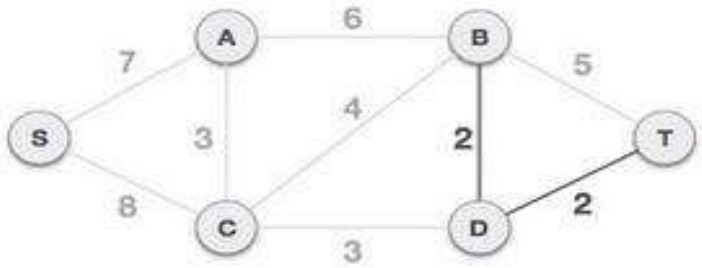


Kruskal's Spanning Tree (Contd…)

- **Step 2 - Arrange all edges in their increasing order of weight**
- The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).

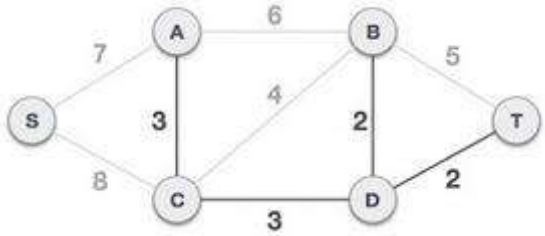
B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

- **Step 3 - Add the edge which has the least weightage**
- Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.

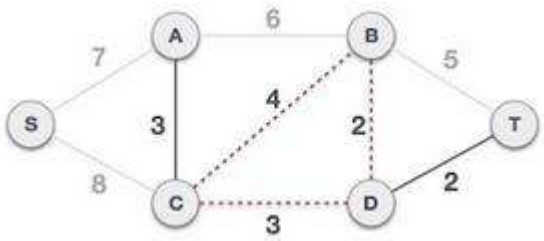


Kruskal's Spanning Tree (Contd…)

- The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.
- Next cost is 3, and associated edges are A,C and C,D. We add them again −

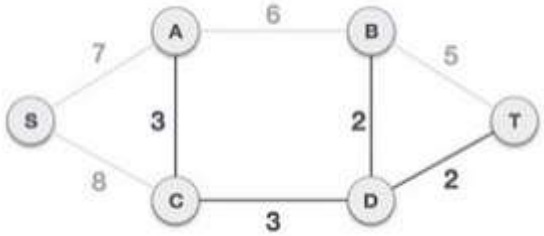


- Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. −

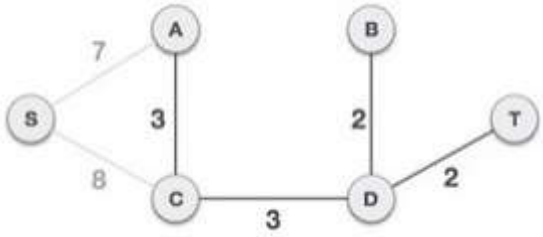


Kruskal's Spanning Tree (Contd…)

- We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



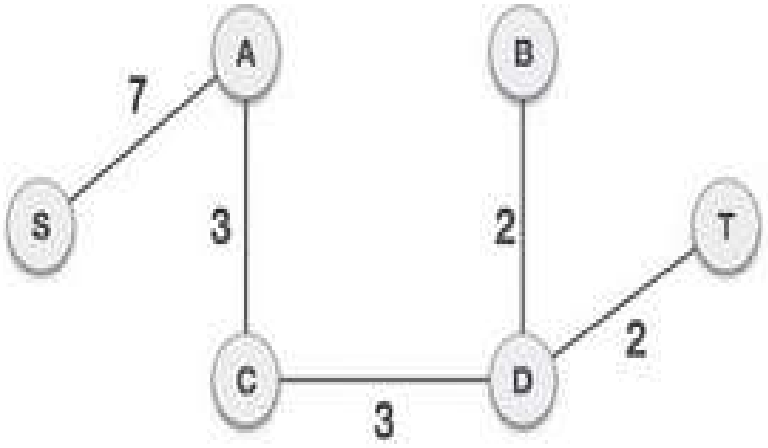
- We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.



- Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.

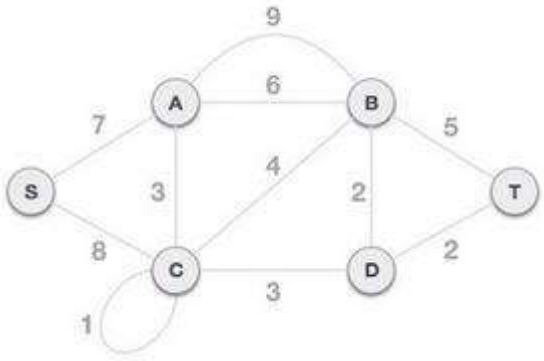
Kruskal's Spanning Tree (Contd…)

- By adding edge S,A we have included all the nodes of the graph and we now have minimum cost spanning tree



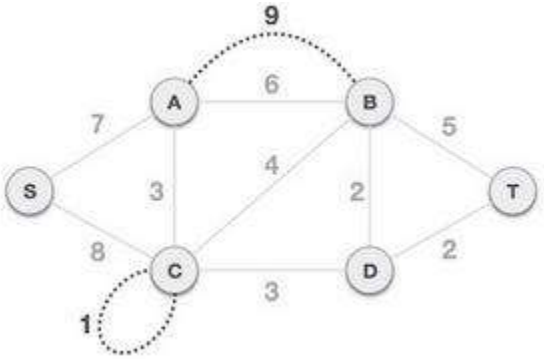
Prim’s Spanning Tree

- Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses the greedy approach. Prim's algorithm shares a similarity with the **shortest path first** algorithms.
- Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.
- To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example −

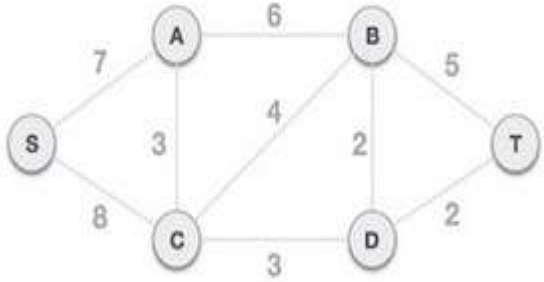


Prim's Spanning Tree (Contd…)

- **Step 1 - Remove all loops and parallel edges**

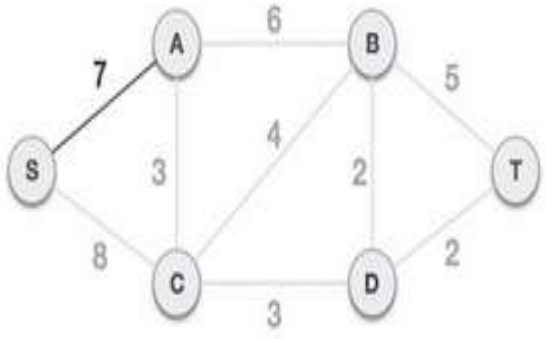


- Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.



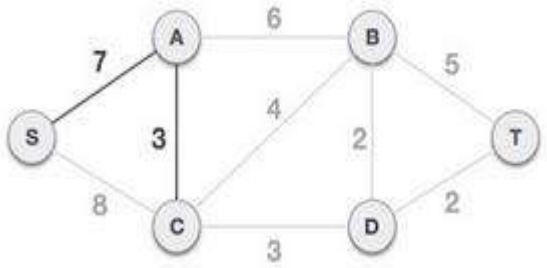
Prim’s Spanning Tree (Contd…)

- **Step 2 - Choose any arbitrary node as root node**
- In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.
- **Step 3 - Check outgoing edges and select the one with less cost**
- After choosing the root node **S**, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.

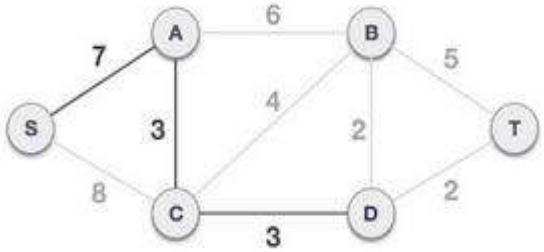


Prim’s Spanning Tree (Contd…)

- Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.

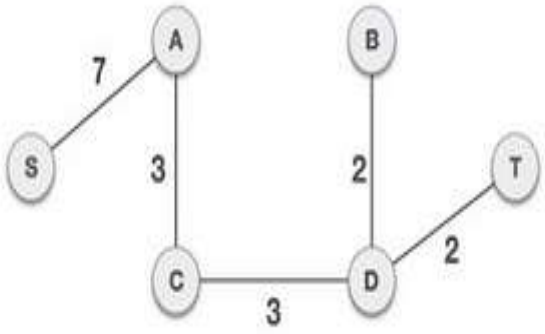


- After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



Prim's Spanning Tree (Contd…)

- After adding node **D** to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.



- We may find that the output spanning tree of the same graph using two different algorithms is same.

Shortest Path

- **Dijkstra's Algorithm**
  - Dijkstra's algorithm solves the single-source shortest-paths problem on a directed weighted graph *G = (V, E)*, where all the edges are non-negative (i.e., *w(u, v)* ≥ 0 for each edge *(u, v) ∈ E*).
  - In the following algorithm, we will use one function ***Extract-Min()***, which extracts the node with the smallest key.

```
Algorithm: Dijkstra's-Algorithm (G, w, s)
for each vertex v ∈ G.V
   v.d := ∞
   v.∏ := NIL
s.d := 0
S := Φ
Q := G.V
while Q ≠ Φ
   u := Extract-Min (Q)
   S := S U {u}
   for each vertex v ∈ G.adj[u]
      if v.d > u.d + w(u, v)
         v.d := u.d + w(u, v)
         v.∏ := u
```

Shortest Path: Dijkstra's Algorithm

- **Analysis**
- The complexity of this algorithm is fully dependent on the implementation of Extract-Min function. If extract min function is implemented using linear search, the complexity of this algorithm is $O(V^2 + E)$.
- In this algorithm, if we use min-heap on which ***Extract-Min()*** function works to return the node from ***Q*** with the smallest key, the complexity of this algorithm can be reduced further.
- **Example**
- Let us consider vertex ***1*** and ***9*** as the start and destination vertex respectively. Initially, all the vertices except the start vertex are marked by ∞ and the start vertex is marked by ***0***.

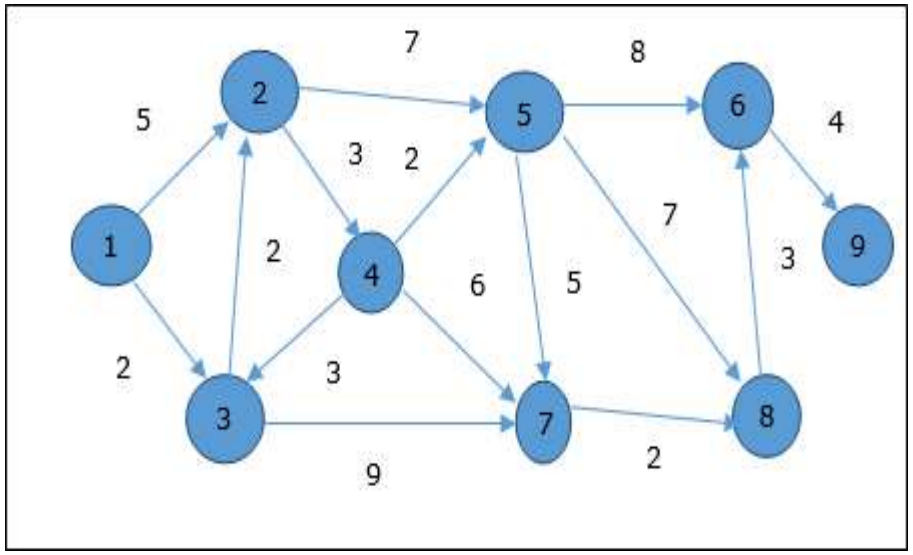
Shortest Path: Dijkstra's Algorithm

- **Example**
  - Let us consider vertex ***1*** and ***9*** as the start and destination vertex respectively. Initially, all the vertices except the start vertex are marked by ∞ and the start vertex is marked by ***0***.

Vertex	Initial	Step1 V_1	Step2 V_3	Step3 V_2	Step4 V_4	Step5 V_5	Step6 V_7	Step7 V_8	Step8 V_6
1	0	0	0	0	0	0	0	0	0
2	∞	5	4	4	4	4	4	4	4
3	∞	2	2	2	2	2	2	2	2
4	∞	∞	∞	7	7	7	7	7	7
5	∞	∞	∞	11	9	9	9	9	9
6	∞	∞	∞	∞	∞	17	17	16	16
7	∞	∞	11	11	11	11	11	11	11
8	∞	∞	∞	∞	∞	16	13	13	13
9	∞	∞	∞	∞	∞	∞	∞	∞	20

Shortest Path: Dijkstra's Algorithm

- Hence, the minimum distance of vertex ***9*** from vertex ***1*** is ***20***. And the path is 1→ 3→ 7→ 8→ 6→ 9
- This path is determined based on predecessor information.



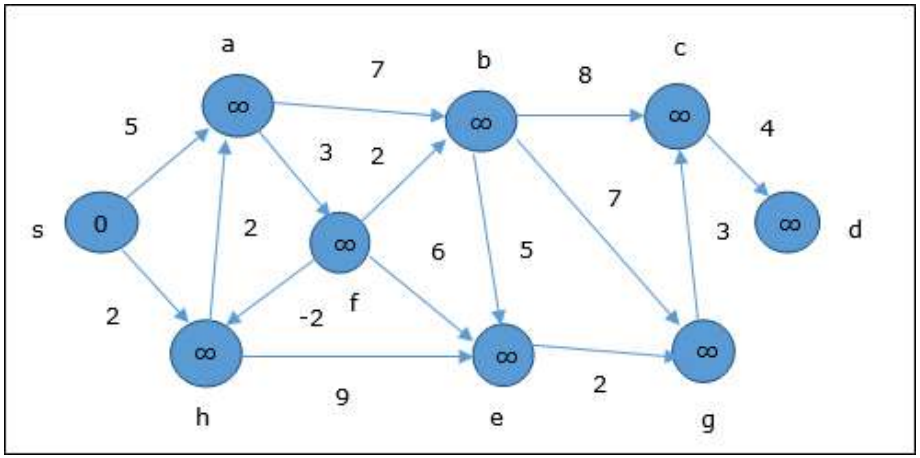
Shortest Path: Bellman Ford Algorithm

- This algorithm solves the single source shortest path problem of a directed graph ***G = (V, E)*** in which the edge weights may be negative. Moreover, this algorithm can be applied to find the shortest path, if there does not exist any negative weighted cycle.

```
Algorithm: Bellman-Ford-Algorithm (G, w, s)
for each vertex v ∈ G.V
   v.d := ∞
   v.∏ := NIL
s.d := 0
for i = 1 to |G.V| - 1
   for each edge (u, v) ∈ G.E
      if v.d > u.d + w(u, v)
         v.d := u.d +w(u, v)
         v.∏ := u
for each edge (u, v) ∈ G.E
   if v.d > u.d + w(u, v)
      return FALSE
return TRUE
```

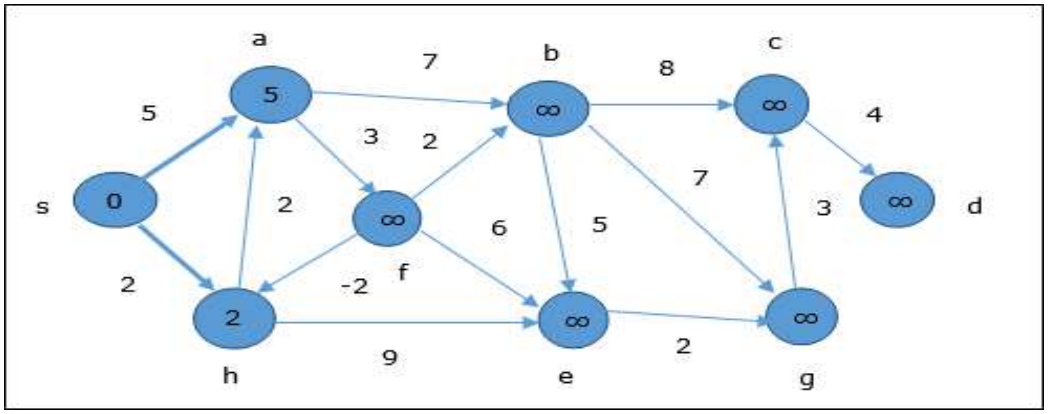
Shortest Path: Bellman Ford Algorithm

- **Analysis**
  - The first **for** loop is used for initialization, which runs in ***O(V)*** times. The next **for** loop runs |***V - 1***| passes over the edges, which takes ***O(E)*** times.
  - Hence, Bellman-Ford algorithm runs in ***O(V, E)*** time.
- **Example**
  - The following example shows how Bellman-Ford algorithm works step by step. This graph has a negative edge but does not have any negative cycle, hence the problem can be solved using this technique.
  - At the time of initialization, all the vertices except the source are marked by ∞ and the source is marked by **0**.

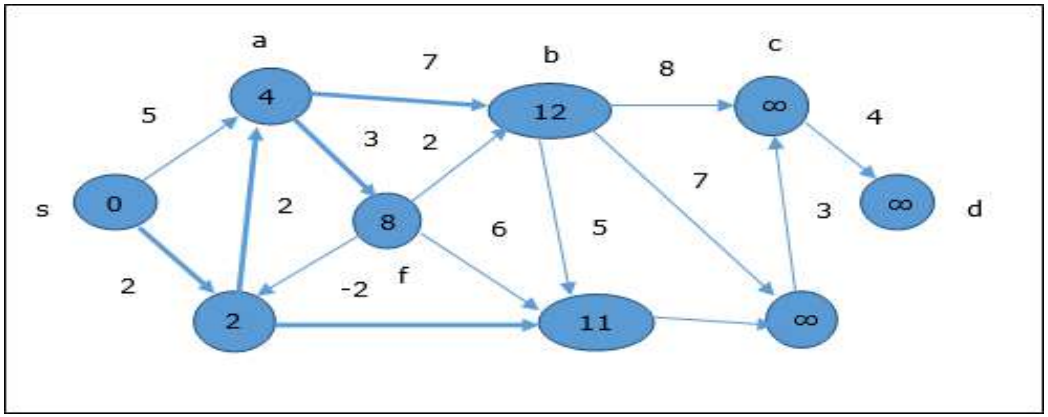


Shortest Path: Bellman Ford Algorithm

- In the first step, all the vertices which are reachable from the source are updated by minimum cost. Hence, vertices ***a*** and ***h*** are updated

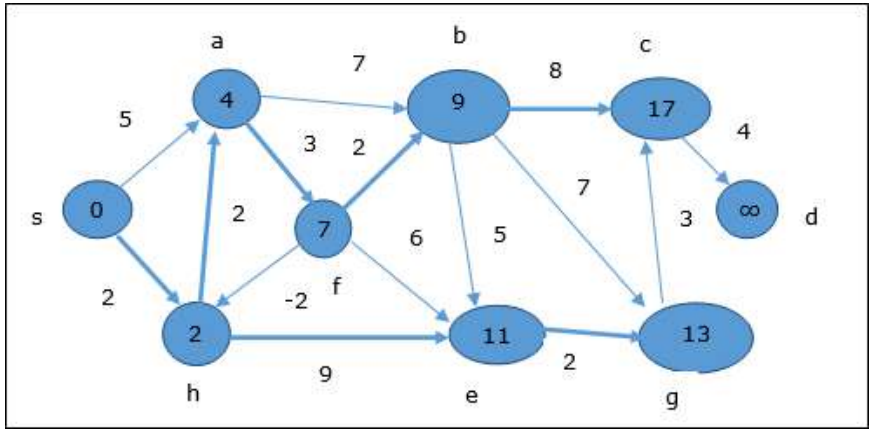


- In the next step, vertices ***a, b, f*** and ***e*** are updated

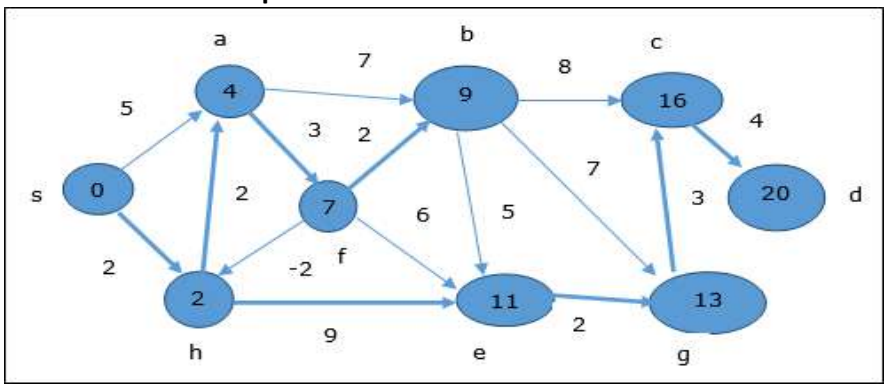


Shortest Path: Bellman Ford Algorithm

- Following the same logic, in this step vertices ***b, f, c*** and ***g*** are updated.



- Here, vertices ***c*** and ***d*** are updated.



- Hence, the minimum distance between vertex **s** and vertex **d** is **20**.
- Based on the predecessor information, the path is s→ h→ e→ g→ c→ d

Dynamic Programming

- Dynamic Programming is also used in optimization problems. Like divide-and-conquer method, Dynamic Programming solves problems by combining the solutions of sub problems.
- Moreover, Dynamic Programming algorithm solves each sub-problem just once and then saves its answer in a table, thereby avoiding the work of re-computing the answer every time.
- Two main properties of a problem suggest that the given problem can be solved using Dynamic Programming. These properties are **overlapping sub-problems and optimal substructure**.

- **Overlapping Sub-Problems**
  - Similar to Divide-and-Conquer approach, Dynamic Programming also combines solutions to sub-problems. It is mainly used where the solution of one sub-problem is needed repeatedly. The computed solutions are stored in a table, so that these don't have to be re-computed. Hence, this technique is needed where overlapping sub-problem exists.
  - For example, Binary Search does not have overlapping sub-problem. Whereas recursive program of Fibonacci numbers have many overlapping sub-problems.

Dynamic Programming (Contd…)

- **Optimal Sub-Structure**
  - A given problem has Optimal Substructure Property, if the optimal solution of the given problem can be obtained using optimal solutions of its sub-problems.
  - For example, the Shortest Path problem has the following optimal substructure property –
  - If a node **x** lies in the shortest path from a source node **u** to destination node **v**, then the shortest path from **u** to **v** is the combination of the shortest path from **u** to **x**, and the shortest path from **x** to **v**.
  - The standard All Pair Shortest Path algorithms like Floyd-Warshall and Bellman-Ford are typical examples of Dynamic Programming.

Dynamic Programming: Steps and Application Areas

- **Steps of Dynamic Programming Approach**
  - Dynamic Programming algorithm is designed using the following four steps −
  - Characterize the structure of an optimal solution.
  - Recursively define the value of an optimal solution.
  - Compute the value of an optimal solution, typically in a bottom-up fashion.
  - Construct an optimal solution from the computed information.

- **Applications of Dynamic Programming Approach**
  - Matrix Chain Multiplication
  - Longest Common Subsequence
  - Travelling Salesman Problem

Knapsack: Dynamic Approach

- 0-1 Knapsack cannot be solved by Greedy approach. Greedy approach does not ensure an optimal solution. In many instances, Greedy approach may give an optimal solution.
- The following examples will establish our statement.
- **Example-1**
  - Let us consider that the capacity of the knapsack is W = 25 and the items are as shown in the following table.
  - Item A B C D Profit 24 18 18 10 Weight 24 10 10 7 Without considering the profit per unit weight (p_i/w_i), if we apply Greedy approach to solve this problem, first item ***A*** will be selected as it will contribute max$_i$mum profit among all the elements.
  - After selecting item ***A***, no more item will be selected. Hence, for this given set of items total profit is ***24***. Whereas, the optimal solution can be achieved by selecting items, ***B*** and C, where the total profit is 18 + 18 = 36

Item	A	B	C	D
Profit	24	18	18	10
Weight	24	10	10	7

Knapsack: Dynamic Approach (Contd…)

- **Example-2**
- Instead of selecting the items based on the overall benefit, in this example the items are selected based on ratio p_i/w_i. Let us consider that the capacity of the knapsack is $W = 60$ and the items are as shown in the following table.
- Item A B C Price 100 280 120 Weight 10 40 20 Ratio 10 7 6 Using the Greedy approach, first item ***A*** is selected. Then, the next item ***B*** is chosen. Hence, the total profit is **100 + 280 = 380**. However, the optimal solution of this instance can be achieved by selecting items, ***B*** and ***C***, where the total profit is **280 + 120 = 400**.

Item	A	B	C
Price	100	280	120
Weight	10	40	20
Ratio	10	7	6

- Hence, it can be concluded that Greedy approach may not give an optimal solution.
- To solve 0-1 Knapsack, Dynamic Programming approach is required.

Knapsack: Dynamic Approach (Contd…)

- **Problem Statement**
  - A thief is robbing a store and can carry a max$_i$mal weight of ***W*** into his knapsack. There are ***n*** items and weight of **ith** item is w_i and the profit of selecting this item is p_i. What items should the thief take?
- **Dynamic-Programming Approach**
  - Let ***i*** be the highest-numbered item in an optimal solution **S** for **W** dollars. Then $S' = S - \{i\}$ is an optimal solution for $W - w_i$ dollars and the value to the solution ***S*** is V_i plus the value of the sub-problem.
  - We can express this fact in the following formula: define **c[i, w]** to be the solution for items **1,2, … , i** and the max$_i$mum weight **w**.
  - The algorithm takes the following inputs
    - The max$_i$mum weight **W**
    - The number of items **n**
    - The two sequences $\mathbf{v} = \langle v_1, v_2, \ldots, v_n \rangle$ and $\mathbf{w} = \langle w_1, w_2, \ldots, w_n \rangle$

Knapsack: Dynamic Approach (Contd…)

```
Dynamic-0-1-knapsack (v, w, n, W)
for w = 0 to W do
   c[0, w] = 0
for i = 1 to n do
   c[i, 0] = 0
   for w = 1 to W do
      if wi ≤ w then
         if vi + c[i-1, w-wi] then
            c[i, w] = vi + c[i-1, w-wi]
         else c[i, w] = c[i-1, w]
      else
         c[i, w] = c[i-1, w]
```

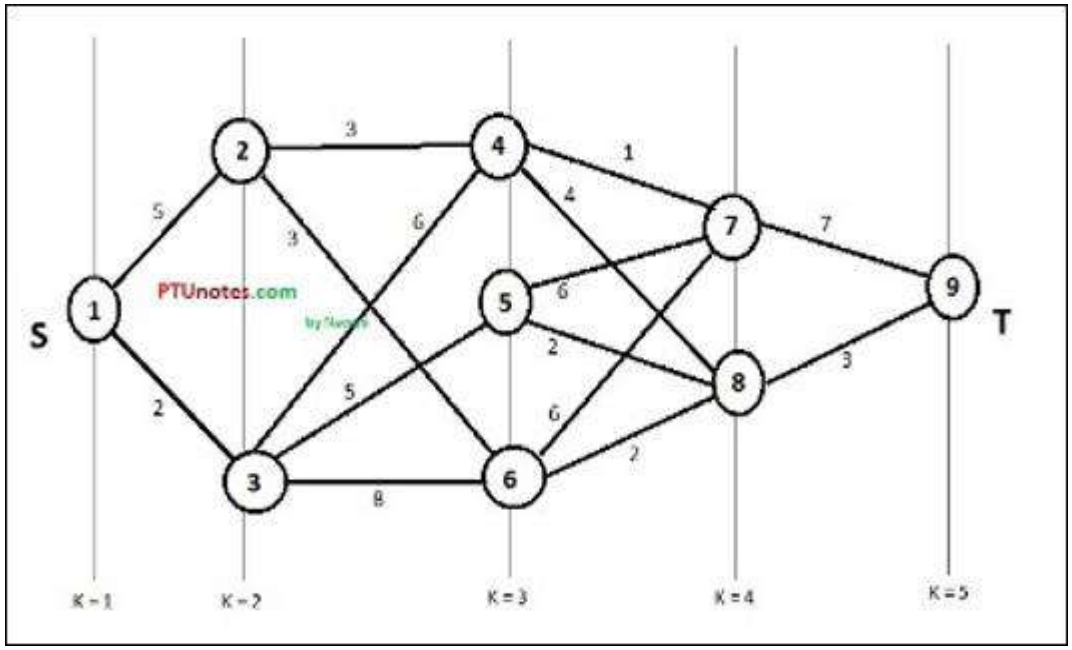
- The set of items to take can be deduced from the table, starting at **c[n, w]** and tracing backwards where the optimal values came from.
- If *c[i, w] = c[i-1, w]*, then item ***i*** is not part of the solution, and we continue tracing with **c[i-1, w]**. Otherwise, item ***i*** is part of the solution, and we continue tracing with **c[i-1, w-W]**.
- **Analysis**
  - This algorithm takes θ(*n*, *w*) times as table *c* has (*n* + 1).(*w* + 1) entries, where each entry requires θ(1) time to compute.

Multistage Graph: Dynamic Approach

- A multistage graph **G = (V, E)** is a directed graph where vertices are partitioned into **k** (where ***k* > 1**) number of disjoint subsets $S = \{s_1, s_2, \ldots, s_k\}$ such that edge *(u, v)* is in E, then $u \in s_i$ and $v \in s_{1+1}$ for some subsets in the partition and $|s_1| = |s_k| = 1$.
- The vertex $s \in s_1$ is called the **source** and the vertex $t \in s_k$ is called **sink**.
- ***G*** is usually assumed to be a weighted graph. In this graph, cost of an edge *(i, j)* is represented by *c(i, j)*. Hence, the cost of path from source ***s*** to sink ***t*** is the sum of costs of each edges in this path.
- The multistage graph problem is finding the path with minimum cost from source ***s*** to sink ***t***.

Multistage Graph Example

- Consider the following example to understand the concept of multistage graph.
- According to the formula, we have to calculate the cost **(i, j)** using the following steps



Multistage Graph Example (Contd…)

- **Step-1: Cost (K-2, j)**
  - In this step, three nodes (node 4, 5. 6) are selected as **j**. Hence, we have three options to choose the minimum cost at this step.
  - *Cost(3, 4) = min {c(4, 7) + Cost(7, 9),c(4, 8) + Cost(8, 9)} = 7*
  - *Cost(3, 5) = min {c(5, 7) + Cost(7, 9),c(5, 8) + Cost(8, 9)} = 5*
  - *Cost(3, 6) = min {c(6, 7) + Cost(7, 9),c(6, 8) + Cost(8, 9)} = 5*
- **Step-2: Cost (K-3, j)**
  - Two nodes are selected as j because at stage k - 3 = 2 there are two nodes, 2 and 3. So, the value i = 2 and j = 2 and 3.
  - *Cost(2, 2) = min {c(2, 4) + Cost(4, 8) + Cost(8, 9),c(2, 6) +*
  - *Cost(6, 8) + Cost(8, 9)} = 8*
  - *Cost(2, 3) = {c(3, 4) + Cost(4, 8) + Cost(8, 9), c(3, 5) + Cost(5, 8)+ Cost(8, 9), c(3, 6) + Cost(6, 8) + Cost(8, 9)} = 10*

Multistage Graph Example (Contd…)

- **Step-3: Cost (K-4, j)**
  - *Cost (1, 1) = {c(1, 2) + Cost(2, 6) + Cost(6, 8) + Cost(8, 9), c(1, 3) + Cost(3, 5) + Cost(5, 8) + Cost(8, 9))} = 12*
  - *c(1, 3) + Cost(3, 6) + Cost(6, 8 + Cost(8, 9))} = 13*

- Hence, the path having the minimum cost is **1→ 3→ 5→ 8→ 9**.

All pair Shortest Path algorithm: Floyd Warshall Algorithm

- **Floyd-Warshall Algorithm**
  - Let the vertices of G be V = {1, 2.....n} and consider a subset {1, 2.....k} of vertices for some k. For any pair of vertices i, j ∈ V, considered all paths from i to j whose intermediate vertices are all drawn from {1, 2.....k}, and let p be a minimum weight path from amongst them. The Floyd-Warshall algorithm exploits a link between path p and shortest paths from i to j with all intermediate vertices in the set {1, 2.....k-1}. The link depends on whether or not k is an intermediate vertex of path p.
  - If k is not an intermediate vertex of path p, then all intermediate vertices of path p are in the set {1, 2.....k-1}. Thus, the shortest path from vertex i to vertex j with all intermediate vertices in the set {1, 2.....k-1} is also the shortest path i to j with all intermediate vertices in the set {1, 2.....k}.
  - If k is an intermediate vertex of path p, then we break p down into i → k → j.
  - Let $d_{ij}^{(k)}$ be the weight of the shortest path from vertex i to vertex j with all intermediate vertices in the set {1, 2.....k}.
  - A recursive definition is given by

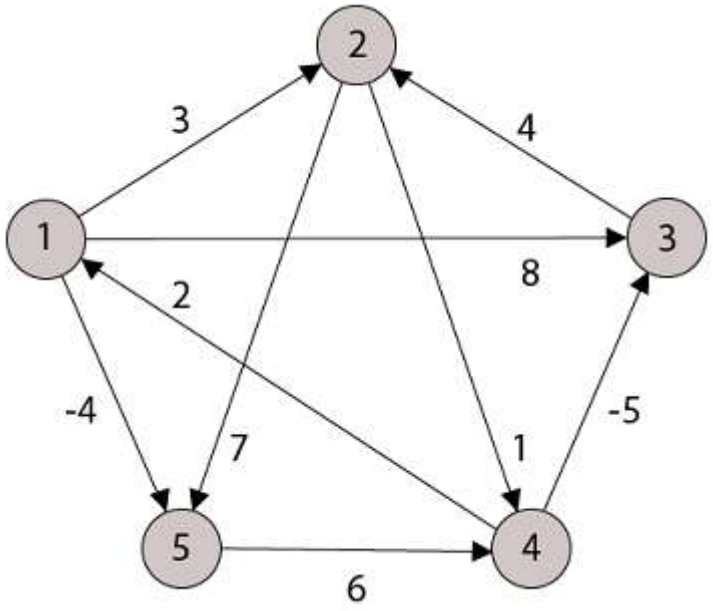
$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k=0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 \end{cases}$$

All pair Shortest Path algorithm: Floyd Warshall Algorithm

```
FLOYD - WARSHALL (W)
 1. n ← rows [W].
 2. D^0 ← W
 3. for k ← 1 to n
 4. do for i ← 1 to n
 5. do for j ← 1 to n
 6. do d_ij^(k) ← min (d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1) )
 7. return D^(n)
```

- The strategy adopted by the Floyd-Warshall algorithm is **Dynamic Programming**. The running time of the Floyd-Warshall algorithm is determined by the triply nested for loops of lines 3-6. Each execution of line 6 takes O (1) time. The algorithm thus runs in time $\theta(n^3)$

Example: Floyd Warshall Algorithm



- **Solution:**

$$d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} \text{ if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} \text{ if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

Example: Floyd Warshall Algorithm

- **Step (i)** When k = 0

$$D^{(0)} = \begin{matrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & -5 & \infty \\ 2 & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{matrix} \qquad \pi^{(0)} = \begin{matrix} NIL & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & 3 & NIL \\ 4 & NIL & NIL & NIL & NIL \\ NIL & NIL & NIL & 5 & NIL \end{matrix}$$

- **Step (ii)** When k =1

$$d_{ij}^{(k)} = \min (d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$$

$$d_{14}^{(1)} = \min (d_{14}^{(0)}, d_{11}^{(0)} + d_{14}^{(0)})$$

$$d_{14}^{(1)} = \min (\infty , 0 + \infty) = \infty$$

$$d_{15}^{(1)} = \min (d_{15}^{(0)}, d_{11}^{(0)} + d_{15}^{(0)})$$

$$d_{15}^{(1)} = \min (-4, 0 + -4) = -4$$

Example: Floyd Warshall Algorithm

$d_{21}^{(1)} = \min (d_{21}^{(0)}, d_{21}^{(0)} + d_{11}^{(0)})$

$d_{21}^{(1)} = \min (\infty, \infty + 0) = \infty$

$d_{23}^{(1)} = \min (d_{23}^{(0)}, d_{21}^{(0)} + d_{13}^{(0)})$

$d_{23}^{(1)} = \min ((\infty, \infty + 8) = \infty$

$d_{31}^{(1)} = \min (d_{31}^{(0)}, d_{31}^{(0)} + d_{11}^{(0)})$

$d_{31}^{(1)} = \min (\infty, \infty + 0) = \infty$

$d_{35}^{(1)} = \min (d_{35}^{(0)}, d_{31}^{(0)} + d_{15}^{(0)})$

$d_{43}^{(1)} = \min (d_{43}^{(0)}, d_{41}^{(0)} + d_{13}^{(0)})$

$d_{43}^{(1)} = \min (\infty, 2 + 8) = 10$

Example: Floyd Warshall Algorithm

$d_{51}^{(1)} = \min (d_{51}^{(0)}, d_{51}^{(0)} + d_{11}^{(0)})$

$d_{51}^{(1)} = \min (\infty, \infty + 0) = \infty$

$$D_{ij}^{(1)} = \begin{matrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & -5 & \infty \\ 2 & 5 & 10 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{matrix} \qquad \pi^{(1)} = \begin{matrix} NIL & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & 3 & NIL \\ 4 & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{matrix}$$

Step (iii) When k = 2

$$d_{ij}^{(k)} = \min (d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$$

$$d_{14}^{(2)} = \min (d_{14}^{(1)}, d_{12}^{(1)} + d_{24}^{(1)})$$

$$d_{14}^{(2)} = \min (\infty, 3 + 1) = 4$$

$$d_{21}^{(2)} = \min (d_{21}^{(1)}, d_{22}^{(1)} + d_{21}^{(1)})$$

$$d_{21}^{(2)} = \min (\infty, 0 + \infty) = \infty$$

Example: Floyd Warshall Algorithm

$$d_{34}^{(2)} = \min (d_{34}^{(1)}, d_{32}^{(1)} + d_{24}^{(1)})$$

$$d_{34}^{(2)} = \min (-5, 4 + 1) = -5$$

$$d_{35}^{(2)} = \min (d_{35}^{(1)}, d_{32}^{(1)} + d_{25}^{(1)})$$

$$d_{35}^{(2)} = \min (\infty, 4 + 7) = 11$$

$$d_{43}^{(2)} = \min (d_{43}^{(1)}, d_{42}^{(1)} + d_{23}^{(1)})$$

$$d_{43}^{(2)} = \min (10, 5 + \infty) = 10$$

$$D_{ij}^{(2)} = \begin{matrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & -5 & 11 \\ 2 & 5 & 10 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{matrix} \qquad \pi^{(2)} = \begin{matrix} NIL & 1 & 1 & 2 & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & 3 & 2 \\ 4 & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{matrix}$$

Example: Floyd Warshall Algorithm

Step (iv) When k = 3

$$d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$$

$$d_{14}^{(3)} = \min\left(d_{14}^{(2)}, d_{13}^{(2)} + d_{34}^{(2)}\right)$$

$$d_{14}^{(3)} = \min(4, 8 + (-5)) = 3$$

$$D_{ij}^{(3)} = \begin{matrix} 0 & 3 & 8 & 3 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & -5 & 11 \\ 2 & 5 & 10 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{matrix} \qquad \pi^{(3)} = \begin{matrix} NIL & 1 & 1 & 3 & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & 3 & 2 \\ 4 & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{matrix}$$

Step (v) When k = 4

$$d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$$

$$d_{21}^{(4)} = \min\left(d_{21}^{(3)}, d_{24}^{(3)} + d_{41}^{(3)}\right)$$

$$d_{21}^{(4)} = \min(\infty, 1 + 2) = 3$$

Example: Floyd Warshall Algorithm

$$d_{23}^{(4)} = \min (d_{23}^{(3)}, d_{24}^{(3)} + d_{43}^{(3)})$$

$$d_{23}^{(4)} = \min (\infty, 1 + 10) = 11$$

$$d_{25}^{(4)} = \min (d_{25}^{(3)}, d_{24}^{(3)} + d_{45}^{(3)})$$

$$d_{25}^{(4)} = \min (7, 1 + (-2)) = -1$$

$$d_{31}^{(4)} = \min (d_{31}^{(3)}, d_{34}^{(3)} + d_{41}^{(3)})$$

$$d_{31}^{(4)} = \min (\infty, -5 + 2) = -3$$

$$d_{32}^{(4)} = \min (d_{32}^{(3)}, d_{34}^{(3)} + d_{42}^{(3)})$$

$$d_{32}^{(4)} = \min (4, -5 + 5) = 0$$

$$d_{51}^{(4)} = \min (d_{51}^{(3)}, d_{54}^{(3)} + d_{41}^{(3)})$$

$$d_{51}^{(4)} = \min (\infty, 6 + 2) = 8$$

Example: Floyd Warshall Algorithm

$$d_{52}^{(4)} = \min (d_{52}^{(3)}, d_{54}^{(3)} + d_{42}^{(3)})$$

$$d_{52}^{(4)} = \min (\infty, 6 + 5) = 11$$

$$d_{53}^{(4)} = \min (d_{53}^{(3)}, d_{54}^{(3)} + d_{43}^{(3)})$$

$$d_{53}^{(4)} = \min (\infty, 6 + 10) = 16$$

$$D_{ij}^{(4)} = \begin{matrix} 0 & 3 & 8 & 3 & -4 \\ 3 & 0 & 11 & 1 & -1 \\ -3 & 0 & 0 & -5 & -7 \\ 2 & 5 & 10 & 0 & -2 \\ 8 & 11 & 16 & 6 & 0 \end{matrix} \qquad \pi^{(4)} = \begin{matrix} NIL & 1 & 1 & 3 & 1 \\ 4 & NIL & 4 & 2 & 2 \\ 4 & 4 & NIL & 3 & 4 \\ 4 & 1 & 1 & NIL & 1 \\ 4 & 4 & 4 & 5 & NIL \end{matrix}$$

Step (vi) When k = 5

$$d_{ij}^{(k)} = \min (d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$$

$$d_{25}^{(5)} = \min (d_{25}^{(4)}, d_{25}^{(4)} + d_{55}^{(3)})$$

$$d_{25}^{(5)} = \min (-1, -1 + 0) = -1$$

Example: Floyd Warshall Algorithm

$d_{23}^{(5)} = \min (d_{23}^{(4)}, d_{25}^{(4)} + d_{53}^{(3)})$

$d_{23}^{(5)} = \min (11, -1 + 16) = 11$

$d_{35}^{(5)} = \min (d_{35}^{(4)}, d_{35}^{(4)} + d_{55}^{(3)})$

$d_{35}^{(5)} = \min (-7, -7 + 0) = -7$

$D_{ij}^{(5)} =$

0	3	8	3	-4
3	0	11	1	-1
-3	0	0	-5	-7
2	5	10	0	-2
8	11	16	6	0

$\pi^{(5)} =$

NIL	1	1	5	1
4	NIL	4	2	4
4	4	NIL	3	4
4	1	1	NIL	1
4	4	4	5	NIL

Resource allocation Problem

- **The Simple Model**
  - You are given X units of a resource and told that this resource must be distributed among N activities.
  - You are also given N data tables ri (x) (for i = 1; ;N and x = 0; 1; ;X) representing the return realized from an allocation of x units of resource to activity i.
  - Further, assume that ri (x) is a non decreasing function of x.
  - The problem is to allocate all of the X units of resource to the activities so as to maximize the total return, i.e. to choose N nonnegative integers xi , i = 1; ;N, that maximize

$$\sum_{i=1}^{N} r_i(x_i)$$

  - subject to the constraint

$$\sum_{i=1}^{N} x_i = X.$$

Resource allocation Problem

- (i) OPTIMAL VALUE FUNCTION:
- Sk (x) = the maximum return obtainable from activities k through N, given x units of resource remaining to be allocated
- (ii) RECURRENCE RELATION:

$$S_k(x) = \max_{j=0,1,\ldots,x} \{r_k(j) + S_{k+1}(x-j)\}, \quad 0 \le x \le X, 1 \le k \le N.$$

- (iii) BOUNDARY CONDITIONS:

$$S_N(x) = r_N(x), \text{ for } x = 1, 2, \ldots, X \text{ and } S_k(0) = 0, \text{ for } k = 1, 2, \ldots, N.$$

- ANSWER TO BE SOUGHT: S1(X).

Resource allocation Problem: Numerical

- Suppose we have four doctors that are to be sent to three different hospitals to help inject the H5N1 vaccine to patients there. Table 1 (left) shows that number of patients the doctors can serve per hour. How should we allocate the doctors so that we can serve the maximum number of patients?

	Hospital		
No. of Doctors	1	2	3
0	0	0	0
1	4	2	5
2	6	4	7
3	9	7	8
4	10	11	9

x	$S_3(x)$
0	0
1	5
2	7
3	8
4	9

- **Table: Number of patients served (left) and boundary conditions for** Sk (x) (right)

Resource allocation Problem: Numerical

x	j	0	1	2	3	4
	$r_2(j)$	0	2	4	7	11
0	$S_3(x-j)$	0				
	+	[0]				
1	$S_3(x-j)$	5	0			
	+	[5]	2			
2	$S_3(x-j)$	7	5	0		
	+	[7]	[7]	4		
3	$S_3(x-j)$	8	7	5	0	
	+	8	[9]	[9]	7	
4	$S_3(x-j)$	9	8	7	5	0
	+	9	10	11	[12]	11

Table: Computation for $S_2(x)$

Resource allocation Problem: Numerical

x	j	0	1	2	3	4
	$r_1(j)$	0	4	6	9	10
4	$S_2(x-j)$	12	9	7	5	0
	+	12	13	13	**14**	10

Table: Computation for $S_1(4)$

Thank You