UNIT – III

Greedy Method: General Method, Job Sequencing with deadlines, Knapsack Problem, Minimum cost spanning trees, Single Source Shortest Paths,

Dynamic Programming: General Method, All pairs shortest paths.0/1 Knapsack, String Editing, Travelling Salesperson problem, Optimal Binary Search Trees.

……

Among all the algorithmic approaches, the simplest and straightforward approach is the Greedy method. In this approach, the decision is taken on the basis of current available information without worrying about the effect of the current decision in future.

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. Greedy method is easy to implement and quite efficient in most of the cases. Hence, we can say that Greedy algorithm is an algorithmic paradigm based on heuristic that follows local optimal choice at each step with the hope of finding global optimal solution.

In many problems, it does not produce an optimal solution though it gives an approximate (near optimal) solution in a reasonable time.

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.

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.

Where Greedy Approach Fails

In many problems, Greedy algorithm fails to find an optimal solution, moreover it may produce a worst solution. Problems like Travelling Salesman and Knapsack cannot be solved using this approach.

General method of greedy

```
Algorithm greedy(a,n)
{
For i=1 to n do
{
 X=select(a)
```

If feasible(x) then

Solution = solution+x;

}

}

2. 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. One general approach to difficult problems is to identify the most restrictive constraint, ignore the others, solve a knapsack problem, and somehow adjust the solution to satisfy the ignored constraints.

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

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

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 x_i of i^{th} item.

$0 \leqslant xi \leqslant 1$

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 $\frac{p_i}{w_i}$, so that $\frac{p_i+1}{w_i+1} \leq \frac{p_i}{w_i}$. Here, ***x*** is an array to store the fraction of items.

Algorithm: Greedy-Fractional-Knapsack (w[1..n], p[1..n], W)

```
for i = 1 to n
  do x[i] = 0
weight = 0
for i = 1 to n
  if weight + w[i] ≤ W then
    x[i] = 1
    weight = weight + w[i]
  else
    x[i] = (W - weight) / w[i]
    weight = W
    break
return x
```

Analysis

If the provided items are already sorted into a decreasing order of **piwi**, then the while loop takes a time in ***O(n)***; Therefore, the total time including the sort is in ***O(n logn)***.

Example

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

Solution

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 $\frac{\mathbf{p_i}}{\mathbf{w_i}}$. 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

After sorting all the items according to *piwi*

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**

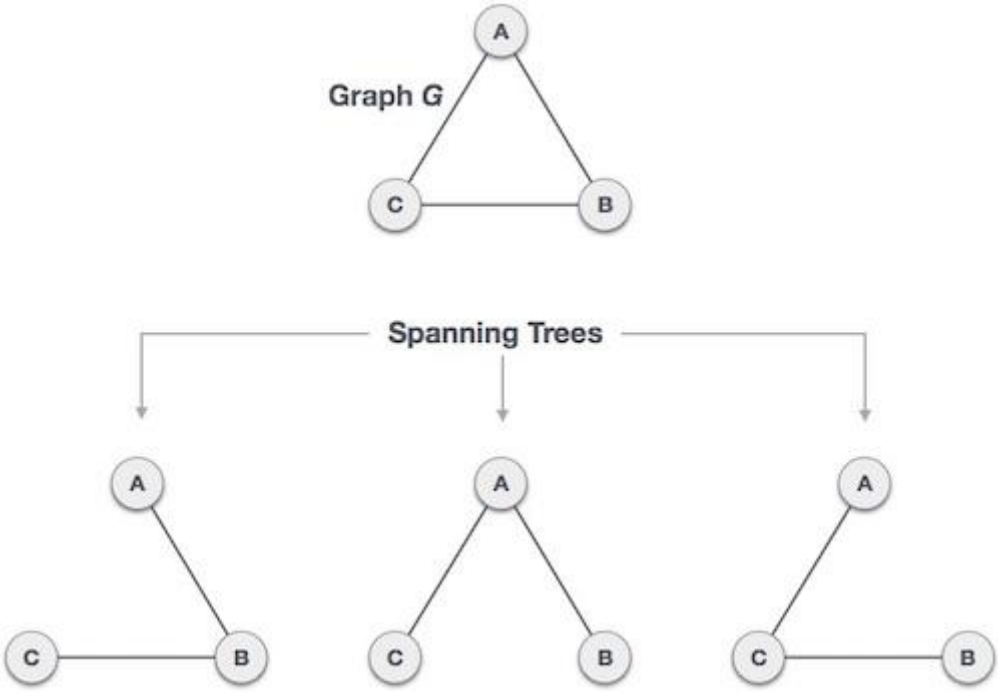
This is the optimal solution. We cannot gain more profit selecting any different combination of items.

3. minimum-cost spanning Trees

What is spanning Tree?

A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected.

By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.



We found three spanning trees off one complete graph. A complete undirected graph can have maximum n^{n-2} number of spanning trees, where **n** is the number of nodes. In the above addressed example, **n is 3,** hence $3^{3-2} = 3$ spanning trees are possible.

Properties of Spanning Tree

- Spanning tree has **n-1** edges, where **n** is the number of nodes (vertices).
- From a complete graph, by removing maximum **e - n + 1** edges, we can construct a spanning tree.
- A complete graph can have maximum n^{n-2} number of spanning trees.

Thus, we can conclude that spanning trees are a subset of connected Graph G and disconnected graphs do not have spanning tree.

Minimum Spanning Tree (MST)

In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph. In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

Minimum Spanning-Tree Algorithm

We shall learn about two most important spanning tree algorithms here −

- Kruskal's Algorithm
- Prim's Algorithm

Both are greedy algorithms.

1. Prim's Algorithm

- Prim's Algorithm is a famous greedy algorithm.
- It is used for finding the Minimum Spanning Tree (MST) of a given graph.
- To apply Prim's algorithm, the given graph must be weighted, connected and undirected.

Prim's Algorithm Implementation-

The implementation of Prim's Algorithm is explained in the following steps-

Step-01:

- Randomly choose any vertex.
- The vertex connecting to the edge having least weight is usually selected.

Step-02:

- Find all the edges that connect the tree to new vertices.
- Find the least weight edge among those edges and include it in the existing tree.
- If including that edge creates a cycle, then reject that edge and look for the next least weight edge.

Step-03:

- Keep repeating step-02 until all the vertices are included and Minimum Spanning Tree (MST) is obtained.

Prim's Algorithm Time Complexity-

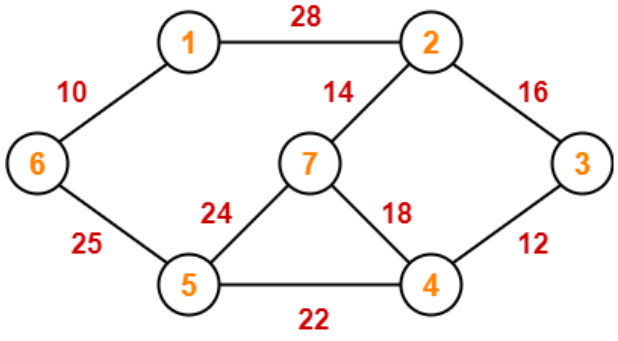
Worst case time complexity of Prim's Algorithm is-

- O(ElogV) using binary heap
- O(E + VlogV) using Fibonacci heap

```
1   Algorithm Prim(E, cost, n, t)
2   // E is the set of edges in G. cost[1 : n, 1 : n] is the cost
3   // adjacency matrix of an n vertex graph such that cost[i, j] is
4   // either a positive real number or ∞ if no edge (i, j) exists.
5   // A minimum spanning tree is computed and stored as a set of
6   // edges in the array t[1 : n − 1, 1 : 2]. (t[i, 1], t[i, 2]) is an edge in
7   // the minimum-cost spanning tree. The final cost is returned.
8   {
9       Let (k, l) be an edge of minimum cost in E;
10      mincost := cost[k, l];
11      t[1, 1] := k; t[1, 2] := l;
12      for i := 1 to n do  // Initialize near.
13          if (cost[i, l] < cost[i, k]) then near[i] := l;
14          else near[i] := k;
15      near[k] := near[l] := 0;
16      for i := 2 to n − 1 do
17      { // Find n − 2 additional edges for t.
18          Let j be an index such that near[j] ≠ 0 and
19          cost[j, near[j]] is minimum;
20          t[i, 1] := j; t[i, 2] := near[j];
21          mincost := mincost + cost[j, near[j]];
22          near[j] := 0;
23          for k := 1 to n do // Update near[ ].
24              if ((near[k] ≠ 0) and (cost[k, near[k]] > cost[k, j]))
25                  then near[k] := j;
26      }
27      return mincost;
28  }
```

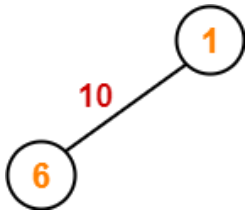
Algorithm 4.8 Prim's minimum-cost spanning tree algorithm

Example: Construct the minimum spanning tree (MST) for the given graph using Prim's Algorithm-

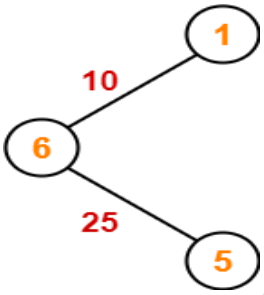


Step-1:

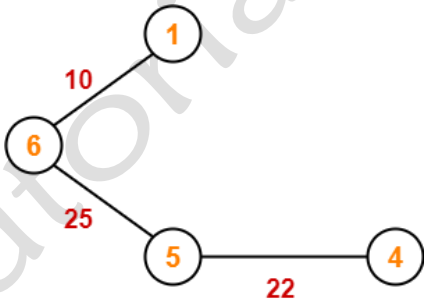
- Randomly choose any vertex. (Here vertex 1)
- The vertex connecting to the edge having least weight is usually selected.



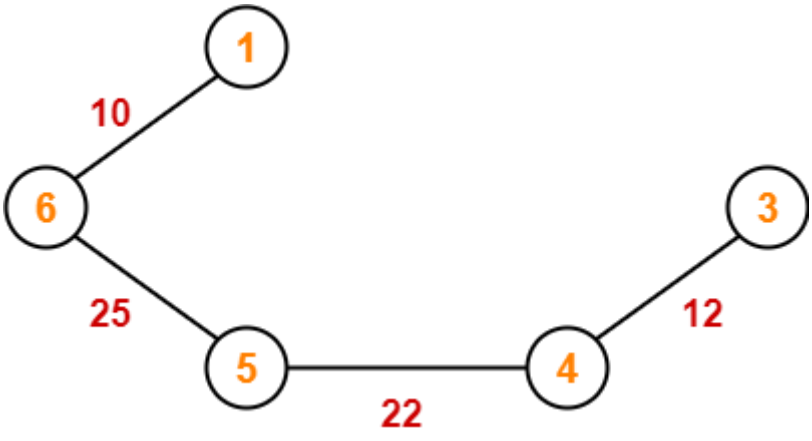
Step-2: Now we are at node / Vertex 6, It has two adjacent edges, one is already selected, select second one.



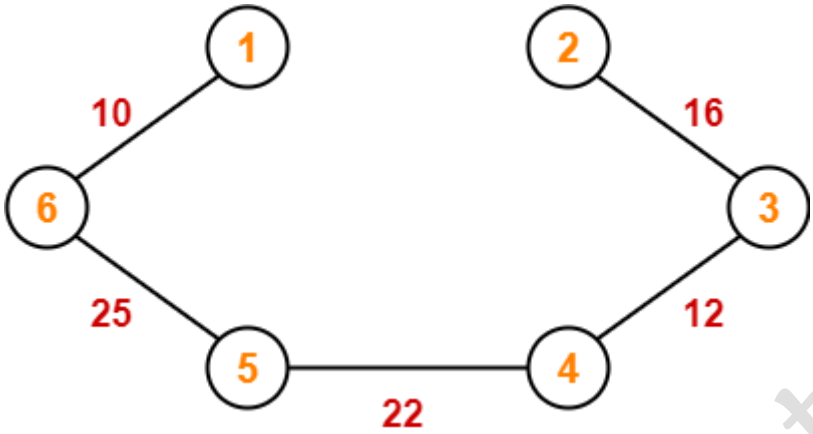
Step-3: Now we are at node 5, it has three edges connected, one is already selected, from reaming two select minimum cost edge (that is having minimum weight) Such that no loops can be formed by adding that vertex.



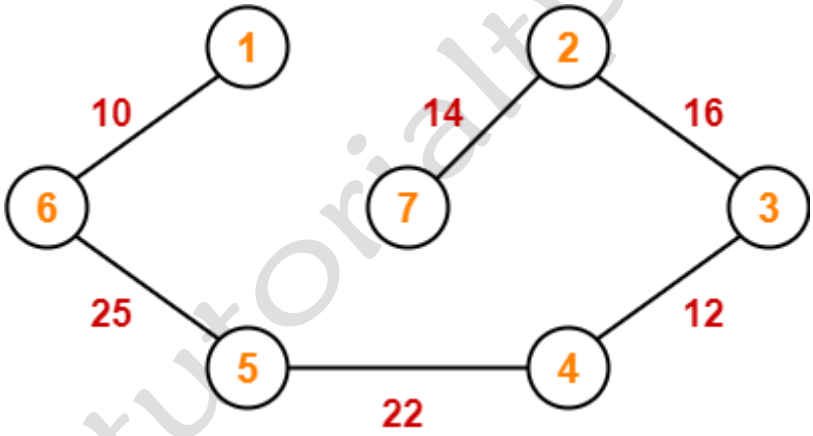
Step-4: Now we are at node 4, select the minimum cost edge from the edges connected to this node. Such that no loops can be formed by adding that vertex.



Step-5: Now we are at node 3, since the minimum cost edge is already selected, so to reach node 2 we selected the edge which cost 16. Then the MST is



Step-6: Now we are at node 2, select minimum cost edge from the edges attached to this node. Such that no loops can be formed by adding that vertex.



Since all the vertices have been included in the MST, so we stop.

Now, Cost of Minimum Spanning Tree

= Sum of all edge weights

= 10 + 25 + 22 + 12 + 16 + 14

= 99 units

Time Complexity: $O(V^2)$, If the input graph is represented using an adjacency list, then the time complexity of Prim's algorithm can be reduced to O(E log V) with the help of a binary heap. In this implementation, we are always considering the spanning tree to start from the root of the graph

2. **Kruskal's Algorithm-**

- Kruskal's Algorithm is a famous greedy algorithm.
- It is used for finding the Minimum Spanning Tree (MST) of a given graph.
- To apply Kruskal's algorithm, the given graph must be weighted, connected and undirected.

Kruskal's Algorithm Implementation-

The implementation of Kruskal's Algorithm is explained in the following steps-

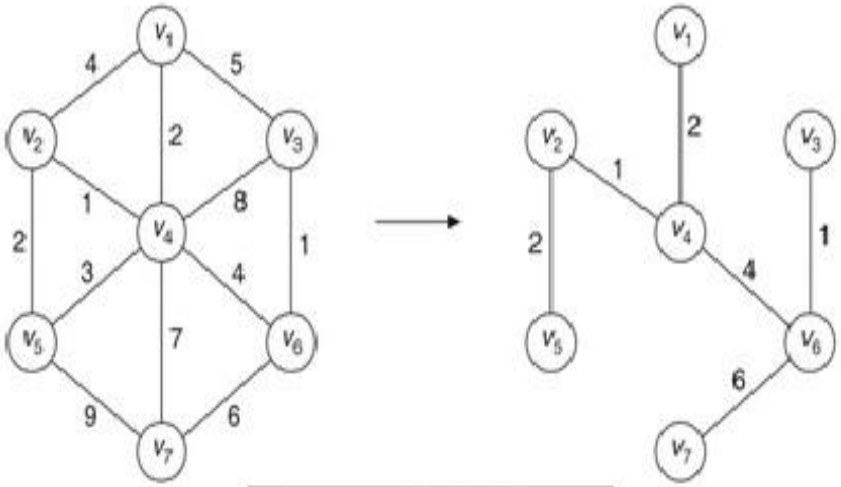
Step-01: Sort all the edges from low weight to high weight.

Step-02:

- Take the edge with the lowest weight and use it to connect the vertices of graph.
- If adding an edge creates a cycle, then reject that edge and go for the next least weight edge.

Step-03:

Keep adding edges until all the vertices are connected and a Minimum Spanning Tree (MST) is obtained.



Edge	Weight	Selection
v_2–v_4	1	✓
v_3–v_6	1	✓
v_2–v_5	2	✓
v_1–v_4	2	✓
v_4–v_5	3	×
v_1–v_2	4	×
v_4–v_6	4	✓
v_1–v_3	5	×
v_6–v_7	6	✓
v_4–v_7	7	
v_3–v_4	8	
v_5–v_7	9	

Figure The minimum spanning tree using Kruskal's algorithm.

```
1   Algorithm Kruskal(E, cost, n, t)
2   // E is the set of edges in G. G has n vertices. cost[u, v] is the
3   // cost of edge (u, v). t is the set of edges in the minimum-cost
4   // spanning tree. The final cost is returned.
5   {
6       Construct a heap out of the edge costs using Heapify;
7       for i := 1 to n do parent[i] := -1;
8       // Each vertex is in a different set.
9       i := 0; mincost := 0.0;
10      while ((i < n - 1)  and (heap not empty)) do
11      {
12          Delete a minimum cost edge (u, v) from the heap
13          and reheapify using Adjust;
14          j := Find(u); k := Find(v);
15          if (j ≠ k) then
16          {
17              i := i + 1;
18              t[i, 1] := u; t[i, 2] := v;
19              mincost := mincost + cost[u, v];
20              Union(j, k);
21          }
22      }
23      if (i ≠ n - 1) then write ("No spanning tree");
24      else return mincost;
25  }
```

Analysis: Where E is the number of edges in the graph and V is the number of vertices, Kruskal's Algorithm can be shown to run in O (E log E) time, or simply, O (E log V) time, all with simple data structures. These running times are equivalent because:

- E is at most V^2 and $\log V^2 = 2 \times \log V$ is O (log V).
- If we ignore isolated vertices, which will each their components of the minimum spanning tree, $V \leq 2E$, so log V is O (log E).

Thus, the total time is

1. O (E log E) = O (E log V).

4. Single Source Shortest Paths.

Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph.

It differs from the minimum spanning tree because the shortest distance between two vertices might not include all the vertices of the graph.

How Dijkstra's Algorithm works

Dijkstra's Algorithm works on the basis that any subpath B -> D of the shortest path A -> D between vertices A and D is also the shortest path between vertices B and D.

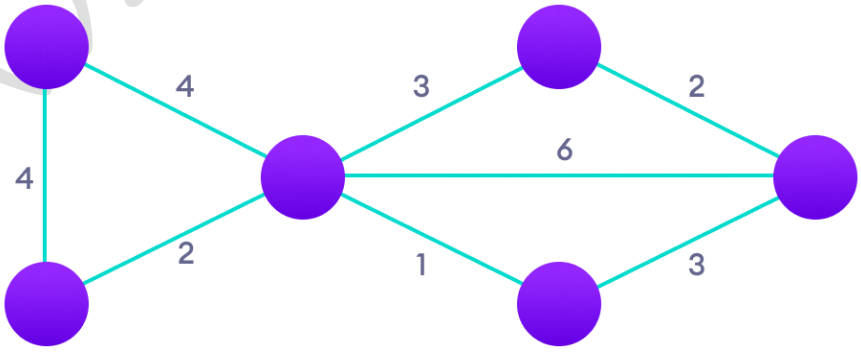
Each subpath is the shortest path

Djikstra used this property in the opposite direction i.e we overestimate the distance of each vertex from the starting vertex. Then we visit each node and its neighbors to find the shortest subpath to those neighbors.

The algorithm uses a greedy approach in the sense that we find the next best solution hoping that the end result is the best solution for the whole problem.

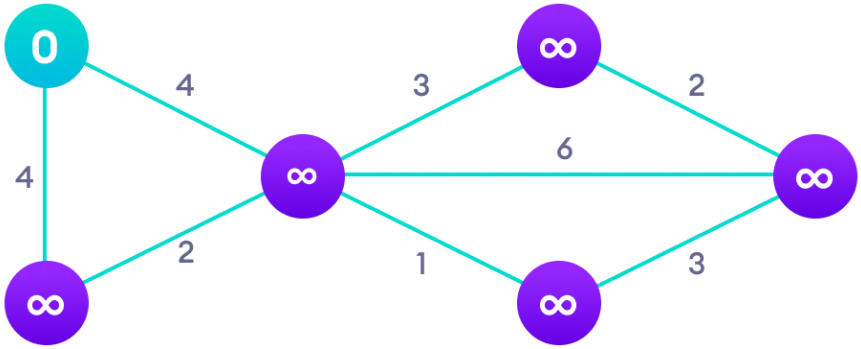
Example of Dijkstra's algorithm

It is easier to start with an example and then think about the algorithm.



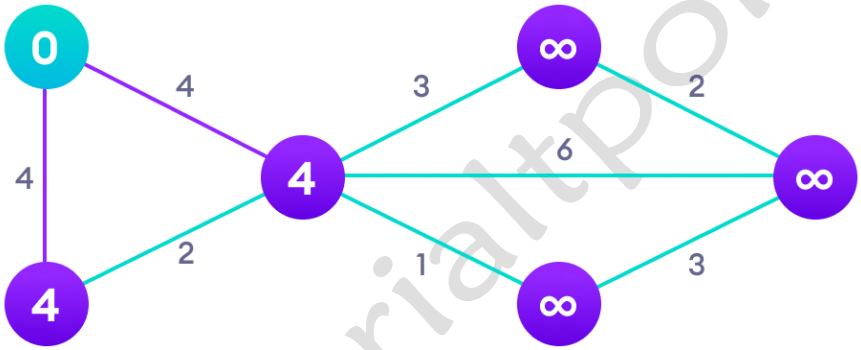
Step: 1

Start with a weighted graph



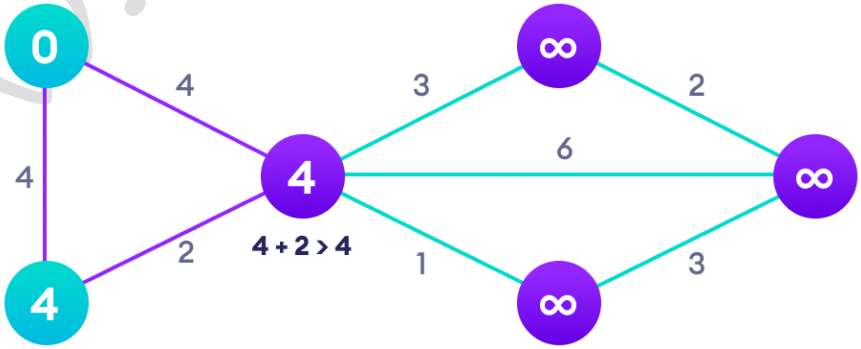
Step: 2

Choose a starting vertex and assign infinity path values to all other devices



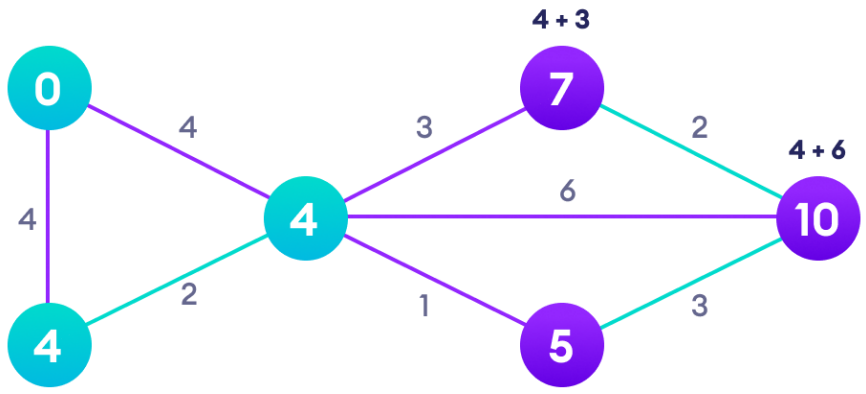
Step: 3

Go to each vertex and update its path length



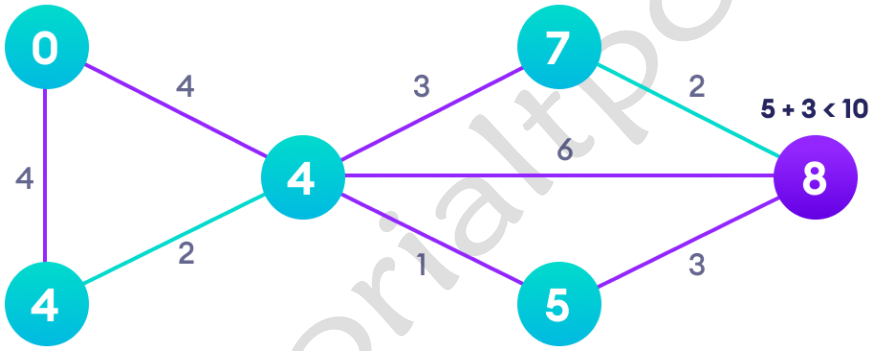
Step: 4

If the path length of the adjacent vertex is lesser than new path length, don't update it



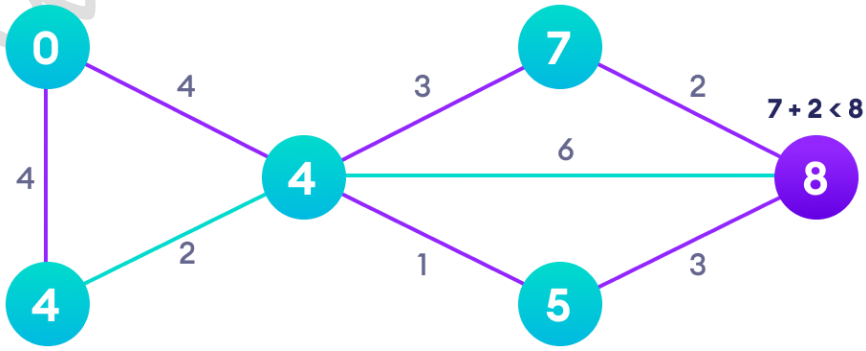
Step: 5

Avoid updating path lengths of already visited vertices



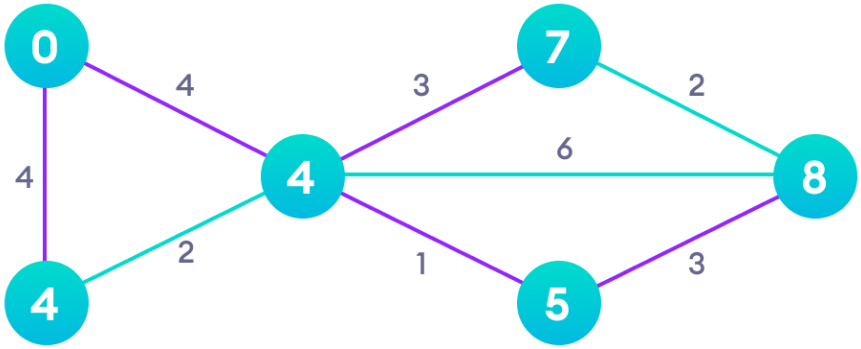
Step: 6

After each iteration, we pick the unvisited vertex with the least path length. So we choose 5 before 7



Step: 7

Notice how the rightmost vertex has its path length updated twice



Step: 8

Repeat until all the vertices have been visited

Djikstra's algorithm pseudocode

We need to maintain the path distance of every vertex. We can store that in an array of size v, where v is the number of vertices.

We also want to be able to get the shortest path, not only know the length of the shortest path. For this, we map each vertex to the vertex that last updated its path length.

Once the algorithm is over, we can backtrack from the destination vertex to the source vertex to find the path.

A minimum priority queue can be used to efficiently receive the vertex with least path distance.

```
function dijkstra(G, S)
    for each vertex V in G
        distance[V] <- infinite
        previous[V] <- NULL
        If V != S, add V to Priority Queue Q
    distance[S] <- 0
    while Q IS NOT EMPTY
        U <- Extract MIN from Q
        for each unvisited neighbour V of U
            tempDistance <- distance[U] + edge_weight(U, V)
            if tempDistance < distance[V]
                distance[V] <- tempDistance
                previous[V] <- U
    return distance[], previous[]
```

5. Job sequencing with deadlines

The arrangement of jobs on a single processor with deadline constraints is named as job sequencing with deadlines. We arrange n jobs on a processor in a sequence to obtain maximum profit subject to dead line. The sequencing problem is stated as follows:

1. Each i^{th} job is associated with a deadline and a profit.
2. For any i^{th} job, the profit pi is earned, if job is completed by its deadline.
3. To complete a job, it is processed on a processor for one unit of time.
4. Only one processor is available for processing all jobs.
5. Not all jobs have to be scheduled.

Job sequencing with deadlines is a classic problem in scheduling theory, where a set of jobs with associated deadlines and profits is given. The goal is to find a sequence of jobs that maximizes the total profit while ensuring that each job is completed before its deadline. This problem is also known as the "Job Sequencing Problem" or the "Single Machine Scheduling Problem with Deadlines."

Approach to Solution-

- A feasible solution would be a subset of jobs where each job of the subset gets completed within its deadline.
- Value of the feasible solution would be the sum of profit of all the jobs contained in the subset.
- An optimal solution of the problem would be a feasible solution which gives the maximum profit.

Greedy Algorithm-

Greedy Algorithm is adopted to determine how the next job is selected for an optimal solution.

The greedy algorithm described below always gives an optimal solution to the job sequencing problem-

Step-1: Sort all the given jobs in decreasing order of their profit.

Step-2:

- Check the value of maximum deadline.
- Draw a Gantt chart where maximum time on Gantt chart is the value of maximum deadline.

Step-3:

- Pick up the jobs one by one.
- Put the job on Gantt chart as far as possible from 0 ensuring that the job gets completed before its deadline

Problem-Given the jobs, their deadlines and associated profits as shown-

Jobs	J1	J2	J3	J4	J5	J6
Deadlines	5	3	3	2	4	2
Profits	200	180	190	300	120	100

Answer the following questions-

1. Write the optimal schedule that gives maximum profit.
2. Are all the jobs completed in the optimal schedule?
3. What is the maximum earned profit?

Solution-

Step-01:

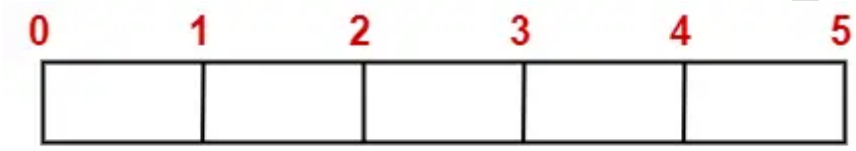
Sort all the given jobs in decreasing order of their profit-

Jobs	J4	J1	J3	J2	J5	J6
Deadlines	2	5	3	3	4	2
Profits	300	200	190	180	120	100

Step-02:

Value of maximum deadline = 5.

So, draw a Gantt chart with maximum time on Gantt chart = 5 units as shown-



Gantt Chart

Now,

- We take each job one by one in the order they appear in Step-01.
- We place the job on Gantt chart as far as possible from 0.

Step-03:

- We take job J4.
- Since its deadline is 2, so we place it in the first empty cell before deadline 2 as-

Step-04:

- We take job J1.
- Since its deadline is 5, so we place it in the first empty cell before deadline 5 as-

Step-05:

- We take job J3.
- Since its deadline is 3, so we place it in the first empty cell before deadline 3 as-

Step-06:

- We take job J2.
- Since its deadline is 3, so we place it in the first empty cell before deadline 3.
- Since the second and third cells are already filled, so we place job J2 in the first cell as-

Step-07:

- Now, we take job J5.
- Since its deadline is 4, so we place it in the first empty cell before deadline 4 as-

Now,

- The only job left is job J6 whose deadline is 2.
- All the slots before deadline 2 are already occupied.
- Thus, job J6 cannot be completed.

Now, the given questions may be answered as-

Part-01:

The optimal schedule is-

J2 , J4 , J3 , J5 , J1

This is the required order in which the jobs must be completed in order to obtain the maximum profit

Part-02

- All the jobs are not completed in optimal schedule.
- This is because job J6 could not be completed within its deadline

Part-03:

Maximum earned profit

= Sum of profit of all the jobs in optimal schedule

= Profit of job J2 + Profit of job J4 + Profit of job J3 + Profit of job J5 + Profit of job J1

= 180 + 300 + 190 + 120 + 200

= 990 units

Algorithm

JobSequence(job[1..n], p[1…n], d[1…n], list[1…n],n)

1. Initialize list with zeros // make all n time slots empty
2. profit=0; // profit = total profit
3. for i=1 to n do // set's i^{th} job dead line to k
4. k=d[i]:
5. while(k>0)
6. if(list[k]=0) // if kth slot is vacant
7. list[k]=job[i]; // assign i^{th} job to k^{th} slot
8. add p[i] to profit; // accumulate total profit
9. got to step 3; // exit from while loop (break)
10. Endif
11. Decrement k by 1; // otherwise get preceding slot.
12. Endwhile
13. Endfor
14. Print list

Chapter-II

Dynamic Programming: General Method, All pairs shortest paths.0/1 Knapsack, String Editing, Travelling Salesperson problem, Optimal Binary Search Trees

……

1 Q) The general method

Dynamic programming approach is similar to divide and conquer in breaking down the problem into smaller and yet smaller possible sub-problems. But unlike, divide and conquer, these sub-problems are not solved independently. Rather, results of these smaller sub-problems are remembered and used for similar or overlapping sub-problems.

Dynamic programming is used where we have problems, which can be divided into similar sub-problems, so that their results can be re-used. Mostly, these algorithms are used for optimization. Before solving the in-hand sub-problem, dynamic algorithm will try to examine the results of the previously solved sub-problems. The solutions of sub-problems are combined in order to achieve the best solution.

So we can say that −

- The problem should be able to be divided into smaller overlapping sub-problem.
- An optimum solution can be achieved by using an optimum solution of smaller sub-problems.
- Dynamic algorithms use Memoization.

Comparison

In contrast to greedy algorithms, where local optimization is addressed, dynamic algorithms are motivated for an overall optimization of the problem.

In contrast to divide and conquer algorithms, where solutions are combined to achieve an overall solution, dynamic algorithms use the output of a smaller sub-problem and then try to optimize a bigger sub-problem. Dynamic algorithms use Memoization to remember the output of already solved sub-problems.

Example

The following computer problems can be solved using dynamic programming approach −

- Fibonacci number series
- Knapsack problem
- Tower of Hanoi
- All pair shortest path by Floyd-Warshall
- Shortest path by Dijkstra
- Project scheduling

Dynamic programming can be used in both top-down and bottom-up manner. And of course, most of the times, referring to the previous solution output is cheaper than recomputing in terms of CPU cycles.

2 Q). All pairs-shortest paths

All Pairs Shortest Path Algorithm is also known as the Floyd-Warshall algorithm and this is an optimization problem that can be solved using dynamic programming.

Let G = <V, E> be a directed graph, where V is a set of vertices and E is a set of edges with nonnegative length. Find the shortest path between each pair of nodes.

L = Matrix, which gives the length of each edge

L[i, j] = 0, if i == j // Distance of node from itself is zero

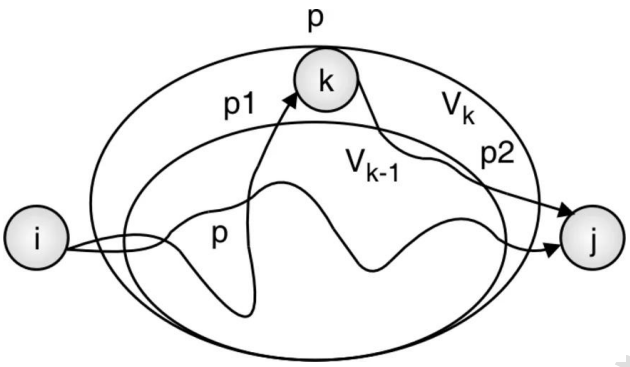
L[i, j] = ∞ , if i ≠ j and (i, j) ∉ E

L[i, j] = w (i, j), if i ≠ j and (i, j) ∈ E // w(i, j) is the weight of the edge (i, j)

Principle of optimality:

If k is the node on the shortest path from i to j, then the path from i to k and k to j, must also be shortest.

In the following figure, the optimal path from i to j is either p or summation of p_1 and p_2.



While considering k^{th} vertex as intermediate vertex, there are two possibilities:

- If k is not part of shortest path from i to j, we keep the distance D[i, j] as it is.
- If k is part of shortest path from i to j, update distance D[i, j] as D[i, k] + D[k, j].

Optimal sub structure of the problem is given as :

$D^k [i, j] = \min\{ D^{k-1} [i, j], D^{k-1} [i, k] + D^{k-1} [k, j] \}$

D^k = Distance matrix after k^{th} iteration

Algorithm for All Pairs Shortest Path

This approach is also known as the **Floyd-warshall** shortest path algorithm. The algorithm for all pair shortest path (APSP) problem is described below

Algorithm FLOYD_APSP (L)

// L is the matrix of size n × n representing original graph

// D is the distance matrix

D ← L

for k ← 1 to n **do**

  for i ← 1 to n **do**

```
    for j ← 1 to n do
        D[i, j]^k ← min ( D[i, j]^k-1, D[i, k]^k-1 + D[k, j]^k-1 )
    end
  end
end
return D
```

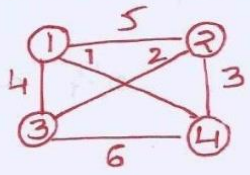
Complexity analysis of All Pairs Shortest Path Algorithm

It is very simple to derive the complexity of all pairs' shortest path problem from the above algorithm. It uses three nested loops. The innermost loop has only one statement. The complexity of that statement is Q(1).

The running time of the algorithm is computed as:

$$T(n) = \sum_{k=1}^{n} \sum_{i=1}^{n} \sum_{j=1}^{n} \Theta(1) = \sum_{k=1}^{n} \sum_{i=1}^{n} n = \sum_{k=1}^{n} n^2 = O(n^3)$$

Eg. Find the shortest distance from each vertex to other vertices in the below graph.



The All Pairs shortest-path problem is to determine a matrix A such that A(i,j) is the length of a shortest path i,j.

From floyd's alg. we have the recurrence relation :

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

$$\Rightarrow A^0(i,j) = cost(i,j),\quad 1\le i\le n,\quad 1\le j\le n$$

$$\boxed{A^k(i,j) = \min\{A^{k-1}(i,j), A^{k-1}(i,k)+A^{k-1}(k,j)\},\quad k\ge 1}$$

The cost matrix for given graph is given by

$$A^0 = cost(i,j) = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 5 & 4 & 1 \\ 5 & 0 & 2 & 3 \\ 4 & 2 & 0 & 6 \\ 1 & 3 & 6 & 0 \end{bmatrix} \end{array}$$

where $cost(i,j) = \begin{cases} 0 & \text{if } i=j \\ \text{edge cost}(i,j) & \text{if } i\ne j \ \&\ (i,j)\in E \\ \infty & \text{if } i\ne j \ \&\ (i,j)\notin E \end{cases}$

and E is set of edges in graph 'G'.

When computing $A^{(1)}$,

- 1st row & 1st column can be copied from $A^{(0)}$ to $A^{(1)}$ as they remain constant.
- all diagonal elements will be 0's always.

$$a^1_{23} = \min\{a^0_{23}, (a^0_{21} + a^0_{13})\} = \min\{2, (5+4)\} = 2$$

$$a^1_{24} = \min\{a^0_{24}, (a^0_{21} + a^0_{14})\} = \min\{3, (5+1)\} = 3$$

$$a^1_{32} = \min\{a^0_{32}, (a^0_{31} + a^0_{12})\} = \min\{2, (4+5)\} = 2$$

$$a^1_{34} = \min\{a^0_{34}, (a^0_{31} + a^0_{14})\} = \min\{6, (4+1)\} = 5$$

$$a^1_{42} = \min\{a^0_{42}, (a^0_{41} + a^0_{12})\} = \min\{3, (1+5)\} = 3$$

$$a^1_{43} = \min\{a^0_{43}, (a^0_{41} + a^0_{13})\} = \min\{6, (1+4)\} = 5$$

$$\therefore A^1 = \begin{bmatrix} 0 & 5 & 4 & 1 \\ 5 & 0 & 2 & 3 \\ 4 & 2 & 0 & 5 \\ 1 & 3 & 5 & 0 \end{bmatrix}$$

11ly $A^2 = \begin{bmatrix} 0 & 5 & 4 & 1 \\ 5 & 0 & 2 & 3 \\ 4 & 2 & 0 & 5 \\ 1 & 3 & 5 & 0 \end{bmatrix}$

11ly $A^3 = \begin{bmatrix} 0 & 5 & 4 & 1 \\ 5 & 0 & 2 & 3 \\ 4 & 2 & 0 & 5 \\ 1 & 3 & 5 & 0 \end{bmatrix}$

$$A^4 = \begin{bmatrix} 0 & 4 & 4 & 1 \\ 4 & 0 & 2 & 3 \\ 4 & 2 & 0 & 5 \\ 1 & 3 & 5 & 0 \end{bmatrix}$$

3.Q) 0/1 knapsack Problem

The 0/1 knapsack problem means that the items are either completely or no items are filled in a knapsack.

For example, we have two items having weights 2kg and 3kg, respectively. If we pick the 2kg item then we cannot pick 1kg item from the 2kg item (item is not divisible); we have to pick the 2kg item completely. This is a 0/1 knapsack problem in which either we pick the item completely or we will pick that item. The 0/1 knapsack problem is solved by the dynamic programming.

Example of 0/1 knapsack problem.

Consider the problem having weights and profits are:

Weights: {2,3,4,5}

Profits: {1,2,5,6}

The weight of the knapsack is 8 kg

The number of items is 4

The above problem can be solved by using the following method:

x_i = {1, 0, 0, 1}

= {0, 0, 0, 1}

= {0, 1, 0, 1}

The above are the possible combinations. 1 denotes that the item is completely picked and 0 means that no item is picked. Since there are 4 items so possible combinations will be:

2^4 = 16; So. There are 16 possible combinations that can be made by using the above problem. Once all the combinations are made, we have to select the combination that provides the maximum profit.

Another approach to solve the problem is dynamic programming approach. In dynamic programming approach, the complicated problem is divided into sub-problems, then we find the solution of a sub-problem and the solution of the sub-problem will be used to find the solution of a complex problem.

ALGORITHM *MFKnapsack*(i, j)

```
//Implements the memory function method for the knapsack problem
//Input: A nonnegative integer i indicating the number of the first
//       items being considered and a nonnegative integer j indicating
//       the knapsack capacity
//Output: The value of an optimal feasible subset of the first i items
//Note: Uses as global variables input arrays Weights[1..n], Values[1..n],
//and table F[0..n, 0..W] whose entries are initialized with −1's except for
//row 0 and column 0 initialized with 0's
if F[i, j] < 0
    if j < Weights[i]
        value ← MFKnapsack(i − 1, j)
    else
        value ← max(MFKnapsack(i − 1, j),
                     Values[i] + MFKnapsack(i − 1, j − Weights[i]))
    F[i, j] ← value
return F[i, j]
```

Eg. Find the optimal solution for the 0/1 knapsack problem making use of dynamic programming.
consider $n=4$, $w=5kg$, $w[1:4]=(2,3,4,5)$ and $P[1:4]=(3,4,5,6)$

Given, Knapsack capacity $(w) = 5kg$
No. of items $(n) = 4$
Weight of each item, $(w_1, w_2, w_3, w_4) = (2, 3, 4, 5)$
and Value of each item, $(P_1, P_2, P_3, P_4) = (3, 4, 5, 6)$

The recurrence relation (formula) is

$$F(i,j) = \begin{cases} \max\{F(i-1,j),\ Value_i + F(i-1, j-weight_i)\} & \text{if } j-w_i \geq 0 \\ F(i-1,j), & j-w_i < 0 \end{cases}$$

Initialize the F-table as below (fill 0^{th} row & 0^{th} col. with 0's) and compute remaining values using above formula

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

i ↑ , j →

computing $F(1,1)$:
$i=1,\ j=1,\ Value_1 = 3,\ weight_1 = 2.$

$\therefore F(1,1) = \max\{F(1-1, 1),\ 3 + F(1-1, 1-2)\}$
$= \max\{F(0,1),\ 3 + F(0,-1)\}$
$= T(0,1)$
$= 0$

↳ ignore as $F(0,-1)$ doesn't exist as $j < weight_i$ i.e; $1 < 2$.

$F(1,2) = \max\{F(1-1,2),\ 3+F(1-1,2-2)\}$

$= \max\{F(0,2),\ 3+F(0,0)\}$

$= \max\{0,\ 3+0\} = 3$

$F(1,3) = \max\{F(0,3),\ 3+F(0,1)\} = \max\{0,\ 3+0\} = 3$

$F(1,4) = \max\{F(0,4),\ 3+F(0,2)\} = \max\{0,\ 3+0\} = 3$

$F(1,5) = \max\{F(0,5),\ 3+F(0,3)\} = \max\{0,\ 3+0\} = 3$

$F(2,1) = \max\{F(1,1),\ 4+F(1,-2)\} = F(1,1) = 0$

$F(2,2) = \max\{F(1,2),\ 4+F(1,-1)\} = F(1,2) = 3$

$F(2,3) = \max\{F(1,3),\ 4+F(1,0)\} = \max\{3,\ 4+0\} = 4$

$F(2,4) = \max\{F(1,4),\ 4+F(1,1)\} = \max\{3,\ 4+0\} = 4$

$F(2,5) = \max\{F(1,5),\ 4+F(1,2)\} = \max\{3,\ 4+3\} = 7$

$F(3,1) = \max\{F(2,1),\ 5+F(2,-3)\} = F(2,1) = 0$

$F(3,2) = \max\{F(2,2),\ 5+F(2,-2)\} = F(2,2) = 3$

$F(3,3) = \max\{F(2,3),\ 5+F(2,-1)\} = F(2,3) = 4$

$F(3,4) = \max\{F(2,4),\ 5+F(2,0)\} = \max\{F(2,4),\ 5+0\} = 5$

$F(3,5) = \max\{F(2,5),\ 5+F(2,1)\} = \max\{7,\ 5+0\} = 7$

$F(4,1) = \max\{F(3,1),\ 6+F(3,-4)\} = F(3,1) = 0$

$F(4,2) = \max\{F(3,2),\ 6+F(3,-3)\} = F(3,2) = 3$

$F(4,3) = \max\{F(3,3),\ 6+F(3,-2)\} = F(3,3) = 4$

$F(4,4) = \max\{F(3,4),\ 6+F(3,-1)\} = F(3,4) = 5$

$F(4,5) = \max\{F(3,5),\ 6+F(3,0)\} = \max\{7,\ 6+0\} = 7.$

∴ The final F-table from above computations is

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	(7)

The last entry represents the max. possible value that can be kept into the knapsack

Here it is 7

∴ Max. value can be in knapsack = 7.

Finding items in knapsack:

The max. value is 7. So check when 7 is in F-table for the 1st time. It is 2nd row.

∴ I2 is to be included, whose wt. is 3 & value = 4

So remaining profit of knapsack = 7 − 4 = 3

Now check when 3 is kept in T-table for the 1st time. It is 1st row.

∴ I1 is to be included, whose wt is 2 & value = 3

So remaining profit = 3 − 3 = 0.

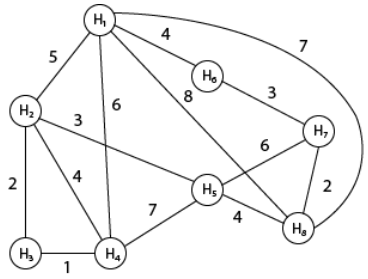
∴ Solution = $(x1, x2, x3, x4) = (1, 1, 0, 0)$.

4 Q) The traveling salesperson problem (TSP)

The travelling salesman problem asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?

Example: A newspaper agent daily drops the newspaper to the area assigned in such a manner that he has to cover all the houses in the respective area with minimum travel cost. Compute the minimum travel cost.

The area assigned to the agent where he has to drop the newspaper is shown in fig:



Solution: The cost- adjacency matrix of graph G is as follows:

$cost_{ij}$ =

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	4	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

The tour starts from area H_1 and then select the minimum cost area reachable from H_1.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
(H1)	0	5	0	6	0	(4)	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

Mark area H_6 because it is the minimum cost area reachable from H_1 and then select minimum cost area reachable from H_6.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
(1) H_1	0	5	0	6	0	(4)	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
(6) H_6	4	0	0	0	0	0	(3)	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

Mark area H_7 because it is the minimum cost area reachable from H_6 and then select minimum cost area reachable from H_7.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
(1) H_1	0	5	0	6	0	(4)	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
(6) H_6	4	0	0	0	0	0	(3)	0
(7) H_7	0	0	0	0	6	3	0	(2)
H_8	7	0	0	0	4	0	2	0

Mark area H_8 because it is the minimum cost area reachable from H_8.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
(1) H_1	0	5	0	6	0	(4)	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
(6) H_6	4	0	0	0	0	0	(3)	0
(7) H_7	0	0	0	0	6	3	0	(2)
(8) H_8	7	0	0	0	(4)	0	2	0

Mark area H_5 because it is the minimum cost area reachable from H_5.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
(1) H_1	0	5	0	6	0	(4)	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
(5) H_5	0	(3)	0	7	0	0	6	4
(6) H_6	4	0	0	0	0	0	(3)	0
(7) H_7	0	0	0	0	6	3	0	(2)
(8) H_8	7	0	0	0	(4)	0	2	0

Mark area H_2 because it is the minimum cost area reachable from H_2.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	(4)	0	7
H_2	5	0	(2)	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	(3)	0	7	0	0	6	4
H_6	4	0	0	0	0	0	(3)	0
H_7	0	0	0	0	6	3	0	(2)
H_8	7	0	0	0	(4)	0	2	0

Mark area H_3 because it is the minimum cost area reachable from H_3.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	(4)	0	7
H_2	5	0	(2)	4	3	0	0	0
H_3	0	2	0	(1)	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	(3)	0	7	0	0	6	4
H_6	4	0	0	0	0	0	(3)	0
H_7	0	0	0	0	6	3	0	(2)
H_8	7	0	0	0	(4)	0	2	0

Mark area H_4 and then select the minimum cost area reachable from H_4 it is H_1.So, using the greedy strategy, we get the following.

4 3 2 4 3 2 1 6

$H_1 \rightarrow H_6 \rightarrow H_7 \rightarrow H_8 \rightarrow H_5 \rightarrow H_2 \rightarrow H_3 \rightarrow H_4 \rightarrow H1$.

Thus the minimum travel cost = 4 + 3 + 2 + 4 + 3 + 2 + 1 + 6 = 25

Let g(i,S) be the length of a shortest path starting at vertex i, going through all vertices in S and terminating at vertex 1. The function g(1, V-{1}) is the length of an optimal salesperson tour.

From the principal of optimality, it follows that

$$g(1,V-\{1\}) = \min_{2 \le k \le n} \{C1k + g(k, V-\{1,k\})\}$$

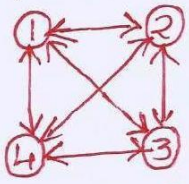
Generalizing above, we obtain for i E S

$$g(i,S) = \min_{j \in S} \{Cij+g(j,S-\{j\})\} \quad -- (1)$$

Time complexity: $O(n^2 2^n)$ as the computation of g(i,S) with $|S| = k$ requires k-1 comparisons when solving equation (1).

Space Complexity: $O(n.2^n)$

Eg. construct an optimal travelling sales person tour using dynamic programming with following data. Assume starting vertex as 1.



graph, G

cost matrix $c = \begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$

The recurrence for optimal travelling sales person tour is

$$g(i, S) = \min_{j \in S} \{ c_{ij} + g(j, S - \{j\}) \} \quad \text{---(1)}$$

and $g(i, \phi) = c_{i1}$, $1 \le i \le n$.

$g(2, \phi) = c_{21} = 5.$

$g(3, \phi) = c_{31} = 6$

$g(4, \phi) = c_{41} = 8.$

Now by using (1), we have

$g(2, \{3\}) = c_{23} + g(3, \phi) = 9+6 = 15$

$g(2, \{4\}) = c_{24} + g(4, \phi) = 10+8 = 18$

$g(3, \{2\}) = c_{32} + g(2, \phi) = 13+5 = 18$

$g(3, \{4\}) = c_{34} + g(4, \phi) = 12+8 = 20$

$g(4, \{2\}) = c_{42} + g(2, \phi) = 8+5 = 13$

$g(4, \{3\}) = c_{43} + g(3, \phi) = 9+6 = 15$

Now let us compute $g(i, S)$ with $|S| = 2$, $i \notin |S|$, $i \neq 1$, $1 \notin S$

$g(2, \{3,4\}) = \min \{ c_{23} + g(3, \{4\}), c_{24} + g(4, \{3\}) \}$

$= \min \{ 9+20, 10+15 \}$

$= \min \{ 29, 25 \}$

$= 25.$

$g(3,\{2,4\}) = \min\{c_{32}+g(2,\{4\}),\ c_{34}+g(4,\{2\})\}$

$= \min\{13+18,\ 12+13\}$

$= 25.$

$g(4,\{2,3\}) = \min\{c_{42}+g(2,\{3\}),\ c_{43}+g(3,\{2\})\}$

$= \min\{8+15,\ 9+18\}$

$= 23.$

finally from (1), we get

$g(1,\{2,3,4\}) = \min\{c_{12}+g(2,\{3,4\}),\ c_{13}+g(3,\{2,4\}),\ c_{14}+g(4,\{2,3\})\}$

$= \min\{10+25,\ 15+25,\ 20+28\}$

$= \min\{35,\ 40,\ 43\}$

$= 35.$

∴ The optimal tour cost is 35.

The tour of this length can be constructed if we retain with each $g(i,s)$ the value of j that minimizes the rt. hand side of eq. (1).

$J(1,\{2,3,4\}) = 2$ ⇒ The tour starts from 1 & moves to 2

$J(2,\{3,4\}) = 4$ ⇒ next we move to 4.

$J(4,\{3\}) = 3$ ⇒ next we move to 3.

∴ The optimal tour is $1 \to 2 \to 4 \to 3$ with cost = 35.