Greedy method

Greedy Method is an algorithmic paradigm used to solve optimization problems.

In Greedy method, solve the problem by making sequence of decisions and each decision is locally optimal. A decision, once made is not changed later. These locally optimal solutions will finally add up to a globally optimal solution.

•Makes the choice that looks best at the moment

•a locally optimal choice will lead to a globally optimal solution

• Everyday examples:

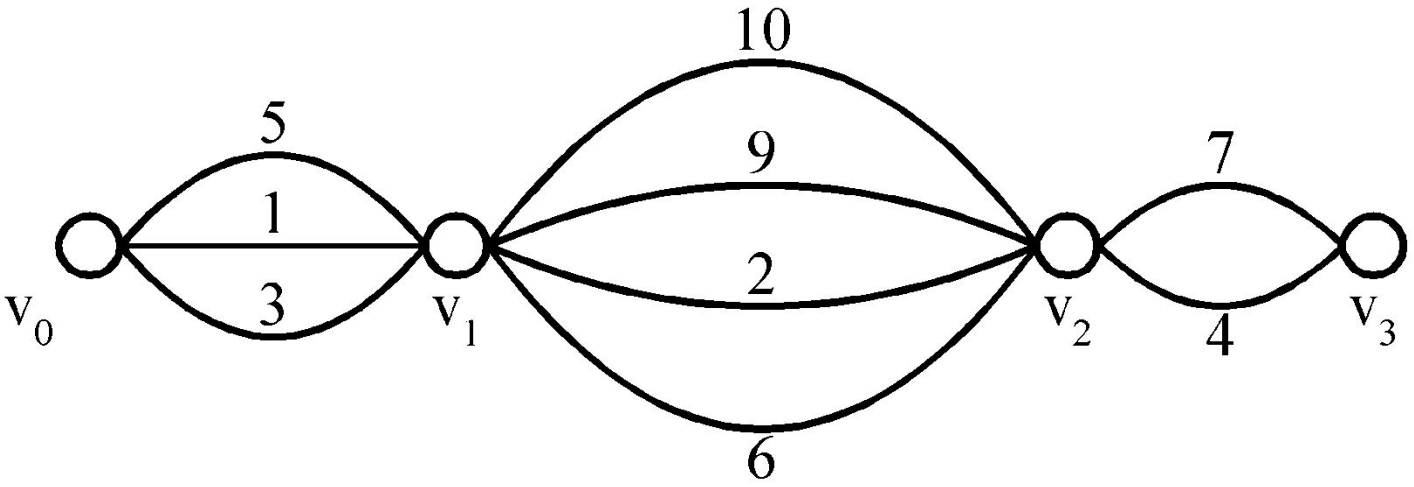
– Driving
-- Shopping

Feasible solutions are potential solutions to an optimization problem that satisfy all the given constraints. In other words, these are the solutions that are within the acceptable limits or boundaries defined by the problem's constraints.

An **optimal solution** is the best possible solution to an optimization problem within the set of all feasible solutions. In other words, it is the solution that either maximizes or minimizes the objective function, depending on the goal of the problem.

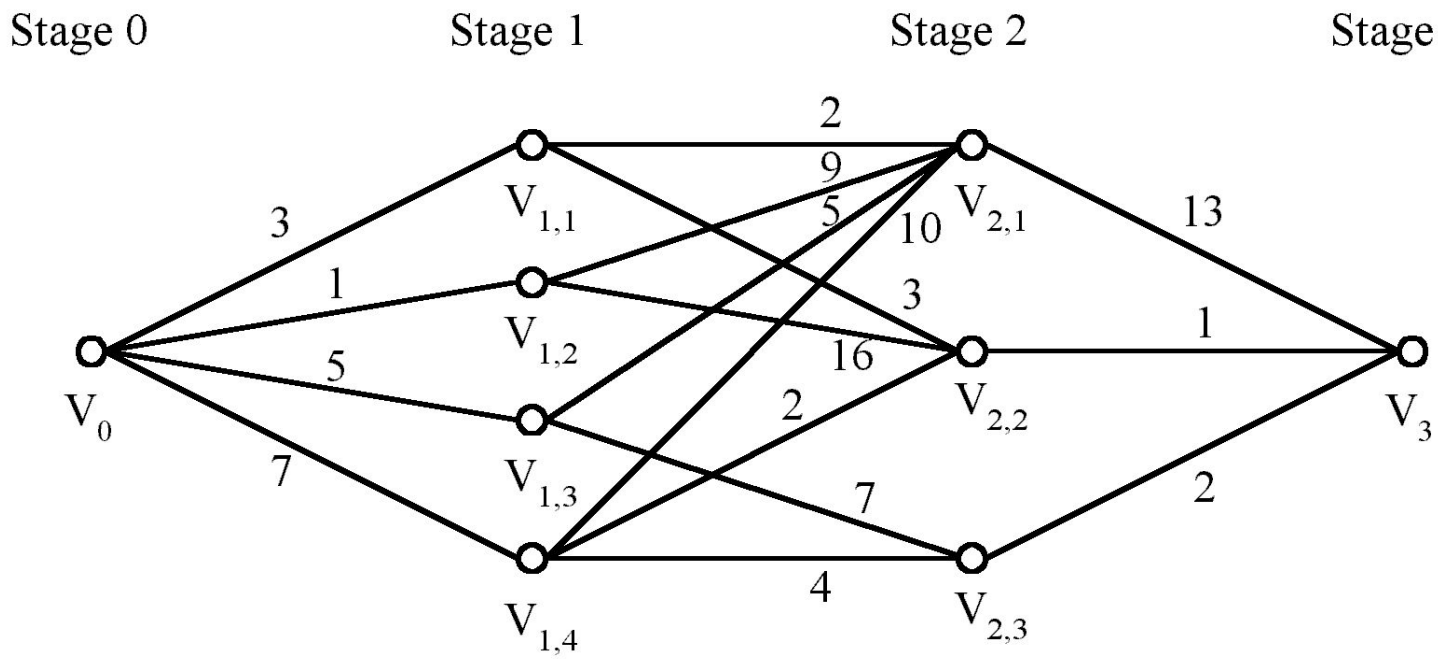
Shortest paths on a special graph

- Problem: Find a shortest path from v_0 to v_3.
- The greedy method can solve this problem.
- The shortest path: 1 + 2 + 4 = 7.



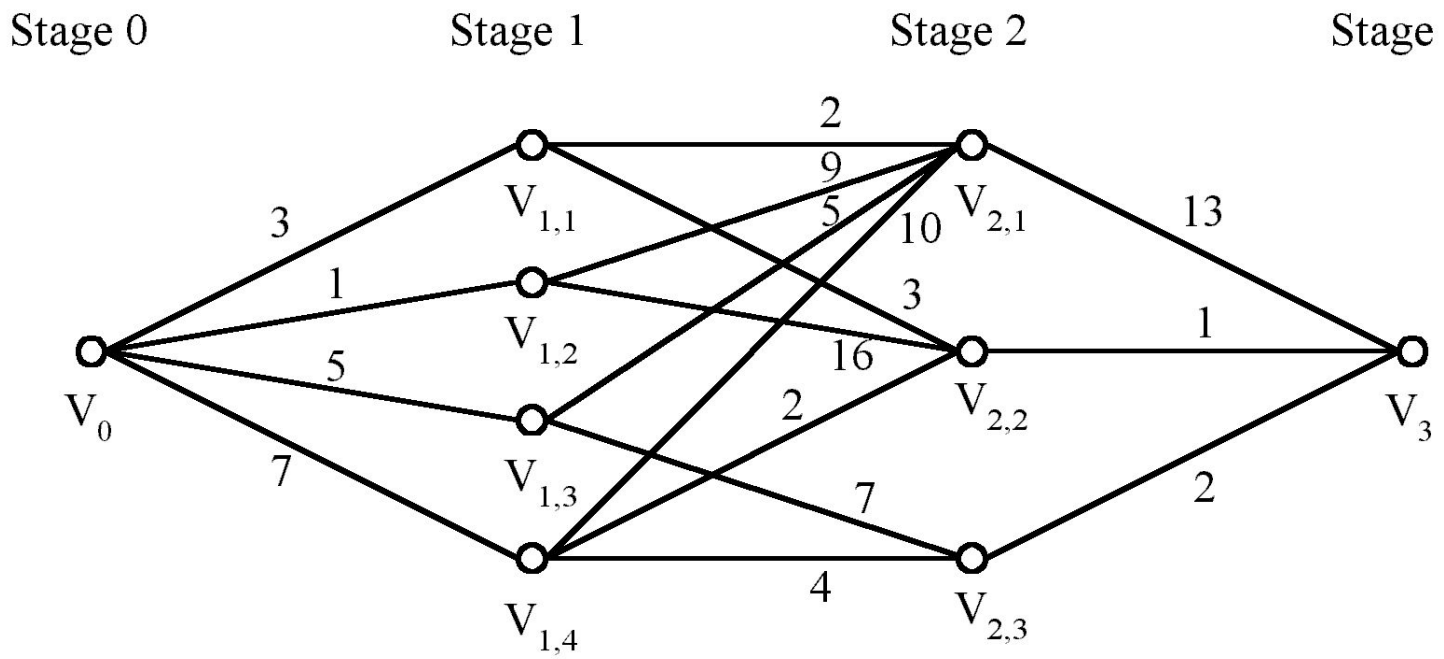
Shortest paths on a multi-stage graph

- Problem: Find a shortest path from v_0 to v_3 in the multi-stage graph.



Shortest paths on a multi-stage graph

- Problem: Find a shortest path from v_0 to v_3 in the multi-stage graph.



Greedy method: $v_0 v_{1,2} v_{2,1} v_3 = 23$
Optimal: $v_0 v_{1,1} v_{2,2} v_3 = 7$
The greedy method does not work.

CONTROL ABSTRACTION FOR GREEDY METHOD: -

```
Algorithm Greedy (a,n)
//a[1:n] contains the 'n' inputs
{     Solution: = 0; //initialize the solution
        for i: = 1 to n do
        {  x: = select (a);
           if Feasible (solution, x) then
    Solution: = Union (solution, x);
        }
        return solution;
}
```

Select is a function which is used to select an input from the set. Feasible is a function which verifies the constraints and determines weather the resultant solution is feasible or not. Union is a function which is used to add elements to the partially constructed set.

Knapsack Problem

We are given ‘n’ objects and a knapsack or bag.

Object ‘i’ has a weight wi and the knapsack has a capacity ‘m’.

If a fraction xi, 0<= xi<=1, of object ‘i’ is placed into the knapsack, then a profit pixi is earned.

The objective is to obtain a filling of the knapsack that maximizes the total profit earned.

Since the knapsack capacity is m, we require the total weight of all the chosen items to be at most m . Formally, the problem can be stated as

$$\text{maximize} \sum_{1 \le i \le n} p_i x_i$$

$$\text{subject to} \sum_{1 \le i \le n} w_i x_i \le m$$

$$\text{and } 0 \le x_i \le 1, \quad 1 \le i \le n$$

Example 4.1 Consider the following instance of the knapsack problem: $n = 3, m = 20, (p_1, p_2, p_3) = (25, 24, 15)$, and $(w_1, w_2, w_3) = (18, 15, 10)$. Four feasible solutions are:

	(x_1, x_2, x_3)	$\sum w_i x_i$	$\sum p_i x_i$
1.	(1/2, 1/3, 1/4)	16.5	24.25
2.	(1, 2/15, 0)	20	28.2
3.	(0, 2/3, 1)	20	31
4.	(0, 1, 1/2)	20	31.5

Of these four feasible solutions, solution 4 yields the maximum profit. As we shall soon see, this solution is optimal for the given problem instance. □

```
1   Algorithm GreedyKnapsack(m, n)
2   // p[1 : n] and w[1 : n] contain the profits and weights respectively
3   // of the n objects ordered such that p[i]/w[i] ≥ p[i + 1]/w[i + 1].
4   // m is the knapsack size and x[1 : n] is the solution vector.
5   {
6       for i := 1 to n do x[i] := 0.0; // Initialize x.
7       U := m;
8       for i := 1 to n do
9       {
10          if (w[i] > U) then break;
11          x[i] := 1.0; U := U − w[i];
12      }
13      if (i ≤ n) then x[i] := U/w[i];
14  }
```

The Greedy method is a natural fit for the Fractional Knapsack Problem because it focuses on making the locally optimal choice at each step.

The Greedy algorithm provides an optimal solution for the Fractional Knapsack Problem.

This is because selecting the items with the highest value-to-weight ratio at each step guarantees that you maximize the value within the given weight constraint.

Spanning treee

- A spanning tree of a graph is just a sub graph that contains all the vertices and is a tree.

- A tree (connected acyclic graph) which contains all the vertices of the graph.

- The weight of a tree is the sum of its edge's weight.

Minimum Cost Spanning treee

- The minimum cost spanning tree of a graph is the spanning tree with minimum weight or cost

Applications:- Find least expensive way to connect a set of cities , terminals, computers, etc.

Prim’s Algorithm

```
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  }
```

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  }
```

Why They Are Greedy:

- **Local Decisions**: Both algorithms make decisions based on local information. Kruskal's picks the smallest edge across the entire graph, while Prim's picks the smallest edge connecting the current MST to the rest of the graph.

- **No Backtracking**: Once a decision is made (an edge is added to the MST), neither algorithm reconsiders previous choices. This is a hallmark of greedy algorithms.

- **Optimal Solution**: Despite being greedy and making local decisions, both algorithms are proven to produce an optimal solution for the MST, which is the tree with the minimum possible total weight that spans all vertices.

JOB SEQUENCING WITH DEADLINES

The problem is stated as below:

There are n jobs to be processed on a machine.

Each job i has a deadline $d_i \geq 0$ and profit $p_i \geq 0$.

Pi is earned iff the job is completed by its deadline.

The job is completed if it is processed on a machine for unit time.

Only one machine is available for processing jobs.

Only one job is processed at a time on the machine.

An optimal solution is a feasible solution with maximum profit value.

Feasible solutions are obtained by various permutations and combination of jobs

Example : Let n = 4, $(p_1,p_2,p_3,p_4) = (100,10,15,27)$, $(d_1,d_2,d_3,d_4) = (2,1,2,1)$

Sr.No.	**Feasible Solution**	**Processing sequence**	profit
1	1	1	100
2	2	2	10
3	3	3	15
4	4	4	27
5	1,2	2,1	110
6	1,3	1,3 or 3,1	115
7	1,4	4,1	127
8	2,3	2,3	25
9	2,4	No feasible solutions	
10	3,4	4,3	42
11	No feasible solutions for combination of 3 jobs		

Example 2:

N=5

p1,p2,p3,p4,p5 =30,25,80,50

d1,d2,d3,d4,d5=1,3,2,1

```
1   Algorithm JS(d, j, n)
2   // d[i] ≥ 1, 1 ≤ i ≤ n are the deadlines, n ≥ 1. The jobs
3   // are ordered such that p[1] ≥ p[2] ≥ ··· ≥ p[n]. J[i]
4   // is the ith job in the optimal solution, 1 ≤ i ≤ k.
5   // Also, at termination d[J[i]] ≤ d[J[i + 1]], 1 ≤ i < k.
6   {
7       d[0] := J[0] := 0; // Initialize.
8       J[1] := 1; // Include job 1.
9       k := 1;
10      for i := 2 to n do
11      {
12          // Consider jobs in nonincreasing order of p[i]. Find
13          // position for i and check feasibility of insertion.
14          r := k;
15          while (d[J[r]] > d[i]) do r := r − 1;
16          if ((d[J[r]] ≤ d[i]) and (d[i] > r)) then
17          {
18              // Insert i into J[ ].
19              for q := k to (r + 1) step −1 do J[q + 1] := J[q];
20              J[r + 1] := i; k := k + 1;
21          }
22      }
23      return k;
24  }
```

```
1   Algorithm JS(d, j, n)
2   // d[i] ≥ 1, 1 ≤ i ≤ n are the deadlines, n ≥ 1. The jobs
3   // are ordered such that p[1] ≥ p[2] ≥ ··· ≥ p[n]. J[i]
4   // is the ith job in the optimal solution, 1 ≤ i ≤ k.
5   // Also, at termination d[J[i]] ≤ d[J[i + 1]], 1 ≤ i < k.
6   {
7       d[0] := J[0] := 0; // Initialize.
8       J[1] := 1; // Include job 1.
9       k := 1;
10      for i := 2 to n do
11      {
12          // Consider jobs in nonincreasing order of p[i]. Find
13          // position for i and check feasibility of insertion.
14          r := k;
15          while ((d[J[r]] > d[i]) and (d[J[r]] ≠ r)) do r := r − 1;
16          if ((d[J[r]] ≤ d[i]) and (d[i] > r)) then
17          {
18              // Insert i into J[ ].
19              for q := k to (r + 1) step −1 do J[q + 1] := J[q];
20              J[r + 1] := i; k := k + 1;
21          }
22      }
23      return k;
24  }
```

The Job Sequencing Problem typically involves scheduling jobs to maximize profit within a given deadline. Each job has a deadline and profit associated with it. The goal is to maximize the total profit by selecting the most profitable jobs that can be completed before their deadlines.

Dynamic programming

To solve a given problem, we need to solve different parts of the problem (subproblems), ***then combine the solutions of the subproblems to reach an overall solution***.

The dynamic programming approach *seeks to solve each subproblem* ***only once***, thus reducing the number of computations:

once the solution to a given subproblem has been computed, it is stored or "memo-ized": the next time the same solution is needed.

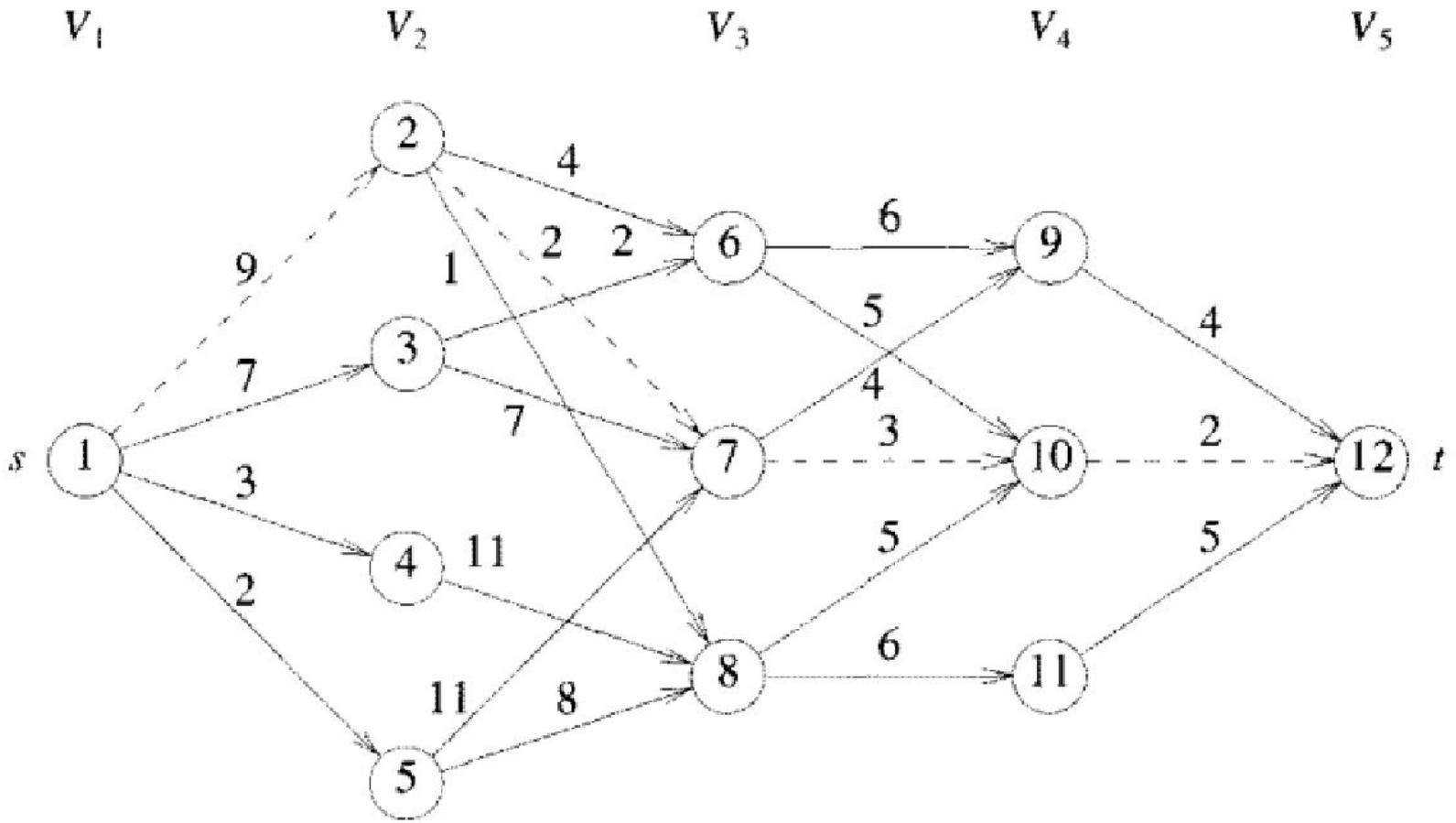


Figure 5.2 Five-stage graph

Elements and characteristics of Dynamic Programming

1. Simple Sub problems:

   We should be able to break the original problem into small sub problem that have the same substructure.

2. optimal substructure:

Optimal substructure is a key concept in dynamic programming (DP) that refers to a property of a problem where the optimal solution can be constructed from the optimal solutions of its sub problems.

In other words, if a problem has an optimal substructure, then the optimal solution to the problem can be composed of optimal solutions to its smaller sub problems.

For example, the Shortest Path problem has 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 combination of shortest path from u to x and shortest path from x to v.

3.Overlapping Subproblems:

In some problems, the same subproblems are solved multiple times as part of the recursive process.

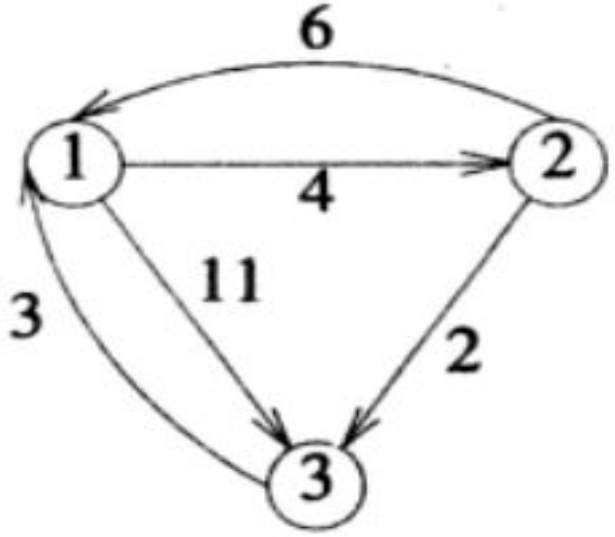
Instead of recomputing the result every time the subproblem is encountered, dynamic programming stores the result (using memoization or tabulation) and reuses it, which significantly reduces the time complexity.

Sr. No.	Greedy Method	Dynamic Programming
1.	Greedy method is used for obtaining **optimum solution.**	Dynamic programming is also for obtaining **optimum solution.**
2.	In Greedy method a set of **feasible solutions** and the picks up the optimum solution.	There is **no** special set of **feasible solutions** in this method.
3.	In Greedy method the optimum selection is **without revising previously** generated solutions.	Dynamic programming considers **all possible sequences** in order to obtain the optimum solution.
4.	In Greedy method there is **no** as such **guarantee** of getting **optimum solution.**	It is guaranteed that the dynamic programming will generate optimal solution using **principle of optimality.**

All pairs shortest Paths

```
0   Algorithm AllPaths(cost, A, n)
1   // cost[1 : n, 1 : n] is the cost adjacency matrix of a graph with
2   // n vertices; A[i, j] is the cost of a shortest path from vertex
3   // i to vertex j. cost[i, i] = 0.0, for 1 ≤ i ≤ n.
4   {
5       for i := 1 to n do
6           for j := 1 to n do
7               A[i, j] := cost[i, j]; // Copy cost into A.
8       for k := 1 to n do
9           for i := 1 to n do
10              for j := 1 to n do
11                  A[i, j] := min(A[i, j], A[i, k] + A[k, j]);
12  }
```

gorithm 5.3 Function to compute lengths of shortest paths



(a) Example digraph

A^0	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

(b) A^0

Travelling Salesman Problem

Let G(V,E) be a directed graph with edges cost Cij ,where Cij>0 for all I and j.

N=number of vertices.

A tour of G is a directed simple cycle that includes every vertex of G. the cost of a tour is the sum of the cost of the edges on the tour. The TSP is to find a tour with minimum cost.

Formula:

$$G(i,s) = \min_{j \in s}(C_{ij} + g(j, S-\{j\}))$$

Here,

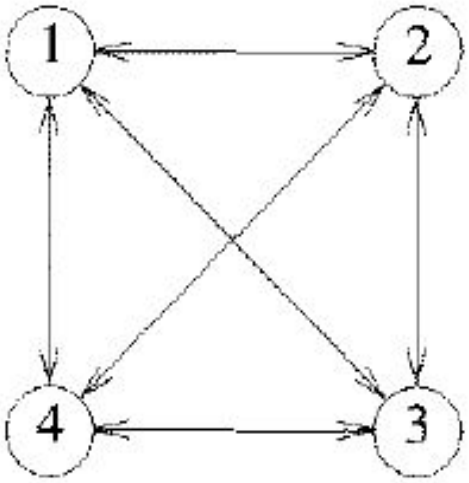
i, salesman is at ith node

s, remaining sets(uncovered vertices), have not been traversed yet

j, is the next node and should be a subset of S

For **n** number of vertices in a graph, there are **(*n* - 1)!** number of possibilities.

Example



(a)

0	10	15	20
5	0	9	10
6	13	0	12
8	8	9	0

(b)

Let 1 will be the starting node

consider sets of 0 element:

$$g(2, \varnothing) = c21 = 5$$

$$g(3, \varnothing) = c31 = 6$$

$$g(4, \varnothing) = c41 = 8$$

consider sets of 1 element:

$g(2,\{3\}) = c23 + g(3, \varnothing) = c23 + c31 = 9 + 6 = 15$

$g(2,\{4\}) = c24 + g(4, \varnothing) = c24 + c41 = 10 + 8 = 18$

$g(3,\{2\}) = c32 + g(2, \varnothing) = 13 + 5 = 18$

$g(3,\{4\}) = C34 + g(4, \varnothing) = 12 + 8 = 20$

$g(4,\{2\}) = C42 + g(2, \varnothing) = 8 + 5 = 13$

$g(4,\{3\}) = C43 + g(3, \varnothing) = 9 + 6 = 15$

consider sets of 2 elements:

$$g(2,\{3,4\})=\min((C23+g(3,\{4\}),C24+g(4,3)$$
$$= \min((9+20),10+15)$$
$$=\min(29,25)$$
$$=25$$

$$g(3,\{2,4\})=\min((C32+g(2,4),C34+g(4,2))$$
$$=\min(13+18,12+13)$$
$$=25$$

$$g(4,\{2,3\})=\min((c42+g(2,3),C43+g(3,2))$$
$$=\min(8+15,9+18)$$
$$=23$$

Finally

g(1,{2,3,4})

=min((C12+g(2,{3,4}),C13+g(3,{2,4}),C14+g(4,{2,3}))

=min(10+25,15+25,20+23)

=35

Optimal tour of the graph has length=35

Optimal tour is 1-2-4-3-1

Bellman ford algorithm

The Bellman-Ford algorithm is used to find the shortest paths from a single source vertex to all other vertices in a weighted graph.

Unlike Dijkstra's algorithm, which works only with graphs that have non-negative weights.

Bellman-Ford can handle graphs with negative weights. (but no negative cycles)

A negative cycle is a path we can follow in circles, where the sum of the edge weights is negative.

Algorithm Steps:

1. **Initialization**:
   - Set the distance to the source vertex to 0 and all other vertices to infinity.

- distance[u]=0
- distance[v]=∞; for all other vertices

2.**Relaxation**:

- Iterate through all edges in the graph, updating the distance to each vertex if a shorter path is found.

  if distance[u]+ C(u,v) < distance[v] then
  distance[v]=distance[u]+ C(u,v)

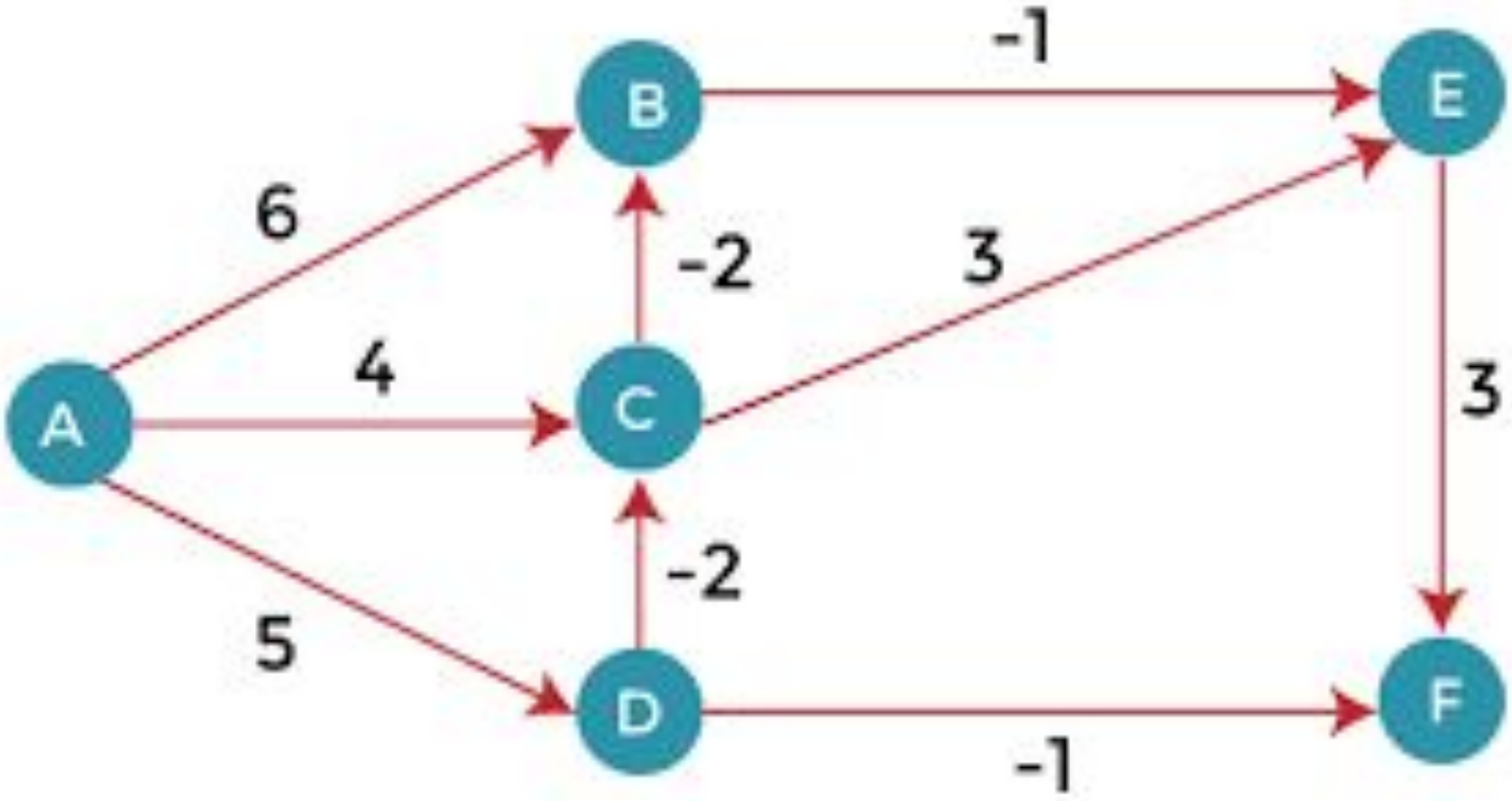
3.Repeat

- Repeat this step for $V-1$ times, where V is the number of vertices in the graph.

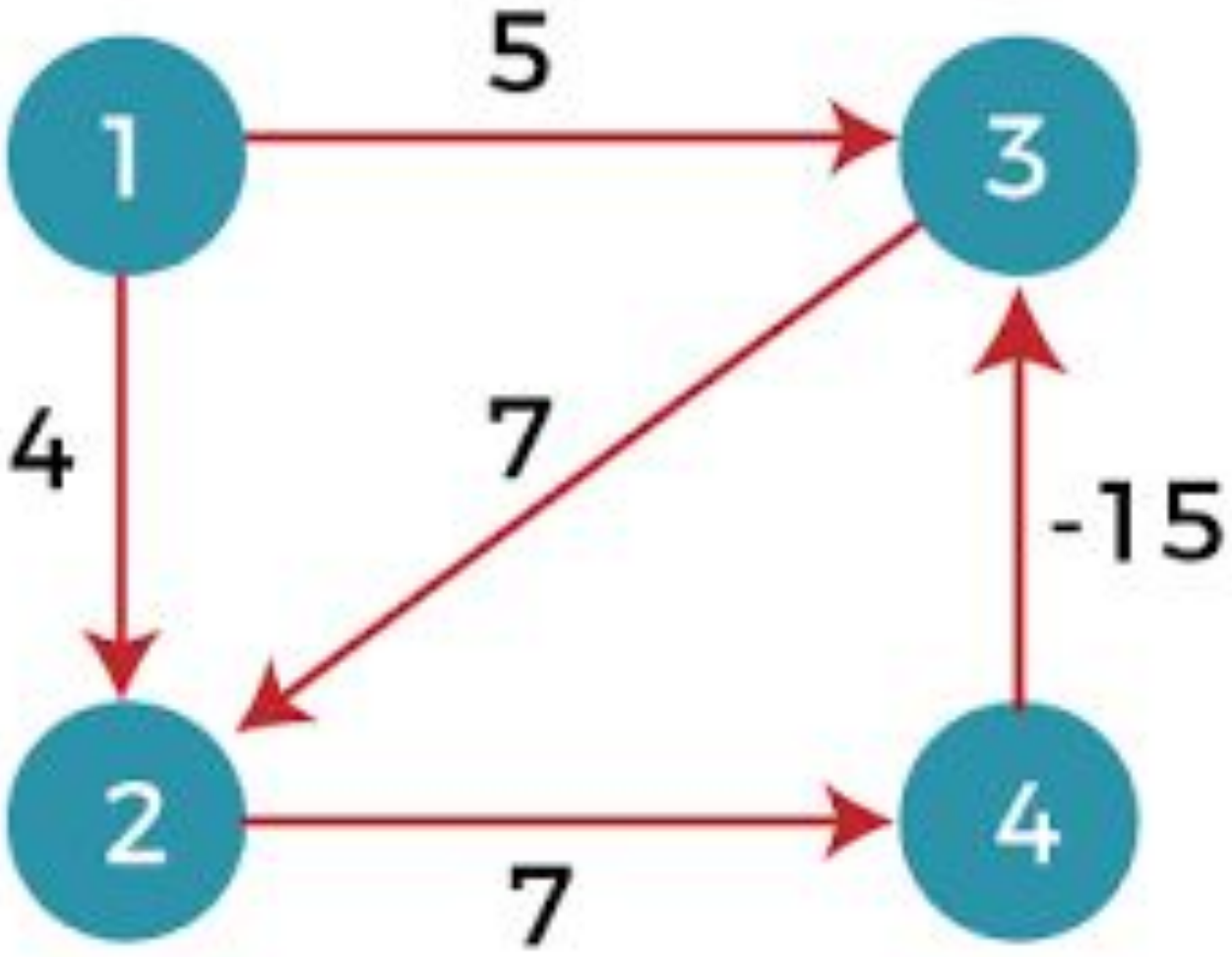
3. **Negative Weight Cycle Check**:

- After the $V-1$ iterations, go through all the edges one more time.

- If you can still reduce the distance to any vertex, it indicates the presence of a negative weight cycle in the graph.



(A,B), (A,C), A,D), (B,E), (C,B), (C,E), (D,C),(D,F), (E,F)



(1,2), (1,3), (2,4), (3,2), (4,3)

```
Algorithm BellmanFord(u, v, C, dist, n)
{
   for i := 1 to n do // Initialize dist.
      dist[i] := ∞;  // Set all distances to infinity initially
      dist[u] := 0; // Distance from source u to itself is 0

   for i := 1 to n-1 do // Repeat n-1 times for each edge (i, v) in the graph do
         if dist[i] + C(i, v) < dist[v] then
            dist[v] := dist[i] + C(i, v);

   // Check for negative weight cycles     for each edge (i, v) in the graph do
      if dist[i] + C(i, v) < dist[v] then
         error "Graph contains a negative weight cycle";
}
```