MODULE 4

Dynamic Programming :The general method

Dynamic programming, as greedy method, is a powerful algorithm design technique that can be used when the solution to the problem may be viewed as the result of a sequence of decisions

- 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.

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**.

Dynamic programming is based on the principle of optimality (also coined by Bellman).

- The principle of optimality states that no matter whatever the initial state and initial decision are, the remaining decision sequence must constitute an optimal decision sequence with regard to the state resulting from the first decision.
- The principle implies that an optimal decision sequence is comprised of optimal decision subsequences.

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.

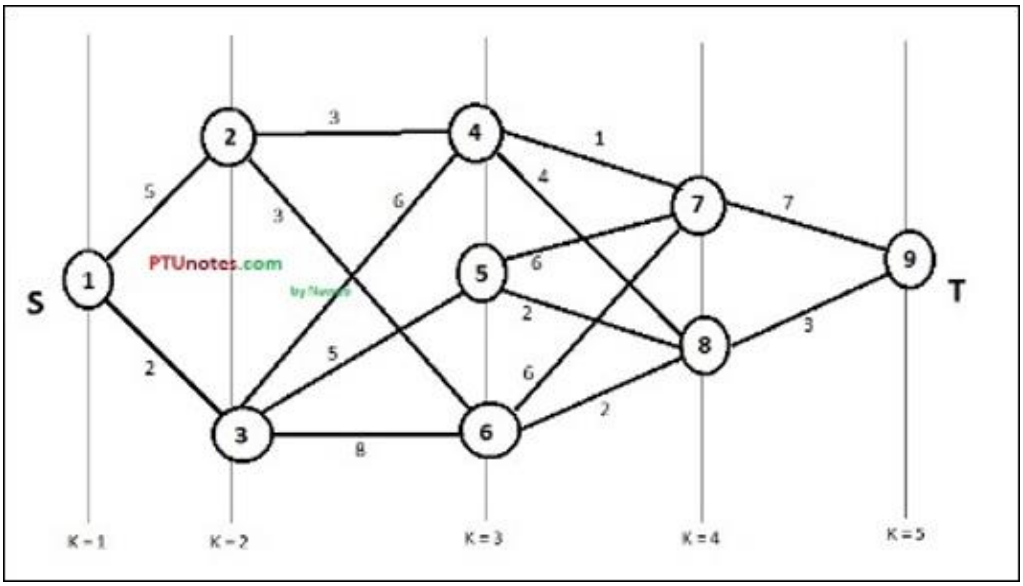
• **Greedy method vs Dynamic programming** –

- In greedy method, only one decision sequence is ever generated –
- In dynamic programming, many decision sequences may be generated –
- Sequences containing suboptimal sequences cannot be optimal because of principle of optimality, and so, will not be generated –

1.MULTISTAGE GRAPHS

1. The multistage graph problem is to find a minimum cost from a source to a sink.
2. A multistage graph is a directed graph having a number of multiple stages, where stages element should be connected consecutively.
3. In this multiple stage graph, there is a vertex whose in degree is **0** that is known as the source. And the vertex with only one out degree is **0** is known as the destination vertex.
4. 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** $|s_1| = |s_k| = 1$.
5. The vertex $s \in s_1$ is called **the source** and the **vertex** $t \in s_k$ **is called sink**.
6. *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.**
7. The multistage graph problem is finding the path with minimum cost from source *s* to sink *t*.

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

cost (i, j) = min {c (j, l) + cost (i + 1, l)}

l c Vi + 1

<j, l> c E

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

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.

```
1.  Algorithm Fgraph(G, k, n, p)
2.  // The input is a k-stage graph G = (V, E) with n vertices // indexed in order or
    stages. E is a set of edges and c [i, j] // is the cost of (i, j). p [1 : k] is a minimum
    cost path.
3.  {
4.  cost [n] := 0.0;
5.  for j:= n - 1 to 1 step – 1 do
6.  { // compute cost [j]
7.  let r be a vertex such that (j, r) is an edge of G and c [j, r] + cost [r] is minimum;
8.  cost [j] := c [j, r] + cost [r];
9.  d [j] := r:
10. }
11. // Find a minimum cost path.


12. p [1] := 1; p [k] := n;
13. for j := 2 to k - 1 do
14. p [j] := d [p [j - 1]];
15. }
```

The multistage graph problem can also be solved using the backward approach. Let bp(i, j) be a minimum cost path from vertex s to j vertex in Vi. Let Bcost(i, j) be the cost ofbp(i, j). From the backward approach we obtain:

Bcost (i, j) = min { Bcost (i –1, l) + c (l, j)}

l e Vi - 1

<l, j> e E

```
1. Algorithm Bgraph(G, k, n, p)
2. // Same function as Fgraph {
3. Bcost [1] := 0.0; for j := 2 to n do { // Compute Bcost [j].
4. Let r be such that (r, j) is an edge of
5. G and Bcost [r] + c [r, j] is minimum;
6. Bcost [j] := Bcost [r] + c [r, j];
7. D [j] := r;
8. } //find a minimum cost path
9. p [1] := 1; p [k] := n;
10. for j:= k - 1 to 2 do p [j] := d [p [j + 1]];
11. }
```

BACKWARD APPROACH:

We use the following equation to find the minimum cost path from t to s: Bcost (i, J) = min {Bcost (i – 1, l) + c (l, J)}

l c vi – 1

<l, j> c E

Bcost (5, 12) = min {Bcost (4, 9) + c (9, 12), Bcost (4, 10) + c (10, 12), Bcost (4, 11) + c (11, 12)}

= min {Bcost (4, 9) + 4, Bcost (4, 10) + 2, Bcost (4, 11) + 5}

Bcost (4, 9) = min {Bcost (3, 6) + c (6, 9), Bcost (3, 7) + c (7, 9)} = min {Bcost (3, 6) + 6, Bcost (3, 7) + 4}

Bcost (3, 6) = min {Bcost (2, 2) + c (2, 6), Bcost (2, 3) + c (3, 6)} = min {Bcost (2, 2) + 4, Bcost (2, 3) + 2} DESIGN AND ANALYSIS OF ALGORITHMS Page 61

Bcost (2, 2) = min {Bcost (1, 1) + c (1, 2)} = min {0 + 9} = 9 Bcost (2, 3) = min {Bcost (1, 1) + c (1, 3)} = min {0 + 7} = 7 Bcost (3, 6) = min {9 + 4, 7 + 2} = min {13, 9} = 9

Bcost (3, 7) = min {Bcost (2, 2) + c (2, 7), Bcost (2, 3) + c (3, 7), Bcost (2, 5) + c (5, 7)}

Bcost (2, 5) = min {Bcost (1, 1) + c (1, 5)} = 2

Bcost (3, 7) = min {9 + 2, 7 + 7, 2 + 11} = min {11, 14, 13} = 11 Bcost (4, 9) = min {9 + 6, 11 + 4} = min {15, 15} = 15

Bcost (4, 10) = min {Bcost (3, 6) + c (6, 10), Bcost (3, 7) + c (7, 10), Bcost (3, 8) + c (8, 10)}

Bcost (3, 8) = min {Bcost (2, 2) + c (2, 8), Bcost (2, 4) + c (4, 8), Bcost (2, 5) + c (5, 8)}

Bcost (2, 4) = min {Bcost (1, 1) + c (1, 4)} = 3

Bcost (3, 8) = min {9 + 1, 3 + 11, 2 + 8} = min {10, 14, 10} = 10 Bcost (4, 10) = min {9 + 5, 11 + 3, 10 + 5} = min {14, 14, 15) = 14

Bcost (4, 11) = min {Bcost (3, 8) + c (8, 11)} = min {Bcost (3, 8) + 6} = min {10 + 6} = 16

2.All-pairs Shortest Path

- In the all pairs shortest path problem, we are to find a shortest path between every pair of vertices in a directed graph G.
- That is, for every pair of vertices (i, j), we are to find a shortest path from i to j as well as one from j to i. These two paths are the same when G is undirected.

- The all pairs shortest path problem is to determine a matrix A such that A (i, j) is the length of a shortest path from i to j.
- The shortest i to j path in G, i ≠ j originates at vertex i and goes through some intermediate vertices (possibly none) and terminates at vertex j.
- If k is an intermediate vertex on this shortest path, then the subpaths from i to k and from k to j must be shortest paths from i to k and k to j, respectively.

Let Ak(i, j) represent the length of a shortest path from i to j going through no vertex of index greater than k, we obtain:

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

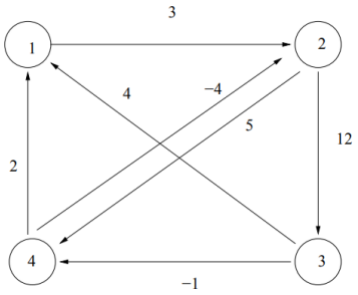
- $1<k<n$

Algorithm All Paths (Cost, A, n)

```
1.  // cost [1:n, 1:n] is the cost adjacency matrix of a graph which
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. }
```

Complexity Analysis: A Dynamic programming algorithm based on this recurrence involves in calculating n+1 matrices, each of size n x n. Therefore, the algorithm has a complexity of O (n3).

Example:



Solution

$$\begin{pmatrix} 0 & 3 & 15 & 8 \\ 7 & 0 & 12 & 5 \\ 1 & 4 & 0 & -1 \\ 2 & -4 & 8 & 0 \end{pmatrix}$$

$$D^0 = \begin{pmatrix} 0 & 3 & 0 & 0 \\ 0 & 0 & 12 & 5 \\ 4 & 0 & 0 & -1 \\ 2 & -4 & 0 & 0 \end{pmatrix} \quad D^1 = \begin{pmatrix} 0 & 3 & 0 & 0 \\ 0 & 0 & 12 & 5 \\ 4 & 7 & 0 & -1 \\ 2 & -4 & 0 & 0 \end{pmatrix} \quad D^2 = \begin{pmatrix} 0 & 3 & 15 & 8 \\ 0 & 0 & 12 & 5 \\ 4 & 7 & 0 & -1 \\ 2 & -4 & 8 & 0 \end{pmatrix}$$

$$D^3 = \begin{pmatrix} 0 & 3 & 15 & 8 \\ 16 & 0 & 12 & 5 \\ 4 & 7 & 0 & -1 \\ 2 & -4 & 8 & 0 \end{pmatrix} \quad D^4 = \begin{pmatrix} 0 & 3 & 15 & 8 \\ 7 & 0 & 12 & 5 \\ 1 & -5 & 0 & -1 \\ 2 & -4 & 8 & 0 \end{pmatrix}$$

3.Single Source Shortest Path Algoritham

Graphs can be used to represent the highway structure of a state or country with vertices representing cities and edges representing sections of highway.

 The edges have assigned weights which may be either the distance between the 2 cities connected by the edge or the average time to drive along that section of highway.

 For example if A motorist wishing to drive from city A to B then we must answer the following questions

- Is there a path from A to B
- If there is more than one path from A to B which is the shortest path

 The length of a path is defined to be the sum of the weights of the edges on that path.

Given a directed graph G(V,E) with weight edge w(u,v). e have to find a shortest path from source vertex S∈v to every other vertex v1∈ v-s.

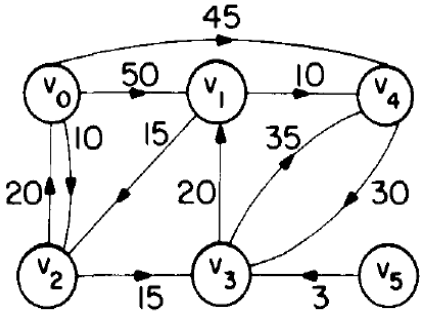
To find SSSP for directed graphs G(V,E) there are two different algorithms.

 Bellman-Ford Algorithm

 Dijkstra's algorithm

 Bellman-Ford Algorithm:- allow –ve weight edges in input graph. This algorithm either finds a shortest path form source vertex S∈V to other vertex v∈V or detect a –ve weight cycles in G, hence no solution. If there is no negative weight cycles are reachable form source vertex S∈V to every other vertex v∈V

 Dijkstra's algorithm:- allows only +ve weight edges in the input graph and finds a shortest path from source vertex S∈V to every other vertex v∈V.



	Path	Length
1)	$v_0\, v_2$	10
2)	$v_0\, v_2\, v_3$	25
3)	$v_0\, v_2\, v_3\, v_1$	45
4)	$v_0\, v_4$	45

Graph and shortest paths from v_0 to all destinations

Consider the above directed graph, if node 1 is the source vertex, then shortest path from 1 to 2 is 1,4,5,2. The length is 10+15+20=45.

 To formulate a greedy based algorithm to generate the shortest paths, we must conceive of a multistage solution to the problem and also of an optimization measure.

 This is possible by building the shortest paths one by one.

 As an optimization measure we can use the sum of the lengths of all paths so far generated.

 If we have already constructed 'i' shortest paths, then using this optimization measure, the next path to be constructed should be the next shortest minimum length path.

 The greedy way to generate the shortest paths from Vo to the remaining vertices is to generate these paths in non-decreasing order of path length.

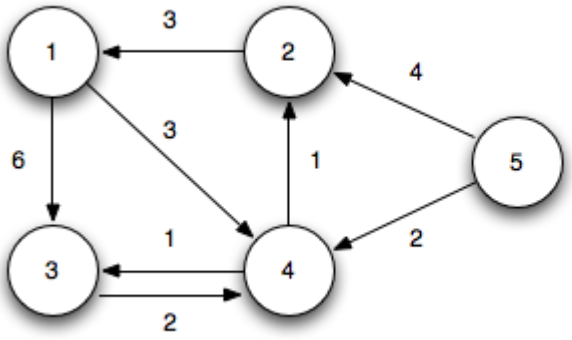
 For this 1st, a shortest path of the nearest vertex is generated. Then a shortest path to the 2nd nearest vertex is generated and so on.

Algorithm for finding Shortest Path

1. Algorithm ShortestPath(v, cost, dist, n)
2. //dist[j], 1≤j≤n, is set to the length of the shortest path from vertex v to vertex j in graph g with n-vertices.
3. // dist[v] is zero
4. { for i=1 to n do{
5. s[i]=false;
6. dist[i]=cost[v,i];
7. }
8. s[v]=true;
9. dist[v]:=0.0; // put v in s
10. for num=2 to n do{
11. // determine n-1 paths from v
12. choose u form among those vertices not in s such that dist[u] is minimum.
13. s[u]=true; // put u in s
14. for (each w adjacent to u with s[w]=false) do
15. if(dist[w]>(dist[u]+cost[u, w])) then
16. dist[w]=dist[u]+cost[u, w];
17. }
18. }

Example

Given the following directed graph



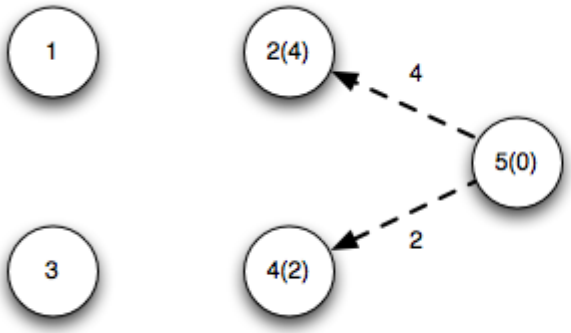
(1,3) = 6
(1.4) = 3
(2,1) = 3
(3,4) = 2
(4,2) = 1
(4,3) = 1
(5,2) = 4
(5,4) = 2

Using vertex 5 as the source (setting its distance to 0), we initialize all the other distances to ∞.

(1,3) = 6
(1.4) = 3
(2,1) = 3
(3,4) = 2
(4,2) = 1
(4,3) = 1
(5,2) = 4
(5,4) = 2

	1	2	3	4	5
d	∞	∞	∞	∞	**0**
π	/	/	/	/	/

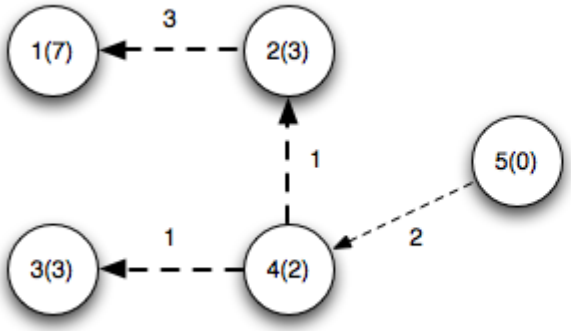
teration 1: Edges (u_5,u_2) and (u_5,u_4) relax updating the distances to 2 and 4



(1,3) = 6
(1.4) = 3
(2,1) = 3
(3,4) = 2
(4,2) = 1
(4,3) = 1
(5,2) = 4
(5,4) = 2

	1	2	3	4	5
d	∞	**4**	∞	**2**	0
π	/	**5**	/	**5**	/

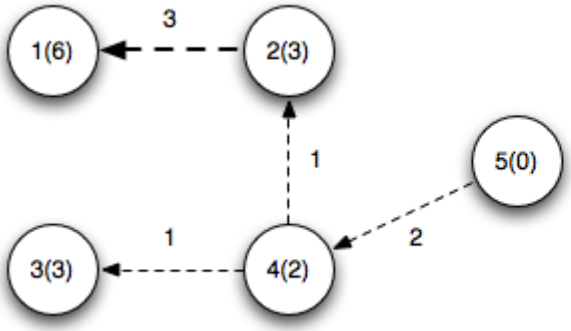
Iteration 2: Edges (u_2,u_1), (u_4,u_2) and (u_4,u_3) relax updating the distances to 1, 2, and 4 respectively. Note edge (u_4,u_2) finds a shorter path to vertex 2 by going through vertex 4



(1,3) = 6
(1.4) = 3
(2,1) = 3
(3,4) = 2
(4,2) = 1
(4,3) = 1
(5,2) = 4
(5,4) = 2

	1	2	3	4	5
d	**7**	**3**	**3**	2	0
π	**2**	**4**	**4**	5	/

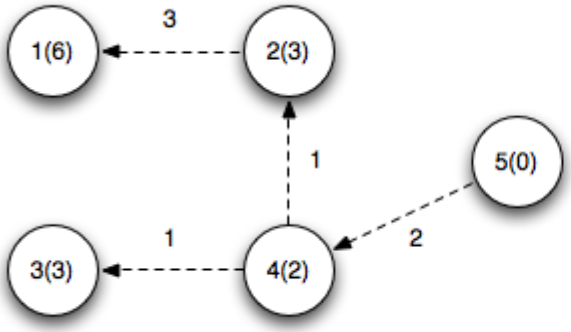
Iteration 3: Edge (u_2,u_1) relaxes (since a shorter path to vertex 2 was found in the previous iteration) updating the distance to 1



(1,3) = 6
(1.4) = 3
(2,1) = 3
(3,4) = 2
(4,2) = 1
(4,3) = 1
(5,2) = 4
(5,4) = 2

	1	2	3	4	5
d	**6**	3	3	2	0
π	**2**	4	4	5	/

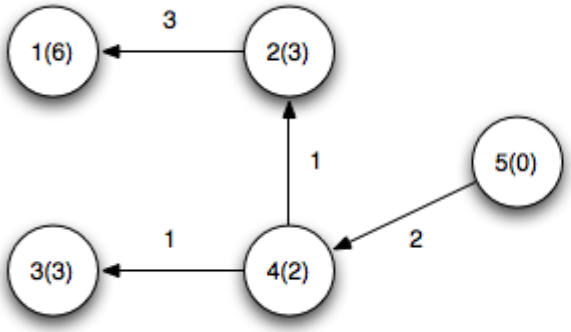
Iteration 4: No edges relax



(1,3) = 6
(1.4) = 3
(2,1) = 3
(3,4) = 2
(4,2) = 1
(4,3) = 1
(5,2) = 4
(5,4) = 2

	1	2	3	4	5
d	6	3	3	2	0
π	2	4	4	5	/

The final shortest paths from vertex 5 with corresponding distances is



	1	2	3	4	5
d	6	3	3	2	0
π	2	4	4	5	/

Complexity

Algorithm	**Time complexity**
Bellman–Ford **algorithm**	O(VE)
	$O(V^2 \log V)$

4.Traveling sales person problem

Travelling Salesman Problem (TSP): Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

- 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.
- 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 Є {1, 2, 3, ... , n}*** that includes ***1***, and ***j Є 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

$$C(S,j)=\min C(S-\{j\},i)+d(i,j) \text{ where } i\in S \text{ and } i\neq j$$

Algorithm: Traveling-Salesman-Problem

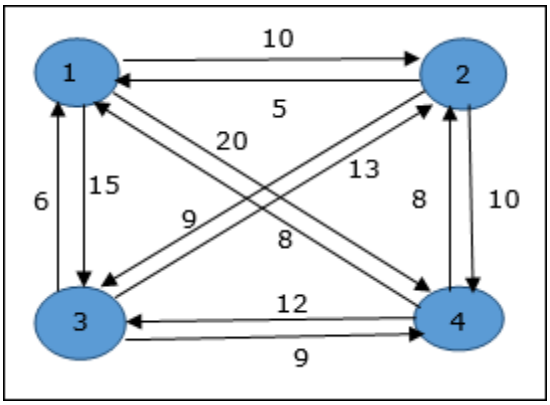
1. C ({1}, 1) = 0
2. for s = 2 to n do
3. for all subsets S Є {1, 2, 3, … , n} of size s and containing 1
4. C (S, 1) = ∞
5. for all j Є S and j ≠ 1
6. C (S, j) = min {C (S – {j}, i) + d(i, j) for i Є S and i ≠ j}
7. Return minj C ({1, 2, 3, …, n}, j) + d(j, i)

Analysis

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

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

S = Φ

Cost(2,Φ,1)=d(2,1)=5

Cost(2,Φ,1)=d(2,1)=5

Cost(3,Φ,1)=d(3,1)=6

Cost(3,Φ,1)=d(3,1)=6

Cost(4,Φ,1)=d(4,1)=8

Cost(4,Φ,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,Φ,1)=9+6=15

Cost(2,{4},1)=d[2,4]+Cost(4,Φ,1)=10+8=18

Cost(3,{2},1)=d[3,2]+Cost(2,Φ,1)=13+5=18

Cost(3,{4},1)=d[3,4]+Cost(4,Φ,1)=12+8=20

Cost(4,{3},1)=d[4,3]+Cost(3,Φ,1)=9+6=15

Cost(4,{2},1)=d[4,2]+Cost(2,Φ,1)=8+5=13

S = 2

Cost(2,{3,4},1)={d[2,3]+Cost(3,{4},1)=9+20=29d[2,4]+Cost(4,{3},1)=10+15=25

Cost(3,{2,4},1)={d[3,2]+Cost(2,{4},1)=13+18=31d[3,4]+Cost(4,{2},1)=12+13=25

Cost(4,{2,3},1)={d[4,2]+Cost(2,{3},1)=8+15=23d[4,3]+Cost(3,{2},1)=9+18=27=

S = 3

Cost(1,{2,3,4},1)= d[1,2]+Cost(2,{3,4},1)=10+25=35

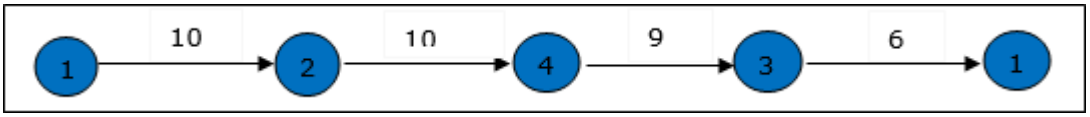
d[1,3]+Cost(3,{2,4},1)=15+25=40

d[1,4]+Cost(4,{2,3},1)=20+23=43=35

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, 2]** but 2 and 4 is already selected. Therefore, we select **d [4, 3]** (two possible values are 15 for d [2, 3] and d [4, 3], but our last node of the path is 4). Select path 4 to 3 (cost is 9), then go to **s = Φ** step. We get the minimum value for **d [3, 1]** (cost is 6).



5.0/1 Knapsack Problem

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack.

- In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively.
- Also given an integer W which represents knapsack capacity,
- find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W.

- You cannot break an item, either pick the complete item, or don't pick it (0-1 property).

Approach for Knapsack problem using Dynamic Programming

Here is an example input :

Weights : 2 3 3 4 6

Values : 1 2 5 9 4

Knapsack Capacity (W) = 10

From the above input, the capacity of the knapsack is 15 kgs and there are 5 items to choose from. Please note that there are no items with zero weight. A very simple idea is to evaluate each item and find out if it should be included in the knapsack or not in order to maximize the value.

Approach

Have a look at the below table and then I will explain the algorithm.

items at our disposal are 2(1), 3(2), 3(5) and 4(9), then the maximum value we can get by picking items from this list would be 10.

- One more time, cell (4, 10) has the value 16. This means that the maximum value we can get from items 2(1), 3(2), 3(5), 4(9) and 6(4) is 16 within the knapsack's weight limit of 10.

.Analysis

The algorithm requires an auxiliary space which is proportional to M * W where M is the length of the weights array. Hence, the space complexity is O(M*W).

The running time of the algorithm is also proportional to M * W because there are two loops and the outer one runs for M times and the inner one runs for W times. Hence the running time would be O(M*W).

EXAMPLE: SOLVING KNAPSACK PROBLEM WITH DYNAMIC PROGRAMMING Selection of n=4 items, capacity of knapsack M=8

Item i	Value v_i	Weight w_i
1	15	1
2	10	5
3	9	3
4	5	4

f(0,g) = 0, f(k,0) = 0

Recursion formula:

f(k,g) ={ f(k-1,g) if w > g

{max {v + f(k-1,g-w), f(k-1,g)} if wk<g and k>0}

Solution tabulated:

		Capacity remaining g=0	g=1	g=2	g=3	g=4	g=5	g=6	g=7	g=8
k=0	f(0,g) =	0	0	0	0	0	0	0	0	0
k=1	f(1,g) =	0	**15**	15	15	15	15	15	15	15
k=2	f(2,g) =	0	**15**	15	15	15	15	25	25	25
k=3	f(3,g) =	0	15	15	15	**24**	24	25	25	25
k=4	f(4, g) =	0	15	15	15	24	24	25	25	**29**

Last value: k=n, g=M

f = f(n,M) = f(4,8) = 29

max Backtracking the solution:

Repeat for k =n ,n-1 ,..., 1

If f(k,g) … f(k-1,g), item k is in the selection, x := 1. Otherwise, x := 0

Capacity for previous items: g := g - w x

g=8

k=4: f(4,8) … f(3,8) Y x = 1 4 g = g - w = 8-4 = 4

k=3 :f(3,4) … f(2,4)Y x = 1 3 g = g - w = 4-3 = 1

k=2, : f(2,1) = f(1,1) Y x = 0 2 g = g - 0 = 1

k=1 f(1,1) … f(0,1)Y x = 1 1 g = g - w = 1-1 = 0

The solution is x = (1,0,1,1) i.e. items 1,3, and 4 are selected. value of the knapsack is 29

Algorithm

```
KnapSack(v, w, n, W)
{
  for (w = 0 to W) V[0, w] = 0;
  for (i = 1 to n)
    for (w = 0 to W)
      if (w[i] ≤ w)
        V[i, w] = max{V[i − 1, w], v[i] + V[i − 1, w − w[i]]};
      else
        V[i, w] = V[i − 1, w];
  return V[n, W];
}
```

Time complexity: Clearly, O(nW).