Analysis of Algorithms
CSC 402
2022-23

Subject Incharge

Bidisha Roy

Associate Professor

Room No. 401

email: bidisharoy@sfit.ac.in

Dynamic Programming

Dynamic Programming

- An algorithm design method that can be used when the solution to a problem can be viewed as a sequence of decisions.
- "***Programming***" refers to *planning.*
- Final solution by combining the solution to sequence of subproblems.

Elements of Dynamic Programming (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 computed information.

Greedy Vs. Dynamic Programming

Greedy algorithm

- The best choice is made at each step and after that the subproblem is solved.
- The choice made by a greedy algorithm may depend on choices so far, but it *cannot depend on* any future choices or on the solutions to subproblems.
- A greedy strategy usually progresses in a *top-down* fashion, making one greedy choice after another, reducing each given problem instance to a smaller one

Dynamic programming

- A choice is made at each step.
- The choice made at each step usually *depends on* the solutions to subproblems.
- Dynamic-programming problems are often solved in a *bottom-up* manner.

Problems to be considered

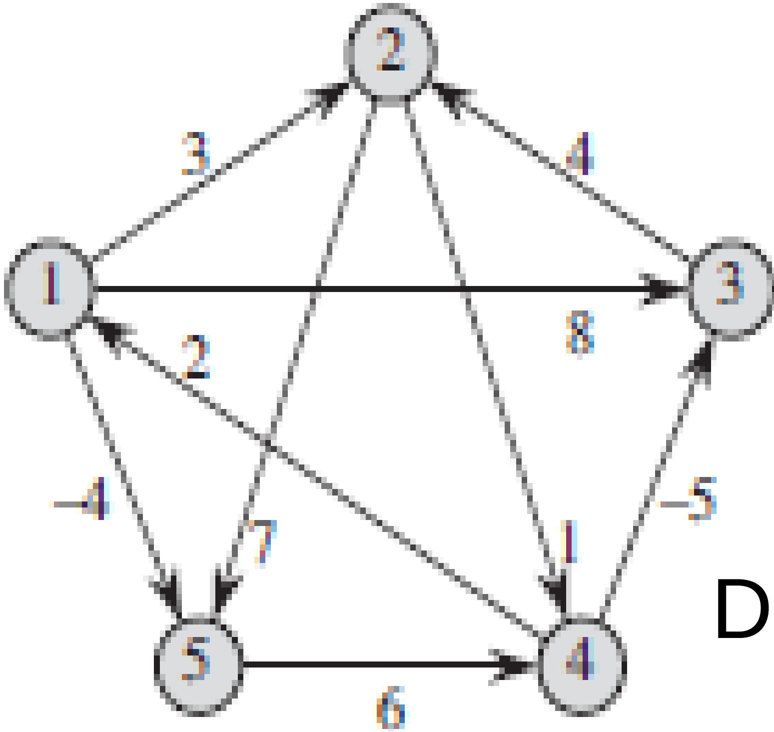
- All pairs shortest path
- Longest Common Subsequence
- Multistage Graphs
- Single source shortest path
- 0/1 knapsack problem
- TSP
- Assembly Line Scheduling

All pairs shortest paths

- Given a graph *G = (V,E)*, find the shortest path between every pair of vertices in the given graph.
- Dynamic Program Technique: Floyd-Warshall's Algorithm
  - Graph may contain negative edges but no negative weight cycles
  - A weight matrix W where
    W(i,j)=0 if i=j.
    W(i,j)=INFINITY if there is no edge between i and j.
    W(i,j)="weight of edge"

Floyd-Warshall's Algorithm

- Floyd-Warshal (W)
  - n ← rows(W)
  - $D^{(0)}$ ← W
  - for k ← 1 to n
    - do for i ← 1 to n
      - do for j ← 1 to n
        - » do $D_{(i,j)}^{k} \leftarrow \min (D_{(i,j)}^{(k-1)}, (D_{(i,k)}^{(k-1)} + D_{(k,j)}^{(k-1)})$
  - Return $D^{(n)}$
- Complexity $O(V^3)$.



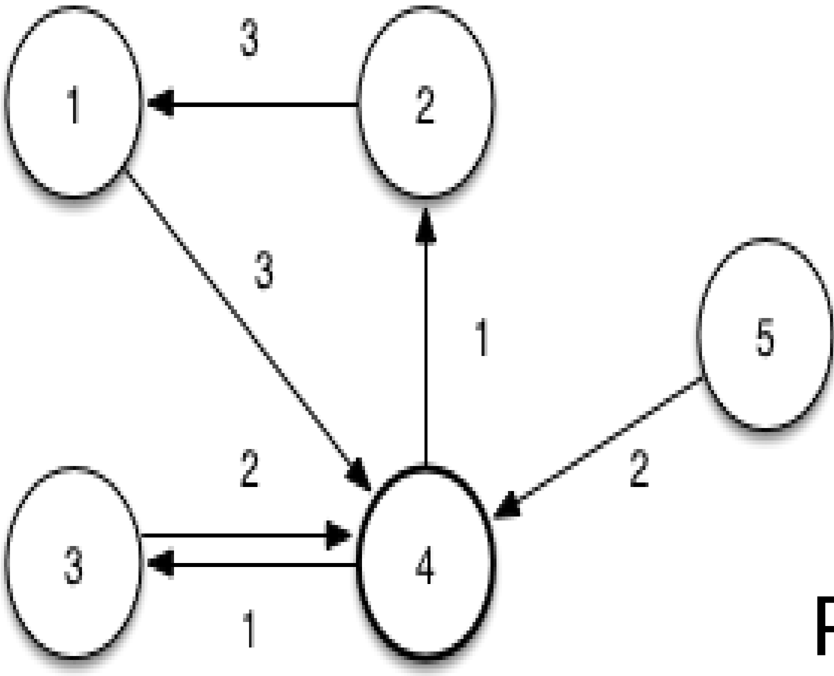
$$D^{(0)} = \begin{pmatrix} \mathbf{0} & \mathbf{3} & \mathbf{8} & \infty & \mathbf{-4} \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

Floyd-Warshall's Algorithm

- Path can be traced by simultaneously tracing a path matrix.
- Path matrix is constructed as follows:
  - If i=j || W[i][j]=∞
    - P[i][j]=0
  - If i != j && W[i][j]!= ∞
    - P[i][j] = i

Floyd-Warshall's Algorithm

- Code modified to trace path
- Floyd-Warshal (W)
  - n ← rows(W)
  - $D^{(0)}$ ← W
  - for k ← 1 to n
    - do for i ← 1 to n
      - do for j ← 1 to n
        - » If $(D_{(i,j)}^{(k-1)} > (D_{(i,k)}^{(k-1)} + D_{(k,j)}^{(k-1)})$
        - » do $D_{(i,j)}^{k} \leftarrow \min(D_{(i,j)}^{(k-1)}, (D_{(i,k)}^{(k-1)} + D_{(k,j)}^{(k-1)})$
        - » P[i][j]=k
  - Return $D^{(n)}$



$$D^{(0)} = \begin{pmatrix} 0 & \infty & \infty & 3 & \infty \\ 3 & 0 & \infty & \infty & \infty \\ \infty & \infty & 0 & 2 & \infty \\ \infty & 1 & 1 & 0 & \infty \\ \infty & \infty & \infty & 2 & 0 \end{pmatrix}$$

$$P^{(0)} = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 4 & 4 & 0 & 0 \\ 0 & 0 & 0 & 5 & 0 \end{pmatrix}$$

Longest Common Subsequence

- Substring and Subsequence
- A **substring** of a string S is another string S′ that occurs in S and all the letters are **contiguous** in S
- E.g. HelloWorld
- substring1 : Hello substring2 : World

- A **subsequence** of a string S is another string S′ that occurs in S and all the letters **need not to be contiguous** in S
- E.g HelloWorld
- subsequence1 : Herd subsequence2 : eow

Longest Common Subsequence

- The Longest Common Subsequence (LCS) problem is as follows:
- We are given two strings: string A of length x and string B of length y.

  We have to find the longest common subsequence :

  The longest sequence of characters that appear left-to-right in both strings.

Example, A= KASHMIR

B= CHANDIGARH

LCS has 3 length and string is **HIR**

Longest Common Subsequence

- Brute Force Method
  - Given two strings X of length m and Y of length n, find a longest subsequence common to both X and Y
  - STEP1 : Find all subsequences of 'X'. 2^m
  - STEP2: For each subsequence, find whether it is a subsequence of 'Y'. $n*2^m$
  - STEP3: Find the longest common subsequence from available subsequences. T.C= $O(n2^m)$
- **To improve time complexity, we use dynamic programming**

Longest Common Subsequence

- **Optimal substructure**

We have two strings

$$X= \{ x_1,x_2,x_3,\ldots\ldots,x_m\}$$

$$Y= \{y_1,y_2,y_3,\ldots\ldots\ldots,y_n\}$$

First compare x_m and y_n. If they matched, find the subsequence in the remaining string and then append the x_m with it.

If $x_m \neq y_n$,

- Remove x_m from X and find LCS from x_1 to x_{m-1} and y_1 to y_n
- Remove y_n from Y and find LCS from x_1 to x_m and y_1 to y_{n-1}

In each step, we reduce the size of the problem into the subproblems. It is optimal substructure.

Longest Common Subsequence

Recursive Equation

$X = \{ x_1, x_2, x_3, \ldots\ldots, x_m \}$

$Y = \{ y_1, y_2, y_3, \ldots\ldots\ldots, y_n \}$

– Let $c[i,j]$ be the length of an LCS of X_i & Y_j

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

Longest Common Subsequence

```
LCS-LENGTH(X, Y)
m ← length[X]
n ← length[Y]
for i ← 1 to m
    do c[i, 0] ← 0
for j ← 0 to n
    do c[0, j] ← 0
for i ← 1 to m
    do for j ← 1 to n
           do if x_i = y_j
                 then c[i, j] ← c[i − 1, j − 1] + 1
                      b[i, j] ← "↖"
                 else if c[i − 1, j] ≥ c[i, j − 1]
                         then c[i, j] ← c[i − 1, j]
                              b[i, j] ← "↑"
                         else c[i, j] ← c[i, j − 1]
                              b[i, j] ← "←"
return c and b
```

Input: Two sequences :

X= $\{ x_1, x_2, x_3, \ldots\ldots, x_m \}$

Y= $\{ y_1, y_2, y_3, \ldots\ldots\ldots, y_n \}$

For i=0..m and j=0..n
C[i,j] holds the LCS of sequences X_i and Y_j

For i=1..m and j=1..n
B[i,j] points to table entries corresponding to optimal sub-problem.

Printing the Solution

```
PRINT-LCS(b, X, i, j)
if i == 0 or j == 0
    return
if b[i, j] == "↖"
    PRINT-LCS(b, X, i − 1, j − 1)
    print x_i
elseif b[i, j] == "↑"
    PRINT-LCS(b, X, i − 1, j)
else PRINT-LCS(b, X, i, j − 1)
```

Initial call:
Print_LCS(b, X, X.length, Y.length)

Analysis

- We have two nested loops
  - The outer one iterates m times
  - The inner one iterates n times
  - A constant amount of work is done inside each iteration of the inner loop
  - Thus, the total running time is $O(mn)$

Application

- Bioinformatics
  - DNA Matching of different organisms

    DNA comprises of {A,C,G,T}.

    DNA1= AGCCTCAGT

    DNA2=ATCCT

    DNA3=AGTAGC

    DNA 1 and DNA 3 are more similar.
- File Comparison
- Revision Control Systems (Git)
- Screen Redisplay

Single source Shortest Path

- Given a graph $G = (V,E)$, to find the shortest path from a given **source** vertex ***s*** ***є V*** to every other vertex ***v є V***.

- Algorithms to be considered in dynamic programming
  - Bellman-Ford Algorithm.

Bellman-Ford Algorithm

- Provides general solution to the SSSP problem.
  - i.e. edge weights may be negative.
- Given a weighted, directed graph $G = (V, E)$ with source ***s***, the Bellman-Ford Algorithm returns a boolean value indicating whether or not there is a negative weight cycle that is reachable from the source.

Bellman-Ford Algorithm

- Bellman-Ford (G, w, s)
  - INITIALIZE-SINGLE-SOURCE(G, s)
  - for i ← 1 to |V[G]| -1
    - do for each edge (u, v) ε E[G]
      - do RELAX (u, v, w)
  - for each edge (u, v) ε E[G]
    - do if d[v] > d[u] + w(u, v)
      - then return FALSE
  - return TRUE

INITIALIZE-SINGLE-SOURCE(G, s)

for each vertex $v \in V[G]$
  do $d[v] \leftarrow \infty$
    $\pi[v] \leftarrow$ NIL
$d[s] \leftarrow 0$

RELAX(u, v, w)

if $d[v] > d[u] + w(u, v)$
  then $d[v] \leftarrow d[u] + w(u, v)$
    $\pi[v] \leftarrow u$

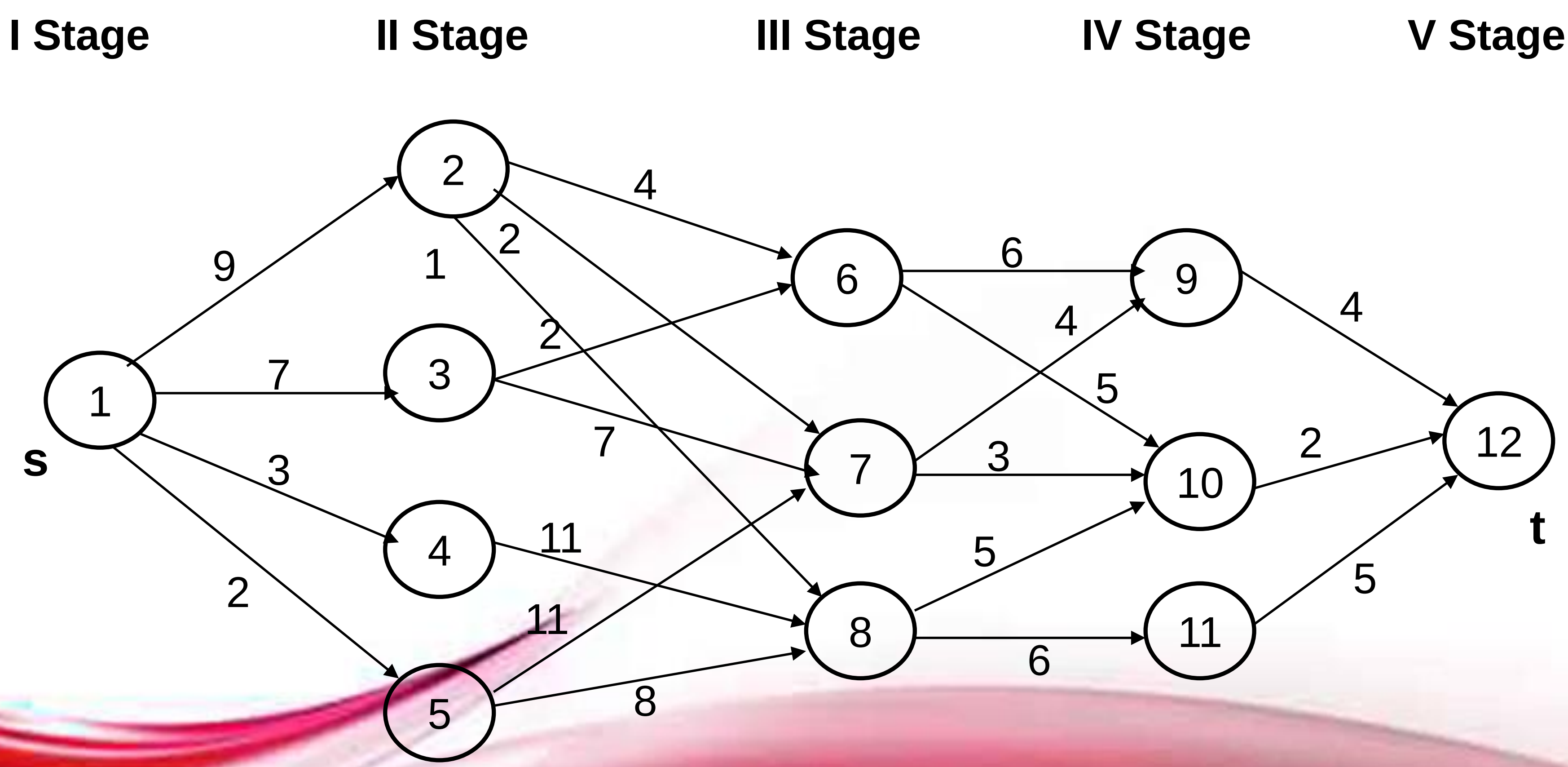
Bellman-Ford Algorithm

- Analysis
  - The for loop executes O(V) times and each time checks for |E| edges, i.e. O(VE)
- Applications
  - Variants of Bellman-Ford Algorithm used in routing protocols like Routing Information Protocol (RIP).

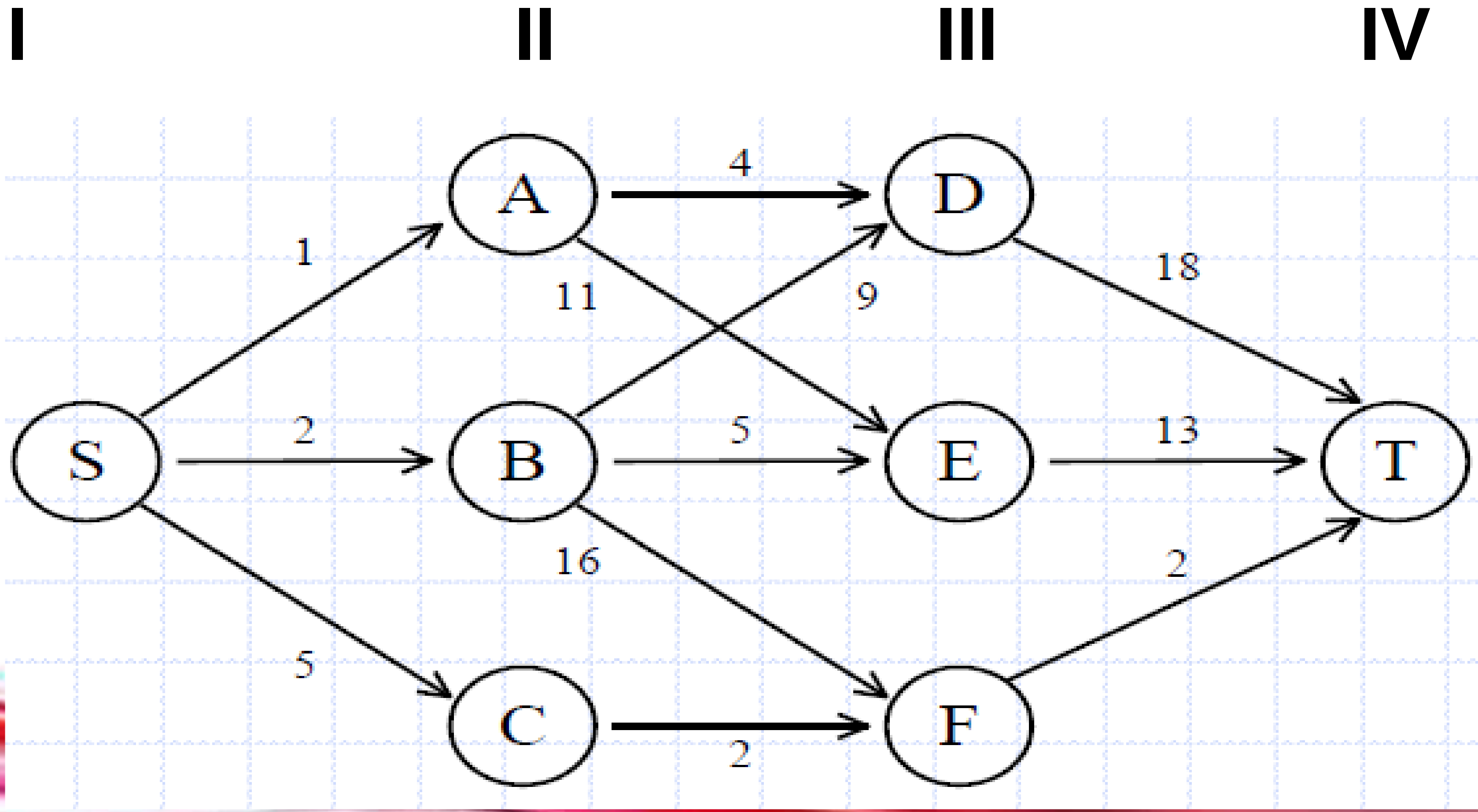
Multistage Graphs

- A multistage graph *G = (V, E)* is a directed graph in which vertices are partitioned into $k \geq 2$ disjoint sets V_i, $1 \leq i \leq k$.
- In addition if *(u, v)* is an edge in E, then $u \in V_i$ and $v \in V_{i+1}$.
- Sets V_1 and V_k are such that $|V_1| = |V_k| = 1$.

Multistage Graphs



Multistage Graphs



Multistage Graphs

- Problem: to find the shortest path between source and sink
- Can be solved using
  - Forward Approach
  - Backward Approach
- For algorithm for both the approaches refer… Fundamentals of Algorithms, Hurowitz, Rajasekaran, Sahani.

Forward Approach

- Solve backwards and trace the path forwards
  - $c(i,j)$ = cost/weight/distance of edge (i,j)
  - $cost(k,j)$ = cost of shortest path for node j in stage k to sink node t
    - E.g. cost(III,6) = cost of shortest path for node 6 in stage III to sink t
- Recursive formula:
  - $$cost(i,j) = \min_{\substack{(j,l) \in E \\ l \in V_{i+1}}} \{c(j,l) + cost(i+1,l)\}$$

Forward Approach

- Initial condition
  - cost(k, t) = 0
  - i.e. cost of shortest path from sink node t in last stage to itself is 0

Backward Approach

- bcost(I,j) = backward cost of node j in stage I to initial source node.
  - $bcost(i,j) = \min \{bcost(i-1,l) + c(l,j)\}$

$$\begin{matrix} (l,j) \in E \\ l \in V_{i-1} \end{matrix}$$

- Initial Condition
  - bcost(I,s) = 0
  - Cost of shortest path from node s to source node in stage I is 0

Applications

- Project Scheduling

0/1 Knapsack Problem

- A thief robbing a store finds *n* items; the i^{th} item is worth c_i cost units and weighs w_i weight units. The thief wants to take as valuable load as possible, but he can carry at most *W* weight units in his knapsack.
- **Which items should he take ???** is the 0-1 knapsack problem (each item can be taken or left)

0/1 Knapsack Problem

- Select objects out of ***n*** objects such that $\Sigma_{1\leq i\leq n} v_i x_i$ is maximum subject to $\Sigma_{1\leq i\leq n} w_i x_i \leq$ ***m*** where $[p_1, p_2, p_3, \ldots p_n]$ is the profit vector, x_i is either 0 or 1, $1\leq i\leq n$ and m is the capacity of knapsack.
- The principal of optimality holds if
  - $V[i,j]=\max\{V[i-1,j], p_i + V[i-1,j-w_i]\}$
  - V is the array of solution of subproblems of size j with i items.

0/1 Knapsack Problem

- Inputs:
  - Set of n items, with weights w_i and profits p_i
  - Knapsack capacity M
- Output: Array V which holds the solution
- Conditions
  - $V[i,j] = 0$ if $i=0$ or $j=0$
  - $V[i,j] = V[i-1,j]$ if $j<w_i$
  - $\text{Max}\ \{V[i-1,j]\ ,\ p_i + V[i-1,j-w_i]\}$ if $j \geq W$

DP_Knapsack(V,p,w,M,n)

- for i ← 1 to n do
  - V[i,0] ← 0
- for j ← 1 to M do
  - V[0,j] ← 0
- for i ← 1 to n do
  - for j ← 0 to M do
    - If w[i] ≤ j
      - V[i,j] ← max {V[i-1,j], p[i]+V[i-1, j-w[i]]}
    - Else
      - V[i,j] ← V[i-1,j]

Complexity: O(n*M)

Trace_Knapsack(w,p,V,M)

- SW ← ϕ //set of weights and profits
- SP ← ϕ // to be added to knapsack
- i←n, j←M
- while (j>0) do
  - if (V[i,j]==V[i-1,j]) then
    - i← i-1
  - else
    - SW ← SW + w[i]
    - SP ← SP + p[i]
    - j← j-w[i]
    - i←i-1

Additional Problems

- Some problems
  - $n=4$, $(w_1, w_2, w_3, w_4) = (1, 5, 3, 4)$, $(p_1, p_2, p_3, p_4) = (15, 10, 9, 5)$ and $m=8$
  - $n=3$, $(w_1, w_2, w_3) = (2, 3, 3)$, $(p_1, p_2, p_3) = (1, 2, 4)$ and $m=6$
  - $n=3$, $(w_1, w_2, w_3) = (2, 3, 4)$, $(p_1, p_2, p_3) = (1, 2, 5)$ and $m=6$.

0/1 Knapsack Problem

- Consider the following case

Weight & Cost	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
	0	0	0	0	0	0	0	0	0	0	0	0
w1=1,p1=1	0	1	1	1	1	1	1	1	1	1	1	1
w2=2,p2=6	0	1	6	7	7	7	7	7	7	7	7	7
w3=5,p3=18	0	1	6	7	7	18	19	24	25	25	25	25
w4=6,p4=22	0	1	6	7	7	18	22	24	28	29	29	40
w5=7,p5=28	0	1	6	7	7	18	22	28	29	34	35	40

0/1 Knapsack Problem

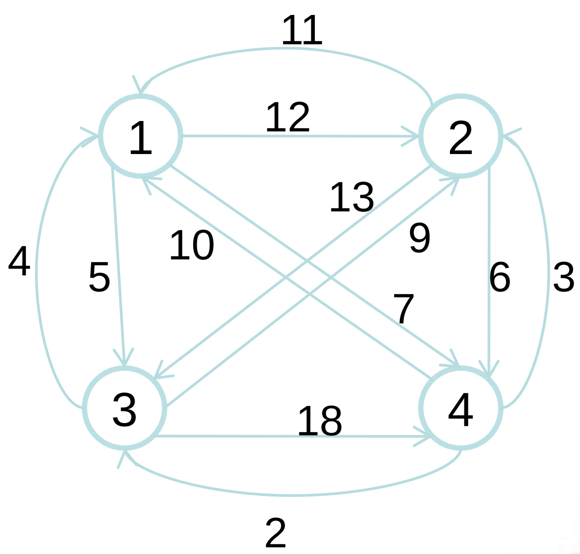
- Applications
  - Data Compression
  - Internet Download Managers
  - Resource Allocation

Travelling Salesman Problem

- Let ***G=(V, E)*** be a directed graph with n vertices and *w(u, v)* be the length/weight of edge (u, v).
- A path starting at a given vertex v1, going thru every vertex exactly once, and finally returning to v1 is called a tour
- Length of a tour is the sum of the lengths of the edges in the path defining the tour

Travelling Salesman Problem

Travelling Salesman Problem… contd



$$
\begin{array}{c c}
 & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 12 & 5 & 7 \\ 11 & 0 & 13 & 6 \\ 4 & 9 & 0 & 18 \\ 10 & 3 & 2 & 0 \end{bmatrix}
\end{array}
$$

Travelling Salesman Problem … contd

- TSP: Finding the tour of minimum length.
- Greedy method fails to find the optimum solution
  - Therefore dynamic programming for optimal solution is used.
- Logic: Every tour consists of an edge (1, k) for some *k ε V – {1, k}* and finally a return path from k to 1.
  - The path from k to 1 goes thru each vertex V – {1, k} exactly once.

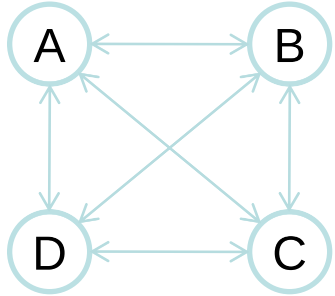
Travelling Salesman Problem… contd

- We calculate shortest path *sp(i, S)* → length of shortest path starting at vertex *i* and going thru all vertices in set S and terminating at 1.
- $sp(1, V-\{1\}) = \min_{2 \le k \le n}\{w_{1k} + sp(k, V-\{1,k\})\}$
- In general for i not belonging to S
  - $sp(i, S) = \min_{j \varepsilon S}\{w_{ij} + sp(j, S - \{j\})\}$
- Initial condition (initially S = Φ)
  - $sp(i, \Phi) = w_{i1}$

Travelling Salesman Problem… contd

- Time complexity
  - Let N be the number of sp(i,S) that have to be computed before general equation can be used to compute sp(1,V-{1})
    - N = $\sum_{k=0}^{n-2}(n-1)\binom{n-2}{k} = (n-1)2^{n-2}$
  - Total time = $O(n^2 2^n)$
  - This is better than enumerating all n! different tours

Travelling Salesman Problem… contd

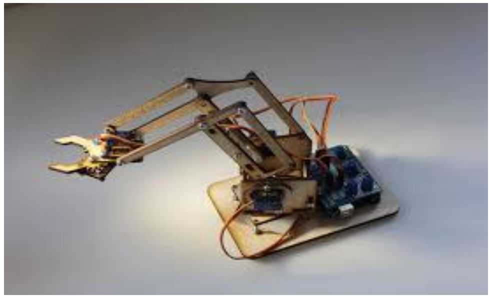


$$
\begin{array}{c c}
 & \begin{array}{cccc} A & B & C & D \end{array} \\
\begin{array}{c} A \\ B \\ C \\ D \end{array} & \begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 5 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}
\end{array}
$$

Applications of TSP

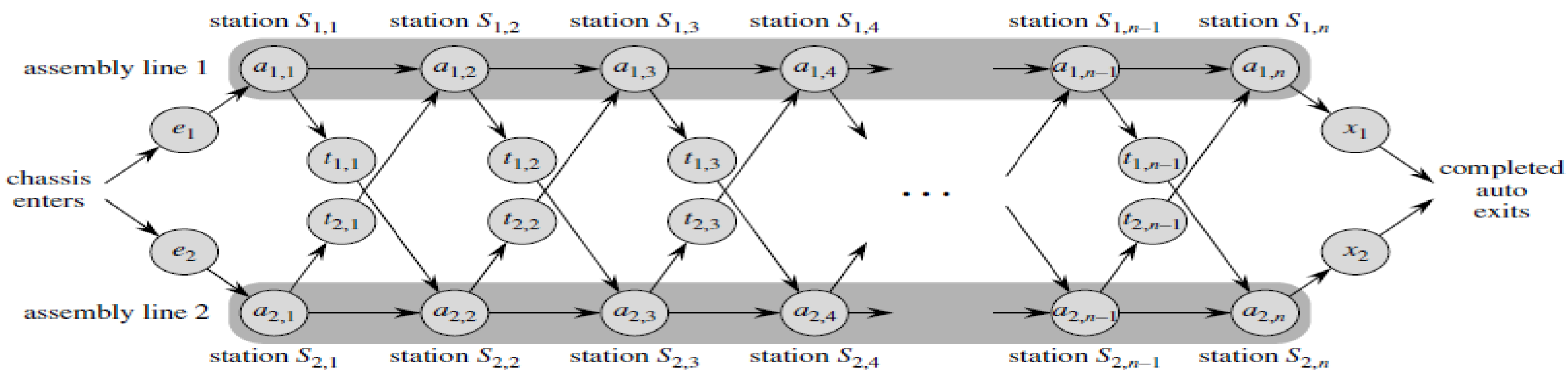
- Route a **postal van** to pick up mail from mail boxes located at n different sites
- In the manufacture of a **circuit board**, it is important to determine the best order in which a laser will drill thousands of holes.

Applications of TSP



- Using a **robot arm** to tighten the nuts on some piece of machinery on an assembly line.
  - The arm will start from its initial position, successively move to each of the remaining nuts and return to the initial position.
  - The path of the arm is a tour on a graph in which vertices represent the nuts.
  - A minimum cost tour will minimize the time needed for the arm to complete its task.

Assembly Line Scheduling



- There are two assembly lines, each with n stations; the j th station on line i is denoted $S_{i,\,j}$ and the assembly time at that station is $a_{i,j}$.
- An automobile chassis enters the factory, and goes onto line i (where $i = 1$ or 2), taking e_i time. After going through the j th station on a line, the chassis goes on to the $(j+1)^{st}$ station on either line. There is no transfer cost if it stays on the same line, but it takes time $t_{i,j}$ to transfer to the other line after station $S_{i,j}$.
- After exiting the n^{th} station on a line, it takes x_i time for the completed auto to exit the factory.
- The problem is to determine which stations to choose from line 1 and which to choose from line 2 in order to minimize the total time through the factory for one auto.

Other Problems

- Chain Matrix Multiplication
- Flowshop Scheduling