Ch 5
Dynamic programming

5.1 general method

Concept

- Dynamic programming is a kind of design method
- The concept is to take the process of solving a problem to be a sequence of decision making → then, find the best one as the final result

- If the decision can obtain the optimal solution
  - → the decision process satisfy the principle of optimality
  - → the way to the optimal solution (i.e., the middle product) of the optimal result is also optimal

Example 5.5: shortest path

- Give a graph. Each edge on the graph has a weight value.
- Assume that we can find a shortest path from node i to node j as $i, i_1, i_2, \ldots, i_k, j$
- Based on the principle of optimality the path $i_1, i_2, \ldots, i_k, j$ is also the shortest path from node i_1 to j

Proof of Example 5.5

- Assume that $i_1, i_2, \ldots, i_k, j$ is not the shortest path
- We can find another path $i_1, r_1, r_2, \ldots, r_k, j$, and this path is shorter than $i_1, i_2, \ldots, i_k, j$
- Which implies the path $i, i_1, i_2, \ldots, i_k, j$ is not optimal
- → contradiction, so $i, i_1, i_2, \ldots, i_k, j$ is optimal

Example 5.6: 0/1 knapsack

- Select n items to put in the knapsack. Those items cannot be divided. (w_i is the weight of i, p_i is the profit of i)

$$
\begin{gathered}
\text{maximize} \sum_{l \le i \le j} p_i x_i \\
\text{subject to} \sum_{l \le i \le j} w_i x_i \le y \\
x_i = 0 \text{ or } 1, \; l \le i \le j
\end{gathered}
$$

- We can use KNAP(l, j, y) to express the above problem
  - l: start of items
  - j: end of items
  - y: capacity

Example 5.6: 0/1 knapsack

- Let $y_1, y_2, \ldots, y_n$ is an optimal 0/1 series to express if the items $x_1, x_2, \ldots, x_n$ are selected
  - For example: when $y_1 = 1 \rightarrow x_1$ is selected, when $y_1 = 0 \rightarrow x_1$ is not selected

- If $y_1 = 0 \rightarrow$ this implies $y_2, y_3, \ldots, y_n$ must be the optimal solution of KNAP(2, n, m); otherwise, the $y_1, y_2, \ldots, y_n$ is not optimal

- If $y_1 = 1 \rightarrow$ this implies $y_2, y_3, \ldots, y_n$ must be the optimal solution of KNAP(2, n, m-w_1); otherwise, we can find another series $z_2, z_3, \ldots, z_n$ that can obtain more profit

Example 5.7

- If node k is a neighbor of node i and node k has the shortest path to node j
  - → The shortest path from node i to node j will include the shortest path from node k to node j

- Example 5.9 → extend to general case
  - If the shortest path from i to j (i, i_1, …, k, p_1, p_2, …, j) includes a middle node k
  - → (i, i_1, …, k) is the shortest path from i to k
  - → (k, p_1, p_2, …, j) is the shortest path from k to j

Example 5.8

- Let $g_j(y)$ represent an optimal solution of KNAP(j+1, n, y)
- → $g_0(m)$ is the optimal solution of KNAP(1, n, m)

- → $g_0(m) = \max \{ g_1(m), g_1(m-w_1) + p_1 \}$

Example 5.10/5.13

- Forward
  - $g_i(y) = \max\{ g_{i+1}(y), g_{i+1}(y-w_{i+1}) + p_{i+1} \}$
  - $g_n(y)=0$ for all $y>=0$ and $g_n(y)=-\inf$ for $y<0$

- Backward
  - $f_i(y) = \max\{ f_{j-1}(y), f_{j-1}(y-w_j) + p_j \}$
  - $f_0(y)=0$ for all $y>=0$ and $f_0(y)=-\inf$ for $y<0$

Example 5.11 (0/1 knapsack)

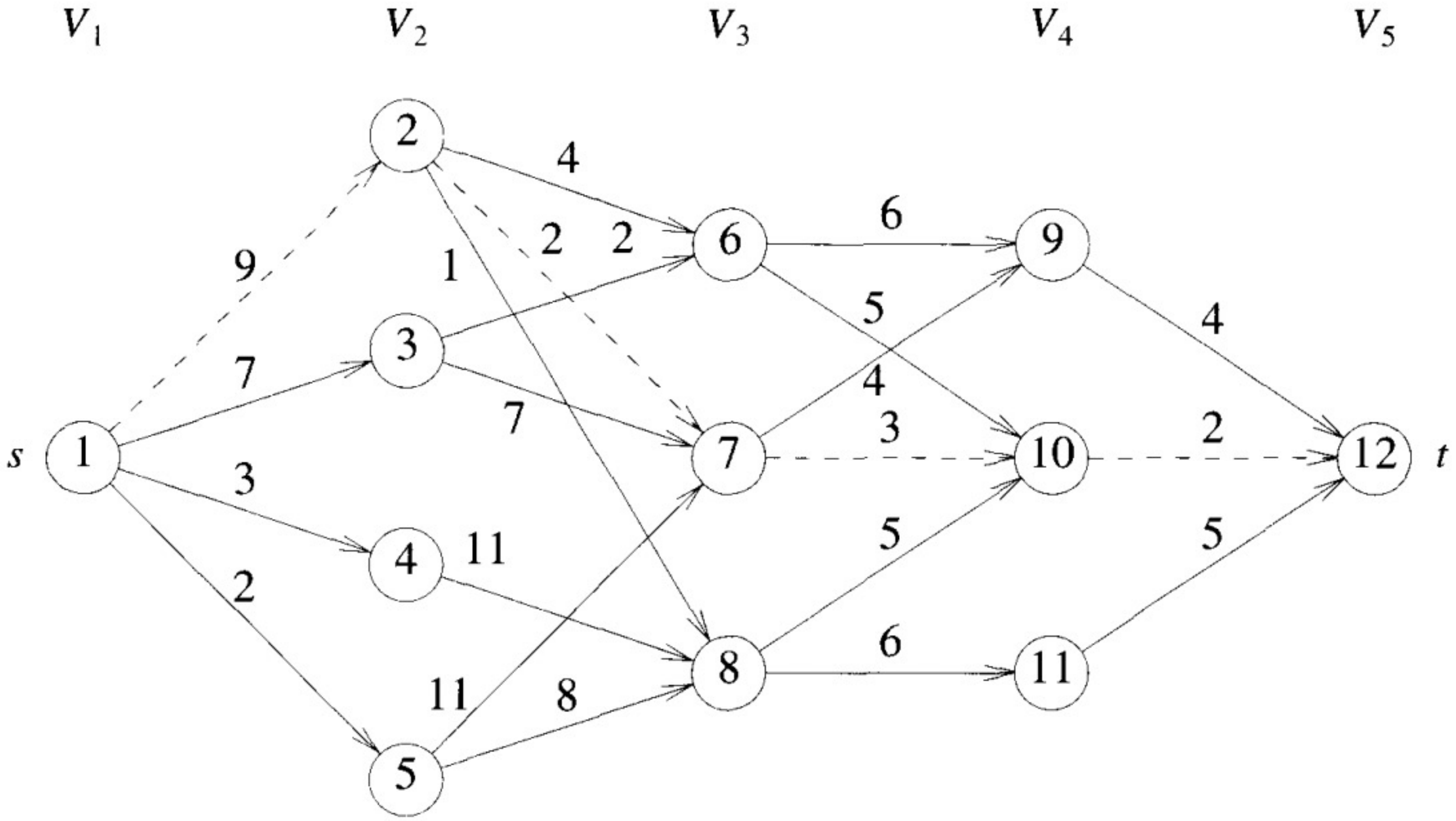
- Assume that
  - n = 3, and m=6 (capacity)
  - w_1=2, w_2=3, w_3=4
  - p_1=1, p_2=2, p_3=5

- To took forward, we aim to obtain $g_0(6)$
- → $g_0(6) = \max\{g_1(6), g_1(6-w_1) + p_1\}$
  $= \max\{g_1(6), g_1(4) + 1\}$

5.2 Multi-stage graph

Model

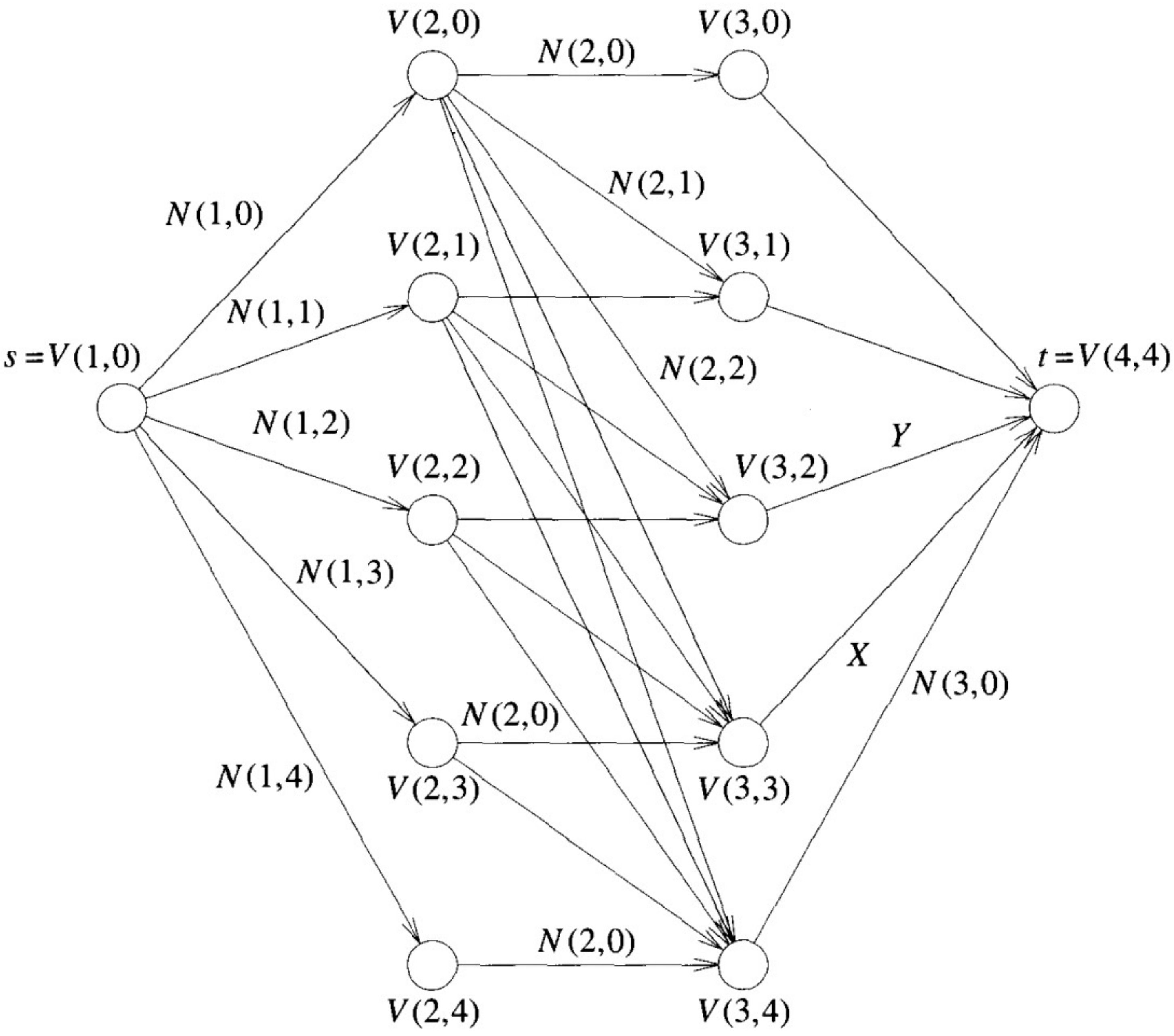
- A multi-stage graph G=(V, E) is a directed graph
- All vertices can be divided as k disjoint sets
- Set v_1 contains one node → as source s
- Set v_k contains one node → as sink t
- The terminals of each edge will be located in different sets
- Each edge has a cost value → ex: the cost of edge (i, j) is c(i, j)
- The multi-stage graph problem is to find a minimum cost path from s to t


V_1
V_2
V_3
V_4
V_5
s
t
1
2
3
4
5
6
7
8
9
10
11
12
9
7
3
2
4
2
2
1
7
11
11
8
6
5
4
3
5
6
4
2
5

A practical application

- There are n resources for r projects
- If a project i has j resources, the profit will be N(i, j)
- → can be expressed as a (r+1) stage graph problem

- 4 resources
- 3 projects



$X = \max\{N(3,0), N(3,1)\}$

$Y = \max\{N(3,0), N(3,1), N(3,2)\}$

Solve by dynamic programming

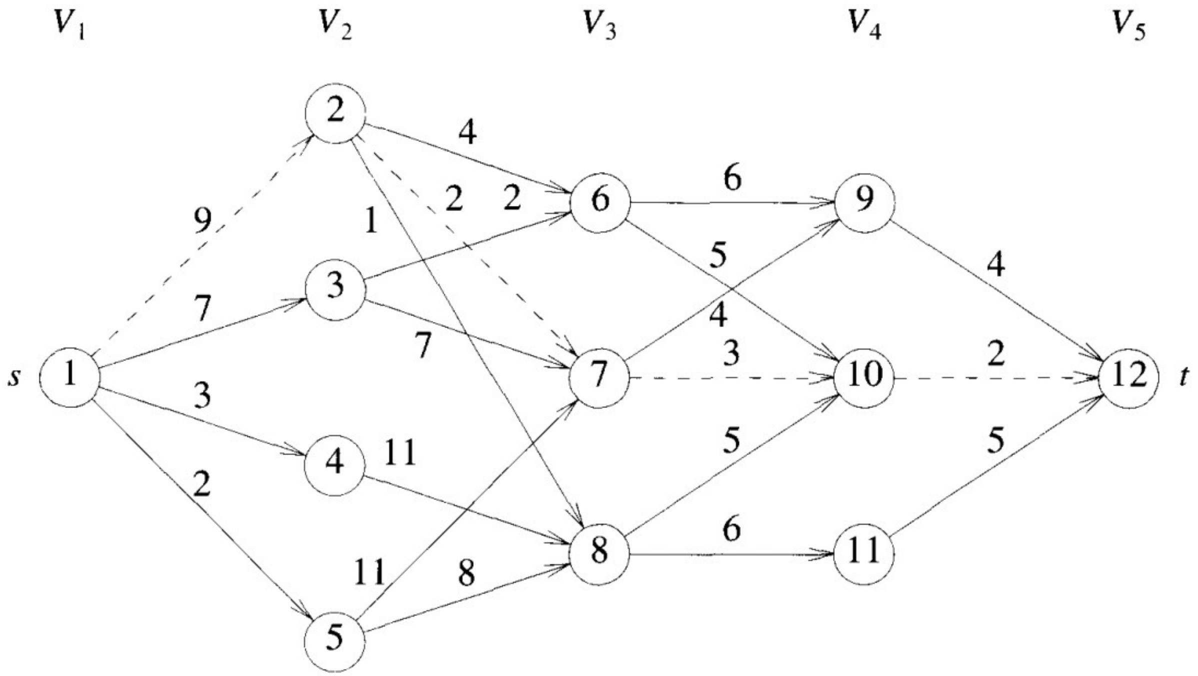
- The i-th decision will decide to use which vertex in V_{i+1} set
  - p(i, j) represent a shortest path from vertex j (in set V_i) to the sink
  - cost(i, j) is the cost of path p(i, j)→ **i** is i-th set V_i, **j** is j-th node in V_i

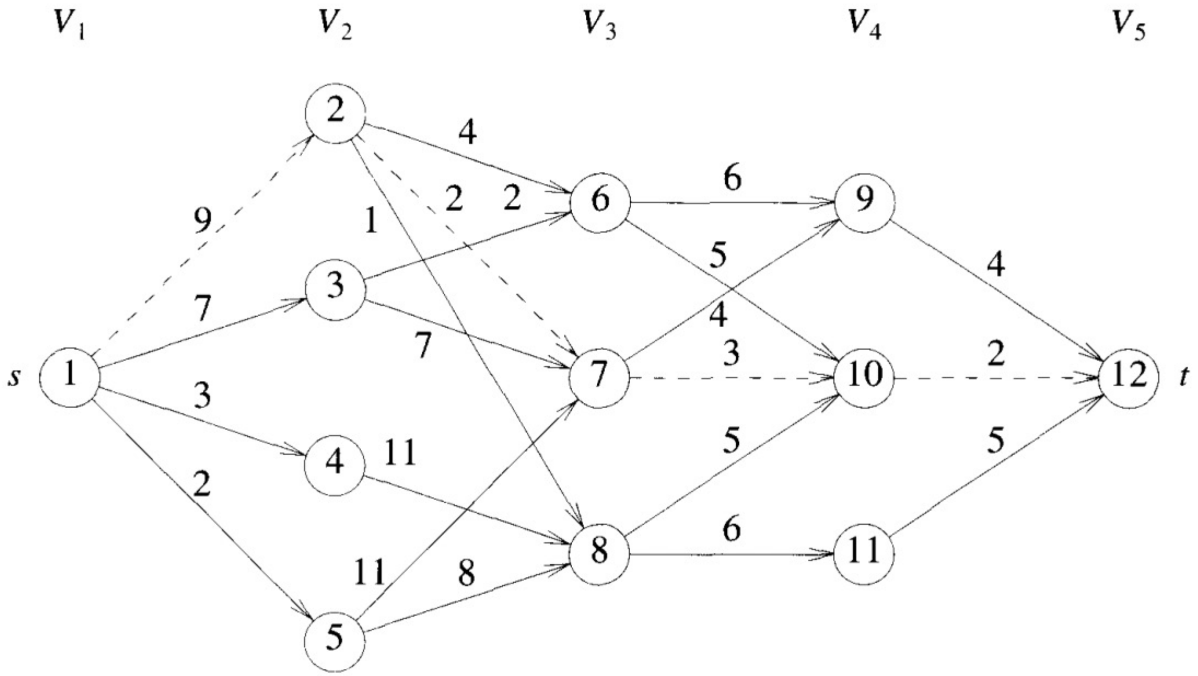
$$cost(i,j) = \min_{\substack{l \in V_{i+1} \\ \langle j,l \rangle \in E}} \{c(j,l) + cost(i+1,l)\}$$

  - cost(k-1, j) = c(j, t) if link (j, t) in E → the cost in the last stage will be the link cost to the sink
  - cost(k-1, j) = inf if link (j, t) not in E

Strategy

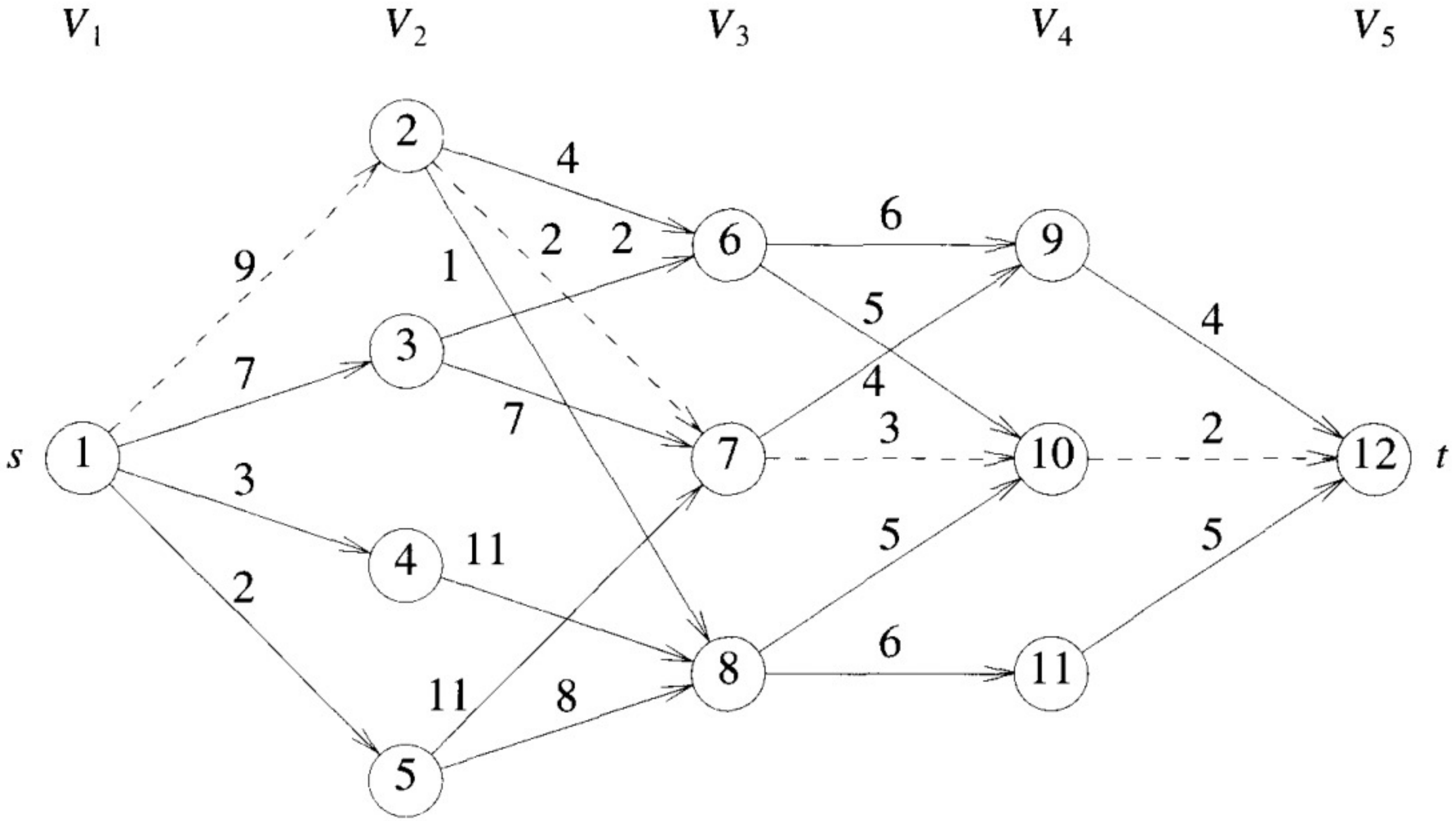
- Start from the last stage
  - → calculate cost(k-2, j), for all j in V_{k-2}
  - → calculate cost(k-2, j), for all j in V_{k-3}
  - .....
  - → cost(1, s)




V_1
V_2
V_3
V_4
V_5
s
t
1
2
3
4
5
6
7
8
9
10
11
12
9
7
3
2
4
1
2
2
7
11
11
8
6
5
4
3
5
6
4
2
5

Path recorder

- After obtaining the final cost value, we can record the process of calculating path
- Use d(i, j) to record the process
  - Ex: d(3, 6) = 10 → A shortest path is by the node 6 in stage 3 going through node 10



$$
\begin{array}{llllllllllll}
d(3,6) & = & 10; & d(3,7) & = & 10; & d(3,8) & = & 10; & & & \\
d(2,2) & = & 7; & d(2,3) & = & 6; & d(2,4) & = & 8; & d(2,5) & = & 8; \\
d(1,1) & = & 2 & & & & & & & & &
\end{array}
$$

Let the minimum-cost path be $s = 1, v_2, v_3, \ldots, v_{k-1}, t$. It is easy to see that $v_2 = d(1,1) = 2, v_3 = d(2, d(1,1)) = 7$, and $v_4 = d(3, d(2,\ d(1,1))) = d(3,7) = 10$.

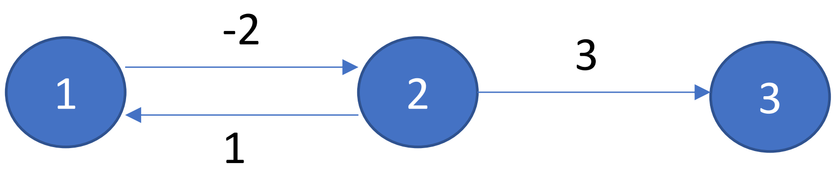
- In the pseudocode cost(i, j) → cost(j)

```
Algorithm FGraph(G, k, n, p)
// The input is a k-stage graph G = (V, E) with n vertices
// indexed in order of 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.
{
    cost[n] := 0.0;
    for j := n − 1 to 1  step −1 do
    { // Compute cost[j].
        Let r be a vertex such that ⟨j, r⟩ is an edge
        of G and c[j, r] + cost[r] is minimum;
        cost[j] := c[j, r] + cost[r];
        d[j] := r;
    }
    // Find a minimum-cost path.
    p[1] := 1; p[k] := n;
    for j := 2 to k − 1 do p[j] := d[p[j − 1]];
}
```

5.3 All pair shortest path

Model

- Give a digraph G=(V, E). Each edge has a cost value.
- The target is to find the shortest paths from every node to all other nodes.
  - The G contains no cycle with negative values



The cost from 1 to 3 will be –inf. →
Because node 1 and node 2 can continuously route to decrease total cost

- The simplest way is to execute shortest path algorithm for n times

Design concept

- The principle of optimality can be preserved → solve by DP
- Give each node an index value
- Let $A^k(i, j)$ be the shortest path from i to j
  - The indices of the middle nodes are all smaller than k

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

From all possible middle nodes, select the smallest cost one

From i to k, the index of the middle node do not exceed k-1

The direct link from i to j

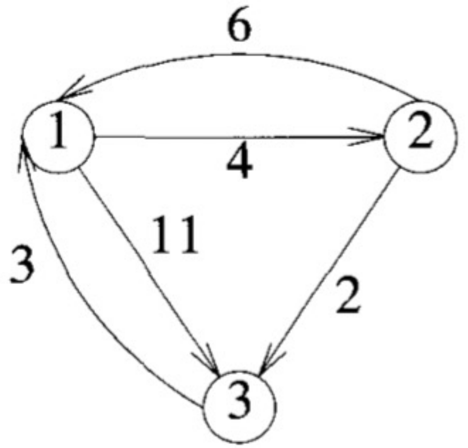
Design concept

- For a $A^k(i, j)$,
  - if the shortest path from i to j passed the node k →
    $A^k(i, j) = A^{k-1}(i, k) + A^{k-1}(k, j)$
  - Otherwise → $A^k (i, j) = A^{k-1}(i, j)$
- → So $A^k(i, j) = \min\{ A^{k-1}(i, j), A^{k-1}(i, k)+A^{k-1}(k, j) \}$

- To derive all pair shortest path → find $A^1(i, j)$→$A^2(i, j)$ …→$A^n(i, j)$
- Since the group has n nodes → $A(i, j) = A^n(i, j)$

```
Algorithm AllPaths(cost, A, n)
// cost[1 : n, 1 : n] is the cost adjacency matrix of a graph with
// n vertices; A[i, j] is the cost of a shortest path from vertex
// i to vertex j. cost[i, i] = 0.0, for 1 ≤ i ≤ n.
{
    for i := 1 to n do
        for j := 1 to n do
            A[i, j] := cost[i, j]; // Copy cost into A.
    for k := 1 to n do
        for i := 1 to n do
            for j := 1 to n do
                A[i, j] := min(A[i, j], A[i, k] + A[k, j]);
}
```

Algorithm 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

A^1	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

(c) A^1

A^2	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

(d) A^2

A^3	1	2	3
1	0	4	6
2	5	0	2
3	3	7	0

(e) A^3

5.4 Single source shortest path: general

Model

- The costs of edges can be negative (without negative loops)
- Let $dist^{l}[u]$ be the shortest path from v to u, and this path contains at most l edges
- Target: calculate $dist^{n-1}[u]$ for all u in G

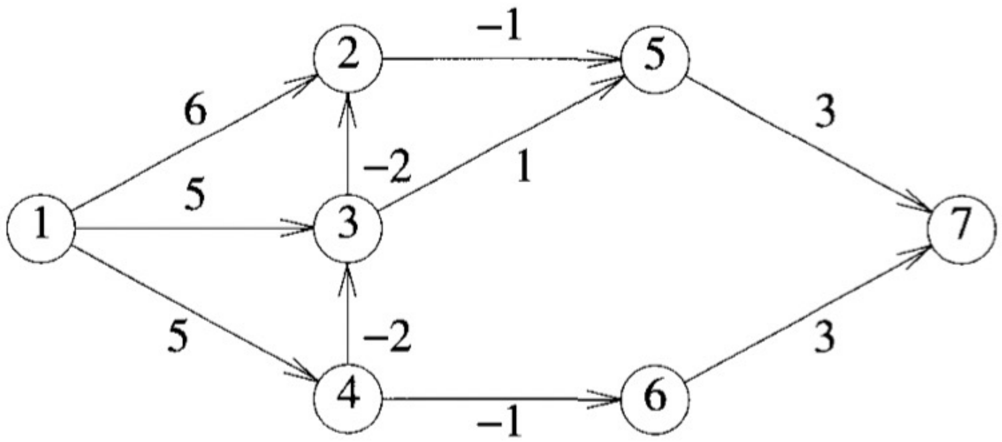
Observations

- If the shortest path from v to u contains at most k edges
  - The shortest path has at most k-1 edges → $dist^{k-1}[u] = dist^{k}[u]$
  - The shortest path has k edges → $dist^{k}[u] = dist^{k-1}[i] + cost[i,u]$, i.e.,

the path to u will pass through i

$dist^{k}[u] = \min \{dist^{k-1}[u] ,\ \min_{\{i\}} \{dist^{k-1}[i] + cost[i,u] \}\}$

Example



k	$dist^k[1..7]$						
	1	2	3	4	5	6	7
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	∞
3	0	1	3	5	2	4	7
4	0	1	3	5	0	4	5
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3

Bellman ford

```
Algorithm BellmanFord(v, cost, dist, n)
// Single-source/all-destinations shortest
// paths with negative edge costs
{
    for i := 1 to n do // Initialize dist.
        dist[i] := cost[v, i];
    for k := 2 to n − 1 do
        for each u such that u ≠ v and u has
                at least one incoming edge do
            for each ⟨i, u⟩ in the graph do
                if dist[u] > dist[i] + cost[i, u] then
                    dist[u] := dist[i] + cost[i, u];
}
```

5.6 String matching

Model

- Give you two strings $X=x_1, x_2, \ldots, x_n$ and $Y=y_1, y_2, \ldots, y_m$
- We aim to change X to be the same as Y

- There are three operations, and they have costs as below
  - Insert: $I(y_j)$ → put the symbol y_j to X
  - Delete: $D(x_i)$ → delete the symbol x_i from X
  - Change: $C(x_i, y_j)$ → change symbol x_i to y_j

- delete & insert → cost: 1,
  change → cost: 2 if changed ; cost: 0 otherwise

Example 5.19

- X = aabab
- Y = babb

- Delete all X and then insert all Y
- Delete x_1, x_2 and then insert y_4

Concept

- cost(i, j) represents the cost of changing $x_1x_2..x_i$ to $y_1y_2…y_j$
- cost(n, m) represents total cost

- For i=0, j=0 → cost(i, j) = 0
- For j=0, i>0 → cost(i, 0) = cost(i-1, 0) + $D(x_i)$
  - This means Y contains nothing and thus delete all symbol in X
- For i=0, j>0 → cost(0, j) = cost(0, j-1) + $I(y_j)$

- For $i \neq 0, j \neq 0$ → the goal is to let $x_1x_2..x_i = y_1y_2...y_j$ →three cases
  - Finished $x_1x_2..x_{i-1} = y_1y_2...y_j$, i.e., x_1 .. x_{i-1} are the same as y_1 .. y_j
    → So, we need to delete the upcoming x_i
    → $cost(i-1, j) + D(x_i)$

  - Finished $x_1x_2..x_{i-1} = y_1y_2...y_{j-1}$, i.e., x_1 .. x_{i-1} are the same as y_1 .. y_{j-1}
    → So, we need to change the upcoming x_i to y_j
    → $cost(i-1, j-1) + C(x_i, y_j)$

  - Finished $x_1x_2..x_i = y_1y_2...y_{j-1}$, i.e., x_1 .. x_i are the same as y_1 .. y_{j-1}
    → So, we need to insert the upcoming y_j
    → $cost(i, j-1) + I(y_j)$

Cost function

- $cost(i, j) =$
  - 0, if $i=j=0$
  - $cost(i-1, 0) + D(x_i)$, if $i>0, j=0$
  - $cost(0, j-1) + I(y_j)$, if $i=0, j>0$
  - $cost'(i, j)$, if $i>0, j>0$

  - $cost'(i, j) = \min\{cost(i-1, j)+D(x_i), cost(i-1, j-1)+C(x_{i,} y_j), cost(i, j-1)+I(y_j)\}$

Example 5.20

$$\begin{aligned} cost(1,1) &= \min\ \{cost(0,1) + D(x_1), cost(0,0) + C(x_1, y_1), cost(1,0) + I(y_1)\} \\ &= \min\ \{2, 2, 2\} = 2 \end{aligned}$$

$$\begin{aligned} cost(1,2) &= \min\ \{cost(0,2) + D(x_1), cost(0,1) + C(x_1, y_2), cost(1,1) + I(y_2)\} \\ &= \min\ \{3, 1, 3\} = 1 \end{aligned}$$

$i \downarrow$ \ $j \rightarrow$	0	1	2	3	4
0	0	1	2	3	4
1	1	2	1	2	3
2	2	3	2	3	4
3	3	2	3	2	3
4	4	3	2	3	4
5	5	4	3	2	3

initialization

5.7 0/1 knapsack

Concept

- In Sec. 5.1 we define a KNAP(1, j, y) problem
  - $f_n(m) = \max \{ f_{n-1}(m), f_{n-1}(m-w_n) + p_n \}$
  - → to find an optimal selection, when there are n items, we need to check 2^n possibilities
  - Ex. 5.21
- A more efficient selection method
  - Do not enumerate all possibilities and we can still find the solution

Concept

- Let S^i=(P, W)
  - P is the current profit after processing numbered i item
  - W is the current weight after processing numbered i item
  - $S^{i+1} = \{ S^i \cup S^i_1 \}$
    - $S^i_1=\{ (P, W) \mid (P-p_{i+1}, W-w_{i+1}) \text{ in } S^i \}$
    - S^i_1 represents that after adding item i+1, the corresponding P and W for rolling back to S^i
- The main idea of reducing computation →
  - If $P_j < P_k$ and $W_j > W_k$, it means that (P_j, W_j) can be ignored

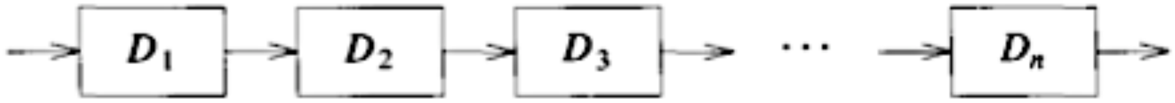
ex 5.21

- $n=3$, $(w_1, w_2, w_3)=(2,3,4)$, $(p_1, p_2, p_3)=(1,2,5)$, $m=6$

5.8 reliability design

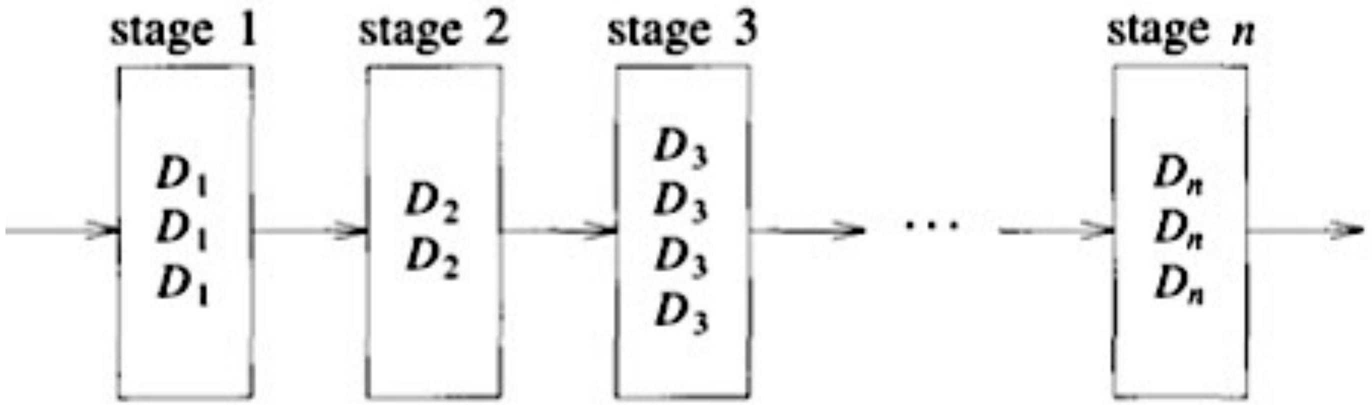
concept

- A system has many modules, and each module has a reliability value



  - Reliability → r_1 x r_2 x … r_n

To improve reliability



- For a stage i, it can have m_i modules
  - The probability that all modules fail at the same time is $(1-r_i)^{mi}$
  - So, the reliability of stage i can be $1- (1-r_i)^{mi}$

The goal

- Each module has a cost value c_i
- The goal is to maximize reliability but the cost should be less than c

$$\text{maximize } \Pi_{1 \leq i \leq n} \; \phi_i(m_i)$$

$$\text{subject to} \sum_{1 \leq i \leq n} c_i m_i \leq c$$

$$m_i \geq 1 \text{ and integer, } \; 1 \leq i \leq n$$

- DP policy

$$f_i(x) = \max_{1 \leq m_i \leq u_i} \{\phi_i(m_i) f_{i-1}(x - c_i m_i)\}$$

Ex 5.25

- The cost of D1, D2, D3 are 30, 15, 20,
- The reliability of D1, D2, D3 are 0.9, 0.8, 0.5.
- The upper bound on cost is 105

5.9 traveling salesperson problem (TSP)

Model

- TSP is more complex than 0/1 knapsack (2^n possibilities)
  - TSP is a permutation problem → n! possibilities

- Give a directed graph G=(V,E). Each edge has a cost value.
- The TSP is to find a tour (with minimum cost) from a start point vertex to go through all other vertices, and then go back to the start point

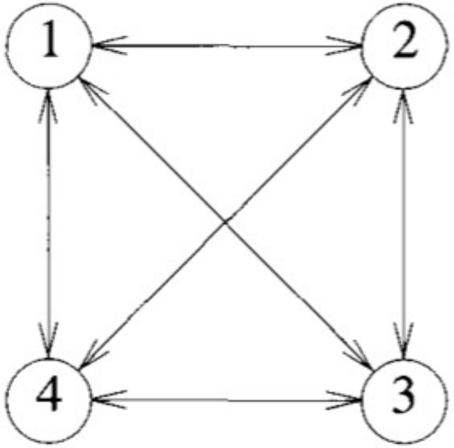
Concept

- Let the start point is vertex 1
- Let g(i, s) is the shortest path from i and going through all vertices in s, and then stop at vertex 1

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

- General equation: $g(i, S)=\min_{j \in S}\{c_{i,j} + g(j, S-\{j\}) \}$
- DP policy: from |S|=1, |S|=2, …

Ex 5.26



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

5.10 flow shop scheduling

Model

- There are *n* jobs. Each job need *m* tasks to complete.
- For a job i, its task has T_{1i}, T_{2i}, …, T_{mi}
- In the system, there are m processes to handle those m tasks
- One process can only execute one task at a time
- For a job, its task should be executed in-sequence

- → To decide a schedule, which can minimize the finish time of n jobs

A special case for two tasks

- Solution:
  - For a job i, its two tasks are a_i and b_i
  - Sort tasks a and b for all jobs in non-increasing order
  - Check in-sequence
    - If the current one is task a_i, put job i to the top-half of the schedule
    - If the current one is task b_i, put job i to the bottom-half of the schedule

Ex 5.28

- Four jobs 1, 2, 3, 4
- $(a_1, a_2, a_3, a_4) = (3, 4, 8, 10)$, $(b_1, b_2, b_3, b_4) = (6, 2, 9, 15)$