Dynamic Programming

SMITA SANKHE
smitasankhe@somaiya.edu

Introduction

- Dynamic Programming is an algorithm design technique for ***optimization problems:*** often minimizing or maximizing.
- Solves problems by combining the solutions to subproblems that contain common sub-problems.

Dynamic Programming

- DP can be applied when the solution of a problem includes solutions to subproblems
- We need to find a recursive formula for the solution
- We can recursively solve subproblems, starting from the trivial case, and save their solutions in memory
- In the end we'll get the solution of the whole problem.

Steps to Designing a Dynamic Programming Algorithm

1. Characterize optimal sub-structure.
2. Recursively define the value of an optimal solution.
3. Compute the value bottom up.
4. (if needed) Construct an optimal solution.

DP Vs D&C

- Divide-and-conquer algorithms split a problem into separate subproblems, solve the subproblems, and combine the results for a solution to the original problem.
- Example: Quicksort, Mergesort, Binary search
- Divide-and-conquer algorithms can be thought of as top-down algorithms

- Dynamic Programming split a problem into subproblems, some of which are common, solve the subproblems, and combine the results for a solution to the original problem.
- Example: Matrix Chain Multiplication, Longest Common Subsequence
- Dynamic programming can be thought of as bottom-up

- In divide and conquer, subproblems are independent.
- Divide & Conquer solutions are simple as compared to Dynamic programming .
- Divide & Conquer can be used for any kind of problems.
- Only one decision sequence is ever generated

- In Dynamic Programming , subproblems are not independent.
- Dynamic programming solutions can often be quite complex and tricky.
- Dynamic programming is generally used for Optimization Problems.
- Many decision sequences may be generated.

Greedy Vs Dynamic Programming

- Greedy strategy:
  - Make a choice at each step.
  - Make the choice before solving the subproblems.
  - Solve top-down.
- Dynamic programming strategy:
  - Make a choice at each step.
  - Choice depends on knowing optimal solutions to subproblems.
  - Solve subproblems first.
  - Solve bottom-up.

Dynamic programming

- An algorithm design method that can be used when the solution can be viewed as the result of a sequence of decisions

Some solvable by Greedy method under the condition

- Condition : an optimal sequence of decisions can be found by

making the decisions one at a time and never making an erroneous decision

For many other problems

- Not possible to make stepwise decisions (based only on local

information) in a manner like Greedy method

Outline – Dynamic Programming

- General Method
- Multistage graphs
- All pair shortest path
- Single source shortest path
- 0/1 knapsack
- Travelling salesman problem
- Matrix chain multiplication

0/1 Knapsack

Definition

The *0-1*, or *Binary, Knapsack Problem* (KP) is: given a set of n *items* and a *knapsack*, with

$$p_j = \textit{profit} \text{ of item } j,$$

$$w_j = \textit{weight} \text{ of item } j,$$

$$c = \textit{capacity} \text{ of the knapsack},$$

select a subset of the items so as to

$$\text{maximize} \quad z = \sum_{j=1}^{n} p_j x_j \tag{2.1}$$

$$\text{subject to} \quad \sum_{j=1}^{n} w_j x_j \leq c, \tag{2.2}$$

$$x_j = 0 \text{ or } 1, \qquad j \in N = \{1, \dots, n\}, \tag{2.3}$$

where

$$x_j = \begin{cases} 1 & \text{if item } j \text{ is selected;} \\ 0 & \text{otherwise.} \end{cases}$$

0-1 Knapsack problem

- Problem, in other words, is to find

$$\max \sum_{i \in T} b_i \text{ subject to } \sum_{i \in T} w_i \le W$$

- The problem is called a *"0-1"* problem, because each item must be entirely accepted or rejected.
- Just another version of this problem is the "*Fractional Knapsack Problem*", where we can take fractions of items.

D & C Approach

1. Partition the problem into subproblems.
2. Solve the subproblems.
3. Combine the solutions to solve the original one.

- **Remark:** If the subproblems are not independent, i.e. subproblems share sub-problems, then a divide and-conquer algorithm repeatedly solves the common sub-problems.
- Thus, it does more work than necessary!
- **Question:** Any better solution?

D P approach

- Dynamic programming is a method for solving optimization problems.
- **The idea:** Compute the solutions to the sub-problems *once* and store the solutions in a table, so that they can be reused (repeatedly) later.
- **Remark:** We trade space for time.

DP solution

- Step 0 - Characterize the structure of an optimal solution.
  – Decompose the problem into smaller problems, and find a relation between the structure of the optimal solution of the original problem and the solutions of the smaller problems.

Step 1: Principle of Optimality

- Express the solution of the original problem in terms of optimal solutions for smaller problems

If items are labeled *1..n*, then a subproblem would be to find an optimal solution for S_k = {*items labeled* 1, 2, .. *k*}

- This is a valid sub-problem definition.
- The question is: can we describe the final solution (S_n) in terms of subproblems (S_k)?
- Unfortunately, we <u>can't</u> do that. Explanation follows….

Step 2- Define the recursive formula

- Recursive formula for subproblems:

$$B[k,w]=\begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- It means, that the best subset of S_k that has total weight w is one of the two:

1) the best subset of S_{k-1} that has total weight w, **or**

2) the best subset of S_{k-1} that has total weight w-w_k plus the item k

Step 3- Compute the solution
0-1 Knapsack Algorithm

```
for w = 0 to W
   B[0,w] = 0
for i = 0 to n
   B[i,0] = 0
   for w = 0 to W
        if w_i <= w // item i can be part of the solution
              if b_i + B[i-1,w-w_i] > B[i-1,w]
                    B[i,w] = b_i + B[i-1,w- w_i]
              else
                    B[i,w] = B[i-1,w]
        else B[i,w] = B[i-1,w]  // w_i > w
```

Step 4- Construct the solution

```
Algorithm KnapsackElements(A,n,W)
{
i=n, k=W;
While (i>0 && k>0)
        {
        if B[i,k]<> B[i-1,k]
                mark ith item in knapsack
                k=k-wi; i=i-1
        else i=i-1
        }
}
```

Knapsack 0-1 problem

- So now we must re-work the way we build upon previous sub-problems…
  - Let **B[k, w]** represent the maximum total value of a subset S_k with weight w.
  - Our goal is to find **B[n, W],** where n is the total number of items and W is the maximal weight the knapsack can carry.
- So our recursive formula for subproblems:

B[k, w] = B[k - 1,w], if $w_k > w$

= max { B[k - 1,w], B[k - 1,w - w_k] + v_k}, otherwise

1) The best subset of S_{k-1} that has total weight w, or
2) The best subset of S_{k-1} that has total weight w-w_k plus the item k

Knapsack 0-1 Problem – Recursive Formula

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- The best subset of S_k that has the total weight w, either contains item k or not.

- **First case:** $w_k > w$
  - Item *k* can't be part of the solution! If it was the total weight would be > w, which is unacceptable.

- **Second case:** $w_k \leq w$
  - Then the item *k* can be in the solution, and we choose the case with greater value.

Knapsack 0-1 Algorithm

```
for w = 0 to W {  // Initialize 1st row to 0's
   B[0,w] = 0
}
for i = 1 to n {  // Initialize 1st column to 0's
   B[i,0] = 0
}
for i = 1 to n {
   for w = 0 to W {
         if wi <= w {  //item i can be in the solution
                  if vi + B[i-1,w-wi] > B[i-1,w]
                           B[i,w] = vi + B[i-1,w- wi]
                  else
                           B[i,w] = B[i-1,w]
         }
         else B[i,w] = B[i-1,w] // wi > w
   }
}
```

Knapsack 0-1 Problem

- Let's run our algorithm on the following data:
  - n = 4 (# of elements)
  - W = 5 (max weight)
  - Elements (weight, value):
    (2,3), (3,4), (4,5), (5,6)

Knapsack 0-1 Example

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

```
// Initialize the base cases
for w = 0 to W
        B[0,w] = 0

for i = 1 to n
        B[i,0] = 0
```

Knapsack 0-1 Example

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0				
2	0					
3	0					
4	0					

$i = 1$
$v_i = 3$
$w_i = 2$
$w = 1$
$w\text{-}w_i = -1$

if $w_i \le w$ //item i can be in the solution

if $v_i + B[i-1,w-w_i] > B[i-1,w]$

$B[i,w] = v_i + B[i-1,w- w_i]$

else

$B[i,w] = B[i-1,w]$

else **B[i,w] = B[i-1,w]** // $w_i > w$

Knapsack 0-1 Example

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3			
2	0					
3	0					
4	0					

$i = 1$

$v_i = 3$

$w_i = 2$

$\mathbf{w = 2}$

$w - w_i = 0$

if w_i <= w //item i can be in the solution

if v_i + B[i-1,w-w_i] > B[i-1,w]

B[i,w] = v_i + B[i-1,w- w_i]

else

B[i,w] = B[i-1,w]

else B[i,w] = B[i-1,w] // w_i > w

Knapsack 0-1 Example

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3		
2	0					
3	0					
4	0					

$i = 1$
$v_i = 3$
$w_i = 2$
$\mathbf{w = 3}$
$w\text{-}w_i = 1$

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i-1,w-w_i] > B[i-1,w]$

$B[i,w] = v_i + B[i-1,w- w_i]$

else

$B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Knapsack 0-1 Example

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	
2	0					
3	0					
4	0					

$i = 1$

$v_i = 3$

$w_i = 2$

$\mathbf{w = 4}$

$w\text{-}w_i = 2$

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i-1,w-w_i] > B[i-1,w]$

$B[i,w] = v_i + B[i-1,w- w_i]$

else

$B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Knapsack 0-1 Example

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0					
3	0					
4	0					

$i = 1$

$v_i = 3$

$w_i = 2$

$\mathbf{w = 5}$

$w - w_i = 3$

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i-1,w-w_i] > B[i-1,w]$

$B[i,w] = v_i + B[i-1,w- w_i]$

else

$B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0				
3	0					
4	0					

$i = 2$
$v_i = 4$
$w_i = 3$
$\mathbf{w = 1}$
$w - w_i = -2$

if $w_i \le w$ //item i can be in the solution

  if $w_i + B[i-1, w-w_i] > B[i-1,w]$

    $B[i,w] = v_i + B[i-1,w- w_i]$

  else

    $B[i,w] = B[i-1,w]$

else **$B[i,w] = B[i-1,w]$** // $w_i > w$

Knapsack 0-1 Example

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3			
3	0					
4	0					

$i = 2$

$v_i = 4$

$w_i = 3$

$\mathbf{w = 2}$

$w\text{-}w_i = -1$

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i-1,w-w_i] > B[i-1,w]$

$B[i,w] = v_i + B[i-1,w- w_i]$

else

$B[i,w] = B[i-1,w]$

else **B[i,w] = B[i-1,w]** // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
$\boxed{2: (3,4)}$
3: (4,5)
4: (5,6)

i / w	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		
3	0					
4	0					

$i = 2$
$v_i = 4$
$w_i = 3$
$\mathbf{w = 3}$
$w - w_i = 0$

if w_i <= w //item i can be in the solution

if $v_i + B[i-1,w-w_i] > B[i-1,w]$

$B[i,w] = v_i + B[i-1,w- w_i]$

else

$B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	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	
3	0					
4	0					

$i = 2$
$v_i = 4$
$w_i = 3$
$\mathbf{w = 4}$
$w\text{-}w_i = 1$

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i-1,w-w_i] > B[i-1,w]$

$B[i,w] = v_i + B[i-1,w- w_i]$

else

$B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	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					
4	0					

$i = 2$
$v_i = 4$
$w_i = 3$
$w = 5$
$w - w_i = 2$

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i-1,w-w_i] > B[i-1,w]$

$B[i,w] = v_i + B[i-1,w- w_i]$

else

$B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Knapsack 0-1 Example

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i / w	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		
4	0					

$i = 3$

$v_i = 5$

$w_i = 4$

$w = 1..3$

$w - w_i = -3..-1$

if w_i <= w //item i can be in the solution

if $v_i + B[i-1,w-w_i] > B[i-1,w]$

$B[i,w] = v_i + B[i-1,w- w_i]$

else

$B[i,w] = B[i-1,w]$

else B[i,w] = B[i-1,w] // w_i > w

Knapsack 0-1 Example

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i / w	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	
4	0					

$i = 3$

$v_i = 5$

$w_i = 4$

$\mathbf{w = 4}$

$w - w_i = 0$

if w_i <= w //item i can be in the solution

if v_i + B[i-1,w-w_i] > B[i-1,w]

B[i,w] = v_i + B[i-1,w- w_i]

else

B[i,w] = B[i-1,w]

else B[i,w] = B[i-1,w] // w_i > w

Knapsack 0-1 Example

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i / w	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					

$i = 3$

$v_i = 5$

$w_i = 4$

$\mathbf{w = 5}$

$w - w_i = 1$

if $w_i \le w$ //item i can be in the solution

    if $w_i + B[i-1, w-w_i] > B[i-1,w]$

        $B[i,w] = v_i + B[i-1, w- w_i]$

    else

        $B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Knapsack 0-1 Example

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i / w	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	

$i = 4$

$v_i = 6$

$w_i = 5$

w = 1..4

$w - w_i = -4..-1$

if $w_i \le w$ //item i can be in the solution

if $v_i + B[i-1,w-w_i] > B[i-1,w]$

$B[i,w] = v_i + B[i-1,w-w_i]$

else

$B[i,w] = B[i-1,w]$

else **B[i,w] = B[i-1,w]** // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	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

$i = 4$
$v_i = 6$
$w_i = 5$
$\mathbf{w = 5}$
$w - w_i = 0$

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i-1,w-w_i] > B[i-1,w]$

$B[i,w] = v_i + B[i-1,w-w_i]$

else

$B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Knapsack 0-1 Example

Items:

1: (2,3)

2: (3,4)

3: (4,5)

$\boxed{4: (5,6)}$

i / w	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**

We're DONE!!

The max possible value that can be carried in this knapsack is ***$7***

Knapsack 0-1 Algorithm

- This algorithm only finds the max possible value that can be carried in the knapsack
  - The value in B[n,W]

- To know the ***items*** that make this maximum value, we need to trace back through the table.

Knapsack 0-1 Algorithm Finding the Items

- Let i = n and k = W

  if B[i, k] ≠ B[i-1, k] then

  mark the i^{th} item as in the knapsack

  i = i-1, k = k-w_i

  else

  i = i-1 // Assume the i^{th} item is not in the knapsack

  // Could it be in the optimally packed knapsack?

Knapsack 0-1 Algorithm
Finding the Items

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Knapsack:

i / w	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

i = 4
k = 5
$v_i = 6$
$w_i = 5$
B[i,k] = 7
B[i-1,k] = 7

i = n , k = W
while i, k > 0
    if *B[i, k]* ≠ *B[i-1, k]* then
        mark the i^th item as in the knapsack
        i = i-1, k = k-w_i
    else
        i = i-1

Knapsack 0-1 Algorithm Finding the Items

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Knapsack:

i / w	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

i = 3
k = 5
$v_i = 5$
$w_i = 4$
B[i,k] = 7
B[i-1,k] = 7

i = n , k = W
while i, k > 0
    if *B[i, k]* ≠ *B[i-1, k]* then
        mark the ith *item as in the knapsack*
        $i = i-1,\ k = k-w_i$
    else
        $i = i-1$

Knapsack 0-1 Algorithm
Finding the Items

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Knapsack:

Item 2

i / w	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

$i = 2$
$k = 5$
$v_i = 4$
$w_i = 3$
$B[i,k] = 7$
$B[i-1,k] = 3$
$k - w_i = 2$

i = n , k = W
while i, k > 0
    if $B[i, k] \neq B[i-1, k]$ then
        mark the i^{th} item as in the knapsack
        $i = i-1, k = k-w_i$
    else
        $i = i-1$

Knapsack 0-1 Algorithm
Finding the Items

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Knapsack:

Item 2
Item 1

i / w	**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

$i = 1$
$k = 2$
$v_i = 3$
$w_i = 2$
$\mathbf{B[i,k] = 3}$
$B[i-1,k] = 0$
$k - w_i = 0$

i = n , k = W
while i, k > 0
  if $B[i, k] \neq B[i-1, k]$ then
    mark the i^th item as in the knapsack
    $i = i-1,\ k = k-w_i$
  else
    $i = i-1$

Knapsack 0-1 Algorithm Finding the Items

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Knapsack:

Item 2
Item 1

i / w	**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

$i = 1$
$k = 2$
$v_i = 3$
$w_i = 2$
$\mathbf{B[i,k] = 3}$
$B[i-1,k] = 0$
$k - w_i = 0$

k = 0, so we're DONE!

The optimal knapsack should contain:
Item 1 and Item 2

Knapsack 0-1 Problem – Run Time

```
for w = 0 to W
    B[0,w] = 0
```

O(*W*)

```
for i = 1 to n
    B[i,0] = 0
```

O(*n*)

```
for i = 1 to n
    for w = 0 to W
          < the rest of the code >
```

Repeat *n* times

O(*W*)

What is the running time of this algorithm?

O(*nW*)**

Remember that the brute-force algorithm takes: **O(2^n)**

Running time

for w = 0 to W

    B[0,w] = 0

for i = 0 to n

    B[i,0] = 0

    for w = 0 to W

        < the rest of the code >

O(W)

Repeat *n* times

O(W)

What is the running time of this algorithm?

O(n*W)

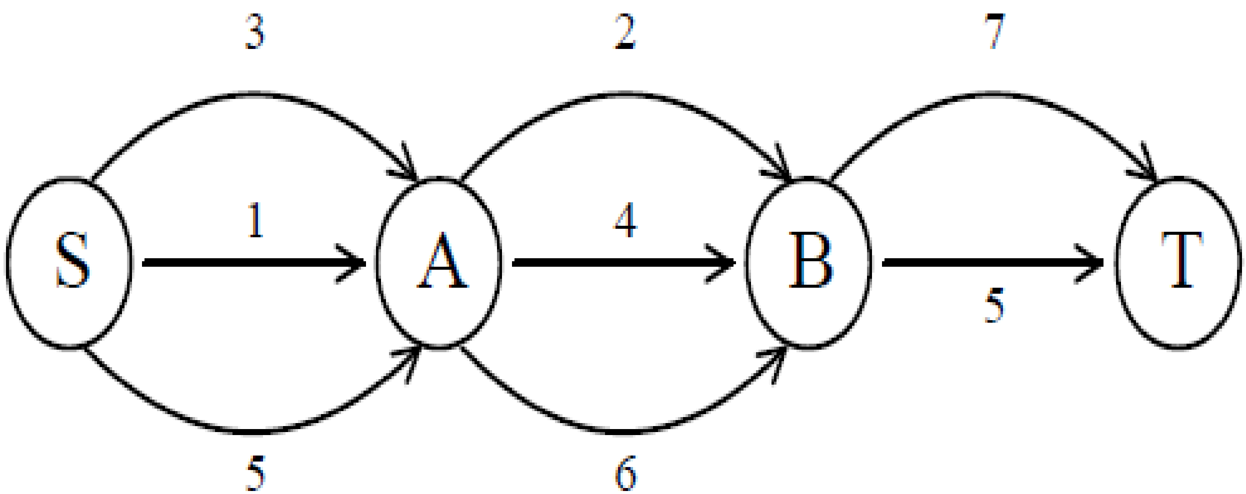
Multistage Graphs

- **Multistage Graph G(V,E)** A directed graph in which the vertices are partitioned into k disjoint sets Vi , $1<i<k$
- If $\langle u,v \rangle \in E$, then $u \in Vi$ and $v \in Vi+1$ for some i, $1<i<k$
- |V1|= |Vk|=1, and s(source) is V1 and t(sink) is Vk
- c(i,j)=cost of edge $\langle i,j \rangle$

- **Multistage graph problem-** Find a minimum-cost path from s to t of the Multistage Graph.

Shortest Path in Multistage Graph

- To find a shortest path in a multi-stage graph



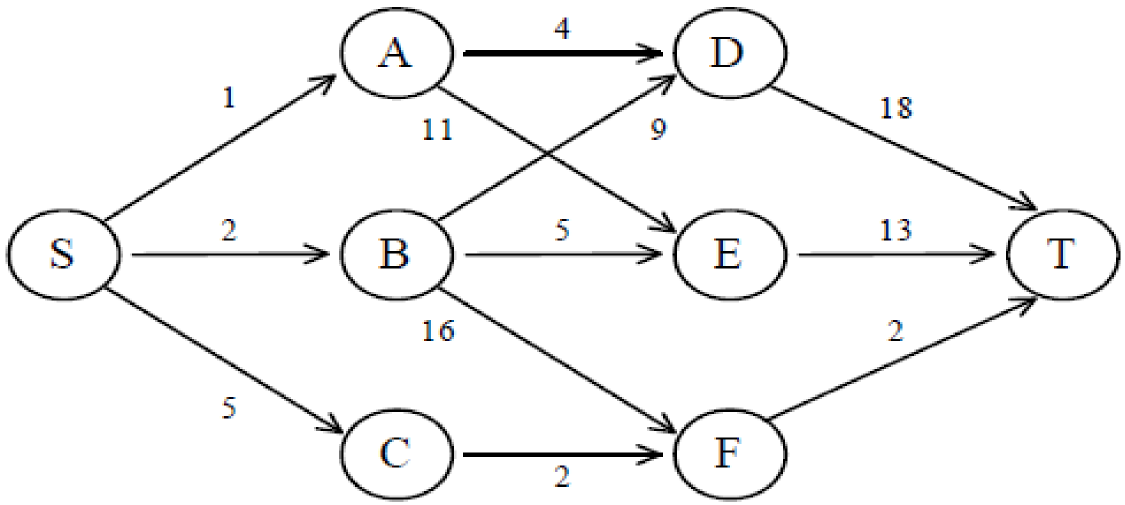
- Apply the greedy method :

the shortest path from S to T :

$$1 + 2 + 5 = 8$$

Shortest Path in Multistage Graph.. cntd

- e.g.

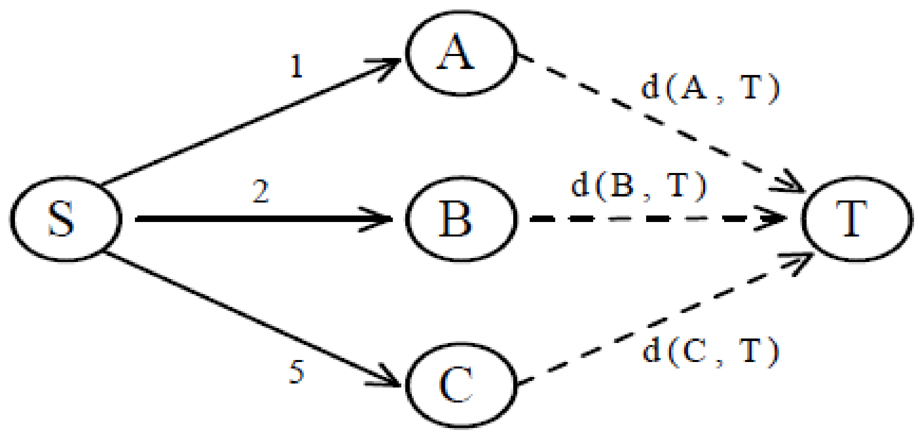


- The greedy method can not be applied to this case: (S, A, D, T) 1+4+18 = 23.
- The real shortest path is:

  (S, C, F, T) 5+2+2 = 9.

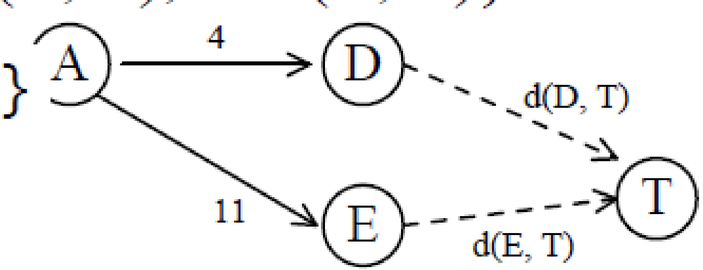
Dynamic Programming Approach

- Dynamic programming (Forward Approach):

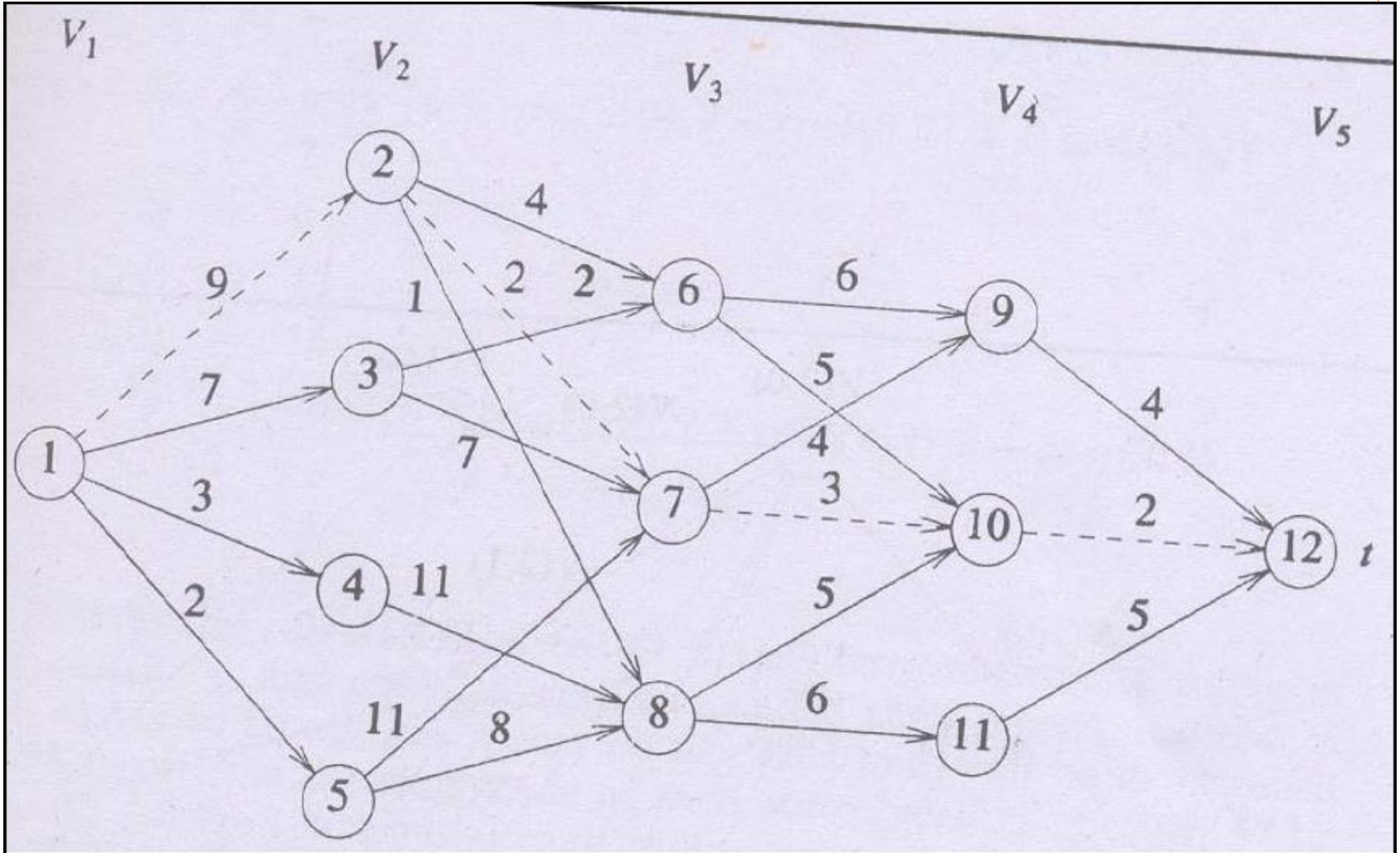


- d(S, T) = min{1+d(A, T), 2+d(B, T), 5+d(C, T)}

- d(A,T) = min{4+d(D,T), 11+d(E,T)} = min{4+18, 11+13} = 22.



Example



Steps to Designing a Dynamic Programming Algorithm

1. Characterize optimal sub-structure
2. Recursively define the value of an optimal solution
3. Compute the value bottom up
4. Construct an optimal solution

Solution- Multi Stage Graphs

Step 1- Characterize optimal sub-structure

- Every s to t path is the result of a sequence of k-2 decisions
- The principle of optimality holds (Why?)
- The principle of optimality states that whatever may be initial state and initial decision are , the remaining decision must constitute an optimal policy with regard to the state resulting from the first decision.

Step 2 - Recursively define the value of an optimal solution

- p(i, j) = a minimum cost path from vertex j in Vi to vertex t,
- cost(i, j) = cost of path p(i , j)

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

$$\operatorname{bcost}(i, j) = \min_{\substack{l \in V_{i-1} \\ <j,l> \in E}} \{\operatorname{bcost}(i-1, l) + c(l, j)\}$$

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

Step 3- Compute the value bottom up

- Solve with forward approach or backward approach

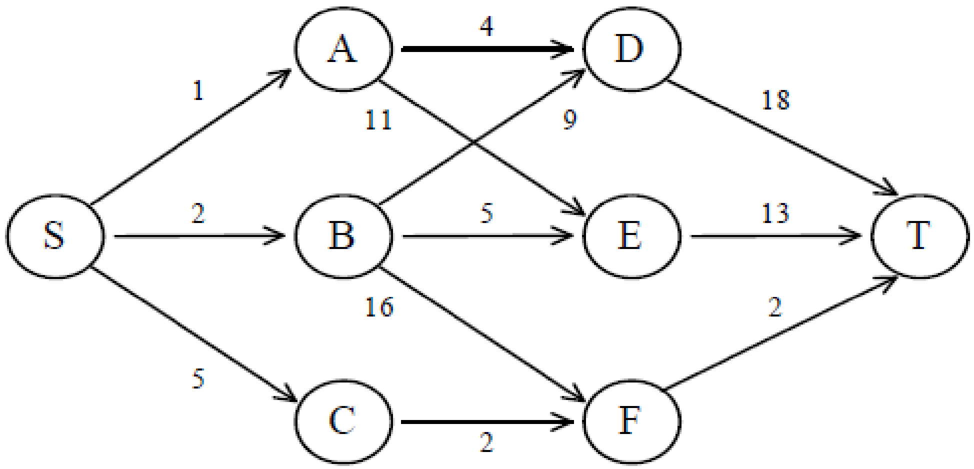
Step 4 - Construct an optimal solution

- Remember the best values along the path and construct the solution

```
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]];
}
```

Solve the following MSG problems

- e.g.



- The greedy method can not be applied to this case: (S, A, D, T) 1+4+18 = 23.
- The real shortest path is:

  (S, C, F, T) 5+2+2 = 9.

All Pairs Shortest Path

"Shortest Path"

- Given graph G=(V,E) with positive weights W(u,v) on the edges (u, v), and given two vertices a and b.
- Find the "shortest path" from a to b (where the length of the path is the sum of the edge weights on the path). Perhaps we should call this the minimum weight path!

Dynamic Programming

- The problem can be recursively defined (by the sub-problem of the same kind)
- A table is used to store the solutions of the subproblems (the meaning of "programming" before the age of computers).

Designing a DP solution

- How are the subproblems defined?
- Where are the solutions stored?
- How are the base values computed?
- How do we compute each entry from other entries in the table?
- What is the order in which we fill in the table?

Dynamic Programming

let {1,2,…,n} denote the set of vertices.

Sub-problem formulation:

- M[i,j,k] = min length of any path from i to j that uses *at most* k edges.

All paths have at most n-1 edges, so $1 \leq k \leq n-1$.

Minimum paths from i to j are found in M[i,j,n-1]

To simplify the notation, we assume that $V = \{1, 2, \ldots, n\}$.

Assume that the graph is represented by an $n \times n$ matrix with the weights of the edges:

$$w_{ij} = \begin{cases} 0 & \text{if } i = j, \\ w(i,j) & \text{if } i \neq j \text{ and } (i,j) \in E, \\ \infty & \text{if } i \neq j \text{ and } (i,j) \notin E. \end{cases}$$

Output Format: an $n \times n$ matrix $D = [d_{ij}]$ where d_{ij} is the length of the shortest path from vertex i to j.

Recursive formula

- $A[i,j] = \min\{ \min_{1<=k<=n}\{ A^{k-1}[i,k] + A^{k-1}[k,j]\}, cost[i,j]\}$

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

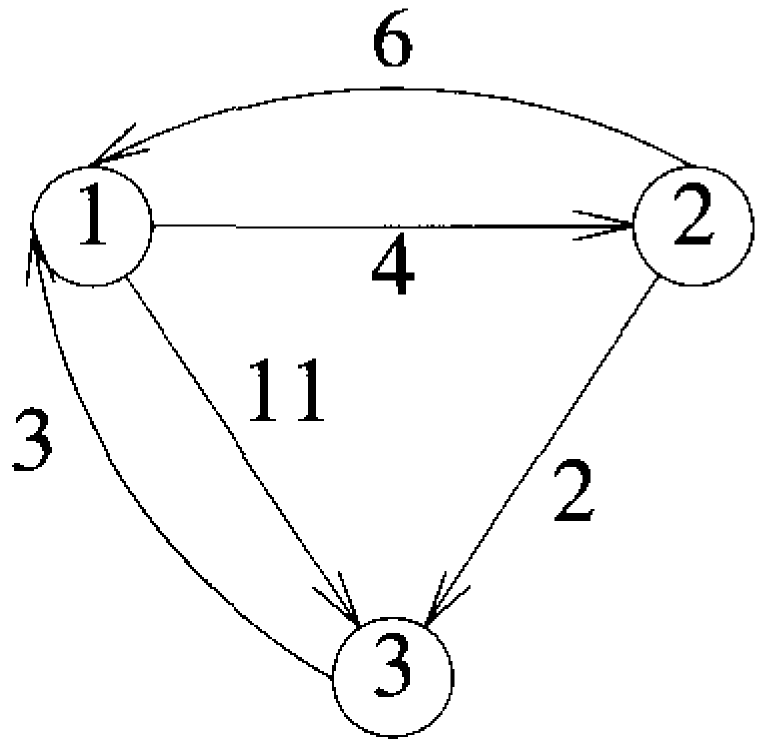
$$A(i,j) = \min\ \{ \min_{1\le k\le n} \{A^{k-1}(i,k) + A^{k-1}(k,j)\}, cost(i,j)\}$$

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

Algorithms- All Pairs Shortest Path

```
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]);
}
```

Example



A^0	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

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

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

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

Running time analysis

```
For k = 1 to n-1
   for j = 1 to n
     for i = 1 to n
        M[i,j,k] = min{min{M[i,x,k-1] + w(x,j): x in V},
                              M[i,j,k-1]}
```

- How many entries do we need to compute? $O(n^3)$

  $1 \leq i \leq n$; $1 \leq j \leq n$; $1 \leq k \leq n-1$

Applications

- Shortest paths in directed graphs
- Optimal routing.
- Fast computation of Pathfinder networks. Widest paths/Maximum bandwidth paths

Single Source Shortest Path

- When there are no cycles of negative length, there is shortest path between any two vertices of n-vertex graph that has most n-1 edges on it
- If there are cyles, elimination of cycles from path results in another path with same source & same destination

Step 1-Optimal Substructure

Let $dist^{\ell}[u]$ be the length of a shortest path from the source verte to vertex u under the constraint that the shortest path contains at mos edges. Then, $dist^1[u] = cost[v,u]$, $1 \leq u \leq n$. As noted earlier, when th are no cycles of negative length, we can limit our search for shortest pa to paths with at most $n-1$ edges. Hence, $dist^{n-1}[u]$ is the length of unrestricted shortest path from v to u.

1. If the shortest path from v to u with at most k, $k > 1$, edges has more than $k - 1$ edges, then $dist^k[u] = dist^{k-1}[u]$.

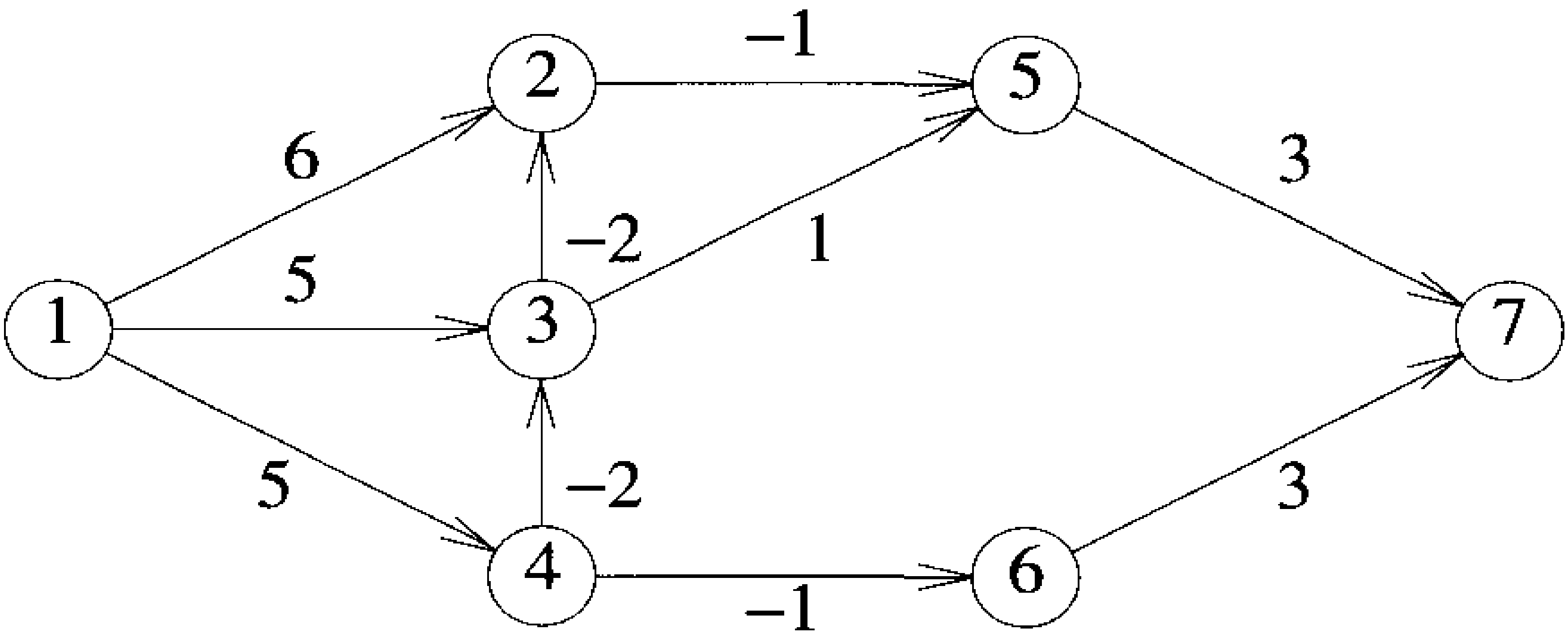
2. If the shortest path from v to u with at most k, $k > 1$, edges h exactly k edges, then it is made up of a shortest path from v to so vertex j followed by the edge $\langle j, u \rangle$. The path from v to j has k - edges, and its length is $dist^{k-1}[j]$. All vertices i such that the ec $\langle i, u \rangle$ is in the graph are candidates for j. Since we are interested i shortest path, the i that minimizes $dist^{k-1}[i] + cost[i, u]$ is the corr value for j.

Step 2- Define Recursive formula

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

This recurrence can be used to compute $dist^k$ from $dist^{k-1}$, for $k = 2, 3, \ldots, n-1$.

Example



(a) A directed graph

Step 3- Compute Solution

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

(b) $dist^k$

Step 4 is optional

Algorithm

```
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];
}
```