알고리즘

04. Dynamic Programming

2014/11/19

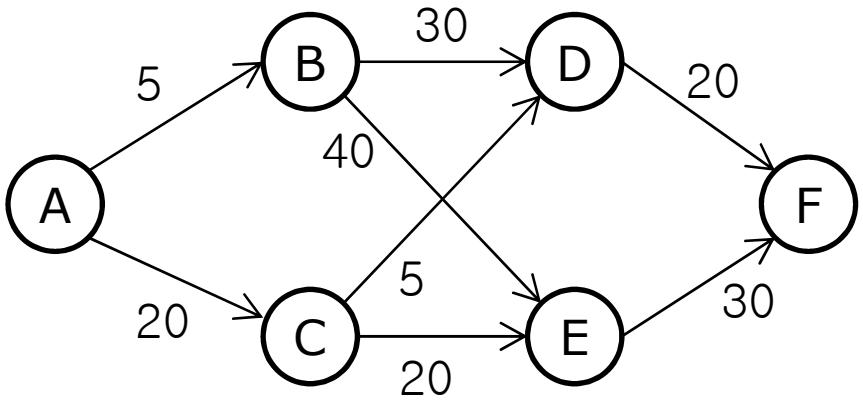
미디어소프트웨어학과
민경하

Contents

Contents

4.0 Introduction

- Key idea of dynamic programming (1)
  - Comes from greedy algorithm
    - A sequence of selections
    - The counterexample of greedy algorithm
      - Find a shortest path from A to F: Cost(A → F)?

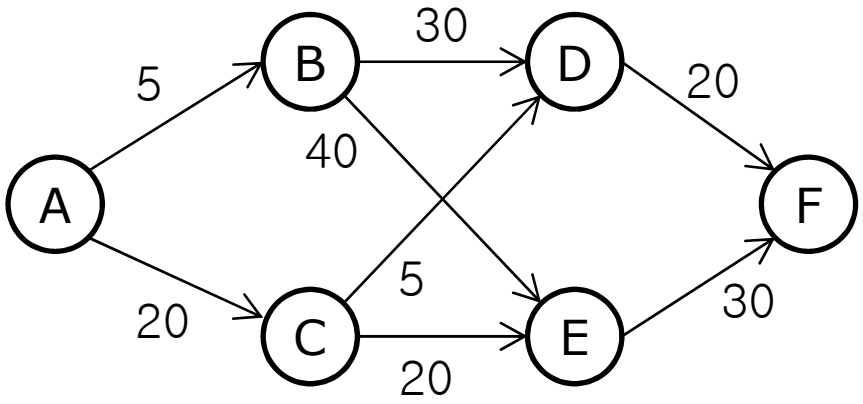


Can a greedy algorithm solve this problem?

4.0 Introduction

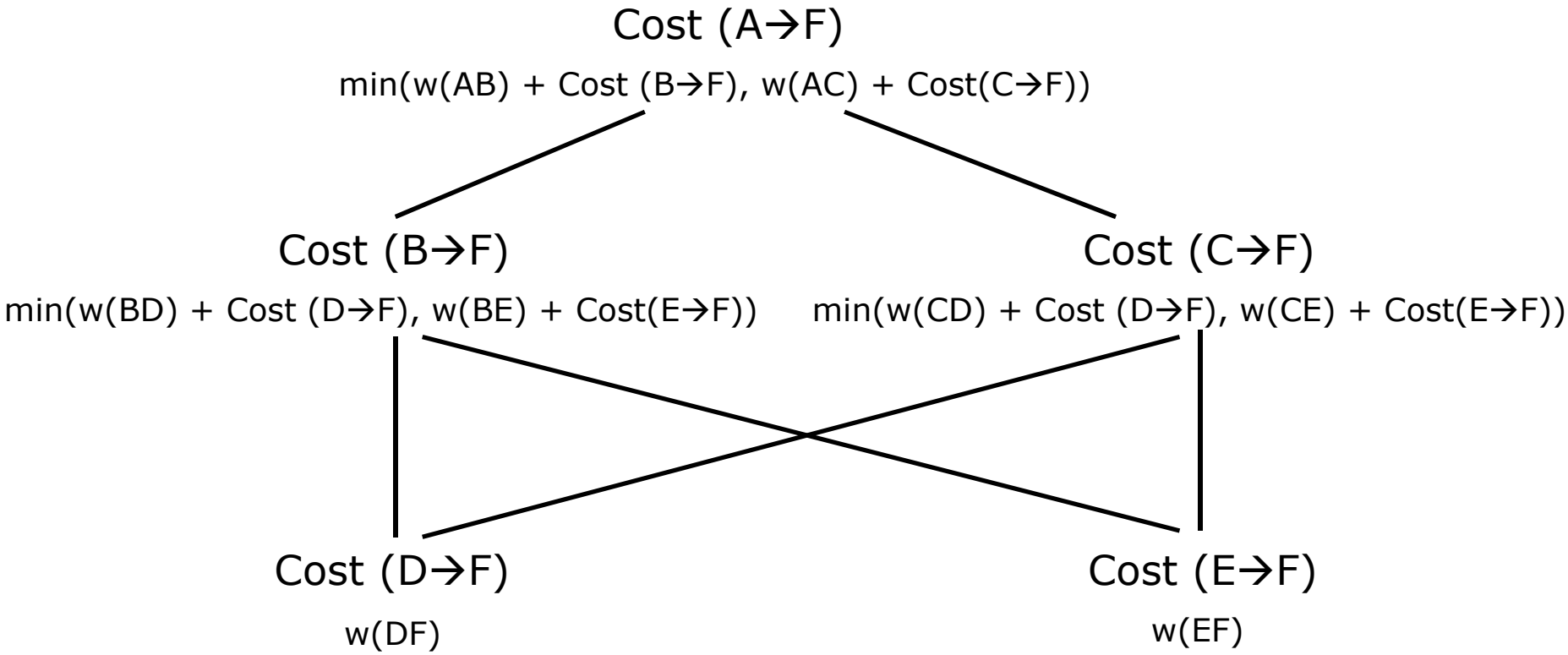
- Key idea of dynamic programming (2)
  - Comes from divide & conquer
    - Decomposing a problem into a set of sub-problems
    - Recursive structure of solution
      - Cost (A → F)?

Cost (A→F) = min(w(AB) + Cost (B→F), w(AC) + Cost(C→F))



4.0 Introduction

- Key idea of dynamic programming (3)
  - Recursive decomposition of a problem



4.0 Introduction

- Principle of optimality

> **Whatever the initial state and decision are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision.**

- Leads to an recurrence relation
- Two approaches
  - There are two ways to formulate the recurrence relations
  - Forward approach
    - The formulation for decision x_i is made in terms of optimal decision sequences x_{i+1}, …, x_n.
  - Backward approach
    - The formulation for decision x_i is made in terms of optimal decision sequences x_1, …, x_{i-1}.

4.1 0/1 Knapsack

Review: Knapsack (Greedy algorithm)

- Problem:
  - We are given n objects and a knapsack.
  - Object **i** has a weight $\mathbf{w_i}$ and a profit $\mathbf{p_i}$, and the knapsack has a capacity **M**.
  - If a fraction $\mathbf{x_i}$, $0 \le x_i \le 1$, of object i is placed into the knapsack, then the profit of $p_i x_i$ is earned.
  - The objective is to obtain a filling of the knapsack that maximizes the total profit earned.

$$\text{Maximize} \sum_{1 \le i \le n} p_i x_i \text{ subject to } \sum_{1 \le i \le n} w_i x_i \le M$$

  - The solution is a set $(x_1, x_2, \ldots, x_n)$

4.1 0/1 Knapsack

Review: Knapsack (Greedy algorithm)

- Strategy:
  - At each step, we include that object which has the maximum profit per unit of capacity.
  - In the order of the ratio p_i / w_i.
- Example:
  - n = 3
  - M = 20
  - (p_1, p_2, p_3) = (25, 24, 15)
  - (w_1, w_2, w_3) = (18, 15, 10).
  - What is the solution (x_1, x_2, x_3) ?

4.1 0/1 Knapsack

Review: Knapsack (Greedy algorithm)

- Counter example:
  - Can a greedy algorithm solve the following knapsack problem when x_i is only 0 or 1?

- Example:
  - n = 6,
  - M = 100,
  - $(p_1, p_2, p_3, p_4, p_5, p_6) = (40, 35, 18, 4, 10, 2)$,
  - $(w_1, w_2, w_3, w_4, w_5, w_6) = (100, 50, 45, 20, 10, 5)$.

  - What is the solution ?

4.1 0/1 Knapsack

- Principle of optimality
  - The key background of dynamic programming
  - The property of an optimal sequence of decisions is

Whatever the initial state and decision are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision.

4.1 0/1 Knapsack

- Problem
  - Similar to the knapsack problem except that x_i can be either 0 or 1
  - KNAP (1, n, M)

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

$$\text{subject to} \sum_{i=1}^{n} w_i x_i \leq M$$

$$x_i = 0 \text{ or } 1$$

4.1 0/1 Knapsack

- Strategy
  - Build a recurrence relation using KNAP (1, n, M)
  - KNAP (1, n, M)
    - If $x_1 = 0$, then KNAP(1, n, M) = KNAP (2, n, M)
    - If $x_1 = 1$, then
      KNAP(1, n, M) = KNAP (2, n, M – w_1)

4.1 0/1 Knapsack

- Strategy
  - Let g_j (y) be the value of an optimal solution to KNAP (j+1, n, y).
  - g_0 (M) is the value of an optimal solution of KNAP (1, n, M).

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

$$g_i(y) = \max\{g_{i+1}(y),\ g_{i+1}(y - w_{i+1})\ +\ p_{i+1}\}$$

4.1 0/1 Knapsack

- Strategy
  - Degenerate case
    - $g_{n-1}(M') = \begin{cases} p_n, & if\ M' \geq w_n \\ 0, & otherwise \end{cases}$
    - $g_i(M') = 0,\ if\ M' < \min(w_{i+1},\ldots,w_n)$
    - $g_i(M') = \sum_{k=i+1}^{n} p_k,\ \ if\ \sum_{k=i+1}^{n} w_k \leq M'$

4.1 0/1 Knapsack

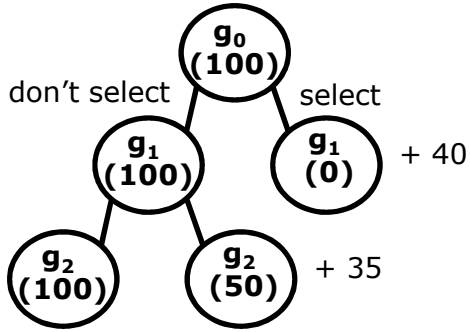
- Example
  - n = 6,
  - M = 100,
  - $(p_1, p_2, p_3, p_4, p_5, p_6) = (40, 35, 18, 4, 10, 2)$,
  - $(w_1, w_2, w_3, w_4, w_5, w_6) = (100, 50, 45, 20, 10, 5)$.

  - What is the solution ?

  - Smart degenerate case?

$$g_i(M) = \sum_{k=i+1}^{n} p_k, if \sum_{k=i+1}^{n} w_k < M$$

4.1 0/1 Knapsack

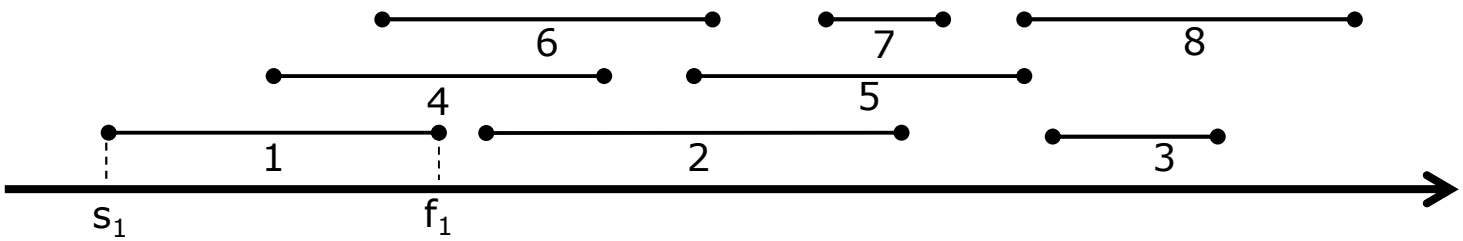


4.1 0/1 Knapsack

```
int KNAP( int i, int n, int capacity )
{
    if ( capacity < min_w ( i, n ) )
        return 0;

    if ( capacity > sum_w ( i, n ) )
        return sum_p ( i, n );

    return max ( KNAP (i-1, n, capacity-w[i]) + p[i]),
                 KNAP (i-1, n, capacity );
}
```

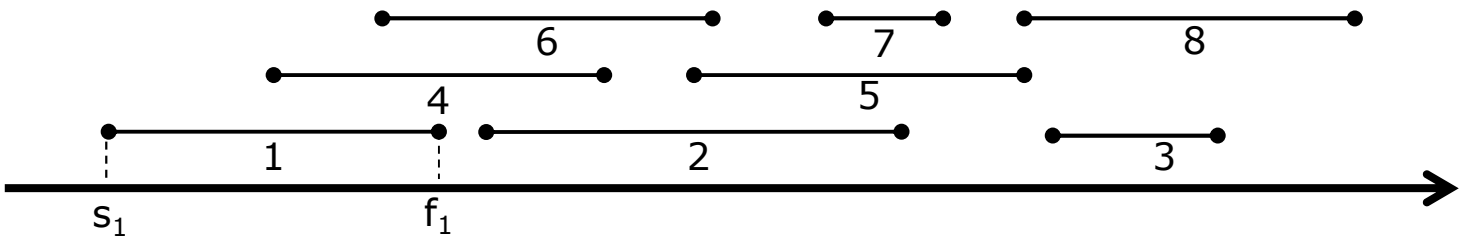
4.2 (Weighted) interval scheduling

- Interval scheduling
  - A set of n request: {1, …, n}
  - i^{th} request
    - {start time (s_i), finish time (f_i)}
  - Compatible
    - A subset of requests is **compatible**, if no two of them overlap in time.
  - Goal
    - Find a set of maximum compatible subsets



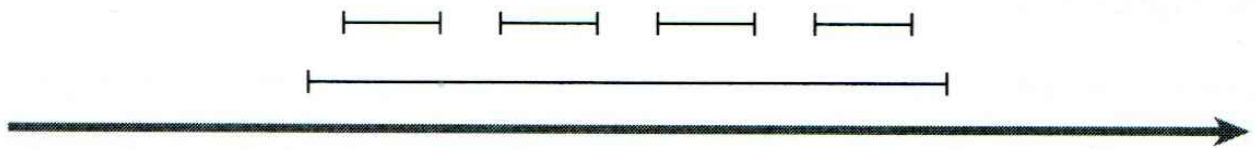
4.2 (Weighted) interval scheduling

- Designing a greedy algorithm
  - Basic idea
    - Select a first request i_1.
    - Reject all the requests that are not compatible with i_1.
    - Select the next request i_2.
    - Reject all the requests that are not compatible with i_2.
    - Repeat this process until we run out of requests.
  - Greedy algorithm
    - Which rule to select the requests.



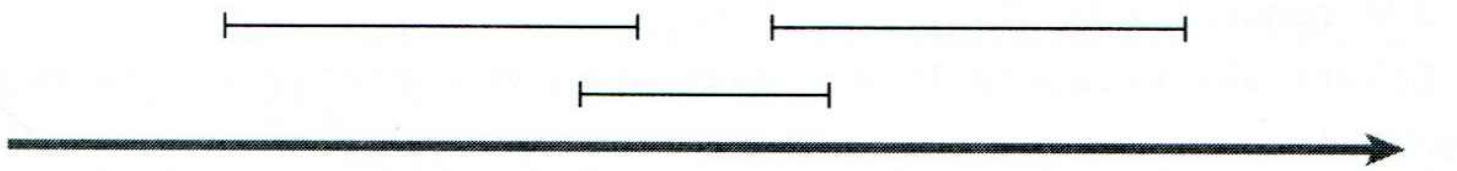
4.2 (Weighted) interval scheduling

- Designing a greedy algorithm
  - Rule 1.
    - Choose the earliest starting interval.
    - Choose an interval i such that s_i is minimum.
    - Counterexample



4.2 (Weighted) interval scheduling

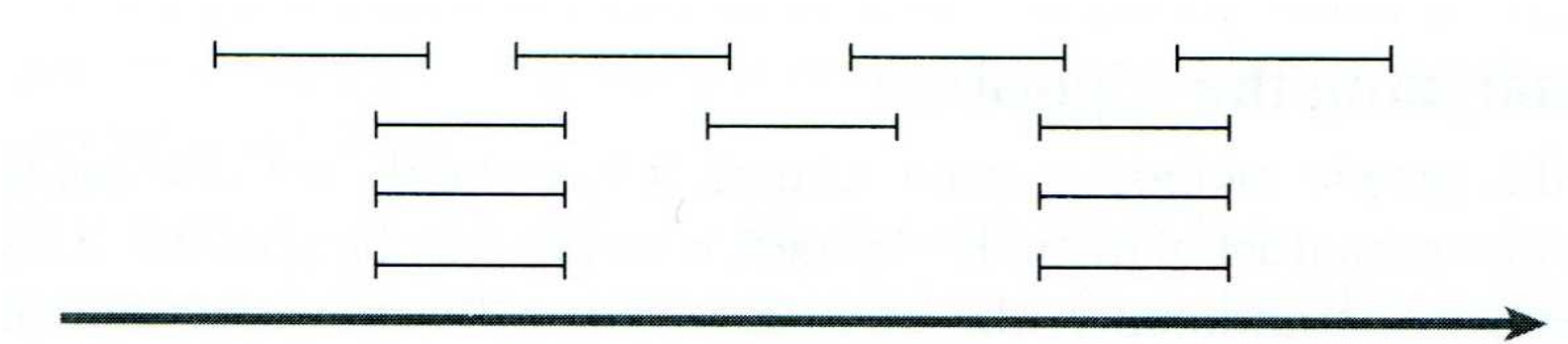
- Designing a greedy algorithm
  - Rule 2.
    - Choose the shortest interval.
    - Choose an interval i such that $f_i - s_i$ is minimum.
    - Counterexample



4.2 (Weighted) interval scheduling

- Designing a greedy algorithm
  - Rule 3.
    - Choose the interval with least conflicts.
    - Choose an interval i that overlaps least intervals.

    - Counterexample



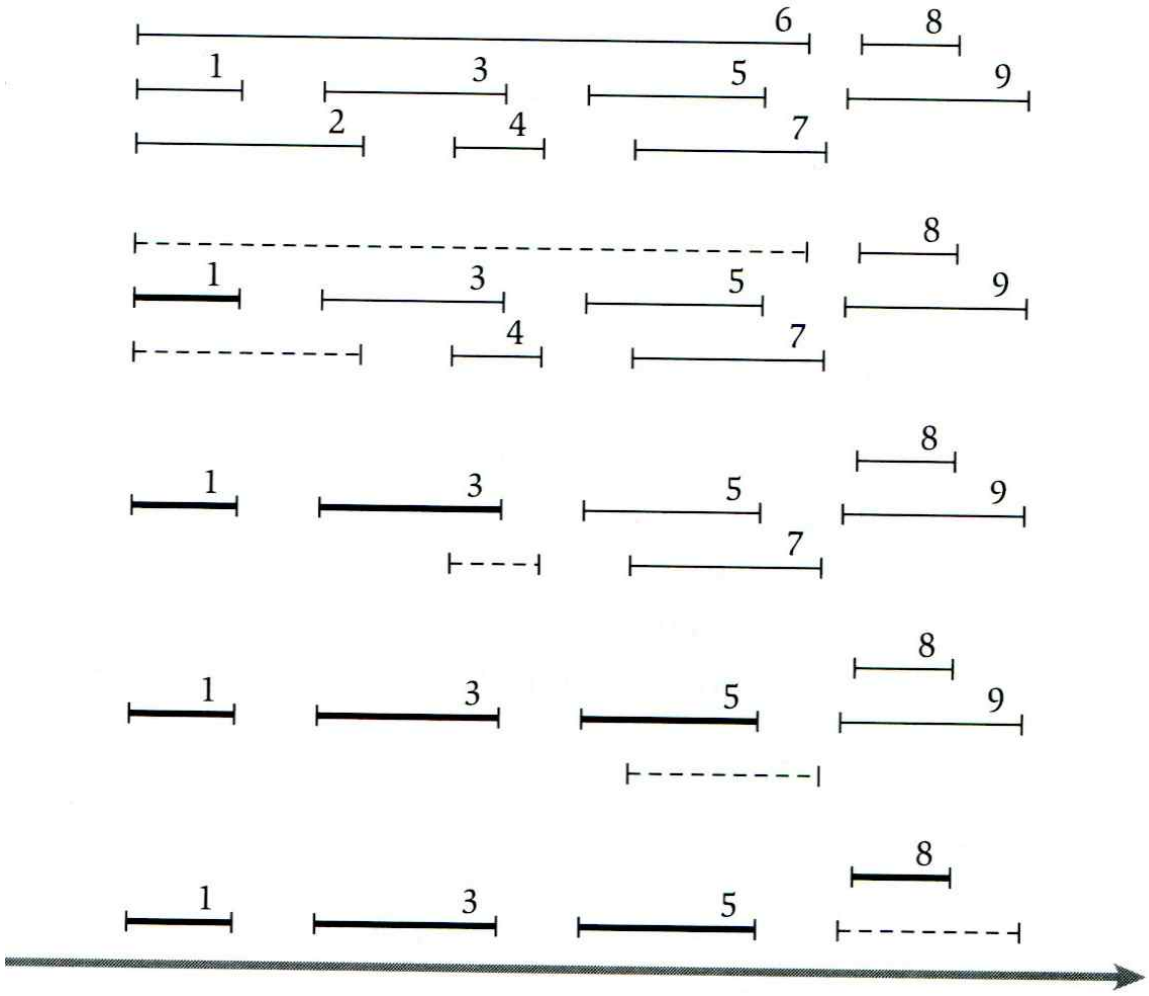
4.2 (Weighted) interval scheduling

- Designing a greedy algorithm
  - Rule 4.
    - Choose the earliest finishing interval.
    - Choose an interval i such that f_i is minimum.

```
Interval Scheduling( int n, request R )
{
    request A; //      solution
    A ← Φ;
    while ( R is not Φ )
        choose a request i from R whose finish time is minimum;
        add i to A;
        delete all the requests from R that are not compatible with i;

    return A;
}
```

4.2 (Weighted) interval scheduling

- Example



4.2 Weighted interval scheduling

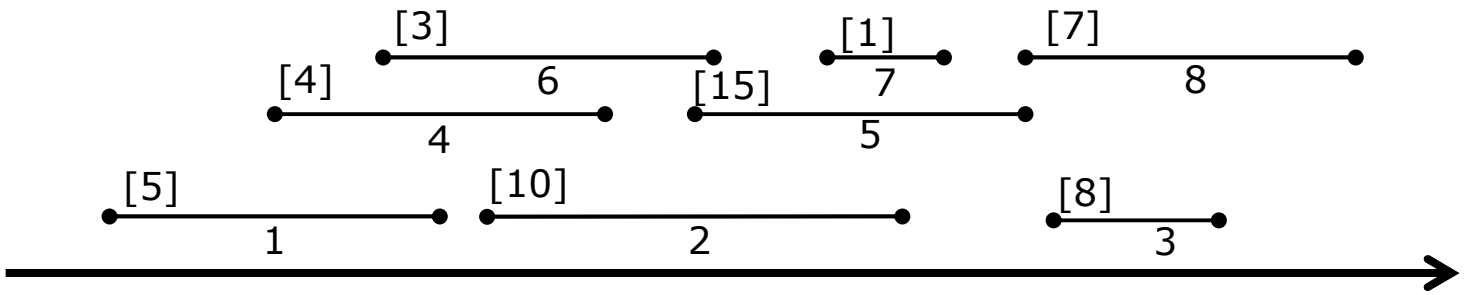
- Extended problem

  → Weighted interval scheduling

  – Interval with different weights

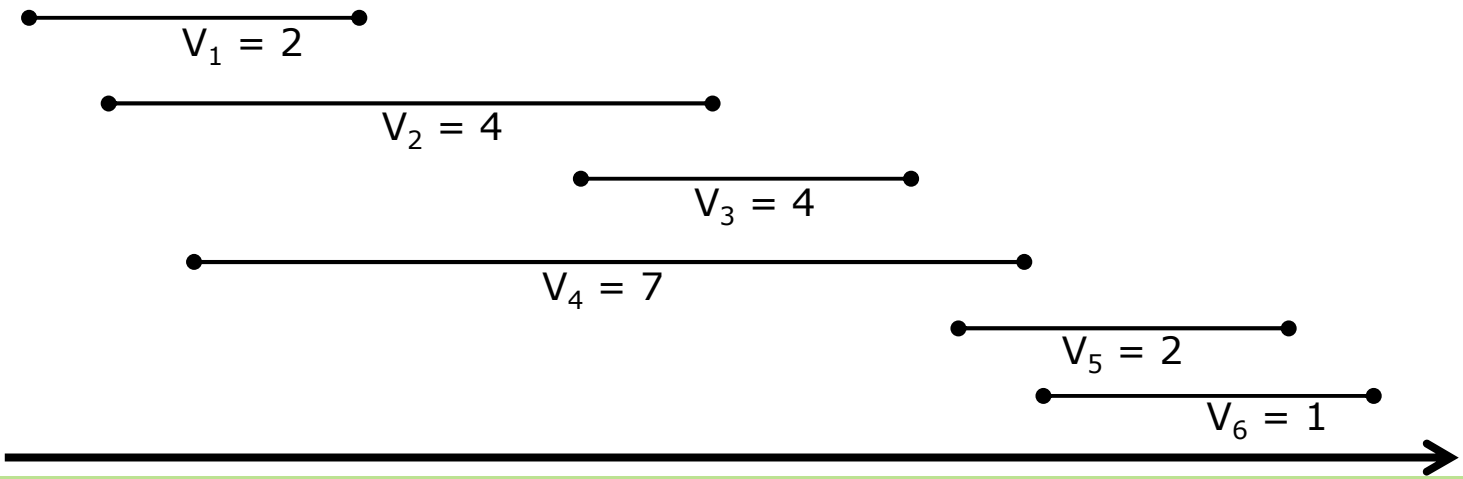
  – Complete requests with maximum weights

Q1. Is there a greedy algorithm that solves this problem?



4.2 Weighted interval scheduling

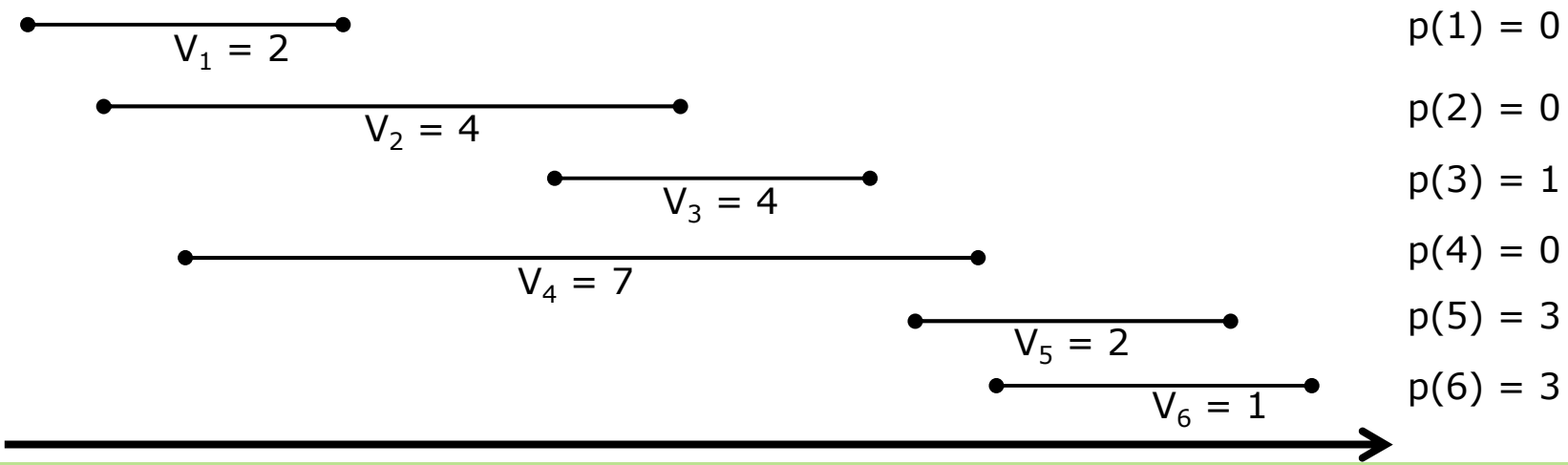
- Problem
  - Each interval is defined by three values:
    {start time (s_i), finish time (f_i), weight (v_i)}
  - Goal
    - Find a set of maximum compatible subsets (Greedy)
      ➔ **Find a set of compatible intervals such that the sum of the weights is maximum**



4.2 Weighted interval scheduling

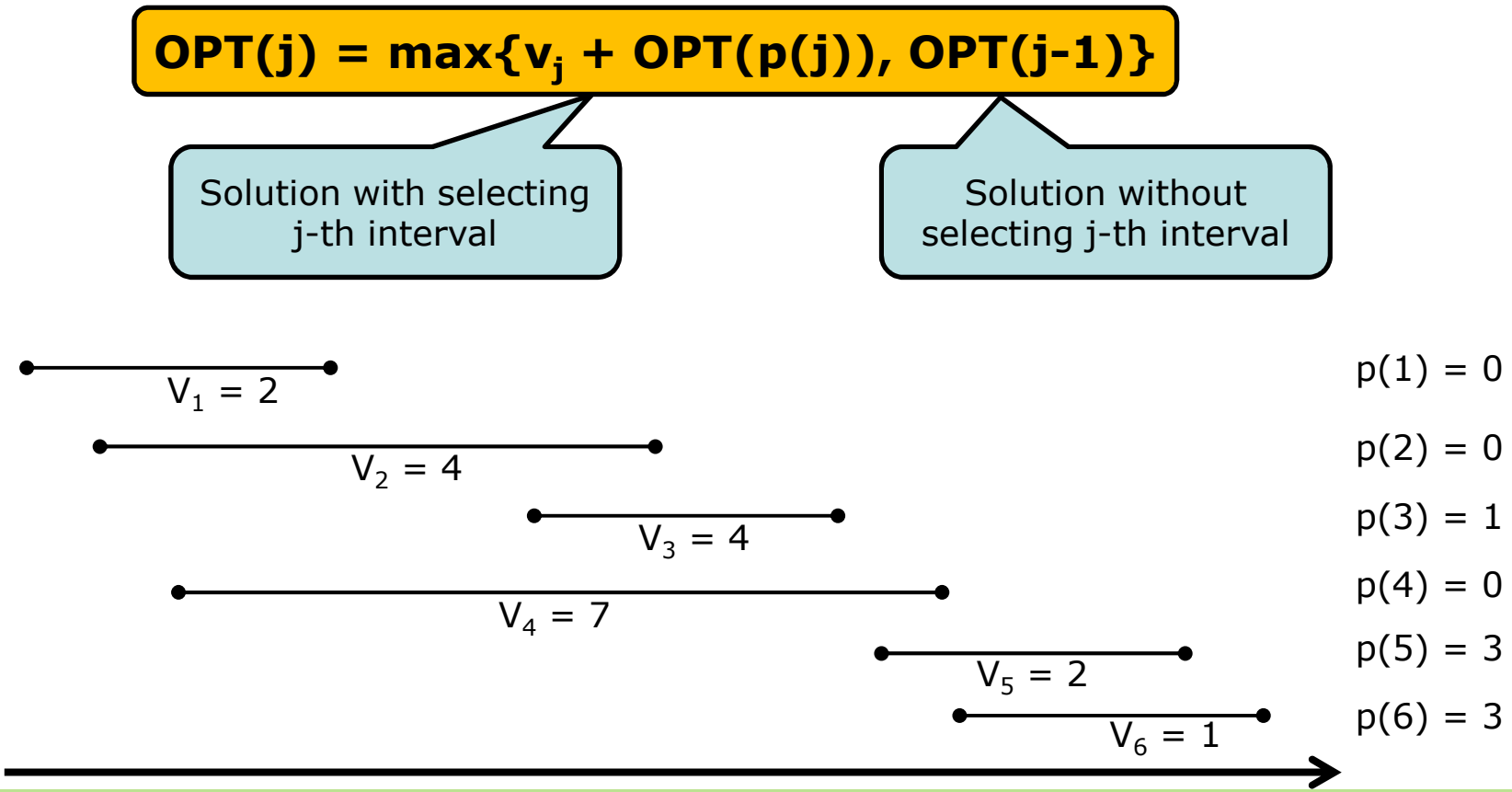
- Strategy
  - Sort intervals according to the decreasing order of finish time

    {6, 5, 4, 3, 2, 1}
  - Define p(j) for an interval j
    - p(j) = the largest index i < j such that intervals i & j are disjoint



4.2 Weighted interval scheduling

- Solution
  - OPT(j)
    - The optimal solution of j intervals

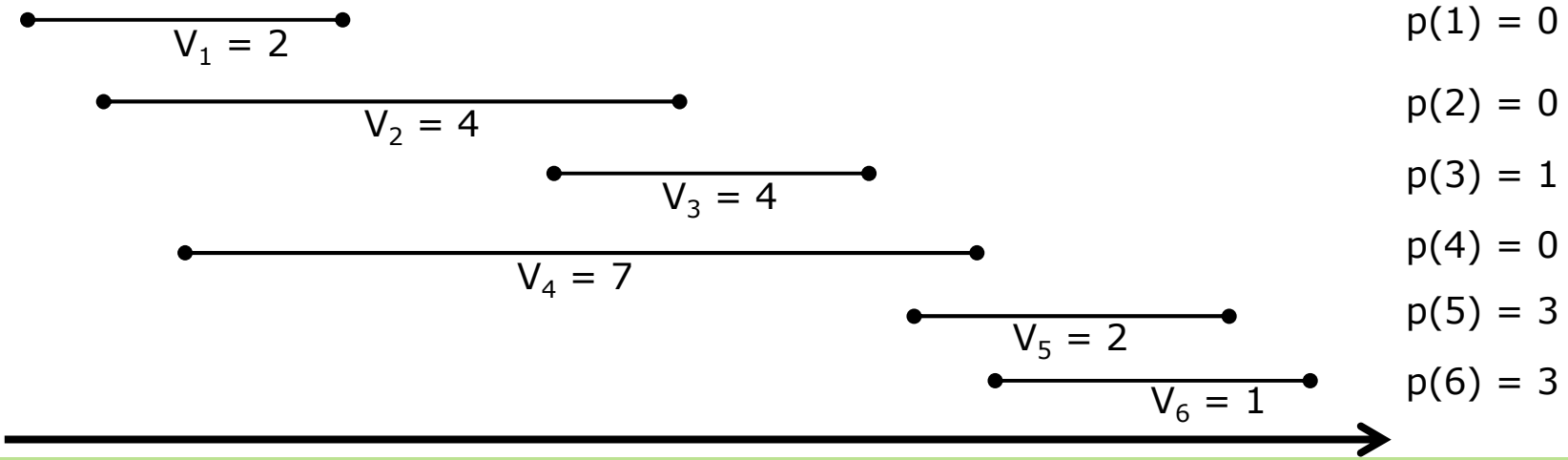


4.2 Weighted interval scheduling

- Example

$OPT(j) = \max\{v_j + OPT(p(j)), OPT(j-1)\}$

  - 6 intervals
    - $OPT(6) = \max\{v_6 + OPT(p(6)), OPT(5)\}$
      $= \max\{1 + OPT(3), OPT(5)\}$
    - $OPT(5) = \max\{2 + OPT(3), OPT(4)\}$
    - $OPT(4) = \max\{7 + OPT(0), OPT(3)\}$
    - $OPT(3) = \max\{4 + OPT(1), OPT(2)\}$
    - $OPT(2) = \max\{4 + OPT(0), OPT(1)\}$
    - $OPT(1) = \max\{2 + OPT(0), OPT(0)\}$



4.2 Weighted interval scheduling

- Algorithm (recursive)

```
int OPT ( int j )
{
    if ( j == 0 )
        return 0;

    return max( v[j] + OPT(P[j]), OPT(j-1) );
}
```

  - Problem?
    - Redundantly calling OPT(x)
    - Use an extra memory to void redundant recursive call

4.2 Weighted interval scheduling

- Algorithm (memorized & recursive)

```
int OPT ( int j )
{
    if ( j == 0 )
        return 0;

    if ( M[j] is not empty )
        return M[j];

    M[j] = max( v[j] + OPT(P[j]), OPT(j-1) );
    return M[j];
}
```

  - Time complexity?

4.2 Weighted interval scheduling

- Algorithm (memorized & non-recursive)

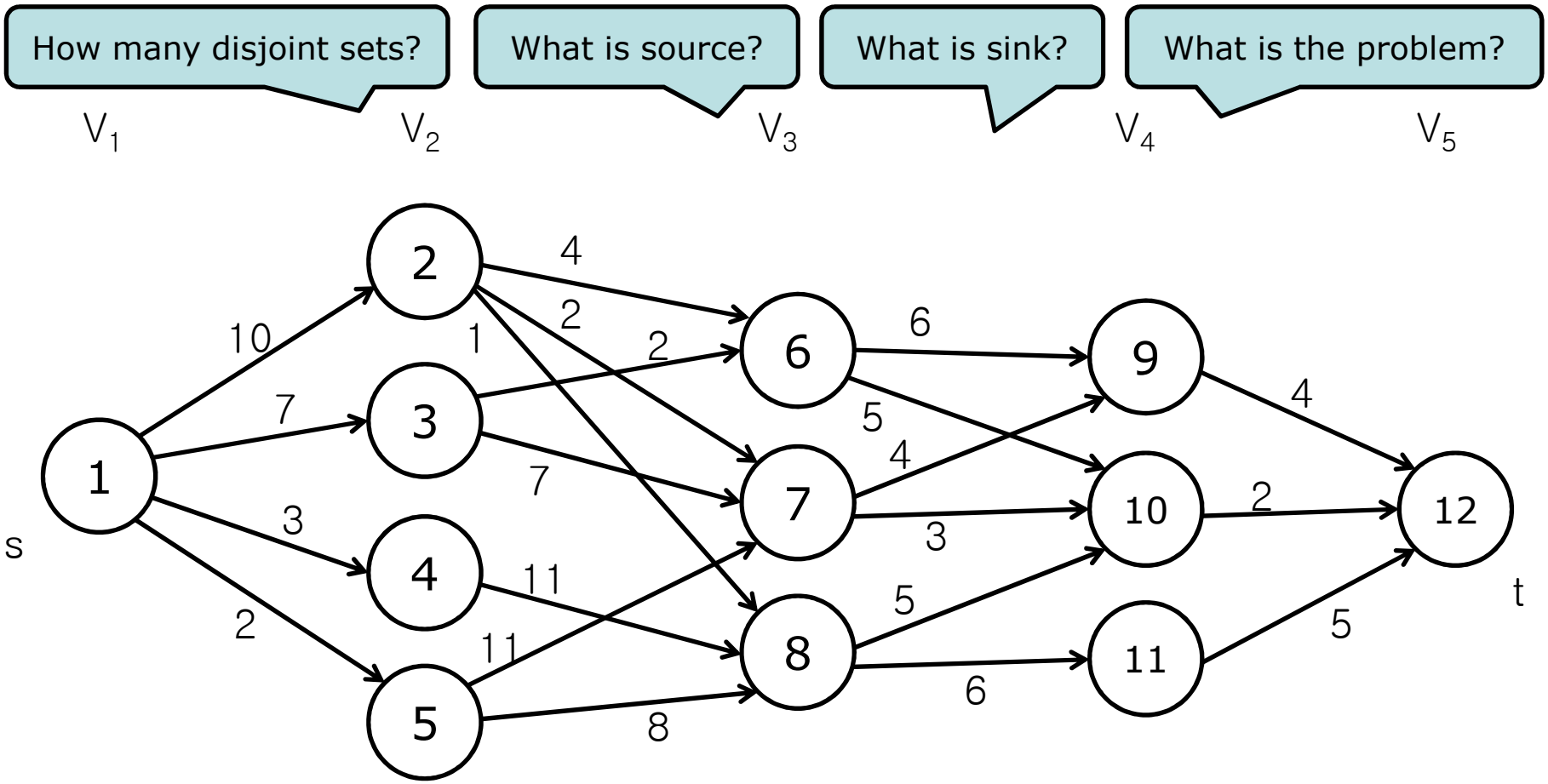
```
int OPT ( int j )
{
    int M[];

    M[0] = 0;

    for ( i = 1; i <= n; i++ )
        M[i] = max( v[i] + M[P[i]], M[i-1] );
}
```

4.3 Multistage graph

- Definition of multistage graph
  - A directed graph
    - The vertices are partitioned into $k \geq 2$ disjoint sets V_i, $1 \leq i \leq k$.
    - If <u, v> is an edge in E, then $u \in V_i$ and $v \in V_{i+1}$, for some i.
    - $|V_1| = |V_k| = 1$.
    - If $s \in V_1$, then s is the source
    - If $t \in V_k$, then t is the sink
  - Let c(i, j) be the cost of edge <i, j>.
    - The cost of a path from s to t is the sum of the costs of the edges on the path.
    - The multistage graph problem is to find a minimum cost path from s to t.

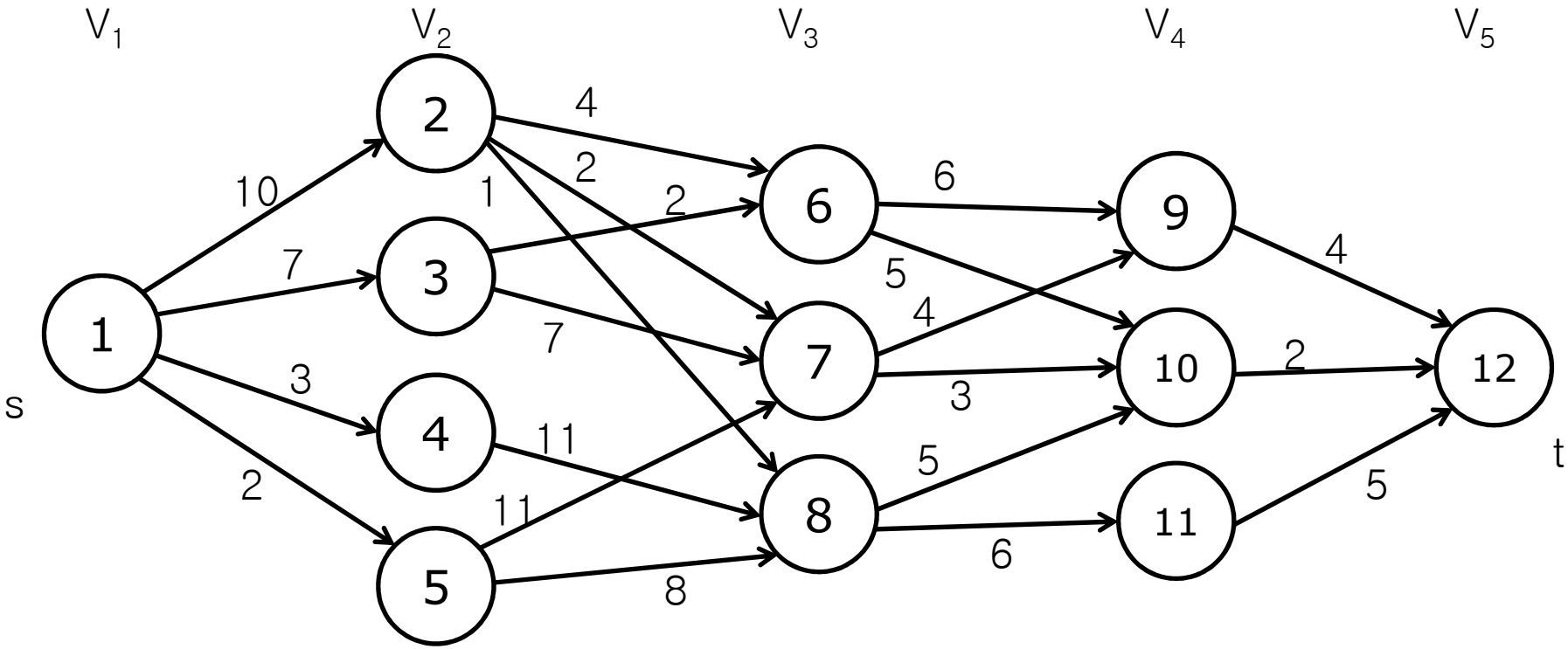
4.3 Multistage graph

- Definition of multistage graph
  - An example of a multistage graph



4.3 Multistage graph

- Problem
  - Find an optimal path from source to sink



4.3 Multistage graph

- Strategy
  - P(i, j)
    - minimum cost path from vertex j in V_i to sink
  - COST(i, j)
    - the cost of P(i, j)

Algorithm

$$COST(i,j) = \min_{l \in V_{i+1}} \{c(j,l) + COST(i+1,l)\}$$

  - What is the problem to solve?
    - P(1, 1)
    - COST(1, 1)

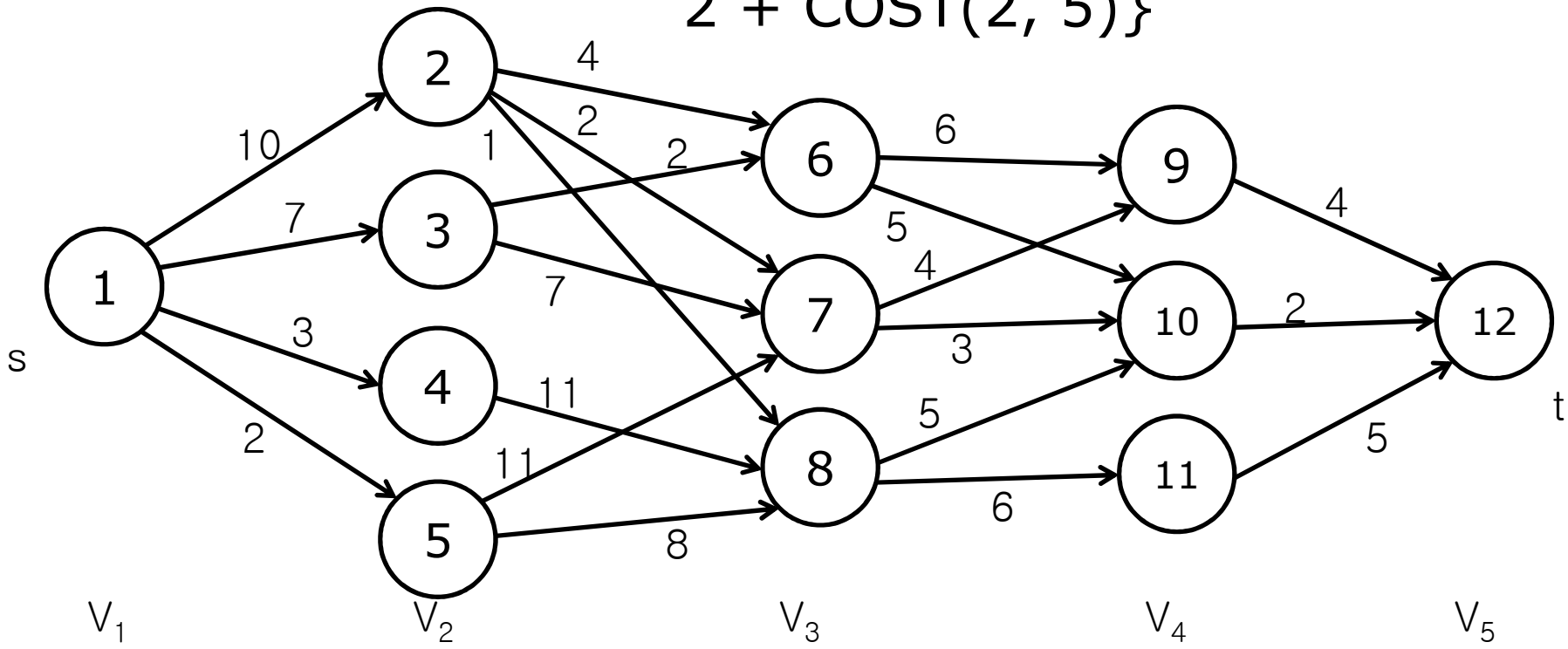
4.3 Multistage graph

- Algorithm

```
void MULTIGRAPH ( Graph G)
{
    int **COST, **D;

    estimate COST(k - 1);
    for ( i = k - 2; i >= 1; i-- ) {
        for all vertices j at stage i
            COST(i, j) ← min(cost(j, l) + COST(i+1, l));
            D(i, j) ← l;
    }
}
```

4.3 Multistage graph

- Example (At V_1)
  - COST(1, 1) = min{10 + COST(2, 2),
    7 + COST(2, 3),
    3 + COST(2, 4),
    2 + COST(2, 5)}



4.3 Multistage graph

- Example (At V_2)
  - COST(2, 2) = min{4 + COST(3, 6),
    2 + COST(3, 7),
    1 + COST(3, 8)}
  - COST(2, 3) = min{2 + COST(3, 6),
    7 + COST(3, 7)}
  - COST(2, 4) = min{11 + COST(3, 8)}
  - COST(2, 5) = min{11 + COST(3, 7),
    8 + COST(3, 8)}

4.3 Multistage graph

- Example (At V_3)
  - COST(3, 6) = min{6 + COST(4, 9),
    5 + COST(4, 10)}
  - COST(3, 7) = min{4 + COST(4, 9),
    3 + COST(4, 10)}
  - COST(3, 8) = min{5 + COST(4, 10),
    6 + COST(4, 11)}

4.3 Multistage graph

- Example (At V_4)
  - COST(4, 9) = 4
  - COST(4, 10) = 2
  - COST(4, 11) = 5

4.3 Multistage graph

- Example (Again at V_3)
  - COST(3, 6) = min{6 + COST(4, 9),
    5 + COST(4, 10)}
    = min{6 + 4, 5 + 2} = 7
  - COST(3, 7) = min{4 + COST(4, 9),
    3 + COST(4, 10)}
    = min{4 + 4, 3 + 2} = 5
  - COST(3, 8) = min{5 + COST(4, 10),
    6 + COST(4, 11)}
    = min{5 + 2, 6 + 5} = 7

4.3 Multistage graph

- Example (Again at V_2)
  - COST(2, 2) = min{4 + COST(3, 6),
    2 + COST(3, 7),
    1 + COST(3, 8)}
    = min{4 + 7, 2 + 5, 1 + 7} = 7
  - COST(2, 3) = min{2 + COST(3, 6),
    7 + COST(3, 7)}
    = min{2 + 7, 7 + 5} = 9
  - COST(2, 4) = min{11 + COST(3, 8)} = 11 + 7 = 18
  - COST(2, 5) = min{11 + COST(3, 7),
    8 + COST(3, 8)}
    = min{11 + 5, 8 + 7} = 15

4.3 Multistage graph

- Example (Again at V_1)
  - COST(1, 1) = min{10 + COST(2, 2),
    7 + COST(2, 3),
    3 + COST(2, 4),
    2 + COST(2, 5)}
    = min{10 + 7, 7 + 9, 3 + 18, 2 + 15}
    = 16
  - P(1, 1) = 1 → 3 → 6 → 10 → 12

4.3 Multistage graph

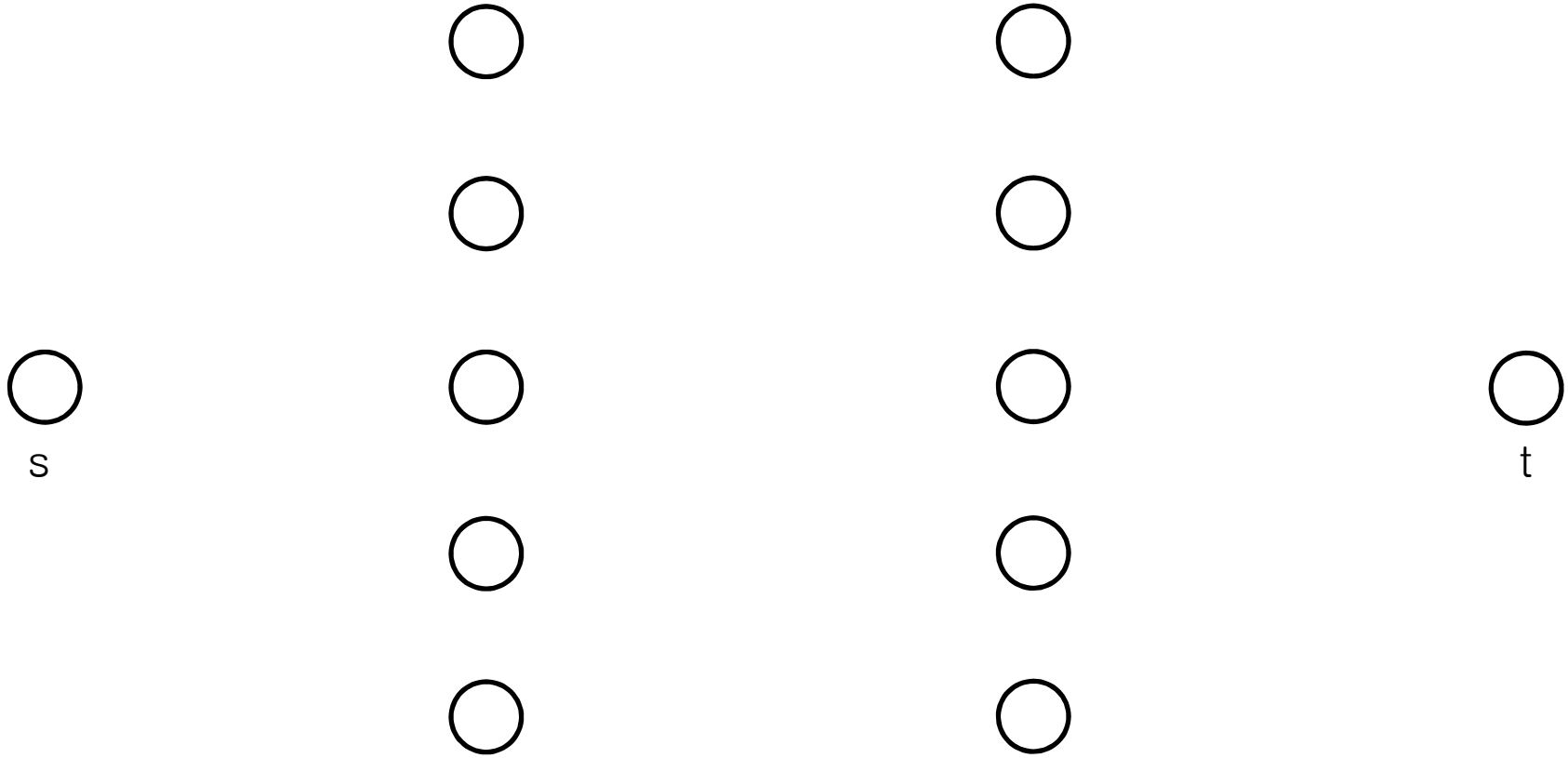
- Application: Resource allocation problem
  - Allocating n units of resources to r projects
  - N(i, j)
    - The net profit of allocating j units of resources to project i.
  - How to maximize total net profit?
  - Formulate this problem using multistage graph

4.3 Multistage graph

- Application: Resource allocation problem
  - Multistage graph
    - (r+1) stage graph problem
    - Stage i represents project i.
    - (n+1) vertices V(i, j) are associated with stage i.
    - Stage 1 and r+1 has one vertex: V(1, 0) = s and V(r+1, n) = t.
    - Vertex V(i, j) represents the stage in which a total of j units of resources have been allocated to projects 1, 2, …, i-1.
    - The edges are of the form <V(i, j), V(i+1, l)>.
      - The cost of the edges is N(i, l-j).

4.3 Multistage graph

- Application: Resource allocation problem
  - Multistage graph of 4 units to 3 teams



4.3 Multistage graph

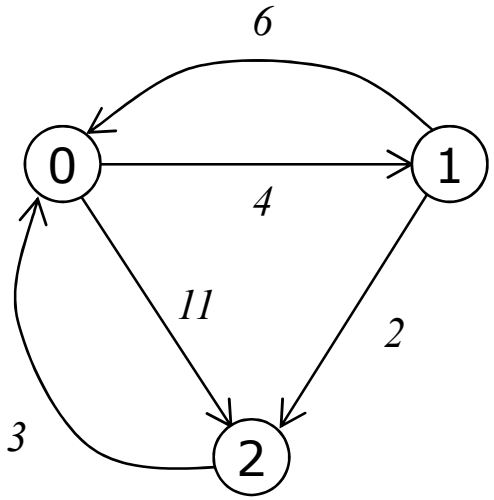
- Example
  - 4 units to 3 teams

	1st team	2nd team	3rd team
N=0	0	5	3
N=1	10	15	5
N=2	20	20	10
N=3	25	22	15
N=4	25	25	25

4.4 All Pairs Shortest path

- Problem: All-pairs shortest path
  - The problem of finding shortest paths for every two vertices *v* to *u*.
  - Solving single-source shortest path for all vertices in G

- Example



	0	1	2
0			
1			
2			

4.4 All Pairs Shortest path

- Strategy
  - Suppose we wish to find a shortest path from vertex i to vertex j.
  - Let A_i be the vertices adjacent to i.
  - Which of the vertices in A_i should be the second vertex?
  - Principle of optimality

Whatever the initial state and decision are, the remaining decisions must constitute an optimal decision sequence

- Let $A_1 = \{i_x\}$.
- The optimal path will be i → i_x → … → j.
- Whatever i_x will be, i_x → j must be optimal.

$$Cost(i, i_1, i_2, \ldots, j) = \min(Cost(i, i_x) + Cost(i_x, \ldots, j))$$

4.4 All Pairs Shortest path

- Strategy
  - Let k be an intermediate vertex on a shortest path from i to j such that {i → i_1 → i_2 → … → k → p_1 → p_2 → … → j}.
  - The path {i → i_1 → i_2 → … → k} and {k → p_1 → p_2 → … → j} should be shortest from i to k and k to j.

$$Cost(i, i_1, i_2, ..., k, p_1, p_2, ..., j)$$
$$= Cost(i, i_1, i_2, ..., k,) + Cost(k, p_1, p_2, ..., j)$$

4.4 All Pairs Shortest path

- Algorithm: Floyd's algorithm
  - Finding the all-pair's shortest path.
    - Input: adjacency matrix of a graph.
    - Output: minimum cost distance + path.
  - The weight of a path between two vertices is the sum of the weights of the edges along that path.
    - Negative weight is allowed.
    - Negative cycle is not allowed.

4.4 All Pairs Shortest path

- Algorithm: Floyd's algorithm
  - $A^k[i][j]$:
    - The cost of the shortest path from vertex i to j, using only those intermediate vertices with an index ≤ k.
  - $A^{-1}[i][j]$: the weight of an edge connecting vertex i and vertex j

Minimum cost with those vertices less than k

Minimum cost from i to k those vertices less than k

Minimum cost from k to j those vertices less than k

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

$$A^{-1}[i][j] = w[i][j]$$

Contents

Contents