Data Structure & Algorithms

Lec 17: Algorithms Design Techniques (Dynamic Programming)

Fall 2017 - University of Windsor
Dr. Sherif Saad

Outlines

Introduction To Dynamic Programming

Dynamic Programming Approach

Solving Problems with Dynamic Programming

Dynamic Programming: Motivation

Computing the n^{th} Fibonacci number recursively:

$$F(n) = F(n-1) + F(n-2)$$

$$F(0) = 0$$

$$F(1) = 1$$

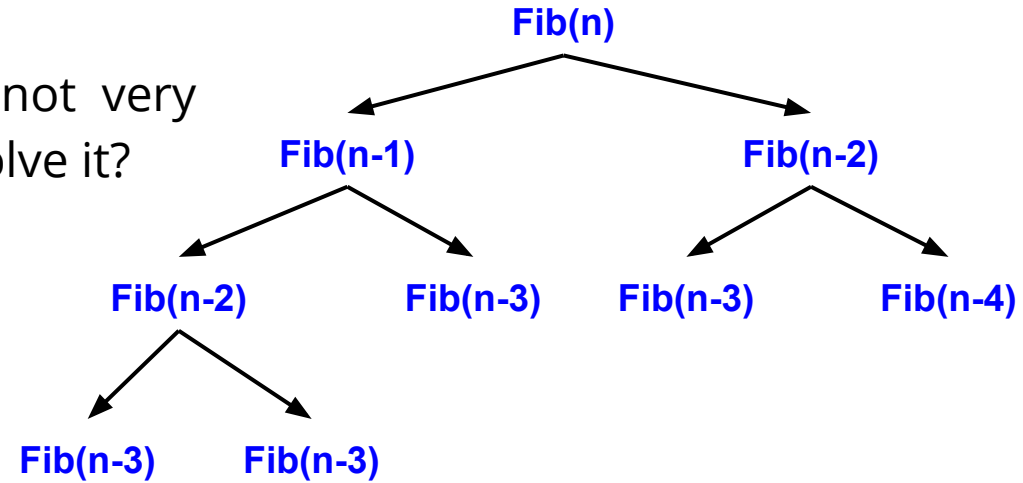
```
12 int Fib(int n){
13
14     if (n==0)
15         return 0;
16
17     if (n==1)
18         return 1;
19
20     return Fib(n-1) + Fib(n-2);
21 }
22
23 int main() {
24
25     cout<<Fib(10);
26     return 0;
27 }
```

Dynamic Programming: Motivation

The recurrence implementation of the Fibonacci number gives:

$T(n) = T(n-1) + T(n-2) +1 = O(2^n)$

It an exponential cost which is not very good, Is there is a better way to solve it?

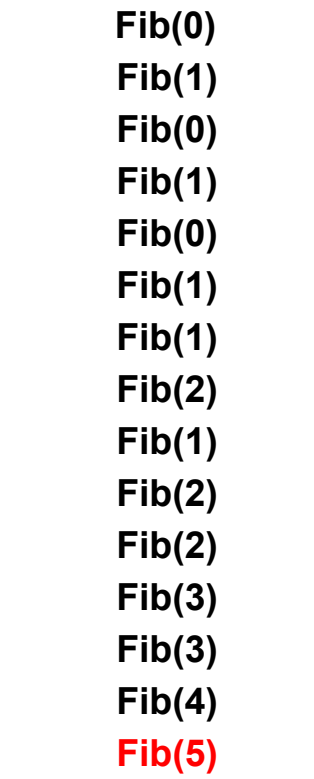


Dynamic Programming: Motivation

The top-down approach to compute the nth Fibonacci number is inefficient, Why??

We had to re-computer the same subproblem more than one time. As we can see from the execution stack on the right, to compute the Fibonacci(5), we calculate Fib(3) two times and Fib(2) three times, which lead to exponential time complexity.

It is clear that we have overlapping subproblems. Could we avoid recomputing or resolving subproblems that we already solved



Dynamic Programming: Definition

Dynamic Programming is a powerful algorithm design technique for solving optimization problems: often minimizing or maximizing.

DP solves problems by combining solutions to sub-problems and avoid recomputing or resolving subproblems we already solved using memoization

The term Dynamic Programming comes from **Control Theory**, not computer science; it is a multistage decision process.

Programming has nothing to do with coding; it refers to the use of tables to construct a solution and the word **Dynamic** to make it sounds impressive.

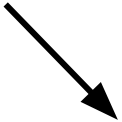
Dynamic Programming: When?

How do we recognize if a dynamic programming could be used for a given problem or not?

- Optimal substructures: an optimal solution to a problem consists of optimal solutions to a finite number of subproblems
- Overlapping Subproblems: a recursive solution contains an x number of distinct subproblems repeated many times.

Dynamic Programming: General Approach

Dynamic Programming = Recursion + Memoization



Recursion:
divide a large problem to a set of subproblems and solve subproblems recursively

Memoization:
an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again

Dynamic Programming: General Approach

In general, there are two approaches for solving DP problems:

- **Top-Down DP:** recursively break-down the problem into subproblems and solve the subproblems. The result of the recursive call is saved (memorized) and before we execute a recursive call we check if the result already know or not
- **Bottom-Up DP:** Try solving the subproblems first and use their solutions to build-on and arrive at solutions to bigger subproblems. Store the results of the smaller subproblems and iteratively generating solutions to bigger and bigger sub-problems by using the solutions to small sub-problems.

Dynamic Programming: Fibonacci Top-Down

```
int fib[MAX_N];

int Fib(int n){

    if (n ==0)
        return 0;
    if (n==1)
        return 1;
    else if (fib[n] != 0)
        return fib[n];
    return fib[n] = Fib(n-1) + Fib(n-2);
}

int main() {
    for(int i =0; i <MAX_N; i++){
        fib[i] =0;

    }
    cout<<Fib(99);
    return 0;
}
```

$T(n) = O(n)$

Dynamic Programming: Fibonacci Button-Up

```
12 int Fib(int n){
13
14     if (n==1 || n== 0)
15         return n;
16
17     fib[0] = 1;
18     fib[1] = 1;
19
20     for (int i=2; i<n; i++){
21         fib[i] = fib[i-1] + fib[i-2];
22     }
23
24     return fib[n-1];
25 }
26
27 int main() {
28     for(int i =0; i <MAX_N; i++){
29         fib[i] =0;
30
31     }
32     cout<<Fib(10);
33     return 0;
34 }
```

T(n) = O(n)

Dynamic Programming: Fibonacci Number

```
int Fib(int n){

    if (n==1 || n== 0)
        return n;

    fib[0] = 1;
    fib[1] = 1;

    for (int i=2; i<n; i++){
        fib[i] = fib[i-1] + fib[i-2];
    }

    return fib[n-1];
}

int main() {
    for(int i =0; i <MAX_N; i++){
        fib[i] =0;

    }
    cout<<Fib(10);
    return 0;
}
```

Button-Up

```
int fib[MAX_N];

int Fib(int n){

    if (n ==0)
        return 0;
    if (n==1)
        return 1;
    else if (fib[n] != 0)
        return fib[n];
    return fib[n] = Fib(n-1) + Fib(n-2);
}

int main() {
    for(int i =0; i <MAX_N; i++){
        fib[i] =0;

    }
    cout<<Fib(99);
    return 0;
}
```

Top-Down

Dynamic Programming: 0-1 Knapsack

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.

$$maximize\left\{f(x)\right\} = maximize\left\{\sum_{i=1}^{n} x_i p_i\right\}$$

$$subject - to = \sum_{i=1}^{n} x_i w_i \leq W \qquad x_i \in \left\{0,1\right\}, i = 1,2,3,\ldots,n$$

Dynamic Programming: 0-1 Knapsack

Assume we have the following set of items and a knapsack of capacity = 10. Find the items that maximize the profit with the capacity limit

Item	A	B	C
Profit	9	7	8
Weight	5	4	6

0-1 Knapsack: Brute Force Solution

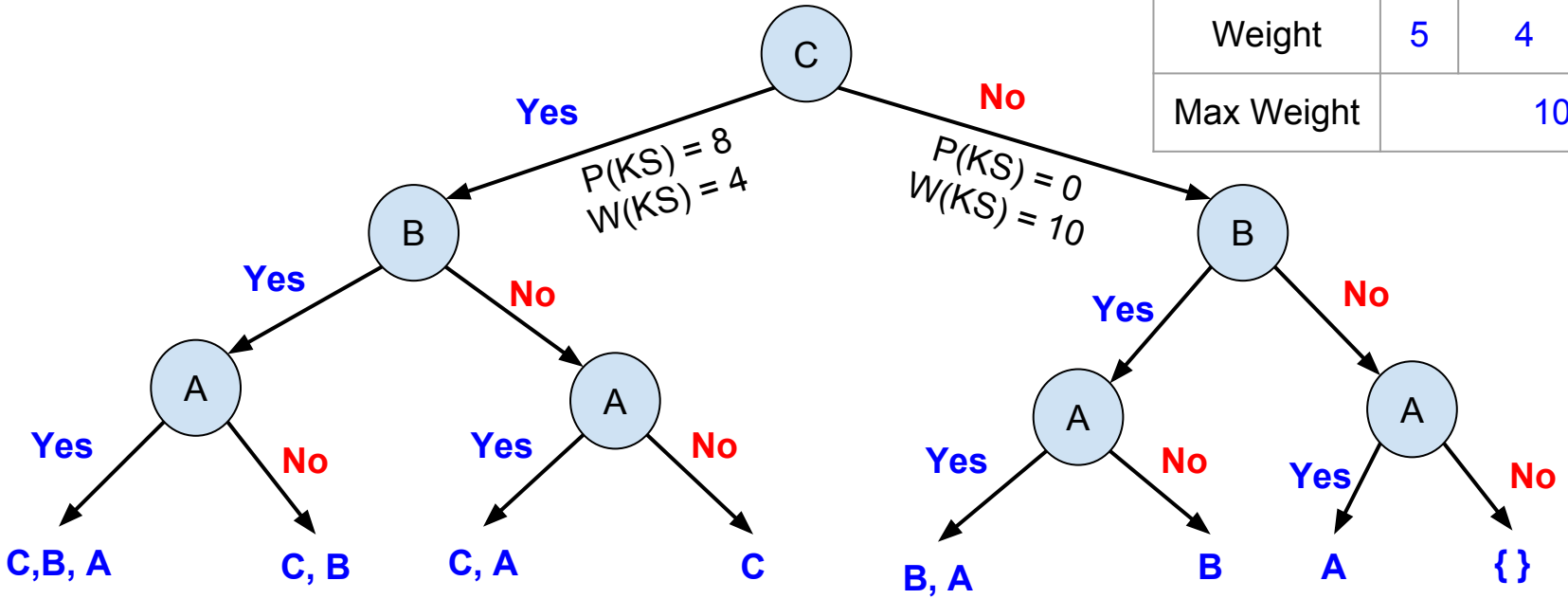
The naive method is to consider all $\mathbf{2^n}$ possible subsets of the n objects and choose the one that fits into the knapsack and maximizes the profit.

To consider all possible subsets of items, there can be two cases for every item:

1. The item is included in the optimal subset.
2. The item is not included in the optimal set.

0-1 Knapsack: Brute Force Solution

Item	A	B	C
Profit	9	7	8
Weight	5	4	6
Max Weight	10		



0-1 Knapsack: Brute Force Solution

The maximum value that can be obtained from n items is the max of the following two options:

1- Maximum value obtained by n-1 items and W weight (excluding nth item).

2- Value of nth item plus maximum value obtained by n-1 items and W minus weight of the nth item (including nth item).

0-1 Knapsack: Brute Force Solution

```
1. algorithm Knapsack(n, C):
2.    if n == 0 ||   C == 0
3.        result = 0
4.    else if weight(n) > C:
5.        result = Knapsack(n-1, C)
6.    else:
7.        opt1 = Knapsack(n-1, c)
8.        opt2 = value(n) + Knapsack(n-1, C- weight(n))
9.        result = max(opt1, opt2)
10.   return result
```

0-1 Knapsack: Dynamic Programming

Construct a table V of size n.c in row-major order.

Filling an entry in a row requires two entries from the previous row: one from the same column and one from the column offset by the weight of the object corresponding to the row.

$$V[i, w] = max(V[i - 1, w], v_i + V[i - 1, w - w_i]$$

Computing each entry takes constant time; the sequential run time of this algorithm is O(nc).

0-1 Knapsack: Dynamic Programming

Item	A	B	C
Profit	9	7	8
Weight	5	4	6
Max Weight	10		

$$V[i, w] = max(V[i - 1, w], v_i + V[i - 1, w - w_i]$$

$$v_i + V[i - 1, w - w_i] = -\infty \quad w_i > w$$

V[i, w]	w =0	1	2	3	4	5	6	7	8	9	10
i = 0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	9	9	9	9	9	9
2	0	0	0	0	7	9	9	9	9	16	16
3	0	0	0	0	7	9	9	9	9	16	16

0-1 Knapsack: Dynamic Programming

```
1. algorithm Knapsack(n, C):
2.     if V[n,c] != nil
3.         return V[n,c]
4.     if n == 0 ||   C == 0
5.         result = 0
6.     else if weight(n) > C:
7.         result = Knapsack(n-1, C)
8.     else:
9.         opt1 = Knapsack(n-1, c)
10.        opt2 = value(n) + Knapsack(n-1, C- weight(n))
11.        result = max(opt1, opt2)
12.    V[n,c] = result
13.    return result
```

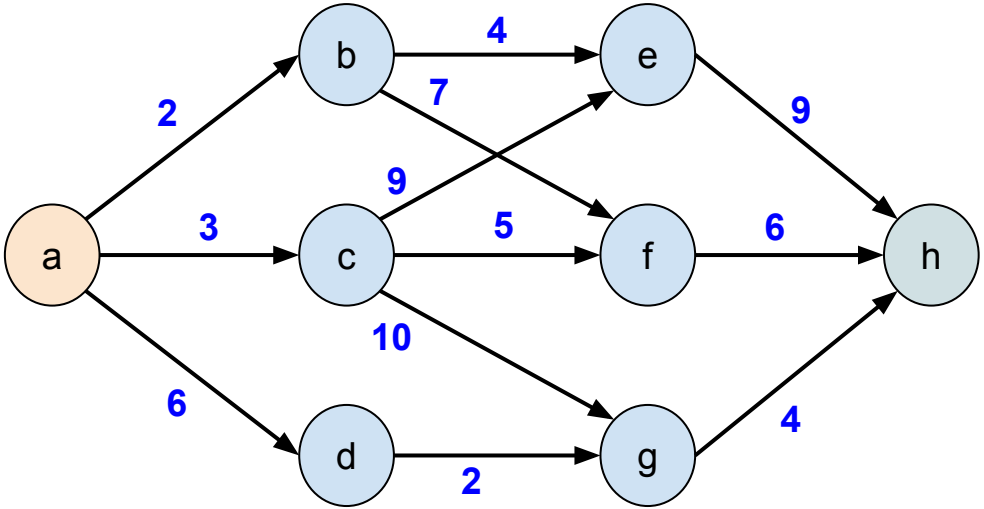
0-1 Knapsack: Dynamic Programming

```
1. algorithm Knapsack(n, C):
2.    if n == 0 ||   C == 0
3.       result = 0
4.    else if weight(n) > C:
5.       result = Knapsack(n-1, C)
6.    else:
7.       opt1 = Knapsack(n-1, c)
8.       opt2 = value(n) + Knapsack(n-1, C- weight(n))
9.       result = max(opt1, opt2)
10.   return result
```

```
1. algorithm Knapsack(n, C):
2.    if V[n,c] != nil
3.       return V[n,c]
4.    if n == 0 ||   C == 0
5.       result = 0
6.    else if weight(n) > C:
7.       result = Knapsack(n-1, C)
8.    else:
9.       opt1 = Knapsack(n-1, c)
10.      opt2 = value(n) + Knapsack(n-1, C- weight(n))
11.      result = max(opt1, opt2)
12.   V[n,c] = result
13.   return result
```

Dynamic Programming: Shortest Path

Given the following weighted directed graph find the shortest path between a and h



Using a greedy approach, the shortest path is {a-b-e-h} with total cost of 15

Dynamic Programming: Shortest Path

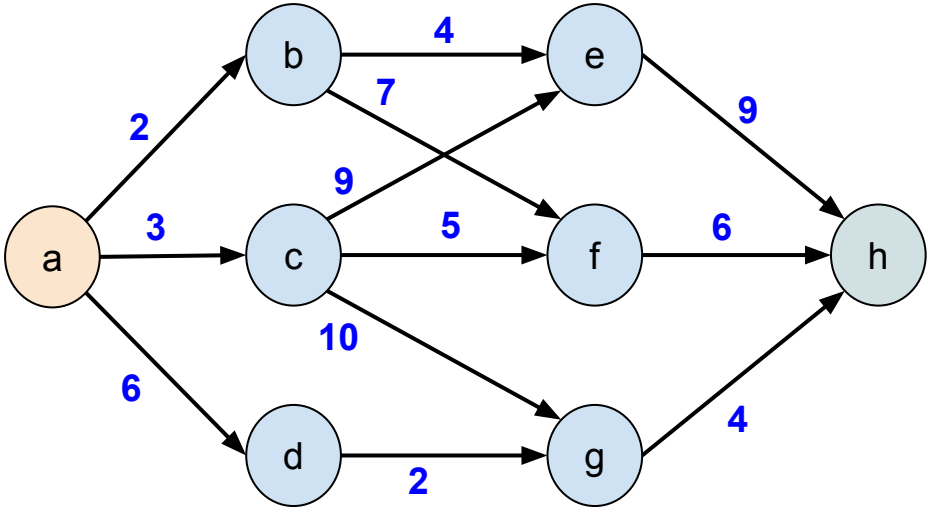
Using Dynamic Programming we can think of the solution as follows:

dis(a, h) = min{ 2 + dis(b, h), 3 + dis(c, h), 6 + dis(d, h) }

dis(b, h) = min{ 4 + dis(e, h), 7 + dis(f, h) }

dis(c, h) = min{ 9 + dis(e, h), 5 + dis(f, h), 10 + dis (g, h) }

dis(d, h) = min (2 + dis(g, h))



Dynamic Programming: All-Pairs Shortest Paths

Floyd-Warshall algorithm use dynamic programming to compute the shortest distance between every single pair of vertices.

The algorithm works with weighted graph (directed|undirected). It runs in **$O(n^3)$**, where n is the number of vertices in G, n = |V|

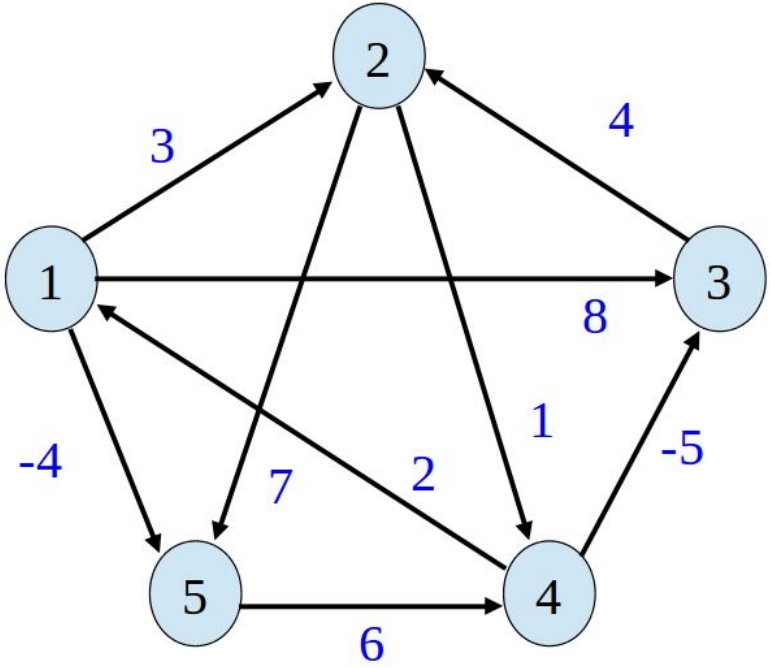
It works with negative-weight edges as along as there is no negative weight cycles.

$$d_{ij}(k) = \begin{cases} w_{ij} & k = 0 \\ min(d_{ij}(k-1), d_{ik}(k-1) + d_{kj}(k-1)) & k \geq 1 \end{cases}$$

Dynamic Programming: All-Pairs Shortest Paths

```
1. algorithm FloyedWarshall(V, E)
2.    for i = 1 to |V|
3.       for j = to |V|
4.          dist[i,j] = e.weight(i,j), // ∞ if no direct edge
5.          path[i,j] = nil
6.    for k =1 to |V|
7.       for i =1 to |V|
8.          for j =1 to |V|
9.             if (dist[i, k]+ dist[k, j]) < dist[i,j]
10.               dist[i, j] = dist[i, k] + dist[k, j]
11.               path[i, j] = path[i, k]
12.   return dist
```

Dynamic Programming: All-Pairs Shortest Paths



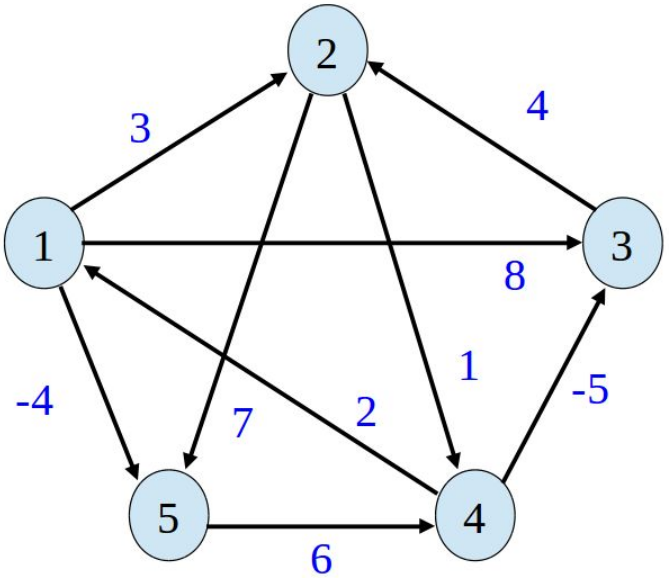
Dynamic Programming: All-Pairs Shortest Paths

0	3	8	∞	-4
∞	0	∞	1	7
∞	4	0	∞	∞
2	∞	-5	0	∞
∞	∞	∞	6	0

Dis (k=0)

nil	1	1	nil	1
nil	nil	nil	2	2
nil	3	nil	nil	nil
4	nil	4	nil	nil
nil	nil	nil	5	nil

Paths (k=0)



$$d_{ij}(k) = \begin{cases} w_{ij} & k = 0 \\ min(d_{ij}(k-1), d_{ik}(k-1) + d_{kj}(k-1)) & k \geq 1 \end{cases}$$

0	3	8	∞	-4
∞	0	∞	1	7
∞	4	0	∞	∞
2	∞	-5	0	∞
∞	∞	∞	6	0

Dis (k=0)

nil	1	1	nil	1
nil	nil	nil	2	2
nil	3	nil	nil	nil
4	nil	4	nil	nil
nil	nil	nil	5	nil

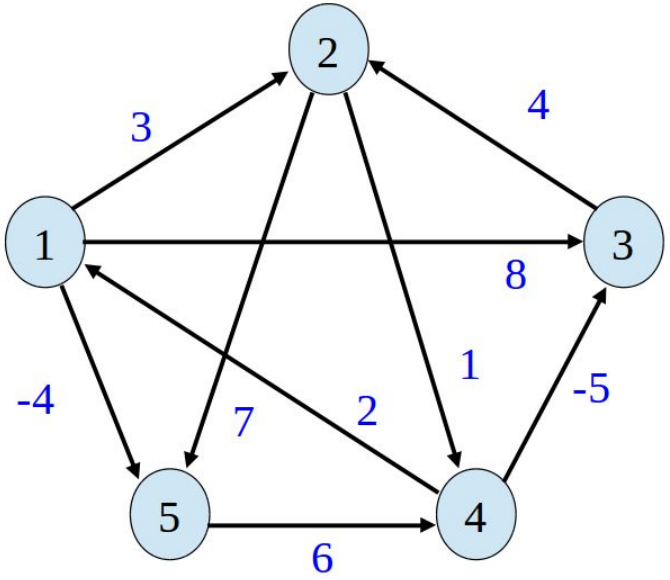
Paths (k=0)

0	3	8	∞	-4
∞	0	∞	1	7
∞	4	0	∞	∞
2	5	-5	0	-2
∞	∞	∞	6	0

Dis (k=1)

nil	1	1	nil	1
nil	nil	nil	2	2
nil	3	nil	nil	nil
4	1	4	nil	1
nil	nil	nil	5	nil

Paths (k=1)



$$d_{ij}(k) = \begin{cases} w_{ij} & k = 0 \\ min(d_{ij}(k-1), d_{ik}(k-1) + d_{kj}(k-1)) & k \geq 1 \end{cases}$$

0	3	8	∞	-4
∞	0	∞	1	7
∞	4	0	∞	∞
2	5	-5	0	-2
∞	∞	∞	6	0

Dis (k=1)

nil	1	1	nil	1
nil	nil	nil	2	2
nil	3	nil	nil	nil
4	1	4	nil	1
nil	nil	nil	5	nil

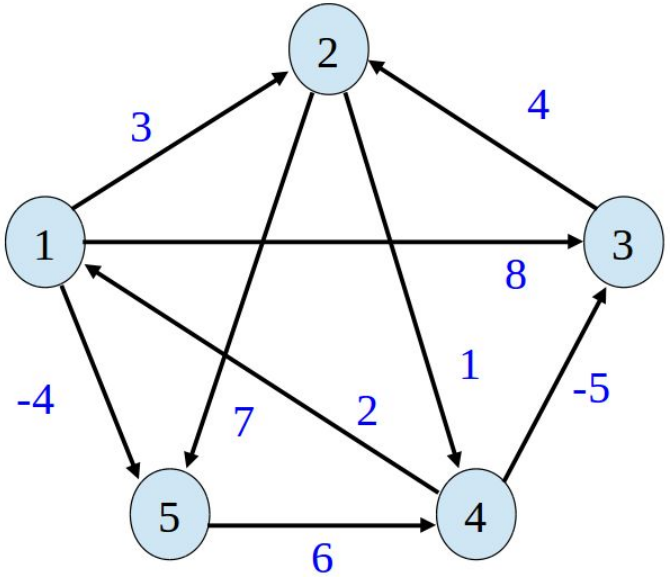
Paths (k=1)

0	3	8	4	-4
∞	0	∞	1	7
∞	4	0	5	11
2	5	-5	0	-2
∞	∞	∞	6	0

Dis (k=2)

nil	1	1	2	1
nil	nil	nil	2	2
nil	3	nil	2	2
4	1	4	nil	1
nil	nil	nil	5	nil

Paths (k=2)



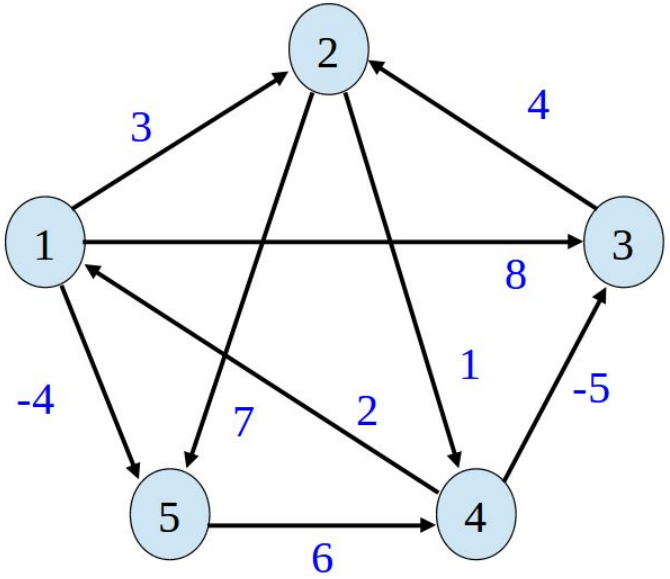
$$d_{ij}(k) = \begin{cases} w_{ij} & k = 0 \\ min(d_{ij}(k-1), d_{ik}(k-1) + d_{kj}(k-1)) & k \geq 1 \end{cases}$$

0	3	8	4	-4
∞	0	∞	1	7
∞	4	0	5	11
2	5	-5	0	-2
∞	∞	∞	6	0

Dis (k=2)

nil	1	1	2	1
nil	nil	nil	2	2
nil	3	nil	2	2
4	1	4	nil	1
nil	nil	nil	5	nil

Paths (k=2)



0	3	8	4	-4
∞	0	∞	1	7
∞	4	0	5	11
2	-1	-5	0	-2
∞	∞	∞	6	0

Dis (k=3)

nil	1	1	2	1
nil	nil	nil	2	2
nil	3	nil	2	2
4	3	4	nil	1
nil	nil	nil	5	nil

Paths (k=3)

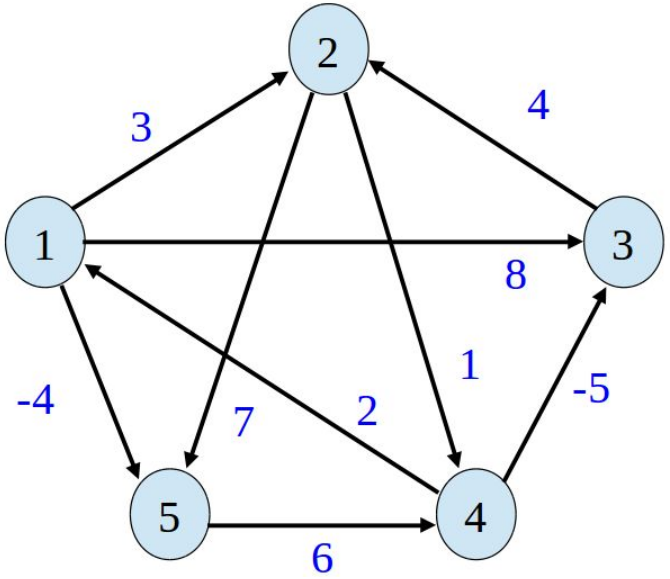
$$d_{ij}(k) = \begin{cases} w_{ij} & k = 0 \\ min(d_{ij}(k-1), d_{ik}(k-1) + d_{kj}(k-1)) & k \geq 1 \end{cases}$$

0	3	8	4	-4
∞	0	∞	1	7
∞	4	0	5	11
2	-1	-5	0	-2
∞	∞	∞	6	0

Dis (k=3)

nil	1	1	2	1
nil	nil	nil	2	2
nil	3	nil	2	2
4	3	4	nil	1
nil	nil	nil	5	nil

Paths (k=3)



0	3	-1	4	-4
3	0	-4	1	-1
7	4	0	5	3
2	-1	-5	0	-2
8	5	1	6	0

Dis (k=4)

nil	1	4	2	1
4	nil	4	2	1
4	3	nil	2	1
4	3	4	nil	1
4	3	4	5	nil

Paths (k=4)

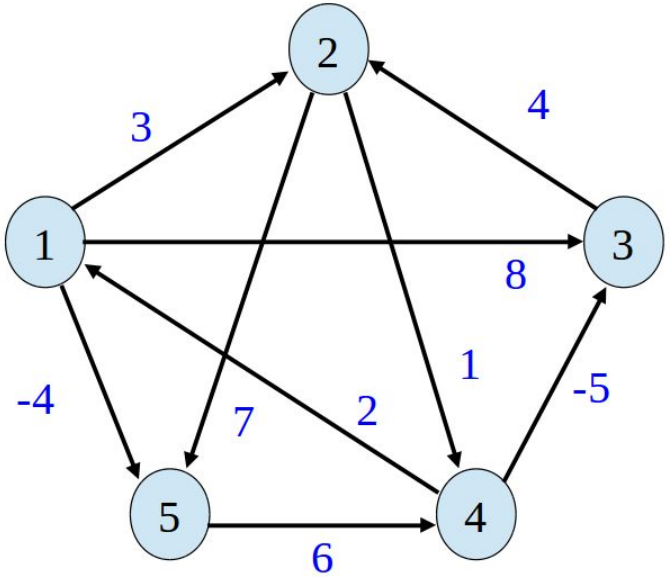
$$d_{ij}(k) = \begin{cases} w_{ij} & k = 0 \\ min(d_{ij}(k-1), d_{ik}(k-1) + d_{kj}(k-1)) & k \geq 1 \end{cases}$$

0	3	-1	4	-4
3	0	-4	1	-1
7	4	0	5	3
2	-1	-5	0	-2
8	5	1	6	0

Dis (k=4)

nil	1	4	2	1
4	nil	4	2	1
4	3	nil	2	1
4	3	4	nil	1
4	3	4	5	nil

Paths (k=4)



0	1	-3	2	-4
3	0	-4	1	-1
7	4	0	5	3
2	-1	-5	0	-2
8	5	1	6	0

Dis (k=4)

nil	3	4	5	1
4	nil	4	2	1
4	3	nil	2	1
4	3	4	nil	1
4	3	4	5	nil

Paths (k=4)

$$d_{ij}(k) = \begin{cases} w_{ij} & k = 0 \\ min(d_{ij}(k-1), d_{ik}(k-1) + d_{kj}(k-1)) & k \geq 1 \end{cases}$$

Dynamic Programming

Summary

1. Problem with optimal substructures
2. Overlapping subproblems
3. Recursive Solution
4. Memorize Intermediate Results to avoid unnecessary computations.