Ch. 6 Dynamic Programming 동적 계획법

Divide-and-conquer, graph exploration, greedy choice 등 특정한 문제 해결을 위한 알고리즘이 아닌 두 개의 알고리즘계의 양손망치(오함마)에 대하여 알아봄

-Dynamic programming

-Linear programming

폭넓게 적용할 수 있고 일반적인 방법에 비해 더 효율적

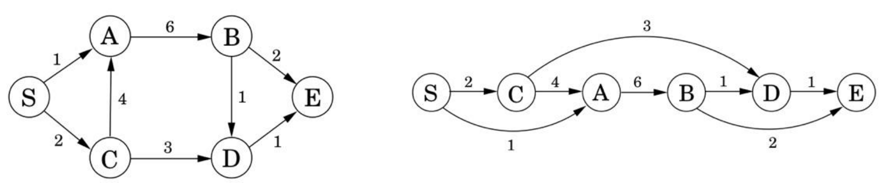
6.1 Shortest paths in dags, revisited

Dag 의 특징: 노드가 linearized 될 수 있음.

D 로 가는 가장 짧은 길을 찾는다면,

Dist(D) = min{dist(B)+1, dist(C)+3}

Figure 6.1 A dag and its linearization (topological ordering).



Node v 에 도착하면 dist(v)를 계산하기 위한 모든 정보를 가지게 됨

그래서 single pass 로 모든 길이를 계산할 수 있음

initialize all dist$(\cdot)$ values to ∞
dist$(s) = 0$
for each $v \in V \backslash \{s\}$, in linearized order:
    dist$(v) = \min_{(u,v)\in E}\{$dist$(u) + l(u,v)\}$

-Dynamic Programming 은 subproblem 들의 모음을 해결하는 것.

-가장 작은 dist(s)=0 을 해결한 뒤 더 큰 subproblems 들을 해결함

-해당 subproblem 에 도달하기 전에 다른 subproblem 을 여러 개를 해결해야 된다면 large subproblem 이라고 할 수 있음

-정리: Dynamic programming 은 subproblem 의 모음을 정의하고 가장 작은 것부터 하나씩 해결하는 것

-Dynamic programming 에서 dag 는 implicit 하기에 주어지지 않음.

6.2 Longest increasing subsequences 최장증가부분순열

Input: sequence of numbers, a1~an

Subsequence: any subset of numbers taken in order of the form ai1, ai2, …, aik where 1〈=i1〈i2…〈ik〈=n

Increasing subsequence: numbers are getting strictly larger

Task: Find the increasing subsequence of greatest length

5, 2, 8, 6, 3, 6, 9, 7 ➔ 2, 3, 6, 9

Node i for each element ai, directed edges (i, j), i〈j and ai 〈 aj

(1) Graph G=(V, E) is a dag, since all edges(I, j) have i〈j

(2) increasing subsequences 와 path 간의 1:1 대응

그러므로 우리의 목표는 dag 에서 가장 긴 path 를 찾는 것

`for` $j = 1, 2, \ldots, n$:
    $L(j) = 1 + \max\{L(i) : (i, j) \in E\}$
`return` $\max_j L(j)$

L(j)는 longest path=longest increasing subsequence

Node j 까지의 path 라면 predecessor 를 하나는 지나야 됨

그래서 L(j)는 1 + predecessors 들의 max L(.) value

우리의 원래 문제를 해결하려면 subproblem 의 모음 {L(j): 1〈=j〈=n}을 해결해야 됨

L(j) = 1 + max{L(i): (i, j) ∈ E}

How long does it take?

Requires the predecessors of j to be known: linear time

Computation of L(j): |E|

At most **O(n^2),** when input array is sorted in increasing order

How to recover subsequence itself?

While computing L(j), also note down **prev(j)**

6.3 Edit distance

두 문자열 사이의 거리: 정렬할 수 있는 범위

정렬: 다른 문자열 위에 문자열을 쓰는 방법

Edit distance 는 최소한의 수정 횟수로도 생각할 수 있음

S	—	N	O	W	Y
S	U	N	N	—	Y

Cost: 3

—	S	N	O	W	—	Y
S	U	N	—	—	N	Y

Cost: 5

A dynamic programming solution

목표: 두 문자열 사이의 edit distance 를 구하는 것.

subproblem 으로 첫번째와 두번째 문자열의 접두사의 edit distance 를 구하고(E(i, j)) 최종 목표인 E(m, n)을 구함

For this to work, we need to somehow express $E(i,j)$ in terms of smaller subproblems. Let's see—what do we know about the best alignment between $x[1\cdots i]$ and $y[1\cdots j]$? Well, its rightmost column can only be one of three things:

$$\begin{array}{c} x[i] \\ - \end{array} \quad \text{or} \quad \begin{array}{c} - \\ y[j] \end{array} \quad \text{or} \quad \begin{array}{c} x[i] \\ y[j] \end{array}$$

E(i, j)를 smaller subproblems 로 표현:

E(i, j) → E(i-1, j), E(i, j-1), E(i-1, j-1)

Subproblems 와 순서가 정해지면 base case 가 남아있음

Base case: 가장 작은 subproblems. Ex. E(0, ·), E(·, 0)

The underlying dag

Each node as a subproblem

Put **weights** on the edges

6.4 Knapsack 배낭문제 (조합최적화)

During a robbery, a burglar finds much more loot than he had expected and has to decide what to take. His bag (or "knapsack") will hold a total weight of at most W pounds. There are n items to pick from, of weight $w_1, \ldots, w_n$ and dollar value $v_1, \ldots, v_n$. What's the most valuable combination of items he can fit into his bag?[1]

For instance, take $W = 10$ and

Item	Weight	Value
1	6	$30
2	3	$14
3	4	$16
4	2	$9

Dynamic programming 을 사용하면 O(nW) 시간 안에 해결할 수 있음

Knapsack with repetition

1. Smaller knapsack capacities, 2. Fewer items

K(w) = maximum value achievable with a knapsack of capacity w

K(w)가 아이템 i 를 포함한다면, 아이템을 제외하면 optimal solution 이 K(w-wi)가 됨

K(w) = K(w-wi) + vi

$K(0) = 0$
for $w = 1$ **to** W**:**
  $K(w) = \max\{K(w - w_i) + v_i : w_i \leq w\}$
return $K(W)$

Knapsack without repetition

Add a second parameter 0〈=j〈=n

K(w, j) = max value achievable using a knapsack of capacity w and items 1, …, j

더 작은 subproblems 로 표현한다면?

$$K(w, j) = \max\{K(w - w_j, j - 1) + v_j, K(w, j - 1)\}.$$

```
Initialize all K(0,j) = 0 and all K(w,0) = 0
for j = 1 to n:
   for w = 1 to W:
      if w_j > w:   K(w,j) = K(w,j-1)
      else:   K(w,j) = max{K(w,j-1), K(w-w_j,j-1) + v_j}
return K(W,n)
```