제 3 장　동적 프로그래밍 (Dynamic Programming)

- 일반적인 방법
- 다단계 그래프
- 모든 쌍들의 최단 경로

일반적인 방법

- 일련의 결정들(sequence of decisions)의 결과로 해를 구하는 알고리즘 설계 방법
- 예제 1 (배낭 문제)

  x_i의 값을 차례로 결정해야 한다. 즉, x_1의 값을 결정하고, 다음으로 x_2의 값을 결정하고, x_3 등등. 결정들의 최적인 순서열은 목적 함수 $\Sigma p_i \cdot x_i$ 를 최대화하는 것이다.
- 예제 2 (최단 경로)

  정점 i 로부터 정점 j 로의 최단 경로를 찾는 하나의 방법은, 어느 정점이 두 번째 정점이 될지, 어느 정점이 세 번째 정점이 될지, … 등등과 같이 정점 j 에 도달할 때까지 계속 정점들을 결정해 나가는 것이다. 결정들의 최적 순서열은 최단 경로를 결과로 내는 것이다.

최적성 원칙 (Principle of Optimality)

- 중간 결정(decision)에 대하여 그 이전의 결정들과 그 이후의 결정들 모두가 optimal 이어야 한다.
- 욕심쟁이 방법과의 차이점

  욕심쟁이 방법: 오직 하나의 decision sequence 가 만들어진다.

  동적 프로그래밍: 많은 decision sequence(모든 가능성을 고려)가 만들어진다. 최적이 될 가능성이 없는 순서열은 생성하지 않게 한다.
- 예제 3 (최단 경로)

  정점 i 로부터 j 까지의 최단 경로가 $i, i_1, i_2, \cdots, i_k, j$ 라 하자.

  i 로부터 i_1 으로 가는 결정이 내려지면 i_1 에서 j 까지 최단 경로를 구해야 하며 이 최단 경로는 $i_1, i_2, \cdots, i_k, j$ 가 되어야만 한다.

- 예제 4 (0/1 배낭 문제)

  KNAP(l, j, y) 문제는 다음과 같이 정의된다.

  maximize $\Sigma_{l \le i \le j}\ p_i \cdot x_i$,

  subjet to $\Sigma_{l \le i \le j}\ w_i \cdot x_i \le y$ and

  $x_i = 0$ or 1, $l \le i \le j$

  이때 n 개의 물건에 대한 원래 문제는 KNAP(1, n, m) 이 된다.

- $y_1, y_2, \ldots, y_n$ 을 $x_1, x_2, \ldots, x_n$에 대한 0/1 값의 최적 순서열이라 하자.
- 만약 $y_1 = 0$ 이면, $y_2, y_3, \ldots, y_n$ 은 KNAP(2, n, m) 에 대해 최적의 순서열이 되어야만 한다.

  만약 $y_1 = 1$ 이면, $y_2, y_3, \ldots, y_n$ 은 KNAP(2, n, m−w_1) 에 대해 최적의 순서열이 되어야만 한다.

  → 최적성 원리 존재

중간 상태의 결정에서도 존재

- 예제 5 (최단 경로)
  정점 k 가 정점 i 로부터 j 까지의 최단 경로 사이의 한 정점이라면
  즉, $i, i_1, i_2, \cdots, k, p_1, p_2, \cdots, j$ 가 최단 경로라면,
  ➔
  경로 $i, i_1, i_2, \cdots, k$ 는 i 로부터 k 까지의 최단 경로이고,
  경로 $k, p_1, p_2, \cdots, j$ 는 k 로부터 j 까지의 최단 경로이다.
- 예제 6 (0/1 배낭)
  $y_1, y_2, \ldots, y_n$ 을 KNAP(1, n, m)의 최적 해라고 하자.
  y_j 에 대하여 $y_1, y_2, \ldots, y_j$ 은 KNAP$(1, j, \Sigma_{1\le i\le j}\, w_i \cdot y_i)$,
  $y_{j+1}, \ldots, y_n$ 은 KNAP$(j+1, n, m-\Sigma_{1\le i\le j}\, w_i \cdot y_i)$에 대해
  최적 해가 되어야만 한다.

forward 접근

- x_i 의 결정은 x_{i+1}, x_{i+2}, …, x_n 의 최적 결정에 따라 결정된다.
- 예: 0/1 배낭 문제에서

  $g_1(m)$ 을 KNAP(1, n, m) 에 대해 최적 해의 값이라 하자.

  $g_1(m) = \max \{ g_2(m), g_2(m-w_1)+p_1 \}$

  일반화시키면, (단, $g_{n+1}(y) = 0$ 이고 $g_i(y) = -\infty$, $y < 0$)

  $g_i(y) = \max \{ g_{i+1}(y), g_{i+1}(y-w_i)+p_i \}$
- 예제:

  $n = 3$, $w_i = (2, 3, 4)$, $p_i = (1, 2, 5)$, $m = 6$ 일 때, $g_1(6)$을 계산한다.

  $g_1(6) = \max \{ g_2(6), g_2(4)+1 \}$

  $g_2(6) = \max \{ g_3(6), g_3(3)+2 \}$

  $g_3(6) = \max \{ g_4(6), g_4(2)+5\} = \max \{0, 5\} = 5$

  $g_3(3) = \max \{ g_4(3), g_4(3-4)+5\} = \max \{0, -\infty\} = 0$

  따라서 $g_2(6) = \max \{ g_3(6), g_3(3)+2 \} = \max \{5, 2\} = 5$ 이다.

  $g_2(4) = \max \{ g_3(4), g_3(4-3)+2 \}$

  $g_3(4) = \max \{ g_4(4), \underline{g_4(4-4)+5}\} = \max \{0, 5\} = 5$

  $g_3(1) = \max \{ g_4(1), g_4(1-4)+5\} = \max \{0, -\infty\} = 0$

  따라서 $g_2(4) = \max \{ \underline{g_3(4)}, g_3(1)+2 \} = \max \{5, 2\} = 5$ 이다.

  원 식에서, $g_1(6) = \max \{ g_2(6), \underline{g_2(4)+1} \} = \max \{5, 5+1 \} = 6$.

  → $x_1 = 1$, $x_2 = 0$, $x_3 = 1$

backward 접근

- x_i 의 결정은 $x_1, x_2, \ldots, x_{i-1}$ 의 최적 결정에 따라 결정된다.
- 예: 0/1 배낭 문제에서

  $f_n(m)$ 을 KNAP(1, n, m) 에 대해 최적 해의 값이라 하자.

  $f_n(m) = \max \{ f_{n-1}(m), f_{n-1}(m-w_n)+p_n \}$

  일반화시키면, (단, $f_0(y) = 0$ 이고 $f_i(y) = -\infty$, $y < 0$)

  $f_j(y) = \max \{f_{j-1}(y), f_{j-1}(y-w_j)+p_j \}$
- 앞의 예제에 대해

  $n = 3$, $w_i = (2, 3, 4)$, $p_i = (1, 2, 5)$, $m = 6$ 일 때, $f_3(6)$을 계산한다.

  $f_3(6) = \max \{ f_2(6), f_2(2)+5 \}$

  $f_2(6) = \max \{ f_1(6), f_1(3)+2 \}$

  $f_1(6) = \max \{ f_0(6), f_0(4)+1\} = \max \{0, 1\} = 1$

  $f_1(3) = \max \{ f_0(3), f_0(3-2)+1\} = \max \{0, 1\} = 1$

  따라서 $f_2(6) = \max \{f_1(6), f_1(3)+2 \} = \max \{1, 3\} = 3$ 이다.

  $f_2(2) = \max \{ f_1(2), f_1(2-3)+2 \}$

  $f_1(2) = \max \{ f_0(2), \underline{f_0(2-2)+1}\} = \max \{0, 1\} = 1$

  따라서 $f_2(2) = \max \{ \underline{f_1(2)}, f_1(2-3)+2 \} = \max \{1, -\infty\} = 1$ 이다.

  원 식에서, $f_3(6) = \max \{f_2(6), \underline{f_2(2)+5} \} = \max \{3, 1+5 \} = 6$.

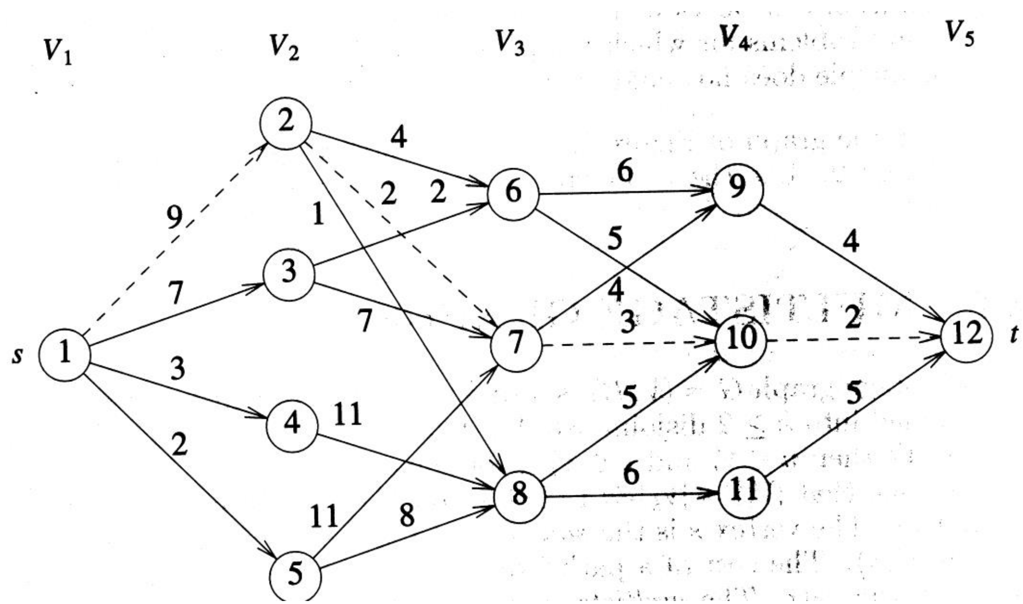
  → $x_3 = 1$, $x_2 = 0$, $x_1 = 1$

동적 프로그래밍의 일반적인 경향

- 최악의 경우, 모든 가능한 decision sequence를 살펴보게 된다.
  즉, $O(d^n)$ 시간복잡도를 갖게 된다.
  d는 선택 가능한 값의 수이며, 0/1 배낭 문제에서 d = 2 이다.
- 그러나 대부분의 경우 suboptimal 에 해당하는 decision 들을 제외시킴으로써 살펴보는 decision의 수를 대폭 줄일 수 있다.
- 그리고 recurrence 에 대한 계산을 매번 반복하는 대신에, 이전에 계산된 값을 이용함으로써 빠른 시간에 수행할 수도 있다.

1. 다단계 그래프 (multistage graphs)

- 다단계 그래프 G = (V, E)는 정점들이 k ≥ 2 개의 서로 소(disjoint)인 집합 V_i (1≤i ≤ k)로 분할된 방향 그래프이다.
- 예: 5-단계 그래프



- s: 출발지(source), t: 도착지(destination), c(i, j): 간선 <i, j>의 비용
- 다단계 그래프 문제는 s 에서 t 까지의 최소-비용 경로를 찾는 문제이다.

forward 방식

- k-단계 그래프 문제에 대한 동적 프로그래밍 구성은 s 에서 t 로의 모든 경로가 k-2 개의 결정(decision)들의 순서열이다.
- p(i, j)를 V_i 내의 정점 j 로부터 정점 t 로의 최소 비용 경로라 하고, cost(i, j)를 이 경로의 비용이라 하자.

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

- 여기서 cost(1, s)와 p(1, s)를 구하는 것이 목표이다.

  cost(1,s) 를 알기 위해 <s, j> ∈ E 인 모든 간선들에 대해 c(s, j)+cost(2, j)를 계산하여 최소값을 구해야 하며, cost(2, j)를 알기 위해 이러한 과정을 반복해야 한다.

그림(9번 슬라이드)의 예(forward)

- 최종으로, p(1,1)과 cost(1,1)을 구하려고 한다.
- V_4: cost(4,9) = 4, cost(4,10) = 2, cost(4,11) = 5
- V_3: cost(3,6) = min{6+cost(4,9), 5+cost(4,10)} = 7
  cost(3,7) = min{4+cost(4,9), 3+cost(4,10)} = 5
  cost(3,8) = min{5+cost(4,10), 6+cost(4,11)} = 7
- V_2: cost(2,2) = min{4+cost(3,6), 2+cost(3,7), 1+cost(3,8)} = 7
  cost(2,3) = 9, cost(2,4) = 18, cost(2,5) = 15
- V_1: cost(1,1) = min{9+cost(2,2), 7+cost(2,3), 3+cost(2,4),
  2+cost(2,5)} = 16
- d(i, j) = cost(i, j)의 최소값을 결정하는 l 값, 즉 <j, l> 간선 사용.

  d(4,9) = 12, d(4,10) = 12, d(4,11) = 12

  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

  ➔ 최소비용 경로를 s=1, v_2, v_3, v_4, t=12 라 할 때, v_2 = d(1,1) = 2,
  v_3 = d(2,2) = 7, v_4 = d(3,7) = 10 으로 결정된다.

시간복잡도

- 각 정점에서의 최소 비용 cost(i, j) 는 그 정점에서 진출하는 간선들의 c(j, l)+cost(i+1,l) 중 최소값으로 결정되므로, 총 시간은 그 정점에서 진출하는 간선의 수에 비례한다.
- 이 비용은 모든 정점에 대해 계산되므로 전체 시간은 O(|V| + |E|)에 해당한다. 최소 비용 경로에 대해서도 각 정점의 d 값을 이용해서 이 시간 내에서 결정할 수 있다.

backward 방식

- bp(i, j)를 정점 s로 부터 V_i 내의 정점 j 로의 최소 비용 경로라 하고, bcost(i, j)를 이 경로의 비용이라 하자.

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

- 여기서 bcost(k, t)와 bp(k, t)를 구하는 것이 목표이다. bcost(k,t) 를 알기 위해 <j, t> ∈ E 인 모든 간선들에 대해 bcost(k−1,j)+c(j,t) 를 계산하여 최소값을 구해야 하며, bcost(k−1, j)를 알기 위해 이러한 과정을 반복해야 한다.

그림의 예(backward)

- 최종으로, bp(5,12)과 bcost(5,12)를 구하려고 한다.
- V_2: bcost(2,2) = 9, bcost(2,3) = 7, bcost(2,4) = 3, bcost(2,5) = 2
- V_3: bcost(3,6) = min{bcost(2,2)+4, bcost(2,3)+2} = 9

  bcost(3,7) = 11

  bcost(3,8) = 10
- V_4: bcost(4,9) = min{bcost(3,6)+6, bcost(3,7)+9} = 15

  bcost(4,10) = 14, bcost(4,11) = 16
- V_5: bcost(5,12) = min{bcost(4,9)+4, bcost(4,10)+2, bcost(4,11)+5} = 16
- d(i, j) = bcost(i, j)의 최소값을 결정하는 I 값, 즉 <I, j> 간선 사용.
- 최소비용 경로도 forward 방식과 유사하게 구할 수 있다.
- 전체 시간 역시 O(|V| + |E|)에 해당한다.

2. 모든 쌍들의 최단 경로들

- G = (V, E)를 n 개의 정점들을 갖는 방향 그래프라 하고, cost는 간선들 상의 가중치를 갖는 인접 행렬이다.
- 모든 쌍들의 최단 경로 문제는 서로 다른 정점의 쌍 u와 v 간의 최단 경로의 길이를 구하는 문제이다(단 음수 길이의 사이클이 존재하지 않을 때). 즉, A(i, j)가 정점 i 로부터 j 까지의 최단 경로의 길이인 행렬 A 를 결정하는 것이다.
- 방법 1: 각 정점을 출발점으로 하여 Dijkstra 알고리즘을 n번 적용
- 방법 2: 동적 프로그래밍 적용(Ford 알고리즘)
- 위의 두 방법들 모두 시간복잡도가 $O(n^3)$ 이나 방법 2가 훨씬 작은 상수를 가진다.

동적 프로그래밍 기법

- 최적성 원칙: i 에서 j 로의 최단 경로에서 k가 이 최단 경로 상의 중간에 있는 정점이라면, i 에서 k로의 부분 경로와 k 에서 j로의 부분 경로가 각각 최단 경로가 되어야만 한다. 만약 그렇지 않다면 다른 경로가 최단 경로가 되기 때문이다.
- 원리:

  $A^k(i,j)$ = k 보다 큰 인덱스를 갖는 중간 정점을 통과하지 않고 i 에서 j 로 가는 최단 경로 길이

  $A^0, A^1, \ldots, A^n$ 을 순서대로 구할 때 A^n 이 모든 쌍의 최단 경로 길이가 된다
- 순환식:

  $A^0(i,j) = cost(i, j), 1 \le i \le n$ 이다.

- A^{k-1} 에서 A^k 를 만드는 방법:

  임의의 정점 쌍 i, j 에 대하여 다음의 두 규칙 중 한 가지를 적용한다.

  (i) 쌍 i, j의 최단 경로가 정점 k를 통과하지 않을 때

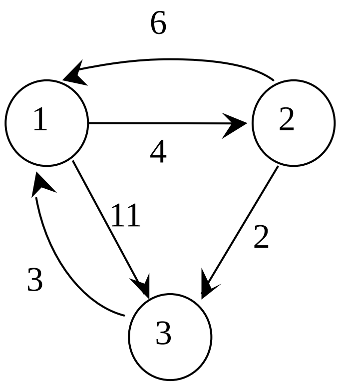
  $A^k(i,j) = A^{k-1}(i,j)$

  (ii) 쌍 i, j의 최단 경로가 정점 k를 통과할 때

  $A^k(i,j) = A^{k-1}(i,k) + A^{k-1}(k,j)$

  따라서 (i) (ii)를 한 식으로 나타내면,

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



(a) 예제 방향그래프

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

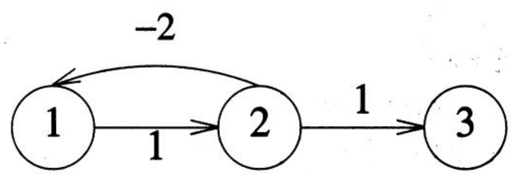
모든 쌍의 최단경로(lec8-1)

```
void AllPaths(float cost[][], float A[][], int 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 (int i=1; i<=n; i++)
        for (int j=1; j<=n; j++)
           A[i][j] = cost[i][j]; // Copy cost into A.
     for (int k=1; k<=n; k++)
        for (i=1; i<=n; i++)
           for (int j=1; j<=n; j++)
              A[i][j] = min(A[i][j], A[i][k]+A[k][j]);
  }
```

- 시간복잡도: $O(n^3)$

음인 길이의 순환 경로를 갖는 경우

- 이 경우는 AllPaths 알고리즘이 정확히 동작하지 않는다.
- 예제 5.14:



$$\begin{bmatrix} 0 & 1 & \infty \\ -2 & 0 & 1 \\ \infty & \infty & 0 \end{bmatrix}$$

- 이 그래프에서 $A^2(1,3) \neq \min\{ A^1(1,3), A^1(1,2)+A^1(2,3) \} = 2$
  실제 최단 경로의 비용은 $A^2(1,3) = -\infty$ 이어야 한다.
  경로 1,2,1,2,1,2, …,1,2,3 의 길이를 임의로 작게 만들 수 있기 때문이다.