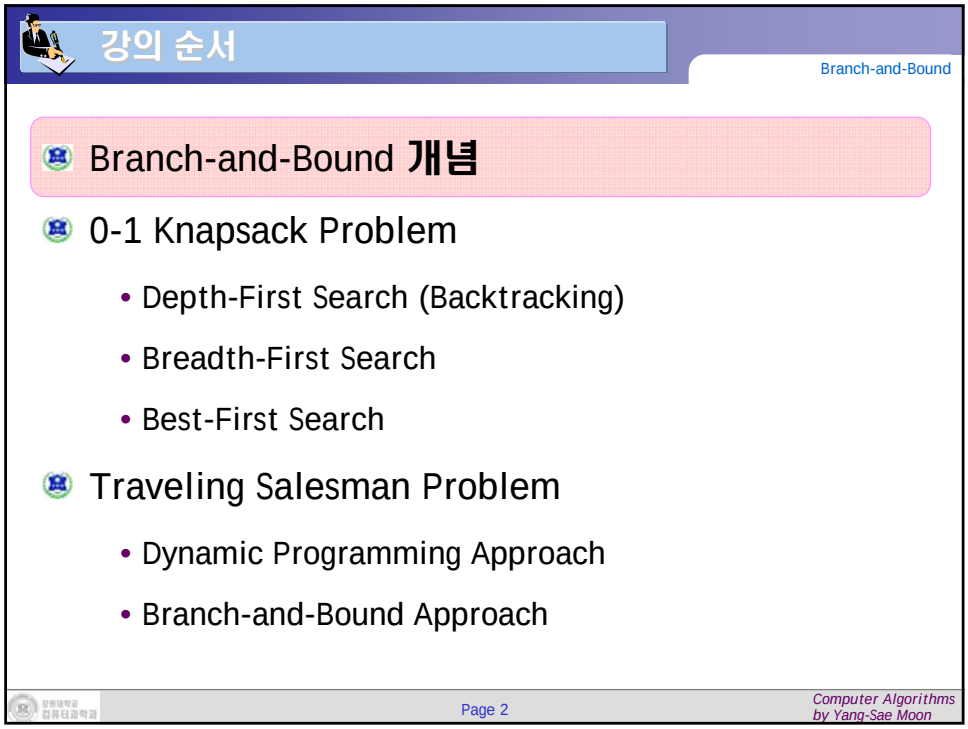

알고리즘(Algorithm) - Branch-and-Bound (분기한정)
문양세 (컴퓨터과학전공, IT특성화대학, 강원대학교)


강의 순서
Branch-and-Bound
Branch-and-Bound 개념
0-1 Knapsack Problem
Depth-First Search (Backtracking)
Breadth-First Search
Best-First Search
Traveling Salesman Problem
Dynamic Programming Approach
Branch-and-Bound Approach
Page 2
Computer Algorithms
by Yang-Sae Moon

Branch-and-Bound? – 특징

- 되추적 기법과 유사하게 상태공간트리를 구축하여 문제를 해결한다.
- 최적의 해를 구하는 문제(optimization problem)에 적용할 수 있다.
- 최적의 해를 구하기 위해서는 궁극적으로 모든 해를 다 고려해 보아야 한다.

→ 해를 찾거나 찾지 못하는 여부가 트리를 순회(traverse)하는 방법에 구애 받지는 않는다.

Branch-and-Bound? – 원리

- 각 노드를 검색할 때 마다, 그 노드가 유망한지의 여부를 결정하기 위해서 한계치(bound)를 계산한다.
- 그 한계치는 그 노드로부터 가지를 뻗어나가서(branch) 얻을 수 있는 해답치의 한계를 나타낸다.
- 따라서 만약 그 한계치가 지금까지 찾은 최적의 해답치 보다 좋지 않은 경우는 더 이상 가지를 뻗어서 검색을 계속할 필요가 없으므로, 그 노드는 유망하지 않다고 할 수 있다.
  (이 경우, 해당 서브트리를 전지(pruning)한다.)

강의 순서

- Branch-and-Bound **개념**
- 0-1 Knapsack Problem
  - Depth-First Search (Backtracking)
  - Breadth-First Search
  - Best-First Search
- Traveling Salesman Problem
  - Dynamic Programming Approach
  - Branch-and-Bound Approach

0-1 Knapsack – Depth-First-Search 개념

- **분기한정 가지치기로 깊이우선검색 (= 되추적)**
  - **상태공간트리를 구축하여 되추적 기법으로 문제를 푼다.**
  - **루트 노드에서 왼쪽으로 가면 첫번째 아이템을 배낭에 넣는 경우이고, 오른쪽으로 가면 첫번째 아이템을 배낭에 넣지 않는 경우이다.**
  - **동일한 방법으로 트리의 수준 1에서 왼쪽으로 가면 두 번째 아이템을 배낭에 넣는 경우이고, 오른쪽으로 가면 그렇지 않는 경우이다.**
  - **이런 식으로 계속하여 상태공간트리를 구축하면, 루트 노드로부터 리프 노드까지의 모든 경로는 해답후보가 된다.**
  - **이 문제는 최적의 해를 찾는 문제(optimization problem)이므로 검색이 완전히 끝나기 전에는 해답을 알 수가 없다.**
  - **따라서 검색을 하는 과정 동안 항상 그 때까지 찾은 최적의 해를 기억해 두어야(메모리에 저장해 두어야) 한다.**

0-1 Knapsack – DFS 기반 Generic Algorithm

```
void checknode(node v) {
  node u;

  if(value(v) is better than best)
    best = value(v);
  if(promising(v))
    for(each child u of v)
      checknode(u);
}
```

- best: 지금까지 찾은 제일 좋은 해답치
- value(v): 노드 v에서의 해답치

0-1 Knapsack – DFS 기반 Algorithm Sketch (1/2)

- 다음 값들을 각 노드에 대해서 계산한다.
  - *profit*: 그 노드에 오기까지 넣었던 아이템의 값어치의 합.
  - *weight*: 그 노드에 오기까지 넣었던 아이템의 무게의 합.
  - *bound*: 노드가 수준 *i*에 있다고 하고, 수준 *k*에 있는 노드에서 총무게가 *W*를 넘는다고 하자. 그러면, 다음과 같이 *bound*를 구할 수 있다.

$$totweight = weight + \sum_{j=i+1}^{k-1} w_j$$

$$bound = \left(profit + \sum_{j=i+1}^{k-1} p_j \right) + (W - totweight) \times \frac{p_k}{w_k}$$

  - *maxprofit* : 지금까지 찾은 최선의 해답이 주는 값어치
- w_i와 p_i를 각각 *i*번째 아이템의 무게와 값어치라고 하면, p_i / w_i 의 값이 큰 것부터 내림차순으로 아이템을 정렬한다. (일종의 탐욕적인 방법이 되는 셈이지만, 알고리즘 자체는 탐욕적인 알고리즘은 아니다.)

0-1 Knapsack – DFS 기반 Algorithm Sketch (2/2)

- **초기값(루트 노드)**: *maxprofit* := $0; *profit* := $0; *weight* := 0
- **깊이우선순위로 각 노드를 방문하여 다음을 수행한다**:
  1. **그 노드의** *profit***와** *weight***를 계산한다**.
  2. **그 노드의** *bound***를 계산한다**.
  3. *weight* < *W* and *bound* > *maxprofit***이면**, **검색을 계속한다**;
     그렇지 않으면, **되추적한다**.
- **상기 과정을 모든 노드를 방문(실제로는 전지(가지치기)가 이뤄지므로, 모든 노드를 방문하지는 않음)할 때까지 수행한다**.
- **고찰**: **최선이라고 여겼던 노드를 선택했다고 해서 실제로 그 노드로부터 최적해가 항상 나온다는 보장은 없다**.

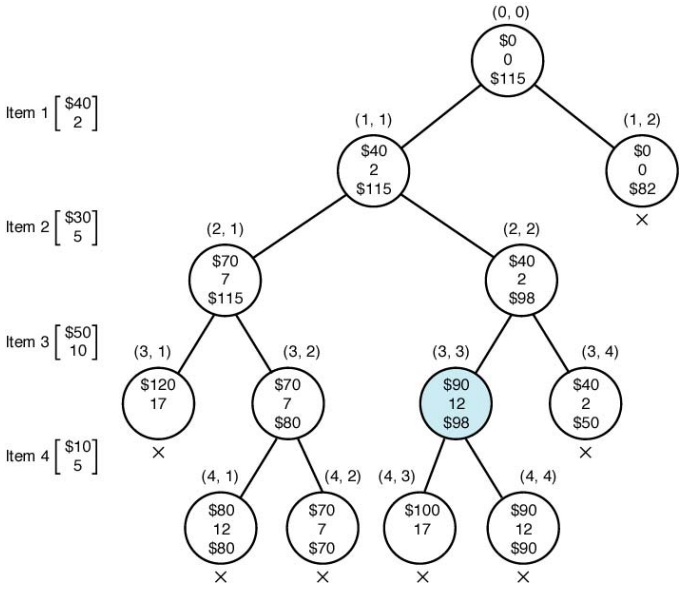
0-1 Knapsack – DFS 기반 Algorithm 적용 예제 (1/2)

- **보기**: *n* = 4, *W* = 16**이고**, **다음과 같은 아이템 내역을 가진다**.

i	p_i	w_i	$\frac{p_i}{w_i}$
1	$40	2	$20
2	$30	5	$6
3	$50	10	$5
4	$10	5	$2

- **이때**, **되추적을 사용하여 구축되는 전지가 이루어진 상태공간트리를 그려 보시오**.
  (**교재** p. 214**의 예제** 5.6 → 다음 페이지 그림 참조)

0-1 Knapsack – DFS 기반 Algorithm 적용 예제 (2/2)



0-1 Knapsack – DFS 기반 Algorithm 분석 (2/2)

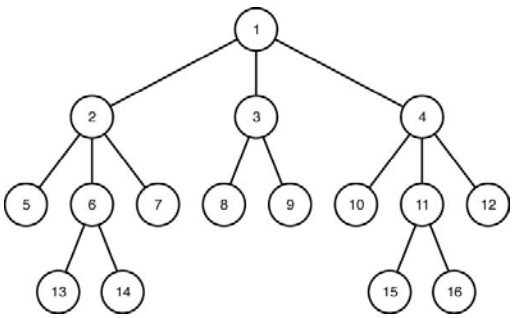
- 이 알고리즘이 점검하는 노드의 수는 $\Theta(2^n)$이다.
- 예제의 경우: 점검한 노드는 13개이다. 이 알고리즘이 DP 기반으로 설계한 알고리즘 보다 좋은가?
  - 확실하게 대답하기 불가능 하다.
  - Horowitz와 Sahni(1978)는 Monte Carlo 기법을 사용하여 되추적 알고리즘이 DP 기반 알고리즘 보다 일반적으로 더 빠르다는 것을 입증하였다.
  - Horowitz와 Sahni(1974)가 분할정복과 DP 기법을 적절히 조화하여 개발한 알고리즘은 $O(2^{n/2})$의 시간복잡도를 가지는데, 이 알고리즘은 되추적 알고리즘 보다 일반적으로 빠르다고 한다.

강의 순서

- Branch-and-Bound **개념**
- 0-1 Knapsack Problem
  - Depth-First Search (Backtracking)
  - Breadth-First Search
  - Best-First Search
- Traveling Salesman Problem
  - Dynamic Programming Approach
  - Branch-and-Bound Approach

Breadth-First-Search (1/2)

- **너비우선검색**(Breadth-first Search)**순서**:
  (1) **루트 노드를 먼저 검색한다**.
  (2) **다음에 수준** 1**에 있는 모든 노드를 검색한다**.
  (**왼쪽에서 오른쪽으로**)
  (3) **다음에 수준** 2**에 있는 모든 노드를 검색한다**
  (**왼쪽에서 오른쪽으로**)
  (4) …



Breadth-First-Search (2/2)

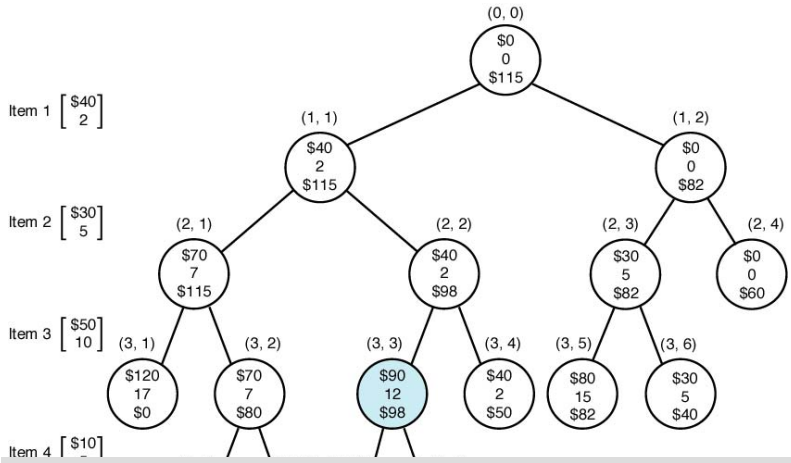
- A Generic Algorithm for Breadth-First-Search
  - **재귀**(recursive) **알고리즘을 작성하기는 상당히 복잡하다**.
  - **따라서 대기열(queue)을 사용한다**.

```
void breadth_first_search(tree T)
{
   queue_of_node Q;
   node u, v;
   initialize(Q);
   v = root of T;
   visit v;
   enqueue(Q,v);
   while(!empty(Q)) {
      dequeue(Q,v);
      for(each child u of v) {
         visit u;
         enqueue(Q,u);
      }
   }
}
```


BFS based Branch-and-Bound Algorithm

```
void breadth_first_branch_and_bound
     (state_space_tree T, number& best)
{
   queue_of_node Q;
   node u, v;
   initialize(Q);              // Q는 빈 대기열로 초기화
   v = root of T;              // 루트 노드를 방문
   enqueue(Q,v);
   best = value(v);
   while(!empty(Q)) {
      dequeue(Q,v);
      for(each child u of v) {        // 각 자식 노드를 방문
         if(value(u) is better than best)
            best = value(u);
         if(bound(u) is better than best)
            enqueue(Q,u);
      }
   }
}
```

0-1 Knapsack – BFS 기반 상태트리



분기한정 가지치기로 BFS를 하여 상태공간트리를 그려보면, 검색하는 노드의 개수는 17이다.
→ 되추적 알고리즘(DFS 기반 해결책)보다 좋지 않다!

강의 순서

- Branch-and-Bound **개념**
- 0-1 Knapsack Problem
  - Depth-First Search (Backtracking)
  - Breadth-First Search
  - Best-First Search
- Traveling Salesman Problem
  - Dynamic Programming Approach
  - Branch-and-Bound Approach

Best-First-Search – Concept

- 최적의 해답에 더 빨리 도달하기 위한 전략:
  1. 주어진 노드의 모든 자식노드를 검색한 후,
  2. 유망하면서 확장되지 않은(unexpanded) 노드를 살펴보고,
  3. 그 중에서 가장 좋은(최고의) 한계치(bound)를 가진 노드를 확장한다.
- (일반적으로) 최고우선검색(Best-First Search)을 사용하면, 너비우선검색에 비해서 검색 성능이 좋아짐

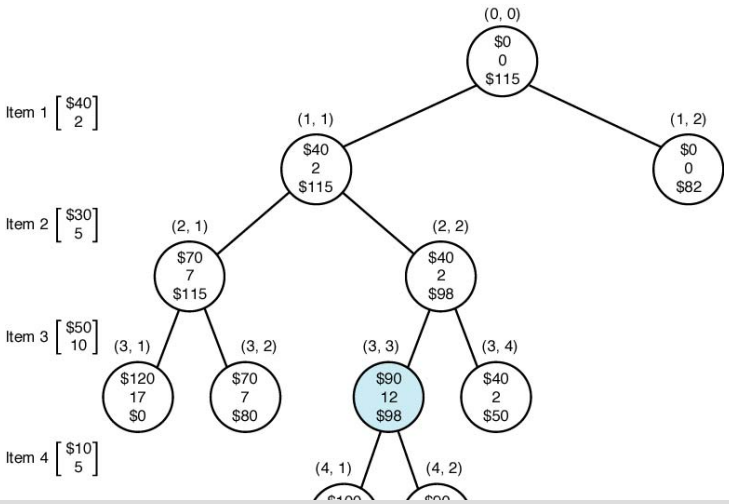
Best-First-Search – Strategy

- 최고의 한계를 가진 노드를 우선적으로 선택하기 위해서 우선순위 대기열(Priority Queue)을 사용한다.
- 우선순위 대기열은 힙(heap)을 사용하여 효과적으로 구현할 수 있다.

Best-FS based Branch-and-Bound Algorithm

```
void best_first_branch_and_bound
     (state_space_tree T, number best)
{
   priority_queue_of_node PQ;
   node u,v;
   initialize(PQ);        // PQ를 빈 대기열로 초기화
   v = root of T;
   best = value(v);
   insert(PQ,v);
   while(!empty(PQ)) {    // 최고 한계 값을 가진 노드를 제거
      remove(PQ,v);
      if(bound(v) is better than best) // 노드가 아직 유망한 지 점검
         for(each child u of v) {
            if(value(u) is better than best)
                best = value(u);
            if(bound(u) is better than best)
                insert(PQ,u);
         }
   }
}
```


0-1 Knapsack – Best-FS 기반 상태트리



분기한정 가지치기로 최고우선검색을 하여 상태공간트리를 그려보면, 검색하는 노드의 개수는 11로서, 앞서의 BFS보다 우수함을 알 수 있다.

강의 순서

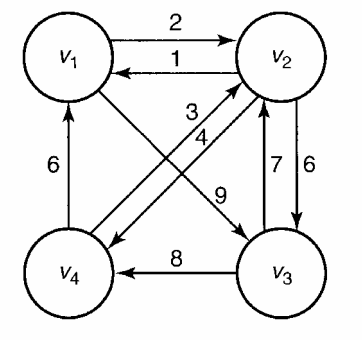
- Branch-and-Bound **개념**
- 0-1 Knapsack Problem
  - Depth-First Search (Backtracking)
  - Breadth-First Search
  - Best-First Search
- Traveling Salesman Problem
  - Dynamic Programming Approach
  - Branch-and-Bound Approach

Traveling Salesman Problem – 개요 (1/2)

- **외판원이 자신의 집이 위치하고 있는 도시에서 출발하여 다른 도시들을 각각 한번씩만 방문하고, 다시 자기 도시로 돌아오는 가장 짧은 일주여행경로**(tour)**를 결정하는 문제이다.**
- **일반적으로, 이 문제는 음이 아닌 가중치가 있는, 방향성 그래프를 대상으로 한다.**
- **그래프 상에서 일주여행경로는 한 정점을 출발하여 다른 모든 정점을 한번씩 만 거쳐서 다시 그 정점으로 돌아오는 경로이다.**
- **여러 개의 일주여행경로 중에서 길이가 최소가 되는 경로가 최적 일주여행경로**(optimal tour)**가 된다.**

Traveling Salesman Problem – 개요 (2/2)

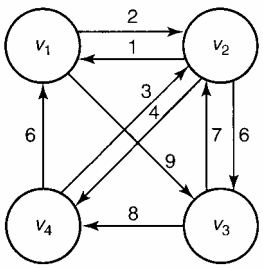
- 무작정 알고리즘: 가능한 모든 일주여행경로를 다 고려한 후, 그 중에서 가장 짧은 일주여행경로를 선택한다.
  → 가능한 일주여행경로의 총 개수는 $(n-1)!$이다.
- Why? (다음 예제를 보고 생각해 보세요.)



	1	2	3	4
1	0	2	9	∞
2	1	0	6	4
3	∞	7	0	8
4	6	3	∞	0

TSP – DP 기반 접근법 개념 (1/2)

- V는 모든 정점의 집합이고, A는 V의 부분집합이라고 하자.
- $D[v_i][A]$는 A에 속한 각 정점을 정확히 한번씩 만 거쳐서 v_i에서 v_1로 가는 최단경로의 길이라고 하자.
- 그러면 우리 예제에서 $D[v_2][\{v_3, v_4\}]$의 값은? (=20)



$D[v_2][A] = min(len[v_2, v_3, v_4, v_1], len[v_2, v_4, v_3, v_1])$

$min(20, \infty) = 20$

TSP – DP 기반 접근법 개념 (2/2)

- **최적 일주여행경로의 길이**:

$$D[v_1][V-\{v_1\}] = min_{2\le j\le n}(W[1][j] + D[v_j][V-\{v_1, v_j\}])$$

v_1에서 v_j로의 거리와 v_j를 뺐을 때 거리 합

- **일반적으로 표현하면 $i \neq 1$이고, v_i가 A에 속하지 않을 때, 다음과 같이 나타난다.**

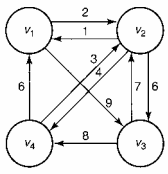
$$D[v_i][A] = min_{v_j \in A}(W[i][j] + D[v_j][A-\{v_j\}]) \text{ if } A \neq 0$$
$$D[v_i][\varnothing] = W[i][1]$$

- **예제 → 다음 슬라이드**

TSP – DP 기반 접근법 예제 (1/2)

- **최적 일주여행경로의 길이** = $D[v_1][\{v_2, v_3, v_4\}]$
- **공집합인 경우**

$$D[v_2][\varnothing] = 1$$
$$D[v_3][\varnothing] = \infty$$
$$D[v_4][\varnothing] = 6$$



- **하나의 구성요소만 포함하는 경우**

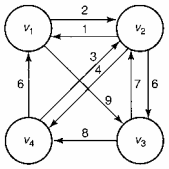
$$D[v_3][\{v_2\}] = \min_{v_j \in \{v_2\}}(W[3][j] + D[v_j][\{v_2\} - \{v_j\}])$$
$$= W[3][2] + D[v_j][\varnothing] = 7 + 1 = 8$$

$$D[v_4][\{v_2\}] = 3 + 1 = 4$$
$$D[v_2][\{v_3\}] = 6 + \infty = \infty$$
$$D[v_4][\{v_3\}] = \infty + \infty = \infty$$
$$D[v_2][\{v_4\}] = 4 + 6 = 10$$
$$D[v_3][\{v_4\}] = 8 + 6 = 14$$

TSP – DP 기반 접근법 예제 (2/2)

- **두 개의 구성요소를 포함하는 경우**

$$D[v_4][\{v_2,v_3\}] = \min_{v_j \in \{v_2,v_3\}}(W[4][j] + D[v_j][\{v_2,v_3\} - \{v_j\}])$$
$$= \min(W[4][2] + D[v_2][\{v_3\}], W[4][3] + D[v_3][\{v_2\}])$$
$$= \min(3+\infty, \infty+8) = \infty$$



$$D[v_3][\{v_2,v_4\}] = \min(7+10, 8+4) = 12$$
$$D[v_2][\{v_3,v_4\}] = \min(6+14, 4+\infty) = 20$$

- **최적 일주여행경로**

$$D[v_1][\{v_2,v_3,v_4\}] = \min_{v_j \in \{v_2,v_3,v_4\}}(W[1][j] + D[v_j][\{v_2,v_3,v_4\} - \{v_j\}])$$
$$= \min(\ W[1][2] + D[v_2][\{v_3,v_4\}],$$
$$W[1][3] + D[v_3][\{v_2,v_4\}],$$
$$W[1][4] + D[v_4][\{v_2,v_3\}])$$
$$= \min(2+20, 9+12, \infty+\infty) = 21$$

TSP – DP 기반 접근법 알고리즘 (1/2)

- **문제**: 가중치(음수가 아닌 정수)가 있는 방향성 그래프에서 최적 일주여행경로를 결정하시오.
- **입력**:
  - 가중치가 있는 방향성 그래프
  - 그래프에 있는 정점의 개수 *n*
  - 그래프는 행렬 *W*로 표시가 되는데, 여기서 *W*[*i*][*j*]는 v_i에서 v_j를 잇는 이음선 상에 있는 가중치를 나타낸다.
  - *V*는 그래프 상의 모든 정점의 집합을 나타낸다.
- **출력**:
  - 최적일주여행경로의 길이 값을 가지는 변수 *minlength*
  - 배열 *P* (이 배열로부터 최적일주여행경로를 구축할 수 있다).
    P[*i*][*A*]는 *A*에 속한 각 정점을 정확히 한번씩만 거쳐 v_i에서 v_1로 가는 최단경로 상에서, v_i 다음의 도달하는 첫 번째 노드의 인덱스이다.

TSP – DP 기반 접근법 알고리즘 (2/2)

- 알고리즘

```
void travel(int n, const number W[][], index P[][],
            number& minlength)
{
   index i, j, k;
   number D[1..n][subset of V-{v1}];

   for(i=2; i<=n; i++)
      D[i][emptyset] := W[i][1];

   for(k=1; k<=n-2; k++)
      for(all subsets A ⊆ V-{v1} containing k vertices)
         for(i such that i≠1 abd vi ∉ A){
            D[i][A] = minimum_{vj∈A}(W[i][j] + D[vj][A-{vj}]);
            P[i][A] = value of j that gave the minimum;
         }
   D[1][V-{v1}] = minimum_{2≤j≤n}(W[1][j] + D[vj][A-{v1}]);
   P[1][V-{v1}] = value of j that gave the minimum;
   minilength = D[1][V-{v1}];
}
```


TSP – DP 기반 접근법 분석 (1/3)

- **정리**: $n \geq 1$를 만족하는 모든 n에 대해서 다음이 성립한다.

$$\sum_{k=1}^{n} k \binom{n}{k} = n\,2^{n-1}$$

- 증명은 생략

TSP – DP 기반 접근법 분석 (2/3)

- **단위연산**: 중간에 위치한 루프가 수행시간을 지배한다.
  - 왜냐하면 이 루프는 여러 겹으로 쌓여 있기 때문이다.
  - 따라서, 단위연산은 v_j의 각 값에 대해서 수행되는 명령문이다. (덧셈하는 명령문 포함)
- **입력 크기**: 그래프에서 정점의 개수 n
- **시간 복잡도**: 알고리즘에서 두 번째 for-루프가 시간복잡도를 좌우한다.

```
for(k=1; k<=n-2; k++)
  (1)  for(V-{v1})의 부분집합 중에서 k개의 정점을 가진 모든 부분집합 A)
  (2)    for(i=1이 아니고 vi가 A에 속하지 않는 모든 i)
  (3)      D[i][A] = minimum_{vj∈A}(W[i][j] + D[vj][A-{vj}]);
           P[i][A] = value of j that gave the minimum;
```


TSP – DP 기반 접근법 분석 (3/3)

- (1)번 루프는 $\binom{n-1}{k}$ 번 반복하고 (n-1개의 정점에서 k개를 뽑는 경우의 수), (2)번은 n - k - 1번 반복하고 (v_1을 제외하고 A에 속하지 않는 정점 개수), (3)번 루프는 A의 크기가 k이므로 k번 반복한다(A에 속한 정점의 개수).
- 따라서 시간복잡도는 다음과 같다.

$$T(n) = \sum_{k=1}^{n-2} (n-k-1)k\binom{n-1}{k} \in \Theta(n^2 2^n)$$

- 공간복잡도: 배열 $D[v_i,A]$와 $P[v_i,A]$가 얼마나 커야 하는지를 결정하면 된다. V - $\{v_1\}$는 n - 1개의 정점을 가지고 있기 때문에, 이 배열은 2^{n-1}개의 부분집합 A를 가지고 있다. (n개의 아이템이 포함되어 있는 어떤 집합의 부분집합의 개수는 2^n이다.)
- 따라서 공간복잡도는 $M(n) = 2 \times n \times 2^{n-1} = n2^n \in \Theta(n2^n)$이 된다.

TSP – 무작정 vs. DP

- n = 20**일 때**,
  - **무작정 알고리즘**: **각 일주여행경로의 길이를 계산하는데 걸리는 시간은** 1μsec**이라고 할 때**, (20 - 1)! = 19!μsec = 3857**년이 걸린다**.
  - DP **기반 알고리즘**: **기본동작을 수행하는데 걸리는 시간을** 1μsec**이라고 할 때**, $T(20) = (20 - 1)(20 - 2)2^{20-3}\mu sec$ = 45**초가 걸리나**, $M(20) = 20 \times 2^{20}$ = 20,971,520**의 배열의 슬롯이 필요하다**.
- n = 40**이라면**? DP **또한** 6**년 이상 걸린다**.
- Dynamic Programming **방법 또한 실용적인 해결책인 아니다**.

TSP – DP 기반 알고리즘 – 최적경로 출력

$$P[1,\{v_2,v_3,v_4\}]=3 \Rightarrow P[3,\{v_2,v_4\}]=4 \Rightarrow P[4,\{v_2\}]=2$$

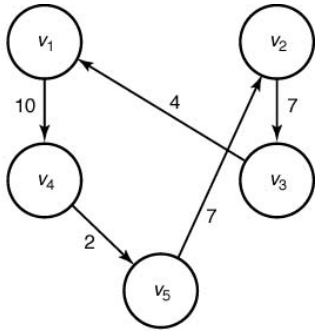
- **따라서 최적 일주여행경로는** [v_1, v_3, v_4, v_2, v_1]**이다**.

TSP – BnB 기반 접근법 – Running Example

- n = 40일 때, 동적계획법 알고리즘은 6년 이상이 걸린다. 그러므로 분기한정법을 시도해 본다.
- **보기**: 다음 인접행렬로 표현된 그래프를 살펴보시오.

$$\begin{bmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{bmatrix}$$

완전 연결그래프의 인접행렬 표현



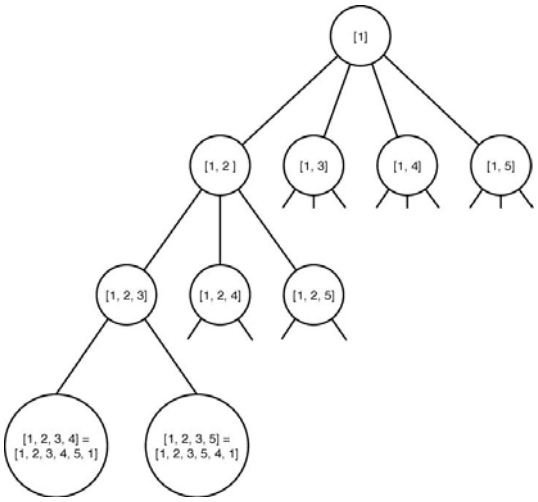
최적 일주여행경로

TSP – BnB 기반 상태공간트리 구축 (1/3)

- 각 노드는 출발노드로부터의 일주여행경로를 나타내게 되는데, 몇 개 만 예를 들어 보면, 다음과 같다.
  - 루트노드의 여행경로는 [1]이 되고, 루트노드에서 뻗어 나가는 수준 1에 있는 여행경로는 각각 [1,2], [1,3], …, [1,5]가 된다.
  - 노드 [1,2]에서 뻗어 나가는 수준 2에 있는 노드들의 여행경로는 각각 [1,2,3], …, [1,2,5]가 된다.
  - 이런 식으로 뻗어 나가서 단말노드에 도달하게 되면 완전한 일주여행경로를 가지게 된다.

TSP – BnB 기반 상태공간트리 구축 (2/3)

- 구축된 상태공간 트리의 일부 예



TSP – BnB 기반 상태공간트리 구축 (3/3)

- 최적일주여행경로를 구하는 방법:
  단말노드에 있는 일주여행경로를 모두 검사하여 그 중에서 가장 길이가 짧은 일주여행경로를 찾으면 된다.
- 참고: 위 예에서 각 마디에 저장되어 있는 마디가 4개가 되면 더 이상 뻗어 나갈 필요가 없다. 왜냐하면, 남은 경로는 더 이상 뻗어 나가지 않고도 알 수 있기 때문이다.

TSP – Best First Search 기반 접근법 개요 (1/5)

- 분기한정 가지치기로 최고우선 검색을 사용하기 위해서 각 마디의 **한계치**(bound)를 구할 수 있어야 한다.
- 이 문제에서는 주어진 마디에서 뻗어 나가서 얻을 수 있는 여행경로의 길이의 **하한**(최소치, lower bound)을 구하여 한계치로 한다.
  - 각 마디를 검색할 때 최소여행경로의 길이 보다 한계치가 작은 경우 그 마디는 유망하다고 한다.
  - 반대로, 최소 여행경로의 길이가 한계치보다 큰 경우는 가지치기를 수행하여 검색 공간을 줄인다.
- 한계치(lower bound)의 변화
  - 최소여행경로의 초기값은 ∞로 놓는다.
  - 완전한 여행경로를 처음 얻을 때 까지는 한계치가 무조건 최소여행경로의 길이 보다 작게 되므로 모든 마디는 유망하다.
  - 완전한 여행경로를 얻은 후에는 한계치가 갈수록 증가하여 가지치기의 효과가 커진다.

TSP – BFS 기반 접근법 개요 (2/5)

- 각 노드의 한계치는 어떻게 구하나?
  - [1, …, k]의 여행경로를 가진 마디의 한계치는 다음과 같이 구한다.
  - Let A = V - ([1,…,k] 경로에 속한 모든 노드의 집합)
  - bound = [1, …, k] 경로 상의 총 거리
    + v_k에서 A에 속한 정점으로 가는 이음선의 길이들 중에서 최소치
    + $\Sigma_{i \in A}$(v_i에서 $A \cup \{v_1\} - \{v_i\}$에 속한 정점으로 가는 이음선의 길이들 중 최소치)

  → 다음 페이지의 예제를 참조할 것

TSP – BFS 기반 접근법 개요 (3/5)

루트노드의 하한 구하기

- 근거: 어떤 일주여행경로라도, 각 정점을 최소한 한번은 떠나야 하므로, 각 정점을 떠나는 이음선의 최소값의 합이 하한이 된다.
  - v_1 → min(14, 4, 10, 20) = 4
  - v_2 → min(14, 7, 8, 7) = 7
  - v_3 → min(4, 5, 7, 16) = 4
  - v_4 → min(11, 7, 9, 2) = 2
  - v_5 → min(18, 7, 17, 4) = 4

$$\begin{bmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{bmatrix}$$

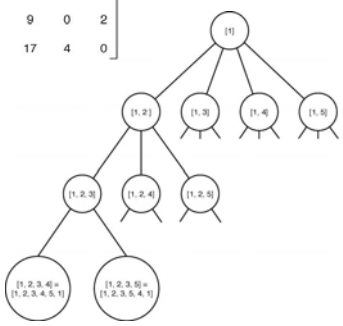
- 따라서, 일주여행경로 길이의 하한은 21(= 4+7+4+2+4)이 된다.
- 주의할 점은 "이 길이의 일주여행경로가 있다는 말이 아니라, 이보다 더 짧은 일주여행경로가 있을 수 없다"는 것이다.
  그래서 하한(lower bound)이라는 말을 사용한다.

TSP – BFS 기반 접근법 개요 (4/5)

노드 [1, 2]를 선택한 경우의 하한 구하기

- 근거: 이미 v_2를 선택하였음을 의미하므로, v_1 → v_2의 비용은 이음선의 가중치인 14가 된다. 나머지는 앞서와 동일한 방법으로 구한다.
  - v_1 → = 14
  - v_2 → min(7, 8, 7) = 7
  - v_3 → min(4, 7, 16) = 4
  - v_4 → min(11, 9, 2) = 2
  - v_5 → min(18, 17, 4) = 4

$$\begin{bmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{bmatrix}$$

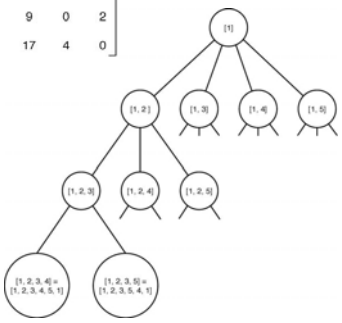


- 따라서, [1, 2]를 포함하는 노드에서 확장하여 구한 일주여행경로 길이의 하한은 31(= 14+7+4+2+4)가 된다.

TSP – BFS 기반 접근법 개요 (5/5)

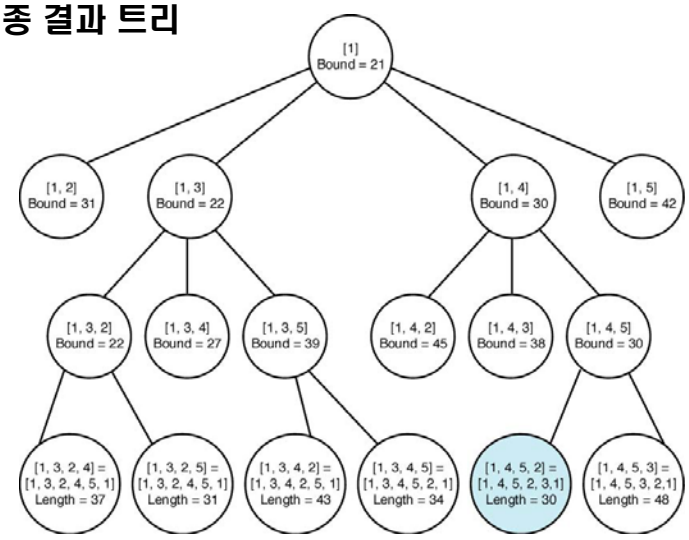
- **노드 [1, 2, 3]를 선택한 경우의 하한 구하기**
  - **근거: 이미 v_2와 v_3를 선택하였음을 의미하므로, v_1 → v_2 → v_3의 비용은 21(=14+7)이 된다. 나머지는 앞서와 동일한 방법으로 구한다.**
    - v_1 → = 14
    - v_2 → = 7
    - v_3 → min(7, 16) = 4
    - v_4 → min(11, 2) = 2
    - v_5 → min(18, 4) = 4
  - **따라서, [1, 2, 3]을 포함하는 노드에서 확장하여 구한 일주여행경로 길이의 하한은 34(= 14+7+7+2+4)이 된다.**

$$\begin{bmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{bmatrix}$$



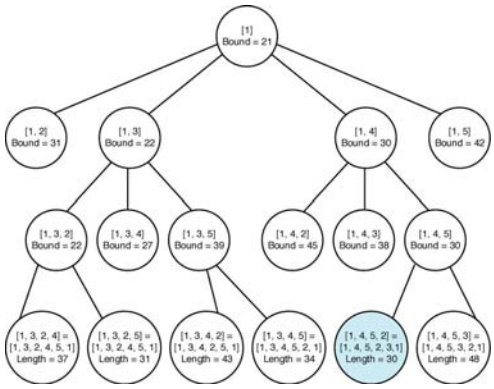
TSP – BFS 기반 상태공간트리 구축 (1/5)

- **최종 결과 트리**



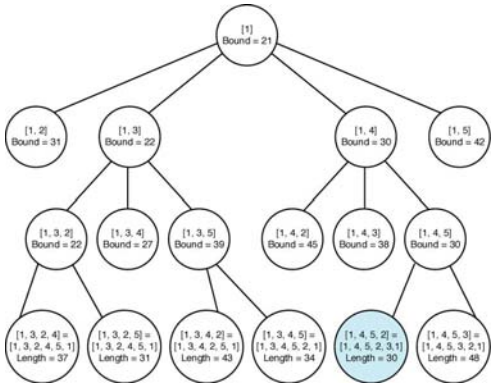
TSP – BFS 기반 상태공간트리 구축 (2/5)

- **루트노드 구성** (LB = 21, minLen = ∞)
- **노드** [1, 2] (LB = 31)
- **노드** [1, 3] (LB = 22)
- **노드** [1, 4] (LB = 30)
- **노드** [1, 5] (LB = 42)
- BFS**에 따라 한계 값이 가장 작은** [1, 3]**을 방문한다**.



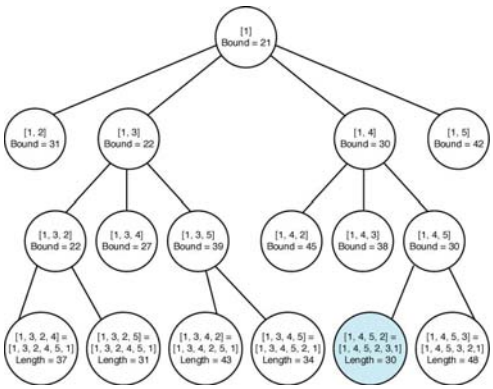
TSP – BFS 기반 상태공간트리 구축 (3/5)

- **노드** [1, 3, 2] (LB = 22)
- **노드** [1, 3, 4] (LB = 27)
- **노드** [1, 3, 5] (LB = 39)
- BFS**에 따라 한계 값이 가장 작은** [1, 3, 2]**를 방문한다**.



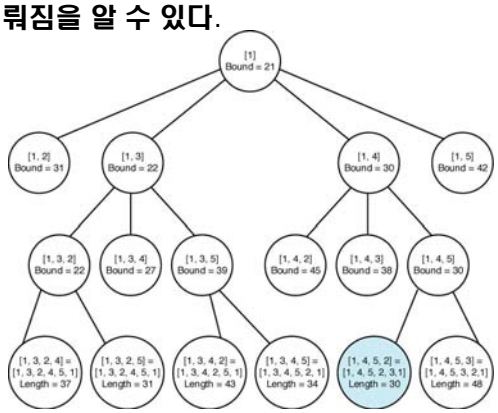
TSP – BFS 기반 상태공간트리 구축 (4/5)

- 노드 [1, 3, 2, 4]
  - 단말노드 이므로 일주여행경로의 길이를 계산한다.
  - 이 길이가 37이고, 37 < ∞이므로, minLen = 37이 된다.
  - [1, 5]와 [1, 3, 5]는 한계값(각각 42, 39)이 minLen보다 크므로 가지치기 할 수 있다.
- 노드 [1, 3, 2, 5]
  - 방문 결과 minLen = 31이 된다.
  - [1, 2]를 가지치기 할 수 있다.
- 다음으로, [1, 3, 4]를 선택한다.
- 상기 과정을 계속 반복하면, 왼편 그림의 Length = 30을 최소 길이로 구할 수 있다.



TSP – BFS 기반 상태공간트리 구축 (5/5)

- 상태공간 트리의 전체 노드 개수는 41개이다.
  왜냐면, 1 + 4 + 4 x 3 + 4 x 3 x 2 = 41이기 때문이다.
- 반면에, 왼편 그림을 보면 노드 개수가 17개이다.
- 결국, 효과적인 가지치기가 이뤄짐을 알 수 있다.



TSP – BFS 기반 알고리즘

- 자세한 알고리즘은 생략 (관심 있는 학생은 교재 p. 247 참조)

- 아직도 알고리즘의 시간복잡도는 지수적이거나 그보다 못하다!
- 다시 말해서 n = 40이 되면 문제를 풀 수 없는 것과 다름없다고 할 수 있다.

- 다른 방법이 있을까?
- 근사(approximation) 알고리즘: 최적의 해답을 준다는 보장은 없지만, 무리 없이 최적에 가까운 해답을 주는 알고리즘이다.
  → 교재 제6.3절의 확률적 추론(진단) 방법