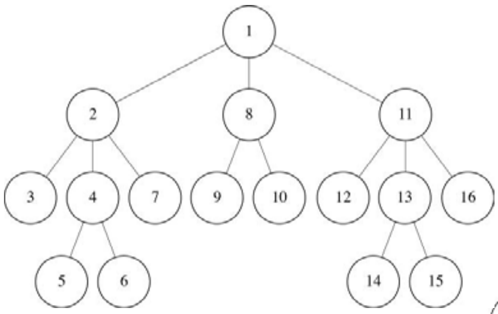
Chapter 6
Branch and Bound

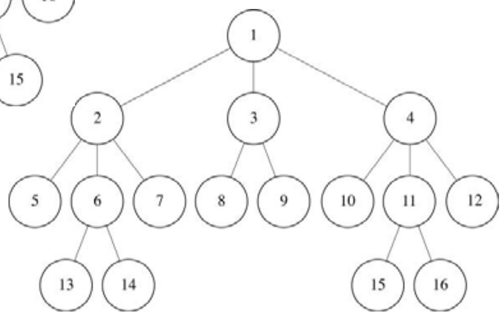
- 최적화 문제를 위한 것으로서 Backtracking 알고리즘을 개선
- State Space Tree 상에서 가능성이 큰 것을 먼저 검사
- Worst case time complexity는 여전히 높지만 average case time complexity는 효율적이다.

Branch and Bound

- Branch and bound
  - similar to backtracking
  - no limit to any particular traversing way
  - need data structure for state space tree
  - only for optimization problems
- Traversing of state space tree
  - (Backtracking) Depth-first search with pruning (recursive call)
  - (Branch and bound) Best-first search with pruning (while loop with priority queue)
  - Breadth-first search with pruning (use queue)
- Best first search
  - visit the child of the best bound ➔ find an optimal solution node faster

Depth-first Search and Breadth-first Search



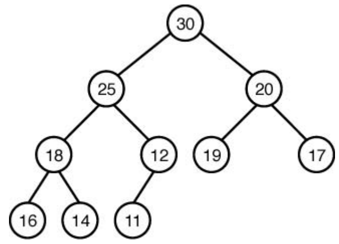


Breadth-First Search with Branch-and-Bound Pruning

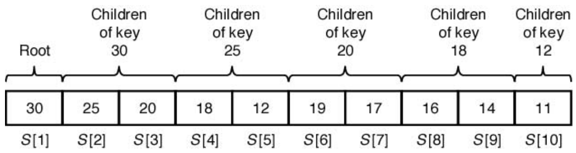
- ✓ (skip)

Priority Queue (7.6.2 Heap Data Structure)

- ✓Complete binary tree (Fig. 7.5)
- ✓Value of each node is greater than equal to its children's values (max heap)
- ✓Tree on a linear array (7.8)
  - ✓Left child index = 2*parent
  - ✓Right child index = 2*parent + 1

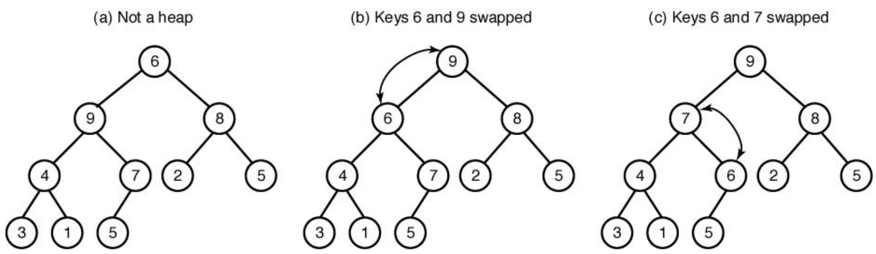



```
struct heap {
  keytype S[1..n];
  int heapsize;
}
```



Heap Operation

- ✓Insert : append as the last and shift up it
- ✓Remove: remove the root, move the last on the root and shift down it
- ✓Shift up
- ✓Shift down example: Fig. 7.6



Insert and Remove

```
typedef struct heap {
    keytype S[n+1];          // S[0]: not used
    int heapSize;            // number of entries
} heap;

void Remove(heap &H, keytype &key)
{
    key = H.S[1];
    H.heapSize--;
    if (H.heapSize > 0) {
        H.S[1] = H.S[H.heapSize+1];
        shiftDown(&H, 1);
    }
}
void Insert(heap &H, keytype &key)
{
    H.heapSize++;
    H.S[H.heapSize] = key;
    shiftup(&H, H.heapSize);
}
```

```
void shiftDown(heap &H, int i)
{
    ...        // maxheap
}
void shiftUp(heap &H, int i)
{
    ...        // maxheap
}
```


Best-First Search with Branch-and-Bound Pruning

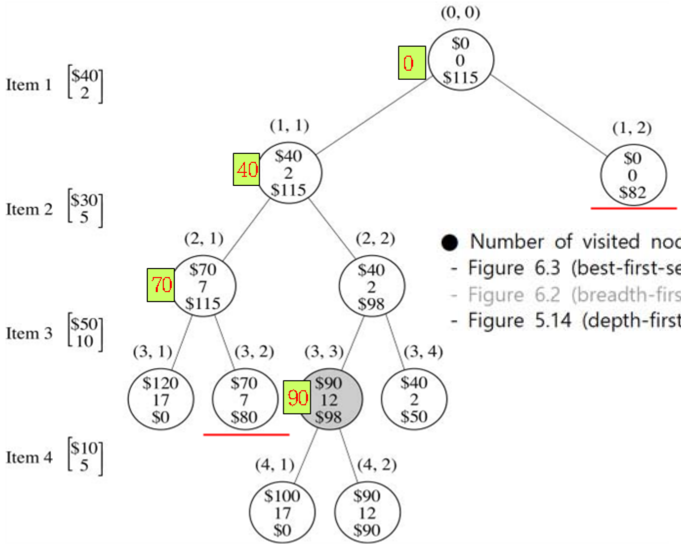
```
void best_first_branch_and_bound(state_space_tree T,
  number &best)
{  priority_queue_of_node PQ; node u, v;

   initialize(PQ);
   v = root of T;
   best = value(v);
   insert(PQ, v);
   while ( ! empty(PQ) ) {
      remove(PQ, v);
      if (bound(v) > best)
          for (each child u of v) {
              if (value(u) > best)
                  best = value(u);
              if (bound(u) > best)
                  insert(QP, u);
          }
   }
}
```

● 0-1 Knapsack Problem (Ex. 6.1 and Fig. 6.3)

(p,w) : ($40, 2), ($30, 5), ($50, 10), ($10, 5)

($profit, weight, $bound)



● Number of visited nodes

- Figure 6.3 (best-first-search) : 11
- Figure 6.2 (breadth-first-search) : 17
- Figure 5.14 (depth-first-search) : 13

● Algorithm 6.2 ***knapsack3***

(0-1 Knapsack with Best-First-Search)

- Fields of each node on the queue

```
struct node {
    int level, profit, weight;
    float bound;      // the upper bound of profit
    ...               // fields to maintain queue
   // boolean include[1..n]   // to print items
}
```

```
bound(node u)   // same as knapsack2 p.256
{
  if the weight of u exceeds W
     return 0
  else
     return the weight of fractional knapsack
}
```

Algorithm 6.2 *knapsack3*

```
void knapsack3 (int n,
                const int p[], const int w[],
                int W,
                int& maxprofit)
{
  priority_queue_of_node PQ;
  node u, v;

  initialize(PQ);                                      // Initialize PQ
  v.level = 0; v.profit = 0; v.weight = 0;             // empty.
  maxprofit = 0;                                       // Initialize v
  v.bound = bound(v);                                  // root.
  insert(PQ, v);
```



```
    while (!empty(PQ)){
      remove(PQ, v);                                   // Remo
      if (v.bound > maxprofit){                        // best
        u.level = v.level + 1;                         // Chec
        u.weight = v.weight + w[u.level];              // pron
        u.profit = v.profit + p[u.level];              // Set
                                                       // that
        if (u.weight <= W && u.profit > maxprofit)     // next
          maxprofit = u.profit;
        u.bound = bound(u);
        if (u.bound > maxprofit)
          insert(PQ, u);
        u.weight = v.weight;                           // Set u
        u.profit = v.profit;                           // that
        u.bound = bound(u);                            // the n
        if (u.bound > maxprofit)
          insert(PQ, u);
      }
    }
}
```

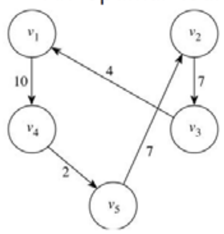
Traveling Salesperson Problem

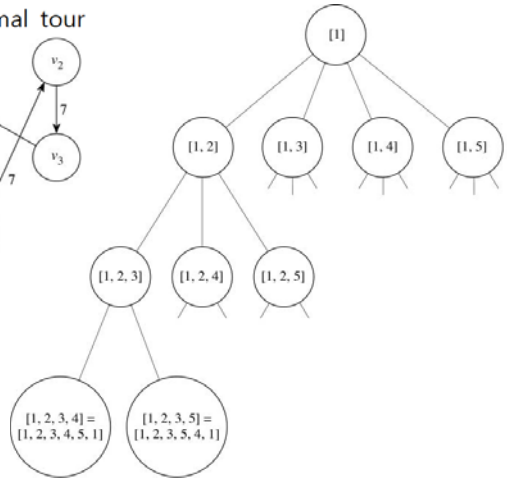
✓State-Space Tree : Fig. 6.4 and Fig. 6.5

Given graph

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

An optimal tour





Algorithm 6.3 *travel2*

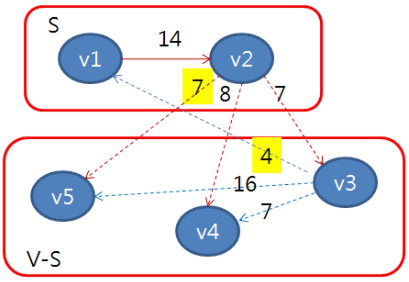
```
void travel2 (int n,
              const number W[] [],
              ordered-set& opttour,
              number& minlength)
{
  priority_queue_of_node PQ;
  node u, v;
```

```
initialize(PQ);                                          // Initial
v.level = 0;
v.path = [1];                                            // Make fi
v.bound = bound(v);                                      // startin
minlength = ∞;
insert(PQ, v);
while (! empty(PQ)){
  remove(PQ, v);                                         // Remove n
  if (v.bound < minlength){
    u.level = v.level + 1;                               // Set u to
    for (all i such that 2 ≤ i ≤ n && i is not i
      u.path = v.path;
      put i at the end of u.path;
      if (u.level == n - 2){                             // Check if
        put index of only vertex                         // complete
        not in u.path at the end of u.path;
        put 1 at the end of u.path;                      // Make firs
        if (length(u) < minlength){                      // Function
          minlength = length(u);                         // length o
          opttour = u.path;
        }
      }
      else{
        u.bound = bound(u);
        if (u.bound < minlength)
          insert(PQ, u);
      }
    }
  }
}
```


Lower Bound of a Tour Length

(1) current length + sum of the minimum outgoing weights of unvisited vertices and the last

```
bound1(u) {
    len = length(u.path);
    S = set of vertices of u.path;
    for (each vertex v of the last or not in S)
        len += min(W[v][j] for vj of v1 or not in S;
    return len;
}
```



Lower Bound of a Tour Length

(2) current length + sum of the minimum inward weights of unvisited vertices and v_1

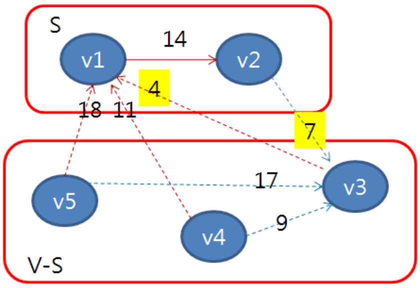
```
bound2(u) {
    len = length(u.path);
    S = set of vertices of u.path;
    for (each vertex v of v1 or not in S)
      len += min(W[j][v] for vj of last or not in S;
    return len;
}
```

(3) average of the two

```
bound3(u) {
    return (bound1(u) + bound2(u)) / 2;
}
```

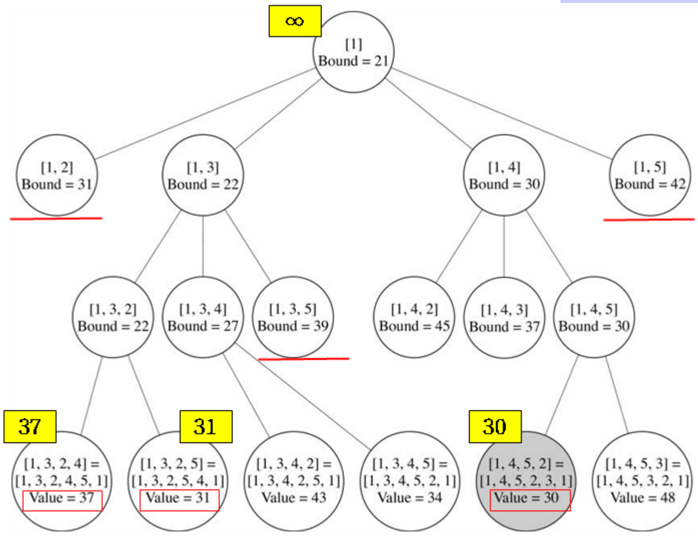
(4) max of the two

```
bound4(u) {
    return max(bound1(u), bound2(u));
}
```



Example: Fig. 6.4 and Fig 6.6 (bound1 적용)

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



Number of Visited Nodes

- ✓Total nodes in the state space tree : 41
- ✓Bound1: Number of visited nodes : 17 nodes
- ✓Bound3: 15 nodes → bound 계산 시간과 검사 node 수의 tradeoff

- ✓To bound the worst-case computation time ?
- ➔ Approximation algorithm (Section 9.5)

Summary

- ✓State space tree and Best-first search
- ✓Branch and bound algorithm
- ✓0-1 Knapsack problem
- ✓Traveling salesperson problem
- ✓Comparison of Backtracking and Branch-and-bound