CSW3010 알고리즘 입문
9장 해 탐색 알고리즘

담당교수: 장직현
서강대학교 컴퓨터공학과

9.1 Backtracking

- **The General Method**

해를 탐색 (state space tree를 탐색) 가면서 도중에 막히면
되돌아가서 다시 해를 찾아가는 기법이다.

In the search for fundamental principles of algorithm design,
backtracking represents one of the most general techniques.

Backtracking is based on the Depth First Tree Searching
same as the preorder tree traversal.

Many problems which deal with searching for a set of solutions
or which ask for an optimal solution satisfying some constraints
can be solved using the backtracking formulation.

- In many applications of backtrack method,
  the desired is expressible as an n-tuple $(x_1, x_2, \ldots, x_n)$
  where x_i is chosen from some finite set S_i.

Often the problem to be solved calls for finding one vector
that maximize (or minimize or satisfies)
a criterion function $P(x_1, x_2, \ldots, x_n)$.

Sometimes it seeks all vectors that satisfy P.

The backtracking (branch-and-bound) algorithm has its virtue
the ability to yield the same answer much more efficiently
than the brute force method.

- Its basic idea is

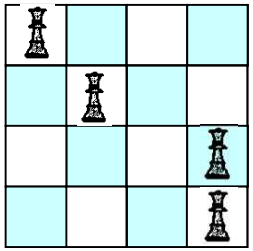
  to build up the solution vector one component at a time

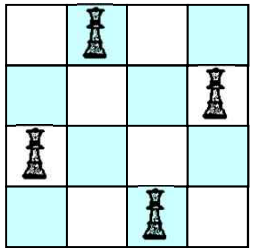
  and to use modified criterion functions $P(x_1, x_2, \ldots, x_i)$

  (sometime called bounding functions) to test

  whether the vectors being formed has a chance to success.

- *n*-Queens Problem
  - We have $n \times n$ chess board and n queens.
  - The goal is to position n queens on the chessboard so that no two queens are in the same row, column or diagonal.
  - Example:

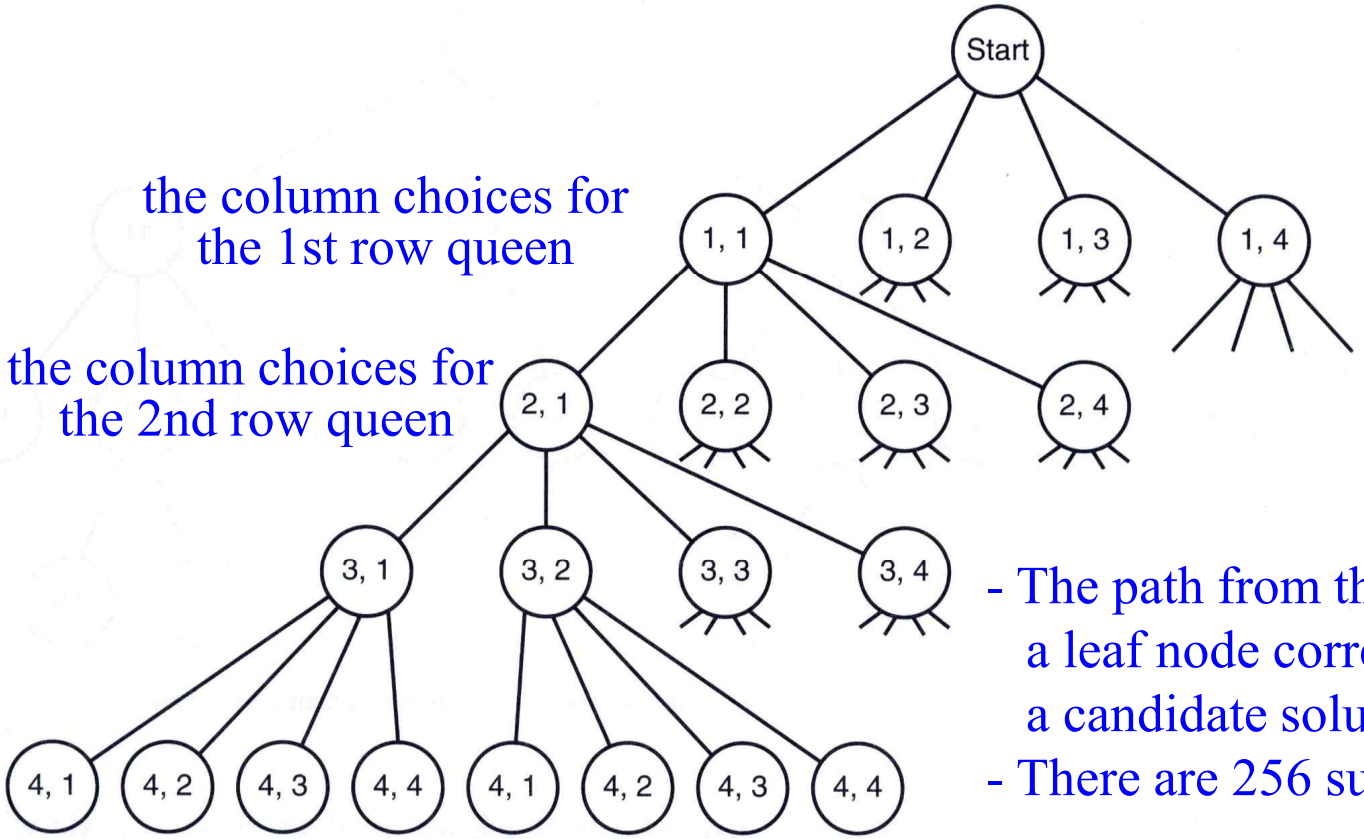




  - Observation
    - No two queens can be in the same row.
    - Assign each queen a different row and check which column combinations yields solutions.
    - We have $4 \times 4 \times 4 \times 4 = 256$ candidate solutions for $n = 4$.

The State Space Tree

- A tree representing all possible candidate solutions.
- Example: The 4-queens problem



the column choices for the 1st row queen

the column choices for the 2nd row queen

- The path from the root to a leaf node corresponds to a candidate solution.
- There are 256 such path.

State Space Tree

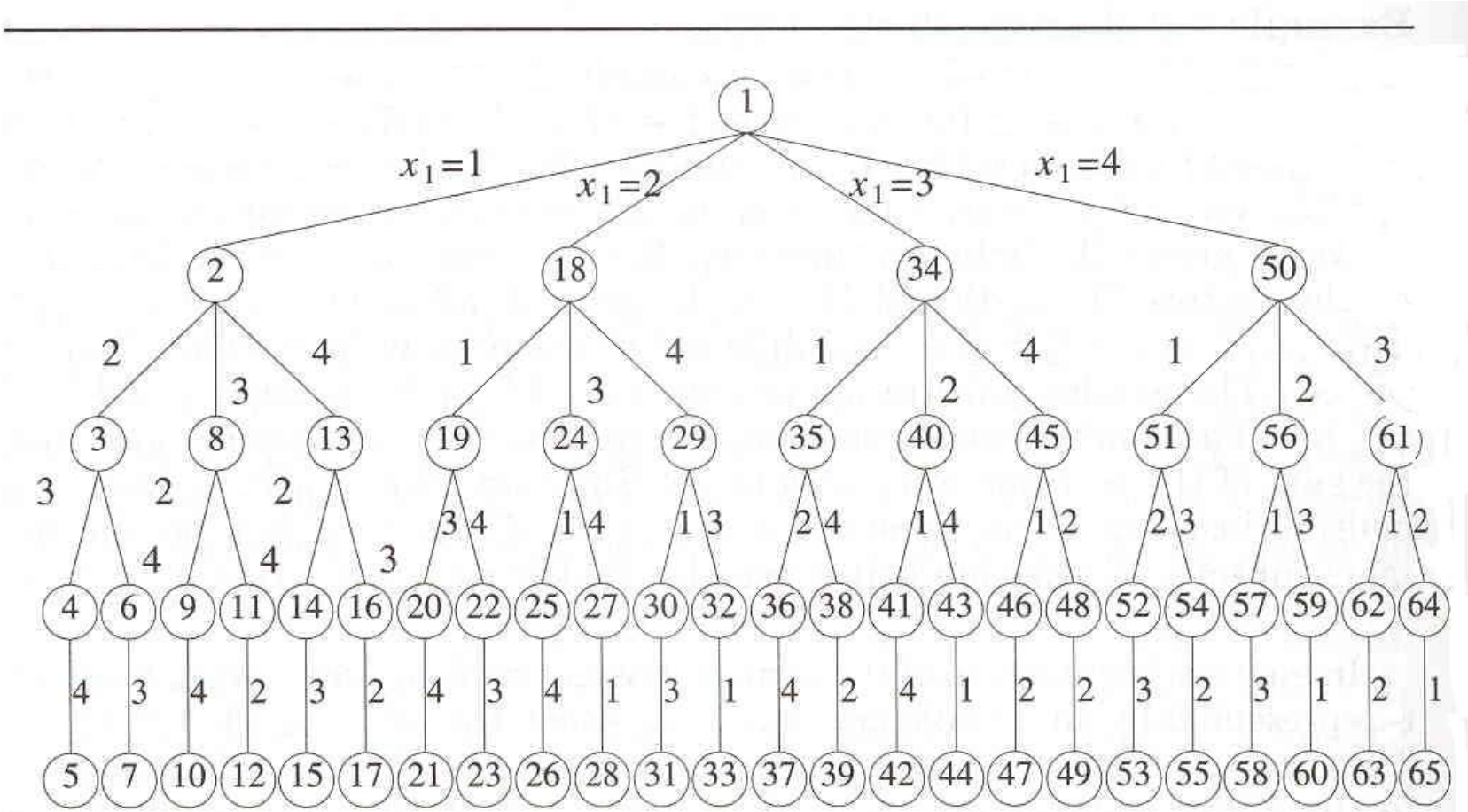


Figure 7.2 Tree organization of the 4-queens solution space. Nodes are numbered as in depth first search.

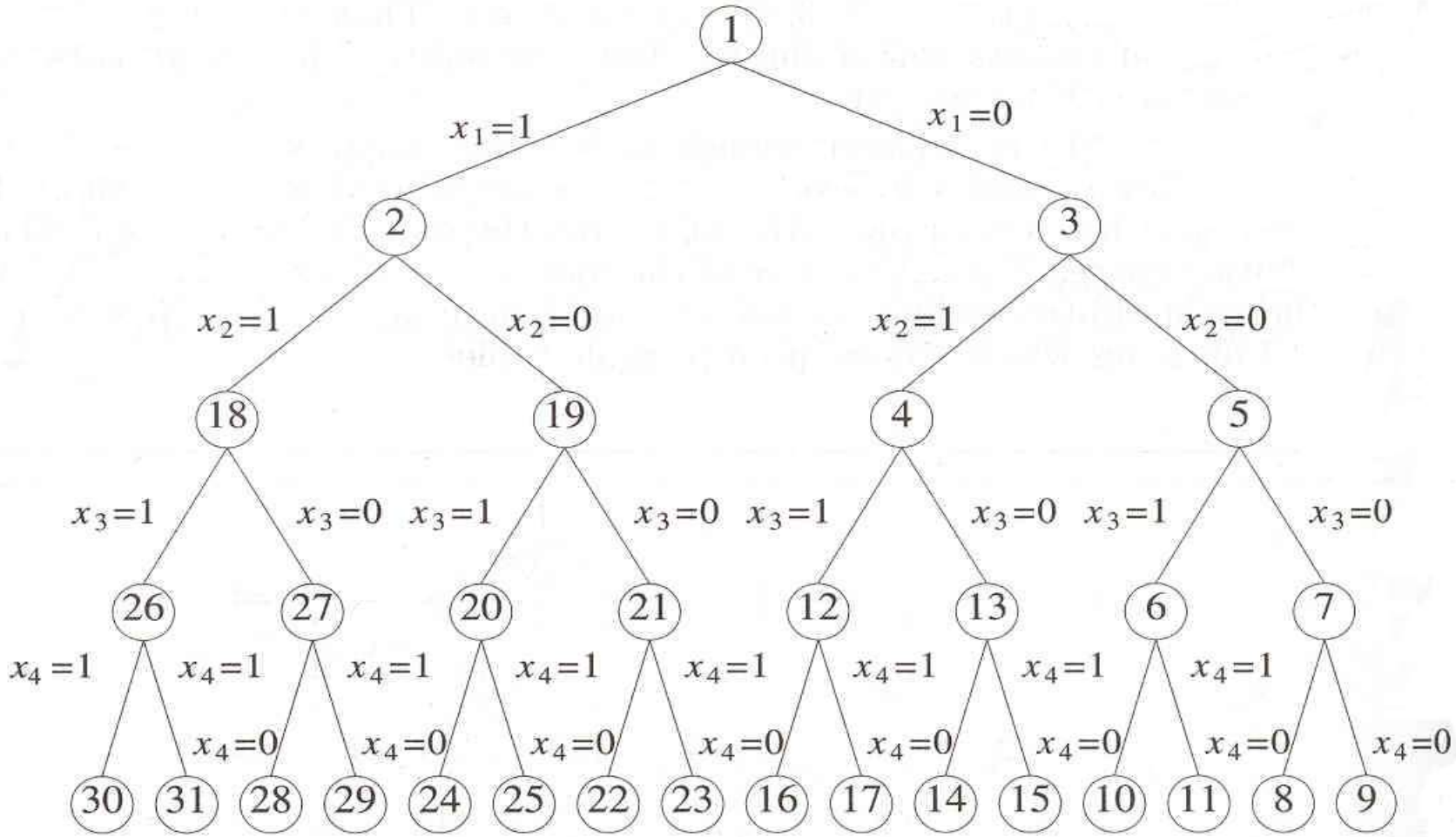


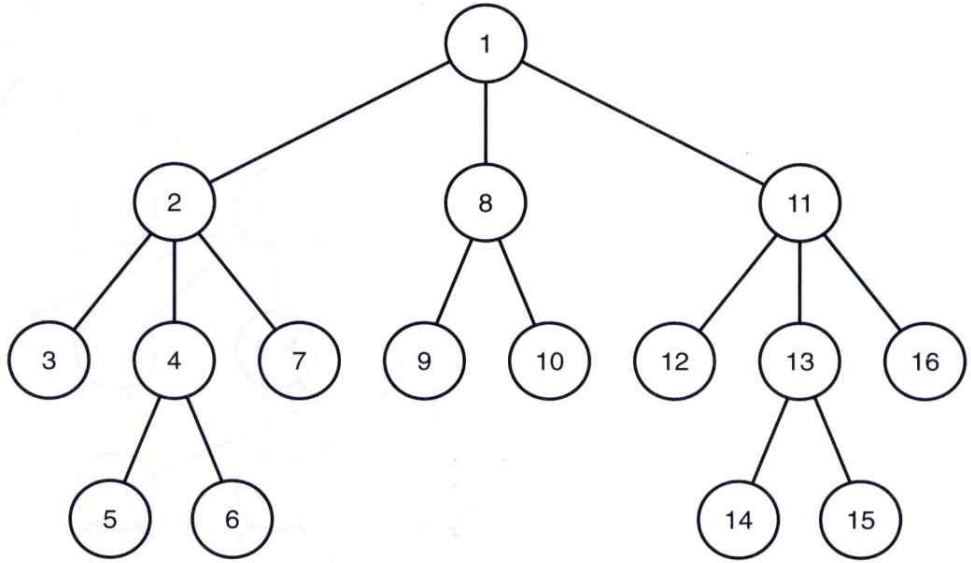
Figure 7.4 Another possible organization for the sum of subsets problems. Nodes are numbered as in *D*-search.

Depth First Tree Searching

- The same as the preorder tree traversal.
- The Procedure

```
void depth_first_tree_search (node v){
  node u;
  visit v;
  for (each child u of v)
      depth_first_tree_search(u);
}
```

- Example



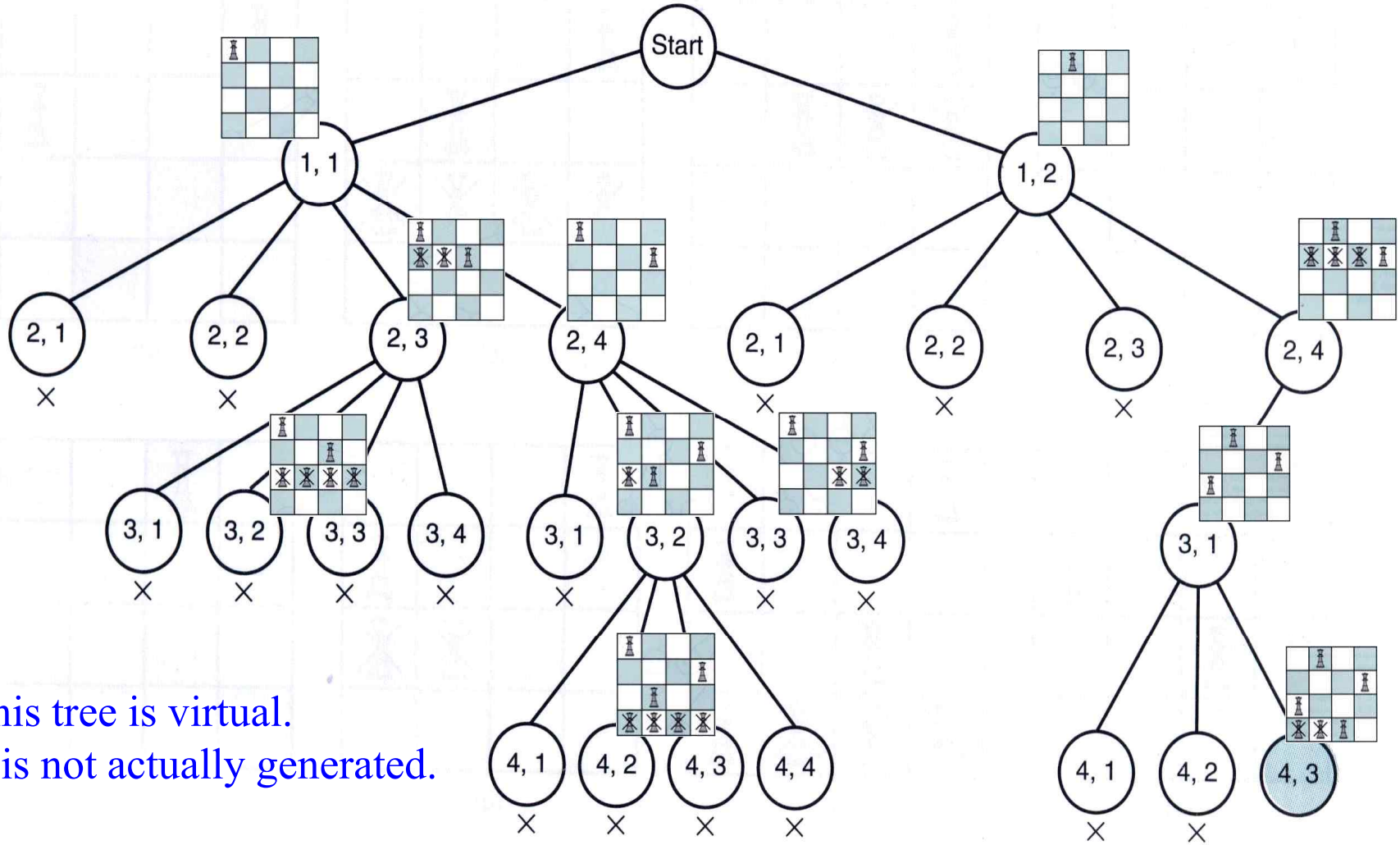
Backtracking (백트래킹)

- A procedure to search the state space tree.
- If a node can not leads to a solution (nonpromising), go back ('backtrack') to the node's parent and proceed with the search on the next child. This is called pruning.
- If a node has a possibility to lead a solution, we call it promising.
- A generic procedure :

```
void checknode (node v){
  node u;
  if (promising(v))
     if (there is a solution at v)
        write the solution;
     else
        for (each child u of v)
           checknode(u);
}
```

Promising function : application dependant

Example (The 4-queen problem)



This tree is virtual.
It is not actually generated.

- Backtracking algorithm for the *n*-queen problem

```
void queens (index i){
  index j;
  if (promising(i))
     if (i == n)
        cout << col[1] through col [n];
     else
        for (j = 1; j <= n; j++){ // See if queen in (i + 1)st
           col[i + 1] = j;        // row can be positioned
           queens(i + 1);         // in each of the n columns.
        }
}
```

  - col[i] : the column where the queen in the *i*-th row is located.
  - col[1], col[2], …, col[i–1] : store the current promising queen's location.

The Function promising for the *n*-queen problem

```
bool promising (index i){
    index k;  bool switch;
    k = 1;
    switch = true;                    // Check if any queen threatens
    while (k < i &&  switch){         // queen in the ith row.
        if (col[i] == col[k] || abs(col[i] - col[k]) == i - k)
            switch = false;
        k++;
    }
    return switch;
}
```

- col[i] == col[k] : check if the two queens are at the same column.
- abs(col[i] – col[k]) == i – k : check if diagonally located.

- Analysis
  - The total number of nodes in the state space tree

$$1 + n + n^2 + n^3 + \cdots + n^n = \frac{n^{n+1} - 1}{n - 1}$$

19,173,961 nodes for n=8

  - The promising nodes are at most

$$1 + n + n(n-1) + n(n-1)(n-2) + \cdots + n!$$

  - Actual number of nodes

109,601 nodes for n=8

n	Number of Nodes Checked by Algorithm 1†	Number of Candidate Solutions Checked by Algorithm 2‡	Number of Nodes Checked by Backtracking	Number of Nodes Found Promising by Backtracking
4	341	24	61	17
8	19,173,961	40,320	15,721	2057
12	9.73×10^{12}	4.79×10^{8}	1.01×10^{7}	8.56×10^{5}
14	1.20×10^{16}	8.72×10^{10}	3.78×10^{8}	2.74×10^{7}

Trying $n!$ possible solutions.

DFS without backtracking (i.e., The number of nodes in the state space tree).

Backtracking Algorithm for Vertex Coloring Problem

Vertex Coloring Problem :

Given a graph G = (V, E) and an integer *m* (number of colors),
is G *m*-colorable?

- Brute Force Algorithm
  - Enumerate all possible color assignment to vertices
    and then determine if the assignment is valid.
  - How many possible assignments? n^m.
  - Time complexity?

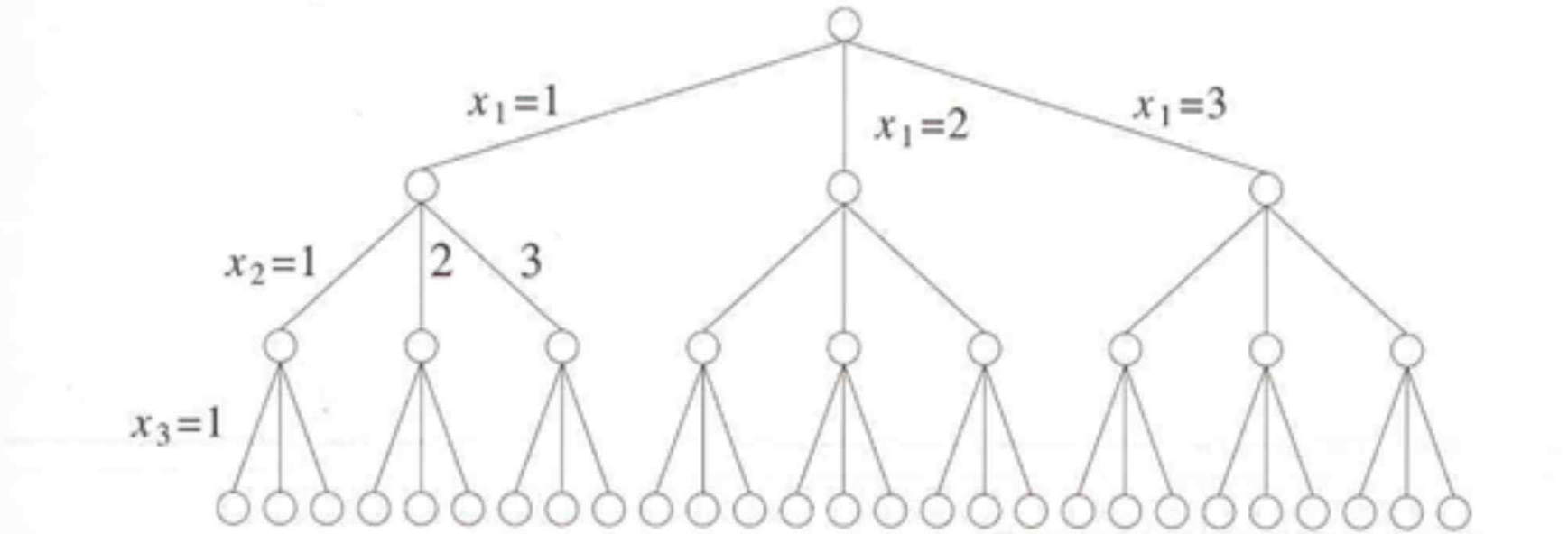


Figure 7.13 State space tree for `mColoring` when n =3 and m = 3

- Backtracking algorithm for the Vertex Coloring problem

```
void m-Coloring (int k)
{
  if (promising(k))
       if (k == n)
            cout << x[1] through x[n];
        else
            for (int j = 1; j <=m; j++) {
                 x[k+1] = j;
                 m-Coloring(k+1);
              }
   }
```

- x[k] : the color assigned to vertex *k*.
- x[1],x[2], …, x[k−1] : store the current promising assignment.

✦ The Function promising for the Vertex Coloring problem.

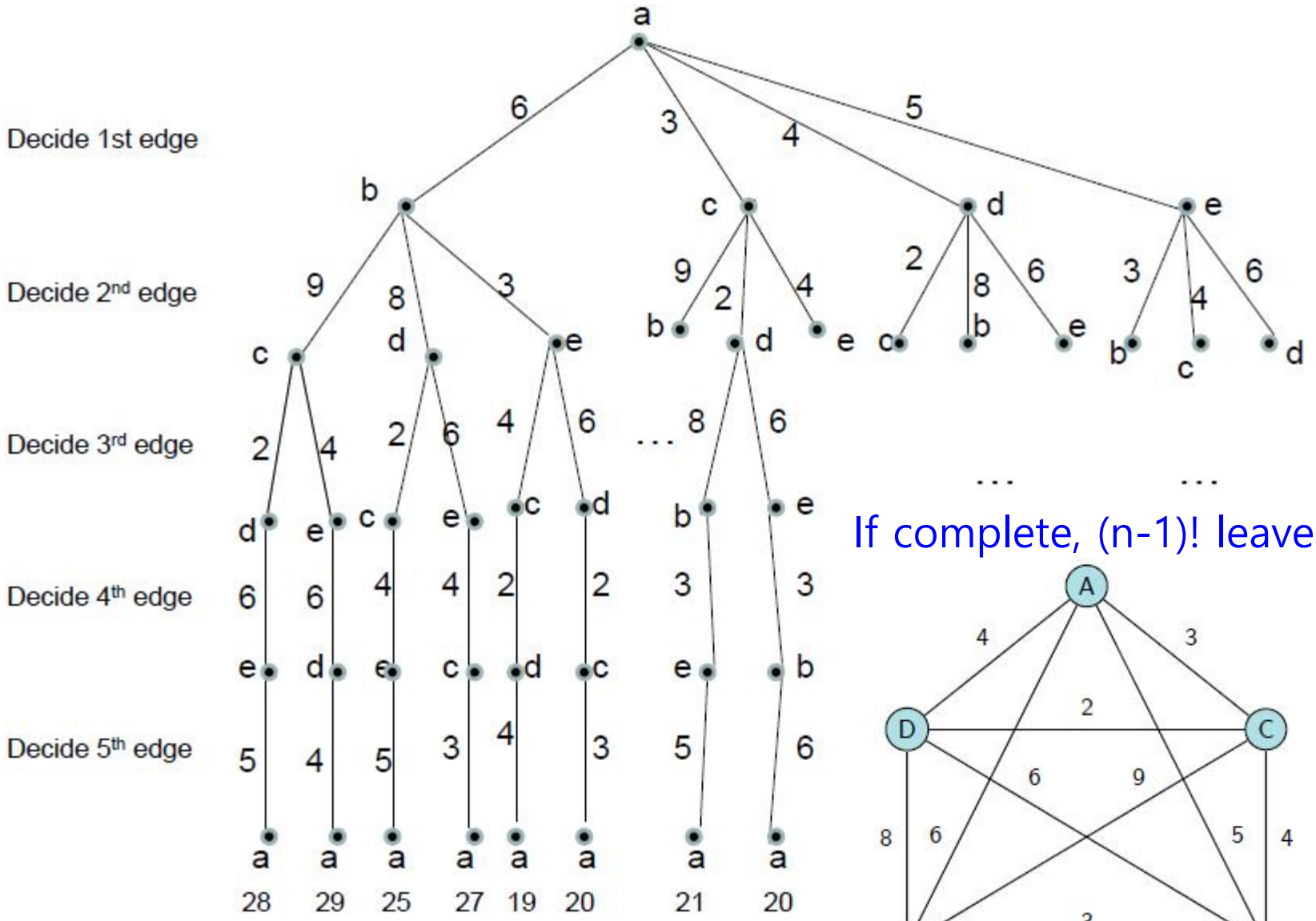
```
boolean promising (int k)
{
   int j;
   for (j = 1; j < k; j++)
      if (G[k][j] && (x[k] == x[j]))
                        return FALSE;
   return TRUE;
}
```

◆ G[][] : adjacency matrix for the graph.

◆ G[k][j] && (x[k] == x[j]) : vertex *k* and vertex *j* are adjacent and assigned the same color.

- Backtracking Algorithm for TSP
  - Brute Force Algorithm
    - Enumerate all tours, and pick one having the smallest cost.
    - How many possible solutions? At most $n!$.
    - Time complexity?

✦State Space Tree



If complete, (n-1)! leaves

✦ Algorithm Outline

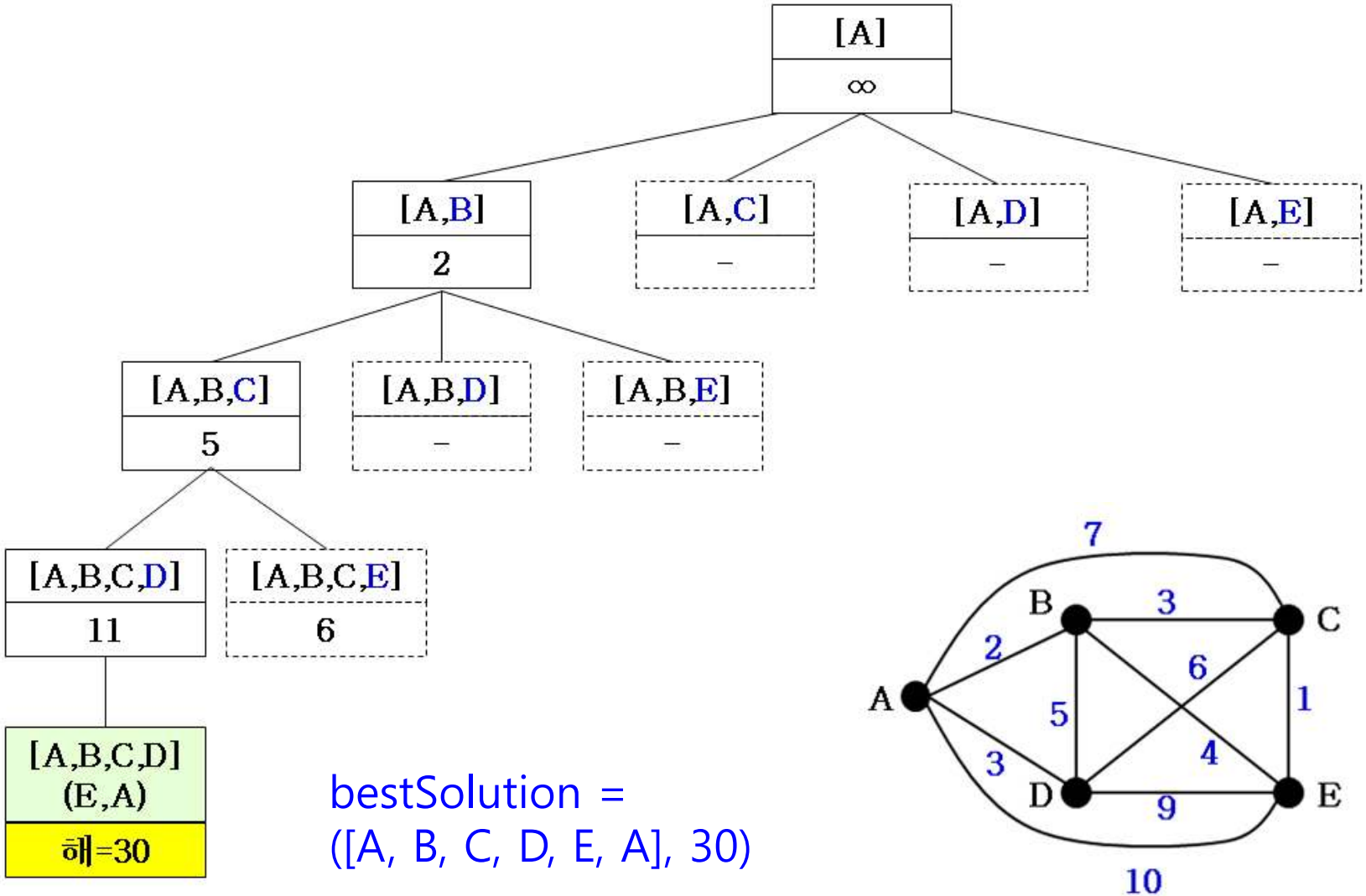
tour = [시작정점] // tour는 a sequence of vertices.

//bestSolution : 현재까지 발견된 최소 cost tour with its cost.

bestSolution = (tour, ∞) // 초기화.

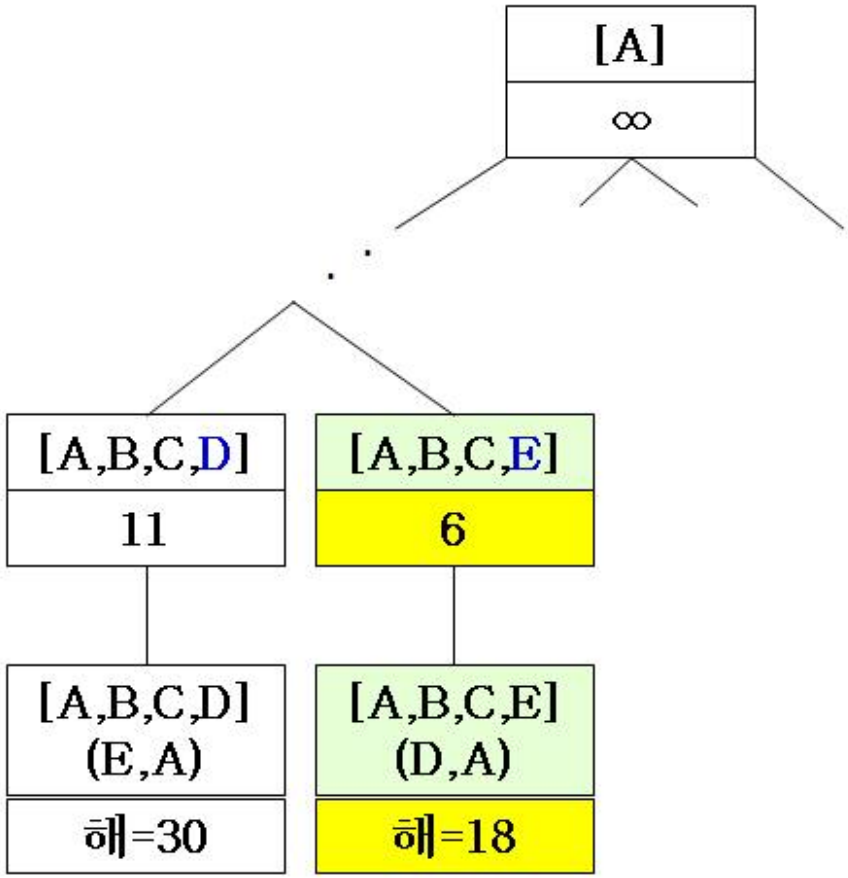
```
BacktrackTSP ( tour )
1.  if (tour가 완전한 해이면)
2.      if (tour의 거리 < bestSolution의 거리)  // 더 짧은 해라면
3.          bestSolution = (tour, tour의 거리);
4.  else {
5.      for (tour를 확장 가능한 각 정점 v에 대해서) {
            newTour = tour + v        // 기존 tour의 뒤에  v 를 추가
7.          if (newTour의 거리 < bestSolution의 거리)
8.              BacktrackTSP(newTour);
9.      }
10. }
```

✦Running Example (1/6)

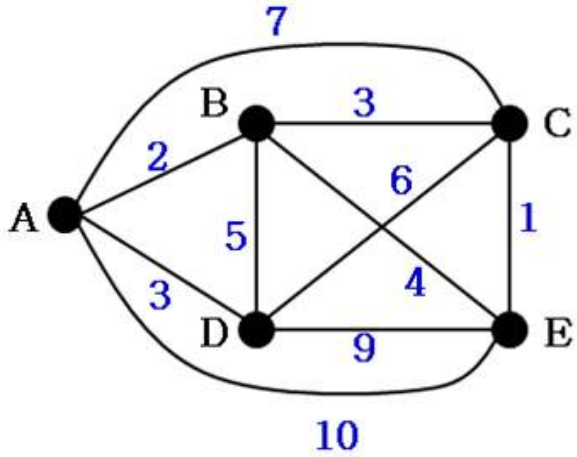


bestSolution =
([A, B, C, D, E, A], 30)

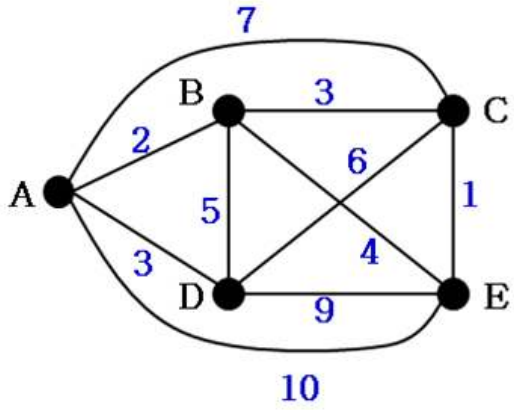
✦Running Example (2/6)



bestSolution =
([A, B, C, E, D, A], 18)

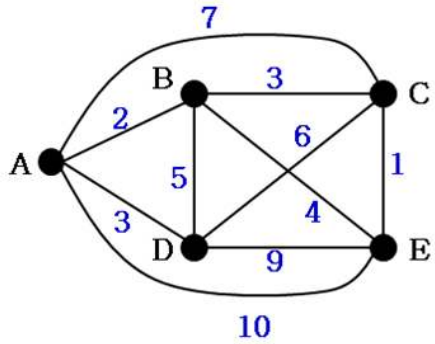


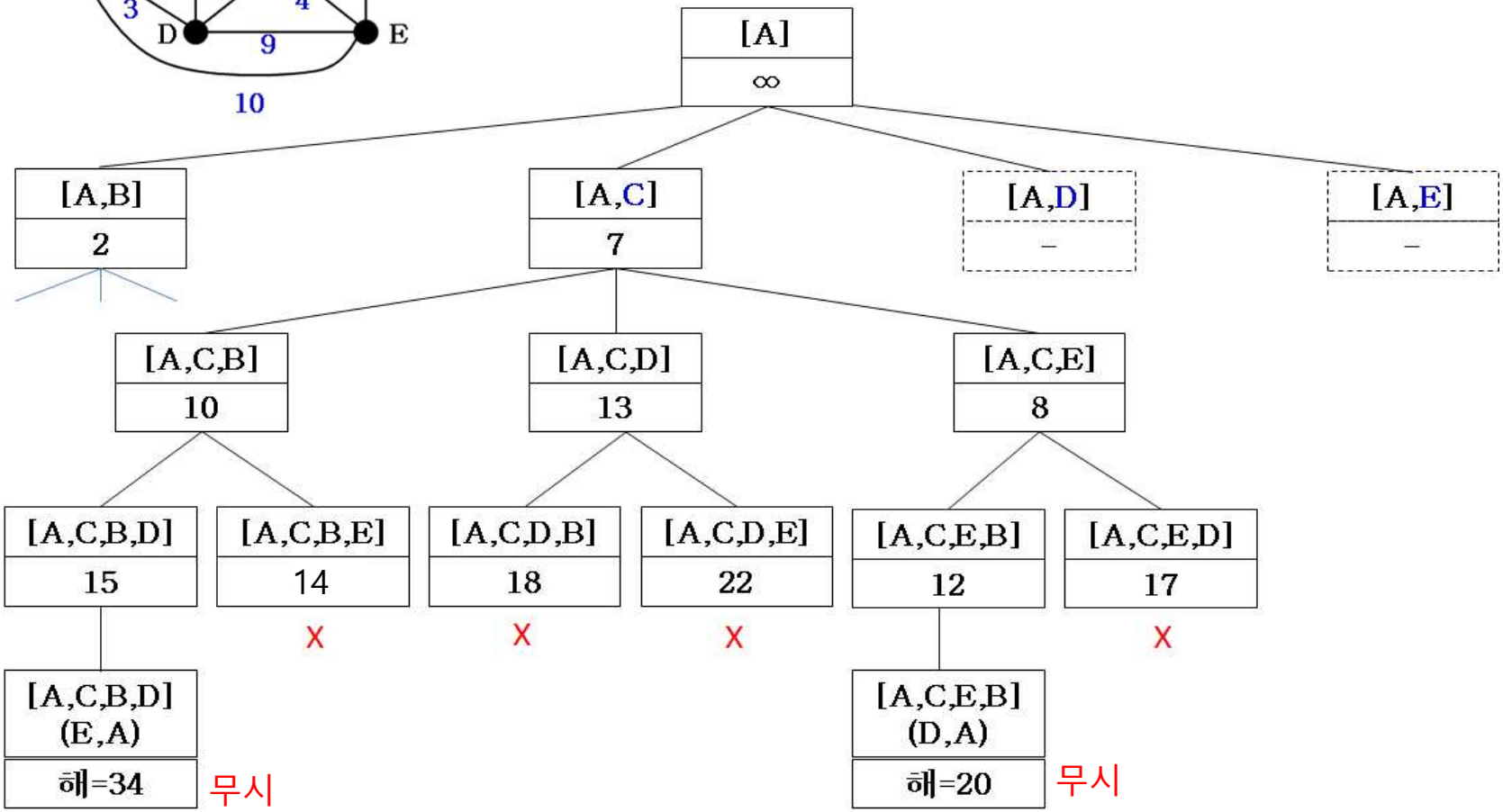
✦Running Example (3/6)



bestSolution = ([A, B, E, C, D, A], 16)

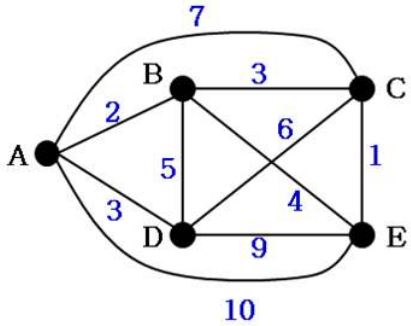
✦Running Example (4/6)

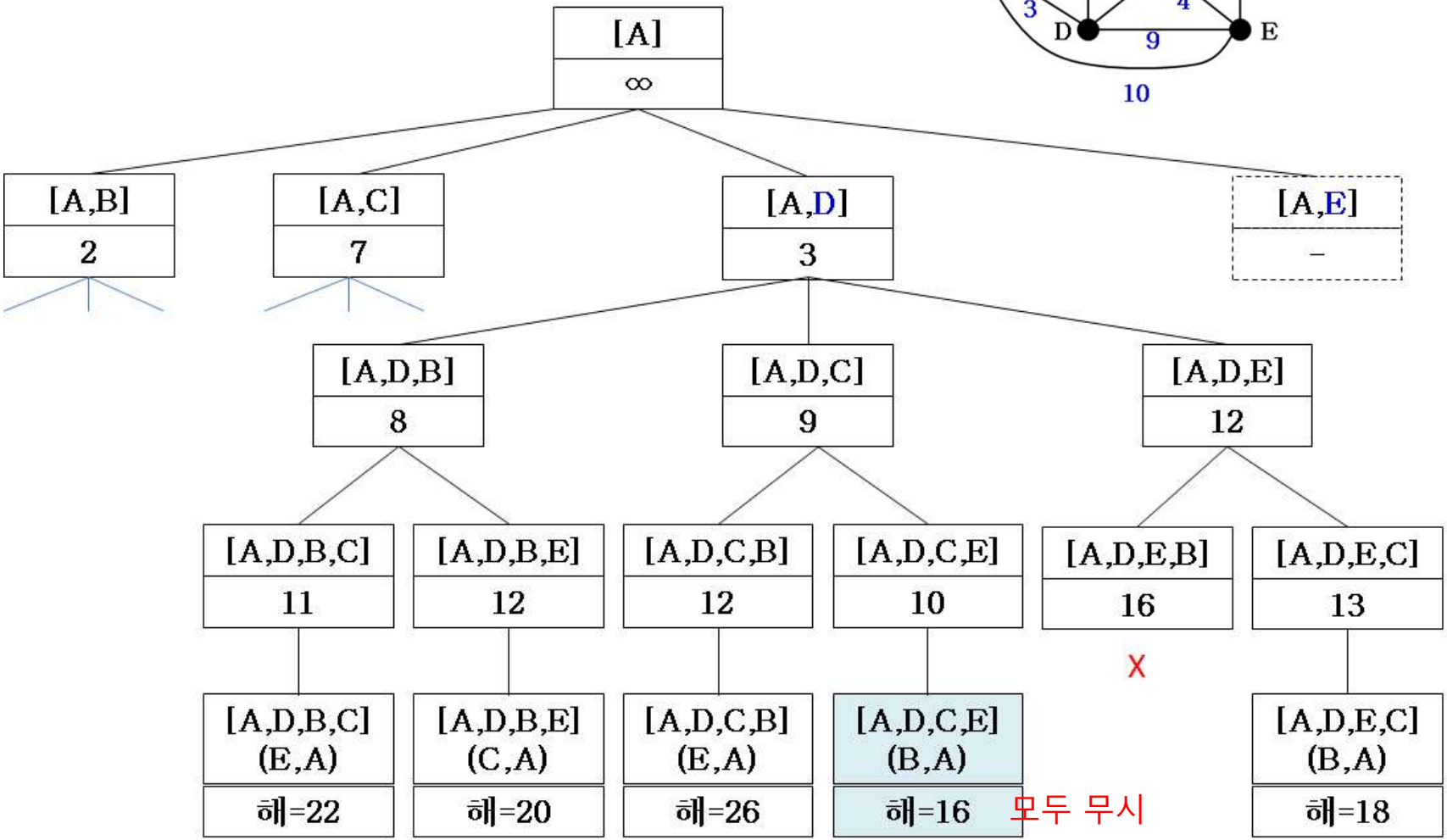




bestSolution = ([A, B, E, C, D, A], 16)

✦Running Example (5/6)

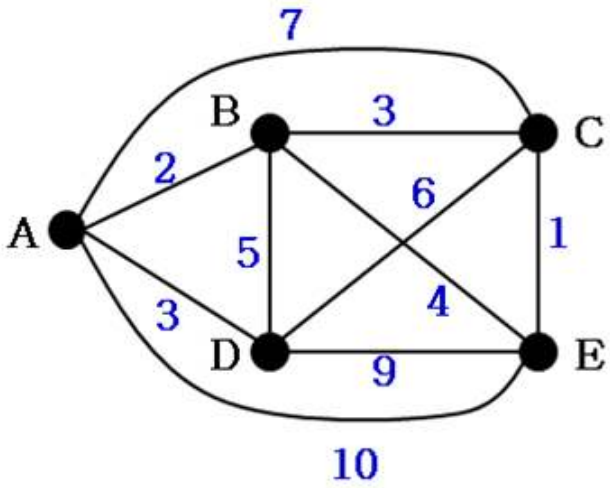


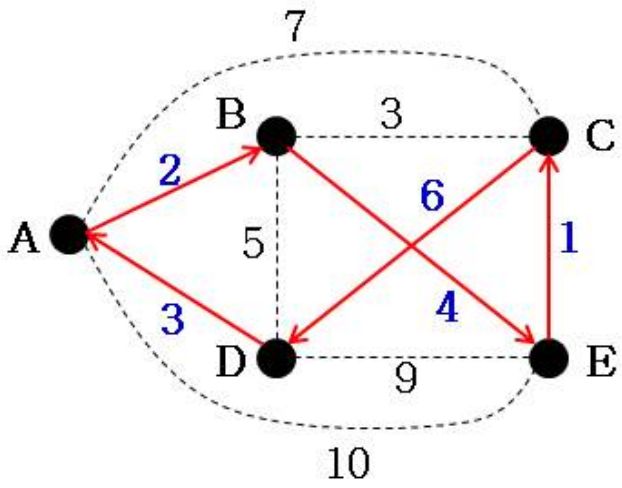


bestSolution = ([A, B, E, C, D, A], 16)

Running Example (6/6)

- 마지막으로 tour=[A,D]에 대해서 탐색을 수행하여도 bestSolution보다 더 우수한 해는 발견되지 않음
- 따라서 최종해= [A,B,E,C,D,A]이고, 거리=16이다.





- 함수를 39번 호출하여 n! = 5! = 120 가지 경우에서 최소 tour를 선택 (state space tree에서 방문 노드 수는 51개)
- 함수 호출 회수와 노드 방문 회수는 입력에 따라 다르다.

- Implementation
  - A[1…n–1] : array of vertex indices. Initially, A[i] = i, i=1..n-1.
  - C[ ][ ] : cost matrix. ∞ if no edge.
  - el : 현재 여기까지 진행됨.
  - lengthSoFar : A[1], A[2],…, A[el]까지 path의 길이.
  - B[1…n–1], minCost : currently best tour and cost.

```
int TSP_Backtrack (int *A,  int n, int **C, int el,  int lengthSoFar,
                                int minCost, int *B  ) {
    int newLength, newCost ;
    if ( el == n && (minCost > lengthSoFar + C[A[n]][A[1]]) )  {
        // found a tour
        minCost = lengthSoFar + C[A[n]][A[1]];
        copy A to B;    // save the currently best solution
    }
   else { … }
    return minCost;
}
```

- Implementation (else part)
  - A[1], …, A[el]은 고정시킨 채로 나머지 A[el+1],…,A[n–1]의 모든 경우에 대해 가능한 tour length를 계산한다.
    - 이를 위하여 (*)에서 A[el+1]과 A[el+1], A[el+2], …, A[n]과 교환하고, 탐색을 마친 후 (**)에서 복구한다.
  - 만일, A[1], …, A[el], A[el+1]까지 경로의 길이가 minCost보다 크다면 더 이상의 탐색을 포기한다.

```
for ( int i = el + 1; i <= n; i++ ) {
    swap( &(A[el + 1]), &(A[i]) );    // check for every possible cases(*)
    newLength = lengthSoFar + C[A[el]][A[el+1]];
     if ( newLength < minCost ) {
         newCost = TSP_Backtrack(A, n, C, el+1, newLength, minCost, B);
         if ( minCost > newCost )  minCost = newCost ;
     }
     swap( &(A[i]), &(A[el + 1]) ); // (**)
}
```

✦ Implementation (전체 함수)

◆ 함수호출 : minCost = TSP_Backtrack(A, n, C, 1, 0, ∞, B);

```
int TSP_Backtrack (int *A, int n, int **C, int el, int lengthSoFar, int minCost, int *B) {
    int newLength, newCost ;
    if ( el == n && (minCost > lengthSoFar + C[A[n]][A[1]]) )  {
        minCost = lengthSoFar + C[A[n]][A[1]]; copy A to B;    // save the current best
   }
   else
      for ( int i = el + 1; i <= n; i++ ) {
         swap( &(A[el + 1]), &(A[i]) );    // check for every possible cases(*)
         newLength = lengthSoFar + C[A[el]][A[el+1]];
         if ( newLength < minCost ) {
              newCost = TSP_Backtrack(A, n, C, el+1, newLength, minCost, B);
              if ( minCost > newCost )  minCost = newCost ;
         }
         swap( &(A[i]), &(A[el + 1]) ); // (**)
      }
    return minCost;
}
```

- Backtracking Algorithm for 0/1 Knapsack Problem.
  - Brute Force Algorithm
    - Enumerate all possible subsets, and pick one feasible subset having the largest profit.
    - How many possible solutions? At most 2^n.
    - Time complexity?
  - State space tree for Backtracking Algorithm
    - The root of the state space tree represents the state that no item has been considered yet.
    - Left child at level i represents the inclusion of i-th item to the state of its parent.
    - Right child at level i represents the disclusion of i-th item to the state of its parent.
    - The state is represented by an array with value 0 or 1.

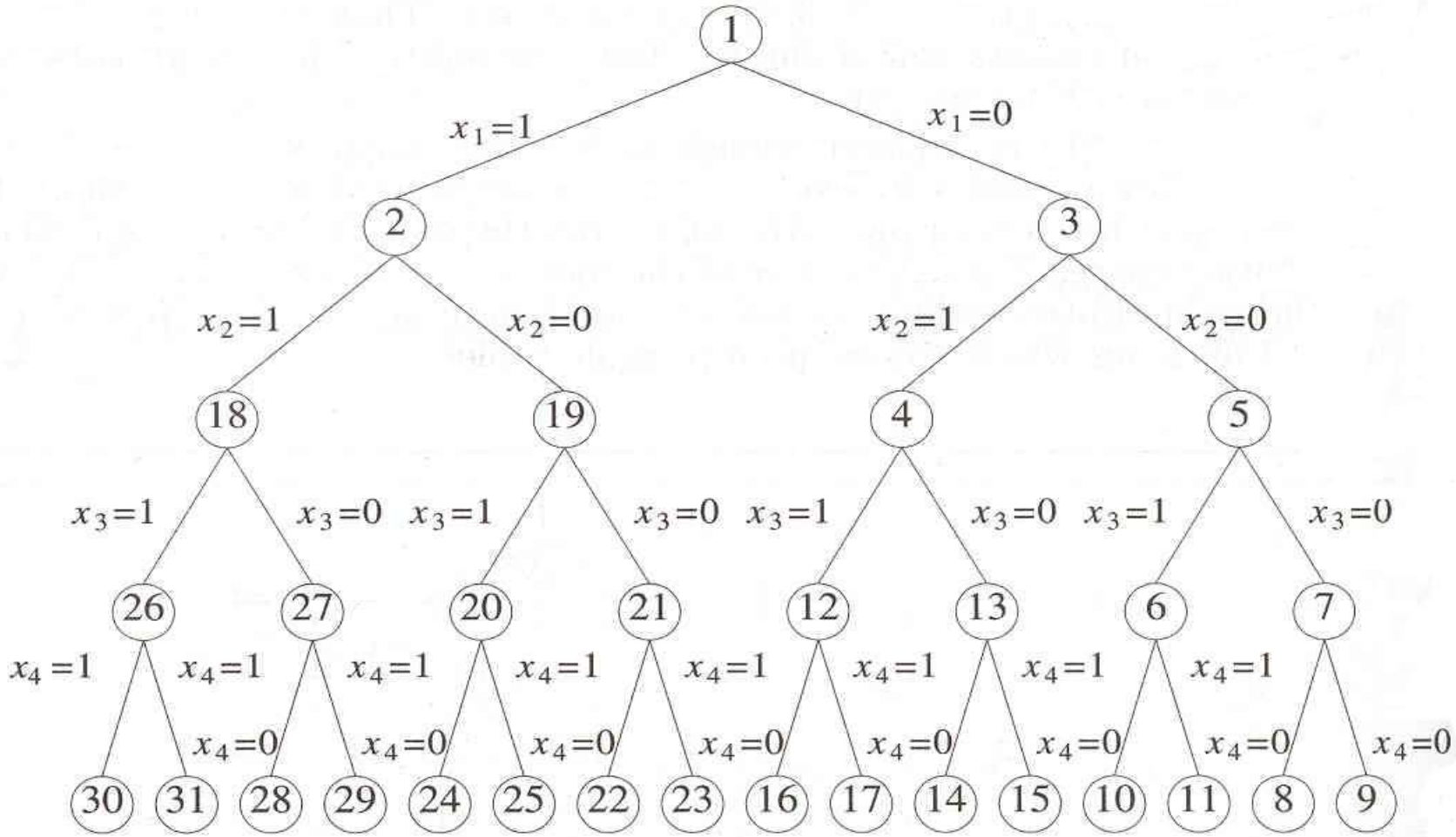


Figure 7.4 Another possible organization for the sum of subsets problems. Nodes are numbered as in *D*-search.

- Assumptions for Backtracking algorithm

  n : the number of items.

  p[] and w[] : profits and weight od items respectively.

  C : the capacity of knapsack.

  x[] : represents the state of the current best solution.

  y[] : represents the state of the current node.

  fp and fw : maximum profit and its weight.

The items are arranged such that $p[i]/w[i] \geq p[i+1]/w[i+1]$.

In the algorithm, a node is not promising

if the node can not be expanded (i.e., the state of that node is not feasible) or it is not possible to generate a solution better than the current best solution.

- Backtracking algorithm for 0/1 Knapsack problem

```
void BKnap (int k, float cp, float cw)
{
  if (cw + w[k] <= C)   {        // left child is promising.
      y[k] = 1;
      if (k < n)  BKnap(k+1, cp + p[k], cw + w[k]);
      if ((cp+p[k] > fp) && (k == n))  {
                fp = cp + p[k];   fw = cw + w[k];
                for (j = 1; j <= n; j++)  x[j] = y[j];
        }
    }
    if (Bound(cp, cw, k) >=  fp)   {     // right child is promising.
      y[k] = 0;
      if (k < n)  BKnap(k+1, cp, cw );
      if ((cp > fp) && (k == n))  {
                fp = cp;   fw = cw;
                for (j = 1; j <= n; j++)  x[j] = y[j];
                  }
     }
}                                  // initial call : BKnap(1, 0, 0)
```

Bounding function for Knapsack problem

```
float Bound (float cp, float cw, int k)
{       // using the greedy algorithm for fractional Knapsack.
  float a = cp;  b = cw;
  int j;
  for (j = k+1; j <= n; j++)   {
       b += w[j];
      if (b < C)   a += p[j];
      else  return (a + (1 – (C – b)/w[j]) * p[j]);
     }
   return (a);
  }
```

[Example] An instance : n = 4, p = (40, 30, 50, 10),
w = (2, 5, 10, 5), C =16.

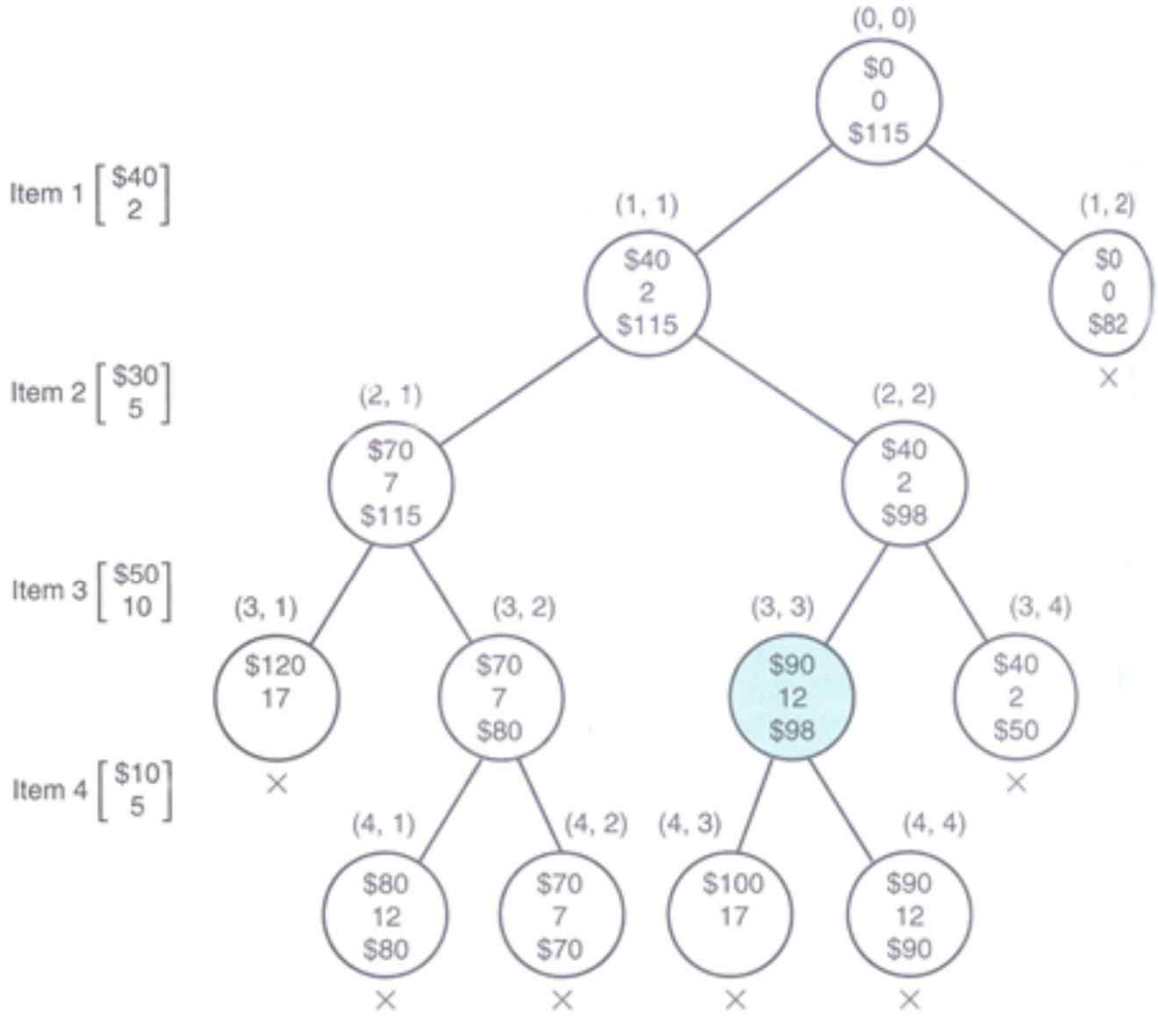


Figure 5.14 • The pruned state space tree produced using backtracking in Example 5.6. Stored at each node from top to bottom are the total profit of the items stolen up to the node, their total weight, and the bound on the total profit that could be obtained by expanding beyond the node. The optimal solution is found at the node shaded in color. Each nonpromising node is marked with a cross.

9.2 Branch-and-Bound(분기 한정 기법)

- Backtracking 기법은 State space tree를 Depth First Search 방식으로 해를 찾아가기 때문에 입력의 크기가 클 경우,

  특히 최적화 문제에 있어 해를 찾는데 시간이 많이 걸릴 뿐 아니라, 경우에 따라 해를 찾는 것이 불가능 할 수 도 있다.

- 분기한정 기법은 State space tree를, 예를 들어, Breadth First Search 방식 으로 해를 찾는 기법이다.

- 분기한정 기법은 State space tree의 각 노드(부분해)에 특정한 값(한정값)을 부여하고 , 이 값을 활용하여 가지치기를 함으로써 Backtracking 기법 보다 빠르게 해를 찾는다.

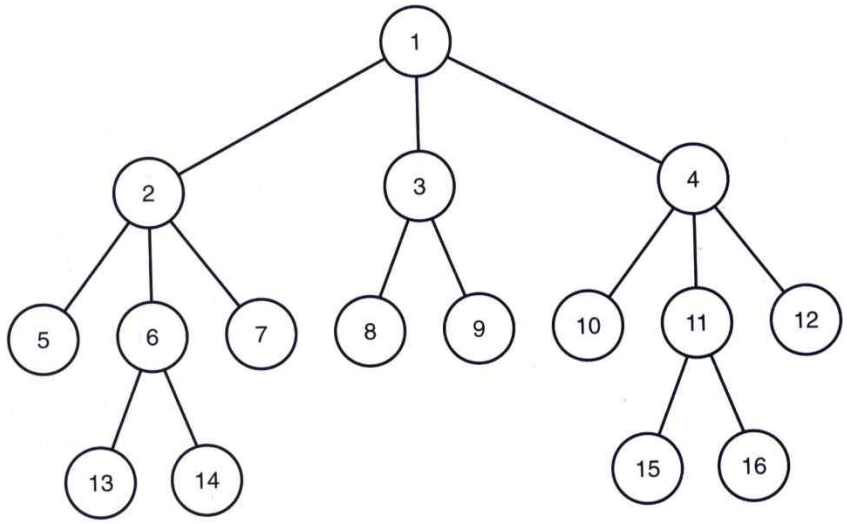
  - Best First Search 방식

Breath First Tree Searching

- Visits nodes level by level (from low to high)

```
void breadth_first_tree_search (tree T){
  queue_of_node Q;  node u, v;
  initialize(Q); // Initialize Q to be empty.
  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);
} } }
```

- Example



Branch-and-Bound (분기 한정 기법)

- Search the state space tree in BFS like fashion.
- Use the same strategy of promising checking to stop or continue searching as the backtracking.
- If no preference in selecting a node at each level (just search FIFO based), we call the method breath-first search with branch-and-bound pruning.
- We may give some preference in selecting a node at each level for searching. In this case we call the method best-first search with branch-and-bounding pruning.

- Lower Bound (한정값)
  - State space tree를 탐색하는 도중, 어떤 노드에 도착했을 때 얻을 수 있는 cost의 최소 값.
  - 계속 탐색해도 lower bound 보다 작은 cost를 갖는 최적해를 구할 수 없다.
- Upper Bound
  - State space tree를 탐색하는 도중, 어떤 노드에 도착했을 때 이후 탐색으로 얻을 수 있는 cost의 최대 값.
  - 어떠한 경우에도 최적해의 cost는 upper bound 보다 클 수 없다.

  예를 들어 TSP에서 현재 best solution의 tour length를 L이라고 하면, optimal tour의 length는 L보다 작거나 같기 때문에 이후 탐색 노드에서 lower bound가 L보다 크다면 더 확장할 필요가 없다.
- 따라서, 탐색 중 lower bound 나 upper bound를 현재 best solution의 값과 비교하여 그 아래 레벨로의 탐색이 필요한지 또는 무의미한지를 결정할 수 있다.

Branch-and-Bound 알고리즘의 형태

- 최소값을 최적해로 갖는 문제에 대한 Best-first search에 의한 branch-and-bound 알고리즘이다.
- 노드들을 priority queue를 사용하여 유지하여야 한다.

Branch-and-Bound(S)

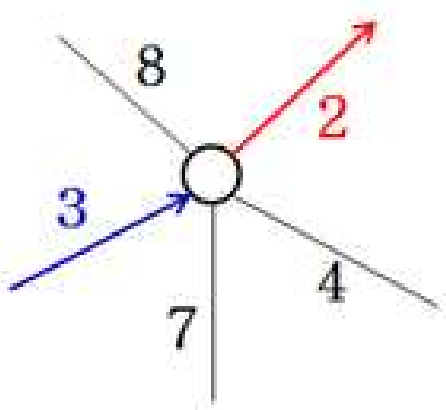
1. 상태 S의 한정값을 계산한다. // lower bound 계산
2. activeNodes = { S } // 탐색되어야 하는 상태의 집합
3. bestValue = ∞ // 현재까지 탐색된 해 중의 최소값
4. while (activeNodes ≠∅) {
5. S_{min}= activeNodes의 상태 중에서 한정값이 가장 작은 상태
6. S_{min}을 activeNodes에서 제거한다.
7. S_{min}의 자식 (확장 가능한) 노드 S'_1, S'_2, …, S'_k를 생성하고, 각각의 한정값을 계산한다.

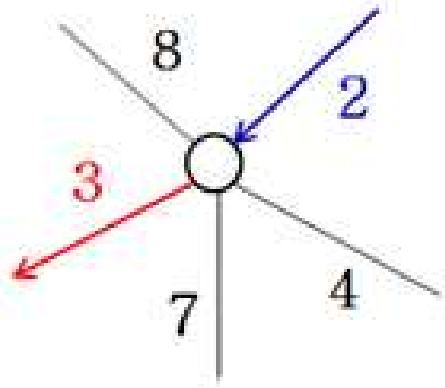
- 한정값 계산은 문제에 따라 다르다.
- Priority queue로 heap, 2-3 tree, Red-Black tree 등의 구조를 사용할 수 있다.

```
8.    for i = 1 to k {      // 확장한 각 자식 S'i에 대해서
9.        if (S'i의 한정값 ≥ bestValue)
10.           S'i를 가지치기한다.  // S'i로부터 탐색은 무의미.
11.       else if (S'i가 완전한 해이고 S'i의 값 < bestValue)
12.          bestValue = S'i의 값
13.          bestSolution = S'i
14.       else
15.          S'i를 activeNodes에 추가. // 나중에 차례가 되면
                                        // S'i로부터 탐색을 수행.
          }
    }
```

A Branch-and-Bound TSP Algorithm

- Lower Bound (한정값) 계산
  - TSP 문제에서 tour란 하나의 vertex에서 시작하여 graph의 모든 다른 vertex들을 정확히 한번씩만 방문한 후 시작 vertex로 돌아오는 cycle이다.
  - 따라서, G의 각 vertex에 인접한 edge 중 정확히 두 개의 edge가 TSP tour에 포함될 것이다.
  - 임의의 한 vertex 관점에서 볼 때 인접한 edge 중 cost가 작은 두 개의 edge가 tour에 포함 되는 것이 가장 이상적이다.





- 앞장의 내용을 근거로 각 노드의 한정값(lower bound) L은 다음과 같이 계산한다.

현재 노드의 label이 $[v_1, v_2, \ldots, v_k]$라 하자.

$T = [v_1, v_2, \ldots, v_k]$ 는 현재 만들어진 중간 단계 tour;

모든 정점 v에 대해 인접한 두 개의 edge es1과 es2를 아래와 같이 선택하여 그 가중치를 합을 LB라 하자.

v_1 : edge (v_1, v_2)과 v_1 에 인접한 다른 egde 중 가중치가 가장 작은 edge.

v_k : edge (v_{k-1}, v_k)과 v_k 에 인접한 다른 egde 중 가중치가 가장 작은 edge.

v_i $(1<i<k)$: edge (v_{i-1}, v_i)과 edge (v_i, v_{i+1}).

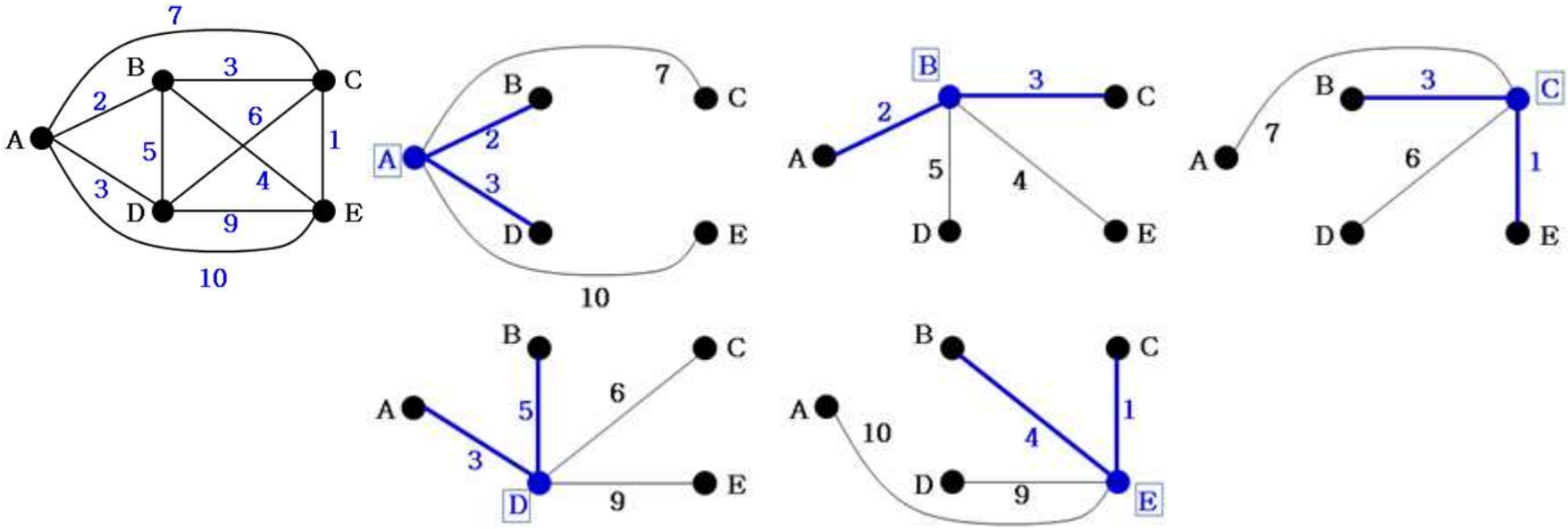
T에 속하지 않은 v : v에 인접한 edge 중 가중치가 가장 작은 두 개의 edge.

$L = LB/2$.

- 마지막에 2로 나눈 것은 한 vertex에서 나가는 edge는 인접 vertex에서는 들어오는 edge이므로 중복 계산되기 때문.

- 이렇게 구한 L 값은 T를 바탕으로 tour를 만들었을 때 얻을 수 있는 최적의 tour 길이보다 작거나 같다.

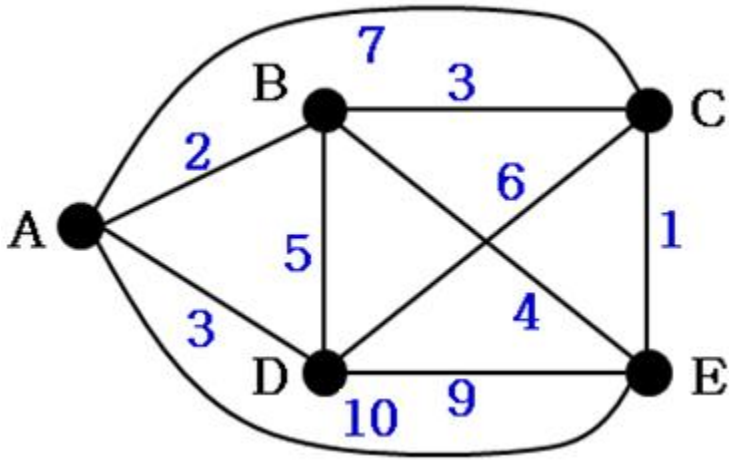
◆Example : T = [A] 인 경우 (즉, 시작할 때) lower bound



- 각 vertex에서 가중치가 작은 두 edge의 가중치

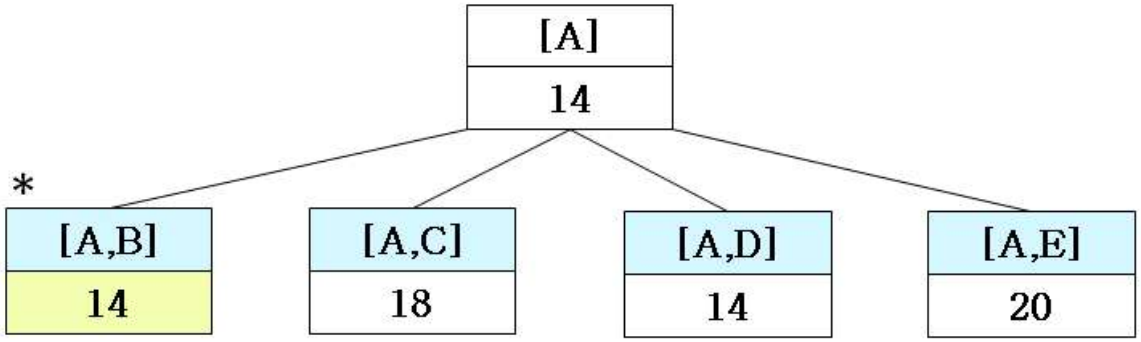
  A : (2, 3), B : (2, 3), C : (1, 3), D : (3, 5), E : (1, 4)
- LB = [(2+3)+(2+3)+(1+3)+(3+5)+(1+4)]/2 = 27/2 = 14
- 참고
  - Optimal tour의 가중치 합은 14보다 작을 수 없다.
  - 반올림 하지 않고 소수점 이하를 버려도 무방 (즉, 13)

- Running Example
  - Input Instance



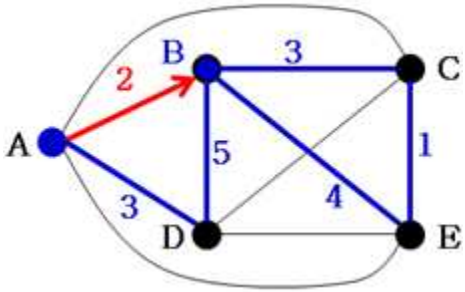
  - A = 시작 vertex.
  - 초기 상태 T = [A].
  - Upper bound, 즉, bestValue = ∞
  - Branch-and-Bound([A])를 호출하여 탐색 시작.
  - Lower Bound 값은 14 (앞에서 계산)

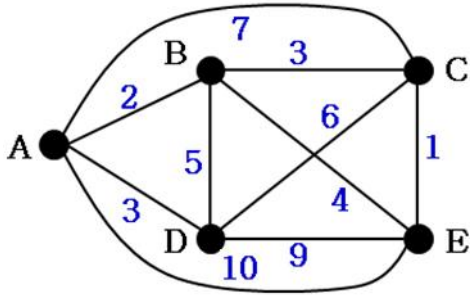
◆[A]의 child 노드 생성



◆Lower bound of each state node

[A,B]

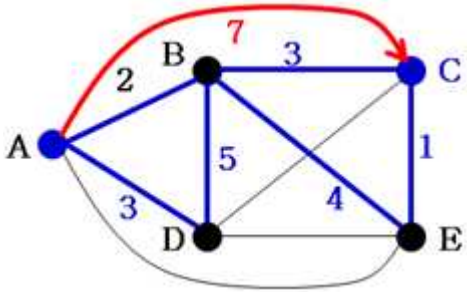


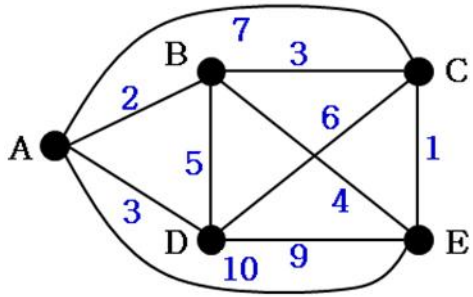


LB = ((2+3)+(2+3)+(3+1)+(4+1)+(3+5))/2
= 14

- Lower bound of each state node

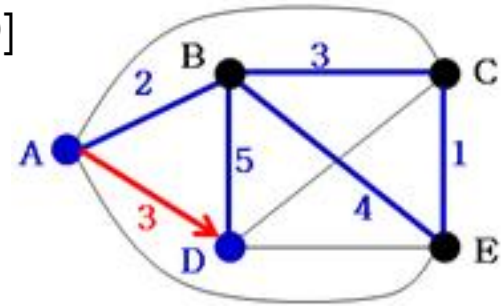
[A,C]





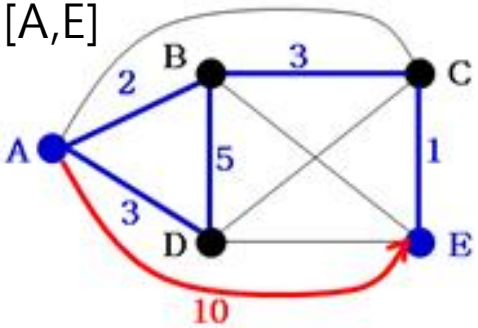
LB = ((2+7)+(2+3)+(7+1)+(3+5)+(4+1))/2
= 18

[A,D]



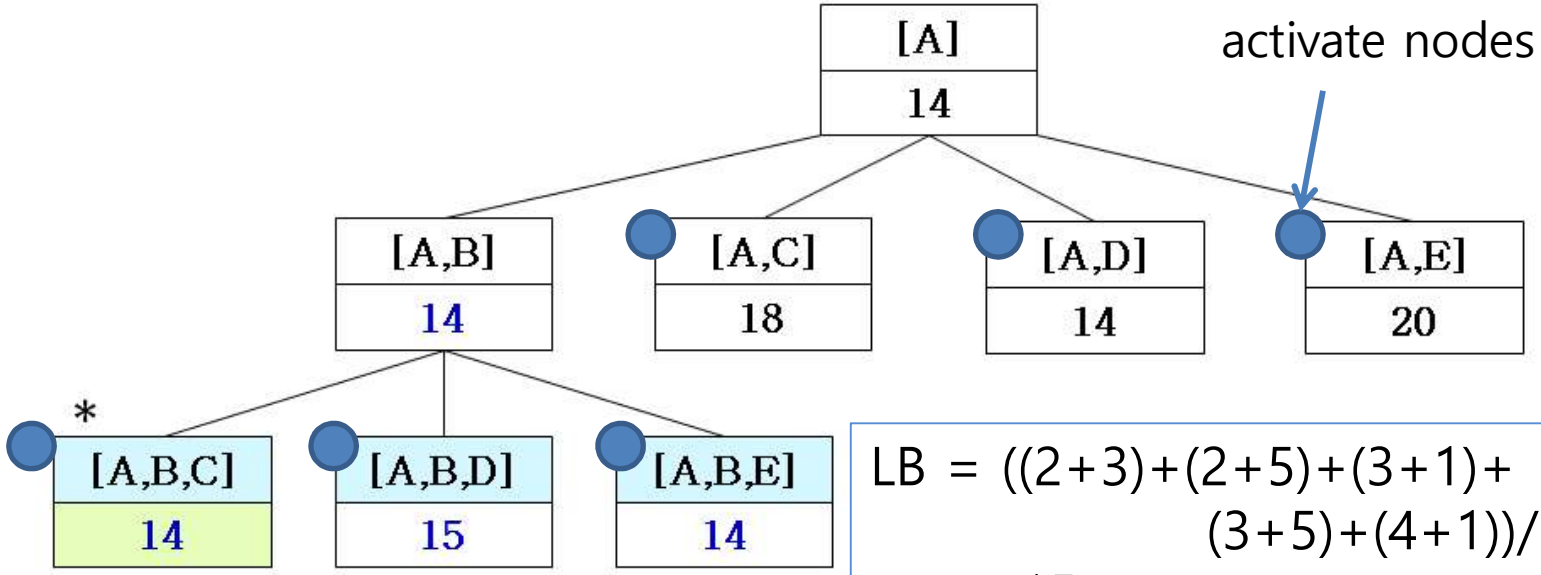
LB = 14 ; the same as [A,B]

[A,E]



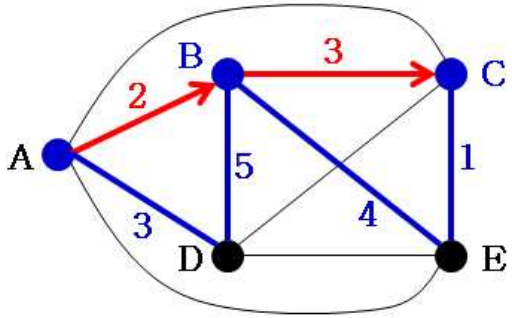
LB = ((2+10)+(2+3)+(3+1)+
(3+5)+(10+1))/2
= 20

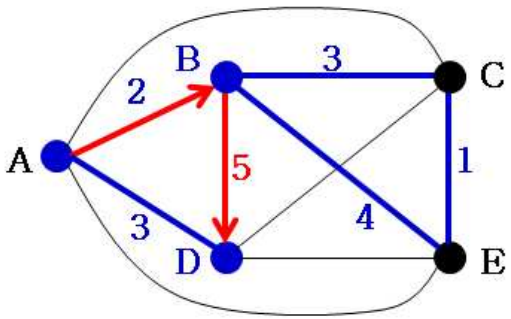
- Next active node is [A,B] or [A,D]
- Create the children of [A, B]

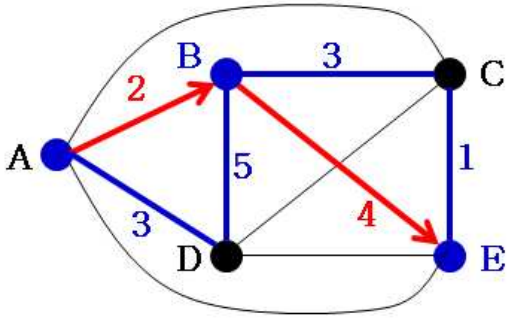


LB = ((2+3)+(2+5)+(3+1)+
(3+5)+(4+1))/2)
= 15

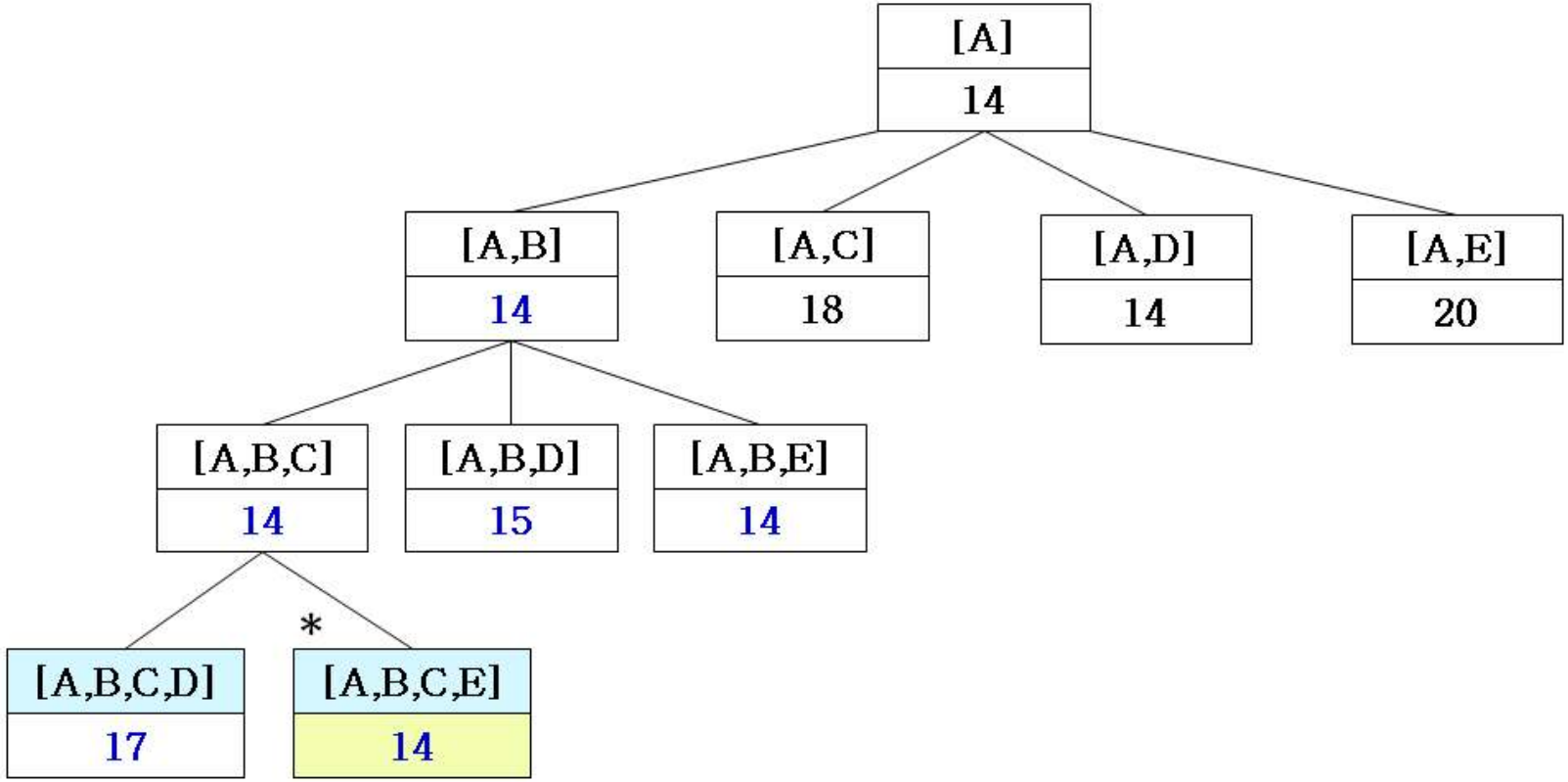
- Lower Bound Calculation





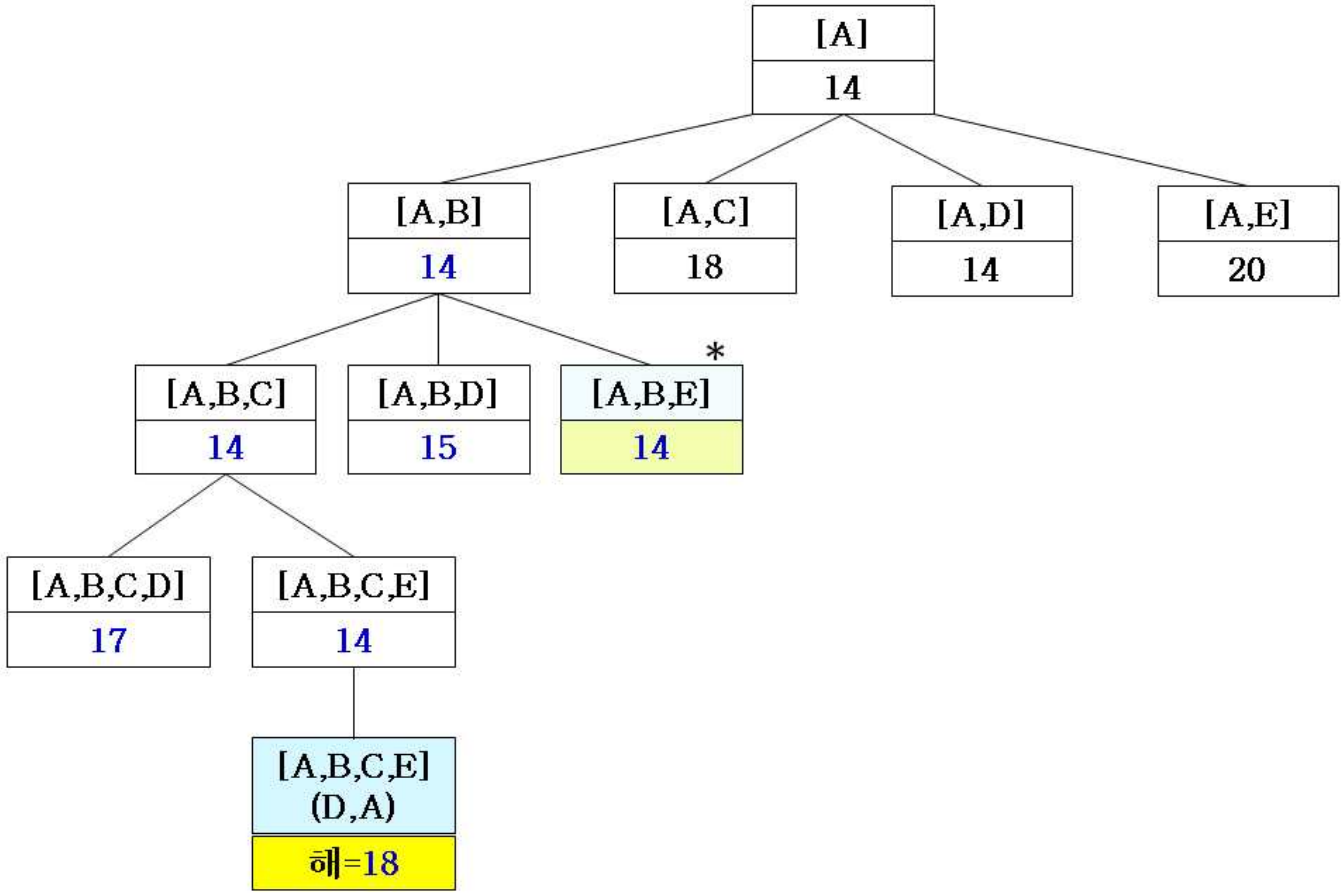


- Next active node is [A,B,C] , [A,B,E] or [A,D].



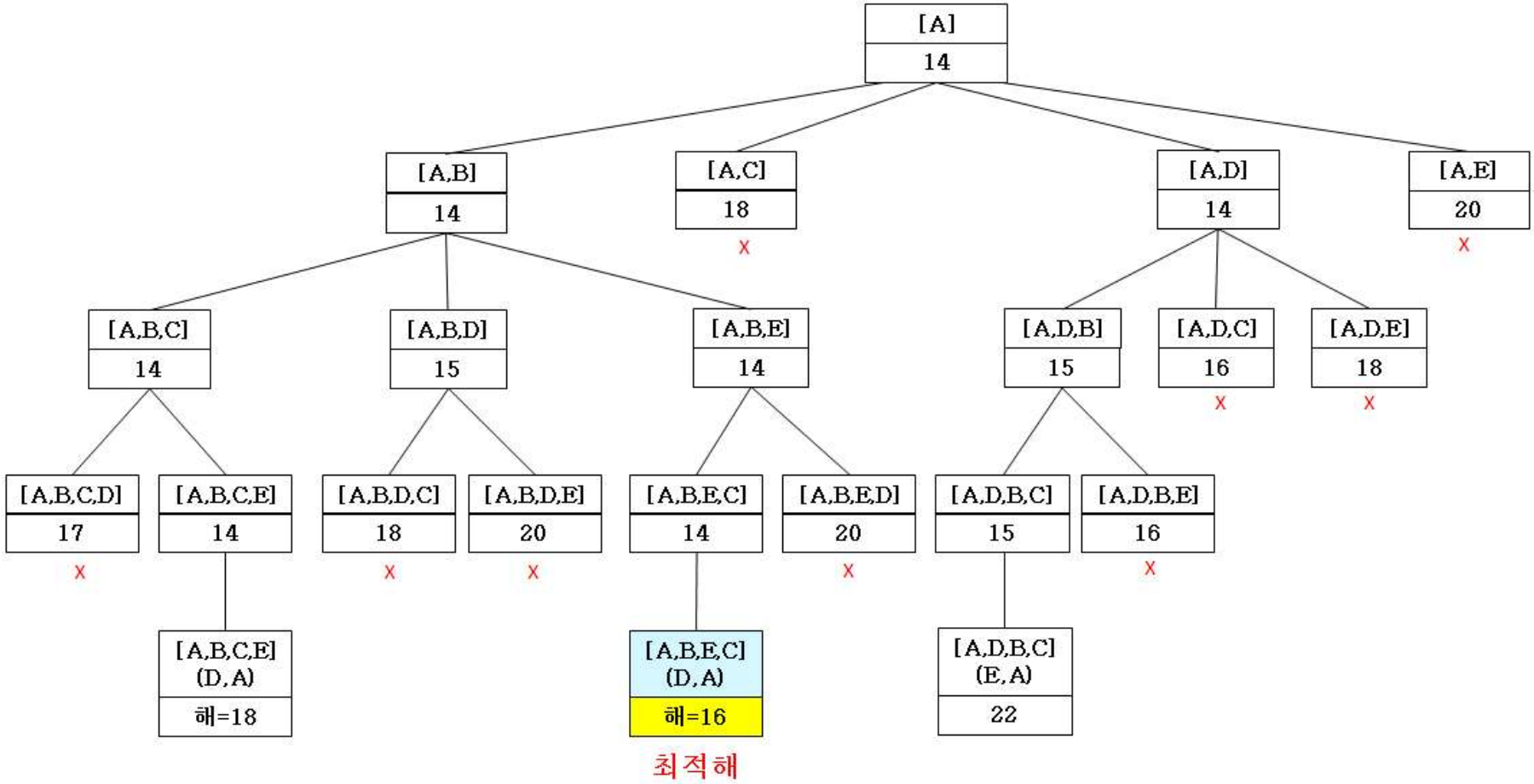
- [A,B,C,D]의 LB: ([2+3]+[2+3]+[6+3]+[3+6]+[1+4])/2 = 33/2 = 17
- [A,B,C,E]의 LB: ([2+3]+[2+3]+[1+3]+[3+5]+[1+4])/2 = 27/2 = 14

- Choose next active node is [A,B,C,E]



- A tour is found.
- Set upper bound from ∞ to 18.

Final Search Result



- Branch & Bound : 22 nodes visited.
- Backtracking : 51 nodes visited.
- Branch & Bound is better (not always)

- A Branch-and-Bound Algorithm for 0/1 Knapsack Problem.
  - Upper Bound (한정값) 계산
    - 앞에서 사용한 함수 Bound를 사용.

[Example]

An instance : n = 4, p = (40, 30, 50, 10),

w = (2, 5, 10, 5), C =16.

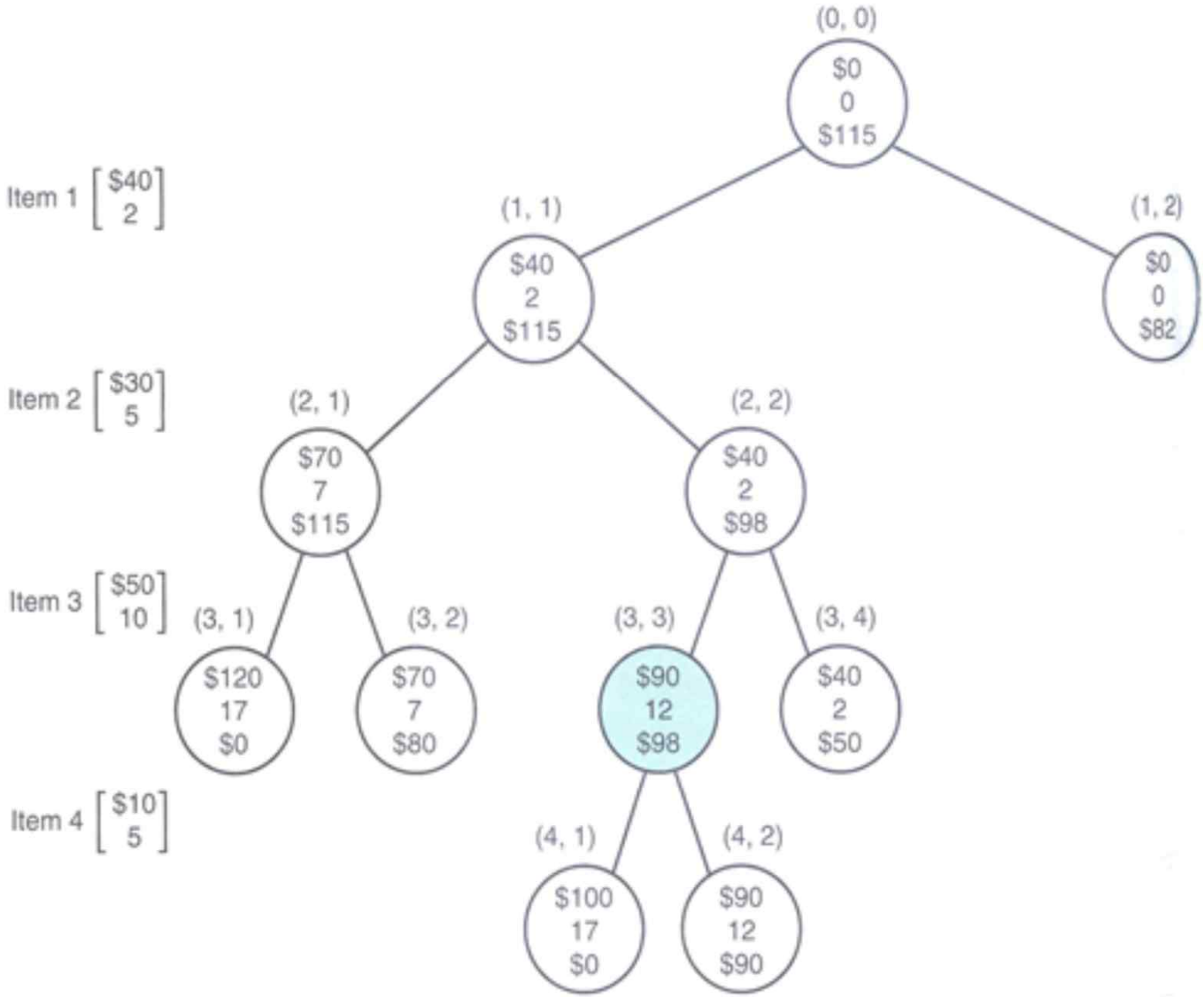


Figure 6.3 ● The pruned state space tree produced using best-first search with branch-and-bound pruning in Example 6.2. Stored at each node from top to bottom are the total profit of the items stolen up to the node, their total weight, and the bound on the total profit that could be obtained by expanding beyond the node. The node shaded in color is the one at which an optimal solution is found.

- 시간이 된다면
  - Genetic Algorithm과 Simmulated Annealing에 대해서도 교재를 읽어 보세요.