Branch-and-Bound Algorithm

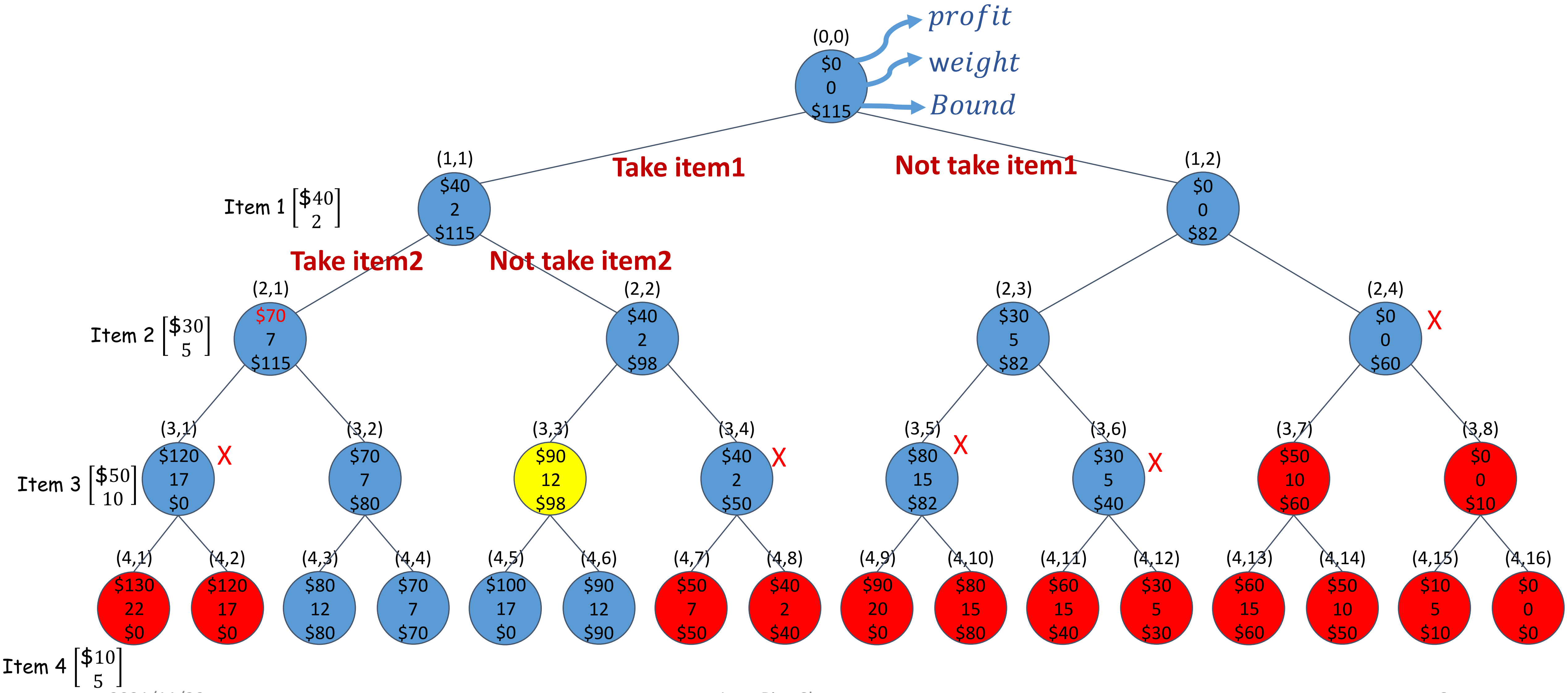
0-1 knapsack problem

- Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the **total weight is less than or equal to a given limit** and **the total value is as large as possible**.
- Example : n=4 ,W=16

i : item
p_i : profit
w_i : weight

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

Breadth-First Search (take or not take)



Calculation of Bound

If the node is at level i, and the node at level k is the one whose weight would bring the weight above W, then

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

Breadth-First Search with Branch-and-Bound Pruning

Calculate the upperbound

- (0, 0)

  $totweight = 2 + 5 = 7$

  $bound = (40 + 30) + (16 - 7) * (50/5)$

- (1, 1) ---take item 1

  $totweight = 2 + 5 = 7$

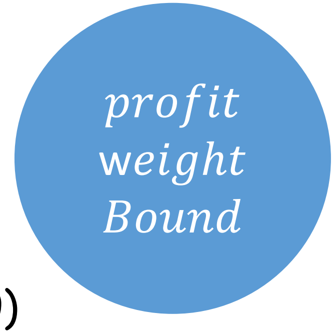
  $bound = (40 + 30) + (16 - 7) * (50/5) = 115$

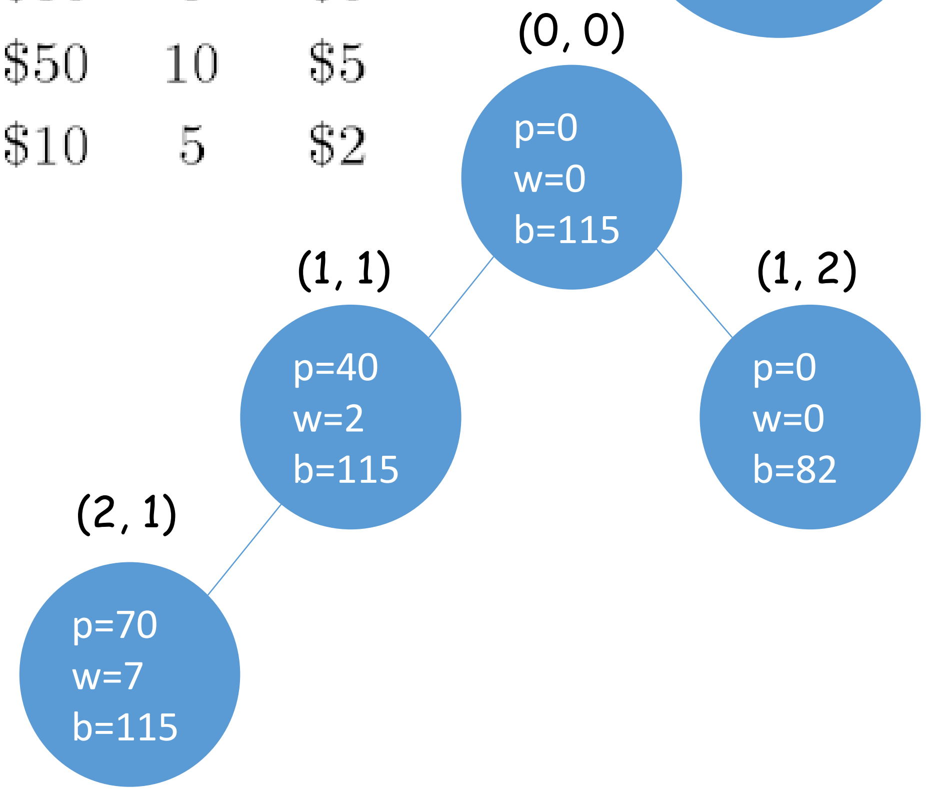
- (1, 2) ---don't take item 1

  $totweight = 5 + 10 = 15$

  $bound = (30 + 50) + (16 - 15) * (10/5) = 82$

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





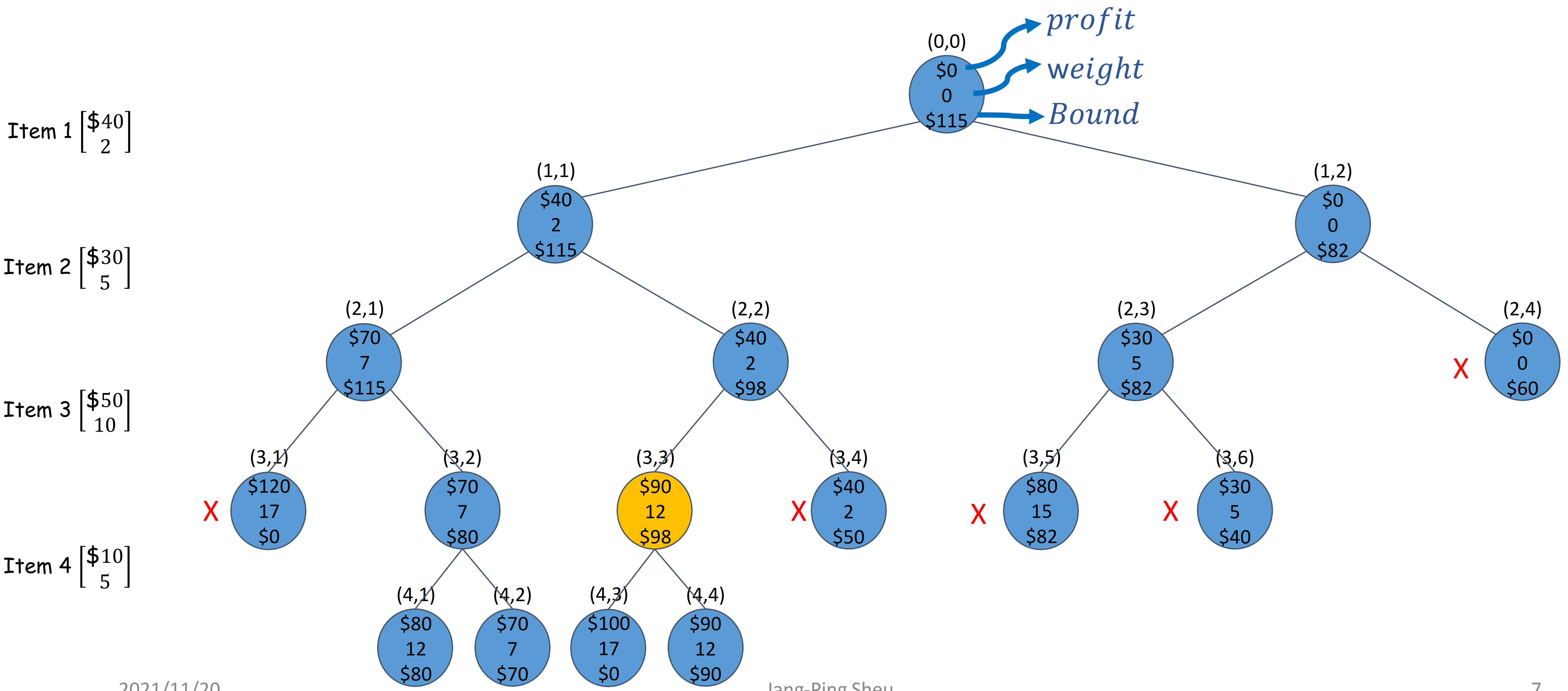
Non-promising

- A node is non-promising if this bound is less than or equal to $maxprofit$, which is the value of the best solution found up to that point.

- Recall that a node is also non-promising if

$$weight \geq W$$

Breadth-First Search with Branch-and-Bound



General algorithm

```
void breadth_first_branch_and_bound (state_space_tree T,
                                     number& best)
{
  queue_of_node Q;
  node u, v;

  initialize(Q);                          // Initialize Q to be empty.
  v = root of T;                          // Visit root.
  enqueue(Q, v);
  best = value(v);
  while (! empty(Q)){
     dequeue(Q, v);
     for (each child u of v){             // Visit each child.
        if (value(u) is better than best)
           best = value(u);
        if (bound(u) is better than best)
           enqueue(Q, u);
     }
  }
}
```

The Breadth-First Search with Branch-and-Bound Pruning Algorithm for the 0-1 Knapsack problem

Problem: Let n items be given, where each item has a weight and a profit. The weights and profits are positive integers. Furthermore, let a positive integer W be given. Determine a set of items with maximum total profits, under the constraint that the sum of their weights cannot exceed W.

Inputs: positive integers n and W, arrays of positive integers w and p, each indexed from 1 to n, and each of which is sorted in nonincreasing order according to the values of $p[i]/w[i]$

Outputs: an integer $maxprofit$ that is the sum of the profits in an optimal set

```
struct node
{
  int level;                // the node's level in the tree
  int profit;
  int weight;
};
```

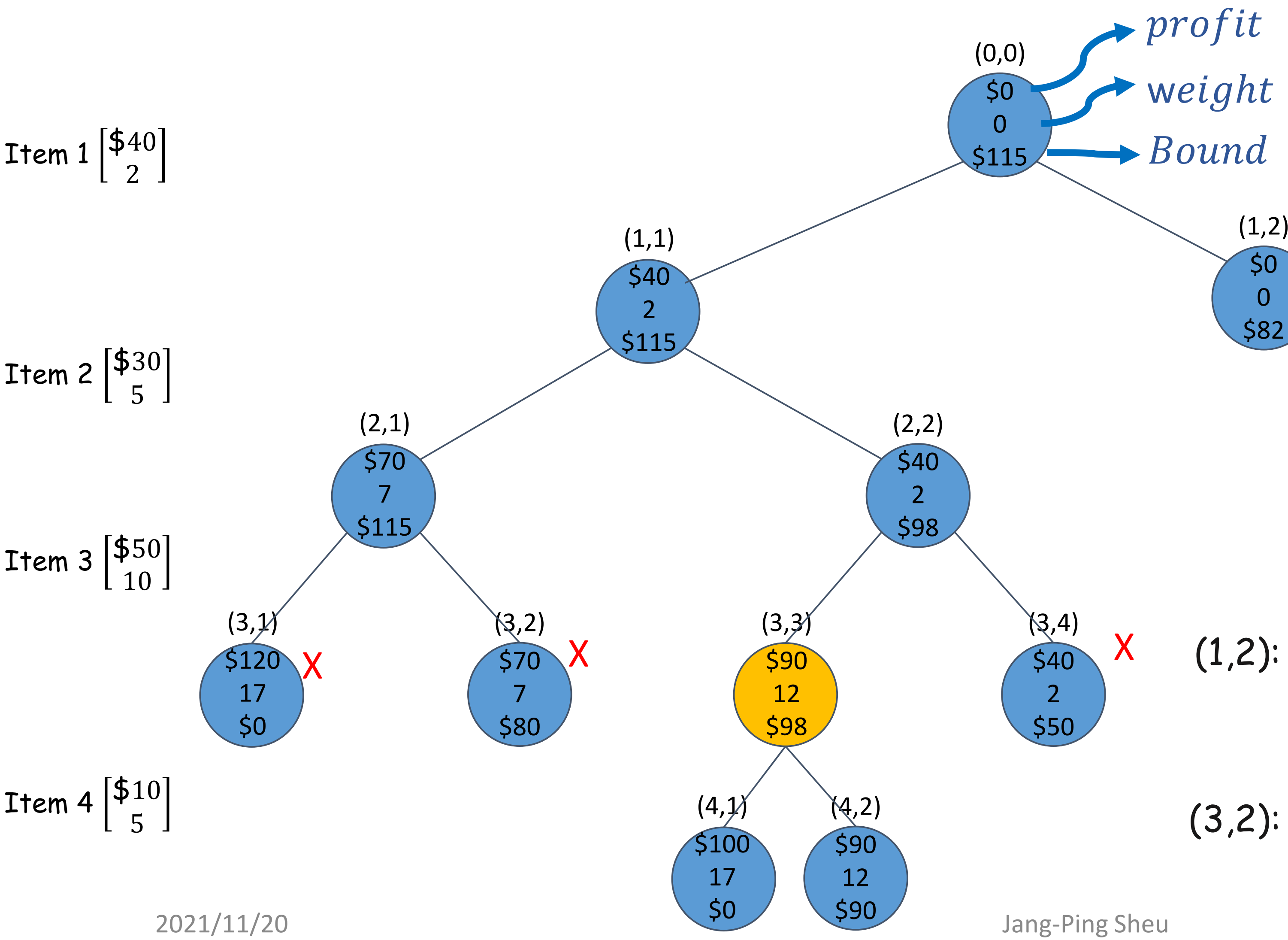
A specific algorithm for the 0-1 knapsack problem(1/2)

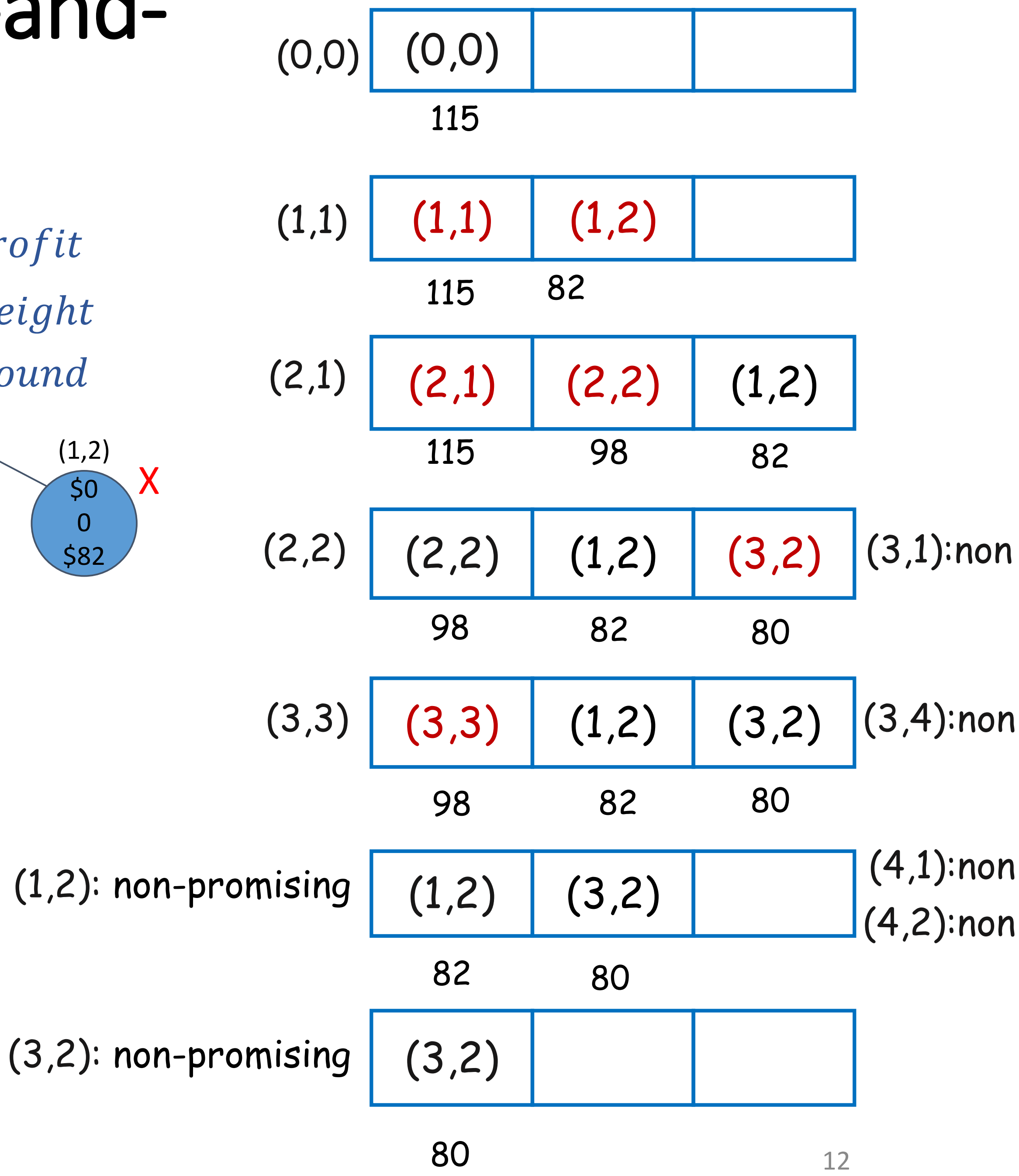
```
void knapsack2 (int n,
                const int p[], const int w[],
                int W,
                int& maxprofit)
{
  queue_of_node Q;
   node u, v;

   initialize(Q);                              // Initialize Q to be empty.
   v.level = 0; v.profit = 0; v.weight = 0;
                                               // Initialize v to be the root.
   maxprofit = 0;
   enqueue(Q, v);
   while (! empty(Q)){
      dequeue(Q, v);
      u.level = v.level + 1;                   // Set u to a child of v.
      u.weight = v.weight + w[u.level];        // Set u to the child
      u.profit = v.profit + p[u.level];        // that includes the
                                               // next item.
      if (u.weight <= W && u.profit > maxprofit)
         maxprofit = u.profit;
      if (bound(u) > maxprofit)
         enqueue(Q, u);
      u.weight = v.weight;                     // Set u to the child that
      u.profit = v.profit;                     // does not include the
      if (bound(u) > maxprofit)                // next item.
         enqueue(Q, u);
   }
}
```

A specific algorithm for the 0-1 knapsack problem(2/2)

```
float bound (node u)
{
   index j, k;
   int totweight;
   float result;
   if (u.weight >= W)
      return 0;
   else{
      result = u.profit;
      j = u.level + 1;
      totweight = u.weight;
      while (j <= n && totweight + w[j] <= W){
         totweight = totweight + w[j];          // Grab as many items
          result = result + p[j];               // as possible.
         j++;
      }
      k = j;                                    // Use k for consistency
      if (k <=n)                                // with formula in text.
         result = result + (W- totweight) * p[k]/w[k];
                                                // Grab fraction of kth
      return result;                            // item.
   }
}
```

Best-First Search with Branch-and-Bound





General algorithm

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

  initialize(PQ);                          // Initialize PQ to be empty.
  v = root of T;
  best = value(v);
  insert(PQ, v);
  while (! empty(PQ)){                     // Remove node with best
    remove(PQ, v);                         // bound.
    if (bound(v) is better than best)      // Check if node is still
      for (each child u of v){             // promising.
        if (value(u) is better than best)
          (best = value(u);
        if (bound(u) is better than best)
          insert(PQ, u);
      }
  }
}
```

The Best-First Search with Branch-and-Bound Pruning Algorithm for the 0-1 Knapsack problem

```
struct node
{
   int level;          // the node's level in the tree
   int profit;
   int weight;
   float bound;
};

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

```
    while (!empty(PQ)){                              // Remove node with
        remove(PQ, v);                               // best bound.
        if (v.bound > maxprofit){                    // Check if node is still
            u.level = v.level + 1;                   // promising.
            u.weight = v.weight + w[u.level];        // Set u to the child
            u.profit = v.profit + p[u.level];        // that includes the
                                                     // next item.
            if (u.weight <= W && u.profit > maxprofit)
                maxprofit = u.profit;
            u.bound = bound(u);
            if (u.bound > maxprofit)
                insert(PQ, u);
            u.weight = v.weight;                     // Set u to the child
            u.profit = v.profit;                     // that does not include
            u.bound = bound(u);                      // the next item.
            if (u.bound > maxprofit)
                insert(PQ, u);
        }
    }
}
```

```
float bound (node u)
{
   index j, k;
   int totweight;
   float result;
   if (u.weight >= W)
      return 0;
   else{
      result = u.profit;
      j = u.level + 1;
      totweight = u.weight;
      while (j <= n && totweight + w[j] <= W){
         totweight = totweight + w[j];             // Grab as many items
          result = result + p[j];                  // as possible.
         j++;
      }
      k = j;                                       // Use k for consistency
      if (k <=n)                                   // with formula in text.
         result = result + (W- totweight) * p[k]/w[k];
                                                   // Grab fraction of kth
      return result;                               // item.
   }
}
```

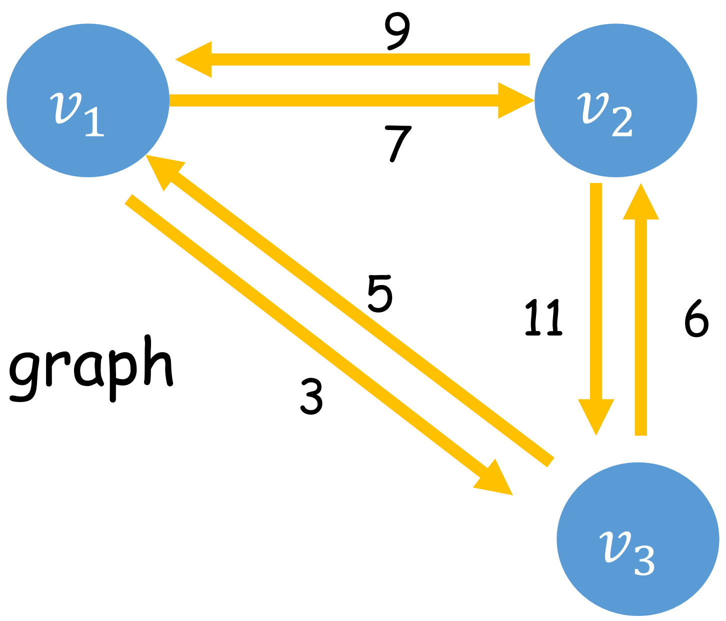
The Travelling Salesman Problem (TSP)

- The TSP asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?"

- *An adjacency matrix representation of a graph*

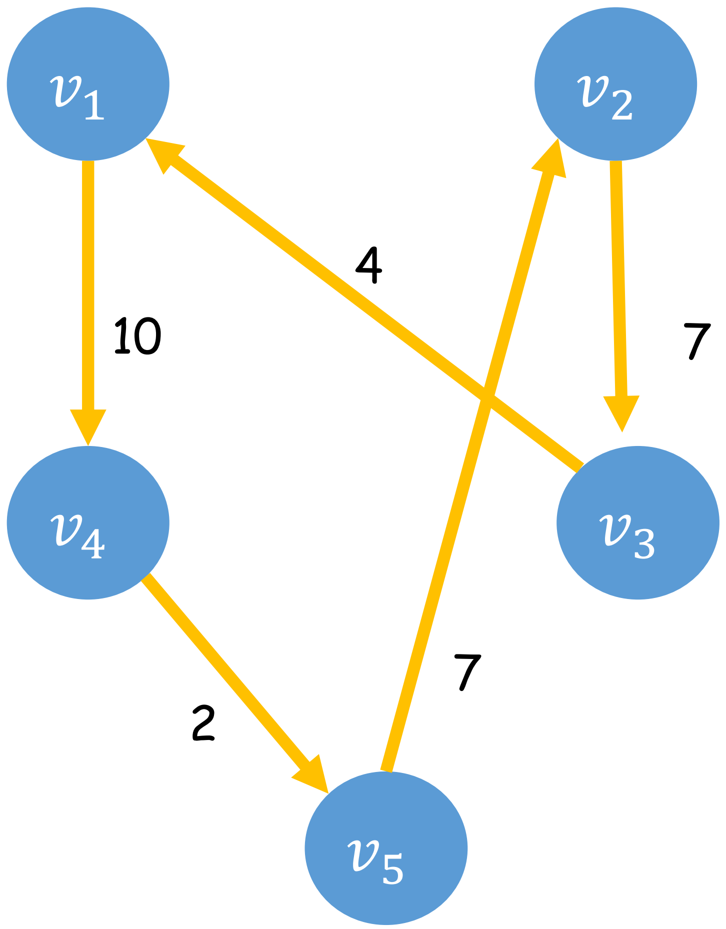
$$
\begin{array}{c c}
 & \begin{array}{ccc} v1 & v2 & v3 \end{array} \\
\begin{array}{c} v1 \\ v2 \\ v3 \end{array} & \begin{bmatrix} 0 & 7 & 3 \\ 9 & 0 & 11 \\ 5 & 6 & 0 \end{bmatrix}
\end{array}
$$

- *A weighted directed graph*



Example:

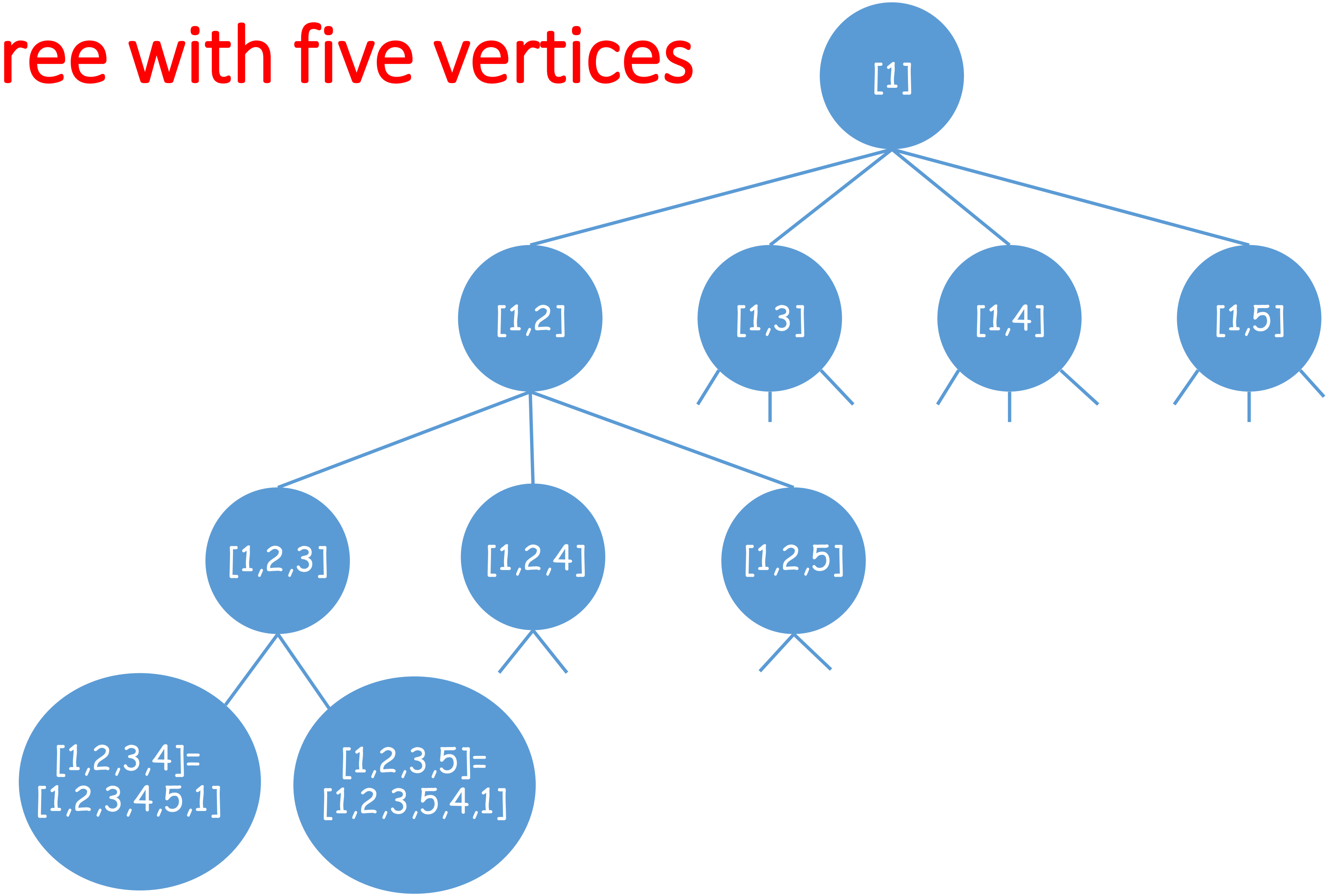
$$\begin{bmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{bmatrix}$$



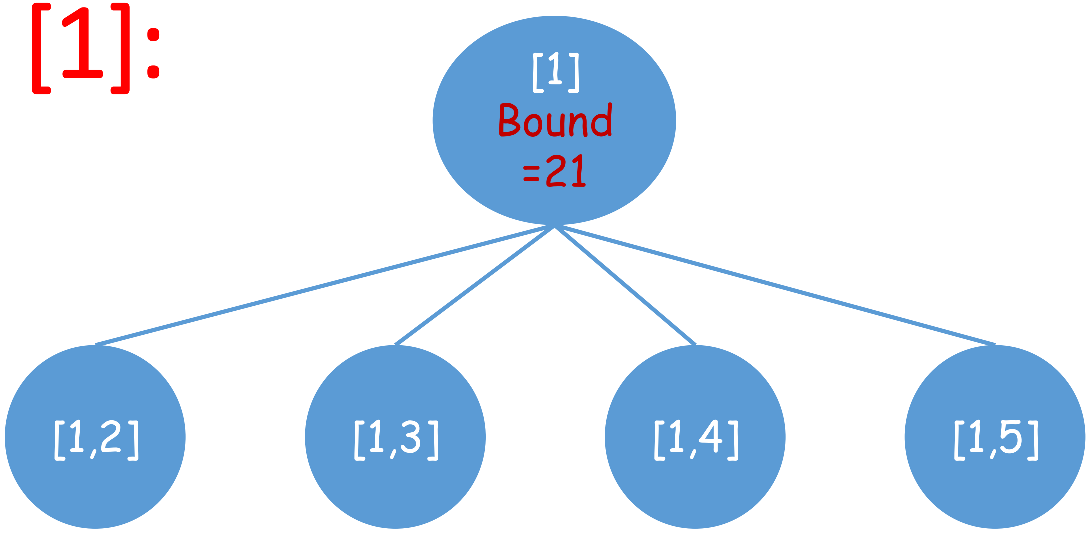
- An adjacency matrix representation of a graph

- An optimal tour: [1, 4, 5, 2, 3, 1]

A state space tree with five vertices



Computed the lower bound [1]:



$v_1 \quad minimum(14, 4, 10, 20) = 4$

$v_2 \quad minimum(14, 7, 8, 7) = 7$

$v_3 \quad minimum(4, 5, 7, 16) = 4$

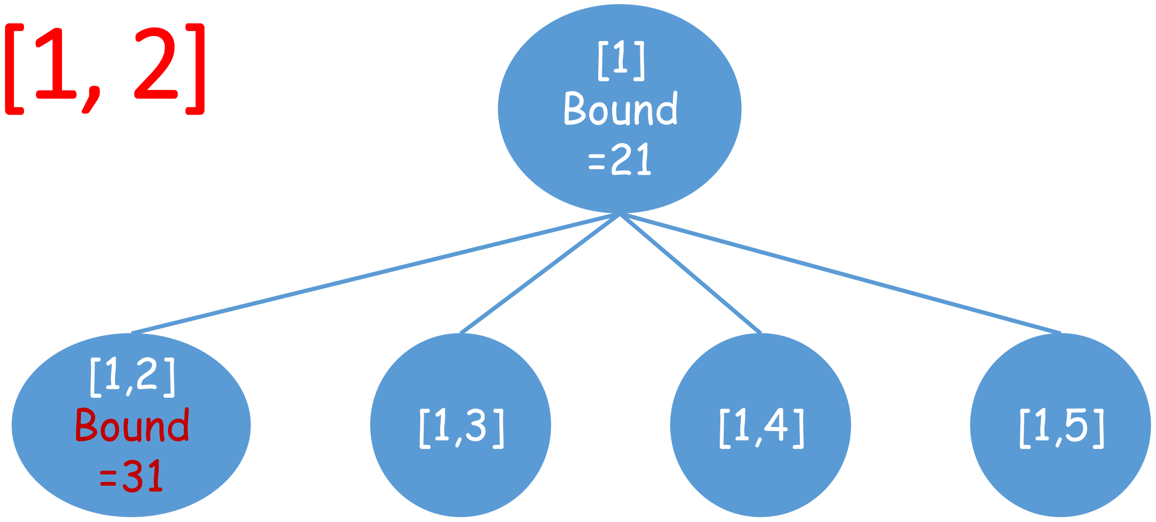
$v_4 \quad minimum(11, 7, 9, 2) = 2$

$v_5 \quad minimum(18, 7, 17, 4) = 4$

$Lower\ bound = 4 + 7 + 4 + 2 + 4 = 21$

	v_1	v_2	v_3	v_4	v_5
v_1	0	14	4	10	20
v_2	14	0	7	8	7
v_3	4	5	0	7	16
v_4	11	7	9	0	2
v_5	18	7	17	4	0

Computed the lower bound [1, 2]



v_1 $\qquad\qquad\qquad\qquad\qquad\qquad 14$

$v_2 \qquad minimum(7, 8, 7) \qquad = 7$

$v_3 \qquad minimum(4, 7, 16) \qquad = 4$

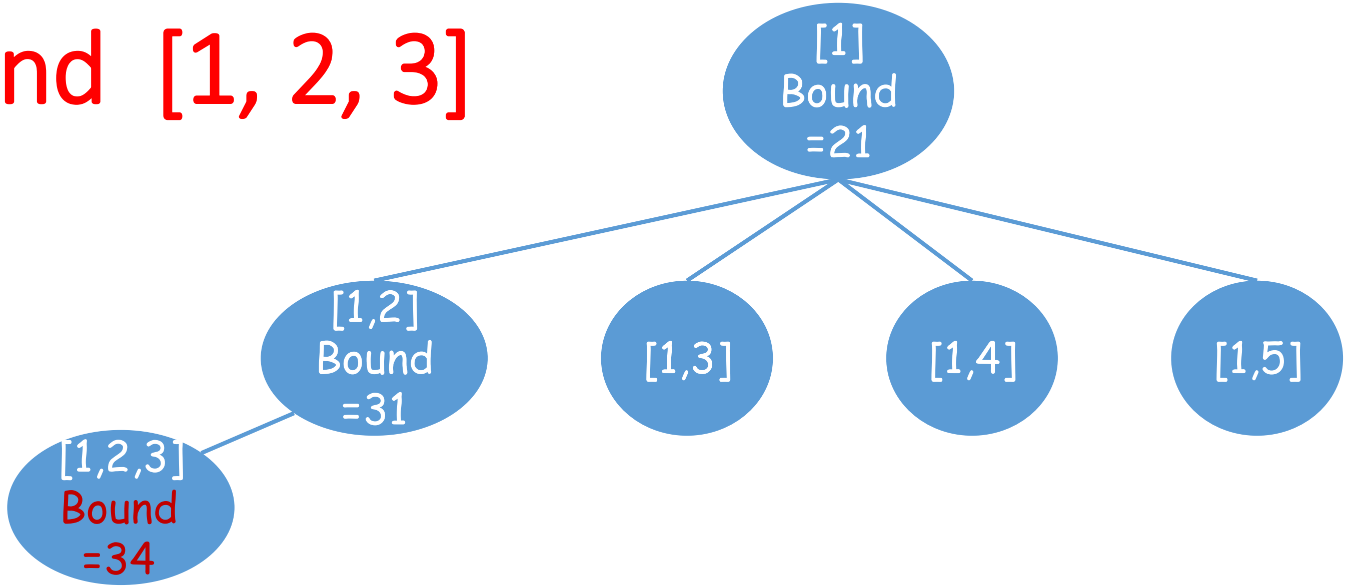
$v_4 \qquad minimum(11, 9, 2) \qquad = 2$

$v_5 \qquad minimum(18, 17, 4) \qquad = 4$

$Lower\ bound\ = 14 + 7 + 4 + 2 + 4 = 31$

	v_1	v_2	v_3	v_4	v_5
v_1	0	14	4	10	20
v_2		0	7	8	7
v_3	4	5	0	7	16
v_4	11	7	9	0	2
v_5	18	7	17	4	0

Computed the lower bound [1, 2, 3]



$v_1 \qquad 14$

$v_2 \qquad 7$

$v_3 \qquad minimum(7, 16) = 7$

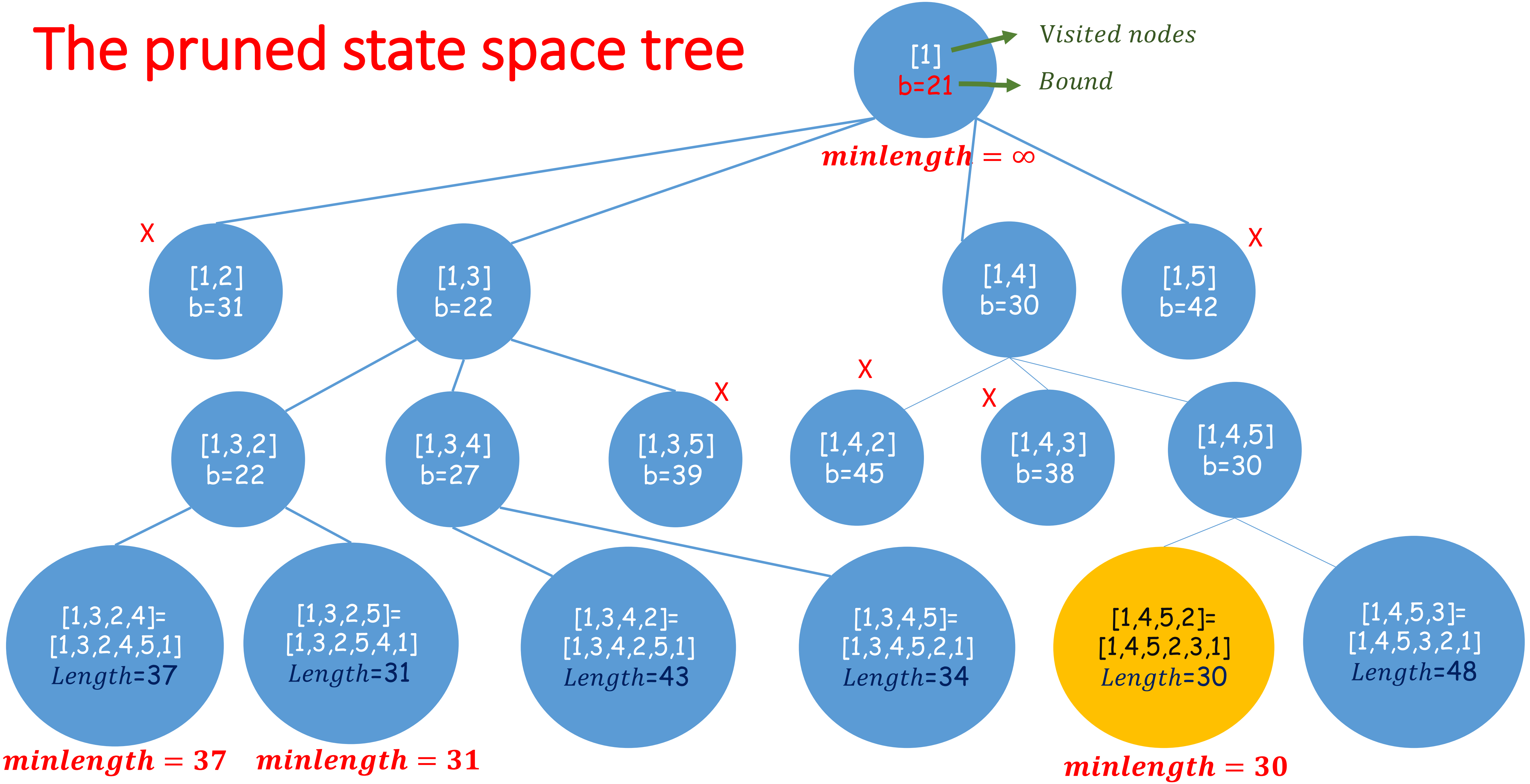
$v_4 \qquad minimum(11, 2) = 2$

$v_5 \qquad minimum(18, 4) = 4$

$Lower\ bound = 14 + 7 + 7 + 2 + 4 = 34$

	v_1	v_2	v_3	v_4	v_5
v_1	0	14	[illegible]	10	20
v_2		0	7	8	7
v_3		5	0	7	16
v_4	11	7	9	0	2
v_5	18	7	17	4	0

The pruned state space tree



- A node is nonpromising if this bound is greater than or equal to $\boldsymbol{minlength}$

The Best-First Search with Branch-and-Bound Pruning Algorithm for the TSP

Problem: Determine an optimal tour in a weighted, directed graph. The weights are nonnegative numbers.

Inputs: a weighted, directed graph, and n, the number of vertices in the graph. The graph is represented by a two-dimensional array W, which has both its rows and columns indexed from 1 to n, where $W[i][j]$ is the weight on the edge from the ith vertex to the jth vertex.

Outputs: variable $minlength$, whose value is the length of an optimal tour, and variable $opttour$, whose value is an optimal tour.

The Best-First Search with Branch-and-Bound Pruning Algorithm for the the Traveling Salesman Problem

```
struct node
{
  int level;                  // the node's level in the tree
  ordered_set path;
  number bound;
};

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

```
initialize(PQ);                                  // Initialize PQ to be empty.
v.level = 0;
v.path = [1];                                    // Make first vertex the
v.bound = bound(v);                              // starting one.
minlength = ∞;
insert(PQ, v);
while (! empty(PQ)){
  remove(PQ, v);                                 // Remove node with best bound.
  if (v.bound < minlength){
    u.level = v.level + 1;                       // Set u to a child of v.
    for (all i such that 2 ≤ i ≤ n && i is not in v.path){
      u.path = v.path;
      put i at the end of u.path;
      if (u.level == n − 2){                     // Check if next vertex
        put index of only vertex                 // completes a tour.
        not in u.path at the end of u.path;
        put 1 at the end of u.path;              // Make first vertex last one.
        if (length(u) < minlength){              // Function length computes the
            minlength = length(u);               // length of the tour.
            opttour = u.path;
        }
      }
      else{
      u.bound = bound(u);
      if (u.bound < minlength)
          insert(PQ, u);
     }
    }
  }
}
```

Homework

- Write a program to solve the Travelling Salesman Problem (TSP) problem by using branch-and-bound strategy.
- Can the branch-and-bound design strategy be used to solve the matrix-chain multiplication problem? Please explain your answer.