CSH2G3: Design & Analysis of Algorithm

Branch and Bound

Rimba Whidiana Ciptasari

The idea behind the BnB

- Like backtracking,
  - It constructs state space tree whose solution set is exponential in size
  - The complexity, in worst case, exponential
  - Significantly decrease execution time by use of “promising” function

The idea behind the BnB

- The difference between BnB and Backtracking,
  - It is a method to solve combinatorial optimization problems
  - It is necessary to provide a bound on the best value of the objective function.
  - The way it traverses the state space tree → BFS (*Breadth First Search*)

Minimization problem: lower bound

Maximization problem: upper bound

Two strategies in traversing tree

- Breadth-first branch and bound
- Best-first branch and bound

General scheme of Breadth First

```
optimumType BranchAndBoundBreadthFirst (StateSpaceTree T){
    optimumType temp, optimum;
    queue<nodeType> Q;
    nodeType u, v;

    v = T.root(); // operation root() returns T's root//
    Q.enqueue(v);
    optimum = value(v);
    while(Q.length() is not zero){
        v = Q.dequeue();
        for (each child u of v){
            temp = value(u);
            if (temp is better than optimum){
              optimum = temp;
            }
            if (bound(u) better than optimum){
                Q.enqueue(u);
            }
        }
    }
return optimum;
}
```

General scheme of Best First

```
optimumType BranchAndBoundBestFirst (StateSpaceTree T){
      optimumType temp, optimum;
      priorityQueue<nodeType> PQ;
      nodeType u, v;

      v = T.root(); // operation root() returns T's root//
      PQ.AddInorder(v);
      optimum = value(v);
      while(PQ.length() is not zero){
         v = PQ.dequeue();
         if (bound(v) is better than optimum){
            for (each child u of v){
                 temp = value(u);
                 if (temp is better than optimum){
                    optimum = temp;
                 }
                 if (bound(u) better than optimum){
                    PQ.AddInorder(u);
                 }
            }
         }
      }
return optimum;
}
```

Case 1. Assignment Problem

Let us consider *n* people who need to be assigned *n* jobs, one person per job (each person is assigned exactly one job and each job is assigned to exactly one person).

Suppose that the cost of assigning job *j* to person *i* is *C(i,j)*. Find an assignment with a minimal total cost.

Mathematical description.

Find $(s_1,\ldots,s_n)$ with s_i in {1,…n} denoting the job assigned to person i such that:

- s_i <> s_k for all i<>k (different persons have to execute different jobs)
- $C(1,s_1)+C(2,s_2)+\ldots.+C(n,s_n)$ is minimal

Assignment Problem

Generate only the assignments which have the chance to be optimal.

Estimate a bound of the cost of solutions

Assignment Problem

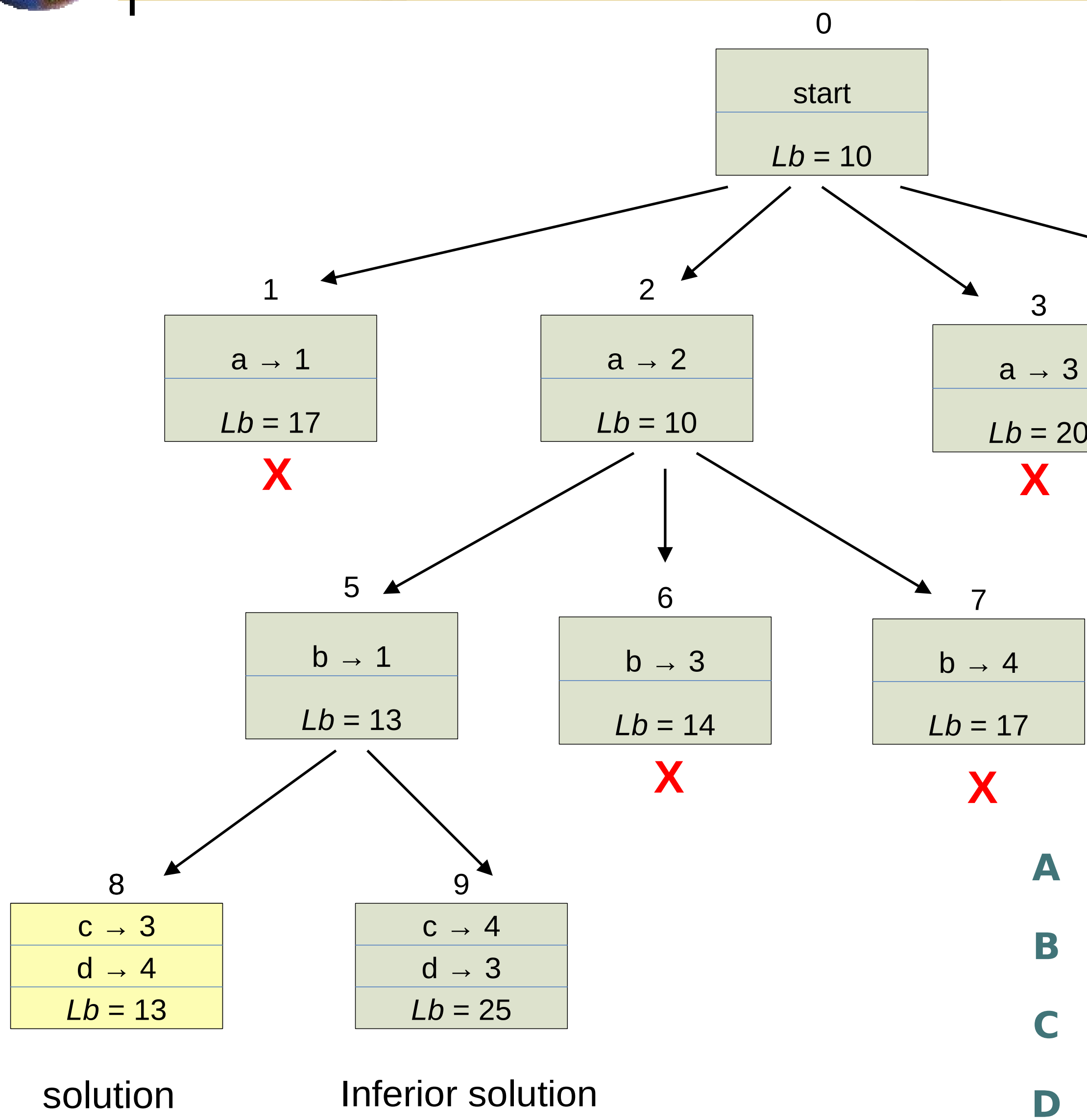
	Job 1	Job 2	Job 3	Job 4
A	9	**2**	7	8
B	6	4	**3**	7
C	5	8	**1**	8
D	7	6	9	**4**

It is just a of any legitimate selection

$$\text{lower bound } (lb) = 2+3+1+4 = 10$$

The cost function could be calculated using two approaches:

1. For each worker, we choose job with minimum cost from list of unassigned jobs (**take minimum entry from each row**) → adopted in this lecture.
2. For each job, we choose a worker with lowest cost for that job from list of unassigned workers (**take minimum entry from each column**)



	Job 1	Job 2	Job 3	Job 4
A	9	2	7	8
B	6	4	3	7
C	5	8	1	8
D	7	6	9	4

Example. 0/1 Knapsack

- Order the *value-to-weight ratios* in descending order

  $v_1/w_1 \geq v_2/w_2 \geq \ldots \geq v_n/w_n$

- Bound:

  $ub = v+(W-w)(v_{i+1}/w_{i+1})$

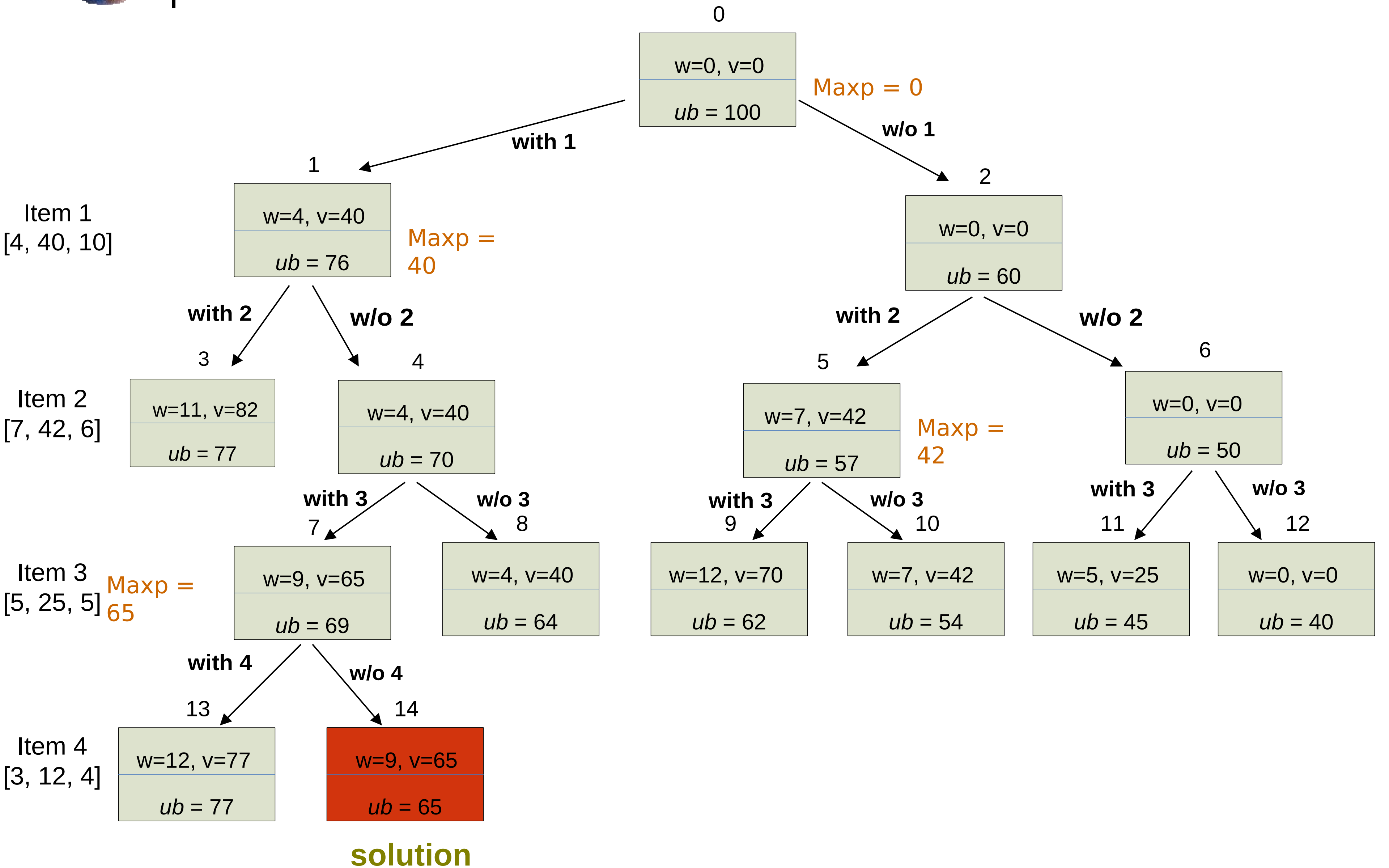
0/1 Knapsack

- Consider the following instance of the 0/1 Knapsack

  $n=4, W=10$

i	v_i	w_i	v_i/w_i
1	404	10	
2	427	6	
3	255	5	
4	123	4	

Breadth-First_0/1Knapsack



0
w=0, v=0
ub = 100
Maxp = 0
with 1
w/o 1
1
w=4, v=40
ub = 76
Maxp = 40
2
w=0, v=0
ub = 60
Item 1
[4, 40, 10]
with 2
w/o 2
with 2
w/o 2
3
w=11, v=82
ub = 77
4
w=4, v=40
ub = 70
5
w=7, v=42
ub = 57
Maxp = 42
6
w=0, v=0
ub = 50
Item 2
[7, 42, 6]
with 3
w/o 3
with 3
w/o 3
with 3
w/o 3
7
w=9, v=65
ub = 69
8
w=4, v=40
ub = 64
9
w=12, v=70
ub = 62
10
w=7, v=42
ub = 54
11
w=5, v=25
ub = 45
12
w=0, v=0
ub = 40
Item 3
[5, 25, 5]
Maxp = 65
with 4
w/o 4
13
w=12, v=77
ub = 77
14
w=9, v=65
ub = 65
Item 4
[3, 12, 4]
solution

Breadth-first_0/1 Knapsack algorithm

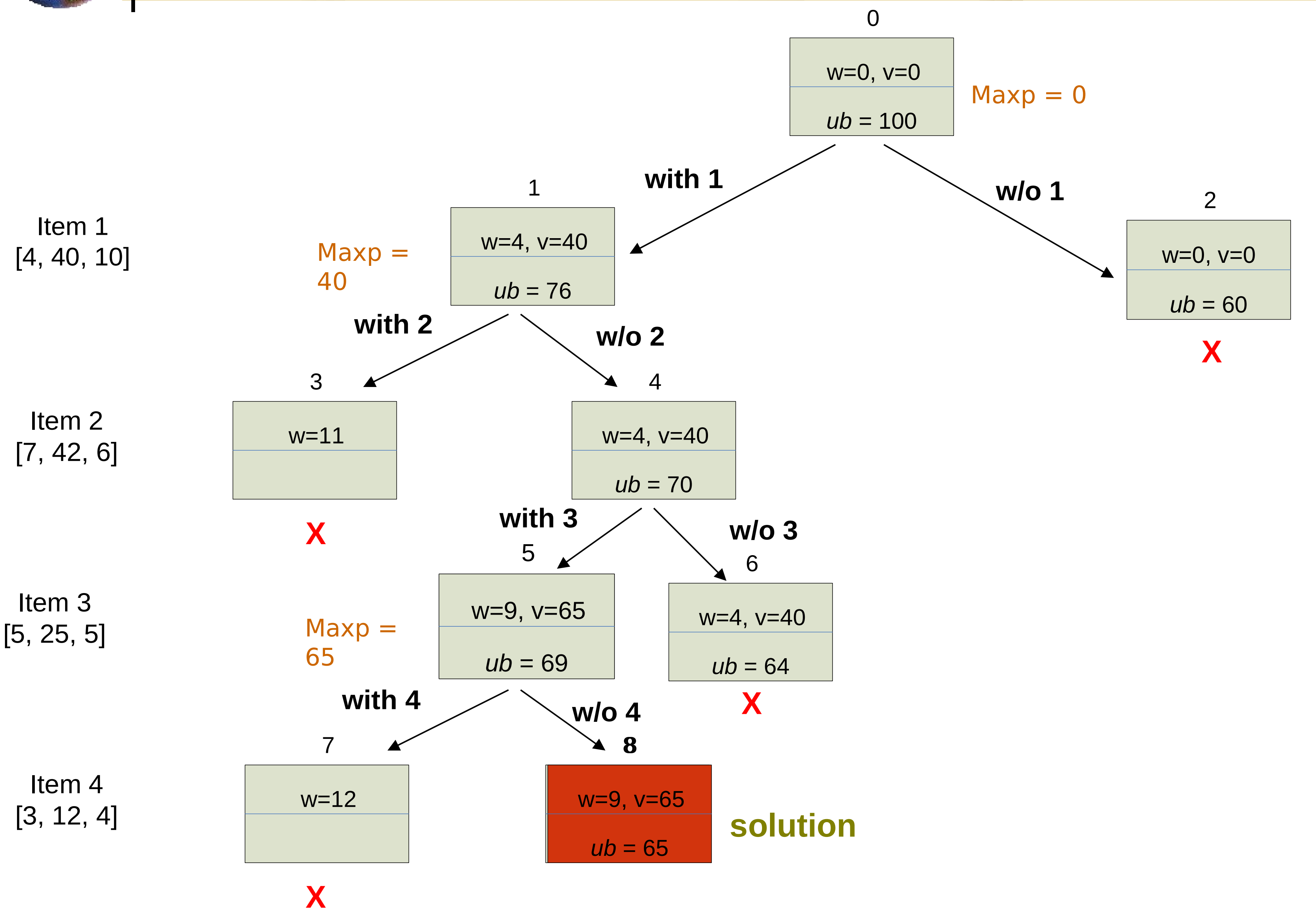
```
Procedure knapsack(n: integer; p,w: array[1..n] of integer; W: integer; var maxprofit: integer)
Var Q: queue of node; u,v: node;
Begin
   Initialized(Q);
   v.level:=0; v.weight:=0; v.profit:=0;
   maxprofit:=0;
   v.bound:=bound(v);
   insert(Q,v);
   while not empty(Q) do
      remove(Q,v);
      u.level:=v.level+1;
      u.weight:=v.weight+w[u.level];
      u.profit:=v.profit + p[u.level];
      if u.weight ≤ W and u.profit > maxprofit then
          maxprofit:=u.profit;
      end;
      if u.weight ≤ W and bound(u) > maxprofit then
         insert(Q,u);
      end;
      u.weight:=v.weight; u.profit:=v.profit;
      if u.weight ≤ W and bound(u) > maxprofit then
         insert(Q,u);
      end;
end;
End;
```

$n=4$, $W=10$

i	v_i	w_i	v_i/w_i
1	40	4	10
2	42	7	6
3	25	5	5
4	12	3	4

```
Function bound(u: node): real;
j,k : index; totweight: integer
Algorithm
If u.weight>=W then
        bound = 0
Else
        bound = u.profit
        j = u.level +1
        totweight = u.weight;
        k=j
        promising = true
        While j<=n and promising do
                if (totweight + w[j]) <= W then
                        totweight = totweight + w[j]
                        bound = bound + p[j]
                        k=j
                        j = j+1
                else
                        promising = false
        {end while}
        If k <= n then
                bound = bound + (W-totweight)*p[k]/w[k]
```

Best-First with Branch-and-Bound Pruning 0/1_Knapsack


0
w=0, v=0
ub = 100
Maxp = 0
with 1
w/o 1
1
w=4, v=40
ub = 76
Maxp = 40
2
w=0, v=0
ub = 60
X
Item 1
[4, 40, 10]
with 2
w/o 2
3
w=11
X
4
w=4, v=40
ub = 70
Item 2
[7, 42, 6]
with 3
w/o 3
5
w=9, v=65
ub = 69
Maxp = 65
6
w=4, v=40
ub = 64
X
Item 3
[5, 25, 5]
with 4
w/o 4
7
w=12
X
8
w=9, v=65
ub = 65
solution
Item 4
[3, 12, 4]

Best-First_0/1 Knapsack Algorithm

```
Procedure knapsack(n: integer; p,w: array[1..n] of integer; W: integer; var maxprofit: integer)
Var PQ: priority_queue of node; u,v: node;
Begin
  Initialized(PQ);
  v.level:=0; v.weight:=0; v.profit:=0;
  maxprofit:=0;
  v.bound:=bound(v);
  insert(PQ,v);
  while not empty(PQ) do
    remove(PQ,v);
    if v.bound > maxprofit then
          u.level:=v.level+1;
          u.weight:=v.weight+w[u.level];
          u.profit:=v.profit + p[u.level];
          if u.weight ≤ W and u.profit > maxprofit then
              maxprofit:=u.profit;
          end;
          if u.weight ≤ W and bound(u) > maxprofit then
            insert(PQ,u);
          end;
          u.weight:=v.weight; u.profit:=v.profit;
          if u.weight ≤ W and bound(u) > maxprofit then
            insert(PQ,u);
          end;
    end;
  end;
End;
```

Exercise

- Solve the following instance of the 0/1 knapsack problem by constructing the state-space-tree.

  $n=4, W=16$

  i v_i w_i v_i/w_i

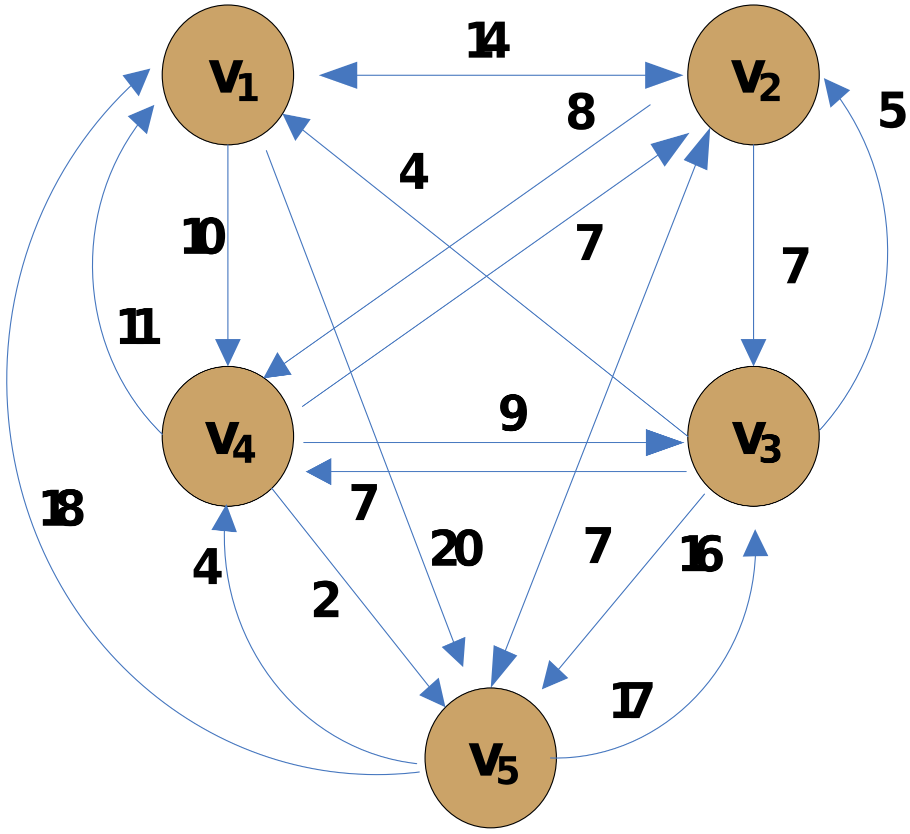
  1 402 20

  2 305 6

  3 50105

  4 105 2

Example. Traveling Salesman Problem



Adjacency matrix

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

Traveling Salesman Problem

- Lower bound on the cost of leaving vertex v_1 is given by the minimum of all nonzero entries in row 1 of the adjacency matrix,
- Lower bound on the cost of leaving vertex v_2 is given by the minimum of all nonzero entries in row 2 of the adjacency matrix,
- And so on..

Traveling Salesman Problem

- Lower bound on the cost of leaving the five vertices are:

  v_1 *minimum* (14, 4, 10, 20) = 4

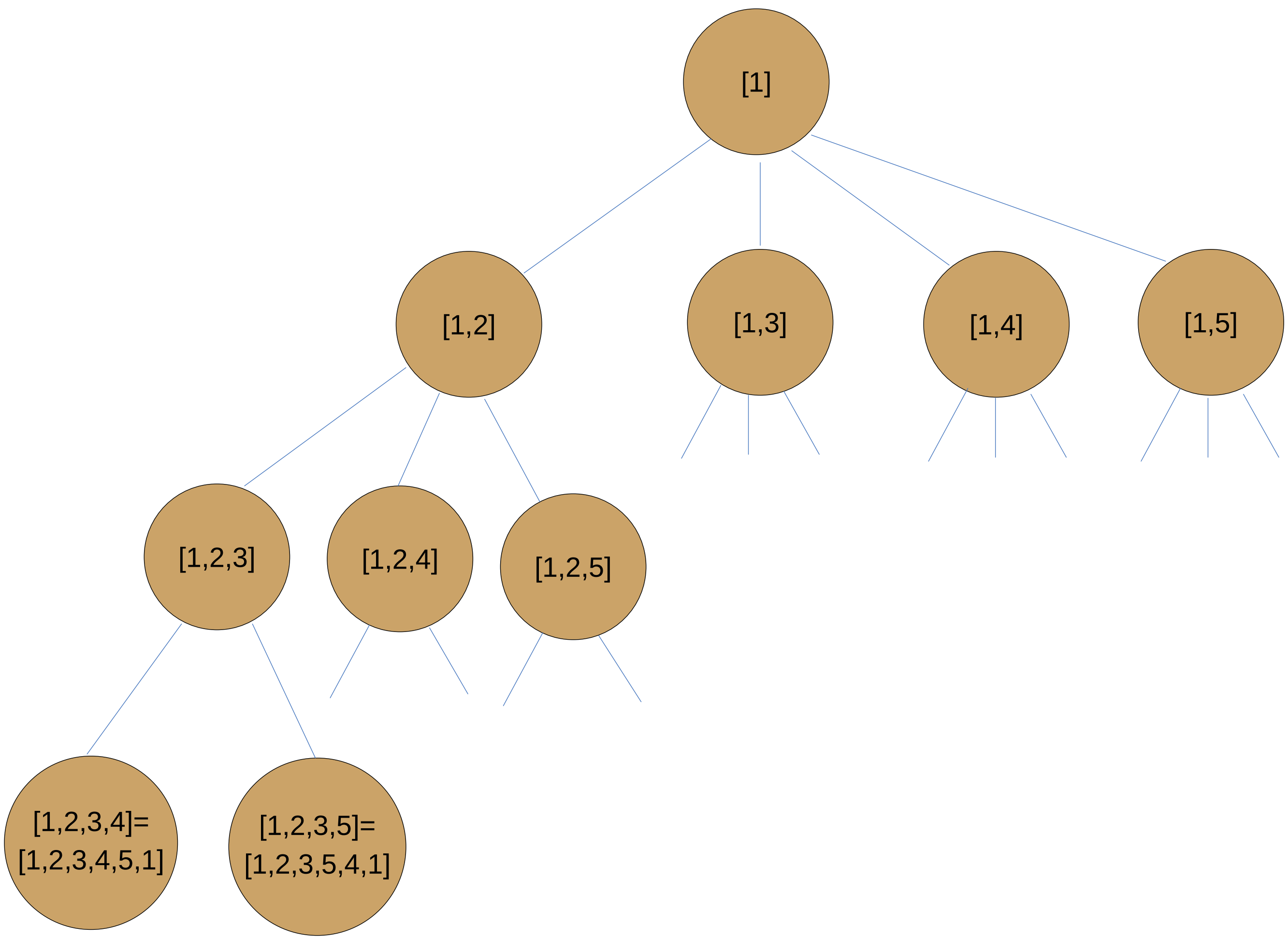
  v_2 *minimum* (14, 7, 8, 7) = 7

  v_3 *minimum* (4, 5, 7, 16) = 4

  v_4 *minimum* (11, 7, 9, 2) = 2

  v_5 *minimum* (18, 7, 17, 4) = 4
- The sum of these minimums is 21

Lower bound



Lower bound

- Lower bound on the node containing [1,2] :
  - The cost of getting to v_2 is 14
  - Obtain the minimum for v_2, it doesn't include the edge to v_1
  - Obtain the minimums for the other vertices it doesn't include v_2 because it's already been at v_2.

  v_1 = 14
  v_2 minimum(7, 8, 7) = 7
  v_3 minimum(4, 7, 16) = 4
  v_4 minimum(11, 9, 2) = 2
  v_5 minimum(18, 17, 4) = 4
- Lower bound obtained by expanding beyond the node containing [1,2] is 14+7+4+2+4=31

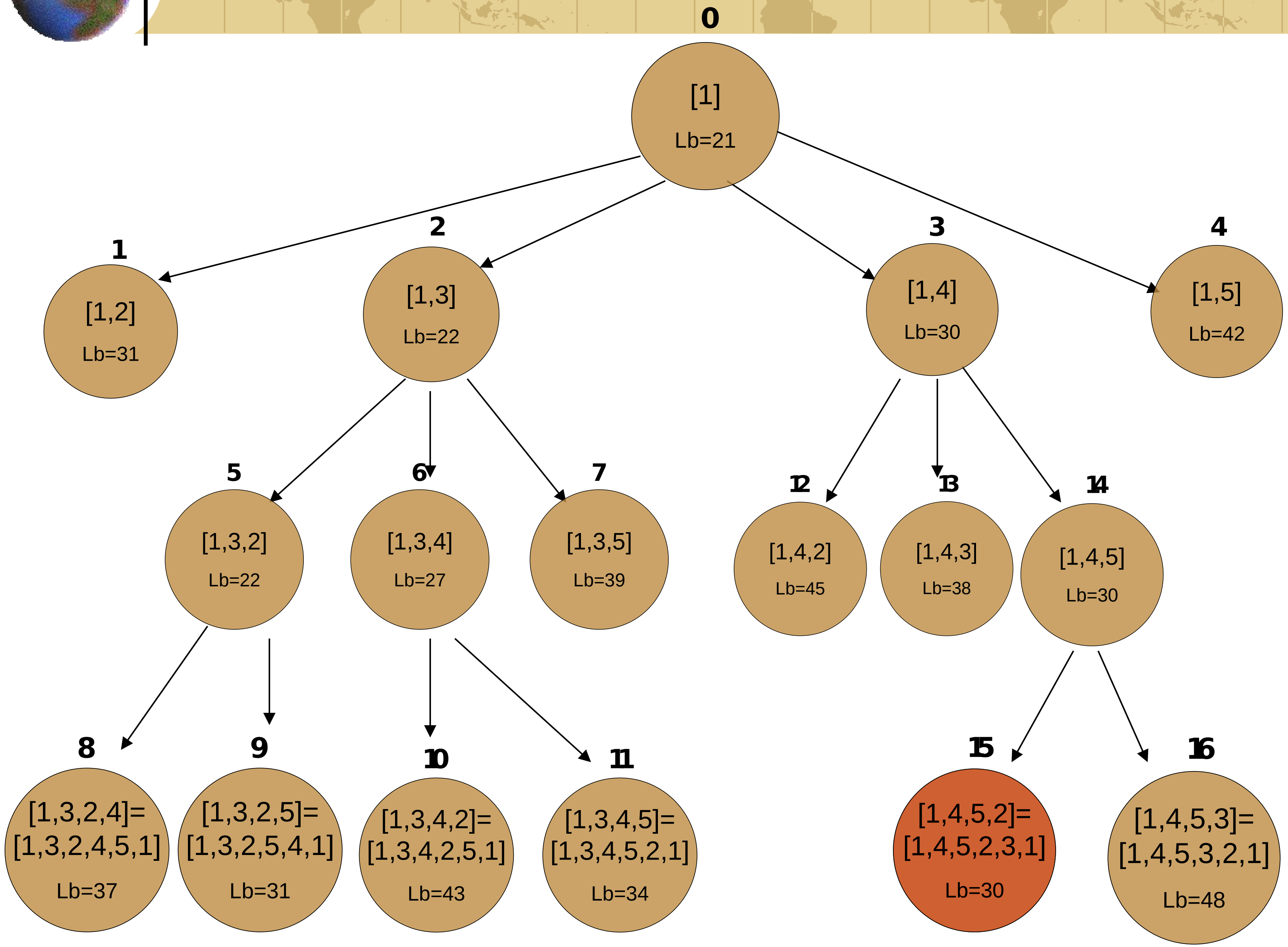
Lower bound

- Lower bound on the node containing [1,2,3]. Any tour obtained by expanding beyond this node has the following lower bound on the cost of leaving the vertices:

v_1		=	14
v_2		=	7
v_3	minimum(7, 16)	=	7
v_4	minimum(11, 2)	=	2
v_5	minimum(18, 4)	=	4

- The lower bound on the node [1,2,3] is 14+7+7+2+4=34

Best-first search with branch-and-bound pruning


0
[1]
Lb=21
1
[1,2]
Lb=31
2
[1,3]
Lb=22
3
[1,4]
Lb=30
4
[1,5]
Lb=42
5
[1,3,2]
Lb=22
6
[1,3,4]
Lb=27
7
[1,3,5]
Lb=39
12
[1,4,2]
Lb=45
13
[1,4,3]
Lb=38
14
[1,4,5]
Lb=30
8
[1,3,2,4]=
[1,3,2,4,5,1]
Lb=37
9
[1,3,2,5]=
[1,3,2,5,4,1]
Lb=31
10
[1,3,4,2]=
[1,3,4,2,5,1]
Lb=43
11
[1,3,4,5]=
[1,3,4,5,2,1]
Lb=34
15
[1,4,5,2]=
[1,4,5,2,3,1]
Lb=30
16
[1,4,5,3]=
[1,4,5,3,2,1]
Lb=48

Problem: determine an optimal tour in a weighted graph. The weights are nonnegative numbers

Inputs: a weighted, directed graph *W*, and *n* the number of vertices in *W*. *W* is represented by matrix, where *W[i,j]* is the weight on the edge from v_i to v_j

Outputs: variable *minlength*, whose value is the length of an optimal tour, and variable *optour*, whose value is an optimal tour

```
Procedure travel(n:int; W:array[1..n,1..n]of number; var
   optour:ordered_set;var minlength:number)
Var
   u,v:node;PQ:priority_queue_of_node
Begin
   initialize(PQ);
   v.level=0;v.path=[1];v.bound=bound(v);minlength=∞;
   insert(PQ,v);
   while not empty(PQ)do
       remove(PQ,v);
       if v.bound < minlength then
           u.level=v.level+1;
           if u.level=n-1 then
               u.path=v.path;
               add 1 to the end of u.path;
               if length(u) < minlength then
                   minlength=length(u);
                   optour=u.path
           else
             for i such that 2≤i≤n and i is not in v.path do
               u.path=v.path;
               add i to the end of u.path;
               u.bound=bound(v);
               if u.bound<minlength then
                   insert(PQ,u)
```

```
Type node = record
  level:integer;
  path :ordered_set;
  bound:number
End;
```

TSP: an optimal tour