Backtracking

Sum of Subsets

and

Knapsack

Backtracking

- Two versions of backtracking algorithms
  - Solution only needs to be feasible (satisfy problem's constraints)
    - sum of subsets
  - Solution needs also to be optimal
    - knapsack

The backtracking method

- A given **problem** has a set of constraints and possibly an objective function
- The **solution** must be feasible and it may optimize an objective function
- We can represent the **solution space** for the problem using a **state space tree**
  - The *root* of the tree represents 0 choice,
  - Nodes at depth 1 represent first choice
  - Nodes at depth 2 represent the second choice, etc.
  - In this tree a *path* from a root to a leaf represents a candidate solution

Sum of subsets

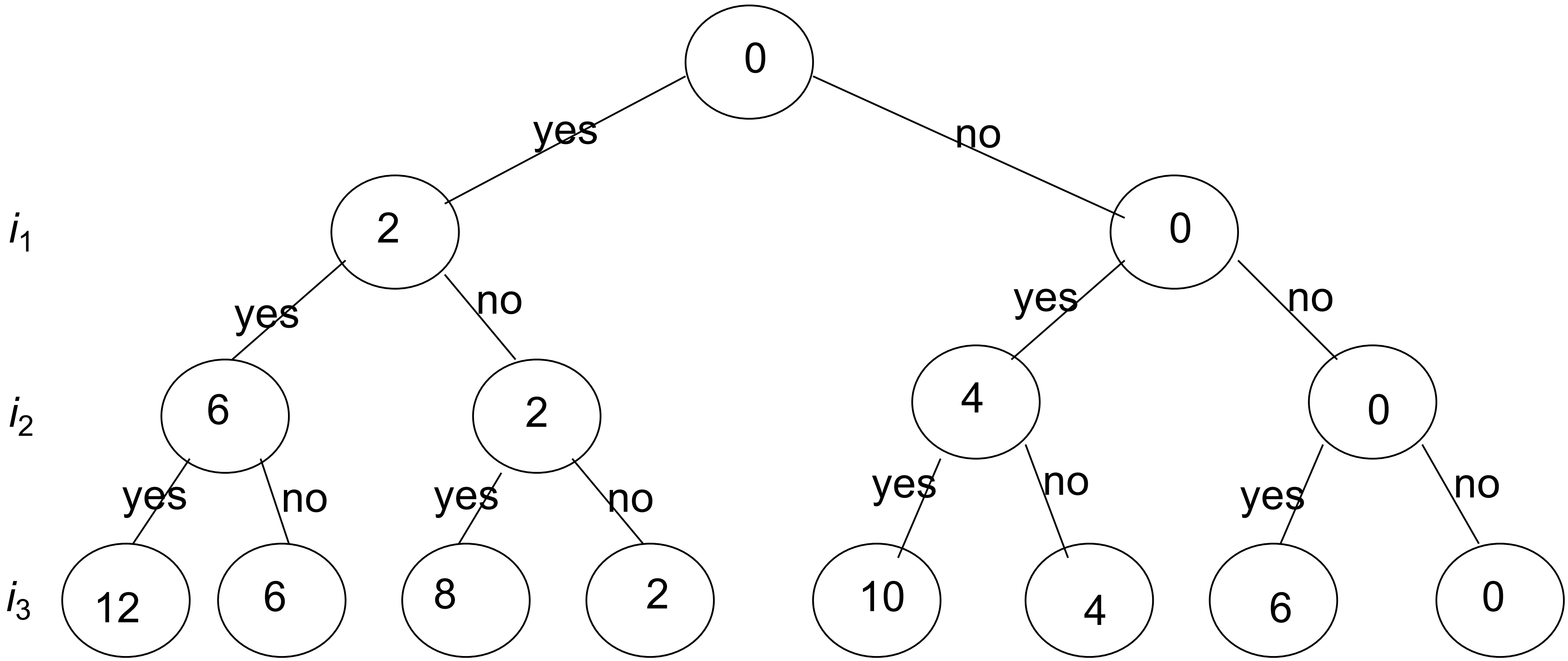
- **Problem**: Given *n* positive integers $w_{1,} \dots w_n$ and a positive integer S. Find all subsets of $w_{1,} \dots w_n$ that sum to S.
- **Example**:
  n=3, S=6, and w_1=2, w_2=4, w_3=6
- **Solutions**:
  {2,4} and {6}

Sum of subsets

- **We will assume a binary state space tree.**
- **The nodes at depth 1 are for including (yes, no) item 1, the nodes at depth 2 are for item 2, etc.**
- **The left branch includes w_i, and the right branch excludes w_i**
- **The nodes contain the sum of the weights included so far**

Sum of subset Problem:

State SpaceTree for 3 items: $w_1 = 2$, $w_2 = 4$, $w_3 = 6$ and $S = 6$



The sum of the included integers is stored at each node.

A Depth First Search solution

- Problems can be solved using depth first search of the (implicit) state space tree
- Each node will save its depth and its (possibly partial) current solution
- DFS can check whether node v is a leaf
  - If it is a leaf then check if the current solution satisfies the constraints
  - Code can be added to find the optimal solution

A DFS solution

- **Such a DFS algorithm will be very slow.**
- **It does not check for every solution state (node) whether a solution has been reached already**
- **Neither does it check whether or not a *partial* solution can lead to a *feasible* solution**
- **Is there a more efficient solution?**

Backtracking

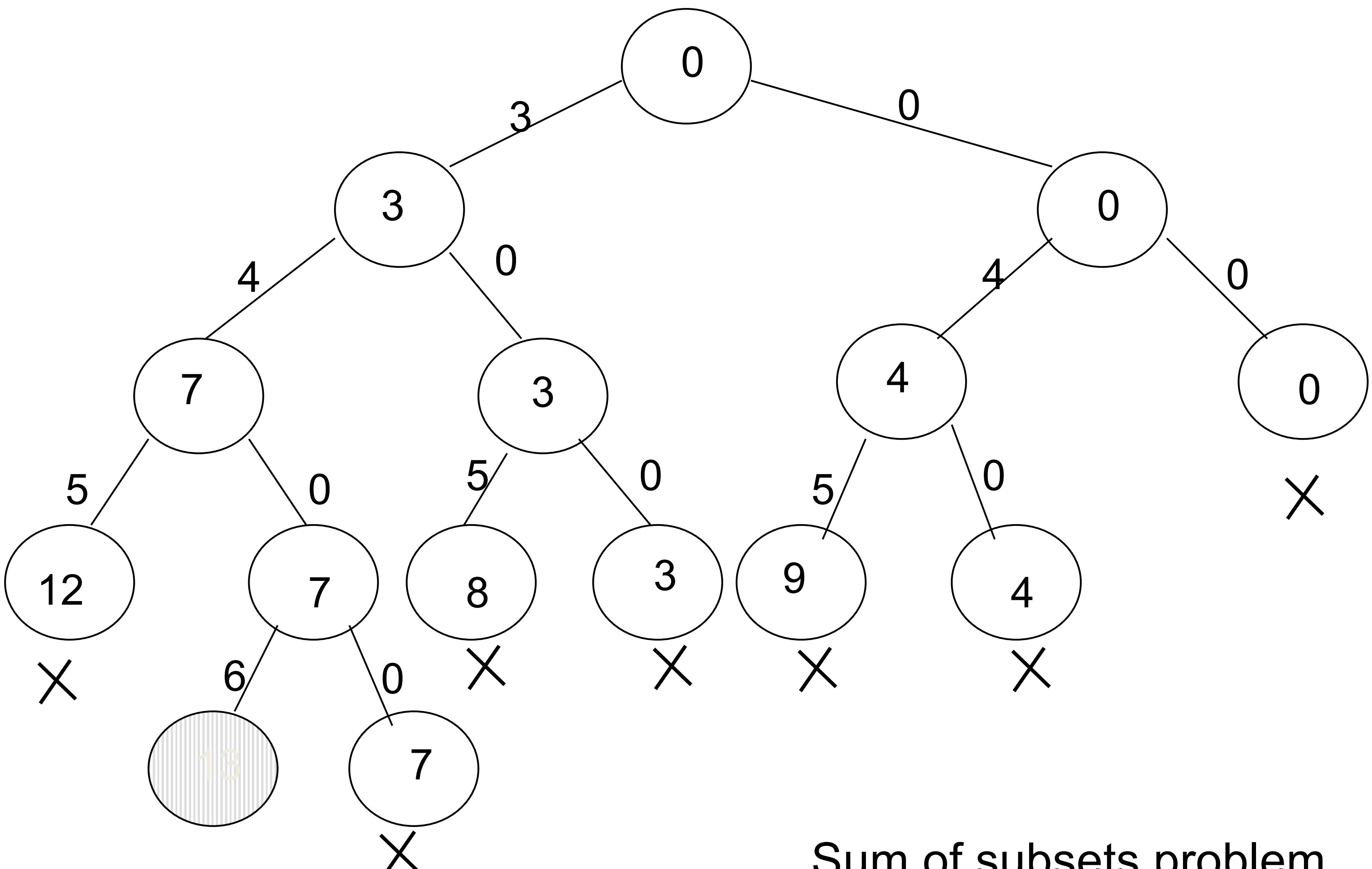
- **Definition**: We call a node *nonpromising* if it cannot lead to a feasible (or optimal) solution, otherwise it is *promising*
- **Main idea**: Backtracking consists of doing a DFS of the state space tree, checking whether each node is promising. If the node is nonpromising, backtrack to its parent.

Backtracking

- The state space tree consists of expanded nodes only (called the *pruned state space tree*)
- The following slide shows the pruned state space tree for the sum of subsets example
- There are only 15 nodes in the pruned state space tree
- The full state space tree has 31 nodes

A Pruned State Space Tree (find all solutions)

$w_1 = 3,\ w_2 = 4,\ w_3 = 5,\ w_4 = 6;\quad S = 13$



Sum of subsets problem

Backtracking algorithm

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

   if (promising ( v ))
      if (aSolutionAt( v ))
         write the solution
      else //expand the node
         for ( each child u of v )
            checknode ( u )
}
```

Checknode

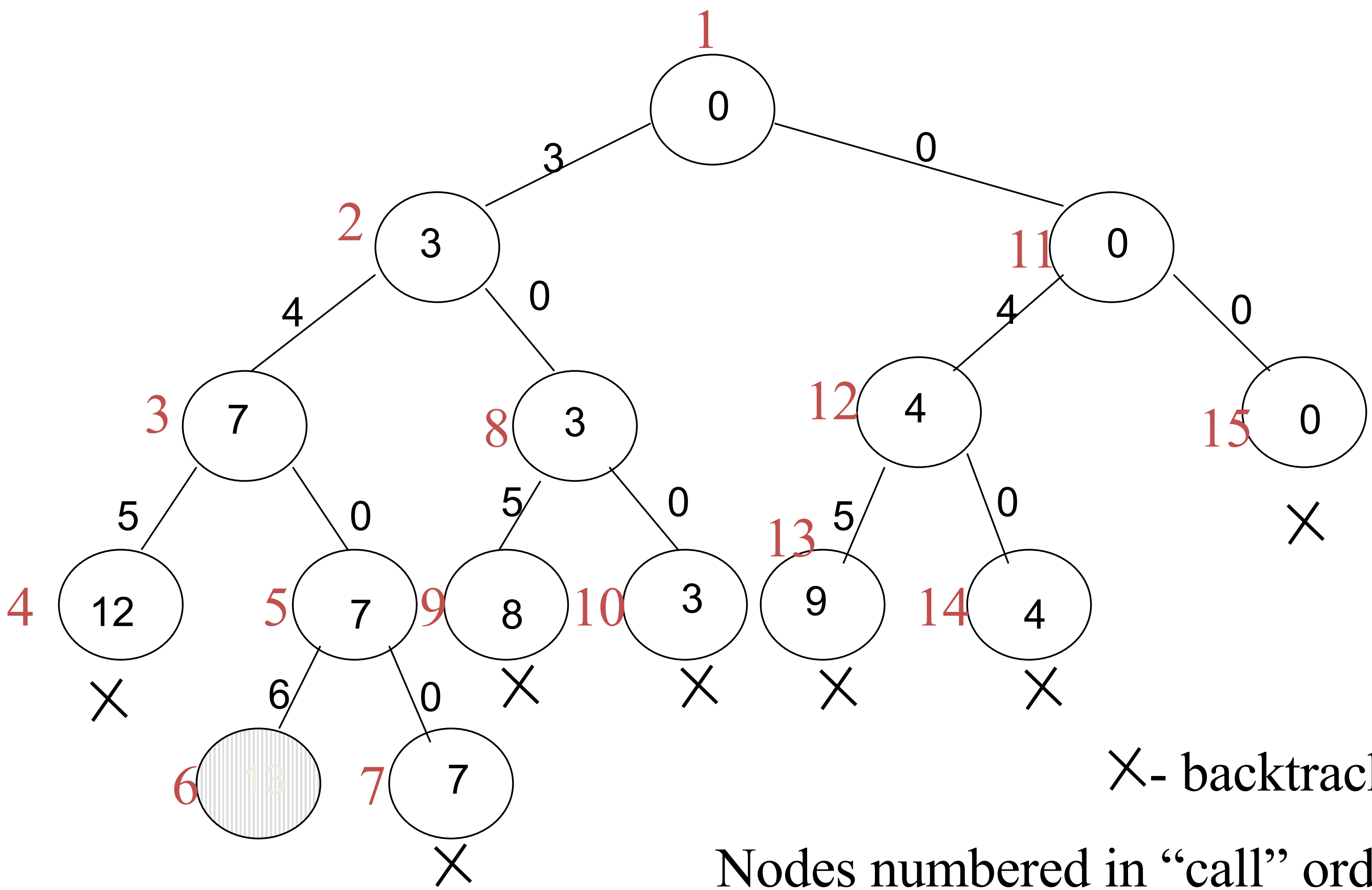
- Checknode uses the functions:
  - *promising*(*v*) which checks that the partial solution represented by *v* can lead to the required solution
  - *aSolutionAt*(*v*) which checks whether the partial solution represented by node *v* solves the problem.

Sum of subsets – when is a node "promising"?

- Consider a node at depth i
- *weightSoFar* = weight of a node, i.e., sum of numbers included in the partial solution that the current node represents
- *totalPossibleLeft* = weight of the remaining items i+1 to n (for a node at depth i)
- A node at depth i is non-promising
  if (*weightSoFar* + *totalPossibleLeft* < S)
  or (*weightSoFar* + w[i+1] > S)
- To be able to use this "promising function" the w_i must be sorted in <u>*non-descending*</u> order

A Pruned State Space Tree

$w_1 = 3,\ w_2 = 4,\ w_3 = 5,\ w_4 = 6;\quad S = 13$



X- backtrack

Nodes numbered in “call” order

```
sumOfSubsets ( i, weightSoFar, totalPossibleLeft )
  1) if (promising ( i ))                         //may lead to solution
  2)    then if ( weightSoFar == S )
  3)         then print include[ 1 ] to include[ i ]     //found solution
                  return
  4)    else  //expand the node when weightSoFar < S
  5)         include [ i + 1 ] = "yes"                   //try including
  6)         sumOfSubsets ( i + 1, weightSoFar + w[i + 1],
                  totalPossibleLeft - w[i + 1] )
  7)         include [ i + 1 ] = "no"                    //try excluding
  8)         sumOfSubsets ( i + 1, weightSoFar , totalPossibleLeft –
             w[i + 1] )
  9) return // nonpromising
```

Initial call: sumOfSubsets(0, 0, $\sum_{i=1}^{n} w_i$)

```
boolean promising (i )
  1) return ( weightSoFar + totalPossibleLeft ≥ S)  and
        ( weightSoFar == S  or weightSoFar +  w[i + 1] ≤ S )
```

Prints all solutions!

Backtracking for optimization problems

- To deal with optimization we compute:
  - *best* : value of the best solution achieved so far
  - *value*(*v*): value of the solution at node *v*
  - Modify *promising*(*v*)
- *Best* is initialized to a value that is equal to a candidate solution or worse than any possible solution
- *Best* is updated to *value*(*v*) if the solution at *v* is "better"
- By "better" we mean:
  - larger in the case of maximization and
  - smaller in the case of minimization

Modifying promising

- A node is ***promising*** when
  - it is **feasible** and **can lead to a feasible solution** and
  - "there is a chance that a **better solution than the (current)** ***best*** can be achieved by expanding it"

- Otherwise it is *nonpromising*

- A ***bound*** on the best solution that can be achieved by expanding the node is computed and compared to *best*

- If the ***bound*** > ***best*** for maximization, (< *best* for minimization) the node is promising

Modifying promising for Maximization Problems

- **For a *maximization* problem the bound is an *upper bound***
  - **The largest possible solution that can be achieved by expanding the node is smaller than or equal to the *upper bound***
- **If *upper bound > best* so far, a better solution may be found by expanding the node and the feasible node is *promising***

Modifying promising for Minimization Problems

- **For *minimization,* the bound is a *lower bound***
  - **The smallest possible solution that can be achieved by expanding the node is larger than or equal to the *lower bound***

- **If *lower bound < best,* a better solution may be found and the feasible node is *promising***

Template for backtracking in the case of optimization problems

Procedure *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* is the best value so far
- *value*(*v*) is the value of the solution at node v

0-1 Knapsack problem

- Solve 0-1 knapsack problem via backtracking
- How to compute the upper bound?
  - Use the optimal greedy algorithm for solving the fractional knapsack to compute the upper bound
  - Do you remember what the optimal fractional knapsack algorithm is?

Notation for knapsack

- We use *maxprofit* to denote *best*
- *profit*(*v*) to denote *value*(*v*)

The state space tree for knapsack

- Each *node v* will include 3 values:
  - *profit* (v) = sum of profits of all items included in the knapsack (on a path from root to *v*)
  - *weight* (v)= the sum of the weights of all items included in the knapsack (on a path from root to *v*)
  - *upperBound(v)* is the maximum benefit that can be found by expanding the whole subtree of the state space tree with root *v.*
- The nodes are numbered in the order of expansion

Promising nodes for 0/1 knapsack

- Node *v* is *promising* if *weight(v)* < *C*, and *upperBound(v)* > *maxprofit*
- Otherwise it is not promising
- Note that when weight(v) = C or upperbound(v) = maxprofit the node is non-promising

Main idea for upper bound

- **Main idea**: *KWF (knapsack with fraction)* is used to compute upper bound
- **Theorem**: The optimal profit for 0/1 knapsack $\leq$ optimal profit for *KWF*

- *Discussion:* Clearly the optimal solution to 0/1 knapsack is a possible solution to *KWF*. So the optimal profit of *KWF* is greater or equal to that of 0/1 knapsack

Computing the upper bound for 0/1 knapsack

- Given node *v* at depth *i*.
- *UpperBound(v) =*

  KWF2(i+1, weight(v), profit(v), w, p, C, n) where w and p are arrays of weights and profits
- *KWF2 requires* that the items be sorted in non-ascending p_i / w_i order. If we arrange the items in this order before applying the backtracking algorithm, *KWF2* will pick the remaining items in the required order.

KWF2(i, weight, profit, w, p, C, n)

```
bound = profit
for j = i to n
   x[j] = 0  //initialize variables to 0
while (weight < C && i <= n)        //not “full”and more items
    if weight + w[i] <= C           //room for next item
         x[i]=1                     //item i is added to knapsack
         weight = weight + w[i]; bound = bound +p[i];
   else
        x[i]=(C - weight)/w[i]  //fraction of  i added to knapsack
        weight = C; bound = bound + p[i]*x[i]
   i=i+1                    // next item
return bound
```

- KWF2 is in O(n) (assuming items sorted before applying backtracking)

Pseudo code

- **The arrays *w*, *p*, *include* and *bestset* have size n+1.**
- **Location 0 is not used**
- ***include* contains the current solution**
- ***bestset* the best solution so far**

Knapsack

```
num = 0;    //number of items considered
maxprofit = 0;
knapsack(0,0,0);
cout << maxprofit;
for (i = 1; i <= num; i++)
    cout << bestset[i]; //the best solution
```

- *maxprofit* is initialized to $0, which is the worst profit that can be achieved with positive p_i.
- In Knapsack - before determining if node *v* is promising, *maxprofit* and *bestset* are updated

knapsack(i, profit, weight)

```
if ( weight <= C && profit > maxprofit )
  // save better solution
   maxprofit = profit //save new profit
   num = i; bestset = include; //save solution
if promising(i)
  include[i + 1] = " yes"
  knapsack(i+1, profit + p[i+1], weight + w[i+1])
  include[i+1] = "no"
  knapsack(i+1,profit,weight)
```

Promising(i)

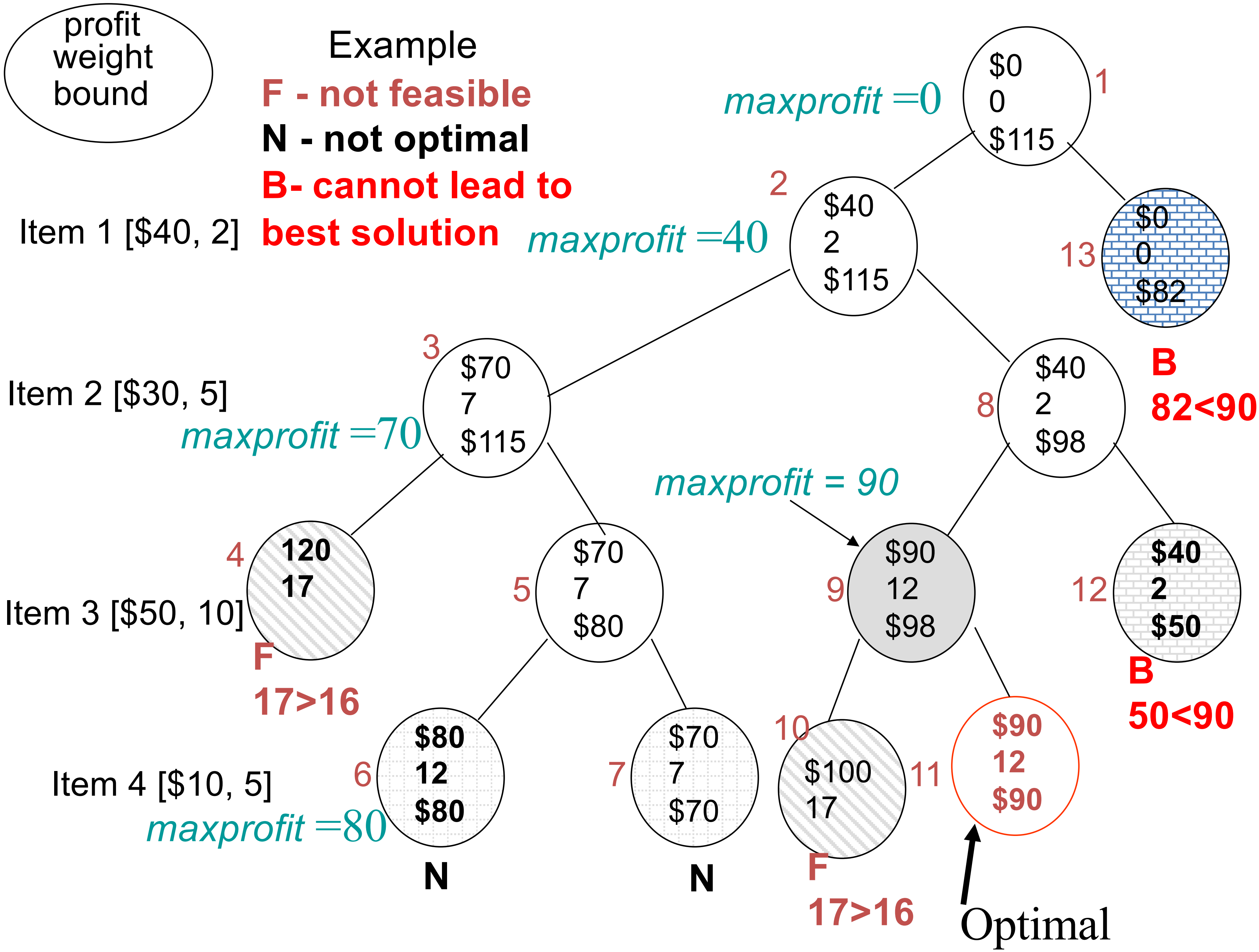
```
promising(i)
{
   //Cannot get a solution by expanding node
   if weight >= C return false

   //Compute upper bound
   bound = KWF2(i+1, weight, profit, w, p, C, n)
   return (bound > maxprofit)
}
```

Example

- Suppose n = 4, C = 16, and we have the following:

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

- Note the items should be in the *correct order* needed by *KWF (largest profit/weight first)*



profit
weight
bound
Example
F - not feasible
N - not optimal
B- cannot lead to best solution
Item 1 [$40, 2]
Item 2 [$30, 5]
Item 3 [$50, 10]
Item 4 [$10, 5]
maxprofit =0
maxprofit =40
maxprofit =70
maxprofit =80
maxprofit = 90
1
$0
0
$115
2
$40
2
$115
13
$0
0
$82
B
82<90
3
$70
7
$115
8
$40
2
$98
4
120
17
F
17>16
5
$70
7
$80
9
$90
12
$98
12
$40
2
$50
B
50<90
6
$80
12
$80
N
7
$70
7
$70
N
10
$100
17
F
17>16
11
$90
12
$90
Optimal

The calculation for node 1

maxprofit = \$0 (n = 4, C = 16)

Node 1

a) *profit* = \$ 0

weight = 0

b) $bound = profit + p_1 + p_2 + (\boldsymbol{C} - 7) * p_3 / w_3$

= \$0 + \$40 + \$30 + (16 -7) X \$50/10 =\$115

c) 1 is promising because its weight =0 < ***C*** = 16 and its bound \$115 > 0 (*maxprofit).*

The calculation for node 2

Item 1 with profit \$40 and weight 2 is included

maxprofit = \$40

a) *profit* = \$40

weight = 2

b) $bound = profit + p_2 + (\boldsymbol{C} - 7) \text{ X } p_3 / w_3$

$= \$40 + \$30 + (16 - 7) \text{ X } \$50/10 = \$115$

c) 2 is promising because its weight =2 < ***C*** = 16 and its bound \$115 > \$40 the value of *maxprofit.*

The calculation for node 13

Item 1 with profit \$40 and weight 2 is not included

At this point maxprofit=\$90 and is not changed

a) *profit* = \$0
   weight = 0

b) $bound = profit + p_2 + p_3 + (\boldsymbol{C} - 15) \text{ X } p_4 / w_4$
   $= \$0 + \$30 + \$50 + (16 - 15) \text{ X } \$10/5 = \$82$

c) 13 is nonpromising because its bound \$82 < \$90 the value of *maxprofit.*

Worst-case time complexity

Check number of nodes:

$$1+2+2^2+2^3+...+2^n = 2^{n+1}-1$$

Time complexity:

$$\theta(2^n)$$

When does it happen?

For a given n, W=n

$P_i = 1$, $w_i=1$ (for $1 \le i \le n-1$)

$P_n=n$ $w_n=n$

Comparing the dynamic programing with the backtracking algorithm

- The worst-case number of entries that is computed by the dynamic programming algorithm for the 0-1 Knapsack problem is in O $(minimum\ (2^n, nW))$.
- In the worst case, the backtracking algorithm checks $\Theta\ (2^n)$ nodes. Owing to the additional bound of nW , it may appear that the dynamic programming algorithm is superior.
- In backtracking algorithms the worst case gives little insight into how much checking is usually saved by backtracking. With so many considerations, it is difficult to analyze theoretically the relative efficiencies of the two algorithms.
- Horowitz and Sahni (1978) ran both algorithms on many sample instances and found that the back-tracking algorithm is usually more efficient than the dynamic programming algorithm.

Branch-and-Bound

Knapsack

Characteristics

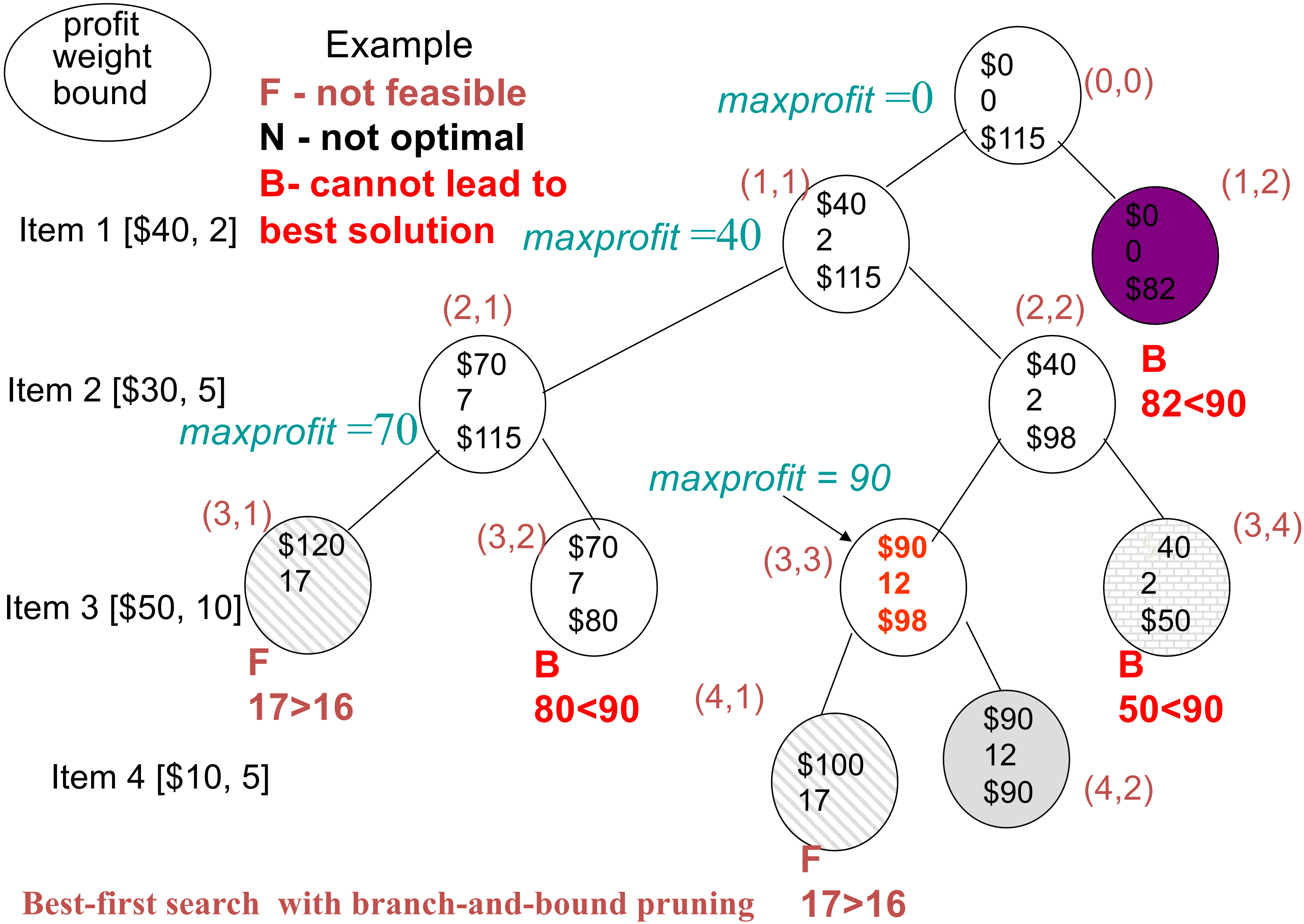
– Use strategy similar to breadth-first-search with some modification

– Visit all the children of a given node to look at all the promising, unexpanded nodes and expand beyond the one with the best bound (e.g., greatest bound)

– Exponential-time in the worst case (same as backtracking algorithm), but could be very efficient for many large instances.

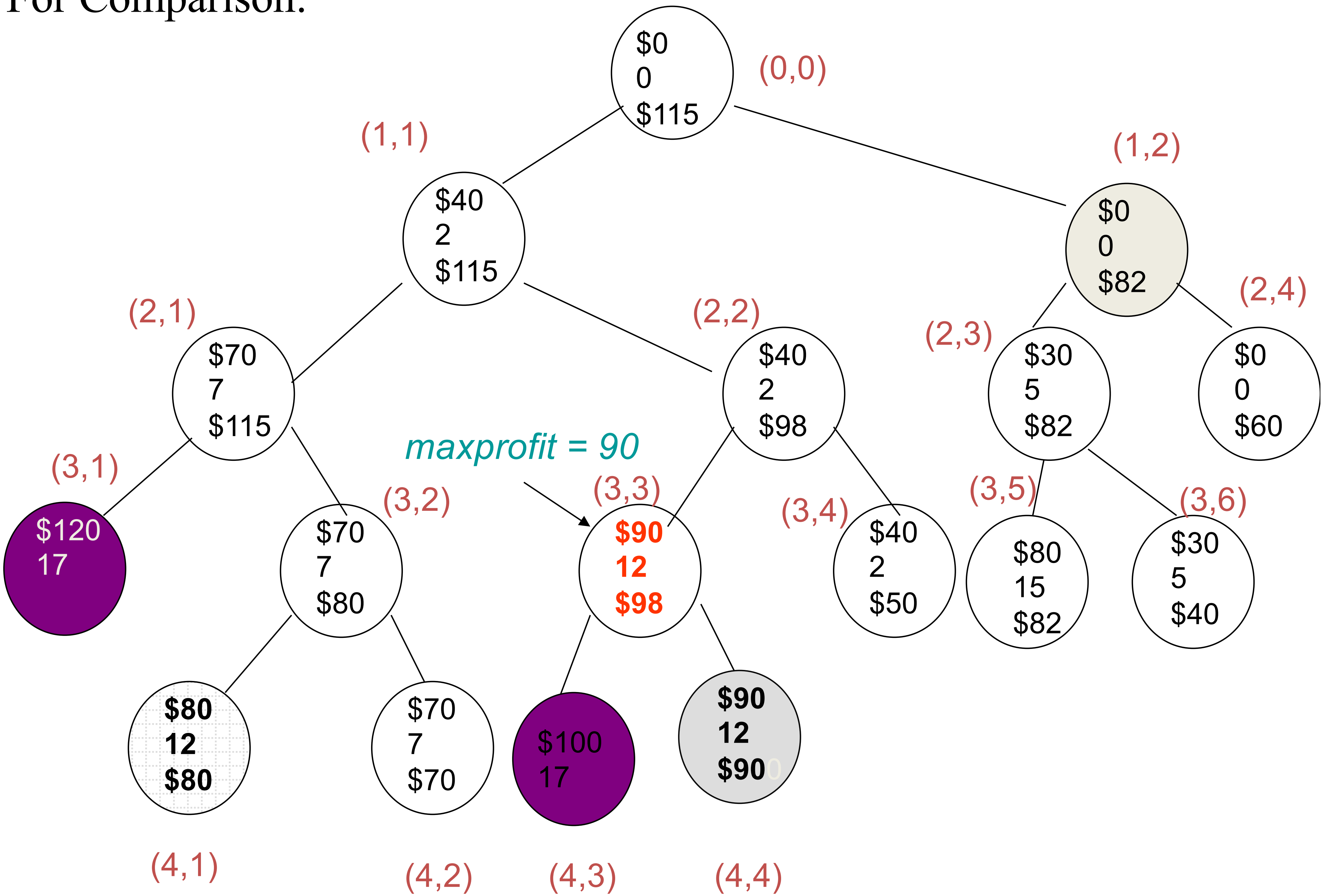
Characteristics

- In backtracking algorithm, the promising function returns false if the value of bound is not greater than the current value of maxprofit, which does not exploit the real advantage of using branch-and-bound.
- Besides using the bound to determine whether a node is promising, we can compare the bounds of promising nodes and visit the children of the one with the best bound.
- In this way we often can arrive at an optimal solution faster than we would by methodically visiting the nodes in some predetermined order (such as a depth-first search). This approach is called best-first search with branch-and-bound pruning.



Best-first search with branch-and-bound pruning
(Expand the unexplored node with the greatest bound)

For Comparison:



Breath-first search with branch-and-bound pruning