HANDONG GLOBAL UNIVERSITY

Knapsack Problem

Algorithm Analysis

School of CSEE

The Knapsack Problem

- The famous *knapsack problem*:
  - A thief breaks into a museum. Fabulous paintings, sculptures, and jewels are everywhere. The thief has a good eye for the value of these objects, and knows that each will fetch hundreds or thousands of dollars on the clandestine art collector's market. But, the thief has only brought a single knapsack to the scene of the robbery, and can take away only what he can carry. What items should the thief take to maximize the value of packed objects?

The Knapsack Problem

0/1 ≤ Fractional

There are two versions of the problem:

(1) "0-1 knapsack problem" and → Item을 선택(1)/선택X(0)

(2) "Fractional knapsack problem" → Item 분할가능.

(1) Items are indivisible: you either take an item or not.

→ 물건 개수가 적다

→ dynamic programming. , 꼬리는 동일이

(2) Items are divisible: you can take any fraction of an item.

→ 일부를 가져올 수 있는

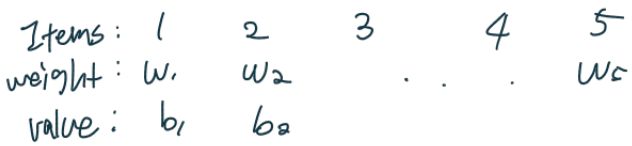
→ Greedy programming , dp 할 수 있긴해.

Fractional knapsack problem

- To solve the fractional problem, we first compute the value per pound *vi/wi for each item. Obeying a greedy strategy, the thief begins by taking* as much as possible of the item with the greatest value per pound. If the supply of that item is exhausted and he can still carry more, he takes as much as possible of the item with the next greatest value per pound, and so forth until he can't carry any more.

0-1 Knapsack Problem

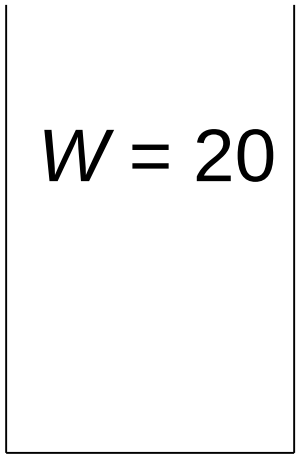
- A knapsack with maximum capacity W, and a set S consisting of n items.
- Each item i has some weight w_i and benefit value b_i.

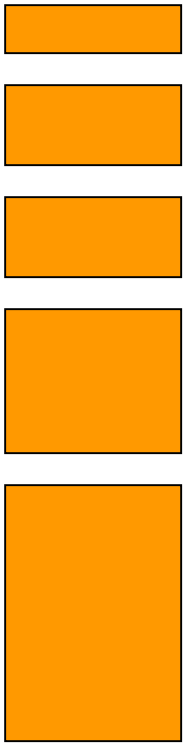


- All w_i, b_i and W are integer values.

0-1 Knapsack Problem

This is a knapsack
Max weight: $W = 20$





Items	Weight w_i	Benefit value b_i
	2	3
	3	4
	4	5
	5	8
	9	10

[1] brute force approach

Let's first solve this problem with a straightforward algorithm.

- Since there are n items, there are 2^n possible combinations of items.
- We go through all combinations and find the one with the most total value and with total weight less or equal to W.
- Running time will be $\Theta(2^n)$.

0-1 Knapsack Problem

- Can we do better?
- [2] Greedy approach does not work for this problem.

  (But it works for fractional knapsack problem.)
- Yes, with an algorithm based on [3] dynamic programming
- We need to carefully identify the subproblems

Let's try this:

If items are labeled *1..n*, then a subproblem would be to find an optimal solution for S_k = *{items labeled 1, 2, .. k}*

[3] Dynamic programming

If items are labeled *1..n*, then a subproblem would be to find an optimal solution for S_k = *{items labeled 1, 2, .. k}*

- This is a valid subproblem definition.
- Q : Can we describe the final solution (S_n) in terms of subproblems (S_k)?
- A : Unfortunately, we can't do that.

  Explanation follows….

[3] Dynamic programming

$w_1 = 2$	$w_2 = 4$	$w_3 = 5$	$w_4 = 3$	
$b_1 = 3$	$b_2 = 5$	$b_3 = 8$	$b_4 = 4$	

Max weight: W = 20

For S_4:

Total weight: 14

total value: 20

$w_1 = 2$	$w_2 = 4$	$w_3 = 5$	$w_5 = 9$
$b_1 = 3$	$b_2 = 5$	$b_3 = 8$	$b_5 = 10$

For S_5:

Total weight: 20

total value: 26

Item #	Weight w_i	Value b_i
1	2	3
2	4	5
3	5	8
4	3	4
5	9	10

S_4

S_5

Solution for S_4 is not part of the solution for S_5!!!

Defining a problem

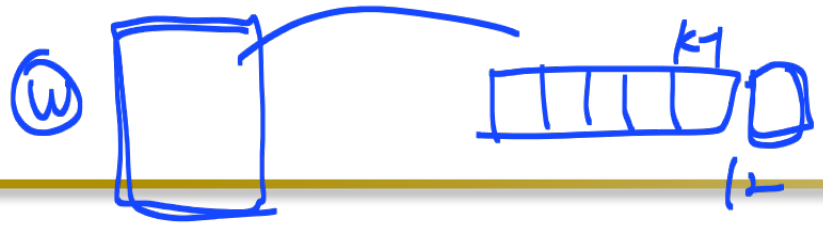
- As we have seen, the solution for S_4 is not part of the solution for S_5.
- So our definition of a subproblem is flawed and we need another one!
- Let's add another parameter: w, which will represent the exact weight for each subset of items.
- The subproblem then will be to compute $B[k,w]$.

Recursive Formula for subproblems

$$B[k, W] = \begin{cases} B[k-1, W] & \text{if } w_k > W \\ \max\{ B[k-1, W], B[k-1, W-w_k] + b_k \} & \text{otherwise} \end{cases}$$

최종 solution.

- The best subset of S_k that has the total weight W, either contains item k or not.
- First case: $w_k > W$. Item k can't be part of the solution, since if it was, the total weight would be > W, which is unacceptable.
- Second case: $w_k <= W$. Then the item k may be in the solution, and we choose the case with greater value.



Recursive Formula for subproblems

Recursive formula for subproblems:

$$B[k, W] = \begin{cases} B[k-1, W] & \text{if } w_k > W \\ \max\{ B[k-1, W], B[k-1, W-w_k] + b_k \} & \text{otherwise} \end{cases}$$

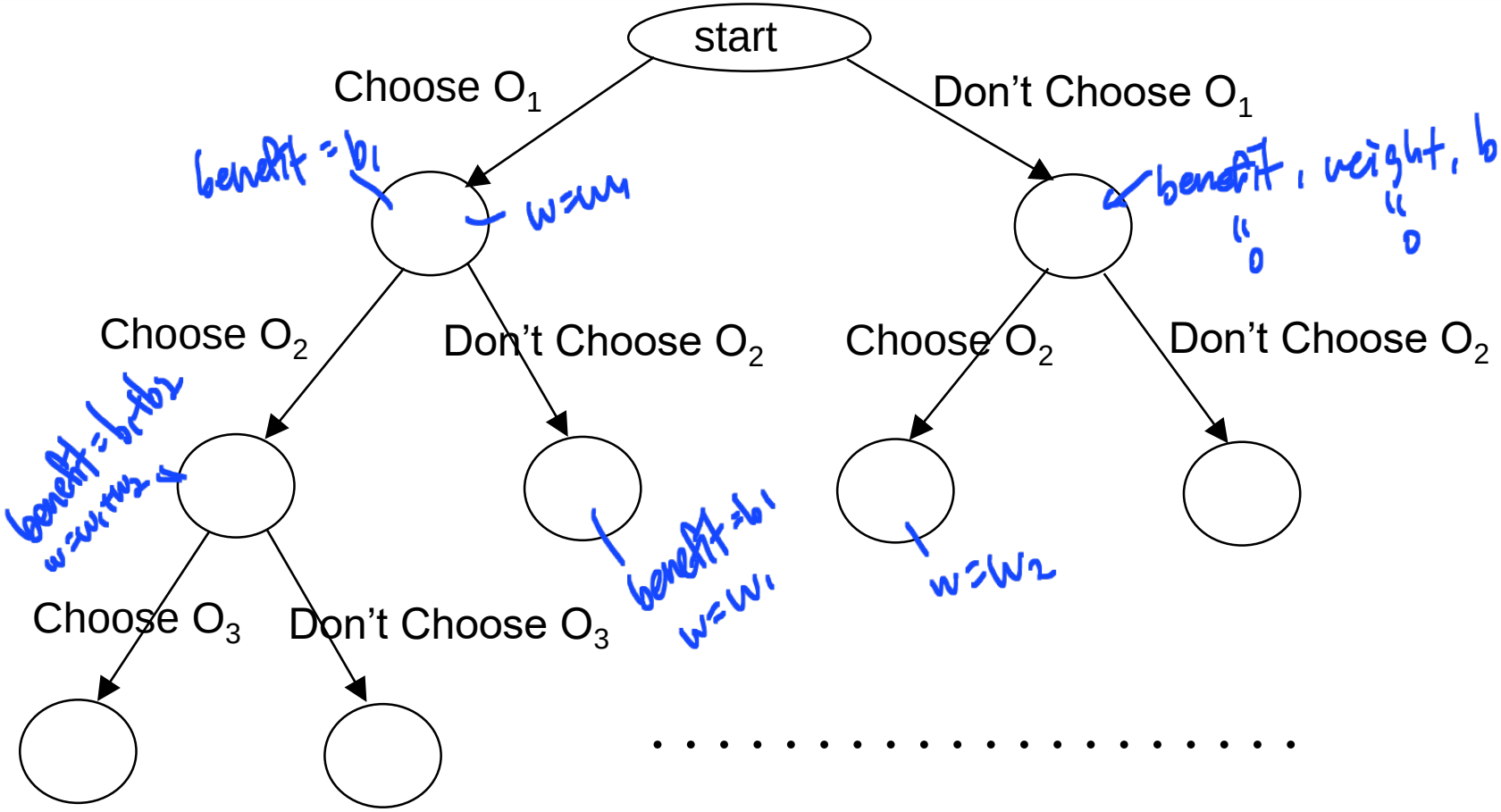
The second case means, that the best subset of S_k that has total weight W is one of the two:

1) the best subset of S_{k-1} that has total weight W, or
2) the best subset of S_{k-1} that has total weight W-w_k plus the item k

Pseudocode

```
for w = 0 to W
    B[0,w] = 0
for i = 1 to n
    B[i,0] = 0
    for w = 1 to W
        if w_i <= w // item i can be part of the solution
            if b_i + B[i-1,w-w_i] > B[i-1,w]
                B[i,w] = b_i + B[i-1,w- w_i]
            else    B[i,w] = B[i-1,w]
        else B[i,w] = B[i-1,w]  // w_i > w
```

[4] Branch and Bound – Best First Search



We do not visit every node, but determine if it is worth or not.

ITEM should be sorted in descending order according to b_i/w_i.

Branch and Bound

- At each node we compute three local values.

  benefit : sum of benefit value up to this node

  weight : sum of weights up to this node

  bound : upper bound on the benefit we could achieve by expanding beyond this node

- Computing '*bound*'

  At level *i*, initially *tot_weight = weight*, then add w_{i+1}, w_{i+2}, … w_{k-1} to *tot_weight* as long as *tot_weight* does not exceed *W*.

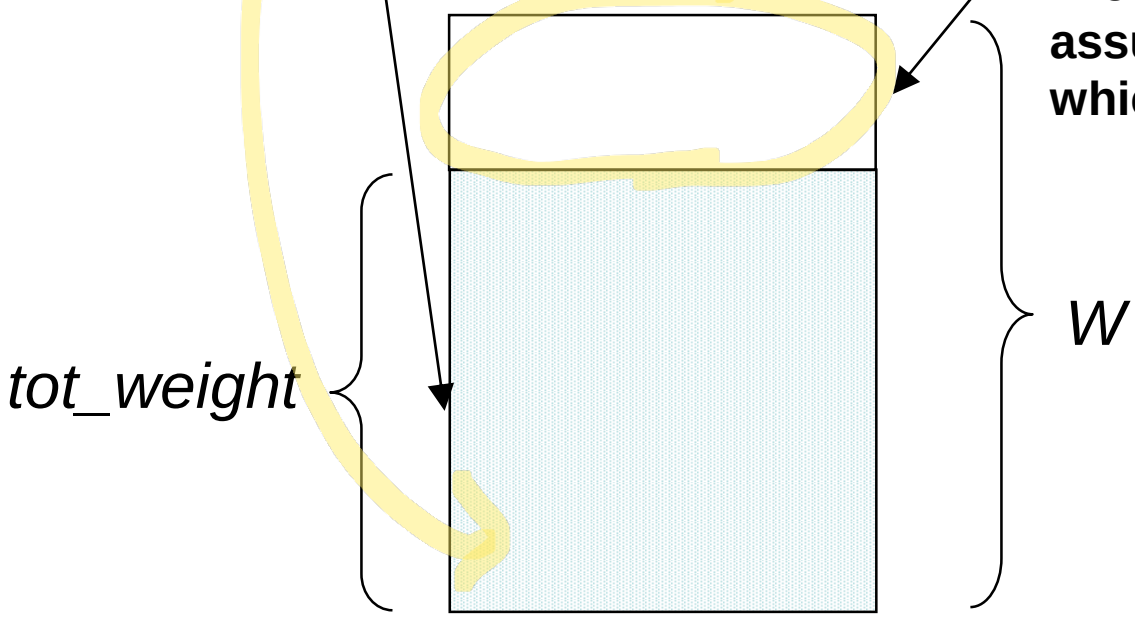
$$tot\_weight = weight + \sum_{j=i+1}^{k-1} w_j$$

Branch and Bound

Then

$$bound = benefit + \sum_{j=i+1}^{k-1} b_j + (W - tot\_weight) \times \frac{b_k}{w_k}$$

Profit for this part assuming fraction is taken, which is upper bound.



Branch and Bound

- And we need one more global variable '*max_benefit*'

  : benefit in the best solution found so far.

  Initially, *max_benefit* = 0.

  Afterwards at each node, if *weight* < *W*,

  max_benefit = max(*max_benefit*, *benefit*)

- Q : Can you guess the relationship between local variable '*bound*' and global variable '*max_beneit*'?

  bound : upper bound on the benefit we could achieve by expanding beyond this node

  max_benefit : benefit in the best solution found so far

Branch and Bound

- The node is evaluated as promising or non-promising.
  - promising : We will expand beyond this node.
  - nonpromising : We will stop here.
- The vertex is non-promising

  if *bound* ≤ *max_benefit* or *weight* > *W*

  (when *bound* ≤ *max_benefit*, we can't get the benefit more than *max_benefit* even if we expand the nodes.)
- The vertex is promising

  if *bound* > *max_benefit* and *weight* ≤ *W*

Branch and Bound

Two non-promising case

1) Case 1: *bound* ≤ *max_benefit*

   The '*benefit*' computed at this node is effective.

   We just do not expand beyond this node.

1) Case 2: *weight* > *W*

   The '*benefit*' computed at this node is not effective.

   So '*max_benefit*' will not be updated and we do not expand beyond this node.

Branch and Bound

Basically apply BFS and choose promising and unexpanded node with greatest bound.

Each node has

benefit

weight

bound

Example) 4 items, $W = 16$

i	*bi*	*wi*	*bi/wi*
1	$40	2	$20
2	$30	5	$6
3	$50	10	$5
4	$10	5	$2

Decreasing order of *bi/wi*

Branch and Bound

$$tot\_weight = weight + \sum_{j=i+1}^{k-1} w_j$$

$$bound = benefit + \sum_{j=i+1}^{k-1} b_j + (W - tot\_weight) \times \frac{b_k}{w_k}$$

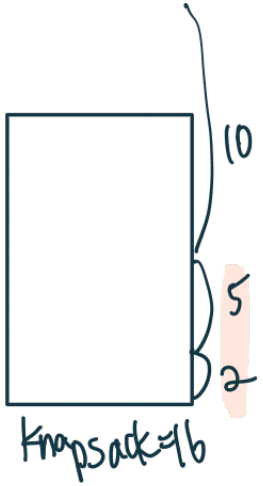
Vertex (0,0)

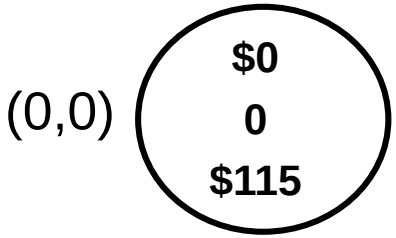
benefit = 0, weight = 0 ≤ W

max_benefit = $0

tot_weight = 0 + (2+5) = 7 ≤ W

bound = $0+($40+$30)+(16-7)*$50/10=$115 > max_benefit





b_i	w_i
$40	2
$30	5
$50	10
$10	5

Branch and Bound

Vertex (1,1)

benefit = 0+$40 = $40

weight = 0+2 = 2 ≤ W

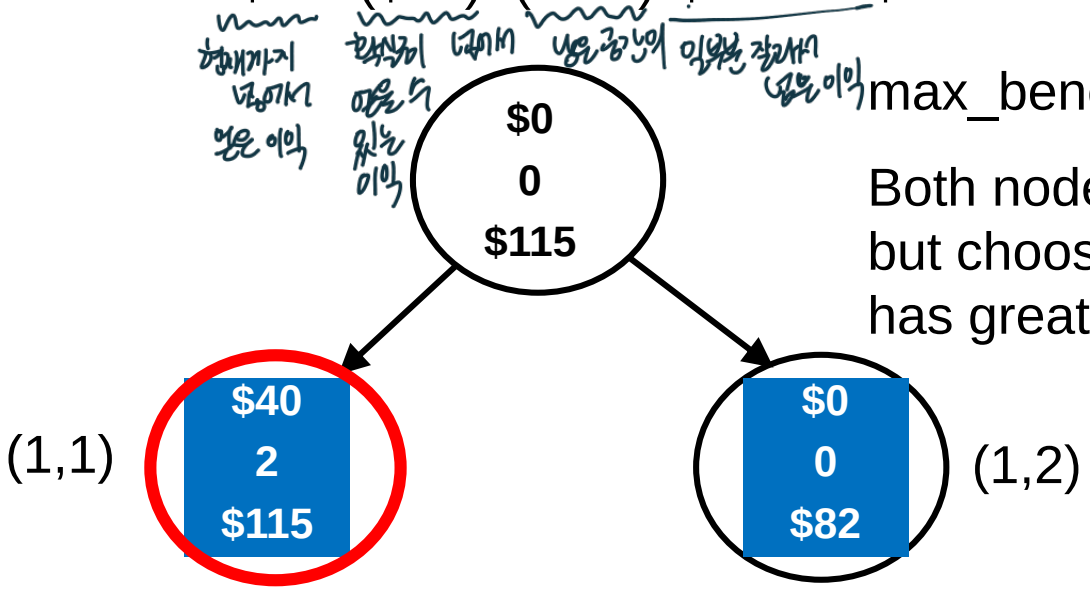
max_benefit = max($0, $40) = $40

tot_weight = 2+ (5) = 7 ≤ W

bound = $40+($30)+(16-7)*$50/10=$115 > max_benefit

$$tot\_weight = weight + \sum_{j=i+1}^{k-1} w_j$$

$$bound = benefit + \sum_{j=i+1}^{k-1} b_j + (W - tot\_weight) \times \frac{b_k}{w_k}$$



max_benefit = $40

Both nodes are promising, but choose (1,1) since it has greatest bound

b_i	w_i
$40	2
$30	5
$50	10
$10	5

Branch and Bound

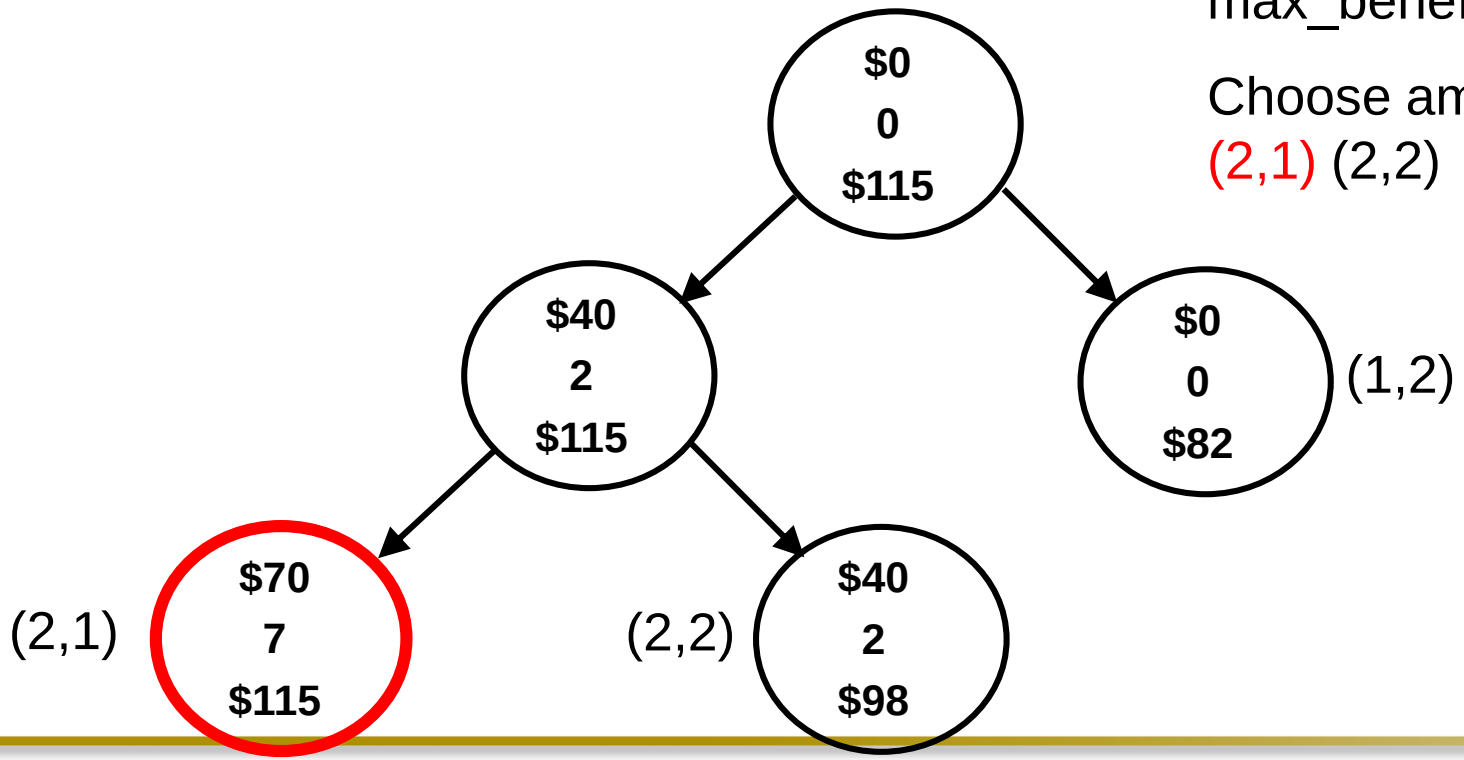
Vertex (2,1)

$$tot\_weight = weight + \sum_{j=i+1}^{k-1} w_j$$

$$bound = benefit + \sum_{j=i+1}^{k-1} b_j + (W - tot\_weight) \times \frac{b_k}{w_k}$$

max_benefit = $70

Choose among (1,2) (2,1) (2,2)



Branch and Bound

$$tot\_weight = weight + \sum_{j=i+1}^{k-1} w_j$$

$$bound = benefit + \sum_{j=i+1}^{k-1} b_j + (W - tot\_weight) \times \frac{b_k}{w_k}$$

Vertex (3,1)

benefit = \$70+\$50 = \$120

weight = 7+10 = 17 > W

max_benefit is still \$70

Vertex (3,2)

benefit = \$70+0 = \$70

weight = 7 $\leq$ W

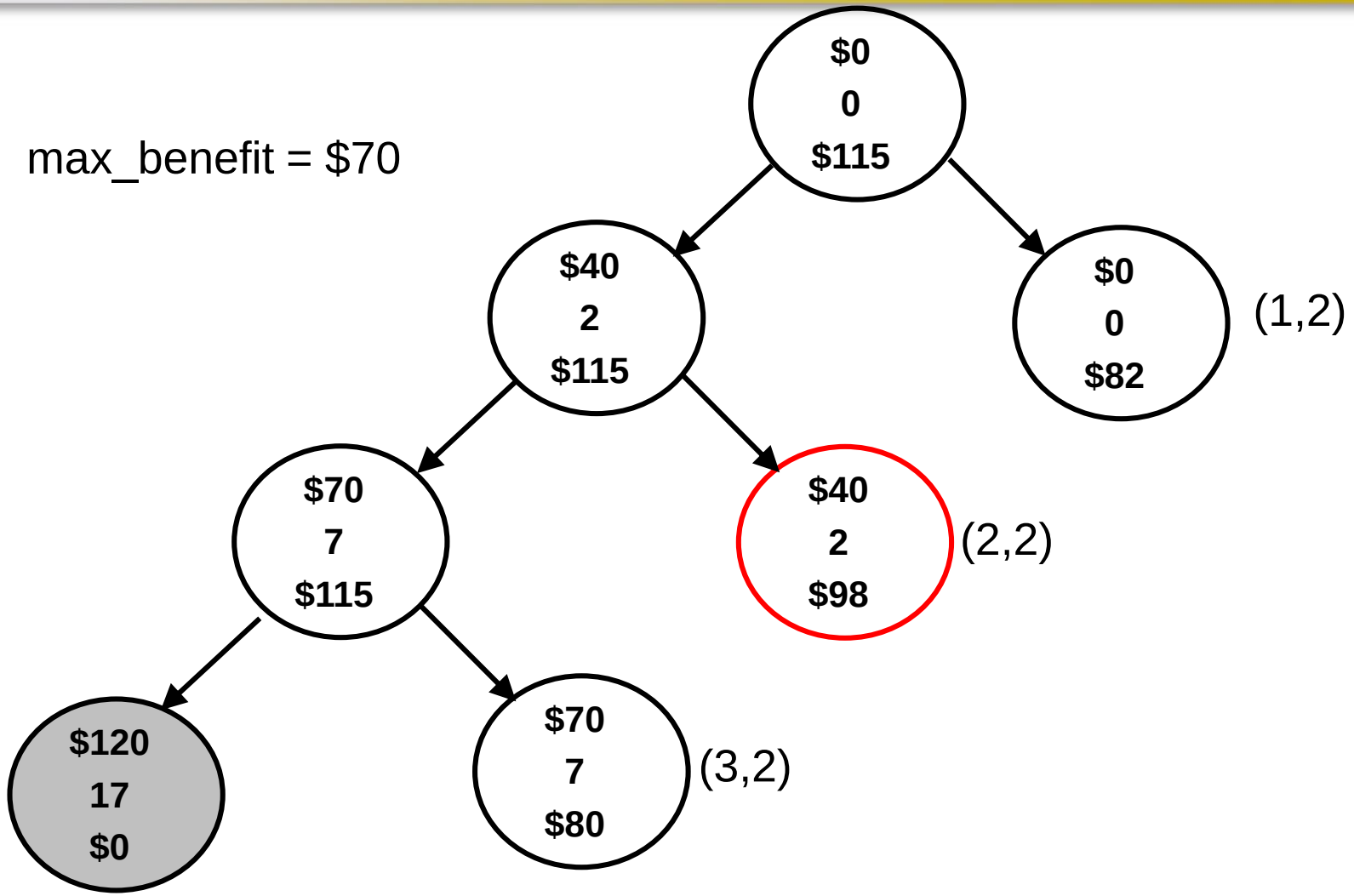
max_benefit = max(\$70, \$70) = \$70

tot_weight = 7+ (5) = 12 $\leq$ W

bound = \$70+\$10 + (16-12)*? = \$80 > max_benefit

Branch and Bound

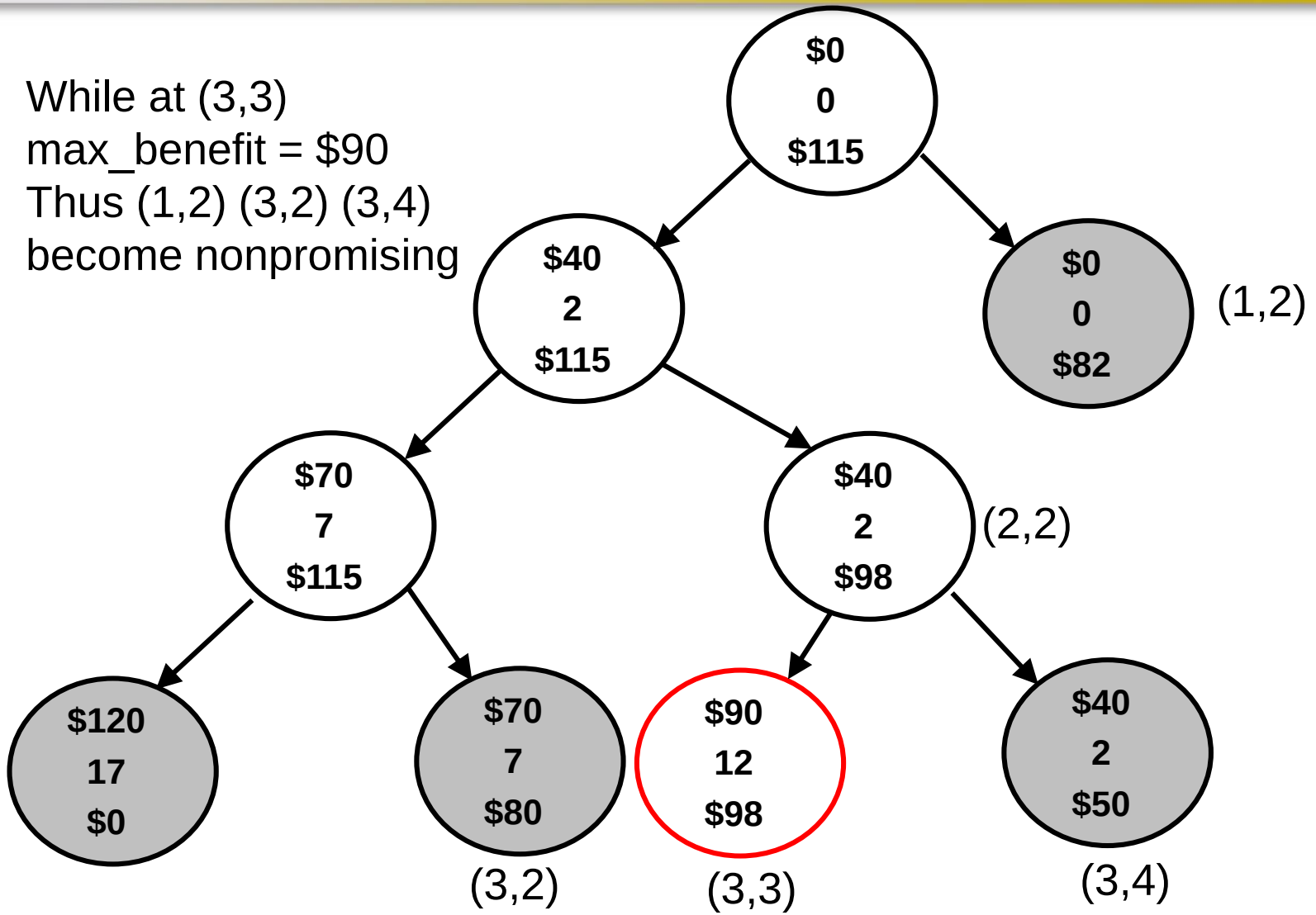
max_benefit = $70



Nonpromising since weight > W

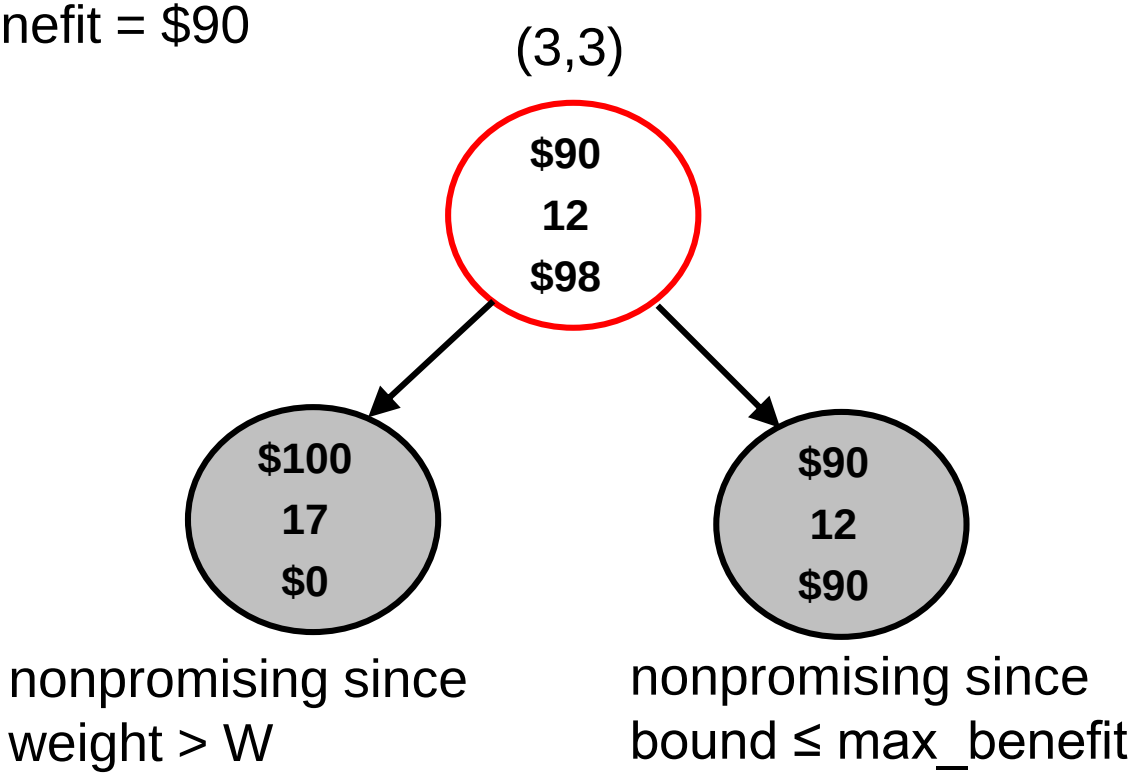
Branch and Bound

While at (3,3)
max_benefit = $90
Thus (1,2) (3,2) (3,4)
become nonpromising



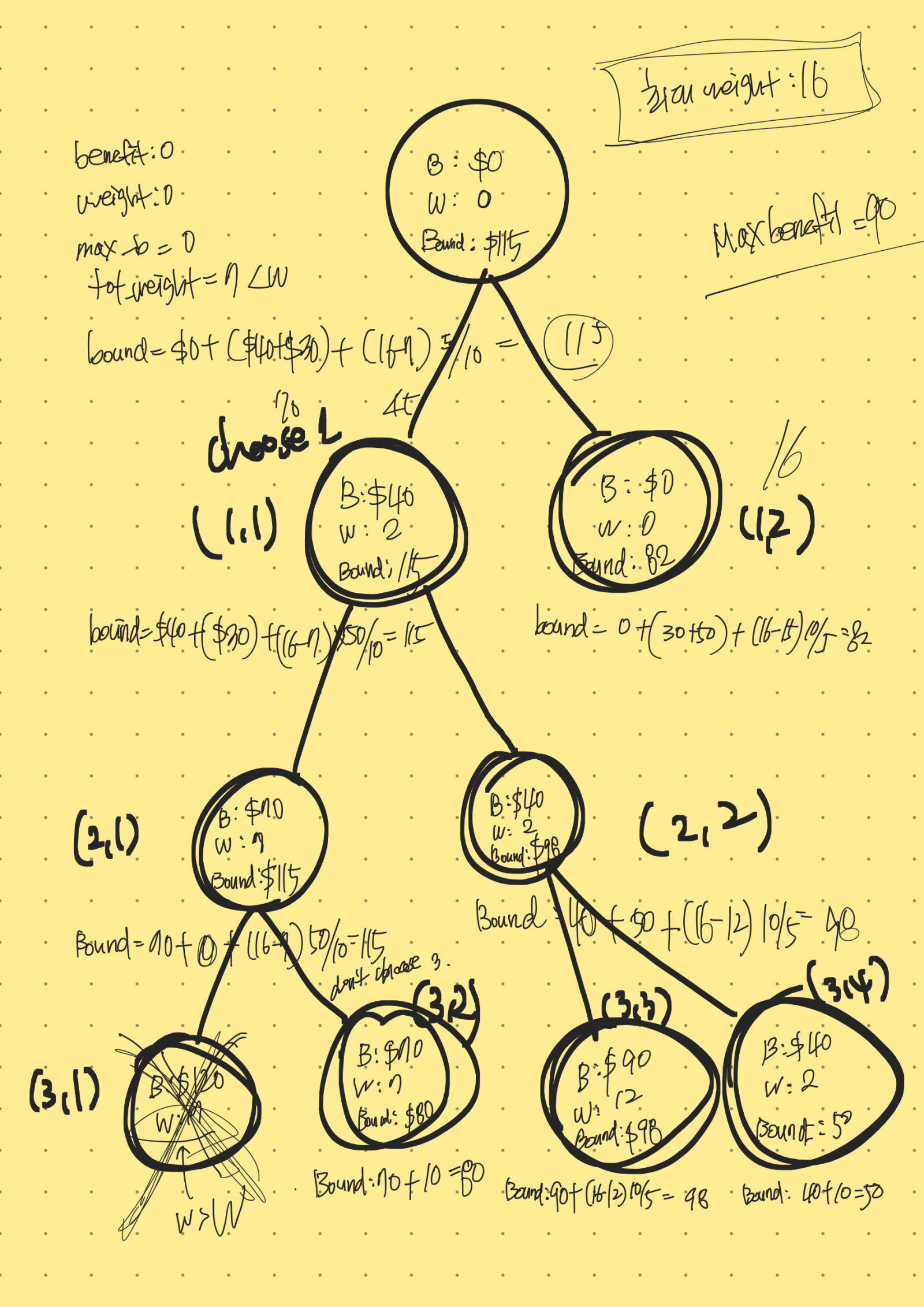
Branch and Bound

max_benefit = $90



Therefore, the vertex (3,3) has max_benefit of $90.

i.e., choose O_1 and O_3


bag weight : 16
benefit : 0
weight : 0
max_bo = 0
tot_weight = n < W
B : $0
W : 0
Bound : $115
Max benefit = 90
bound = $0 + ($40+$30) + (16-n) * 50/10 = 115
choose 1
take
(1,1)
B : $40
W : 2
Bound : 115
(1,2)
B : $0
W : 0
Bound : 82
bound = $40 + ($30) + ((16-n)*50/10 = 115
bound = 0 + (30+50) + (16-14) 10/5 = 82
(2,1)
B : $10
W : 9
Bound : $115
(2,2)
B : $40
W : 2
Bound : $98
Bound = 40 + 50 + (16-12) 10/5 = 98
Bound = 10 + 0 + (16-9) 50/10 = 115
don't choose 3.
(3,1)
B : $10
W : 9
W > W
(3,2)
B : $10
W : 9
Bound : $80
Bound : 10 + 10 = 50
(3,3)
B : $90
W : 12
Bound : $98
Bound : 90 + (16-12) 10/5 = 98
(3,4)
B : $40
W : 2
Bound : 50
Bound : 40 + 10 = 50