Branch and Bound (1)

By: Aminul Islam

Based on Chapter 6 of Foundations of Algorithms

Objectives

Objectives

- Describe the branch-and-bound technique for solving optimization problems
- Contrast the branch-and-bound technique with the backtracking
- Apply the branch-and-bound technique to solve the 0-1 Knapsack Problem

Branch-and-Bound Design Strategy

Branch-and-Bound Design Strategy

- Branch-and-bound design strategy is similar to backtracking

Branch-and-Bound Design Strategy

- Branch-and-bound design strategy is similar to backtracking
  - State space tree used to solve problem

Branch-and-Bound Design Strategy

- Branch-and-bound design strategy is similar to backtracking
  - State space tree used to solve problem
- Difference between branch-and-bound and backtracking:

Branch-and-Bound Design Strategy

- Branch-and-bound design strategy is similar to backtracking
    - State space tree used to solve problem
- Difference between branch-and-bound and backtracking:
    1. branch-and-bound is not limited to a particular tree traversal

Branch-and-Bound Design Strategy

- Branch-and-bound design strategy is similar to backtracking
  - State space tree used to solve problem
- Difference between branch-and-bound and backtracking:
  1. branch-and-bound is not limited to a particular tree traversal
  2. branch-and-bound is usually used for optimization problems

Branch-and-Bound Design Strategy (2)

Branch-and-Bound Design Strategy (2)

- Bound (which is generally a number) is computed at a node to determine whether the node is promising

Branch-and-Bound Design Strategy (2)

- Bound (which is generally a number) is computed at a node to determine whether the node is promising
- Bound indicates the value of the solution that could be obtained by expanding beyond the node.

Branch-and-Bound Design Strategy (2)

- Bound (which is generally a number) is computed at a node to determine whether the node is promising
- Bound indicates the value of the solution that could be obtained by expanding beyond the node.
    - We had a similar concept in "upper bound" of 0-1 Knapsack using Backtracking

Branch-and-Bound Design Strategy (2)

- Bound (which is generally a number) is computed at a node to determine whether the node is promising
- Bound indicates the value of the solution that could be obtained by expanding beyond the node.
  - We had a similar concept in "upper bound" of 0-1 Knapsack using Backtracking
- If Bound is not better than the value of the best solution found so far, node is non-promising

Branch-and-Bound Design Strategy (2)

- Bound (which is generally a number) is computed at a node to determine whether the node is promising
- Bound indicates the value of the solution that could be obtained by expanding beyond the node.
  - We had a similar concept in "upper bound" of 0-1 Knapsack using Backtracking
- If Bound is not better than the value of the best solution found so far, node is non-promising
  - Otherwise, the node is promising

Breadth-first Search Tree

Breadth-first Search Tree

- Branch and Bound uses (variations) of breadth-first search on the tree

Breadth-first Search Tree

- Branch and Bound uses (variations) of breadth-first search on the tree
- Recap of the breadth-first search

Breadth-first Search Tree

- Branch and Bound uses (variations) of breadth-first search on the tree
- Recap of the breadth-first search
  - Use Queue - FIFO

Breadth-first Search Tree

- Branch and Bound uses (variations) of breadth-first search on the tree
- Recap of the breadth-first search
    - Use Queue - FIFO
    - Visit root first

Breadth-first Search Tree

- Branch and Bound uses (variations) of breadth-first search on the tree
- Recap of the breadth-first search
  - Use Queue - FIFO
  - Visit root first
  - Visit all nodes at level 1 next

Breadth-first Search Tree

- Branch and Bound uses (variations) of breadth-first search on the tree
- Recap of the breadth-first search
  - Use Queue - FIFO
  - Visit root first
  - Visit all nodes at level 1 next
  - Visit all nodes at level 2 next

Breadth-first Search Tree

- Branch and Bound uses (variations) of breadth-first search on the tree
- Recap of the breadth-first search
  - Use Queue - FIFO
  - Visit root first
  - Visit all nodes at level 1 next
  - Visit all nodes at level 2 next
  - Visit all nodes at level n

General Branch and Bound Structure based on Breadth First Search

General Branch and Bound Structure based on Breadth First Search

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

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

General Branch and Bound Structure based on Breadth First Search

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

   initialize(Q);          // Make an empty queue
   v = root of T;
   enqueue(Q, v);
   best = value(v);
   while (! empty(Q)){
       dequeue(Q, v);
       for (each child u of v){
           if (value(u) is better than best)
               best = value(u);
           if (bound(u) is better than best)
               enqueue(Q, u);
       }
   }
}
```

General Branch and Bound Structure based on Breadth First Search

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

  initialize(Q);          // Make an empty queue
  v = root of T;
  enqueue(Q, v);
  best = value(v);
  while (! empty(Q)){
     dequeue(Q, v);
     for (each child u of v){
        if (value(u) is better than best)     // Check if a better
           best = value(u);                   // solution is found
        if (bound(u) is better than best)
           enqueue(Q, u);
     }
  }
}
```

General Branch and Bound Structure based on Breadth First Search

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

  initialize(Q);          // Make an empty queue
  v = root of T;
  enqueue(Q, v);
  best = value(v);
  while (! empty(Q)){
     dequeue(Q, v);
     for (each child u of v){
        if (value(u) is better than best)
           best = value(u);
        if (bound(u) is better than best)
           enqueue(Q, u);
     }
  }
}
```

// Check if a better
// solution is found

// Check if promising

0-1 Knapsack Problem using Breadth-First Search with Branch and Bound

0-1 Knapsack Problem using Breadth-First Search with Branch and Bound

- Similar to the Backtracking version but breadth-first rather than depth-first

0-1 Knapsack Problem using Breadth-First Search with Branch and Bound

- Similar to the Backtracking version but breadth-first rather than depth-first
- Let *weight* and *profit* be the total weight and total profit of the items that have been included up to a node.

Determine if a Node is Promising

Determine if a Node is Promising

Promising cases (Same as backtracking):

Determine if a Node is Promising

Promising cases (Same as backtracking):

- Sum of the weights up to the node $< W$ (i.e, knapsack capacity)

Determine if a Node is Promising

Promising cases (Same as backtracking):

- Sum of the weights up to the node $< W$ (i.e, knapsack capacity)
- A node at level i is promising if:
  $(\text{upper bound})_i > \text{maxprofit}$

Example of 0-1 Knapsack Problem using Breadth-First Search with Branch and Bound

Example of 0-1 Knapsack Problem using Breadth-First Search with Branch and Bound

$n = 4, \ W = 16$

Item 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

Promising:
- (current node) bound $>$ maxprofit
- (current node) weight $< W$

maxprofit $=$

Example of 0-1 Knapsack Problem using Breadth-First Search with Branch and Bound

$n = 4,\ W = 16$

Item 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

Promising:
- (current node) bound $>$ maxprofit
- (current node) weight $< W$

maxprofit $=$ \$0

Example of 0-1 Knapsack Problem using Breadth-First Search with Branch and Bound

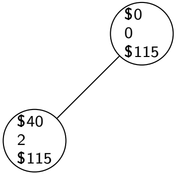
$n = 4, \ W = 16$

Item 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

Promising:

- (current node) bound $>$ maxprofit
- (current node) weight $< W$

maxprofit $= \$40$



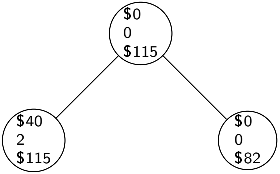
Example of 0-1 Knapsack Problem using Breadth-First Search with Branch and Bound

$n = 4, \ W = 16$

Item 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

Promising:
- (current node) bound $>$ maxprofit
- (current node) weight $< W$

maxprofit $=$ \$40



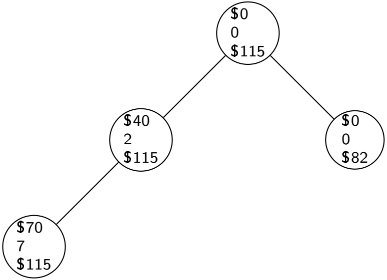
Example of 0-1 Knapsack Problem using Breadth-First Search with Branch and Bound

$n = 4, \; W = 16$

Item 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

Promising:
- (current node) bound $>$ maxprofit
- (current node) weight $< W$

maxprofit $= \$70$



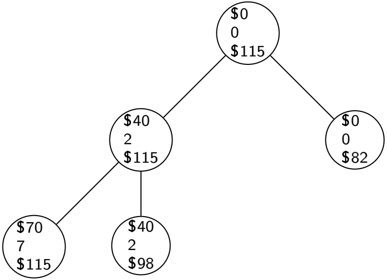
Example of 0-1 Knapsack Problem using Breadth-First Search with Branch and Bound

$n = 4, \ W = 16$

Item 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

Promising:
- (current node) bound $>$ maxprofit
- (current node) weight $< W$

maxprofit $= \$70$



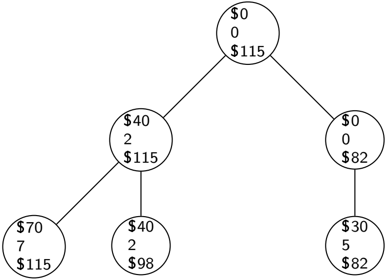
Example of 0-1 Knapsack Problem using Breadth-First Search with Branch and Bound

$n = 4,\ W = 16$

Item 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

Promising:
- (current node) bound $>$ maxprofit
- (current node) weight $< W$

maxprofit $= \$70$



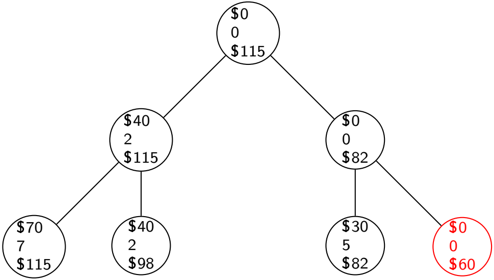
Example of 0-1 Knapsack Problem using Breadth-First Search with Branch and Bound

$n = 4, \ W = 16$

Item 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

Promising:
- (current node) bound $>$ maxprofit
- (current node) weight $< W$

maxprofit $= \$70$



Example of 0-1 Knapsack Problem using Breadth-First Search with Branch and Bound

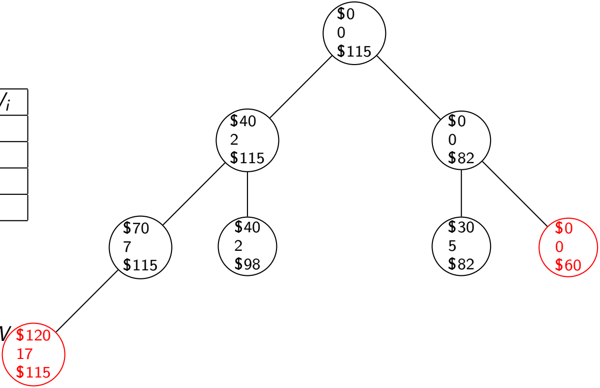
$n = 4,\ W = 16$

Item 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

Promising:

- (current node) bound $>$ maxprofit
- (current node) weight $< W$

maxprofit $= \$70$



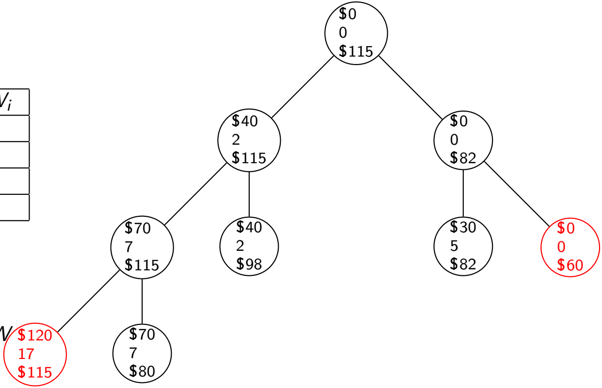
Example of 0-1 Knapsack Problem using Breadth-First Search with Branch and Bound

$n = 4, \ W = 16$

Item 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

Promising:
- (current node) bound > maxprofit
- (current node) weight < W

maxprofit = \$70



Example of 0-1 Knapsack Problem using Breadth-First Search with Branch and Bound

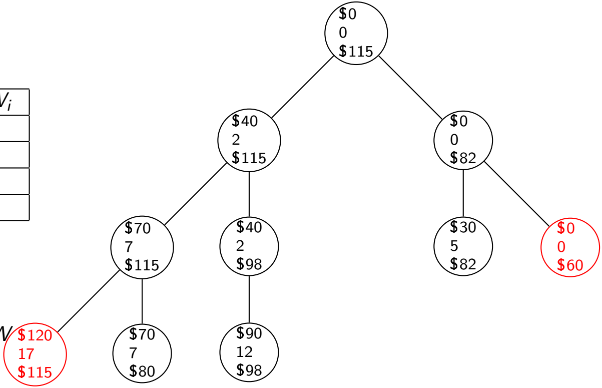
$n = 4, \ W = 16$

Item 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

Promising:

- (current node) bound > maxprofit
- (current node) weight < W

maxprofit $= \$90$



Example of 0-1 Knapsack Problem using Breadth-First Search with Branch and Bound

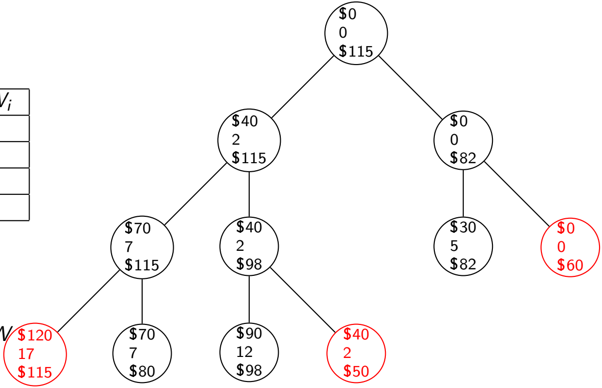
$n = 4, \; W = 16$

Item 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

Promising:

- (current node) bound > maxprofit
- (current node) weight < W

maxprofit $= \$90$



Example of 0-1 Knapsack Problem using Breadth-First Search with Branch and Bound

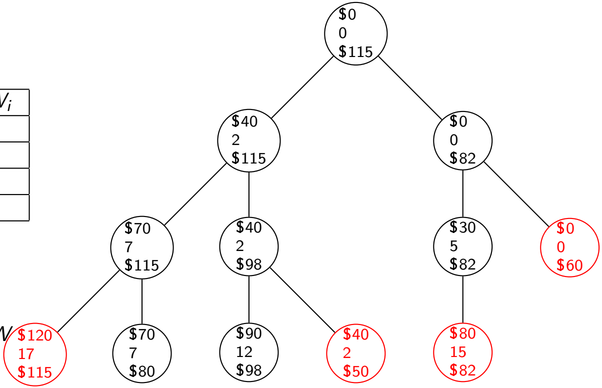
$n = 4, \ W = 16$

Item 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

Promising:

- (current node) bound $>$ maxprofit
- (current node) weight $< W$

maxprofit $=$ \$90



Example of 0-1 Knapsack Problem using Breadth-First Search with Branch and Bound

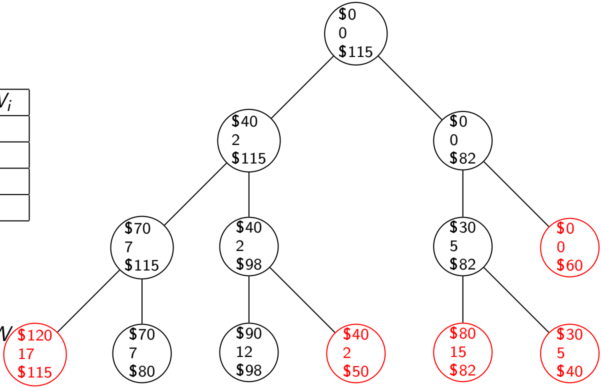
$n = 4, \ W = 16$

Item 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

Promising:

- (current node) bound $>$ maxprofit
- (current node) weight $< W$

maxprofit $=$ \$90



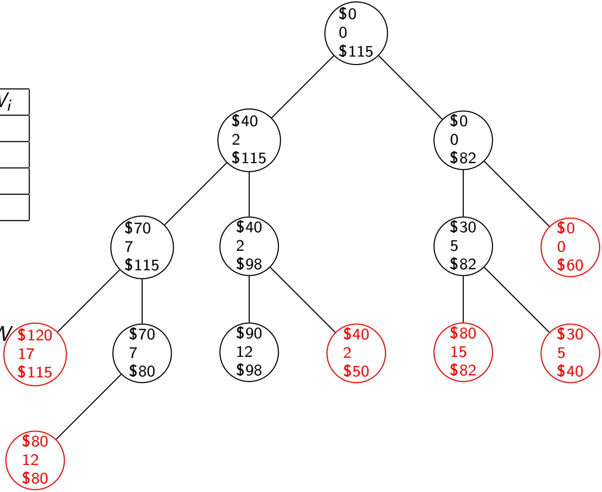
Example of 0-1 Knapsack Problem using Breadth-First Search with Branch and Bound

$n = 4, \ W = 16$

Item 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

Promising:
- (current node) bound $>$ maxprofit
- (current node) weight $< W$

maxprofit $= \$90$



Example of 0-1 Knapsack Problem using Breadth-First Search with Branch and Bound

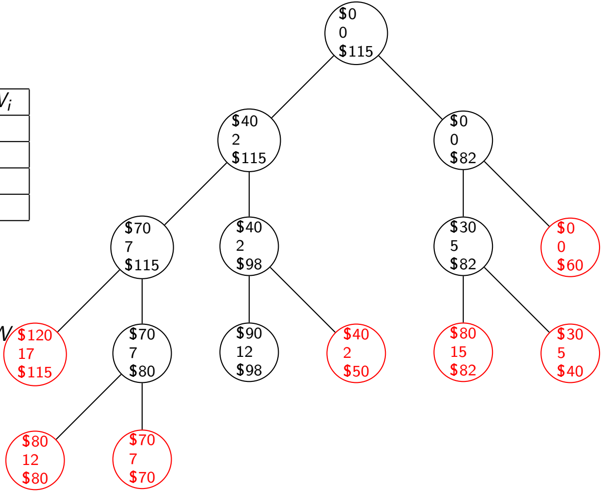
$n = 4,\ W = 16$

Item 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

Promising:

- (current node) bound $>$ maxprofit
- (current node) weight $< W$

maxprofit $= \$90$



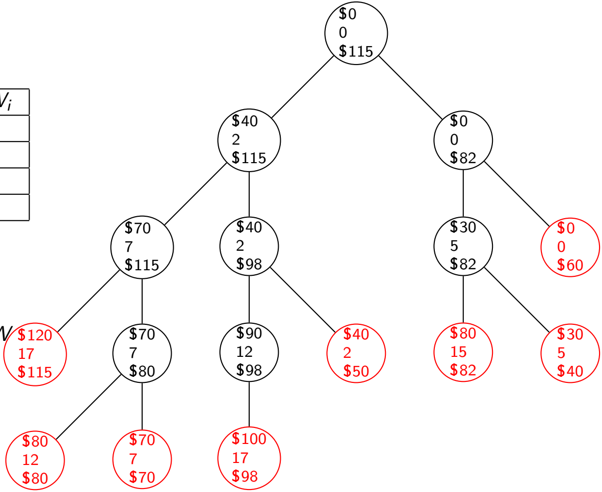
Example of 0-1 Knapsack Problem using Breadth-First Search with Branch and Bound

$n = 4, \ W = 16$

Item 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

Promising:
- (current node) bound > maxprofit
- (current node) weight < W

maxprofit = \$90



Example of 0-1 Knapsack Problem using Breadth-First Search with Branch and Bound

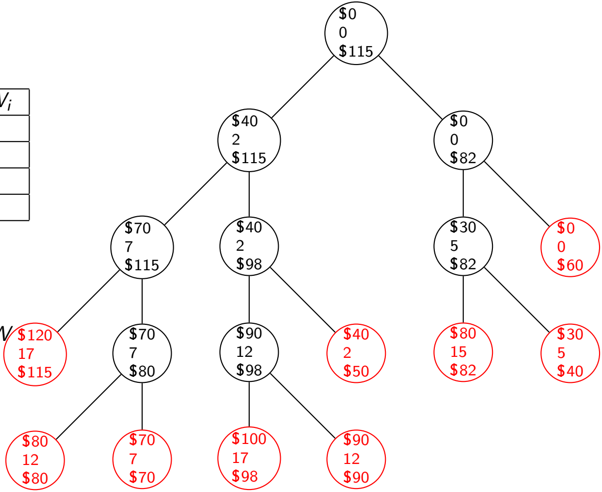
$n = 4,\ W = 16$

Item 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

Promising:

- (current node) bound $>$ maxprofit
- (current node) weight $< W$

maxprofit $=$ \$90



Summary of breadth-first search with Branch-and-Bound Pruning

Summary of breadth-first search with Branch-and-Bound Pruning

- In general, Breadth-First Search strategy has no advantage over a depth-first search (backtracking)

Summary of breadth-first search with Branch-and-Bound Pruning

- In general, Breadth-First Search strategy has no advantage over a depth-first search (backtracking)
- However, we can improve our search by using our bound to do more than just determine whether a node is promising.

Use Bound to Improve Search

Use Bound to Improve Search

- Why not using Bound to guide us to the solution quicker?

Use Bound to Improve Search

- Why not using Bound to guide us to the solution quicker?
  - The node with a higher Bound will have a better potential for an optimal solution.

Use Bound to Improve Search

- Why not using Bound to guide us to the solution quicker?
  - The node with a higher Bound will have a better potential for an optimal solution.
- Order for expansion can be determined by the best bound rather than pre-determined methods (i.e., DFS or BFS)

Use Bound to Improve Search

- Why not using Bound to guide us to the solution quicker?
  - The node with a higher Bound will have a better potential for an optimal solution.
- Order for expansion can be determined by the best bound rather than pre-determined methods (i.e., DFS or BFS)
- Use priority queue for implementation

Example of 0-1 Knapsack Problem using Best-First Search with Branch and Bound

Example of 0-1 Knapsack Problem using Best-First Search with Branch and Bound

$n = 4, \; W = 16$

Item 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

• Determine promising, unexpanded node with the greatest bound

Promising:
• (current node) bound $>$ maxprofit
• (current node) weight $< W$

maxprofit $=$

Example of 0-1 Knapsack Problem using Best-First Search with Branch and Bound

$n = 4, \ W = 16$

Item 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

- Determine promising, unexpanded node with the greatest bound

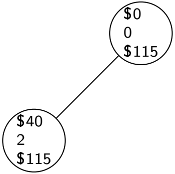
Promising:
- (current node) bound $>$ maxprofit
- (current node) weight $<$ W

maxprofit $=\$0$

Example of 0-1 Knapsack Problem using Best-First Search with Branch and Bound

$n = 4, \ W = 16$

Item 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



• Determine promising, unexpanded node with the greatest bound

Promising:
• (current node) bound $>$ maxprofit
• (current node) weight $<$ W

maxprofit $=\$40$

Example of 0-1 Knapsack Problem using Best-First Search with Branch and Bound

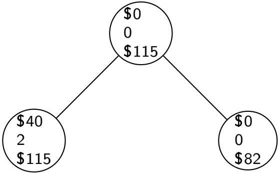
$n = 4, \ W = 16$

Item 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

- Determine promising, unexpanded node with the greatest bound

Promising:
- (current node) bound > maxprofit
- (current node) weight $< W$

maxprofit =\$40



Example of 0-1 Knapsack Problem using Best-First Search with Branch and Bound

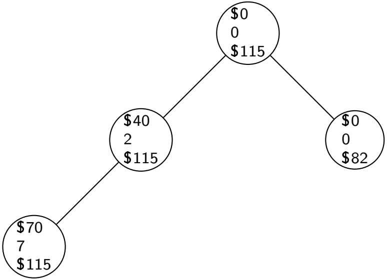
$n = 4, \ W = 16$

Item 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

- Determine promising, unexpanded node with the greatest bound

Promising:
- (current node) bound $>$ maxprofit
- (current node) weight $< W$

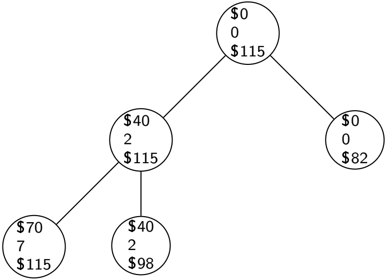
maxprofit $=\$70$



Example of 0-1 Knapsack Problem using Best-First Search with Branch and Bound

$n = 4,\ W = 16$

Item 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



- Determine promising, unexpanded node with the greatest bound

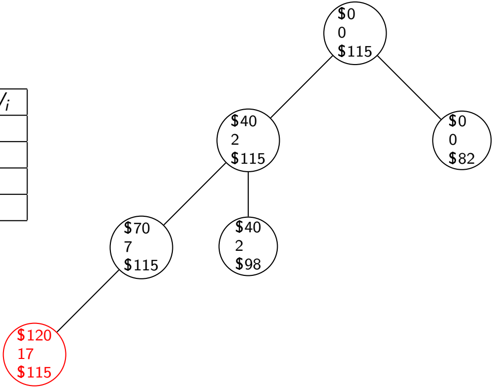
Promising:
- (current node) bound $>$ maxprofit
- (current node) weight $< W$

maxprofit $=\$70$

Example of 0-1 Knapsack Problem using Best-First Search with Branch and Bound

$n = 4, \ W = 16$

Item 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



- Determine promising, unexpanded node with the greatest bound

Promising:
- (current node) bound $>$ maxprofit
- (current node) weight $< W$

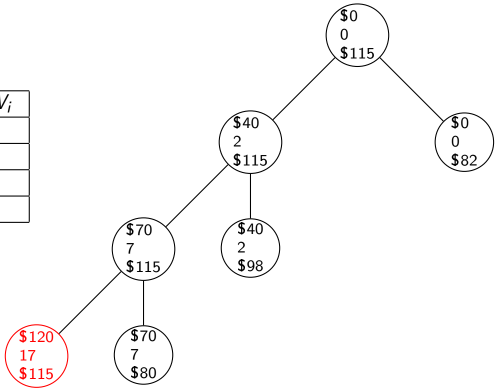
maxprofit $=\$70$

Example of 0-1 Knapsack Problem using Best-First Search with Branch and Bound

$n = 4, \ W = 16$

Item 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

- Determine promising, unexpanded node with the greatest bound



Promising:
- (current node) bound > maxprofit
- (current node) weight < W

maxprofit =\$70

Example of 0-1 Knapsack Problem using Best-First Search with Branch and Bound

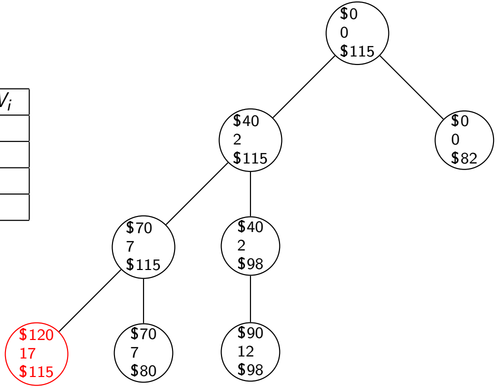
$n = 4, \ W = 16$

Item 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

- Determine promising, unexpanded node with the greatest bound

Promising:
- (current node) bound $>$ maxprofit
- (current node) weight $< W$

maxprofit $=$\$90



Example of 0-1 Knapsack Problem using Best-First Search with Branch and Bound

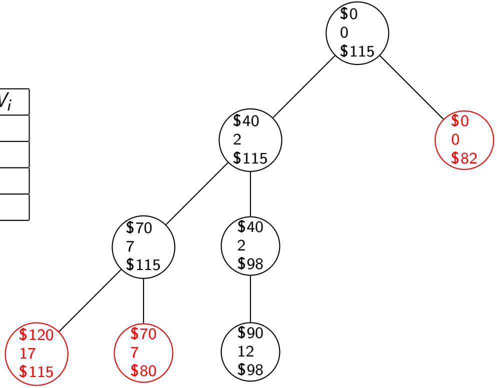
$n = 4, \; W = 16$

Item 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

- Determine promising, unexpanded node with the greatest bound

Promising:
- (current node) bound > maxprofit
- (current node) weight < W

maxprofit =\$90



Example of 0-1 Knapsack Problem using Best-First Search with Branch and Bound

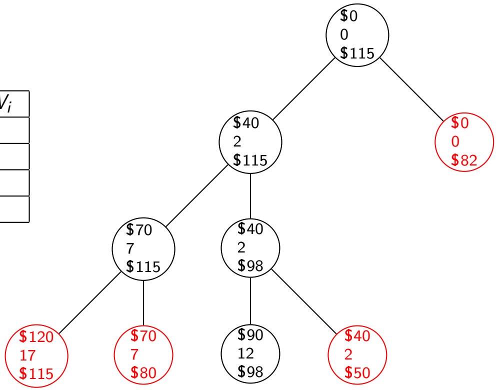
$n = 4, \ W = 16$

Item 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

- Determine promising, unexpanded node with the greatest bound

Promising:
- (current node) bound $>$ maxprofit
- (current node) weight $< W$

maxprofit $=\$90$



Example of 0-1 Knapsack Problem using Best-First Search with Branch and Bound

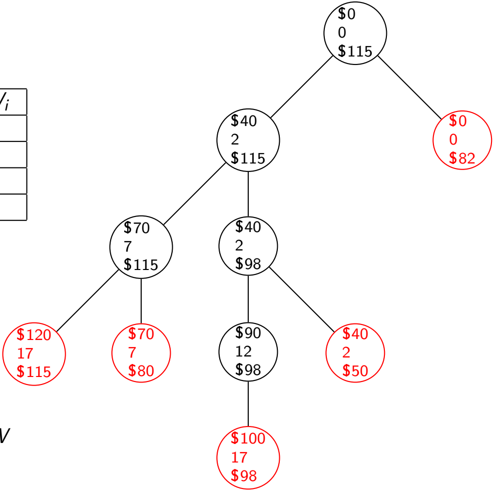
$n = 4, \; W = 16$

Item 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

• Determine promising, unexpanded node with the greatest bound

Promising:
• (current node) bound > maxprofit
• (current node) weight < W

maxprofit =\$90



Example of 0-1 Knapsack Problem using Best-First Search with Branch and Bound

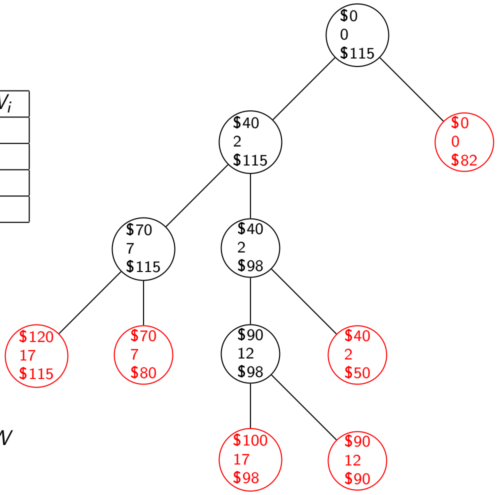
$n = 4, \ W = 16$

Item 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

- Determine promising, unexpanded node with the greatest bound

Promising:
- (current node) bound $>$ maxprofit
- (current node) weight $< W$

maxprofit $=\$90$



General Algorithm for B&B based on Best First Search

General Algorithm for B&B based on Best First Search

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

  initialize(PQ);
  v = root of T;
  best = value(v);
  insert(PQ, v);
  while (! empty(PQ)){
     remove(PQ, v);
    if (bound(v) is better than best)
        for (each child u of v){
            if (value(u) is better than best)
               (best = value(u);
            if (bound(u) is better than best)
               insert(PQ, u);
        }
    }
}
```

General Algorithm for B&B based on Best First Search

```
void best_first_branch_and_bound (state_space_tree T,
                                  number& best)
{
  priority_queue_of_node PQ;          // Priority queue
  node u, v;                          // instead of a
                                      // normal queue
  initialize(PQ);
  v = root of T;
  best = value(v);
  insert(PQ, v);
  while (! empty(PQ)){
     remove(PQ, v);
    if (bound(v) is better than best)
        for (each child u of v){
            if (value(u) is better than best)
               (best = value(u);
            if (bound(u) is better than best)
               insert(PQ, u);
        }
   }
}
```

General Algorithm for B&B based on Best First Search

```
void best_first_branch_and_bound (state_space_tree T,
                                  number& best)
{
  priority_queue_of_node PQ;            // Priority queue
  node u, v;                            // instead of a
                                        // normal queue
  initialize(PQ);
  v = root of T;
  best = value(v);
  insert(PQ, v);                        // Check if the node
  while (! empty(PQ)){                  // is still promising
    remove(PQ, v);
    if (bound(v) is better than best)
       for (each child u of v){
          if (value(u) is better than best)
             (best = value(u);
          if (bound(u) is better than best)
             insert(PQ, u);
       }
  }
}
```

B&B Algorithm based on Best First Search to Solve 0-1 Knapsack Problem

B&B Algorithm based on Best First Search to Solve 0-1 Knapsack Problem

```
void knapsack3 (int n,
                const int p[], const int w[],
                int W,
                int& maxprofit)
{
  priority_queue_of_node PQ;
  node u, v;

  initialize(PQ);
  v.level = 0; v.profit = 0; v.weight = 0;
  maxprofit = 0;
  v.bound = bound(v);
  insert(PQ, v);
```

Continue ...

Continue ...

```
while (!empty(PQ)){
   remove(PQ, v);
   if (v.bound > maxprofit){
      u.level = v.level + 1;
      u.weight = v.weight + w[u.level];
      u.profit = v.profit + p[u.level];

      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;
      u.profit = v.profit;
      u.bound = bound(u);
      if (u.bound > maxprofit)
         insert(PQ, u);
   }
  }
}
```

Bound Function for the 0-1 Knapsack Problem

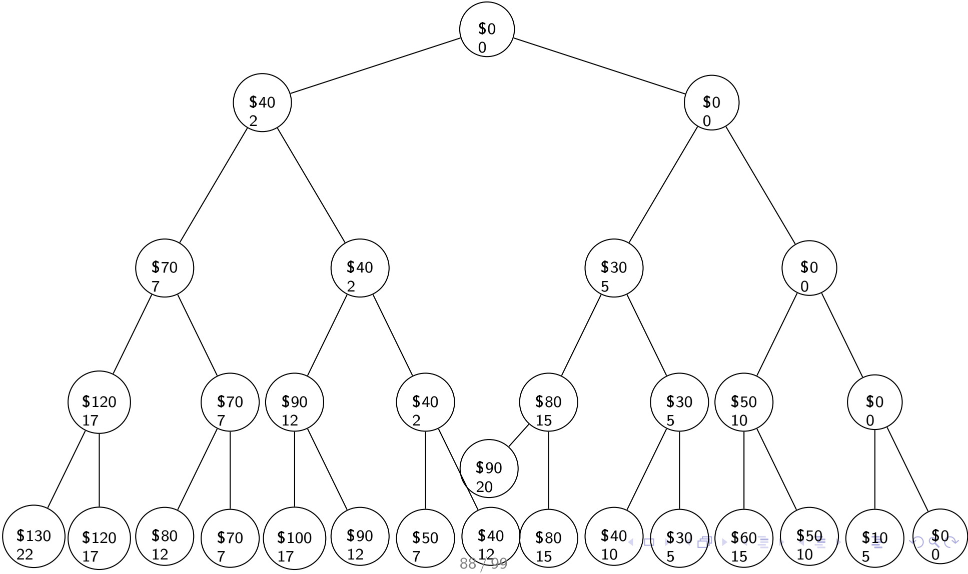
Bound Function for the 0-1 Knapsack Problem

```
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];
         result = result + p[j];
        j++;
      }
      k = j;
      if (k <=n)
        result = result + (W- totweight) * p[k]

      return result;
   }
}
```

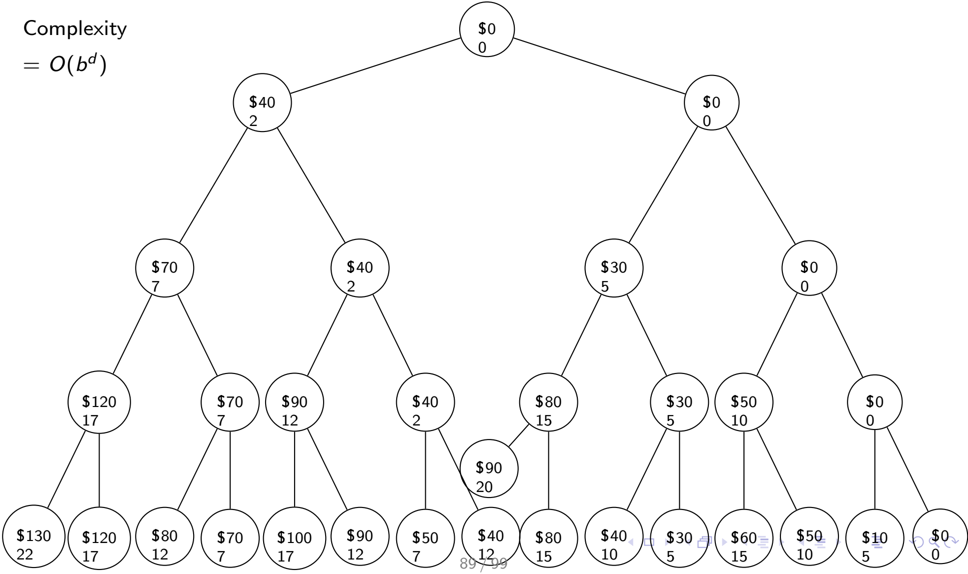
Time Complexity and Space Complexity of **Breadth-First Search**

Time Complexity and Space Complexity of **Breadth-First Search**



Time Complexity and Space Complexity of **Breadth-First Search**

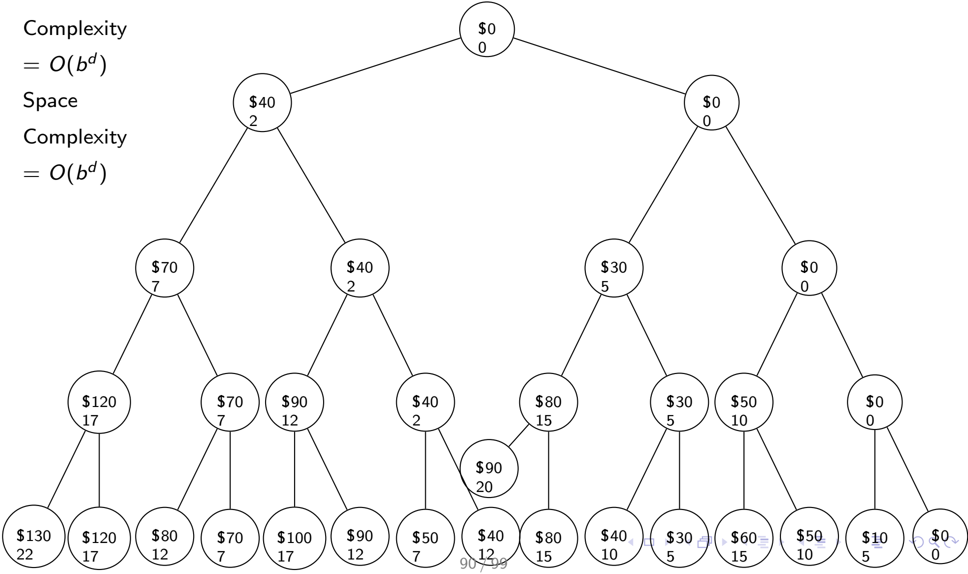
Time Complexity $= O(b^d)$



Time Complexity and Space Complexity of **Breadth-First Search**

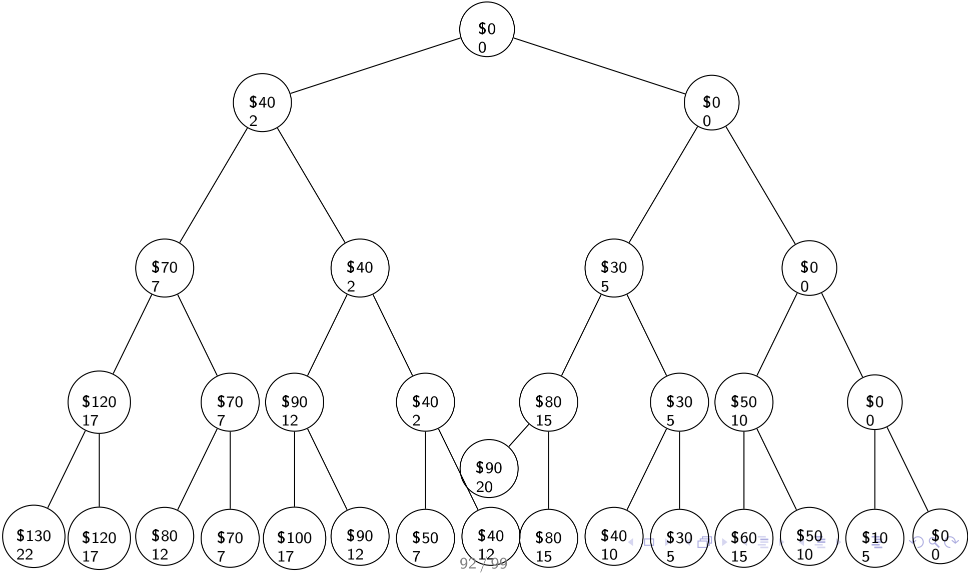
Time Complexity $= O(b^d)$

Space Complexity $= O(b^d)$



Time Complexity and Space Complexity of **Depth-First Search**

Time Complexity and Space Complexity of **Depth-First Search**

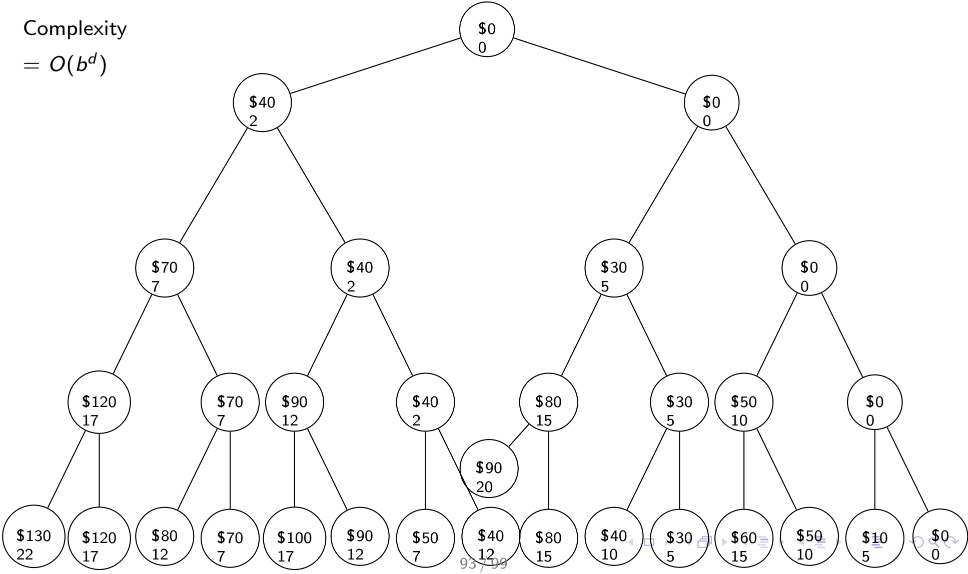


Time Complexity and Space Complexity of **Depth-First Search**

Time

Complexity

$= O(b^d)$

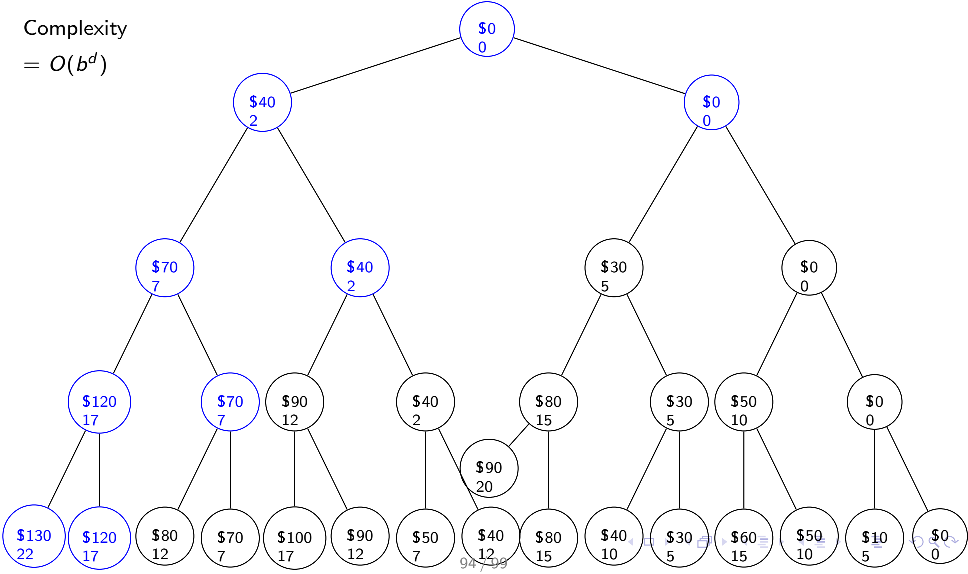


Time Complexity and Space Complexity of **Depth-First Search**

Time

Complexity

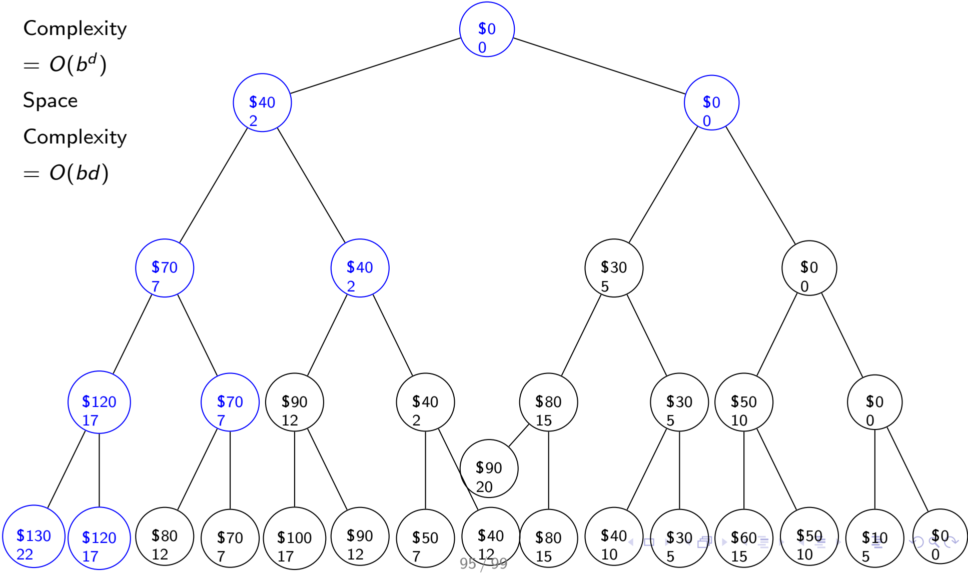
$= O(b^d)$



Time Complexity and Space Complexity of **Depth-First Search**

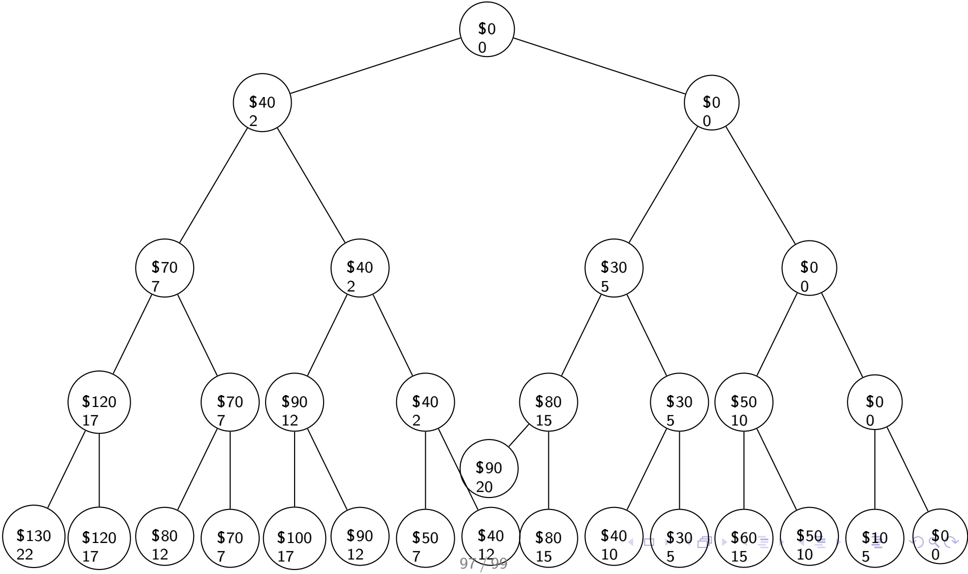
Time Complexity $= O(b^d)$

Space Complexity $= O(bd)$



Time Complexity and Space Complexity of **Best-First Search**

Time Complexity and Space Complexity of **Best-First Search**



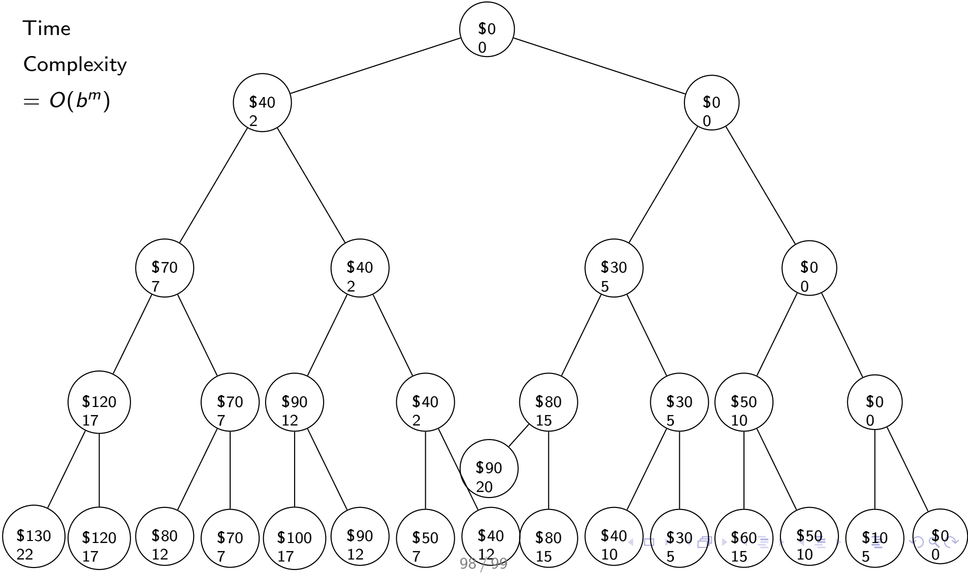
Time Complexity and Space Complexity of **Best-First Search**

Worst case

Time

Complexity

$= O(b^m)$



Time Complexity and Space Complexity of **Best-First Search**

Worst case

Time

Complexity

$= O(b^m)$

Worst case

Space

Complexity

$= O(b^m)$