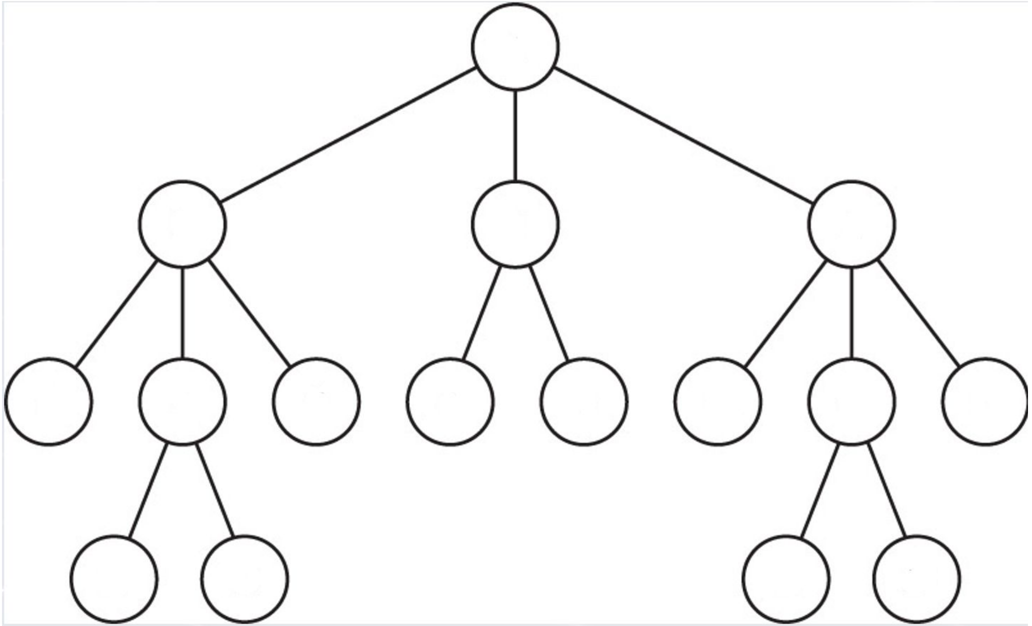
Lecture 20: Chapter 6 Part 1

Branch-and-Bound

CS3310

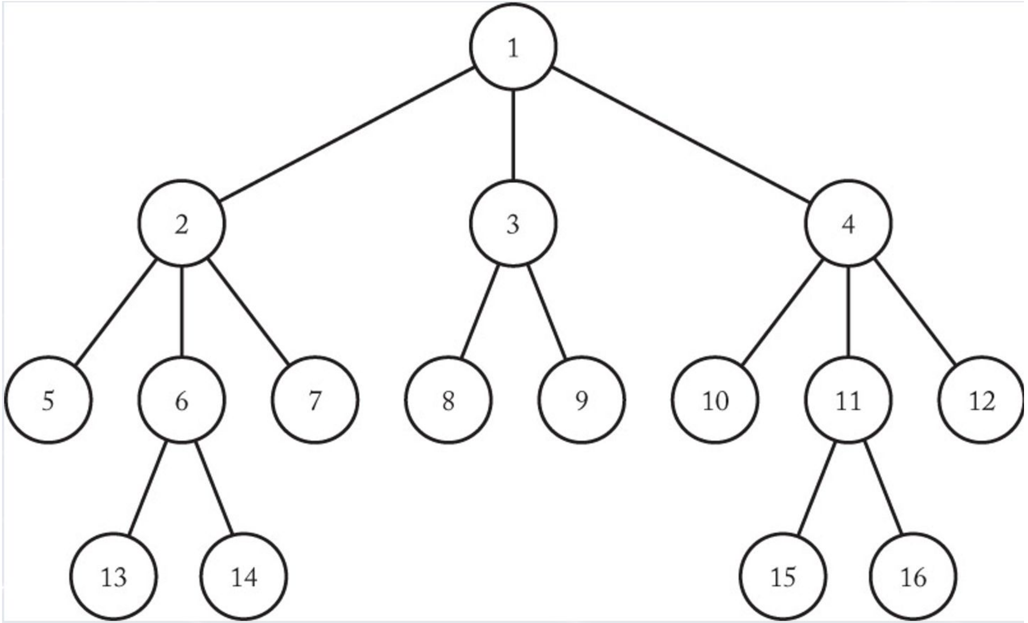
Breadth-first Search Review

- In which order does a breadth-first search visit the nodes of the following tree?



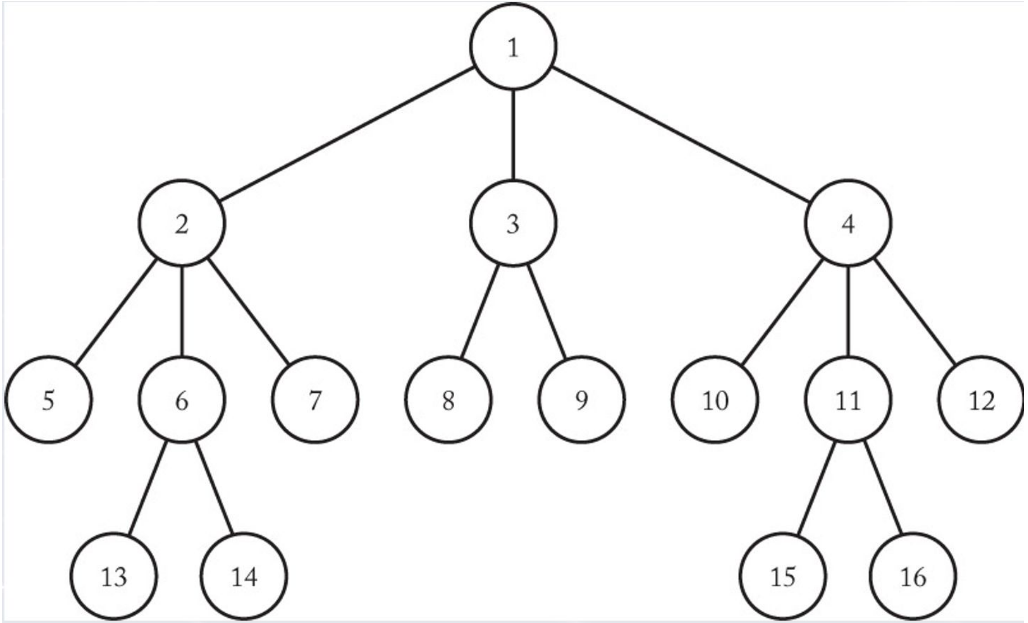
Breadth-first Search Review

- In a breadth-first search, we first visit the root.
- We then visit all nodes at level 1 before we move on to visit the nodes at level 2
- What type of data structure do we use in this kind of search?



Breadth-first Search Review

- In a breadth-first search, we first visit the root.
- We then visit all nodes at level 1 before we move on to visit the nodes at level 2
- What type of data structure do we use in this kind of search? A queue!



Breadth-first Search Review

In a breadth-first search, we start by adding the root to a queue.

- In a loop, we continually take the following steps until the queue is empty:
    - Dequeue the node at the front of the queue.
    - Visit and enqueue its children from left to right.

- i.e. After visiting the root, we visit all nodes at level 1 before we move on to visit the nodes at level 2, etc...

Breadth-first Search Review

```
void breadth_first_tree_search (tree T)
{
     queue_of_nodes Q;
     node u, v;
     initialize (Q);                  // Initialize the queue to be empty
     v = root of T;
     enqueue(Q, v);                   // Add the root to the queue
     while (!empty(Q))
     {
          dequeue(Q, v);              // Remove node at front of queue
          for (each child u of v)
          {
               visit u;               // visit that node's children and enqueue them
               enqueue(Q, u);
          }
     }
}
```

Branch-and-Bound

- We will discuss two branch-and-bound solutions for the 0-1 Knapsack problem.
  - The first is a simple version called breadth-first search with branch-and-bound pruning.
  - The second is a major improvement on the first and is called a **best-first search** with branch-and-bound pruning, which is the technique to use on homework #4

Breadth-First Search with Branch-and-Bound Pruning

- Suppose we have the following instance of the 0-1 Knapsack problem
  - **Note**: The items have already been sorted in nonincreasing order by p_i / w_i
- We proceed exactly as we did in the backtracking solution, except we perform a breadth-first search instead a depth-first one.

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

Breadth-First Search with Branch-and-Bound Pruning

- In the breadth-first search, we again determine `totweight` and `bound` for each node.
- Recall that *i* is the level we are currently on and therefore the most recent item we have either accepted or rejected.
  - i.e. at level 2 in the state space tree, we have either accepted or rejected item 2.
- We greedily take the remaining items until an item would take us over W.
- *k* represents the level of the item that would take the weight over W
  - i.e. if the 5th item would take us over W, then *k* = 5.
- `totweight` is the weight of the items we *can* greedily take after the current item
  - i.e. the current weight + the weights of items *j* through *k* - 1.

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

Breadth-First Search with Branch-and-Bound Pruning

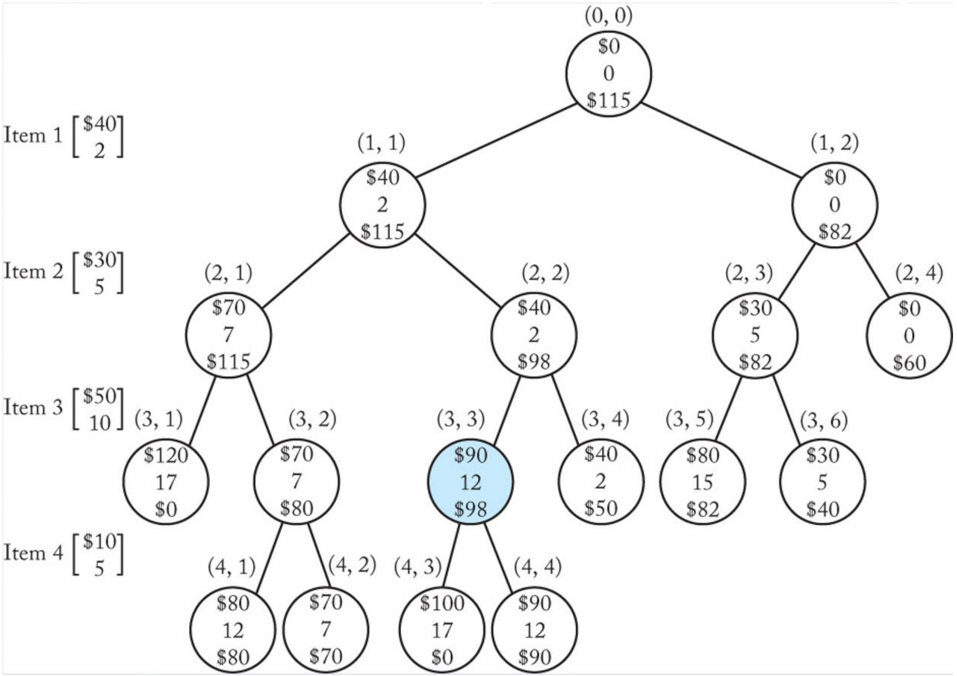
- The `bound` of a node is the current profit, plus the profit of the extra items we can fit in the bag, plus the profit of the fraction of the first item we can't take (item k).
- A node is nonpromising if its `bound` is less than or equal to `maxprofit` (the value of the best solution found up to that point)
  - A node is also nonpromising if `weight` >= W

$$bound = \underbrace{\left(profit + \sum_{j=i+1}^{k-1} p_j\right)}_{\text{Profit from first k-1 items taken}} + \underbrace{(W - totweight)}_{\text{Capacity available for kth item}} \times \underbrace{\frac{p_k}{w_k}}_{\text{Profit per unit weight for kth item}}$$

Breadth-First Search with Branch-and-Bound Pruning

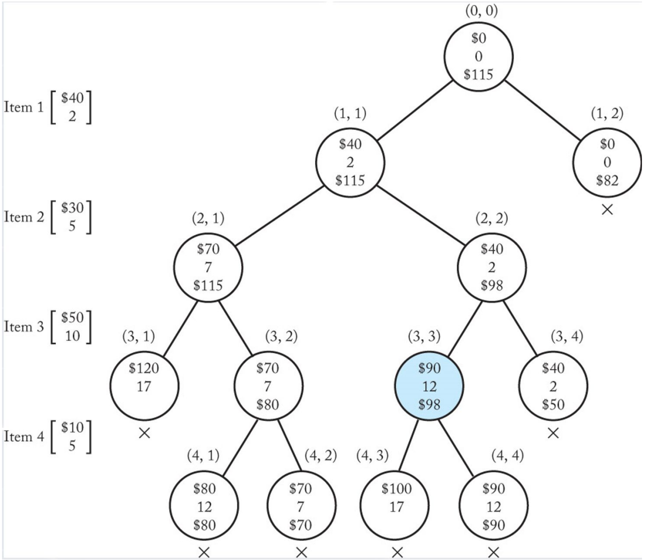
In this illustration of the pruned state-space tree, each node contains:

1. The profit of the items stolen up to that node
2. The weight of the items stolen up to that node
3. The bound on the total profit that could be obtained by expanding beyond the node.

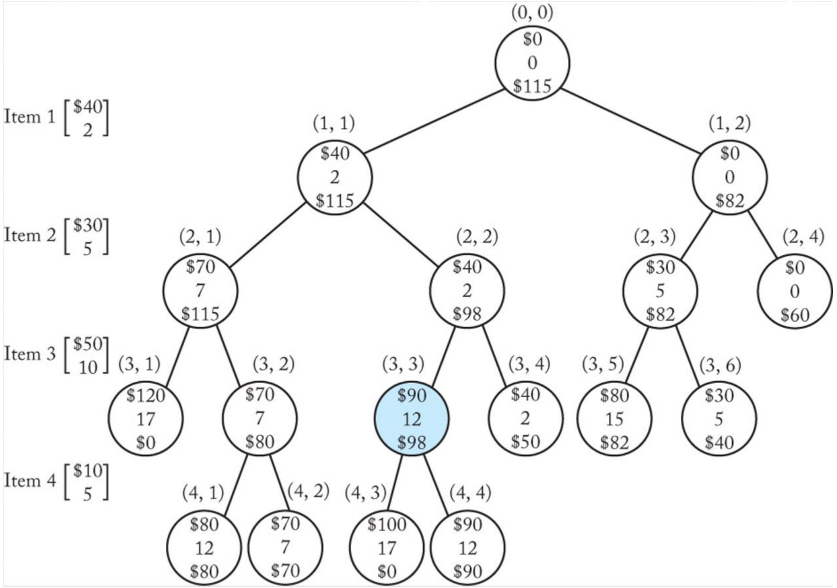


Breadth-First Search with Branch-and-Bound Pruning

Depth-First

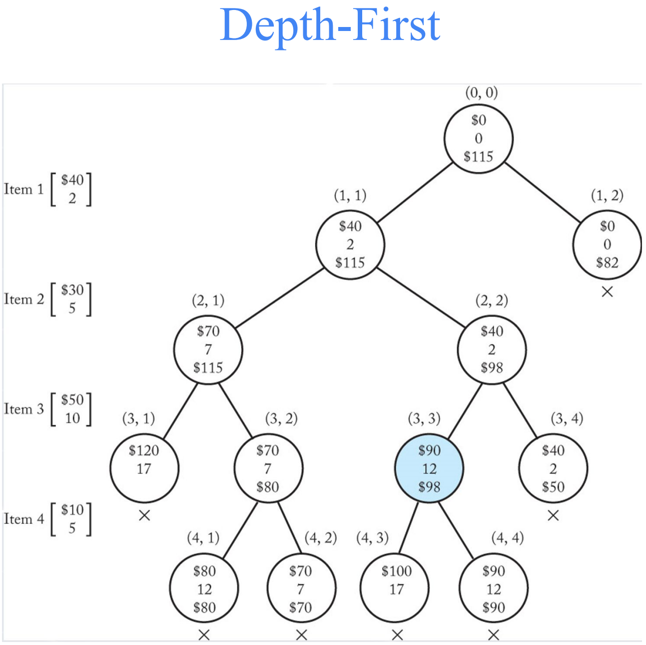


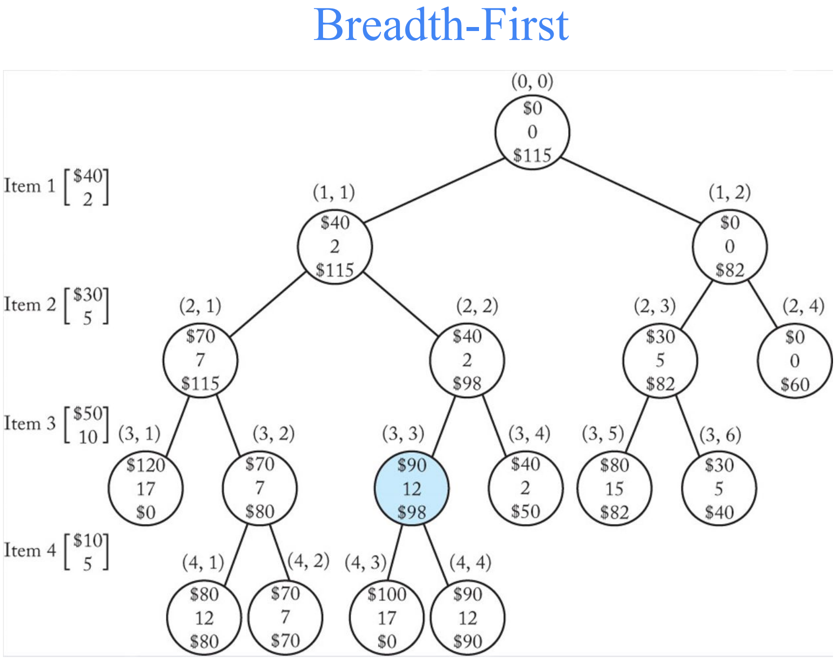
Breadth-First



In this example, a breadth-first search performs *worse* than a depth-first search. Why?

Breadth-First Search with Branch-and-Bound Pruning





- When we visit node (1, 2), `maxprofit` is $90 in the depth-first search. We can backtrack since the node's bound is $82
- In the breadth-first search, maxprofit is $40 when we visit this node, so we continue

Breadth-First Search with Branch-and-Bound Pruning

- With a backtracking algorithm, we implicitly created the state space tree by setting values in an array.
- For a breadth-first search, we implicitly create the state space tree by placing `node` objects in a queue.
  - Once a `node` is removed from the queue, it can be deleted.
- Each object stores info about a visited node:

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

- **Note**: Do NOT use the following algorithm for homework #4!

Breadth-First Search with Branch-and-Bound Pruning

```
void knapsack (int n, const int p[], const int w[], int W, int &maxprofit)
     queue_of_node Q;   node u, v; initialize(Q);
     v.level = 0; v.profit = 0; v.weight = 0;      // initialize v to the root

     maxprofit = 0;
     enqueue(Q, v);                                // place v in the queue
     while (!empty(Q))
          dequeue(Q, v);
          u.level = v.level + 1;                   // set u to be the left child of v
          u.weight = v.weight + w[u.level];        // (i.e. try taking the next item)
          u.profit = v.profit + p[u.level];

          if (u.weight <= W && u.profit > maxprofit)
               maxprofit = u.profit;               // taking u gives us the best profit so far
          if (bound (u) > maxprofit)
               enqueue(Q, u);                      // if u is promising, add to queue
          u.weight = v.weight;
          u.profit = v.profit;                     // set u to the right child of v
          if (bound(u) > maxprofit)                // (i.e. try rejecting the next item
               enqueue(Q, u);
```

Breadth-First Search with Branch-and-Bound Pruning

```
float bound (node u)
      index j, k;
      int totweight;
      float result;
      if (u.weight >= W)
            return 0;                                // u is nonpromising. Return 0 for its bound
      else
            result = u.profit;
            j = u.level + 1;                              // j = the first unconsidered item
            totweight = u.weight;
            while (j <= n && totweight + w[j] <= W)  // greedily grab as many items as possible
                  totweight = totweight + w[j];
                  result = result + p[j];
                  j++;
      k = j;
      if (k <= n)
            result = result + (W - totweight) * p[k] / w[k];
      return result;
```

- `bound` is similar to `promising` from the backtracking algorithm, except it returns the calculated bound of node *u* rather than true or false.

Best-First Search with Branch-and-Bound Pruning

- We can make significant improvements on the breadth-first technique by performing a **best-first search**.
  - With this strategy, we do more with `bound` than just determine whether a node is promising.
- After we visit a node's children, we look at the promising nodes whose children we have not yet visited. We expand beyond the one with the *best* bound.
- What type of data structure can we use to accomplish this?

Best-First Search with Branch-and-Bound Pruning

- We can make significant improvements on the breadth-first technique by performing a **best-first search**.
    - With this strategy, we do more with `bound` than just determine whether a node is promising.
- After we visit a node's children, we look at the promising nodes whose children we have not yet visited. We expand beyond the one with the *best* bound.
- What type of data structure can we use to accomplish this?
    - A **priority queue** with the entries sorted in nondecreasing order by bound.

Best-First Search with Branch-and-Bound Pruning

- For a best-first search, each node stores its `bound` since it might be promising when we add it to the priority queue but nonpromising when we remove it.
    - Rather than calculate a node's bound twice, we calculate it and store the result when we *visit* the node.
- When we remove a node from the priority queue, we check the previously calculated `bound` against `maxprofit` to see if we should visit its children
    - i.e. is it still promising?

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

Best-First Search with Branch-and-Bound Pruning Init

How do we initialize this problem?

- **Note**: We will be using the same set from the backtracking example.
- For the sake of space, w and p are not printed on each slide:

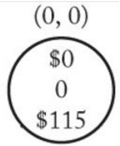
$w = \{ 2, 5, 10, 5 \}$ $p = \{ 40, 30, 50, 10 \}$

W = 16 *maxprofit* = PQ = { }

Best-First Search with Branch-and-Bound Pruning Init

How do we initialize this problem?

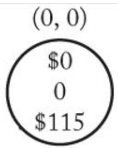
- Visit the dummy node, (0, 0).
  - v.level = 0;
  - v.profit = \$0;
  - v.weight = 0;
- Initialize `maxprofit` to \$0, compute `v.bound` to be \$115 and insert node (0, 0) to PQ

W = 16 *maxprofit* = \$0 PQ = {  }

Best-First Search with Branch-and-Bound Pruning Step 1

- Remove node from front of priority queue, assign it to *v*:
- Visit its left child first.

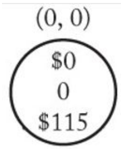
v:



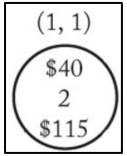
W = 16 *maxprofit* = \$0 PQ = { }

Best-First Search with Branch-and-Bound Pruning Step 1

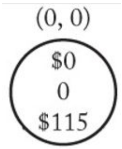
- Remove node from front of priority queue, assign it to *v*:

v: 

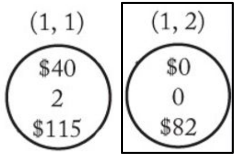
- Visit its left child first.
  - `u.level = v.level + 1`= 1;
  - `u.profit = v.profit + p[u.level]`= $0 + $40 = $40;
  - `u.weight = v.weight + w[u.level]`= 0 + 2 = 2;
- Since `u.weight < 16` and `u.profit > $0`, update `maxprofit` to $40.
- `u.bound` is computed to be $115, which is promising.
- Insert *u* (node (1, 1)) to PQ

W = 16 *maxprofit* = $40 PQ = {  }

Best-First Search with Branch-and-Bound Pruning Step 2

v: 

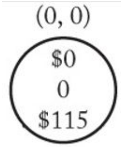
- v still has an unvisited child
- Visit its right child, (1, 2)
  - `u.level = v.level + 1` = 1
  - `u.profit = v.profit` = \$0
  - `u.weight = v.weight` = 0
- Its bound is computed to be \$82, which is promising.
- Insert u (node (1, 2)) to PQ
- We have visited all of (0, 0)'s children. What's next?

W = 16 *maxprofit* = \$40 PQ = {  }

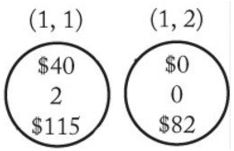
Best-First Search with Branch-and-Bound Pruning Step 2

- What's next?
  - v has no more unvisited children.
  - We next determine the promising, unexpanded node with the greatest bound.
    - i.e. the node at the front of PQ!

v:



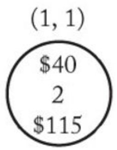
W = 16 *maxprofit* = $40 PQ = {



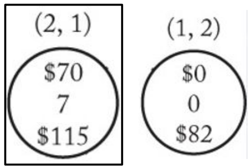
}

Best-First Search with Branch-and-Bound Pruning Step 3

- Remove node from front of priority queue, assign it to *v*:

v: 

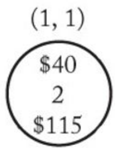
- *v* has two unvisited children. Visit its left child (2, 1)
  - `u.level = v.level + 1` = 2
  - `u.profit = v.profit + p[u.level]` = \$40 + \$30 = \$70
  - `u.weight = v.weight + w[u.level]` = 2 + 5 = 7
- Since `u.weight` < 16 and `u.profit` > \$40, update `maxprofit` to \$70.
- `u.bound` is computed to be \$115, which is promising.
- Insert *u* (node (2, 1)) to PQ.
  - It goes in the front since its bound is higher than that of (1, 2)

W = 16 *maxprofit* = \$70 PQ = {  }

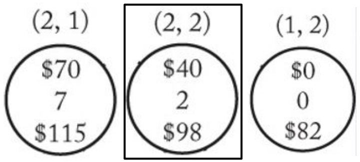
Best-First Search with Branch-and-Bound Pruning Step 4

- *v* still has an unvisited child
- Visit its right child (2, 2)
  - `u.level = v.level + 1`= 2
  - `u.profit = v.profit`= $40
  - `u.weight = v.weight`= 2
- `u.bound` is computed to be $98, which is greater than `maxprofit`
- Insert *u* (node (2, 2)) to PQ.
- We next determine the promising, unexpanded node with the greatest bound.
  - Again, this is the node at the front of the priority queue!

v:



W = 16   *maxprofit* = $70   PQ = {

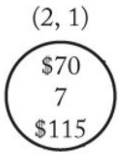


}

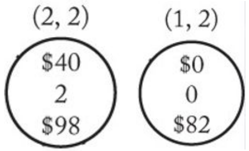
Best-First Search with Branch-and-Bound Pruning Step 5

- Remove node from front of PQ, assign it to *v*:
- *v* has two unvisited children. Visit its left child (3, 1)
  - `u.level = v.level + 1` = 3
  - `u.profit = v.profit + p[u.level]` = \$70 + \$50 = \$120
  - `u.weight = v.weight + w[u.level]` = 7 + 10 = 17
- (3, 1) is nonpromising because its weight 17 is greater than 16.
  - The `bound` method indicates this by returning 0.
- Since its bound of 0 is less than `maxprofit` we do not add it to PQ

v:



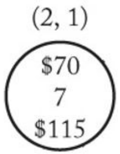
W = 16 *maxprofit* = \$70 PQ = {



}

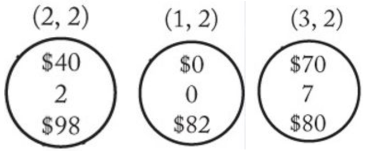
Best-First Search with Branch-and-Bound Pruning Step 6

v:



- *v* still has an unvisited child
- Visit its right child (3, 2)
  - `u.level = v.level + 1` = 3
  - `u.profit = v.profit` = $70
  - `u.weight = v.weight` = 7
- `u.bound` is $80 which is greater than `maxprofit`. Add (3, 2) to PQ
- The node at the front of the priority queue is the promising, unexpanded node with the greatest bound.

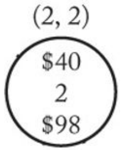
W = 16 *maxprofit* = $70 PQ = {



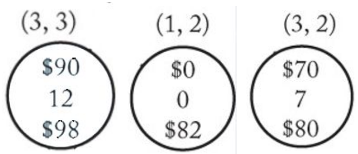
}

Best-First Search with Branch-and-Bound Pruning Step 7

- Remove node from front of PQ, assign it to *v*:

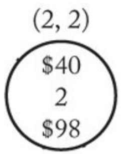
v: 

- *v* has two unvisited children. Visit its left child (3, 3)
  - `u.level = v.level + 1` = 3
  - `u.profit = v.profit + p[u.level]` = \$40 + \$50 = \$90
  - `u.weight = v.weight + w[u.level]` = 2 + 10 = 12
- `u.weight` < 16 and `u.profit` > `maxprofit`, which we update to \$90
- Calculate `u.bound` to be \$98.
  - Add it to PQ (it goes in the front)

W = 16 *maxprofit* = \$90 PQ = {  }

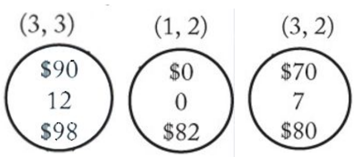
Best-First Search with Branch-and-Bound Pruning Step 8

v:



- *v* still has an unvisited child
- Visit its right child (3, 4)
  - `u.level = v.level + 1` = 3
  - `u.profit = v.profit` = $40
  - `u.weight = v.weight` = 2
- Calculate its bound to be $50, which is less than `maxprofit`. Do not add to PQ.
- The node at the front of the priority queue is the promising, unexpanded node with the greatest bound.

W = 16 *maxprofit* = $90 PQ = {

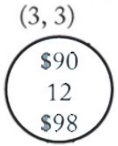


}

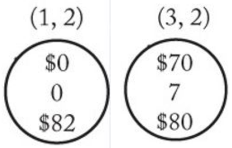
Best-First Search with Branch-and-Bound Pruning Step 9

- Remove node from front of PQ, assign it to *v*:
- Visit its left child (4, 1)
  - `u.level = v.level + 1` = 4
  - `u.profit = v.profit+ p[u.level]` = \$90 + \$10 = \$100
  - `u.weight = v.weight + w[u.level]` = 12 + 5 = 17
- Since `u.weight > W`, set bound to \$0, which is less than \$90. Do not add to PQ

v:

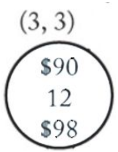


W = 16 *maxprofit* = \$90 PQ = {

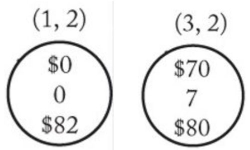


}

Best-First Search with Branch-and-Bound Pruning Step 10

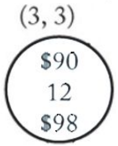
v: 

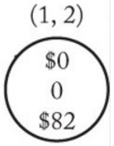
- *v* still has an unvisited child
- Visit its right child (4, 2)
  - `u.level = v.level + 1`= 4
  - `u.profit = v.profit`= $90 = $90
  - `u.weight = v.weight`= 12
- Compute bound to be $90, which is not greater than `maxprofit`. Do not add to PQ.
- What’s next?

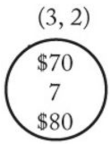
W = 16 *maxprofit* = $90 PQ = {  }

Best-First Search with Branch-and-Bound Pruning Step 10

- *v* still has an unvisited child
- Visit its right child (4, 2)
  - `u.level = v.level + 1`= 4
  - `u.profit = v.profit`= \$90 = \$90
  - `u.weight = v.weight`= 12
- Compute bound to be \$90, which is not greater than `maxprofit`. Do not add to PQ.
- What's next? Both nodes in PQ have bounds less than `maxprofit`. Remove them and we are done!

v: 

W = 16 *maxprofit* = \$90 PQ = { 

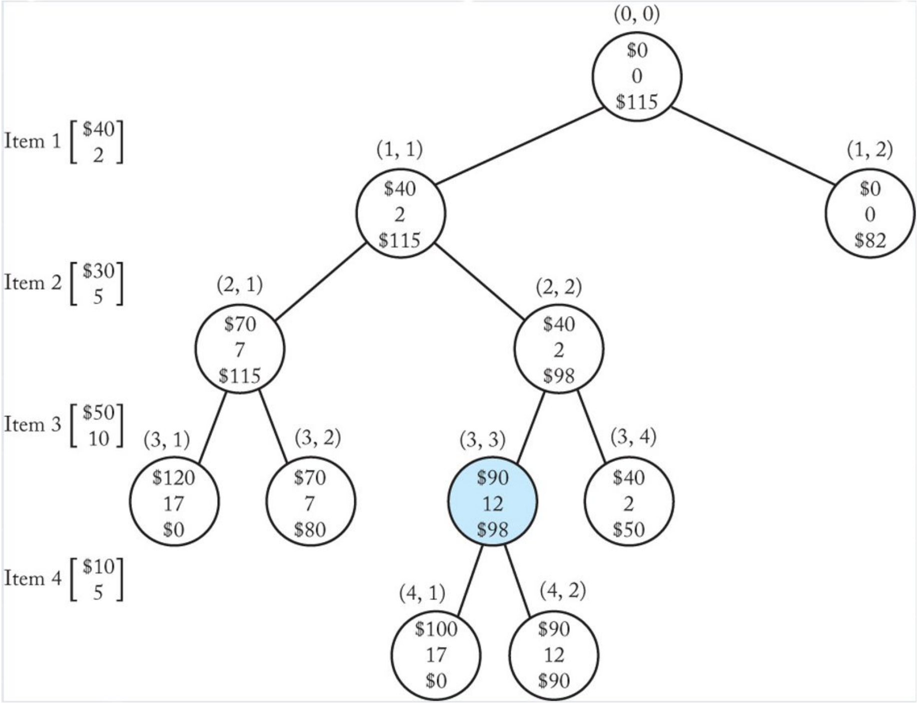


}

Best-First Search with Branch-and-Bound Pruning

Using best-first search, we have checked only 11 nodes, vs. 13 with a depth-first search.

A savings of 2 nodes isn't much, but savings can be significant as the problem set grows.



Best-First Search with Branch-and-Bound Pruning

- The best-first search algorithm is very similar to the breadth-first search, except we use a priority queue rather than a regular queue.
  - **Note**: the nodes must be sorted in nondecreasing order by `p[i] / w[i]`
- To begin, initialize variables and insert the root of the tree in the priority queue.
  - **Note**: Use *this* algorithm for homework #4!

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

Best-First Search with Branch-and-Bound Pruning

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

In each iteration of the loop we visit both children of the node at the front of the priority queue. Those with a higher bound than `maxprofit` are added to PQ.

In-Class Exercise

Use the best-first search with branch-and-bound pruning algorithm on the following instance. Draw the state-space tree of nodes searched and the contents of PQ after each step. W = 13

i	p_i	w_i	p_i / w_i
1	\$20	2	10
2	\$30	5	6
3	\$35	7	5
4	\$12	3	4
5	\$3	1	3