Branch and Bound

Week 14

Required Activities

- Check Announcements regularly (every 2-3 days)
- Read FA book: Chapter 6
- Start working on Assignment 5

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 used only for optimization problems
- Bound (number) is computed at a node to determine whether the node is promising
- Calculate Bound for a node
  - Bound best possible solution obtainable by expanding this node (minimum or maximum value)
  - Choose technique for calculating bound that produces best solution
  - If bound is no better than the value of the best solution found so far, node is non-promising (do not expand the node)
  - Otherwise, the node is promising
- Like backtracking, algorithms have typically exponential time complexity but can be efficient for large instances

Breadth-First Tree 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

- Algorithm:

```
void beadth_first_tree_search(Tree t)
{
        Queue Q;
        Node u, v;
        initialize(Q); //initialize queue to be empty
        r = root of T;
        visit v;
        enqueue(Q, v);
        while (!empty(Q))
        {
                dequeue(Q, v);
                for (each child u of v)
                {
                        visit u;
                        enqueue(Q, u)
                }
        }

}
```

Breadth-First with Bound

```
Node beadth_first_branch_and_bound(Tree t)
{
		Queue Q;
		Node u, v;
		int best;
		Q.initialize(); //initialize queue to be empty
		v = root of T;
		Q.enQueue(v);
		best = v.getValue()
		while (! Q.empty())
		{
				v = Q.deQueue();
				for (each child u of v)
				{
						if (u.getValue() > best)
								best = u.getValue()
						If (bound(u) > best)
								Q.enQueue(u)
				}
		}
}
```

0-1 Knapsack Problem

- Weight and profit at node are total weight and total profit of the items that have been included up to a node (same as in backtracking)
- Determine if a node is promising
  - If node’s weight is >= W OR **bound is <= maxprofit** (the value of the **best solution** found up to that point) then non-promising
  - Otherwise
    - Initialize *totweight* to item’s weight and *bound* to item’s profit
    - Loop to add items weight to *totweight* and items profit to *bound* until item’s weight brings *totweight* above W
    - Grab the fraction of the item allowed by available weight and add profit of the fraction to bound
  - *Bound is calculated same as in backtracking*
- Traverse next to node with **bound greater** than current *maxprofit*

Best-First with Bound

```
Node beadth_first_branch_and_bound(Tree t)
{
        PriorityQueue PQ;
        Node u, v;
        int best;
        PQ.initialize(); //initialize priority queue to be empty
        v = root of T;
        PQ.insert(v);
        best = v.getValue()
        while (! PQ.empty())
        {
                v = PQ.remove();   // removes the highest valued item
                If (bound(v) > best)  {
                        for (each child u of v)
                        {
                                if (u.getValue() > best)
                                        best = u.getValue()
                                If (bound(u) > best)
                                        PQ.insert(u)
                        }
                }
        }
}
```

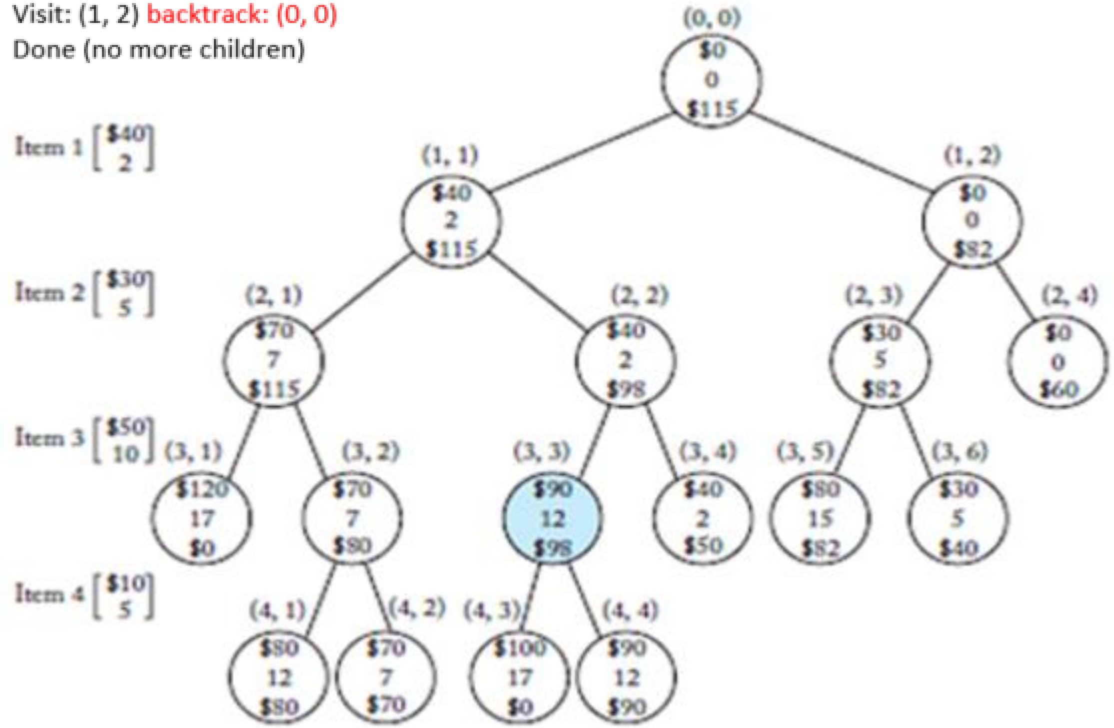
Best-First Search with Branch-and-Bound pruning

- Can obtain solution quicker than breath-first-search or depth-first-search
- May not obtain optimum solution
- Uses bound to improve search
  - After visiting all of the children of a given node, expand the one with the **best** bound
  - Determine promising unexpanded node with greatest bound
  - Order determined by best bound rather than pre-determined
  - Uses priority queue to hold the nodes of state space

Fig 6.2 (A6.1 breath) and Fig 6.3 (A6.2 best)

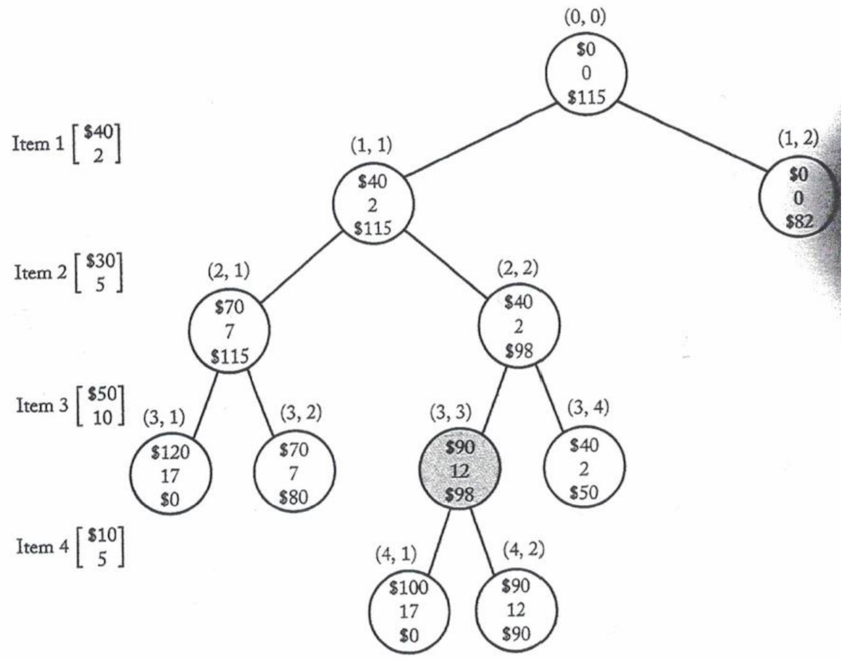
Backtracking Algorithm 5.7:

Visit: (0, 0), (1, 1), (2, 1), (3, 1) backtrack: (2, 1)
Visit: (3, 2), (4, 1) backtrack: (3, 2)
Visit: (4, 2) backtrack: (1, 1)
Visit: (2, 2), (3, 3), (4, 3) backtrack: (3, 3)
Visit: (4, 4) backtrack: (2, 2)
Visit: (3, 4) backtrack: (0, 0)
Visit: (1, 2) backtrack: (0, 0)
Done (no more children)



Bound Algorithm 6.2:

Visit: (0, 0), (1, 1), (1, 2) then determine best bond (1, 1) so go to children
Visit: (2, 1), (2, 2) then best bond (2, 1)
Visit: (3, 1), (3, 2) then best bound (2, 2)
Visit: (3, 3), (3, 4) then best bound (3, 3)
Visit: (4, 1), (4, 2)
Done (no more promising unexpanded nodes)



Questions ?

- Post in the discussions
- Send email to RMcFadden@HarrisburgU.edu
- Respond usually within 48hours