# **Chapter 6: Branch and Bound Algorithms**

## **Introduction**

**The branch and bound approach** to solving problems is **similar to the backtracking** approach.  Both **use the idea of a tree for exploring the possible solutions** to a problem.  The **main differences** between branch and bound and backtracking are:

1. Branch and bound is **only used for optimization problems** (i.e., problems in which the solution can be represented as a number, with some way of picking the best out of two numbers)
2. Branch and bound **does not use a depth first search** through the tree

The **full name for branch and bound** is **best-first search with branch-and-bound pruning**.  Pruning we already understand from our look at backtracking algorithms last week.  Recall that in backtracking, we used **a depth-first search**, and now in branch and bound we're using a **best-first**search.  **What is a best-first search?**

To understand that, let's look at a slightly simpler form, called **breadth-first search with branch-and-bound pruning**.  Recall from when we covered trees that a breadth first search visits nodes from top to bottom and left to right.

**A breadth-first search version of branch and bound** is almost identical in concept to backtracking.  Except instead of a depth-first search, we perform a breadth-first search.  We **eliminate nonpromising branches** **using the same rules** as in a backtracking algorithm.

## **Breadth-First 0-1 Knapsack Problem**

As an example, let's go back to the 0-1 Knapsack Problem.  This is an optimization problem, so we can use branch and bound for it.  Remember, what we are going to look at right now is **not the best-first search**, but the **breadth-first search**.

We draw the tree in the same way as for the backtracking algorithm.  Let's draw part of the tree for three items, with profits of **$40, $30, $50** and weights of **2, 5, 10**.  We can indicate in the node itself the **total profit** so far and the **weight of items** in the knapsack.

**When doing a breadth-first search, we place child items into a queue for later processing if they are promising.  If they are non-promising, we leave them out of the queue, so that branch will not be processed.**

Page 255 in your book has the **algorithm for the breadth-first approach to the 0-1 Knapsack Problem.**Page 256 shows the bound function, which is same as function promising in Algorithm 5.7.

The one basic difference between this algorithm and the backtracking algorithm is that **we calculate an integer value** for each node that represents the maximum possible profit if we pick that node.  **If that maximum possible profit is not greater than the best total profit we've found so far, we do not bother going down that branch.**  **If the total weight exceeds W then the profit for that solution is 0 (it is an invalid solution).**   Otherwise, the maximum possible profit we could get from picking a node is that **returned by the Fractional Knapsack algorithm.**

So far, this is **no improvement over the backtracking algorithm** (in fact, we're using the same rules and covering the same portions of the tree).  Let's look at page 253 in the book to see the tree covered for a certain set of items.

## **Best-First Branch and Bound**

What makes **branch and bound** more efficient than backtracking is that, instead of traversing the tree in a predetermined order (depth-first, breadth-first, etc.), **we traverse it based on the optimization criteria for the problem.**

For example, in the 0-1 Knapsack Problem we want to maximize **profit**.  So, we pick a child to process based on **which one has the highest maximum possible profit**.  This allows us to quickly determine a pretty good solution.  **That pretty good solution can be used to eliminate inferior solutions more quickly than in backtracking or breadth-first searching.**

It's important to realize that all the children yet to process are included in the set of possible next nodes.  We pick the one with the highest maximum possible profit, so we may be jumping around the tree quite a bit.  **When we process a node, if the node is promising, then we immediately add its children to the set of possible next nodes.**

Page 258 in your book has the tree covered for the same items we looked at previously.   Let's see how it traverses the tree. You can see that fewer nodes are actually visited, because we quickly found a pretty good solution that helped to eliminate other branches.  This is what makes **branch and bound** an effective problem solving technique for optimization problems.

Note that **branch and bound** has the same problem that backtracking had.  You cannot know for any particular set of data whether **branch and bound** will be efficient or not.   However, over a large number of times with random data, you can determine whether **branch and bound** is efficient for a particular problem.

So, how do we write a best-first search?  Recall that for a breadth-first search, we put nodes into a queue.  **For a best-first search, we put items into a data structure called a priority queue.**  **In a priority queue, you can put items into the queue in any order, but you always remove the one with the maximum value first**.  In this way, we put the children of a node into the priority queue, and when we pull them out, **we process the one with the maximum possible profit first.**

So how do we write a priority queue?  Usually it's implemented as a heap, which we discussed earlier. Now that we know how the best-first search works for the 0-1 Knapsack Problem, so please work through Example 6.2 shown in page 257 of your textbook.

## **Traveling Salesperson Problem**

Let's look at another optimization problem and see how branch and bound works.   Recall that the traveling salesperson problem was to determine the shortest path from one vertex in a graph that went through every other vertex in the graph exactly once, ending up back at the starting vertex.

We can visualize a tree for this by making one child of the root for each possible vertex after the starting vertex.  In the worst case, a complete graph, this will be N children.  Each child will then have N-1 children, representing the next vertex taken.  For a complete graph, the tree would have N! nodes.  But remember, the idea with branch and bound is to traverse only a small number of those nodes.

Let's look at a portion of the tree that is shown on page 264 in your book, for the graph shown on page 263 (draw just the root and its immediate children).

For branch and bound algorithms, we need to calculate an integer value for a node that represents the best possible solution we think we can get if we go down that branch.   But we need to calculate this value without actually going down the branch.   The method used in the traveling salesperson problem is to add up the minimum edge out of every remaining vertex.  Remember, we're trying to optimize for the shortest tour.  If everything worked out exactly right, we still could not get a tour shorter than if we used the minimum edge leaving every vertex.

Let's calculate the bounds for the various children of the root.  Now we know which branch we want to expand further...the one with the minimum value (we're optimizing for the shortest path).  You can see on page 267 in your book the nodes which are checked.  Let's walk through this example to see the order in which nodes are checked.

Branch and bound has the potential for solving the traveling salesperson problem efficiently.  Unfortunately, we can't know how quickly it will solve any set of data until we try.  **In the absolute worst case, the branch and bound algorithm for the traveling salesperson problem is O(n!).**  Most of the time, though, **it performs better than the dynamic programming algorithm** for the same problem.

Now please walk through using this algorithm to solve the traveling salesperson problem for the exercise in your book on page 281, number 7.