7 Branch and Bound

7.1 Preliminaries

So far we have covered Dynamic Programming, Greedy Algorithms, and (some) Graph Algorithms. Other than a few examples with Knapsack and Travelling Salesman, however, we have mostly covered **P** algorithms. Now we turn to look at some **NP** algorithms. Recognize, that, we are not going to come up with any **P** algorithms for these problems but we will be able to radically improve runtime over a naïve algorithm.

Specifically we are going to discuss a technique called Branch and Bound. Let's first understand the intuition behind the technique. Assume we are trying to maximize a function f over a exponentially large set X. However, not only is the set exponentially large but f might be computationally intensive on this set. Fortunately, we know of a function h such that $f \leq h$ everywhere.

Our naïve strategy is to keep a maximum value m (initially at $-\infty$) and for each $x \in X$, update it by $m \leftarrow \max\{m, f(x)\}$. However, an alternative strategy would be to calculate $h(x)$ first. If $h(x) \leq m$ then we know $f(x) \leq h(x) \leq m$ so the maximum will not change. So we can effectively avoid computing $f(x)$ saving us a lot on computation! However, if $h(x) > m$ then we cannot tell anything about $f(x)$ so we would have to compute $f(x)$ to check the maximum. So, in the worst case, our algorithm could take the same time as the naïve algorithm. But if we have selected a function h that is a tight bound (i.e. $h - f$ is very small) then we can save a lot of computational time.

Note: this is not Branch and Bound; this is only the intuition behind the technique. Branch and Bound requires more structure but has a very similar ring to it.

7.2 Knapsack, an example

Now that we've gone over some broader intuition, let's try an example and then come back for the formalism. Let's consider the Knapsack problem. We can rewrite Knapsack as a n-level decision problem, where at each level i, we choose whether to include item i in our bag or not. Figure 7.1 gives the intuition for this decision tree. As drawn, the left hand decision matches choosing the item and the right hand decision matches leaving the item.

In the end, we are left with 2^n nodes, each representing a unique choice of items and its respective weight and value. We are then left with maximizing over all choices (leaves) that satisfying our weight constraint W.

However, this wasn't efficient! We spent a lot of time traversing parts of the tree that we knew had surpassed the weight constraint. So, a smarter strategy would be to truncate the tree at any point where the weight up to that point has passed the weight constraint. We see this in Figure 7.2. Now, the problem is a lot faster as we've seriously reduced our runtime. Notice though, we cannot guarantee any truncations of the graph, so the asymptotic runtime of this problem is still $O(2^n)$ as we have 2^n possibilities for item sets.

7.3 Formalism

Let's formalize the intuition we built from Knapsack and generate a rigorous structure which we can use to solve other Branch and Bound problems. These are the qualities we are looking for in a Branch and Bound problem.

Properties of Branch and Bound Algorithm.

1. The problem should be expressible as a maximization of a function $f : L \to \mathbb{R}$ where L is the set of leaves of some tree T.
2. We can define a function $h : T \to \mathbb{R}$ defined on all nodes of the tree such that $f(\ell) \leq h(t)$ if ℓ is a descendant leaf of t. (Here t is any node in the graph. Note ℓ and t could be the same).

Note we can also write minimization problems in this manner by considering $(-f)$. So, for the rest of this lecture, I am only going to discuss maximization problems without loss of generality. This gives us a natural generic algorithm for solving a Branch and Bound problem.

Algorithm 7.1 (Branch and Bound Algorithm)**.** For any problem,

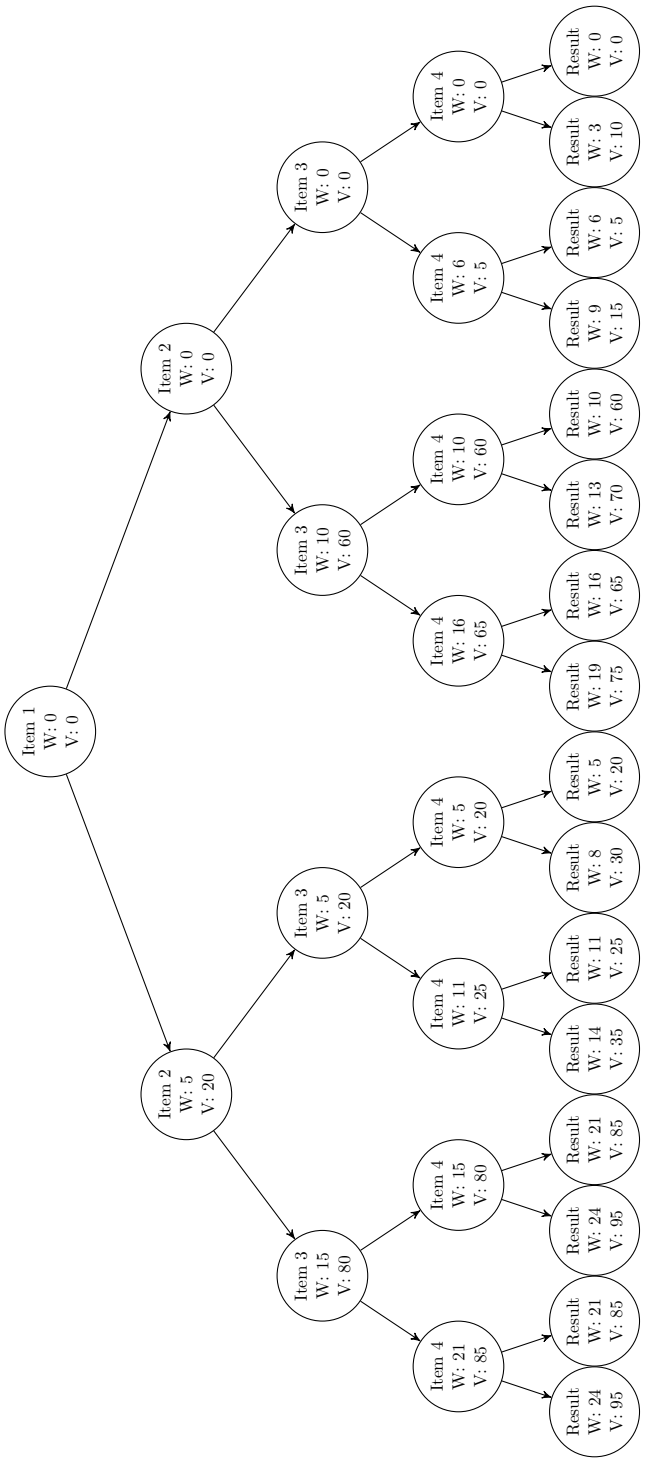

Figure 7.1: A graph illustration of the Knapsack decision problem. Assume 4 items with weights 5, 10, 6, 3, and values, 20, 60, 5, 10, respectively. Set the weight constraint to be 10. Here all paths are considered. Next figure shows the truncation we can do.

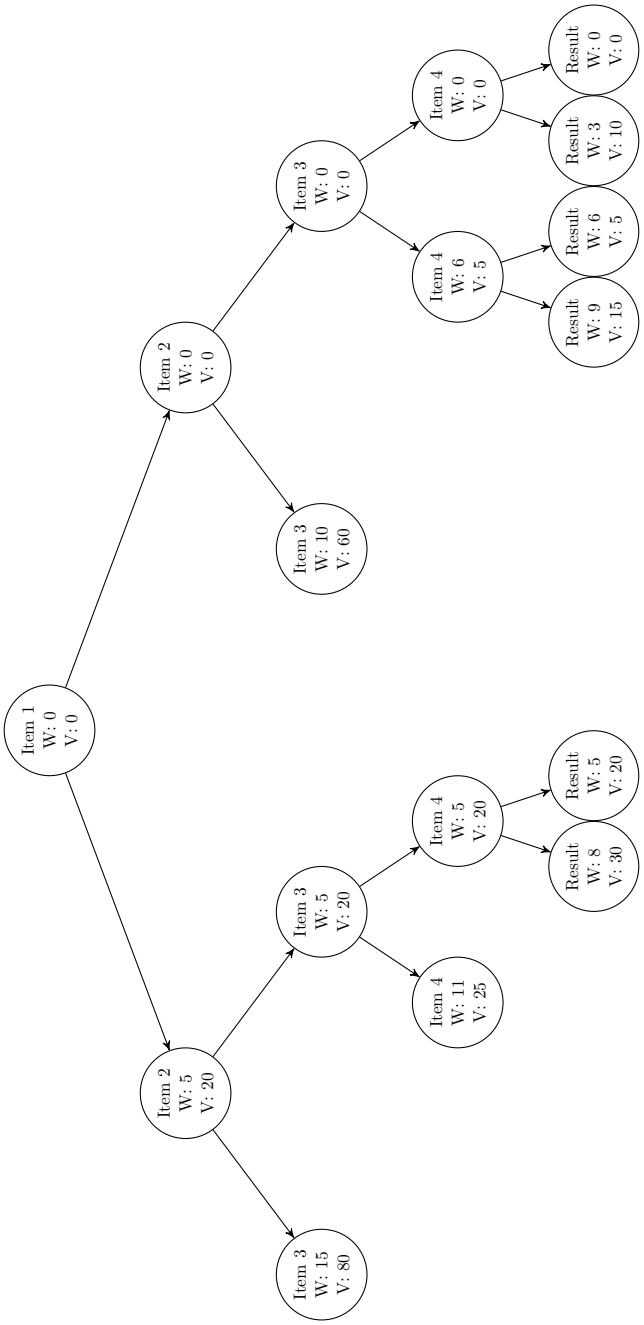

Figure 7.2: A graph illustration of the Knapsack decision problem. Assume 4 items with weights 5, 10, 6, 3, and values, 20, 60, 5, 10, respectively. Set the weight constraint to be 9. Here we truncate unfeasible paths early.

1. Write the problem as a maximization of $f : L \to \mathbb{R}$ where L is the set of leaves of a tree T.
2. Let $m \leftarrow -\infty$ and $x \leftarrow$ null. This is the current maximum value achieved and the leaf achieving it.
3. Beginning at the root r of T, traverse the tree in pre-order (i.e. run a calculation at node t, the traverse recursive each of its children). For every node t encountered do the following:
   (a) If t isn't a leaf, check if $h(t) < m$. If so, then truncate the traversal at t (i.e. don't consider any of t's descendants)
   (b) If t is a leaf, calculate $f(t)$. If $f(t) < m$, $m \leftarrow f(t)$ and $x \leftarrow t$ (i.e. update the maximum terms)
4. Return x.

Algorithm Correctness. The naïve algorithm would run by traversing the tree and updating the maximum at each leaf ℓ and then returning the maximum. The only difference we make is, we claim we can ignore all the descendants of a node t if $h(t) < m$. Why? For any descendant ℓ of t by the property above $f(\ell) \leq h(t)$. As $h(t) < m$ then $f(\ell) < m$ as well. Therefore, the maximum will never update by considering ℓ. As this is true for any descendant ℓ, we can ignore its entire set of descendants.

7.4 Formalizing Knapsack

Let's apply this formalism concretely to the Knapsack problem. We've already seen the natural tree structure. So let's define the functions f and h. Every leaf L can be expressed as a vector in $\{0,1\}^n$ where the ith index is 1 if we use item i and 0 if not. So $L = \{0,1\}^n$. Furthermore, we can define T as

$$T = \left\{ \{0,1\}^k \;\middle|\; 0 \leq k \leq n \right\} \tag{7.1}$$

Informally, every node in T at height k is similarly expressible as $\{0,1\}^k$ where the ith index again represents whether item i was chosen or not. Each node $v \in \{0,1\}^k$ for $0 \leq k \leq n$

has children $(v, 1)$ and $(v, 0)$ in the next level.

We can define the function $f : L = \{0, 1\}^n \to \mathbb{R}$ as follows:

$$f(\ell) = f(\ell_1 \dots \ell_n) = \mathbb{1}_{\{\ell_1 w_1 + \ldots + \ell_n w_n \leq W\}} \cdot (\ell_1 v_1 + \ldots \ell_n v_n) \tag{7.2}$$

Here $\mathbb{1}_{\{\cdot\}}$ is the indicator function. It is 1 if the statement inside is true, and 0 if false. Therefore, what f is saying in simple terms is that the value of the items is 0 if the weights pass the capacity and otherwise is the true value $\ell_1 v_1 + \ldots + \ell_n v_n$. Define the function $h : T \to \mathbb{R}$ as:

$$h(\mathbf{t}) = f(t_1 \dots t_k) = \begin{cases} \infty & \text{if } t_1 w_1 + \ldots t_k w_k \leq W \\ 0 & \text{otherwise} \end{cases} \tag{7.3}$$

Let's verify that f and h have the desired relation. If $h(t) = \infty$ then obviously $f(\ell) \leq h(t)$, so we don't need to check this case. If $h(t) = 0$ then $t_1 w_1 + \ldots + t_k w_k > W$. Any descendant ℓ of t will satisfy $\ell_1 = t_1, \ldots, \ell_k = t_k$. Therefore,

$$(\ell_1 w_1 + \ldots + \ell_k w_k) + (\ell_{k+1} w_{k+1} + \ldots + \ell_n w_n) \geq t_1 w_1 + \ldots t_k w_k > W \tag{7.4}$$

Therefore, the indicator function is 0 so $f(\ell) = 0$ so $f(\ell) \leq h(t)$. So, we have completely written Knapsack in the Branch and Bound formalism. Effectively, we've converted our intuition of disregarding any item set that exceeds the weight capacity early into a formal function.

7.5 Traveling Salesman Problem

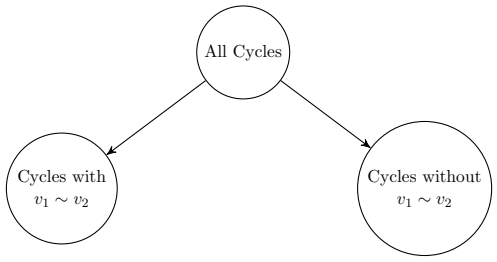

Figure 7.3: 1st level breakdown of the tree

Remember the traveling salesman problem? Given a set of verticies $V = \{v_1, \ldots, v_n\}$ and

a weight function $w : V^2 \to \mathbb{R}^+$, we want to find a Hamiltonian cycle of least weight.

So, if we were to define a tree structure to pose the Traveling Salesman Problem as a Branch and Bound problem, we would want the leaves of the tree to be Hamiltonian cycles. What about the rest of the tree? Well consider the tree as shown in Figure 7.3. At each level of the tree, we restrict ourselves to cycles that either contain or don't contain a specific edge. We do this for all $\binom{n}{2} = O(n^2)$ levels. Now, some of these branches will produce paths that aren't Hamiltonian cycles (more than two edges in the cycle per vertex) and we can truncate the tree there. Everything leaf remaining, is a Hamiltonian cycle.

Naïvely the tree contains an exponential number of vertices, so we are going to come up with a truncating function. The leaves of our tree are precisely Hamiltonian cycles and the objective function is $f(H) =$ sum of weight of edges in Hamiltonian cycle H.

As TSP is a minimizing problem, the truncating function h is going to be a lower bound for f. A simple naive lowerbound for the TSP is

$$\frac{1}{2}\sum_{v\in V}(\text{the 2 minimum weight edges incident on } v) \tag{7.5}$$

We can extend this intuition to the entire tree. If we are forced to include edge $v_i \sim v_j$ in our cycle, then for vertex v_j, one of the two edges weights we include in the truncating function sum must be $w(v_i, v_j)$. Similarly, if we are forced to exclude edge $v_i \sim v_j$ then $w(v_i, v_j)$ cannot be one of the two edges weights we include in the truncating function sum.

Its a little tedious, but you can work through the details to verify that this truncation function is well defined.[1]

[1] It is one of the few cases where it is easier to describe the function in words rather than symbol manipuations.