CS 38 Introduction to Algorithms **Week 7 Recitation Notes**

Branch and Bound, Divide and Conquer, and More

TA: Chinmay Nirkhe (chinmay@caltech.edu) & Catherine Ma (cmma@caltech.edu)

1 Branch and Bound

1.1 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.

1.2 Knapsack as a Branch and Bound 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 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 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.

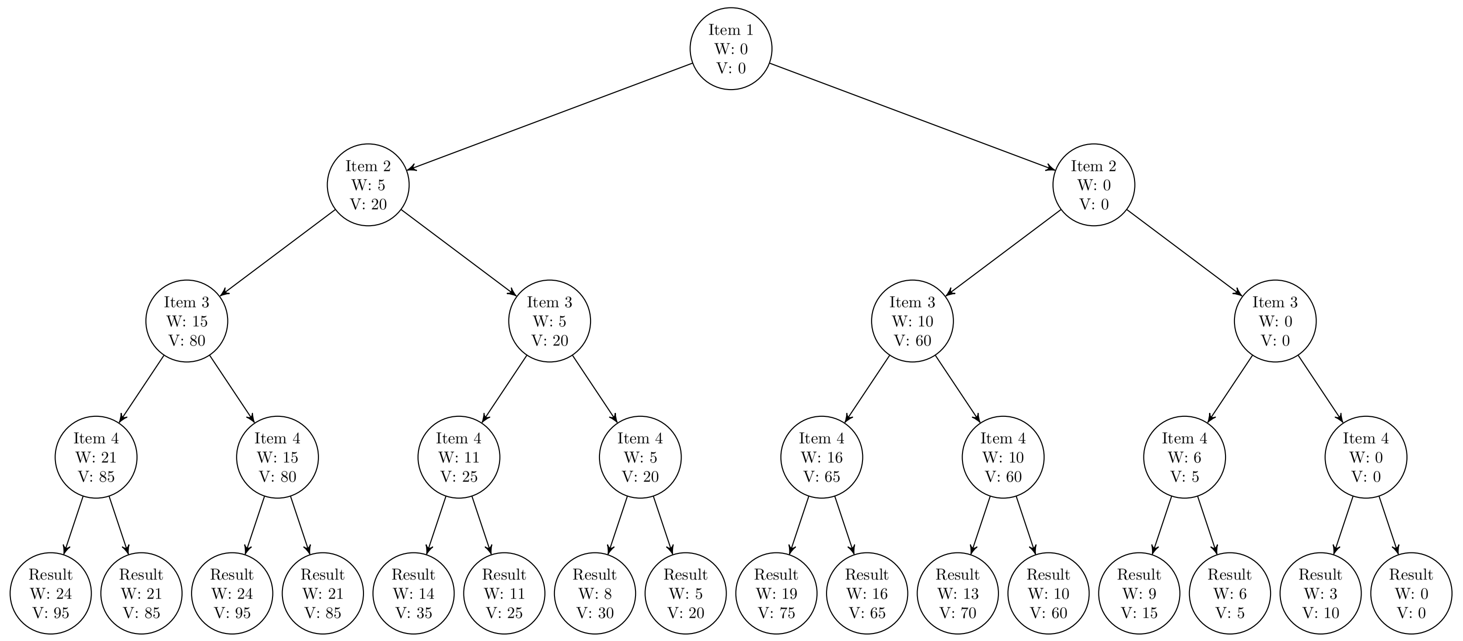

Figure 1: A graph illustration of the Knapsack decision problem. Assume 4 items with weights 5, 10, 6, 3, and values, 20, 60, 5, 10, respectively. Set the weight constraint to be 10. Here all paths are considered. Next figure shows the truncation we can do.

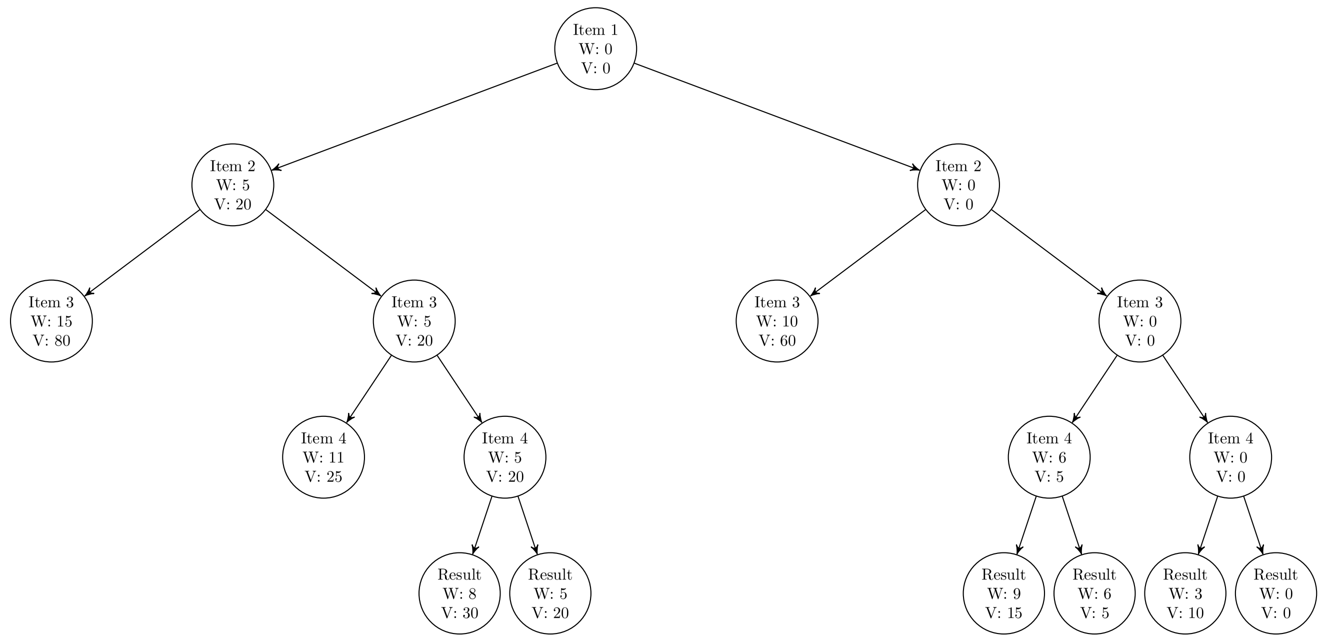

Figure 2: A graph illustration of the Knapsack decision problem. Assume 4 items with weights 5, 10, 6, 3, and values, 20, 60, 5, 10, respectively. Set the weight constraint to be 9. Here we truncate unfeasible paths early.

1.3 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.

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.

1.3.1 Generic Algorithm

Branch and Bound Algorithm

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.

1.3.2 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

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

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:

$$f(\ell) = f(\ell_1 \dots \ell_n) = \mathbb{1}_{\{\ell_1 w_1 + \ldots + \ell_n w_n \leq W\}} \cdot (\ell_1 v_1 + \ldots \ell_n v_n) \tag{2}$$

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:

$$h(\mathbf{t}) = f(t_1 \ldots t_k) = \begin{cases} \infty & \text{if } t_1 w_1 + \ldots t_k w_k \leq W \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

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,

$$(\ell_1 w_1 + \ldots + \ell_k w_k) + (\ell_{k+1} w_{k+1} + \ldots + \ell_n w_n) \geq t_1 w_1 + \ldots t_k w_k > W \tag{4}$$

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.

2 Divide and Conquer

The divide and conquer technique is a recursive technique that splits a problem into 2 or more subproblems of equal size. General problems that follow this technique are sorting, multiplication, and discrete Fourier Transforms.

2.1 Generic Algorithm Design

Algorithm

1. For positive integer $b > 1$, divide the problem into b parts
2. (Divide) Recursively solve the subproblems
3. (Conquer) Consider any situations that transcend subproblems

Complexity By the construction, the time complexity of the algorithm $T(n)$ satisfies the recurrence relation:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \tag{5}$$

where $f(n)$ is the time it takes to compute step 3 (above). In class, we looked at the following theorem about runtime:

Theorem 2.1. *(Master Theorem) If $T(n)$ satisfies the above recurrence relation, then if*

- *If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.*
- *If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.*
- *If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.*

Proof. Convince yourself that $a^{\log_b n} = n^{\log_b a}$. Drawing a recursion tree makes it easy to see that

$$T(n) = \sum_{k=0}^{\log_b n - 1} a^k f\left(\frac{n}{b^k}\right) + a^{\log_b n} T\left(\frac{n}{b^{\log_b n}}\right) = \sum_{k=0}^{\log_b n - 1} a^k f\left(\frac{n}{b^k}\right) + \Theta(n^{\log_b a}) \tag{6}$$

since $T(1)(=\Theta(1))$ is constant. For the first case,

$$\begin{aligned}
T(n) &= O\Big(\sum_{k=0}^{\log_b n-1} a^k(\frac{n}{b^k})^{\log_b a-\epsilon}\Big) + \Theta(n^{\log_b a}) \\
&= O\Big(\sum_{k=0}^{\log_b n-1} a^k(\frac{n^{\log_b a-\epsilon}b^{k\epsilon}}{a^k})\Big) + \Theta(n^{\log_b a}) \\
&= O\Big(n^{\log_b a-\epsilon}\sum_{k=0}^{\log_b n-1} b^{k\epsilon})\Big) + \Theta(n^{\log_b a}) \\
&= O\Big(n^{\log_b a-\epsilon}\frac{n^\epsilon-1}{b^\epsilon-1}\Big) + \Theta(n^{\log_b a}) \\
&= O\Big(n^{\log_b a-\epsilon}O(n^\epsilon)\Big) + \Theta(n^{\log_b a}) \\
&= O\Big(n^{\log_b a}) + \Theta(n^{\log_b a}) \\
&= \Theta(n^{\log_b a})
\end{aligned} \tag{7}$$

For the second case,

$$\begin{aligned}
T(n) &= \Theta\Big(\sum_{k=0}^{\log_b n-1} a^k(\frac{n}{b^k})^{\log_b a}\Big) + \Theta(n^{\log_b a}) \\
&= \Theta\Big(\sum_{k=0}^{\log_b n-1} a^k(\frac{n^{\log_b a}}{a^k})\Big) + \Theta(n^{\log_b a}) \\
&= \Theta\Big(n^{\log_b a}\sum_{k=0}^{\log_b n-1} 1)\Big) + \Theta(n^{\log_b a}) \\
&= \Theta(n^{\log_b a}\log_b n) + \Theta(n^{\log_b a}) \\
&= \Theta(n^{\log_b a}\log n)
\end{aligned} \tag{8}$$

For the third case, $f(\frac{n}{b}) \le \frac{c}{a}f(n)$ implies $f(\frac{n}{b^k}) \le (\frac{c}{a})^k f(n)$ for depth k in the recursion tree.

$$\begin{aligned}
T(n) &\le O\Big(\sum_{k=0}^{\log_b n-1} a^k(\frac{c}{a})^k f(n)\Big) + \Theta(n^{\log_b a}) \\
&= O\Big(f(n)\sum_{k=0}^{\log_b n-1} c^k\Big) + \Theta(n^{\log_b a}) \\
&\le O\Big(f(n)\sum_{k=0}^{\infty} c^k\Big) + \Theta(n^{\log_b a}) \\
&= O\Big(f(n)\frac{1}{1-c}\Big) + \Theta(n^{\log_b a}) \\
&= O(f(n)) + \Theta(n^{\log_b a}) \\
&= O(f(n))
\end{aligned} \tag{9}$$

since $f(n) = \Omega(n^{\log_b a+\epsilon})$ for some $\epsilon > 0$. Furthermore,

$$\sum_{k=0}^{\log_b n-1} a^k f\left(\frac{n}{b^k}\right) = \Omega(f(n)) \tag{10}$$

since it is a sum of $f(n)$'s. So,

$$T(n) = \Omega(f(n)) + \Theta(n^{\log_b a}) = \Omega(f(n)) \tag{11}$$

Thus, $T(n) = \Theta(f(n))$.

2.2 Closest Two Points

Problem Given a set $S = \{s_1, \ldots s_n\}$ where $s_i = (x_i, y_i)$ find the two closest points in euclidean distance.

Algorithm [1] We consider a divide and conquer algorithm. First sort the points by both x and y coordinate. This gives us two distinct permutations π, σ such that

$$x_{\pi(1)} \leq \ldots \leq x_{\pi(n)} \qquad y_{\sigma(1)} \leq \ldots \leq y_{\sigma(n)} \tag{12}$$

Following the divide and conquer technique we calculate d the minimum distance of the following two recursive problems: $S_1 = \{s_{\pi(1)}, \ldots, s_{\pi(n/2)}\}$ and $S_2 = \{s_{\pi(n/2+1)}, \ldots, s_{\pi(n)}\}$. (Note these are presorted so we don't need to consider sorting them again.)

Now, we need to consider any point pairs that transcend this seperation. Recognize that we only care about points whose euclidean distance is less than d. Therefore, their x distance is also $< d$. So, we need to consider any point pairs s_i, s_j that have $x_i, x_j \in [x_{\pi(n/2)} - d, x_{\pi(n/2)} + d]$. Okay now let $T_1 \subseteq S_1$ be the subset $\{s_i \ : x_i \geq x_{\pi(n/2)} - d\}$ adn $T_2 \subseteq S_2$ be the subset $\{s_j : x_j \leq x_{\pi(n/2)} + d\}$. So we're interested in pairs from (T_1, T_2).

Naïvely, this will take $O(n^2)$ time because T_1 could equal S_1 and T_2 equal S_2. However, we know points are seperated by at least d on either side of $x_{\pi(n/2)}$. Therefore, there can only be one point per $(d/2) \times (d/2)$ square on either side. Therefore, for any point $s_i \in T_1$ we only need to consider the ten closest points in y distance to s_i from T_2.[2]

We've already sorted by y, so if we construct T_1 and T_2 in $O(n)$ time from the y sorted list, we can then find pairs that transcend the seperation in $10 \cdot n/2$ comparisons which is $O(n)$. By the Master Theorem, we get a runtime of $O(n \log n)$.

[1]The way I've written an algorithm here is not the way you should write one in this course. I've interjected a lot of unneccessary details to guide the explanation.

[2]If this is confusing, the Wikipedia page for this problem has a good picture.