Greedy Algorithms

Sukanya Pandey

February 20, 2024

1 Lesson plan

1.1 Learning Outcomes

At the end of the lecture, the students should be able to:

1. explain what a greedy algorithm is, what is the greedy choice property
2. appreciate the difference between greedy algorithms and dynamic programming
3. design greedy algorithms
4. prove the correctness of a greedy strategy
5. analyze the efficiency of an algorithm in terms of its time complexity

1.2 Teaching/Learning Activities

To achieve the above intended learning outcomes, the following teaching/learning activities will be carried out.

1. Lectures supplemented with notes to understand what is a greedy algorithm.
2. Illustrating through an example problem (0-1 knapsack vs fractional knapsack) the distinction between a greedy strategy and dynamic programming;

   Think-pair-share: what are the similarities and differences between the two?
3. Multiple choice question: Considering the example of interval scheduling, asking students to pick the correct greedy strategy among a few options;
4. demonstrating how to prove the correctness of a greedy algorithm
5. tutorial session to practice designing greedy algorithms as well as proving their correctness, and analyzing their time-complexity.

The actual content of the lecture starts on the next page.

2 Introduction

Greedy algorithms are those which build the solution step-by-step, making the optimal choice at every step. Typically, greedy algorithms tend to be faster than other algorithmic strategies. However, they are risky and do not always yield an optimum solution. For example, in the lecture we see the 0-1 Knapsack problem, where the greedy strategy may yield a sub-optimal solution. A variant of the problem where greedy strategy works is the Fractional Knapsack.

To show the correctness of the greedy strategy, one needs to prove the following:

- **Greedy Choice Property** We need to prove that the greedy choice that we make, indeed yields an optimal solution. To do so, we show that there exists some optimal solution that contains the best option as per our greedy strategy.
- **Optimality Principle** To demonstrate optimal substructure of the problem, we need to show that if we combine the greedy choice with the optimal solution of the subproblem we are left with, we get the optimum solution to the original problem. [1]

2.1 Fractional Knapsack

In the fractional knapsack problem, you get as input a list $I = \{I_1, I_2, \dots I_n\}$ of n items, their values $v = \{v_1, v_2, \dots v_n\}$ and their weights $w = \{w_1, w_2, \dots w_n\}$. Your goal is to fill your knapsack with items of a total weight W, and of maximum possible value. The catch is that you are allowed to pick a fraction of any item. If f is the fraction of the item picked, its corresponding value is $v \cdot f$.

What is a good greedy strategy here?

Let us try to pick the items with the highest value to weight ratio until we either exhaust the item's supply or reach the maximum weight limit. We do this iteratively. Now, we ought to show that this strategy works, i.e, it gives us an optimal solution.

Claim 2.1. *There exists an optimum solution to fractional knapsack that contains the items in decreasing order of their value to weight ratio.*

Proof. Suppose that $O = \{o_1, o_2, \dots o_l\}$ is the list of items in the optimum solution. Let $G = \{g_1, g_2, \dots g_m\}$ be the choices made by our greedy strategy. We assume that both O and G are sorted in decreasing order of value-to-weight ratio. Let i be the first index at which the optimum solution differs from the greedy solution, i.e either $o_i \neq g_i$ or the amount of o_i is not the same as that of g_i. Then, $\frac{v_{o_i}}{w_{o_i}} \leq \frac{v_{g_i}}{w_{g_i}}$ by our choice of the greedy strategy. Remove x amount of o_i from the optimum solution and replace it with x amount of g_i. The change

[1] Note the difference in the proof strategy from the one for dynamic programming lies in the fact that after making the greedy choice, we are left with only one subproblem. In contrast, in a dynamic program, there are multiple subproblems to consider at every step.

in value is $x \cdot (\frac{v_{g_i}}{w_{g_i}} - \frac{v_{o_i}}{w_{o_i}}) \geq 0$. Therefore, we obtain a solution of value at least as much as the optimum solution. Hence, G must be optimal too. □

Exercise: Show that the optimality principle holds for the subproblem.

3 Interval Scheduling

In this problem, we have a single resource and a set of n jobs which are processed by the resource. Each job j_i comes with a deadline d_i and a contiguous interval of execution time t_i. We are required to assign an interval of time $[s(i), f(i)]$ to each job such that $f(i) = s(i) + t_i$, and no two of them overlap. A job is said to be late if it is finished after its deadline. The lateness of a job j_i is defined as $l_i = \max\{0, f(i) - d_i\}$. Our goal is to schedule all jobs using non-overlapping intervals, so that the maximum lateness, $L = \max_i l_i$, is minimized.

Greedy Strategy We see a few natural greedy strategies in the lecture that do not yield an optimal solution. Here, we shall see one that does. We schedule the job with the earliest deadline first. In other words, we sort the jobs in increasing order of their deadlines, and iteratively select the job with the earliest deadline first. The subsequent subproblem would be to decide the order in which the remaining jobs need to be scheduled to minimize the maximum lateness.

Exercise: Show that the aforementioned subproblem has an optimal substructure.

Next, we shall show that the greedy choice property holds for our strategy. We make a few observations about the optimal schedule first. We say that the resource is idle when it has no jobs to execute. The time interval for which no job is scheduled is called idle time.

Claim 3.1. *There exists an optimal solution with no idle time.*

Proof. Suppose O is an optimal solution with idle time between jobs j_i and j_{i+1} and let the time interval be t'. Thus $s(i+1) = f(i) + t'$ and $f(i+1) = s(i+1) + t_{i+1}$. By removing the gap in the solution, i.e, making $f(i) = s(i+1)$, we either decrease the lateness of j_{i+1} by t' if j_{i+1} was late, or make no difference to its lateness. Thus, the new solution obtained is as good as O, if not better and is therefore optimal. □

We say that there is an inversion in the schedule if there exists a pair of jobs j_a and j_b, with $a < b$ but $d_a > d_b$. Note that our greedy strategy always produces schedules with no inversions. Hence, to show that it produces an optimal solution, we must show that there exists some optimal solution with no inversions.

Claim 3.2. *There exists an optimal solution with no inversions.*

Proof. Let O be the optimal schedule with an inversion. Then O must contain a pair of consecutive jobs that are inverted. Let the consecutive jobs inverted be j_a and j_b, with $f(a) < f(b)$ and $d_a > d_b$. We swap the two jobs to get the schedule O'. In the new schedule, $f'(b) < f'(a)$. Also note that $f'(a) = f(b)$.

Note that the lateness of j_b cannot increase in O'. We also show that the lateness of j_a cannot be more than the lateness of j_b in O. Let $l'_a = f'(a) - d_a$ be the lateness of j_a in O'.

$$l'(a) = f'(a) - d_a = f(b) - d_a < f(b) - d_b = l_b \tag{1}$$

Therefore, the maximum lateness of O' is no more than that of O. □

Claim 3.3. *All schedules with no idle time and no inversions have the same maximum lateness.*

Proof. Two different schedules without idle time and inversions can only differ in the order of jobs with the same deadline. Among the jobs with the same deadline, the last one must have the maximum lateness. The finishing time of the last job does not depend on the order in which the jobs are executed. Therefore, the maximum lateness remains the same. □

4 Huffman Codes

Suppose you want to send an encoded message over a communication channel. To increase the efficiency to communication, you would want the encoded message to be as short as possible. Say we want to encode an n-letter alphabet.

A binary code assigns to each letter of the alphabet a string of 0's and 1's. A prefix-free binary code is one in which no code for any letter is a prefix of another. Prefix codes can be represented as full binary trees, called code trees. The encoded letters of the alphabet form the leaves of the tree. The code for each letter is given by the path from the root to the leaf corresponding to the letter, where you concatenate a 0 when you go left and a 1 if you go right.

Note that not all letters of the alphabet occur with the same frequency in the message to be encoded. We can use this to our leverage by mapping the more frequent letters to shorter binary strings and the less frequent ones longer. In other words, given the frequency $f(i)$ of each letter i of the alphabet, we wish to create a prefix-free code that minimizes the total length of the encoded message. Note that the length of the code of any letter corresponds to the depth of the corresponding leaf in the full binary tree. Mathematically, we wish to minimize

$$\sum_{i=1}^{n} f(i) \cdot depth(i) \tag{2}$$

Greedy Strategy Huffman proposed the following greedy strategy. Merge the two least frequent characters into one new character and then recurse. The

new character has a frequency equal to the sum of the frequencies of the constituent characters. We construct the corresponding binary tree, called the Huffman tree, bottom-up. We make the two least frequent characters leaves with a common parent. The parent represents the new character. We recursively build the remaining tree.

We now prove that the greedy strategy gives us an optimal prefix free code. However, we need the following lemma first.

Lemma 4.1. *Let x and y be the two least frequent characters. Then, there exists an optimal code tree in which x and y are siblings and have the greatest depth.*

Proof. Let T be an optimal code tree of depth d. The leaf at depth d must have a sibling (or else we could get rid of its parent to get a shorter code!). Suppose that the two leaves at depth d are distinct from x and y. We'll call them a and b. Swap the positions of x and a. We call the new tree thus obtained T'. After swapping, the depth of x increases while that of a decreases. Let $\delta = d\text{–}depth_T(x)$. The total length of T' is $L_{T'} = L_T + \delta * (f[x] - f[a])$. Since a is neither x nor y, $f[a] \geq f[x]$. Therefore, the total length of T' is less than or equal to that of T. But T was optimal so $L_T \leq L_{T'}$. Therefore, T' must be optimal. □

Claim 4.2. *Huffman tree is an optimal prefix-free code tree.*

Proof. Suppose that $f[1 \ldots n]$ is the list of input frequencies, and $n \geq 3$. Without loss of generality, assume that $f[1]$ and $f[2]$ are the lowest frequencies. Define $f[n+1] = f[1] + f[2]$. We know from the previous lemma that 1 and 2 are the deepest siblings in some optimal tree. Let their depth be d.

Base Case: When $n = 1$ or 2, the code tree has depth $= 1$.

Induction Hypothesis: The Huffman tree T' for frequencies $f[3 \ldots n+1]$ is optimal. (The total size of the alphabet is 1 less than before).

Induction Step: In T' we replace the node $n+1$ by the leaves 1 and 2, to get back the original tree T. We need to show that T is optimal for the list of frequencies $f[1 \ldots n]$.

Suppose that the length of the optimum code tree for the list of frequencies $f[1 \ldots n]$ is L_{OPT}. Then $L_{OPT} = L_{OPT'} + f[1] + f[2]$, where OPT' is an optimum code tree for the frequencies $f[3 \ldots n+1]$. This is because in the tree OPT, we add the letters 1 and 2 at depth one more than the depth of the letter $n+1$. Hence, we need to add $f[1] \cdot 1 + f[2] \cdot 1$ to the length of OPT.

Next let us examine the length of the code tree T.

$$
\begin{aligned}
L_T &= \sum_{i=1}^{n} f[i] \cdot depth(i) && (3)\\
&= \sum_{i=3}^{n+1} f[i] \cdot depth(i) + f[1] \cdot depth(1) + f[2] \cdot depth(2) && (4)\\
&\quad - f[n+1] \cdot depth(n+1) && (5)\\
&= L_{T'} + f[1] \cdot d + f[2] \cdot d - f[n+1] \cdot (d-1) && (6)\\
&= L_{T'} + (f[1] + f[2] - f[n+1]) \cdot d + f[n+1] && (7)\\
&= L_{T'} + f[1] + f[2] && (8)
\end{aligned}
$$

Since T' is an optimal prefix-free code tree, we conclude that T is an optimal prefix-free code tree.

□