Analysis of Algorithms I: Greedy Algorithms

Xi Chen

Columbia University

Optimization problems: Given an instance, there is a (usually very large, e.g., exponential in the input size) set of “feasible” solutions. Each solution is associated with a number, called its cost or value. We wish to find an optimal solution, among all feasible solutions, to either minimize the cost or maximize the value:

> In the Traveling Salesman Problem, we are given a list of cities and their pairwise distances. A salesman needs to make a tour: visiting each city exactly once and finishing at the city he/she starts from. Here a feasible solution is a tour that visits each city exactly once (starts and finishes at the same city). Each tour has a cost: total travel distance, and we wish to find an optimal tour that minimizes the cost.

Note that in an optimization problem, we need to find “an” optimal solution instead of “the” optimal solution, because there may be several solutions that achieve the same optimal value (and we only need to find one of them).

Given an optimization problem, a greedy algorithm “tries” to find an optimal solution by making a sequence of “greedy” choices. In each step, it makes the choice that looks best at the moment, according to some local criterion. For example,

> A greedy algorithm for the traveling salesman problem: Pick an arbitrary city and call it city 1. Find a city with the smallest distance from city 1, and call it city 2. Find a city in the rest of the $n-2$ cities with the smallest distance from city 2 ... Output the tour: City 1 $\rightarrow$ City 2 $\rightarrow \cdots \rightarrow$ City $n \rightarrow$ City 1.

Unfortunately, for many difficult optimization problems, greedy algorithms fail to find an optimal solution, because the greedy choices are made according to some local criterion (and thus, are somewhat short sighted). Remember the goal here is to find a "globally" optimal solution. In many optimization problems (especially in those difficult ones) the choices made in a "globally" optimal solution may not be "locally" optimal. When this is the case, greedy algorithms will fail. For example, the greedy algorithm from the last slide usually outputs a tour worse than the optimal.

In this class, we look at two problems where the greedy strategy works perfectly. We will also learn how to prove the correctness of a greedy algorithm when it works. In the next class, we introduce a more sophisticated scheme, dynamic programming, for solving optimization problems.

Activity Selection Problem:

1. Input: A set of n activities A to share a resource. Each activity $a = [s, f)$ has a start time s and a finish time f, where $f > s > 0$. We say two activities $a = [s, f)$ and $a' = [s', f')$ are compsatible if they do not overlap: either $s \geq f'$ or $s' \geq f$.
2. Feasible solutions: A (feasible) solution here is a sequence of activities $a_1 = [s_1, f_1), \ldots, a_k = [s_k, f_k)$ that satisfy

$$f_1 \leq s_2,\ f_2 \leq s_3,\ \ldots,\ f_{k-1} \leq s_k$$

3. Optimal solution: We wish to find a feasible solution that maximizes the number of activities in it.

Greedy choice: Which activity in A to pick as the first activity we schedule in the solution? The one with the earliest start time? or the one with the earliest finish time? The latter! Intuition: It leaves the resource available for as many other activities as possible. For example, compare $a = [3, 6)$ and $a' = [1, 7)$. Even though a' starts earlier than a, if we pick a' as the first activity then the resource will not be available until 7, while a leaves the resource available from 6. So if there is another activity, e.g. $b = [6, 8)$, we can schedule b right after a while it overlaps with a'.

Greedy algorithm for the Activity selection problem (A):

1. If $|A| = 0$, return nil; If $|A| = 1$, return the activity in A
2. Find an activity a with the earliest finish time
3. Remove a and all activities that overlap with a from A
4. Denote the new set of activities A'. Recursively find an optimal solution S' (i.e., a sequence of activities) for A'
5. Return $S = (a, S')$, the concatenation of a and S'

Easy to show that the running time is $\Theta(n)$ in the worst case. (A greedy algorithm, when it works, is usually very efficient.) But how can we prove that it always returns an optimal solution?

We use the following three steps to prove the correctness (commonly used in proving the correctness of a greedy algorithm):

1. Step 1: The first greedy choice is "safe" or "correct": Show that there is always an optimal solution S^* for A, in which a (the activity with the earliest finish time) is its first activity.
2. Step 2: Optimal substructure: Show that if S' is an optimal solution for A' (obtained by removing a and those overlap with a from A), then $S = (a, S')$ is a feasible solution as good as S^* and thus, S is optimal as well. (The proof uses Step 1.)
3. Step 3: Use induction and Step 2 to conclude (we usually skip this step but you should understand how it is done).

Lemma

Let $a = [s, f)$ be an activity with the earliest finish time in A. Then there is an optimal solution S^* for A, which starts with a.

We prove it using the following "exchange argument". Let T be an optimal sequence for A. If the first activity in T is a, then we are done. Otherwise, let $a' = [s', f')$ denote the first activity in T, then $f' \geq f$ (why?). From this, we can exchange the a' in T with a, and the result must still be a feasible solution (why?). Denote the new sequence by S^*. Then S^* is feasible and has the same number of activities as T. As a result, S^* is an optimal sequence as well. The lemma follows because a is the first activity of S^*.

Lemma

Let $a = [s, f)$ be an activity with the earliest finish time in A. Let S' denote an optimal sequence for

$$A' = A - \left\{ a \text{ and all activities that overlap with } a \right\}.$$

Then $S = (a, S')$ must be an optimal solution for A.

First, all activities in A' are compatible with a, so S is feasible. Second, from the last lemma, there is an optimal sequence S^* for A, in which the first activity is a. So to prove that S is optimal, it suffices to show that the number of activities in $S \geq$ the number of activities in S^*. To see this, all activities in S^* after a do not overlap with a. Thus, $S^* - a$ is a mutually compatible subset of A' and by the optimality of S', we have

$$|S'| \geq |S^* - a| = |S^*| - 1$$

As a result, $|S| = 1 + |S'| \geq |S^*|$ and S is also optimal. QED.

We use the second lemma to prove the correctness by induction:

1. Basis: If $|A| = 0$ or 1, the algorithm is correct. Trivial.
2. Induction Step: Assume the algorithm is correct for all sets of size $\leq k$, for some $k \geq 1$. We show that the algorithm is also correct for sets of size $k+1$. Let A denote a set of $k+1$ activities. The algorithm starts by finding a, an activity with the earliest finish time in A. Then it makes a recursive call on A' to obtain a subset S' of A'. By the inductive hypothesis, S' must be an optimal solution for A' as $|A'| \leq |A| - 1 = k$. From the second lemma we know $S = (a, S')$ is optimal for A.

Huffman codes: Given a set C of n characters and the frequency $0 \leq f(c) \leq 1$ of each character $c \in C$, we need to design an optimal prefix code for C. Equivalently, we need to construct a binary tree T with n leaves (see examples in Figure 16.4) in which each leaf is labelled with a distinct character $c \in C$. So any such binary tree T is a feasible solution. Given T, we define its cost:

$$\text{cost}(T) = \sum_{c \in C} f(c) \cdot \text{depth}_T(c)$$

Here $\text{depth}_T(c)$ denotes the depth of c (or the depth of the leaf labelled with c) in T. We wish to construct an optimal binary tree T, with respect to C and the frequencies f, to minimize its cost.

Check the textbook for the motivation behind this classical problem: What is a prefix code? How does such a binary tree represent a prefix code for C? Why does an optimal binary tree with the minimum cost represent an optimal prefix code for C with the frequencies f?

Before we discuss the greedy strategy, it is easy to prove that an optimal binary tree for C and f must be full: every internal node has two children. (Why? Show that if it is not full then we can improve it.) So we will restrict our attention to full binary trees.

Now we present the greedy algorithm invented by Huffman. Given C and the frequencies $f(c)$, it constructs an optimal binary tree with the minimum cost. What is the greedy choice here?

Intuition: Let T be an optimal binary tree for C. Let u denote an internal node with the largest depth. Because T is a full binary tree, u has two children and both are leaves. Which two characters should we put at these two leaves? Intuitively we want the frequencies of the two characters there as low as possible because these two leaves have the largest depth among all leaves of T. So let x and y be two characters in C with the smallest frequencies. Then intuitively they should be siblings in an optimal binary tree.

Inspired by this observation, we present Huffman's algorithm:

1. If $|C| = 1$ or 2, trivial; Otherwise
2. Find two characters $x, y \in C$ with the smallest frequencies
3. Remove x, y from C (Greedy choice: x, y will be siblings in the tree we construct); add a new character z with frequency

   $$f(z) = f(x) + f(y)$$

   Denote the new set by C'; note that $|C'| = |C| - 1$
4. Recursively find an optimal binary tree T' for C'
5. Locate the leaf labelled with z in T' and replace it by an internal node having x and y as its two children
6. Return the new tree T, now a binary tree for C

Again we break the proof of its correctness into three steps:

1. Step 1: The first greedy choice is “safe” or “correct”: Show that there is always an optimal binary tree T^* for C, in which x and y are indeed siblings.
2. Step 2: Optimal substructure: Show that if T' is an optimal binary tree for C', then the tree T obtained from T' must be optimal for C. Here C' is obtained from C by removing x, y and adding z with $f(z) = f(x) + f(y)$; T is obtained from T' by replacing z with an internal node having x, y as children.
3. Step 3: Use induction and the lemma from Step 2 to conclude that Huffman’s algorithm always outputs an optimal solution.

Lemma

Let x, y be two characters in C with the smallest frequencies. Then there is an optimal binary tree for C in which x and y are siblings.

Again we use an exchange argument. Let T denote an optimal binary tree for C. Let u be an internal node of T with the largest depth. If x and y are children of u, we are done. If neither x nor y is a child of u, then we use $a, b \in C$ to denote the children of u. By exchanging x with a and y with b, we get a new binary tree T'. Because u has the maximum depth and because x, y have the smallest frequencies in C, we must have $\text{cost}(T') \leq \text{cost}(T)$ (why?) and T' is optimal as well. Thus, we get an optimal binary tree for C in which x, y are siblings. The last case, where either x or y is a child of u, can be handled similarly.

Lemma

If T' is an optimal binary tree for C', then the tree T, obtained from T' by replacing the leaf for z with an internal node having x and y as children, must be optimal for C. Here C' is obtained from C by removing x, y and adding z with $f(z) = f(x) + f(y)$.

First of all, it is easy to check that

$$\text{cost}(T) = f(x) + f(y) + \text{cost}(T')$$

using the fact that all characters have the same depth in T as in T' except x, y, where $\text{depth}_T(x) = \text{depth}_T(y) = \text{depth}_{T'}(z) + 1$.

By the last lemma, there must be an optimal binary tree T^* for C in which x and y are siblings. Given T^*, we remove x and y and label their parent with z. Denote the new tree T'', then T'' is a binary tree for C'. Similarly, one can show that

$$\text{cost}(T^*) = f(x) + f(y) + \text{cost}(T'')$$

Due to the optimality of T', we have

$$\text{cost}(T') \leq \text{cost}(T'')$$

and thus, $\text{cost}(T) \leq \text{cost}(T^*)$. We conclude that T is also an optimal binary tree for C (because its cost is no more than the cost of an optimal).

We skip Step 3 (induction). As for the time complexity of Huffman's algorithm, if we use a red-black tree (or any data structure that supports Insert, Delete, and Min in worst case $O(\lg n)$ time, e.g., a binary min-heap discussed in Chapter 6) to store the characters, sorted using their frequencies as the keys, then the running time is $O(n \lg n)$ in the worst case.