COMPSCI 330: Design and Analysis of Algorithms 1/26/2016 and 1/28/2016

Greedy Algorithm

Lecturer: Debmalya Panigrahi *Scribe: Tianqi Song, Fred Zhang*

1 Overview

This lecture introduces a new algorithm type, greedy algorithm. General design paradigm for greedy algorithm is introduced, pitfalls are discussed, and four examples of greedy algorithm are presented along with running time analysis and proof of correctness.[1]

2 Introduction to Greedy Algorithm

Greedy algorithm is a group of algorithms that have one common characteristic, making the best choice locally at each step without considering future plans. Thus, the essence of greedy algorithm is a choice function: given a set of options, choose the current best option. Because of the myopic nature of greedy algorithm, it is (as expected) not correct for many problems. However, there are certain problems that can easily be solved using greedy algorithm, which can be proved to be correct.

3 Activity Selection (Interval Scheduling) Problem

3.1 Problem Description

There is a set of activities $S = \{a_1, a_2, ..., a_n\}$. Each activity a_i has a start time s_i and a finish time f_i, where $f_i > s_i \geq 0$. The duration of activity a_i is $[s_i, f_i)$. Two activities a_i and a_j are compatible if $s_i \geq f_j$ or $s_j \geq f_i$. We want a subset of S that has maximum number of activities that are mutually compatible.

3.2 Algorithm

Sort the activities by finish time. Initialize S_m as $\emptyset$. Take the activity with the earliest finish time in S and copy it to S_m. Delete this activity and all activities that are not compatible with it from S. Keep doing this until S is empty, and then return S_m.

3.3 Correctness

Proof is by an inductive way. Basis: if $S = \emptyset$, the algorithm is correct. Inductive step: for $S \neq \emptyset$, let activity a be the activity with the earliest finish time. Let S_a be the subset of activities that contains a and all activities that are not compatible with a. Let $S_l = S - S_a$, so $S_a \cup S_l = S$ and $S_a \cap S_l = \emptyset$. Assume that the algorithm gives correct answer for input S_l. We need to prove that it also gives correct answer for S. Let W be a subset of S that contains maximum number of activities that are mutually compatible. We can divide W into W_a and W_l such that $W_a \subseteq S_a$ and $W_l \subseteq S_l$, so $W_a \cup W_l = W$ and $W_a \cap W_l = \emptyset$. We have $|W| = |W_a| + |W_l|$. Let $\{a\} \cup K_l$ be the result for input S by our algorithm. Base on the assumption, we have that K_l is an optimal

[1]Some of the material in this note is from a previous note by Yilun Zhou for this class in Fall 2014.

solution for S_l and then $|K_l| \geq |W_l|$. $|W_a| = 1$, otherwise, it is contradictory to that W is an optimal solution or the fact that a is the activity with the earliest finish time. Therefore $|\{a\} \cup K_l| = |W|$ and the algorithm gives correct answer for S.

3.3.1 Time Complexity

The time complexity is dominated by the cost of sorting the n items by finish time, which is $O(n \log n)$.

4 Interval Coloring

4.1 Problem Description

Let us turn to a new problem called *Interval Coloring* or *Interval Partitioning*. Suppose we are given a set of n lectures, and lecture i starts at time s_j and ends at time f_j. The goal is to use the minimum number classrooms to schedule all lectures so that no two occur at the same time in the same room. As an illustration of the problem, consider the sample instance:

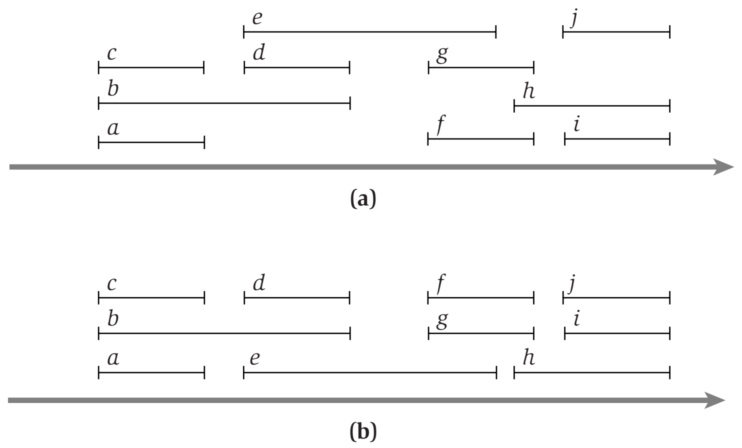

Figure 1: (a) An instance of the Interval Partitioning Problem with ten intervals, or rather, lectures (a through j). (b) A solution in which all lectures are scheduled using 3 classrooms: each row represents a set of lectures that can all be scheduled in a single classroom.

Now, is there any hope of using just two classrooms in this instance? Clearly the answer is no. We need at least three classrooms since, for example, lectures a, b, and c all pass over a common point on the time-line, and hence must require three classrooms. Can we generalize this argument? Let us define that the *depth* of a set of intervals (lectures) to be the maximum number that pass over any single point on the time-line. Then we observe that the number of classrooms needed is at least the depth of the input set. Hence, any schedule that uses a number of classrooms that equals to depth is in fact *optimal*, since we cannot do any better.

4.2 Algorithm

Then can we always find an optimal schedule? The answer is yes, and we now design a simple greedy algorithm that schedules all lectures using a number of classrooms equal to the depth. Consider lectures in increasing order of start time and assign lecture to any compatible classroom. If there all classrooms are taken when we try to assign a lecture, then open a new classroom. We keep track of the number of classrooms opened in the variable k.

Algorithm 1 Compute an optimal schedule

```
function SCHEDULE($s_1, s_2, \cdots, s_n$)
    Sort the intervals by starting time in non-decreasing order.
    $d \leftarrow 0$
    for $j = 1$ to $n$ do
        if lecture $j$ is compatible with some classroom $k$ then
            schedule lecture $j$ in classroom $k$
        else
            allocate a new classroom $d+1$
            schedule lecture $j$ in the new classroom
            $d \leftarrow d+1$
```

4.3 Correctness and Running Time

The running time of the algorithm is dominated by the sorting step, so it runs in $O(n \log n)$ time. Now let us prove it indeed generates a feasible and optimal schedule that minimizes the number of classrooms. First we observe that the greedy algorithm never schedules two incompatible lectures in the same classroom, simply by its definition. To establish optimality, let d be the number of classrooms that the greedy algorithm allocates. Then classroom d was opened because we needed to schedule a lecture, say j, that is incompatible with all $d-1$ other classrooms. It follows that these d lectures each end after s_j. Since we sorted by start time, these lectures also start no later than s_j. Hence, we have d lectures overlapping at this moment, or technically, at $s_j + \varepsilon$ for some small constant ε. This implies the depth is at least d and our schedule is optimal.

5 Knapsack Problem

Consider the situation when a burglar breaks into a house. After seeing all items in the house, he has a clear idea of what the value of each item is and what the volume of each item is. He wants to take everything but he only has a ”knapsack” of a certain size. Thus, he wants to take a subset of items of maximum total value but still fits his knapsack. In addition, all items are indivisible (i.e. he cannot take half of TV and get half of its value). What should he take?

Definition 1. *In a knapsack problem, there is a set I containing n items, labeled $1, 2, ..., n$. Each item is associated with a value $v_1, v_2, ..., v_n$ and a weight $w_1, w_2, ..., w_n$. In addition, there is a constraint W. The problem asks to find the subset $I' \subseteq I$ that maximizes the total value of items in it $\sum_{i \in I'} v_i$ subject to the constraint $\sum_{i \in I'} w_i \leq W$.*

The solution to this knapsack problem will be presented in a later lecture and this problem is a computational hard problem.

5.1 Fractional Knapsack Problem

Although the previous knapsack problem is not easy to solve, a variant of it, fractional knapsack problem, can be solved efficiently using greedy algorithm.
Now suppose instead the burglar breaks into a grocery store. All items in the grocery store are divisible. For example, he can take half a bag of flour and get half of its value. Still having the constraint of the total size of his knapsack, what should be his strategy to take items?

Definition 2. *In a fractional knapsack problem, there is a set I containing n items, labeled $1,2,...,n$. Each item is associated with a value $v_1,v_2,...,v_n$ and a weight $w_1,w_2,...,w_n$. In addition, there is a constraint W. The problem asks to find the weight of each item to take $w'_1,w'_2,...,w'_n$ that maximizes the total value of taken items $\sum_{i=1}^{n} v_i \frac{w'_i}{w_i}$ subject to the constraint $\sum_{i=1}^{n} w'_i \leq W$ and $w'_i \leq w_i$.*

5.1.1 Algorithm

The solution to fractional knapsack problem is relatively easy to come up with. Because you can take fraction of items, you are guaranteed to be able to fill the knapsack (note that this is not true in the previous knapsack problem: you may have some empty room left that is smaller than all remaining items). Thus, with a constant total weight, finding the maximum value becomes finding the maximum value per unit weight. Thus, you should first sort the items by value per unit weight and then take items in decreasing order (most valuable item first). Take the whole item when possible and take as much as you can for a given item when there is no space for the whole item.

5.1.2 Correctness

We define the value per unit weight $p_i = \frac{v_i}{w_i}$. The items are sorted by value per unit weight as $p_{k1} \geq p_{k2} \geq ... \geq p_{kn}$. Let the weight of item ki in the greedy solution be $w_{ki}*$. The greedy algorithm has the property that for any $i < j$, if $w_{ki}* < w_{ki}$, then $w_{kj}* = 0$. Let a general solution take $\hat{w_{ki}}$ amount (weight) of item ki. We find an item ki such that $\hat{w_{ki}} < w_{ki}*$ and an item kj such that $\hat{w_{kj}} > w_{kj}*$, where $i < j$, if possible. Take out $\min(w_{ki}* - \hat{w_{ki}}, \hat{w_{kj}} - w_{kj}*)$ amount of item kj and add in the same amount of item ki. The total value in the knapsack will not decrease because $p_{ki} \geq p_{kj}$ for $i < j$. We keep doing this kind of transfer until it has to stop. The transfer activities will definitely stop because each transfer will at least make one item have the property that its weight is equal to greedy solution and then this item is out of the future transfer activities. Note that, among the items with a different weight from the greedy solution, the one with the largest value per unit weight must have a smaller weight than the greedy solution, otherwise, the greedy algorithm is not greedy. Based on this fact, when the transfer activities stop, there might be two possible situations: (i) We reach the greedy solution; (ii) Each item that may participate in the future transfer activities has a weight that is smaller than the greedy solution. Case (ii) cannot happen because the total weight is not changed by a transfer. Therefore, any solution can be transformed to the greedy solution and then its total value cannot be larger than the greedy solution.

5.1.3 Time Complexity

The time complexity is dominated by the cost of sorting the n items by value per unit weight, which is $O(n \log n)$.

5.1.4 Relation to Original Knapsack Problem

As mentioned before, in the original knapsack problem you cannot take fractions so you may end up wasting some spaces. So in original knapsack problem, $a_1, a_2, ..., a_n = \{0, 1\}$. This difference adds (a huge amount of) complication and makes the difference of a problem that can be solved efficiently (in polynomial time) and a problem that cannot, as we will see later this semester.

6 Huffman Code

Consider I which is a data composed of characters drawn from an alphabet C. Each character $c \in C$ has a frequency $f(c)$ which is the number of times c shows up in I. We encode each character $c \in C$ by a codeword which is a binary string, so we need $\sum_{c \in C} f(c)l(c)$ number of bits to encode I, where $l(c)$ is the length of the codeword of c.

	a	b	c	d
Frequency	20	10	5	1
Fixed-length code	00	01	10	11
Variable-length code	1	01	000	001

Table 1: The fiexed-length code and variable-length code of a data composed of characters drawn from alphabet $\{a, b, c, d\}$

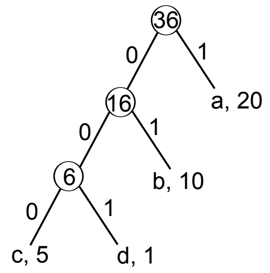

Figure 2: The tree of the variable-length code (which is a prefix code) in Table 1. The number associated with a character is its frequency. The number in an internal node is the sum of the frequencies of its children nodes.

There are multiple kinds of codes. A simple one is fixed-length code, where the codeword of each character is of the same size. For example, as shown in Table 1, each character has a 2-bit codeword and it needs $(20+10+5+1)*2 = 72$ bits to encode the data. Another way is variable-length code where frequent characters have short codewords and infrequent characters have long codewords. If we use the variable-length code in Table 1, it only needs $20*1+10*2+5*3+1*3 = 58$ bits to encode the data. Here, we only consider prefix codes in which no codeword is a prefix of another codeword. Prefix codes are preferred because they do not cause ambiguity. Each prefix code can be described as a tree (see one example in Figure 2), where a leaf represents a character, and the path from the root to the leaf of a character specifies the codeword of the character.

6.1 Huffman Algorithm

Huffman code is a prefix code created by an greedy algorithm. The algorithm constructs the tree as following: Initialize of node set S, where each character in the alphabet C is allocated a node. Choose two nodes with least frequencies and merge them as a new node whose frequency is the sum of the two nodes. Assign the two nodes as the children of the new node. Delete the two nodes from S and add in the new node. Keep doing this until there is only one node in S and then return the tree.

6.2 Correctness

Proof is by an inductive way. Basis: If there are only two characters in C, the algorithm encodes each of them by one bit and this is optimal. Inductive step: Let the characters with least frequencies in C be α and β. Let C' be $C - \{\alpha, \beta\} \cup \{\gamma\}$, where γ is achieved by merging α and β. Assume that the algorithm gives optimal solution (tree) to C' denoted by T'. We want to prove that it also gives optimal solution to C: The solution to C, denoted by T, costs $L(T) = \sum_{c \in C} f(c) l_T(c) = L(T') + f(\alpha) + f(\beta)$. If the solution T to C is not optimal, we have another solution T'' such that $L(T'') < L(T)$. In T'', a and b can be siblings because for any tree constructed for C, we can transform it such that a and b are siblings without increasing the cost. We delete the nodes of a and b from T'' to get $T*$ and we have $L(T*) = L(T'') - f(a) - f(b) < L(T) - f(a) - f(b) = L(T')$, which is contradictory to our assumption.

6.3 Time Complexity

There are $n-1$ rounds of merging activities. The major cost of each round is to find the two nodes with least frequencies, which is $O(\log n)$ time if we use binary min-heap, and then the total cost is $O(n \log n)$.