Lecture 24: Greedy algorithms (Ch 16)

CS303: Algorithms

Last update: March 3, 2014

1 Review

1. Sequence alignment problem

2 Idea of greedy algorithms

A very useful design strategy for optimization problems. DP is one way to solve optimization problems but sometime it is an overkill by considering quite a few cases locally.
Idea of Greedy algorithms When we have a choice to make, make the one that looks best *right now*. Make a *locally optimal choice in hope of getting a globally optimal solution.*

Greedy algorithms don't always yield an optimal solution. But sometimes they do. We'll see a problem for which they do. Then we'll look at some general characteristics of when greedy algorithms give optimal solutions.

3 Example 1: activity selection

n activities require exclusive use of a common resource. For example, scheduling the use of a classroom.

Set of activities $S = a_1, a_2, ..., a_n$. a_i needs resource during period $[s_i, f_i)$, which is a half-open interval, where s_i =start time and f_i =finish time.

Goal

Select the largest possible set of non-overlapping (mutually compatible) activities.
Note

Could have many other objectives:

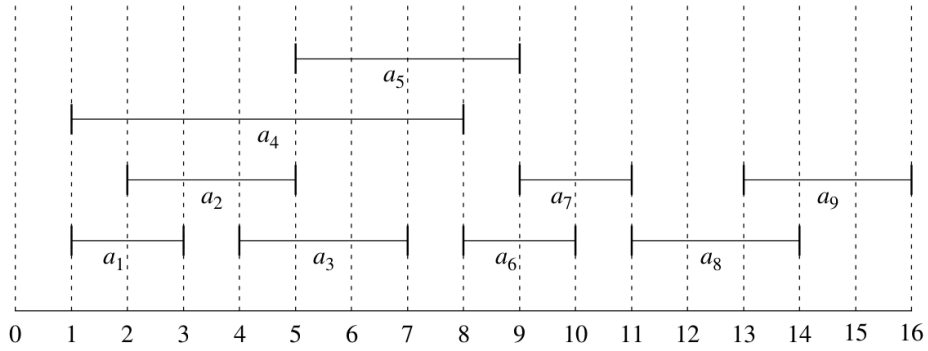
- Schedule room for longest time.
- Maximize income rental fees.

Assume that activities are sorted by finish time: $f_1 \leq f_2 \leq f_3 ... \leq f_{n-1} \leq f_n$

Example

Suppose S have been sorted by finish time

i	1	2	3	4	5	6	7	8	9
s_i	1	2	4	1	5	8	9	11	13
f_i	3	5	7	8	9	10	11	14	16



The maximum-size mutually exclusive set is $\{a_1, a_3, a_6, a_8\}$. Note this isn't a unique solution because $\{a_2, a_5, a_7, a_9\}$ is also optimal.
We will first try to solve the problem using DP, then show that greedy choice is better in this case.

3.1 Optimal substructure of activity selection

$S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$

S_{ij} is basically a set that start after a_i and finish before a_j starts.
For example, in our figure above, $S_{18} = \{a_3, a_5, a_6, a_7\}$ and $S_{17} = \{a_3\}$.

Activities in S_{ij} are compatible with

- all activities that finish by f_i , and
- all activities that start no earlier than s_j .

Let A_{ij} be a maximum-size set of mutually compatible activities in S_{ij} . Let $a_k \in A_{ij}$ be some activity in A_{ij}. Then we have two subproblems:

- Find mutually compatible activities in S_{ik} (activities that start after a_i finishes and that finish before a_k starts). $\rightarrow A_{ik}$
- Find mutually compatible activities in S_{kj} (activities that start after a_k finishes and that finish before a_j starts).$\rightarrow A_{kj}$

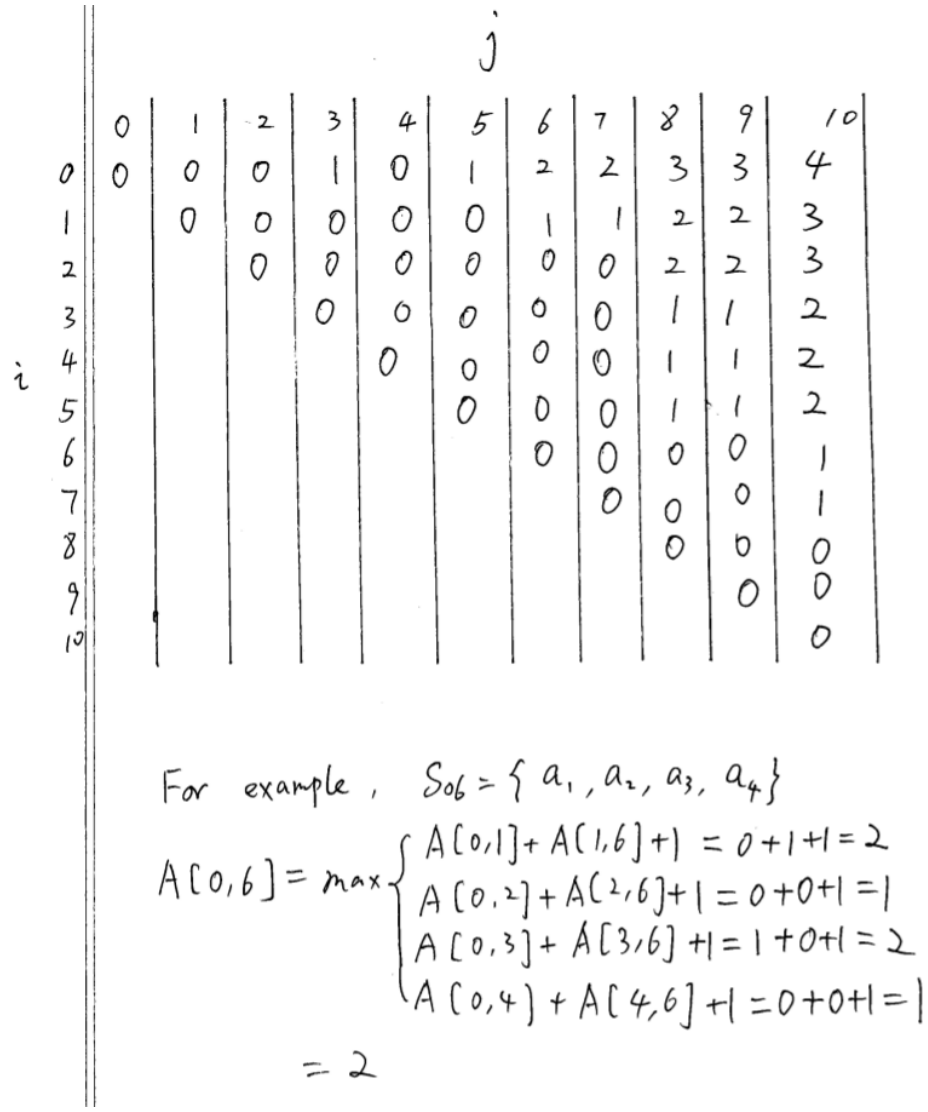
Thus $|A_{ij}| = |A_{ik}| + A_{kj}| + 1$ with our goal as $A_{0,n+1}$
Thus by using dynamic programming, we can do the following

$$A[i,j] = \begin{cases} 0 & if\ S_{ij} = \varnothing \\ max_{a_k \in S_{ij}} A[i,k] + A[k,j] + 1 & if\ S[i,j] \neq \varnothing \end{cases}$$

By examining the recursive relationship, we can fill in the table diagonally and in fact, only the top half is used because $a_k \in S_{ij}$ only when $i < k < j$.

To compensate the first event and the last event, we can introduce a_0 whose start time is $-\infty$ and ends at s_1. Similarly, a_{n+1} has a start time of f_n and ends at ∞.

In our previous example, here is the DP table.



4 Making the greedy choice

Choose an activity to add to optimal solution before solving subproblems. For activity-selection problem, we can get away with considering only the greedy choice: the activity that leaves the resource available for as many other activities as possible.

Q:/ Which activity leaves the resource available for the most other activities?
A:/ The first activity to finish. (If more than one activity has earliest finish time, can choose any such activity.) Since activities are sorted by finish time, just choose activity a_1.

Q:/ Why not the shortest activity?
A:/ Max number of activities is more about the relationship of activities instead of individual activities.

That leaves only one subproblem to solve: finding a maximum size set of mutually compatible activities that start after a_1 finishes.

Thus the subproblem changes to $S_k = \{a_i \in S : S_{\geq} f_k\}$=activities that start after a_k finishes.
Thus we are look at after finding a_1, we try to solve S_1. We can prove that a_1 is always part of some optimal solution.

Proof: Suppose A is an optimal solution for S. Pick the earliest finish time activity and call it a_k. If $a_k \neq a_1$, then we can basically get rid of a_k and put in a_1 and there won't be any conflict since a_1 finishes first. The resulted new set is also an optimal solution since it has the same number of activities as the original A.

So overall the idea is to use recursion to find the solution of a sub-problem after we make a greedy decision.
The solution of the activity selection problem in our original example is a_1, a_3, a_6, a_8. As can be seen, we do miss one optimal solution if we use the greedy approach.

Difference between greedy and DP

- Dynamic programming
  - Make a choice at each step.
  - Choice depends on knowing optimal solutions to subproblems. Solve subprob- lems first.
  - Solve bottom-up.
- Greedy
  - Make a choice at each step.
  - Make the choice before solving the subproblems.
  - Solve top-down.

5 Huffman tree

It is one of the most important algorithms in compression. It compresses information by thinking about how many information is needed for each symbol it needs to represent.

A few definitions:
Coding: assignment of bit strings to alphabet characters
Codewords: bit strings assigned for characters of alphabet

There are two types of codes:

1. fixed-length encoding (e.g., ASCII)

2. variable-length encoding (e,g., Morse code)

However, by introducing variable-length encoding, we carried one cach: how can we tell how many bits of an encoded text represent the ith character?

e.g.
Using Morse code, what does it mean?
.–.
Does it mean P or WE?

To avoid this, we can introduce prefix free codes:
Prefix-free codes: no codeword is a prefix of another codeword

Thus if the code is prefix free, we can simply scan a bit string until we get the first group of bits that is a codeword for a symbol.

Problem: If frequencies of the character occurrences are known, what is the best binary prefix-free code?

One approach is to build a tree with edges labeled 0 and 1 and symbols as leaves. Hopefully the tree will be optimal such that the average length for each symbol is optimal.

Idea: If the frequencies of the symbols are known in advance, then we can use that information to place symbols into leaves (more frequent ones will be higher upper in the tree (i.e. less 0 and1 for it)).

We need the frequency of each symbol in the text we will encode

5.1 Huffman's algorithm

Initialize n one-node trees with alphabet characters and the tree weights with their frequencies.

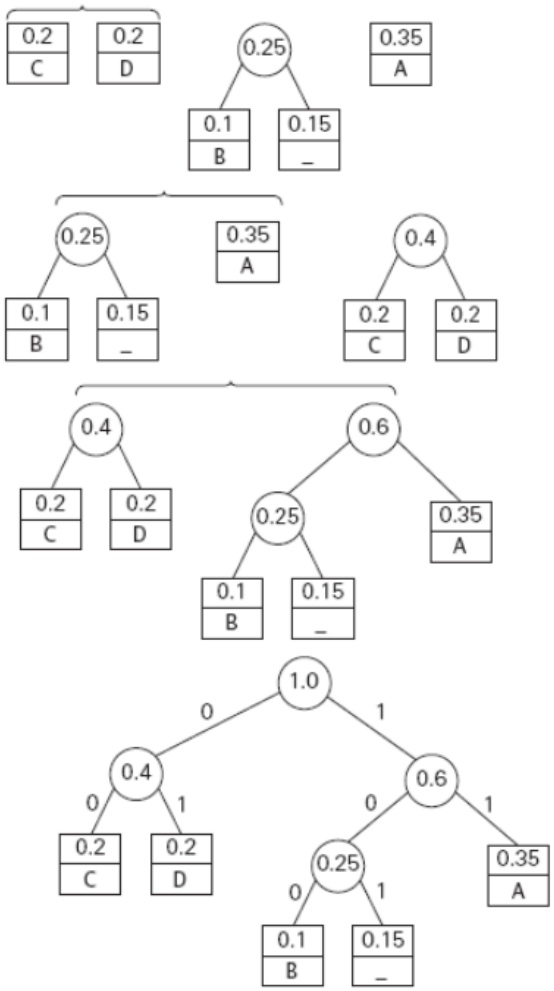
Repeat the following step $n - 1$ times:

- join two binary trees with smallest weights into one (as left and right subtrees) and make its weight equal the sum of the weights of the two trees.
- Mark edges leading to left and right subtrees with 0's and 1's, respectively.

Example

e.g.
If the frequency of ABCD and space are the following, build a Huffman's tree to encode each character.

character	A	B	C	D	-
frequency	.25	.1	.2	.2	.15



Thus the code words are
A:11
B:110
C:00
D:01
-:101

average bits per character: 2.25
for fixed-length encoding: 3
compression ratio: (3-2.25)/3*100% = 25%