EECS3101 notes:: Practice on greedy algorithms

Jerry Wu

2022-03-14

Exercises

Example 1 :: task scheduling

Given a collection L of n jobs with L_i being the length of job i to be run on a single core CPU, order the jobs to **minimize** the total waiting time.

Example input:

$$L = \{3, 1, 4, 1, 6\}$$

What is the total waiting time if the schedule is

> *"Keep an eye on me in case I make a wrong calculation"-Larry YL Zhang 2023*

- $S_1 = \{4, 3, 1, 6, 1\}, Time = 0 + 4 + 7 + 8 + 14 = 33$
- $S_2 = \{6, 4, 3, 1, 1\}, Time = 0 + 6 + 10 + 13 + 14 = 43$
- $S_3 = \{1, 1, 3, 4, 6\}, Time = 0 + 1 + 2 + 5 + 9 = 17$

This problem is applicable in a real scenario, since we can use the information generated to measure the performance of a CPU. For the above 3 examples, we can observe that the one where the times are sorted from smallest to greatest has the least total waiting time. Can we prove that this is the optimal solution? Yes we can! This is called SJF (shortest job first).

```
1 while $\exists$ unfinished jobs:
2     run the shortest unfinished job
```

Of course, real world algorithms for task scheduling are a lot more complex, since CPUs have multiple cores, and we have no knowledge on how long exactly a task will take. This will be gone over in EECS3221.

Proof of SJF

Recall that the purpose of the proof is to prove there does not exist a more optimal solution than the greedy solution.
Take an SG as the greedy solution to the problem, i.e. the permutation of the list of jobs such that $\exists SO$ where $Opt(SO) > Opt(SG)$ for the sake of contradiction. We know that SG is a list of jobs that are sorted in increasing order for their respective waiting times. So here, we don't have to sort anything ourselves since the solution is already in sorted order. Define index k such that $SG_k < SO_k$, i.e. the first index where the solutions are different. Here, $SG_k < SO_k$ because the greedy approach will always choose the best possible choice (smallest value in this case). Construct a new solution set SO' such that SG_k comes before SO_k, then we have a time interval at which the waiting times for all jobs between SG_k and SO_k are shifted by $\Delta t = SO_k - SG_k$. Thus, the waiting times of all subsequent jobs are reduced by this amount of time, leading to a reduction of the total waiting time across the board, whereby we have a contradiction, since $Opt(SO') > Opt(SO)$, meaning SG is indeed the most optimal. ■

Example 2 :: Huffman coding for data compression

Given a piece of data, we wish to compress (minimize) its size by storing it in a different format. We can do both lossy and lossless compression.

- With audio, it's fine if we lose a bit of information.
- When zipping files into a zip folder, we must use lossless since we want to unzip and still get everything we originally had. Otherwise, we get corrupted files.
- For images, the JPEG compression takes neuroscience and visual perception into account, since the pixels will be arranged in such a way that the viewer's own cognitve perception of the image will make up for lost pixels from said compression. This is gone in depth in another course.

This example will go over a lossless compression method. The idea is that we can start off with a code of fixed length, but change it in such a way that we can get a code of variable length. Some of them might be shorter, but they can also be longer. Our job is to minimize the number of long codes and maximize the number of short codes. To do this, we can assign short codewords to the frequently occuring characters, and longer codewords to characters that occur less frequently.

Possible issue (unique decoding)

Say we have the string 101010. Should the variable length codewords be divided into 10+1010 or 101+010? This generates ambiguity, and we dont want this for a lossless compression. We can deal with this issue using a concept known as **prefix code**.

- Organize codewords into a **prefix tree** (binary tree) where each leaf node represents a valid codeword.
- No codeword can be a prefix of another codeword.
- The prefix of any given code is an **ancestor** of the code (including parent).

Take the following frequencies of characters::

$$\{a : 45, b : 13, c : 12, d : 16, e : 9, f : 5\}$$

What is the optimal way of building the prefix tree for these frequencies? To be greedy, we can start with the two least frequent letters (e and f) to be at the bottom level of the tree, since we want the longest codes to be assigned to the least frequent occurences. The combined frequencies of e and f together would be 9+5=14. The next level would be b and c, which occur 13 and 12 times respectively, where their combined frequencies are 25. Then combine d with e and f to get 14:16=30. Then we get 55 in total combined with 45 a's which gives 100. In this tree, each parent's weight is the sum of the weights of their children. (tree on slide 14)

"Excuse me for the shape of the tree"-Larry YL Zhang 2023

In essence, we want to always merge the **minimum two** nodes to get the optimal solution. Now we have to prove that the greedy approach is always the optimal approach.

Lemma 1

If characters x and y have the lowest frequencies, then $\exists$ an optimal prefix code in which they are sibling leaves (share the same parent).

Proof

- Base case:: The string is 1 character long. This is trivial.
- Inductive hypothesis:: Assume the huffman tree is optimal for a k-character alphabet.
- Now consider an alphabet with $k+1$ characters. Then,
  - Let x and y be two characters with the lowest frequencies, and T_{k+1} to be the resulting Huffman tree where x and y would be merged first.
  - Merge x and y into z where z has the combined frequency of x and y
  - By IH, running Huffman on the k character alphabet results in an optimal tree T_k.
  - For the sake of contradiction, let us assume that T_{k+1} is **not** optimal. Then $\exists$ an optimal tree R_{k+1} where $Opt(R_{k+1}) > Opt(T_{k+1})$ that has x and y being sibling leaves.
  - From here, we can conclude that merging x and y would give us an R_k such that $Opt(R_k) > Opt(T_k)$, which is a contradiction since we assumed T_k to be the most optimal solution.■

Closing remarks and summary

The reason we want to know dynamic programming and greedy algorithms is not to determine the exact solution to a problem, but to construct and validate a solution.