CS 3000: Algorithms & Data — Spring 2024

Due Sunday March 17 at 11:59pm via Gradescope

Name: Ryan Tietjen
Collaborators:

- Make sure to put your name on the first page. If you are using the LaTeX template we provided, then you can make sure it appears by filling in the `yourname` command.
- This homework is due Sunday March 17 at 11:59pm via Gradescope. No late assignments will be accepted. Make sure to submit something before the deadline.
- Solutions must be typeset. If you need to draw any diagrams, you may draw them by hand as long as they are embedded in the PDF. We recommend that you use LaTeX, in which case it would be best to use the source file for this assignment to get started.
- We encourage you to work with your classmates on the homework problems, but also urge you to attempt all of the problems by yourself first. If you do collaborate, you must write all solutions by yourself, in your own words. Do not submit anything you cannot explain. Please list all your collaborators in your solution for each problem by filling in the `yourcollaborators` command.
- Finding solutions to homework problems on the web, or by asking students not enrolled in the class is strictly forbidden.

Problem 1. *(2 + 8 + 8 + 2 = 20 points) Minimizing weighted delay at a bank*

A bank manager has n clients waiting to be served, all of whom arrived at the same time. Let us call the time at which these clients arrive to be 0. The manager estimates that client i requires t_i time units of service. They also know the clients well and assign each client a weight according to how important the client is to the bank (say, the outstanding balance of their bank account): client i has weight w_i. The manager would like to determine a good order in which to serve the clients.

In a given order, define the delay of client i to be the time at which client i starts getting served. The total weighted delay of a given order is the sum, over each client i, of the product of w_i and the delay of i. The goal of this problem is to determine an order to serve the clients that minimizes the total weighted delay.

For example, consider 4 clients with $t_1 = 5$, $t_2 = 2$, $t_3 = 7$, $t_4 = 4$. And weights $w_1 = 1$, $w_2 = 3$, $w_3 = 2$, and $w_4 = 2$. Consider the order $1, 3, 2, 4$. The delay for client 1 is 0, for client 2 is $t_1 + t_3 = 5 + 7 = 12$, for client 3 is $t_1 = 5$, and for client 4 is $t_1 + t_3 + t_2 = 5 + 7 + 2 = 14$. The total weighted delay of the order is then $1 \cdot 0 + 3 \cdot 12 + 2 \cdot 5 + 2 \cdot 14 = 36 + 10 + 28 = 74$.

(a) Describe precisely what your algorithm is given as input and what it needs to output.

Solution:

The algorithm is given 2 arrays as input(let n be the number of clients):

-An array of integers t[1...n] where t[i] denotes the time units of service that client i requires.

-An array of integers w[1...n] where w[i] denotes the weight that client i holds.

-Optional: Take in the number of clients n (not required since both t[] and w[] of are size n)

The algorithm should output an array of integers c[1...n] that contains each client number (i.e. every integer from 1 to n) and represents the order in which the clients should be served. This array should be ordered such that the total weighted delay of the order is minimized.

(b) Prove the following greedy choice property for the problem of finding an order that minimizes the total weighted delay: there exists an optimal solution in which the first client to be served is the client i that has the smallest value of t_i/w_i.

(Hint: Start by proving the statement when there are only two clients. Extend to the general case using an exchange argument.)

Solution:

We will begin by proving the statement when there are only two clients:

First, suppose there are two clients A and B where t_A and t_B represent the service times of A and B and w_A and w_B represent the weights of A and B. Also, suppose that $\frac{t_A}{w_A} < \frac{t_B}{w_B}$.

If A is served before B, then the total weighted delay is $t_A \cdot w_B$

If B is served before A, then the total weighted delay is $t_B \cdot w_A$

We must show that serving client A first is at least as good as serving client B first (i.e. If B were to be served before A, then flipping A and B would only decrease the total weighted delay) Hence, we must show:

$t_A \cdot w_B \leq t_B \cdot w_A$

Divide both sides by w_A and w_b:

$\frac{t_A}{w_A} \leq \frac{t_B}{w_B}$

Since $\frac{t_A}{w_A} < \frac{t_B}{w_B}$, clearly $\frac{t_A}{w_A} \leq \frac{t_B}{w_B}$. So, serving A before B only results in a smaller total weighted delay. Hence, the greedy choice property holds for two clients.

Now, we can extend to the general case using an exchange argument.

Consider the optimal solution O.

-Case: In O, the first client has the smallest value of $\frac{t_i}{w_i}$ is trivial.

-Case: In O, the first client does not have the smallest value of $\frac{t_i}{w_i}$.

We will denote the client that has the smallest value of $\frac{t_i}{w_i}$ to be client i at position k where $k > 1$.

Then, we can swap client i with the client one position before them (the client at position $k-1$). As shown above, swapping any two consecutive clients will only decrease the total weighted delay. So, swapping these two clients preserves the optimal solution. Hence, we can continue swapping client i with the client in the position directly before them until client i reaches the first position.

So, the client in the first position would be have the smallest value of $\frac{t_i}{w_i}$ as required.

Hence, the greedy choice property holds for the problem of finding an order that minimizes the total weighted delay.

(c) Based on the claim of part (b), give a greedy algorithm to determine an order that minimizes total weighted delay. Describe your algorithm in pseudocode.

Solution:

Algorithm 1: Determines the order of clients that results in the minimum total weighted delay

```
Function minWeightedDelay(t[1...n], w[1...n]):
    ratios[1...n]                                  ▷ Declare empty array of size n
    clients[1...n]                                 ▷ Declare empty array of size n
    For i from 1 to n
        ratios[i] = (t[i] / w[i], i)
    ratios = MergeSort(ratios)                     ▷ Sort ratios (ascending) by first value in the pair
    For i from 1 to n
        clients[i] = ratios[i].second              ▷ Second value is original client index
    Return clients
```

Assume MergeSort is a sorting algorithm (merge sort) that runs in $O(nlogn)$ time

(d) Analyze the worst-case running time of your algorithm.

Solution:

In the worst case:

-Initializing ratios[] and clients[] takes $O(n)$.

-Calculating the ratios of each client takes $O(n)$ (first loop).

-Sorting ratios[] takes $O(nlogn)$

-Storing the results in clients[] takes $O(n)$ (second loop).

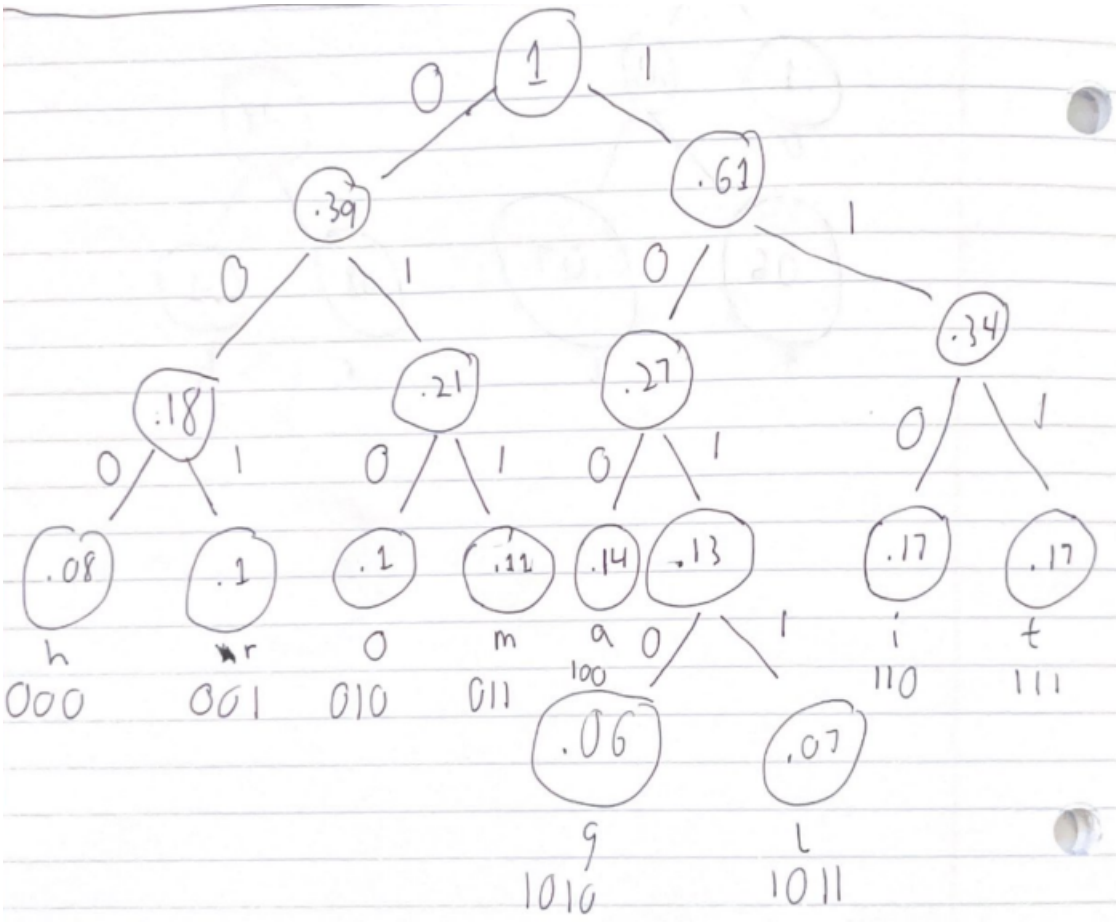
Since the dominant runtime is $O(nlogn)$, the total runtime of the algorithm is O(nlogn)

Problem 2. *(7 + 2 + 6 = 15 points) Huffman Code and Entropy*

(a) Given the following alphabet $\Sigma = \{a, g, h, i, l, m, o, r, t\}$ and corresponding frequencies, use Huffman's algorithm to compute an optimal prefix-free code. Represent the prefix-free code as a binary tree, with either 0 or 1 on each edge.

Letter	a	g	h	i	l	m	o	r	t
Frequency	0.14	0.06	0.10	0.17	0.07	0.11	0.10	0.08	0.17

Solution:



(b) What is the expected encoding length (average number of bits per element) of your Huffman code for this alphabet?

Solution:

.08(3) + .1(3) + .1(3) + .11(3) + .14(3) + .06(4) + .07(4) + .17(3) + .17(3) = $\boxed{3.13}$ average number of bits per element (expected encoding length)

Entropy is a mathematical formulation of the uncertainty and/or the amount of information in a data set. Consider a data set D consisting of n characters, each character independently chosen from a set C according to a specified probability distribution p. That is, for $c \in C$ and $0 \leq i < n$, the probability that the ith character of D is c is $p(c)$. Note that $\sum_{c \in C} p(c) = 1$. The entropy of data set D is then defined to be

$$n \sum_{c \in C} p(c) \log_2(1/p(c)).$$

Intuitively, the entropy measures the information-theoretic minimum number of bits needed to represent the data set.

(c) Calculate the entropy of a corpus, which contains the letters in the same frequencies as (a). View the frequencies as probabilities for the entropy calculation. (Note that the frequencies sum up to 1.) Compare the entropy with the compression achieved by Huffman coding and indicate whether Huffman encoding yields the best possible compression. More generally, do you think entropy captures the limit of how much that corpus can be compressed? Explain briefly in words.

Solution:

$\sum_{c \in C} p(c) \log_2(\frac{1}{p(c)}) =$

$0.14 \cdot \log_2(\frac{1}{0.14}) + 0.06 \cdot \log_2(\frac{1}{0.06}) + 0.1 \cdot \log_2(\frac{1}{0.1}) + 0.17 \cdot \log_2(\frac{1}{0.17}) + 0.07 \cdot \log_2(\frac{1}{0.07}) + 0.11 \cdot \log_2(\frac{1}{0.11}) + 0.1 \cdot \log_2(\frac{1}{0.1}) + 0.08 \cdot \log_2(\frac{1}{0.08}) + 0.17 \cdot \log_2(\frac{1}{0.17}) =$

$.397 + .244 + .332 + .435 + .269 + .35 + .332 + .292 + .435 =$

$\boxed{3.086}$

So, the entropy of a corpus that contains the letters in the same frequencies as (a) would be $3.086n$ where n is the number characters in the corpus.

Comparing the entropy with the compression achieved by Huffman coding shows that Huffman coding does not yield the best possible compression ($3.13 > 3.086$) However, the low margin of error between entropy and Huffman coding shows that Huffman coding is still a relatively efficient algorithm. Huffman coding is unable to achieve the entropy limit because it deals with bits, which are integers (i.e. each character can only be represented by an integer number of bits). Entropy, however, is not restricted by having to use integer numbers, so it is better able to capture the limit of how much a corpus can be compressed.

(d) **(0 bonus points)** Prove that if all the probabilities are powers of 2 (i.e., for every c there exists an $i \geq 0$ such that $p(c) = 1/2^i$), then the expected number of bits used in the Huffman encoding of D exactly equals its entropy.

Solution:

Problem 3. *(2 + 8 + 8 + 2 = 20 points) Project management under marginally decreasing benefits*

You are leading a major division in your software firm. You have just been promoted by the CEO and have been asked to manage a large number N of software projects. You have under your disposal M equally talented programmers and you are given the task of distributing these M programmers among these N projects.

After some careful thought, you figure out how much benefit i programmers will bring to project j. View this benefit as a number. Formally put, for each project j, you have pre-computed an array $A_j[0 \ldots M]$ where $A_j[i]$ is the benefit obtained by assigning i programmers to project j. (Assume that all of the programmers are equally competent, and it is ok to assign zero programmers to a project.)

You know that $A_j[i]$ is nondecreasing with increasing i; that is, $A_j[i] \geq A_j[i-1]$ for all $i \geq 1$. But you notice something interesting: the benefits satisfy the economically sound assumption that the marginal benefit obtained by assigning an ith programmer to a project is nonincreasing as i increases. That is,

$$\text{For all } j \text{ and } i \geq 1,\ A_j[i+1] - A_j[i] \leq A_j[i] - A_j[i-1].$$

In this problem, you will save your job by designing a greedy algorithm to determine how many programmers to assign to each project such that the total benefit obtained over all projects is maximized.

The running time of your algorithm should be at most $O(NM)$, enough to save your job and earn full credit. We note that a better running time of $O(N + M \log N)$ is achievable.

(a) Describe precisely what your algorithm is given as input and what it needs to output.

Solution:

inputs:

-The number of software projects N

-The number of programmers M

-A pre-computed array $A_j[0...M]$ where $A_j[i]$ is the benefit obtained by assigning i programmers to project j.

This algorithm will output an array programmers$[0...M]$ where programmers$[j]$ is the number of programmers assigned to project j

(b) Design a greedy algorithm that iterates from 1 to M and determines in each step the project to which the next programmer will be assigned. Describe your greedy algorithm in pseudocode.

Solution:

Algorithm 2: Determines the optimal (greedy solution) number of clients that should be assigned to each project

```
Function assignProgrammers(N, M, A_j[0...M]):
    programmers[1...N]                          ▷ Empty array of size N
    gain[1...N]                                 ▷ Gain in benefit from adding another programmer
    For j from 1 to N
        gain[j] = A_j[1] − A_j[0]
    For i from 1 to M
        bestGain = -1
        bestProject = -1
        For a from 1 to N
            If gain[a] > bestGain :
                bestGain = gain[a]
                bestProject = a
        programmers[bestProject] = A_bestProject[programmers[bestProject] + 1] −
          A_bestProject[programmers[bestProject]]
    Return programmers
```

(c) Justify the correctness of your algorithm. For full credit, prove using induction that for all i between 1 and M, the assignment obtained after the ith greedy step of your algorithm is a subset of an optimal assignment. (Your induction step will be an exchange argument.) For partial credit, your justification not need be formal, just convincing.

Solution:

Proof by induction on i:

Inductive Hypothesis:

Let H(i) be the statement: The greedy algorithm produces a subset of the optimal assignment for all $1 < i < M$.

Base Case:

When $i = 1$, the greedy algorithm assigns the first programmer to the project with the greatest marginal benefit ($A_j[1] - A_j[0]$).

In the optimal solution, the first programmer is assigned to the project that provides the highest increase in benefit.

So, the greedy algorithm and the optimal solution assign the first programmer to the same project. Hence, the greedy solution is a subset of the optimal solution for $i = 1$

Inductive Step:

We will show that for every $1 < i < M$, H(i) $\implies$ H($i + 1$). That is, when programmer $(i + 1)$ is assigned by the greedy algorithm, the resulting assignment is a subset of the optimal assignment. Assume H(i) holds, that is, after i assignments, the greedy algorithm produces a subset of the optimal solution.

Now, suppose that the greedy choice (highest marginal benefit) for programmer $(i + 1)$ does not result in a subset of the optimal solution. Hence, there is another project in the optimal solution for which when programmer $(i + 1)$ is assigned to it, the result is a higher overall benefit than in the greedy solution.

However, since assigning a programmer p to a project is nonincreasing as p increases, moving p from a project with a lower marginal benefit to a project with a greater marginal benefit cannot decrease the total benefit. Hence, we could create a more optimal solution by assigning programmer $(i + 1)$ to the project chosen by the greedy algorithm. This contradicts that the optimal solution was indeed optimal.

Hence, moving programmer $(i + 1)$ to the project chosen by the greedy algorithm will not result in a lower total benefit that the optimal solution. Therefore, H(i) is true for all $1 < i < M$: The greedy algorithm produces a subset of the optimal assignment.

(d) Analyze the running time of your algorithm.

Solution:

-Initializing gain takes $O(n)$

-Determining the best way to assign each programmer takes $O(NM)$ since we have a for loop from 1 to N inside a for loop from 1 to M.

-Since O(NM) dominates O(N), the runtime of this algorithm is $\boxed{O(\text{NM})}$

Problem 4. *(7 + 8 = 15 points) Coloring graphs and walking on them*

These two exercises are primarily to help you reason about graphs and traversing them.

(a) Call an undirected graph k-colorable if one can assign each vertex a color drawn from $\{1, 2, \ldots, k\}$ such that no two adjacent vertices have the same color. We know that bipartite graphs are 2-colorable.

Prove that if the degree of every vertex of G is at most Δ, then the graph is $(\Delta+1)$-colorable. (Hint: Iterate through the vertices, assigning them colors one by one.)

Next, show how to construct, for every integer $\Delta > 0$, a graph with maximum degree Δ that requires $\Delta + 1$ colors.

Solution:

To show that if the degree of every vertex of G is at most Δ, then the graph is $(\Delta + 1)$-colorable:

Start with an uncolored graph G where every vertex of G has a degree of at most Δ.

For each vertex v_i in G, since the degree of v_i is at most Δ, there are at most Δ vertices connected to v_i. Hence, there are at most Δ colors being occupied by the vertices connected to v_i. Since there are $\Delta + 1$ colors available, there is at least 1 color available to be assigned to v_i (by the pigeonhole principle). This process can be iteratively repeated for each v_i in G until each vertex in G is colored.

To construct a graph G with maximum degree Δ that requires $\Delta + 1$ colors (where $\Delta > 0$):

Let G be a graph of $\Delta + 1$ vertices where each vertex in G is connected to every other vertex. So, each vertex has degree Δ. Hence, the maximum degree of G is Δ. Also, since every vertex is connected to every other vertex, no two vertices can share the same color. Hence, each vertex must be assigned a unique color. Thus, there must be at least $\Delta + 1$ colors. So, G would be $(\Delta + 1)$-colorable.

(b) Describe an algorithm that takes as input an undirected graph $G = (V, E)$ and returns a path that traverses every edge of G exactly once in each direction. You need not present pseudocode, but your description should be clear, broken down into steps, and clearly explain how you build the path that is returned. Your algorithm should run in $\Theta(n + m)$ time in the worst-case, where m is the number of edges and n is the number of vertices. (Hint: Use depth-first search.)

Solution:

For the sake of the problem, assume that there exists a path that traverses every edge of G exactly once in each direction

In G, every vertex in V has a color as DFS progresses (from lecture 15, slide 23):

WHITE = undiscovered

GRAY = discovered, not finished (not done exploring it)

BLACK = finished (have found everything reachable from it)

Each vertex is initially colored WHITE

Begin by initializing a stack S of vertices and a list L of edges.

-S will be used to assist with DFS

-L will store the path that traverses every edge of G exactly once in each direction

Push the starting vertex onto S and color it GRAY (the starting vertex can be any vertex in G so long as there exists a path that traverses every edge of G exactly once in each direction. If G contains any vertices with an odd degree, then the starting vertex must be one of these vertices).

Next, perform DFS:

-While the stack is not empty:

-Peek at the top vertex u in S.

-Look for any edges from u that are available for traversal (any edge to a WHITE vertex)

-If there exists an edge $e = (u, v)$ where $e \in E$ and $u, v \in V$ that is available:

-Color v GRAY and push it onto S

If there does not exist an edge that is available for traversal:

-Add u to the beginning of L

-Color u BLACK

-Pop u from S

Once S is empty, return L

The worst-case runtime of this algorithm is $\Theta(n + m)$ because each edge and vertex is visited at most once. By coloring each vertex in G, we ensure that no edges are traversed unnecessarily.