CS 170 HW 6

Due **2021-10-11, at 10:00 pm**

1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, you must explicitly write none.

In addition, we would like to share correct student solutions that are well-written with the class after each homework. Are you okay with your correct solutions being used for this purpose? Answer "Yes", "Yes but anonymously", or "No"

2 Online Matching

Consider an unweighted, undirected bipartite graph $G = (L, R, E)$ where L and R are disjoint subsets of vertices, L denotes the left side vertices and R denotes the right side vertices. The maximum bipartite matching problem asks to provide a subset of edges of maximal size such that no two edges share a vertex. Intuitively, you can think about it as trying to create as many pairs as possible of vertices from L and from R.

Now, consider the following harder, *online* setting. In the online version, L is known, but the vertices in R arrive one at a time. When a vertex $u \in R$ arrives, its incident edges are also revealed, and at this moment, we must make a immediate and irrevocable decision to match u to one of its available neighbors in L or not to match it at all. The decision is irrevocable in the sense that once a vertex is matched or left unmatched, it cannot be later rematched (to another vertex).

The goal is to maximize the matching size.

(a) Consider the deterministic greedy algorithm, where for each incoming $u \in R$, if there is at least one available neighbor, u is matched to an arbitrary neighbor out of all the available neighbors. Let OPT denote the maximum possible matching size in the offline setting (*i.e.*, when the graph is fully known). Show that the greedy online algorithm obtains a matching of size at least half of OPT.
Hint: Consider an edge (l, r) in the optimal offline matching. Can we say anything about the number of edges adjacent to l or r in the matching obtained by the greedy algorithm?

(b) Show that no deterministic online algorithm can achieve strictly better than half of OPT on all possible inputs.
Hint: Construct two inputs, where if the algorithm does better than half of OPT in one of them, it must achieve at most half of OPT in another. Use the fact that the algorithm cannot see future. Keep the construction simple—four vertices are enough!

(c) *Insanely hard challenge yet worth no point*: Show that the following randomized algorithm achieves $1 - 1/e$ fraction of OPT on any input. Give a random ordering of the vertices in L. When $u \in R$ arrives, if u has an unmatched neighbor in L, then we choose the unmatched neighbor $v \in L$ that comes earliest in the random ordering. (Otherwise,

leave it unmatched.)
Hint: Don't spend too much time on it.

Solution:

(a) First, we will prove that for any $u \in L, v \in R$ that are connected, either u or v must be matched with some vertex after the greedy algorithm was run. Indeed, when the algorithm encountered v, there could be two cases: either it had no available neighbors, in this case u was already matched, or it had some available neighbors, and it was matched to one of the neighbors.

Now, consider the optimal matching, specifically, consider an edge (u, v) in this matching. Since u and v are connected by this edge, either u or v must be matched by the greedy algorithm. This shows that any edge in the optimal matching shares an endpoint with one of the edges in the greedy algorithm's solution.

Assume that the algorithm produced a matching of size k. Since each edge has two endpoints and each edge in the optimal solution shares an endpoint with one of the edges in the greedy algorithm's solution, this means that the optimal solution can have at most $2k$ edges.

(b) Suppose $L = \{i_1, i_2\}$ and $R = \{j_1, j_2\}$. Consider two possible inputs. In both of them, vertex j_1 arrives first and reveals that it is connected to both i_1 and i_2. Then vertex j_2 arrives and reveals that it has only one neighbor: in Input 1 this neighbor is i_1; in Input 2 it is i_2.

Notice that the maximum matching has size 2 in both of these inputs: j_2 can be matched to its only neighbor, whereas j_1 can be matched to the remaining element of L. Also notice that in both cases, this is the unique matching of size 2. Therefore, an online algorithm that seeks to select the maximum matching faces a predicament: first it must match j_1 to one of its neighbors, there is a unique choice that is consistent with picking the maximum matching, and there is no way to know which choice this is until later j_2 is revealed. Formally, on Input 1, if the algorithm makes the correct choice to match j_1 with i_2 and goes on to match j_2 with i_1, then it has to mess up on Input 2; that is, matching j_1 with i_2 necessarily leads to a matching of size 1 on Input 2. On the other hand, if the algorithm does the opposite, then it is just going to mess up on Input 1.

Hence, we conclude that for any deterministic online algorithm, we can find an input that causes the algorithm to select a matching of size at most 1, while the maximum matching has size 2.

Note that this proof is based on the fact that the algorithm is deterministic. Such and other limitations of deterministic algorithms is one of the reasons why later in the course, we will study randomized algorithms.

(c) The algorithm is known as RANKING, first proposed by Karp, Vazirani and Vazirani in 1990; in fact, two of them were and still are Berkeley professors. The initial proof is very complicated.

Over the years, the analysis has been simplified a lot, but this is still way beyond the scope of this course. We refer you to `https://arxiv.org/abs/1804.06637` or `https://www.cs.cornell.edu/~rdk/papers/RPDFinal.pdf`.

3 Twenty Questions

Your friend challenges you to a variant of the guessing game 20 questions. First, they pick some word $(w_1, w_2, ..., w_n)$ according to a known probability distribution $(p_1, p_2, ..., p_n)$, i.e. word w_i is chosen with probability p_i. Then, you ask yes/no questions until you are certain which word has been chosen. You can ask any yes/no question, meaning you can eliminate any subset S of the possible words with the question "Is the word in S?".

Define the cost of a guessing strategy as the expected number of queries it requires to determine the chosen word, and let an optimal strategy be one which minimizes cost. Design an $O(n \log n)$ algorithm to determine the cost of the optimal strategy.

Give a 3-part solution.

Note: *We are only considering deterministic guessing strategies in this question. Including randomized strategies doesn't change the answer, but it makes the proof of correctness more difficult.*

Solution:

This solution is inspired by the observation that in binary coding, each bit of a codeword we read further narrows the possible symbols being encoded, just like a question in the game above. This correspondence is made rigorous in the proof of correctness.

Main idea: Create a Huffman tree on $(w_{1\cdots n})$ with weights $(p_{1\cdots n})$ and return the expected length of a codeword under the corresponding encoding.

Proof of correctness: Note that any guessing strategy gives a prefix-free binary encoding of the words $(w_{1\cdots n})$, where each word w_i is encoded by sequences of yes/no answers which would lead you to conclude that w_i was chosen. This encoding is prefix-free because the game only ends when all words except one have been eliminated.

Additionally, any prefix-free encoding of the words can be made into a guessing strategy as follows. Let $x_n \in \{0,1\}^n$ represent the sequence of yes/no answers received on the first n questions, with 1 corresponding to yes and 0 corresponding to no. Then asking at step $n+1$ whether the word can be encoded by a string with the prefix $x_n \circ 1$ (i.e. the answers so far followed by a yes) will result in the final sequence of answers being a valid encoding of the chosen word.

In this correspondence, the expected code length equals the cost of a guessing strategy. Therefore finding an optimal strategy is equivalent to finding a prefix-free encoding of the words with minimum expected codelength, which is exactly what Huffman coding does. To get the final answer, we calculate the expected codelength of the optimal strategy, which can be done by DFS from the root of the Huffman tree.

Runtime: A Huffman tree can be constructed in $n \log(n)$ time (this is dominated by the time to sort the probabilities). The average codelength (and thus cost of the associated strategy) can be calculated in $O(n)$ time by DFS. Therefore the total runtime is $O(n \log(n))$.

4 Balloon Popping Problem.

You are given a sequence of n-balloons with each one of a different size. If a balloon is popped, then it produces noise equal to $s_{left} \cdot s_{popped} \cdot s_{right}$, where s_{popped} is the size of the popped balloon and s_{left} and s_{right} are the sizes of the balloons to its left and to its right. If

there are no balloons to the left, then we set $s_{left} = 1$. Similarly, if there are no balloons to the right then we set $s_{right} = 1$, while calculating the noise produced.

After popping a balloon, the balloons to its left and right become neighbors. (Note that the total noise produced depends on the order in which the balloons are popped.)

Design a dynamic programming algorithm to compute the the maximum noise that can be generated by popping the balloons.

Example:
Input (Sizes of the balloons in a sequence): ④ ⑤ ⑦
Output (Total noise produced by the optimal order of popping): 175

Walkthrough of the example:

- **Current State** ④ ⑤ ⑦
  Pop Balloon ⑤
  Noise Produced $= 4 \cdot 5 \cdot 7$

- **Current State** ④ ⑦
  Pop Balloon ④
  Noise Produced $= 1 \cdot 4 \cdot 7$

- **Current State** ⑦
  Pop Balloon ⑦
  Noise Produced $= 1 \cdot 7 \cdot 1$

- **Total Noise Produced** $= 4 \cdot 5 \cdot 7 + 1 \cdot 4 \cdot 7 + 1 \cdot 7 \cdot 1$.

(a) Define your subproblem.

(b) What are the base cases?

(c) Write down the recurrence relation for your subproblems. What is the runtime of a dynamic programming algorithm using this subproblem?

Solution:

(a) We want to form subtrees $C(i, j)$ representing the *maximum* amount of noise produced by popping balloons in the sublist $i, i+1, \ldots j$ first before the other balloons. The smallest subproblem is when there is only 1 balloon to pop. The size of the subproblem is the number of multiplications.

(b) Base case: When the size of sublist to pop out is only one balloon s_{popped}, return the noise $s_{left} \cdot s_{popped} \cdot s_{right}$. In addition, we need to initialize $s_0 = 1$ and $s_{n+1} = 1$ assuming the input is from 1 to n. In our algorithm we'll never actually pop these two balloons as they are just dummy balloons on the left and right.

(c)

$$C(i,j) = \max_{i \le k \le j} \{C(i,k-1) + C(k+1,j) + s_{i-1} \cdot s_k \cdot s_{j+1}\}$$

Justification: The leaves of the tree represent the last balloon being popped, so here, k represents the index of the last balloon being popped. Then we can recurse up, and at each level find the splitting point k that maximizes the value of the subtree (noise produced for that sequence). Runtime of this algorithm is $O(n^3)$ because there are $O(n^2)$ subproblems, each of which requires $O(n)$ time to compute given the answers to smaller subproblems.