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!

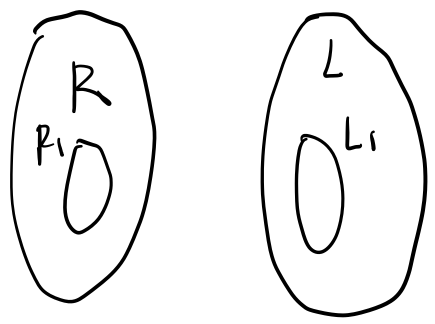
(c) *Insanely hard challenge yet worth no point*: Show that the following randomized algorithm achieves $1 - 1/e$ fraction of OPT on any input. Give a random ordering of the vertices in L. When $u \in R$ arrives, if u has an unmatched neighbor in L, then we choose the unmatched neighbor $v \in L$ that comes earliest in the random ordering. (Otherwise, leave it unmatched.)
Hint: Don't spend too much time on it.

1.

3037534676 Shenghan Zhang

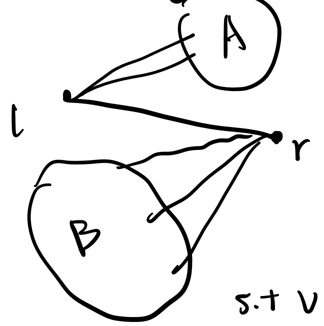
2.

(a)



Suppose the vertices forming the maximum pairs comes from R_1, L_1. Then $|R_1| = |L_1| = OPT$

For (l, r) that's in the offline match, the remove of (l, r) shouldn't generate more than 1 available edges.



Suppose l's neighbours are in Set A.

r's neighbours are in Set B.

Then there are 3 possibilities: 1) $B \subseteq L_1$, $\exists v \in A$ s.t $v \notin R_1$

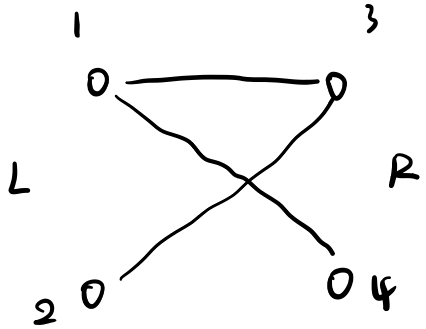
2) $A \subseteq R_1$, $\exists v \in B$ s.t $v \notin L_1$

3) $A \subseteq R_1$, $B \subseteq L_1$

Thus, to remove 1 edge (l, r), we take at most 2 vertices (Assume we choose l and remove r, for 1), 2) we can find another edge, for the other end of the edge, it's now used. Moreover, r can't be chosen any more. For 3), we can't take another edge, so only r can't be chosen any more)

Since we can redirect to optimal solution to the greedy algorithm by removing edges and adding edges and the cost is at most 2 vertices $\Rightarrow$ the matching size $\geq \frac{1}{2}$ OPT

b)



2 input : 3 4

4 3

OPT = 2

For 4, 3, since the only choice for 4 is (1, 4)

In 3, 4, we first pick (1, 3), thus 4 can't pick anyone. Since $1 \leq 2 \cdot \frac{1}{2}$, the statement is proved

(we will not reveal (1, 4) before reaching 4, (1, 3) (2, 3) are the same from 3's view)

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.*

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:

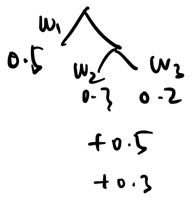
- **Current State** ④ ⑤ ⑦
  Pop Balloon ⑤
  Noise Produced $= 4 \cdot 5 \cdot 7$

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

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

3. ① Main Idea

Use Huffman Coding on $(w_1, \cdots, w_n)$ acording to $(p_1, \cdots, p_n)$, suppose we always put the branch with higher probability on left.



```
    Sum=0
    r = root
    opt=0
While r-> left :
      If sum==1 : break
    ask "Is the word in Left branch of r"
    opt += 1
   1) If yes, sum+=probability of right branch (stored in r-> right)
r = r-> Left

   2) Else, sum += probability of left branch (Stored in r-> left node)
r= r-> right
    If sum!=1 :
        return opt
    If Sum==1:
        return opt -1
```

② Correctness

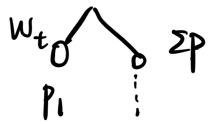
Sum represents the sum of posibility that we have excluded

r is the current root node of the branch

opt represents the number of quieries we made

The strategy is optimal since we always ask the branch with higher possibility and Huffman guarantees we have the shortest encoding for each node (word)

When sum = 1, we are done since we have block all possibilities. Moreover, we should reduce 1 query since the sum of possibily for right branch is 0

(That means after previous query, we get 

$p_1 > 0$, $\Sigma p = 0$. We ask "Is word W_t" and get yes and add p_1 to sum. Then sum = 1, but we don't need this query since $\Sigma p = 0$)

When r has no branch, we are done since r is the word.

③ Runtime

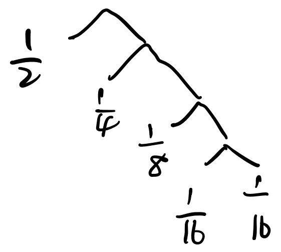
Construction for the tree using Huffman Coding

$O(n\log n)$

Maximum queries to make $= O(\#\text{ depth})$

$= O(n)$

Total runtime $= O(n\log n)$



- **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?

Define an array to hold integer input (nums)

(a) Let dp[i][j] represent the max noise from balloon i to balloon j

Goal: dp[1][n]

(b)

if $j=i$ return nums[i-1] * nums[i] nums[i+1]

(if index exceeds the range, use 1 instead)

if $j-i=1$ return max{ nums[i] nums[j] nums[j+1] + nums[i-1]nums[i] nums[j+1] , nums[i-1] nums[i] nums[j] + nums[i-1] nums[j] nums[j+1] } (2 situations, pop i first or pop j first)

(c) dp [left][right] = max{ dp[left][right] , nums[left -1] nums[k] * nums[right +1] + dp[left][k-1] + dp[k+1] [right] }

(left < k < right) Assume i to be the last balloon

(

Runtime :

There are 3 loops in algorithm

First loop for i (from length-2 to 0)

Second loop for j (from i+2 to length-1)

Third loop for k (from i+1 to j-1), to get the max value

The operation in loop for k is O(1)

∴ Total runtime = O (n^3)