Homework 2

Liwen Ouyang

February 24th 2020

Bipartite Matchings

The student is wrong. Here's a counterexample. Let the bipartite graph $G = V+E$ where $V = \{A, B, C, 1, 2, 3\}$ and $E = \{(A,1),(A,2),(B,2),(B,3),(C,2)\}$. It is obvious that there is only one perfect matching for G : $\{(A,1),(B,3),(C,2)\}$. Now we can construct preference lists for the vertices. Since any vertex puts its neighbors before non-neighbors. We can construct the following preference lists: $A : 2- > 1- > 3, B : 3- > 2- > 1, C : 2- > 1- > 3$ and $1 : A- > B- > C, 2 : A- > C- > B, 3 : B- > A- > C$. In this case, A prefers 2 than 1, and 2 prefers A than C. Therefore we have an instability $(A, 2)$, which invalids the student's argument, as GS algorithm will never give a matching that is not stable. Therefore it will never give us the matching $\{(A,1),(B,3),(C,2)\}$.

Problem 2

The algorithm is wrong. Let $S = \{8, 7, 6, 5, 4\}$. Clearly it can be partitioned into $A = \{8, 7\}$ and $B = \{6, 5, 4\}$, where the difference of the sum is 0. However, if we run the algorithm, we will end up with the partition $A' = \{8, 6\}$ and $B' = \{7, 5, 4\}$, which have a difference of sum of 4.

Problem 3

We will sort the jobs based on their starting time from early to late. First we put the starting time and ending time of the first job as an interval in to a stack. Then we check for every job whether it overlaps with the interval on the top of the stack. If they do, that means we can check them together in the interval on which they overlap. We will put this interval into the stack and proceed with next job. For the next job, we will check whether the job overlap with the previous interval we have, if it does, it means they can be checked together and we will update the interval to be the overlap. If not, that means this job cannot be checked together, we will put the job itself as an interval into the stack. We continue to do this until we run out of jobs, and in the end we will have all the intervals that we can choose our time points from in the stack.

procedure TIMEPOINTS(L) ▷ given list of tuples representing starting time s and finishing time f of jobs
  sort n jobs by their starting time in ascending order $j_1, j_2, \ldots j_n$. ▷ $O(n \log n)$ sorting
  initialize a Stack S to store the solution
  push the starting time and finishing time of j_1: (s_1, f_1) onto S
  for jobs j_i from j_2 to j_n **do**. ▷ : $O(n)$ loop
    $s_x, f_x \leftarrow S.peek()$
    if $s_i < f_x$ **then**
      S.pop()
      push (s_i, f_x) to S
    else
      push (s_i, f_i) to S ▷ : all $O(1)$ operation.
    end if
  end for ▷ in the S, we have all the intervals that we have to check once to check all jobs.

end procedure

The algorithm's run time is dominated by the sorting which is $O(n \log n)$. We want to show that our solution is correct using induction. If there is only one job, then its starting time and finishing time will be in the stack S, meaning the most optimal case is that we'll have to check once in the interval (s, f) which is trivially true. Now let's assume our algorithm produces an optimal solution for $n-1$ jobs. Let the interval on the top of the stack be (s_x, f_x)Now we add one more job, j_n to it. We want to show that the solution in the stack is still optimal. If (s_n, f_n) overlaps with the interval on the stack, meaning we will only update the last interval to be (s_n, f_x), which does not change the number of time points we need. Now since our solution is optimal for $n-1$ job, and we can check for n jobs without adding one more time point in this case. Then the solution for n must also be optimal. If j_n does not overlap with (s_x, f_x), it does not overlap with any intervals in the stack. That means we will have to check at least one more time for this job in the most optimal case. In the stack, another interval (s_n, f_n) will be pushed onto it, which agrees with the most optimal case. This completes our proof.

Problem 4

Applying greedy approach we set out to always pair the two elements from the two sets that have the smallest absolute difference. We will first put all possible pairing into a priority queue Q, which has insertion and deletion of $O(\log n)$. Each pairing (h_i, l_i) will have priority $|h_i - l_i|$.

procedure LEASTDIFFERENCE(H,L) ▷ Given list H and L that contains n numbers each
  Enqueue all elements of $H \times L$ into a priority queue Q. ▷ pairing and enqueue n^2 times:
  Initialize a list S for solution ▷ $O(n^2 \log(n))$
  while Q is not empty **do**. ▷ We will dequeue n times
    Q.dequeue() $\rightarrow (h_s, l_s)$
    push(h_s, l_s) onto S
    for all pairs $P = (h_i, l_i)$ in Q **do** ▷ we will delete $n^2 - n$ times
      if $h_s \in P \vee l_s \in P$ **then**
        delete P from Q ▷ Each deletion costs $O(\log n)$, which in total gives $O((n^2 - n) \log n)$
      end if
    end for
  end while
  S is the optimal solution. ▷ Finally we have $O(n^2 \log(n) + (n^2 - n) \log n) = O(n^2 \log n)$
end procedure

We will use inductive "greedy-stays-ahead"argument to prove the correctness. For the base case if we have only 1 pair, it is trivially true that the solution is optimal. Now let's assume for $n-1$ pairs, our algorithm gives the optimal solution. Now the algorithm choose another pair, based on our greedy algorithm, it is going to choose the one with the least difference, which combined with the $n-1$ pairs that is optimal by our hypothesis, should yield an optimal solution.

Problem 5

We will use a $n \times n$ array to represent the solution. Applying a greedy approach, we will first color the row that have the most number of black arbitrarily, decrement the column array accordingly. Then we will do it again for the second largest number in row and third and so on. If there's a tie, we will choose arbitrarily.

procedure COLOR(R,C) ▷ Given R and C as two arrays of integers
  Initialize a $n \times n$ array S to store solution with initial value of every entry as uncolored
  while R is not empty **do** ▷ $O(n)$ loop
    $r_{max} \leftarrow max(R)$
    $row = R.indexOf(r_{max})$
    delete r_{max} from R

```
        x = number of nonzero entries in C                    ▷ this is O(n)
        if x < r_max then
            Exit loop and declare no solution exists
        else
            for i from 1 to n do
                if C[i] > 0 and r_max > 0 then
                    color the cell S[row][i]
                    C[i] − −
                    r_max − −
                end if
            end for
        end if
    end while
    Return S
end procedure
```

This algorithm has a $O(n^2)$ running time, which is dominated by two nested $O(n)$ loop. The algorithm always terminate because we remove 1 element from R every time and the loop stops when R is empty. At the end of every outer while loop, the partially filled solution S always satisfies the constraint we removed from R and C, because otherwise the if branch will be executed, meaning we have insufficient number of blocks to be filled because we cannot fill x black blocks in y where $y < x$, and the algorithm will declare no solution exist. That is to say, in the end, our algorithm will either declare no solution exists or give a correct solution.

Problem 6

To begin with we'll prove a lemma.

Lemma 1. *Let G be a connected graph with n nodes, and T_1, T_2 are spanning trees of G. Let the number of shared edges between T_1, T_2, that is the edges of G that are both in T_1 and T_2, be m. Then the least number of steps to transform T_1 to T_2 is $n - 1 - m$.*

Proof. We'll use proof by induction. First let our base case be a graph that has 1 nodes. Obviously there is only one spanning of this graph and 0 edges. And therefore the least step to transform is $1 - 1 - 0 = 0$. In this case $n = 1$ and $m = 0$, which shows our lemma holds. Now assume this lemma holds for an arbitrary graph G, we want to show that after adding a node v and arbitrary number of edges from v to nodes of G, the lemma still holds for the extended spanning trees T_1' and T_2' of G'.

We know in order for T_1' and T_2' to span G', there must be an edge e_1 that connect v to T_1, and there must be an edge e_2 that connect v to T_2 to form T_1', T_2'. If e_1 is e_2, then we know T_1' and T_2' shares $m + 1$ edges. The least number of steps should not change because transforming T_1' to T_2' is essentially the same as transforming T_1 to T_2 as adding a same edge will not affect the steps. Now note that G' has $n + 1$ nodes, therefore we should have $n + 1 - 1 - (m + 1) = n - 1 - m$ number of steps, which is the same as our claim. This proves this case.

If e_1 is different from e_2. Now the number of edges shared by T_1' and T_2' is the same as before: m. Since from T_1, T_2 to T_1', T_2' we added one more pair (e_1, e_2) edges, we'll have to go one more step to transform T_1' to T_2', which is $n - 1 - m + 1 = n - m$. Now note that G' has $n + 1$ nodes, and T_1' and T_2' share m edges, then the least number steps: $n - m = n + 1 - 1 - m$, which agrees with our hypothesis. This completes the proof. □

Now that we have the proof, it is trivial to design the algorithm. Since essentially we just need to count the number of shared edges in T_1 and T_2. Let the edges relation of them be stored in two arrays E_1, E_2, where $|E_1| = |E_2| = n - 1$ We just need to have the number of elements that are both in E_1 and E_2. Using brute force approach we check each element in $E_1 \times E_2$ and return m as the number of pairs that have 2 same

edges. Then we compute the number :$n-1-m$, which is $O(1)$. This will give us a $O((n-1)^2)+O(1) = O(n^2)$ algorithm.