HW1 CS 180

ASHER CHRISTIAN 006-150-286

1. Exercise 3, P 22

Is there always a stable pair of schedules? No. Consider a schedule with 2 time slots and the shows for the respective networks.

$$A = \{a_1 = 4, a_2 = 2\}.$$
$$B = \{b_1 = 3, b_2 = 1\}.$$

There are precisely four pairs of schedules (S, T) as follows with their payoffs

$$S = (a_1, a_2)T = (b_1, b_2) \to (2, 0)$$
$$S = (a_1, a_2)T = (b_2, b_1) \to (1, 1)$$
$$S = (a_2, a_1)T = (b_1, b_2) \to (1, 1)$$
$$S = (a_2, a_1)T = (b_2, b_1) \to (2, 0)$$

There is clearly no pure nash equilibrium for if the payoff is $(2, 0)$ by inspection B must swap their shows to increase their payoff by 1. Similarly by inspection if the payoff is $(1, 1)$ A can increase their payoff to $(2, 0)$ by switching the shows of their network. Thus one network always has incentive to deviate and there is no stable matching.

2. Exercise 6, P 25

Let $S := \{s_i : i \in \{1, ...n\}\}$ Be the set of all ships and $P := \{p_i : i \in \{1, ..., n\}\}$ Be the set of all ports. For each s_i ship let c_i denote the calendar of that ship detailing where it will be on each of the m days. For each s_i using c_i create C_i the condensed schedule of the ship including only the days where the ship s_i is at port in the form of a list of ports such that the first element of the list is the first port to be visited and so on. I define the algorithm as so. The steps of the algorith to create a set of truncations are as follows:

1. set every ship unmatched to a port
2. pick an arbitrary unmatched ship s_i and attempt to match it with the first unasked port p_j
   There are two possibilities:
   (i) p_j is unmatched in which case s_i matches with p_j.
   (ii) p_j is already matched to s_k. If s_i visits p_j after s_k visits p_j then s_i matches with p_j and s_k becomes unmatched. Else s_i remains unmatched.
3. Repeat step 2. until either there are no unmatched ships or every ship has asked every port already

Date: 16.01.25.

The truncations are precisely the matches such that if s_i is matched to p_j then s_i truncates its schedule upon visiting p_j. I first must show that this algorithm results in a truncation or matching for each s_i and then show that this matching does not violate the requirement that no two ships can be in the same port on the same day.

1. *every ship matches with exactly one port*
   First note that once a port gets matched it is never without a match since it can only be matched to a different ship. Also note that not two ships can be matched to the same port or multiple ports to one ship. Lastly note that once a port has been asked to match it will stay matched forever as corollary from the first point.
   Assume for contradiction there exists a ship s_i that is not matched to any port. Then by the algorithm it must have asked every port to match at least once since every port appears within its calendar. Thus every port must be matched with a unique ship at this point but since s_i is unmatched there can be at most $n-1$ matched ships which contradicts the fact that all n ports are matched.
2. *No two ships can be in the same port on the same day*
   Assume again for contradiction that two ships s_i and s_j are at the same port p_k on the same day d and that it is the first such day that they violate this rule. This can only be the case if either s_i or s_j is currently truncated. This is because we can assume that the regular schedules of the ships contain no days in which two are at the same port. Assume without loss of generality that s_i is truncated. Additionally note that both ships cannot be simultaneously truncated at this time since each ship is matched with a unique port. So s_j must be matched with a port that it visits after p_k. This implies that during the algorithm, s_j attempted to match with p_k since the algorithm requires ships to ask in order of their visit. There are two possibilities, either (s_j, p_k) were matched and later unmatched or p_k was alreday matched to s_l. In the first case s_i would never be able to match with p_k because it visits p_k before s_j and all future matches of p_k must visit even later. In the second case s_i also would not be able to match with p_k because it would have already matched to a ship with a later date and if $s_l = s_i$ then p_k would have matched with s_j since s_j visits after s_i. This leads to a contradiction since none of the possibilities follow the logic of the algorithm and thus the rule must never have been violated.

3. Exercise 4, P 67

I propose the following order

1. $2^{\sqrt{log(n)}}$
2. $n(log(n))^3$
3. $n^{\frac{4}{3}}$
4. $n^{log(n)}$
5. 2^n
6. 2^{n^2}
7. 2^{2^n}

I will use the fact that if $f(x)$ is $O(g(x))$ then $log(f(x))$ is $O(log(g(x)))$ First to show 1. is order 2.

$$n(log(n))^3 > 2^{\sqrt{log(n}} \to log(n) + 3log(log(n)) > \sqrt{log(n)}log(2).$$

Verifiable since $log(n) > \sqrt{log(n)}$ for large n and for all future cases assume the inequality holds under the assumption of large n. now 2. < 3. we know by theorem 2.8 that $log_b(n)$ is $O(n^x)$ for all $x > 0, b > 1$ and so

$$n(log(n))^3 < n * (n^{\frac{1}{9}})^3 = n^{\frac{4}{3}}.$$

Further

$$n^{\frac{4}{3}} < n^{log(n)}.$$

whenever $log(n) > \frac{4}{3}$ For 4. < 5.

$$log(n^{log(n)}) = (log(n))^2 < (n^{.5})^2 log(2) = nlog(2) = log(2^n).$$

and so

$$n^{log(n)} < 2^n.$$

The last two are trivial to prove since by inspection $n < n^2 < 2^n$

4. Exercise 3, P 107

In the book it has already been shown that the algorithm for producing a topological ordering of a DAG operates in $O(m + n)$ time. I propose to use the same algorithm with only a slight modification. The original algorithm is as follows If

Algorithm 1 To compute a topological ordering of G :

```
Find a node $v$ with no incoming edges and order it first
delete $v$ from $G$
Recursively compute a topological ordering of $G - \{v\}$ and append this order after $v$
```

this algorithm completes successfuly it computes a proper topological ordering of G. The only requirement for the successful completion of this algorithm is the existence of a node v with no incoming edges. It has also been shown in the book that if there is no such v satisfying this property that there exists a cycle in G. Thus modifying the code as follows I claim that this algorithm meets the specifications. Firstly if the graph is a DAG then the algorithm works because it doesnt change anything from the base case when the graph is a DAG. If the graph is not a dag then the first if statement will not always execute and at some point the else statement will execute at this point there is no node with no incoming edges and so by the textbook there is an edge that can be found in m steps. I find this edge in m steps and return the cycle. If the graph is a DAG this algorithm has been shown to run in $O(m + n)$ time. If the graph is not a DAG then $O(m + n)$ operations occur before the else statement executes. After which $O(m)$ operations occur. So the algorithm is $O(m + n)$

Algorithm 2 To compute a topological ordering of G or return a cycle:

```
if There is a node $v$ with no incoming edges then
    order it first
    delete $v$ from $G$
    Recursively compute a topological ordering of $G - \{v\}$ and append this order after $v$
else
    Pick an arbitrary node $v$ and arbitrarily take an edge backwards from $v$ recording the node visited in a linked list repeating $|G| \leq m$ times and recording times visited each node stopping short if visit the same node twice
    Find a duplicate element and return the traversal back from that group element until it is reached again returning a cycle
end if
```

5. Exercise 5, HW

1. *Prove (by induction) that sum of the first n integers $(1+2+...+n)$ is $\frac{n(n+1)}{2}$* For $n=1$ this can be verified immediately. assume it holds for the sum up to n integers. then
$$\sum_{i=1}^{n+1} i = \sum_{i=1}^{n} i + n + 1 = \frac{n^2+n}{2} + \frac{2n+2}{2} = \frac{n^2+3n+2}{2} = \frac{(n+1)(n+2)}{2}.$$
Thus proving the inductive step.
2. *What is $1^3 + 2^3 + 3^3 + ... + n^3$? Prove by induction* For $n = 1$ the following holds
$$\sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}.$$
This holds for $n = 1$ assume it holds up to $n-1$ then the sum
$$\sum_{i=1}^{n} i^3 = \sum_{i=1}^{n-1} i^3 + n^3 = \frac{n^4 - 2n^3 + n^2}{4} + \frac{4n^3}{4} = \frac{n^4 + 2n^3 + n^2}{4} = \frac{n^2(n+1)^2}{4}.$$
And so the inductive step holds

6. Exercise 6, HW

Given an array A of size N. The elements of the array consist of positive integers. You have to find the largest element with minimum frequency I propose the following algorithm

1. Initialize an empty hash table mapping positive integers values to positive integer frequencies
2. Pass through the array one time updating the hash table to reflect the frequency of each element
(i.e.) checking if the element is in the hash table. If it is update the frequency by one if not set it and set the frequency to 1.
3. Pass through all key-value pairs in the hash table keeping track of two variables minf and maxn initializing them to be the frequency and value of the first pair respectively.

For each pair if the frequency < minf: set minf = frequency, maxn = value. Else if frequency = minf and n > maxn: set maxn = n

4. return maxn This algorithm operates in $O(N)$ time because it passes through the N array once doing each operation in $O(1)$ time generating a hash map with $O(N)$ elements and passes through the key value pairs of that hash map once each time doing an $O(1)$ operation. Assume for contradiction that the algorithm works that is the algorithm returns n when it should have returned m. There are two cases

$$(i) \quad m > n \quad \text{and} \quad f(m) = f(n).$$
$$(ii) \quad f(m) < f(n).$$

Note first that the algorithm finds the element of minimal frequency for it checks the frequency of each element and if it is less than the previously recorded minimum it changes the minimum and it checks every element So the first case is the only one that could happen. If the algorithm checked the n key value pair first. upon checking the m key value pair it would overwrite n with m because $m > n$ and they have the same frequency. If it checked m first it would not overwrite m with n because $n < m$ but they have the same frequency. Thus this contradiction cannot occur and the algorithm works.