Stav Yosef, 316298876

Advanced Algorithms - Homework 3

Question 1

A

Assume we can solve this problem in polynomial time.
Let G' be a graph with the same vertices and edges but $a(v') = 1$ and, $b(v') = 0$, the creation of G' is polynomial time.
If we can solve this problem in polynomial time hence, we can solve the **vertex cover** problem in poly time which is NP-complete it, contradiction! this problem is np-hard.
Why it's working? for each vertex v', $a(v') > b(v') = 0$, therefore we "blocking" the opportunity for any algorithm to choose $a(v')$ over $b(v')$ → any algorithm will prefer to use as many as it can $b(v')$ values (v' is not in the vertex cover) and least $a(v')$ values (v' is in the vertex cover).

B

We will choose random vertex v and run BFS in order to get the order of the tree.
We'll travel in post order and for each vertex v we apply:
If any vertex has no children then $M_{in}[v] = a(v)$ and $M_{out}[v] = b(v)$,
Else

1. $M_{in}[v] = a(v) + \sum_{c \in childrens(v)} min(M_{in}[c], M_{out}[c])$
2. $M_{out}[v] = b(v) + \sum_{c \in childrens(v)} M_{in}[c]$

The optimal value is $min(M_{in}[v], M_{out}[v])$ (where v is the random we chose above)

Correctness

Any vertex could be in or out from the vertex cover, for each of the vertices we calculate what happens if any vertex in or out.

1. If a vertex is in then its value a(v) is in the vertex cover and we can either include or exclude each of his children, we are adding the option which adds least weight.
2. If a vertex is not in, then its value b(v) is in the vertex cover then we must add all his children into the vertex cover.

In the end we choose the least weight of the root vertex.

C - Best result is 25

	a	b	c	d	e	f	g	h	i	j	k
order	1	2	4	9	3	5	11	10	8	6	7
M_{in}	3	-5	3	15	1	8	25	12	11	4	1
M_{out}	2	7	3	33	-2	2	35	5	1	5	6

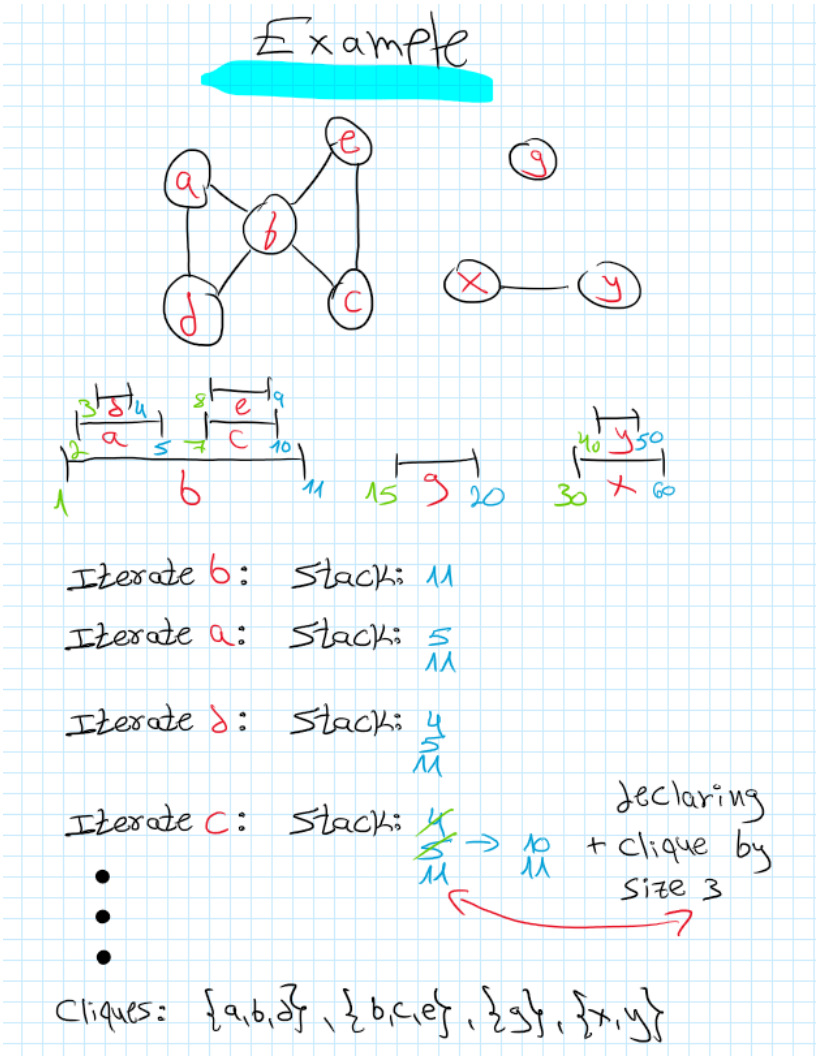
Stav Yosef, 316298876

Question 2

Algorithm

1. Build a laminar family of intervals as following:
    a. initiate $t = 0$
    b. travel recursively through each of the trees, start from the root, when reaching a new node (include the root of the tree) add new $interval$ with $start\ time = t$ and increase t by 1. when there are no more children to travel into, set the $end\ time$ of the node's $interval$ to $end\ time = t$ and again increase t by 1. $O(|Forest|)$
2. Run over the $intervals$ (they are sorted). For each $interval$ add its $end\ time$ to a stack, if the $start\ time$ is bigger than the first element (top element) in the stack, declare on the new clique and start to pop all the elements in the stack that smaller than the $start\ time$. $O(|S|)$
3. **Note -** Equivalent algorithm is to perform DFS from each of the roots to each of the leafs and declare a clique to each of the paths (root to leaf)

Overall, $O(|S|)$



Stav Yosef, 316298876

Correctness

We are given a forest of rooted trees and each vertex corresponds to an interval and v_1 is an ancestor of a vertex v_2 if and only if $I2 \subseteq I1$ therefore we know for sure each vertex's interval his children and their children intervals are contained in the vertex's interval. This way we can created a sorted laminar family.
Given a laminar family therefore we know either they don't intersect at all or one of them is contained in the other. The algorithm works correctly because we are creating a clique with the most vertices possible. We know the clique is with the most vertices possible because we keep adding vertices to the clique until $l_{k+1_{start}} > l_{k_{end}}$. If $l_{k+1_{start}} < l_{k_{end}}$ then $l_{k+1} \subseteq l_k \rightarrow k, k+1$ in the same clique.
This algorithm will produce us minimum clique cover because we are creating cliques biggest as possible so there isn't a clique that contained in another clique.
Complexity time - creating laminar family of intervals is $O(|FOREST| = s)$ and creating minimum clique cover is $O(s)$ because over each interval we are executing $O(1)$ commands, where $s = |S|$.
Overall, $O(s)$

Stav Yosef, 316298876

Question 3

Assume G is a bipartite graph whose vertices can be divided into two disjoint and independent set M (morning activities) and A (afternoon activities). Each edge between M and A is an activity selection that a worker made.

Rules

1. For each $v \in M$ and $\deg(v) \geq k_2 + 1$ must be in the solution → add v into morning activities that Oren selecting (only if $k_1 > 0$). Remove all edges associated with v and decreased k_1 by one. **Justification -** If v is not in the solution that's mean we need to choose at least $k_2 + 1$ activities in A because we need to make sure at least each worker receives one activity (Morning/Afternoon).
2. For each $v \in A$ and $\deg(v) \geq k_1 + 1$ must be in the solution → add v into afternoon activities that Oren selecting (only if $k_2 > 0$). Remove all edges associated with v and decreased k_2 by one. **Justification -** Same as rule 1 justification.
3. If $k_1 = 0, k_2 = 0, |E| > 0$ return False. **Justification -** If we can't choose more activities and there are still workers without at least one activity there is not solution.

Kernel Size

If we reached this far we know for sure that each of the vertices in the morning their degree is maximum k_2 and the vertices in the afternoon their degree is maximum k_1 therefore, in the worst case we can have $k_1k_2 + k_2k_1 = 2k_1k_2 \leq (k_1 + k_2)^2$ edges and maximum $(k_1k_2 + k_1) + (k_2k_1 + k_2) = 2k_1k_2 + k_1 + k_2 \leq (k_1 + k_2)^2$ vertices

Algorithm

1. Iterate each of the vertices that have degree bigger than 0 (1 and above), for each of them apply the rules. This cost us at maximum run though $2n$ vertices and n edges → complexity time is $O(n)$.
2. Perform brute-force on our kernel which has at maximum $(k_1 + k_2)^2$ edges and vertices, $f(k_1 + k_2) = \left(((k_1 + k_2)^2)^{k_1+k_2}(k_1 + k_2)^2\right)$.

Overall complexity time - $O\left(((k_1 + k_2)^2)^{k_1+k_2}(k_1 + k_2)^2 + n\right) = O(f(k_1 + k_2) \cdot n)$.

Stav Yosef, 316298876

Question 4

Processing time	100	1	1
Release time	12:00	13:00	13:00

$SPT - 100 + 101 + 102 = 303$
$Optimal - 2 + 3 + 103 = 108$

$\mathrm{SPT} > 2 \cdot \mathrm{Optimal}$

When adding 1 to each of the jobs this causing a "snowball" effect, the first job adding 1 to the new solution, the second job on the same machine adding 2 and so on…
We have 2 machines and each of the machines handles k jobs (We know that because there isn't release time and the jobs being sorted before, $n = 2k$).
In general case with m machines, $\mathrm{z'} = \mathrm{z} + \left(1 + 2 + \cdots + \frac{\mathrm{n}}{\mathrm{m}}\right) \cdot \mathrm{m}$, and in our case with 2 machines, $\mathrm{z'} = \mathrm{z} + \left(1 + 2 + \cdots + \frac{\mathrm{n}}{2}\right) \cdot 2$

Stav Yosef, 316298876

Question 5

I read and understood

Yes, the algorithm $\mathrm{A'_k}$ can replace the algorithm $\mathrm{A_k}$
We know that the first k jobs are scheduled optimally.
All the machines in the algorithm $\mathrm{A'_k}$ are busy at the time $\mathrm{job_j}$, that's what determine the makespan → this is a property of any list scheduling without intended idle.
The $\mathrm{job_{k+1}}$ starts its processing in the non increasing sorted order because of step b.