1. (15 points) **(Finding Good Routes for DHL Couriers)** We are given a graph $G = (V, E)$ with n vertices and edges, each edge e has a non-negative cost $c(e)$. The DHL van originates at a source vertex s, and the goal is to compute a tour which visits every vertex in V of minimum total cost and finally returns to the source s. You may traverse the same edge many times provided you pay its cost as many times.

   (a) (5 points) What is the natural decision version of this problem? Show that it is NP-hard by reducing from the Hamilton Cycle problem (which known to be NP-hard).

Solution:

Proof. The current problem is as follows: Given a graph $G = (V, E)$ with n vertices and edges, each edge e has a non negative cost $c(e)$. Let C be the cost of all the edges on the walk from the source s, visiting each vertex in G and going back to the source s. We have to minimise C in the given problem.
The decision version of the problem is: Given a $C', G = (V, E), c(e) \forall e \in E$, we have to find if there exists a walk that starts from some source vertex s, visits every vertex and finally resturns to the source s such that the cost of all edges in the walk is lesser than equal to C'. We ouput yes if a walk exists such that $C \leq C'$, and no otherwise.
Let us assume that there exists a polynomial time algorithm for our decision version. Given a graph G, to find if there exists a hamiltonian cycle within that graph, we consider the same graph with $c(e) = 1, \forall e \in E$. Consider $C' = |V|$ and apply the decision version poly time algorithm of the problem on this problem. If the answer received is yes, means there is a walk with cost as $|V|$. Now as all costs have been set to 1, means that there are $|V|$ edges on the walk. Also as our decision problems covers all vetrtices and returns back to the source s means that we cannot have less than $|V|$ cost, the fact that the decision version replies yes implies that the walk is of cost $|V|$ such that each vertex is covered exactly once. As we have found a cycle that visits each vertex exactly once and covers all vertes, hence we have found a hamiltonian cycle. If the decision version answers no, means there is no walk that has less than equal to $|V|$ edges. Means there is no walk with $|V|$ cost. A hamiltonian cycle has $|V|$ cost, since there doesn't exist such a walk implies there is no Hamiltonian cycle. Hence we have shown the problem of Hamiltonian cycle to be poly time solvable. But as it is known that Hamiltonian cycle is NP-hard, decision version of our problem is also NP-hard. □

   (b) (10 points) Devise a good approximation algorithm for this problem by using the minimum spanning tree as a sub-routine. What is the approximation factor?

Solution:

Proof. We consider any tree for the moment and find out an algorithm that can

find a walk on the tree that covers all vertices in our tree. Let T be a spanning tree on the graph G. Let $C'/2$ be the cost of all the edges on the tree, we will first show that we can find a walk with cost C' that covers all vertices in the tree.
We perform a depth-first search (on no fixed destination) on the root vertex. As each vertex gets looked at in the depth first search algorithm we add it into our walk. Depth first search visits every vertex in a connected graph, the algorithm ends when every vertex has been marked visited by the DFS. As our walk also adds vertices as they are visited, the walk also contains all vertices. Algorithm 1, is applied. We note that the walk is a valid walk because only adjacent vertices are picked in our walk, we visit the children one by one(4) and go back to the parent to visit each of the remaining children(10). We note that the number of vertices in the walk are equal to $2|V|$(vertices get added only twice 4,10). And each edge also is added only twice as (u to v added on 4 and then v to u is added back on 10). Hence the cost of the walk is twice the sum of all edges on the graph i.e $C = 2\sum_{i=1}^{|E|} c(e_i)$.
We now create a 2-approximation for the giving problem by running the minimum spanning tree algorithm and applying our modified dfs algorithm to obtain a walk, we claim that this walk is a 2 approximation. Note that the minimum spanning tree is the minimum sum of $|V| - 1$ edges that covers each vertex exactly once, the most optimal walk must have a cost strictly greater than this cost (there is one more edge the one that makes the walk return back to the original source, hence there are —V— edges atleast in the walk). Let T be the minimum spanning tree.

$$AlgCost = 2 * \sum c(e) | e \in E(T)$$
$$OptCost > \sum c(e) | e \in E(T)$$
$$\Rightarrow Algcost < 2 * Optcost$$

Hence the given algorithm is a 2-approximation algorithm. □

Algorithm 1 DFS with walk

```
1: Initialize visited of all vertices as false and walk as an empty array
2: procedure DFS(u)                                          ▷ The vertex being visited
3:     visited[u] = True
4:     walk.append(u)                                        ▷ The vertex being visited is added
5:     for  v = all neighbours of u do
6:         if visited[v] = True then
7:             continue
8:         end if
9:         DFS(v)
10:        walk.append(u)  ▷ u is appended to maintain a valid walk, i.e walk goes back to
   the rooted vertex to visit remaining children
11:    end for
12:    return
13: end procedure
```

2. (30 points) **(To be or not to be NP-hard?)** We now look at some basic problems and study their complexity. If they are NP-hard, you will design good approximation algorithms for these problems, and mention any non-approximability results you can derive.

   (a) (10 points) You are given n integers in the range $[1, n]$ and a target B. The goal is to check if any sub-set of them add to exactly B.

Solution:

Proof. Let S be the set that contains n integers. This is a poly time solvable problem. We provide a dynamic programming solution for the problem that runs in $O(n^3)$. We first note that the maximum sum possible is n^2. The sum of n elements less than equal to n, has to be less than equal to n^2. We define to be a 2-D array such that the number of possible subsets in the set $\{S_1, S_2, \ldots, S_i\}$ whose sum is equal to j is given by $dp[i][j]$. The recurrence relation is as follows,

$$dp[i][j] = 0 \qquad \text{We initialise dp[i][j] = 0}$$
$$dp[i][j] = dp[i][j] + dp[i-1][j] \quad \text{if } S_i \text{ is not chosen in the subset considered}$$
$$dp[i][j] = dp[i][j] + dp[i-1][j-S_i] \quad \text{if } S_i \text{ is chosen in the subset considered}$$

We initialise the state with $dp[1][0] = 1$ and $dp[1][S_1] = 1$ and all other $dp[1][j] = 0 | \forall j,\ j \neq 0\ \&\ j \neq S_1$.
The number of states in the dp are $O(n) * O(n^2)$, that is of order $O(n^3)$. Each state takes $O(1)$ time in computation, i^{th} stage is computed when elements of the $(i-1)^{th}$ stage have been precomputed. Hence in $O(n^3)$ time the whole table is computed.
If $dp[n][B] \geq 1$, we report that a sub-set has been found otherwise there exists no subset. Hence polytime solvable. □

   (b) (20 points) You're given a graph $G = (V, E)$. The goal is to choose a set $S \subseteq V$ of at most k vertices so that for every vertex $v \in V$, either $v \in S$, or there exists a neighbour w of v (i.e. $(u, v) \in E$) such that $w \in S$. In the optimization version of this problem, we want to minimize the size of the set S we select.

Solution:

Proof. The given algorithm is the decision version of the well known dominating set problem. We show NP-Hardness first and then move on to a greedy algorithm that is an approximation algorithm of $\ln \Delta + 2$.
To show NP-Hardness we show a reduction from the set cover problem. Consider a set cover problem with the universe U and the set containing sets of the

elements in U to be S.

$$U = \{e_1, e_2, \ldots, e_n\}$$
$$S = \{S_1, S_2, \ldots, S_m\} \qquad |S_j \subseteq U$$

Now we construct a graph $G = ((U, S), E)$ as follows:

$$(S_i, e_j) \in E \Leftrightarrow e_j \in S_i$$
$$(S_i, S_j) \in E \mid \forall i, jj \neq i \qquad \text{S is a clique in G}$$

Now we assume that the dominating set algorithm is poly time solvable, we run the decision version of the algorithm with k as the parameter on this graph.
(a) Lets say the answer to the decision problem be yes. Let the set we recieve with k vertices have some vertices in U and some in S and is called D. We first show that we can replace it with vertices entirely in S and that also is a valid answer to the decision version of the problem. Construct $D' \subseteq S$, by replacing every element $e \in D \cap U$ with any element $s \in S$ such that $(e, s) \in E$. We argue that if D is a dominating set D' is also a dominating set. Since there exists atleast one vertex of $D' \in S$, dom(D') covers S. Lets assume that it some vertes isn't dominated by D'. That vertex must belong in U as dom(D') covers S. Since D is a dominating set, it must have been covered by dom(D), as it isn't being covered by dom(D') means u is one of the elements that were replaced in D or is a neighbour of one of the elements that were replaced. u cannot be a neighbour as only elements in $D \cap U$ were replaced, and there is no edge in our graph that joins two vertices in U. This implies that u is one of the elements that were replaced, but for each element that was replaced we add a neighbour of it in D'. Hence this is also covered by dom(D'). As we cannot find a vertex, implies that $dom(D') = dom(D)$. Also $|D'| \leq |D|$ as only one element was added for each removed element. A newly added element might already exist in D, hence the less than sign. Now we have a set D' such that it is a subset of S. We claim that this is the set cover for the original problem that we started with. This is trivial to see from the construction because every element in U is dominated by some element in D' which means it is connected to some element in D' which means that it is covered by some element in D'. Since $D' \subseteq S$. We have found a set covering.
(b) Now we prove that if the decision version answers no, then there is no set cover. We show the contrapositive statement that if there exists a set cover then the decision version answers yes. Let the subset of S that covers U and of size at most k be X. We choose the same set of vertices in G. We claim that every vertex is covered in dom(X). As there exists one vertex in $S \cap X$, the whole set S is covered in dom(X). Also all elements in U are covered by dom(X) because of the construction of edges.
Hence by running dominating set algorithm on the graph we have found out if

the set cover decision problem is poly time solvable or not. Hence dominating set is NP-Hard.
We suggest a greedy algorithm for the approximation. The greedy algorithm is as follows: Among all vertices that have not yet been chosen, add that vertex that causes the maximum increase in the amount of nodes covered by the dominating set chosen uptill now. Repeat the above process till the whole graph is dominated by the chosen set.
We show that this algorithm is a $\ln \Delta + 2$ approximation. The proof is based on an amortized analysis. Let us consider the optimal dominating set. Assign each vertex that is either part a neighbor or the vertex itself to some vertex in the dominating set, do that for each vertex. Hence we get some centres as subset of the considered optimal dominating set. In each round of the algorithm we increase our cost by 1. Instead of increasing each round cost by 1, we could introduce costs to each vertices that are covered. Let n_i be the number of elements covered in the i^{th} round by the dominating set. We give a cost of $1/n_i$ to each vertex that was covered on this round. We consider a centre $v*$. Let u be the vertex added on a round of our greedy algorithm. We know that the gain in adding u is greater than equal to gain in adding $v*$, hence the max cost for each vertex is $1/(deg(v*) + 1)$. Similarly in the next stage max cost is $1/deg(v*)$. After all rounds the cost is less than a harmonic in $(\ln \Delta + 2)$, we skip the entire specifics as the question only asks us to mention any non-approximability results. □

3. (30 points) **(Restricted MAX-CUT)** Consider MAX-CUT with the additional constraint that specified pairs of vertices be on the same/opposite sides of the cut. Formally, we are given two sets of pairs of vertices, S_1 and S_2. The pairs in S_1 need to be separated, and those in S_2 need to be on the same side of the cut sought. Under these constraints, the problem is to find a maximum-weight cut.

   (a) (10 points) Firstly, sometimes there may not even be any feasible solution. Show that you can check if there exists feasible solutions efficiently in polynomial time.

Solution:

Proof. We show here that finding an existence of a solution is equivalent to solving a version of the two sat problem. All pairs in S_1 need to be separated, let (u, v) be a pair in S_1. We assume that for each vertex we are assigning a true or false value denoted by x_u and x_v respectively. A true value indicates the set being present in one part of the cut and false indicates presence in a separate part. Since pairs need to be separated, we observe that the xor of the two variables must be true, writing this in 2-SAT form.

$$(x_u \vee x_v) \wedge (\neg x_u \vee \neg x_v)$$

For each such pair we and all the constraints to get a 2-SAT satisfiable problem. Similarly for pairs in S_2 they must belong in the same side of the max cut, for them the satisfiability condition is:

$$(x_u \vee \neg x_v) \wedge (\neg x_u \vee x_v)$$

Combining all constraints and solving the equivalent 2-SAT problem in polynomial time we can check for existence in poly time as 2-SAT is poly time solvable. □

(b) (20 points) Now, if there is a feasible solution, show how to adapt the algorithm in class for MAX-CUT to get approximation algorithms for this problem. Explain the changes you'll make for the SDP discussed in class.

Solution:

Proof. We begin by defining the max cut SDP.

$$\text{maximize: } \sum_{ij \in E} w_{ij} \frac{(1 - v_i.v_j)}{2}$$
$$||v_i|| = 1$$
$$v_i \in R^n$$

The additional constraints can be represented as follows:

$$\forall (i,j) \in S_1 \Leftrightarrow v_i.v_j \leq 0$$
$$\forall (i,j) \in S_2 \Leftrightarrow v_i.v_j \geq 0$$

We then optimize the SDP under these conditions. The changes that we would make for the SDP are as follows: The algorithm just assigns a random gaussian vector and then assigns the vertices to the two classes based on the inner product. As the SDP we solve already satisfies the set of constraints. We now need to assign the random gaussian vector so that it doesn't violate our constraints. □

4. (25 points) **(Of Multi-terminal Flows and Cuts)** You should recall the famous *max-flow min-cut theorem* which ascertains the following beautiful theorem: given a undirected graph $G = (V, E)$ where each edge has unit capacity, and a source vertex s and sink vertex t, consider the two quantities: Let $F_{\max}$ denote the maximum flow which s can send to t while respecting edge capacities; let $C_{\min}$ denote the minimum number of edges crossing any partition of the form $(S, V \setminus S)$ where $s \in S$ and $t \in V \setminus S$. Then $F_{\max} = C_{\min}$. In fact, one such proof of this theorem is by using LP duality. In this question, we will study the same from a more practical setting.

In this question, there are many source,destination requests of the form (s_1, t_1), (s_2, t_2),

$\ldots, (s_k, t_k)$ where s_1 wants to establish a flow to t_1, s_2 wants to send some flow to t_2, and so on. The goal is to maximize the total throughput, i.e., establish some λ_i flow between s_i to t_i such that $\sum_i \lambda_i$ is maximized. Of course, the total flow going through any edge (both directions combined) should be at most its capacity of 1. You may assume that all sources and sinks are distinct vertices (i.e., no s_i is the same vertex as another s_j or t_j).

We now state below, the Linear Program which captures this problem. The set $P_{(s_i,t_i)}$ is the set of all simple paths from s_i to t_i, and every element $p \in P_{(s_i,t_i)}$ is a simple path from s_i to t_i. Ignore the fact that there are exponentially many variables — we are not trying to implement this in a computer, we are only using LPs to understand/prove some very fundamental mathematical properties about flows and cuts in this question. In fact, this exercise shows how to use algorithmic techniques like LPs and Duality to show deep mathematical properties.

$$\begin{aligned}
\max & \sum_{i=1}^{k} \lambda_i \\
\text{s.t} & \sum_{p \in P_{(s_i,t_i)}} f_p \geq \lambda_i && \forall 1 \leq i \leq k \\
& \sum_{i} \sum_{p \in P_{(s_i,t_i)} \text{ s.t } e \in p} f_p \leq 1 && \forall e \in E \\
& f_p \geq 0 && \forall i, \forall p \in P_{(s_i,t_i)}
\end{aligned}$$

(a) (5 points) Let $F_{\max}$ then denote the maximum flow, i.e., optimum value of the above LP. Likewise, analogous to $C_{\min}$ defined above, we can define a similar cut value: consider partitioning the graph vertices into disjoint sets S_1, S_2, $\ldots, S_p$ for any $2 \leq p \leq n$. Call such a partition a *separating partition* if for each $1 \leq i \leq k$, s_i and t_i are in different sets (i.e., all sources and sinks are separated). As an extreme example, the partition where each vertex is in its own set $\{v\}$ is one such separating partition with $t = n$. Now, let $C_{\min}$ denote the *fewest number of edges cut* (which go between different sets) among all separating partitions. Show that $F_{\max} \leq C_{\min}$.

Solution:

Proof. Consider the edge constraints, as the minimum number of edges going between two disjoint sets is equal to C_{min}, hence the flow between these two sets has to be less than equal to C_{min}.
We can solve this by considering duality, primal is maximization so by weak duality, any feasible solution to primal will have obj value less than equal to dual feasible solution's obj value.
In part(c) we show that the dual's feasible solution is the problem of finding the

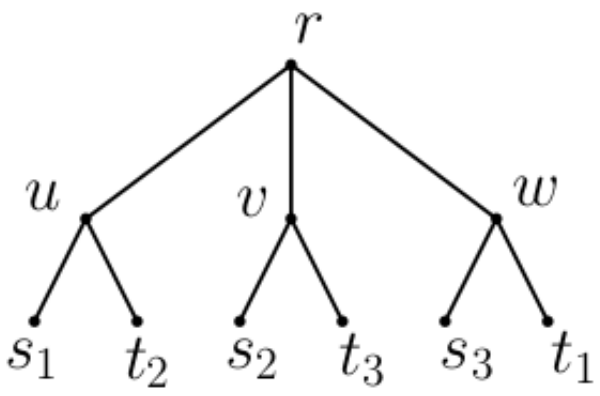

Figure 1: Figure for Gap Example

minimum cut. Hence combining both of them we have that the $F_{\max} \leq C_{\min}$. What remains to be shown is part c which we do below. □

(b) (10 points) Consider the following instance in Figure 1, and show that $F_{\max}$ can be strictly smaller than $C_{\min}$. This shows that having more than one source or sink makes the max flow-min cut theorem false!

Solution:

Proof. The problem is as follows:

$$\begin{aligned}
&\max \sum_{i=1}^{3} \lambda_i \\
&\text{s.t} \sum_{p \in P_{(s_i,t_i)}} f_p \geq \lambda_i && \forall 1 \leq i \leq 3 \\
&\text{s.t} f_{s_1-u-r-w-t_1} \geq \lambda_1 \\
&\text{s.t} f_{s_2-v-r-u-t_2} \geq \lambda_2 \\
&\text{s.t} f_{s_3-w-r-v-t_3} \geq \lambda_3 \\
&\quad f_{s_1-u-r-w-t_1} + f_{s_2-v-r-u-t_2} \leq 1 && e = (u,r) \\
&\quad f_{s_2-v-r-u-t_2} + f_{s_3-w-r-v-t_3} \leq 1 && e = (v,r) \\
&\quad f_{s_1-u-r-w-t_1} + f_{s_3-w-r-v-t_3} \leq 1 && e = (w,r) \\
&\quad f_p \geq 0 && \forall i, \forall p \in P_{(s_i,t_i)}
\end{aligned}$$

We can clearly see here that $C_{min} = 3$, as if you partition the vertices such that s_1, s_2, s_3 are separated from t_1, t_2, t_3 you need atleast three edges. And solving the three equations we can see that all value of λ are equal to 0.5, and the value of $F_{max} = 3$. Hence in this case $F_{max} < C_{min}$. □

(c) (10 points) Write the dual of the LP above, and say why it is the *LP relaxation* of the problem of finding the minimum cut defined above in part (a).

Solution:

Proof. Optimizing function in primal is max:$\sum_{i=1}^{k} \lambda_i$. This can be written as max:$\sum_{i=1}^{k} \sum_{p \in P_{(s_i,t_i)}} f_p$, which can be witten as $\sum_p c_p f_p$, where $c_p = \sum_{i=1}^{k}(\{p\} \cap P_{(s_i,t_i)})$. We create $|E|$ dual variables, i.e one dual variable for each edge, now objective function is: minimize($\sum_{e=1}^{|E|} y_e$). For each simple path in the graph that lies between a source and a destination we can write a dual constraint as $\sum_{e \in p} y_e \geq c_p$. And $\forall e$, $y_e \geq 0$.

$$
\begin{aligned}
minimize(\sum_{e=1}^{|E|} y_e) & \\
\sum_{e \in p} y_e \geq c_p & \qquad \text{p is a valid path connecting two sources} \\
\forall e, y_e \geq 0 & \\
c_p = \sum_{i=1}^{k} (\{p\} \cap P_{(s_i,t_i)}) &
\end{aligned}
$$

We show equivalence between minimum cut and our dual optimization function now:
$c_p = 1$ as there can be only one path connecting two sources, hence we must just ensure that for each valid path the sum of edges on that path is greater than equal to 1.
Let C_{min} be the minimum cut between two separating partitions, S_i and S_j. Now because of the cut edges we have C_{min} different paths, and as these paths correspond to some valid path between a source and a destination, hence the minimum value of the dual must be atleast C_{min}.

□

5. (25 points) **(Dominating Set for Bounded Treewidth Graphs)**

(a) (5 points) Define the treewidth of a graph G?

Solution:

Proof. The tree width of a graph G is the minimum width among all possible tree decompositions of G. The width of a tree decomposition is the size of its largest set minus one. Equivalently the width is the size of the largest clique minus 1 in a given tree decomposition. □

(b) (20 points) Assuming that a binary tree decomposition of G is given, design an

algorithm to compute a minimum dominating set with running time $f(k) * n^{O(1)}$, where n is the size of the graph and k is the width of the given tree decomposition.

Solution:

Proof. To give an algorithm for the same we follow a dynamic programming approach, assume a tree decomposition is given to us, we root the decomposition at some root vertex r. We define the state of the dp as follows: $dp[r][X]$ is defined as the minimum dominating set with T_r as the graph to be considered and X be the vertex set that is the subset of the bag containing r(called B_r) where it not be needed to necessarily dominate the vertices in X.
To calculate the dp recursively we consider cases where there is some y that is a subset of B_r that is chosen in our dominating set in the end. The answer is brute forcing through all possible subsets of B_r, choosing each subset of y and finding the dp solution to be minimum among all the possible subsets. We can see that if even one vertex is chosen in B_r, then we have covered everything in the bag B_r due to the maximum clique property of the tree decomposition hence we can remove the vertices that were already covered from our subsequent problems, now to calculate we must find the answer in this case as:

$$dp[r][X] = min_{\text{among all } y \subseteq B_r} \sum_{\text{for each child i}} (dp[r_i][B_{r_i} \cap y \cap N(y) \cap X])$$

To find the final answer we must calculate $dp[root][\phi]$, because every vertex must be dominated at the start. The complexity of the algorithm is calculated as follows: $O(n) * O(2^k)$ are all the possible dp states, to calculate one dp state you must do a computation of $O(2^k) * O(\text{degree of root})$ which can be roughly stated as $O(2^k) * O(n)$. Hence the complexity of the problem calculates to be $O(n^2) * O(4^k)$ which is equal to $O(n^{O(1)})f(k)$. The leaves can be easily initialized as the leaves are cliques in which the dominating set has size 1. The proof is very similar to the vertex cover proof done in the class. □

6. (25 points) **(Feedback Vertex Set)**

   (a) (15 points) Write the LP relaxation for the feedback vertex set of a graph G. You can assume that all weights are 1. Show that the LP relaxation can be solved in polynomial time.

Solution:

Proof. In Feedback vertex set we want the graph to be cycle free. We denote each vertex to be 0 or 1 indicating if it is chosen as the vertex to be removed. We want to minimize $\sum_{i=1}^{|V|} x_i$, where for each cycle possible in the graph i.e

$\forall C, \sum_{v \in C} x_v \geq 1$. For LP relaxation we assume that $0 \leq x_v \leq 1$.

$$\textbf{Minimize: } \sum_{i=1}^{|V|} x_i$$
$$\forall C, \sum_{v \in C} x_v \geq 1$$
$$0 \leq x_v \leq 1$$

As LP's can be solved in polynomial time the given LP also can be solved in polynomial time. □

(b) (10 points) Show that the shortest cycle in an undirected graph can be found efficiently (linear time?). How long is the shortest cycle in a graph in which minimum degree is at least 3.? How does this help in designing an approximation algorithm for feedback vertex set?

Solution:

Proof. The shortest cycle in an undirected graph can be found by running a BFS algorithm from any vertex, we do not stop at a particular vertex and only stop once all the vertices in the connected component are visited. Thus whenever there is a cycle there is an edge going back to the visited set, we can easily figure the cycle length from the difference in the BFS levels in which the tree is formed, i.e if there is a back edge observed from level 4 to level 1 in the BFS means there is a cycle length of 4 in this graph. Thus running the BFS and maintaining the minimum cycle length we can find the answer in linear time in the number of edges. (BFS runs in $O(|V| + |E|)$).
The longest shortest cycle in the graph with minimum degree 3 is $O(\log n)$, this result was mentioned in the class and we mention here the same without proving it. We use this result to create a $\log n$ approximation.

1. Remove the vertices of degree 1 simply removing them (they won't contribute to any cycle). And remove the vertices of degree 2 by the short circuiting algorithm discussed in class.
2. Graph has minimum degree 3.
3. Find the shortest cycle in the graph, this is of size atmost $O(\log n)$. Lets call the cycle C.
4. Remove C from the graph.

5. Repeat the above steps and remove cycles till graph is acyclic.

Let $C_1, C_2, C_3, \ldots, C_k$ be the cycles created from this process. They correspond to vertex disjoint cycles in the original graph as the vertices that were chosen in an earlier cycle were removed from the graph as soon as they were chosen. Output the FVS formed by all the vertices in these k cycles. The size of the FVS is less than equal to $O(\log n)k$ as the max cycle size is $O(\log n)$, and the minimum optimum FVS is of size greater than equal to k, as we have found k vertex disjoint cycles, hence these many vertices must be removed be removed to make the graph acyclic. Hence a $O(\log n)$ approximation. □