COL351 Assignment 4

Pratik Nimbalkar, Shreya Sonone

TOTAL POINTS

60 / 60

QUESTION 1

Hitting Set 17 pts

1.1 NP class **5 / 5**

✓ **+ 5 pts** *Correct*

1.2 NP completeness **12 / 12**

✓ **+ 12 pts** *Correct*

QUESTION 2

Tracking Set 28 pts

2.1 NP class **10 / 10**

✓ **+ 6 pts** *Correct Approach*

✓ **+ 3 pts** *Proof of Correctness*

✓ **+ 1 pts** *Showing $$poly(n)$$ time complexity*

2.2 NP completeness **18 / 18**

✓ **+ 18 pts** *Correct*

QUESTION 3

Flows and Cuts 15 pts

3.1 part (i) **10 / 10**

✓ **+ 3 pts** *Algorithm Correctness*

✓ **+ 5 pts** *Correctness Proof*

✓ **+ 2 pts** *Running Time Analysis Correctness*

3.2 part (ii) **5 / 5**

✓ **+ 2 pts** *Algorithm Correctness*

✓ **+ 2 pts** *Algorithm Correctness Proof*

✓ **+ 1 pts** *Time Analysis*

COL351 Assignment 4

Pratik Nimbalkar (2020CS10607)
Shreya Sonone (2020CS10384)

1 Hitting Set

Consider a set U = {u1, . . . , un} of n elements and a collection A1, A2, . . . , Am of subsets of U. That is, Ai ⊆ U, for i ∈ [1, m]. We say that a set S ⊆ U is a hitting-set for the collection A1, A2, . . . , Am if S ∩ Ai is non-empty for each i. The Hitting-Set Problem (HS) for the input (U, A1, . . . , Am) is to decide if there exists a hitting set S ⊆ U of size at most k.

Part 1

1. Hitting Set is in NP class:
Definition of NP class: The class of ALL decision problems which have Polynomial-time Verifier.
Hence we need to find a polynomial time algorithm which at a given instance and given answer to that instance of our problem, verifies whether the answer is correct or not. Here any given instance will be (U, A1, A2, ….Am).

The algorithm to verify our solution to the problem is:

1. We need to check whether the size of solution S is at most k. This, we can do in polynomial time O(n) since k<=n.
2. We also need to check whether at the instance I, Ai ∩ S ≠ Φ for all i.

   We will run the following algorithm m times on each Ai (A1, A2, … Am) in instance I:

We will find Ai ∩ S (for each i). This can be computed in O(n*k) i.e. O(n^2) time using is_intersection function. If is_intersection (Ai, S) comes out to be False, then it means that given answer S is an incorrect solution to our problem, else the solution would be correct. Since i<= m, at max, time complexity = O(m* n^2).

The intersection can be simply found by using this function:

```
Function is_intersection ( set P, set Q)
        for each element in Q:
                If there is element in P, then return True
        return false
```

This function would required O(|P| |Q|) time.

The above algorithm tells that a given solution is correct only when S satisfies the conditions of the instance that is S is a hitting set of (U, A1, A2..Am).

Also the time complexity of above algorithm is O(m*P*Q)=O(m*n*k)=O(mnk) = O(m*n^2) that is polynomial in that instance I.
Hence we can say that the Hitting Set belongs to the NP class.

1.1 NP class **5 / 5**

✓ **+ 5 pts** *Correct*

Part 2

2. Hitting Set is NP-complete by reducing Vertex-cover to Hitting Set:

We can reduce a problem which we already know that it is NP complete to Hitting Set Problem to prove that Hitting Set is NP complete as well. This known problem which is already NP complete can be taken as Vertex Cover problem here. Hence we reduce the Vertex Cover(VC) problem to hitting set(HS) problem.

We define the Vertex Cover Problem as follows:
A vertex cover of an undirected graph is a subset of its vertices such that for every edge (u, v) of the graph, either ‘u’ or ‘v’ is in the vertex cover.

We define instance I of vertex cover as follows. We say that for vertex cover problem, given graph G=(V,E), we have to find subset $T \subseteq V$ ($|T| \leq k$) such that $\forall$ edges $e(x,y) \in E$ either x lies in T or y lies in T.

We can convert I instance of vertex cover problem to I’ instance of Hitting Set problem as U=V, Ai = {x, y} $\forall$ e. Here e is the same as before (e = (x,y) E).
Since 1<= i<= m, we can say that there would be m sets corresponding to each edge in graph G (since each Ai has 1 edge). And according to the definition of Hitting Set problem, we have to find solution $S \subseteq U$ ($|S| \leq k$) such that $S \cap Ai$ is non empty $\forall$ $i \in [1,m]$.

This construction of Instance I’ from I will take polynomial time only. Constructing U will take O(|V|) i.e. O(n) time. For constructing m Ai’s, O(m) time would be required as each would get constructed in

constant time only. Hence total time = O(m) + O(n) = O(m+n) which is polynomial in terms of input sizes.

Now to prove that vertex cover can be reduced in the Hitting Set, we have to show that if instance I of Vertex Cover is “YES” then instance I’ of Hitting Set would also be “YES” and vice versa.

1. Vertex Cover => Hitting Set:
   We have to prove that if solution T of instance I of VC is there, then the same set will satisfy instance of HS I’.
   We can say that T (or S) ∩ Ai ≠ Φ since the set T has at least one end points of each edge (since it is the vertex cover) and the I’ has all the sets Ai which contains end points of some edge e. This is stated according to the definition. That is, the intersection of T or S with Ai is non empty. Using contradiction, if we assume this is true, i.e. Ai ∩ S is empty then that implies that there is an edge in instance I of vertex cover and none of it’s end points are in the set T. This leads to a contradiction. Hence we say that T is a solution of I’. (Size of T <= k as T is a vertex cover of G of at most size k). Hence when I is a YES instance of Vertex Cover problem, instance I’ of hitting set problem is mapped from “ I is also YES instance of HS”.

2. Hitting Set => Vertex Cover
   We have to prove that the solution S of instance I’ of HS is the solution of instance I of VC.
   Since S is hitting set, Ai ∩ S ≠ Φ for all i∈[1,m] and there exist an Ai that will contain end points of edge e for all edges e ∈ E. So the intersection is non empty for all Ai means that for each edge e, S will contain at least one of its endpoints.
   Hence S = T for our conditions to be satisfied.
   Size of S <=k as it is hitting set of size at most k, hence we can say that S is a solution of I.

Hence when I' is a YES instance of HS problem, instance I of VC is mapped to " I' is also YES instance of VC".

Hence we reduced the vertex cover problem which is NP complete to the Hitting Set problem in polynomial time and proved that both have "YES" solution only together.
Hence the Hitting Set is NP complete.

1.2 NP completeness **12 / 12**

✓ **+ 12 pts** *Correct*

2. Tracking Shortest Paths

Part-1

NP class is the class of ALL decision problems having polynomial time verifier. Hence If we have a solution T of the given problem, then we prove that the verifier of this solution is polynomial time.
a) To check the size of set T and verify that it is less than k, O(n) time will be taken as T has atmost n elements.
b) We calculate all shortest paths from s to t (algorithm is given below), This algorithm takes O(n+m) time as described below. Let P be collection of these shortest paths $P_1, P_2 ... P_l$. l is the number of shortest paths which has maximum value O(m+n) as bfs traversal takes O(m+n) time. Finding $S_i = T \cap P_i$ will take $|T| * |P_i|$ time and hence atmost time of $O(n^2)$. We verify that all $S_i \neq S_j$ for any i, j hence our verification will be done in $O(n^2 * (m + n))$ i.e polynomial time at max.
Algorithm for finding all P_i:
We do a breadth first search for finding all shortest paths from source(s) to destination(t)
1. Maintain an array of paths, initialize it as empty.
2. Begin Breadth First Search traversal from source vertex s.
3. We maintain an array of distance from source node to all other nodes and initialize it with infinity and an array of parent node of all nodes while doing BFS traversal, with parent of source node as -1 and distance of source node as 0.
4. For every node, we store all parent nodes for which that node has shortest distance from s.
5. Using this parent array, we keep on appending one node into the path and call for its parent nodes till we encounter parent node as -1. Once parent node is -1, we get a shortest path and then append this path to the array of paths.
6. When all paths are appended to final array of paths, we return this array of paths.
Time Complexity: This algorithm takes time same as time taken for bfs traversal and that it O(m+n).
Algorithm:

Algorithm 1 All_paths(G,s,t):

```
list_of_paths[][] ← φ
path[] ← φ
parent[] ← φ
parent[s] ← −1
dist[] ← inf
dist[s] ← 0
queue ← q
q.push(s)
while (!q.empty()) do
    node ← q.front()
    q.pop()
    nbrs[] = adjacent_nodes(node)
    i ← 1
    while i <= nbrs.size() do
        u ← nbrs[i]
        if dist[u] == dist[node] + 1 then
            parent[u].append(node)
        else if dist[u] > dist[node] + 1 then
            dist[u] ← dist[node] + 1
            q.push(u)
            parent[u] ← φ
            parent[u].append(node)
        end if
    end while
end while
func1(list_of_paths[][], path[], parent[], dist[], s, node)
Return list_of_paths
```

Algorithm 2 func1(list_of_paths[][],path[],parent[],dist[],s,node):

```
if (node == −1) then
    list_of_paths.append(path)
    return
end if
list[] ← parent[node]
i ← 1
while i <= list.size() do
    p ← list[i]
    path.push(node)
    func1(list_of_paths[][], path[], parent[], dist[], s, p)
    path.pop_back()
```

2.1 NP class **10 / 10**

✓ **+ 6 pts** *Correct Approach*

✓ **+ 3 pts** *Proof of Correctness*

✓ **+ 1 pts** *Showing $$poly(n)$$ time complexity*

Part-2

To prove that Tracking shortest path problem (TSP) is NP-complete problem, we take a known NP- complete problem X and reduce an instance of this X problem to an instance of TSP problem in polynomial time. If "Yes" instance of TSP problem also gives "Yes" instance of problem X and vice versa, then TSP problem is NP-complete.
We take known problem X as vertex cover problem
A vertex cover of an undirected graph is a subset of its vertices such that for every edge (u, v) of the graph, either 'u' or 'v' is in the vertex cover.
Given an instance I=(G(V,E),k) of vertex cover problem, we convert this instance into an instance of I'=((G'(V',E')),k') of TSP problem
Now we reduce instance I to I' in polynomial time as follows:
Graph G has set of vertices $V = v_1, v_2....v_n$ and set of edges $E = e_1, e_2, ...e_m$.
For graph G', set of vertices V' includes all vertices in V, all edges in set E and 5 more vertices (p,q,r,s,t) .i.e. $V' = v_1, v_2...v_n, e_1, e_2...e_m, p, q, r, s, t$
s and t are source and destination vertices respectively, vertex p is connected to all e_i and s through an edge, vertex q is connected to s and r through an edge and vertex r is connected to p,q and t through an edge.
Other than these edges, v_i and e_j has an edge between them if in graph G the edge e_j has one of its end point at v_i
This reduction takes the following time:
Constructing set of vertices V' takes $|V| + |E| + 5$ time i.e. O(m+n) time
Constructing set of edges E' takes $|V| * |E|$ time i.e. O(n*m) time as we check for all v_i and e_j.
Hence the reduction is polynomial time in terms of n and m.
The below diagram shows instance I'

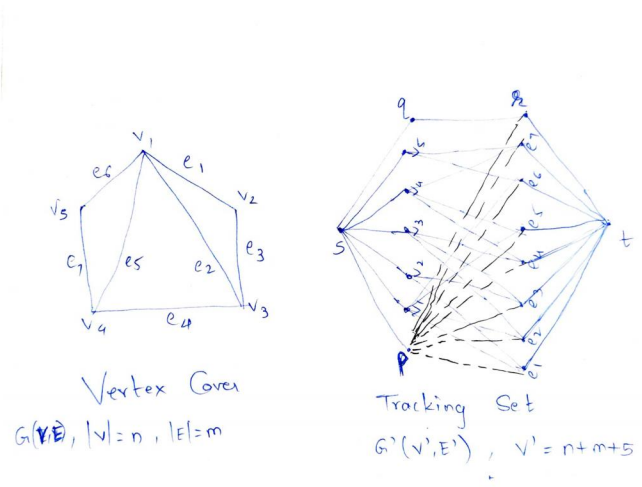


Figure 1: Reduction of instance I to I'

Claim 1: If I=(G(V,E),k) is a "YES" instance of VC then I'=(G'(V',E'),k') is also a "YES" instance of TSP, where k'=k+m+1

Proof: Vertex Cover => Tracking Set

Given solution W to vertex cover problem instance I, where $|W| = k$, We prove that set $T = W \cup E \cup \{p\}$, This set is a solution to tracking set path of instance I'. Paths from s to t having shortest length (i.e 4 in this case) include these type of paths P:

1. (s, v_i, e_j, t), Here, $T \cap P = \{e_j\}$ or $T \cap P = \{v_i, e_j\}$
2. (s, p, e_i, t), Here, $T \cap P = \{p, e_j\}$
3. (s, p, q, t), Here, $T \cap P = \{p\}$
4. (s, q, r, t), Here, $T \cap P = \phi$

Take any two shortest paths P_1 and P_2

The only case where problem arises is when: Both P_1 and P_2 is of type 1:

In this case, let $P_1 = (s, v_i, e_j, t)$ and $P_2 = (s, v_a, e_b, t)$

Now let us assume, $e_j = e_b$, so since P_1 and P_2 are different paths, $v_i \neq v_a$, therefore, v_i and v_a makes an edge e_j in the vertex cover problem graph G, hence atleast one of v_i and v_a is in W, and hence in T. Without loss of generality let $v_i \in T$ then $T \cap P_1 = \{v_i, e_j\}$ and $T \cap P_2 = \{e_j\}$

Hence we proved that Both are not same

In all other cases,it is trivial to show that $T \cap P_1 \neq T \cap P_2$ as atleast one vertex is different in both intersections and hence the paths are distinguishable

Claim 2: If I'=(G'(V',E'),k') is a "YES" instance of TSP then I=(G(V,E),k) is also a "YES" instance of VC, where k'=k+m+1

Proof: Tracking Set => Vertex Cover

Consider a solution T of tracking set of given instance I'. We prove that $W = T \backslash E \cup p$ is vertex cover set of instance I=(G(V,E),k). As mentioned in claim 1, paths are of type:

1. (s, v_i, e_j, t), Here, $T \cap P = p, e_j$ or $T \cap P = p, v_i, e_j$
2. (s, p, e_i, t), Here, $T \cap P = p, e_j$
3. (s, p, q, t), Here, $T \cap P = p$
4. (s, q, r, t), Here, $T \cap P = \phi$

We are concerned only with the case when paths P_1 and P_2 are of type 1, because only that path contains a vertex from set V.

Let $P_1 = (s, v_i, e_j, t)$ and $P_2 = (s, v_a, e_b, t)$, Now since T is tracking set of G', therefore $T \cup P_1 \neq T \cup P_2$, hence $T \cap P_1 = (v_i, e_j) or (e_j)$

$T \cap P_2 = (v_a, e_b) or (e_b)$

Problem arises when, $e_j = e_b$, In all other cases, we are able to distinguish both paths clearly, however in the problematic case, either $T \cap P_1 = (v_i, e_j)$ or $T \cap P_2 = (v_a, e_j)$, therefore atleast one of v_i and $v_a \in T$, Therefore for all $e_j = (v_i, v_a) \in E$, atleast one of v_i, v_a belongs to T, Hence T contains the vertex cover of instance I=(G(V,E),k). Vertex Cover set =W=$T \cap V = T \backslash (E \cup p)$

2.2 NP completeness **18 / 18**

✓ **+ 18 pts** *Correct*

3 Flows and Cuts

Let G = (V, E) be an undirected graph. For any $S \subseteq V$, let E(S) denote the set of edges in G that have both endpoints in S. We define the density of S as |E(S)|/|S|.

Part 1

1. We design a polynomial time algorithm that given a rational number α determines if there exists a set S with density at least α.

We define a subgraph of given graph G (V,E) as G'(S, E(S)).

Let D be the density of the desired subgraph.
We firstly make a construction to reduce our problem to several minimum capacity cut computations. These we will solve using computations involving min-cut max flow problem and other network flow methods. This would help us in determining whether there exists subgraph of density at least α as well as finding maximum density subgraph.

<u>Construction of the Network:</u>

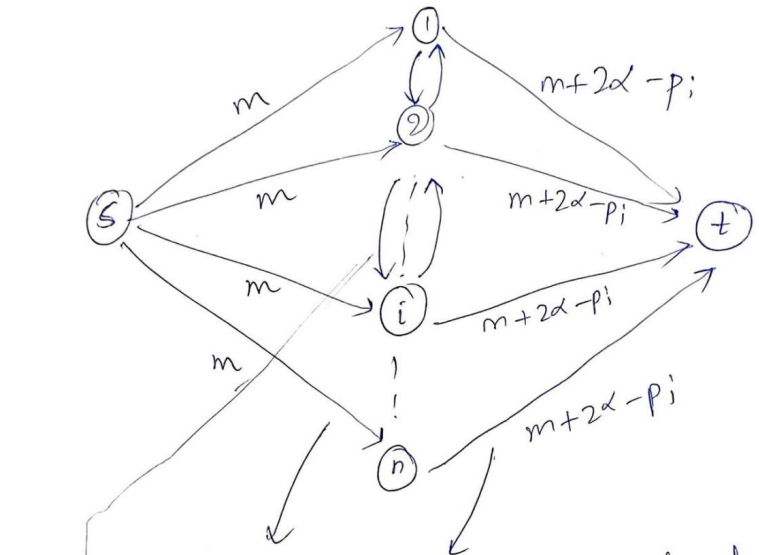
Our given graph is G(V, E). Let |V| = m and |E| = n.
Let the degree of vertex i be pi.
We are converting this graph to a new network flow graph. Let this be N (V1, E1).

i. Firstly since each network flow has a source where flow originates and a sink where flow ends, we add source s and sink t to V (vertices of G).

ii. Now, we need to convert our given undirected graph to a directed graph for simplicity. We do this by adding 2 directed edges in both directions connecting any 2 vertices. For eg if x — y is an undirected edge, add 2 directed edges x → y and x ← y. Hence now our new graph N is directed. The capacity of each edge can be kept as 1.

iii. Lastly the vertex of the graphs need to be connected. This can be done by connecting the source to all vertices of the graph by edges of capacity equal to the number of edges in G i.e. |E| =m. Also, the vertices need to be connected with sink t. This can be done by connecting vertex i with sink t using edge of capacity= (m + 2α - pi). Here pi is the degree of ith vertex. Similarly we have to connect all vertices (v1, v2,... vi, … vn).

Pictorially, the construction is as follows:



arrow direction denotes direction of flow.
↳ we can see that 2 directed edges are added in opposite directions b/w all vertices of original G graph.

Here c is capacity:
c (s→ i) = m, c(i → t) = m + 2α - pi, where i is any vertex in V
c(i → j) =1 where (i – j) is any edge in E. Here i and j are all vertices except s and t.

Since the degree of any vertex cannot be more than the total number of vertices, m - pi >0 and hence c (i → t) > 0. Thus all the capacities in our network N are positive.

Now, we define a cut which will partition our graph into 2 parts or sets: one containing source and other containing sink. Let the set which contains source be Vs and let it contain |Vs| nodes and the one which contains t be Vt and contain |Vt| nodes. Please note that these |Vt| and |Vs| number of vertices don't include the sink vertex and source vertex itself respectively.
We now focus on calculating the capacity of the cut.
If after partitioning, the part containing source contains none other than source vertex, then |Vs|= 0 and the capacity of cut will directly be m |V| since there are no other vertices in this part and only flow coming to the cut is from s which is equal to flow source to n vertices each having capacity of m.

Otherwise,
We need to consider all 3 different type of edges to calculate total capacity:

i. Total capacity of edges which carry flow from source to Vt vertices.
It would be: m*|Vt| (as capacity of each is m)

ii. Total capacity of edges which carry flow from Vs to the sink.
It would be:

$$\sum_{V_i}(m + 2\alpha - pi)$$

here i goes from 1 to |Vs| (basically cover all vertices in Vs)

$$m|Vs| + 2\alpha|Vs| - \sum_{V_i} p_i$$

=>

iii. Total capacity of edges in the graph through which the cut passes.
It would be: ∑ c(a→b) (here a ∈ Vs and b ∈ Vt)

Adding all 3,

=> m|Vt| + m|Vs| + 2α|Vs| - ∑ pi (here i ∈ Vs) + ∑ c(a→b)
=> m |V| + 2|Vs| (α - (∑ pi (here i ∈ Vs) - ∑ c(a→b)) /2 |Vs|)
(since |Vt| + |Vs| = |V| and took 2|Vs| common from rest)

=>

$$m|V| + 2|Vs| * (\alpha - \frac{(\sum_{V_i} p_i) - (\sum_{a,b} 1\ )}{2 * |Vs|})$$

Here a ∈ Vs, b ∈ Vt and i ∈ Vs

Now, we see that here we are subtracting the total capacity of edges connecting Vs and Vt from the total sum of degrees of vertices Vs. This entire term is divided by 2* |Vs|.
Now we know that subtracting the total capacity of edges connecting Vs and Vt from the total sum of degrees of vertices Vs will give us double of the number of edges of Vs.

Now, if double of |Vs| is divided by the obtained double of number of edges of Vs, then we will get density of Vs. (according to definition).

Hence,

$$\frac{(\sum_{V_i} p_i) - (\sum_{a,b} 1\ )}{2 * |Vs|} = D_s$$

Hence total capacity which we are calculating = m|V| + 2|Vs|(α - Ds)

Algorithm:

Now we relate our algorithm to the maxflow min cut problem.
We assume that we are getting a minimum capacity cut according to the min- cut max- flow problem.
Now our algorithm states that if |Vs| = 0 (i.e. when after partitioning, if the part containing source contains none other than source vertex), then the density of all the subgraphs would be less than or equal to α.
If this is not the case, i.e. |Vs| ≠ 0, then we can get subgraphs of density(D) greater than or equal to α.
i.e . D - α >= 0 if |Vs| ≠ 0
And D - α <= 0 if |Vs| = 0

Correctness:

As we already mentioned, if we have |Vs| = 0 i.e. only source in 1 part and rest of the network in other when partitioned into 2 by the cut, the capacity of the cut = m|V|. Now if there exists a smaller cut then its capacity would be less than m|V|.
Now if max flow min cut algorithm is runned on the network and a cut is obtained in which |Vs| ≠ 0, then the capacity of the cut would less than m|V|.
ie . m|V| >= capacity of this smaller cut
=> m|V| >= m|V| + 2|Vs|(α - Ds)
=> 2|Vs|(α - Ds) <= 0

=> Ds>= α

Now, if we don't obtain this cut where |Vs| ≠ 0 then |Vs| would be 0 and the source would be in 1 part and remaining network in the other part after partitioning of the cut. Hence the minimum capacity cut in this case = m|V|.

If we solve this by contradiction, we assume there exists a cut with Density > α. Let subgraph be (Vsg, Esg) Capacity = m|V| + 2|Vsg| (α - Density) <= (capacity of cut with source in 1 part and remaining network in other)
Now since Vs is empty (if it wasn't then the cut of source in 1 part and remaining network in other would not be a min cut),
we would not get any density having value > α.
In this case, the density of subgraph of G is <= α.
=> Ds <= α

Time Complexity:

1. Time required for construction:
   Only we are adding edges for all vertices (connecting them with source and sink) and other edges connecting to one another have been added directed edges of capacity 1.
   Hence total time = O(n) + O(m) = O(m+n)

2. Time required for our algorithm:
   The Edmonds Karp algorithm can be used to find the minimum capacity. The residual graph can be obtained by Edmonds Karp algorithm and the minimum capacity can be computed from there.
   Total time for Edmonds Karp = O(mn(m+n))..... Since the number of iterations in this algorithm are mn and for each iteration, O(m+n) time is required.
   Hence total time = O(mn(m+n)) + O(m+n)

= O(mn(m+n))

Hence we find a polynomial time solution for our given problem.

3.1 part (i) **10 / 10**

✓ **+ 3 pts** *Algorithm Correctness*

✓ **+ 5 pts** *Correctness Proof*

✓ **+ 2 pts** *Running Time Analysis Correctness*

Part 2

2. Present a polynomial time algorithm to find a set S of vertices with maximum density.

Lemma 1:
If a subgraph of a given graph G has density D and there is not any other subgraph which has density >= D + 1/ n(n-1), then that subgraph is the maximum density subgraph.

Proof:
The density is the (number of edges) / (number of vertices) for some subgraph of given graph i.e. |E(s)| / |S|

Now let the number of edges of that subgraph be ei and the number of vertices be vi.
We know that the range of ei is : 0<= ei<= m (since the number of edges in subgraph cannot exceed number of edges in parent graph and the subgraph at minimum can have only 1 vertex and 0 edges)

and range of vi is: 1<= vi<= n (same reason)

Density = ei/ vi.

Using simple maths, we can see that the range of Density would vary from 0 to m.

There are a variety of possible values of densities for different combinations of subgraphs. We have to find the one having maximum density. Now, we find the smallest difference between any 2 possible values of D.

Let this difference be S.D.

Let us find the difference of densities between 2 subgraphs having edges and vertices as: e1, v1 and e2, v2.

S.D = e1/v1 - e2/v2
S.D = (e1v2 - v2e1)/ v1v2

Now if v1 = v2,
S.D = (e1-e2)/ v1
Now, v1<= n
=> 1/v1 >= 1/n 1
Also |S.D| = |e1-e2|/ v1 and |e1-e2| >= 0
Now if e1 ≠ e2, then the minimum difference between them would be 1 (since the number of edges is a whole number and any 2 whole numbers if not the same and not 0, differ by minimum one).
Hence, |e1-e2|>=1
So, |S.D| >= 1/v1 2
Now from 1 and 2,
|S.D| >= 1/n

Now, if v1 ≠ v2,
We know that v1 <= n
At max, lets assume v1 is n.
Now at max, v2 can be (n-1) (since v1 ≠ v2 and v2 =n, at max)
Hence, at max, v1v2 = n(n-1)
Hence, v1v2 <= n(n-1)
=> 1/ v1v2 >= 1/ (n)(n-1)

Now, |S.D| = |(e1v2 - e2v1)|/ v1v2
Again since all e1, e2, v1, v2 are whole numbers and if not equal to 0, |(e1v2 - e2v1)| >= 1, the multiplication, if not equal may differ by at max 1.

Hence, |S.D| >= 1/v1v2 >= 1/ n(n-1)

Now, since we know that at maximum 2 densities can differ by 1/n or 1/n(n-1) depending on whether v1=v2 or not.

Now, 1/n(n-1) < 1/n. Hence if we have a subgraph of density D and at minimum, if we are not able to find subgraph having density more than or equal to D + 1/ n(n-1) (Since D + 1/n(n-1) < D + 1/n, this condition would automatically satisfy v1 =v2 case), then our subgraph will be the one having maximum density.

Algorithm:

We will find the maximum density subgraph using binary search. We will use the above mentioned lemma to decide where we have to stop our search if we find that if there is not any other subgraph which has density >= D + 1/ n(n-1), the subgraph of density D is our required answer.

Using binary search,

We set p to be 0 initially. ($p \leftarrow 0$)
and q to be the number of edges m ($q \leftarrow m$)
Also, initially, |Vs| is assigned 0.

Now, we will perform our algorithm for q >= p + 1/n(n-1).

while q >= p + 1/n(n-1)

set a to be (p+q)/2
construct the network N (V1, E1).
Then we find a min cut Cm.

if |Vs| = 0
we set q as α (q←α)
Else
We set p as α (p ← α)

and Vs as Set containing all nodes except source (S1 - {s})... Here S1 the partition containing source s obtained after partitioning the graph using the cut

And again run our algorithm on these new p and q.

We return the subgraph of G induced by Vs.

Correctness:

When the algorithm is running, Vs will have subgraphs whose density is more than p. Now, for stopping our binary search, we will set the condition that there is no subgraph having density greater than or equal to p + 1/ n(n-1)
And then this current p will be our maximum density.

We will use the above mentioned lemma to decide where we have to stop our search if we find that there is not any other subgraph which has density >= p + 1/ n(n-1), the subgraph of density p is our required answer.

We have provided the proof of this in Lemma 1 proof itself.

We have initially set Vs as empty where only source is present in the part containing source obtained after partitioning the network into 2 parts using the cut.

We will perform our algorithm for q >= p + 1/n(n-1) because For q < p + 1/n(n-1), then it means we still have not got a subgraph having density more than subgraph having density p.
Hence, if the new subgraph's density becomes more than p + 1/n(n-1), then subgraph having density > p do exists. Thus if there does not exist such a subgraph having density > p + 1/n(n-1), the subgraph with density p is the maximum density subgraph.

Now we have set α as (p+q)/2. This is the standard technique for binary search.

Next, constructing the network is done as mentioned at the starting of the solution and min cut is found in polynomial time using the Edmond Karp algorithm.

Now, if only source is present in the part containing source obtained after partitioning the network into 2 parts using the cut, then our q is set to α (this we had proved in part 1 that if |Vs| =0, D - α <=0, hence D <= α and maximum density in this case would be α.
We can directly return the subgraph which has this maximum density α as our answer.

Now, if this is not the case, then D-α>=0 (as mentioned in part 1 also).
Hence we set the minimum limit p as α and Vs as every node in the partition containing source s obtained after introducing the cut except the source itself. Now again our algorithm is runned on this pair of new p and q while initial condition of q>= p + 1/(n)(n-1) is satisfied.
Then we would return that particular subgraph after the algorithm terminates at the situation mentioned above.

Hence, all our mentioned lemmas and conditions are correct and are proved by us and thus our algorithm is correct.

Time Complexity:

We have only 1 loop in our algorithm.

We are doing binary search in the following array:

$$(0, \frac{1}{(n)(n-1)}, \frac{2}{(n)(n-1)}, \frac{3}{(n)(n-1)}, \ldots, 1, 1+\frac{1}{(n)(n-1)}, 1+\frac{2}{(n)(n-1)}, \ldots, 2, \ldots, j+\frac{i}{(n)(n-1)}, \ldots, m)$$

Here 0<= j<= m-1

Number of elements = m(n)(n-1)

This would be executed the following number of times on applying binary search:

log ((m)(n)(n-1))

=> O(log (m*(n^2)))

Now, the step which will take maximum time will be the finding of min-cut.
Number of vertices present in our new network = n+2 (adding sink and source)
=> O(n)
And number of edges = 2m + 2n (connecting each vertex with source + sink and connecting vertices with each other - directed, both directions).
=> O(m+n)

The Edmond Karp algorithm would required O (|edges||vertices|(|edges| + |vertices|)) = O((m+n)(n) ((m+n)+n))

Total time = O (log (m*(n^2) ((m+n)(n) (m+n+n)))
Which is polynomial in the input m and n.

3.2 part (ii) **5 / 5**

✓ **+ 2 pts** *Algorithm Correctness*

✓ **+ 2 pts** *Algorithm Correctness Proof*

✓ **+ 1 pts** *Time Analysis*