CS 6363 - ASSIGNMENT-4 - PART (b)

PROBLEM-5:

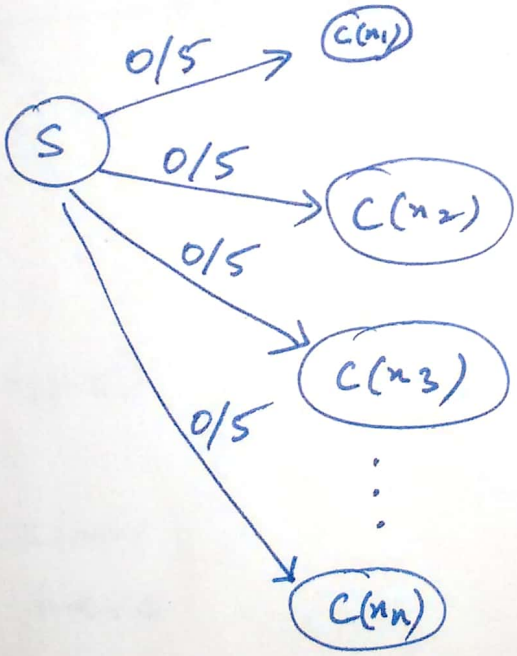
(a). S : set of students.

For $x \in S$, $C(x)$ = wishlist of student x.

C : set of classes.

We have to model this problem as a max-flow problem.

1) Let $S = \{x_1, \ldots, x_n\}$ be the set of students.
Then $C(x_1), C(x_2) \ldots C(x_n)$ refers to the wishlist of the individual students.
We create a node in the graph for S (set of students) and nodes for each of $C(x_1), C(x_2) \ldots C(x_n)$.



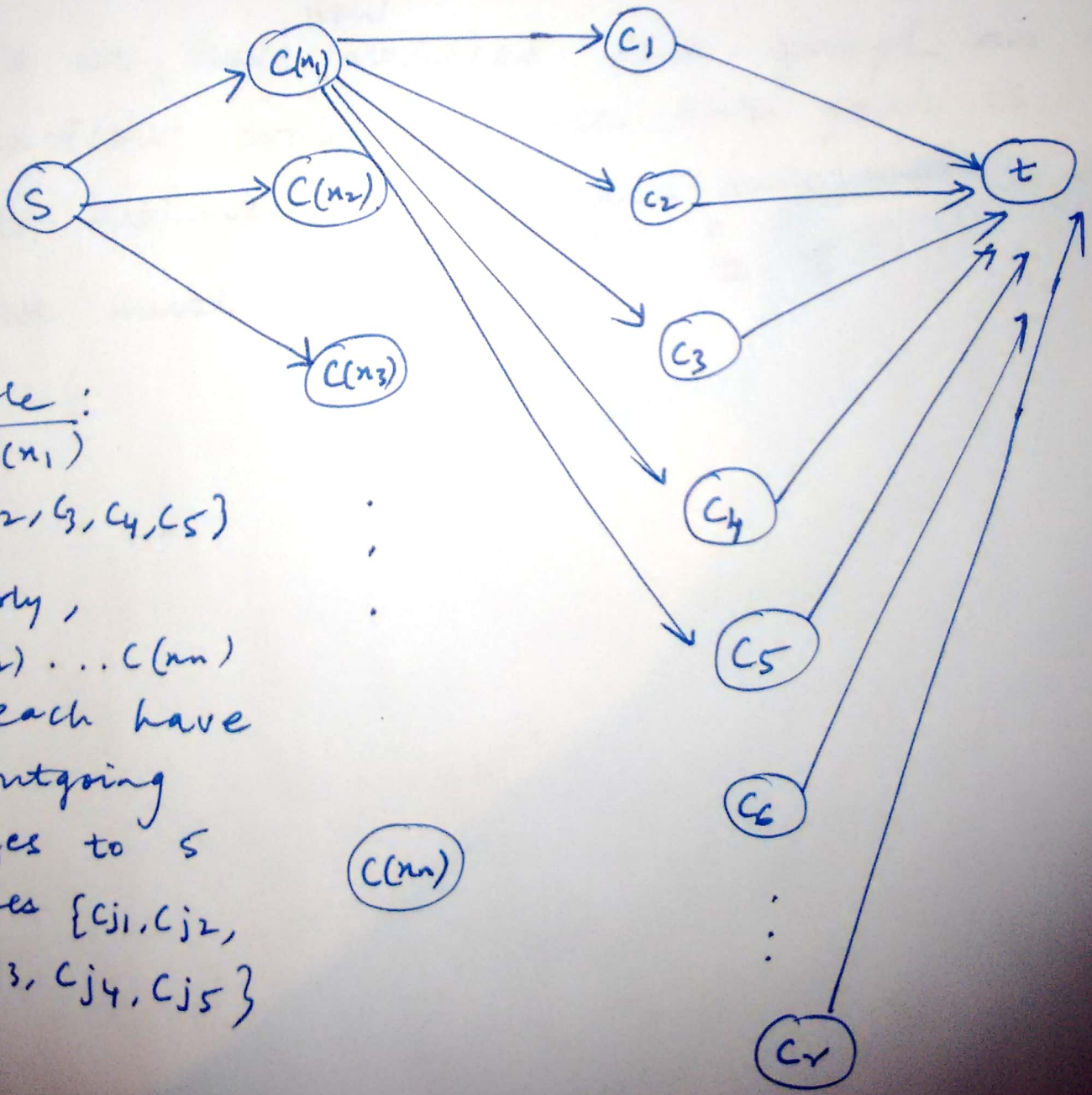
We add an edge (directed) from S to each $C(x_i)$ where $i = 1, 2, \ldots n$ with a capacity of 5, since a student can take atmost 5 classes (hence, wishlist can have atmost 5 classes)

) Let C be the set of classes.

$$C = \{c_1, \ldots, c_r\}$$

We make a node for each c_i, $i = 1, \ldots r$.

ode $C(x_i)$
ach, We add a directed edge from $C(x_i)$ to each class $c_k \in C(x_i)$ with capacity 1 (since each student can register for a given class atmost once.



Example:
Here $C(x_1) = \{c_1, c_2, c_3, c_4, c_5\}$

Similarly, $C(x_2) \ldots C(x_n)$ will each have 5 outgoing edges to 5 nodes $\{c_{j1}, c_{j2}, c_{j3}, c_{j4}, c_{j5}\}$

3) Finally, we add node t, which denotes the maximum total enrolment in all classes. We add directed edges from each C_i, $i = 1, \ldots, r$ to t with a capacity of 20. (Since each class can have atmost 20 students and hence, can contribute atmost 20 to the total enrolment).

4) Since we have now modeled the graph as a max-flow problem. Since our goal is to maximize total class enrolment, we must maximize flow from s to t.

Algorithm UniqueCut(G,s,t,f*)

{

Compute G_R = residual graph of $G = (V,E)$ for max flow f*

Compute X = set of vertices reachable from s in G_R

Reverse graph Gr (call it G_R')

Compute Y = set of vertices reachable from t in G_R'

if (X ∪ Y = V)

return YES

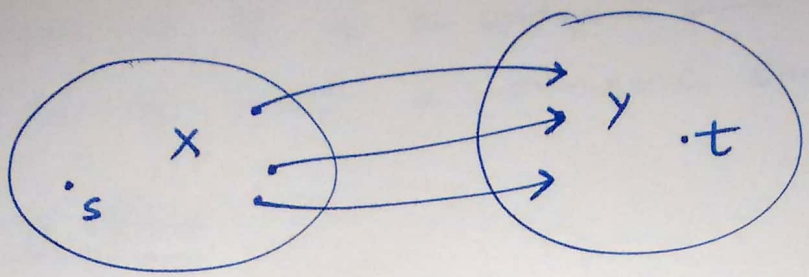
else

return NO

}

Explanation:

1) Suppose there is a min-cut in G. Let us call it $\{X, Y\}$.



2) Let f^* be the max flow. Since we know that the max-flow saturates every edge in min-cut, every edge from X to Y has flow = capacity.

3) In the residual graph G_r of G, we will not have any edges from X to Y. However, there will be edges from Y to X.

4) We first compute the residual graph and then compute:
X = set of vertices reachable from s in G_r

5) Note that $\{X, V \setminus X\}$ defines a min-cut.

6) We then compute G_r', the reversed graph of G_r. There would be no edges from Y to X in G_r' (follows from step 3)

7) We compute Y, the set of vertices reachable from t. The partition $\{Y, V \setminus Y\}$ also defines a mincut

6) Since we now have 2 min cuts, we must check if they are the same.
Hence, we check if $X \cup Y = V$
if yes ⇒ it is a unique cut
if no ⇒ not a unique cut.

Running time:

1) Computing Residual graph takes $O(|E|)$ time.
2) Finding set X takes $O(|V|+|E|)$ time.
3) Reversing G_r takes $O(|V|+|E|)$ time.
4) Find set Y takes $O(|V|+|E|)$ time.
5) The remaining operations take $O(1)$ time.

$\therefore T(n) = O(|V|+|E|)$

Algorithm DoubleTargetFlow(G, s, t_1, t_2)

{

Add vertex t' to G (lets call this graph G')

$c_0(t_1 \to t') = \infty$

$c(t_2 \to t') = \infty$

result = BlackBox(G, s, t')

Return result

}

Here, BlackBox(G, s, t') computes the maximum flow from s to t' in G.

i.e. it solves the max-flow problem in a standard flow graph.

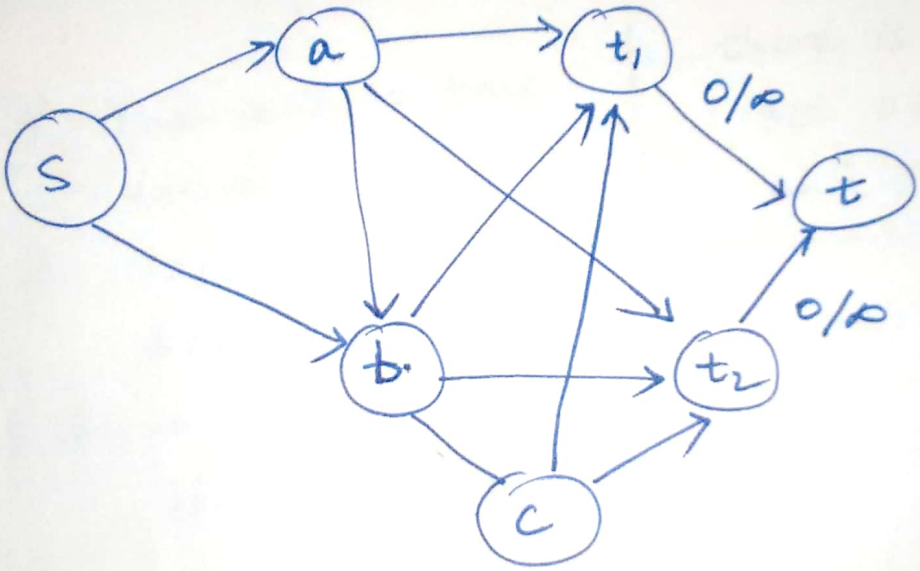
c). Explanation:

1) In the double target flow network, we must maximize the flow from s to both t_1 and t_2.

2) We can add a new node t' in our problem graph. We also add directed edges from t_1 and t_2 to t'.

3) We set the capacities of these edges ($t_1 \to t'$ and $t_2 \to t'$) equal to ∞.

Example:



4) Clearly maximising the flow from s to t_1 and t_2 is the same as maximising the flow from s to t'.

5) Maximising flow from s to t' is the same as computing the maxflow in a standard black flow problem, which is done by the black-box.

PROBLEM-6:

(a). Algorithm MaxIndSet(G)

```
{
    n = |V|
    return Helper(G, 0, n)
}

Algorithm Helper(G, i, j)
{
    if Ind-Set(G, j) = true
        return j
    if i = j
        return i
    if i = j - 1
        if Ind-Set(G, j) = true
            return j
        else
            return i
    K = (i+j)/2
    if Ind-Set(G, K) = true
        return Helper(G, K, j)
    else
        return Helper(G, i, K-1)
}
```

Input: Graph G

Output: No. of vertices in the maximum independent of G.

Assumption:

Blackbox Ind-Set(G, K) which returns true if there is an independent set of size $\geq K$ and false otherwise.

Explanation:

1) We assume that we have a blackbox which computes the answer to the Indepen Set (G, K) decision problem at any step.

2) We use a binary search approach.

3) We call Max Ind set (G, 0, |V|) in the beginning.

4) At each call of Helper (G, i, j), we first check if Ind-set(j) = true. If yes, our work is done.

5) Otherwise, we call Ind-set ((i+j)/2)
if this evaluates to true, we call
Helper (G, (i+j)/2, j)
If it evaluates to false, we call
Helper (G, i, (i+j)/2 - 1)
This follows from the definition of the Ind-Set problem.

6). When we reach our base cases,
1) i = j
2) i = j - 1
we return the appropriate value.

Running time:

1) The base cases take $O(1)$ time.

2) Maximum no. of recursive calls

$$= \log(|V|)$$

$$\therefore T(n) = O(\log |V|)$$

1) We have to prove that 4-SAT is np-hard.

2) To prove this, we reduce 3-SAT, a known np-hard problem, to 4-SAT.

3) We can change each clause in a 3CNF formula:

$$a \vee b \vee c \longrightarrow (a \vee b \vee c \vee x) \wedge (a \vee b \vee c \vee \bar{x})$$

where $\bar{x}$ is a new variable (arbitrarily set).

4) This reduction can be done in polynomial time.

∴ 3SAT has been reduced to a 4SAT problem plus a polynomial time procedure

5) The reduction implies that if we had a polynomial time procedure for 4SAT, then we'd have a polynomial time procedure for 3SAT, an np-hard problem which would imply ⇒ P=NP

∴ 4SAT is an np-hard problem.

To Prove: Set-cover is np-hard.

1) We use a reduction from vertex cover to set cover.

2) The vertex cover problem asks if there is a subset $I \subseteq V$ of at most K vertices which are adjacent (cover) all edges in the graph.

3). The set cover problem asks if there is are K subsets $S_1, \ldots, S_K \in C$ such that $\bigcup_{i=1}^{K} S_i = S$, where C is a collection of subsets of S.

4). We model (reduce) the ~~set~~ vertex cover problem as a set cover problem.

vertex-cover: E $e \in E$ (i is endpoint) K mapped to.

E → S, $e \in E$ → S_i, K → K

5) One set S = set of edges E in G.

6) We label all nodes in G from 1 to n. Each subset S_i is the set of edges with node i as an endpoint.

7) • All the nodes in the graph represent subsets $S_1, \ldots, S_n$. Thus, the set of vertices V is mapped to C.

$i \in L$

8) The value k is mapped to k.

i.e. Finding a vertex cover with atmost K vertices

↔ Finding a vertex set which covers all edges with size atmost k.

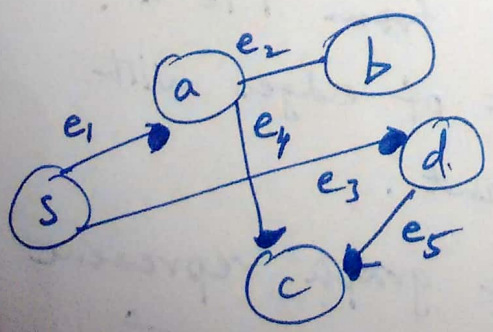
↔ Finding • k subsets which cover the set of all edges (set S).

This reduction can be done in O(|V|+|E|) time.

Thus, this reduction means that if there is a polynomial time procedure for solving • set cover, then there would be a polynomial time procedure for solving vertex cover which would imply P = NP.

∴ Set-cover is np-hard.

Example:



Mapping →

s-1, a-2, b-3, c-4, d-5.

$S = \{e_1, e_2, e_3, e_4, e_5\}$

$S_1 = \{e_1, e_3\}$ $S_2 = \{e_1, e_2, e_4\}$

$S_3 = \{e_2, e_4\}$ $S_4 = \{e_3, e_5\}$

$S_5 = \{e_3, e_5\}$ $\bigcup_{i=1}^{5} S_i = S$

K → k

To Prove: Knapsack problem is np-hard.

1) The knapsack problem asks given non-negative $V[1....n]$, $S[1...n]$, B, k, is there a subset of indices I such that $\sum_{i \in I} V[i] \geq k$ and $\sum_{i \in I} S[i] \leq B$.

2) The subset problem asks if given non-negative integers $a_1, \ldots a_n$, is there a subset of these numbers with a total sum t?

3) We reduce the subset problem to an instance of the knapsack problem.

Suppose $V = \{v_1, \ldots, v_n\}$ and $S = \{s_1, \ldots s_n\}$

4) We map:

$$a_i = v_i \;, \; i = 1, \ldots, n$$

$$a_i = s_i \;, \; i = 1, \ldots n$$

$$t = B = k$$

5) Now, if there is a subset I of indices such that:

$$\sum_{i \in I} a_i \leq B \Leftrightarrow \sum_{i \in I} s_i \leq t \quad \text{—— (1)}$$

6) Also, $\sum_{i \in I} a_i \geq k \Leftrightarrow \sum_{i \in I} v_i \geq t$ —— (2).

$$\therefore \sum_{i \in I} a_i \leq B \Leftrightarrow \sum_{i \in I} s_i \leq t$$

$$\sum_{i \in I} a_i \geq t \Leftrightarrow \sum_{i \in I} v_i \geq k$$

$$\Leftrightarrow \sum_{i \in I} s_i = t$$

∴ There is a subset of indices I with total sum ≥ K

⇔ There is a subset of indices such that

$$\sum_{i \in I} v_i \geq K \quad \text{and} \quad \sum_{i \in I} s_i \leq B$$

7) Hence, we have reduced the np-hard sub-set problem to the knapsack problem.
Clearly, this reduction can be done in polynomial time ($\because O(n)$)

8) If there was a polynomial time algorithm to solve knapsack problem, then it implies there would be a polynomial time algorithm to solve the subset problem, which would imply P=NP.

∴ The knapsack problem is np-hard.