UCSD CSE 101 Section A00, Winter 2016 FINAL

March 17, 2016

KEY

NAME:

Student ID:

Question	Points	Score
1	18	
2	12	
3	10	
4	10	
5	10	
TOTAL	60	

INSTRUCTIONS. Be clear and concise. Write your answers in the space provided. **Do not write your answers on the back of pages, since we will scan into Gradescope.** Use the scratch page at the end and/or ask for extra sheets, for your scratchwork. Make sure to check your work. Good luck!

You may freely use or cite the following subroutines from class:

- dfs(G). This returns three arrays of size $|V|$: *pre*, *post*, and *cc*. If G is undirected and has k connected components, then the *cc* array assigns each node a number in the range 1 to k. dfs(G) has runtime complexity $\mathcal{O}(|V| + |E|)$.
- bfs(G, s), dijkstra(G, l, s), bellman-ford(G, l, s). Each of these returns two arrays of size $|V|$: *dist* and *prev*. bfs(G, s) has runtime complexity $\mathcal{O}(|V| + |E|)$.
  dijkstra(G, l, s) has runtime complexity $\mathcal{O}((|V| + |E|)log(|V|))$ assuming a binary heap PQ implementation.
  bellman-ford(G, l, s) has runtime complexity $\mathcal{O}(|V||E|)$.

QUESTION 1. Short Answer [Note: In the T/F questions, you must justify your answer to receive points.]

(a) *(2 points)* True or False: $n^{2.1} \in O(n^2 log(n))$ [Explain why]

Answer:

False.

Explanation:

$$\lim_{n\to\infty} \frac{n^{2.1}}{n^2 \log(n)} = \lim_{n\to\infty} \frac{n^{0.1}}{\log(n)}$$

Using L'hopital rule

$$\lim_{n\to\infty} \frac{(0.1\, n^{-0.9})}{(1/n)} = \lim_{n\to\infty} 0.1\, n^{0.1} \to \infty.$$

(b) *(2 points)* How many times will the following program print "**Hello World**"? Give the recursive expression and solve it using the Master Theorem. You may assume that n is a power of 2.

```
define Foo(A[i_1, i_2, ..., i_n]):
  if n == 0:
    return
  if n == 1:
    print "Hello World"
    return
  print "Hello World"
  Foo(A[i_1, ..., i_⌊n/2⌋])
  Foo(A[i_⌊n/2⌋+1, ..., i_n])
  return
```

Recursive Expression:

$$T(n) = 2T(n/2) + O(1)$$

Closed-form Expression:

$T(n) = O(n)$. ; $T(n) = an + b$.

$T(1) = 1$, $T(0) = 0$

$T(2) = 2$

putting the values $T(n) = n$.

(c) *(4 points)* Given a graph in which all edges have equal weights. Describe in English an efficient algorithm to compute an MST in the graph. Explain your algorithm's runtime complexity. (Your algorithm should be more efficient than algorithms such as Prim's algorithm, Kruskal's algorithm, etc. which find an MST in a general edge-weighted graph.)

Give an English description of your algorithm:

All spanning trees will have same weight. To compute MST we will run BFS to find tree which will have weight = ~~...~~ c|V-1|. where c is weight of edge & |V| is no. of vertex in graph.

Analyze the runtime complexity of your algorithm:

Runtime complexity = O(|V|+|E|)

(d) *(3 points)* Draw lines to match the three dynamic programming algorithms on the left to the appropriate recurrences in the right column.

i. Bellman-Ford — a. $T_j^{(k+1)} = \min\{T_j^k, \min_i (T_i^{(k)} + d_{ij})\}$

ii. Longest Common Subsequence — c.

iii. Knapsack — d.

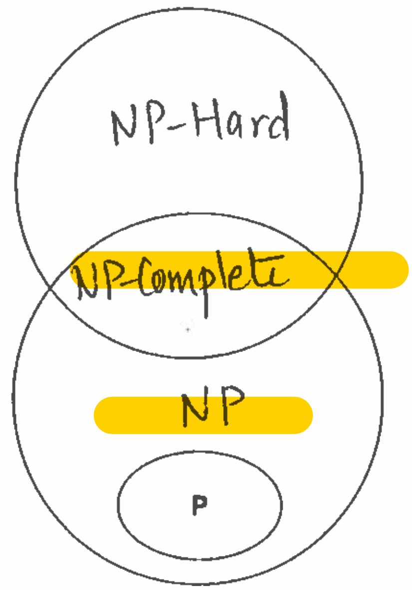
a. $T_j^{(k+1)} = \min\{T_j^k, \min_i (T_i^{(k)} + d_{ij})\}$

b. $T_{i,j} = \min\{1 + T_{i-1,j}, 1 + T_{i,j-1}, d_{i,j} + T_{i-1,j-1}\}$

c. $T(i,j) = \begin{cases} T(i-1,j-1)+1 & x(i) = y(i) \\ \max(T(i,j-1), T(i-1,j)) & \textit{otherwise} \end{cases}$

d. $T_k(y) = \max\{T_{k-1}(y), T_k(y - w_k) + v_k\}$

(e) *(3 points)* With the assumption that P $\neq$ NP, please label the following figure to indicate the relationships among the classes P, NP, NP-Complete and NP-Hard. P has been drawn already.



(f) *(4 points)* True or False: If a problem $L1$ is polynomial-time reducible to a problem $L2$, and $L2$ has a polynomial-time algorithm, then $L1$ has a polynomial-time algorithm. [If True, explain why. If False, show a counterexample.]

Answer:

True

Explanation/Counterexample:

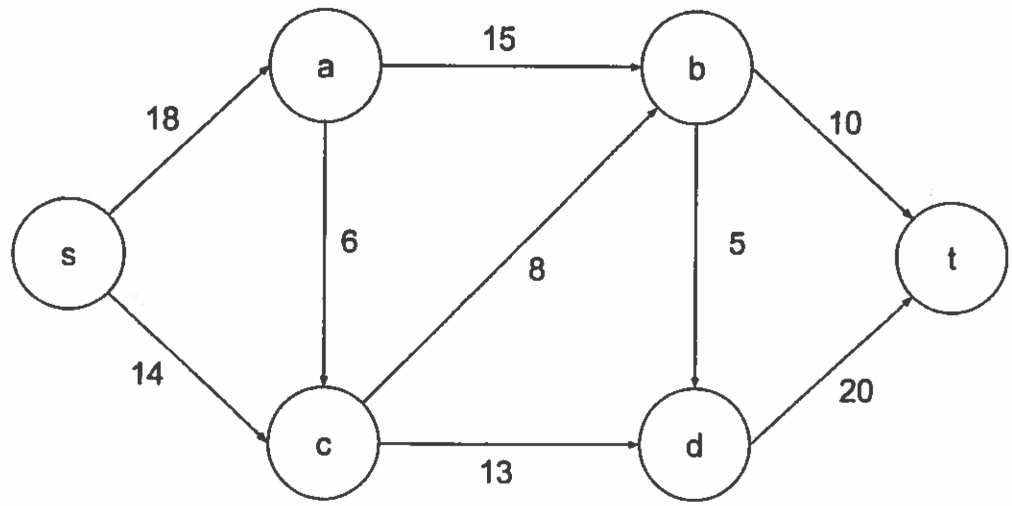
L_1 is polynomial-time reducible to problem L_2.
this means L_1 is as simple as L_2.
L_2 is polynomial time $\Rightarrow$ L_1 is polynomial time.

OR

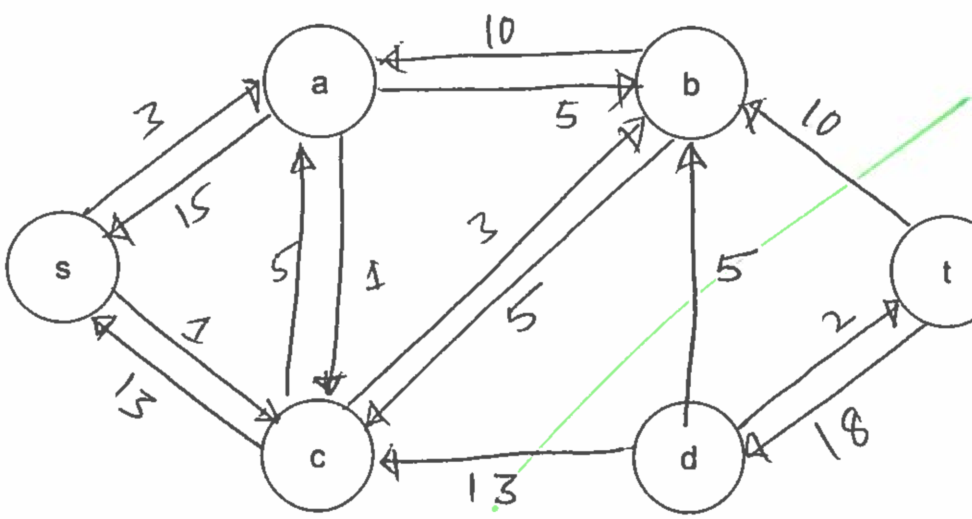
L_1 can be done in polynomial time + polynomial steps to polynomial algorithm $\Rightarrow$ L_1 is polynomial time algorithm

QUESTION 2. Network Flow/Linear Programming

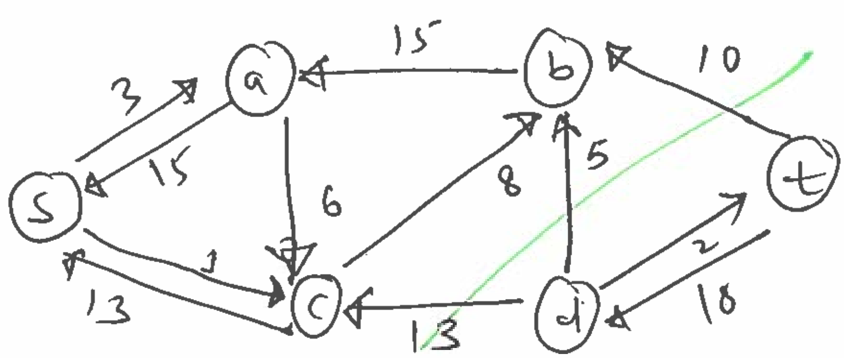
For this question, refer to the flow network below.



(a) *(3 points)* Run the Ford-Fulkerson Algorithm on the given network. Draw the final residual graph G^f into the figure provided below. Extra copies of the network are given in the last sheet of the exam.



Sol ②:



(b) *(2 points)* What is the value of the maximum flow?

28

(c) *(3 points)* List the edges that are in the minimum cut that corresponds to the maximum flow.

b-t

b-d

c-d

(d) *(4 points)* Write the problem of determining the maximum *s*-*t* flow in the given network as a linear program. (Don't leave out any constraints!)

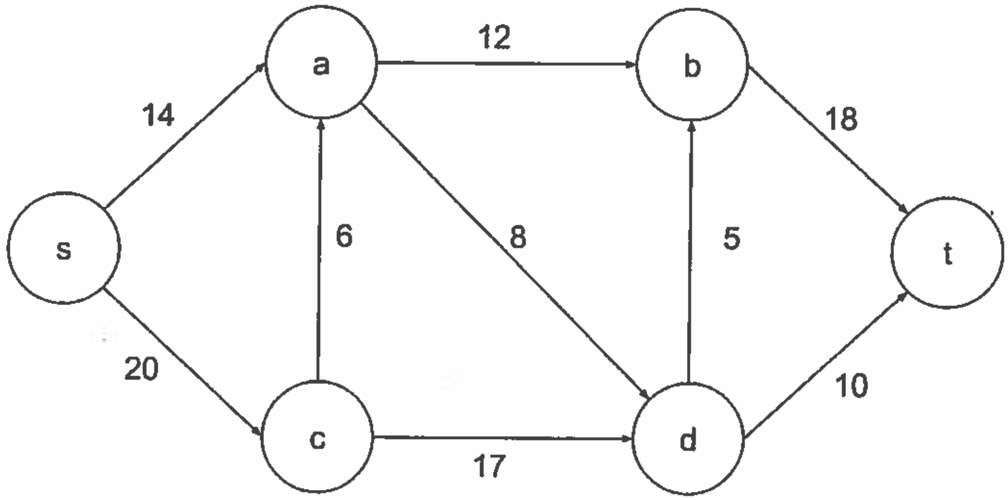
Maximise $f_{sa} + f_{sc}$ subject to

$$\left.\begin{array}{l} 0 \leqslant f_{sa} \leqslant 18 \\ 0 \leqslant f_{sc} \leqslant 14 \\ 0 \leqslant f_{ac} \leqslant 6 \\ 0 \leqslant f_{ab} \leqslant 15 \\ 0 \leqslant f_{cb} \leqslant 8 \\ 0 \leqslant f_{cb} \leqslant 13 \\ 0 \leqslant f_{bd} \leqslant 5 \\ 0 \leqslant f_{bt} \leqslant 10 \\ 0 \leqslant f_{dt} \leqslant 20 \end{array}\right\} \text{Edge constraints}$$

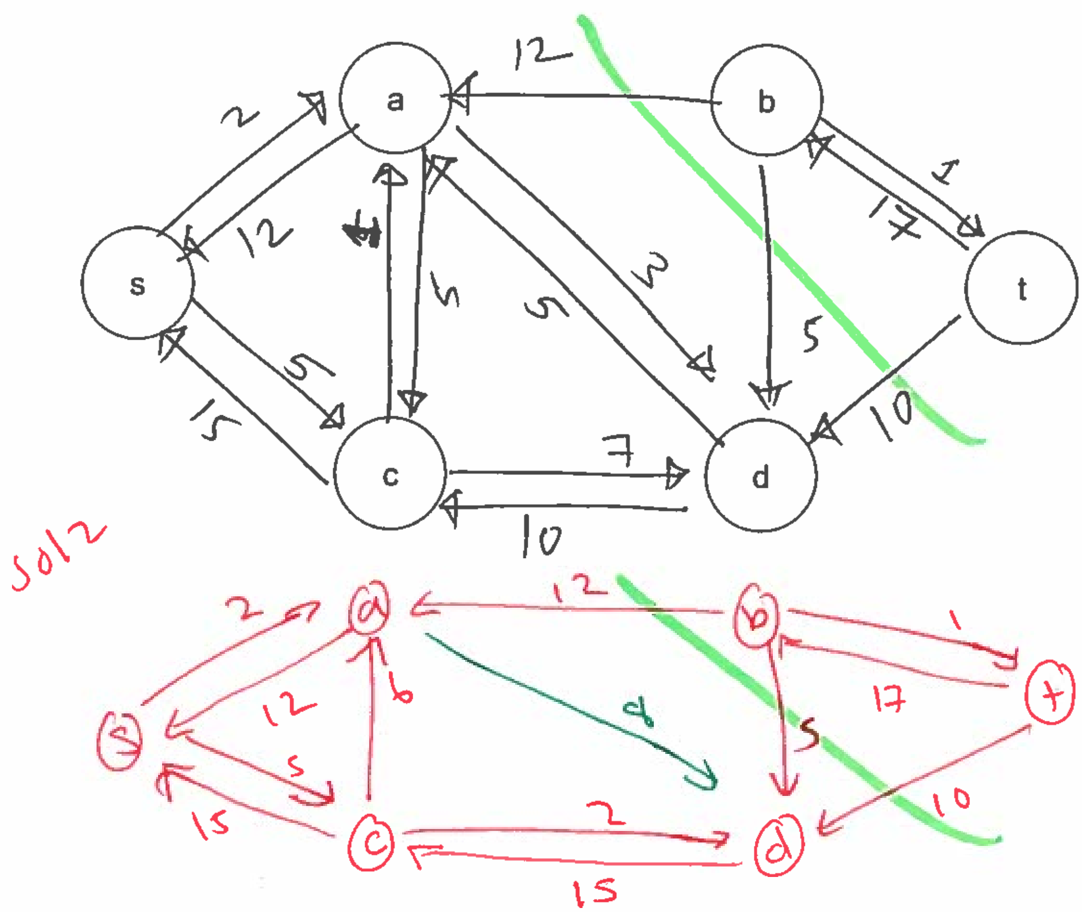
$$\left.\begin{array}{l} f_{sa} = f_{ab} + f_{ac} \\ f_{sc} + f_{ac} = f_{cb} + f_{cd} \\ f_{ab} + f_{cb} = f_{bd} + f_{bt} \\ f_{bd} + f_{cd} = f_{dt} \end{array}\right\} \text{Conservation constraint}$$

QUESTION 2. Network Flow/Linear Programming

For this question, refer to the flow network below.



(a) *(3 points)* Run the Ford-Fulkerson Algorithm on the given network. Draw the final residual graph G^f into the figure provided below. Extra copies of the network are given in the last sheet of the exam.



(b) *(2 points)* What is the value of the maximum flow?

27

(c) *(3 points)* List the edges that are in the minimum cut that corresponds to the maximum flow.

a – b
d – b
d – t

(d) *(4 points)* Write the problem of determining the maximum s–t flow in the given network as a linear program. (Don't leave out any constraints!)

Maximise $f_{sa} + f_{sc}$ subject to

$$\left.\begin{aligned}
0 &\leq f_{sa} \leq 14 \\
0 &\leq f_{sc} \leq 20 \\
0 &\leq f_{ab} \leq 12 \\
0 &\leq f_{ad} \leq 8 \\
0 &\leq f_{bt} \leq 18 \\
0 &\leq f_{ca} \leq 6 \\
0 &\leq f_{cd} \leq 17 \\
0 &\leq f_{db} \leq 5 \\
0 &\leq f_{dt} \leq 10
\end{aligned}\right\} \text{Edge constraints}$$

$$\left.\begin{aligned}
f_{ca} + f_{sa} &= f_{ab} + f_{ad} \\
f_{sc} &= f_{ca} + f_{cd} \\
f_{ab} + f_{db} &= f_{bt} \\
f_{cd} + f_{ad} &= f_{db} + f_{dt}
\end{aligned}\right\} \text{conservation constraints}$$

QUESTION 3. Minimum Spanning Tree

Given a connected, undirected, edge-weighted graph $G = (V, E)$ with all edge weights distinct.

Given a specific edge $e = (u, v)$ of G, design an $O(|V| + |E|)$ algorithm to decide whether e is contained in the minimum spanning tree of G.

[*Hint:* According to the Cycle Property, what would imply that e is NOT in the MST of G?]

(a) *(3 points)* Give an English description of your algorithm.

We will run a modified BFS algorithm from vertex u. We will ignore edges having weight greater than weight of e. If we are able to visit v, then it is not possible for e to be contained in the minimum spanning tree of G.

The above algorithm shows that if we are able to visit v then e is the heaviest edge in above cycle. By cycle property if e is the heaviest edge, it cant be contained in MST.

(b) *(3 points)* Give the pseudocode of your algorithm.

```
EdgeInMST ( G=(V,E), e(u,v))
    ∀ w∈V , mark w.visited = false.
    <stack> toVisit
    toVisit.push(u)
     u.visited = true.
      while toVisit is not empty:
          cur = toVisit.pop()
          for n in cur.neighbors:
                 if cost(cur, n) < cost(u,v) and n is not visited:
                       n.visited = true
                       toVisit.push(n)
      if v.visited : return False
          else      : return true
```

(c) *(4 points)* Explain the correctness and runtime complexity of your algorithm.

We are able to visit v from u using edges lighter than edge e, edge e is the heaviest edge in cycle. Using cycle property edge e can not be contained in MST.

We used modified BFS, so overall runtime for the algorithm is O(|V|+E|)

QUESTION 4. **Dynamic Programming**

A palindrome is a string that reads the same from front and back. Any string can be viewed as a sequence of palindromes if we allow a palindrome to consist of one letter.

The MinPal(w) problem is: Given a string of letters, w, what is the minimum number of palindromes whose concatenation is w? Note that a single letter is a palindrome.

For example, MinPal("bobseesanna") = 3, since "bobseesanna" = "bob" + "sees" + "anna" and we cannot write "bobseesanna" with less than 3 palindromes.

As another example, MinPal("abdcaba") = 5, since "abdcaba" = "a" + "b" + "d" + "c" + "aba".

Give an $\mathcal{O}(n^3)$ dynamic programming algorithm to find MinPal(w) for a given string w, where MinPal(w) is defined above.

[*Hint:* You will have to fill a $N \times N$ table for a string of size N.]

(a) *(2 points)* Give notation for, and explain, what is a subproblem that is solved in your dynamic programming algorithm.

Subproblem definition

Let minPal(i,j) be the minimum number of palindromes whose cocatenation is s[i,...j].

(b) *(2 points)* Write the recurrence used in your dynamic programming algorithm.

Recursive Formulation

$$\text{minPal}(i,j) = \begin{cases} 1 & \text{if } s[i..j] \text{ is a palindrome} \\ \min\limits_{i \le k < j} \{ \text{minPal}(i,k) + \text{minPal}(k+1,j) \} & \text{o/w.} \end{cases}$$

(c) *(3 points)* Give the pseudocode of your dynamic programming algorithm.

```
IsPalindrome (s)
    n = len(s)
    for i = 1 to n/2
        if s[i] is not equal to s[n-i]
            return False
    return True.

Minimum_Palindrome (s):
    minPal [i][i] = 1
    for j = 1 to n
        for i = 1 to n-j
            if IsPalindrome (s[i, i+j])
                minPal [i][i+j] = 1
            else minPal[i][i+j] = min_k { minPal[i][k] + minPal[k+1][i+j] }
    return minPal [1][n]
```

(d) *(1 point)* Analyze the runtime complexity of your dynamic programming algorithm.

We fill a table of size NxN

Filling each entry take O(N) time as we choose the k which minimizes no of palindrome.

Therefore total runtime complexity is $O(N^3)$

(e) *(2 points)* Fill the following table for string "aabab". Circle the answer for MinPal(aabab) in the filled-in table.

1	1	2	2	2
	1	2	1	2
		1	2	1
			1	2
				1

QUESTION 5. NP-Completeness Reduction

****PLEASE DO ONE OF THE TWO NP-COMPLETENESS REDUCTIONS MENTIONED BELOW (i.e. DOUBLE-SAT or SET_COVER)****

(1.) Reduction from SAT to DOUBLE-SAT

In Boolean logic, a formula F is in conjunctive normal form (CNF) if it is a conjunction (logical and, denoted $\wedge$) of clauses, each consisting of the disjunction (logical or, denoted $\vee$) of several literals, where a literal is either a Boolean variable or its negation. One example is:

$$F(x_1, x_2, x_3, x_4) = (x_1 \vee \neg x_2) \wedge (x_2 \vee x_3 \vee \neg x_1) \wedge (x_4 \vee \neg x_3).$$

A *satisfying* assignment is an assignment of 0 or 1 to each variable so that F evaluates to 1.

SATISFIABILITY

Input: A Boolean formula F in CNF.

SAT(F) Decision Problem: Does F have a satisfying assignment?

DOUBLE-SAT(F) Decision Problem: Does F have at least **TWO** distinct satisfying assignments?

Example 1: Consider $F1(x_1, x_2, x_3) = (x_1) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3)$.
SAT($F1$) = YES as $F1$ evaluates to 1 if $x_1 = 1, x_2 = 1$ and $x3 = 1$
DOUBLE-SAT($F1$) = NO as there does not exist an assignment of literals other than $x_1 = 1, x_2 = 1$ and $x3 = 1$ for which $F1$ evaluates to 1.

Example 2: Consider $F2(x_1, x_2, x_3) = (x_1) \wedge (x_2 \vee x_3) \wedge (\neg x_1 \vee x_3)$.
SAT($F2$) = YES as $F2$ evaluates to 1 if $x_1 = 1, x_2 = 1$ and $x3 = 1$
DOUBLE-SAT($F2$) = YES as there also exists another assignment of literals, precisely $x_1 = 1, x_2 = 0$ and $x3 = 1$ for which $F2$ evaluates to 1.

Given that SAT(F) is NP-complete, prove that DOUBLE-SAT(F) is NP-complete.

(a) *(3 points)* DOUBLE-SAT(F) $\in$ NP

For any input formula F and any two guess assignment we can verify in polynomial time, if they satisfy F. To verify ~~xxx~~ ~~xxx~~ we will plug in ~~li~~terals to see if it satisfies F.

(b) *(7 points)* SAT(F) $\leq_p$ DOUBLE-SAT(F)

let F denote the input Boolean formula to SAT problem and suppose that the set of variables in F are $X = \{x_1, \dots, x_n\}$. We construct a DOUBLE-SAT problem with Boolean formula F' over a new variable set X as follows:

- $X = \{x_1, \dots, x_n, y\}$.
- $F' = F \wedge (y \vee \neg y)$

Polynomial time for conversion.

Claim: F is satisfiable iff F' has at least 2 satisfying assignments.

If F is satisfiable, then we can add any ~~po~~ value of y (true or False) to give at least two satisfying assignmt for F'.

If F' has two satisfying assignments then F is satisfiable as well. For F' to be satisfiable both F and $(y \vee \neg y)$ will be satisfiable.

(2.) Reduction from VERTEX_COVER to SET_COVER

Given a set U of elements and a collection of subsets $S = S_1, S_2, \cdots, S_n$ of U whose union equals U, a *set_cover* is a sub-collection of S whose union equals U.
For example, consider $U = \{1, 2, 3, 4, 5\}$ and the collection of subsets $S = \{\{1, 2, 3\}, \{2, 4\}, \{3, 4\}, \{4, 5\}\}$ of U. Clearly the union of S is U. However, we can cover all of the elements with the following, smaller number of sets: $\{\{1, 2, 3\}, \{4, 5\}\}$.

SET_COVER(U, S, k)

Input: Given a set U of elements and a collection $S = S_1, S_2, \cdots, S_n$ of subsets of U whose union equals U.
Decision Problem: Is there a collection of k elements of S that covers U?

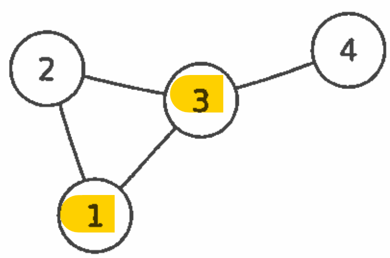
For the above example, Set_Cover$(U, S, 1)$ = NO, Set_Cover$(U, S, 2)$ = YES and Set_Cover$(U, S, 4)$ = YES

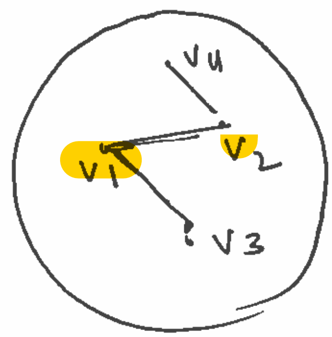
A *vertex_cover* of a graph $G(V, E)$ is a set $S \subseteq V$ of vertices such that every edge $e \in E$ has at least one endpoint in S.

VERTEX_COVER(G, k)

Input: An undirected graph $G = (V, E)$ and a nonnegative integer k.
Decision Problem: Does G have a vertex_cover of size k?

$F = \{ \{(v_1, v_2), (v_1 v_3)\},$
$\{(v_2, v_4)\} \{(v_2 v_1)\}$
$\{(v_3 v_1)\},$
$\}(v_4, v_2);$





G

Consider the undirected graph G shown above.
Example 1: VERTEX_COVER(G,2) = NO
Example 2: VERTEX_COVER(G,3) = YES as {1,2,3} is a vertex_cover
Example 3: VERTEX_COVER(G,4) = YES as {1,2,3,4} is a vertex_cover.

Given that VERTEX_COVER(G, k) is NP-complete, prove that SET_COVER(U, S, k) is NP-complete.

(a) *(3 points)* SET_COVER$(U, S, k) \in$ NP

A collection of set acts as a certificate and certifier can check in polynomial time if guess solution is subset of A and size of guess solution is less than or equal to k. And all the elements of U belongs to the set.

(b) *(7 points)* VERTEX_COVER$(G, k) \leq_p$ SET_COVER(U, S, k)

Given an instance of vertex cover $(G=(V,E), k)$ we will construct instance of Set Cover.
$\boxed{\text{let } U = E}$. we will define n subsets of U as follows: label each vertex of G from 1 to n. and let S_i be the edges incident to vertex i. and k is same as in vertex cover.

Suppose (G,k) is an Yes instance of vertex cover. Let T be such a set of nodes. By our construction T corresponds to a collection C of subsets of X. For any edge e in G, since T is a vertex cover for G at least one of endpoints is in T. Therefore C contains at least a set associated with endpoints of e, And by definition, there both contains e.

Now suppose there is a set cover C of size k. Since each set is naturally associated with a vertex in G, let T be the set of these vertex; $|T| = |C|$ thus T will contain at least one k vertex. Now C contains set that includes e for all $e \in E$. Thus T must contain at least one endpoints of e