COMP 3804 Assignment 3:

1. a) the three matterials have a maximum Volume of 60 meters:

$$m_1 + m_2 + m_3 \leq 60$$

the maximum weight is 100 tons, this can be expressed as:

$$100 \geq 2m_1 + m_2 + 3m_3$$

the total amounts of each matterial is as follows:

$$m_3 \leq 20$$
$$m_2 \leq 30$$
$$m_1 \leq 40$$

there is a net Positive amount of matterial: $0 \leq m_1, m_2, m_3$

We must find a linear Program which optimizes revenue w/in the Constraints of the objective

$$\boxed{\square\ Max(1000m_1 + 1200m_2 + 12\,000m_3)}$$

Its most optimal to max the highest Value goods in favour of the lowest Value ones, thus the optimal Sol^{n} is:

$$m_1 = 5 \rightarrow 5 \cdot 1000 = 5000$$
$$m_2 = 30 \rightarrow 30 \cdot 1200 = 36000$$
$$m_3 = 20 \rightarrow 20 \cdot 12000 = 240000$$

$\Rightarrow 5000 + 3600 + 240000 \boxed{= 281\ 000}$

thus, the most optimal revenue is: 281 000

Linear Program:

m_1 = matterial 1; m_2 = matterial 2; m_3 = matterial 3

$$Max(1000m_1 + 1200m_2 + 12\,000m_3)$$

$$\begin{cases} 0 \leq m_1, m_2, m_3 \\ 60 \geq m_1 + m_2 + m_3 \\ 100 \geq 2m_1 + m_2 + 3m_3 \\ 40 \geq m_1 \\ 30 \geq m_2 \\ 20 \geq m_3 \end{cases}$$

b) See Attachment on the next Page

Simplex method

Intermediate operations **(show/hide details)**

Pivot row (Row 2):
10 / 2 = 5
2 / 2 = 1
0 / 2 = 0
0 / 2 = 0
0 / 2 = 0
1 / 2 = 0.5
0 / 2 = 0
-1 / 2 = -0.5
-3 / 2 = -1.5

Row 1:
10 - (1 * 5) = 5
1 - (1 * 1) = 0
0 - (1 * 0) = 0
0 - (1 * 0) = 0
1 - (1 * 0) = 1
0 - (1 * 0.5) = -0.5
0 - (1 * 0) = 0
-1 - (1 * -0.5) = -0.5
-1 - (1 * -1.5) = 0.5

Row 3:
40 - (1 * 5) = 35
1 - (1 * 1) = 0
0 - (1 * 0) = 0
0 - (1 * 0) = 0
0 - (1 * 0) = 0
0 - (1 * 0.5) = -0.5
1 - (1 * 0) = 1
0 - (1 * -0.5) = 0.5
0 - (1 * -1.5) = 1.5

Row 4:
30 - (0 * 5) = 30
0 - (0 * 1) = 0
1 - (0 * 0) = 1
0 - (0 * 0) = 0
0 - (0 * 0) = 0
0 - (0 * 0.5) = 0
0 - (0 * 0) = 0
1 - (0 * -0.5) = 1
0 - (0 * -1.5) = 0

Row 5:
20 - (0 * 5) = 20
0 - (0 * 1) = 0
0 - (0 * 0) = 0
1 - (0 * 0) = 1
0 - (0 * 0) = 0
0 - (0 * 0.5) = 0
0 - (0 * 0) = 0
0 - (0 * -0.5) = 0
1 - (0 * -1.5) = 1

Row Z:
276000 - (-1000 * 5) = 281000
-1000 - (-1000 * 1) = 0
0 - (-1000 * 0) = 0
0 - (-1000 * 0) = 0
0 - (-1000 * 0) = 0
0 - (-1000 * 0.5) = 500
0 - (-1000 * 0) = 0
1200 - (-1000 * -0.5) = 700
12000 - (-1000 * -1.5) = 10500

Tableau 4			1000	1200	12000	0	0	0	0	0
Base	**C_b**	**P_0**	**P_1**	**P_2**	**P_3**	**P_4**	**P_5**	**P_6**	**P_7**	**P_8**
P_4	0	5	0	0	0	1	-0.5	0	-0.5	0.5
P_1	1000	5	1	0	0	0	0.5	0	-0.5	-1.5
P_6	0	35	0	0	0	0	-0.5	1	0.5	1.5
P_2	1200	30	0	1	0	0	0	0	1	0
P_3	12000	20	0	0	1	0	0	0	0	1
Z		281000	0	0	0	0	500	0	700	10500

☐ Show results as fractions.

The optimal solution value is Z = 281000
$X_1 = 5$
$X_2 = 30$
$X_3 = 20$

ℓP = Linear Program

② Linear Program:

$$\max(x, y)$$
$$\begin{cases} ① & ax + by \le 1 \\ ② & x, y \ge 0 \end{cases}$$

a) The linear Program is never infesible because no matter the value of a or b, the origin value (0,0) will satisfy the constraints of $ax+by \le 1$, no matter the value chosen for a or b.

b) For the ℓP to be unbounded, either a or b must be less then or equivilent to 0. $(a,b \le 0)$

- When $a,b \le 0$, then x (for a) or y (for b) can be increased to any amount w/out making $ax+by \nleq 1$.

this will not work when $a,b > 0$

Math:

$\min(a,b)x + \min(a,b)y \le ax+by$

$1 \ge ax+by$

$x+y \ge 0$

$\min(a,b)x + \min(a,b)y \le 1$

$x+y \le 1/[\min(a,b)]$

$\frac{1}{\min(a,b)}$ ∝ • Cannot be unbounded because it will approch Zero, breaking $ax+by \le 1$

- The minimum of a & b will be greater then zero, as both a & b are themselves greater then zero
- $x, y \ge 0$, thus if the minimum of a & b is positive, then when x or y are unbounded, the product of $ax+by$ will also be unbounded.

therefore, for the ℓP to be unbounded, a or b must be less then or equal to zero.

c) For the linear Program to have a unique optimal Sol$^{\underline{n}}$:

- As seen above, we know that when a & b are both positive, the ℓP has a specific Sol$^{\underline{n}}$ w/in the programs constraints $\boxed{(a,b>0)}$
- having identical values for a & b can allow for an array of possible Sol$^{\underline{n}}$'s in x & y $(a=b)$. x & y can be interchanged, allowing for multiple valid Sol$^{\underline{n}}$'s. This means that for there to be a unique Sol$^{\underline{n}}$, we can't have $a=b$ $(a \ne b)$
  - If $a=b$ then any permutation of x & y such that $x+y = 1/b$ is optimal, thus there is no unique optimal Solution.
- If the ℓP is unbounded, then there can't be any optimal Sol$^{\underline{n}}$

— The optimal Sol$^{\underline{n}}$ is the Sol$^{\underline{n}}$ w/ the Max. or Min. possible Problem.

— When $a>b$, $x=0$; when $b>a$, $y=0$

Case 1

- increasing y by any amount will decrease the result. increasing x by any amount will do the same.

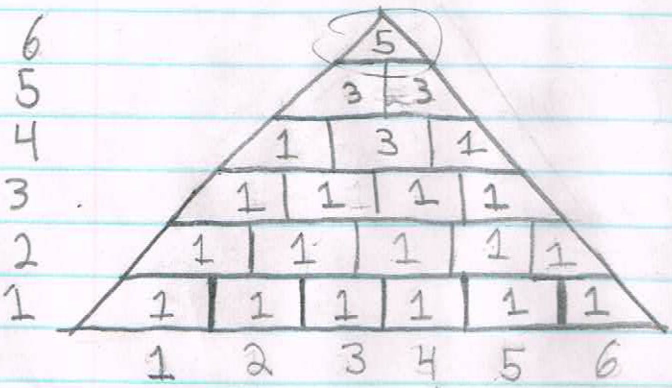
(2)

— The optimal value is represented as a line, w/ the optimized value of this line representing the limits of the ℓP's constraints. When the optimal value line intersects w/ the constraints, it represents an optimal Sol$^{\underline{n}}$: when $a=b$, the lines fully overlap, causing there to be ∞ points of interception

please see the * $a \ne b$

③ a)

- we will have a String of length n
- the last and first letters are the Same
  - If we Count from the left side of the Subsequence, then that letter Should match the right side of the Subsequence at the Same index:
    - For example: 1 2 3 4 5 6
      - 1 Should be the Same letter as 6, 2 Should match 5, & 3 Should match 4.
- we can find Some Stuff in less then n^2 time, for example, getting the length of the String Should take $O(n)$ time
- because we are allowed to drop characters w/in the String, the # of Sol'n's is taken as a Square of the total length of the String. Thus, we must check all Potential Sol'n's by Storing them in a 2D representation (i.e., a 2D array)
- we can check over the String to find in $O(n^2)$ time by comparing at every element to all future elements. For example:



- XYZGYX has a Shortest Subsequence of: XYZYX, the Alg. would find this out the following operations:
  - Level 1: XYZGYX — i & j cover all the letters
    - index i: ↑ . . . . . .
    - index K: ↑ . . . . . ↑
  - Level 2: XYZGYX — i & j cover exactly n-1 letters
    - i: ↑ . . . . .
    - K: ↑ . . . . . .
  - Level 3: XYZGYX — n-2 letters
    - i: ↑ . . .
    - K: ↑ . . .
  - Level 4: XYZGYX — n-3 letters
    - i: ↑ . . . . . .
    - K: ↑ . .
  - Level 5: XYZGYX — n-4 letters
    - i: ↑ . . . . .
    - K: ↑ .
  - Level 6: XYZGYX
    - i: ↑ . . . . .
    - K: ↑

- Rules: If a String has 2 or more characters, it must be a Palendrome of Size 3 minimum; for example:

AXYZA

↓ we can cut out any character we want

AXA → Size 3

Correctness:

- If 2 characters next to each other may form a Palendrome w/ 2 characters later in the String, then the Palendrome is of length 5. For example:

$$AB X_1 X_2 X_3 BA$$
$$\downarrow$$
$$AB X_3 BA \longrightarrow \text{length } 5$$

- We may conclude that any character sequence that can form a Palendrome later in the String will add 2 to the length of the longest Palendrome.
- If a character can be added to the middle of the Subsequence, we add +1 to its length. This can only occur when index k is 2 greater then index i.

Algorithm:

```
-> largestPalindromicSubsequence (String);
   — length = SizeOf(String) //Get length of String, O(n) time
   — Array has dimention of length by length
   — Set all elements of Array to have size 1
            //each character can be a Singleton Palendrome
   — FOR i in Size length:
   —     let a = 0 & p = i
   —     FOR k in Size length:
             IF ( p >= length ):
                break
             ELSEIF ( String[a] == String[p+a] ):
                IF ( abs(a-p) < 2 ):
                   Array[a][p] = 2
                ELSE:
                   Array[a][p] = Array[a+1][p-1] + 2
                ENDIF
             ELSE:
                Array[a][p] = (Array[a+1][p] > Array[a][p-1]) ? Array[a+1][p] : Array[a][p-1]
             ENDIF
             let a be a+1
             let p be p+1
```

END FOR
END FOR
Return largestIn(Array)
END largestPalindromicSubsequence

— Note:

Array[i+1][p-1] checks if the current element being searched extends the outside of a palindrome: i.e.

BAB → PBABP

extends Pal. by 2

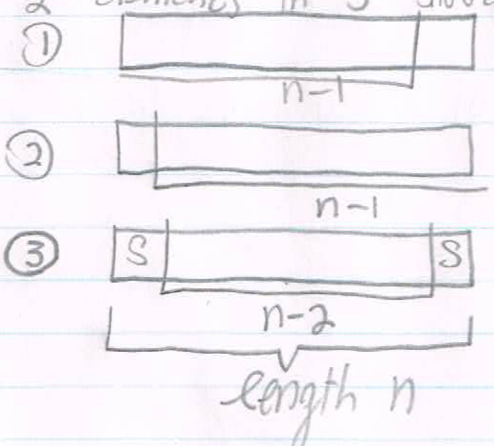
Array[a+1][p] & Array[a][p-1] both represent the fact we can't remove characters in between a palendrome.

b) The Alg. has 2 FOR loops running on the list size, n; we can conclude the worst case runtime is

$$O(n) \cdot O(n) = O(n^2)$$

Proof for Alg.:

— We have an array of size n, & we can get a palendrome between 2 elements in 3 different ways:



— We can find the maximum palendromic subsequence by splitting the excess elements using one of these three ways.

Induction:

Base case: Size 1

return String, its a singleton palendrome [by ① or ②]

Hypothesis:

the max. palindrome can always be produced by splitting a subsequence using method 1, 2, or 3.

Step:

take the max of ①, ②, ③

this is the correct awnser, it works for all 3 cases:

Case 1: the last value to the right isn't in the Palendrome

$\underbrace{SXYZYXS}\Delta \Rightarrow$ returns correct Pal.

Case 2: the left value isn't in the Palendrome

$\Delta\underbrace{SXYZYXS} \rightarrow$ ret. correct Pal.

Case 3: the whole substrings in the Palendrome

$\underbrace{SXYZYXS} \rightarrow$ correct Pal.

④

Consider the 5 matrixes:

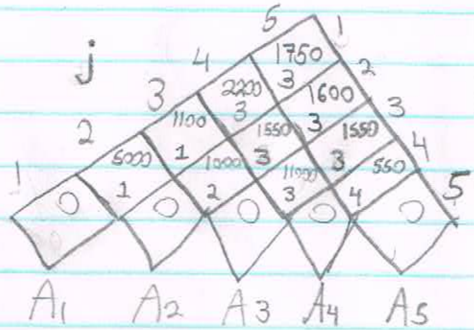
A_1 : 10×5

A_2 : 5×100

A_3 : 100×2

A_4 : 2×55

A_5 : 55×5 →



Top row is the m table, & bottom is the S table.

Compute $m(1,2)$: $A_1 \times A_2$

$A_1 \times A_2 = 10 \times 5 \times 100 = 5000$ — $A_1 = A$

Compute $m(2,3)$: $A_2 \times A_3$ — $A_2 = B$

$A_2 \times A_3 = 5 \times 100 \times 2 = 1000$ — $A_3 = C$

Compute $m(3,4)$: $A_3 \times A_4$ — $A_4 = D$

$A_3 \times A_4 = 100 \times 2 \times 55 = 11000$ — $A_5 = E$

Compute $m(4,5)$: $A_4 \times A_5$

$A_4 \times A_5 = 2 \times 55 \times 5 = 550$

Compute $m(1,3)$: $A_1 \times A_2 \times A_3$

$K_1 = 10 \times 5 \times 2 = 100 + \overset{m(2,3)}{1000} = 1100$ → Smaller : $A_1(A_2A_3)$

$K_2 = 10 \times 100 \times 2 = 2000 + \underbrace{5000}_{m(1,2)} = 7000$: $(A_1A_2)A_3$

Compute $m(2,4)$: $A_2 \times A_3 \times A_4$

$K_2 = 2$: $5 \times 100 \times 55 = 27500 + \overset{m(3,4)}{11000} = 38500$: $A_2(A_3A_4)$

$K_3 = 3$: $5 \times 2 \times 55 = 550 + \underbrace{1000}_{m(2,3)} = 1550$: $(A_2A_3)A_4$

Compute $m(3,5)$: $A_3 \times A_4 \times A_5$

K_3: $100 \times 2 \times 5 + \overset{m(4,5)}{550} = 1550$: $A_3(A_4A_5)$

K_4: $100 \times 55 \times 5 + \underbrace{11000}_{m(3,4)} = 38500$: $(A_3A_4)A_5$

Compute $m(1,4)$: for $A_1 \times A_2 \times A_3 \times A_4$

K_1: $A_1(A_2A_3A_4)$: $10 \times 5 \times 55 + \overset{m(2,4)}{\boxed{1550}} = 4300$

K_2: $(A_1A_2)(A_3A_4)$: $10 \times 100 \times 55 + \underbrace{5000}_{m(1,2)} + \underbrace{11000}_{m(3,4)} = 71000$

K_3: $(A_1A_2A_3)A_4$: $10 \times 2 \times 55 + \underbrace{1100}_{m(1,3)} = 2200$

Compute m(2,5): for $A_2A_3A_4A_5$

K_2: $A_2(A_3A_4A_5)$: $5 \times 100 \times 5 + \overset{m(3,5)}{\boxed{1550}} = 4050$

K_3: $(A_2A_3)(A_4A_5)$: $5 \times 2 \times 5 + \overset{m(2,3)}{\boxed{1000}} + \overset{m(4,5)}{\boxed{550}} = 1600$

K_4: $(A_2A_3A_4)A_5$: $5 \times 55 \times 5 + \underset{m(2,4)}{\boxed{1550}} = 2925$

Compute m(1,5): for $A_1 \times A_2 \times A_3 \times A_4 \times A_5$

K_1 : $A_1(A_2A_3A_4A_5)$: $10 \times 5 \times 5 + \overset{m(1,1)}{\boxed{0}} + \overset{m(2,5)}{\boxed{1600}} = 1850$

K_2 : $(A_1A_2)(A_3A_4A_5)$: $10 \times 100 \times 5 + \overset{m(1,2)}{\boxed{5000}} + \overset{m(3,5)}{\boxed{1650}} = 11550$

K_3 : $(A_1A_2A_3)(A_4A_5)$: $10 \times 2 \times 5 + \overset{m(1,3)}{\boxed{1100}} + \overset{m(4,5)}{\boxed{550}} = 1750$

K_4 : $(A_1A_2A_3A_4)A_5$: $10 \times 55 \times 5 + \overset{m(1,4)}{\boxed{2200}} + \overset{m(5,5)}{\boxed{0}} = 4950$

∴ The optimal Parenthesization is:

(ABC)(DE) which can be broken further into

((A(BC))(DE))

the # of operations taken is: 1750

- optimal Parenthisization is obtained through the S table because the S table tracks the Progress of the m table.
  - The S value tells us what matrix we should break our multiplication at, Recording the S value allows us to know the optimal Parenthesization for each combination of matrixes.

- Enteries are derived based on the # of operations needed for the matrix multiplication. we are trying to get the smallest # from our multiplication table, as this represents the most efficient way to multiply the matrixes.

⑤ Find the optimal Value for the following linear Program.

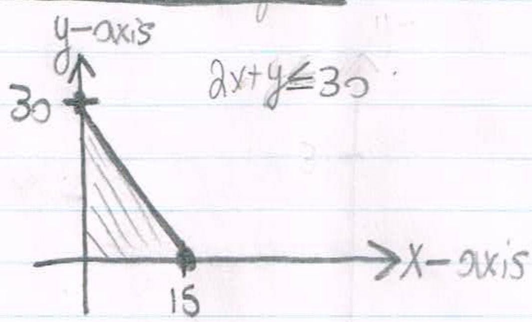
- Plot feasible reagon
- Evaluate the value of the objective function at each of the vertices
  - □ Find which one achives the optimum
- linear Program:

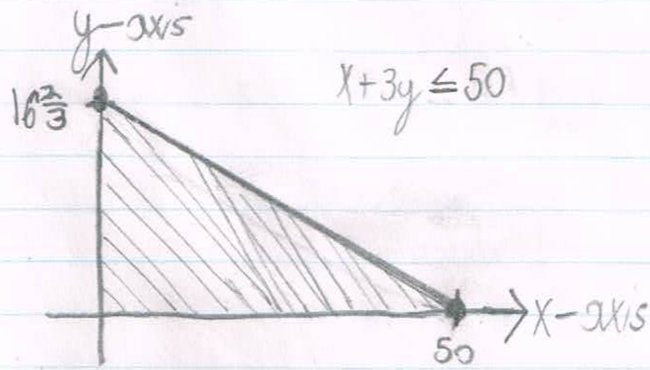
Maximize $(x+y)$

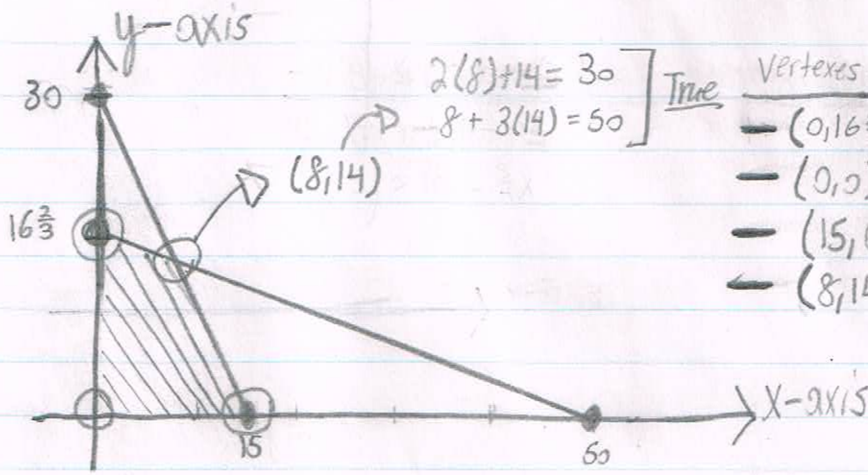
$$\begin{cases} x \geq 0 \\ y \geq 0 \\ 2x + y \leq 30 \\ x + 3y \leq 50 \end{cases}$$

$$\frac{50}{3} = 10 + \frac{20}{3}$$

Plot feasible Reagon:







$$\left.\begin{array}{l} 2(8)+14 = 30 \\ 8 + 3(14) = 50 \end{array}\right] \text{True}$$

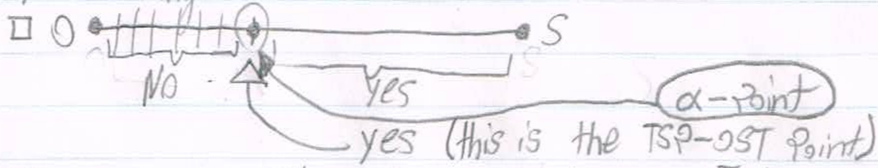
Vertexes:		total max(x+y):
$(0, 16\frac{2}{3})$	→	$16\frac{2}{3}$
$(0,0)$	→	0
$(15,0)$	→	15
$(8,14)$	→	22

∴ The Vertex at (8,14) achives the optimum, which is a value of 22

⑥

— Assuming TSP Can be Solved in Polynomial time, we must Solve TSP-OPT in Polynomial time.

- let S be the Sum of all the TSP Routs in the graph
- The minimum Possible TSP-Rout is in the range of 0 to S
  - min(E) = 0 when theres a TSP-Rout of Length zero
  - min(E) = S when theres 1 Rout, or if all other TSP-Routs are length 0 in the graph
- The machine TSP can tell if a Paths Possible, we can thereby tell if a Value is the Correct Path.
- Theres a range of Paths between 0 & S where TSP will return a yes, & a range where it returns a no.



- let us assume all Rout lengths are Integers, the S (the Sum of all edge weights) is also an Integer.
- We can Simply run a binary Search through each int in S in order to determine the α-Point (the TSP-OST).
  - To do this, we must run a binary Search through the range of Zero to S, & find a int that TSP returns a true for, but won't return true for the Previous int.
  - A binary Search will take S log(S) time, where S is the Integer Value of all elements in the range of Zero to S.
    - The S is because we run a binary Search on each int
    - The log S is the runtime for each binary Search
    - The total runtime is S log(S) as Stated before
      - S log(S) is a Polynomial runtime

∴ The TSP-OPT Can return the Shortest tour in Polynomial time, the above Procedure would take O(S log(S)).

(7) We can prove NP—Completness using two steps: first we must show Stingy SAT is in NP, then reduce it to NP—Complete.

— Prove Stingy SAT is NP:

- let a be the number of variables
- let b be the number of clauses
- We can check if a input satisfies the problem by checking clause by clause that the input satifies them
- This will requires us to search all variables for each clause, which will take O(a) time. This problem will also require us to search each clause which will take O(b) time.
- Finally, we check if the # of true variables is larger then K by iterating through through all the variables & counting the # of trues (takes O(a) time it iterate).

— Reduce Stingy SAT to NP—Complete:

- We need to reduce SAT to Stingy SAT
- let P be an instance of SAT which has j variables. We can simply take the input & run the Stingy SAT algorithm on it, using the known number of variables (j) to represent the integer K.

☐ SAT is NP-Complete, & Stingy SAT is easier the SAT

Proof: Prove that a given input for SAT is correct only if the input w/ its j variables is also a correct input for Stingy SAT

- (for SAT): If SAT is correct, then all n variables are true, as they were all evaluated in SAT. Stingy SAT has a subset of the variables from the original input, ∴ Stingy SAT must also be correct as all the variables are true anyways.
- (for STINGY-SAT): If Stingy-SAT returns true for the same input w/ j variables, then the evaluated instance satisfied all the constraints, & this evaluated instance would also evaluate to true for SAT
- If either SAT or STINGY-SAT return false for a shared instance, then Neither can evaluate that instance to True.

∴ (STINGY-SAT $\geq_P$ SAT) Stigy SAT's harder then SAT

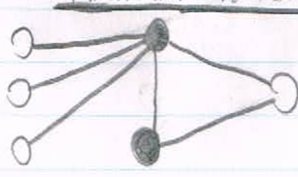
- We know SAT $\in$ NP-Complete : SAT's NP-Complete (as its NP-Hard)

∴ STINGY-SAT $\in$ NP-Complete → STINGY-SAT is NP-Complete

⑧

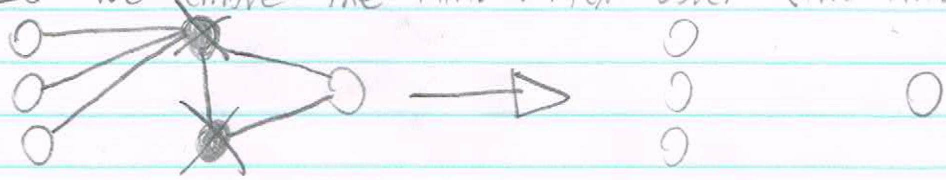
- We must prove the Smallest Killer Set is a NP-hard Problem
  - We must thus show that the Problem is at least as hard as any NP-Complete Problem.
    - To do this, a reduction from any known NP-complete Problem must be done.
- <u>Killer-Set Problem:</u>
  - $G=(V,E)$ is both undirected & unweighted
  - let S be the subset of all vertices
  - $G[S]$ is graph w/ vertex set S & an edge between two vertices $u,v \in S$ Provided $\{u,v\}$ is an edge of G.
  - Killer Set - A subset K of G, wherein the deletion of K kills all the edges of G, such that $G[V-K]$ has no edges.
  - Prove that finding the Smallest Killer Set of G is an NP-hard Problem.
- <u>Reduce Killer-Set to NP-Complete:</u>
  - Finding the Smallest Killer Set would require the Algorithm to be an optimization Problem, as a value must be minimized.
    - This means we cannot apply the theory of NP-Completness
    - We can however reduce to NP to prove its atleast NP-hard
  - In order to Solve this Algorithm, we must find the Vertex Cover of the Graph, Specifically the minimum vertex Cover.
    - The vertex cover is a set of verticies that allow each each edge of the graph to be incident to atleast 1 vertex of the Set.
    - A min. vertex cover is the Smallest Possible vertex Cover
  - In order for a K to be a Killer Set, it must be incident upon all edges in G, Such that its deletion destroys all edges in G. Thus, it is by definition the minimum vertex Cover.
  - illustration: See the example below

<u>minimum Vertex Cover:</u>



- Notice that if we delete the vertexes w/ their respective edges, then all edges in the Graph are Killed, ∴ the min. vertex cover is the Killer Set.

- If we remove the min. vertex cover (the Killer Set):



  - Notice that all edges were indeed Killed
- Finding the min. vertex cover is NP-hard, & finding the vertex cover is NP, to show the reduction; follow below:
  - The vertex cover can be reduced from the clique Algorithm:
    - Clique $\leq_p$ Vertex-Cover:

① $f: \underbrace{(G,K)}_{\text{input for clique}} \longrightarrow \underbrace{(G',K')}_{\text{input for Vertex-cover}}$

② G has a clique of size K $\Longleftrightarrow$ G' has a vertex of size K'

③ f can be computed in polynomial time

- Clique & Vertex-cover are known NP-complete Problems

- Vertex-Cover is NP-Complete: Reduce vertex-cover to min. vertex cover
  let g be an algorithm that computes the vertex covers of a Graph & returns the min. vertex cover, then g must always have at least the same difficulty as vertex cover:
  thus, min.-vertex cover $\geq_p$ vertex-cover

— In conclusion:

- A Killer Sets a min. vertex cover
- A min. vertex cover is atleast as hard as a vertex cover
- A vertex-covers at least as hard as a clique
- Thus: Killer-Set = min.-vertex-cover $\geq_p$ Vertex-Cover $\geq_p$ Clique
- We know clique & vertex cover $\in$ NP-Complete

∴ min.vertex-cover $\geq_p$ NP-Complete & min. vertex-cover = Killer-Set

- Killer-Set $\geq_p$ NP-Complete
- ∴ Killer-Set is NP-hard