1: Easy relatives of 3-SAT

(a) For any given CNF to be satisfiable, every associated clause needs to be true for the given set of variables. let's take the example of given clause $x_1 \vee x_2$. let's write a truth table considering this as the only clause. From the below truth table, it is evident that either both of the variables or at least one of them needs to be TRUE for the CNF to be satisfiable.

x_1	x_2	*output*
false	false	false
false	true	true
true	false	true
true	true	true

Now let's consider the implication formula and analyze the same in terms of the truth table mentioned for $x_1 \vee x_2$. for cases where x_1 is true, we can see that irrespective of whatever value x_2 carries, the output will be true, hence in k-map reduction the term x_2 becomes insignificant. for cases where x_1 is false, we can see that the formula is satisfiable (output is true) if and only if x_2 is true. which essentially means that the k-map reduces to a form where x_2 has to be complement of x_1 and hence we have the implication proof $\overline{x}_1 \implies x_2$ (if at all x_1 is false, x_2 must be true for the CNF to be satisfiable, the complementary condition also holds good but is insignificant) inferred logically from the truth table.

(b) we will prove the theory by contradiction. Let's assume that there exists a path between x_i to $\overline{x}_i$ in a given graph G, and there also exists a satisfying assignment $\theta(x_1, x_2...x_n)$ for a formula in ϕ. The determining of existence of any edge between two given nodes(vertices's) can be done through either BREADTH FIRST SEARCH or DEPTH FIRST SEARCH algorithms.

Proof. :

Let's consider a first case where the satisfying assignment $\theta(x_1, x_2...x_n)$ results in X=TRUE.

Let's imagine a graph of path x_i to $\overline{x}_i$ given as
$x_i -> .... -> \alpha -> \beta -> .... -> \overline{x}_i$.

From a given set of graph clauses, we know that there exists an edge between P to Q in graph G IFF there is a clause $(\overline{P} V Q)$ in ϕ. The edge from P to Q represents that if P is TRUE, then Q must be TRUE (for the clause to be TRUE). Now since X is TRUE, all literals in path from x_i to α (including α) must be TRUE. Similarly, all literals in the path from β to $\overline{x}_i$ (including β) must be FALSE (because $\overline{x}_i$=FALSE). This results in an edge between α and β, with α = TRUE and β = FALSE. Consequently the clause $(\overline{\alpha} \vee \beta)$ becomes FALSE, contradicting our assumption that there exists a satisfying assignment $\theta(x_1, x_2....x_n)$ for ϕ.

Let's consider the second case where the satisfying assignment $\theta(x_1, x_2...x_n)$ results in X=FALSE.

Let's imagine the same graph of path $\overline{x}_i$ to x_i given as
$\overline{x}_i -> .... -> \alpha -> \beta -> .... -> x_i$.

keeping the rules of directed graph explained above in mind, let's approach the scenario. Now since X is FALSE, all literals in path from $\overline{x}_i$ to α (including α) must be FALSE. Similarly, all literals in the path from β to x_i (including β) must be TRUE (because x_i=TRUE). This results

in an edge between α and β, with α = FALSE and β = TRUE. Consequently the clause $(\alpha \vee \overline{\beta})$ becomes TRUE, contradicting our assumption that there exists a satisfying assignment $\theta(x_1, x_2....x_n)$ for ϕ.

Hence, by checking for the existence of an edge between nodes x_i to $\overline{x}_i$ and/or $\overline{x}_i$ to x_i path in the graph G, we can decide whether the corresponding 2 CNF expression ϕ is satisfiable or not. Traversal algorithms like BFS and DFS take polynomial time of O(V + E) time complexity. where V = number of vertices's and E = number of edges in graph G. Hence 2SAT can be solved in polynomial time P.

(c) For a given CNF with 3 literals per clause, let us assume 3 arbitrary literals (x_p, x_q, x_r) in a clause, where each literal can be within (x_i, $\overline{x_i}$, for variables $\{x_i\}_{i=1}^{n}$).

we shall approach the *2-or-more 3-SAT* as *1-or-more 2-SAT* problem, that is we will divide every 3-SAT clause into CNF of 2-SAT clauses. for example

$(x_p \vee x_q \vee x_r)$ can be written as

$((x_p \vee x_q) \wedge (x_p \vee x_r) \wedge (x_q \vee x_r))$

with every 3 literal clause split as these two literal clauses in CNF form, now we will check for at least 1 literal in every clause evaluating to true, which can be implemented in any standard algorithm. we shall take the Krom's algorithm to implement the same -

Algorithm: :

1)Reduce the original 2-CNF Boolean formula clauses using Krom's inference rule.

$(x_i \vee x_j) \wedge (\overline{x}_i \vee x_k) => (x_j \vee x_k)$

2) While preserving consistency, for each variable in the original formula, if you have generated these special clauses: $(x_i \vee x_i) \wedge (\overline{x}_i \vee \overline{x}_i)$, then there is no solution.

3)Otherwise, if for some variable x_i we have generated neither $(x_i \vee x_i)$ nor $(\overline{x}_i \vee \overline{x}_i)$, add one of them as you please and go back to step 1.

4)Output the truth assignment satisfying all literals x_i for which we have generated $(x_i \vee x_i)$.

Correctness: : The implementation can also be verified using Breadth first search algorithm as well. the implications reasoned by retaining the original formula of the CNF can be used to reconstruct the 3-SAT problem by adding a variable and it's complement in CNF form for every 2 literals or the same variable can be OR'd thrice to reconstruct a one literal function as 3-SAT.

Running time: : the complexity is derived by multiplying the number of variables by the $O(n^2)$ number of pairs of clauses involving a given variable, to which the inference rule may be applied. Thus, it is possible to determine whether a given 3-SAT/2-CNF instance is satisfiable in time $O(n^3) + constant$ which is polynomial, where the *constant* time here is to split a given clause into 2-CNF clause's.

2: Decision vs Search

Proof: For any given graph G = (V, E) and any $v \in$ V let $\overline{G}_v$ denote the graph formed from G by removing v and all edges adjacent to it. Let G be the given graph, and let $v_1, ..., v_n$ be its vertex set. By problem statement we are given an Oracle Θ which takes (G, K) as inputs and outputs 1

if G has an independent set of size K, and 0 otherwise.

The pseudocode to implement the polynomial time call to oracle is given as follows:

Algorithm: :

If Oracle(G, K) = 0 then return *nil* and halt

For i = 1. . .n do

If Oracle($\overline{G}_{vi}, K$) == 1 then

$G \leftarrow \overline{G}_{vi}$

End //for loop

Repeat

Arbitrarily pick a vertex v in G and let $G \leftarrow \overline{G}_v$

Until G has exactly K nodes

Output G

The idea is to remove nodes, one by one from G, and see if it affects the independent set size. For the analysis, let G_i be the graph after the i^{th} execution of the loop. $G_0 = G$ as the first iteration will have the whole graph as input. the independent set here can also be an out put whose vertex count can be greater than or equal to K.

We know from the first step that G_0 has a independent set of size K, if at all it existed in the first place, else we shall crash out of the algorithm. The following loops will be true for all $i \geq 1$:

(1) If v_i is not removed, then all independent sets of size at least K in G_{i-1} require v_i to be part of it.

(2) G_i contains an independent set of size K or more.

Correctness: : For a given vertices's w, as independent set in G_w is also a independent set in G_k for $w \geq k$. this essentially means that for any given vertices's w, the graph G_w also has an independent set of size K, and for any $k \leq w$, if v_k is in G_w, then it is a member of all independent sets of size K in G_w. This means that in G_n, there is a independent set of $size \geq K$, and all vertices's are members of all independent sets of $size \geq K$. So G_n must itself be an independent set of $size \geq K$.

Running time: : the outer for loop runs O(n) times considering the total vertices's set of n. The Oracle returning a 1/0 inside for loop may be running at a polynomial time, let us assume it to be O(n) from the question. The arbitrary v deletion also can happen until $w_s - K$ times, where w_s is a independent set with w vertices's which is $\leq n$ but $\geq K$, and is outside the for loop. so we can assume the complexity as O(nxn) which is polynomial in nature with input vertices's n and ignore the arbitrary function complexity.

3: ILP/SAT reductions

(a) The hardness of a problem P can be devised by a polynomial reduction from another problem Q that is known to be NP-HARD. In this case we will reduce a SAT problem from ILP and show that there exists a function f that maps each instance SAT_i of the problem SAT to an instance

of the problem ILP such that: SAT_i is a *yes* instance for $SAT \leftrightarrow f(SAT_i)$ is a *yes* instance for ILP. This implies that solving $f(SAT_i)$ must be "at least as difficult" as solving SAT_i in itself.

Let's take a individual SAT clause $C_1 = (x_1 \vee x_2 \vee \overline{x}_3)$. This is satisfied with the same solution as the solution for the following set of equations and hence can be reduced to the below form -

the instance constraints can be defined as -

$x_i = y_i \in 0, 1$

$\overline{x}_i \equiv (1 - y_i)$

the SAT can now be expressed as

$x_1 + x_2 + 1 - x_3 \geq 1$

where $0 \leq x_i \leq 1$

This is in ILP form now and for any given m such clauses $(C_1..C_m)$ with n variables, there can be $(m + 2n)$ such equations for a SAT problem. Despite some specific instances of ILP which can be *solved* in polynomial time, in general ILP's are NP-HARD to *solve* i.e. major instances of ILP that are NP-HARD.

(b) for a given set of n ILP equations, unless there are additional operations such as maximize or minimize are requested, every ILP problem can be $verified$ in polynomial time. Thus, in general, ILP's are NP-HARD to *solve*, but can be $verified$ in polynomial time. therefore, ILP's are NP-complete.

(c) YES!! for a given $LINEQ$(mod2) style of equations, the given ILP can be reduced to a SAT. starting with the first row, create a boolean variable for representing each bit in coefficient a_{1j}, one more boolean variable to represent the monomial x_j, and a last variable to represent b_1. Make an *adder* circuit adding all m such rows.

Then make a comparison circuit, declaring the sum to $< b_1$.

Convert these two circuits to CNF, filling in the a_{1j} variables and b_1 since they are computed using boolean variables.

Repeat for all rows, but reuse the x_j variables between them.

The final CNF will contain all the constraints from original equations, and is in polytime owing to polytime reduction.

(d) For a given $QUADEQ$(mod2) equation of the form $\sum_{i,j} c_{ij} x_i x_j = b$, where c_{ij} and b are 0/1 (and addition/multiplication are mod 2), to prove that it is NP-complete, we need to show that it is polytime reducible from a 3-SAT and determine the term which results in a NP-HARD complexity and break the sub-problem into .

let's try to reduce a SAT expression into a $QUADEQ$. let's assume a clause expression

$Y_2 = (x_1 \vee x_2)$, which can be written as

$Y_2 = x_1 \; XOR \; x_2 \; XOR \; x_1 * x_2$

which is equivalent to $x_1 + x_2 + x_1 * x_2$ (where addition is mod2).

similarly a 3-SAT can be expressed as

$Y_3 = (x_1 \vee x_2 \vee x_3)$

$Y_3 = x_1\ XOR\ x_2\ XOR\ X_3\ XOR\ x_1 * x_2 * x_3$

which is equivalent to $x_1 + x_2 + x_3 + x_1 * x_2 * x_3$

from the above expression, $x_1 * x_2 * x_3$ kind of cubic expressions are NP-HARD complex.

let us try to break this cubic expression by introducing a new variable x_i such that

$x_i = x_2 * x_3 * x_4 \implies (x_i - (x_2 * x_3 * x_4) = 0)$

we can further replace the negated terms in 3-SAT clause with $(1 - x_i)$.

The final resulting expression will have some linear terms. we can simply raise their power by 1 to make them quadratic and the equation will still have the same solutions. The overhead comes with addition of m number of such x_i variables per clause for m such clauses from a given equation. From the above equations, it's clear that 3-SAT solution is NP-HARD to come-by, but can be $verified$ in polynomial time and hence given $QUADEQ$ forms are NP-complete.

4: Graphs-definitions

(a) we shall try to prove the case by contradiction. Let's consider a finite graph G without any cycle and with a longest path, say L. Let z be the final vertex in our path L. Since each vertex of G has a degree of 2 or more, let's assume that z has a degree of 2 as well. Since z has a degree of 2, it has 2 edges, say e_1 and e_2. If e_1 was the edge used in the path L to reach z, we still have an unused edge e_2 which cannot be connected to any other vertex in L as this would create a cycle such as $z \to e_2 \to arbitrary\ vertices\ in\ L\ \to e_1 \to z$. If e_2 is connected to some vertex outside L, it will contradict our assumption of L being the longest path. Hence, graph G must contain a cycle if the degree of each vertex is greater than 2.

(b) any undirected graph can be bound using clauses described below $\quad 1 \to$ The maximum number of edges in any undirected graph should be $\leq \frac{n\ (n-1)}{2}$ where n is the total number of nodes. So, for the given example with 10 *nodes*, there can be a maximum of 45 edges while it has $\frac{\sum_{i=1}^{10} e_i}{2} \approx 18$ edges and hence the given graph nodes are good candidates for undirected graph.
$2 \to$ Each node can have a maximum of $(n-1)$ degree, so the degree of each vertex for the given 10 *nodes* can go up to 9. examining the given degrees i.e. 2, 3, 4, 4, 7, 1, 4, 5, 3, 2, it can be verified that none of the vertices's has degree ≥ 9.

hence we can conclude that the given nodes/degree data can be used to construct a undirected graph.

(c) : to prove this algorithm we will try to prove the following-

claim: If G is bipartite, it has no cycles of odd length

Proof. : We will prove this claim by contradiction. Let's assume that G is bipartite graph and has a cycle of odd length. call the odd cycle O_c, and let the vertices's in it be $(v_1, ..., v_n)$ where n is odd as we are looking out for odd length cycles. Since G is assumed to be bipartite, we can split the vertices's into two group sets X, Y such that there are no edges within same sets. Since the graph is bipartite and it is strongly connected, every consecutive vertices's from $(v_1, v_2, v_3, ..., v_n)$ starting from v_1 will go into different sets, i.e., no two consecutive vertices's v_i (where $i \leftarrow 1..n$) lie in the same set. To analyze the odd length cycles, we need to see if the n^{th}

vertex lies in the same set as v_1 or not. if it lies in the same set as v_1, then we have a cycle of odd length and hence the graph G cannot be bipartite, else a cycle of even length signifies otherwise.

Algorithm: :

A modified Breadth First Search algorithm coloring each node a different color from its parent will help us realise our approach.

isGraphBipartite(G):

start at any node arbitrary node S, and color it blue (opposite color will be red), put S in the queue Q.

while Q not empty

$currentNode = \text{pop}(Q)$

for every neighbor n of $currentNode$

if: n is not colored, then

color it the opposite color of $currentNode$ and put it in Q

if: n is colored and is the same color as $currentNode$, then

halt and output NOT BIPARTITE

else:

output G IS BIPARTITE

Correctness: : In case of a bipartite graph G, the start vertex is either in X or Y, which we will call red or blue in the following discussion. All of the neighbors of S must be the opposite color of $CurrentNode$ and so on. Let G be a graph that is not bipartite, then no matter how we partition the vertices into two groups, there will always be an edge within one of the groups. In particular, from the coloring of our algorithm, either the red group or the blue group must have an edge that points to two of the same colors, which is exactly what the algorithm tests for.

Running time: : The running time of this algorithm is the same as BFS which is $O(|V| + |E|)$ since the we have $O(1)$ work $|V|$ times with putting things in and out of the queue Q. We also have $O(1)$ work for every color checking of the neighbors which happens $2 * edges$ times (once for each side of the edge).

5: Weary traveler

From the problem, we are to select a flight so as to come up with the shortest arrival time from Source airport (A) to the destination airport (B). with a given parameter - n (total number of airports), each of which can be represented as a vertex in a directed graph. The edges of the graph are flights and will include 2 weights i.e. the $DepartureTime$ and $ArrivalTime$. There are two constraints in the problem as below : a. The problem requires us to select a flight F such that there is at least $10 minutes$ delay between the arrival and departure against the next connecting flight. b. We also want to select a flight(next flight) which has the minimum arrival time.

We can approach this problem with a modified Dijkstra's Algorithm to include two priority queues, One for finding the shortest time to reach each of the airport and another to select the flight with the shortest travel time.

Parameters :

G => The Directed graph of airports V and flights E, where V - vertices's and E - edges.

Edges include the D_t and A_t of the flight as weights, where D_t - departure time and A_t - arrival time.

A => Source Airport

B => Destination Airport

S_{at} => Shortest Arrival time at a airport

Algorithm: :

$SmallestTravelTime(G, A, startTime)$

Initialize the arrival Time at source $S_{at}[A]$ to startTime

For all the remaining V, set the arrival time as ∞

Initialize a priority queue - Q, based on the S_{at} at each vertex

while Q is not empty

do

remove the vertex V with the minimum A_t from Q

for each vertex i which is adjacent to vertex V

do

Initialize a priority queue, Q_w with flights F, according to their A_t at w,

having D_t at $V \geq S_{at}[V] + 10\ mins$

Initialize the flight time t to ∞

Select the minimum flight time from Q_w queue (if not empty) and assign it to t

if $t < S_{at}[w]$ then

update $S_{at}[w]$ with t and w in Q

return S_{at}

Running time:

worst case is when all n airports are interconnected. The time complexity of building the initial priority queue Q using heap implementation is $O(n)$. For each of the vertices's V, there can be $(n-1)$ connections to other adjacent vertices's w, making a new flight priority queue i.e Q_w. So, the total number of flight priority queues made is $n\ times * (n-1)$. Therefore, the time complexity of building a priority queue using heap implementation is $O(n^2)$. hence, total time complexity of the entire system is $O(n^2 \log n)$ as each queue-pop from queue Q requires $(\log n)$ time.