1: Easy relatives of 3-SAT

(a) The clause $(\overline{x_1} \implies x_2)$ means that whenever $\overline{x_1}$ is true, x_2 has to be true and when $\overline{x_1}$ is false then x_2 can have any value. The truth table for this implication will be

$$\begin{aligned}
&\overline{x_1} = 0, x_2 = 0, y = 1 \qquad (x_1 = 1)\\
&\overline{x_1} = 0, x_2 = 1, y = 1 \qquad (x_1 = 1)\\
&\overline{x_1} = 1, x_2 = 0, y = 0 \qquad (x_1 = 0)\\
&\overline{x_1} = 1, x_2 = 1, y = 1 \qquad (x_1 = 0)
\end{aligned}$$

where y is the output of this expression. For the four possible values of $\overline{x_1}$ in the table, the value of x_1 is 1,1,0, and 0. This table can also be inferred as when either of x_1 or x_2 is true, the output is also true. Therefore this expression can also be written as $(x_1 \vee x_2)$.

(b) The requirement of the path in the implication graph given in the question basically means if there is a cycle from x_i to x_i that contains $\overline{x_i}$.

A 2-SAT formula is unsatisfiable if some clause $(x_i \vee x_j)$ is forced to take the values $x_i = 0$ and $x_j = 0$. That's the only way the formula is unsatisfiable. This means that somewhere in the conjunction there are variables $\overline{x_i}$ and $\overline{x_j}$. These literals can be written as the following clauses,

$$\begin{aligned}
(\overline{x_i} \vee \overline{x_i}) &\equiv (x_i \Rightarrow \overline{x_i})\\
(\overline{x_j} \vee \overline{x_j}) &\equiv (x_j \Rightarrow \overline{x_j})
\end{aligned}$$

Additionally the clause $(x_i \vee x_j)$ can be written as the conjunction $(\overline{x_i} \Rightarrow x_j) \wedge (\overline{x_j} \Rightarrow x_i)$. In the implication graph consider the edges, $x_i \to \overline{x_i}$, $\overline{x_i} \to x_j$, $x_j \to \overline{x_j}$, and $\overline{x_j} \to x_i$. This forms a cycle from x_i to x_i that contains $\overline{x_i}$. Hence we can say that if a 2-SAT formula is unsatisfiable then its implication graph will have the mentioned path.

Proving the other way round. Let's say that there is a cycle in the implication graph that starts from x_1 and ends in x_1 and contains $\overline{x_1}$. The clasues for this cycle will look like,

$$(x_1 \Rightarrow x_2) \wedge (x_2 \Rightarrow x_3) \wedge \cdots \wedge (x_i \Rightarrow \overline{x_1}) \wedge (\overline{x_1} \Rightarrow x_{i+1}) \wedge \cdots \wedge (x_n \Rightarrow x_1)$$

Consider the argument that $(x_i \Rightarrow x_j) \wedge (x_j \Rightarrow x_k)$ can be written as $(x_i \Rightarrow x_k)$. This is beacuse, $(x_i \Rightarrow x_j) \wedge (x_j \Rightarrow x_k) \equiv (\overline{x_i} \vee x_j) \wedge (\overline{x_j} \vee x_k) \equiv (\overline{x_i} \vee x_k) \equiv (x_i \Rightarrow x_k)$ (Resolution proof).

Therefore the implication cycle can be reduced to $(x_1 \Rightarrow \overline{x_1}) \wedge (\overline{x_1} \Rightarrow x_1)$, which is equivalent to $(\overline{x_1} \vee \overline{x_1}) \wedge (x_1 \vee x_1) \equiv \overline{x_1} \wedge x_1 \equiv 0$. Hence we show that if the implication graph contains such a cycle, then the formula is unsatisfiable.

Consequently, we can say that 2-SAT has a polynomial time algorithm. As we first need to build an implication graph rewriting the clasuses as implications. Then we can run a BFS search algorithm to find such a cycle.

(c) Let's say there is a 3-SAT clause, $(x_1 \vee x_2 \vee \neg x_3)$. Let's break this clause into three 2-SAT clauses as follows,

$$(x_1 \vee x_2) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_3 \vee x_1)$$

This expression is only satisfied when atleast two of the three literals $x_1, x_2, \neg x_3$ are true. If none or only one of them is true then there is atleast one clause which is not satisfied and results in whole expression being 0. Therefore, this breaking up of the initial 3-SAT clause accounts for the condition of $2 - or - more$ 3-SAT.

Given a 3-SAT formula we can break each clause into three 2-SAT clauses (by taking disjunction of pairwise literals). If the 3-SAT problem has m clauses, then there will be atmost $3m$ such 2-SAT clauses and this conversion is polynomial in the size of 3-SAT formula. We can then run the polynomial time algorithm discussed in the previous part to check for satisfiablity of the newly created 2-SAT formula.

2: Decision vs Search

Algorithm: : Algorithm 1 presents an algorithm to find an independent set (returned as I) of size k in a graph G. The $Orc()$ is the oracle procedure that, when given inputs, graph G and value k, returns Yes if that graph contains an independent set of size k otherwise No. We first perform a check if the graph contains an independent set of size k. If there is no such set we immediately return the empty set I. The variable v can be used to store nodes (initially it contains no node). The procedures $remove(v)$, $get\_a\_node(G)$, and $remove\_N(v)$ in the while loop, remove the node v from the graph, get some random node from the graph, and remove the node v and its neighbours respectively.

Algorithm 1 Independet Set

```
1:  procedure search-IndSet(G, k)
2:      I = φ
3:      chk = Orc(G, k)
4:      if chk == No then
5:          return I
6:      end if
7:      v = φ
8:      while k > 0 do
9:          G = G.remove(v)
10:         v = get_a_node(G)
11:         G' = G.remove_N(v)
12:         chk = Orc(G', k − 1)
13:         if chk == Yes then
14:             I = I ∪ v
15:             G = G'
16:             k = k − 1
17:         end if
18:     end while
19:     return I
20: end procedure
```

For the first iteration, the first step of while loop basically does nothing. The second step get some node from G. We then remove v and its neighbours from the graph and check if the new graph G' has an independent set of size $k - 1$. We remove v and its neighbours from G because if

v is part of the independent set, then its neighbours can't be. If the $Orc(G', k-1)$ returns No, then node v is not part of the independent set but its neighbours can be. Therefore, in the next iteration we just remove v from the graph (first step of the while loop). However, if the $Orc(G', k-1)$ returns Yes, then it means that v can be part of the independent set and add it to I. Also now we need to look for an independent set of size $k-1$ in G'. Therefore, we update the necessary variables in the If block.

The loop ends when $k = 0$ and the required independent set is contained in I that we return.

Correctness: The initial check for the independent set of size k guarantees that the graph contains the required independent set. The basis of the algorithm is that a node is in the independent set if by removing the node and its neighbours from the graph, we can still get an independent set of size $k-1$ from the residual graph. We only include such nodes in the I and then reduce k. If we the residual graph doesn't contain independent size of $k-1$, then it means that one or more of its neighbours must be in the independent set and we remove that node only. So we're always reducing the graph by one or more nodes in every iteration.

Running time: The loop will sometimes run for same value of k and sometimes for different value of k depending on whether we enter the If block. Since we know that loop terminates, it must run k times for different values of k. It will atmost run $n-k$ times (n is the total number of nodes) for same values of k in the worst case. Therefore, the loop will run atmost $k+n-k = n$ times. In each iteration we are selecting some node and removing some node and its neighbours which can be done polynomial time. Also the number of calls to oracle are upper bounded by n.

3: Reductions, reductions

(a) Reducing a NP-hard problem to ILP will be sufficient to say that ILP is NP-hard. We will try to reduce the problem of SAT to ILP. Let's understand it with the following SAT problem.

$$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee x_4)$$

For this problem to be satisfiable, both the clauses need to be True. Consider the following interpretation of the above problem,

$$\begin{aligned}
&x_1 + 1 - x_2 + x_3 \geq 1,\\
&x_2 + x_4 \geq 1,\\
&x_1 \geq 0, x_1 \leq 1,\\
&x_2 \geq 0, x_2 \leq 1,\\
&x_3 \geq 0, x_3 \leq 1,\\
&x_4 \geq 0, x_4 \leq 1
\end{aligned}$$

here the variables, $x_1 \ldots x_4$ are can have integer values. The last 8 inequalities restrict these variables from having any value other than 0 and 1. The first two inequalities can only be satisfied if the respective clauses in the Boolean formula are satisfied. For example, the Boolean formula can be satisfied with the assignment, $x_1 = 1, x_2 = 0, x_3 = 0, x_4 = 1$. The $L.H.S.$ of the first inequality becomes 2 (≥ 1) and the second becomes 1 (≥ 1). One of the unsatisfying assignment for the Boolean formula is $x_1 = 0, x_2 = 0, x_3 = 0, x_4 = 0$. In this the $L.H.S.$ of the first inequality is although 1 (≥ 1) but the second one is 0 which is less than $R.H.S.$

Next let's say we have a SAT problem with m clauses and n variables. A conversion of this problem to the set of inequalities in the above manner will result in $m + 2n$ inequalities (one for

each clause and two for each variable). We now have an ILP problem at our hands, a solution for which is a solution for the initial SAT problem. Therefore, we have shown a reduction from SAT to ILP. Hence ILP is NP-hard.

(b) If we only consider solving for the variables given these inequalities (as above), I think the problem is NP-complete. Consider that, if we are given a solution for the variables, then I think we can verify the solution in polynomial time making the problem NP. Being NP-hard and NP, the problem becomes NP-complete. But if we need to maxmize or minimize some expression given these inequalities (as is generally the case with ILP problems), then verifying the solution does not seem to be polynomial and the problem is not NP-complete.

(c) Consider the following LINEQ (mod 2) ,

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m \end{aligned}$$

Adding all these m equations,

$$\begin{aligned} (a_{11} + a_{21} + \cdots a_{m1}+)x_1 + (a_{12} + a_{22} + \cdots + a_{m2})x_2 + \ldots \\ \cdots + (a_{1n} + a_{2n} + \cdots + a_{mn})x_n = (b_1 + b_2 + \cdots + b_m) \end{aligned}$$

Let the n coeffecients for the n variables be $c_1, c_2, \ldots c_n$ and the sum on the *R.H.S.* of the equality be b. Therefore, the above sum now becomes,

$$c_1x_1 + c_2x_2 + \cdots + c_nx_n = b$$

Since the problem is modulo 2, we know that $c_1, \ldots, c_n, b \in \{0, 1\}$. Consider the following cases. If all the c_i are 0 and $b \neq 0$ then we can say that the problem is infeasible. If some $c_i \neq 0$ and $b = 0$ then assign the corresponding $x_i = 0$ and remaining x_j can be assigned anything. If some $c_i \neq 0$ and $b = 1$ then assign any one of the x_i to be one and the remaining to be zero.

Thie overall process involves adding all the equations and then trivially finding an assignment to $x_1 \ldots x_n$ by examining the value of b. I think this can be carried out in polynomial time.

(d) Before starting the solution first consider the following observation that we will use.

We are given that $x_i \in \{0, 1\}$, therefore, increasing the exponent of a variable does not make any difference on the solution of the equalities. For example, the equation $x_1 + x_2x_3 = 1$ has same solutions as $x_1^2 + x_2x_3 = 1$.

Now let's move on to the solution. If we are given a solution to this problem, then it can be verified in polynomial time as for each equation we need to plug in the value and compare it to the *R.H.S.* Therefore, the problem is in NP.

To prove that the problem is NP-hard, we will reduce 3-SAT to it. Consider the following 3-SAT clause,

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

This clause is satisfiable iff the following equation is satisfiable,

$$x_1 + x_2 + x_3 + x_1x_2 + x_2x_3 + x_3x_1 + x_1x_2x_3 = 1$$

where, $x_1, x_2, x_3 \in \{0, 1\}$ and the addition is modulo 2. If the SAT clause had a negated variable x_i, then it can be replaced with $1 - x_i$ or $1 + x_i$ ($-1 = 1 \pmod 2$). As an example,

$$(x_1 \vee \neg x_2 \vee x_3)$$

This clause is satisfiable iff the following equation is satisfiable,

$$x_1 + (1 + x_2) + x_3 + x_1(1 + x_2) + (1 + x_2)x_3 + x_3x_1 + x_1(1 + x_2)x_3 = 1$$

This conversion of the 3-SAT clause results in a equation that has linear, quadratic and cubic terms. This equation can be easily made "cubic" by raising the exponent of linear terms by 2, raising the exponent of anyone of the variable in a quadratic term by 1 and leaving the cubic term as it is, e.g. for the 3-SAT clause $(x_1 \vee x_2 \vee x_3)$

$$x_1^3 + x_2^3 + x_3^3 + x_1^2x_2 + x_2^2x_3 + x_3^2x_1 + x_1x_2x_3 = 1,$$

In this way we can say that 3-SAT reduces to the "cubic" version of the problem and hence it is NP-hard.

Inorder to write the above equation as quadratic the problem is with the "cubic" term. Let's say that the term is $x_ix_jx_k$. We can write this as x_ix_t (introduce a new variable) and include one more equation in addition to the one we get by converting the 3-SAT clause,

$$\begin{aligned} x_t &= x_jx_k \pmod 2 \\ x_t - x_jx_k &= 0 \pmod 2 \\ x_t + x_jx_k &= 0 \pmod 2 \\ x_t^2 + x_jx_k &= 0 \pmod 2 \end{aligned}$$

Therefore, the 3-SAT clause $(x_1 \vee x_2 \vee x_3)$ is satisfiable iff the following two equations have a solution,

$$\begin{aligned} x_1^2 + x_2^2 + x_3^2 + x_1x_2 + x_2x_3 + x_3x_1 + x_1x_{23} &= 1 \pmod 2 \\ x_{23}^2 + x_2x_3 &= 0 \pmod 2 \end{aligned}$$

where x_{23} is the newly introduced variable.

Therefore, if we a 3-SAT problem with m clauses, then it can be reduced to a set of atmost $2m$ quadratic equation with atmost m new variables. If we can solve this set of equations then we have a solution to the 3-SAT problem. Therefore, the quadratic problem is NP-hard. Since it lies both in NP and NP-hard, it is NP-complete.

4: Graphs−definitions

(a) All vertices have degree ≥ 2. Let's say there are n vertices. Therefore, the sum of all degress, $\sum deg \geq 2n$. Now because $\sum deg = 2|E|$, therefore, $|E| \geq n$.

I will now show that if a graph has $|E| \geq n$, then it has a cycle. We will try to build a cycle free graph with n edges. First let's build a cycle free graph with $n-1$ edges.

Let's consider two sets **Used** and **Unused**. Let's keep all the nodes in **Unused** initially. Pick two nodes from this set and make an edge between them. Remove these two nodes from **Unused** and keep them in **Used**. Next take a new node from **Unused** and make an edge between this node and any of the previous node from the graph and keep this in **Used** and remove it from **Unused**. The nodes in the set **Used** are connected to each other. As long as for each new edge we take a new node from **Unused** there will not a cycle as we are always expanding our graph and never coming back to a previously used node. Therefore, after running this process for $n-1$ edges we will have a connected graph with no cycles. We can always find a path between any two nodes.

Now let's try to insert the n^{th} edge. Since the initial graph was already connected, a new edge will create a new path between the two nodes that it now connects. So we will have two different paths between two nodes and these two paths will result in a cycle. Therefore, a graph can't be constructed with n edges and no cycle. In other words, if the number of edges $\geq n$, then the graph will have a cycle.

(b) The sum of the ten degrees is, $\sum deg = 2+3+4+4+7+1+4+5+3+2 = 35$. Now $\sum deg = 2|E|$, therefore for a valid graph the sum of degree must be an even number as each edge contributes to 2 degrees of the connecting nodes. Therefore, this graph with $\sum deg = 35$ cannot exist.

(c) Algorithm: The algorithm for finding odd-length cycles in a directed graph is given in

Algorithm 2. First we check for self loops and if we find one we return that because that's a cycle of length 1.

Next we perform BFS search by selecting some root r. During the BFS, we maintain a parity attribute of each node depending on the length of the path from root to this node. If the length is even then parity is even and vice versa. Also $parentBFS$ is named so because we also perform DFS later in the procedure. So there can be $parentBFS$ and $parentDFS$ for each node.

If in the loop we find an edge between two nodes having the same parity we break from both the loops (the else part in the for loop). The nodes corresponding to this edge are U and V connected as $U \to V$. This break statement is only executed if there is an odd-length cycle in the graph otherwise the BFS process will run to completion.

In the next part of the procedure, we check if there was a break encountered in the previous loop. If it wasn't, then we return No Odd Cycle. But if there was, we first perform a DFS search from V to r and store the length of this path in d.

If *V.parity* is even, it means the path from r to V ($r \rightsquigarrow V$) is even length. Now if d is odd, we can just return the path $r \rightsquigarrow V \rightsquigarrow r$ as an odd-length cycle, otherwise $r \rightsquigarrow U \to V \rightsquigarrow r$ becomes the odd-length cycle.

If *V.parity* is odd, then the above two cases are just switched.

Algorithm 2 Odd-Cycle

```
1: procedure odd_cycle(G)
2:     r = G.select_root()
3:     Check for self loops in G; If found return that loop as odd cycle
4:     for each node v, v.distance = ∞, v.parentBFS = NIL
5:     r.distance = 0; r.parity = 0
6:     Q.enqueue(r)
7:     while Q is not φ do
8:         curr = Q.dequeue()
9:         for each node n adjacent to curr do
10:            if n.distance == ∞ then
11:                n.distance = curr.distance + 1
12:                n.parentBFS = curr
13:                n.parity = ¬curr.parity
14:            else
15:                if n.parity == curr.parity then
16:                    V = n
17:                    U = curr
18:                    break from both the loops
19:                end if
20:            end if
21:        end for
22:    end while
23:    if break from the above loops then
24:        Perform DFS from V to r; Let the DFS distance from V to r be d
25:        if V.parity == 0 then
26:            if d is odd then
27:                return the path r ⇝ V ⇝ r
28:            else
29:                return the path r ⇝ U → V ⇝ r
30:            end if
31:        else
32:            if d is odd then
33:                return the path r ⇝ U → V ⇝ r
34:            else
35:                return the path r ⇝ V ⇝ r
36:            end if
37:        end if
38:    else
39:        return No Odd Cycle
40:    end if
41: end procedure
```

Correctness: The premise for the algorithm is that if there is no edge between two nodes with same parity, then there can't be an odd-cycle length. Let's consider the undirected version of our directed graph. We just ignore the directional information of the edges and have an undirected

graph. A undirected graph is bipartitie if and only if there is no odd-length cycle in it. If the undirected graph is not bipartite, then it means that its directed version is also not bipartite. As a result, if the undirected graph can't have an odd cycle, the directed version can't have that too (as the directed graph can have atmost same cycles as the undirected graph and not any extra cycles). We perform a bipartiteness check in our algorithm (else part in while loop). If the check succeeds, then it means that directed and the undirected version are bipartite, and because of the above reasoning the directed graph doesn't have an odd cycle.

If there is an odd-length cycle (bipartiteness test fails) in the undirected version then that cycle may or may not exist when we add directional information. There might be some node with the two edges pointing in it or pointing out. Let's say while traversing the undirected version of this cycle from left to right, there is a node in the directed version like $u \to t \leftarrow v$. The edge $t \leftarrow v$ is the problem. But since our graph is strongly connected, there will always a path from $t \rightsquigarrow v$. If this path is even length, then the cycle $t \rightsquigarrow v \to t$ is an odd-length cycle. On the other hand if this path is odd length, then the cycle $t \rightsquigarrow v \rightsquigarrow u \to t$ is odd length. This tells us that if the bipartiteness test is failed we're guaranteed to have an odd-length cycle in the directed, strongly connected graph, we just need to find it; the procedure of finding it is described in the algorithm.

Running time: To check for self loops, we need to iterate over all the edges once, therefore this is bounded by $|E|$. The while loop is just BFS search with an additional else part which will only execute if the search is about to stop. Therefore, this search is bounded by $|V| + |E|$. If we break from the loop, we perform a DFS which is again bounded by $|V| + |E|$. Next only one of the four possible cases can be true . For each case we need to find the appropriate path. This is trivial as all the necessary search have already been done, we just need to traverse along $parentDFS$ and $parentBFS$ and find the path. This is bounded by $|V|$.

Therefore, the algorithm is bounded by $O(|V| + |E|)$.

5: Weary Traveler

Algorithm: The algorihtm for minimum travel time is shown in Algorithm 3. The algorithm is basically built upon Dijkstra's algorithm with one modification. Let's say we have n airports and m flights connecting them. This can be construed as a directed graph, G, with n vertices and m edges. Notice that any pair of vertices can have mutiple edges in either direction (there can be multiple flights between a pair of airports). If there are k flights between two airports u and v then the departure times from u are denoted as $u_{1d}, u_{2d}, \ldots, u_{kd}$ and flight times as $f_1, f_2, \ldots, f_k$. Also for the algorithm, time is assumed to be in minutes and is an increasing quantity without being divided into hours, days or any higher unit. $time[u]$ indicates the arrival time at the airport u and $prev[u]$ indicates the airport from which you arrived at u.

The algorithm gives the optimum schedule to fly to any other $n-1$ airports given a source aiport src. The time at which the person arrives at the source airport is stored in $time[src]$ (this time is assumed to finite). Initially, for every other airport we set $time[v]$ to be ∞ and $prev[v]$ to be $undefined$. We add all nodes in the priority queue Q. Next we run a while loop until Q is empty. For each iteration, we assign u the element in Q with smallest $time[]$ and remove that element from Q. Then for each neighbour v of u we find out the best possible flight from u and this time duration is stored in $duration$. The inner min statement finds the flight that provides minimum of sum of waiting time ($u_{id} - time[u]$) and the flight time (f_i) subject to the constraint that

$u_{id} - time[u] \geq 10$. If none of the flight is possible to catch from $u \to v$ then the inner min is undefined and we assign $duration = \infty$.

Algorithm 3 Minimum Travel Time

```
1: procedure min_travel_time(G, src)
2:     time[src] = Arrival Time at the src
3:     Initialize priority queue Q
4:     for each vertex v do
5:         if v ≠ src then
6:             time[v] ← ∞
7:             prev[v] ← undefined
8:         end if
9:         Q.add(v, time[v])
10:    end for
11:    while Q is not empty do
12:        u ← Q.extract_min
13:        for each neighbour v of u do
14:            duration = min{∞, min_{i=1...k} {u_id − time[u] + f_i | u_id − time[u] ≥ 10}}
15:            tmp = time[u] + duration
16:            if tmp < time[v] then
17:                time[v] ← tmp
18:                prev[v] ← u
19:                Q.update(v, tmp)
20:            end if
21:        end for
22:    end while
23:    return time[], prev[]
24: end procedure
```

We check if by taking this flight we get a reduction in the arrival time at v, *i.e.* $time[v]$. If there is an improvement we update this information. The algorithm returns $time[]$, which gives the arrival times of $n-1$ airports from src, and $prev[]$ where $prev[v]$ tells us the airport from which we should fly to v.

Correctness: The algorithm is basically Dijkstra's algorithm where we want to reduce the quantity $time[v]$ for every v except src. The main difference is that there can be multiple edges connecting two nodes u and v. We need to pick the best possible of these edges (flights) and then see if it provides any improvement over the previously designated path to reach v. The selection of the best possible flight accounts for the wait time and flight time as both them are a part of the total traveling time. Therefore, the value of $time[]$ that we get as the output is the optimal value given the src.

Running time: The priority queue, Q, in the algorithm needs $\log n$ time for $extract\_min$ and $update$ operations. The while loop is executed exactly n times as each node is assigned to u exactly once. For each u there can be atmost $n-1$ neighbours. Therefore, for each u the for loop will be executed atmost $n-1$ times. Also for each neighbour we perform a min operation (scan all the outdegree of u) on the outdegree of u. This will sum upto order of m. The two loops will

result in order of n^2 as for each node there are $n-1$ possible neighbours. For each iteration we will a $\log n$ term for the priority queue. Not that $\log n$ is not needed to be multiplied with m term as the *update* operation will be performed on just one of the paths and not all of them. Therefore, the overall complexity becomes $O(n^2 \log n + m)$