1: Easy Relatives of 3-SAT

(a) Using a truth-table, we can prove that $x_1 \vee x_2$ and $\overline{x_1} \Rightarrow x_2$ are logically equivalent.

x_1	x_2	$\overline{x_1}$	$x_1 \vee x_2$	$\overline{x_1} \Rightarrow x_2$
0	0	1	0	0
0	1	1	1	1
1	0	0	1	1
1	1	0	1	1

$\overline{x_1} \Rightarrow x_2$ can be restated as "if $\overline{x_1}$ is true, then x_2 must be true". On the first row of the truth table, we see a contradiction when $\overline{x_1}$ is true but x_2 is false. In all other cases the implication holds.

(b)

(c)

2: Decision vs. Search

Algorithm:

Setup: Create a set C and populate it with all $v \in V$ in G and a set S that is initially empty.

Step 1: If $\mathcal{O}(G, k)$ reports NO, report that an independent subset of size k does not exist in G. Otherwise continue.

Step 2: Randomly select and **remove** a vertex v from C. Remove v from G to form G'. Keep some state so we can add v back to G' to restore G.

Step 3: Call $\mathcal{O}(G', k)$ and if the oracle answers NO add v to S and add v back to G' to restore G. If the oracle responds with YES, replace G with G' (this vertex is discarded).

Step 4: If $|S| < k$ repeat from step 2. If $|S|$ equals k, S is an independent set of size k and we are done.

Correctness: From step 1, we know initially if G contains an independent subset of size k. If a vertex v can be removed from G to form G' and $\mathcal{O}(G', k)$ returns YES, we know G contains an independent subset of size k and v is not part of that subset. If $\mathcal{O}(G', k)$ returns NO, G no longer contains an independent subset so we know v is part of a independent subset so v is added to S.

Running time: In the worst-case, each $v \in V$ in G must be removed from G (one at a time) and a call to $\mathcal{O}$ is made. Assuming the representation of the graph is an adjacency matrix, a node can be removed or added in $O(|V|)$ time. Giving $O(|V|)$ calls to $\mathcal{O}$ and $O(|V|^2)$ work to remove vertices.

3: Reductions, Reductions

(a) To prove the Integer Linear Programming (ILP) problem is NP-hard, we can provide a polynomial time reduction of 3-SAT (a known NP-complete problem) to ILP. 3-SAT is the problem of given a formula of clauses in conjunctive normal form (CNF) where each clause contains at most three literals.

Reduction: Let the variables in the 3-SAT problem be y_1, y_2, y_3, ..., y_n. There will be identical variables x_1, x_2, x_3, ..., x_n in our LIP problem restricted to the values $\{0, 1\}$ where 0 represents false and 1 represents true. Each linear constraint will contain the constants a_1, a_2, a_3, ..., a_n and b.

For each clause in the 3-SAT problem construct a linear constraint. If y_i is present then a_i is set to -1 otherwise a_i is set to 0. The constant b is set to -1. If y_i is negated in the clause, then x_i is set to $(1 - y_i)$ otherwise x_1 is set to y_i.

The formula is satisfiable if there exists integers x_i that satisfy all the linear constraints.

(b) The ILP problem is NP-Complete if it is in NP and is NP-Hard. By reducing a known NP-Complete problem (3-SAT) to ILP we showed ILP is NP-Hard. ILP is in NP because given integers x_i we can verify in polynomial time if they satisfy all the constraints. Thus ILP is NP-Complete.

(c)

(d)

4: Graphs - Definitions

(a)

(b) There cannot exist an undirected graph consisting of 10 nodes with degrees 2, 3, 4, 4, 7, 1, 4, 5, 3, 2 respectively.

Proof: The *degree sum formula* states, given an undirected graph G, the sum of the degrees of all the vertices V in G is equal to twice the number of edges E in G. Where the degree of a vertex is the number of edge incident to the vertex.

$$\sum_{v \in V} deg(v) = 2|E|$$

Each edge in an undirected graph connects exactly two vertices. In the case of a looping edge that connects a vertex to itself, this adds two to the degree of that vertex.

We consider the sum of the given vertices.

$$\sum_{v \in V} deg(v) = 2|E|$$

$$35 = 2|E|$$

$$|E| = 17.5$$

The number of edges must be an integer value. Thus we know that the given graph does not satisfy the *degree sum formula* and is an invalid graph.

(c)

5: Weary Traveler

Algorithm: A directed graph G is constructed where airports are represented by vertices and flights are represented by edges. Each vertex contains the name of an airport. Edges contain the arrival time of the flight, departure time of the flight, and a flight id that uniquely identifies the flight (used to construct flight schedule when the shortest travel path is found).

Four dictionaries are used to keep track of the following: the previous vertex in a path, the minimum travel time from the source to a given vertex, what time you will arrive at a given vertex to achieve the minimum travel time to that vertex from the source, and an id of the flight used to arrive at the a given vertex.

The helper method *arrival(u, w)* returns the arrival time of the edge that connects u and w, *depart(u, w)* returns the departure time, and *id(u, w)* returns the flight id.

Algorithm 1 Minimum Total Travel Time

```
 1: Initialize
 2:     travel_time[v] = $\infty$ For all $v \in V$ in G
 3:     prev[v] = null For all $v \in V$ in G
 4:     arrival_time[v] = $-\infty$ For all $v \in V$ in G
 5:     flight_id[v] = null For all $v \in V$ in G
 6:
 7:     travel_time[source] = 0
 8:     arrival_time[source] = time arrived at source airport
 9:     PriorityQueue q = [source]
10: end Initialize
11:
12: procedure MINIMUM_TOTAL_TRAVEL_TIME(G, SOURCE, DEST)
13:     while q is non-empty do
14:         u = q.peek() # Get the top priority item but don't remove it
15:         if u == dest then
16:             return schedule with smallest total travel time
17:         end if
18:         for neighbors w in u do # Consider each vertex with an edge from u
19:             new_travel_time = arrival(u, w) - arrival_time[u] + travel_time[u]
20:             if arrival_time[u] + 10 $\leq$ depart(u, w) && new_dist < travel_time[w] then
21:                 travel_time[w] = new_travel_time
22:                 arrival_time[w] = arrival(u, w)
23:                 flight_id[w] = id(u, w)
24:
25:                 q.update(w) # Add w to the priority queue or update it's priority
26:             end if
27:         end for
28:         q.pop() # Remove from the queue the vertex with the highest priority
29:     end while
30:     return No path exists
31: end procedure
```

The algorithm is Dijkstra's algorithm with small modifications. On line 14, if we de-queue the destination vertex we are done and can construct the shortest path. This is accomplished using the *prev* and *flight_id* dictionaries starting at the destination vertex and working backwards to the source vertex. On line 19, the travel time to travel from u to w is calculated, combining flight time and time spent waiting in airports. In contrast to traditional weighted graphs, this is necessary since time will be spent traveling while inside airports (nodes) and while on flights (edges).

A priority queue implemented with a min-heap is used to store the vertices waiting to be processed so we can efficiently get the vertex with the highest priority. The travel time from the source to the vertex is stored in the priority queue with the vertex and is used to determine priority.

Correctness:

Running time: