Note: Your TA probably will not cover all the problems. This is totally fine, the discussion worksheets are not designed to be finished in an hour. They are deliberately made long so they can serve as a resource you can use to practice, reinforce, and build upon concepts discussed in lecture, readings, and the homework.

1 A Reduction Warm-up

In the Undirected Rudrata path problem (aka the Hamiltonian Path Problem), we are given a graph G with undirected edges as input and want to determine if there exists a path in G that uses every vertex exactly once.

In the Longest Path in a DAG, we are given a DAG, and a variable k as input and want to determine if there exists a path in the DAG that is of length k or more.

Is the following reduction correct? If so, provide a concise proof. If not, justify your answer and provide a counter example.

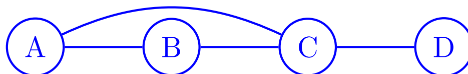
Undirected Rudrata Path can be reduced to Longest Path in a DAG. Given the undirected graph G, we will use DFS to find a traversal of G and assign directions to all the edges in G based on this traversal. In other words, the edges will point in the same direction they were traversed and back edges will be omitted, giving us a DAG. If the longest path in this DAG has $|V| - 1$ edges then there must be a Rudrata path in G since any simple path with $|V| - 1$ edges must visit every vertex, so if this is true, we can say there exists a Rudrata path in the original graph. Since running DFS takes polynomial time ($O(|V| + |E|)$), this reduction is valid.

Solution:

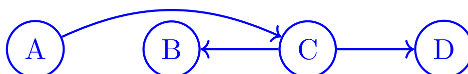
It is incorrect.

It is true that if the longest path in the DAG has length $|V| - 1$ then there is a Rudrata path in G. However, to prove a reduction correct, **you have to prove both directions**. That is, if you have reduced problem A to problem B by transforming instance I to instance I' then you should prove that I has a solution **if and only if** I' has a solution. In the above "reduction," one direction does not hold. Specifically, if G has a Rudrata path then the DAG that we produce does not necessarily have a path of length $|V| - 1$–it depends on how we choose directions for the edges.

For a concrete counterexample, consider the following graph:



It is possible that when traversing this graph by DFS, node C will be encountered before node B and thus the DAG produced will be



which does not have a path of length 3 even though the original graph did have a Rudrata path.

2 California Cycle

Prove that the following problem is NP-hard.

Input: A directed graph $G = (V, E)$ with each vertex colored blue or gold, i.e., $V = V_{\text{blue}} \cup V_{\text{gold}}$.

Goal: Find a *Californian cycle* which is a directed cycle through all vertices in G that alternates between blue and gold vertices.

Hint: Directed Rudrata Cycle.

Solution: We reduce Directed Rudrata Cycle to Californian Cycle, thus proving the NP-hardness of Californian Cycle. Given a directed graph $G = (V, E)$, we construct a new graph $G' = (V', E')$ as follows:

- V' contains two copies of the vertex set V. So, for each $v \in V$, there's a corresponding blue node v_b and a gold node v_g in V'.
- For each $v \in V$, add an edge from v_b to v_g in G'.
- For each $(u, v) \in E$, add edge (u_g, v_b) in G'.

  Another way to view this is that for each node $v \in V$, we are redirecting all its incoming nodes to v_g, and all its outgoing nodes originate from v_b (in G').

To see why this works correctly, observe that every cycle in G, will have a corresponding cycle in G' that alternates between blue and gold vertices. So if G contains a Rudrata cycle, then G' will contain a Californian cycle. Also, every edge in G' that goes from a blue node to a gold node will have both its end points correspond to the same vertex in G. So, G' has a Californian cycle, we can ignore all the blue-gold edges in that cycle and obtain a Rudrata cycle in G.

3 Cycle Cover

In the cycle cover problem, we have a directed graph G, and our goal is to find a set of directed cycles $C_1, C_2, \ldots C_k$ in G such that every vertex appears in exactly one cycle (a cycle cannot revisit vertices, e.g. $a \to b \to a \to c \to a$ is not a valid cycle, but $a \to b \to c \to a$ is), or declare none exists.

In the bipartite perfect matching problem, we have a undirected bipartite graph (a graph where the vertices can be split into L, R, and there are no edges between two vertices in L or two vertices in R), and our goal is to find a set of edges in this graph such that every vertex is adjacent to exactly one edge in the set, or declare none exists.

Give a reduction from cycle cover to bipartite perfect matching.

(Hint: In a cycle cover, every vertex has one incoming and one outgoing edge.)

Solution: Given the cycle cover instance G, we create a bipartite graph G' where L has one vertex v_L for every vertex in G, and R has one vertex v_R for every vertex in G. For an edge (u, v) in G, we add an edge (u_L, v_R) in the bipartite graph. We claim that G has a cycle cover if and only if G' has a perfect matching.

If G has a cycle cover, then the corresponding edges in the bipartite graph are a bipartite perfect matching: The cycle cover has exactly one edge entering each vertex so each v_R has exactly one edge adjacent to it, and the cycle cover has exactly one edge leaving each vertex, so each v_L has exactly one edge adjacent to it.

If G' has a perfect matching, then G has a cycle cover, which is formed by taking the edges in G corresponding to edges in G': If we have e.g. the edges $(a_L, b_R), (b_L, c_R), \dots, (z_L, a_R)$ in the perfect matching, we include the cycle $a \to b \to c \to \dots, z \to a$ in G in the cycle cover. Since v_L and v_R are both adjacent to some edge, every vertex will be included in the corresponding cycle cover.

If there exists a polynomial reduction from problem A to problem B, problem B is at least as hard as problem A. From this, we can define complexity class which sort of gauge 'hardness'.

Complexity Definitions

- NP: a problem in which a potential solution can be verified in polynomial time.
- P: a problem which can be solved in polynomial time.
- NP-Complete: a problem in NP which all problems in NP can reduce to.
- NP-Hard: any problem which is at least as hard as an NP-Complete problem.

Prove a problem is NP-Complete
To prove a problem is NP-Complete, you must prove the problem is in NP and it is in NP-Hard.

To prove that a problem **is in NP,** you must show there exists a polynomial verifier for it.

To prove that a problem **is NP hard,** you can reduce an NP-Complete problem to your problem.

4 NP or not NP, that is the question

For the following questions, circle the (unique) condition that would make the statement true.

(a) If B is NP-complete, then for any problem $A \in \mathsf{NP}$, there exists a polynomial-time reduction from A to B.

Always True True iff $\mathsf{P} = \mathsf{NP}$ True iff $\mathbf{P} \neq \mathbf{NP}$ Always False

Solution: Always True: this is the definition of NP-hard, and all NP-complete problems are NP-hard

(b) If B is in NP, then for any problem $A \in \mathbf{P}$, there exists a polynomial-time reduction from A to B.

Always True True iff $\mathsf{P} = \mathsf{NP}$ True iff $\mathsf{P} \neq \mathsf{NP}$ Always False

Solution: Always true: since we have polynomial time for our reduction, we have enough time to simply solve any instance of A during the reduction.

(c) 2 SAT is NP-complete.

Always True True iff $\mathsf{P} = \mathsf{NP}$ True iff $\mathsf{P} \neq \mathsf{NP}$ Always False

Solution: True iff $\mathsf{P} = \mathsf{NP}$:
By definition, in order to be NP-Complete a problem must be in NP, and there must exist a polynomial reduction from every problem in NP.
If $\mathsf{P} \neq \mathsf{NP}$, then there does not exist a polynomial time reduction from NP-Complete problems like 3-SAT to 2-SAT.
If $\mathsf{P} = \mathsf{NP}$, then a polynomial reduction is as follows:
since $\mathsf{P} = \mathsf{NP}$ there must exist a polynomial times algorithm to solve $3 - SAT$. Thus, when we are preprocessing 3-SAT we can solve for whether there exists a solution in the instance or not. If the instance has a solution, then we will map it to an instance of $2 - SAT$ that has a solution,

and if it doesn't have a solution, we will map it to an instance that doesn't have a solution. Thus all problems in NP will have a polynomial time reduction to 2-SAT as all problems in NP are reducible to 3-SAT.

(d) Minimum Spanning Tree is in NP.

Always True   True iff $\mathsf{P} = \mathsf{NP}$ True iff $\mathsf{P} \neq \mathsf{NP}$ Always False

Solution: Always True. MST is solvable in polynomial time, which means it is verifiable in polynomial time.
Note that explicitly, the decision problem would be "does there exist a spanning tree whose cost is less than a budget b?".

5 Runtime of NP

True or False (with brief justification): Suppose we can show for some fixed k, an NP-complete problem P has a time $O(n^k)$ algorithm. Then every problem in NP has a $O(n^k)$ time algorithm.

Solution: False. The reduction f_L from an arbitary problem $L \in \mathsf{NP}$ is guaranteed to run in time $O(n^{c_L})$ and produce a problem $f(x)$ of the NP-complete problem of size $O(n^{c'_L})$ for constants c_L and c'_L. However, these can be arbitrarily larger than k.