Solutions to Homework 7

(10 pts) 1. **Self assessment.**

Carefully read and understand the posted solutions to the **midterm exam**. Identify **one problem from the list below** for which your own solution has the most room for improvement (e.g., has unsound reasoning, doesn't show what was required, could be significantly clearer or better organized, etc.).

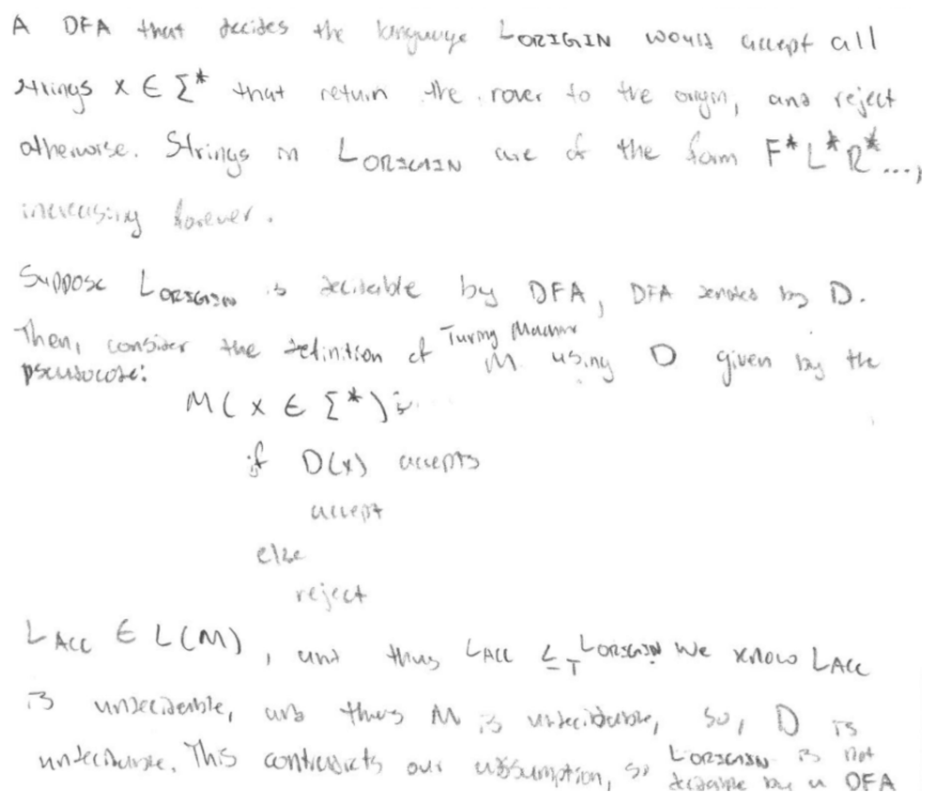
- Q10 Chaotic Potential
- Q13 Greedy Cat Photographing
- Q14 DFA Impossibility
- Q15 SpongeBob vs Squidward
- Q16 Machine Disagreement

Copy or screenshot your solution, then in a few sentences, explain what was deficient and how it could be fixed.

(Alternatively, if you think one of your solutions is significantly *better* than the posted one, copy it here and explain why you think it is better.)

If you didn't take the exam, review this solution by GPT 4o instead (pick one problem and explain the room for improvement).

Solution: My solution for Q14 of the midterm is as follows:



A DFA that decides the language L_{ORIGIN} would accept all strings $x \in \Sigma^*$ that return the rover to the origin, and reject otherwise. Strings in L_{ORIGIN} are of the form $F^*L^*R^*\ldots$, increasing forever.

Suppose L_{ORIGIN} is decidable by DFA, DFA denoted by D. Then, consider the definition of Turing Machine M using D given by the pseudocode:

```
M(x ∈ Σ*):
    if D(x) accepts
        accept
    else
        reject
```

$L_{ACC} \in L(M)$, and thus $L_{ACC} \leq_T L_{ORIGIN}$. We know L_{ACC} is undecidable, and thus M is undecidable, so, D is undecidable. This contradicts our assumption, so L_{ORIGIN} is not decidable by a DFA

My solution was fundamentally deficient because of the way that I approached the problem. Instead of using the definition of a DFA, and taking advantage of the fact that a DFA has a finite amount of states with the pigeonhole principle, I tried to define a contradictory turing machine that would accept the language, and then show that the language was turing rediculable to L_{ACC}. This was a convoluted approach, but I did have the right idea to contradict a supposed DFA that decides tha language. I could have fixed this by using the definition of a DFA, and showing that the language is not regular by the pigeonhole principle.

2. **Understanding P, NP, and NP-hardness.**

(4 pts) (a) Prove that all efficiently decidable languages are efficiently verifiable. In other words, show that every language in P has an efficient verifier.

Solution: Since every language in P has a polynomial-time decider D, construct a verifier V that, on input (x, y), simply ignores y and simulates D on x. If D accepts, then V accepts; otherwise, V rejects. The simulation runs in polynomial time, so all languages in P admit efficient verifiers.

(4 pts) (b) Prove that every language in NP is decidable. In other words, show that every language in NP has a decider.

Solution: Let $L \in \mathsf{NP}$. By definition, there is a nondeterministic polynomial-time Turing machine N deciding L. To decide L deterministically, on input x, systematically simulate every nondeterministic branch of N on x. Since each branch runs in polynomial time, this procedure halts on every branch and accepts exactly when $x \in L$. Thus, L is decidable.

(4 pts) (c) Prove or disprove: All NP-hard languages are decidable.
(A disproof would consist of an NP-hard language that is also undecidable.)

Solution: Disprove; define

$$L = \{ \varphi \mid \varphi \in \text{3SAT}\} \cup \{ \langle M \rangle \mid M \text{ halts on the empty input}\}.$$

1. Reduction: Given an instance φ of 3SAT, let $f(\varphi) = \varphi$.
2. Polynomial Time: This is the identity function; polynomial time.
3. Correctness:

- $(\Rightarrow)$ If φ is satisfiable, then $\varphi \in L$, so $f(\varphi) \in L$.
- $(\Leftarrow)$ If $f(\varphi) = \varphi \in L$, then φ must be a satisfiable 3SAT instance.

Hence 3SAT $\leq_p L$, so L is NP-hard. However, deciding L requires solving the Halting Problem, so L is not decidable. Therefore, not all NP-hard languages are decidable.

3. **Verifying ϒ's verifiers.**

We highly recommend you to read the Discussion of Completeness and Soundness and Discussion of Efficiency sections before attempting the following problem.

Recall that a *multiset* S is just like a set, but it can have duplicate elements. A submultiset $S^* \subseteq S$ also can have duplicate elements, as long as it does not have more copies of an particular element than S does.

Consider the following language

$$\textsc{SubsetSum} = \{(S, k) : S \text{ is a } \textit{multiset} \text{ of integers} \geq 0, \text{ and } \exists\, S' \subseteq S \text{ such that } \sum_{s \in S'} s = k\}.$$

(4 pts) (a) Professor ϒ claims that $\textsc{SubsetSum} \in \mathsf{NP}$, and presents the following verifier:

```
Input: A multiset of nonnegative integers $S = \{s_1, s_2, \ldots, s_n\}$, an integer $k$, and
    a multiset of nonnegative integers $C = \{c_1, c_2, \ldots, c_m\}$
1: function VerifierA($S, k, C$)
2:     if $m > n$ then reject // $C$ can't be a submultiset of $S$ in this case
3:     $sum \leftarrow 0$
4:     for $c \in C$ do // iterate over elements (integers) in $C$
5:         if $c \notin S$ then reject
6:         else $sum \leftarrow sum + c$
7:     if $sum = k$ then accept
8:     reject
```

Correctness: For correctness,

$$\begin{aligned}(S, k) \in \textsc{SubsetSum} &\iff \text{there exists a submultiset } C \subseteq S \text{ s.t. } \sum_{c \in C} c = k \\ &\iff \textsc{VerifierA}(S, k, C) \text{ accepts}\end{aligned}$$

Runtime: After ruling out the possibility that $m > n$ in line 2, the for loop will iterate over at most n elements, while other lines take constant time, the overall runtime is $O(n)$, which is polynomial with respect to the input size.

Explain why VerifierA does not show that $\textsc{SubsetSum} \in \mathsf{NP}$. You may use an example to support your reasoning. You do not have to fix the verifier, though it is a good exercise.

Solution: VerifierA does not ensure that C is a valid submultiset of S. Line 6 only checks whether each $c \in C$ appears in S, not whether the total number of occurrences of each integer in C exceeds its count in S. If $S = \{1, 1\}$ and $C = \{1, 1, 1\}$, line 6 verifies each 1 in C independently and never catches that C has too many 1's. This failure to enforce correct repeats allows invalid certificates to be accepted, so VerifierA does not establish $\textsc{SubsetSum} \in \mathsf{NP}$.

(4 pts) (b) ϒ realizes their mistake and gives you another verifier, which takes a *collection* of nonnegative integers *multisets* as certificate (these multisets may or may not be submultisets of S) as follows.

```
Input: A multiset of nonnegative integers $S = \{s_1, s_2, \dots, s_n\}$, an integer $k$, and
    a collection of nonnegative integers multisets $C = \{C_1, C_2, \dots, C_m\}$
1: function VerifierB($S, k, C$)
2:     for $C_i \in C$ do // Iterate over multisets in $C$
3:         Confirm that $C_i \subseteq S$
4:         if $\sum_{c \in C_i} = k$ then accept
5:     reject
```

Correctness: For correctness,

$$\begin{aligned}(S,k) \in \textsc{SubsetSum} &\iff \text{there exists a submultiset } C_i \subseteq S \text{ s.t. } \sum_{c \in C_i} c = k \\ &\iff \text{there exists a collection } C \text{ that contains } C_i \\ &\iff \textsc{VerifierB}(S,k,C) \text{ accepts}\end{aligned}$$

Runtime: In line 4, since C_i is confirmed to be a submultiset of S, $|C_i| \le n$ and hence summing all items takes $O(n)$. Since there are m sets in C, the overall runtime is $O(mn)$, which is polynomial with respect to the input size.

Explain why VerifierB does not show that SubsetSum $\in$ NP. You may use an example to support your reasoning. You do not have to fix the verifier, though it is a good exercise.

Solution: VerifierB takes a "collection of submultisets" as the certificate without restricting its size to be polynomial in $|S|$. Since there can be exponentially many submultisets in such a collection, the run time (in terms of $|S|$ alone) can become unbounded. For example, given $S = \{1, 2, \dots, n\}$, a certificate could include all 2^n possible submultisets of S. Even though each submultiset can be checked in polynomial time, the total number of submultisets makes verification exponential in $|S|$. Because the runtime must be polynomial in the size of the *original input* to show SubsetSum $\in$ NP, VerifierB fails to establish membership in NP.

(4 pts) (c) Frustrated by Υ's mistakes, Lambda presents the following verifier:

```
Input: A multiset of nonnegative integers $S = \{s_1, s_2, \dots, s_n\}$, an integer $k$, and
    an array of bits $C[1, \dots, n]$
1: function VerifierC($S, k, C$)
2:     $sum \leftarrow 0$
3:     for $i = 1, \dots, n$ do
4:         if $C[i] = 1$ then $sum \leftarrow sum + s_i$
5:     if $sum = k$ then accept
6:     reject
```

Υ claims that Lambda's verifier is incorrect with the following counterexample:

Consider $S = \{1, 1, 2, 5\}$, $k = 7$, and $C = [0, 1, 0, 1]$. Clearly, $S \in$ SubsetSum, but $1 + 5 = 6 \neq 7$ so the verifier rejects. Therefore, VerifierC must be incorrect!

Explain why Υ's claim is unfounded.

Solution: A single failing certificate for an instance in a language does not invalidate a verifier. By definition, NP only requires that some certificate (not *every* certificate) is accepted for each instance in the language. The bit array $[0, 1, 0, 1]$ does not yield a sum of 7, so the verifier correctly rejects it. Another bit array ($[1, 1, 0, 1]$ here) leads to a sum of 7, which would be accepted, satisfying correctness.

4. **Understanding poly-time mapping reductions (PTMRs, aka Karp reductions).**

If you haven't already, carefully read Handout 6: NP-Hardness Proofs and apply it to this problem and all subsequent ones.

(5 pts) (a) Prove or disprove: if $A \le_p B$, then $A \le_T B$.
(A disproof would consist of a pair of languages such that $A \le_p B$ but $A \not\le_T B$.)

Solution: Suppose there is a polynomial-time mapping reduction from A to B via a function f. On input x:

1. Compute $f(x)$ in polynomial time.
2. Query the oracle for B on $f(x)$.
3. Accept if the oracle says "yes," and reject otherwise.

This is a Turing reduction from A to B, so $A \le_T B$.

(5 pts) (b) Prove or disprove: if $A \le_T B$, then $A \le_p B$.
(A disproof would consist of a pair of languages such that $A \le_T B$ but $A \not\le_p B$.)

Solution: Turing reductions allow multiple adaptive queries to an oracle for B. Polynomial-time mapping reductions require an efficient and specific transformation of input before a new membership query. There exist languages A and B such that $A \le_T B$ yet no polynomial-time computable function f satisfies

$$x \in A \iff f(x) \in B.$$

For instance, many languages are Turing reducible to the Halting Problem, but they do not admit a single polynomial-time (efficient) mapping to that problem. Hence there is a pair (A, B) with $A \le_T B$ but $A \not\le_p B$.

(6 pts) (c) Prove that PTMRs are *transitive*: if $A \le_p B$ and $B \le_p C$, then $A \le_p C$.

Solution: Suppose there is a polynomial-time mapping reduction f transforming instances of A into instances of B, and a polynomial-time mapping reduction g transforming instances of B into instances of C. Define $h(x) = g(f(x))$. h is the composition of two polynomial-time functions, so it is also polynomial-time. For every x, $x \in A \iff f(x) \in B \iff g(f(x)) \in C \iff h(x) \in C$. Thus $A \le_p C$.

(6 pts) (d) The languages Σ^* and $\emptyset$ are called the *trivial* languages, and all other languages are *non-trivial.* Prove that if $\mathsf{P} = \mathsf{NP}$, then *every* non-trivial language is NP-hard.
Be sure to state where your proof relies on non-triviality (because the trivial languages are *not* NP-hard, regardless of whether $\mathsf{P} = \mathsf{NP}$!).

Solution: Let L be any NP-complete language. Assume $\mathsf{P} = \mathsf{NP}$. Then L can be decided by some polynomial-time algorithm. Let A be a non-trivial language. Since $A \neq \Sigma^*$ and $A \neq \emptyset$, there is at least one string $s \in A$ and at least one string $t \notin A$.

1. Construct a polynomial-time function f from inputs of L to instances of A:

$$f(x) = \begin{cases} s & \text{if } x \in L, \\ t & \text{if } x \notin L. \end{cases}$$

2. This reduction clearly runs in polynomial time, since deciding L is (by assumption) a polynomial-time procedure, and the construction of either s or t from x takes polynomial time.

3. *Correctness:*

$$x \in L \quad \Longleftrightarrow \quad f(x) = s \in A, \qquad x \notin L \quad \Longleftrightarrow \quad f(x) = t \notin A.$$

   Hence

$$x \in L \quad \Longleftrightarrow \quad f(x) \in A.$$

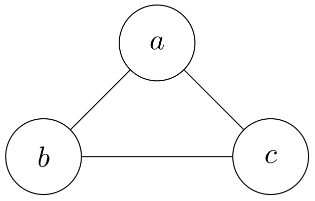
   Thus, $L \leq_p A$.

Since L is NP-complete, A is NP-hard.

The existence of strings $s \in A$ and $t \notin A$ is essential. Such strings do not exist for Σ^* or $\emptyset$, so, if A is trivial, then f cannot be constructed. Also, this means that the trivial languages are never NP-hard.

5. **Dominating sets.**

Consider an undirected graph $G = (V, E)$. A *dominating set* of a graph is a subset of vertices D such that every vertex in V is either in D or connected to some vertex/vertices in D. For example, consider the following graph:



The graph has a dominating set of size 1, for example, $D = \{a\}$ is a dominating set because $a \in D$ and both b and c are connected to a.

Notice the difference between a dominating set and a vertex cover: In the graph above, $D = \{a\}$ is *not* a vertex cover because it does not cover the edge (b, c). However, $C = \{b, c\}$ is both a vertex cover and a dominating set.

Let G be an undirected graph and k be a nonnegative integer. Consider the following language:

$$\textsc{Dominating-Set} = \{(G, k) = G \text{ has a dominating set of size (at most) } k\}.$$

In this problem, you will show that $\textsc{Dominating-Set}$ is NP-complete.

(6 pts) (a) Prove that $\textsc{Dominating-Set} \in \mathsf{NP}$. Specifically, you need to

(i) define a certificate[1] and argue that it is polynomial in size with respect to the instance[2],

(ii) give an efficient verifier for the language that takes in an instance and a certificate as input,

(iii) give a correctness argument for the verifier, and

(iv) give a runtime analysis for the verifier.

In your verifier, you may assume that the certificate is well-formed, so you do not have to check for it's *data type* (e.g., a graph), but you still have to check for its *validity*.

Solution:

(i) A suitable certificate is a set of vertices $D \subseteq V$ with $|D| \leq k$. This set can be encoded using a list of vertex identifiers, which is of size polynomial in the size of the input.

(ii) The verifier takes as input (G, k) and the certificate D. It checks that $|D| \leq k$. Then, for each vertex $v \in V$, it checks whether $v \in D$ or there is an edge from v to some $u \in D$. If all vertices satisfy this property, the verifier accepts; otherwise, it rejects.

[1]By "defining a certificate," we mean describing the *data type* of the certificate. For example, the certificate may be an array, a set, a graph, etc. You may define it in the "Input" section of the algorithm, similar to Υ's and Λ's verifiers in Question 3. You should not require the certificate to satisfy any criteria that would trivialize its validity.

[2]The specific polynomial can depend on the verifier, but the same polynomial bounds the certificate size for *all* instances of the language.

(iii) If there is a dominating set of size at most k, then providing that set as a certificate causes the verifier to accept. If there is no dominating set of size at most k, then no certificate can pass the verifier's checks.

(iv) Checking, for each v, whether there is an edge to some $u \in D$ can be done in polynomial time by scanning adjacency lists or an adjacency matrix. Hence, the verification runs in polynomial time.

(4 pts) (b) Since Dominating-Set has some resemblance to VERTEX-COVER, a good source of inspiration is the 3SAT $\leq_p$ VERTEX-COVER reduction from lecture.

We modify the reduction as follows: Given a 3SAT instance (a 3CNF formula) ϕ, we output $f(\phi) = (G, k)$ where the "budget" $k = n$ is the number of variables in ϕ, and graph G is constructed as follows.

- For each variable x_i in ϕ we create a "variable gadget" consisting of **four** vertices labeled T_i (for true), F_i (for false), α_i, and β_i, along with edges (T_i, F_i), (α_i, T_i), (α_i, F_i), (β_i, T_i), and (β_i, F_i).
- For each clause C_j in ϕ we create a **single** vertex labeled C_j, and introduce an edge between C_j and T_i if the literal x_i appears in C_j, and an edge between C_j and F_i if the literal $\overline{x_i}$ appears in C_j.

This reduction is correct, which you will prove in Part c.

Now, consider the following "yes" instance in 3SAT:

$$\phi = (x_1 \vee \overline{x}_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee x_3) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3)$$

Determine $f(\phi) = (G, k)$, i.e., give the graph output by the reduction given ϕ and specify what k should be. Then, show that $\phi \in$ 3SAT by giving *a* dominating set of G that has (at most) k vertices (if there are multiple, just give any *one* of them).

Solution: $k = 3$. For each variable x_i, create four vertices $T_i, F_i, \alpha_i, \beta_i$ with edges

$$(T_i, F_i),\ (\alpha_i, T_i),\ (\alpha_i, F_i),\ (\beta_i, T_i),\ (\beta_i, F_i).$$

For each clause C_j, create a single vertex labeled C_j and add edges from C_j to T_i if x_i appears in C_j, and from C_j to F_i if $\overline{x_i}$ appears in C_j. Here, this looks like:

$$C_1 \text{ connects to } T_1,\ F_2,\ T_3, \quad C_2 \text{ connects to } F_1,\ T_2,\ F_3,$$

$$C_3 \text{ connects to } F_1,\ F_2,\ T_3, \quad C_4 \text{ connects to } F_1,\ F_2,\ F_3.$$

To show that $\phi \in$ 3SAT, an assignment for ϕ is $x_1 = \text{false},\ x_2 = \text{false},\ x_3 = \text{true}$.

$$D = \{\, F_1,\ F_2,\ T_3 \}.$$

Each variable gadget is dominated because α_i and β_i connect to both T_i and F_i, and each clause vertex C_j has an edge to at least one chosen T_i or F_i. D is a dominating set of size at most $k = 3$.

(8 pts) (c) Prove that the reduction in Part b is correct, i.e., show that

$$\phi \in \text{3SAT} \iff f(\phi) = (G, k) \in \textsc{Dominating-Set}.$$

Moreover, the reduction takes polynomial time in the size of the input formula, because it creates $O(1)$ vertices and edges per vertex and clause. Since 3SAT is NP-hard, we can conclude that Dominating-Set is NP-hard.

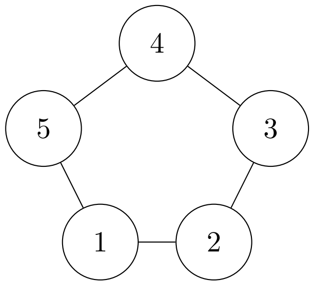
Solution:

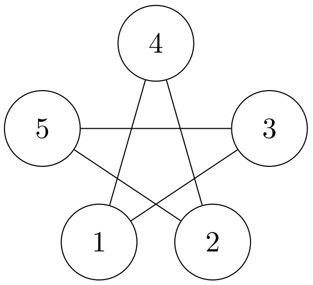
- *Forward direction:* If ϕ is satisfiable, for each variable x_i, if $x_i = \text{true}$, pick the vertex T_i. Otherwise, pick F_i (if $x_i = \text{false}$). This set has size equal to the number of variables and dominates all clause vertices (each clause connects to a satisfied literal) as well as all vertices α_i, β_i.
- *Reverse direction:* If there is a dominating set of size k in G, at most one vertex among $\{\alpha_i, \beta_i\}$ can dominate those vertices, so each variable gadget must include exactly one of T_i or F_i. This induces a satisfying assignment for ϕ: setting $x_i = \text{true}$ if $T_i \in$ the dominating set and $x_i = \text{false}$ if $F_i \in$ the dominating set. Since each clause vertex is dominated by one of the chosen T_i or F_i, every clause has a satisfied literal.
- The graph has $O(n)$ vertices from variables plus $O(m)$ vertices from clauses (where n is the number of variables and m is the number of clauses). Each edge is added in constant time per literal. Hence the construction is polynomial in the size of ϕ.

Since 3SAT is NP-hard, it follows that Dominating-Set is NP-hard.

6. **Subgraph isomorphism.**

For two graphs $G = (V, E)$ and $H = (W, F)$, an *isomorphism* between them is a *bijection* $f : V \to W$ such that $(u, v) \in E$ if and only if $(f(u), f(v)) \in F$. Two graphs are said to be *isomorphic* if there exists an isomorphism between them. For example, the following two graphs are isomorphic, by the bijection f where $f(1) = 1$, $f(2) = 3$, $f(3) = 5$, $f(4) = 2$, and $f(5) = 4$.





Recall that a subgraph $G' = (V', E')$ of $G = (V, E)$ is a graph such that $V' \subseteq V$ and $E' \subseteq E$. Let G and H be undirected graphs. Consider the following language

$$\textsc{Subgraph-Isomorphism} = \{(\langle G\rangle, \langle H\rangle) : G \text{ has a subgraph } G' \text{ that is isomorphic to } H\}.$$

In this problem, we will prove that Subgraph-Isomorphism is NP-complete.

(6 pts) (a) Prove that Subgraph-Isomorphism $\in$ NP. Remember to include all steps as in Question 5 Part a.

Solution:

(i) Let the instance be (G, H), with $G = (V, E)$ and $H = (W, F)$. A valid certificate can be specified by a function $f : W \to V$, represented as a list of vertex images $\big(f(w_1), f(w_2), \ldots, f(w_{|W|})\big)$. This certificate has size polynomial in $|V|$ and $|W|$.

(ii) Given (G, H) and the certificate f, the verifier performs the following checks:

(a) Ensure that f is injective; that is, no two distinct vertices of W map to the same vertex of V.

(b) For every edge $(w_i, w_j) \in F$, check that $\big(f(w_i), f(w_j)\big) \in E$.

(c) For every non-edge $(w_i, w_j) \notin F$, check that $\big(f(w_i), f(w_j)\big) \notin E$ if subgraph isolation requires preserving the absence of edges among only those vertices in the induced subgraph on $f(W)$. (Alternatively, it suffices to check edge preservation if the definition of subgraph isomorphism does not require completeness of non-edges. Standard definitions typically only require adjacency preservation, so many presentations omit this step.)

(iii)
- If G contains a subgraph isomorphic to H, there exists an injective mapping f that preserves adjacency. This mapping will pass the verifier's checks.
- If no such subgraph exists, no certificate f can satisfy all adjacency requirements.

(iv) Checking injectivity can be done in $O(|W|)$ time by recording used vertices in V. Checking all edges in F against E is $O(|F|)$. All these steps run in polynomial time in the sizes of G and H. Hence the verifier runs in polynomial time.

(12 pts) (b) Prove that CLIQUE $\leq_p$ Subgraph-Isomorphism. Specifically, you need to

(i) give a PTMR from CLIQUE to Subgraph-Isomorphism,

(ii) give a correctness argument for the reduction, and

(iii) give a runtime analysis for the reduction.

Since CLIQUE is NP-hard, we can conclude that Subgraph-Isomorphism is NP-hard.

Hint: Consider different ranges of k in relation to $|V|$.

Solution:

(i) Let (G, k) be an instance of the CLIQUE problem, where $G = (V, E)$. Construct an instance $\big(G, K_k\big)$ of Subgraph-Isomorphism, where K_k is the complete graph on k vertices.

(ii)
- ($\Rightarrow$) If G contains a k-clique, then there is a set of k vertices in G such that all pairs in that set are adjacent. Those k vertices span a subgraph of G isomorphic to K_k, so (G, K_k) is a "yes" instance of Subgraph-Isomorphism.
- ($\Leftarrow$) If (G, K_k) is a "yes" instance of Subgraph-Isomorphism, then G has a subgraph on k vertices that is isomorphic to K_k. That subgraph corresponds to a clique of size k in G.

(iii) The graph K_k can be constructed in $O(k^2)$ time by adding k vertices and connecting every pair. Incorporating G unchanged is $O(|V| + |E|)$. Hence the entire reduction runs in polynomial time with respect to the size of (G, k).

Thus, CLIQUE $\leq_p$ Subgraph-Isomorphism. Since CLIQUE is NP-hard and $(G, k) \mapsto (G, K_k)$ is a PTMR, Subgraph-Isomorphism is also NP-hard.

(8 pts) 7. **Big independent sets.**

Recall that an *independent set* of an undirected graph $G = (V, E)$ is a subset $I \subseteq V$ such that no two (distinct) vertices in I have an edge between them.

In lecture, we proved that the following language is NP-complete.

$$\text{IS} = \{(G, k) : G = (V, E) \text{ has an independent set of size (at least) } k\}.$$

In this problem, we will explore the following variant of IS.

$$\text{BIG-IS} = \{G : G = (V, E) \text{ has an independent set of size (at least) } |V| - 1\}.$$

Either prove that BIG-IS $\in \mathsf{P}$ (by giving a polynomial-time algorithm that decides BIG-IS) or that BIG-IS is NP-hard (by showing that IS $\leq_p$ BIG-IS). If you are giving a reduction, remember to include all steps as in Question 6 Part b.

Solution: BIG-IS $\in \mathsf{P}$. Let $G = (V, E)$ be an input graph. The following algorithm decides whether $G \in$ BIG-IS in polynomial time:

1. If $E = \emptyset$, accept, because $|V|$ vertices form an independent set.
2. Otherwise, pick any edge $(u, v) \in E$.
3. Check whether every edge in E is incident on u. If so, accept.
4. Check whether every edge in E is incident on v. If so, accept.
5. Otherwise, reject.

Correctness follows from the observation that a graph admits an independent set of size $|V| - 1$ exactly when there is at most one vertex that all edges share as an endpoint. If there are two such vertices, then they are adjacent, and the independent set cannot include both. If there are no such vertices, then all vertices are independent. If there are more than two, the algorithm correctly rejects.

The runtime of the algorithm is polynomial in $|V| + |E|$.

BIG-IS $\in \mathsf{P}$.