Combinatorial Optimization

Problem set 8: solutions

1. Fix constants $a \in \mathbb{R}$ and $b > 1$. For $n \in \mathbb{N}$, let $f(n) = n^a$ and $g(n) = b^n$. Prove that $f(n) = o\big(g(n)\big)$.

▷ **Solution.** First we observe that $g(n) \geq 0$ for all $n \in \mathbb{N}$ and $\lim_{n\to\infty} g(n) = \lim_{n\to\infty} b^n = \infty$ because $b > 1$. Now we consider three cases based on the value of a.

Case 1: $a \leq 0$. Then for all $n \in \mathbb{N}$ we have $0 \leq n^a \leq n^0 = 1$, so $0 \leq f(n)/g(n) \leq 1/g(n)$ for all $n \in \mathbb{N}$. Therefore, by the squeeze theorem,

$$0 \leq \lim_{n\to\infty} \frac{f(n)}{g(n)} \leq \lim_{n\to\infty} \frac{1}{g(n)} = 0,$$

so $\lim_{n\to\infty} f(n)/g(n) = 0$, which is to say, $f(n) = o\big(g(n)\big)$.

Case 2: a is a positive integer. Since $a > 0$, $\lim_{n\to\infty} f(n) = \lim_{n\to\infty} n^a = \infty$, so the form of the limit $\lim_{n\to\infty} f(n)/g(n)$ is ∞/∞, which is an indeterminate form. Therefore, by L'Hôpital's rule,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^a}{b^n} \overset{\text{L'H}}{=} \lim_{n\to\infty} \frac{an^{a-1}}{b^n \ln b} \overset{\text{L'H}}{=} \lim_{n\to\infty} \frac{a(a-1)n^{a-2}}{b^n (\ln b)^2} \overset{\text{L'H}}{=} \cdots \overset{\text{L'H}}{=} \lim_{n\to\infty} \frac{a!}{b^n (\ln b)^a} = 0,$$

the last equality following from the fact that the numerator is a constant while the denominator tends to infinity. So $f(n) = o\big(g(n)\big)$.

Case 3: $a > 0$ is not an integer. Let $A = \lceil a \rceil$. Then for all $n \in \mathbb{N}$ we have $0 \leq n^a/b^n \leq n^A/b^n$, and by case 2 above we know that $\lim_{n\to\infty} n^A/b^n = 0$, so by the squeeze theorem $\lim_{n\to\infty} f(n)/g(n) = \lim_{n\to\infty} n^a/b^n = 0$. Hence $f(n) = o\big(g(n)\big)$. □

2. Carefully state the decision (recognition) version of the minimum spanning tree problem. Prove that this problem is in P.

▷ **Solution.** In MINIMUM SPANNING TREE, the decision version of the minimum spanning tree problem, an instance is a simple undirected graph $G = (V, E)$, a function $w : E \to \mathbb{Q}$ specifying edge weights, and a threshold value $k \in \mathbb{Q}$, and the question is whether G contains a spanning tree with total edge weight no greater than k. (It is not particularly important that the codomain of w and the set from which k is chosen be $\mathbb{Q}$.)

The graph G in the instance may not be connected, so we first perform the following transformation to convert the instance (G, w, k) into another instance (G', w', k) such that G' is *complete* (i.e., every two vertices are adjacent) and such that G' contains a spanning tree with total edge weight no greater than k if and only if G does. Let z be the sum of all negative edge weights in the instance:

$$z = \sum_{\substack{e \in E \\ \text{such that} \\ w(e) < 0}} w(e).$$

Then take $G' = (V, E')$, where $E' = \big\{ \{u, v\} : u, v \in V,\ u \neq v \big\}$. Define the edge weights $w'(\{u, v\})$ for $\{u, v\} \in E'$ as follows:

$$w'(\{u,v\}) = \begin{cases} w(\{u,v\}), & \text{if } \{u,v\} \in E; \\ k - z + 1, & \text{otherwise.} \end{cases}$$

The value of z can be found in $O(|E|) = O(|V|^2)$ time, and the construction of E' and w' can be done in $O(|V|^2)$ time, so this whole transformation can be done in $O(|V|^2)$ time.. Now, any spanning tree in G is also a spanning tree in G' with the same total edge weight, so if G contains a spanning tree with weight no greater than k, then so does G'. Conversely, no spanning tree in G' with weight no greater than k can contain an edge of weight $k - z + 1$, because the total edge weight of a subset of edges containing such an edge is at minimum $(k-z+1)+z=k+1>k$. Therefore, if G' contains a spanning tree with weight no greater than k, then it uses only edges that are also edges in G (of the same weight), so G also contains a spanning tree with weight no greater than k. Hence we may solve the original instance (G,w,k) by solving the instance (G',w',k).

As we showed in class on June 17, a minimum spanning tree for a connected graph can be found using Prim's algorithm. The specific implementation of Prim's algorithm that was presented in the lecture is as follows:

1. Initialize $U := \{u_1\}$, where u_1 is an arbitrary vertex in V, and $T := \emptyset$.
2. If $U = V$, we are done. Stop.
3. Out of all edges having exactly one endpoint in U, choose one with minimum weight, say $\{u,v\}$ where $u \in U$ and $w \notin U$.
4. Add v to U and add $\{u,v\}$ to T.
5. Go back to step 2.

The initialization of this algorithm can be done in constant time. The set U begins with a single vertex, each iteration of the algorithm adds one vertex to U, and the algorithm stops when $U = V$, which means the algorithm will run for $|V|-1 = O(|V|)$ iterations. In each iteration, the algorithm must select one edge in step 3. To do this, we must first identify the edges that have exactly one endpoint in U; this can be done by iterating through all edges in E and, for each edge $\{u,v\}$, iterating through V attempting to find u and iterating through V attempting to find v, and then testing whether ($u \in U$ and $v \notin U$) or ($u \notin U$ and $v \in U$). This part can be done in $O(|E| \cdot |V|)$ time. Then, out of those edges, of which there are at most $|E|$, we must find the one that has minimum weight; this can be done in $O(|E|)$ time (see the algorithm for finding the minimum element in an unordered list described in class on June 29). So step 3 of the algorithm can be done in $O(|E| \cdot |V|) = O(|V|^3)$ time. Steps 2, 4, and 5 can be done in constant time. Steps 2 through 5 are performed $O(|V|)$ times, so the whole algorithm runs in $O(|V|^4)$ time.

After we run Prim's algorithm to find a minimum spanning tree in G', we calculate its total edge weight W; this can be done in $O(|V|)$ time because the tree has $|V|-1$ edges. Then we compare W to k, which can be done in constant time, and we return "yes" if $W \le k$ or "no" if $W > k$.

This whole algorithm—the transformation from (G,w,k) to (G',w',k) in $O(|V|^2)$ time, the running of Prim's algorithm in $O(|V|^4)$ time, and the final calculations and comparison in $O(|V|)$ time—can be done in $O(|V|^4)$ time, which is polynomial in the size of the instance. Therefore MINIMUM SPANNING TREE is in P. □

3. Prove that if we had a polynomial-time algorithm for *computing the length* of the shortest TSP tour, then we would have a polynomial-time algorithm for *finding* the shortest TSP tour. (In other words, show how to solve the optimization version of the traveling salesman problem in polynomial time given a polynomial-time algorithm for solving the evaluation version.)

▷ **Solution.** An instance of the traveling salesman problem consists of a positive integer n and an $n \times n$ matrix C of costs c_{ij}. An instance therefore contains $\Theta(n^2)$ numbers. Note that we are not assuming that $c_{ij} = c_{ji}$; the costs may be asymmetric. We do assume that the costs c_{ij} are rational numbers, which is a reasonable assumption. (Allowing arbitrary real-valued costs introduces some troublesome complications, among which is the issue of representation—there are uncountably many real numbers, but only countably many finite strings of symbols from a fixed finite alphabet, so it is impossible to represent all real numbers in a practical model of computation.)

Suppose that we have a polynomial-time algorithm $\mathcal{L}$ for computing the length of the shortest tour; so the input to $\mathcal{L}$ is a pair (n,C), and the output is a single number giving the length of the shortest tour. Observe that finding a shortest tour is equivalent to finding the *set of arcs* (i,j) that the tour uses. (Converting from a set of arcs to a sequence of nodes, or vice versa, can easily be done in polynomial time.)

Let us first assume that all costs are integers. Let us also assume that the shortest tour is unique. Then we can find the shortest tour as follows: Given an instance (n,C), use $\mathcal{L}$ to compute the length z^* of the shortest tour. Now, for each arc (i,j), $i \neq j$, let $C'_{(ij)}$ be the matrix obtained from C by replacing the single entry c_{ij} with $c_{ij}+1$, and use $\mathcal{L}$ to compute the length $z'_{(ij)}$ of the shortest tour in $(n, C'_{(ij)})$. If $z'_{(ij)} = z^*$, then the arc (i,j) is not used in the unique shortest tour in (n,C), because increasing its cost did not increase the cost of this tour. On the other hand, if $z'_{(ij)} > z^*$, then the arc (i,j) *is* used in the shortest tour in (n,C). So let $U = \{\,(i,j) : z'_{(ij)} > z^*\,\}$. The arcs in U are precisely those used in the unique shortest tour in (n,C). Each matrix $C'_{(ij)}$ can be constructed in $O(n^2)$ time from the matrix C, and the rest of the calculations for a given arc (i,j) can be done in a constant number of elementary operations plus one call to $\mathcal{L}$. These steps are repeated for each of the $\Theta(n^2)$ arcs (i,j), so this algorithm requires $O(n^4)$ elementary operations plus $O(n^2)$ calls to $\mathcal{L}$. If $\mathcal{L}$ runs in polynomial time, then this gives us a polynomial-time algorithm to find the shortest tour.

If the costs are rational numbers rather than integers, then we can first multiply all of the costs by their least common (positive) denominator d, which will convert them to integers. There are exactly n arcs in every tour, so this will multiply the cost of every tour by the same positive factor nd, and therefore a tour is optimal for this modified cost matrix if and only if it was optimal for the original matrix. This transformation can be performed with $O(n^2)$ arithmetic operations (counting lcm as a single arithmetic operation), which includes both finding the value of d and multiplying every cost by this value. [To be more careful, the least common denominator can be found using the Euclidean algorithm, which can be shown to run in polynomial time in the number of bits in the input, so this is not a problem.] Therefore, once we have a polynomial-time algorithm for finding the shortest tour in an instance with integer costs, we can also use it to find the shortest tour in an instance with rational costs in polynomial time.

Now we remove the assumption that the shortest tour is unique. For $1 \le i \le n$ and $1 \le j \le n$, let $p_n(i,j) = n(i-1)+j$. So $p_n(1,1)=1$, $p_n(1,2)=2$, $\ldots$, $p_n(1,n)=n$, $p_n(2,1)=n+1$, $p_n(2,2)=n+2$, $\ldots$, $p_n(2,n)=2n$, $p_n(3,1)=2n+1$, $\ldots$, $p_n(n,n)=n^2$; the function p_n assigns distinct integers in $\{1,2,\ldots,n^2\}$ to all of the ordered pairs (i,j) with $1 \le i \le n$ and $1 \le j \le n$. Given a TSP instance (n,C) with rational costs, first perform the transformation described above to transform it to an instance (n,C') with all integer costs. Then transform this to another instance (n,C'') by replacing each cost c'_{ij} with $c''_{ij} = c'_{ij} + 1/2^{p_n(i,j)}$; so c'_{11} is increased by $1/2$, c'_{12} is increased by $1/4$, c'_{13} is increased by $1/8$, and so on. [This second transformation can be done with $O(n^2)$ arithmetic operations.] Note that the fractional parts of the costs c''_{ij} are distinct positive integer powers of $1/2$, so no two sums of subsets of these costs can have the same fractional part, which means that the optimal tour in (n,C'') must be unique. Furthermore, because the costs in C' are all integers, the cost of any non-optimal tour in (n,C) is at least 1 greater than the cost of an optimal tour; and the maximum amount that the cost of a tour in (n,C') is increased in (n,C'') is certainly no more than $\sum_{k=1}^{n^2} 1/2^k < 1$, so an optimal tour in (n,C') cannot be made to have larger total weight in (n,C'') than a tour that is not optimal in (n,C'), so the unique shortest tour in (n,C'') is also optimal in (n,C'), and hence in (n,C). Thus we can use the algorithm described above to find the unique shortest tour in (n,C'') in polynomial time, which is also a shortest tour in (n,C). $\square$

[In fact, there is a way to use this last trick to find a shortest tour in (n,C) with only *one* call to $\mathcal{L}$: Transform (n,C) to (n,C'') as described above, and then use $\mathcal{L}$ to find the length z^* of the shortest tour in (n,C''). By the construction of the costs c''_{ij}, the fractional part of z^*, when written in binary, will contain a 1 in position $p_n(i,j)$ if and only if the arc (i,j) is used in the shortest tour! To get the same effect in base ten rather than binary, use $1/10$ rather than $1/2$ in the construction of the costs c''_{ij}.]

4. In the VERTEX COVER problem, an instance is a simple undirected graph $G = (V, E)$ and a positive integer $k \le |V|$, and the question is whether there exists a *vertex cover* of size no greater than k, that is, a subset $V' \subseteq V$ with $|V'| \le k$ such that for every edge $\{u, v\} \in E$ at least one of the endpoints u and v belongs to V'. Prove that VERTEX COVER is in NP.

▷ **Solution.** To prove that VERTEX COVER is in NP, we need to describe a verifier and, for every "yes" instance, a polynomial-size certificate that the verifier will accept in polynomial time; and we need to ensure that the verifier will *not* accept any fraudulent certificate for a "no" instance.

A certificate will consist of a set V'. For a "yes" instance, the certificate that will be accepted by the verifier will be a vertex cover $V' \subseteq V$ of size no greater than k, which must in turn be no greater than $|V|$. Such a certificate consists of a list of k vertices in V, so it has size $O(|V|)$ [or $O(|V| \log |V|)$ if we are counting bits]. This is polynomial in the size of the instance.

The verifier will act as follows. Remember that the input to the verifier consists of an instance and a certificate, so the verifier has access to V, E, and k as well as the certificate V'.

- First, the verifier must verify that $|V'| \le k$. This can be done by iterating through the elements of the set V' given in the certificate and counting them, and then comparing the final tally with k. For a "yes" instance and an appropriate certificate, we have $|V'| \le k \le |V|$, so this stage of the verification can be done in $O(|V|)$ time. If this stage fails (i.e., if it is determined that $|V'| > k$), then the verifier will reject the certificate.
- Next, the verifier must verify that $V' \subseteq V$. This can be done by iterating through the elements of V' and, for each one, iterating through the elements of V until a match is found or the end of V is reached without a match. For a "yes" instance and an appropriate certificate, we have $|V'| \le k \le |V|$, so there will be $O(|V|)$ "outer" iterations, and each of these outer iterations will take $O(|V|)$ "inner" iterations through the elements of V. So this stage of the verification can be done in $O(|V|^2)$ time. If this stage fails (i.e., if it is determined that $V' \nsubseteq V$), then the verifier will reject the certificate.
- Then the verifier must verify that for every edge $\{u, v\} \in E$ at least one of the endpoints u and v belongs to V'. This can be done by iterating through the elements of E. For each edge $\{u, v\} \in E$, the verifier iterates through the elements of V' attempting to find u, and if that fails it iterates through the elements of V' attempting to find v. For a "yes" instance and an appropriate certificate, we have $|V'| \le k \le |V|$, so the verification for each edge $\{u, v\} \in E$ takes at most $2|V'| = O(|V|)$ comparisons, and this is done for each of $|E| = O(|V|^2)$ edges, so this stage of the verification can be done in $O(|V|^3)$ time. If this stage fails (i.e., if it is determined that E contains an edge $\{u, v\}$ such that neither u nor v is in V'), then the verifier will reject the certificate.
- If the verifier gets to this point, then it has verified that $|V'| \le k$, $V' \subseteq V$, and for every edge $\{u, v\} \in E$ at least one of the endpoints u and v belongs to V'. This means that V' is indeed a vertex cover of size no greater than k, so the answer to the instance is "yes," and the verifier will accept the certificate.

For a "yes" instance and an appropriate certificate, the three stages of this verification process can be done in $O(|V|)$ time, $O(|V|^2)$ time, and $O(|V|^3)$ time, respectively, so the entire verification process takes $O(|V|^3)$ time, which is polynomial in the size of the instance. Moreover, the verifier will accept the certificate *only if* the answer to the instance really is "yes," because the verifier will accept the certificate only if the certificate gives a vertex cover of size no greater than k. Therefore VERTEX COVER is in NP. □

5. In the 3-COLORABILITY problem, an instance is a simple undirected graph $G = (V, E)$, and the question is whether there exists a proper 3-coloring of the vertices of G, that is, a function $f : V \to \{1, 2, 3\}$ such that $f(u) \neq f(v)$ for every edge $\{u, v\} \in E$. (Think of the function f as assigning each vertex a "color" in the set $\{1, 2, 3\}$; then no two adjacent vertices can be assigned the same color.) The 4-COLORABILITY problem is defined analogously. Describe and justify a polynomial-time transformation from 3-COLORABILITY to 4-COLORABILITY.

▷ **Solution.** Let $G = (V, E)$ be an instance of 3-COLORABILITY. Our goal is to construct a graph $G' = (V', E')$, an instance of 4-COLORABILITY, such that G' has a proper 4-coloring if and only if G has a proper 3-coloring.

Let $V' = V \cup \{\alpha\}$, where α is a new vertex not in V, and let $E' = E \cup \{\, \{\alpha, v\} : v \in V \,\}$, that is, join α to every vertex in V. This construction can be done in $O(|V|)$ time if the instance G can be modified in place (the only necessary modifications are the addition of one more vertex and $|V|$ more edges), or in $O(|V| + |E|)$ time if the instance G' must be written from scratch. In any case, it can be done in polynomial time in the size of G.

Suppose G' has a proper 4-coloring $\phi : V' \to \{1, 2, 3, 4\}$. The vertex α must receive some color; without loss of generality, suppose $\phi(\alpha) = 4$. Because α is adjacent to every other vertex in G', no other vertex may be given the same color as α, so $\phi(v) \in \{1, 2, 3\}$ for all $v \in V$. Furthermore, any two vertices u and v that are adjacent in V are also adjacent in V', because $E \subseteq E'$, so $\phi(u) \neq \phi(v)$. Therefore, restricting the domain of ϕ to $V = V' \setminus \{\alpha\}$ yields a proper 3-coloring $\phi\big|_V = f : V \to \{1, 2, 3\}$ of G.

Conversely, suppose G has a proper 3-coloring $f : V \to \{1, 2, 3\}$. Define $\phi : V' \to \{1, 2, 3, 4\}$ by $\phi(v) = f(v)$ if $v \in V$ and $\phi(\alpha) = 4$. Let $\{u, v\} \in E'$ be an edge in G'. If $u \in V$ and $v \in V$, then $\{u, v\} \in E$, so $\phi(u) = f(u) \neq f(v) = \phi(v)$. Otherwise, without loss of generality, $u = \alpha$ and $v \in V$, so $\phi(u) = 4$ and $\phi(v) = f(v) \in \{1, 2, 3\}$, which means $\phi(u) \neq \phi(v)$. Hence ϕ is a proper 4-coloring of G'.

Thus G' has a proper 4-coloring if and only if G has a proper 3-coloring, so this construction is a polynomial-time transformation from 3-COLORABILITY to 4-COLORABILITY. □

6. In the SUBGRAPH ISOMORPHISM problem, an instance consists of two simple undirected graphs $G = (V_G, E_G)$ and $H = (V_H, E_H)$, and the question is whether G contains a subgraph isomorphic to H, that is, a subgraph $K = (V_K, E_K)$ with $V_K \subseteq V_G$ and $E_K \subseteq E_G$ such that there exists a bijection $f : V_H \to V_K$ with $\{u, v\} \in E_H$ if and only if $\{f(u), f(v)\} \in E_K$. Prove that SUBGRAPH ISOMORPHISM is NP-complete.

▷ **Solution.** First we show that SUBGRAPH ISOMORPHISM is in NP. A certificate will consist of sets V_K and E_K and a function $f : V_H \to V_K$. For a "yes" instance, the certificate that will be accepted by the verifier will specify a subgraph $K = (V_K, E_K)$ of G and a bijection $f : V_H \to V_K$ such that $\{u, v\} \in E_H$ if and only if $\{f(u), f(v)\} \in E_K$. Such a certificate consists of lists of vertices and edges in G, together with a list of pairs $\big(v, f(v)\big)$ specifying the function f, so it has size $O(|V| + |E| + |V_H|)$, which is $O(|V|^2)$ because $|E| = O(|V|^2)$ and $|V_H| \leq |V|$ for a "yes" instance. [The certificate has size $O(|V|^2 \log |V|)$ if we are counting bits.] This is polynomial in the size of the instance. The verifier will act as follows:

- First, the verifier must verify that $V_K \subseteq V_G$. This can be done by iterating through the elements of V_K and, for each one, iterating through the elements of V_G until a match is found or the end of V_G is reached without a match. For a "yes" instance and an appropriate certificate, we have $|V_K| \leq |V_G|$, so there will be $O(|V_G|)$ "outer" iterations, and each of these outer iterations will take $O(|V_G|)$ "inner" iterations through the elements of V_G. So this stage of the verification can be done in $O(|V_G|^2)$ time. If this stage fails (i.e., if it is determined that $V_K \not\subseteq V_G$), then the verifier will reject the certificate.

- Then the verifier must verify that $E_K \subseteq E_G$. For a "yes" instance and an appropriate certificate, this stage of the verification can be done in $O(|E_G|^2) = O(|V_G|^4)$ time. If this stage fails (i.e., if it is determined that $E_K \nsubseteq E_G$), then the verifier will reject the certificate.
- Next, the verifier must verify that (V_K, E_K) is a graph—that for every edge $\{u, v\} \in E_K$ we have $u \in V_K$ and $v \in V_K$ and $u \neq v$. This can be done by iterating through the elements of E_K. For each edge $\{u, v\} \in E_K$, the verifier iterates through the elements of V_K attempting to find u, iterates again through the elements of V_K attempting to find v, and compares u and v to ensure they are distinct. For a "yes" instance and an appropriate certificate, we have $|E_K| \leq |E_G| = O(|V_G|^2)$ and $|V_K| \leq |V_G|$, so there are $O(|V_G|^2)$ "outer" iterations, and each of these outer iterations will take $O(|V_G|)$ "inner" iterations through the elements of V_K plus a single comparison to ensure that $u \neq v$. So this stage of the verification can be done in $O(|V_G|^3)$ time. If this stage fails (i.e., if it is determined that some edge $\{u, v\} \in E_K$ has $u \notin V_K$ or $v \notin V_K$ or $u = v$), then the verifier will reject the certificate.
- The verifier must then verify that the function $f : V_H \to V_K$ is well defined, which is to say that the list of pairs $\big(v, f(v)\big)$ specifying the function f gives one and only one value $f(v)$ in V_K for every $v \in V_H$. This can be done in two substages. First, the verifier iterates through the pairs $\big(v, f(v)\big)$, and for each such pair it iterates through the elements of V_H attempting to find v and iterates through the elements of V_K attempting to find $f(v)$. Second, the verifier iterates through the elements of V_H and, for each $u \in V_H$, iterates through the pairs $\big(v, f(v)\big)$, counting the number of times that $v = u$. For a "yes" instance and an appropriate certificate, the number of pairs $\big(v, f(v)\big)$ in the list specifying f is $|V_H|$, and we have $|V_H| = |V_K| \leq |V_G|$, so each of these two substages takes $O(|V_G|^2)$ time, which means that the stage as a whole takes $O(|V_G|^2)$ time. If this stage fails [i.e., if it is determined that some pair $\big(v, f(v)\big)$ has $v \notin V_H$ or $f(v) \notin V_K$, or that for some $u \in V_H$ the number of pairs $\big(v, f(v)\big)$ such that $v = u$ is not exactly 1], then the verifier will reject the certificate.
- Then the verifier must verify that f is a bijection. This can be done by iterating through the elements of V_K and, for each $u \in V_K$, iterating through the pairs $\big(v, f(v)\big)$ that specify the function f, counting the number of times that $f(v) = u$. For a "yes" instance and an appropriate certificate, we have $|V_H| = |V_K| \leq |V_G|$ and the number of pairs $\big(v, f(v)\big)$ is $|V_H|$, so there will be $O(|V_G|)$ "outer" iterations, and each of these outer iterations will take $O(|V_G|)$ "inner" iterations through the pairs $\big(v, f(v)\big)$. So this stage of the verification can be done in $O(|V_G|^2)$ time. If this stage fails [i.e., if it is determined that for some $u \in V_K$ the number of pairs $\big(v, f(v)\big)$ such that $f(v) = u$ is not exactly 1], then the verifier will reject the certificate.
- Finally, the verifier must verify that $\{u, v\} \in E_H$ if and only if $\{f(u), f(v)\} \in E_K$. To do this, the verifier iterates through the vertices in V_H, letting u be each such vertex in turn; for each vertex u, the verifier iterates through the vertices in V_H, letting v be each such vertex in turn; for each pair (u, v), the verifier iterates through the list of pairs specifying f in order to determine $f(u)$ and $f(v)$, iterates through the elements of E_H attempting to find $\{u, v\}$, iterates through the elements of E_K attempting to find $\{f(u), f(v)\}$, and then compares the results of these last two searches to see if they both succeeded or both failed. For a "yes" instance and an appropriate certificate, we have $|V_H| \leq |V_G|$ and $E_H = E_K = O(|V_G|^2)$, and the number of pairs specifying f is $|V_H|$. So there will be $|V_H| = O(|V_G|)$ u-iterations, each of which consists of $|V_H| = O(|V_G|)$ v-iterations, each of which consists of $O(|V_G|)$ iterations through the pairs specifying f, $O(|V_G|^2)$ iterations through the elements of E_H, and $O(|V_G|^2)$ iterations through the elements of E_K. Hence this stage of the verification can be done in $O(|V_G|^4)$ time. If this stage fails [i.e., if it is determined that there exist $u, v \in V_H$ such that $(\{u, v\} \in E_H$ but $\{f(u), f(v)\} \notin E_K)$ or $(\{u, v\} \notin E_H$ but $\{f(u), f(v)\} \in E_K)$], then the verifier will reject the certificate.

- If the verifier gets to this point, then it has verified that $V_K \subseteq V_G$, $E_K \subseteq E_G$, $K = (V_K, E_K)$ is a graph, the function $f : V_H \to V_K$ is well defined, f is a bijection, and $\{u, v\} \in E_H$ if and only if $\{f(u), f(v)\} \in E_K$. This means that K is a subgraph of G that is isomorphic to H, so the answer to the instance is "yes," and the verifier will accept the certificate.

For a "yes" instance and an appropriate certificate, the six stages of this verification process can be done in $O(|V_G|^2)$ time, $O(|V_G|^4)$ time, $O(|V_G|^3)$ time, $O(|V_G|^2)$ time, $O(|V_G|^2)$ time, and $O(|V_G|^4)$ time, respectively, so the entire verification process takes $O(|V_G|^4)$ time, which is polynomial in the size of the instance. Moreover, the verifier will accept the certificate *only if* the answer to the instance is "yes," because the verifier will accept the certificate only if the certificate gives a subgraph of G that is isomorphic to H. Therefore SUBGRAPH ISOMORPHISM is in NP.

Next we show that CLIQUE polynomially transforms to SUBGRAPH ISOMORPHISM. Let (Γ, k) be an instance of CLIQUE; that is, let $\Gamma = (V, E)$ be a simple undirected graph and let k be a positive integer. (The question is whether Γ contains a clique of size k.) Our goal is to construct an instance (G, H) of SUBGRAPH ISOMORPHISM such that G contains a subgraph isomorphic to H if and only if Γ contains a clique of size k.

First of all, if $k > |V|$, then the answer to the CLIQUE instance is obviously "no," so we can take G to be a graph with exactly one vertex and H to be a graph with two vertices, for example; then G obviously does not contain a subgraph isomorphic to H. So we may as well assume that $k \le |V|$.

Take $G = \Gamma$ and $H = K_k$, the complete graph on k vertices; so $H = (V_H, E_H)$ where $V_H = \{1, 2, \ldots, k\}$ and $E_H = \big\{ \{u, v\} : u, v \in V_H,\ u \neq v \big\}$. This construction can be done in $O(|V|^2)$ time, because the number of edges in E_H is $k(k-1)/2 = O(|V|^2)$. So the running time of this construction is polynomial in the size of the CLIQUE instance.

Suppose that G contains a subgraph isomorphic to H. Then there is a subgraph $K = (V_K, E_K)$ with $V_K \subseteq V_G$ and $E_K \subseteq E_G$ such that there exists a bijection $f : V_H \to V_K$ with $\{u, v\} \in E_H$ if and only if $\{f(u), f(v)\} \in E_K$. But $\{u, v\} \in E_H$ for every pair of distinct vertices u and v in V_H, so every two vertices in V_K are adjacent, which means that V_K is a clique in G. Since f is a bijection, we have $|V_K| = |V_H| = k$. So $G = \Gamma$ contains a clique of size k.

Conversely, suppose $\Gamma = G$ contains a clique $Q \subseteq V$ of size k. Take $V_K = Q$ and $E_K = \big\{ \{u, v\} : u, v \in Q,\ u \neq v \big\}$. Since $|Q| = k$, there exists a bijection $f : \{1, 2, \ldots, k\} \to Q$. Now $\{u, v\} \in E_H$ for every pair of distinct vertices u and v in $V_H = \{1, 2, \ldots, k\}$, and every two vertices in Q are adjacent, so G contains a subgraph $K = (V_K, E_K)$ isomorphic to H.

Thus G contains a subgraph isomorphic to H if and only if Γ contains a clique of size k, so this is a polynomial transformation from CLIQUE to SUBGRAPH ISOMORPHISM. We proved in class (July 7; Theorem 15.3 in Papadimitriou and Steiglitz) that CLIQUE is NP-complete, so SUBGRAPH ISOMORPHISM is also NP-complete. □

[Two other possible approaches are to show a polynomial transformation from HAMILTONIAN CIRCUIT or HAMILTONIAN PATH to SUBGRAPH ISOMORPHISM by taking H to be a cycle with $|V|$ vertices or a path with $|V|$ vertices, respectively. Note that taking H to be the graph with k vertices and no edges does *not* give a polynomial transformation from INDEPENDENT SET to SUBGRAPH ISOMORPHISM, because *every* graph G with at least k vertices contains a subgraph isomorphic to this H, whether or not G has an independent set of size k.]