**CS 344: Design and Analysis of Computer Algorithms** **Rutgers: Fall 2019**

# "Homework" #6 Solutions

December 13, 2019

**Problem 1.** You are given an $n \times n$ matrix and a set of $k$ cells $(i_1, j_1), \ldots, (i_k, j_k)$ on this matrix. We say that this set of cells can *escape* the matrix if: (1) we can find a path from each cell to any arbitrary *boundary cell* of the matrix (a path is a sequence of *neighboring* cells, namely, top, bottom, left, and right), (2) these paths are all *disjoint*, namely, no cell is used in more than one of these paths. See Figure 1 for an example.

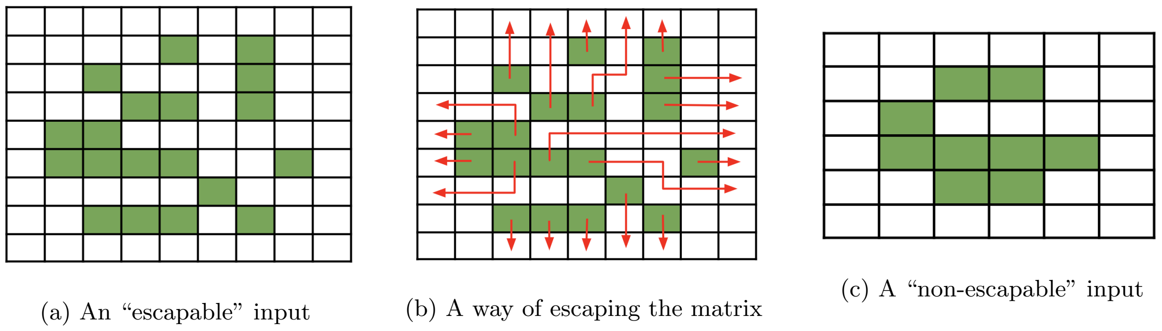
(a) An "escapable" input (b) A way of escaping the matrix (c) A "non-escapable" input

Figure 1: The green cells correspond to the input $k$ cells of the matrix.

Design an $O(n^3)$ time algorithm that given the matrix and the input cells, determines whether these cells can escape the matrix (together) or not[1]. **(25 points)**

*Hint:* Whenever you want to find a collection of paths with "bounded intersection" (totally disjoint in this problem), try network flow first.

**Solution.** *Algorithm (reduction to maximum flow problem):*

1. Create the following flow network $G(V, E)$:
   - For every cell in the matrix, add a vertex to the graph $G(V, E)$ and connect that vertex to all neighboring vertex-cells by bi-directed edges of capacity onee.
   - Add a new vertex $s$ and connect it to the $k$ vertex-cells $(i_1, j_1), \ldots, (i_k, j_k)$ with edges of capacity one. Moreover, create a new vertex $t$ and connect all boundary vertex-cells to $t$ with edges of capacity one.
   - Assign a capacity of one to all *vertices* of this network other than $s, t$ (by using the technique of splitting each vertex into two, explained both in previous homework and lecture note 23).
2. Find the value of maximum $s$-$t$ flow in this network and output the cells are escapable if and only if the max-flow value is $k$.

*Proof of Correctness:* By assigning capacity of one to all vertices and connecting $s$ to $k$ cells that needs to escape and connecting all boundary vertices to $t$, computing max-flow in this network is equivalent to finding maximum vertex disjoint paths from $k$ cells to any boundary vertices. We prove that number of vertex-disjoint paths (i.e., value of flow) is $k$ if and only if the cells are escapable.

[1]This (type of) problem is used typically to connect multiple components of a circuit to the power sources that can only be placed at the boundary of the circuit – disjointness of the paths ensures that the components of the circuit do not interfere with each other.

(i) If the number of vertex disjoint paths (i.e., value of max-flow) is $k$ then the cells are escapable: We can simply use the paths find by the algorithm to escape the cells since the paths are vertex disjoint and thus no cell will be used more than once. Moreover, since $s$ is connected to $k$ sources with edges of capacity one, the only way to have flow of $k$ is to escape all the $k$ input cells.

(ii) If the cells are escapable then the number of vertex disjoint paths (i.e., value of max-flow) is $k$: We define the vertex-disjoint paths (or the flow function) by picking each path that is used to escape the cells (namely, send one unit of flow over that path); since all $k$ cells can escape this allows for a flow of size $k$ which is feasible because no vertex will be shared in the paths. Also max-flow in this network is at most $k$ since edges incident on $s$ have capacity $k$ in total and we clearly cannot send more flow.

*Runtime Analysis:* Number of edges in this network is $O(n^2)$ and the value of max-flow is at most $k$, hence using Ford-Fulkerson algorithm for max-flow will give a runtime of $O(n^2 \cdot k)$. Since $k = O(n^2)$ this runtime is $O(n^4)$ and *not* $O(n^3)$ asked in the question.

To get the runtime of $O(n^3)$ asked in this question we need to use a better max-flow algorithm (tailored to vertex-disjoint paths) which can indeed achieves $O(n^3)$ time for this problem. However, considering we have not covered algorithms other than Ford-Fulkerson for max-flow, this question should have only asked for an $O(n^4)$ runtime *not* $O(n^3)$ and you may treat the $O(n^3)$ runtime in the problem statement as a *typo*[2].

---

**Problem 2.** A *path cover* of a directed graph $G(V, E)$ is a collection of directed paths $P_1, \ldots, P_k$ such that every vertex of $v$ appears in *exactly* one of these paths. Our goal in this problem is to study algorithms for finding a *minimum* path cover (a path cover of minimum size).

(a) Design an algorithm that finds a path cover of a given *directed acyclic graph (DAG)* in $O(mn)$ time.

**(12.5 points)**

*Hint:* Consider the bipartite graph obtained by making two copies of each vertex $v$ called $v^l$ and $v^r$, and adding an edge $\{u^l, v^r\}$ to this undirected graph whenever $(u, v)$ is an edge in the original graph. Prove that the *maximum matching* size in this graph is $n - k$ if and only if the size of the minimum path cover is $k$, and use the algorithm for maximum matching (by network flow) to solve this problem.

**Solution.** *Algorithm (reduction to the maximum bipartite matching problem):* We do exactly as suggested in the hint by creating the given bipartite graph and computing a maximum matching in it.

*Proof of Correctness:* Let $H$ denote the bipartite graph. We prove that $G$ has a path cover of size $k$ if and only if there is a matching of size $n-k$ in $H$. This will then imply the correctness of the algorithm as minimizing path cover corresponds to maximizing the matching size. Like any reduction, we need to prove the following two parts:

(i) If $G$ has a path cover of size $k$, then $H$ has a matching of size $n - k$: Let $P_1, \ldots, P_k$ be the path cover of $G$ and let $P = \cup_{i=1}^{k} P_i$ be the set of edges used in the path cover. Every vertex of the graph $G$ has in-degree 0 or 1 in $P$, and each path only has one vertex of in-degree 0. This means that the number of edges in $P$ is exactly $n - k$. Now consider the set of edges $M$ in $H$ obtained by connecting $v^l$ to $u^r$ whenever $(v, u)$ is an edge in $P$. We claim that $M$ is a matching because in-degree and out-degree of each vertex in $P$ is at most one and hence each vertex in $M$ can have degree at most one as well, making $M$ a matching. So, there is a matching of size $n - k$ in $H$.

(ii) If $H$ has a matching of size $n - k$, then $G$ has a path cover of size $k$: Let $M$ be a matching in $H$ and define the set of edges $P$ in $G$ by adding any edge $(v, u)$ to $P$ whenever $\{v^l, u^r\}$ belongs to $M$. By definition of a matching, each vertex in $P$ will have in-degree and out-degree at most

[2]If you are interested, the following approach solves this problem in $O(n^3)$ time (but if you are not interested, do not worry about this since this is completely out of scope of our course and you can completely ignore this!). Instead of using Ford-Fulkerson algorithm, one should run Hopcroft-Karp algorithm for bipartite matching which can also solve vertex-disjoint path problem on arbitrary graphs (with some modification) in $O(m\sqrt{n})$ time where $m$ is number of edges and $n$ is number of vertices of the graph – considering our graph has $O(n^2)$ edges and $O(n^2)$ vertices, this gives the advertised runtime of $O(n^3)$.

one. As such, $P$ forms a collection of disjoint (directed) paths in $G$ (since $G$ is a DAG, such a collection of edges cannot have any directed cycle inside it). Moreover, since number of edges in $M$ (and hence $P$) is $n - k$, there can only be $k$ vertices in the *entire* graph $G$ such that their in-degree is 0 in $P$ and hence $P$ forms $k$ disjoint paths (note that it is possible we have singleton vertices that are not matched by $M$ and hence we should add those vertices as singleton paths to the path cover as well but the total number of singleton paths and "normal" paths is $k$ by the above argument). This means that we found a path cover of size $k$ for $G$.

*Runtime Analysis:* We created a graph with $O(n)$ vertices and $O(m)$ edges and using the network-flow based algorithm for bipartite matching, we can solve this problem in $O(mn)$ time.

**Remark:** This part of the problem is somewhat on the harder side when it comes to applications of max-flow and hence do not get discouraged if you could not solve it (in particular, your final exam will not have a network flow problem with this level of difficulty as can be seen from the practice exam).

---

(b) Prove that this problem is NP-hard in general, i.e., when $G$ is no longer a DAG but any arbitrary directed graph. In order to do so, use a reduction from the following NP-hard problem:

- **Hamiltonian Path Problem**: Given a directed graph $G(V, E)$ is there a path in $G$ that passes through every vertex, namely, a Hamiltonian path?

**(12.5 points)**

**Solution.** *Reduction (from Hamiltonian Path problem):*

(a) Suppose we have an algorithm $A$ for the path cover problem on arbitrary graphs. We use this to solve the Hamiltonian path problem on any given graph $G(V, E)$.

(b) Run algorithm $A$ on $G(V, E)$ to determine the size of the path cover. Output that $G$ has a Hamiltonian path if and only if size of path cover is one.

*Proof of Correctness:* We prove that $G$ has a Hamiltonian path if and only if it has a path cover of size one (the proof is very basic!).

(i) If $G$ has a Hamiltonian path, then it has a path cover of size one: Simply let the Hamiltonian path be the a single-path path cover. Since every vertex appears in a Hamiltonian path, we obtain a path cover consisting of a single path.

(ii) If $G$ has a path cover of size one, then it has a Hamiltonian path: Simply let the single path in the path cover to be the Hamiltonian path. Since path cover has a single path, this path should go through all vertices and hence is a Hamiltonian path.

*Runtime Analysis:* The reduction simply involves running $A$ on the original graph $G$ and hence if we have a poly-time algorithm $A$ we obtain a poly-time algorithm for Hamiltonian path also. But since Hamiltonian path is an NP-hard problem having a poly-time algorithm for it implies that P = NP. As such, having a poly-time algorithm for path cover problem also implies P = NP, implying that this problem is also NP-hard.

**Remark:** Unlike the previous part, this part of the problem is a very easy reduction and something at this level can easily appear on your final exam as well.

---

**Problem 3.** Recall that in the class, we focused on *decision* problems when defining NP. Solving a decision problem simply tells us whether a solution to our problem exists or not but it does not provide that solution when it exists. Concretely, let us consider the 3-SAT problem on an input formula $\Phi$. Solving 3-SAT on $\Phi$ would tell us whether $\Phi$ is satisfiable or not but will not give us a satisfying assignment when $\Phi$ is satisfiable. What if our goal is to actually find the satisfying formula when one exists? This is called a *search* problem.

It is easy to see that a search problem can only be "harder" than its decision variant, or in other words, if we have an algorithm for the search problem we will obtain an algorithm for the decision problem as well. Interestingly, the converse of this is also true for all NP problems and we will prove this in the context of the 3-SAT problem in this problem. In particular, we reduce the 3-SAT-SEARCH problem (the problem of finding a satisfying assignment to a 3-CNF formula) to the 3-SAT (decision) problem (the problem of deciding whether a 3-CNF formula has a satisfying assignment or not).

(a) Suppose you are given, as a black-box, an algorithm $A$ for solving 3-SAT (decision) problem that runs in polynomial time. Design a poly-time algorithm that given a 3-CNF formula $\Phi$ and a *single* variable $a$ in $\Phi$, decides whether there exists a satisfying assignment of $\Phi$ in which $a$ =True or not (note that this is still a decision problem). **(10 points)**

**Solution.** *Algorithm:*

(1) Set the variable $a =$ True in $\Phi$, remove any clause that is already satisfied by setting $a =$ True, and remove the literal $\neg a$ from any clause that contains it to get formula $\Phi'$.

(2) Run algorithm $A$ on the resulting formula $\Phi'$ (which is still a 3-CNF formula since size of no clause was increased) and output Yes if and only if the new formula is satisfiable.

*Proof of Correctness:* After setting $a$ =True, the algorithm removes any clause from $\Phi$ which is satisfied by this assignment. Also any clause which contains $\neg a$ and hence is not yet satisfied is not going to have variable $a$ any more. Hence, the formula $\Phi'$ does not have variable $a$. As such, satisfying $\Phi'$ is equivalent to finding a satisfying assignment in $\Phi$ in which $a$ is set to be True. In other words, if $\Phi'$ is satisfiable, we can pick the satisfying assignment of $\Phi'$ and append it with $a$ =True to get a satisfying assignment of $\Phi$ and if $\Phi$ is satisfiable when $a$ is set to True, we can pick the assignment of the remaining variables other than $a$ and satisfy $\Phi'$. This implies the correctness of the algorithm.

*Runtime Analysis:* Creating the new formula takes linear time by simply reading the formula once and updating it accordingly. Hence, if $A$ runs in poly-time, the entire algorithm runs in poly-time as well.

---

(b) Use your algorithm from the first part to find a satisfying assignment of a given 3-CNF formula $\Phi$ or output that no such assignment exists, i.e., solve the 3-SAT-SEARCH problem. Your algorithm should run in polynomial time assuming that you are given a poly-time algorithm $A$ for solving 3-SAT decision problem. **(15 points)**

*Hint:* Iterate over variables of $\Phi$ one by one and use part (a) to decide whether they should be assigned True or False iteratively.

**Solution.** *Algorithm:*

(1) Run $A$ on the entire formula $\Phi$ and if $A$ declares $\Phi$ does not have a satisfying assignment, output the same and terminate.

(2) Pick an arbitrary ordering of variables $a_1, \ldots, a_n$. For $i = 1$ to $n$:

   i. Run the algorithm in part (a) on the formula $\Phi$ with variable $a_i =$ True. If it says there is a satisfying assignment for $\Phi$ in which $a_i =$ True, set $a_i =$ True and otherwise set $a_i =$ False.

   ii. *Update* $\Phi$ by removing any clause that got true by the assignment of $a_i$ and removing variable $a_i$ or $\neg a_i$ from any remaining clause (that did not get true by this assignment).

(3) Return the assignment found for $a_1, \ldots, a_n$ as a satisfying assignment of $\Phi$.

*Proof of Correctness:* Firstly, if $\Phi$ is not satisfiable the first line of the algorithm correctly declares it. Now, in every step, if there is a satisfying assignment in which $a_i =$ True, the algorithm decides it correctly by part (a) and set $a_i$ =True and update the remaining clauses – this ensures that by finding a satisfying assignment of the remaining clauses (which we know are satisfiable by part (a)), we obtain a satisfying assignment of $\Phi$ as well. On the other hand, if setting $a_i =$ True does not lead to any satisfying assignment for the remainder of $\Phi$, then $a_i$ =False surely gives a satisfying assignment (recall that $\Phi$ has a satisfying assignment) and hence the algorithm picks this assignment correctly. As such, in every step, the choice of assignment for $a_i$ ensures that every removed clause is satisfied *and* there is a way to satisfy the remaining clauses.

At the end of the algorithm, no clause remains in $\Phi$ and hence the assignment found by the algorithm satisfies all clauses, thus solving 3-SAT-SEARCH problem.

*Runtime analysis:* The algorithm takes $O(n)$ iteration of for-loop and in each one runs algorithm in part (a). Hence the total runtime is $O(n)$ more than runtime of $A$ which is still poly-time in total.

---

**Problem 4.** Prove that each of the following problems is NP-hard and for each problem determine whether it is also NP-complete or not.

(a) **7-Degree Spanning Tree:** Given an undirected graph $G(V, E)$, does $G$ contain a spanning tree in which every vertex has a degree of *at most* 7? **(6 points)**

**Solution.** *Reduction (from Undirected s-t Hamiltonian Path):*

(1) Given an instance $G(V, E)$ of the undirected $s$-$t$ Hamiltonian path problem, we create an instance $G'$ of the 7-degree spanning tree problem as follows.

(2) For any vertex $v \in V \setminus \{s, t\}$, add 5 new dummy vertices to $G'$ and connect them to $v$. Add 6 new dummy vertices for each of $s$ and $t$, and connect them to $s$ and $t$.

(3) We run the algorithm for 7-degree spanning tree on $G'$ and output $G$ has such a $s$-$t$ Hamiltonian path if and only if the algorithm outputs $G'$ has a 7-degree spanning tree.

*Proof of correctness.* We show that $G$ has a $s$-$t$ Hamiltonian path if and only if $G'$ has a 7-degree spanning tree.

(i) If $G$ has a $s$-$t$ Hamiltonian path $P$, then $G'$ has a 7-degree spanning tree: The tree in $G'$ where each original vertex is connected to all its dummy vertices and its neighbor(s) in $P$ is a spanning tree of $G'$ where degree of each vertex is at most 7.

(ii) If $G'$ has a spanning tree $T$ in which every vertex has a degree of at most 7, then $G$ has a $s$-$t$ Hamiltonian path: Since the dummy vertices can only be connected to their corresponding original vertex, removing them from $T$ does not change the connectivity of remaining (original) vertices in $T$. Moreover, after removing them, degree of each of $s$ and $t$ is exactly 1 and remaining vertices have degree 2 in this new tree, hence the tree is a hamiltonian path between $s$ and $t$ in $G$.

*Runtime analysis:* This reduction can be implemented in $O(n + m)$ time and hence a poly-time algorithm for 7-Degree spanning tree implies a poly-time algorithm for undirected $s$-$t$ Hamiltonian path which in turn implies P = NP (by definition of $s$-$t$ Hamiltonian path being NP-hard). Thus a poly-time algorithm for 7-Degree spanning tree also implies P = NP, making this problem NP-hard.

*NP-completeness?* Yes. 7-Degree spanning tree belongs to NP because we can have a *verifier* that takes such a spanning tree as a proof and simply check whether degree of all vertices is $\leq 7$ and all edges belong to $G$ and this is indeed a tree, all in polynomial time. Since this problem is both in NP and also NP-hard, it is also NP-complete.

---

(b) 1/2-**Path:** Given an undirected *connected* graph $G(V, E)$, does $G$ contain a path that passes through *more than* half of the vertices in $G$? **(6 points)**

**Solution.** *Reduction (from Undirected s-t Hamiltonian Path):*

(1) Given an instance $G(V, E)$ of the undirected $s$-$t$ Hamiltonian path problem, we create an instance $G'$ of 1/2-Path as follows.

(2) Obtain $G'$ by adding $n + 1$ dummy vertices to $G$ and connecting them to $s$ and adding one additional dummy vertex and connecting it to $t$.

(3) We then run the algorithm for 1/2-Path on $G'$ and output $G$ has $s$-$t$ Hamiltonian path if and only if the algorithm outputs $G'$ has a path passing more than half the vertices.

*Proof of correctness.* We show that $G$ has an $s$-$t$ Hamiltonian path if and only if $G'$ has a path that passes through more than half of the vertices in $G$. Note that the number of vertices in $G'$ is $2(n+1)$, hence any valid path for the 1/2-Path problem in $G'$ has to be of length at least $n+2$.

(i) If $G$ has a $s$-$t$ Hamiltonian path $P$, then $G'$ has a path passing through more than half of vertices: The path $P' = d_s \rightarrow s \rightsquigarrow_P t \rightarrow d_t$ (the notation $s \rightsquigarrow_P t$ is used to show we copy the path $P$ from $s$ to $t$ here), where $d_s$ is one of dummy vertices connected to $s$ and $d_t$ is the dummy vertex connected to $t$, is a path in $G'$ that passes through exactly $n+2$ vertices, meaning that it is a valid answer for the 1/2-Path problem (recall that $G'$ has $2n+2$ vertices).

(ii) If $G'$ has a path that passes through more than half of the vertices, then $G$ has a $s$-$t$ Hamiltonian path: Note that any path of length more than 3 cannot contain two dummy vertices connected to $s$ and hence the maximum length of a path in $G'$ is $n+2$. Moreover, any path of length $n+2$ in $G'$, must contain a dummy vertex connected to $s$, all original vertices ($n$ vertices), and the dummy vertex connected to $t$ and consequently has to start from and end in a dummy vertex. Hence if $G'$ has a path of length $n+2$, then removing the first and last (dummy) vertices results in a Hamiltonian path from $s$ to $t$ in $G$.

*Runtime analysis:* This reduction can be implemented in $O(n+m)$ time and hence a poly-time algorithm for 1/2-Path implies a poly-time algorithm for undirected $s$-$t$ Hamiltonian path which in turn implies P = NP (by definition of $s$-$t$ Hamiltonian path being NP-hard). Thus a poly-time algorithm for 1/2-Path also implies P = NP, making this problem NP-hard.

*NP-completeness?* Yes. 1/2-Path belongs to NP because we can have a *verifier* that takes such a path as a proof and simply check whether this is path that goes through more than half the vertices and all edges of the path belong to the graph, all in polynomial time. Since this problem is both in NP and also NP-hard, it is also NP-complete.

---

(c) **4-Coloring:** Given an undirected graph $G(V, E)$, is there a 4-coloring of $G$? (A 4-coloring is an assignment of colors $\{1, 2, 3, 4\}$ to vertices so that no edge gets the same color on both its endpoints).

**(6 points)**

**Solution.** *Reduction (from 3-Coloring):*

(1) Given an instance $G(V, E)$ of the 3-coloring problem, we create an instance $G'$ of 4-Coloring as follows.

(2) Obtain $G'$ by adding a new vertex $z$ to $G$ and connect $z$ to all vertices in $G$.

(3) We then run the algorithm for 4-Coloring on $G'$ and output $G$ has 3-coloring if and only if the algorithm outputs $G'$ has a 4-coloring.

*Proof of correctness.* We show that $G$ has a 3-coloring if and only if $G'$ has a 4-coloring.

(i) If $G$ has a 3-coloring, then $G'$ has a 4-coloring: Color all vertices inside $G'$ that were originally in $G$ using the 3-coloring of $G$ and color the new vertex $z$ using a completely new color. This is a valid 4-coloring of $G'$ as no edge inside $G$ can have the same color (by definition of 3-coloring) and edges between $z$ and other vertices always have two different colors (since color of $z$ is not used in coloring $G$ at all).

(ii) If $G'$ has a 4-coloring, then $G$ has a 3-coloring: In any 4-coloring of $G'$, vertex $z$ should get a different color than all vertices copied from $G$ since it is connected to all of them. This means that only 3 colors are used to color vertices inside $G'$ copied from $G$ and thus we can rename the colors to become $\{1, 2, 3\}$ and get a 3-coloring of $G$.

*Runtime analysis:* This reduction can be implemented in $O(n+m)$ time and so a poly-time algorithm for 4-Coloring implies a poly-time algorithm for 3-Coloring which in turn implies P = NP (by definition of 3-Coloring being NP-hard). Thus a poly-time algorithm for 4-Coloring implies P = NP, making this problem NP-hard.

*NP-completeness?* Yes. 4-Coloring belongs to NP because of the same exact reason 3-Coloring belongs to NP. Since this problem is both in NP and also NP-hard, it is also NP-complete.

---

(d) **Minimum Vertex Cover:** Given an undirected graph $G(V, E)$, what is the size of minimum vertex cover in $G$? (A vertex cover is a set of vertices such that every edge has at least one end point in the vertex cover). **(7 points)**

**Solution.** *Reduction (from Maximum Independent Set):*

(1) Given an instance $G(V, E)$ of the maximum independent set problem, we simply run the minimum vertex cover algorithm on $G$.

(2) Let $S$ be the vertex cover found by the algorithm; return $V - S$ as a maximum independent set.

*Proof of correctness.* We show that for any graph $G(V, E)$ and any set $S$ of vertices, $S$ is a vertex cover if and only if $V - S$ is an independent set. Since minimizing size of $S$ is equivalent to maximizing size of $V - S$, this ensures that minimum vertex cover is simply complement of maximum independent set, proving correctness of the algorithm.

The claim about $S$ and $V - S$ is true because for $S$ to be a vertex cover, every edge should have an endpoint in $S$ which means there is no edge with both endpoints in $V - S$, which means $V - S$ is an independent set. Similarly whenever $V - S$ is an independent set, $S$ would become a vertex cover as no edge of $G$ is inside $V - S$ and hence all edges are incident on $S$.

*Runtime analysis:* This reduction simply requires running the poly-time algorithm for minimum vertex cover the graph and hence is poly-time itself. As before, this implies minimum vertex cover is NP-hard.

*NP-completeness?* No. Minimum vertex cover does *not* belong to NP because it is not a decision problem and hence it cannot be NP-complete.

---

You may assume the following problems are NP-hard for your reductions:

- **Undirected $s$-$t$ Hamiltonian Path:** Given an undirected graph $G(V, E)$ and two vertices $s, t \in V$, is there a Hamiltonian path from $s$ to $t$ in $G$? (A Hamiltonian path is a path that passes every vertex).
- **Maximum Independent Set:** Given an undirected graph $G(V, E)$, what is the size of maximum independent set in $G$? (An independent set is a collection of vertices with no edges between them).
- **3-Coloring:** Given an undirected graph $G(V, E)$, is there a 3-coloring of $G$? (A 3-coloring is an assignment of colors $\{1, 2, 3\}$ to vertices so that no edge gets the same color on both its endpoints).