EECS 376: Foundations of Computer Science
University of Michigan, Winter 2024

Solutions to Homework 8

This homework has 8 questions, for a total of 100 points and 8 extra-credit points.
Unless otherwise stated, each question requires *clear*, *logically correct*, and *sufficient* justification to convince the reader.
For bonus/extra-credit questions, we will provide very limited guidance in office hours and on Piazza, and we do not guarantee anything about the difficulty of these questions.
We strongly encourage you to typeset your solutions in LaTeX.
If you collaborated with someone, you must state their name(s). You must *write your own solution* for all problems and *may not use any other student's write-up.*

(0 pts) 0. **Before you start; before you submit.**

- Read Handout 3: NP-Hardness Proofs to understand the components of such proofs.
- If applicable, state the name(s) and uniqname(s) of your collaborator(s).

Solution:

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

Carefully read and understand the posted solutions to the previous homework. Identify one part 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.). Copy or screenshot this 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 turn in the previous homework, then (1) state that you didn't turn it in, and (2) pick a problem that you think is particularly challenging from the previous homework, and explain the answer in your own words. You may reference the answer key, but your answer should be in your own words.

Solution:

2. **Review of last week's materials.**

(4 pts) (a) Define the terms P, NP, NP-Hard, and NP-Complete. (You do not need to define the terms that are used in these definitions.)

Solution:

- $\mathsf{P} = \{L : L$ is efficiently decidable (by some polynomial-time Turing Machine)$\}$.
- $\mathsf{NP} = \{L : L$ is efficiently verifiable$\}$.
- "L is NP-Hard" means that $A \leq_p L$ for every $A \in \mathsf{NP}$.
- "L is NP-Complete" means that $L \in \mathsf{NP}$ and L is NP-Hard.

(8 pts) (b) For each of the following statements, state, with brief justification, that it is *known to be true*, *known to be false*, or its truth/falsehood is *unknown*. Note: as always, if a statement has any counterexample(s), it is false.

(i) Let V be an efficient verifier for $L \in \mathsf{NP}$ and x, c be strings; if $V(x, c)$ accepts, then $x \in L$.

(ii) Let V be an efficient verifier for $L \in \mathsf{NP}$ and x, c be strings; if $V(x, c)$ rejects, then $x \notin L$.

(iii) If $\mathsf{P} \neq \mathsf{NP}$ and L is NP-Complete, then $L \notin \mathsf{P}$.

(iv) If L is NP-Hard, then $L \notin \mathsf{P}$.

Solution:

(i) **Known to be true.** This holds immediately by the definition of correctness for a verifier, which says that $x \in L$ if and only if there exists c such that $V(x, c)$ accepts. Since such c exists here, $x \in L$.

(ii) **Known to be false.** In general, "yes" instances can have "unconvincing"/"invalid" certificates that the verifier rejects; all that's required is that *some* convincing certificate exists for each "yes" instance.

A specific counterexample is $L = \text{TSP}$ with the verifier V from lecture and the course notes, $x = (G, k) \in \text{TSP}$ is a "yes" instance (i.e., graph G has a tour within budget k), but c is a "junk" certificate, e.g., with no vertices in it. Then $V(x, c)$ rejects because the certificate is not even a tour, but there is a different certificate c' that does make $V(x, c')$ accept.

(iii) **Known to be true**. By definition of NP-Complete, L is NP-Hard, i.e., every language in NP poly-time mapping reduces to L. So, if $L \in \mathsf{P}$ then every language in NP would be in P, implying that $\mathsf{P} = \mathsf{NP}$. This contradicts the hypothesis that $\mathsf{P} \neq \mathsf{NP}$, so with this hypothesis we must have that $L \notin \mathsf{P}$.

(iv) **Unknown.** If $\mathsf{P} \neq \mathsf{NP}$, then by the reasoning from the previous statement, we necessarily have $L \notin \mathsf{P}$, so the statement would be true.

But if $\mathsf{P} = \mathsf{NP}$, then any NP-hard language $L \in \mathsf{NP} = \mathsf{P}$ (e.g., $L = \text{SAT}$) is a counterexample, making the statement false.

Since it is unknown whether $\mathsf{P} = \mathsf{NP}$, the truth/falsehood of the statement is unknown.

(4 pts) (c) Briefly explain what the "Boolean satisfiability" (SAT) problem is, and give one example instance that is satisfiable, and one that is unsatisfiable. Each example must include at least 2 variables and at least 3 (not necessarily distinct) logical operators (AND/OR/NOT).

Solution: The Boolean satisfiability decision problem (aka SAT) is the problem of determining, given a Boolean formula, whether there exists a true/false assignment to all the formula's variables that make the formula evaluate to true.

An example of a satisfiable Boolean formula is $(A \vee B) \wedge (\neg A \vee B)$, because the assignment $A =$ false and $B =$ true makes the formula true. (There are other assignments that would make the formula true, but we need only one to show that the formula is satisfiable.)

An example of an unsatisfiable Boolean formula is $A \wedge \neg A \wedge B \wedge C$, because both A and its negation $\neg A$ must be true to make the formula true, but this is logically impossible.

(4 pts) (d) Towards proving the Cook-Levin Theorem in lecture, we outlined a proof of the following statement:

> Let language $L \in \mathsf{NP}$ be arbitrary, and fix an efficient verifier $\textsc{VerifyL}$ for L. Given any instance x of L, in polynomial time we can construct an instance ϕ of SAT such that ϕ is satisfiable *if and only if* there *exists* a c such that the tableau of $\textsc{VerifyL}(x, c)$ has q_{accept} in it.

Briefly explain why proving this statement proves that SAT is NP-Hard.

Solution: By definition of verifier, there exists such a c if and only if $x \in L$. Then by definition, the construction is a poly-time mapping reduction from L to SAT, so $L \leq_p$ SAT for all $L \in \mathsf{NP}$, so SAT is NP-hard by definition.

(10 pts) 3. **Turing reductions vs. poly-time mapping reductions.**

Suppose that $\mathsf{P} \neq \mathsf{NP}$, and let $A \in \mathsf{P}$ be arbitrary. Prove that there is a *Turing* reduction from SAT to A (i.e., SAT $\leq_T A$), but there is *no polynomial-time mapping reduction* from SAT to A. (i.e., SAT $\not\leq_p A$).

Solution: We first prove that SAT $\leq_T A$, by showing that SAT is decidable. A formula φ of size n has at most n variables, so there are at most 2^n possible assignments to its variables. A brute-force algorithm decides SAT simply by iterating over all of the assignments of its input formula, accepting if any of them satisfies the formula, and rejecting otherwise. (Although this is not *efficient*, it does *decide* SAT.) As we showed in a previous homework, any decidable language (here, SAT) Turing-reduces to any other language (here, A); recall that the reduction does not even use its oracle. So, SAT $\leq_T A$.

We now prove that there is no polynomial-time mapping reduction from SAT to A, under the hypothesis that $\mathsf{P} \neq \mathsf{NP}$. For the purposes of contradiction, if SAT $\leq_p A$, then because $A \in \mathsf{P}$, we would have SAT $\in \mathsf{P}$. This is turn would imply that $\mathsf{P} = \mathsf{NP}$, because SAT is NP-Hard. But this contradicts the question's hypothesis that $\mathsf{P} \neq \mathsf{NP}$.

4. **Subset sum.**

Consider the following reduction involving 3SAT and the language

$$\text{SUBSETSUM} = \{(A, s) : A \text{ is a } \textit{multiset} \text{ of integers} \geq 0, \text{ and } \exists\, I \subseteq A \text{ such that } \sum_{a \in I} a = s\}.$$

A *multiset* A is just like a set, but it can have duplicate elements. A submultiset $I \subseteq A$ also can have duplicate elements, as long as it does not have more copies of an particular element than A does.[1]

The reduction is given a 3CNF formula φ with n variables $x_1, \ldots, x_n$ and m clauses $c_1, \ldots, c_m$. We assume without loss of generality that there is no repeated clause. (Recall that each clause is the OR of exactly three literals, where a literal is either some variable x_i or its negation $\overline{x_i}$.)

The reduction constructs a collection of numbers as follows:

1. For each clause c_j, it constructs integers a_j and b_j, each $n + m$ decimal digits long, where the jth digit of both a_j and b_j is 1, and all other digits are 0.
2. For each variable x_i, it constructs integers t_i and f_i, each $n+m$ decimal digits long, where:
   (a) The $(m + i)$th digits of both t_i and f_i are 1.
   (b) The jth digit of t_i is 1 if literal x_i appears in clause c_j.
   (c) The jth digit of f_i is 1 if literal $\overline{x_i}$ appears in clause c_j.
   (d) All other digits of t_i and f_i are 0.
3. It constructs a target sum s that has $n + m$ decimal digits, where the first m digits are 3 and the last n digits are 1.

The reduction outputs (A, s), where A is the multiset of all the numbers a_j, b_j, t_i, and f_i.

For example, for the formula $\varphi = (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3})$, the reduction constructs:

Number	$j = 1$	$j = 2$	$i = 1$	$i = 2$	$i = 3$
a_1	1	0	0	0	0
b_1	1	0	0	0	0
a_2	0	1	0	0	0
b_2	0	1	0	0	0
t_1	1	0	1	0	0
f_1	0	1	1	0	0
t_2	0	1	0	1	0
f_2	1	0	0	1	0
t_3	1	0	0	0	1
f_3	0	1	0	0	1
s	3	3	1	1	1

Answer the following:

[1] In class we defined a slightly different version of SUBSETSUM with ordinary sets instead of multisets. With a little more work, the version with ordinary sets can also be proved NP-Complete.

(2 pts) (a) Fill in the blanks: the above reduction is for showing that ________ $\leq_p$ ________.

Solution: 3SAT $\leq_p$ SubsetSum.

(15 pts) (b) Prove that the reduction is correct by proving the following, for any 3CNF formula φ:

- $\varphi \in$ 3SAT $\implies (A, s) \in$ SubsetSum;
- $(A, s) \in$ SubsetSum $\implies \varphi \in$ 3SAT.

Solution: For the first direction, we must show that any *satisfiable* formula φ maps to a multiset A of the numbers t_i, f_i, a_i, b_i and target number s, where some submultiset $I \subseteq A$ sums to s. To do this, we will show that *any* particular satisfying assignment of φ yields a *corresponding* specific subset $I \subseteq A$ that sums to s.

For example, for the specific formula φ given above, the assignment $x_1 = T, x_2 = T, x_3 = F$ satisfies it. A corresponding submultiset consists of t_1, t_2, f_3 and a_1, b_1, a_2, which do indeed sum to 33111.

By assumption, there exists a true/false (T/F) assignment of the variables x_i that satisfies φ. We define

$$v_i = \begin{cases} t_i & \text{if } x_i = T \\ f_i & \text{if } x_i = F. \end{cases}$$

We can see that $s' = \sum_{i=1}^{n} v_i$ will have all its least-significant n digits equal to 1, because we are choosing exactly one of t_i or f_i for each $1 \leq i \leq n$ (and there are no 'carries' among these digits).

Now, observe that because the assignment satisfies φ, under this assignment, each clause has either one, two, or three true literals—but not zero! By construction of the t_i and f_i values, this means that each of the m most-significant digits of s' is either one, two, or three. So, for each $1 \leq j \leq m$ we additionally choose either both, one, or zero (respectively) of a_j and b_j, to make each of our sum's m most-significant digits equal exactly 3. Notice that including a_j and/or b_j increases only the jth most-significant digit, and leaves all the others unchanged (because there are no 'carries'). So, we have shown how any satisfying assignment of φ induces a submultiset of A that sums to s, as desired.

For the other direction, by hypothesis there exists some submultiset I of A that sums to s; we show how this yields a satisfying assignment for the original formula φ. First observe that for each $1 \leq i \leq n$, *exactly one* of f_i or t_i must be in I, because the n least-significant digits of the sum must match those of s, and *no carries are possible* when adding numbers from A. We construct the following truth assignment to the variables of φ:

$$x_i = \begin{cases} T & \text{if } t_i \in I \\ F & \text{if } f_i \in I. \end{cases}$$

While the set I may also contain some of the a_j and b_j numbers, we know that for any $1 \leq j \leq m$ these a_j, b_j can contribute at most 2 to the jth most-significant digit of the sum. So by construction of the numbers, the sum of *just* the t_i and f_i numbers in I must be at least 1 in each of its m most-significant digits.

By the construction of the numbers t_i, f_i, this means that under our assignment, each clause of φ has at least one true literal. Because each clause is an OR of three literals, each clause is true, hence the entire formula is satisfied under our assignment, as desired.

(5 pts) (c) Why do we set the first m digits of s to be 3? Would the reduction still be correct if we used 2 instead? What about 4?

Solution: The reduction would be *incorrect* if we set the first m digits of s to be 2; specifically, the $\Longleftarrow$ direction does not hold. Consider $\varphi = (x_1 \vee x_1 \vee x_1) \wedge (\overline{x_1} \vee \overline{x_1} \vee \overline{x_1})$. Clearly there is no satisfying assignment, but we can choose a_1, a_2, b_2, t_1 and get a sum of 221. The problem is that the "padding" numbers a_j, b_j can be used to match the 2 digits, without any of the literals of the clauses being true.

Similarly, the reduction is incorrect if we set the first m digits of s to be 4; specifically, the $\Longrightarrow$ direction does not hold. Consider $\varphi = (x_1 \vee x_1 \vee \overline{x_1}) \wedge (\overline{x_1} \vee \overline{x_1} \vee x_1)$. Clearly there is a satisfying assignment with $x_1 = T$, but there is no submultiset of A that sums to s, because we cannot make the most-significant digits of any subset-sum equal 4.

(18 pts) 5. **Tours and cycles.**

Recall the following definitions (in this problem, all graphs are undirected):

$$\textsc{HamCycle} = \{G : G \text{ is an unweighted graph with a Hamiltonian cycle}\}$$
$$\text{TSP} = \{(G, k) : G \text{ is a weighted, complete graph with a tour of weight} \le k\} .$$

Prove that $\textsc{HamCycle} \le_p \text{TSP}$. (Recall from lecture and the handout what such a proof involves.) Since $\textsc{HamCycle}$ is NP-Hard, is follows that TSP is also NP-Hard.

Reminder: an instance of $\textsc{HamCycle}$ is an arbitrary unweighted graph, while an instance of TSP is a *complete* graph where each edge has a weight.

Solution: We construct a function f that takes a graph $G = (V, E)$ as input and outputs a graph-integer pair $(G', k = |V|)$ such that $G \in \textsc{HamCycle} \iff f(G) = (G', |V|) \in \text{TSP}$. In other words, f maps "yes" instances of $\textsc{HamCycle}$ to "yes" instances of TSP, and "no" instances of $\textsc{HamCycle}$ to "no" instances of TSP.

The key idea is to generate a weighted complete graph G' on the *same vertices* as G, where each edge has weight 1 if the edge is in G, and has weight 2 (or any larger weight) if the edge is not in G. The TSP budget is set to be the number of vertices in the graph. Then if the input has a Hamiltonian cycle, the same cycle works as a TSP tour that is within the budget. Conversely, any TSP tour that is within the budget must have all edge weights equal to 1, and hence is a Hamiltonian cycle in the original graph.

```
1: function $f(G = (V, E))$
2:     Define $E^* := \{(u, v) : u, v \in V\}$. Note that $E \subseteq E^*$.
3:     for all $e \in E^*$, assign edge weights in $G'$ as follows: do
4:         if $e \in E$ then
5:             $wt(e) := 1$
6:         else
7:             $wt(e) := 2$
8:     return $(G' = (V, E^*), k = |V|)$
```

Correctness analysis: We prove the correctness of this reduction. In both parts of this analysis, let $G = (V, E)$ and $(G' = (V, E^*), k = |V|) = f(G)$.

Suppose that $G \in$ HAMCYCLE. Then by definition, there is a Hamiltonian cycle in G, which by definition has exactly $|V|$ edges. Each edge in this cycle is assigned a weight of 1 in the constructed graph G', so G' has a TSP tour of weight exactly $|V|$. Thus, $f(G) = (G', k = |V|) \in$ TSP.

Conversely, suppose that $f(G) \in$ TSP. Then by definition, the graph G' has a TSP tour of weight *at most* $k = |V|$. Since all edges in G' have weight at least 1, and all TSP tours in G' must have exactly $|V|$ edges, any TSP tour in G' must have a total weight of at least $|V|$. Consequently, G' has a tour of weight *exactly* $|V|$, and all edges in the tour have weight 1, hence all of them are in the original edge set E. Since any TSP tour is a Hamiltonian cycle, we have that $G \in$ HAMCYCLE.

(There is a slight ambiguity in the final claim above, depending on the precise definition of TSP "tour." In some definitions it must be a Hamiltonian cycle, so no edge can be reused. In other definitions it just needs to visit every vertex exactly once and then return to the starting vertex. These definitions are equivalent, except on the graph consisting of just two vertices u, v with an edge (u, v) between them: $u \to v \to u$ is a tour under the latter definition, but not under the former one. With the latter definition, since the graph has a TSP tour but not a Hamiltonian cycle, we can tweak the reduction to treat this one input graph as a special case and output some fixed "no" instance of TSP on it.)

Efficiency analysis: we argue that the reduction runs in polynomial time. The first step of the reduction creates edges by finding every pair of vertices, which is $O(|V|^2)$. The next step creates a graph using these edges, and so is $O(|V| + |V|^2) = O(|V|^2)$. The next step iterates through the edges, and so is $O(|V|^2)$. The next step simply returns the new graph and number of vertices, and so is $O(|V|^2 + |V|) = O(|V|^2)$. Every step of the reduction is polynomial time in the size of the input graph $G = (V, E)$, so the reduction as a whole runs in polynomial time. Altogether, we have shown that HAMCYCLE $\leq_p$ TSP.

6. **More Knapsack!**

Recall the 0-1 knapsack problem: an instance is a list of n item weights $W = (W_1, W_2, \dots, W_n)$, their corresponding values $V = (V_1, V_2, \dots, V_n)$, and a weight capacity C. (All values are non-negative integers.) The goal is to select items having maximum total value, such that their total weight does not exceed the capacity. We can make this a decision problem by introducing a "budget" K and asking whether a total value of at least K can be achieved (again, subject to the capacity constraint):

$$\textsc{Knapsack} = \{(W, V, C, K) : \exists\, S \subseteq \{1, \dots, n\} \text{ such that } \sum_{i \in S} W_i \le C \text{ and } \sum_{i \in S} V_i \ge K\}.$$

(14 pts) (a) Prove that Knapsack is NP-Hard, by showing that SubsetSum $\le_p$ Knapsack. (See Question 4 for the definition of SubsetSum.)

Solution: To prove that SubsetSum $\le_p$ Knapsack, we exhibit an efficiently computable function f such that

$$(A, s) \in \textsc{SubsetSum} \iff f(A, s) \in \textsc{Knapsack}.$$

Simply define $f(A, s) = (W = A, V = A, C = s, K = s)$. (To be precise, A is a multiset of numbers, which we can list in arbitrary order in an array, using the same order for both W and V.) In words: given a subset-sum instance with integers A and target sum s, we map it to a knapsack instance with an item for each number in A, whose weight and value both equal that number; moreover, the knapsack capacity C and target value K both equal the target sum s.

We show that f has the required properties. It is clearly efficiently computable. For the $\implies$ direction, if some subset of A sums to s, then the corresponding collection of knapsack items has total weight $s \le C$ (the knapsack capacity), and has total value $s \ge K$ (the target value); in fact, both of these are equalities. For the $\impliedby$ direction, if some subset of the knapsack items has total weight $\le C = s$ and total value $\ge K = s$, then because each item's weight equals its value, the total weight and value are both *exactly* s. This means that the corresponding collection of numbers in A sums to s, as required.

(6 pts) (b) Recall that we previously gave a dynamic programming algorithm that solves Knapsack in $O(nC)$ time. Does this prove that $\mathsf{P} = \mathsf{NP}$? Why or why not?

Solution: Despite it running in time $O(nC)$, the DP algorithm is actually *not* efficient. Recall that C is an integer, and it is represented in binary using some t digits, possibly $t = n$ (the number of items) or more, whereas the rest of the instance W, V, K might take only $O(n^2)$ or even $O(n)$ bits to represent. So the value of C can be 2^n or more, meaning that the DP algorithm's running time is *exponential* in the input length. So, this does not show that $\mathsf{P} = \mathsf{NP}$. (Nor does it show that $\mathsf{P} \ne \mathsf{NP}$. We have only shown that a *particular* algorithm for SubsetSum takes exponential time; we have not ruled out the possibility of a different algorithm that runs in polynomial time.)

(8 EC pts) 7. **Optional extra-credit question: Network reliability.**

The Republic of Alvonia owns the internal internet infrastructure for the country and leases out connections to Internet Service Providers (ISPs). We represent that network as an undirected graph. Assume that each edge in the graph has a positive integer *rental cost* associated with it. An ISP is confronted with the problem of spending the minimum amount of money on link rental so that it can provide adequately reliable service to its customers.

Here is how we quantify reliability: We say that two paths in the network are *disjoint* if they have no vertices in common, except for possibly their endpoints. For example, there can be two disjoint paths from vertex v_{42} to v_{100}, but v_{42} and v_{100} can be the only vertices that these paths have in common (since these vertices are the endpoints of the paths). In the interest of reliability, it is desirable to have multiple disjoint paths between pairs of nodes in the network. Some ISPs provide more reliability than others, but they may charge their clients more.

The Network Reliability Optimization Problem (NROP) is now defined as follows. An instance is an undirected graph with n vertices $v_1, \dots, v_n$, a non-negative integer weight on each edge, and an n-by-n symmetric matrix R_{ij}. The objective is to find a subset S of the edges such that the total cost of the edges in S is minimized, with the requirement that for every pair of vertices v_i and v_j, there are at least R_{ij} disjoint paths from v_i to v_j such that all paths use only edges in S.

You've been hired as a summer intern to develop an efficient algorithm for solving NROP. After working on an algorithm for awhile, your team conjectures that the problem is NP-hard. Here is your task:

1. Define the decision version of this problem, which we will call NRDP;
2. Prove that NRDP is NP-hard via a reduction from an NP-hard problem from lecture.

Solution: The decision version of the problem is as follows. Given a weighted undirected graph G on n vertices, an n-by-n matrix R, and a budget C, does there exist a subset of edges of total cost at most C such that every pair of vertices v_i, v_j have at least $R_{i,j}$ disjoint pairs between them using the given set of edges?

(Although this is not required for this problem, note that the problem is in NP because we can verify a claimed solution as follows. The form of the certificate is a set of edges that make up a claimed solution, *and* a collection of paths. We then check that the edges are indeed in the graph, that their total cost doesn't exceed the budget C, and that the certificate includes R_{ij} disjoint paths for each pair of vertices (i, j). Notice that the number of vertex disjoint paths between any two vertices cannot exceed n, so this can be verified in polynomial time.)

The reduction f is from HamCycle. Given an undirected graph G, it constructs an instance of NROP as follows: the graph is G with edge weights 1, the matrix has 2 in all its non-diagonal entries and 0 in its diagonal entries, and the budget is n, the number of vertices in the graph.

This reduction clearly takes time polynomial in the size of the input G. To see that $G \in \text{HamCycle} \implies f(G) \in \text{NROP}$, observe that if there is a Hamiltonian cycle in G, then the edges in such a cycle together cost exactly $C = n$, and for every pair of (distinct) vertices v_i, v_j, there are exactly two disjoint paths that use these edges, that go in the two "directions" around the cycle. (Notice that disjointness relies on the fact that the cycle is Hamiltonian.)

We now show that $f(G) \in \text{NROP} \implies G \in \textsc{HamCycle}$. By hypothesis, there is a set of n edges such that using these n edges, there are at least two disjoint paths between every pair of vertices. We show that these n edges must form a Hamiltonian cycle.

First, observe that in the subgraph H induced by those n edges, every vertex must have degree at least two, since each vertex has at least two disjoint paths to every other vertex. Since the sum of the degrees in a graph equals twice the number of edges, the degree sum is exactly $2n$. If any vertex has degree more than two, then some other vertex has degree less than two, which contradicts the fact that every vertex has degree at least two; therefore, each vertex in H has degree *exactly* two. Thus, the set of selected edges form one or more cycles, and every vertex is on one such cycle. Since each vertex can reach every other vertex, the graph H is connected, and therefore it consists of a *single* cycle. Since the vertices of H are all the vertices of G, we conclude that G has a Hamiltonian cycle, as claimed.