Gradiance Online Accelerated Learning

Zayd

- Home Page
- Assignments Due
- Progress Report
- Handouts
- Tutorials
- Homeworks
- Lab Projects
- Log Out

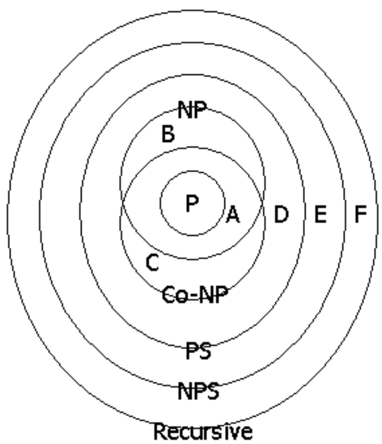
Help

Submission number:	84191
Submission certificate:	EF868552
Submission time:	2014-05-09 03:06:23 PST (GMT - 8:00)

Number of questions:	6
Positive points per question:	3.0
Negative points per question:	1.0
Your score:	10

Questions about languages classes NP and above, based on Sections 10.1, 11.1, 11.2, and 11.3 of HMU.

1. In the diagram below we see certain complexity classes (represented as circles or ovals) and certain regions labeled A through F that represent the differences of some of these complexity classes.



The state of our knowledge regarding the existence of problems in the regions A-F is imperfect. In some cases, we know that a region is nonempty, and in other cases we know that it is empty. Moreover, if P=NP, then we would know more about the emptiness or nonemptiness of some of these regions, but still would not know everything.

Decide what we know about the regions A-F currently, and also what we would know if P=NP. Then, identify the true statement from the list below.

a) Region A is definitely not empty.

b) If P=NP, it would still not be known whether region B is empty.

c) Region C is definitely empty.

d) Region F is definitely not empty.

Answer submitted: **b)**

Your answer is incorrect.

If it happens that P=NP, what could we conclude about region B? See Section 11.1 (p. 484).

Question Explanation:

Since we do not know whether P=NP, or whether NP=co-NP (i.e., whether NP is closed under complementation), we do not know whether any of A, B, or C is empty. If we know P=NP, then surely A and B are empty. But if P=NP, then co-NP=NP=P, since P is closed under complementation. Thus, C would be empty as well.

Savitch's theorem tells us that PS=NPS; i.e., region E is empty. We also know that there are arbitrarily complex recursive languages, so region F is definitely *not* empty. Finally, we do not know about region D, since it is open whether PS=NP or even PS=P. And even if we knew P=NP, we would still not be sure whether or not PS=NP=P.

The correct choice is: **d)**

---

2. Let us denote a problem X as NP-Easy if it is polynomial-time reducible to some problem Y that is in NP. Let us denote as NP-Equivalent, the class of problems that are both NP-Easy and NP-Hard. Let A,B,C,D and E be problems such that A is NP-Hard, B is NP-Complete, C is NP-Equivalent, D is NP-Easy and E is in NP. Which of the following statements is TRUE?

a) If P=NP then D is in P.

b) If B is in P then so is A.

c) If P=NP then A is in P.

d) If A is in P then E may or may not be in P

Answer submitted: **a)**

You have answered the question correctly.

Question Explanation:

The question is based on definitions of NP, NP-hardness, NP-completeness, and Theorems 10.4 and 10.5 in Section 10.1.6, p. 434--435. Some key facts we are using include:

1. If a problem A reduces to a problem B, then A is at least as hard as B.
2. Since A is NP-Hard, for every language L in NP, there is a polynomial-time reduction from L to A.
3. Since C is NP-Easy, it implies that C is reducible in polynomial time to another problem F in NP. F is reducible in polynomial time to A. So C is polynomial-time reducible to A. A similar reasoning holds for D.
4. Since A is NP-Hard, every problem in NP reduces to A in polynomial time. If A is in P, all such problems R can be solved in polynomial time using the algorithm for reducing R to A combined with the algorithm for A. This would imply that P = NP.
5. We denote a problem X as NP-Easy if it is polynomial-time reducible to some problem Y that is in NP. Then the fact that Y is in NP, combined with the reduction, implies that X is also in NP. Moreover, if X is NP-equivalent, then X is also NP-Hard. Hence the notions of NP-Equivalence and NP-Completeness are the same.

---

3. Consider the following problems:

1. SP (Shortest Paths): given a weighted, undirected graph with nonnegative integer edge weights, given two nodes in that graph, and given an integer limit *k*, determine whether the length of the shortest path between the nodes is *k* or less.

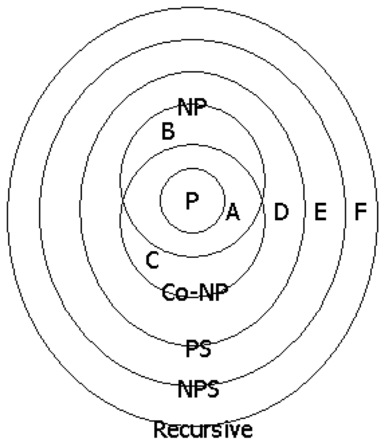
2. WHP (Weighted Hamilton Paths): given a weighted, undirected graph with nonnegative integer edge weights, and given an integer limit *k*, determine whether the length of the shortest Hamilton path in the graph is *k* or less.

3. TAUT (Tautologies): given a propositional boolean formula, determine whether it is true for all possible truth assignments to its variables.

4. QBF (Quantified Boolean Formulas): given a boolean formula with quantifiers for-all and there-exists, such that there are no free variables, determine whether the formula is true.

In the diagram below are seven regions, P and A through F.

Copyright © 2007-2013 Gradiance Corporation.



Place each of the four problems in its correct region, on the assumption that NP is equal to neither P nor co-NP nor PS.

a) Problem QBF is in region F.
b) Problem SP is in region D.
c) Problem WHP is in region A.
d) Problem QBF is in region D.

Answer submitted: **d)**

You have answered the question correctly.

Question Explanation:

SP is in P. You can use Dijkstra's algorithm to solve the problem in quadratic time.

WHP is NP-complete. It is a generalization of Hamilton-path, but the weights don't affect the nondeterministic-polynomial-time "guessing" algorithm to solve it. On the assumption that NP is neither P nor co-NP, WHP must be in region B.

TAUT is essentially the complement of SAT. Since SAT is NP-complete, TAUT must be in region C, unless NP is equal to one of P or co-NP (which we assume not to be the case).

QBF is complete for PS. Thus, assuming PS is not NP, QBF is in D.

---

4. Suppose there are three languages (i.e., problems), of which we know the following:

1. L1 is in P.
2. L2 is NP-complete.
3. L3 is not in NP.

Suppose also that we do not know anything about the resolution of the "P vs. NP" question; for example, we do not know definitely whether P=NP. Classify each of the following languages as (a) Definitely in P, (b) Definitely in NP (but perhaps not in P and perhaps not NP-complete) (c) Definitely NP-complete (d) Definitely not in NP:

- L1 [union] L2.
- L1 ∩ L2.
- L2cL3, where c is a symbol not in the alphabet of L2 or L3 (i.e., the *marked concatenation* of L2 and L3, where there is a unique marker symbol between the strings from L2 and L3).
- The complement of L3.

Based on your analysis, pick the correct, definitely true statement from the list below.

a) The complement of L3 is definitely not NP-complete.
b) The complement of L3 is definitely not in P.
c) L1 ∩ L2 is definitely in P.

d) L1 [union] L2 is definitely NP-complete.

Answer submitted: **d)**

Your answer is incorrect.

Hint: Consider the case where L1 is all strings over the alphabet of L2. What would you then know about how L1 ∪ L2 could be recognized? Is that consistent with L1 ∪ L2 being NP-complete?

Some general observations:

1. Read Section 10.1.5 (p. 433) on polynomial-time reductions and Section 10.1.6 (p. 434) on what it means if a problem is NP-complete.
2. If an NP-complete problem such as L2 polynomially reduces to a problem in P, then we would know P=NP. Since we don't know that, a choice that lets you conclude L2 is in P cannot be correct.
3. If a problem L polynomially reduces to some problem in NP (even an NP-complete problem), then L must be in NP. This observation applies to L3, for example.

Question Explanation:

Let L = L1 [union] L2. It is possible that L1 is empty; that is certainly one language in P. If so, then L = L2, and L is NP-complete. On the other hand, suppose L1 is all strings over the alphabet of L2 --- another language in P. Then L = L1, a language in P. Since we don't know whether P = NP, we cannot conclude L is definitely in P and we cannot conclude that L is definitely NP-complete.

On the other hand, we can conclude definitely that L is in NP. A nondeterministic, polynomial-time algorithm for L starts by applying a polynomial-time algorithm for L1 to its input, and if the result is negative applies the NP-recognizer for L2 to the same input. It accepts the input if either recognizer accepts.

The argument for L = L1 ∩ L2 is essentially the same, although now L is in P if L1 is empty and L is NP-complete if L1 is all strings over the alphabet of L2. Also, the nondeterministic polynomial-time algorithm for L works by trying the polynomial-time recognizer for L1 first, and only accepting if that recognizer accepts and the NP-recognizer for L2 also accepts.

Now, consider L = L2cL3. Suppose L had an NP-recognizer. Let x be some string in L2 (since L2 is NP-complete, and we do not know that P = NP, we can be sure L2 is not empty, so x exists). Then there is a nondeterministic polynomial-time algorithm for L3 that works as follows. Take input w, and test it for membership in L3 by feeding xcw to the NP-recognizer for L. Respond exactly as this recognizer responds. Since we know x is in L2, the recognizer for L accepts xcw if and only if w is in L3. We now have an NP-recognizer for L3, but we were told that L3 is *not* in NP. Thus, our assumption that L is in NP must be false.

Last, let L be the complement of L3. If L is in P, then the complement of L, which is L3, is also in P. But we know L3 is not even in NP. We do not even know that L is in NP; for example, L3 could be an undecidable problem. On the other hand, L could be NP-complete. For example, the complement of the problem SAT is not known to be in NP. But it is possible that L3 is the complement of SAT and therefore L = SAT, a known NP-complete problem.

The correct choice is: **b)**

---

5. There is a Turing transducer *T* that transforms problem P1 into probem P2. *T* has one read-only input tape, on which an input of length *n* is placed. *T* has a read-write scratch tape on which it uses O(S(n)) cells. *T* has a write-only output tape, with a head that moves only right, on which it writes an output of length O(U(n)). With input of length *n*, *T* runs for O(T(n)) time before halting. You may assume that each of the upper bounds on space and time used are as tight as possible.

A given combination of S(n), U(n), and T(n) may:

1. Imply that *T* is a polynomial-time reduction of P1 to P2.
2. Imply that *T* is NOT a polynomial-time reduction of P1 to P2.
3. Be impossible; i.e., there is no Turing machine that has that combination of tight bounds on the space used, output size, and running time.

What are all the constraints on S(n), U(n), and T(n) if *T* is a polynomial-time reducer? What are the constraints on feasibility, even if the reduction is not polynomial-time? After working out these constraints, identify the true statement from the list below.

a) S(n) = n^2; U(n) = n^3; T(n) = n^4 is a polynomial-time reduction

b) $S(n) = n^2$; $U(n) = n^3$; $T(n) = n^4$ is possible, but not a polynomial-time reduction.

c) $S(n) = n$; $U(n) = n^2$; $T(n) = 2^n$ is a polynomial-time reduction

d) $S(n) = n^2$; $U(n) = 1$; $T(n) = n^{10}$ is not physically possible.

Answer submitted: **a)**

You have answered the question correctly.

Question Explanation:

There is one constraint on *T* being a polynomial-time reduction: T(n) must be a polynomial. However, there are also some constraints on feasibility. First, in time T(n), *T* cannot write more than T(n) cells on either its scratch or output tape. Thus, S(n) and U(n) must both be O(T(n)). Second, if T(n) is larger than $O(n \log S(n) c^{S(n)})$ for any constant *c*, then *T* must repeat an "ID" consisting of the state, scratch-tape contents, and head positions on the input and scratch tapes.

---

6. The classes of languages P and NP are closed under certain operations, and not closed under others, just like classes such as the regular languages or context-free languages have closure properties. Decide whether P and NP are closed under each of the following operations.

1. Union.
2. Intersection.
3. Intersection with a regular language.
4. Concatenation.
5. Kleene closure (star).
6. Homomorphism.
7. Inverse homomorphism.

Then, select from the list below the true statement.

a) P is not closed under concatenation.

b) NP is not closed under homomorphism.

c) NP is not closed under Kleene closure.

d) P is not closed under Kleene closure.

Answer submitted: **b)**

You have answered the question correctly.

Question Explanation:

Both P and NP are closed under each of these operations, except for homomorphism. To see why neither class is closed under homomorphism, start with a very hard language *L*, say one that requires time 2^{2^n}. Make it easy by appending to each word of length *n* exactly 2^{2^n} *c*'s, where *c* is a new symbol. That is, let $L' = Lc^{2^{2^n}}$. Then *L'* is surely in P and NP. But *L* is h(L'), if *h* is the homomorphism that sends *c* to ε and is the identity on all symbols of *L*. If P or NP were closed under homomorphism, then *L* would be in NP, which it is not.

Here are the constructions that show P and NP closed under the other six operations:

1. L1 [union] L2: Apply the tests for membership in L1 and then for membership in L2. Accept if either accepts. If L1 and L2 are in P, then both tests are polynomial, so the entire process is polynomial. If L1 and L2 are in NP, then there is a nondeterministic polynomial algorithm for the entire process.

2. L1 ∩ L2: The argument is the same, but accept only if both tests accept.

3. Intersection with a regular set: The argument is the same as (2), since a regular language is surely in P and NP.

4. L1L2: If L1 and L2 are in NP, just guess the point on the input where the string in L2 begins, and apply the nondeterministic polynomial tests to the two parts of the input. If L1 and L2 are in P, we have to try systematically all possible breakpoints between the prefix of the input that is in L1 and the suffix that is in L2. If the tests for L1 and L2 are polynomial, say p(n) and q(n) running time, then we must apply each at most *n* times on input of length *n*. Since n(p(n)+q(n))

is a polynomial, we can recognize L1L2 in polynomial time.

5. L*: If *L* is in NP, guess all the places on the input where strings in *L* end. Run the nondeterministic polynomial-time test on each segment. Since there are at most *n* segments, the whole process is nondeterministic polynomial. If *L* is in P, we again must be systematic. For each pair of input positions *i* and *j*, run the polynomial-time test to see whether positions *I* through *j* of the input is in *L*. If *L* has a p(n)-time algorithm, then this process takes $O(n^2p(n))$ time, a polynomial. Then, use a dynamic programming algorithm, taking $O(n^3)$ time, to determine whether positions *i* through *j* can be composed of 1, 2,...,*n* strings in *L*. At the end, we only care about whether positions 1 through *n* can be composed of some number of strings in *L*.

6. $h^{-1}(L)$: Any homomorphism *h* can only expand the length of the string to which it is applied by a constant factor. To recognize $h^{-1}(L)$, apply *h* to the input *w*, and see whether h(w) is in *L*. If there is a polynomial-time, or nondeteministic polynomial-time test for membership in *L*, this test will, in the same order of magnitude time complexity tell us whether *w* is in $h^{-1}(L)$.
