

CSCI 303 Homework 5

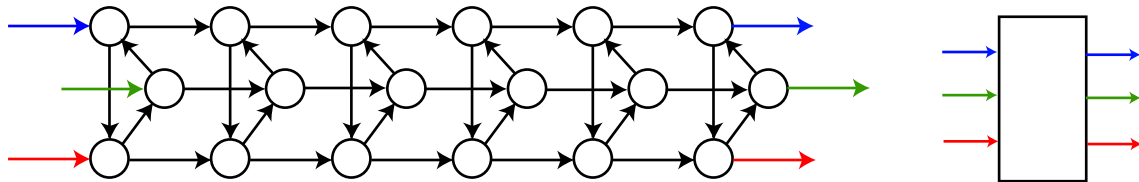
Problem 1 (Not in book):

Using the polynomial time reduction given in class, give an instance of the Directed Hamiltonian Cycle problem that corresponds to the instance of the 3-SAT problem where

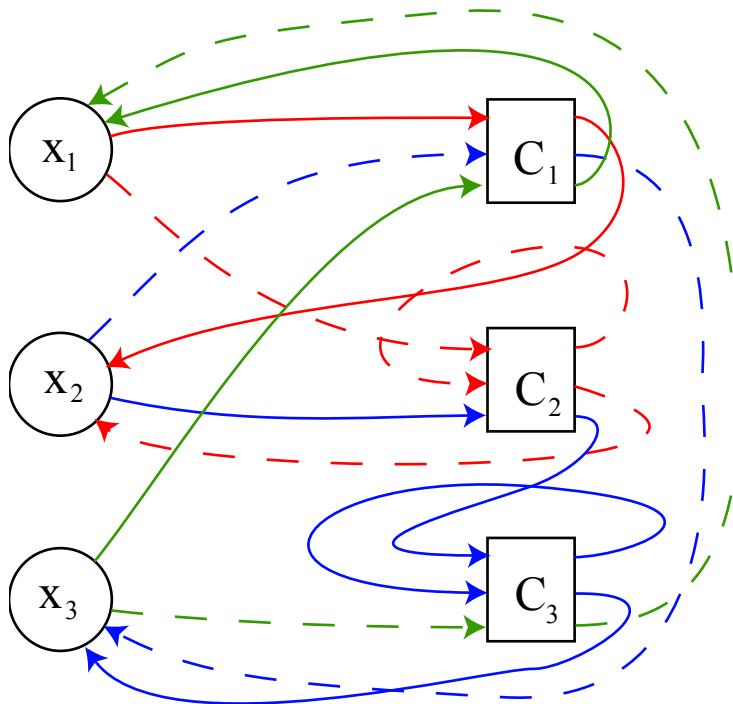
$$\phi = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_1) \wedge (x_2 \vee x_2 \vee \neg x_3).$$

Solution 1:

We will use the gadget discussed in class, and drawn below, first as a graph, then in the simpler box representation:



In the diagram below, each box represents a clause and each circle represents a variable. Solid red arrows represent the literal x_1 and dashed red arrows represent $\neg x_1$. Solid blue arrows represent x_2 and dashed blue arrows represent $\neg x_2$. Solid green arrows represent x_3 and dashed green arrows represent $\neg x_3$. A literal selection that satisfies ϕ will, using the appropriate colored and styled lines, form a directed Hamiltonian cycle. A literal selection that does not satisfy ϕ will not form a directed Hamiltonian cycle, as it will miss at least one of the gadgets.



Problem 2 (34.1-5):

Show that an otherwise polynomial time algorithm that makes at most a constant number of calls to polynomial time subroutines runs in polynomial time, but that a polynomial number of calls to polynomial time subroutines may result in an exponential time algorithm.

Solution 2:

Assume that algorithm A runs in polynomial time, not counting calls to subroutines. That is to say, A has worst-case asymptotic complexity $O(n^k)$ for some k , again not counting calls to subroutines. A makes constant number c of calls to polynomial time subroutines B_1, B_2, \dots, B_c , each of which has worst-case asymptotic complexity $O(n^s)$, for some s . The size of the outputs of the subroutines must be polynomial in the size of their inputs (otherwise the subroutines wouldn't run in polynomial time); in fact, the size of the outputs of the subroutines is $O(n^s)$. Assume A is run with an input of size n . It then runs subroutine B_1 with an input of size n_1 . How large can n_1 be? Since A runs in time $O(n^k)$, it follows that n_1 must also be $O(n^k)$. Assume* that n_1 is exactly n^k . How long does B_1 take? It takes $O(n_1^s)$ time. Since B_1 runs in time $O(n_1^s)$, the size of its output is at most n_1^s . A then runs B_2 on an input of size n_2 . How large can n_2 be? By the same reasoning as before, n_2 is at most n_1^s (because B_1 only has $O(n_1^s)$ time to act on its input). B_2 takes time $O(n_2^s)$. Continuing in this way, the output of subroutine B_i providing the input of subroutine B_{i+1} , we continue for c steps. How long does this take in total? It takes time for A itself, plus the time for B_1 , plus the time for B_2 , etc. This is $O(n^k) + O(n_1^s) + O(n_2^s) + \dots + O(n_c^s)$. Since $n_1 = n^k$, and $n_2 = n_1^s$, and $n_3 = n_2^s$, the total time is

$$O(\underbrace{(((\dots((n^k)^s \dots)^s)^s)}_c) = O(n^{ks^c}),$$

which is polynomial in n . Thus, when A makes a constant number of calls to polynomial time subroutines, A still runs in polynomial time.

Taking the above argument and replacing c with n shows that A might take exponential time if it makes a polynomial number of calls to polynomial time subroutines. For a more concrete example, suppose a subroutine B takes a string w and returns the concatenation of w with itself, or ww . That is, on input "10010" the subroutine would return "1001010010". Clearly, B can run in polynomial time. Suppose an algorithm A takes as input a number n and calls B exactly n times, giving as input to the subroutine the output of the previous call to the subroutine. Let $s_0 = 1$. First A calls B on input s_0 , which outputs $11 = s_1$. It then calls B on s_1 to get s_2 . It continues in such manner for n iterations, finally returning s_n . Since the length of the string doubles every time, the length of s_n is 2^n after only n calls to B . But it takes at least 2^n steps to output a string of that length. Thus A must take exponential time even though it only makes a polynomial number of calls to a polynomial time subroutine.

* It's not technically correct to do this, but we haven't studied the formalism of Big-O notation deeply enough to do it the "right" way, and the "right" way ends up being the same basic argument, just messier.

Problem 3 (Derived from 34.3-2):

Show that if Decision Problem 1 is polynomial time reducible to Decision Problem 2 and Decision Problem 2 is polynomial time reducible to Decision Problem 3, then Decision Problem 1 is polynomial time reducible to Decision Problem 3.

Solution 3:

If Decision Problem 1 is polynomial time reducible to Decision Problem 2, then there exists a polynomial time algorithm A_1 that is membership-preserving. That is to say, A_1 maps positive instances of Decision Problem 1 to positive instances of Decision Problem 2 and maps negative instances of Decision Problem 1 into negative instances of Decision Problem 2. Likewise, if Decision

Problem 2 is polynomial time reducible to Decision Problem 3, then there exists a polynomial time algorithm A_2 that maps positive instances of Decision Problem 2 to positive instances of Decision Problem 3 and maps negative instances of Decision Problem 2 into negative instances of Decision Problem 3.

Consider an algorithm A that is the concatenation of algorithms A_1 and A_2 . A takes, as inputs, instances of Decision Problem 1 and uses A_1 to map them to instances of Decision Problem 2, then uses A_2 to map these instances of Decision Problem 2 into instances of Decision Problem 3. A must map positive instances of Decision Problem 1 to positive instances of Decision Problem 3, and must map negative instances of Decision Problem 1 to negative instances of Decision Problem 3, thus A is membership-preserving. Since A_1 and A_2 are both polynomial time, and the worst-case asymptotic complexity of A is at most the sum of the worst-case asymptotic complexities of A_1 and A_2 , A itself must be polynomial time. Therefore Decision Problem 1 is polynomial time reducible to Decision Problem 3. This result says that polynomial time reductions are transitive.

Problem 4 (Not in book):

Prove that the 3-SAT problem is polynomial time reducible to the Vertex Cover problem.

Solution 4:

In class we saw that the 3-SAT problem is polynomial time reducible to the Clique problem, and that the Clique problem is polynomial time reducible to the Vertex Cover problem. Since we know that polynomial time reductions are transitive from problem 3, it immediately follows that the 3-SAT problem is polynomial time reducible to the Vertex Cover problem.

Problem 5 (Derived from 34.3-3):

Let D be a decision problem, then the complement of D , denoted \overline{D} , is the decision problem such that positive instances of D are negative instances of \overline{D} and negative instances of D are positive instances of \overline{D} . For example,

The 3-SAT Problem:

INPUT: ϕ , a Boolean Formula in 3-CNF form
 OUTPUT: 1 if ϕ is *not* satisfiable
 0 if ϕ is satisfiable

The Search Problem:

INPUT: $(a_1, a_2, \dots, a_n; b)$ are numbers
 OUTPUT: 1 if there does *not* exist an i such that $a_i = b$
 0 if if there does exist an i such that $a_i = b$

- a. Prove or disprove:
The search problem is polynomial time reducible to the search problem.
- b. Prove or disprove:
For all decision problems D , D is polynomial time reducible to \overline{D} .
- c. Prove or disprove:
The 3-SAT problem is polynomial time reducible to the 3-SAT problem.

Solution 5:

- a. The search problem *is* polynomial time reducible to the search problem, and to prove it, it is sufficient to give a polynomial time algorithm A that maps positive instances of the

search problem to positive instances of the search problem, and negative instances of the search problem to negative instances of the search problem.

The search problem itself has polynomial time algorithms that solve it, such as UNSORTED-SEARCH. On input $(a_1, a_2, \dots, a_n; b)$, let A call UNSORTED-SEARCH as a subroutine to determine whether there exists an i such that $a_i = b$. If the answer is “yes” (i.e., the input was a positive instance of the search problem), A will output $(1, 2; 3)$, which is a positive instance of the search problem, because 3 is not in the list 1, 2. If the answer is “no” (i.e., the input was a negative instance of the search problem), A will output $(1, 2; 2)$, which is a negative instance of the search problem, because 2 is in the list 1, 2. A clearly runs in polynomial time, so the search problem is polynomial time reducible to the search problem.

- b. It turns out that it is *not* true that all decision problems D are polynomial time reducible to \overline{D} . This result says that polynomial time reductions are not commutative. Unfortunately, we haven’t yet studied the tools we will need to prove this. Stay tuned, this will be a problem in Homework 7.
- c. It is currently *unknown* whether the 3-SAT problem is polynomial time reducible to the 3-SAT problem. This question is likely just as hard, and may even be harder than, the P vs. NP question. If you think you can answer it one way or the other, let me know. I’d like to coauthor a paper with you.

Problem 6 (Derived from 34.2-3):

Consider the following problem

The Hamiltonian Cycle Identification Problem:

INPUT: G , a directed graph

OUTPUT: a list of nodes, in order, of a Hamiltonian cycle of G , if one exists
0 if G does not contain a Hamiltonian cycle

Suppose that someone gives you a polynomial time algorithm to solve the Hamiltonian Cycle problem. Describe how to use this algorithm to find a polynomial time algorithm that solves the Hamiltonian Cycle Identification problem.

Solution 6:

Assume that there exists a polynomial time algorithm A that solves the Hamiltonian Cycle problem. Let G be a graph. Run A on input G . If A outputs 0, then there is no Hamiltonian cycle in G , so output 0. Otherwise, remove one edge from G to form a new graph G_1 and run A on input G_1 . If A outputs 0, then the edge we removed was needed in all Hamiltonian cycles in G , so put it back in. If A outputs 1 on input G_1 , then there is a Hamiltonian cycle in G_1 , so leave the edge out. Continue trying to remove edges, one at a time, putting them back in if A outputs 0 on the resulting graph and leaving them out if A outputs 1 on the resulting graph. After trying to remove every edge once, the graph that remains will be a Hamiltonian cycle that is contained in G .

What is the worst-case asymptotic complexity of this algorithm? If G contains m edges, then we ran A precisely m times. Since A ran in polynomial time, our algorithm is also polynomial time.

Problem 7 (34.5-5):

The Set-Partition problem takes as input a set S of numbers. The question is whether the numbers can be partitioned into two sets A and $\overline{A} = S - A$ such that $\sum_{x \in A} x = \sum_{x \in \overline{A}} x$. Show that the Set-Partition problem is NP-complete.

Solution 7:

To show that the Set-Partition problem is NP-complete, we will show that the Subset-Sum problem is polynomial time reducible to the Set-Partition problem. Recall that the Subset-Sum problem takes as input a set W of numbers and a target number t and outputs 1 iff there is a subset of W that sums to t . We need to find a membership-preserving, polynomial time algorithm A that maps instances of the Subset-Sum problem into instances of the Set-Partition problem.

We are given an instance of the Subset-Sum problem, W, t . Let $k = \sum_{x \in W} x$ and let $\ell > \max(k, t)$. Create a new set $S = W \cup \{\ell - t, \ell - k + t\}$. The sum of the elements of S is 2ℓ . We claim that there exists a subset of the numbers in W that add up to t if and only if there exists a partition of S into two sets A and $\bar{A} = S - A$ such that the sum of the elements in A is the same as the sum of the elements in \bar{A} .

Assume there exists a subset V of W whose elements sum to t . Then the elements of $\bar{V} = W - V$ sum to $k - t$. Partition S into $A = V \cup \{\ell - t\}$ and $\bar{A} = \bar{V} \cup \{\ell - k + t\}$. The elements of A sum to ℓ , and the elements of \bar{A} also sum to ℓ , so if (W, t) is a positive instance of the Subset-Sum problem, then S is a positive instance of the Set-Partition problem.

Conversely, assume S is a positive instance of the Set-Partition problem. Then we can partition S into A and \bar{A} such that the elements of A sum to the same thing as the elements of \bar{A} do. Since the sum of all the elements of S is 2ℓ , the elements of A must sum to ℓ . We claim that one of $\ell - t$ and $\ell - k + t$ must be in A and the other must be in \bar{A} . If they were both in the same set, then the sum of the elements in that set would be at least $2\ell - k > k$, but the sum of all the other elements is exactly k . So, without loss of generality, assume $\ell - t \in A$. But the sum of all the elements in A is ℓ , so the elements of $A - \{\ell - t\}$ must sum to t . Since $A - \{\ell - t\} \subseteq W$, there exists a subset of W that sums to t , so (W, t) is a positive instance of the Subset-Sum problem.