NP-Completeness Problems

Sample Solved Exercises

Prof. Pedro C. Diniz

Departamento de Engenharia Informática (DEI)
Faculdade de Engenharia da Universidade do Porto (FEUP)

Spring 2023

Exercise 34.1-4
Is the dynamic-programming algorithm for the 0-1 knapsack problem that is asked for in Exercise 16.2-2 a polynomial-time algorithm? Explain your answer.

Solution:
The dynamic programming algorithm for the 0-1 knapsack problem has running time O(nW), where n is the number of items and W is the maximum weight that the knapsack can hold. This is **not** a polynomial running time for any reasonable representation of the input. In a reasonable representation, all numeric values (the weights, the values, etc.) will be given in binary (base 2, or perhaps even a higher base). To represent the value W takes log W bits; thus the running time of O(nW) is **exponential** in the size of the input (although polynomial in the magnitude).

Exercise 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:
If an otherwise polynomial-time algorithm makes at most a constant number of calls to polynomial-time subroutines then the total running time for the algorithm must be polynomial because if the subroutines being called run in time O(nk) for some constant k and there are a total of c such calls then the total running time for the algorithm will be O(cnk) = O(nk). On the other hand, a polynomial number of calls to a polynomial-time subroutine may result in an exponential-time algorithm. Consider the following.

```
RUN-SLOW(n)
1  s ← a
2  for i = 1 to n do
3     s ← CONCATENATE(s, s)
```

Exercise 34.2-1
Consider the language GRAPH-ISOMORPHISM = {G_1, G_2: G1 and G2 are isomorphic graphs}. Prove that GRAPH-ISOMORPHISM $\in$ NP by describing a polynomial-time algorithm to verify the language.

Solution:
To show that GRAPH-ISOMORPHISM is in NP, we must construct a non-deterministic algorithm that decides GRAPH-ISOMORPHISM in polynomial time. The following is a nondeterministic algorithm that does so:

```
GRAPH-ISOMORPHISM(G, H )
1 if G does not have the same number of vertices as H then return false.
2  non-deterministically guess a permutation (bijection) π of m vertices.
3 for each pair of vertices (x, y) in G do
4 verify that (x, y) is an edge in G if and only if (π(x), π(y)) is an edge of H .
5 end for
6 if all edges agree, then return true.
7 else return false.
```

Exercise 34.2-2
Prove that if G is an undirected bipartite graph with an odd number of vertices, then G is non-hamiltonian.

Solution:
Because bipartite graph does not have an odd-length cycle, it cannot have an Hamitonian cycle.

Exercise 34.2-3
Show that if HAM-CYCLE $\in$ P, then the problem of listing the vertices of a hamiltonian cycle, in order, is polynomial-time solvable.

Solution:

Exercise 34.2-5
Show that any language in NP can be decided by an algorithm running in time $2^{(\cdot)}$ for some constant k.

Solution:

Exercise 34.2-8
Let ϕ be a boolean formula constructed from the boolean input variables x_1 , x_2 , . . . , x_k , negations ($\neg$), AND's ($\wedge$), OR's ($\vee$), and parentheses. The formula ϕ is a tautology if it evaluates to 1 for every assignment of 1 and 0 to the input variables. Define TAUTOLOGY as the language of boolean formulas that are tautologies. Show that TAUTOLOGY $\in$ co-NP.

Solution:
Let the certificate be the values of the k input variables that make ϕ evaluate to false. This can be verified in polynomial time and is enough to prove that ϕ is NOT a tautology. Therefore, since TAUTOLOGY $\in$ NP, it follows that TAUTOLOGY $\in$ co-NP.

Exercise 34.3-3

Prove that $L \leq_P \bar{L}$ if and only if $\bar{L} \leq_P L$.

Solution:

If $L \leq_p \bar{L}$ then by definition, there exists a function f such that $x \in L \Leftrightarrow f(x) \in \bar{L}$. But this means that $x \in L \Leftrightarrow f(x) \in \bar{L}$, or in other words, $x \in \bar{L} \Leftrightarrow f(x) \in L$ and it follows that $\bar{L} \leq_p L$. The reverse direction can be proven similarly.

Exercise 34.4-3

Professor Jagger proposes to show that SAT $\leq_P$ 3-CNF-SAT by using only the truth-table technique in the proof of Theorem 34.10, and not the other steps. That is, the professor proposes to take the boolean formula ϕ, form a truth table for its variables, derive from the truth table a formula in 3-DNF that is equivalent to $\neg\phi$, and then negate and apply DeMorgan's laws to produce a 3-CNF formula equivalent to ϕ. Show that this strategy does not yield a polynomial-time reduction.

Solution:

This strategy does not yield a polynomial-time reduction because forming the truth table for a boolean formula over n variables will require 2^n rows and thus the reduction would run in time $\Omega(2^n)$.

Exercise 34.4-6

Suppose that someone gives you a polynomial-time algorithm to decide formula satisfiability. Describe how to use this algorithm to find satisfying assignments in polynomial time.

Solution:

Suppose we are given a polynomial-time algorithm to decide formula satisfiability. Call this algorithm S AT and imagine that its running time is $O(n^k)$. If we are to find a satisfying assignment for a boolean formula ϕ, we first run SAT(ϕ). If the algorithm returns false then we quit, otherwise we proceed as follows. Form two new formulae ϕ_1 and ϕ_2 such that we substitute 1 in for x_1 in ϕ_1 and 0 in for x_1 in ϕ_2. Since ϕ is satisfiable we are assured that at least one of ϕ_1 or ϕ_2 is satisfiable. Now as we did initially, run SAT(ϕ_1) and SAT(ϕ_2) to determine whether x_1 is true or false. Whatever is the case, make the appropriate substitution permanent and continue doing this for every variable. If there are n variables then the preceding algorithm will run in time $O(nxn^k) = O(n^{k+1}) = O(n^k)$.

Note: This is a case where we are making a polynomial number of calls to a polynomial-time subroutine (SAT). Does this contradict the first problem? Why or why not?

Exercise 34.4-6

Suppose we are given a polynomial-time algorithm to decide formula satisfiability. Call this algorithm SAT and imagine that its running time is $O(n^k)$.

Solution:

If we are to find a satisfying assignment for a boolean formula ϕ, we first run SAT(ϕ). If the algorithm returns false then we quit, otherwise we proceed as follows. Form two new formulae ϕ_1 and ϕ_2 such that we substitute 1 in for x_1 in ϕ_1 and 0 in for x_1 in ϕ_2 . Since ϕ is satisfiable we are assured that at least one of ϕ_1 or ϕ_2 is satisfiable. Now as we did initially, run S AT (ϕ_1) and SAT(ϕ_2) to determine whether x_1 is true or false. Whatever is the case, make the appropriate substitution permanent and continue doing this for every variable. If there are n variables then the preceding algorithm will run in time $O(n \times n^k) = O(n^{k+1}) = O(n^k)$.

Note: This is a case where we are making a polynomial number of calls to a polynomial-time subroutine

(SAT). Does this contradict the first problem? Why or why not?

Exercise 34.4-7

Let 2-CNF-SAT be the set of satisfiable boolean formulas in CNF with exactly 2 literals per clause. Show that 2-CNF-SAT is in P. Make your algorithm as efficient as possible. (Hint: Observe that x ∨ y is equivalent to ¬x ⇒ y. Reduce 2-CNF-SAT to a problem on a directed graph that is efficiently solvable.)

Solution:

By the definition of 2-CNF-SAT, every formula is a conjunction of clauses of the form x ∨ y. For the formula to be satisfiable, each one of these clauses must be satisfied. Following the provided hint, x ∨ y is equivalent to ¬x ⇒ y and (because of the commutativity of ∨) ¬y ⇒ x. Using this "translation" we construct a graph for every formula whose nodes are the literals that appear in the formula (i.e., there is a different node for x and ¬x. Then, we add an edge between v_1 and v_2 if $v_1 \Rightarrow v_2$ (e.g., in the example above we would add an edge from ¬x to y and another from ¬y to x. Then, we compute the SCC graph of the graph that is produced. Consider a sink node/component of this graph (i.e., one with no outgoing edges). If there are both x and ¬x in that node, this implies that there is a path from ¬x to x (and vice versa) which implies ¬x ⇒ x (and the converse). Thus, there is no assignment of truth value to x that can satisfy this clause. As a result, we can safely conclude that the formula is unsatisfiable. Otherwise, i.e., if the sink component under consideration does not contain any such pairs of "incompatible" literals, we can assign the value true to all the literals in the component (this means that if x is in the component we set x = true while if ¬y is in the component we set ¬y = true, i.e., y = false. We claim that this way every clause of the original formula that contains a variable that appears in the component is satisfied. For, suppose by way of contradiction that x is in the component and there exists an a not in the component and a clause ¬x ∨ a. Then, x ⇒ a would be part of the graph, which means that a is reachable from x. But since x is in a sink component, a must also be in the same component. Thus, a would have been set to true satisfying and eliminating the clause ¬x ∨ a. As a result, we can safely eliminate those clauses from the formula (or equivalently remove the considered sink SCC from the SCC graph) and proceed to solve the resulting (smaller) problem. This algorithm will obviously terminate eventually (since the size of the recursive sub-problem always decreases and the size of the original formula is finite) and will produce a correct value assignment, as explained by the argument above, if such an assignment exists, or identify that the formula is unsatisfiable, if an incompatible pair exists. Finally, this algorithm operates in linear time on the size of the graph, which is the same as the size of the formula.

Exercise 34.5-1

The subgraph-isomorphism problem takes two graphs G1 and G2 and asks whether G1 is isomorphic to a subgraph of G2 . Show that the subgraph-isomorphism problem is NP-complete.

Solution:

To show that a problem is NP -complete, we must first show that the given problem is in NP and then show that every problem in NP polynomial-time reduces to the given problem.

It is easy to see that SUBGRAPH-ISOMORPHISM is in NP because we can non-deterministically guess the mapping of vertices. To show that every problem in NP is polynomial-time reducible to SUBGRAPH-ISOMORPHISM, we will reduce from the known NP-complete problem CLIQUE. That is, we will show that CLIQUE $\leq_p$ SUBGRAPH-ISOMORPHISM. The following algorithm computes the reduction:

F(G,m)
1 Construct the complete graph K_m
2 Output K_m, G

One can see that G has a clique of size m if and only if K_m is isomorphic to a subgraph of G. Furthermore, since this reduction can be computed in polynomial time, it follows that SUBGRAPH-ISOMORPHISM is NP-complete.

Exercise 34.5-2
Given an integer m-by-n matrix A and an integer m-vector b, the 0-1 integer-programming problem asks whether there is an integer n-vector x with elements in the set {0, 1} such that $Ax \leq b$. Prove that 0-1 integer programming is NP-complete. (Hint: Reduce from 3-CNFSAT.)

Solution:
It should be clear that given a correct solution one can check in polynomial time that it satisfies $Ax \leq b$. For NP-hardness, we can reduce 3-CNF-SAT to 0-1 integer-programming as follows: Let ϕ be a formula that is an instance of 3-CNF-SAT. For every clause of the form $(x \vee y \vee z)$, produce an equation $f(x) + f(y) + f(z) \leq 2$, where $f(x) = x$, if x is a positive literal, and $f(x) = 1 - x$ otherwise. For example, $(\neg x \vee \neg y \vee w)$ would be translated to $1 - x + 1 - y + w \leq 2 \Rightarrow w - x - y \leq 0$.

Finally, let (perhaps counter-intuitively) $x = 0$ in the integer program if $f(x)$ = true in 3-CNF-SAT, and $x = 1$ if $f(x)$ = false for all variables. If 3-CNF-SAT is satisfiable, every clause is satisfiable, i.e., at least one of the disjuncts is true (i.e., 0), and as a result the corresponding equation is a sum of three numbers, each of which is no greater than 1 and one of which is definitely 0. It should be obvious that their sum is always ≤ 2, i.e., the equation is satisfied, i.e., there is a solution for the integer program (which is the same as the solution for 3-CNF-SAT by replacing true with 0 and false with 1, as explained above). Conversely, if there is a solution for integer programming, every equation must be satisfied. Observe that the only way to satisfy a formula of the form $x + y + z \leq 2$ where $x, y, z \in \{0, 1\}$ is to have at least one of them be 0. As a result, in the corresponding clause at least one of the literals is true, i.e., the formula is satisfiable.

Exercise 34.5-3
The integer linear-programming problem is like the 0-1 integer-programming problem given in Exercise 34.5-2, except that the values of the vector x may be any integers rather than just 0 or 1. Assuming that the 0-1 integer-programming problem is NP-hard, show that the integer linear-programming problem is NP-complete.

Solution:
This problem is a generalization of the 0-1 Integer Programming problem which has been shown to the NP-Complete. It is thus also NP-Complete.

Exercise 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 $A = S - A$ such that $\sum_{x \in A} x = \sum_{x \in A} x$. Show that the set-partition problem is NP-complete.

Solution:
SET-PARTITION is in NP since a subset of the numbers whose sum is equal to half the sum of the total set is clearly a certificate of a solution and can be checked in polynomial time. For NP-hardness we are going to reduce SUBSET-SUM to SET-PARTITION. That is, we take an instance $I = (S, t)$ of the SUBSET-SUM problem and reduce it in polynomial time to an instance $f(I)$ of the SET-PARTITION, so that I is a "yes" instance iff $f(I)$ is a "yes" instance of the corresponding problems. Let $S(I) = \sum_{x \in S} x$. We construct an instance $f(I) = (S)$ of the SET-PARTITION problem by adding to the set S the number $n = |S(I) - 2t|$, i.e., $S = S \cup \{|S(I) - 2t|\}$. We consider two cases:
• $S(I) \geq 2t$. Then the added number has value $S(I) - 2t$. The total value of all numbers in
$f(I)$ is $2S(I) - 2t$. If the numbers can be split to A, A as required so that $\sum_{x \in A} x = \sum_{x \in A} x$ then each partition has a total value of $S(I) - t$. Consider the partition that contains the added number n. If we remove it we get a subset of value $S(I) - t - (S(I) - 2t) = t$. Thus if $f(I)$ is a "yes" instance of SET-PARTITION, I is a "yes" instance of SUBSET-SUM. Conversely, if I is a "yes" instance of SUBSET-SUM,

there is a subset of numbers whose sum is t, and adding n to them yields a partition of value S (I) – t, i.e., half the amount in f (I)'s set, that is f (I) is a "yes" instance of SET-PARTITION.

• S (I) < 2t. Then, the added number n has value 2t – S (I) and the total value of S in f (I) is 2t. If f (I) is a "yes" instance then S can be divided into A and A, each of whose sum of elements is t. Without loss of generality let A be the subset that does not contain n, then A is a subset of S and as a result a solution to the SUBSET-SUM problem for I . Conversely, if I is a "yes" instance, there is a subset of b ⊆ S in I whose sum of elements is S (I) – t. Then B ∪ {n} is a partition of the elements of S in f (I) and lB ∪ {n}l = S (I) – t + 2t – S (I) = t. As a result, B ∪ {n}, S – (B ∪ {n}) is a solution for the SET-PARTITION problem, i.e., f (I) is a "yes" instance of SET-PARTITION,

.