Cornell University, Fall 2021	CS 6820: Algorithms
Lecture notes on NP-completeness	Oct 2021

These lecture notes concern problems that are thought to be computationally hard. A distinguishing feature of computational complexity theory is that most problems of interest can be organized into a relatively small number of equivalence classes, called "complexity classes", under the relation of polynomial-time reducibility (to be defined below). The existence of a polynomial-time for any problem in one of these classes would imply that all the other problems in that same complexity class can be solved in polynomial time. A major challenge in theoretical computer science is to prove that these complexity classes really are distinct; the most famous special case of this challenge is the "P versus NP" problem.

1 Defining NP, co-NP, #P

Many important complexity classes can be defined in terms computing the value of an exponentially large formula. Let $\odot$ be a binary operation that is both commutative and associative, taking values in some set Ω. For example, Ω could be the natural numbers and $\odot$ could be addition, or Ω could be the set of Boolean truth values $\{0,1\}$ (with 1 interpreted as TRUE and 0 interpreted as FALSE) and $\odot$ could be the Boolean OR ($\vee$) or AND ($\wedge$) operation.

Suppose that $V(\cdot,\cdot)$ is a polynomial-time algorithm that takes two inputs, x and y, and produces a value in Ω. We will denote the number of bits needed to describe x and y by $|x|$ and $|y|$, respectively, and we will assume they are related by $|y| = p(|x|)$, where p is a function of at most polynomial growth. Then one can consider the problem of computing the function

$$F(x) = \bigodot_{y \in \{0,1\}^{|p(x)|}} V(x,y).$$

When one instantiates this definition for $\odot \in \{\vee, \wedge, +\}$ one obtains the classes NP, coNP, and #P.

Definition 1. A function $F : \{0,1\}^* \to \{0,1\}$ belongs to NP if and only if there exists a polynomial-time algorithm V such that $F(x) = \bigvee_y V(x,y)$.

A function $F : \{0,1\}^* \to \{0,1\}$ belongs to coNP if and only if there exists a polynomial-time algorithm V such that $F(x) = \bigwedge_y V(x,y)$.

A function $F : \{0,1\}^* \to \{0,1\}$ belongs to #P if and only if there exists a polynomial-time algorithm V such that $F(x) = \sum_y V(x,y)$.

In the context of the class NP, the algorithm V is often called a "polynomial-time verifier" for the problem F. The verifier's two inputs, x and y, are interpreted as follows: x is an input instance of the decision problem described by F, and y is a "candidate solution." The verifier outputs 1 (true) if y is a valid solution to x, and it outputs 0 (false) if y is not a valid solution. The task of computing $F(x) = \bigvee_y V(x,y)$ can then be interpreted as deciding whether the problem specified by x has a solution.

For example, the GRAPH 3-COLORABILITY problem is the following decision problem: given a graph $G = (V, E)$, is there a function $h : V \to \{0, 1, 2\}$ such that $h(u) \neq h(v)$ whenever G contains an edge with endpoints u, v. To interpret GRAPH 3-COLORABILITY as a problem in NP, first let x denote a binary string encoding the graph G, for example a string of n^2 bits that lists the entries of the adjacency matrix of G in row-major order. Then let y denote a binary string encoding a potential coloring of G, for example by representing each of the colors in $\{0, 1, 2\}$ as an element of $\{0, 1\}^2$ and then concatenating the encodings of the colors of each of the n vertices of G into a string of $2n$ bits. A verifier V for GRAPH 3-COLORABILITY works as follows: if runs through the adjacency matrix to find all the edges (u, v). For each edge, it looks up the binary encodings of $h(u)$ and $h(v)$ in the string y and tests that $h(u), h(v)$ both belong to $\{0, 1, 2\}$ and that $h(u) \neq h(v)$. If y passes all of these tests then $V(x, y) = 1$; otherwise $V(x, y) = 0$. Clearly the verifier can perform these tests in polynomial time, and the verifier outputs 1 if and only if y is the binary encoding of a valid 3-coloring of G. Hence, the value of the function $F(x) = \bigvee_{y \in \{0,1\}^{2n}} V(x, y)$ is 1 if x is the binary encoding of a 3-colorable graph, and 0 otherwise.

The class coNP is defined similarly to NP but using Boolean AND rather than OR. In other words, when F belongs to coNP, computing $F(x)$ for an input x is tantamount to deciding whether the verifier *always* outputs 1 on input pair (x, y) regardless of the value of y. An archetypical example of a coNP problem is the TAUTOLOGY problem: given a quantifier-free Boolean logical formula on n variables, decide if the formula is a tautology. In this case, the verifier V takes a binary string x that describes the Boolean formula, and a binary string y encoding a potential truth assignment to the n variables, and $V(x, y)$ outputs the truth value of the formula represented by x, when one assigns the variables the truth values represented by y.

The class #P consists of *counting problems*. An archetypical example is the problem of counting the number of perfect matchings of a graph G. In this example, V could be an algorithm that takes a string x representing the adjacency matrix of G, and a string y representing the adjacency matrix of a subgraph H, and V could perform the following tests:

- y is a symmetric matrix: $y[i, j] = y[j, i]$ for all i, j.
- y is the adjacency matrix of a subgraph of G: $y[i, j] \leq x[i, j]$ for all i, j.
- y is the adjacency matrix of a perfect matching: $\sum_j y[i, j] = 1$ for all i.

Matrices y that pass these tests are in one-to-one correspondence with perfect matchings in G. Hence, computing $\sum_{y \in \{0,1\}^{n^2}} V(x, y)$ is tantamount to counting the number of perfect matchings of G.

1.1 A remark about data types

Above, we have adopted the convention that the input to a problem must be a binary string, $x \in \{0, 1\}^*$. Although any type of data can be encoded using a binary string, in practice we want to talk about the complexity of problems whose inputs are graphs, logical formulas, and the like. Insisting that the definition of computational problem must include a specification

of how to encode the input as a binary string tends to take one's attention away from the computational essence of the problem and refocus it on relatively meaningless low-level details of the data encoding. (Are we presenting graphs using an adjacency matrix or an adjacency list? When the input to a problem is a sequence of integers, how does the input string indicate where the digits of one number end and the digits of the next one begin?)

Accordingly, it is convenient to generalize our notion of computational problems as follows. We assume the problem is defined by:

- a set of input instances, $\mathscr{X}$, accompanied by a measure of *input size* $\text{size} : \mathscr{X} \to \mathbb{N}$.
- a set of candidate solutions, $\mathscr{Y}$, accompanied by a function that assigns to each $x \in \mathscr{X}$ a finite subset $\mathscr{Y}(x) \subseteq \mathscr{Y}$.
- a set of output values, Ω, with a commutative and associative binary operation $\odot$.
- a binary encoding function $\text{enc} : \mathscr{X} \cup \mathscr{Y} \cup \Omega \to \{0,1\}^*$, whose restriction to each of the sets $\mathscr{X}, \mathscr{Y}, \Omega$ is one-to-one. Furthermore there are polynomial functions $p(n), q(n)$ such that for each $x \in \mathscr{X}$ we have $|\text{enc}(x)| \leq p(\text{size}(x))$ and for each $y \in \mathscr{Y}(x)$ we have $|\text{enc}(y)| \leq q(\text{size}(x))$.

A verifier V is a function that takes (the binary encodings of) $x \in \mathscr{X}$ and $y \in \mathscr{Y}(x)$, and outputs (the binary encoding of) a value in Ω. The computational problem corresponding to V is then: given x, compute $F(x) = \bigodot_{y \in \mathscr{Y}(x)} V(x,y)$.

This formalism allows us to abstract away the binary encoding operation. For example, we could simply say that the verifier for *graph 3-colorability* takes a graph $G = (V,E)$ and a function $h : V \to \{0,1,2\}$ and tests that the endpoints of each edge are assigned distinct colors, or that the verifier for the problem of counting perfect matchings takes a graph $G = (V,E)$ and an edge set $M \subseteq E$ and tests that every vertex belongs to exactly one element of M.

2 Reductions and Completeness

Reductions constitute the most important tool for reasoning about relative computational complexity of different problems.

Definition 2. If $A : \mathscr{X}_A \to \Omega$ and $B : \mathscr{X}_B \to \Omega$ are two computational problems, a *Karp reduction* from A to B is a function $R : \mathscr{X}_F \to \mathscr{X}_G$ such that $A(x) = B(R(x))$ for all $x \in \mathscr{X}_A$. A *polynomial-time Karp reduction* is a Karp reduction such that there exists an algorithm to compute $R(x)$ running in time $O(p(\text{size}(x)))$ for some polynomial function p.

Karp reductions constitute a particularly simple way of solving problem A using a subroutine that solves B. When thought of in this way, two important limitations of Karp reductions deserve to be mentioned:

1. the subroutine that solves B can only be called once;

2. the subroutine's output must be used as the solution to A.

We write $A \leq_P B$ if there is a polynomial-time Karp reduction from A to B. This is a reflexive, transitive relation because the identity function is a reduction from A to A, and because one can compose polynomial-time reductions from A to B and from B to C to obtain a polynomial-time reduction from A to C. If $A \leq_P B$ and $B \leq_P A$ then we write $A \equiv_P B$ and say that A and B are *computationally equivalent* under polynomial-time Karp reductions. Note that $\equiv_P$ is an equivalence relation whose equivalence classes are partially ordered by $\leq_P$.

If $\mathscr{C}$ is a class of problems, a problem B is called *$\mathscr{C}$-hard* (under polynomial-time Karp reductions) if $A \leq_P B$ for all $A \in \mathscr{C}$, and B is called *$\mathscr{C}$-complete* if $B \in \mathscr{C}$ and B is $\mathscr{C}$-hard. In other words, a $\mathscr{C}$-complete problem is a maximal element of $\mathscr{C}$ under the ordering induced by the polynomial-time reducibility relation, $\leq_P$.

A more general type of reduction called a *polynomial time Turing reduction* or *Cook reduction* does away with these two limitations, allowing a polynomial number of calls to the subroutine that solves B, and allowing any polynomial-time postprocessing procedure to extract the solution to A. In a Turing reduction the calls to B can even be adaptive, meaning that the input to one subroutine call may depend on the output of a previous one. Since every Karp reduction is a Turing reduction, equivalence under polynomial-time Turing reductions is a coarser equivalence relation than $\equiv_P$. This is, in fact, the main advantage of Karp reductions: they allow us to make distinctions between classes of problems that are equally hard to solve using polynomial-time algorithms, but differ in other important ways. For example, NP-complete and coNP-complete problems are equivalent under polynomial-time Turing reductions, but they are believed to be inequivalent under polynomial-time Karp reductions. This reflects a genuine computational difference between an NP-complete problem such as GRAPH 3-COLORABILITY and a coNP-complete problem such as GRAPH NON-3-COLORABILITY: it is easy to prove that a graph G is 3-colorable (just present the 3-coloring), but it seems hard in general to prove that a graph is not 3-colorable.

3 The Cook-Levin Theorem

In this section we present (without proof) a seminal theorem that constitutes the starting point for identifying problems as NP-complete.

Definition 3. If $x_1, \ldots, x_n$ is a list of Boolean variables, a *literal* is any of the $2n$ terms $x_1, \bar{x}_1, \ldots, x_n, \bar{x}_n$ formed by taking the variables and their negations.

A *CNF clause* (of width k) is any term formed by connecting k literals using the Boolean OR operation.

A *CNF formula* is a conjunction of CNF clauses. It is called a k-CNF if each of its clauses has width at most k.

A *truth assignment* for variables $x_1, \ldots, x_n$ is a function $y : \{x_1, x_2, \ldots, x_n\} \to \{0, 1\}$; it extends uniquely to a function $y : \{x_1, \bar{x}_1, \ldots, x_n, \bar{x}_n\} \to \{0, 1\}$ satisfying $y(x_i) \neq y(\bar{x}_i)$ for each i. A truth assignment y satisfies clause C if at least one of the literals in C is mapped to 1 by y; it satisfies CNF formula ϕ if it satisfies every clause of ϕ.

Theorem 1 (Cook-Levin)**.** *For every $k \geq 3$ the following problem known as k-SAT is NP-complete: given a k-CNF formula ϕ, decide whether there exists a truth assignment that satisfies ϕ.*

It is evident that k-SAT belongs to NP because a verifier V given a formula ϕ and truth assignment y can easily check whether y satisfies ϕ. The non-trivial content of the Cook-Levin Theorem is the assertion that *every* other problem in NP reduces to 3-SAT.

Armed with the Cook-Levin Theorem, it becomes quite easy to show that other problems are NP-complete. This is due to the following observation.

Lemma 2. *Suppose B is a problem in NP. If A is an NP-complete problem and $A \leq_P B$ then B is NP-complete.*

Proof. By assumption B belongs to NP. To show that B is NP-hard, consider any other problem C in NP. Since A is NP-complete, we have $C \leq_P A$. By assumption $A \leq_P B$. The proof of the lemma concludes by observing that $C \leq_P B$ because $\leq_P$ is a transitive relation. □

The lemma implies the following recipe for showing that a problem, B, is NP-complete.

1. Show that B belongs to NP. This is usually accomplished by describing a polynomial-time verifier for B.
2. Choose another NP-complete problem, A, and present a polynomial-time Karp reduction from A to B.

In the next section we will elaborate on how the second step is usually accomplished.

4 Designing reductions between NP problems

At the end of the previous section we saw that the key to proving a problem B is NP-complete is to identify another NP-complete problem A (such as 3-SAT) and a polynomial-time Karp reduction R from A to B.

To understand how to construct and analyze a reduction R, it helps to recall the definitions of NP and of Karp reductions. If V_A and V_B are the verifiers for A and B, respectively, then we require that for all $x \in \mathscr{X}_A$, $A(x) = B(R(x))$. Now recall that $A(x) = 1$ if and only if there exists $y \in \mathscr{Y}(x)$ such that $V_A(x, y) = 1$, and that $B(R(x)) = 1$ if and only if there exists $z \in \mathscr{Y}(R(x))$ such that $V_B(R(x), z) = 1$.

Lemma 3. *Suppose A and B are NP problems with verifiers V_A, V_B, respectively. Suppose there exist three functions*

$$R : \mathscr{X}_A \to \mathscr{X}_B \quad T_{AB} : \mathscr{Y}_A \to \mathscr{Y}_B \quad T_{BA} : \mathscr{Y}_B \to \mathscr{Y}_A$$

the relations

$$V_A(x,y) = 1 \Longleftarrow T_{AB}(y) \in \mathscr{Y}(R(x)) \text{ and } V_B(R(x), T_{AB}(y)) = 1$$
$$V_B(R(x),z) = 1 \Longleftarrow T_{BA}(z) \in \mathscr{Y}(x) \text{ and } V_A(x, T_{BA}(z)) = 1$$

hold for all $x \in \mathscr{X}_A, y \in \mathscr{Y}(x), z \in \mathscr{Y}(R(x))$. *If* R *is computable in polynomial time, then* $A \leq_P B$.

Proof. We must show that $A(x) = B(R(x))$ for all x. Since A and B are $\{0,1\}$-valued, this is equivalent to showing that $A(x) = 1 \Leftrightarrow B(R(x)) = 1$. If $A(x) = 1$ it means that there exists $y \in \mathscr{Y}(x)$ such that $V_A(x,y) = 1$. Then, by assumption, $z = T_{AB}(y) \in \mathscr{Y}(R(x))$ and $V_B(R(x),z) = 1$, so $B(R(x)) = 1$. Conversely, if $B(R(x)) = 1$ then there exists $z \in \mathscr{Y}_B(R(x))$ such that $V_B(R(x),z) = 1$. Then, by assumption, $y = T_{BA}(z) \in \mathscr{Y}(x)$ and $V_A(x,y) = 1$ so $A(x) = 1$. □

Although proving $A \leq_P B$ only requires coming up with the function R, in practice to prove the correctness of a reduction it is usually helpful to define the functions T_{AB} and T_{BA} as well. Each of the three functions R, T_{AB}, T_{BA} occupies a distinct conceptual role in analyzing the relationship between problems A and B.

- R "encodes" an instance of problem A using an instance of problem B. Often, the output of the reduction is a structure consisting of disjoint substructures called "gadgets" that correspond to different portions of the input instance $x \in \mathscr{X}_A$. For example, when reducing 3-SAT to a graph problem, the output of the reduction may be a graph made up of disjoint subgraphs corresponding to the variables and the clauses of the given 3-CNF formula.

- T_{AB} "encodes" a candidate solution for an instance of problem A as a candidate solution for the corresponding instance of B. Most often, this is "co-designed" with R, meaning that the creative thought process that leads to defining the function R simultaneously leads to defining T_{AB}. If the definition of R mainly consists of specifying "gadgets", the definition of T_{AB} mainly consists of specifying an "intended usage" for each gadget.

- T_{BA} "decodes" a candidate solution for $R(x)$ to produce a solution for x. Most often, this is the hardest step to get right. If one conceptualizes T_{AB} as defining an "intended usage" for the gadgets of the reduction, then the process of defining T_{BA} typically involves showing that there is no "unintended usage" for the gadgets. In more detail, defining T_{BA} typically involves showing that for any $z \in \mathscr{Y}(R(x))$, either $V_B(R(x),z) = 0$ — in which case $T_{BA}(z)$ can be defined arbitrarily — or else z must use each of the gadgets in $R(x)$ "as intended", which makes it easy to use the information in z to extract a solution $y \in \mathscr{Y}(x)$ such that $V_A(x,y) = 1$. This will all become clearer below when we present some examples.

4.1 3-SAT to Independent Set

An *independent set* in a graph G is a set of vertices such that every edge of G has at most one endpoint in the set. In the INDEPENDENT SET problem we are given a graph G and

a natural number k, and we are asked to decide whether G contains an independent set of size k. This problem clearly belongs to NP: given G and S, a verifier can test that S has k elements and that no edge of G has both of its endpoints in S.

In this section we will design and analyze a reduction from 3-SAT to INDEPENDENT SET. If ϕ is a 3-CNF formula with variables $x_1, \ldots, x_n$ and clauses $C_1, \ldots, C_m$, our goal is to construct a graph G such that truth assignments satisfying ϕ correspond to large independent sets in G. The first step is to figure out how the variables of ϕ will be represented in G. Since a variable has exactly two potential truth values, it is logical to try encoding each variable x_i using a subgraph $U_i \subset G$ that has exactly two maximum independent sets. The easiest way to do this is by defining U_i to consist of a single edge with endpoints u_i^0, u_i^1. The vertices of U_i will be disjoint from all the other gadgets our reduction creates. The two maximum independent sets of U_i are $\{u_i^0\}$ and $\{u_i^1\}$. The "intended use" of the gadget when transforming a truth assignment to a large independent set S is that setting $x_i = 0$ will correspond to including u_i^0 in S and setting $x_i = 1$ will correspond to including u_i^1 in S. Now we come to the issue of representing the clauses of ϕ in the graph G. Again, our aim is to ensure that truth assignments which satisfy ϕ should transform into large independent sets in G. Hence, it is logical to encode a clause C_j as a structure in G containing one or more vertices that can be added to our intended independent set S, but only if at least one of the corresponding variables is assigned a truth value that satisfies the clause. We will accomplish this by representing a clause C_j of width w as a set of w vertices V_j that form a clique in G. (This connectivity pattern ensures that each independent set contains at most one element of V_j.) If C_j contains a literal x_i then u_i^0 will be connected to one of the vertices in V_j, and if C_j contains a literal $\bar{x}_i$ then u_i^1 will be connected to one of the vertices in V_j. Distinct literals in C_j are connected to distinct vertices in V_j.

In summary, the reduction R and transformation T_{AB} transform a 3-CNF formula ϕ and truth assignment y into a graph G, natural number k, and vertex set S defined as follows.

- Let $x_1, \ldots, x_n$ and $C_1, \ldots, C_m$ be the variables and clauses of ϕ.
- The vertex set of G is partitioned into disjoint sets $(U_i)_{i=1}^n$ and $(V_j)_{j=1}^m$ such that $U_i = \{u_i^0, u_i^1\}$ and $|V_j|$ is the width of clause C_j. Each of these $n+m$ sets will be called a "gadget".
- The edge set of G contains two types of edges: every two vertices that belong to the same gadget are joined by an *intra-gadget edge*, and every vertex in a clause gadget V_j belongs to exactly one *inter-gadget edge* whose other endpoint is in a variable gadget U_i.
- If literal x_i appears in C_j, then u_i^0 belongs to an inter-gadget edge whose other endpoint is in V_j. If literal $\bar{x}_i$ appears in C_j then u_i^1 belongs to an inter-gadget edge whose other endpoint is in V_j. Note that in an inter-gadget edge from U_i to V_j, *the superscript of the vertex from U_i corresponds to the truth value for x_i which **does not** satisfy C_j.*
- The desired independent set size, k, is defined to be equal to the number of gadgets, $n + m$.

- If y is a truth assignment, then the vertex set $S = T_{AB}(y)$ is defined to contain u_i^0 for every i such that $y(x_i) = 0$ and S contains u_i^1 for every i such that $y(x_i) = 1$. Finally, for every clause C_j, if at least one literal of C_j is satisfied by y, we choose one such literal (breaking ties arbitrarily if there is more than one literal to choose from) and we include the corresponding vertex of V_j in S.

Note that the reduction runs in polynomial time: for each variable and clause of ϕ, it takes constant time to create the corresponding vertices of G and their intra-gadget edges, and it takes an additional $O(m)$ time to create the inter-gadget edges of G.

Note also that, by construction, if y satisfies ϕ then $S = T_{AB}(y)$ contains exactly one vertex from each gadget, and accordingly we have $|S| = k = n + m$. Furthermore, S is an independent set in G: intra-gadget edges have at most one endpoint in S because S contains only one vertex from each gadget, and any inter-gadget edge $e = (u, v)$ with $u \in U_i$ and $v \in V_j$ has at most one endpoint in S because if $v \in S$ then the truth assignment y satisfies the literal corresponding to v. However, if $u = u_i^a$, then as noted above, the superscript a is the truth value of x_i that *does not* satisfy C_j, so u_i^a does not belong to S.

To finish analyzing the reduction we need to design a reverse transformation T_{BA} such that if S is an independent set of $n + m$ vertices in $G = R(\phi)$ then $y = T_{BA}(S)$ is a truth assignment that satisfies ϕ. Since the vertex set of the graph $R(\phi)$ is partitioned into $n + m$ subsets ("gadgets") and every two vertices belonging to the same gadget are joined by an edge, we know that every independent set in G contains at most one vertex from each gadget, and every independent set of size $n+m$ contains *exactly* one vertex from each gadget. If S is such an independent set, define $y = T_{BA}(S)$ to be the truth assignment that assigns to each variable x_i the truth value a such that $u_i^a \in S$. We must show that y satisfies each clause of ϕ. If C_j is a clause of ϕ then S contains an element $v \in V_j$. The corresponding literal in C_j must be satisfied by y, since otherwise, S would contain both endpoints of the intra-gadget edge originating at v.

4.1.1 3-SAT to Indepedent Set in Graphs of Maximum Degree 3

Consider the special case of the independent set problem in which every vertex has degree at most 3, i.e. every vertex belongs to at most 3 edges. We would like to show that this special case of the problem is still NP-complete. Clearly it belongs to NP, but we must modify the reduction designed above to output a graph with maximum degree 3. The vertices in the "clause gadgets" V_j already have degree 3, but the vertices in the "variable gadgets" U_i can have high degree. In particular, a vertex corresponding to a literal that appears in c clauses will have degree $c + 1$.

To modify the reduction to output a graph with maximum degree 3, we must modify the variable gadgets. Since we still want to use intra-gadget edges to join the variable gadgets to the clause gadgets, and a variable gadget can belong to as many as m clauses, the variable gadgets must be large enough to have m edges protruding from them without violating the maximum-degree constraint. This will necessitate enlarging the variable gadgets to contain a large number of vertices. Recall that the essential property of the variable gadgets U_i was that they should have exactly two maximum independent sets. One way to accomplish this is

by making U_i into a cycle of length $2m$ comprising vertices $u_i^{a,b}$ for $(a, b) \in \{0, 1\} \times [m]$. The edges of the cycle join $u_i^{1,b}$ to $u_i^{0,b}$ and $u_i^{0,b+1}$, where the second superscript is interpreted mod m, so $u_i^{0,m+1}$ denotes the same vertex as $u_i^{0,1}$. Now U_i has exactly two maximum independent sets, both of size m. The intragadget edges now connect V_j with all vertices $u_i^{0,j}$ such that x_i belongs to C_j and all vertices $u_i^{1,j}$ such that $\bar{x}_i$ belongs to C_j. Note that every vertex of G now belongs to two intra-gadget edges and at most one inter-gadget edge, so the maximum degree of G is indeed 3. The modified value of k (the desired size for the independent set) in this reduction is $nm + m$. Since $G = R(\phi)$ can be partitioned into n variable gadgets with maximum independent set size m, and m clause gadgets with maximum independent set size 1, the only independent sets in G with size $nm+m$ are those that contain m vertices of each variable gadget and one vertex of each clause gadget. With this observation in hand, it is easy to modify the proof of correctness of the previous reduction, to show that this new reduction is also correct.