Journal of Universal Computer Science, vol. 7, no. 12 (2001), 1114-1124
submitted:11/5/01, accepted: 23/10/01, appeared: 28/12/01

A Generic NP-hardness Proof for a Variant of Graph Coloring

Hans L. Bodlaender
(Institute of Information and Computing Sciences, Utrecht University
hansb@cs.uu.nl)

Abstract: In this note, a direct proof is given of the NP-completeness of a variant of GRAPH COLORING, i.e., a generic proof similar to the proof of Cook of the NP-completeness of SATISFIABILITY. Then, transformations from this variant of GRAPH COLORING to INDEPENDENT SET and to SATISFIABILITY are shown.
These proofs could be useful in an educational setting, where basics of the theory of NP-completeness must be explained to students whose background in combinatorial optimisation and/or graph theory is stronger than their background in logic. In addition, I believe that the proof given here is slightly easier than older generic proofs of NP-completeness.
Keywords: NP-completeness, computational complexity, graphs, education
Category: F.2.2

1 Introduction

Cook's proof of the NP-completeness of SATISFIABILITY from 1971 [1] is a cornerstone in the theory of the complexity of combinatorial (and other) problems. In an attempt to provide a starting point for the theory of NP-completeness that could be easier to students with a background in combinatorial optimisation and/or graph algorithms, I give in this note a variant of Cook's proof for a variant of the GRAPH COLORING problem.

A major drive of algorithms research is the wish to design algorithms that solve important problems 'fast enough', i.e., a reasonable implementation of the algorithm should give a correct output in reasonable time. It has been observed that there are large classes of combinatorial problems, many with very important applications, that do not have such algorithms available. In some cases, there is a proof that these problems require 'much' time, or that no algorithm exists at all for these problems. In other cases, while no 'efficient' algorithms are known for the problem, also no proof of non-existence of such 'efficient' algorithms is available. This note deals with the most famous (and probably most important) of these classes: the class of NP-complete problems.

The class was defined first by Cook in 1971 in [1]. In that paper, Cook also showed that an NP-complete problem indeed existed: he gave a 'generic' proof of the NP-completeness of SATISFIABILITY, a problem from logic. In this note, I will review some of the most basic notions of NP-completeness, and then give a new generic proof of NP-completeness for a different problem, namely a variant of the GRAPH COLORING problem, called here RESTRICTED GRAPH COLORING. The structure of this proof will be rather similar to Cook's proof, but I belief the generic transformation to a problem from graph theory could be easier to follow for some groups of students, and at a few details, this proof is easier.

To illustrate that the Restricted Graph Coloring problem is also a viable starting point for starting a theory of NP-complete problems, the problem is used as a starting point to proof NP-completeness of Independent Set and Satisfiability.

Other problems can be used as well as starting point for NP-completeness. One of them is the Square Tiling problem (also called the *Jig-Saw* problem, as it models the problem to solve a jig-saw puzzle, for which a generic NP-hardness proof is also known (see [6]).

An interesting discussion of NP-completeness, especially for those unfamiliar with the topic, appears in [4], where Kaye shows that the computer game Minesweeper is NP-complete using a reduction from Satisfiability.

2 Notions and Definitions

2.1 Turing machines

A Turing Machine is a standard model of computation, introduced by Alan Turing in 1936 [5]. In this note, I use the following Turing machine model.

A Turing machine consists of the following parts: a 'finite state control', a 'tape', and a 'read-write head'. The finite state control is a set S of *states*, and at any point in time, the machine is in exactly one of these states. The tape is an infinite array of *cells*, each cell contains at each point in time exactly one *symbol* from an *alphabet* Σ. The *read-write head* (or, in short: the head) is always pointing to exactly one cell at the tape.

There are two special states, s_0 and s_A. s_0 is the *start state*, i.e., when the machine starts its operation, the state of the machine is s_0. s_A is the *accept state*, and it will be explained below what its role is.

The cells are numbered with integers from Z, and it is assumed in this note that the machine starts with the head pointing to cell 0.

The behaviour of the machine is described by a set of *transitions*: this is a set $\delta \subseteq (S \times \Sigma \times S \times \Sigma \times \{-1, 0, +1\})$. This works as follows: at each time step, the machine is in some state s and there is some symbol σ below the head. If there is no 5-tuple in δ of the form $(s, \sigma, \ldots, \ldots, \ldots)$, then the machine halts. In case the machine halts in s_A, then we say that the machine *accepts*, otherwise we say the machine *rejects*. If there is exactly one 5-tuple of this form, say $(s, \sigma, s', \sigma', \Delta)$, then three things happen. First, the symbol s' will be written on the place of the read-write head. (I.e., the symbol s on this place will be replaced by s'.) Secondly, the machine will go to state σ'. Thirdly, if $\Delta \neq 0$, then the read-write head will be moved: if $\Delta = 1$, then it will be moved one position to the right, and if $\Delta = -1$, then it will be moved one position to the left. It will not move when $\Delta = 0$. I.e., if the read-write head pointed to cell i before the step, it will point to cell $i + \Delta$ after the step.

Now, we distinguish two different types of Turing machines. For a deterministic Turing machine, for each $s \in S$ and $\sigma \in \Sigma$, there is at most one tuple of the form $(s, \sigma, \ldots, \ldots, \ldots) \in \delta$, i.e., either there is one transition or none.

In a non-deterministic Turing machine, or in short, NDTM, it is possible that there are more tuples of such a form for given $s \in S$ and $\sigma \in \Sigma$. If there are more tuples, then we say the machine makes a *non-deterministic step*: it can actually choose which of the transitions it will follow. We say the NDTM accepts if there is at least one possible sequence of transitions that leads to halting in the accept state.

There are several closely related models of Turing machines that all have the same expressive power. Here, we will assume that the alphabet Σ consists of three symbols: $\Sigma = \{0, 1, e\}$, where e denotes an 'empty' cell. Now, to each (deterministic or non-deterministic) Turing machine M, we associate the *language, accepted by the machine* M, $L(M)$: for a string $s \in \{0, 1\}^*$, we have $s \in L(M)$, if and only if M accepts when it starts with a tape, that contains the string s in consecutive cells, (i.e., if $s = s_0 s_1 \cdots s_{k-1}$, then for each i, $0 \leq i \leq k-1$, cell i contains symbol s_i), and all other cells are empty (i.e., contain symbol e). Note that the head starts at the first symbol of the string.

2.2 P, NP, and related notions

At first, we restrict ourselves to looking at combinatorial problems that make a decision *yes* or *no*, i.e., problems that can be represented by the set of strings that code the inputs with an answer *yes*.

The time a deterministic Turing Machine M takes on input $s \in \{0, 1\}^*$ is the number of steps that is taken before the machine halts, when it starts with s on the input tape with the head on the first character of s (as above.) The time a non-deterministic Turing machine takes on input s is the maximum number of steps that the machine can take before halting, starting again with s on the input tape.

A Turing Machine is said to use *polynomial time*, if there is a polynomial p, such that on any string $s \in \{0, 1\}^*$ with n characters the machine takes time at most $p(n)$.

The class P is defined as the set of languages L, for which there is a deterministic Turing Machine M that accepts L (i.e., $L(M) = L$), and that uses polynomial time.

The notion is important, because Turing machines can simulate other models of computation, that correspond with real-life computers. These simulations only bring a small (polynomially bounded) overhead with them, which means that the class P captures exactly those problems for which there is an algorithm, solving it in polynomial time on a 'normal' computer.

P is a 'worst-case concept': if there is a problem that has an algorithm that is almost always fast (using polynomial time), but that sometimes must use much more time (e.g., exponential in the size of the input), then we cannot say the problem belongs to the class P, while it still may be efficiently solvable in *almost* all practical cases.

Still, the class P well coincides with what is felt to be *practically doable*. Unfortunately, there are many problems for which we do not know that they belong to P. In several cases, there is a proof that shows that the problem does not belong to the class P, e.g., for some problems, it is known that any algorithm for the problem uses worst case exponential time. There are many other cases where such a proof is not available. Probably the most notable case of a class of problems for which it is unknown whether they belong to P, but for which it is expected they do not are the NP-hard problems.

The class NP is defined as the set of languages L for which there is a non-deterministic Turing Machine that accepts L and that uses polynomial time.

As every deterministic Turing machine is also a non-deterministic Turing machine, we clearly have: P $\subseteq$ NP.

So far, the Turing machines discussed only compute decision problems: problems with an answer *yes* or *no*. However, in order to define the notions of NP-hardness and NP-completeness, we need also to be able to talk about the computations of *functions*. A possible definition is the following: a deterministic Turing machine is said to compute function f, if, when it starts with a tape with a string x on consecutive non-empty cells, then it halts with the string $f(x)$ on consecutive non-empty cells. If the machine uses at most $p(n)$ steps when it starts with an input of length n with p a polynomial, then the function is *polynomial time computable*. It can be seen that this notion of polynomial time computability coincides with the usual notion used in algorithm design. In general, when we denote algorithms, we do not use a Turing machine, but a machine model that is much more resembling the machines that are on our desks, and we allow ourselves to write down programs in a structured language with loops, conditional statements, and use pointers, integer variables, objects, etc. It actually can be proved that all these constructions can be translated to a Turing machine, without crossing the borderline around polynomial time computability. This, however, will not be proven here, but the fact will used implicitly in proofs given later in this note.

Now, we can define the notion of a *polynomial time transformation*. Let $L_1 \subseteq \{0,1\}^*$ and $L_2 \subseteq \{0,1\}^*$ be languages. A function $f : \{0,1\}^* \rightarrow \{0,1\}^*$ is a polynomial time transformation from L_1 to L_2, if there is a deterministic Turing machine that computes f in polynomial time, and for all $x \in \{0,1\}^*$, $x \in L_1$, if and only if $x \in L_2$.

A language $L \subseteq \{0,1\}^*$ is said to be *NP-hard*, if for every language L' in NP, there is a polynomial time transformation from L' to L. L is said to be *NP-complete*, if it is NP-hard and $L \in$ NP.

The question: *Is there a polynomial time algorithm for an NP-complete problem?* has many equivalent formulations. Usually, it is expressed as the P=NP-question. This problem is one of the most important unsolved problems in theoretical computer science and mathematics, and despite many efforts for many years by many researchers, it still is unsolved. Many researchers think that the answer will be that P $\neq$ NP — such a result would imply that all NP-hard problems cannot be solved in polynomial time. Thus, an NP-hardness or NP-completeness proof for a problem L can be seen as strong evidence for the conjecture that L does not have a polynomial time algorithm solving it. Much more on this issue can be said; readers unfamiliar with the topic are advised to consult e.g. [3].

2.3 Combinatorial problems

Below, three combinatorial problems are given. The first one is a rather arbitrary special case of the well known Graph Coloring problem, and the latter two are well known problems.

Restricted Graph Coloring
Instance: Directed graph $G = (V, E)$, set of colors C, for each edge $(v, w) \in E$, a set of allowed color-pairs $A(v, w) \subseteq C \times C$, and for each vertex $v \in V$, a set of allowed colors $B(v) \subseteq C$.

Question: Is there a coloring $c : V \to C$, such that for every edge $(v,w) \in E$, $(c(v), c(w)) \in A(v,w)$, and for every vertex $v \in V$, $c(v) \in B(v)$?

Independent Set
Instance: Undirected graph $G = (V,E)$, integer $K \leq |V|$.
Question: Does G have an independent set of size at least K, i.e., is there a set $W \subseteq V$ with for all $\{v,w\} \in E$: $v \notin W$ or $w \notin W$, and $|W| \geq K$?

Satisfiability
Instance: Set of Boolean variables $x_1, \ldots, x_n$; formula F of the form $\bigwedge_{1 \leq i \leq r}(\bigvee_{1 \leq j \leq c_i} l_{ij}))$, where each l_{ij} is of the form x_i or $\neg x_i$ with $1 \leq i \leq n$.
Question: Can we assign to each variable x_i, $1 \leq i \leq n$ a value *true* or *false*, such that formula F becomes true?

None of these problems is a language of the form $L \subseteq \{0,1\}^*$. In order to fit these problems into the formal framework as given above with Turing machines, we need to *code* the inputs of the problems as binary strings. Thus, we assume a 'natural' mapping of the inputs to a binary string. Note that such a mapping will be anyhow used for any representation of the input on a computer, so we do not make any strange assumption when we stipulate that we have such a coding. For details, see e.g., [3, Chap. 2.1].

3 A Generic Proof of NP-completeness of Restricted Graph Coloring

Now, the main result of this note is given. The construction is similar to the proof of Cook [1] of the NP-completeness of Satisfiability.

Theorem 1. Restricted Graph Coloring *is NP-complete.*

Proof. To proof NP-completeness for a problem R, two things must be proved. First, it must be shown that the problem belongs to NP, and secondly, from every problem Q in NP, there must be a polynomial time transformation from Q to R. Membership of Restricted Graph Coloring in NP is not so hard to prove, and omitted from this note.

Now, it will be shown that for every problem R in NP, there is a polynomial time transformation from R to Restricted Graph Coloring.

Let R be an arbitrary problem in NP. Because R belongs to NP, we know that there is a non-deterministic Turing Machine M that recognises R, and that uses on an input of n bits at most $p(n)$ time steps, where p is a polynomial. We make the following assumptions on M:

- M does not access any tape location before location 0.
- M has one accepting state s_A with no transitions out of s_A.
- S is the set of states of M.
- The alphabet used for the tape is Σ.
- The head starts at time 0 at position 0, and M starts in state s_0.

– The set of transitions is $T \subseteq S \times \Sigma \times S \times \Sigma \times \{-1, 0, +1\}$.
– When there are two different transitions possible from a (state, symbol)-pair, then they can differ in the result state and in the direction the head moves, but not in the symbol written, i.e., if $(s, \sigma, s', \sigma', \delta) \in T$ and $(s, \sigma, s'', \sigma'', \delta') \in T$, then $\sigma' = \sigma''$. (Note that one can easily simulate a TM to another TM with this property that uses at most twice as many time steps: split a non-deterministic step in a non-deterministic step that only moves non-deterministically to a different state, but does not change the head or the symbol below the head, and a deterministic step that writes a symbol and possibly moves the head.)

To the set of transitions T, we add transitions that keep the TM running in the accepting state: for each $\sigma \in \Sigma$, add the transition $(s_A, \sigma, s_A, \sigma, 0)$. The original Turing Machine halts at or before time $p(n) - 1$ in state s_A, if and only if the modified TM is in state s_A at time $p(n) - 1$. Let T' be this new set of transitions, and let M' be the modified TM.

Note that M' can only access tape locations 0, 1, ..., $p(n) - 1$.

A description of the construction of G, C, A, and B now follows. This description is interleaved with an explanation of why this construction is made in this way.

The graph $G = (V, E)$ has the following kind of vertices:

– We have vertices $v_{t,l}$, $0 \leq t \leq p(n) - 1$, $0 \leq l \leq p(n) - 1$, i.e., for each time step t, $0 \leq t \leq p(n) - 1$ and for each location on the tape l, $0 \leq l \leq p(n) - 1$, we have a vertex. This vertex will denote what symbol can be found on location l at time step t.
– We have vertices s_t, $0 \leq t \leq p(n) - 1$, i.e., for each time step t, we have one vertex. This vertex will get a color that denotes what position the head is, what is the current state, and (duplicating information for a vertex of the first set) what symbol is below the head.

There are two kinds of colors:

– Each pair (σ, b), with $\sigma \in \Sigma$ and $b \in \{true, false\}$ is a color in C. Write $X = \Sigma \times \{true, false\}$.
– Each triple $(s, l, \sigma) \in S \times \{0, 1, \ldots, p(n) - 1\} \times \Sigma$ is a color in C. Write $S' = S \times \{0, 1, \ldots, p(n) - 1\} \times \Sigma$.

So, $C = X \cup S'$.

Now, the description of B is given.

– For vertices $v_{0,l}$, $0 \leq l \leq p(n) - 1$, if the tape has symbol σ at position l before the run of the Turing Machine, i.e., at time 0, then take $B(v_{0,l}) = \{(\sigma, false)\}$ for $l \geq 0$, and take $B(v_{0,0}) = \{(\sigma, true)\}$. This forces that a run of M', simulated by the coloring indeed starts with the correct initial configuration. The Boolean values denote whether the head is at that symbol during the corresponding time step.
– For vertices $v_{t,l}$, $1 \leq t \leq p(n) - 1$, $0 \leq l \leq p(n) - 1$, take $B(v_{t,l}) = X$. The first argument of the color given to these vertices denotes the symbol that is on position l at time t during the run of M', simulated by the coloring of G, and the second argument is a boolean that tells whether the head is at that location.

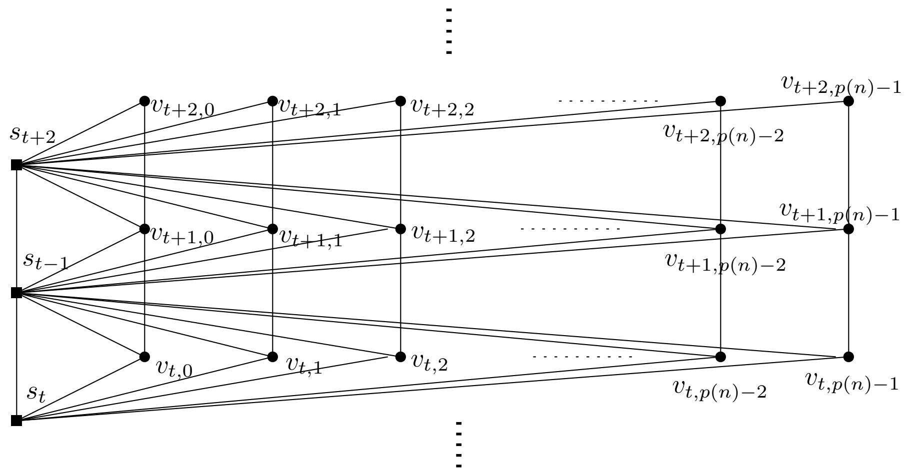

Figure 1: The constructed graph G: detail

– For vertex s_0, $B(s_0) = \{(s_0, 0, \sigma)\}$, where σ is the symbol at position 0 of the input, i.e., at time 0. This makes that the color of s_0 denotes that the head starts at position 0, and that the automaton starts in state s_0.
– For vertices s_t, $1 \leq t \leq p(n) - 2$, take $B(s_t) = S'$. This forces that at each time during the run of M', simulated by the coloring of G, the head is at a location, and the machine is in a state.
– For vertex $s_{p(n)-1}$, take $B(s_{p(n)-1}) = \{(s_A, l, \sigma) \mid 0 \leq l \leq p(n) - 1,\ \sigma \in \Sigma\}$. This forces that the run of M', simulated by the coloring of G is in an accepting state at time $p(n) - 1$.

Edges, and the allowed color-pairs for edges are used to force that the coloring of G simulates a run of M' that is allowed by the set of transitions.

We have the following kind of edges:

– Edges that check consistency at a certain time step. For each time step t, $0 \leq t \leq p(n) - 1$, and each location l, $0 \leq l \leq p(n) - 1$, we take an edge $(s_t, v_{t,l})$, with $A(s_t, v_{t,l}) = \{((s, l, \sigma), (\sigma, true)) \mid \sigma \in \Sigma, s \in S\} \cup \{((s, l', \sigma), (\sigma', false)) \mid \sigma, \sigma' \in \Sigma,\ s \in S,\ 0 \leq l' \leq p(n) - 1,\ l' \neq l\}$. This edge forces that when s_t gets a color (s, l, σ), then $v_{t,l}$ gets color $(\sigma, true)$, and all $v_{t,l'}$, $l \neq l'$, get the value $false$ as second argument. So, these check consistency of the color of s_t with the colors of the vertices $v_{t,l}$ under the 'interpretation' given to these colors.
– Edges that check that the tape is not changed on locations without the head. For each time step t, $0 \leq t \leq p(n) - 2$, and each location l, $0 \leq l \leq p(n) - 1$, take an edge $(v_{t,l}, v_{t+1,l})$ with $A(v_{t,l}, v_{t+1,l}) = \{((\sigma, false), (\sigma, b)) \mid b \in \{true, false\},\ \sigma \in \Sigma\} \cup \{((\sigma, true), X) \mid \sigma \in \Sigma\}$. This makes sure that the symbol at a tape-location can only be changed at a time step when the head is at that location.

– Edges that check that symbols are written correctly. For each time step t, $0 \leq t \leq p(n) - 2$, and each location l, $0 \leq l \leq p(n) - 1$, we take an edge $(s_t, v_{t+1,l})$. $A(s_t, v_{t+1,l})$ is the union of the following two sets:
  - $\{((s,l,\sigma),(\sigma',b)) \mid b \in \{true, false\},\ (s,\sigma,s',\sigma',\delta) \in T$ for some $s' \in S$, $\delta \in \{-1, 0, +1\}\}$. If we are in state s and symbol σ is below the head at position l, then at time step $l+1$, the symbol at position l must be the unique symbol for which there is a transition of the form $(s,\sigma,s',\sigma',\delta)$.
  - $\{((s,l',\sigma),(\sigma',b)) \mid (s,l',\sigma) \in S',\ (\sigma',b) \in X, l \neq l'\}$. If s_t has the color (s,l',σ) and $l \neq l'$, then $v_{t+1,l}$ denotes the content of a cell at time $t+1$ for a location where the head was not at time t. In such a case, we do not need to enforce anything that was not already enforced by other by other conditions.

– Edges that check that states are changed and the head is moved according to the transition rules. For every time t, $0 \leq t \leq p(n) - 1$, we have an edge (s_t, s_{t+1}), with $A(s_t, s_{t+1}) = ((s,l,\sigma),(s',l',\sigma')) \mid \sigma' \in \Sigma,\ (s,\sigma,s',\sigma'',l'-l) \in T$ for some $\sigma'' \in \Sigma\}$. If we are in state s at time t and σ is under the head at location l, and we go to state s' with the head at location l', then the head is moved $l'-l$, and a transition of the form $(s,\sigma,s',\sigma'',l'-l)$ must belong to T. (We do not specify what symbol is written. This was checked by the previous set of edges.)

This finishes the construction. We now have:

Claim 2 *There is a coloring of G, satisfying the conditions posed by color-pairs A and color sets B, if and only if the non-deterministic Turing Machine M accepts the input.*

The claim can be proved by using induction, and proving the following stronger claim.

Claim 3 *There is a coloring c of G consistent with the conditions posed by A and B, if and only if there is a possible execution of M', such that*

– *σ is at location l at time t and the head is not at location l at time t, if and only if $c(v_{t,l}) = (\sigma, false)$ ($0 \leq t \leq p(n) - 1$, $0 \leq l \leq p(n) - 1$, $\sigma \in \Sigma$.*

– *σ is at location l at time t and the head is at location l at time t, if and only if $c(v_{t,l}) = (\sigma, true)$ ($0 \leq t \leq p(n) - 1$, $0 \leq l \leq p(n) - 1$, $\sigma \in \Sigma$.*

– *M' is in state s at time t with the head at location l, with symbol σ at head-location l, if and only if $c(s_t) = (s,l,\sigma)$ ($0 \leq t \leq p(n) - 1$, $0 \leq l \leq p(n) - 1$, $\sigma \in \Sigma$.*

Comments given with the construction show the correctness of this claim.

Now, we have a polynomial time transformation from problem R to Restricted Graph Coloring.

4 Transformations

Generic proofs of NP-completeness are in general much harder than proving NP-completeness with transformations to problems known to be NP-complete. In this section, a few such transformations are given. These also show that Restricted Graph Coloring is a reasonable choice for starting proofs of NP-completeness with.

Theorem 4. INDEPENDENT SET *is NP-complete.*

Proof. Membership in NP is easy to see.

To show NP-hardness, a transformation from RESTRICTED GRAPH COLORING is used.

Let $G = (V, E)$, C, A, and B be given as input of the RESTRICTED GRAPH COLORING problem.

A graph $H = (W, F)$ will now be constructed. The set of vertices of H is $W = \{w_{v,c} \mid v \in V,\ c \in B(v)\}$. The set of edges F of H is the union of the following edge sets:

- $\{\{w_{v,c}, w_{v,c'}\} \mid v \in V,\ c, c' \in B(v),\ c \neq c'\}$. I.e., for every v, the vertices of the form $w_{v,c}$ form a clique.
- $\{\{w_{v,c}, w_{v',c'}\} \mid (v, v') \in E,\ (c, c') \notin A(v, v')\}$.

Claim 5 *H has an independent set of size at least $|V|$, if and only if G has a graph coloring satisfying the conditions posed by A and B.*

Proof. Suppose G has such a coloring c. Now, let $X = \{w_{v,c(v)} \mid v \in V\}$. One easily checks that X is an independent set of size $|V|$.

Suppose H has an independent set X of size at least $|V|$. For each vertex $v \in V$, X can contain at most one vertex of the form $w_{v,c}$ (because these vertices form a clique for given v). As $|X| \geq |V|$, we have that X contains exactly one vertex of the form $w_{v,c}$ for each v. Now, write $c(v) = c$, if $w_{v,c} \in X$. This is a function $V \to C$. By construction, $c(v) \in B(v)$. For every edge $(v, v') \in E$, $(w_{v,c(v)}, w_{v',c(v')}) \notin E$ (as X is an independent set), hence $(c(v), c(v')) \in A(v, v')$.

As the transformation can be done in polynomial time, NP-hardness of INDEPENDENT SET follows.

Theorem 6. SATISFIABILITY *is NP-complete.*

Proof. Clearly, SATISFIABILITY belongs to NP.

To show NP-hardness, a transformation from RESTRICTED GRAPH COLORING is used.

Let $G = (V, E)$, C, A, and B be given as input the RESTRICTED GRAPH COLORING problem.

This input is now transformed to an instance of SATISFIABILITY. This latter instance uses the following set of Boolean variables: $X = \{x_{v,c} \mid v \in V \mid c \in B(v)\}$. Intuitively, $x_{v,c}$ is the variable that denotes whether v gets color c.

The set of clauses I consists of the following subsets:

- Clauses that make sure every vertex v has at least one color from $B(v)$. For every vertex $v \in V$, take a clause $\bigvee_{c \in B(v)} x_{v,c}$.
- Clauses that make sure every vertex has at most one color. For every vertex $v \in V$, and every two distinct colors $c, c' \in B(v)$, $c \neq c'$, take a clause $\neg x_{v,c} \vee \neg x_{v,c'}$.
- Clauses that make sure that for every edge (v, v'), the colors given to v and v' do not form a pair outside of $A(v, v')$. For every edge $(v, v') \in E$, and every $c \in B(v)$, $c' \in B(v')$ with $(c, c') \notin A(v, v')$, take a clause $\neg x_{v,c} \vee \neg x_{v',c'}$.

Claim 7 *G has a coloring fulfilling the conditions posed by A and B, if and only if the set of variables X has a truth assignment that makes all clauses in I true.*

Proof. Suppose we have such a truth assignment. Now, set $c(v) = c$, iff $x_{v,c} = true$. By the first set of clauses, such a c exists for every $v \in V$. By the second set of clauses, at most one such c exists per v. So, $c : V \to C$ is a function. By construction $c(v) \in B(v)$ for each $v \in V$. Now, for every $(v, v') \in E$, $(c(v), c(v')) \in A(v, v')$. Suppose there is an edge $(v, v') \in E$ with $(c(v), c(v')) \notin A(v, v')$. Then the clause $\neg x_{v,c} \vee \neg x_{v',c'}$ is not satisfied, contradiction.

Suppose we have such a coloring c. Set $x_{v,c}$ to true, iff $c(v) = c$. One easily verifies that this truth assignment satisfies all clauses.

Thus, the construction given is a polynomial time transformation from SATISFIABILITY to RESTRICTED GRAPH COLORING, and hence the theorem follows.

5 Conclusions

This note showed the NP-completeness of some problems using a generic proof. For purposes of presentation of the theory of NP-completeness to students with a background in combinatorial optimisation, operations research, or graph theory, this approach could be more insightful, at least for some groups of students, than following the classic proof of Cook. Those familiar with Cook's proof [1], (see also [3]) may notice the inspiration I had from that proof. I believe that translating Cook's proof to the RESTRICTED COLORING PROBLEM gives a slightly simpler construction. There is, however, ample room for improvement.

In my view, the main shortcoming of the presentation here is the reliance on Turing machines. As a model of computing, Turing machines are much more distinct from computing devices that are nowadays used than, e.g., the model of a Random Access Machine. Thus, to argue that NP-completeness indeed is viable evidence for the conjectured non-existence of polynomial time / efficient algorithms for the problem, one needs to argue that a simulation of a machine like a Random Access Machine on a Turing machine has only a polynomial loss of efficiency - a fact that needs a detailed proof and may be not very appealing to the group of students the proofs in this note were meant for. (The argument was briefly hinted at in Section 2, but not given.) Thus, a nice question could be to give a generic proof of NP-completeness for some graph problem based upon the Random Access Machine model. It can be observed that the construction used here actually does give some freedom, and it seems that using a RAM with words of $O(\log n)$ bits could be used instead of Turing Machines. An informal generic proof of the NP-completeness of CIRCUIT SATISTIABILITY appears in [2].

A different option could be to use hypergraphs instead of graphs for the first generic proof. Consider the following problem:

RESTRICTED HYPERGRAPH COLORING
Instance: Hypergraph $G = (V, E)$, with for every $e \in E$, $|e| \leq 3$; set of colors C; for every $e \in E$, set of functions C_e, of the form $f : e \to C$.
Question: Is there a coloring $c : V \to C$, such that for all $e \in E$, the restriction of c to e belongs to C_e?

A generic proof of NP-completeness of this problem is actually somewhat shorter than the proof of RESTRICTED GRAPH COLORING, but can go along the same lines. However, the problem seems harder to understand, and may be less easy to start reductions from.

Acknowledgement

This research was partially supported by EC contract IST-1999-14186: Project ALCOM-FT (Algorithms and Complexity - Future Technologies).

References

1. S. A. Cook. The complexity of theorem-proving procedures. In *Proc. 3rd Ann. Symp. on Theory of Computing*, pages 151–158, New York, 1971. ACM.
2. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Mass., USA, 1989.
3. M. R. Garey and D. S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, 1979.
4. R. Kaye. Minesweeper is NP-complete. *The Mathematical Intelligencer*, 22:9–15, 2000.
5. A. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc. Ser. 2*, 42:230–265, 1936.
6. P. van Emde Boas. The convenience of tilings. In A. Sorbi, editor, *Complexity, Logic and Recursion Theory*, pages 331–363. Lecture Notes in Pure and Applied Mathematics, vol. 187, Marcel Dekker Inc., 1997.