A Refutation of Guinea’s “Understanding SAT is in P”

Jackson Abascal* and Shir Maimon*
Department of Computer Science
University of Rochester
Rochester, NY 14627, USA
November 15, 2017

Abstract
In this work, we summarize and critique the paper “Understanding SAT is in P” by Alejandro Sánchez Guinea [SG15]. The paper claims to present a polynomial-time solution for the NP-complete language 3-SAT. We show that Guinea’s algorithm is flawed and does not prove 3-SAT is in P.

1 Introduction
The problem of finding a satisfying assignment to a boolean formula, referred to as SAT, has strong historical significance in computer science. It was the first natural problem proven to be NP-complete [Coo71] and remains an area of active theoretical and practical interest.

Alejandro Sánchez Guinea’s paper “Understanding SAT is in P” [SG15] claims to provide a polynomial time algorithm for 3-SAT [SG15], a restricted version of SAT. Like SAT, 3-SAT is NP-complete [Kar72].

If correct, Guinea’s paper would resolve the P-versus-NP problem by showing that \( P = NP \). However, we show in Section 3 that the algorithm is based on multiple erroneous assumptions and admits counterexamples.

We begin with some definitions. A literal is either a boolean variable \( x \) or a negated variable \( \neg x \). A clause is an expression of literals connected by \( \lor \) (logical or) operators. We say a clause is satisfied if it evaluates to true. For example, one satisfying assignment of \( (x \lor \neg y \lor z) \) is \( x = \text{True}, y = \text{True}, \) and \( z = \text{False} \). The 3-SAT problem is defined as follows: Given a collection of variables and clauses where each clause contains exactly three literals, determine whether there exists an assignment of the variables that satisfies every clause. Distinct clauses may share variables.

*Work supported in part by a CRA-W CREU grant.
1This critique is written with respect to the most recent available revised version: Version 4 (the version of September 16, 2016) of arXiv.org report 1504.00337 [SG15].
Guinea defines 3-SAT in a somewhat nonstandard fashion by disallowing any clause that contains the same variable twice. This is not an issue, as standard 3-SAT can be reduced in polynomial time to Guinea’s form (see Appendix).

Given an instance of 3-SAT, Guinea defines an understanding \( \bar{u} \) as a function from the set of literals to the set \( \{t, f, \epsilon\} \). For any literal \( \lambda \), we say that \( \lambda \) is “true” under \( \bar{u} \) if \( \bar{u}(\lambda) = t \), is “false” if \( \bar{u}(\lambda) = f \), and is “free” otherwise. The definition as Guinea presents it does not enforce consistency between literals and their negations. For example, both \( x \) and \( \neg x \) may be true under \( \bar{u} \).

When evaluating a 3-SAT expression with respect to an understanding \( \bar{u} \), we consider a clause to be satisfied if any of its literals are true under \( \bar{u} \). Note that if an understanding satisfies the clauses of an expression, this does not mean that there exists a satisfying assignment in the traditional sense. We call two understandings equivalent if they satisfy the same set of clauses.

Let \( \phi \) be a clause, and for a literal \( \lambda \) in \( \phi \) define the concept of \( \lambda \) in \( \phi \) as the multiset \( \{\bar{u}(l_1), \bar{u}(l_2)\} \) where \( l_1, l_2 \) are the two literals in \( \phi \) which are not \( \lambda \). A literal \( \lambda \) has no concept in \( \phi \) if \( \lambda \notin \phi \). Although it is never explicitly stated, Guinea assumes that the literals and the clause associated with a concept are implicitly stored and can be recovered. We say that a concept \( C \) is of type \( C^* \) if it contains the true symbol \( t \), and of type \( C^+ \) otherwise. Intuitively, if a concept with respect to \( \lambda \) is of type \( C^+ \) then \( \lambda \) can be set to free, as some other literal already satisfies its associated clause. If it is of type \( C^+ \) then \( \lambda \) must evaluate to true in order to satisfy the clause (assuming no other values change).

For any set of clauses \( \phi \) and literal \( \lambda \), define \( \bar{C}[\lambda] \) as the set of concepts \( C \) such that \( C \) is the concept of \( \lambda \) in \( \phi \) for some \( \varphi \in \phi \). Define \( \bar{C}[\lambda]^− \) as the set of concepts of type \( C^+ \) in \( \bar{C}[\neg \lambda] \). Intuitively, \( \bar{C}[\lambda]^− \) is the set of concepts associated to clauses which would be unsatisfied if \( \lambda \) were true. A set of concepts is of type \( \bar{C}^* \) if all of the concepts in it are of type \( \bar{C}^* \), and a set of concepts is of type \( \bar{C}^+ \) if at least one of the concepts within it are of type \( \bar{C}^+ \).

An understanding is called defined if it satisfies the following property:

\[
\bar{u}(\lambda) = \begin{cases} 
\epsilon & \text{if } \bar{C}[\lambda] \text{ is empty or } (\bar{C}[\lambda]^− \text{ is empty and } \bar{C}[\lambda] \text{ is of type } \bar{C}^*) \\
t & \text{if } \bar{C}[\lambda] \text{ is of type } \bar{C}^+ \text{ and } \bar{C}[\lambda]^− \text{ is empty} \\
f & \text{if } \bar{C}[\lambda]^− \text{ is not empty and } \bar{C}[\lambda] \text{ is not of type } \bar{C}^+
\end{cases}
\]

The case where \( \bar{C}[\lambda] \) is of type \( \bar{C}^+ \) and \( \bar{C}[\lambda]^− \) is not empty is purposefully omitted. In this case \( \bar{u} \) is considered undefined. If an understanding is defined with respect to some set of clauses then \( \lambda \) and \( \neg \lambda \) will not have conflicting assignments. In particular, \( \bar{u}(\lambda) = t \) implies \( \bar{u}(\neg \lambda) = f \), and \( \bar{u}(\lambda) = \epsilon \) implies \( \bar{u}(\neg \lambda) = \epsilon \). A set of clauses \( \phi \) is satisfiable if and only if there is some understanding defined with respect to \( \phi \). Guinea’s algorithm claims to produce a defined understanding for the clauses of an instance of 3-SAT if and only if the instance is satisfiable.

Observe that although the existence of a satisfying assignment implies the existence of a defined understanding, this understanding will not necessarily assign to each literal the
value given by the satisfying assignment. In a defined understanding, a literal \( \lambda \) maps to \( t \) only if there is a clause in which \( \lambda \) is the only literal assigned to true. Furthermore, \( \lambda \) can only be assigned to \( f \) if there is a clause in which \( \neg \lambda \) would be the only literal assigned to true. In any other case, \( \lambda \) must be free. For example, with respect to the 3-SAT instance consisting of the single clause \((x \lor y \lor z)\), the understanding that assigns each of \( x \), \( y \), and \( z \) to \( t \) is not defined, whereas the understanding that assigns \( x \) to \( t \) and both \( y \) and \( z \) to \( \varepsilon \) is considered defined.

2 Algorithms

Guinea’s descriptions of his algorithms are often difficult to read and occasionally ambiguous. We have written in this section a more precise exposition of each of the algorithms presented in the paper.

Guinea presents three algorithms: Algorithm G, Algorithm D, and Algorithm \( \tilde{U} \). Algorithm \( \tilde{U} \) is claimed to ultimately solve 3-SAT in polynomial time. He also defines an operation \( \langle \text{Compute } \tilde{u} \rangle \), which is used in all three algorithms. We will describe \( \langle \text{Compute } \tilde{u} \rangle \) first.

2.1 \( \langle \text{Compute } \tilde{u} \rangle \)

The exact description of \( \langle \text{Compute } \tilde{u} \rangle \) in the paper is as follows:

\[
\langle \text{Compute } \tilde{u} \rangle = \text{Compute } \tilde{u} \text{ for each literal } \lambda \text{ and its negation for which the type } \tilde{C}[\lambda] \text{ has changed, until there is no change of type on any subset of concepts of } \tilde{C} \quad \text{[SG15]}. 
\]

We found this description ambiguous. Because Guinea calls \( \langle \text{Compute } \tilde{u} \rangle \) an operation rather than an algorithm, we believe he means that we can insert its description anywhere he writes “\( \langle \text{Compute } \tilde{u} \rangle \)” and interpret the description in a way that makes sense in context. This would imply that for each \( \langle \text{Compute } \tilde{u} \rangle \) call, the first literals acted on are those for which the type of \( \tilde{C}[\lambda] \) has changed in the operations directly preceding the call. Thus we can consider \( \langle \text{Compute } \tilde{u} \rangle \) as a function taking in some set of literals whose values under \( \tilde{u} \) we would like to modify.

Furthermore, it is unclear what it means to “Compute \( \tilde{u} \) for a literal \( \lambda \).” We have taken this phrase to be equivalent to the procedure \( \text{RECALCULATE}(\tilde{u}, \tilde{C}, \lambda) \) defined below, where \( \tilde{u} \) is an understanding, \( \tilde{C} \) is a set of concepts, and \( \lambda \) is a literal.

1: \textbf{procedure} \( \text{RECALCULATE}(\tilde{u}, \tilde{C}, \lambda) \) 
2: \quad \textbf{if} \( \tilde{C}[\lambda] \) is empty \textbf{or} \( (\tilde{C}[\lambda])^- \) is empty \textbf{and} \( \tilde{C}[\lambda] \) is of type \( \tilde{C}^* \) \textbf{then} 
3: \quad \textbf{return} \( \varepsilon \) 
4: \quad \textbf{else if} \( \tilde{C}[\lambda] \) is of type \( \tilde{C}^+ \) \textbf{and} \( \tilde{C}[\lambda]^- \) is empty \textbf{then} 
5: \quad \textbf{return} \( t \) 
6: \quad \textbf{else if} \( \tilde{C}[\lambda]^- \) is not empty \textbf{and} \( \tilde{C}[\lambda] \) is not of type \( \tilde{C}^+ \) \textbf{then} 
7: \quad \textbf{return} \( f \)
We can now define our interpretation of \(\langle \text{Compute } \tilde{u} \rangle\). We have defined it as a function that takes as arguments an understanding \(\tilde{u}\), a set of concepts \(\tilde{C}\), and an initial set \(S\) of literals to be recomputed.

```plaintext
1: procedure Compute(\(\tilde{u}\), \(\tilde{C}\), \(S\))
2: while \(S\) is not empty do
3:   Take any \(\lambda \in S\) and remove \(\lambda\) from \(S\)
4:   Set \(\tilde{u}(\lambda) := \text{Recalculate}(\tilde{u}, \tilde{C}, \lambda)\)
5:   if \(\tilde{u}(\lambda) = \text{UNDEFINED}\) then
6:     return \(\text{UNDEFINED}\)
7:   end if
8:   For any concept \(C\) in \(\tilde{C}\) containing \(\lambda\), update \(C\) using \(\tilde{u}\)
9:   for all literals \(\lambda'\) for which the type of \(\tilde{C}[\lambda']\) changed in line 8 do
10:      Add \(\lambda'\) to \(S\) ⊲ \(\tilde{C}\) changed, so there may be literals whose concepts changed
11:   end for
12: end while
13: return \((\tilde{u}, \tilde{C})\)
14: end procedure
```

\(\langle \text{Compute } \tilde{u} \rangle\), while ambiguously defined, is not the basis of our refutation. Under any reasonable interpretation of \(\langle \text{Compute } \tilde{u} \rangle\), our counterexamples remain valid.

### 2.2 Algorithm G

Algorithm G takes as arguments a set of clauses \(\phi\), an understanding \(\tilde{u}\), a set of concepts \(\tilde{C}\) defined with respect to \(\tilde{u}\), and a literal \(\lambda\) that is free under \(\tilde{u}\). Let \(\phi_\lambda \subseteq \phi\) be the clauses in which \(\lambda\) or \(\neg \lambda\) appears. Guinea claims the algorithm will return “True” if and only if there is an understanding \(\tilde{u}_\lambda\) defined with respect to \(\phi_\lambda\) such that \(\tilde{u}_\lambda(\lambda) = t\). Guinea does not specify how exactly to assign \(\tilde{u}(l_1)\) and \(\tilde{u}(l_2)\) in line 6 meaning the algorithm is ill-defined. Due to Algorithm G’s heavy reliance on \(\langle \text{Compute } \tilde{u} \rangle\), rather than analyzing this algorithm our refutation will assume its validity.

```plaintext
1: procedure Algorithm_G(\(\phi, \tilde{u}, \tilde{C}, \lambda\))
2:   Set \(\tilde{u}(\lambda) := t\)
3:   Set \(\tilde{u}(\neg \lambda) := f\)
4:   for all concepts \(C\) in \(\tilde{C}[\lambda]\) do
5:     Let \(l_1\) and \(l_2\) be the literals in \(C\)
6:     Set both \(\tilde{u}(l_1)\) and \(\tilde{u}(l_2)\) to either \(\epsilon\) or \(f\)
7:     Let \(S\) be the literals \(\lambda\) such that \(\tilde{C}[\lambda]\) changed type in the previous step
8:     Set \((\tilde{u}', \tilde{C}') := \text{Compute}(\tilde{u}, \tilde{C}, S)\)
9:     if \((\tilde{u}', \tilde{C}') \neq \text{UNDEFINED}\) then
10:        return True ⊲ The algorithm terminates successfully
```

4
2.3 Algorithm D

Algorithm D claims to do the following: Given an understanding \( \tilde{u} \), a set of concepts \( \tilde{C} \), and a literal \( \lambda \) that is false under \( \tilde{u} \), construct a defined understanding \( \tilde{u}' \) equivalent to \( \tilde{u} \) such that \( \lambda \) is free under \( \tilde{u}' \), or report that no such \( \tilde{u}' \) exists. Guinea refers to a set \( H \) without explicitly specifying its purpose, saying only \( H \) is “considered empty, if not given” [SG15]. To properly call Algorithm D recursively, we modify the definition to be dependent on \( H \) by adding the condition that \( \tilde{u}'(\lambda) = \tilde{u}(\lambda) \) if \( \lambda \in H \).

1: \textbf{procedure} Algorithm\_D(\( \tilde{u}, \tilde{C}, \lambda, H \))
2: \hspace{1em} \textbf{while} there is a concept in \( \tilde{C}[\lambda]^- \) not already considered \textbf{do}
3: \hspace{2em} Take any \( C \in \tilde{C}[\lambda]^− \) not already considered
4: \hspace{2em} Let \( Q \) be the set of literals \( l \) in \( C \) such that \( l /\not\in H \)
5: \hspace{2em} \textbf{while} \( Q \) is not empty \textbf{do}
6: \hspace{3em} Take any \( l \in Q \) and remove \( l \) from \( Q \)
7: \hspace{3em} Set \( (\tilde{u}', \tilde{C}') = \text{FAIL} \)
8: \hspace{3em} \textbf{if} \( \tilde{u}(l) = f \) \textbf{then}
9: \hspace{4em} Set \( (\tilde{u}', \tilde{C}') \leftarrow \text{Algorithm\_D}(\tilde{u}, l, H \cup \{\lambda\}) \)
10: \hspace{4em} \textbf{if} \( (\tilde{u}', \tilde{C}') = \text{FAIL} \) \textbf{then}
11: \hspace{5em} go to line 5
12: \hspace{4em} \textbf{end if}
13: \hspace{3em} \textbf{end if}
14: \hspace{2em} \textbf{if} Algorithm\_G(\( \tilde{u}, l \)) \textbf{then}
15: \hspace{3em} \textbf{if} \( (\tilde{u}', \tilde{C}') \neq \text{FAIL} \) \textbf{then}
16: \hspace{4em} Set \( \tilde{u} := \tilde{u}' \)
17: \hspace{4em} Set \( \tilde{C} := \tilde{C}' \)
18: \hspace{3em} \textbf{end if}
19: \hspace{3em} \textbf{end if}
20: \hspace{2em} \textbf{if} \( \tilde{u}(l) = t \)
21: \hspace{3em} \textbf{Set} \( \tilde{u}(-l) := f \)
22: \hspace{3em} \textbf{Let} \( S \) be the literals \( l' \) such that \( \tilde{C}[l'] \) changed type in lines 19 and 20
23: \hspace{3em} \( (\tilde{u}, \tilde{C}) := \text{Compute}(\tilde{u}, \tilde{C}, S) \)
24: \hspace{3em} \textbf{go to line 2}
25: \hspace{2em} \textbf{end if}
26: \hspace{1em} \textbf{end while}
27: \hspace{1em} \textbf{return} \( \text{FAIL} \)
28: \hspace{1em} \textbf{end while}
29: \textbf{return} \( (\tilde{u}, \tilde{C}) \)
30: \textbf{end procedure}
2.4 Algorithm $\tilde{U}$

Algorithm $\tilde{U}$ takes a 3-SAT problem instance $\Phi$ as its only argument. Guinea claims the algorithm will produce an understanding $\tilde{u}$ defined with respect to the clauses of $\Phi$, or report that none exist.

1: procedure $\text{ALGORITHM}_{\tilde{U}}(\Phi)$
2: Let $\tilde{u}$ be an understanding assigning every literal in $\Phi$ to $\epsilon$
3: Let $\phi$ be $\emptyset$
4: Let $\tilde{C}$ be $\emptyset$
5: while there is some clause in $\Phi$ not in $\phi$ do
6: Let $\varphi$ be a clause in $\Phi$ but not $\phi$
7: if all literals $\lambda \in \varphi$ are false under $\tilde{u}$ then
8: for all literals $\lambda \in \varphi$ do
 9: Set $(\tilde{u}', \tilde{C}') := \text{ALGORITHM}_D(\tilde{u}, \tilde{C}, \lambda, \emptyset)$
10: if $(\tilde{u}', \tilde{C}') \neq \text{FAIL}$ then
11: Set $\tilde{u} := \tilde{u}'$
12: Set $\tilde{C} := \tilde{C}'$
13: go to line 18
14: end if
15: end for
16: return $\text{FAIL}$ $\triangleright \Phi$ is unsatisfiable
17: end if
18: for all literals $\lambda$ in $\varphi$ (first taking literals that are not false under $\tilde{u}$) do
19: Add the concept of $\lambda$ in $\varphi$ to $\tilde{C}$
20: Let $S$ be the set of literals $l$ such that $\tilde{C}[l]$ changed type in the previous step
21: $(\tilde{u}, \tilde{C}) := \text{COMPUTE}(\tilde{u}, \tilde{C}, S)$
22: Add $\varphi$ to $\phi$
23: end for
24: end while
25: return $\tilde{u}$ $\triangleright$ We have found a satisfying assignment for $\Phi$
26: end procedure

3 Refutation

3.1 Refutation of Algorithm D

A major error in Algorithm D stems from the fact that concepts removed from $\tilde{C}[\lambda]$ throughout the execution of the algorithm can be readded. Because Guinea states that we only consider each concept at most once, some concept that is first removed from $\tilde{C}[\lambda]$ and then later added to $\tilde{C}[\lambda]$ will not be processed a second time. We can use this fact to construct a case in which the algorithm terminates successfully, even though there is no defined understanding in which $\lambda$ is free.
Consider the following instance of 3-SAT:

\[(a \lor b \lor c) \land (a \lor \neg b \lor c) \land (a \lor b \lor \neg c) \land (\neg a \lor \neg y) \land (\neg a \lor y) \land (\neg d \lor \neg y) \land (\neg e).
\]

and some initial defined understanding \(\tilde{u}\) which satisfies \(\tilde{u}(x) = f, \tilde{u}(a) = t, \text{ and } \tilde{u}(y) = t\). An example is the understanding \(\tilde{u}\) defined by the following assignments:

\[
\begin{align*}
\tilde{u}(a) &= t, & \tilde{u}(-a) &= f, & \tilde{u}(b) &= \epsilon, & \tilde{u}(-b) &= \epsilon, \\
\tilde{u}(c) &= \epsilon, & \tilde{u}(-c) &= \epsilon, & \tilde{u}(d) &= f, & \tilde{u}(-d) &= t, \\
\tilde{u}(e) &= \epsilon, & \tilde{u}(-e) &= \epsilon, & \tilde{u}(x) &= f, & \tilde{u}(-x) &= t, \\
\tilde{u}(y) &= t, & \tilde{u}(-y) &= f.
\end{align*}
\]

Calling Algorithm D(\(\tilde{u}, C, x, \emptyset\)) should return an understanding under which \(x\) is free or report that one does not exist. The literal \(a\) is forced true by the first four clauses, and therefore either \(y \lor \neg a\) or \(\neg y \lor \neg a\) is false. By the fifth and sixth clauses \(\neg x\) must be true and \(x\) must be false. Thus no understanding defined with respect to these clauses can assign \(x\) to \(\epsilon\), and the algorithm should terminate unsuccessfully.

We will simulate a run of Algorithm D(\(\tilde{u}, C, x, \emptyset\)). On line 2 the algorithm takes a concept in \(\tilde{C}[x]^-\). The only such concept is the concept of \(\neg x\) in \((\neg x \lor y \lor \neg a)\). Next, on line 6 we choose any literal in this concept. Suppose we choose \(\neg y\). Assuming Algorithm D is correct, our recursive call Algorithm D(\(\tilde{u}, \tilde{C}, -y, \{x\}\)) will find that there does exist an understanding \(\tilde{u}'\) in which \(x\) is not changed and \(\neg y\) evaluates to free. An example is the understanding \(\tilde{u}'\) defined by the following assignments:

\[
\begin{align*}
\tilde{u}'(a) &= t, & \tilde{u}'(-a) &= f, & \tilde{u}'(b) &= \epsilon, & \tilde{u}'(-b) &= \epsilon, \\
\tilde{u}'(c) &= \epsilon, & \tilde{u}'(-c) &= \epsilon, & \tilde{u}'(d) &= f, & \tilde{u}'(-d) &= t, \\
\tilde{u}'(e) &= t, & \tilde{u}'(-e) &= f, & \tilde{u}'(x) &= f, & \tilde{u}'(-x) &= t, \\
\tilde{u}'(y) &= \epsilon, & \tilde{u}'(-y) &= \epsilon.
\end{align*}
\]

Assuming Algorithm G is correct, it will likewise report that there exists an understanding \(\tilde{u}'\) with respect to the clauses containing \(y\) or \(\neg y\) in which \(\neg y\) is true, for example the understanding \(\tilde{u}'\) defined by the following assignments:

\[
\begin{align*}
\tilde{u}'(a) &= t, & \tilde{u}'(-a) &= f, & \tilde{u}'(b) &= \epsilon, & \tilde{u}'(-b) &= \epsilon, \\
\tilde{u}'(c) &= \epsilon, & \tilde{u}'(-c) &= \epsilon, & \tilde{u}'(d) &= t, & \tilde{u}'(-d) &= f, \\
\tilde{u}'(e) &= \epsilon, & \tilde{u}'(-e) &= \epsilon, & \tilde{u}'(x) &= f, & \tilde{u}'(-x) &= t, \\
\tilde{u}'(y) &= \epsilon, & \tilde{u}'(-y) &= \epsilon.
\end{align*}
\]

At this point, the Algorithm D will set \(\tilde{u}\) to the understanding returned from the recursive call to itself on line 9 and then set \(\tilde{u}(y) = f\) on line 20. The Compute call on line 22 will not have any effect, since \(\tilde{u}\) will be defined. When we began, we required only that \(\tilde{u}(x) = f, \tilde{u}(a) = t, \text{ and } \tilde{u}(y) = t\). At this point \(\tilde{u}(x)\) is still equal to \(f\) since \(\tilde{u}(x)\) was guaranteed not to change by the recursive call. Similarly, \(\tilde{u}(a)\) is still equal to \(t\) since it
must be true in any defined understanding. However, \( \tilde{u}(\neg y) \) is now equal to \( t \). Note that the 3-SAT formula given stays the same if we swap \( \neg y \) and \( y \). Now, \( \neg x \lor y \lor \neg a \) is the only clause in \( \tilde{C}[x] \). By the exact argument given above, after running one more iteration of the while loop starting on line 22 we will reach a state satisfying our initial conditions that \( \tilde{u}(x) = f, \tilde{u}(a) = t, \) and \( \tilde{u}(y) = t \). The only clause in \( C[x] \) has already been considered, and so the algorithm halts and returns successfully on line 28.

We have already shown that there is no defined understanding in which \( x \) is free, so Algorithm D returns an incorrect result. If we try to fix this issue by allowing the same clause to be processed multiple times, this example also shows that Algorithm D can return to a previous state and run forever.

### 3.2 Refutation of Algorithm \( \tilde{U} \)

Even if Algorithm D were correct, Algorithm \( \tilde{U} \) could still fail.

Algorithm \( \tilde{U} \) maintains a set of satisfied clauses \( \phi \). The algorithm will choose some clause \( \varphi \) not in \( \phi \) and attempt to satisfy \( \varphi \) while keeping all clauses in \( \phi \) satisfied. The algorithm will then add \( \varphi \) to \( \phi \), and continue this process until \( \phi \) contains all clauses in \( \Phi \) or we fail in satisfying some clause.

The algorithm may fail when \( \varphi \) has every literal assigned false. Guinea claims that if there is a defined understanding that satisfies all clauses in \( \Phi \), then

“The call to Algorithm D . . . defines successfully an understanding with respect to \( \phi \) under which one literal \( \lambda \) in \( \phi \) is free” [SG15].

We believe the second \( \phi \) in this quote is intended to be \( \varphi \), as this is what Algorithm D claims to return in this context. However, Guinea’s assumption is incorrect.

Consider the following instance of 3-SAT:

\[
\begin{align*}
(x \lor a \lor b) \land (x \lor \neg a \lor b) \land (\neg x \lor a \lor \neg b) \land (\neg x \lor \neg a \lor \neg b) \land \\
(y \lor c \lor d) \land (y \lor \neg c \lor d) \land (\neg y \lor c \lor \neg d) \land (\neg y \lor \neg c \lor d) \land \\
(z \lor e \lor f) \land (z \lor \neg e \lor f) \land (\neg z \lor e \lor \neg f) \land (\neg z \lor \neg e \lor \neg f) \land \\
(x \lor y \lor z).
\end{align*}
\]

Depending on the order that the algorithm chooses clauses in line 6 and literals in line 8, it is possible that the program reaches a state in which

\[
\phi = \{(x \lor a \lor b), (x \lor \neg a \lor b), (\neg x \lor a \lor \neg b), (\neg x \lor \neg a \lor \neg b), \\
(y \lor c \lor d), (y \lor \neg c \lor d), (\neg y \lor c \lor \neg d), (\neg y \lor \neg c \lor d), \\
(z \lor e \lor f), (z \lor \neg e \lor f), (\neg z \lor e \lor \neg f), (\neg z \lor \neg e \lor \neg f)\},
\]

\[
\varphi = (x \lor y \lor z),
\]

\[
\tilde{u}(x) = f,
\]

\[
\tilde{u}(y) = f,
\]

\[
\tilde{u}(z) = f.
\]
In this case, the condition on line 7 will hold true because all literals in \( \varphi \) are false under \( \tilde{u} \). There is no satisfying assignment for the clauses in \( \phi \) in which any of \( x \), \( y \) or \( z \) is free. To see this let \( x \) be free. Then to satisfy all clauses we need to satisfy \((a \lor b) \land (\neg a \lor b) \land (a \lor \neg b) \land (\neg a \lor \neg b)\), but this is impossible. It is clear that analogous statements hold for \( y \) and \( z \). Thus Algorithm D will always return FAIL in line 9 of Algorithm \( \tilde{U} \) (assuming correctness of Algorithm D), so Algorithm \( \tilde{U} \) will return FAIL in line 16. However, the 3-SAT instance given is satisfiable via the assignment:

\[
\begin{align*}
  x &= \text{True}, \\
  a &= \text{True}, \\
  b &= \text{False}, \\
  y &= \text{True}, \\
  c &= \text{True}, \\
  d &= \text{False}, \\
  z &= \text{True}, \\
  e &= \text{True}, \\
  f &= \text{False}.
\end{align*}
\]

Therefore Algorithm \( \tilde{U} \) does not solve 3-SAT as claimed.

## 4 Conclusion

We have given counterexamples for both Algorithm D and Algorithm \( \tilde{U} \), even assuming all methods they call work as described by Guinea. These two algorithms are fundamental flaws in Guinea’s approach to solving 3-SAT in polynomial time. Guinea has not shown that 3-SAT is in P and has not resolved P-versus-NP.

**Acknowledgments** We thank Professor Lane A. Hemaspaandra, Michaela Houk, and Jesse Stern for their comments on previous drafts of this paper.

**References**

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *IN STOC*, pages 151–158. ACM, 1971.

[Kar72] R. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[SG15] A. Sánchez Guinea. Understanding SAT is in P. *ArXiv e-prints*, April 2015.
A Appendix. Reduction from 3-SAT to 3-SAT with No Duplicate Variables in Any Clause

Given an instance of \( \Phi \) 3-SAT, we would like to reduce it in polynomial time to some other instance \( \Phi' \) that is satisfiable if and only if \( \Phi \) is satisfiable, where no clause in \( \Phi' \) contains the same variable twice.

Take any clause \( \varphi \) in \( \Phi \) that contains a duplicate variable \( x \). If both \( x \) and \( \neg x \) are in \( \varphi \), then we remove \( \varphi \) from the collection of clauses completely as it is trivially always satisfied. Otherwise, we may replace a duplicate \( x \) or \( \neg x \) with some variable that always evaluates to false, and this will not affect satisfiability under any assignment. Take any symbols \( a \), \( b \), \( c \) not already appearing in any clause and add the clauses \((a \lor b \lor c)\), \((a \lor b \lor \neg c)\), \((a \lor \neg b \lor c)\), and \((a \lor \neg b \lor \neg c)\). Then replace the duplicate literal with \( \neg a \). No matter what the values of \( b \) and \( c \) are, in one of our four new clauses \( a \) will be surrounded by two literals evaluating to false, so \( a \) must always evaluate to true. Thus by replacing the duplicate literal with \( \neg a \), we are not affecting the satisfiability of our formula but we are strictly reducing the number of duplicate variables. Repeating this process until no duplicate variables remain yields our formula \( \Phi' \).