The idea behind our solution is the following: If we want to satisfy a clause \( C = \lambda_1 \lor \lambda_2 \), we have to impose certain restrictions on the truth values of \( \lambda_1 \) and \( \lambda_2 \). In particular, if \( \lambda_1 \) is false, then \( \lambda_2 \) must be true; and if \( \lambda_2 \) is false, then \( \lambda_1 \) must be true. Note, however, that there are no constraints on the value of \( \lambda_2 \) is \( \lambda_1 \) is true, and vice versa.

We model the constraints imposed by the clauses of a formula \( F \) using an implication graph \( G \) with \( 2n \) vertices and \( 2m \) edges, where \( n \) and \( m \) are as in the question, that is, \( n \) is the number of variables and \( m \) is the number of clauses. For every variable \( x_i \), \( G \) contains two vertices \( x_i \) and \( \bar{x}_i \). For every clause \( \lambda_1 \lor \lambda_2 \), there are two directed edges \( \bar{\lambda}_1 \rightarrow \lambda_2 \) and \( \bar{\lambda}_2 \rightarrow \lambda_1 \). For example, the clause \( x_1 \bar{x}_2 \) is represented by the edges \( \bar{x}_1 \rightarrow \bar{x}_2 \) and \( x_2 \rightarrow x_1 \). See Figure 1.

\[
F = (x_1 \lor x_2) \land (\bar{x}_1 \lor x_3) \land (\bar{x}_2 \lor \bar{x}_3)
\]

Figure 1: Illustration of the definition of the implication graph.

Our goal is to prove the following theorem.

**Theorem 1.** Formula \( F \) is satisfiable if and only if no strongly connected component of \( G \) contains a variable \( x_i \) and its negation \( \bar{x}_i \).

**Corollary 1.** Given a formula \( F \) in 2-CNF, it takes \( O(m) \) time to decide whether \( F \) is satisfiable if the variables of \( F \) are numbered 1 through \( n \). If this is not the case, the complexity of the algorithm becomes \( O(m \log n) \).
Proof. If the variables are numbered 1 through $n$, the construction of graph $G$ takes $O(n + m)$ time. We create the $2n$ vertices of $G$ in $O(n)$ time and add pointers to them to a table with $2n$ slots indexed by the variable numbers. Now we iterate sequentially through the clauses of formula $F$ and, for every clause, add the corresponding edges to $G$. Since we have pointers to the vertices corresponding to the variables in the clause, this takes constant time per clause, $O(m)$ time in total.

Given graph $G$, we can test whether $F$ is satisfiable by computing the strongly connected components of $G$ and checking whether any of them contains a variable and its negation. Computing the strongly connected components takes $O(n + m)$ time. The result is a labeling of the vertices of $G$ such that two vertices have the same label if and only if they belong to the same component. Thus, we iterate over the $n$ variables and test for each of them whether $x_i$ and $\overline{x}_i$ have the same label. This takes another $O(n)$ time. Thus, the total complexity of the algorithm is $O(n + m)$. It remains to observe that $n \leq 2m$. Hence, $O(n + m) = O(m)$.

If the variables are not numbered 1 through $n$, it is more difficult to map all occurrences of the same variable in different clauses to the same vertex in $G$. We get around this by storing the pointers to the vertices representing variables we have seen already in a red-black tree instead of an array. We start with an empty tree and iterate over the clauses as before. When inspecting a clause, we check whether its two variables are already in the tree. If not, we create two new vertices in $G$ for each missing variable and insert the variable with pointers to the vertices into the red-black tree. Then we add the edges between the vertices representing the variables as before. This takes $O(\log n)$ time per clause, $O(m \log n)$ in total. Once graph $G$ has been constructed in this manner, we continue to test whether $F$ is satisfiable as described in the previous paragraph.

The key, therefore, is proving Theorem 1. To do so, we make a few useful observations about the structure of graph $G$. Throughout our discussion, we use $\lambda_1 \rightarrow \lambda_2$ to denote the implication “if $\lambda_1$ is true, then, in order to satisfy $F$, $\lambda_2$ must be true”, as well as the edge from vertex $\lambda_1$ to $\lambda_2$ in $G$ that represents this implication. We also write $\lambda_1 \Rightarrow \lambda_2$ to indicate that there exists a path $\lambda_1 = \lambda_1' \rightarrow \lambda_2' \rightarrow \ldots \rightarrow \lambda_q' = \lambda_2$ from $\lambda_1$ to $\lambda_2$ in $G$. Note that the sequence of edges in this path represent implications that say: if $\lambda_1'$ is true, then $\lambda_2'$ must be true, which in turn implies that $\lambda_3'$ must be true, and so on. Thus, $\lambda_1 \Rightarrow \lambda_2$ also means that $\lambda_2$ must be true if $\lambda_1$ is true. We also write $\lambda_1 \Leftrightarrow \lambda_2$ if vertices $\lambda_1$ and $\lambda_2$ belong to the same strongly connected component of $G$.

Observation 1. If $\lambda_1 \Rightarrow \lambda_2$, then $\overline{\lambda}_2 \Rightarrow \lambda_1$.

Proof. Let $\lambda_1 = \lambda_1' \rightarrow \lambda_2' \rightarrow \ldots \rightarrow \lambda_q' = \lambda_2$ be the path from $\lambda_1$ to $\lambda_2$ in $G$. By the definition of graph $G$, every edge $\lambda_i' \rightarrow \lambda_{i+1}'$ in $G$ implies that $G$ also contains the “opposite” edge $\overline{\lambda}_{i+1}' \rightarrow \overline{\lambda_i}'$.
Thus, $G$ contains the path $\lambda_2 = \lambda'_q \rightarrow \lambda'_{q-1} \rightarrow \ldots \rightarrow \lambda'_1 = \bar{\lambda}_1$.

Observation 2. $\lambda_1 \cong \lambda_2$ if and only if $\bar{\lambda}_1 \cong \bar{\lambda}_2$.

Proof. If $\lambda_1 \cong \lambda_2$, then $\lambda_1 \Rightarrow \lambda_2$ and $\lambda_2 \Rightarrow \lambda_1$. Thus, by the previous observation $\lambda_1 \Rightarrow \lambda_2$ and $\bar{\lambda}_2 \Rightarrow \bar{\lambda}_2$, that is, $\bar{\lambda}_1 \cong \bar{\lambda}_2$.

What Observation 2 says is that $G$ has two types of strongly connected components: A component that contains a variable and its negation in fact contains both the nonnegated and negated form of every literal contained in the component. The components that do not contain a variable and its negation come in pairs $(C_1, C_2)$ such that a literal $\lambda$ belongs to $C_1$ if and only if $\bar{\lambda}$ belongs to $C_2$.

The following observation is also handy.

Observation 3. If $\lambda_1 \cong \lambda_2$, then both have to have the same value in a satisfying truth assignment of $F$.

Proof. Since $\lambda_1 \cong \lambda_2$, we have $\lambda_1 \Rightarrow \lambda_2$ and $\lambda_2 \Rightarrow \lambda_1$. The former prevents us from setting $\lambda_1$ to true without doing the same to $\lambda_2$. The latter prevents us from setting $\lambda_2$ to true without doing the same to $\lambda_1$. Hence, either $\lambda_1$ or $\lambda_2$ are both false or both true.

We are now ready to prove the main theorem.

Proof of Theorem 1. First assume that there exists a variable $x_i$ such that $x_i \cong \bar{x}_i$ in $G$. By Observation 3, both $x_i$ and $\bar{x}_i$ have to have the same value in a satisfying truth assignment of $F$. Since $\bar{x}_i$ is the negation of $x_i$, this is impossible. Hence, $F$ is not satisfiable in this case.

Now assume that there exists no such variable $x_i$. We construct a satisfying truth assignment of $F$. Similar to a vertex, we define the in-degree and out-degree of a strongly connected component $C$ as the number of edges in $G$ that have only their heads or only their tails in $C$.

We choose a strongly connected component $C_1$ of in-degree 0, as well as its counterpart $C_2$; that is, $C_2$ contains the negations of all literals in $C_1$. By Observation 1, $C_2$ has out-degree 0. We set all literals in $C_1$ to false and all literals in $C_2$ to true, thereby obtaining a valid truth assignment to the variables in these two components. Then we remove $C_1$ and $C_2$ from $G$ and repeat this procedure until $G$ is empty. Note that we can always find such a component $C_1$ because the graph obtained by contracting every strongly connected component into a vertex is acyclic, that is, there always exists a strongly connected component of in-degree 0.

Since we assign opposite values to a literal and its negation, this procedure does in fact construct a valid truth assignment to the variables in $F$. It remains to prove that this truth assignment satisfies $F$. 

3
By the definition of graph $G$, a truth assignment satisfies $F$ if for every edge $\lambda_1 \rightarrow \lambda_2$ in $G$, the implication $\lambda_1 \rightarrow \lambda_2$ is true. We prove that this is the case for the computed truth assignment, and we do so by induction on the number of strongly connected components of $G$. Note that the components of $G$ come in pairs, that is, the number of these components is even.

If $G$ has two strongly connected components, then by Observation 1, there are no edges between these components. Setting all literals in one component to false and all literals in the other component to true satisfies all implications inside the components, and these are the only implications in $G$.

If $G$ has more than two strongly connected components, let $(C_1, C_2)$ be the pair of components we choose and remove, and let $G'$ be the resulting graph. Since $G'$ has fewer connected components than $G$, the inductive claim implies that all implications in $G'$ are satisfied by the computed truth assignment. By setting all literals in $C_1$ to false and all literals in $C_2$ to true, we satisfy all the implications in $C_1$ and $C_2$. Thus, it remains to consider the out-edges of $C_1$ and the in-edges of $C_2$. (Recall that $C_1$ has no in-edges and $C_2$ has no out-edges.) The implications corresponding to these edges are also satisfied because every out-edge of $C_1$ has its tail set to false, making the implication true no matter what truth value we give to the head; similarly the head of every in-edge of $C_2$ is set to true, making the implication true no matter what truth value we give to the tail.

We observe that the proof of Theorem 1 immediately allows us to prove a stronger version of Corollary 1. We cannot only test whether $F$ is satisfiable, but we can also compute a satisfying truth assignment in linear time if $F$ is satisfiable. To do so, we compute the in-degree and out-degree of every strongly connected component of $G$, which is easily done in $O(m)$ time by iterating over the edges of $G$ and counting the in- and out-edges of all components. Then we maintain a list of strongly connected components of in-degree 0 and a list of components of in-degree greater than 0. We repeatedly choose a component from the first list, assign truth values to its literals and remove the component and its complement from $G$, and decrease the in-degrees of the remaining components accordingly. If the in-degree of a component becomes 0 as a result, it is moved to the list of in-degree 0 components. The details are similar to the topological sorting algorithm discussed in the tutorial and are easily seen to take $O(m)$ time.

Thus, we obtain the following final result.

**Theorem 2.** Given a Boolean formula $F$ in 2-CNF, it takes $O(m)$ time to test whether $F$ is satisfiable and, if so, construct a satisfying truth assignment of $F$. This claim holds under the assumption that the variables of $F$ are numbered 1 through $n$. Otherwise, the complexity of the algorithm is $O(m \log n)$. 

4