

# Formal Methods in Software Engineering

**Boolean Satisfiability** — Spring 2025

Konstantin Chukharev

# §1 Boolean Satisfiability

## Boolean Satisfiability Problem (SAT)

SAT is the classical NP-complete problem of determining whether a given Boolean formula is *satisfiable*, that is, whether there exists an assignment of truth values to its variables that makes the formula true.

$$\exists X. f(X) = 1$$

SAT is a *decision* problem — the answer is either “yes” or “no”. In practice, we are mainly interested in *finding* the actual satisfying assignment if it exists — this search problem is called *functional* SAT.

## Cook–Levin Theorem

Historically, SAT was the first problem proven to be NP-complete, independently by Stephen Cook [1] and Leonid Levin [2] in 1971.

**Theorem 1** (Cook–Levin): SAT is NP-complete.

That is, SAT is in NP, and *any* problem in NP can be *reduced* to SAT in polynomial time.

Cook's original proof was based on the concept of *Turing reductions* (now called *Cook reductions*).

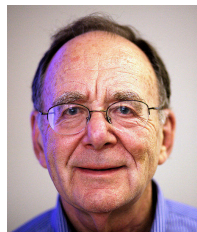
Richard Karp [3] later refined this with a stronger concept of polynomial-time *many-one reductions* (now known as *Karp reductions*), establishing the modern framework for NP-completeness.



Stephen Cook



Leonid Levin



Richard Karp

## Karp Reduction

**Definition 1** (Many-one reduction<sub>□</sub>): A polynomial-time *many-one reduction* from problem  $A$  to  $B$ , denoted  $A \leq_p B$ , is a polynomial-time computable function  $f$  such that for every instance  $x$  of  $A$ ,  $x$  is a “yes” instance of  $A$  if and only if  $f(x)$  is a “yes” instance of  $B$ .

A reduction of this kind is also called a *polynomial transformation* or *Karp reduction*.

## Proving the Cook–Levin Theorem

**Proof** ( $SAT \in NP$ ): A satisfying assignment serves as a *certificate* verifiable in linear time. □

**Proof of Theorem 1:** *Any problem in NP can be reduced to SAT in polynomial time.*

A problem  $L$  is in NP if there exists a polynomial-time *verifier* (Turing machine)  $V(x, c)$  that checks whether  $c$  is a valid certificate for  $x \in L$ .

A *Karp reduction* from  $L$  to SAT is a polynomial-time computable function mapping instances  $x$  of  $L$  to propositional formulas  $\varphi_x$ , such that  $\varphi_x$  is satisfiable iff  $x \in L$ .

Construct a formula  $\varphi_x$  encoding the computation of  $V(x, c)$  for input  $x$  and certificate  $c$ .

- Variables encode the Turing machine state, tape cells, and head position at each step  $t$ .
- Clauses enforce the initial configuration (fixed input  $x$ , free variables for certificate  $c$ ), valid transitions (per  $V$ 's rules), and acceptance at step  $T \in \mathcal{O}(p(|x|))$ .

A satisfying assignment to  $\varphi_x$  corresponds to a valid certificate  $c$  causing  $V(x, c)$  to accept, thus  $x \in L$ . □

This foundational result shows that SAT is a “universal” problem for NP.

## Solving General Search Problems with SAT

---

SAT solvers are powerful tools for solving general search problems. Given a problem, we can encode it as a SAT instance and use a SAT solver to find a solution.

To model a search problem as a SAT instance, the general approach is as follows:

1. Define propositional variables to represent the problem's *state*.
2. Encode the problem's *constraints* using propositional formulas.
3. Translate the formulas into a *clausal form*.
4. Run a SAT solver to *find* a satisfying assignment or *prove* its non-existence.

## Example: Graph Coloring

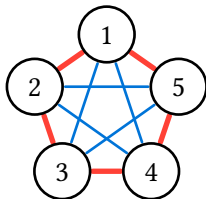
Graph  $G = (V, E)$  consists of a set  $V$  of vertices and a set  $E$  of edges, where each edge is an unordered pair of vertices. A complete graph on  $n$  vertices, denoted  $K_n$ , is a graph with  $|V| = n$  such that  $E$  contains all possible pairs of vertices. In total,  $K_n$  has  $\frac{n(n-1)}{2}$  edges.

Given a graph, color its vertices such that no two adjacent vertices have the same color.

Given a complete graph  $K_n$ , color its edges using  $k$  colors without creating a monochromatic triangle.

What is the largest complete graph for which this is possible for a given number of colors?<sup>1</sup>

- For  $k = 1$ , the answer is  $n = 2$ .
  - The graph  $K_2$  has only one edge, which can be colored with a single color.
- For  $k = 2$ , the answer is  $n = 5$ .
  - See the example of 2-colored  $K_5$  on the right.
- For  $k = 3$ , the answer is  $n = 16$ .
  - This is the work for a SAT solver. See the next slides.



---

<sup>1</sup>For a more general case, see [Ramsey's theorem](#)



# Modelling and Solving the Graph Coloring Example

---

## 1. Describe states using propositional variables.

- A simple (*one-hot*, or *direct*) encoding uses three (as  $k = 3$ ) variables for each edge:  $e_1$ ,  $e_2$ , and  $e_3$ . There are  $2^3 = 8$  possible combinations of values of three variables, but only 3 of them are *valid*.

## 2. Describe constraints using propositional formulas.

- Each edge must be colored with exactly one color. For each edge  $e$ :

$$(e_1 \vee e_2 \vee e_3) \wedge \neg(e_1 \wedge e_2) \wedge \neg(e_1 \wedge e_3) \wedge \neg(e_2 \wedge e_3)$$

- No monochromatic triangles are allowed. For each triangle  $(e, f, g)$ :

$$\neg(e_1 \wedge f_1 \wedge g_1) \wedge \neg(e_2 \wedge f_2 \wedge g_2) \wedge \neg(e_3 \wedge f_3 \wedge g_3)$$

## 3. Translate the formula into a clausal form.

## 4. Run a SAT solver to find a satisfying assignment or prove its non-existence.

Now, perform this process for increasing values of  $n$ , and find the largest  $n$  for which the formula is satisfiable. The answer is  $n = 16$  for  $k = 3$ .

## Code for Graph Coloring Example

```
n = 17
k = 3
m = n * (n - 1) // 2

edges = {}
for u in range(1, n + 1):
    for v in range(u + 1, n + 1):
        edges[(u, v)] = len(edges) + 1

def color(e, c):
    return (e - 1) * k + c

clauses = []
for e in range(1, m + 1):
    # At least one color is assigned to edge e
    clauses.append([
        color(e, c) for c in range(1, k + 1)
    ])
    # At most one color is assigned to edge e
    for c1 in range(1, k + 1):
        for c2 in range(c1 + 1, k + 1):
            clauses.append([
```

```
                -color(e, c1), -color(e, c2)
            ])
# No mono-chromatic triangles
for v1 in range(1, n + 1):
    for v2 in range(v1 + 1, n + 1):
        for v3 in range(v2 + 1, n + 1):
            e12 = edges[(v1, v2)]
            e23 = edges[(v2, v3)]
            e13 = edges[(v1, v3)]
            for c in range(1, k + 1):
                clauses.append([
                    -color(e12, c),
                    -color(e23, c),
                    -color(e13, c)
                ])

print(f"p cnf {color(len(edges), k)} {len(clauses)}")
for clause in clauses:
    print(" ".join(map(str, clause)) + " 0")
```

# TODO

---

- ☐ Encodings
- ☐ SAT Solvers
- ☐ Applications
- ☐ Exercises

## Bibliography

- [1] S. A. Cook, “The complexity of theorem-proving procedures,” in *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, 1971, pp. 151–158. doi: [10.1145/800157.805047](https://doi.org/10.1145/800157.805047).
- [2] L. A. Levin, “Universal sequential search problems,” *Problemy Peredachi Informatsii*, vol. 9, no. 3, pp. 115–116, 1973, [Online]. Available: <http://mi.mathnet.ru/ppi914>
- [3] R. M. Karp, “Reducibility among Combinatorial Problems,” *Complexity of Computer Computations*. Springer, pp. 85–103, 1972. doi: [10.1007/978-1-4684-2001-2\\_9](https://doi.org/10.1007/978-1-4684-2001-2_9).