Lecture 2

Ishaan Lal

January 24, 2025

1 SAT Solvers

1.1 Compact SAT Representation

Last week, we started to explore the SAT problem. One fascinating observation about boolean formulas in CNF is that the order of the clauses does not matter, and the order of the variables within the clause doesn't matter. That is to say, the following boolean formulas are all structurally equivalent:

$$\varphi_1 = (x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_2 \vee \overline{x_4} \vee x_5)$$
$$\varphi_2 = (x_2 \vee \overline{x_3} \vee x_1) \wedge (x_2 \vee \overline{x_4} \vee x_5)$$
$$\varphi_3 = (x_2 \vee \overline{x_4} \vee x_5) \wedge (x_1 \vee x_2 \vee \overline{x_3})$$

Due to these observations, we can represent an instance of a formula using set notation, where elements consist of a concatenation of literals. For example, the above three equations can be represented as:

$$\varphi_1 = \{x_1 x_2 \overline{x_3}, x_2 \overline{x_4} x_5\} = \{12\overline{3}, 2\overline{4}5\}$$
$$\varphi_2 = \{x_2 \overline{x_3} x_1, x_2 \overline{x_4} x_5\} = \{2\overline{3}1, 2\overline{4}5\}$$
$$\varphi_3 = \{x_2 \overline{x_4} x_5, x_1 x_2 \overline{x_3}\} = \{2\overline{4}5, 12\overline{3}\}$$

1.2 MiniSAT

MiniSAT is a very important SAT solver that was created by two Swedish PhD students, Niklas Een and Niklas Sorensson, in the early 2000s. It was very small – only consisting of around 600 lines of C++ code – but it worked really well. At the 2005 SAT Competition (a yearly affair), MiniSAT won the silver medal. The structure of MiniSAT inspired a lot of other SAT solvers, like the one we will build together soon, as well as the solver we will use, called PicoSAT.

1.3 PicoSAT

PicoSAT is a very simple to use, open source, SAT solver that we will use quite frequently. It was created by Professor Armin Biere, who was one of the leading researchers on SAT, and published many papers and books on the problem, including *the Handbook of Satisfiability*. PicoSAT won the 2007 SAT competition, taking home the gold medal. The solver was written in C, but has bindings to Python and other languages, making it useful for our purposes.

Solving with PicoSAT is easy. First, install it through pip via:

```
pip install pycosat
```

Or, depending on your setup, you may have to use pip3:

```
pip3 install pycosat
```

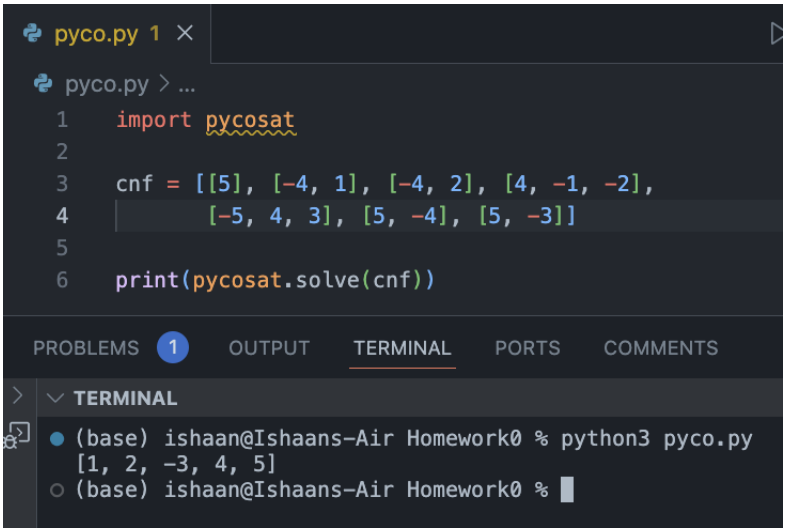
Now, we can represent a compact boolean equation as a list of lists, where each inner list consists of the values that create the clause. A unique number represents a unique variable. A positive number represents a positive literal, and a negative number represents a negative literal. The full transformation from CNF-SAT to compact form to pycosat input can be seen here:

$$\begin{aligned}
\varphi &= x_5 \wedge (\overline{x_4} \vee x_1) \wedge (\overline{x_4} \vee x_2) \wedge (x_4 \vee \overline{x_1} \vee \overline{x_2}) \wedge (\overline{x_5} \vee x_4 \vee x_3) \wedge (x_5 \vee \overline{x_4}) \wedge (x_5 \vee \overline{x_3}) \\
&\rightarrow \{5, \overline{4}1, \overline{4}2, 4\overline{12}, \overline{5}43, 5\overline{4}, 5\overline{3}\} && \text{(Compact)} \\
&\rightarrow [[5], [-4, 1], [-4, 2], [4, -1, -2], [-5, 4, 3], [5, -4], [5, -3]] && \text{(as input)}
\end{aligned}$$

To solve the equation, we simply use the method

```
pycosat.solve()
```

as seen here:



Our solver returned $[1, 2, -3, 4, 5]$ which corresponds to the following assignment:

$$x_1 = \texttt{TRUE}, x_2 = \texttt{TRUE}, x_3 = \texttt{FALSE}, x_4 = \texttt{TRUE}, x_5 = \texttt{TRUE}$$

Note that PicoSAT is deterministic, so it will always give the same assignment. So even if there are multiple satisfying assignments, it will return the same one. We must also note that often, if a formula has one satisfying assignment, then it likely has many.

Tricky Question

Suppose we don't like the assignment that we got above and wanted to obtain a different one. How could we change the formula to allow this to happen?

SOLUTION: We can add a new clause to our formula that constrains the solver to never return the assignment that we got. In the example above, we can add the following clause to the formula:

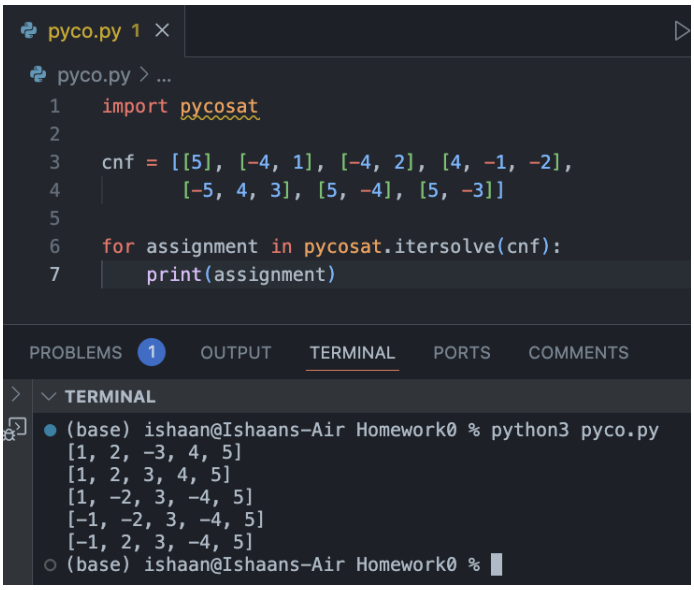
$$(\overline{x_1} \vee \overline{x_2} \vee x_3 \vee \overline{x_4} \vee \overline{x_5})$$

We see that now, satisfying everything but this new clause is the same as before, but with the assignment $[1, 2, -3, 4, 5]$, this last clause will get evaluated to `FALSE`, making the entire formula equivalent to `FALSE`. Thus, our solver would have to return a new assignment. And note that this "new assignment" would still be applicable to the formula before adding the new clause.

But of course, we may want to see all possible satisfying assignments at once. This can be done using the following method:

```
pycosat.itersolve()
```

as seen here:



2 Encodings

So what's the big deal with SAT? Why do we care so much about it? SAT is great because of its easy-to-understand structure, which allows it to be the go-to problem for reductions. *Reductions* are a mathematical procedure of describing one problem as an instance of another problem. We won't take the time to explain the full theory, but the idea is that we can use a *reduction* to show that two problems are equivalent, by encoding one problem as an instance of the other. Before we start exploring some other problems, we need to broaden our logical vernacular...

Warmup

Convert the following expressions to compact CNF:

1. $x \implies y$
2. $x \iff y$
3. $x \oplus y$
4. if x then y else z

SOLUTION:

(a) $x \implies y \equiv \neg x \vee y = \{\overline{x}y\}$

(b) $x \iff y \equiv (x \implies y) \wedge (y \implies x) \equiv (\neg x \vee y) \wedge (\neg y \vee x) \equiv \{\overline{x}y, \overline{y}x\}$

(c)

$$\begin{aligned} x \oplus y &= (x \wedge \overline{y}) \vee (\overline{x} \wedge y) \\ &= (x \vee \overline{x}) \wedge (x \vee y) \wedge (\overline{x} \vee \overline{y}) \wedge (\overline{y} \vee y) \\ &= \texttt{TRUE} \wedge (x \vee y) \wedge (\overline{x} \vee \overline{y}) \wedge \texttt{TRUE} \\ &= (x \vee y) \wedge (\overline{x} \vee \overline{y}) \\ &= \{xy, \overline{x}\overline{y}\} \end{aligned}$$

(d)

$$\begin{aligned} \text{if } x \text{ then } y \text{ else } z &\equiv (x \implies y) \wedge (\overline{x} \implies z) \\ &\equiv (\overline{x} \vee y) \wedge (x \vee z) \\ &= \{\overline{x}y, xz\} \end{aligned}$$

Warmup Part 2

Convert the following expressions to compact CNF:

1. All$(x_1, ..., x_n)$
2. AtLeastOne$(x_1, ..., x_n)$ aka ALO$(x_1, ..., x_n)$
3. AtMostOne$(x_1, ..., x_n)$ aka AMO$(x_1, ..., x_n)$
4. ExactlyOne$(x_1, ..., x_n)$

SOLUTION:

(a) Given that "All" returns true if and only if all variables are true, then we have:

$$\begin{aligned}\text{All}(x_1, .., x_n) &= x_1 \wedge x_2 \wedge ... \wedge x_n \\ &= \{x_1, x_2, ..., x_n\}\end{aligned}$$

That is, each variable is its own clause.

(b) For "At Least One", we will leverage the OR operator:

$$\begin{aligned}\text{ALO}(x_1, ..., x_n) &= x_1 \vee x_2 \vee ... \vee x_n \\ &= \{x_1 x_2 ... x_n\}\end{aligned}$$

That is, we have a singular clause consisting of the disjunction of all the variables.

(c) This is a difficult one, and there are many correct, natural ways of representing it. One way is as follows: if at most one variable is true, then for every pair of variables, it must be the case that we cannot have both of them being true. We can encode this constraint for a single pair of variables x_i, x_j as $(\overline{x_i} \vee \overline{x_j})$. Thus, putting it all together, we have:

$$\begin{aligned}\text{AMO}(x_1, ..., x_n) &= (\overline{x_1} \vee \overline{x_2}) \wedge (\overline{x_1} \vee \overline{x_3}) \wedge ... \wedge (\overline{x_{n-1}} \vee \overline{x_n}) \\ &= \{\overline{x_i x_j} \mid 0 \leq i < j \leq n\}\end{aligned}$$

We can see from the compact form that the formula will be evaluated to `FALSE` if and only if at least one clause evaluates to `FALSE`. Suppose $\overline{x_i} \vee \overline{x_j}$ is the clause that evaluates to `FALSE`. The only way this is possible is if $x_i =$ `TRUE` and $x_j =$ `TRUE`, which implies that we indeed do not have "at most one", as desired.

(d) Leveraging 2. and 3., we get:

$$\text{ExactlyOne}(x_1, ..., x_n) = \text{ALO}(x_1, ..., x_n) \wedge \text{AMO}(x_1, ..., x_n)$$

2.1 Binary AMO Encoding

As mentioned above, there are many ways of encoding AMO. What is the complexity of the solution provided. In other words, how many clauses is the encoding? Well, it is just the number of pairs of variables, which is $\binom{n}{2}$ or $\mathcal{O}(n^2)$. Is there an encoding that uses fewer clauses? Yes, but it may introduct some extra variables.

Binary AMO Encoding

$AMO(x_1, ..., x_n)$:

- Retain the original variables $x_1, ..., x_n$
- Let $m = \lceil \lg n \rceil$, and introduce new variables $b_1, ..., b_m$. Each b_j represents that ith bit of $\mathcal{T}$, where $x_{\mathcal{T}}$ is the $(\leq)$ one `TRUE` variable
- Our clauses will be described as:

$$\left\{ \left(x_i \implies \begin{cases} b_j & \text{if the } j\text{th bit of } i \text{ is } 1 \\ \overline{b_j} & \text{if the } j\text{th bit of } i \text{ is } 0 \end{cases} \right) \mid \forall x_i, b_j \right\}$$

Don't worry too much if you don't understand why this encoding works. What matters here is that we have introduced $\lg n$ extra variables, and in total, we have $n \lg n$ clauses – a significant improvement from before.

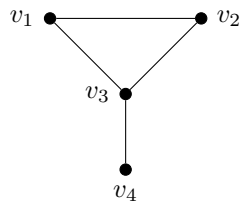
2.2 Does the Size of the Formula Matter?

Interesting, the size of the formula doesn't really correlate with how difficult the formula is to solve. But the performance of a SAT solver can vary depending on the encoding itself. Different encodings work better in different cases. So there is no "universal best encoding".

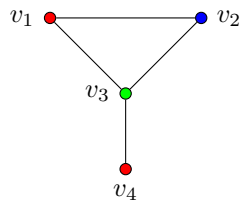
3 The Graph Coloring Problem

PROBLEM: Given a graph $G(V, E)$ (a set of vertices and a set of edges, where an edge connects exactly two vertices), and given k colors, is it possible to color the vertices of G with $\leq k$ colors such that each vertex has a color, and no two vertices that share an edge are the same color?

As an example, given the following graph:



and given $k = 3$, we would return YES, because the graph can be colored in ≤ 3 colors as follows:



However, if we had $k = 2$, we would return "NO", because it is not possible to color the graph with ≤ 2 colors so as to have adjacent vertices be different colors.

3.1 Encoding Graph Coloring as CNF-SAT

We'd now like to transform an instance of the Graph Coloring problem as an instance of CNF-SAT. To do this, we must consider what our variables should be/represent, as well as consider what constraints we have in the Graph Coloring problem, and ensure that they are represented in the SAT formula we construct.

Let us be very clear in what we desire. We want a *general* procedure of converting a graph, G, into an instance, φ, of CNF-SAT such that if the graph G is k-colorable, then φ is satisfiable. And if G is not k-colorable, then φ is not satisfiable.

Another way of thinking about it is that once we have our graph coloring, we should be able to look at the coloring and immediately know what the satisfying assignment for φ is. Conversely, once we have φ and find its satisfying assignment, we should be able to immediately know what the coloring for G is.

With this in mind, let us consider how we can define the variables for φ. A natural idea is to have a variable for each vertex-color pair. That is, suppose our vertices are $v_1, v_2, ..., v_n$, and our colors are $c_1, c_2, ..., c_k$, then, we will have variables $x_{v_1,c_1}, x_{v_1,c_2}, ..., x_{v_n,c_k}$. Ideally, if $x_{v_i,c_j} = \texttt{TRUE}$ in the satisfying assignment, then we will intend to say that "vertex i is colored with color j".

With these vertices in mind, we now will attempt to encode the constraints of the graph problem, and that will develop our CNF formula.

Constraint 1: Every vertex must be colored with exactly one color.

This is true, but remember when we were looking at the encoding for "Exactly one" (section 2, warmup part 2, question 4)? We leveraged the encoding for AMO and ALO, and we also said that AMO is difficult to encode and adds a lot of overhead. Let's amend our constraint slightly to:

Constraint 1

Constraint 1: Every vertex must be colored with at least one color.

Let's think about why this is actually a valid constraint. Suppose we find that vertex v_i can be colored with BLUE or with GREEN, and in either case, we would have a valid coloring. Well, this still tells us that we have a valid coloring, and we just have additional information. So let us accept this additional information for the sake of simplicity.

Encoding Constraint 1

Let us consider how we would encode constraint the first constraint. Recall that our variables are of the form x_{v_i,c_j}, where if $x_{v_i,c_j} = \texttt{TRUE}$, then vertex v_i will be colored with color c_j.

And we are aiming to ensure that for each vertex, it has at least one color. Then, consider an arbitrary vertex v_a, and consider all of its associated variables: $x_{v_a,c_1}, x_{v_a,c_2}, ..., x_{v_a,c_k}$. We want at least one of these to be true. Turning to the encoding rule we discussed in section two, we have:

$$\begin{aligned} \text{ALO}(x_{v_a,c_1}, x_{v_a,c_2}, ..., x_{v_a,c_k}) &= x_{v_a,c_1} \vee x_{v_a,c_2} \vee ... \vee x_{v_a,c_k} \\ &= \{x_{v_a,c_1} x_{v_a,c_2} x_{v_a,c_k}\} \qquad \text{(compact form)} \end{aligned}$$

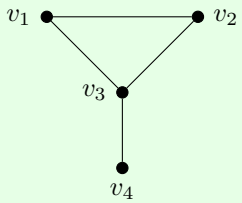
But of course, we need this clause for all vertices, not just for one vertex a_i, so we say that our compact form for t his constraint is:

$$\{x_{v_i,c_1}, x_{v_i,c_2}, ..., x_{v_i,c_k} \mid \forall v_i \in V\}$$

Before moving to the next constraint(s), let us work with the example graph and encode the first constraint:

Encoding Constraint 1 on Example Graph

Recall our example graph:



We will assume that $k = 3$. That is, we want to see if G is colorable using $\leq k$ or equivalently, ≤ 3 colors.

Our vertices are $\{v_1, v_2, v_3, v_4\}$, and we will say our colors are $\{c_1, c_2, c_3\}$. Thus, the variables that we will use will be:

$$x_{v_1,c_1}, x_{v_1,c_2}, x_{v_1,c_3}$$
$$x_{v_2,c_1}, x_{v_2,c_2}, x_{v_2,c_3}$$
$$x_{v_3,c_1}, x_{v_3,c_2}, x_{v_3,c_3}$$
$$x_{v_4,c_1}, x_{v_4,c_2}, x_{v_4,c_3}$$

To ensure that each vertex will be colored with at least one color, we will have:

$$\begin{aligned}\varphi = &(x_{v_1,c_1} \vee x_{v_1,c_2} \vee x_{v_1,c_3}) \\ \wedge &(x_{v_2,c_1} \vee x_{v_2,c_2} \vee x_{v_2,c_3}) \\ \wedge &(x_{v_3,c_1} \vee x_{v_3,c_2} \vee x_{v_3,c_3}) \\ \wedge &(x_{v_4,c_1} \vee x_{v_4,c_2} \vee x_{v_4,c_3})\end{aligned}$$

NOTE: This is NOT the encoding of the entire problem, as we still have some constraint(s) to add. Those constraint(s) will make our formula grow. However, at this point in time, if φ were to evalue to `TRUE`, then we would see that each clause must have at least one literal evaluating to `TRUE`, which is equivalent to saying that each vertex has at least one color, as desired.

The most important part of the graph coloring problem is that adjacent vertices must be of different colors. Let us now aim to encode this constraint:

Constraint 2

Constraint 2: If $(u, v) \in E$, then u and v have different colors.

Encoding Constraint 2

With our variables already defined, this should be relatively straightforward. Each edge in our graph adds a new constraint. Consider vertices v_i, v_j that are connected by an edge. If v_i is colored with color c_1, then vertex v_j CANNOT be colored with color c_1. If v_i is colored with c_2, then vertex v_j CANNOT be colored with color c_2. And of course, this must be true for all possible colors, and in the opposite direction as well.

Take a close look at the phrasing: "**IF** v_i is colored with c_1, **THEN** v_2 is NOT colored with c_1". The "**IF**...**THEN**" suggests that we have an implication relation. And the whole "vertex colored with a color" is exactly what our variables represent.

So literally translating the above, we have

$$x_{v_i,c_1} \implies \overline{x_{v_j,c_1}} \equiv \overline{x_{v_i,c_1}} \vee \overline{x_{v_j,c_1}} = \{\overline{x_{v_i,c_1} x_{v_j,c_1}}\}$$

However, this is just for one color. We need to enumerate over all colors and enforce this constraint. Moreover, this constraint must be applied to every edge in the graph, not just one.

In the general form, the entirety of constraint 2 can be written as:

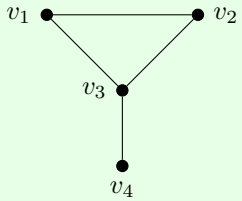
$$\{\overline{x_{v_i,c_c} x_{v_j,c_c}} \mid \forall (v_i, v_j) \in E, \forall c \in [1..k]\}$$

In words, this reads as: For every edge in the graph, we will enumerate over all colors, and ensure that if one endpoint is colored with that color, then the other vertex is not colored with that color.

Note that if we have m edges, then the number of clauses we are adding is $m \cdot k$, where each clause is of size 2.

Encoding Constraint 2 on Example Graph

Recall our example graph:



We'll now look to encode the second constraint. Consider the edge (v_1, v_2). If v_1 is colored with c, then v_2 must not be colored with c. That is, $x_{v_1,c} \implies \overline{x_{v_2,c}} \equiv \overline{x_{v_1,c}} \vee \overline{x_{v_2,c}}$. However, since the actual color of v_1 is unknown to us, we must add this constraint over all colors c_1, c_2, c_3.
That is, to φ, we would add:

$$(\overline{x_{v_1,c_1}} \vee \overline{x_{v_2,c_1}}) \wedge (\overline{x_{v_1,c_2}} \vee \overline{x_{v_2,c_2}}) \wedge (\overline{x_{v_1,c_3}} \vee \overline{x_{v_2,c_3}})$$

But of course, we must do this for all edges in the graph. Upon doing so, and combining it with the φ we had when considering Constraint 1, we get:

$$\begin{aligned}
\varphi = &(x_{v_1,c_1} \vee x_{v_1,c_2} \vee x_{v_1,c_3}) \\
&\wedge (x_{v_2,c_1} \vee x_{v_2,c_2} \vee x_{v_2,c_3}) \\
&\wedge (x_{v_3,c_1} \vee x_{v_3,c_2} \vee x_{v_3,c_3}) \\
&\wedge (x_{v_4,c_1} \vee x_{v_4,c_2} \vee x_{v_4,c_3}) \\
&\wedge (\overline{x_{v_1,c_1}} \vee \overline{x_{v_2,c_1}}) \wedge (\overline{x_{v_1,c_2}} \vee \overline{x_{v_2,c_2}}) \wedge (\overline{x_{v_1,c_3}} \vee \overline{x_{v_2,c_3}}) \\
&\wedge (\overline{x_{v_1,c_1}} \vee \overline{x_{v_3,c_1}}) \wedge (\overline{x_{v_1,c_2}} \vee \overline{x_{v_3,c_2}}) \wedge (\overline{x_{v_1,c_3}} \vee \overline{x_{v_3,c_3}}) \\
&\wedge (\overline{x_{v_2,c_1}} \vee \overline{x_{v_3,c_1}}) \wedge (\overline{x_{v_2,c_2}} \vee \overline{x_{v_3,c_2}}) \wedge (\overline{x_{v_2,c_3}} \vee \overline{x_{v_3,c_3}}) \\
&\wedge (\overline{x_{v_3,c_1}} \vee \overline{x_{v_4,c_1}}) \wedge (\overline{x_{v_3,c_2}} \vee \overline{x_{v_4,c_2}}) \wedge (\overline{x_{v_3,c_3}} \vee \overline{x_{v_4,c_3}})
\end{aligned}$$

Putting our formula φ into a SAT solver, we get the following assignment:

$$x_{v_1,c_1} = \texttt{FALSE} \quad x_{v_1,c_2} = \texttt{FALSE} \quad x_{v_1,c_3} = \texttt{TRUE}$$
$$x_{v_2,c_1} = \texttt{FALSE} \quad x_{v_2,c_2} = \texttt{TRUE} \quad x_{v_2,c_3} = \texttt{FALSE}$$
$$x_{v_3,c_1} = \texttt{TRUE} \quad x_{v_3,c_2} = \texttt{FALSE} \quad x_{v_3,c_3} = \texttt{FALSE}$$
$$x_{v_4,c_1} = \texttt{FALSE} \quad x_{v_4,c_2} = \texttt{FALSE} \quad x_{v_4,c_3} = \texttt{TRUE}$$

With $c_1 = \text{GREEN}, c_2 = \text{BLUE}$ and $c_3 = \text{RED}$, we see that the satisfying assignment gives us the coloring of the graph we saw at the start of section 3.

3.2 Finding the Min-Coloring

What happens if we want to find the minimum coloring of G? Notice that this is a shift from a DECISION problem to an OPTIMIZATION problem. Before, we were given a graph G, and a value k, and we simply returned YES/NO indicating if the graph could be colored. This new problem asks us to find a value k that is as small as possible such that G is still $k-$colorable. This is the problem of finding the *chromatic number* of G, notated $\chi(G)$. Let us call the original problem "GRAPH-COLORING", and this new optimization problem "MIN-COLORING".

One naive approach is to use our solution for "GRAPH-COLORING", by testing $k = 1$, then $k = 2$, and so on and so forth until our solver finds a solution. The runtime of this is $\mathcal{O}(\chi(G))$.

A different approach is to binary search. First, check if G can be colored with $1, 2, 4, 8, 16...$ colors. Stop at the smallest value 2^k such that G is 2^k-colorable. Then, binary search for the true value of $\chi(G)$ in the range $[2^{k-1}, 2^k]$. The runtime of this is $\mathcal{O}(\lg \chi(G))$

3.3 UNSAT

Let's consider another variant of the problem. Given a graph G, and an integer k, we need to decide if there is NO valid k-coloring. That is, if every assignment of the k colors to vertices is invalid, we would return YES. We'll call this problem UNSAT.

In terms of solving time, UNSAT can be **way** harder than SAT. This is because, with SAT, we only need to find one satisfying assignment. Once we find it, we are done. With UNSAT, we need to consider every assignment, and show that none of them are valid. This is a massive search space.

In practice, on a triangle-free graph, checking UNSAT for $k = 5$ can take upwards of 10 minutes with the implementation we have now.

3.4 Solving Time

It turns out that the amount of time to solve the GRAPH-COLORING problem peaks once we approach the chromatic number. In fact, for most instances, setting $k = \chi(G) - 1$ will take the longest time to solve.