LECTURE 3

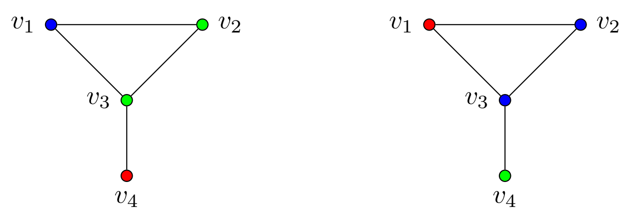
Ishaan Lal

January 31, 2025

1 Symmetry Breaking

UNSAT takes a very long time to solve. Why? The idea is that our SAT solver is oblivious to the fact that we are working with a graph. It treats it as just a typical boolean formula. Because of this, there is a special property about graph coloring that we haven't yet incorporated in our formula: symmetric colorings.

Consider the following two attempted colorings of a graph:



These two colorings are essentially identical. We can form equivalence classes of vertices that are colored the same color, and they would be the same classes for both colorings. The only difference is the actual colors of the classes. So visually, we can tell that these two attempted colorings are identical, but our SAT solver does not know this. Because in the first coloring, the variable $x_{v_1,\text{BLUE}} = \texttt{TRUE}$, but in the second coloring, $x_{v_1,\text{BLUE}} = \texttt{FALSE}$, because v_1 is not colored blue.

This is what we call a symmetric coloring. We can greatly reduce the search space by telling our SAT solver "hey, these two colorings are identical."

But how do we do this? It turns out the solution is very simple. We will pick a few vertices, and fix their colors at the start. For example, if we choose to fix $v_1 = \text{BLUE}$, or equivalently, $x_{v_1,\text{BLUE}} = \texttt{TRUE}$, then the second coloring would never be explored in our search space. This can be done by adding $x_{v_1,\text{BLUE}}$ as a standalone clause in φ. Upon doing this, our runtime improves drastically.

2 Stable Matchings

Problem: We have n men and n women. Each man ranks the n women in order of preference. Each woman ranks the n men in order of preference. The goal of this problem is to find a one-to-one matching of men to women.

A man and a woman who both prefer each other to their matched partners are called a **blocking pair**. A matching is **stable** if it has not blocking pairs.

2.1 Encoding Stable Matching

We will use an unintuitive variable definition for our encoding. Let m_{ip} be a variable that is $\texttt{TRUE}$ if man i is matched to the pth woman or later on his preference list. Similarly, let w_{ip} be a variable that is $\texttt{TRUE}$ if woman i is matched to the pth man or later on her preference list.

For example, suppose we have the following two men and two women with given preference lists:

$$M_1 : W_1 > W_2$$
$$M_2 : W_1 > W_2$$
$$W_1 : M_1 > M_2$$
$$W_2 : M_1 > M_2$$

Further, suppose M_1 is matched with W_1 and M_2 is matched with W_2, then, our variables and their evaluations are given by:

$$m_{1,1} = \texttt{TRUE} \quad m_{1,2} = \texttt{FALSE} \quad m_{2,1} = \texttt{TRUE} \quad m_{2,2} = \texttt{TRUE}$$
$$w_{1,1} = \texttt{TRUE} \quad w_{1,2} = \texttt{FALSE} \quad w_{2,1} = \texttt{TRUE} \quad w_{2,2} = \texttt{TRUE}$$

Constraint 1

Constraint 1: Every man is matched to some partner.

That means, we must have, for arbitrary man m_i, at least one of $m_{i,1}, m_{i,2}, ..., m_{i,p}$ evaluating to true. However, even simpler, we can just look at $m_{i,1}$. If this variable evaluates to `TRUE`, then by definition, it means that man i is matched to the women ranked first or later on the list, and the list contains all women. So, for each man, we need $m_{i,1} = \texttt{TRUE}$.

This can be represented in CNF form by:

$$\varphi = m_{1,1} \wedge m_{2,1} \wedge ... \wedge m_{n,1} = \{m_{i,1} \mid 1 \leq i \leq n\} \qquad \text{(in compact form)}$$

Constraint 2

Constraint 2: If a man gets his pth or later choice, it is also his $(p-1)$th or later choice.

This makes sense from our definition of our variables. That is, if $m_{i,p} = \texttt{TRUE}$, meaning that the ith man is matched to his pth or later woman, then clearly, $m_{i,p-1}$ must also be true. But remember, our formula φ does not know that yet. We know that because of how our variables are defined, but we need to encode this. This can be done as follows:

$$\{m_{i,p} \implies m_{i,p-1} \mid 1 \leq i \leq n, 2 \leq p \leq n\}$$

Constraint 3

Constraint 3: If man i is matched to woman j, then she is matched to him.

Hopefully the need for this constraint is obvious. How we encode this is slightly difficult. How do we know what woman a man is matched to just by the variables we have defined. Well, if man i is matched with woman p, then it must be the case that $m_{i,p} = \texttt{TRUE}$, but also, $m_{i,p+1} = \texttt{FALSE}$. So, this will be our condition. We can encode the constraint as:

$$\{m_{i,p} \wedge \overline{m_{i,p+1}} \implies w_{j,q} \wedge \overline{w_{j,q+1}} \mid 1 \leq i, j \leq n\}$$

where p is the position of woman j in man i's list, and q is the position of man i in woman j's list.

Constraint 4

Constraint 4: If man i is matched to someone worse than woman j, then her match must be better than him with respect to her preference list.

The wording of this is a little bit wonky, but it is essentially ensuring that we do not have any blocking pairs. That is, if woman j is highly ranked for man i, but they are not matched, the only way that they will not form a blocking pair is for woman j to be matched with man k who she ranked higher than man i.

In terms of encoding, we have:

$$\{m_{i,p+1} \implies \overline{w_{j,q}} \mid 1 \leq i, j \leq n\}$$

And those are all of the constraints. By putting the constraints together into a single boolean formula, we can feed it into a SAT solver, and get our matchings.

2.2 But What about Gale-Shapley?

Some of you might remember a fun algorithm from CIS1210 called Gale-Shapley. We won't go into the details of that algorithm in this class, but all you need to know is that this algorithm is a linear time algorithm that solves the matching problem in linear time.

So why did we waste all of that time converting Stable Matching to SAT? Why couldn't we have just used Gale-Shapley? Well, we could have.

The major downfall of Gale-Shapley is its inability to solve mutations of the stable matchings problem.

For example, a common deviant of this problem is SMTI: the stable matching problem where preference lists may be incomplete and contain ties. For SMTI, we can alter our formula φ, by mainly changing the clauses developed by constraint 4, and then shove our formula in a SAT solver. But Gale-Shapley cannot solve SMTI.

Another variant is SM-C: the stable matching problem with couples. You can imagine the setting as follows: There is a group of couples, all of them are applying to residency. Together, the couples construct a preference list of **pairs** of hospitals, instead of creating lists for the individual. Then, we find the optimal matching given these pair preferences. Gale-Shapley cannot handle this, but our SAT solver could be adapted to handle it.

The short reason why Gale-Shapley doesn't work for SMTI and SM-C is because these problems are NP-Complete.

3 Introduction to SAT Algorithms

Hopefully by now, we can all recognize that solving SAT is a difficult problem. Today we will explore the algorithms that go into solving SAT.

A naive approach is to simply try every possible assignment until we find a satisfying assignment or exhaust the search space. One can interpret this as conducting DFS on a search tree, where each branch represents the assignment of a variable. Leaves of this search tree represent the assignment of all variables.

The clearest issue with this approach is the time. The search space is exponential – each variable has two possible assignments, so for n variables, there are 2^n assignments we would have to check.

4 Simplifying the Search Space

Unfortunately, there is no easy way to circumvent exploring the exponential number of assignments by reducing it to a more manageable search space (why? if the search space was smaller than exponential, it would likely be polynomial!).

Therefore, we will have to accept the fact that there may be an exponentially sized search space at worst, but we will still work to reduce the search space by skipping over unnecessary variable assignments that we know will not work.

How we do this is simply implementing observations of the SAT problem.

Critical SAT Facts

Let $\varphi = C_1 \wedge C_2 \wedge ... \wedge C_n$ be a CNF formula, where C_i is a clause of the form $(x_{i_1} \vee x_{i_2} \vee ... \vee x_{i_k})$

1. If φ is satisfiable, then $C_{\forall i} \equiv$ `TRUE`. That is, every clause must be evaluated to `TRUE`. Or equivalently, no clause is evaluated to `FALSE`.
2. If a literal in a clause is satisified, then the entire clause is evaluated to `TRUE`, regardless of the other variables in the clause. That is, if we have clause $C_i = x_{i_1} \vee x_{i_2} \vee ... \vee x_{i_k}$, and we find that $x_{i_1} =$ `TRUE`, then immediately, $C_i =$ `TRUE`, regardless of $x_{i_2}, ..., x_{i_k}$

Fact 2 can be critical in developing a useful SAT solver! In that example, we can essentially ignore clause C_i once we found that $x_{i_1} =$ `TRUE`, because the clause is satisfied, which, when considering Fact 1, puts us a step closer in satisfying φ. Formally, we have:

Observation 1

When we set $x_i =$ `TRUE`, any clause containing the positive literal x_i becomes satisfied, so we no longer need to consider those clauses. We can thus **remove** all clauses containing x_i, which greatly reduces our search space.

A similar observation can be made for clauses containing a negative literal:

Observation 2

When we set $x_i =$ `TRUE`, any clause containing the negative literal $\overline{x_i}$ needs to be satisfied **by a different literal**, so we can ignore $\overline{x_i}$ in that clause. Thus, we can remove $\overline{x_i}$ from all clauses containing it. In logic:

$$(F \vee x_1 \vee x_2 \vee ...) \equiv (x_1 \vee x_2 \vee ...)$$

4.1 The Splitting Rule

These observations have given us something that we call **The Splitting Rule**. To formalize context, we can think of finding a solution to SAT as a process of assigning variables, and then removing variables and/or clauses from our formula if we are allowed to ignore them. Premission to ignore a variable/clause comes from the observations above.

The Splitting Rule

1. When we set $x_i =$ `TRUE`, we can remove/ignore all clauses containing the positive literal x_i (per observation 1)
2. When we set $x_i =$ `TRUE`, we can remove/ignore all instances of the negative literal $\overline{x_i}$ (per observation 2)

After repeatedly applying the splitting rule to formula φ:

(a) If there are **no clauses left**, that is, $\varphi = \emptyset$, then all clauses have been satisfied (removed), thus φ is satisfied.

(b) If φ ever contains an **empty clause**, then all literals in that clause are `FALSE`, so we have made a mistake. Notationally, $\epsilon \in \varphi$ denotes that φ contains an empty clause.

In programming terms, the method of traversing our search tree and re-routing as we hit "bad leaves" is called **backtracking** – a method of repeatedly making a guess to explore partial solutions, and if we hit a "dead end" (contradiction), then undo the last guess. Many problems rely on backtracking, such as printing all permutations of a set of numbers and the N queens problem.

For a bit of notation, for a CNF φ and a literal x, we will define $\varphi \mid x$ (read as "φ given x") to be a new CNF produced by:

1. Removing all clauses containing x
2. Removing $\overline{x}$ from all clauses containing it.

Or, in simpler terms, $\varphi \mid x$ yields a CNF which is equivalent to φ after the Splitting Rule is applied when $x =$ `TRUE`.

One helpful observation is that conditioning is commutative:

$$\varphi \mid x_1 \mid x_2 = \varphi \mid x_2 \mid x_1$$

Cool. We can now present the SAT solver we have come up with in terms of pseudocode:

Pseudocode v1

```
# method to check if φ is satisfiable:
backtrack(φ):
    if φ = ∅: return TRUE
    if ϵ ∈ φ: return FALSE
    let x = pick_variable(φ)
    return backtrack(φ | x) OR backtrack(φ | x̄)
```

Let's see an example:

Example

Consider

$$\varphi = (x_1 \vee x_3 \vee \overline{x_4}) \wedge (\overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2)$$

Let's rewrite φ in compact form:

$$\varphi = \{13\overline{4}, \overline{2}3, \overline{1}2\}$$

We'll now go about trying to find a satisfying assignment. For purposes of this example, we will simply use trial and error, instead of a systematic traversal of a search tree.

First, we may notice the positive literal x_1 in the first clause, so we may choose to set $x_1 = \texttt{TRUE}$. Substituting, we get:

$$\varphi = (T \vee x_3 \vee \overline{x_4}) \wedge (\overline{x_2} \vee x_3) \wedge (F \vee x_2)$$

We can use the first part of The Splitting Rule on the first clause (i.e. ignore it), and the second part of the Splitting Rule on the third clause, which yields:

$$\varphi' = (\overline{x_2} \vee x_3) \wedge (x_2) \equiv \{\overline{2}3, 2\}$$

Suppose we now decided to set $x_2 = \texttt{FALSE}$ (yes, this may seem like a silly decision, but such decisions may not seem silly when we have massive equations with hundreds or thousands of variables and clauses).

Upon setting $x_2 = \texttt{FALSE}$, we obtain:

$$\varphi' = (T \vee x_3) \wedge (F)$$

Now, when we apply the splitting rule to the second clause, we would be left with an empty clause ($\epsilon \in \varphi$), because there are no variables that were in this clause originally that are left to assign. As per (b) in the Splitting Rule, this means that we have made a mistake.

Once we hit a mistake, we can correct the mistake (backtrack). In this case, setting $x_2 = \texttt{TRUE}$ fixes the mistake. As an exercise, you can complete tracing through the Splitting Rule on this example, and find that the satisfying assignment is:

$$x_1 = \texttt{TRUE} \quad x_2 = \texttt{TRUE} \quad x_3 = \texttt{TRUE}$$

A visual example is provided in the Lecture Slides. Please refer to that.

5 Improving our Solver

Okay cool. We've developed a procedure for solving SAT. But is it fast? Well, that depends on how we implement it. In particular, we want:

1. to be able to compute $\varphi \mid x$ quickly
2. to be able to detect empty clauses quickly
3. to be able to backtrack quickly

That's a lot of desiderata. Let's try coming up with some solutions:

Naive Idea 1

We can compute $\varphi \mid x$ quickly by directly implementing the Splitting Rule – that is, delete satisfied clauses and delete literals evaluated to `FALSE` from φ.

Deletion can be a quick, cheap task if we implement φ as a linked list of clauses, where clauses themselves are a linked list of literals. Then, deletion is simply a reroute of pointers. This is better than an array implementation, as it avoids the need for shifting data after a deletion.

To detect an empty clause, we can just check if the linked list representing a clause is empty, however, we would need to check all clauses.

The biggest issue comes with backtracking. Once we make a mistake and need to backtrack, we need to obtain a prior version of the formula. This is especially difficult to do with this implementation.

Naive Idea 2

We'll improve Naive Idea 1, by instead of modifying φ directly, we will create a copy of φ first, and modify that.

Now, the issue of backtracking is solved, because we can just restore the old formula.

However, we have introduced a new issue: it is way to expensive (with respect to time and memory) to copy the formula every time we split. If we have thousands or millions of clauses, this is a lot of memory we are using! Not to mention, we are still working with exponential time!

But we can be smart. In fact, we will try to devise a schema where we don't modify or copy the formula! A key observation to note is that **we must only backtrack once a clause has become empty after the Splitting Rule has been applied**.

Key Observation!

A clause can only become empty if it has just one unassigned literal remaining.
In other words, if φ does not contain an empty clause, but $\varphi \mid x$ does contain an empty clause, then the clause $\overline{x}$ must have existed in φ, because once we apply the splitting rule, we would have removed all instances of $\overline{x}$, resulting in an empty clause.

5.1 1 Watched Literal Scheme

We have built our way to something called the 1 Watched Literal Scheme. The idea is for each clause to "watch" one literal within the clause, and maintain a **watching invariant**: the watched literal is `TRUE` or unassigned.

If the literal being watched becomes `FALSE`, then the clause must watch another literal.

If there are no more `TRUE` or unassigned literals to watch, then the clause must be empty.

Please refer to the lecture slides for a visual walkthrough of the 1 Watched Literal Schema.

5.2 Unit Propagation

Let's use another observation to speed up our solver (that is, reduce the search space).

Unit Clause

A **Unit Clause** is a clause containing only one literal. For any unit clause $\{\ell\}$, we MUST set $\ell =$ `TRUE` for φ to potentially be `TRUE`.

This seems like a very obvious observation, and maybe it is. But it is critical to include, as it can greatly speed up our solver.

You might ask: *How? How can it speed up our solver?* And this is a particularly good question when φ doesn't have any unit clauses. However, as we apply the Splitting Rule and continuously reduce our clauses, we may face some unit clauses, whereby we can immediately set the assignment using the Unit Propagation Rule.

5.3 The DPLL Algorithm

The Davis-Putnam-Logemann-Loveland Algorithm (DPLL) is an implementation of the Unit Propagation algorithm. This is still the basic algorithm behind most state-of-the-art SAT solvers today!

DPLL Algorithm Pseudocode

```
dpll(φ):
    if φ = ∅: return TRUE
    if ϵ ∈ φ: return FALSE
    if φ contains a unit clause {x_ℓ}:
        return dpll(φ | x_ℓ = TRUE)
    let x = pick_variable(φ)
    return dpll(φ | x = TRUE) OR dpll(φ | x = FALSE)
```

Refer to the lecture slides for a visual walkthrough of the DPLL algorithm.

Notice that even with this algorithm, we may make the call to `dpll()` an exponential number of times, so speeding up the algorithm however possible is desired.

In its current form, DPLL takes quite a bit of time on the Unit Propagation step, because it takes time to determine if we have a unit clause. This smells similar to an earlier problem! How did we handle the issue of determining if we had an empty clause? We used the 1-Watched Literal Schema. Now, we need to detect if a clause has **at most one** literal (instead of zero). This gives us...

5.4 2 Watched Literal Schema

Motivating Observation

Each clause can watch two literals and maintains a watching invariant: the watched literals are not False, unless the clause is satisfied. If a watched literal becomes False, then we watch another.

If we are unable to maintain the invariant, then the clause is unit. This is because we will break the invariant once a literal becomes False and we would attempt to watch another literal, but we have run out of literals to watch. But since we are watching 2 literals, there is still one left, meaning we have a unit clause, and can propagate.