15-414: Bug Catching: Automated Program Verification

Lecture Notes on SAT Solving Techniques

Matt Fredrikson Ruben Martins

Carnegie Mellon University
Lecture 20

1 Introduction

In Lecture 13 "SAT Solvers & DPLL", we studied decision procedures for propositional logic which are often referred to as SAT solvers. In particular, we studied the DPLL procedure and how Boolean Constraint Propagation (BCP) plays an important role in reducing the number of interpretations that the SAT solver needs to explore until it either finds an interpretation that satisfies all clauses or proves that no interpretations satisfies the formula, i.e. the formula is unsatisfiable.

In Lab 3, we asked you to wrote a provably correct brute-force search SAT solver. In Lab 4, we ask you to make your SAT solver more efficient by writing and verifying a minimalistic implementation of the DPLL algorithm, which forms the basis of the most modern SAT solvers. In this lecture, we will study efficient data structures for BCP and pre-processing techniques for SAT that simplify a propositional formula φ and transform it into an *equivalent* or *equisatisfiable* formula ϕ. These techniques can be incorporated into your final submission of Lab 4 and be used to improve the efficiency of your SAT solver for the verified SAT competition.

2 Review: BCP & DPLL

In this section, we will review the main components of BCP and DPLL. For a full revision, we refer to the lectures notes of Lecture 13 "SAT Solver & DPLL". [1]

[1] Available at https://www.cs.cmu.edu/~15414/lectures/13-dpll.pdf

2.1 Boolean Constraint Propagation

Consider the following CNF formula:

$$\underbrace{(x_2 \vee x_3)}_{C_0} \wedge \underbrace{(\neg x_1 \vee \neg x_3)}_{C_1} \wedge \underbrace{(\neg x_1 \vee \neg x_2 \vee x_3)}_{C_2} \wedge \underbrace{(x_0 \vee x_1 \vee \neg x_3)}_{C_3} \wedge \underbrace{(\neg x_0 \vee x_1 \vee x_3)}_{C_4} \quad (1)$$

Suppose that your `sat` procedure begins by choosing to assign x_0 to $true$. This leaves us with:

$$\begin{aligned} & (x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_0 \vee x_1 \vee \neg x_3) \wedge (\neg x_0 \vee x_1 \vee x_3) \\ \leftrightarrow\ & (x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (\top \vee x_1 \vee \neg x_3) \wedge (\bot \vee x_1 \vee x_3) \\ \leftrightarrow\ & (x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_3) \end{aligned}$$

Since after propagating x_0 there are no unit clauses, i.e. clauses with a single unassigned literal, then it means we cannot infer any additional information by using BCP. Suppose that we now assign x_1 to $true$. This leaves us with:

$$\begin{aligned} & (x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_3) \\ \leftrightarrow\ & (x_2 \vee x_3) \wedge (\bot \vee \neg x_3) \wedge (\bot \vee \neg x_2 \vee x_3) \wedge (\top \vee x_3) \\ \leftrightarrow\ & (x_2 \vee x_3) \wedge (\neg x_3) \wedge (\neg x_2 \vee x_3) \end{aligned}$$

Notice the clause C_1, which was originally $(\neg x_1 \vee \neg x_3)$, is now simply $(\neg x_3)$. It is obvious that any satisfying interpretation must assign x_3 $false$, so there is really no choice to make given this formula. We say that x_3 is a *unit literal*, which simply means that it occurs in a clause with no other literals. BCP when run in the interpretation $I = \{x_0, x_1\}$ will imply that x_3 has to be assigned to $false$. If we propagate this assignment then this leaves us with:

$$\begin{aligned} & (x_2 \vee x_3) \wedge (\neg x_3) \wedge (\neg x_2 \vee x_3) \\ \leftrightarrow\ & (x_2 \vee \bot) \wedge (\top) \wedge (\neg x_2 \vee \bot) \\ \leftrightarrow\ & (x_2) \wedge (\neg x_2) \end{aligned}$$

After we propagate x_3, we arrive at a contradiction since its propagation implies that x_2 must be assigned to $true$ and $false$. We can then conclude that there is no completion to the partial interpretation $I = \{x_0, x_1\}$ that will satisfy the formula.

2.2 DPLL

BCP can be used to improve the brute-force search SAT solver developed in Lab 3. The natural place to insert this optimization is at the beginning of the DPLL procedure, before `F` is further inspected and any choices are made. This will ensure that if we are

given a formula that is already reducible to a constant through BCP, then we won't do any unnecessary work by deciding values that don't matter. The resulting procedure is called the David-Putnam-Loveland-Logemann or DPLL procedure, as it was introduced by Martin Davis, Hilary Putnam, George Logemann, and Donald Loveland in the 1960s [DP60, DLL62].

```
let rec dpll (F:formula) : bool =
  let Fp = BCP F in
  if Fp = true then true
  else if Fp = false then false
  else begin
    let p = choose_atom(Fp) in
    let Ft = (subst Fp p true) in
    let Ff = (subst Fp p false) in
    dpll Ft || dpll Ff
  end
```

Using an implementation that resembles the one above for such problems would not yield good results in practice. One immediate problem is that the formula is copied multiple times and mutated in-place with each recursive call. While this makes it easy to keep track of which variables have already been assigned or implied via propagation, even through backtracking, it is extremely slow and cumbersome.

Modern solvers address this by using imperative loops rather than recursive calls, and mutating an interpretation rather than the formula itself. The interpretation remains *partial* throughout most of the execution, which means that parts of the formula cannot be evaluated fully to a constant, but are instead *unresolved*.

Definition 1 (Status of a clause under partial interpretation)**.** Given a partial interpretation I, a clause is:

- Satisfied, if one or more of its literals is satisfied
- Conflicting, if all of its literals are assigned but not satisfied
- Unit, if it is not satisfied and all but one of its literals are assigned
- Unresolved, otherwise

For example, given the partial interpretation $I = \{x_1, \neg x_2, x_4\}$:

$(x_1 \vee x_3 \vee \neg x_4)$ is satisfied

$(\neg x_1 \vee x_2)$ is conflicting

$(\neg x_2 \vee \neg x_4 \vee x_3)$ is unit

$(\neg x_1 \vee x_3 \vee x_5)$ is unresolved

As we discussed earlier, when a clause C is unit under partial interpretation I, I must be extended so that C's unassigned literal ℓ is satisfied. There is no need to backtrack on ℓ before the assignments in I that made C unit have already changed, because ℓ's

value was implied by those assignments. Rather, backtracking can safely proceed to the *most recent decision*, erasing any assignments that arose from unit propagation in the meantime. Implementing this backtracking optimization correctly is essential to an efficient SAT solver, as it is what allows DPLL to avoid explicitly enumerating large portions of the search space in practice.

3 Efficient Data Structures for BCP

In the previous section, we have seen that a clause can be in one of four status: satisfied, conflicting, unit and unresolved. The *conflicting* and *unit* status are quite important. The former is an indicator that the current partial interpretation cannot be further extended and the solver needs to backtrack. The latter informs the solver that a literal can be propagated via BCP. A naive approach to detect the status of each clause would be to perform a linear search on all clauses of a propositional formula and for each clause check its corresponding status. Even though this is likely to be the approach followed by many of you in your implementation of Lab 4, it is not the most efficient implementation of BCP. In order to improve its performance, modern SAT solvers use efficient data structures that improve the detection of the status of a clause.

3.1 Adjacency Lists

GRASP [MSS96] was the first SAT solver (1996) to use clause learning and non-chronological backtracking and used adjacency lists as its underlying data structures for BCP.

Consider the same CNF formula shown in Section 2.1:

$$\underbrace{(x_2 \vee x_3)}_{C_0} \wedge \underbrace{(\neg x_1 \vee \neg x_3)}_{C_1} \wedge \underbrace{(\neg x_1 \vee \neg x_2 \vee x_3)}_{C_2} \wedge \underbrace{(x_0 \vee x_1 \vee \neg x_3)}_{C_3} \wedge \underbrace{(\neg x_0 \vee x_1 \vee x_3)}_{C_4}$$

When we assign x_1 to $true$ we do not need to check if all clauses are *unit* clauses! We only need to check the clauses that contain $\neg x_1$ (C_1 and C_2) since those are the only ones that can be turned into unit clauses when x_1 is assigned to $true$. If the solver keeps an adjacency list for each literal l_i to all clauses where $\neg l_i$ occurs, then we significantly reduce the number of clauses that need to be checked.

For example, for the literals x_1 and $\neg x_1$ we would keep the following adjacency lists:

$$x_1 \mapsto \{C_1, C_2\}$$
$$\neg x_1 \mapsto \{C_3, C_4\}$$

Even though we reduced the number of clauses that needs to be checked, for each clause we still need to perform a linear search on all its literals to check the corresponding status. Can we check the status of a clause without checking all its literals? To improve this potential bottleneck, we will need to augment the clause data structure with counters for the number of satisfied and unsatisfied literals. Using these counters we can conclude the following:

- If the number of unsatisfied literals is the same as the size of the clause than we can conclude that the clause is conflicting;
- If the number of satisfied literals is larger or equal to 1 than we can conclude that the clause is satisfied;
- If the number of satisfied literals is 0 and the number of unsatisfied literals is the same the size of the clause minus 1 than we can conclude that the clause is unit;
- On the remaining cases we can conclude that the clause is unresolved.

For example, when we have the partial interpretation $I = \{x_0, x_1, \neg x_3\}$, then we would have the following counters associated with each clause:

- C_0: 1 satisfied, 1 unsatisfied, size 2;
- C_1: 1 satisfied, size 2;
- C_2: 3 unsatisfied, size 3;
- C_3: 3 satisfied, size 3;
- C_4: 1 satisfied, 2 unsatisfied, size 3.

Since C_2 has 3 unsatisfied literals and its size is 3 than we can conclude that it is a conflicting clause without needing to traverse. Note that every time you assign a literal l_i to $true$ than you would need to increase the satisfied counter of all clauses c_i that contain l_i. Similarly, you would also need to increase the unsatisfied counter of all clauses c_i that contain $\neg l_i$. When backtracking, a similar procedure needs to be done by decrease the value of the counters accordingly. The invariant that this data structure maintains is that for all partial interpretations I, the counters must precisely track the number of satisfied and unsatisfied literals for each clause.

Another potential overhead of adjacency lists is that if l_i occurs in all clauses than we will still need to traverse all clauses. Can we do better? In the next two subsections, we will study *lazy data structures* [2] for BCP. Their key insight is that we do not need to know if a clause is *satisfied* or *unresolved*. We only need to know when a clause is *unit* or *conflicting* and to detect if a clause is unit it suffices to track *two* literals of that clause. [3]

3.2 Head-Tail

The Head-Tail data structure [Zha97, MSLM09] associates two references with each clause, the head (H) and the tail (T) literal reference. Initially, the head reference points

[2] Description of lazy data structures and figures based on [MSLM09].

[3] When a formula is satisfiable, the DPLL algorithm terminates if all clauses are satisfied by the current partial interpretation. If we use lazy data structures than we do not know when a clause is satisfied. However, we can change our termination criterion to be when all variables are assigned a truth value and there is no conflicting clauses.

to the first literal and the tail reference points to the last literal. Each time a literal pointed to by either the or tail is assigned, a new unassigned literal is search for. Both pointers move towards to the middle of the clause. In case an unassigned literal is identified, it becomes the new head (or tail) reference, and a new reference is created and associated with the literal's variable. These references guarantee that H/T positions are correctly recovered when the search backtracks. In case a satisfied literal is identified, the clause is declared satisfied. In case no unassigned literal can be identified, and the other reference is reached, then the clause is declared unit, unsatisfied or satisfied, depending on the value of the literal pointed to by the other reference. In contrast to the adjacency lists, in the H/T data structure we do not need to keep a reference from l_i to all clauses where $\neg l_i$ occurs. Instead, we only need to keep references to where $\neg l_i$ occurs and is either the head or tail pointer.

Let i_h the index of the head and i_t the index of the tail in a given clause c_i. The H/T data structure maintains the invariant that all literals in index j with $0 \leq j < i_h$ are unsatisfied. Similarly, all literals in index k with $i_t \leq k < n$ (where n i the size of the clause) are also unsatisfied. Maintaining this invariant is costly when backtracking. When the search process backtracks, the references that have become associated with the head and tail references can be discarded, and the previous head and tail references become activated. Observe that this requires in the worst-case associating with each clause a number of literal references in variables that equals the number of literals.

This data structure is illustrated in Figure 1 (left). We illustrate the H/T data structure for one clause for a sequence of assignments. Each clause is represented by an array of literals. Literals have different representations depending on being unassigned, assigned value 0 (unsatisfied) or assigned value 1 (satisfied). Each assigned literal is associated with a decision level indicating the level where the literal was assigned. In addition, we represent the head (H) and tail (T) pointers that point to a specific literal. Initially, the H/T pointer points to the left/rightmost literal, respectively. These pointers only point to unassigned literals. Hence, each time one literal pointed by one of these pointers is assigned, the pointer has to move inwards. However, a new reference for the just assigned literal is created (represented with a dash line). When the two pointers reach the same unassigned literal the clause is unit. When the search backtracks, the H/T pointers must be moved. The pointers are now placed at its previous positions, i.e. at the position they were placed before being moved inwards.

3.3 Two Watched-Literals

The more recent Chaff SAT solver [MMZ+01, MSLM09] proposed a new data structure, the Watched Literals (WL), that solves some of the problems posed by H/T lists. As with H/T lists, two references are associated with each clause. However, and in contrast with H/T lists, there is no order relation between the two references, allowing the references to move in any direction. The lack of order between the two references has the key advantage that no literal references need to be updated when backtracking takes place. In contrast, unit or unsatisfied clauses are identified only after traversing all the clauses literals; a clear drawback. The identification of satisfied clauses is similar

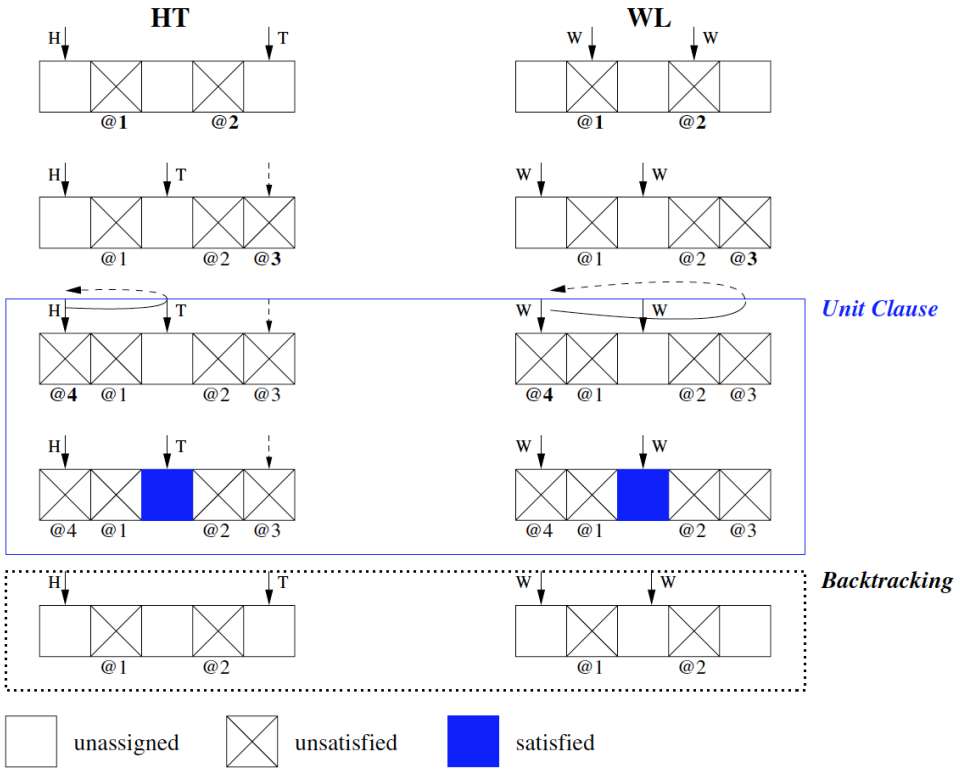

Figure 1: Lazy data structures for BCP [MSLM09]

to H/T lists. The most significant difference between H/T lists and watched literals occurs when the search process backtracks, in which case the references to the watched literals are not modified. Consequently, and in contrast with H/T lists, there is no need to keep additional references. This implies that for each clause the number of literal references that are associated with variables is kept constant. This data structure is also illustrated in Figure 1 (right). The two watched literal pointers are undifferentiated as there is no order relation. Again, each time one literal pointed by one of these pointers is assigned, the pointer has to move inwards. However, in contrast with the H/T data structure, these pointers may move in both directions. This causes the whole clause to be traversed when the clause becomes unit. In addition, no references have to be kept to the just assigned literals, since pointers do not move when backtracking.

Note: All these data structures were created with non-chronological backtracking in mind. Since your SAT solver implementation is likely not to include clause learning, you will be performing chronological backtracking, i.e. when a conflict is detected only the last decision is undo. Therefore, you may not fully benefit from some of the gains of these data structures (e.g. backtracking being free with the two-watched scheme).

For part II of Lab 4, if you choose to improve your data structures, we would suggest to use adjacency lists. Since BCP is performed very frequently, any improvement to its data structures is going to have a huge impact on the performance of your SAT solver.

4 SAT Pre-processing

Formula simplification can significantly improve the performance of SAT solvers and is commonly used on modern SAT solvers. The main goal of pre-processing is to transform a propositional formula φ into an *equivalent* or *equisatisfible* formula φ' that is easier to solve than the original formula.

Definition 2 (Equivalent)**.** Two formulas φ and ϕ are equivalent iff all interpretations of φ are also interpretations of ϕ and vice versa.

Definition 3 (Equisatisfiable)**.** Two formulas φ and ϕ are equisatisfiable if φ is satisfiable whenever ϕ is satisfiable and vice versa, i.e. either both formulas are satisfiable or both are unsatisfiable. However, contrary to equivalence, φ and ϕ do not need to have the same interpretations to be equisatisfiable.

From the pre-processing techniques presented in this section, variable elimination is the only technique that solely preserves equisatisfiablity. The remaining techniques preserve formula equivalence.

4.1 Pure literal rule

Any atom that only appears in either positive or negative literals is called *pure*, and their corresponding atoms must always be assigned in a way that makes the literal $true$. Thus, they do not constrain the problem in a meaningful way, and can be assigned without making a choice. This is called *pure literal elimination* and is one type of simplification that can be applied to CNF formulas.

Consider the following CNF formula:

$$\underbrace{(x_1 \vee x_2)}_{C_0} \wedge \underbrace{(\neg x_1 \vee x_2)}_{C_1} \wedge \underbrace{(x_1 \vee \neg x_2 \vee x_3)}_{C_2} \wedge \underbrace{(\neg x_1 \vee x_2 \vee x_3)}_{C_3}$$

Notice that x_3 appears only as a positive literal in this formula. Hence, we can assign x_3 to $true$ and satisfy the literal. This procedure will simplify the above formula into:

$$\begin{aligned} & (x_1 \vee x_2) \wedge (\neg x_1 \vee x_2) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_1 \vee x_3) \\ \leftrightarrow\ & (x_1 \vee x_2) \wedge (\neg x_1 \vee x_2) \wedge (x_1 \vee \neg x_2 \vee \top) \wedge (\neg x_1 \vee x_1 \vee \top) \\ \leftrightarrow\ & (x_1 \vee x_2) \wedge (\neg x_1 \vee x_2) \end{aligned}$$

In practice, pure literal elimination can significantly reduce the complexity of propositional formulas, and so it is sometimes used as a pre-processing simplification before handing the formula to a solver. However, oftentimes there are no pure literals in a formula.

4.2 Resolution rule

Recall the resolution rule presented in Lecture 13:

$$\text{(res)}\ \frac{\Gamma \vdash P, \Delta \quad \Gamma \vdash \neg P, \Delta}{\Gamma \vdash \Delta}$$

We can use the resolution rule to derive additional clauses to add to a propositional formula. Notice that these resolvents (the succedent of a resolution rule is called a resolvent) are implied by the formula but may allow unit propagation to infer additional information. There is a trade-off between adding too many clauses to the formula and the benefit that these clauses may bring. In practice, one may restrict the number of clauses added to the formula by only adding resolvents of a given size (e.g. binary resolvents).

Consider the same CNF formula shown in Section 2.1:

$$\underbrace{(x_2 \vee x_3)}_{C_0} \wedge \underbrace{(\neg x_1 \vee \neg x_3)}_{C_1} \wedge \underbrace{(\neg x_1 \vee \neg x_2 \vee x_3)}_{C_2} \wedge \underbrace{(x_0 \vee x_1 \vee \neg x_3)}_{C_3} \wedge \underbrace{(\neg x_0 \vee x_1 \vee x_3)}_{C_4}$$

If we resolve C_0 and C_2 on x_2 then we can infer the following clause that can be added to the formula:

$$\text{(res)}\ \frac{(x_2 \vee x_3) \quad (\neg x_1 \vee \neg x_2 \vee x_3)}{(x_3 \vee \neg x_1)}$$

Note that a clause may have several literals where resolution can be performed. When performing resolution, we can observe that the clause $(x_3 \vee \neg x_1)$ is equivalent to $(x_3 \vee x_3 \vee \neg x_1)$. One may also infer tautologies when applying the resolution rule.

Definition 4 (Tautology)**.** A clause ω is a tautology iff it contains a literal l_i with both negative and positive polarities. For example, $(x_1 \vee \neg x_1)$ is a tautology since it is true in all interpretations.

Finally, if the resolution rule derives the empty clause. Then we can conclude that the formula is unsatisfiable. For instance, if the formula contains the clause (x_1) and $(\neg x_1)$ then the formula is unsatisfiable and resolution derives an empty clause:

$$\text{(res)}\ \frac{(x_1) \quad (\neg x_1)}{()}$$

4.3 Variable Elimination

The resolution rule can be recursively applied to determine the satisfiability of a propositional formula. However, this approach suffers from an exponential blow up in memory.

Definition 5 (Variable elimination)**.** A variable x_i can be eliminated from a propositional formula φ by resolving *all* pairs of clauses that contain both x_i and $\neg x_i$. The resolvents can be added to φ and the clauses that contain x_i can be removed from the formula. The new formula φ' is equisatisfiable to φ.

Consider the same CNF formula shown in Section 2.1:

$$\underbrace{(x_2 \vee x_3)}_{C_0} \wedge \underbrace{(\neg x_1 \vee \neg x_3)}_{C_1} \wedge \underbrace{(\neg x_1 \vee \neg x_2 \vee x_3)}_{C_2} \wedge \underbrace{(x_0 \vee x_1 \vee \neg x_3)}_{C_3} \wedge \underbrace{(\neg x_0 \vee x_1 \vee x_3)}_{C_4}$$

Assume we want to eliminate variable x_1 from the above formula. We start by identifying all clauses where x_1 or $\neg x_1$ occurs.

- $x_1 \mapsto \{C_3, C_4\}$
- $\neg x_1 \mapsto \{C_1, C_2\}$

Now we resolve all pairs of clauses that contain x_1 and $\neg x_1$:

$$\text{(res)} \quad \frac{C_3 : (x_0 \vee x_1 \vee \neg x_3) \quad C_1 : (\neg x_1 \vee \neg x_3)}{R_1 : (x_0 \vee \neg x_3)}$$

$$\text{(res)} \quad \frac{C_3 : (x_0 \vee x_1 \vee \neg x_3) \quad C_2 : (\neg x_1 \vee \neg x_2 \vee x_3)}{R_2 : (\neg x_2 \vee x_3 \vee x_0 \vee \neg x_3)}$$

$$\text{(res)} \quad \frac{C_4 : (\neg x_0 \vee x_1 \vee x_3) \quad C_1 : (\neg x_1 \vee \neg x_3)}{R_3 : (\neg x_0 \vee x_3 \vee \neg x_3)}$$

$$\text{(res)} \quad \frac{C_4 : (\neg x_0 \vee x_1 \vee x_3) \quad C_2 : (\neg x_1 \vee \neg x_2 \vee x_3)}{R_4 : (\neg x_0 \vee x_3 \vee \neg x_2 \vee x_3)}$$

Note that the clauses R_2, R_3 and R_4 are tautologies. We can simplify the initial formula by removing C_1, C_2, C_3 and C_4 from the formula and add the resolvent R_1. After eliminating x_3 from the initial formula we obtain:

$$\underbrace{(x_2 \vee x_3)}_{C_0} \wedge \underbrace{(x_0 \vee \neg x_3)}_{R_1}$$

Observe that resolving all pairs of clauses that contain a variable can increase the number of clauses from the simplified formula. Therefore, resolution is often used as a preprocessing technique only when the number of resolvents is *smaller or equal* to the number of clauses that contain the variable to be eliminated.

4.4 Failed literal rule

BCP can also be used as a preprocessing technique. Let φ be a propositional formula, and l_i ($\neg l_i$) a literal to be propagated. If propagating l_i ($\neg l_i$) in φ leads to a conflict, than we can conclude that l_i ($\neg l_i$) must be assigned to *false* (*true*) in all interpretations I of φ. We call this procedure the *failed literal rule*.

Consider again the CNF formula shown in Section 2.1:

$$\underbrace{(x_2 \vee x_3)}_{C_0} \wedge \underbrace{(\neg x_1 \vee \neg x_3)}_{C_1} \wedge \underbrace{(\neg x_1 \vee \neg x_2 \vee x_3)}_{C_2} \wedge \underbrace{(x_0 \vee x_1 \vee \neg x_3)}_{C_3} \wedge \underbrace{(\neg x_0 \vee x_1 \vee x_3)}_{C_4}$$

Performing BCP with the interpretation $I = \{x_1\}$ leads to:

$$\begin{aligned}
&(x_2 \vee x_3) \wedge (\bot \vee \neg x_3) \wedge (\bot \vee \neg x_2 \vee x_3) \wedge (x_0 \vee \top \vee \neg x_3) \wedge (\neg x_0 \vee \top \vee x_3) \\
\leftrightarrow\ &(x_2 \vee x_3) \wedge (\neg x_3) \wedge (\neg x_2 \vee x_3) \\
\leftrightarrow\ &(x_2 \vee \bot) \wedge (\top) \wedge (\neg x_2 \vee \bot) \\
\leftrightarrow\ &(x_2) \wedge (\neg x_2)
\end{aligned}$$

Since propagating x_1 leads to a conflict and no other decision has been made, then it means that x_1 needs to be assigned to *false* in all interpretations of φ.

4.5 Probing

BCP can also be used for *probing*. The key idea behind probing is to propagate l_i and $\neg l_i$ and see if any literal l_j is implied by both propagations. If this is the case than we can conclude that l_j must be assigned to $true$ in all interpretations of φ.

Consider the following CNF formula:

$$\underbrace{(x_1 \vee x_2)}_{C_0} \wedge \underbrace{(\neg x_1 \vee x_2)}_{C_1} \wedge \underbrace{(x_1 \vee \neg x_2 \vee x_3)}_{C_2} \wedge \underbrace{(\neg x_1 \vee x_2 \vee x_3)}_{C_3}$$

When we propagate x_1 and $\neg x_1$, we have the following:

- BCP$(\varphi, x_1) \mapsto \{x_2\}$
- BCP$(\varphi, \neg x_1) \mapsto \{x_2\}$

Therefore, we can conclude that x_2 must be assigned to $true$ in all interpretations of the formula.

References

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Design Automation Conference*, pages 530–535. ACM, 2001.

[MSLM09] Joao Marques-Solva, Inês Lynce, and Sharad Malik. Conflict-Driven Clause Learning SAT Solvers. In *Handbook of Satisfiability*, pages 131–153. IOS Press, 2009.

[MSS96] Joao Marques-Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. In *ICCAD*, pages 220–227. IEEE Computer Society, 1996.

[Zha97] Hantao Zhang. SATO: an efficient propositional prover. In *International Conference on Automated Deduction*, pages 272–275. Springer, 1997.