Off the Trail: Re-examining the CDCL Algorithm*

Alexandra Goultiaeva and Fahiem Bacchus

Department of Computer Science
University of Toronto
{alexia,fbacchus}@cs.toronto.edu

Abstract. Most state of the art SAT solvers for industrial problems are based on the Conflict Driven Clause Learning (CDCL) paradigm. Although this paradigm evolved from the systematic DPLL search algorithm, modern techniques of far backtracking and restarts make CDCL solvers non-systematic. CDCL solvers do not systematically examine all possible truth assignments as does DPLL.

Local search solvers are also non-systematic and in this paper we show that CDCL can be reformulated as a local search algorithm: a local search algorithm that through clause learning is able to prove UNSAT. We show that the standard formulation of CDCL as a backtracking search algorithm and our new formulation of CDCL as a local search algorithm are equivalent, up to tie breaking.

In the new formulation of CDCL as local search, the trail no longer plays a central role in the algorithm. Instead, the ordering of the literals on the trail is only a mechanism for efficiently controlling clause learning. This changes the paradigm and opens up avenues for further research and algorithm design. For example, in QBF the quantifier places restrictions on the ordering of variables on the trail. By making the trail less important, an extension of our local search algorithm to QBF may provide a way of reducing the impact of these variable ordering restrictions.

1 Introduction

The modern CDCL algorithm has evolved from DPLL, which is a systematic search through variable assignments [4]. CDCL algorithms have evolved through the years, various features and techniques have been added [10] that have demonstrated empirical success. These features have moved CDCL away from exhaustive search, and, for example, [9] has argued that modern CDCL algorithms are better thought of as guided resolution rather than as exhaustive backtracking search.

New features have been added as we have gained a better understanding of CDCL both through theoretical developments and via empirical testing. For example, the important technique of restarts was originally motivated by theoretical and empirical studies of the effect of heavy-tailed run-time distributions [7] on solver run-times.

* Supported by Natural Sciences and Engineering Research Council of Canada.

A. Cimatti and R. Sebastiani (Eds.): SAT 2012, LNCS 7317, pp. 30–43, 2012.

Combinations of features, however, can sometimes interact in complex ways that can undermine the original motivation of individual features. For example, phase saving, also called light-weight component caching, was conceived as a progress saving technique, so that backtracking would not retract already discovered solutions of disjoint subproblems [12] and then have to spend time rediscovering these solutions. However, when we add phase saving to restarts, we reduce some of the randomization introduced by restarts, potentially limiting the ability of restarts to short-circuit heavy-tailed run-times. Nevertheless, even when combined, restarts and phase saving both continue to provide a useful performance boost in practice and are both commonly used in CDCL solvers.

When combined with a strong activity-based heuristic, phase saving further changes the behavior of restarts. In this context it is no longer obvious that restarts serve to move the solver to a different part of the search space. Instead, it can be shown empirically that after a restart a large percentage of the trail is re-created exactly as it was prior to the restart, indicating that the solver typically returns to the same part of the search space. In fact, there is evidence to support the conclusion that the main effect of restarts in current solvers is simply to update the trail with respect to the changed heuristic scores. For example, [14] show that often a large part of the trail can be reused after backtracking. With the appropriate implementation techniques reusing rather than reconstructing the trail can speed up the search by reducing the computational costs of restarts.

In this paper we examine another feature of modern SAT solvers that ties them with the historical paradigm of DPLL: the trail used to keep track of the current set of variable assignments. We show that modern SAT solvers, in which phase savings causes an extensive recreation of the trail after backtracking, can actually be reformulated as local search algorithms.

Local search solvers work with complete truth assignments [15], and a single step usually consists of picking a variable and flipping its value. Local search algorithms have borrowed techniques from CDCL. For example, unit propagation has been employed [6,8,2], and clause learning as also been used [1]. However, such solvers are usually limited to demonstrating satisfiability, and often cannot be used to reliably prove UNSAT. Our reformulation of the CDCL algorithm yields a local search algorithm that is able to derive UNSAT since it can perform exactly the same steps as CDCL would. It also gives a different perspective on the role of the trail in CDCL solvers. In particular, we show that the trail can be viewed as providing an ordering of the literals in the current truth assignment, an ordering that can be used to guide clause learning. This view allows more flexible clause learning techniques to be developed, and different types of heuristics to be supported. It also opens the door for potentially reformulating QBF algorithms, which suffer from strong restrictions on the ordering of the variables on the trail.

Section 2 examines the existing CDCL algorithm and describes our intuition in more detail. Section 3 presents a local search formulation of the modern CDCL algorithm and proves that the two formulations are equivalent. Section 4 presents some simple experiments which suggest further directions for research. Section 5 concludes the paper.

Algorithm 1: Modern CDCL algorithm

```
Data: φ—a formula in CNF
Result: TRUE if φ is SAT, FALSE if φ is UNSAT
π ← ∅ ; C ← ∅ while TRUE do
    π ← unitPropagate(φ ∪ C, π)
    if reduce(φ ∪ C, π) contains an empty clause then
        c′ ← clauseLearn(π, φ ∪ C)
        if c′ = ∅ then return FALSE
        C = C ∪ {c′}
        π = backtrack(c′)
    else if φ is TRUE under π then return TRUE
    else
        v ← unassigned variable with largest heuristic value
        v ← phase[v]
        π.append(v)
    end
    if timeToRestart() then backtrack(0)
end
```

2 Examining the CDCL Algorithm

A modern CDCL algorithm is outlined in Algorithm 1. Each iteration starts by adding literals implied by unit propagation to the trail π. If a conflict is discovered clause learning is performed to obtain a new clause $c' = (\alpha \rightarrow y)$. The new clause is guaranteed to be **empowering**, which means that it is able to produce unit implications in situations when none of the old clauses can [13]. In this case, c' generates a new implication y earlier in the trail, and the solver backtracks to the point where the new implication would have been made if the clause had previously been known. Backtracking removes part of the trail in order to add the new implication in the right place. On the next iteration unit propagation will continue adding implications, starting with the newly implied literal y. If all variables are assigned without a conflict, the formula is satisfied. Otherwise, the algorithm picks a decision variable to add to the trail. It picks an unassigned variable with the largest heuristic value, and restores its value to the value it had when it was last assigned. The technique of restoring the variable's value is called **phase saving**. We will say that the **phase** of a variable v, $phase[v]$, is the most recent value it had; if v has never been assigned, $phase[v]$ will be an arbitrary value set at the beginning of the algorithm; if v is assigned, $phase[v]$ will be its current value.

Lastly, sometimes the solver restarts: it removes everything from the trail except for literals unit propagated at the top level. This might be done according to a set schedule, or some heuristic [3].

As already mentioned, after backtracking or restarting, the solver often recreates much of the trail. For example, we found that the overwhelming majority of assignments Minisat makes simply restore a variable's previous value. We

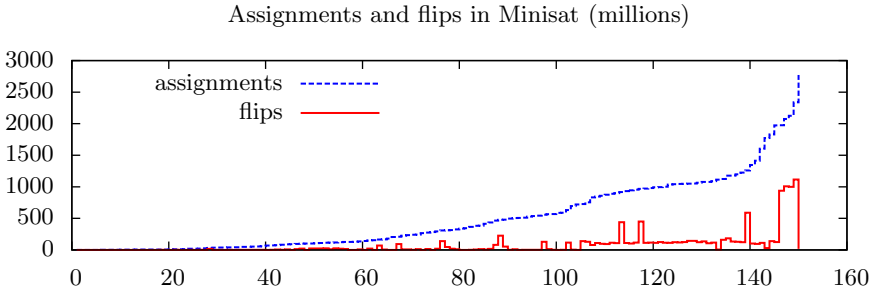

Fig. 1. Assignments and flips on both solved and unsolved (after a 1000s timeout) instances of SAT11 dataset. Sorted by the number of assignments.

have ran Minisat on the 150 problems from the SAT11 dataset of the SAT competition, with a timeout of 1000 seconds. Figure 1 shows the distribution of assignments Minisat made, and the number of "flips" it made, where flips are when a variable is assigned a different value than it had before. On average, the solver performed 165.08 flips per conflict, and 3530.4 assignments per conflict. It has already been noted that flips can be correlated with the progress that the solver is making [3].

Whenever the solver with phase saving backtracks, it removes variable assignments, but unless something forces the variable to get a different value, it would restore the old value when it gets to it. So, we can imagine that the solver is working with a complete assignment, which is the phase settings for all the variables $phase[v]$, and performing a flip from $\neg l$ to l only in one of the following cases. (1) l is implied by a new conflict clause. (2) l is implied by a variable that was moved up in the trail because its heuristic value was upgraded. Or (3) l is implied by another "flipped" variable. Phase saving ensures that unforced literals, i.e., decisions, cannot be flipped.

In all of these cases l is part of some clause c that is falsified by the current "complete" assignment (consisting of the phase set variables); c would then become its reason clause; at the point when l is flipped, c is the earliest encountered false clause; and l is the single unassigned variable in c (i.e., without c, l would have been assigned later in the search). As we will see below, we can use these conditions to determine which variable to flip in a local search algorithm.

Note that we will not consider the randomization of decision variables in this paper, although this could be accommodated by making random flips in the local search algorithm. The benefits of randomizing the decision variables are still poorly understood. In our experiments we found that turning off randomization does not noticeably harm performance of Minisat. Among ten runs with different seeds, Minisat solved between 51 and 59 instances, on average 55. With randomization turned off, it solved 56.

Algorithm 2: Local Search

```
Data: φ - a formula in CNF
Result: TRUE if φ is SAT, FALSE if φ is UNSAT
while TRUE do
    I ← initValues()
    while φ|_I contains FALSE clauses do
        if timeToRestart() then break
        v ← pickVar(I)
        flip(v)
    end
end
return TRUE
```

3 Local Search

Algorithm 2 presents a generic local search algorithm. A local search solver works with a complete assignment I. At each stage in the search, it picks a variable and flips its value. There are different techniques for choosing which variable to flip, from simple heuristics such as minimizing the number of falsified clauses [15], to complicated multi-stage filtering procedures [16].

Typically, the algorithm tries to flip a variable that will reduce the distance between the current complete assignment and a satisfying assignment. However, estimating the distance to a solution is difficult and unreliable, and local search solvers often get stuck in local minima. It was noted that it is possible to escape the local minimum by generating new clauses that would steer the search. Also, if new non-duplicated clauses are being generated at every local minimum, the resulting algorithm can be shown to be complete. An approach exploiting this fact was proposed, using a single resolution step to generate one new clause at each such point [5]. The approach was then extended to utilize an implication graph, and incorporate more powerful clause learning into a local search solver, resulting in the CDLS algorithm [1]. However, as we will see below, CDLS cannot ensure completeness because the clause learning scheme it employs can generate redundant clauses.

The main difficulty for such an approach is the generation of an implication graph from the complete assignment I. The first step consists of identifying **once-satisfied** clauses. A clause c is considered to be **once-satisfied** by a literal x and a complete assignment I if there is exactly one literal $x \in c$ that is true in I ($c \cap I = \{x\}$).

Theoretically, any clause c_f with $\neg x \in c_f$ that is false under I can be resolved with any clause c_o that is once-satisfied by literal x. This resolution would produce a non-tautological clause c_R which is false under I and which can potentially be further resolved with other once-satisfied clauses. However, in order to be useful, the algorithm performing such resolutions needs to ensure that it does not follow a cycle or produce a subsumed clause.

In order to avoid cycles, it is sufficient to define some ordering ψ on variables in I, and only allow the resolution of falsified clauses c_f and once-satisfied clauses c_o when all of the false literals in c_o precede the satisfying literal in the ordering ψ. However, a simple ordering does not ensure that the new clauses are useful.

Clause learning can be guided more effectively by considering the effects of unit propagation. We define an **ordering** ψ on the complete assignment I to be a sequence of literals $\psi = \{x_1, x_2, \ldots, x_k\}$ from I ($\forall i. x_i \in I$). A literal $x_i \in \psi$ is **implied** in ψ if there is some clause $(\neg x_{j_1}, \ldots, \neg x_{j_n}, x_i)$ with $j_1 < j_2 < \cdots < j_n < i$. In this case j_n is called an **implication point** for x_i (the implication point is 0 if the clause is unit). x_i is said to be **implied at** k if k is the smallest implication point for x_i.

Finally, an ordering ψ will be said to be **UP-compatible** if for any $x_i \in \psi$, if x_i is implied at j_n, then it must appear in the ordering ψ as soon after j_n as possible. In particular, UP-compatibility requires that any literals between x_i and its smallest implication point j_n, i.e., the literals $x_{j_{n+1}}, \ldots, x_{i-1}$, also be implied in ψ. For example, for a set of clauses (a), $(\neg a, \neg b, \neg c)$, $(\neg c, d)$, the orderings $\{a, c, d, b\}$ and $\{c, d, b, \neg a\}$ are UP-compatible, but $\{c, d, b, a\}$ or $\{c, b, d, \neg a\}$ are not. In the first case, a is implied by the clause (a), but follows non-implied c. In the second, d is implied by c with $(\neg c, d)$, but follows non-implied b.

A CDCL solver that ignores all conflicts would produce a UP-compatible ordering. However, not every UP-compatible ordering can be produced by a CDCL solver. This is because the definition considers only the given assignment, and does not take into account falsified clauses. So, it is possible that for some literal x_i, $\neg x_i$ is implied by a smaller prefix of the assignment, but this implication is ignored because it disagrees with the current assignment.

Given a complete assignment I and a UP-compatible ordering ψ we can define a **decision literal** to be any literal in ψ that is not implied. For each $x_i \in \psi$, we can define the decision level of x_i to be the number of decision literals in $\{x_1, x_2, \ldots, x_i\}$. For each implied literal, we can say that its reason is the clause that implied it. Note that the reason clauses are always once-satisfied by I. So, the ordering ψ gives us an implication graph over which clause learning can be performed as in a standard CDCL solver.

Consider a false clause $c = (\neg x_{c_1}, \neg x_{c_2}, \ldots, \neg x_{c_n})$ with $c_1 < c_2 < \ldots < c_n$. If $\neg x_{c_n}$ had been in I, it would have been implied at the same decision level as $x_{c_{n-1}}$. We will call such $\neg x_{c_n}$ a **failed implication**. We will say that x_{c_n} is **f-implied** at a decision level i if it is a failed implication at the decision level i but not earlier.

The scheme used by CDLS [1] is to construct a **derived partial interpretation** I'. Let i be the first decision at which a failed implication $\neg x_f$ occurs due to some clause $(\beta, \neg x_f)$. I' is then the prefix of ψ up to and including all variables with decision level i. If $x_f \in I'$, then x_f and $\neg x_f$ are implied at the same decision level, and clause learning can be performed as usual. We will call this kind of failed implication **conflicting**. In this case the execution is identical to a corresponding run of a CDCL solver, so the resulting clause is subject

to all the guarantees a CDCL solver provides. In particular, the new clause is guaranteed to be empowering [13].

If $x_f \notin I'$ then $\neg x_f$ does not cause a conflict. It is a failed implication simply because it is incompatible with the current assignment. We shall call this kind of failed implication a **non-conflicting** implication. In CDLS the learning scheme is only applied when no variable flip is able to reduce the number of falsified clauses. So, there must be some clause (α, x_f) that is once-satisfied by x_i. CDLS then extends I' by including decisions $\{\alpha - I'\}$ as assumptions. In the new I', both α and β are falsified, so both x_f and $\neg x_f$ are implied, which causes a conflict that can be used as a starting point for clause learning. However, no guarantees apply to the new clause in this case. Because the added assumptions are not propagated, it is possible that the newly generated clause is not only not empowering, but is actually subsumed by existing clauses. For example, suppose the formula contains clauses (x_1, x_2, c), $(x_1, x_2, \neg x_3)$ and $(\neg x_3, \neg c)$. Suppose that the current assignment contains $(\neg x_1, \neg x_2, x_3, c)$. If x_3 is chosen as the first decision, the conflict immediately occurs because $\neg c$ is a failed implication at the first level. The implication graph, after adding the necessary assumptions, contains only two clauses, $(\neg x_3, \neg c)$ and (x_1, x_2, c). The resulting clause, $(x_1, x_2, \neg x_3)$, repeats a clause already in the database.

Instead of stopping at the first failed implication, we could use a larger prefix of ψ. Namely, we could apply learning to the first conflicting failed implication. However, this would not guarantee an unsubsumed new clause. It is possible that clause learning generates a clause (α, x) implying x that is the same as one of the previously ignored clauses causing a (non-conflicting) failed implication.

The problem arises because, from the point of view of CDCL, x_f is not a conflict. Instead of doing clause learning, a CDCL algorithm would have flipped x_f and continued with the search. Picking a correct ordering ψ would not help either. The problem here is with objective functions used to guide local search.

The following example is for the objective function that minimizes the number of satisfied clauses. Suppose we have the following clauses: (a, b), (c, d), $(\neg a, c)$, $(\neg b, d)$, $(\neg c, a)$, $(\neg d, b)$. An assignment $\pi = (\neg a, \neg b, \neg c, \neg d)$ is a local minimum: it falsifies two clauses. No literal flip falsifies less, and no ordering of π produces a conflicting implication. If we initially set $\neg a$, the two implications are b and $\neg c$. The first is a failed implication because it disagrees with π. The two possible implications from $\neg c$ are d and $\neg a$. The first is a failed implication, and the second is already set. All the other variables are completely symmetrical.

To avoid this problem, the flips need to be guided using some notion of unit propagation. Intuitively, a non-conflicting failed implication does not give enough information to clause learning, and thus would not produce a useful conflict. So, it should be resolved using a flip rather than clause learning, and should not constitute a local minimum.

Algorithm 3 demonstrates a strategy to guide the local search outlined in Algorithm 2. It selects a UP-compatible ordering ψ on I (of course, this could be updated incrementally and not generated from scratch every time). It then picks the first failed implication on ψ. If it is conflicting, clause learning is performed.

Algorithm 3: pickVar(I)

```
Data: I - a complete assignment
Result: y - next variable to flip
ψ ← UP-compatible ordering on I
y ← first failed implication in ψ
if ¬y is conflicting then
    c ← firstUIP(ψ)
    if c = ∅ then EXIT(FALSE)
    attachClause(c); y ← c.implicate
end
return y
```

Note that the resultant clause c is guaranteed to be FALSE under I, and it would produce a failed implication at an earlier level. If an empty clause is derived, the formula is proven unsatisfiable. Otherwise, the new failed implication is now the earliest, and is non-conflicting, so the new implicate needs to be flipped.

One detail that is left out of the above algorithm is how to pick the ordering ψ. Note that if we are given some base ordering ψ_b, we can construct a UP-compatible ordering ψ in which decision literals respect ψ_b. In this case ψ_b plays the role of the variable selection heuristic. Of course, the heuristic must be chosen carefully so as not to lead the algorithm in cycles. An easy sufficient condition is when ψ_b is only updated after clause learning, as VSIDS is.

3.1 Connection to CDCL

In this section, we will focus on Algorithm 2 guided by the variable selection and clause learning technique presented in Algorithm 3 and with no restarts. We will refer to this as A2. We will refer to Algorithm 1 as A1.

Define a **trace** of an algorithm A to be a sequence of flips performed and clauses learned by A. Note that this definition applies to both A2 and A1: recall that for A1 a flip is an assignment where the variable's new value is different from its phase setting.

Theorem 1. *For any heuristic h there is a heuristic h' such that for any input formula ϕ, A1 with h would produce the same trace as A2 with h' (provided they make the same decisions in the presence of ties).*

Proof. We will say that a heuristic h is **stable** for (a version of) CDCL algorithm A if during any execution of A with h we have $h(v_1) \geq h(v_2)$ for some decision variable v_2 only if $h(v_1) \geq h(v_2)$ also held just before v_2 was last assigned.

Intuitively, a heuristic is stable if the ordering of decision variables is always correct with respect to the heuristic, and is not simply historical. One way to ensure that a heuristic is stable is to restart after every change to the heuristic. For example, the VSIDS heuristic is stable for a version of A1 which restarts after every conflict.

We will first prove the claim for a heuristic h that is stable for A1. Then, we will show that for any h, we can find an equivalent h' that is stable for A1 and such that A1 with h' produces the same trace as with h.

Let the initial assignment of A1 be the same as the initial phase setting of A2, and let both algorithms use the same heuristic h. It is easy to verify that if a partial execution of A1 has the same trace as a partial execution of A2, that means that the phase setting of any variable in A1 matches its value in A2.

To show that A1 and A2 would produce the same trace when run on the same formula ϕ, we will consider partial executions. By induction on n, we can show that if we stop each algorithm just after it has produced a trace of length n, the traces will be identical.

If $n = 0$, the claim trivially holds. Also note that if one algorithm halts after producing trace T, so will the other, and their returned values will match. Both algorithms will return FALSE iff T ends with an empty clause. If A2 has no failed implications, then A1 will restore all variables to their phase values and obtain a solution, and vice versa. Suppose the algorithms have produced a trace T.

Let S be the Next Flip or Learned Clause of A1. Let π be the trail of A1 just before it produced S.

Because h is stable for A1, then the heuristic values of the decision variables in π are non-increasing. That is because if $h(v_1) > h(v_2)$ for two decision variables, then the same must have held when v_2 was assigned. If v_1 had been unassigned at that point, it would have been chosen as the decision variable instead. So, v_1 must have been assigned before v_2.

So, π is a UP-compatible ordering respecting h over the partial assignment: any implication is placed as early as possible in π, and non-implied (decision) literals have non-increasing heuristic value. Because unit propagation was performed to completion (except for possibly the last decision level), and because the heuristic value of all unassigned literals is less than that of the last decision literal, π can always be extended to a UP-compatible ordering ψ on I.

Let $C = \{\alpha, v\}$ be the clause that caused v to be flipped to TRUE if S is a flip; otherwise, let it be the conflicting clause that started clause learning, with v being the trail-deepest of its literals. In both cases, C is FALSE at P_1, so v is a failed implication in ψ. This is the first conflict encountered by A1, so there are no false clauses that consist entirely of literals with earlier decision levels. So, v is the first failed implication in ψ.

If S is a flip, then v is non-conflicting, and A2 would match the flip. Otherwise, v is a conflicting failed implication, and will cause clause learning. For the ψ which matches π, clause learning would produce a clause identical to that produced by A1. So, the next entry in the trace of A2 will also be S.

Let S be the Next Flip or Learned Clause of A2. Let v be the first failed implication just before S was performed, and let ψ be the corresponding UP-compatible ordering. Let π be the trail of A1 just before it produces its next flip or a learned clause. We will show that whenever π differs from ψ, A1 could have broken ties differently to make them match. Let $v_1 \in \pi$ and $v_2 \in \psi$ be the first pair of literals that are different between π and ψ. Suppose v_1 is implied.

Then, because ψ is UP-compatible, v_2 must also be implied by preceding literals. So, A1 could have propagated v_2 before v_1. If v_1 is a decision literal, then so is v_2. Otherwise, v_2 should have been unit propagated before v_1 was assigned. If $h(v_2) < h(v_1)$, this would break the fact that ψ respects the heuristic. So, $h(v_2) \geq h(v_1)$. Then the same must have been true at the time v_1 was assigned. So, v_2 was at worst an equal candidate for the decision variable, and could have been picked instead.

So, provided A1 breaks ties accordingly, it would have the trail π that is a prefix of ψ. It can continue assigning variables until the trail includes all the variables at the decision level at which v is f-implied. Because v is the first failed implication in ψ, no conflicts or flip would be performed up to that point. At this point, there will be some clause $C = (\alpha, v)$. If S is a flip, then v is not conflicting, C will be unit, and a flip will be performed. If S is a learned clause, then v is conflicting, which means that v was among the implied literals at this level. So, clause learning will be performed. Either way, the next entry in the trace of A1 will also be S.

So, we have shown that A2 and A1 would produce the same trace given the same heuristic h' which is stable for A1. Now we will sketch a proof that given any variable heuristic h, we can construct a heuristic h' which is stable for A1 and such that A1 with h would produce the same trace as with h'.

We will define $h'(v) = h(v)$ whenever v is unassigned. Otherwise, we will set $h'(v) = M + V - D + 0.5d$, where M is some value greater than the maximum $h(v)$ of all non-frozen variables, V is the number of variables in the problem, and D is the decision level at which v was assigned when it became frozen, and d is 1 if v is decision and 0 otherwise.

Because a heuristic is only considered for unassigned variables, then the behavior of the algorithm is unaffected, and it will produce the same traces. Also, unassigned values always have a smaller heuristic value than those that are assigned; those assigned later always have a smaller heuristic value than earlier decision literals. So, the heuristic is stable for A1.

As a corollary: because Algorithm 1 is complete, so is Algorithm 2.

3.2 Other Failed Implications

In Algorithm 3 we always choose the first failed implication. However, it is not a necessary condition to generate empowering clauses.

Theorem 2. *Suppose that ψ is UP-compatible ordering on I. Let c be a clause generated by 1UIP on some failed implication x. Suppose $c = (\alpha, y)$ where y is the new implicant. If no failed implication that is earlier than x can be derived by unit propagation from α, then c is empowering.*

Proof. Suppose that c is not empowering. Then y can be derived by unit propagation from α. Because y was not implied by α at that level, then the unit propagation chain contains at least one literal that contradicts the current assignment. Let p be such a literal which occurs first during unit propagation. Then p is a failed implication that can be derived from α.

Note that this is sufficient, but not a necessary condition. It is possible that an earlier failed implication x can be derived from α, but $\alpha \cap x$ still do not allow the derivation of y.

3.3 Potential Extension to QBF Solving

The trail has always played a central role in the formalization of the SAT algorithm. It added semantic meaning to the the chronological sequence of assignments by linking it to the way clause learning is performed.

In SAT, this restriction has no major consequences, since the variables can be assigned in any order. However, in an extension of SAT, Quantified Boolean Formula (QBF) solving, this restriction becomes important.

In QBF variables are either existentially or universally quantified, and the inner variables can depend on the preceding ones. Clause learning utilizes a special **universal reduction** step, which allows a universal to be dropped from a clause if there are no existential variables that depend on it. In order to work, clause learning requires the implication graph to be of a particular form, with deeper variables having larger decision levels. Because of the tight link between the trail and clause learning, the same restriction is applied to the order in which the algorithm was permitted to consider variables. Only outermost variables were allowed to be picked as decision literals.

This restriction is a big impediment to performance in QBF. One illustration of this fact is that there is still a big discrepancy between search-based and expansion based solvers in QBF. The former are constrained to consider variables according to the quantifier prefix, while the latter are constrained to consider the variables in reverse of the quantifier prefix. The fact that the two approaches are incomparable, and that there are sets of benchmarks challenging for one but not the other, suggests that the ordering restriction plays a big role in QBF. Another indication of this is the success of dependency schemes, which are attempts to partially relax this restriction [11].

The reformulation presented here is a step towards relaxing this restriction. We show that the chronological sequence of assignments does not have any semantic meaning, and thus should not impose constraints on the solver. Extending the present approach to QBF should allow one to get an algorithm with the freedom to choose the order in which the search is performed.

To extend to QBF, the definition of UP-compatible ordering would need to be augmented to allow for universal reduction. One way to do this would be to constrain the ordering by quantifier level, to ensure that universal reduction is possible and any false clause would have an implicate. However, this ordering is no longer linked to the chronological sequence of variables considered by the solver, and will be well-defined after any variable flip. At each step, the solver will be able to choose which of the failed implications to consider, according to some heuristic not necessarily linked to its UP-compatible ordering.

So, decoupling the chronological variable assignments from clause learning would allow one to construct a solver that would be free to consider variables in any order, and would still have well-defined clause learning procedure when it encounters a conflict.

Table 1. Problems from SAT11 solved within a 1000s timeout by Minisat with phase saving, and by the modified versions C_n, where n is the number of full runs the solver performs at each restart. C_All is the version that performs exclusively full runs. The number of problems solved is shown for All, True and False instances.

Family	**Minisat**			**C_1**			**C_5**			**C_10**			**C_100**			**C_1000**			**C_All**		
	A	T	F	A	T	F	A	T	F	A	T	F	A	T	F	A	T	F	A	T	F
fuhs (34)	10	**9**	1	10	8	2	9	8	1	9	8	1	10	7	**3**	10	8	2	7	7	0
manthey (9)	3	**3**	0	2	2	0	2	2	0	3	**3**	0	3	**3**	0	1	1	0	0	0	0
jarvisalo (47)	24	8	**16**	24	8	**16**	24	9	15	24	**10**	14	25	9	**16**	17	6	11	15	6	9
leberre (17)	11	4	7	13	**6**	7	14	**6**	**8**	13	**6**	7	13	**6**	7	12	5	7	7	1	6
rintanen (30)	9	**7**	2	8	5	**3**	7	4	**3**	8	5	**3**	8	5	**3**	2	1	1	1	0	1
kullmann (13)	2	2	0	3	3	0	3	3	0	2	2	0	**4**	**4**	0	3	3	0	3	3	0
Total (150)	59	33	26	60	32	28	59	32	27	59	**34**	25	**63**	**34**	**29**	45	24	21	33	17	16

4 Experiments

We have investigated whether subsequent failed implications, mentioned in Section 3.2, can be useful in practice. To evaluate this, we have equipped Minisat with the ability to continue the search ignoring conflict clauses. Note: here we use a version of Minisat with phase saving turned on.

This is equivalent to building a UP-compatible assignment with no non-conflicting failed implications.

For each decision level, it would only store the first conflict clause encountered, because learning multiple clauses from the same decision level is likely to produce redundant clauses. After all the variables are assigned, it would backtrack, performing clause learning on each stored conflict, and adding the new clauses to the database. We will say that one iteration of this cycle is a **full run**.

Obviously, not yet having any method of guiding the selection, the algorithm could end up producing many unhelpful clauses. To offset this problem, and to evaluate whether the other clauses are *sometimes* helpful, we constructed an algorithm that performs a full run only some of the time.

We have added a parameter n so that at every restart, the next n runs of the solver would be full runs. We experimented with $n \in \{1, 5, 10, 100, 1000\}$ and with a version which only performs full runs.

We ran the modified version on the 150 benchmarks from SAT11 set of the Sat Competition, with timeout of 1000 seconds. The tests were run on a 2.8GHz machine with 12GB of RAM.

Table 1 summarizes the results. As expected with an untuned method, some families show improvement, while for others the performance is reduced. However, we see that the addition of the new clauses can improve the results for both satisfiable instances (as in benchmarks sets leberre and kullmann), and unsatisfiable ones (as in fuhs and rintanen).

Figure 2 compares the number of conflicts learned while solving the problems in Minisat and C_100. For instances which only one solver solved, the other solver's value is set to the number of conflicts it learned within the 1000s timeout.

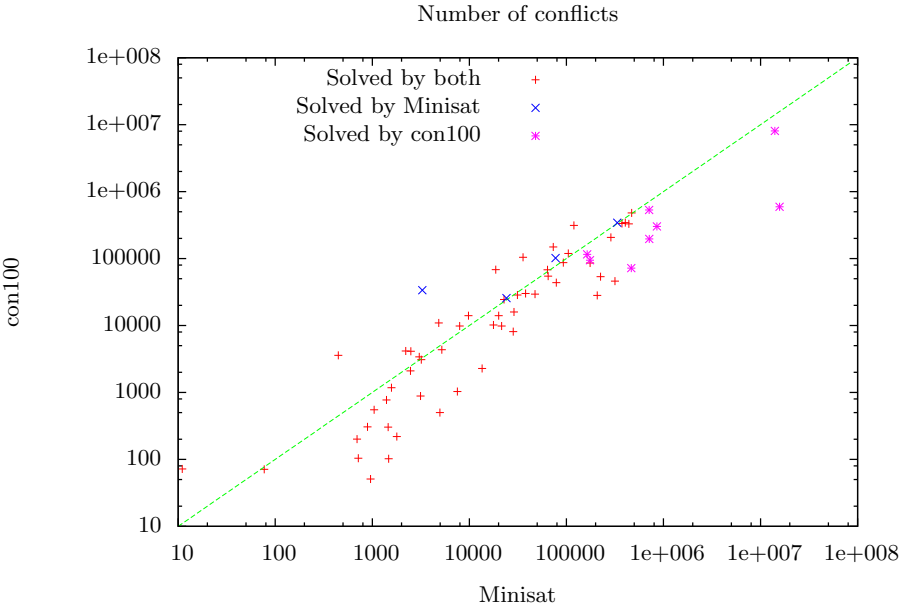

Fig. 2. The number of conflicts needed to solve the problem. Below the line are instances on which C_100 encountered fewer conflicts than Minisat.

We note that in the conflict count for C_100 we include all the conflicts it ever learned, so a single full run might add many conflicts at once. These are unfiltered, so we expect that good heuristics and pruning methods can greatly reduce this number. However, even with all the extra conflicts C_100 encounters, there is a fair number of cases where it needs fewer conflicts to solve the problem than Minisat.

5 Conclusion

We have presented a reformulation of the CDCL algorithm as local search. The trail is shown to be simply an efficient way to control clause learning. By decoupling clause learning from the chronological sequence in which variables are considered, we introduce new flexibility to be studied.

One potential application of this flexibility would be to produce QBF solvers whose search space is not so heavily constrained by the variable ordering. Another is to find good heuristics to choose which conflict clauses are considered during search.

Current CDCL solvers effectively maintain a UP-compatible ordering on the trail by removing the order up to the place affected by a flip, and recomputing it again. An interesting question worth investigating is whether it is possible to develop algorithms to update the order more efficiently.

References

1. Audemard, G., Lagniez, J.-M., Mazure, B., Sais, L.: Learning in local search. In: ICTAI, pp. 417–424 (2009)
2. Belov, A., Stachniak, Z.: Improved Local Search for Circuit Satisfiability. In: Strichman, O., Szeider, S. (eds.) SAT 2010. LNCS, vol. 6175, pp. 293–299. Springer, Heidelberg (2010)
3. Biere, A.: Adaptive Restart Strategies for Conflict Driven SAT Solvers. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 28–33. Springer, Heidelberg (2008)
4. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM 5, 394–397 (1962)
5. Fang, H.: Complete local search for propositional satisfiability. In: Proceedings of AAAI, pp. 161–166 (2004)
6. Gableske, O., Heule, M.J.H.: EagleUP: Solving Random 3-SAT Using SLS with Unit Propagation. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 367–368. Springer, Heidelberg (2011)
7. Gomes, C.P., Selman, B., Crato, N.: Heavy-Tailed Distributions in Combinatorial Search. In: Smolka, G. (ed.) CP 1997. LNCS, vol. 1330, pp. 121–135. Springer, Heidelberg (1997)
8. Hirsch, E.A., Kojevnikov, A.: Unitwalk: A new sat solver that uses local search guided by unit clause elimination. Ann. Math. Artif. Intell. 43(1), 91–111 (2005)
9. Huang, J.: A Case for Simple SAT Solvers. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 839–846. Springer, Heidelberg (2007)
10. Katebi, H., Sakallah, K.A., Marques-Silva, J.P.: Empirical Study of the Anatomy of Modern Sat Solvers. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 343–356. Springer, Heidelberg (2011)
11. Lonsing, F., Biere, A.: Depqbf: A dependency-aware qbf solver. JSAT 7(2-3), 71–76 (2010)
12. Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In: 10th International Conference on Theory and Applications of Satisfiability Testing, pp. 294–299 (2007)
13. Pipatsrisawat, K., Darwiche, A.: On the Power of Clause-Learning SAT Solvers with Restarts. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 654–668. Springer, Heidelberg (2009)
14. Ramos, A., van der Tak, P., Heule, M.J.H.: Between Restarts and Backjumps. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 216–229. Springer, Heidelberg (2011)
15. Selman, B., Kautz, H., Cohen, B.: Local Search Strategies for Satisfiability Testing. In: Tamassia, R., Tollis, I.G. (eds.) GD 1994. LNCS, vol. 894, pp. 521–532. Springer, Heidelberg (1995)
16. Tompkins, D.A.D., Balint, A., Hoos, H.H.: Captain Jack: New Variable Selection Heuristics in Local Search for SAT. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 302–316. Springer, Heidelberg (2011)