Notes on Non-Chronologic Backtracking, Implication Graphs, and Learning

Alan J. Hu
for CpSc 513
Univ. of British Columbia

2004 February 19

These are supplementary notes on these aspects of a modern DPLL-style complete SAT solver. I think part of what makes these things hard to understand sometimes is that multiple related ideas get lumped together. In these notes, I'm going to try to distill out the essential ideas, without worrying about how to do everything in the best possible manner. If you're impatient, jump to Section 5

1 Exhaustive Enumeration

OK, forget for the moment everything you know about SAT, other than the basic problem: you've got a bunch of clauses, and you're trying to find a satisfying assignment, or prove that no satisfying assignment exists. The obvious approach is to systematically generate every possible assigment, and test whether that assignment satisfies the clauses. The approach has to be systematic if we want to be able to prove unsatisfiability – you have to guarantee that you'll try every possibility eventually. (If you don't care about that, you can try random truth assignments, which puts us in the realm of stochastic local search, which is Holger's expertise.)

The easiest way to systematically try all truth assignments is to march through them in numerical (binary) order, the way you'd do a truth table. I'm going to show you something slightly messier, because that leads to the next step.

Assume you have some data structure for holding a (possibly partial) truth assignment to the variables, and another data structure for holding the clauses. Here's some pseudo-C for an implementation. You'd call this function initially with the empty (all variables unassigned) truth assignment:

```
int sat(truth_assignment t, clauses c)
{
  variable v;
```

```
  v = pick_unassigned_variable(t);
  if (v==NULL) {
    /* all variables assigned, check SAT */
    if (evaluate(t,c)==1) return SAT;
    else return UNSAT;
  }

  assign_false(v,t); /* Assign v to 0 in t */
  if (sat(t,c)==SAT) return SAT;

  /* Not satisfiable when v=0, try v=1. */
  unassign(v,t);
  assign_true(v,t);
  if (sat(t,c)==SAT) return SAT;

  /* Not satisfiable regardless of what we assign to v. */
  /* Therefore, this (sub-)problem isn't satisfiable. */
  unassign(v,t);
  return UNSAT;
}
```

In the above code, if `pick_unassigned_variables` always picks variables in the same order, then you'll search things in the same order as a truth table would. However, the code works fine as long as it always picks an unassigned variable. (The different parts of the truth table could have its rows ordered differently.)

The other thing to note about the code is that we are exploiting the way programs work in order to simplify our implementation. In particular, we are using the recursive call stack to keep track of our partial assignments, and the sequential code in the function to remember whether we've tried 0, 1, or both possibilities for assigning a given variable.

2 Backtracking

The above code is actually already doing backtracking. It's just that we're walking all the way to the bottom of the search tree (assigning a value to every variable) before we backtrack. With SAT, as soon as we find that any clause is 0 (with a partial truth assignment), we know that it's useless to continue assigning variables, so we may as well backtrack immediately at that point. This is a big efficiency improvement. To do this, let's assume we have a function `partial_evaluate` that returns SAT if the (possibly partial) assignment already satisfies the clauses, UNSAT if the assignment already falsifies the clauses, and UNKNOWN otherwise.

```
int sat(truth_assignment t, clauses c)
{
  variable v;
```

```
  int temp;

  temp = partial_evaluate(t,c);
  if (temp==SAT) return SAT;
  if (temp==UNSAT) return UNSAT;

  v = pick_unassigned_variable(t);
  /* Must be unassigned variables, otherwise partial_evaluate
     would have returned SAT or UNSAT. */
  assert (v!=NULL);

  assign_false(v,t); /* Assign v to 0 in t */
  if (sat(t,c)==SAT) return SAT;

  /* Not satisfiable when v=0, try v=1. */
  unassign(v,t);
  assign_true(v,t);
  if (sat(t,c)==SAT) return SAT;

  /* Not satisfiable regardless of what we assign to v. */
  /* Therefore, this (sub-)problem isn't satisfiable. */
  unassign(v,t);
  return UNSAT;
}
```

As an implementation note, one could also write the `partial_evaluate` function to return the clauses that result from plugging in the truth assignment. This clause list is what would be passed to the recursive calls to `sat`. Implementing that way means the partial evaluation work gets ammortized over the various partial assignments. The downside is that you'd need to either save the old clause list on a stack, so that you can restore your previous clause list, or have code to undo the effect of the partial evaluation.

This is a natural and easy way to write a SAT solver, and is pretty much what everyone did for the past 30 years or so. The main question would be how to pick a "good" unassigned variable, and various tricks to know when you can avoid making one recursive call or the other. Practical, complete SAT-solving largely stagnated.

3 Non-Chronologic Backtracking — Inspiration

One of the big breakthroughs was non-chronologic (aka conflict-directed) backtracking. This is normally explained along with the unit clause rule, but let's ignore that for now. Doing so makes it much easier to see the basic idea.

Suppose you ran our backtracking solver on the following problem:

$$(a+z)(b+c+d+\cdots+x+y)(\bar{z})$$

and suppose that the `pick_unassigned_variable` function happens to pick the variables in alphabetical order.

You'll see that the code first tries $a=0$, then it will spend a long time trying all 2^{24} possible assignments to $b,\ldots,y$, with each of these searches eventually failing because $a=0$ implies $z=1$ from the first clause, but the last clause says $z=0$.

You can imagine many clever tricks for avoiding this problem (and that's what people did, and some of these tricks are generally useful). However, what people didn't see for a long time, probably because the easy programming structure of the recursive backtracking code blinded them to this possibility, is that one way to avoid this problem is to "backtrack" in a weird way. Intuitively, we get stuck trying to find an assignment to z, and the only relevant variables are a and z. All the intermediate assignments to $b,\ldots,y$ were irrelevant. So rather than backtrack "chronologically" (backing up to the most recent untried possibility in the recursive call stack), we should backtrack "non-chronologically" or "conflict-driven" by backing up to the most recent **relevant** untried possibility. So, we'd try assigning $a=0$, and then $b=0$, etc., down to $y=0$, where we'd backtrack and try $y=1$ (because of the big clause), and then $z=0$, which fails, causing us to try $z=1$. When that fails, too, we'd like to figure out somehow that the assignments to $b,\ldots,y$ aren't relevant, and backtrack all the way back to trying $a=1$.

How do we determine what's relevant? In general, this turns out to be tricky. If you believe that, jump to the next section. If you want to see why this is hard, consider:

$$(a+\bar{y})(b+z)(c+d+\cdots+x)(y+\bar{z})(\bar{a})$$

Again, assuming that we pick the variables in alphabetical order, we'd try $a=0$, $b=0$, and so on, down to $x=0$, oops, backtrack, $x=1$. At that point, we'd try $y=0$, and then $z=0$, which fails the second clause $(b+z)$, so we'd backtrack and try $z=1$, which fails the last clause $(y+\bar{z})$, so we backtrack and try $y=1$, which fails the first clause $(a+\bar{y})$. So, how far back to we backtrack now? If we look at the clause that failed, $(a+\bar{y})$, it's not obvious that the most recent, **relevant** untried possibility is actually to try $b=1$. What we're seeing is that, in general, what variable assignments were relevant is not local to the clause that is failing at a given point in the search; instead, it depends on the history of the backtracking. In this example, b is relevant because $b=0$ forced $z=1$, which eliminated the possible solution for $y=0$. This sort of analysis is expensive in general, and the research community assumes it's therefore not feasible.

4 Conflict and the Unit Clause Rule

It turns out there is a special case in which it's easy to tell what the relevant variable assignments were. If after making an assignment, the resulting clauses insist that a given variable be both true and false at the same time, this is called "conflict". For example, when we looked at:

$$(a+z)(b+c+d+\cdots+x+y)(\bar{z})$$

as soon as we assigned $a = 0$, then the $(a+z)$ clause would insist that $z = 1$, but the $(\bar{z})$ clause insists that $z = 0$, so we have a conflict. In some sense, a conflict is a one-level lookahead in our backtracking, in which we see immediately that both assignments to z are already guaranteed to fail, given our assignments so far. Therefore, **upon seeing a conflict, we can look at only the clauses involved in the conflict, and backtrack to the most recent untried decision in those clauses**. So now, in our example, as soon as we assigned $a = 0$, we'd detect a conflict and immediately backtrack to the most recent untried possibility, namely $a = 1$.

In the messier example:

$$(a+\bar{y})(b+z)(c+d+\cdots+x)(y+\bar{z})(\bar{a})$$

we'd assign $a = 0$, then $b = 0$, etc. down to $y = 0$, which produces a conflict on z, so then we try $y = 1$, which fails (but doesn't produce a conflict), so we don't erroneously backtrack all the way back to a, and we inefficiently backtrack only up to trying $w = 1$, etc.

Hmm... we've seen that our conflict analysis is basically looking ahead slightly for cases where a variable is forced to be true and false at the same time. When is a variable forced to take a value? When the partial assignment causes all the other literals in a clause to be false, so the last literal must be true. For example, in the clause $(c+d+\cdots+x)$, after trying $c = d = \cdots = w = 0$, the clause has only the unassigned literal x, so we are forced to have $x = 1$. A clause in which only one literal is left unassigned is called a "unit clause", and the "unit clause rule" says to immediately make the satisfying assignment to that unassigned literal, since it's forced. The unit clause rule is in some sense a generalization of the lookahead we're using to define conflicts.

Armed with the unit clause rule, both of our examples are solved instantly. For the first one:

$$(a+z)(b+c+d+\cdots+x+y)(\bar{z})$$

The last clause is unit, immediately forcing $z = 0$, which makes the first clause $(a+z)$ unit, forcing $a = 1$. Then, we proceed to pick assignments for b through x, which then triggers the unit clause rule again to make $y = 1$. For the messier example:

$$(a+\bar{y})(b+z)(c+d+\cdots+x)(y+\bar{z})(\bar{a})$$

the last clause is unit, forcing $a = 0$, which makes the first clause unit, forcing $y = 0$, which forces $z = 0$, which forces $b = 1$. Then, we pick an assignment for c through x.

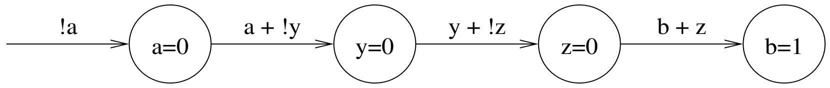
5 Implication Graphs

As we just saw, the unit clause rule can cause a cascade of forced variable assignments. In order to be able to backtrack non-chronologically, we need some way to keep track of and "see through" these forced assignments. The data structure invented to do this is called an "implication graph".

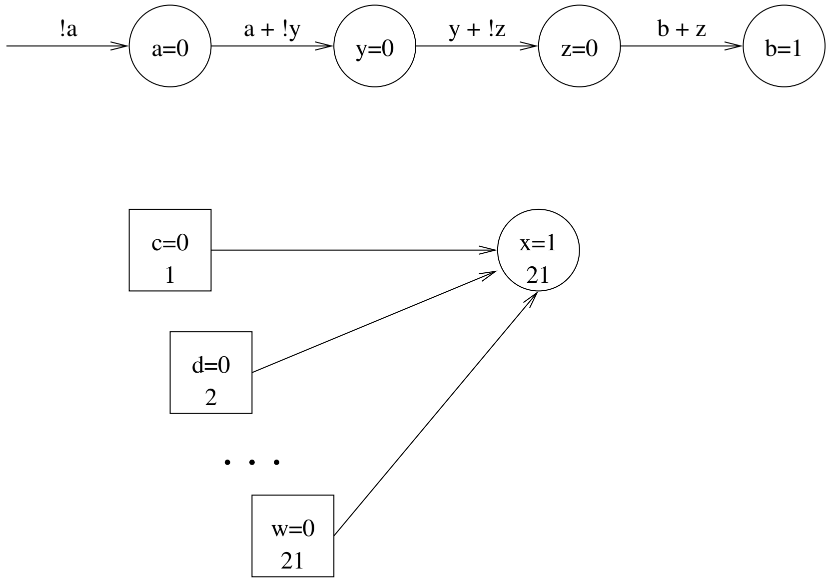
An implication graph is a DAG (directed, acyclic graph). Vertices are labeled with an assignment to a variable. There are two kinds of vertices: decision vertices, which indicate a decision the backtracking search has decided to try, and deduced or implied vertices, whose value is forced by the unit clause rule. An edge leads from one vertex v_1 to another vertex v_2 if the assignment at v_1 was (part of) what became the unit clause that forced v_2. For example, returning to our messier formula:

$$(a+\bar{y})(b+z)(c+d+\cdots+x)(y+\bar{z})(\bar{a})$$

the unit clauses immediately generate a bunch of implied vertices:



As we start the backtracking search, we then start generating decision vertices, which I will draw with squares. Note that decision vertices have no incoming edges, since they were decisions of the search procedure, not implications of other decisions:



The assignment of $x = 1$ is another implied vertex. I didn't bother labeling all the arrows with the big $(c+d+\cdots+x)$ clause that forced that assignment. Note that I've put little numbers on the nodes. These numbers are "decision levels", which indicate how deep in the backtracking we are. We will use these to tell how far back to backtrack non-chronologically, and what parts of the graph to erase when we backtrack.

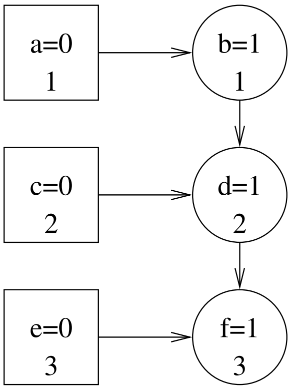
6 Adding Learning

We need one more wrinkle to make everything work right. This wrinkle is called "learning", and it will magically make non-chronologic backtracking happen. Note that our above example has been solved without requiring a backtrack, so we'll need a more complex example. Consider:

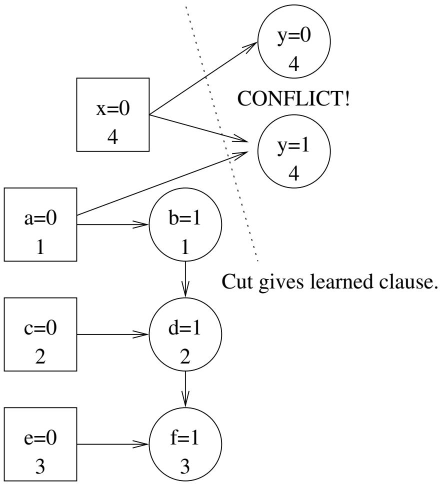
$$(a+x+y)(a+b)(\bar{b}+c+d)(\bar{d}+e+f)(\bar{x}+\bar{y})(x+\bar{y})(\bar{x}+y).$$

Again, assume we'll pick the variables in alphabetical order. We'll therefore start with trying $a = 0$, which implies $b = 1$; then, we'll try $c = 0$, which forces $d = 1$; then, we'll try $e = 0$, which forces

$f = 1$, yielding the implication graph:



The next decision is $x = 0$, which immediately produces conflict:



The key to learning is to note that the graph shows us that certain variable assignments led inexorably to the conflict. If you trace backwards from the conflict in the implication graph, all decision variables that are ancestors of the conflict are what caused the problem. In our example, the problem is the assignments $a = 0$ and $x = 0$. Note that the decision levels 2 and 3 were completely irrelevant, and the implication graph shows us this. Learning is the process of adding a new clause (a "learned clause") that tells us not to ever have $a = 0$ and $x = 0$ again: we add the clause $(a + x)$. In general, any cut of the implication graph between the decision variables and the conflict is a perfectly good learned clause.

Note that it's not clear in general which cut(s)/clause(s) one should learn from a given conflict. If we cut very close to the decision variables, this is good, because in the future, the learned clause will catch us early and prevent us from going down this path again. However, a learned clause near the decision variables won't prevent us from bypassing the learned clause if we end up choosing decision variables in a different order. Learning too many clauses will slow down the SAT solver, because the set of clauses will grow too big. All in all, this is an open research question.

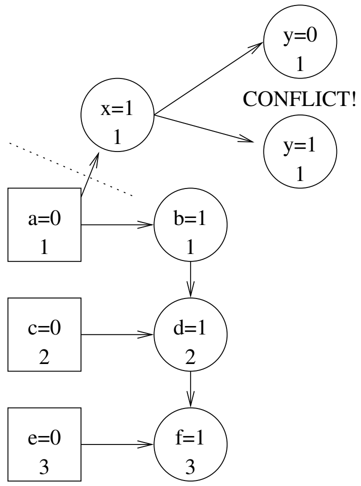
One rule that people agree on, though, is to always learn a clause that will force the search in a different direction, if we were to attempt to choose the variables in the same order. Such a clause is called an "asserting clause". You can guarantee that a clause is asserting by having the cut separate the most recent decision vertex from all of its deduced vertices. Then, you backtrack to the most recent relevant decision vertex before that.

7 Non-Chronologic Backtracking

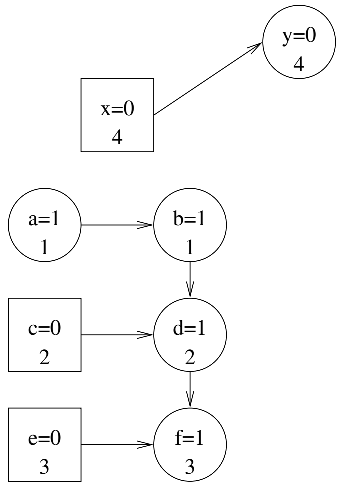
Now, at last, we're ready to look at full non-chronologic backtracking. Returning to our running example, we add the learned asserting clause to our clauses:

$$(a+x+y)(a+b)(\bar{b}+c+d)(\bar{d}+e+f)(\bar{x}+\bar{y})(x+\bar{y})(\bar{x}+y)(a+x),$$

and then we backtrack back to decision level 1. Again, we try $a=0$, but this time, the learned clause becomes unit and forces $x=1$, which produces a conflict: (I've left the decision level 2 and 3 stuff here, out of laziness, but in reality, those nodes had been deleted.)



Now, the asserting clause is just (a). We add that clause, and then backtrack all the way to the beginning, and eventually get:



which is the satisfying assignment.