600.432/632 Declarative Methods - Prof. Jason Eisner
Answers to "SAT Solvers" discussion problems

1. (a) Stays the same.
The decision level only increases when the backtracking search has to guess the value of a variable (not when the value is forced by unit propagation).

(b) Backtracks.
The problem may still be satisfiable if different decisions are made.

(c) None.
Recall that zChaff does not bother to "cross out" these clauses that are now satisfied. They do not need to be visited at all because they cannot cause any unit propagation.

(d) Some.
zChaff only visits clauses that are watching ~X (which has just become false). These are the clauses that may have become unit clauses (if we regard all false literals as having been crossed out), leading to unit propagation.

(e) Always the same.
The only difference between deciding X = 1 and setting it by unit propagation is during backtracking.

(f) At least 1, and at most k-1.
(We also gave credit for an answer of k, because of the confusing "Note" in the problem. The "Note" was trying to deal with the case where the clause contained multiple copies of X or ~X.)

Remember that the 2 watched literals are always non-F, i.e., either T or ?. We are visiting the clause because this invariant property has just been broken: its watched literal ~X is changing status from ? to F. (*)

Here's the pseudocode where W1, W2 indicate the two watched literals. wlog, assume W1 turned false

```
for each literal L among the k-2 unwatched literals
    if L in [T, ?]
        swap L with ~W1 and return

# if we got here, all k-2 unwatched literals must be F
if W2 = ?
    W2 = T # unit propagation
```

In the worst case, this looks at all k-1 literals other than ~W1.
In the best case, this looks at only 1 literal:
- If k >= 3, it could return during the first loop iteration.
- If k = 2, there are 0 loop iterations but it still does have to look at the other watched literal.)

(*) We may not be able to fix the invariant property: if we reach the end of the above code without returning early, then we'll have T F F F F or F T F F F (where the T either was changed from ? by unit propagation above, or else was already T by a previous assignment).

Now there don't exist two non-F literals to watch -- one of the watched literals is in fact F. So the invariant property

actually only holds on clauses that don't have the above form.
Or to put it another way, the REAL invariant property is this:
"Both watched literals are non-F, provided that there are at
least two non-F literals in the clause. If not, then one watched
literal is non-F."

2. The steps are:

[1] Clause 4 is true -> cross it off.
[2] Cross off D from clause 3 -> unit clause -> B = 1.
[3] Clause 1 is true -> cross it off.
[4] Cross off ~B from clause 2.

Pictorially:

```
A v B v C            A v B v C
A v ~B v ~C   ---->  A v ~B v ~C  ---->  A v ~ C
B v D                B                   B
C v ~D
```

3. The heuristic says to pick the variable that crosses off
the most clauses -- except that when it is counting clauses,
it gives short clauses more weight, since they are harder to
cross off.

B=1 or C=1 is the correct answer.
Either of these satisfies a clause of length 2 and a clause of length 3.

A=1 satisfies two clauses of length 3, which is not as good.

4. There's a bug. We have to have C=1 on the stack so we know to
undo it in the assignment table when it's popped.

(That is, this is not really tail recursion, since we still have to
do something after we return from the recursion -- namely cleaning
up the assignment.)

Some of you suggested other problems that were not really problems.
Some of you said that we'd lose the assignment information (but we
wouldn't, since it is traditionally kept in a separate table).
Others said that we'd forget which variables we still had to branch
on (but the whole point of the question was that the stack would
still be used to remember exactly those variables).

5.

(a) Each constraint is a disjunction of literals, such as (A v ~B v D)

(b)

Option A:

```
(X -> Y) ^ (Y -> X)
(~X v Y) ^ (~Y v X)
```

Option B:

```
(X ^ Y) v (~X ^ ~Y)
(X v ~X) ^ (X v ~Y) ^ (Y v ~X) v (Y v ~Y)
(X v ~Y) ^ (Y v ~X)
```

(c)

i. Unit propagation (also known as boolean constraint propagation)

ii.

Option A:

"Most constrained" would give a unit clause high priority.

Option B:

"Most satisfying" might help as well, if shorter clauses are given more weight.

iii. Suppose X=0 is assigned and variable ordering has decided to assign Y next. If the value ordering method picks Y=1, then we'll detect a contradiction immediately (thanks to the clause (X v ~Y)), backtrack, and proceed immediately with Y=0.

iv. It's not guaranteed that GSAT will assign Y=0. GSAT flips the variable which will satisfy the most constraints, and we only know that setting Y=0 will solve one constraint in this example.

6. a. All the constraints on variable A, i.e., the first 3 constraints.
b. (B v D) ^ (C v D) ^ (X v Y).

Here's an explanation:
Following the general rules for variable elimination,
we fuse the above three constraints to get
(A v B) ^ (A v C) ^ (~A v D)
which has this truth table,

A	B	C	D	(AvB)^(AvC)^(~AvD)
0	0	0	0	0
1	0	0	0	0
0	1	0	0	0
1	1	0	0	0
0	0	1	0	0
1	0	1	0	0
0	1	1	0	1
1	1	1	0	0
0	0	0	1	0
1	0	0	1	1
0	1	0	1	0
1	1	0	1	1
0	0	1	1	0
1	0	1	1	1
0	1	1	1	0
1	1	1	1	1

Then we project out A to find only the BCD combos that
can satisfy the constraint for some value of A:

B	C	D	Result
0	0	0	0
1	0	0	0
0	1	0	0
1	1	0	1
0	0	1	1
1	0	1	1
0	1	1	1
1	1	1	1

The latter truth table can be more succinctly expressed in CNF as
(B v D) ^ (C v D)
so our final answer is
(B v D) ^ (C v D) ^ (X v Y).

We don't really have to write out truth tables, though.
As mentioned in class, we can do variable elimination in CNF
by the "resolution" procedure, where each clause containing A
is fused with each clause containing ~A.
Thus, we work directly with the symbols, and turn
(A v B) ^ (A v C) ^ (~A v D)
into
((A v B) ^ (~A v D)) ^ ((A v C) ^ (~A v D))
which is equivalent to
((~A -> B) ^ (A -> D)) ^ ((~A -> C) ^ (A -> D))
which is satisfiable iff the following is satisfiable:

(B v D) ^ (C ^ D)
This is just the reverse of the construction that introduces a switching variable A.

c. Given values of (B, C, D, X, Y) that satisfy Psi, just try extending this assignment with both A=0 and A=1, testing to see whether the resulting assignment satisfies the original formula Phi. It is enough just to check that it satisfies Phi's constraints on A, namely (AvB)^(AvC)^(~AvD).

We are guaranteed that at least one choice of A will work, since our new constraint (BvD)^(CvD) was designed to describe exactly those BCD combinations in the truth table that could be extended with some value of A to satisfy (AvB)^(AvC)^(~AvD).

d. Reduces the number of variables by 1.

e. In general, it may increase the number of clauses, although it didn't in this case.

(If we had fused 99 clauses with A with 99 clauses with ~A, then we would have ended up replacing those 198 clauses with 99*99 = 9801 new clauses. Thus, in the worst case, a SINGLE step of variable elimination may roughly square the number of clauses.)

7.

(a) Pick randomly among the clauses in vector U. This takes O(1) time.

(b) It's just |B_L|. This is stored with the vector B_L and can be looked up in O(1) time.

(c) Roughly speaking, WalkSAT chooses randomly among the variables in C that have the minimum break score. (If this minimum break score is > 0, it has some probability of instead choosing randomly among *all* variables in C.)

(d) U_X and F_X (since X becomes watched)
and W_~X (since ~X is no longer watched, so we have to look for a replacement).

If you are curious about the other updates:

The false literal X becomes true, so:

- visit all clauses in U_X and swap X into watched position.
Then rename U_X to B_X. (We no longer need U_X once X is true.)
Also remove all these clauses from U.
Time: |U_X|*k.

- visit all clauses in F_X and swap X into watched position.
Then rename F_X to W_X. (We no longer need F_X once X is true.)
Time: |F_X|*k.

The true literal ~X becomes false, so:

- For each clause C in B_~X, add C to U
for each literal L in C, add C to U_L.
Now we no longer need B_~X now that ~X is false.
Time: |B_~X|*k.

- For each clause C in W_~X,
if C contains an unwatched true literal L,
swap L with ~X so that L is now watched
add C to W_L
else

```
            let L be the other watched true literal in C
        add C to B_L
```

Now we no longer need W_~X now that ~X is false.
Time: |W_~X|*k.

(e) The union of B_L and W_L, i.e., all watched literals.

(f) REASON 1: zChaff never has to pick a random unsatisfied clause, so it doesn't need to maintain a list U.

REASON 2: in zChaff, false variables are never flipped to true. So we don't need to maintain lists U_L and F_L of unsatisfied or unit clauses to notice when such flips rescue these clauses.

REASON 3: zChaff is solving SAT, not MAX-SAT, so it requires all clauses to be true. It doesn't even consider breaking unit clauses. But in WalkSAT, we need to maintain a list B_L of unit clauses that would be broken by flipping L, so we can compute the break score. Furthermore, by keeping B_L separate from W_L, we can find the length of this list quickly. (We don't really have to keep them separate, though; we could maintain the number |B_L| without keeping the actual clauses separate.)