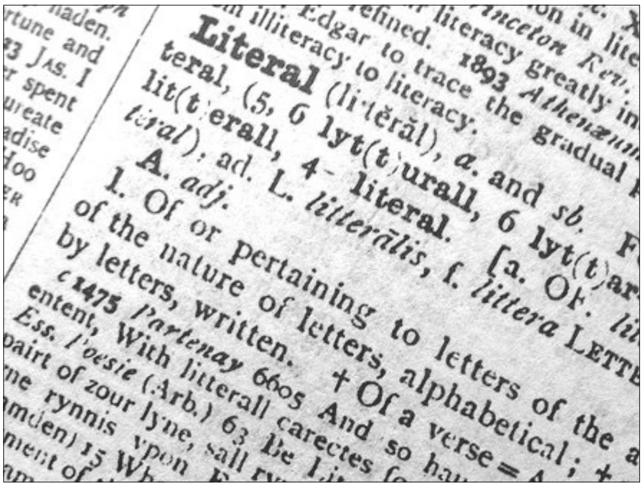
Informatics 1

Michael Fourman

Lecture 10 Searching for Satisfaction

In this lecture we consider the problem of searching for a satisfying valuation for a set of clauses.

We look at three algorithms.



naden.
rtune and
3 JAS. I
r spent
ureate
adise
Hoo
ER

efined. literacy greatly in
nceton Rev. on in lite
1893 Athenæum Edgar to trace the gradual
n illiteracy to literacy.

Literal (li·tĕrăl), *a.* and *sb.* F
teral, (5, 6 **lyt(t)urall,** 6 **lyt(t)ar**
lit(t)erall, 4- **literal.** [a. OF. *li*
teral); ad. L. *litterālis*, f. *littera* LETTE

A. *adj.*

1. Of or pertaining to letters of the a
of the nature of letters, alphabetical; †
by letters, written. †Of a verse = A
c **1475** *Partenay* 6605 And so hau
entent, With litterall carectes fo
Ess. Poesie (Arb.) 63 Be Lit
pairt of zour lyne, sall ry
ne rynnis vpon
mden) 15 Wh
ment of

Naïve search

```
function SAT(Φ,V)
  Φ|V = {}
  ||
    {} ∉ Φ|V
    &&
    let A = chooseLiteral (Φ,V)
    in
       SAT (Φ,V ^ A)
       ||
       SAT (Φ,V ^ ¬A)
```

V is a partial valuation
(a consistent set of literals)
V^A = V∪{A}

Φ is a set of clauses

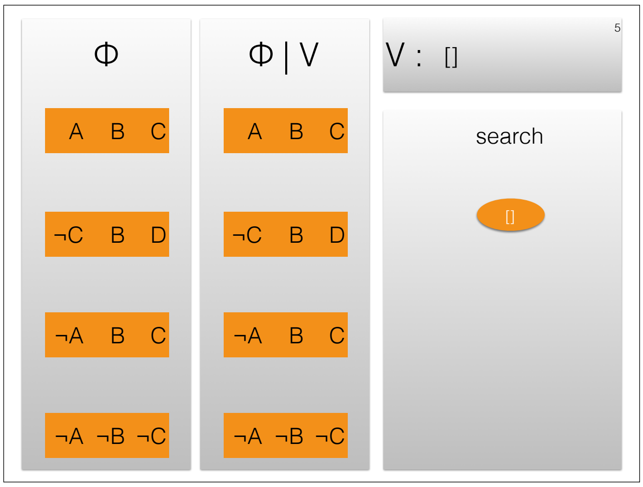
Φ | V is the result of simplifying Φ using V:
For each literal L ∈ V

- remove clauses containing L
- delete ¬L from remaining clauses

chooseLiteral(Φ,V) returns a literal occurring in Φ | V

partial valuations

A partial valuation makes each atom true, false, or unassigned.

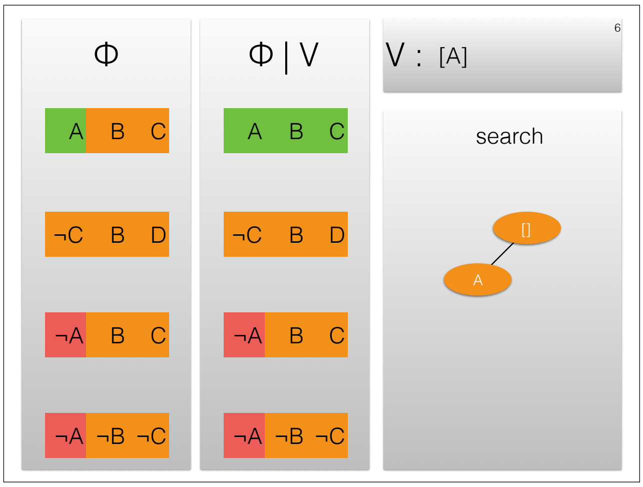


Φ	Φ | V
A B C	A B C
¬C B D	¬C B D
¬A B C	¬A B C
¬A ¬B ¬C	¬A ¬B ¬C

V : []

search

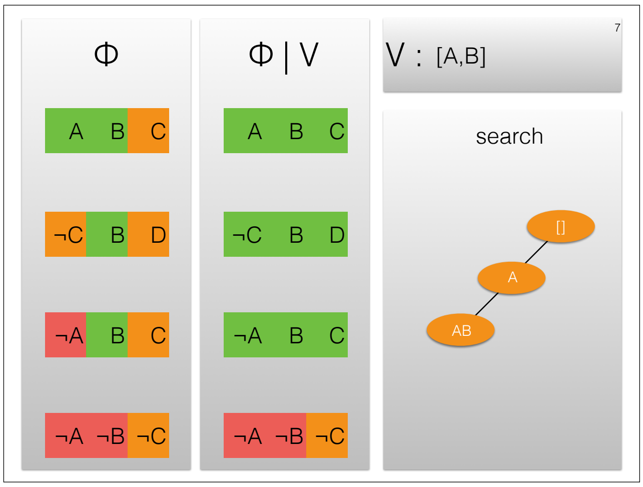
[]



Φ			Φ | V		
A	B	C	A	B	C
¬C	B	D	¬C	B	D
¬A	B	C	¬A	B	C
¬A	¬B	¬C	¬A	¬B	¬C

V : [A]

search



Φ

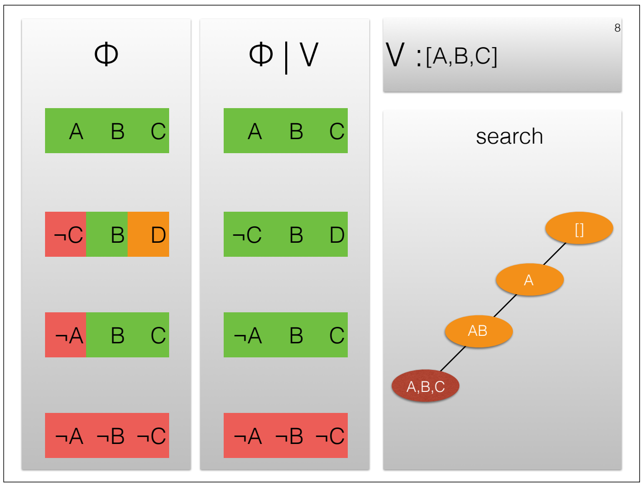
A	B	C
¬C	B	D
¬A	B	C
¬A	¬B	¬C

Φ | V

A	B	C
¬C	B	D
¬A	B	C
¬A	¬B	¬C

V : [A,B]

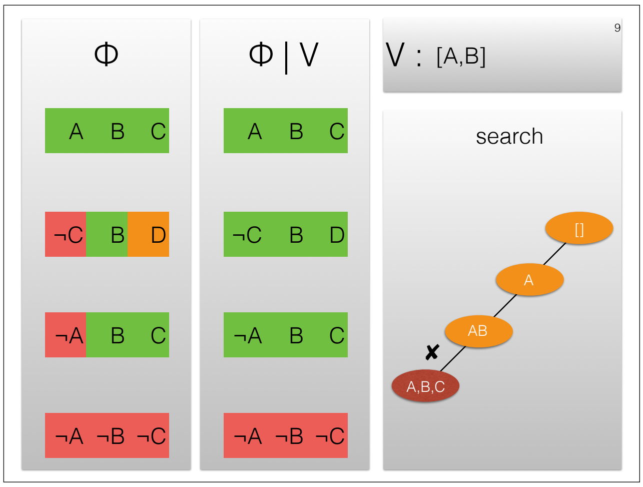
search


8

V : [A,B,C]

Φ	Φ | V
A B C	A B C
¬C B D	¬C B D
¬A B C	¬A B C
¬A ¬B ¬C	¬A ¬B ¬C

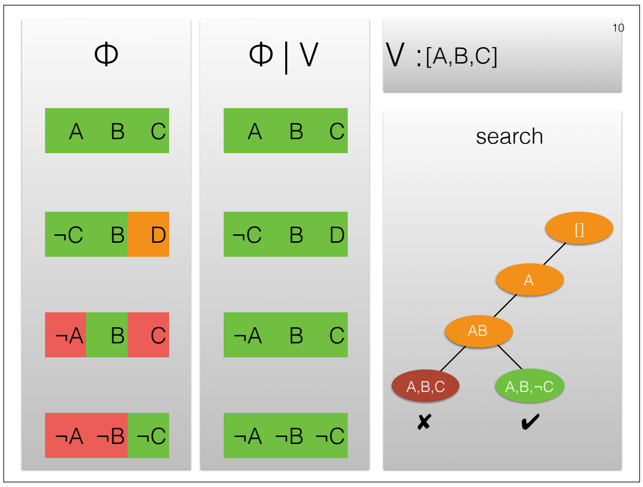
search



Φ	Φ | V
A B C	A B C
¬C B D	¬C B D
¬A B C	¬A B C
¬A ¬B ¬C	¬A ¬B ¬C

V : [A,B]

search

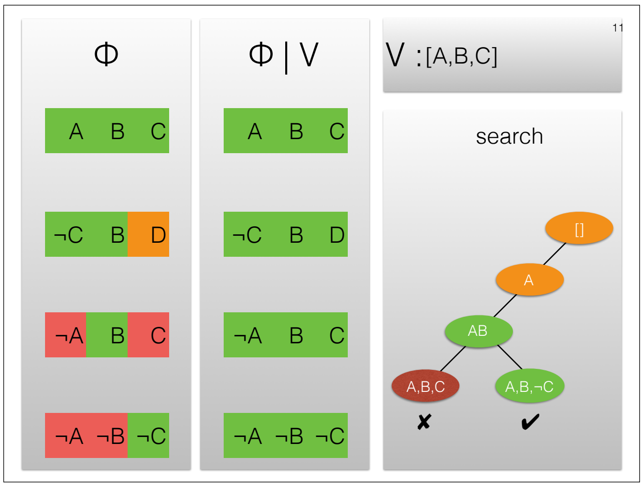


Φ		
A	B	C
¬C	B	D
¬A	B	C
¬A	¬B	¬C

Φ | V		
A	B	C
¬C	B	D
¬A	B	C
¬A	¬B	¬C

V : [A,B,C]

search



Φ

A B C

¬C B D

¬A B C

¬A ¬B ¬C

Φ | V

A B C

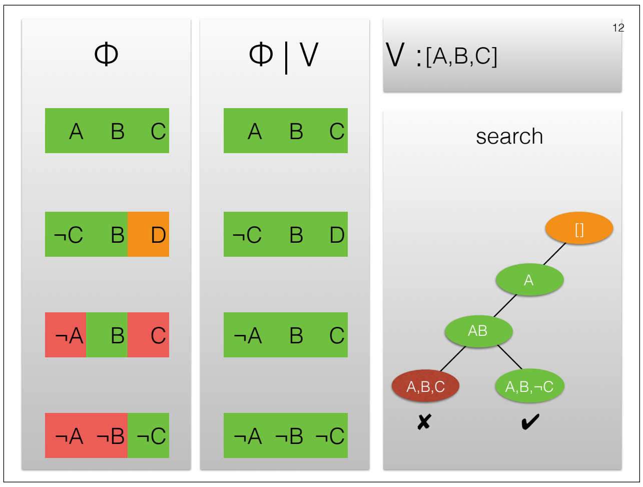
¬C B D

¬A B C

¬A ¬B ¬C

V : [A,B,C]

search

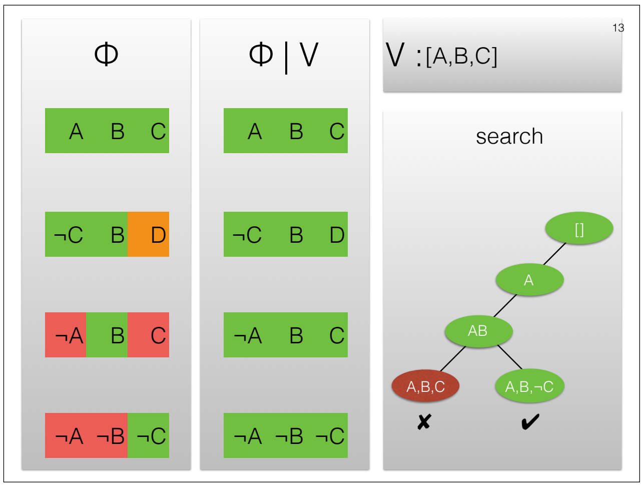


V : [A,B,C]

Φ		
A	B	C
¬C	B	D
¬A	B	C
¬A	¬B	¬C

Φ | V		
A	B	C
¬C	B	D
¬A	B	C
¬A	¬B	¬C

search

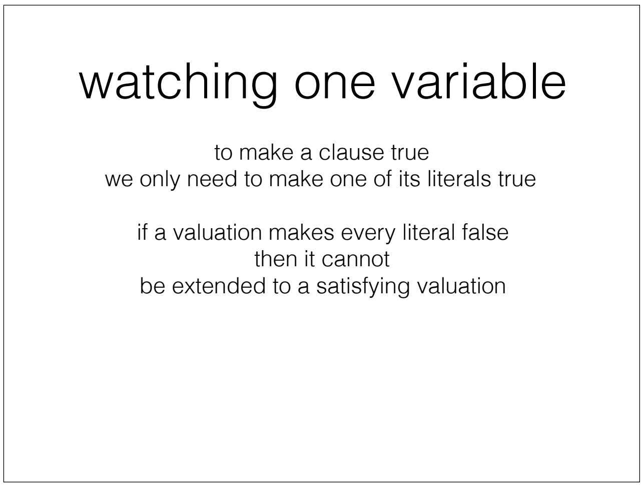


Φ		
A	B	C
¬C	B	D
¬A	B	C
¬A	¬B	¬C

Φ | V		
A	B	C
¬C	B	D
¬A	B	C
¬A	¬B	¬C

V : [A,B,C]

search



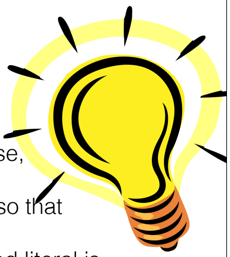
watching one variable

to make a clause true
we only need to make one of its literals true

if a valuation makes every literal false
then it cannot
be extended to a satisfying valuation

watching one literal

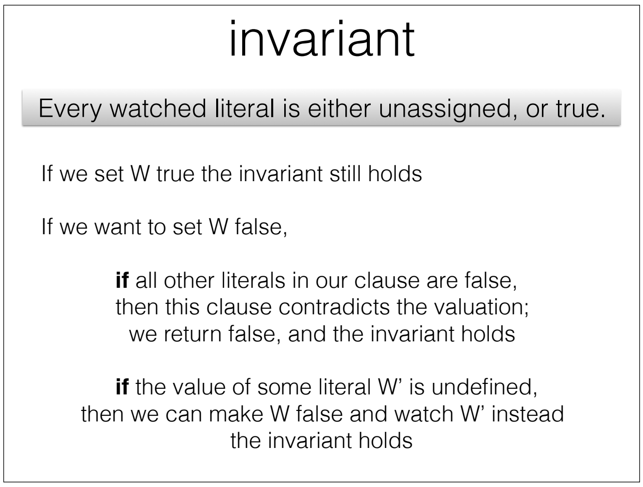
to make a clause true
we can make any one of its literals true
to make it false we must make
every last one false



IDEA we watch one literal in each clause,
as we search for a satisfying valuation,
we change the literal we are watching so that

the valuation of each watched literal is
either undefined or true

if we reach a stage where all watched literals are true
we have a satisfying valuation



invariant

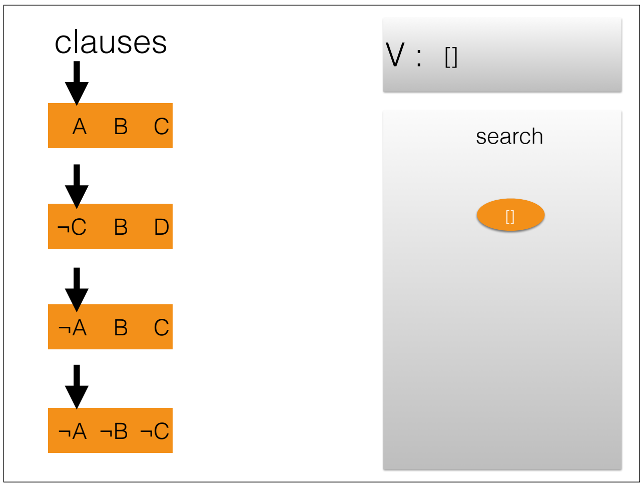
Every watched literal is either unassigned, or true.

If we set W true the invariant still holds

If we want to set W false,

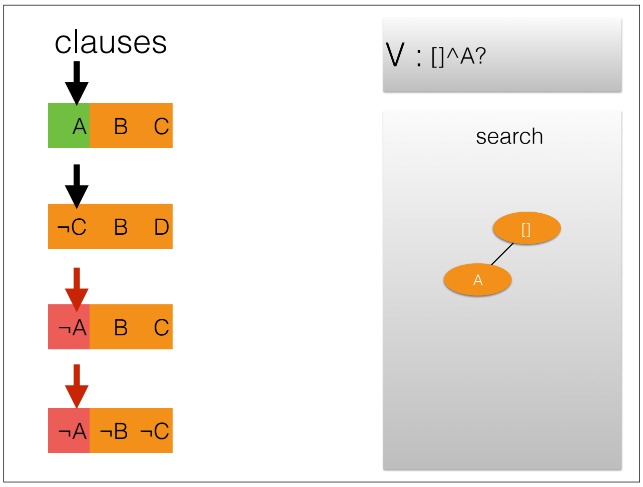
if all other literals in our clause are false,
then this clause contradicts the valuation;
we return false, and the invariant holds

if the value of some literal W’ is undefined,
then we can make W false and watch W’ instead
the invariant holds



The invariant is that **every watched literal is either true or unassigned**

If we want to set a watched literal false we first have to check that we can move the pointers so we can still satisfy the invariant



clauses

A B C

¬C B D

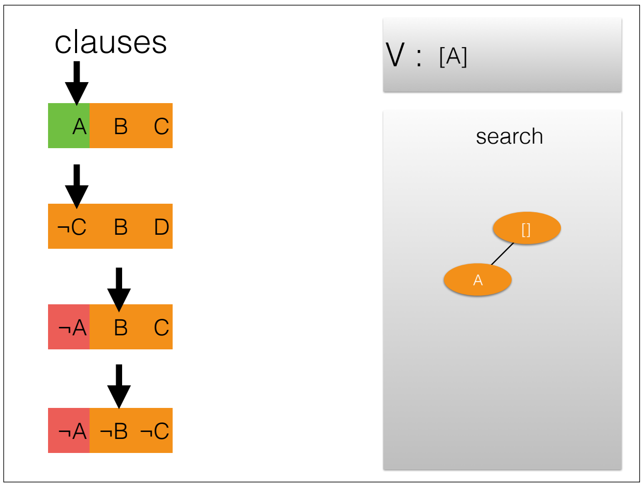
¬A B C

¬A ¬B ¬C

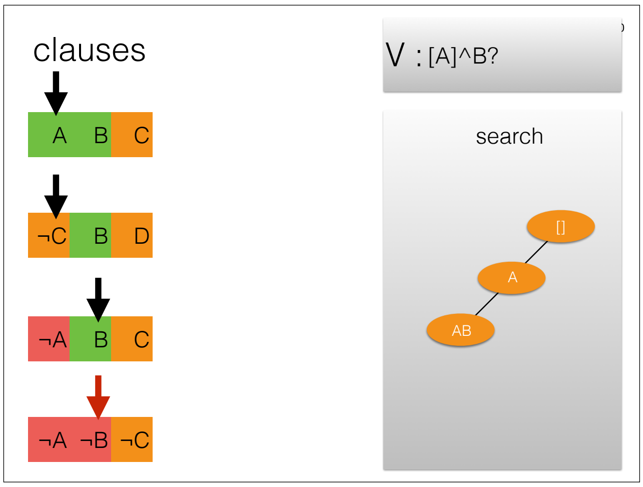
V : []^A?

search

If we want to set A true we will have to move two pointers



This is easily done



clauses

A B C

¬C B D

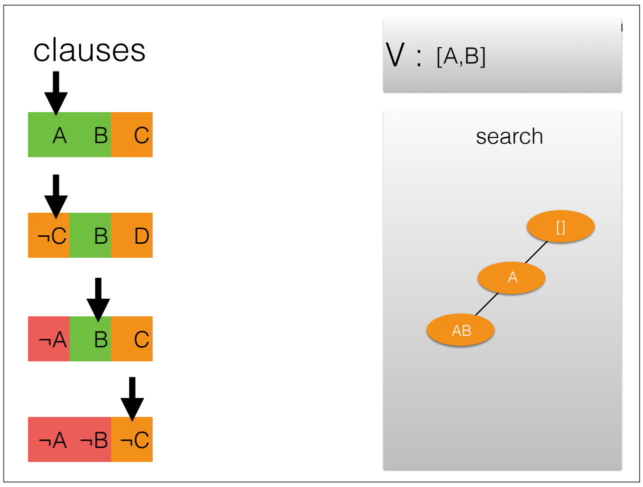
¬A B C

¬A ¬B ¬C

V : [A]^B?

search

If we want to set B true there is one pointer to move



clauses

A B C

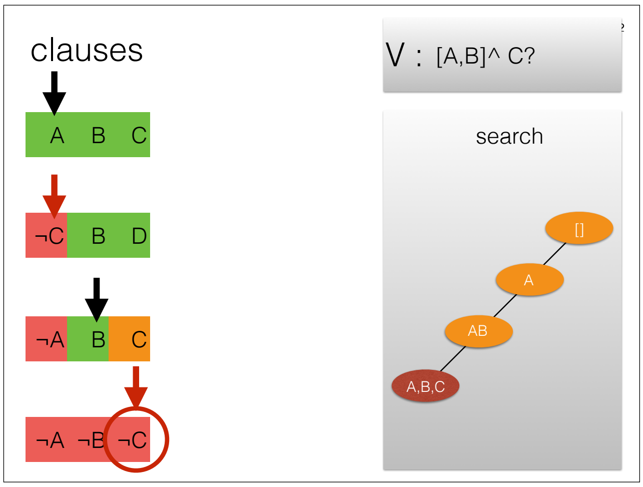
¬C B D

¬A B C

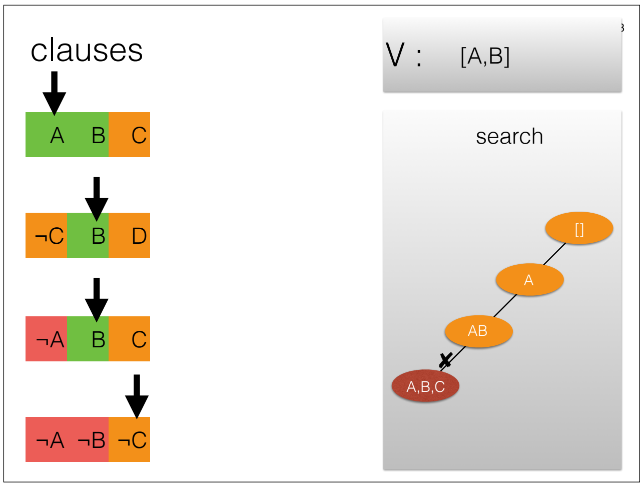
¬A ¬B ¬C

V : [A,B]

search



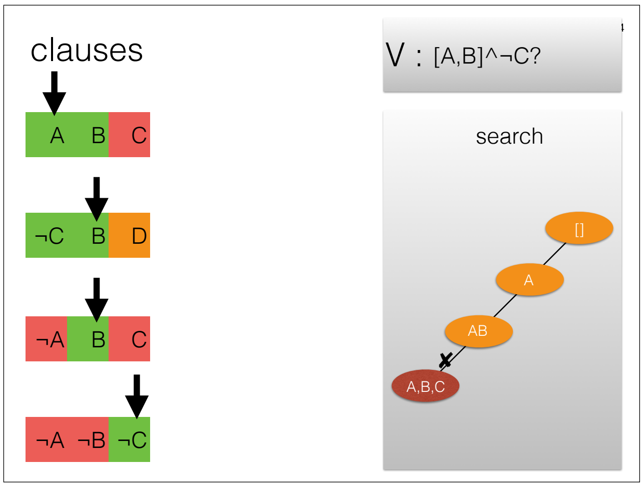
To make C true we would have to move two pointers
– but for one of them there is nowhere left to go!



To make C true we would have to move two pointers
– but for one of them there is nowhere left to go!

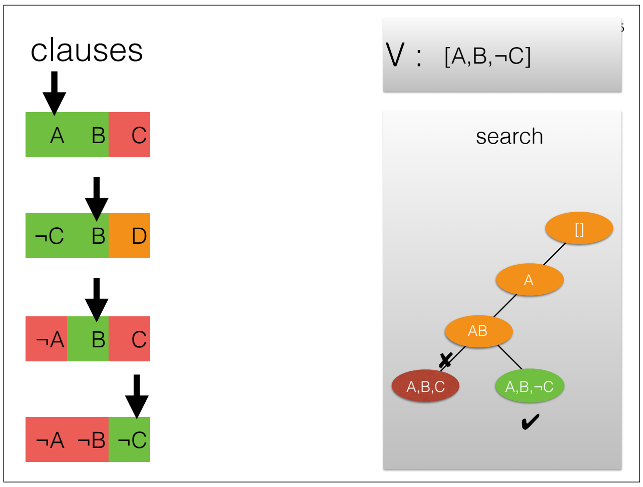
This branch of the search has failed, so we must backtrack and search elsewhere.

Note that it doesn't matter whether or not we move the first pointer, so long as the invariant still holds. In this case, we have moved it.



Note that it doesn't matter whether or not we move the first pointer, so long as the invariant still holds. In this case, we have moved it.

Now we try making C false. The invariant still holds - we don't need to move any pointers.



Note that it doesn't matter whether or not we move the first pointer, so long as the invariant still holds. In this case, we have moved it.

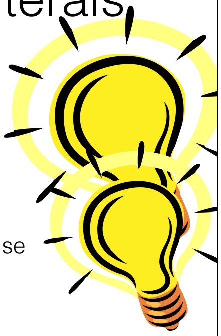
We can see that every clause is satisfied, because every watched literal is true.
You should check that, if we had been watching D instead of B in the second clause, we still have found a satisfying valuation at the next step.

Boolean Constraint Propagation
BCP

if $\Phi \mid V$ contains a unit clause {X}
– that is, a clause with only one literal –
add that literal to V and simplify
$\Phi \mid V$ ^ X

if $\Phi \mid V$ ^ X is inconsistent, so was $\Phi \mid V$
every satisfying valuation for Φ extending V
must make X true

watching two literals



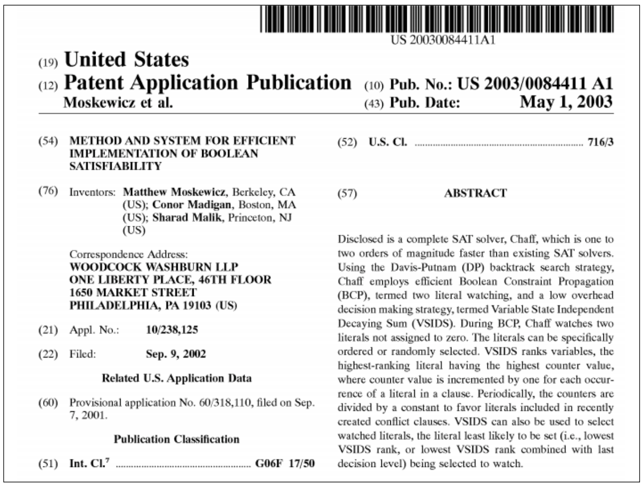
to make a clause true
we can make any one of its literals true

to make it false we must make
every last one false

when we have made the last-but-one false
we have a unit clause

IDEA we watch two literals in each clause

https://docs.google.com/viewer?url=patentimages.storage.googleapis.com/pdfs/US20030084411.pdf



US 20030084411A1

(19) **United States**

(12) **Patent Application Publication** (10) **Pub. No.: US 2003/0084411 A1**

Moskewicz et al. (43) **Pub. Date: May 1, 2003**

(54) **METHOD AND SYSTEM FOR EFFICIENT IMPLEMENTATION OF BOOLEAN SATISFIABILITY**

(76) Inventors: **Matthew Moskewicz**, Berkeley, CA (US); **Conor Madigan**, Boston, MA (US); **Sharad Malik**, Princeton, NJ (US)

Correspondence Address:
WOODCOCK WASHBURN LLP
ONE LIBERTY PLACE, 46TH FLOOR
1650 MARKET STREET
PHILADELPHIA, PA 19103 (US)

(21) Appl. No.: **10/238,125**

(22) Filed: **Sep. 9, 2002**

Related U.S. Application Data

(60) Provisional application No. 60/318,110, filed on Sep. 7, 2001.

Publication Classification

(51) **Int. Cl.^{7}** **G06F 17/50**

(52) **U.S. Cl.** **716/3**

(57) **ABSTRACT**

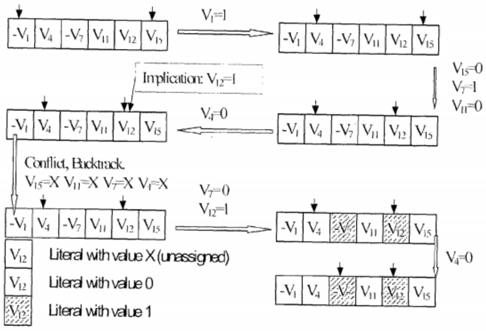
Disclosed is a complete SAT solver, Chaff, which is one to two orders of magnitude faster than existing SAT solvers. Using the Davis-Putnam (DP) backtrack search strategy, Chaff employs efficient Boolean Constraint Propagation (BCP), termed two literal watching, and a low overhead decision making strategy, termed Variable State Independent Decaying Sum (VSIDS). During BCP, Chaff watches two literals not assigned to zero. The literals can be specifically ordered or randomly selected. VSIDS ranks variables, the highest-ranking literal having the highest counter value, where counter value is incremented by one for each occurrence of a literal in a clause. Periodically, the counters are divided by a constant to favor literals included in recently created conflict clauses. VSIDS can also be used to select watched literals, the literal least likely to be set (i.e., lowest VSIDS rank, or lowest VSIDS rank combined with last decision level) being selected to watch.

(60) Provisional application No. 60/318,110, filed on Sep. 7, 2001.

Publication Classification

(51) **Int. Cl.**^{7} **G06F 17/50**

rence of a literal in a clause. Periodically, the counters are divided by a constant to favor literals included in recently created conflict clauses. VSIDS can also be used to select watched literals, the literal least likely to be set (i.e., lowest VSIDS rank, or lowest VSIDS rank combined with last decision level) being selected to watch.



“BCP of the present invention identifies implied clauses and the associated implications while maintaining certain invariants, namely that each clause has two watched literals and that if a clause can become newly implied via any sequence of assignments, then the sequence will include an assignment of one of the watched literals to Zero.”

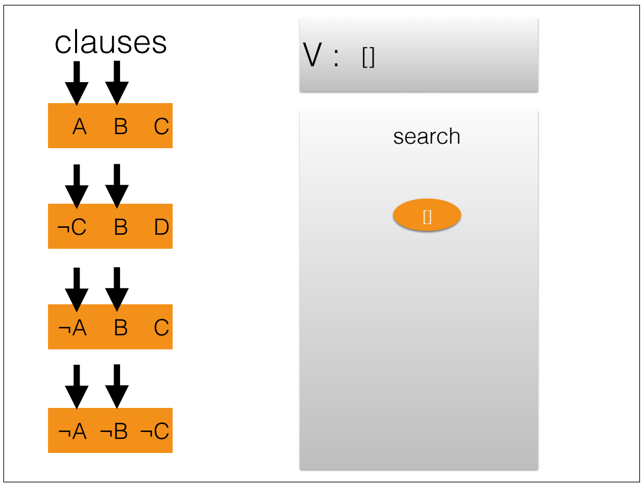
invariant

Every watched literal is either unassigned, or true. 1

At most one watched literal is false, and if one is false then the other is true. 2

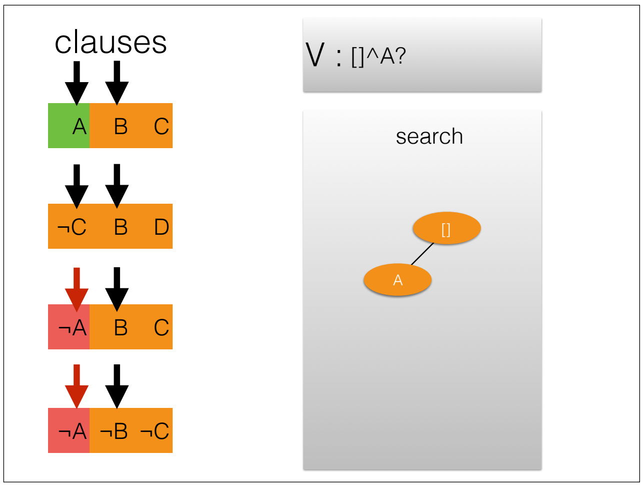
The first invariant implies the second, so we can normally move pointers to maintain the first invariant, just as before.

If we want to make one of the watched literals false, and we have no room to move so that 1 holds, then we must make the other literal true, so that 2 holds, or this branch of the search fails.

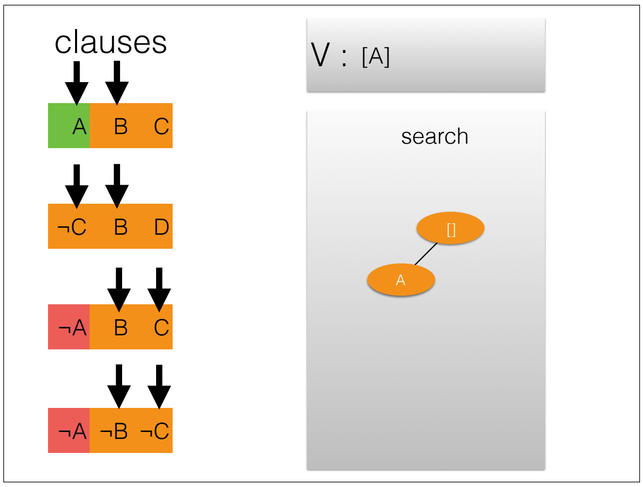


The invariant is that **every watched literal is either true or unassigned**

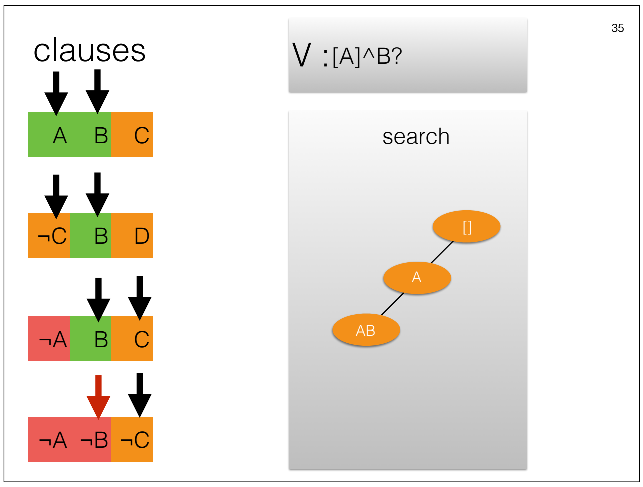
If we want to set a watched literal false we first have to check that we can move the pointers so we can still satisfy the invariant



If we want to set A true we will have to move two pointers



This is easily done



35

clauses

A B C

¬C B D

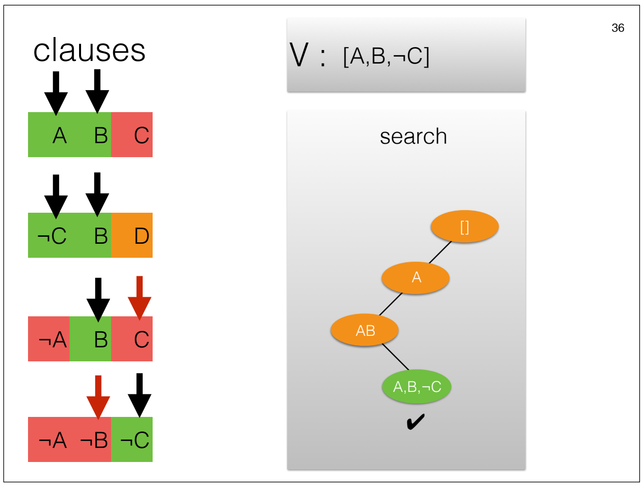
¬A B C

¬A ¬B ¬C

V :[A]^B?

search

If we want to set B true there is one pointer to move, but there is nowhere to go.



clauses

A B C

¬C B D

¬A B C

¬A ¬B ¬C

V : [A,B,¬C]

search

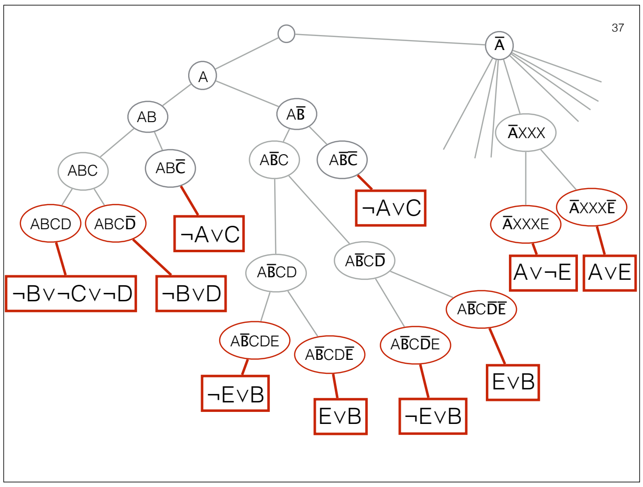
If we want to set B true there is one pointer to move, but there is nowhere to go.

If we make B true we must make C false – so we try this.

If making C false fails then making B true fails;

we would then backtrack to make them both unassigned, and maintain our invariant.

In this case we can already see that all clauses are satisfied as at least one of the watched literals in each clause is true.



For next week's tutorial, you will try both watched literal methods for this example.