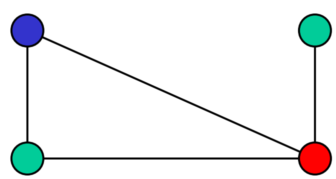
Heuristics for
Efficient SAT Solving

As implemented in GRASP, Chaff and GSAT.

Formulation of famous problems as SAT: *k*-Coloring (1/2)

The *K*-Coloring problem:
Given an undirected graph *G(V,E)* and a natural number *k*, is there an assignment *color*:

$$V \rightarrow \{1, \ldots, k\} \text{ s.t.}$$
$$\forall i, j, (i, j) \in E, color(i) \neq color(j)$$



Formulation of famous problems as SAT: *k*-Coloring (2/2)

$x_{i,j}$ = node i is assigned the 'color' j ($1 \leq i \leq n$, $1 \leq j \leq k$)

Constraints:

i) At least one color to each node: $(x_{1,1} \vee x_{1,2} \vee \ldots x_{1,k} \vee \ldots)$

$$\bigwedge_{i=1}^{n} \bigvee_{j=1}^{k} (x_{i,j})$$

ii) At most one color to each node:

$$\forall_{i=1}^{n} \forall_{j=1}^{k-1} \forall_{t=j+1}^{k} (\neg x_{i,j} \vee \neg x_{i,t})$$

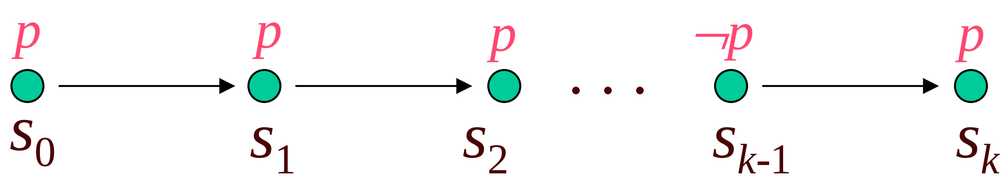
iii) Coloring constraints:

$$\forall_{c=1}^{k} \forall_{i=1}^{n} \forall_{j=1}^{n} (i, j) \in E . (\neg x_{i,c} \vee \neg x_{j,c})$$

Formulation of famous problems as SAT: *Bounded Model Checking* (1/4)

Given a property p: (e.g. "*always* signal_a = signal_b")

Is there a state reachable within k cycles, which satisfies $\neg p$?



Formulation of famous problems as SAT: *Bounded Model Checking* (2/4)

The reachable states in k steps are captured by:

$$I(s_0) \wedge \rho(s_0, s_1) \wedge \rho(s_1, s_2) \wedge \ldots \wedge \rho(s_{k-1}, s_k)$$

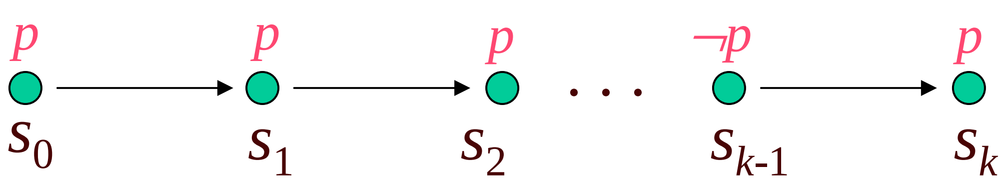
The property p fails in one of the cycles $1..k$:

$$\neg p_1 \vee \neg p_2 \vee \ldots \vee \neg p_k$$

Formulation of famous problems as SAT: *Bounded Model Checking* (3/4)

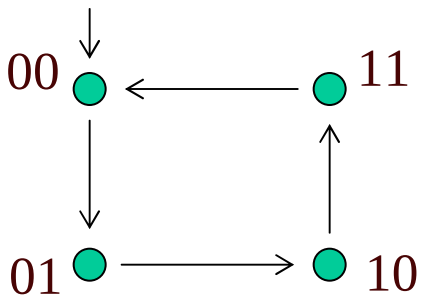
The safety property p is valid up to cycle k iff $\Omega(k)$ is unsatisfiable:

$$\Omega(k): \quad I_0 \wedge \bigwedge_{i=0}^{k-1} \rho(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg p_i$$



Formulation of famous problems as SAT: *Bounded Model Checking* (4/4)

Example: a two bit counter



Initial state: $I_0: \neg l \wedge \neg r$

Transition: $\rho: \begin{array}{l} l' = (l \neq r) \\ r' = \neg r \end{array}$

Property: always $(\neg l \vee \neg r)$.

For $k = 2$:

$$\varphi: (\neg l_0 \wedge \neg r_0) \wedge \begin{bmatrix} l_1 = (l_0 \neq r_0) \wedge r_1 = \neg r_0 \wedge \\ l_2 = (l_1 \neq r_1) \wedge r_2 = \neg r_1 \end{bmatrix} \wedge \begin{bmatrix} (l_0 \wedge r_0) \vee \\ (l_1 \wedge r_1) \vee \\ (l_2 \wedge r_2) \end{bmatrix}$$

For $k = 2$, $\Omega(k)$ is unsatisfiable. For $k = 4$ $\Omega(k)$ is satisfiable

What is SAT?

Given a propositional formula in CNF, find an assignment to Boolean variables that makes the formula true:

$$\omega_1 = (x_2 \vee x_3)$$

$$\omega_2 = (\neg x_1 \vee \neg x_4)$$

$$\omega_3 = (\neg x_2 \vee x_4)$$

$$A = \{x_1=0, x_2=1, x_3=0, x_4=1\}$$

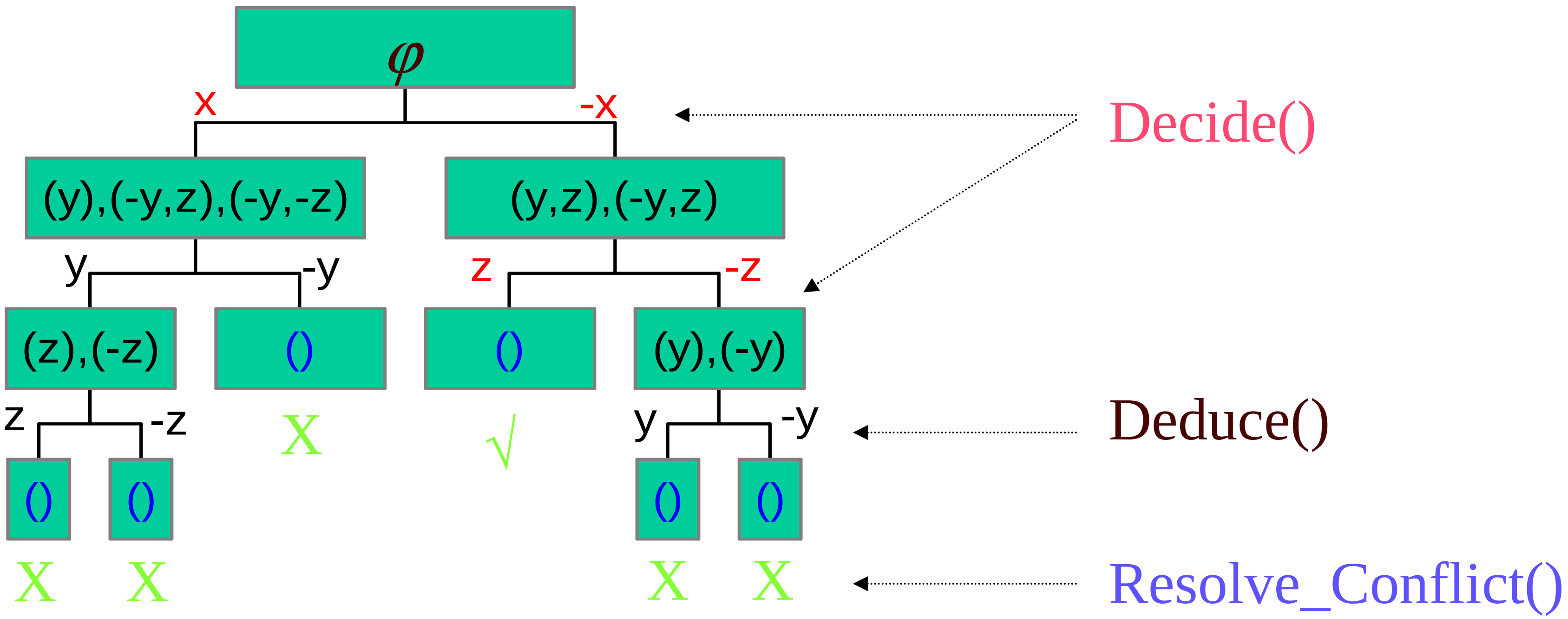
SATisfying assignment!

Why SAT?

- Fundamental problem from theoretical point of view
- Numerous applications:
  - CAD, VLSI
  - Optimization
  - Bounded Model Checking and other type of formal verification
  - AI, planning, automated deduction

A Basic SAT algorithm

Given φ in CNF: (x,y,z),(-x,y),(-y,z),(-x,-y,-z)



A Basic SAT algorithm

```
While (true)
{
        if (!Decide()) return (SAT);
        while (!Deduce())
                if (!Resolve_Conflict()) return (UNSAT);
}
```

Choose the next variable and value. Return False if all variables are assigned

Apply unit clause rule. Return False if reached a conflict

Backtrack until no conflict. Return False if impossible

Basic Backtracking Search

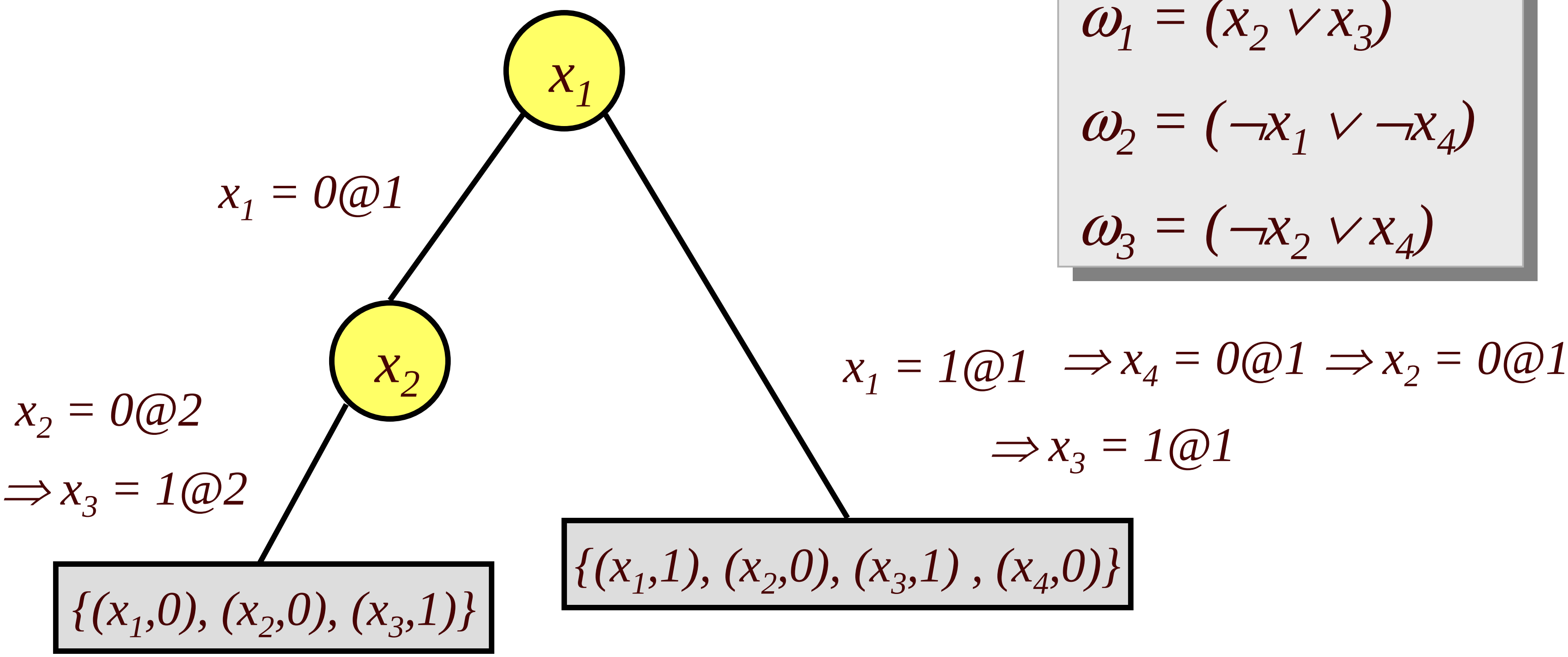
- Organize the search in the form of a decision tree
  - Each node corresponds to a decision
  - Depth of the node in the decision tree $\rightarrow$ decision level
  - Notation: $x=v@d$
    $x \in \{0,1\}$ is assigned to v at decision level d

Backtracking Search in Action

$\omega_1 = (x_2 \vee x_3)$

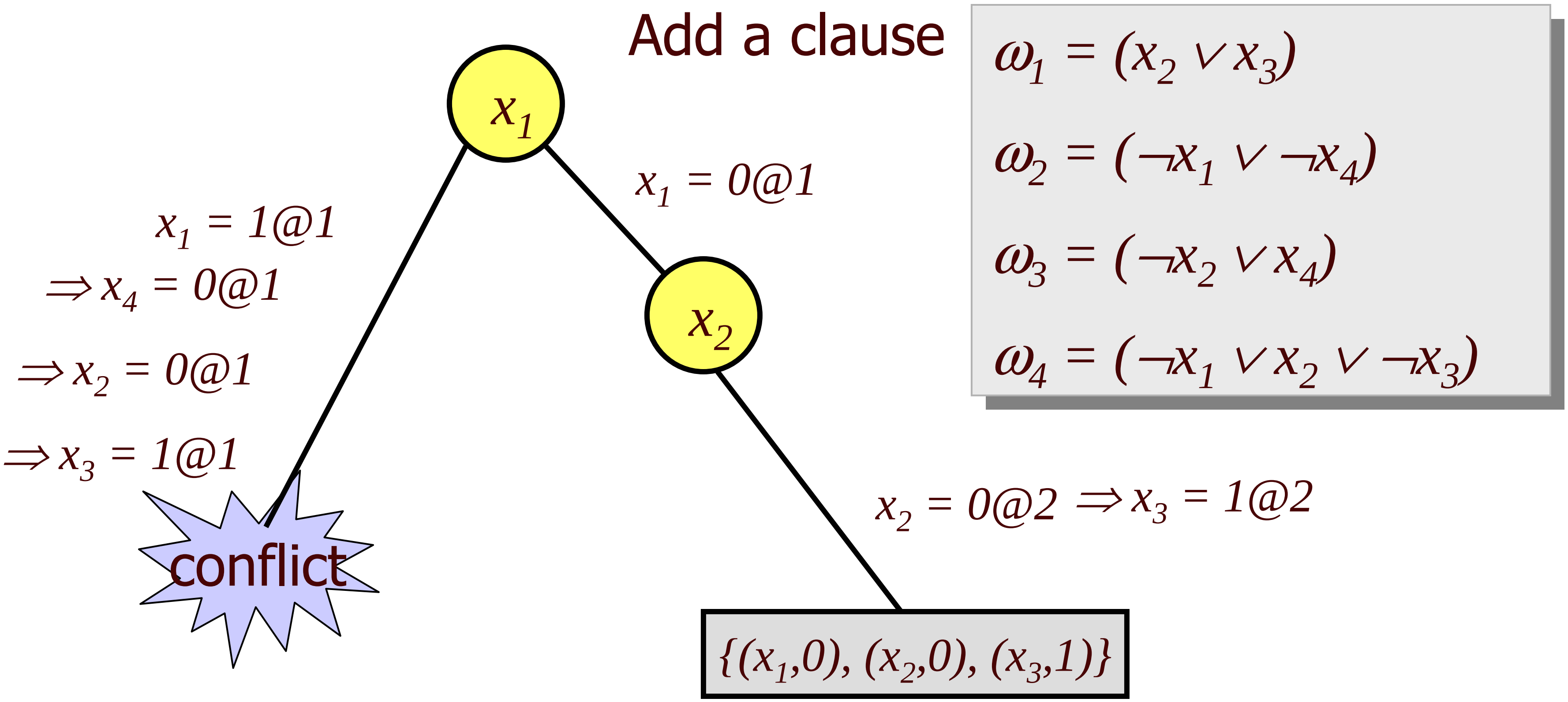
$\omega_2 = (\neg x_1 \vee \neg x_4)$

$\omega_3 = (\neg x_2 \vee x_4)$



No backtrack in this example!

Backtracking Search in Action



Decision heuristics

- <u>DLIS</u> (Dynamic Largest Individual Sum)

  For a given variable x:

  - $C_{x,p}$ – # unresolved clauses in which x appears positively
  - $C_{x,n}$ - # unresolved clauses in which x appears negatively
  - Let x be the literal for which $C_{x,p}$ is maximal
  - Let y be the literal for which $C_{y,n}$ is maximal
  - If $C_{x,p} > C_{y,n}$ choose x and assign it TRUE
  - Otherwise choose y and assign it FALSE
- Requires l (#literals) queries for each decision.
- (Implemented in e.g. **Grasp**)

Decision heuristics

Jeroslow-Wang method

Compute for every clause ω and every variable l (in each phase):

- J(l) := $\sum_{l \in \omega, \omega \in \varphi} 2^{-|\omega|}$
- Choose a variable l that maximizes J(l).
- This gives an exponentially higher weight to literals in shorter clauses.

Decision heuristics

<u>MOM</u> (Maximum Occurrence of clauses of Minimum size).

- Let $f^*(x)$ be the # of unresolved smallest clauses containing x. Choose x that maximizes:

  $$((f^*(x) + f^*(!x)) * 2^k + f^*(x) * f^*(!x)$$

- k is chosen heuristically.

- The idea:
  - Give preference to satisfying small clauses.
  - Among those, give preference to balanced variables (e.g. $f^*(x) = 3, f^*(!x) = 3$ is better than $f^*(x) = 1, f^*(!x) = 5$).

Decision heuristics

VSIDS (Variable State Independent Decaying Sum)

1. Each variable in each polarity has a counter initialized to 0.
2. When a clause is added, the counters are updated.
3. The unassigned variable with the highest counter is chosen.
4. Periodically, all the counters are divided by a constant.

(Implemented in **Chaff**)

Decision heuristics

VSIDS (cont'd)

- **Chaff** holds a list of unassigned variables sorted by the counter value.
- Updates are needed only when adding conflict clauses.
- Thus - decision is made in constant time.

Decision heuristics

VSIDS is a 'quasi-static' strategy:

- *static* because it doesn't depend on current assignment

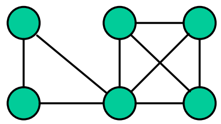
- *dynamic* because it gradually changes. Variables that appear in recent conflicts have higher priority.

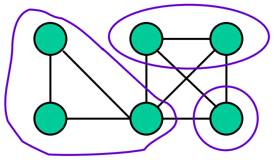
This strategy is a *conflict-driven* decision strategy.

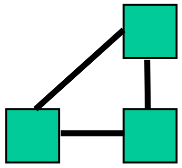
"..employing this strategy dramatically (i.e. an order of magnitude) improved performance ... "

Variable ordering

(Abstract dependency graphs)

A (CNF) dependency graph D (V,E): 

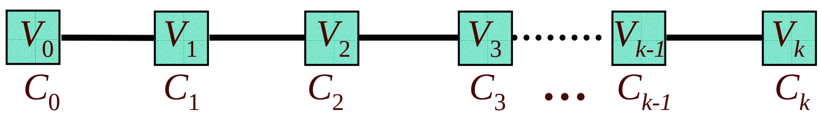
A partitioning $C_1..C_n$: 

An *abstract* dependency graph D'(V', E'): 

Variable ordering

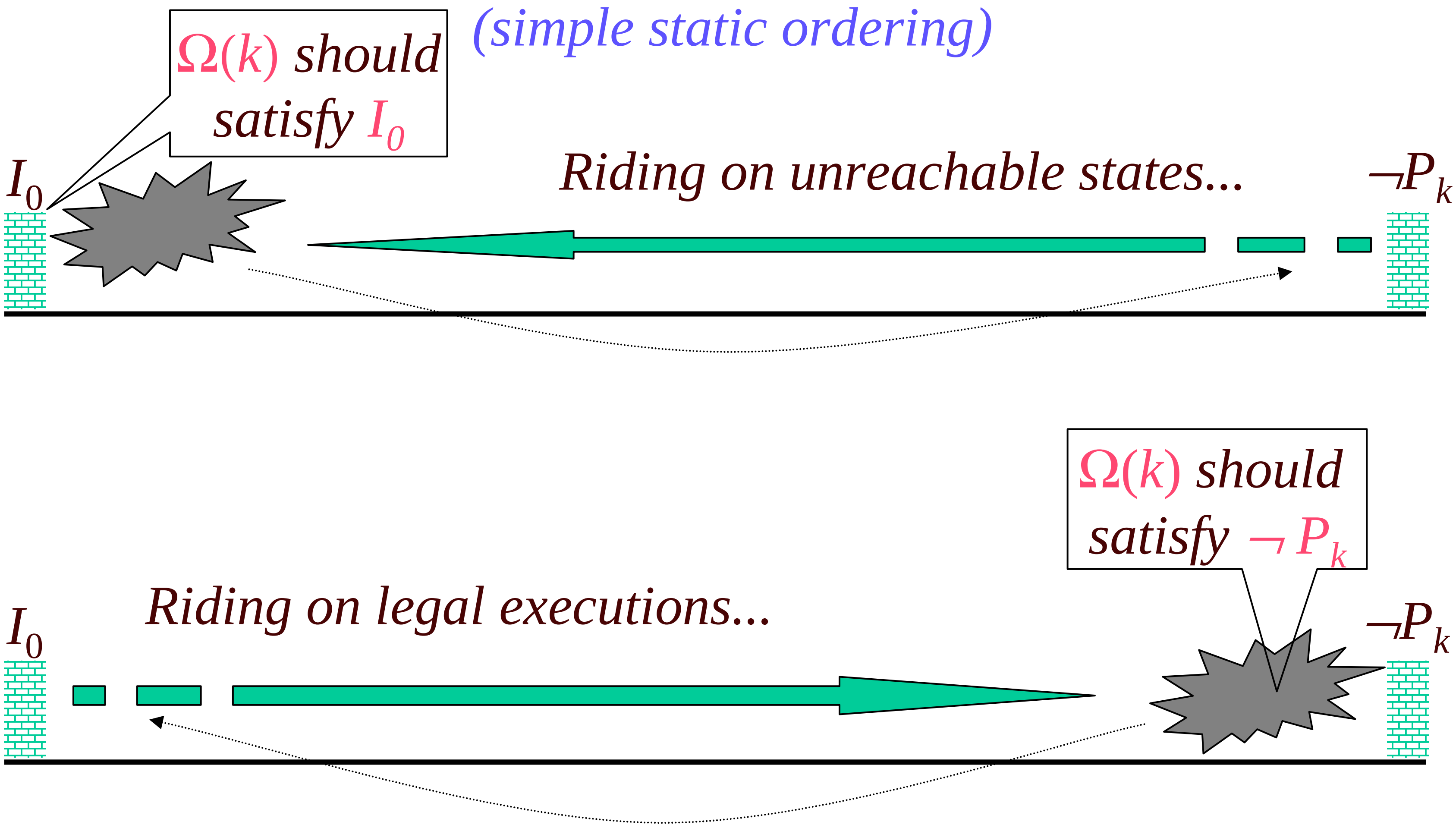
(The natural order of Ω(k))

For $\Omega(k)$ there exists a partition $C_1..C_n$ s.t. the abstract dependency graph is linear



$$\Omega(k): \quad I(s_0) \wedge \bigwedge_{i=1}^{k-1} \rho(s_i, s_{i+1}) \wedge \neg p(s_k)$$

Variable ordering



Implication graphs and learning

Current truth assignment: $\{x_9=0@1, x_{10}=0@3, x_{11}=0@3, x_{12}=1@2, x_{13}=1@2\}$

Current decision assignment: $\{x_1=1@6\}$

$\omega_1 = (\neg x_1 \vee x_2)$

$\omega_2 = (\neg x_1 \vee x_3 \vee x_9)$

$\omega_3 = (\neg x_2 \vee \neg x_3 \vee x_4)$

$\omega_4 = (\neg x_4 \vee x_5 \vee x_{10})$

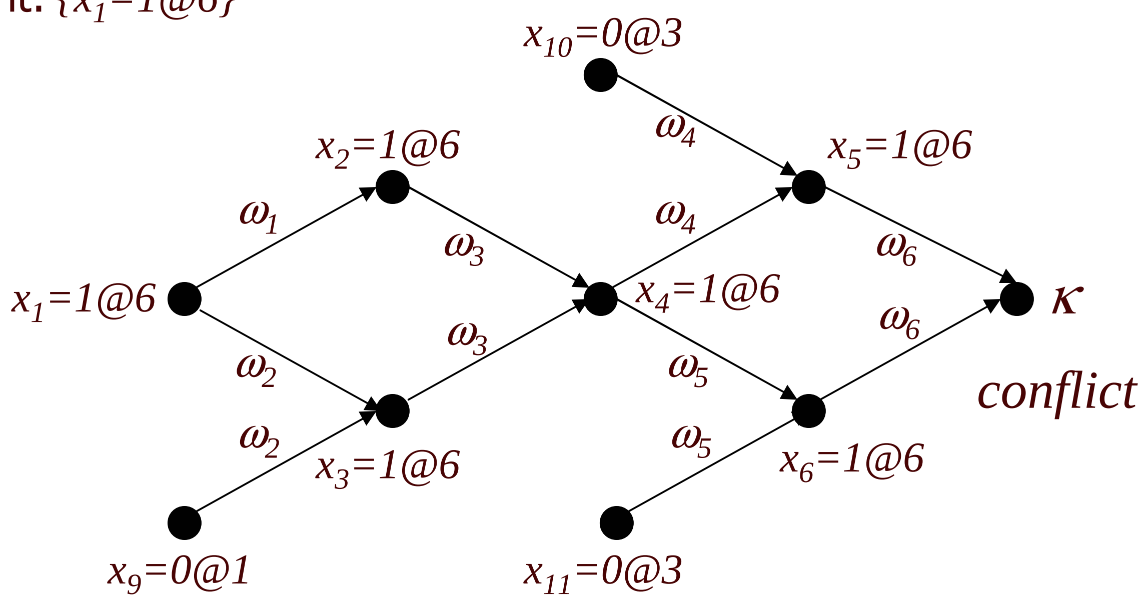
$\omega_5 = (\neg x_4 \vee x_6 \vee x_{11})$

$\omega_6 = (\neg x_5 \vee \neg x_6)$

$\omega_7 = (x_1 \vee x_7 \vee \neg x_{12})$

$\omega_8 = (x_1 \vee x_8)$

$\omega_9 = (\neg x_7 \vee \neg x_8 \vee \neg x_{13})$



We learn the *conflict clause* $\omega_{10} : (\neg x_1 \vee x_9 \vee x_{11} \vee x_{10})$

Implication graph, flipped assignment

$\omega_1 = (\neg x_1 \vee x_2)$

$\omega_2 = (\neg x_1 \vee x_3 \vee x_9)$

$\omega_3 = (\neg x_2 \vee \neg x_3 \vee x_4)$

$\omega_4 = (\neg x_4 \vee x_5 \vee x_{10})$

$\omega_5 = (\neg x_4 \vee x_6 \vee x_{11})$

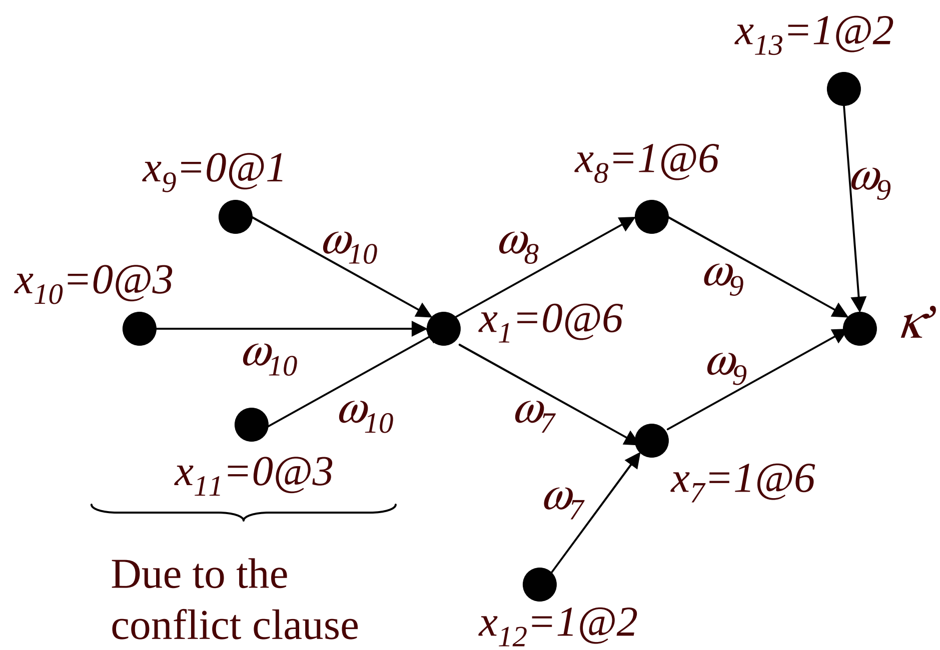
$\omega_6 = (\neg x_5 \vee x_6)$

$\omega_7 = (x_1 \vee x_7 \vee \neg x_{12})$

$\omega_8 = (x_1 \vee x_8)$

$\omega_9 = (\neg x_7 \vee \neg x_8 \vee \neg x_{13})$

$\omega_{10} : (\neg x_1 \vee x_9 \vee x_{11} \vee x_{10})$



Non-chronological backtracking

Which assignments caused the conflicts ?

$x_9 = 0@1$

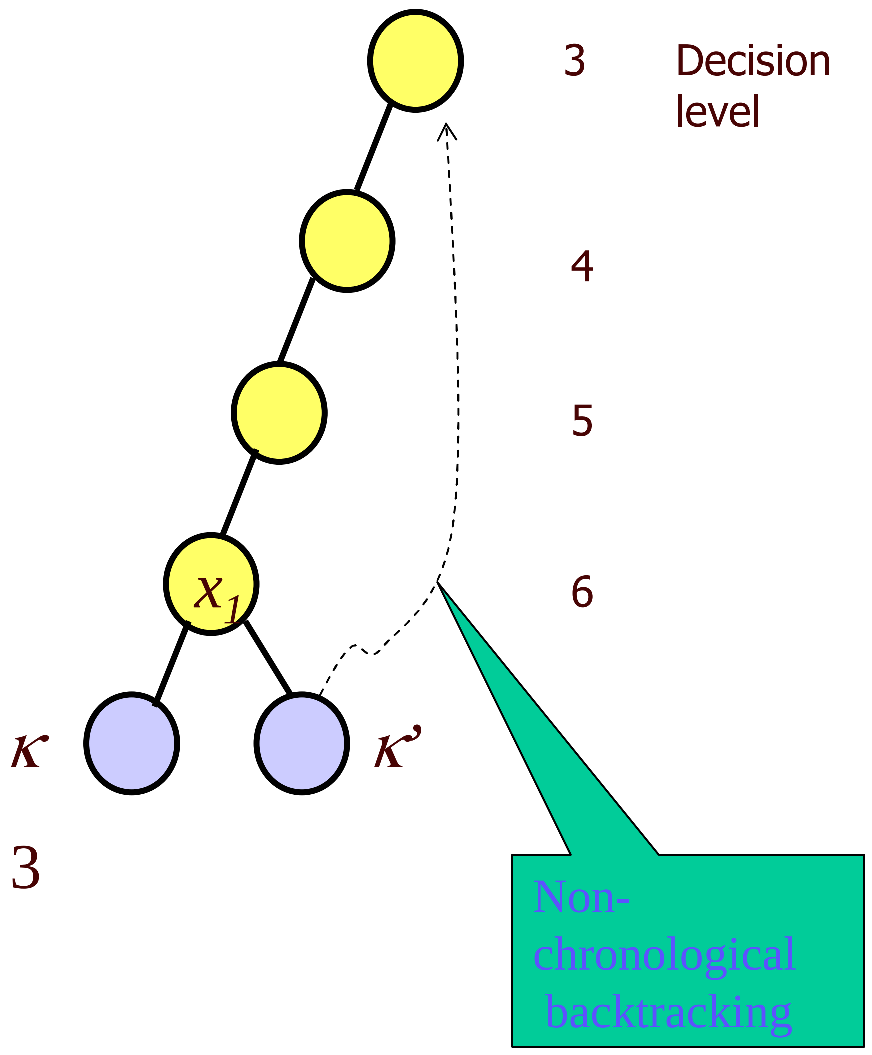
$x_{10} = 0@3$

$x_{11} = 0@3$

$x_{12} = 1@2$

$x_{13} = 1@2$

These assignments Are sufficient for Causing a conflict.



Backtrack to decision level 3

More engineering aspects of SAT solvers

Observation: More than 90% of the time SAT solvers perform Deduction().

Deduction() allocates new implied variables and conflicts.
How can this be done efficiently ?

Grasp implements Deduction() with counters

Hold 2 counters for each clause π:

val1(π) - # of <u>negative</u> literals assigned <u>0</u> in π +
of <u>positive</u> literals assigned <u>1</u> in π.

val0(π) - # of <u>negative</u> literals assigned <u>1</u> in π +
of <u>positive</u> literals assigned <u>0</u> in π.

Grasp implements Deduction() with counters

π is *satisfied*	iff	$val1(\pi) > 0$
π is *unsatisfied*	iff	$val0(\pi) = \|\pi\|$
π is *unit*	iff	$val1(\pi) = 0 \wedge val0(\pi) = \|\pi\| - 1$
π is *unresolved*	iff	$val1(\pi) = 0 \wedge val0(\pi) < \|\pi\| - 1$

.

.

Every assignment to a variable x results in updating the counters for all the clauses that contain x.

Backtracking: Same complexity.

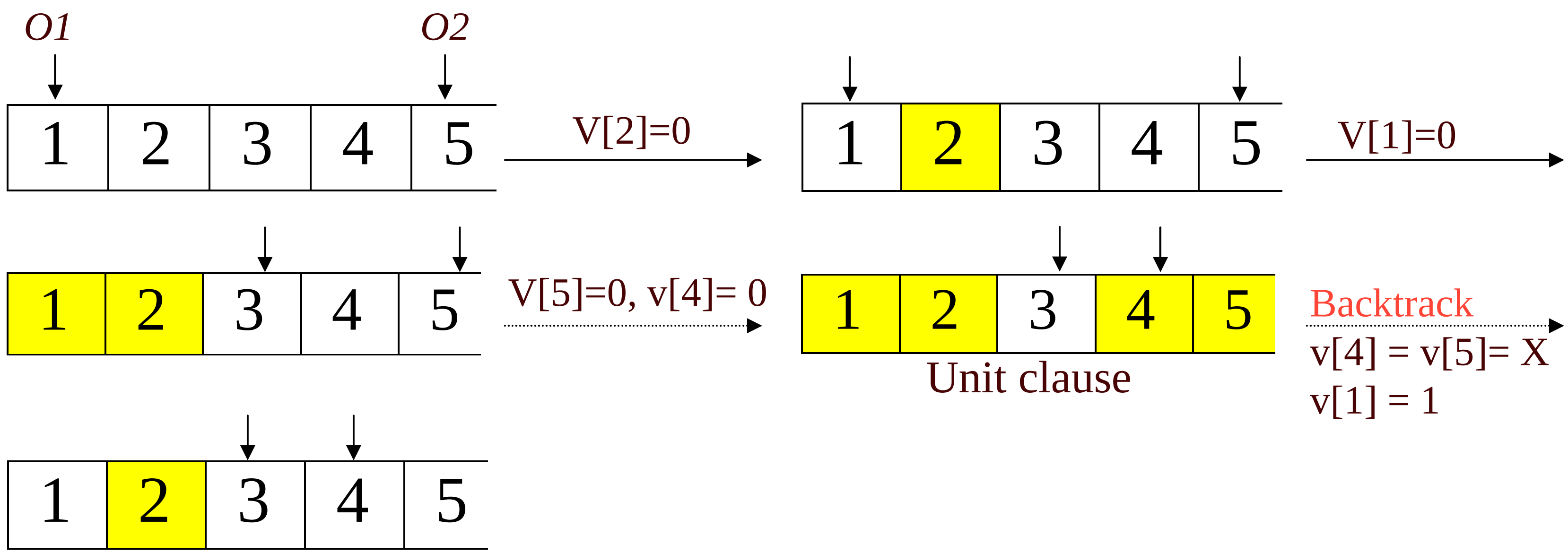
Chaff implements Deduction() with a pair of observers

- Observation: during Deduction(), we are only interested in newly implied variables and conflicts.
- These occur only when the number of literals in π with value ‘false’ is greater than $|\pi|$ - 2
- Conclusion: no need to visit a clause unless ($val0(\pi) > |\pi| - 2$)
- How can this be implemented ?

Chaff implements Deduction() with a pair of observers

- Define two ‘observers’: $O1(\pi)$, $O2(\pi)$.
- $O1(\pi)$ and $O2(\pi)$ point to two distinct π literals which are not ‘false’.
- π becomes *unit* if updating one observer leads to $O1(\pi) = O2(\pi)$.
- Visit clause π only if $O1(\pi)$ or $O2(\pi)$ become ‘false’.

Chaff implements Deduction() with a pair of observers



Both observers of an implied clause are on the highest decision level present in the clause. Therefore, backtracking will un-assign them first. Conclusion: <u>when backtracking, observers stay in place</u>.

Backtracking: No updating. Complexity = constant.

Chaff implements Deduction() with a pair of observers

The choice of observing literals is important.

⇨ Best strategy is - the least frequently updated variables.

The observers method has a *learning curve* in this respect:

1. The initial observers are chosen arbitrarily.

2. The process shifts the observers away from variables that were recently updated (these variables will most probably be reassigned in a short time).

In our example: the next time v[5] is updated, it will point to a significantly smaller set of clauses.

GSAT: A different approach to SAT solving

Given a CNF formula α, choose max_tries and max_flips

```
for i = 1 to max_tries {
  T := randomly generated truth assignment
  for j = 1 to max_flips {
    if T satisfies α return TRUE
    choose v s.t. flipping v's value gives largest increase in
            the # of satisfied clauses (break ties randomly).
    T := T with v's assignment flipped.  } }
```

Improvement # 1: clause weights

Initial weight of each clause: 1

Increase by k the weight of unsatisfied clauses.

Choose v according to max increase in weight

Clause weights is another example of *conflict-driven* decision strategy.

Improvement # 2: Averaging-in

Q: Can we reuse information gathered in previous tries in order to speed up the search ?

A: Yes! Rather than choosing T randomly each time, repeat 'good assignments' and choose randomly the rest.

Improvement # 2: Averaging-in (cont'd)

Let X1, X2 and X3 be equally wide bit vectors.

Define a function `bit_average` : $X1 \times X2 \rightarrow X3$ as follows:

$$b_3^i := \begin{cases} b_1^i & b_1^i = b_2^i \\ \text{random} & \text{otherwise} \end{cases}$$

(where b_j^i is the i-th bit in Xj, $j \in \{1,2,3\}$)

Improvement # 2: Averaging-in (cont'd)

Let T_i^{init} be the initial assignment (T) in cycle i.

Let T_i^{best} be the assignment with highest # of satisfied clauses in cycle i.

T_1^{init} := random assignment.

T_2^{init} := random assignment.

$\forall i > 2$, T_i^{init} := `bit_average`(T_{i-1}^{best}, T_{i-2}^{best})