SAT Solvers:
In Practice

Type Theory and Mechanized Reasoning
Lecture 8

CAS CS 400

Introduction

Administrivia

Homework 2 is due on Thursday 11:59PM. Homework 3 will be released on Thursday (Tomorrow).

I will post about the **Final Project** on Friday. The rough outline:

» Groups of 1, 2, or 3
» Implementation, Proof, or Survey
» Weekly assignments will include progress reports
» By next week, you should have a group and a rough idea

Objectives

Finish our discussion on DPLL.

Talk about encoding with CNF formulas.

Build a sudoku solver with a SAT solver.

Agda Tutorial: Implicit Arguments

Recall: Polymorphism

```
k : (A : Set) -> (B : Set) -> A -> B -> A
k A B x y = x
```

Recall: Polymorphism

```
k : (A : Set) -> (B : Set) -> A -> B -> A
k A B x y = x
```

In Agda, polymorphism comes for free with other language features.

Recall: Polymorphism

```
k : (A : Set) -> (B : Set) -> A -> B -> A
k A B x y = x
```

In Agda, polymorphism comes for free with other language features.

This is an example of parametric polymorphism which means that the definition of **k** is *completely agnostic to the types **A** and **B**.*

Recall: Polymorphism

```
k : (A : Set) -> (B : Set) -> A -> B -> A
k _ _ x y = x
```

Recall: Polymorphism

```
k : (A : Set) -> (B : Set) -> A -> B -> A
k _ _ x y = x
```

A and **B** only aren't necessary for the computation of **k**, only for the type.

Recall: Polymorphism

```
k : (A : Set) -> (B : Set) -> A -> B -> A
k _ _ x y = x
```

A and **B** only aren't necessary for the computation of **k**, only for the type.

So we can replace them with wildcards in the definition of **k**.

Recall: Polymorphism

```
k : {A : Set} -> {B : Set} -> A -> B -> A
k x y = x
```

Recall: Polymorphism

```
k : {A : Set} -> {B : Set} -> A -> B -> A
k x y = x
```

When this happens, we can make the arguments implicit.

Recall: Polymorphism

```
k : {A : Set} -> {B : Set} -> A -> B -> A
k x y = x
```

When this happens, we can make the arguments implicit.

The values of **A** and **B** can be solved based on the inputs given to **k**.

Recall: Polymorphism

```
k : {A : Set} -> {B : Set} -> A -> B -> A
k x y = x
```

When this happens, we can make the arguments implicit.

The values of **A** and **B** can be solved based on the inputs given to **k**.

This solving process is call unification.

Unification and Implicit Arguments

```
k : {A : Set} -> {B : Set} -> A -> B -> A
k x y = x

c : {A B C : Set} -> (A -> B) -> (B -> C) -> (A -> C)
c f g x = g (f x)

i : {A : Set} -> A -> A
i x = x

f : {A B : Set} -> A -> B -> A
f x = c (k x) i
```

Unification and Implicit Arguments

```
k : {A : Set} -> {B : Set} -> A -> B -> A
k x y = x

c : {A B C : Set} -> (A -> B) -> (B -> C) -> (A -> C)
c f g x = g (f x)

i : {A : Set} -> A -> A
i x = x

f : {A B : Set} -> A -> B -> A
f x = c (k x) i
```

We will not cover unification, but hopefully we can intuit it somewhat.

Unification and Implicit Arguments

```
k : {A : Set} -> {B : Set} -> A -> B -> A
k x y = x

c : {A B C : Set} -> (A -> B) -> (B -> C) -> (A -> C)
c f g x = g (f x)

i : {A : Set} -> A -> A
i x = x

f : {A B : Set} -> A -> B -> A
f x = c (k x) i
```

We will not cover unification, but hopefully we can intuit it somewhat.

When a value can be solved via unification, then can made implicit without issues.

An Example with Dependent Types

```
head : {A : Set} -> {n : Nat} -> Vec A (suc n) -> A
head (x :: _) = x

foo : Nat
foo = head (1 :: 2 :: [])
```

After applying head to a value, the input **n** can be solved, so it can be made implicit.

Passing in Implicit Arguments

```
allVals : {n : Nat} -> List (Vec Bool n)
allVals {zero} = [] :: []
allVals {suc n} = go (allVals {n}) where
  go : List (Vec Bool n) -> List (Vec Bool (suc n))
  go [] = []
  go (x :: xs) = (true :: x) :: (false :: x) :: go xs
```

It is occasionally necessary to work directly with implicit arguments. To do this, we pass in values with {_}'s.

Passing in Implicit Arguments

```
allVals : {n : Nat} -> List (Vec Bool n)
allVals {zero} = [] :: []
allVals {suc n} = go (allVals {n}) where
  go : List (Vec Bool n) -> List (Vec Bool (suc n))
  go [] = []
  go (x :: xs) = (true :: x) :: (false :: x) :: go xs
```

It is occasionally necessary to work directly with implicit arguments. To do this, we pass in values with {_}'s.

The Takeaway

```
k : (A : Set) -> (B : Set) -> A -> B -> A
k _ _ x y = x
```

If you can pass a wildcard for argument in the definition of a function, it can be made implicit.

(Even if you can't it may still be useful).

Question

```
foo : Bool -> Set
foo true = Nat
foo false = Nat

bar : (b : Bool) -> foo b -> Nat
bar true x = x
bar false x = x
```

Can ***b*** *in* ***bar*** *be made implicit?*

Recap: SAT Solvers

Recall: Satisfiability

Recall: Satisfiability

Definition. A formula ϕ is **satisfiable** if there is a valuation v such that $\overline{v}(\phi) = \mathsf{true}$.

Recall: Satisfiability

Definition. A formula ϕ is **satisfiable** if there is a valuation v such that $\overline{v}(\phi) = \mathsf{true}$.

Example. $(x \vee y) \wedge (\neg x \vee y) \wedge (\neg x \vee \neg y)$ is satisfied by

$$v(y) = \mathsf{true} \quad v(x) = \mathsf{false} \quad v(\_) = \mathsf{false}$$

Recall: Conjunctive Normal Form

Recall: Conjunctive Normal Form

A **literal** is a propositional variable or its negation.

Recall: Conjunctive Normal Form

A **literal** is a propositional variable or its negation.

A formula is in **conjunctive normal form** if is is a conjunction (and) of disjunctions (or) of literals, e.g.

Recall: Conjunctive Normal Form

A **literal** is a propositional variable or its negation.

A formula is in **conjunctive normal form** if is is a conjunction (and) of disjunctions (or) of literals, e.g.

$$(x \vee \neg y) \wedge \neg y \wedge (\neg y \vee \neg z \vee \neg w)$$

Recall: Conjunctive Normal Form

A **literal** is a propositional variable or its negation.

A formula is in **conjunctive normal form** if is is a conjunction (and) of disjunctions (or) of literals, e.g.

$$\underbrace{(x \lor \neg y)}_{\text{clause}} \land \underbrace{\neg y}_{\text{clause}} \land \underbrace{(\neg y \lor \neg z \lor \neg w)}_{\text{clause}}$$

Recall: SAT

Recall: SAT

The **satisfiability problem (SAT)** is the computational problem:

Recall: SAT

The **satisfiability problem (SAT)** is the computational problem:

Given a CNF formula ϕ, determine a satisfying assignment for ϕ or determine that no such satisfying assignment exists.

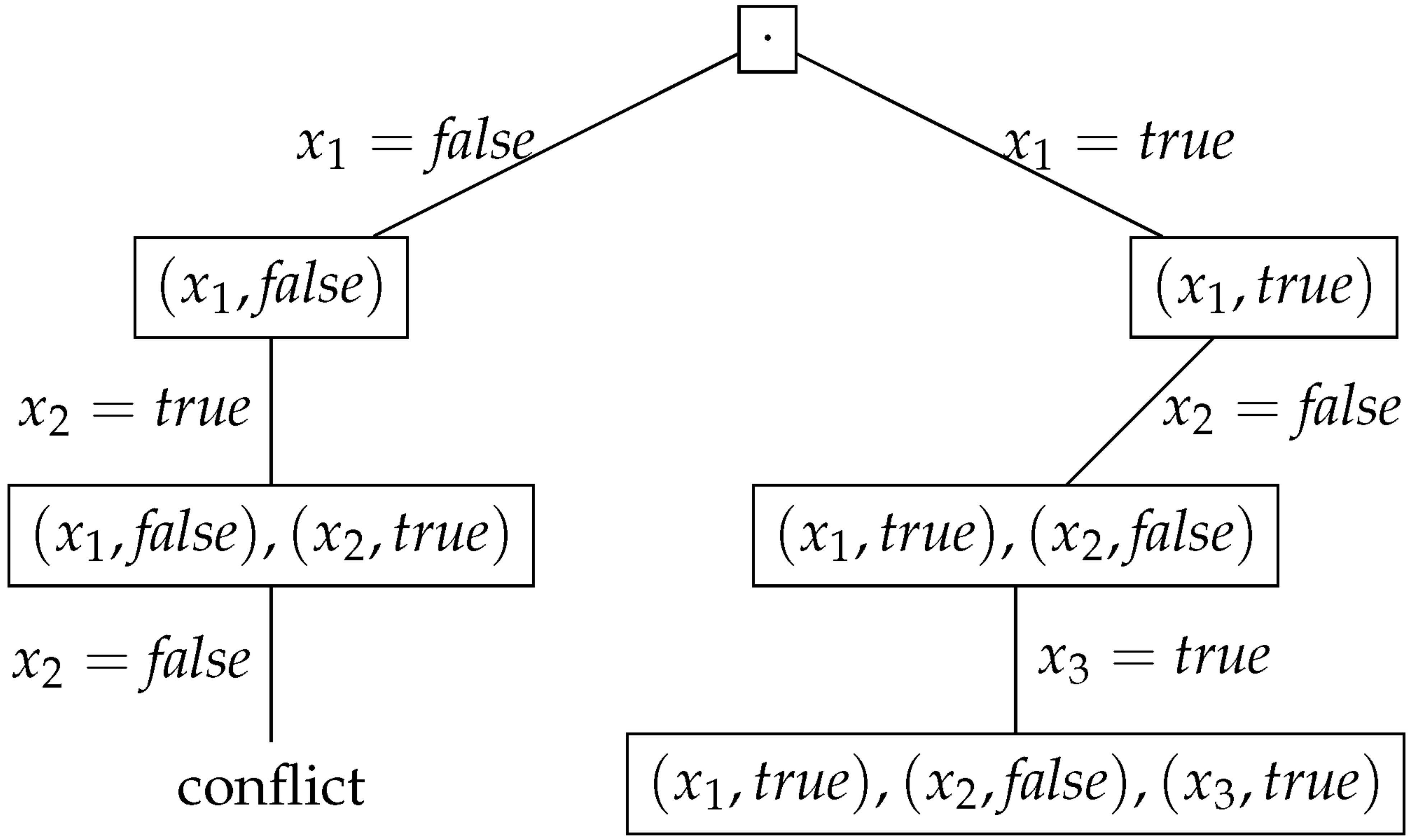
Recall: SAT

The **satisfiability problem (SAT)** is the computational problem:

Given a CNF formula ϕ, determine a satisfying assignment for ϕ or determine that no such satisfying assignment exists.

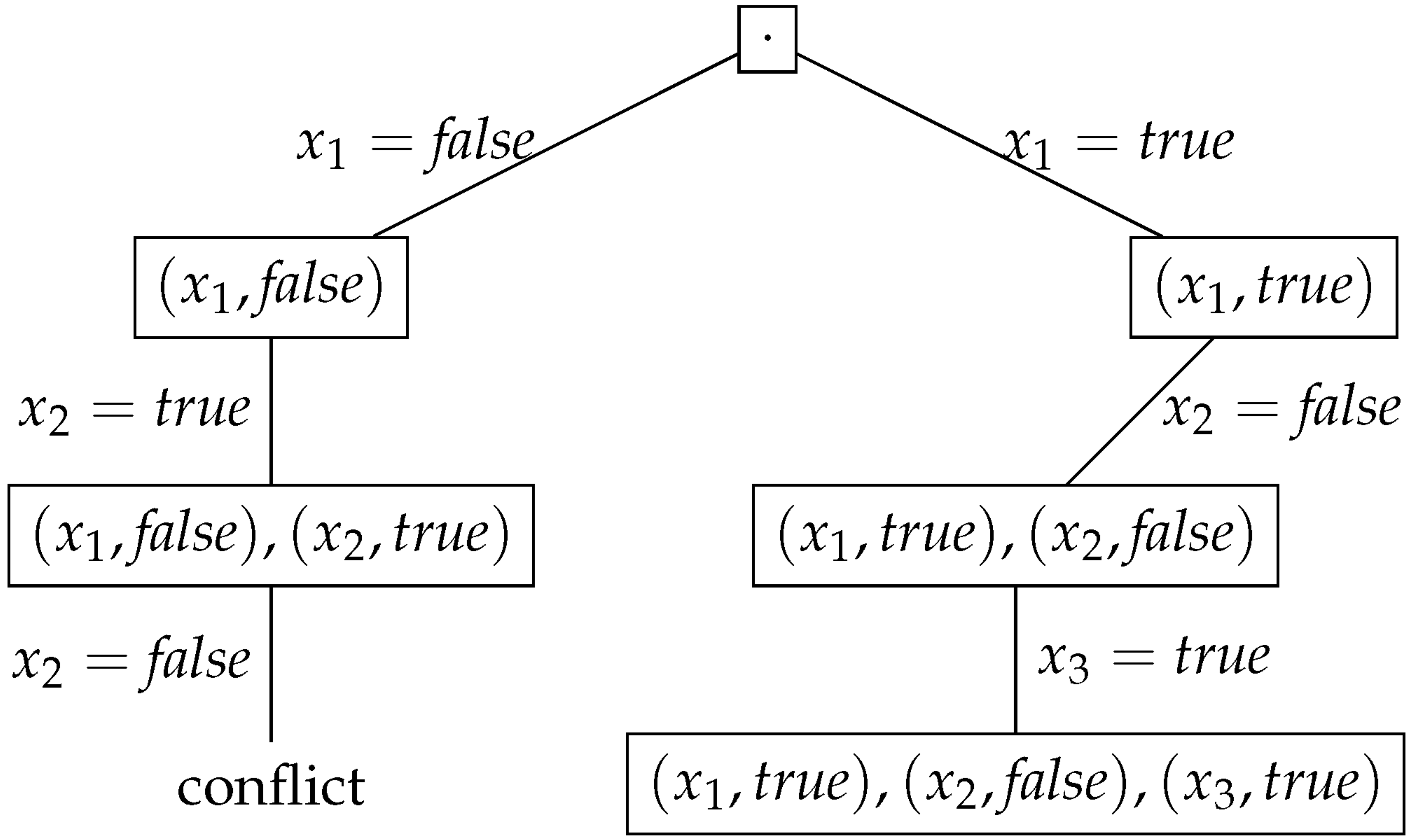
Another view. Can we find an assignment which satisfies **at least one literal of every clause**?

Recall: DPLL



https://www.mdpi.com/2227-7390/10/13/2264

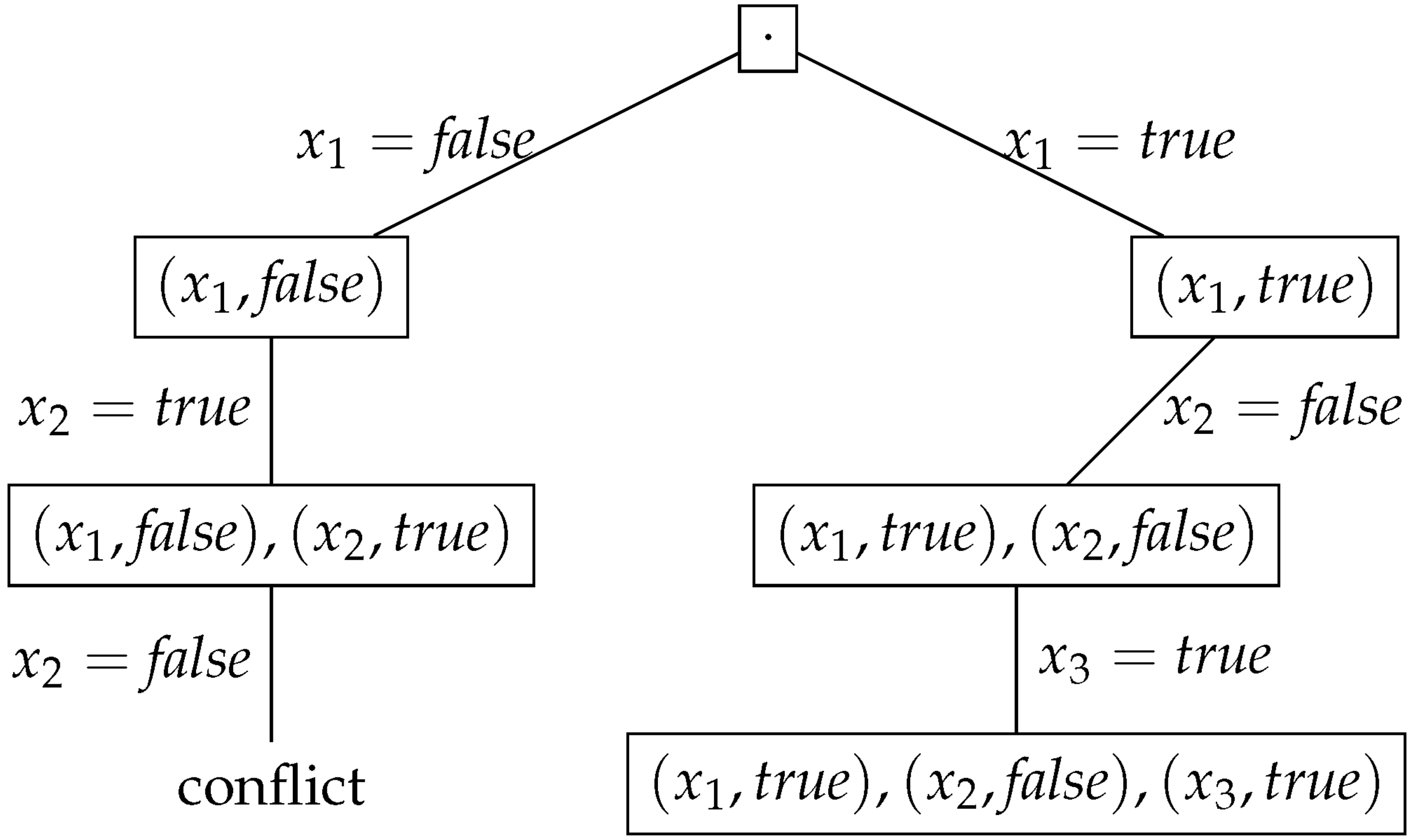
Recall: DPLL



Idea. Build a satisfying assignment one variable at a time, updating the formula at each step.

https://www.mdpi.com/2227-7390/10/13/2264

Recall: DPLL



Idea. Build a satisfying assignment one variable at a time, updating the formula at each step.

This is a backtracking procedure, where a leaf is a satisfying assignment or a **conflict** (the assignment cannot be satisfying).

https://www.mdpi.com/2227-7390/10/13/2264

Recall: Partial Assignments

Definition. A **partial assignment** is a set of literals.

Example. $\{x^1, y^0, z^1\}$

We think of this as a set of *assertions*, i.e., x is true and y is false and z is true.

Restriction by a Literal

Restriction by a Literal

Definition. Given a formula ϕ and a literal l the **restriction** of ϕ by l, written $\phi|_l$ is given by

Restriction by a Literal

Definition. Given a formula ϕ and a literal l the **restriction** of ϕ by l, written $\phi|_l$ is given by

$$C|_l = C \text{ if } l \notin C$$

Restriction by a Literal

Definition. Given a formula ϕ and a literal l the **restriction** of ϕ by l, written $\phi|_l$ is given by

$$C|_l = C \text{ if } l \notin C$$

$$(C \vee x^a \vee D)|_{x^b} = \begin{cases} C \vee D & a \neq b \\ \mathsf{true} & \text{otherwise} \end{cases}$$

Restriction by a Literal

Definition. Given a formula ϕ and a literal l the **restriction** of ϕ by l, written $\phi|_l$ is given by

$$C|_l = C \text{ if } l \notin C$$

$$(C \vee x^a \vee D)|_{x^b} = \begin{cases} C \vee D & a \neq b \\ \text{true} & \text{otherwise} \end{cases}$$

$$(C_1 \wedge C_2 \wedge \ldots \wedge C_k)|_l = (C_1|_l) \wedge (C_2|_l) \ldots \wedge (C_k|_l)^*$$

Restriction by a Literal

Definition. Given a formula ϕ and a literal l the **restriction** of ϕ by l, written $\phi|_l$ is given by

$$C|_l = C \text{ if } l \notin C$$

$$(C \vee x^a \vee D)|_{x^b} = \begin{cases} C \vee D & a \neq b \\ \text{true} & \text{otherwise} \end{cases}$$

$$(C_1 \wedge C_2 \wedge \ldots \wedge C_k)|_l = (C_1|_l) \wedge (C_2|_l) \ldots \wedge (C_k|_l)^*$$

*This can be made more efficient.

General Restriction

Restriction by a partial assignment can be understood as repeated restriction by literals, e.g.*

$$\phi|_{\{l_1,l_2\}} = (\phi|_{l_1})|_{l_2}$$

*This elides questions about order of restrictions

Naive DPLL in Agda

```
{-# TERMINATING #-}
is-sat : CNF -> Bool
is-sat f with find-var f
is-sat f | Nothing = notb (has-empty f)
is-sat f | Just x with is-sat (restrict (x , true) f)
is-sat f | Just x | true = true
is-sat f | Just x | false = is-sat (restrict (x , false) f)
```

High Level: is-sat branches the choice of restricting by x or $\neg x$.

Heuristics

Heuristics

Unit Propagation. If the formula has a clause which is a single literal l, then restrict the formula by l.

Heuristics

Unit Propagation. If the formula has a clause which is a single literal l, then restrict the formula by l.

Example. $(x^0 \wedge (x^1 \vee y^0))|_{x^0}$ becomes y^0

Heuristics

Unit Propagation. If the formula has a clause which is a single literal l, then restrict the formula by l.

Example. $(x^0 \wedge (x^1 \vee y^0))|_{x^0}$ becomes y^0

Pure Literal Rule. If the formula has only appearance of l, then restrict the formula by l.

Heuristics

Unit Propagation. If the formula has a clause which is a single literal l, then restrict the formula by l.

Example. $(x^0 \wedge (x^1 \vee y^0))|_{x^0}$ becomes y^0

Pure Literal Rule. If the formula has only appearance of l, then restrict the formula by l.

Example. $((x^0 \vee y^0) \wedge (x^0 \vee y^1) \wedge z^0)|_{x^0}$ becomes z^0

DPLL Pseudocode

```
Algorithm DPLL
    Input: A set of clauses Φ.
    Output: A truth value indicating whether Φ is satisfiable.
```

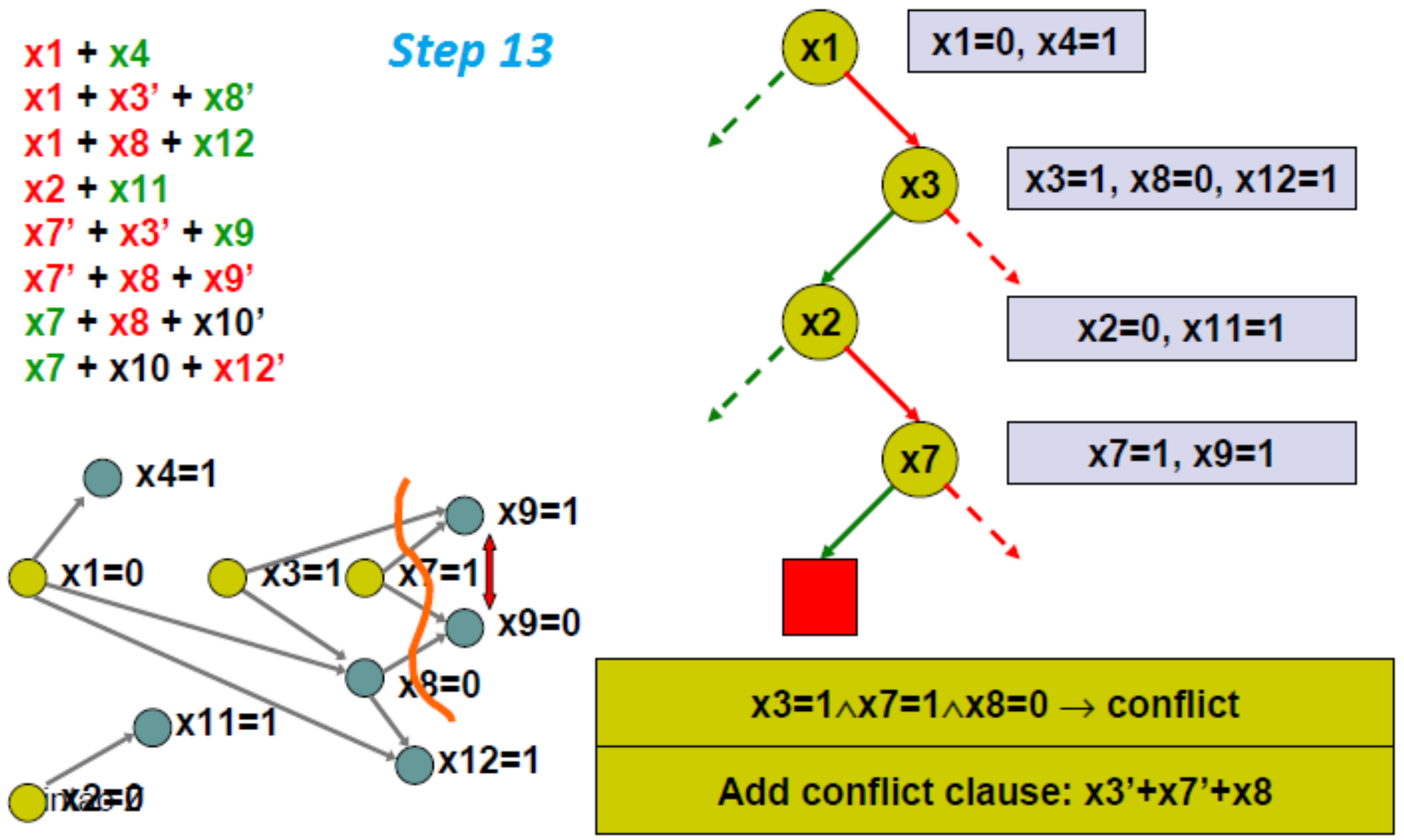
```
function DPLL(Φ)
    // unit propagation:
    while there is a unit clause {l} in Φ do
        Φ ← unit-propagate(l, Φ);
    // pure literal elimination:
    while there is a literal l that occurs pure in Φ do
        Φ ← pure-literal-assign(l, Φ);
    // stopping conditions:
    if Φ is empty then
        return true;
    if Φ contains an empty clause then
        return false;
    // DPLL procedure:
    l ← choose-literal(Φ);
    return DPLL(Φ ∧ {l}) or DPLL(Φ ∧ {¬l});
```

https://en.wikipedia.org/wiki/DPLL_algorithm

That's it. That's the
David–Putnam–Logemann–Loveland Procedure.

backtracking + unit propagation + pure literal rule

A More Complicated Algorithm: CDCL



Modern SAT solvers are built using a heuristic called **conflict driven clause learning (CDCL).**

Idea. When we find that a partial assignment creates a conflict, we can *add* clauses to our formula which might help the solver avoid making the same mistake again.

https://en.wikipedia.org/wiki/Conflict-driven_clause_learning#/media/File:Cdcl13.png

CNF Encodings

Motivation

Motivation

CNF is better for computation, but it is more difficult to express statements in CNF form.

Motivation

CNF is better for computation, but it is more difficult to express statements in CNF form.

Natural encodings in CNF may be very complex, or difficult to work with.

Motivation

CNF is better for computation, but it is more difficult to express statements in CNF form.

Natural encodings in CNF may be very complex, or difficult to work with.

We need to build up a couple tricks for encoding.

"At least one" Encoding

"At least one" Encoding

Given a set of literals $l_1, \ldots, l_n$ we can encode the statement "at least one of $l_1, \ldots, l_n$ is true" as

"At least one" Encoding

Given a set of literals $l_1, \ldots, l_n$ we can encode the statement "at least one of $l_1, \ldots, l_n$ is true" as

$$\text{one}_{\geq}(l_1, \ldots, l_n) = l_1 \vee \ldots \vee l_n$$

"At least one" Encoding

Given a set of literals $l_1, \ldots, l_n$ we can encode the statement "at least one of $l_1, \ldots, l_n$ is true" as

$$\text{one}_{\geq}(l_1, \ldots, l_n) = l_1 \vee \ldots \vee l_n$$

This only introduces one clause.

"At most one" Encoding

"At most one" Encoding

"At most one of $l_1, \ldots, l_n$ is true" is the equivalent to

"At most one" Encoding

"At most one of $l_1, \ldots, l_n$ is true" is the equivalent to

"it can't be that l_i and l_j are both true, for any choice of i and j"

"At most one" Encoding

"At most one of $l_1, \ldots, l_n$ is true" is the equivalent to

"it can't be that l_i and l_j are both true, for any choice of i and j"

$$\text{one}_{\leq}(x_1, \ldots, x_n) \quad = \bigwedge_{1 \leq i < j \leq n} \neg(x_i \wedge x_j) \quad \equiv \quad \bigwedge_{1 \leq i < j \leq n} \neg x_i \vee \neg x_j$$

"At most one" Encoding

"At most one of $l_1, \ldots, l_n$ is true" is the equivalent to

"it can't be that l_i and l_j are both true, for any choice of i and j"

$$\mathsf{one}_{\leq}(x_1, \ldots, x_n) \quad = \bigwedge_{1 \leq i < j \leq n} \neg(x_i \wedge x_j) \quad \equiv \quad \bigwedge_{1 \leq i < j \leq n} \neg x_i \vee \neg x_j$$

Question. *How many clauses does this create?*

"Exactly one" Encoding

Given $l_1, \ldots, l_n$, "exactly one of $l_1, \ldots, l_n$ is true" can be encoded as

$$\text{one}(l_1, \ldots, l_n) = \text{one}_{\leq}(l_1, \ldots, l_n) \wedge \text{one}_{\geq}(l_1, \ldots, l_n)$$

Note that this is still a CNF formula.

Understanding Check

Write a CNF encoding for $\text{two}_{\geq}(l_1, \ldots, l_n)$ *which expresses "at least two of* $l_1, \ldots, l_n$ *hold".*

What about XOR?

What about XOR?

Proposition. Any CNF encoding of $x_1 \oplus \ldots \oplus x_n$ has at least 2^{n-1} clauses.

What about XOR?

Proposition. Any CNF encoding of $x_1 \oplus \ldots \oplus x_n$ has at least 2^{n-1} clauses.

Exclusive or is a very important logical primitive for applications like cryptography.

What about XOR?

Proposition. Any CNF encoding of $x_1 \oplus \ldots \oplus x_n$ has at least 2^{n-1} clauses.

Exclusive or is a very important logical primitive for applications like cryptography.

Some SAT solvers (e.g., CryptoMiniSat) have built-on XOR solvers.

Equisatisfiability

Equisatisfiability

Definition. Formulas ϕ and ψ are **equisatisfiable** if ϕ is satisfiable if and only if ψ is.

Equisatisfiability

Definition. Formulas ϕ and ψ are **equisatisfiable** if ϕ is satisfiable if and only if ψ is.

Example. $(x \oplus y) \wedge (\neg y \oplus z)$ and $x \oplus z$ are equisatisfiable.

Equisatisfiability

Definition. Formulas ϕ and ψ are **equisatisfiable** if ϕ is satisfiable if and only if ψ is.

Example. $(x \oplus y) \wedge (\neg y \oplus z)$ and $x \oplus z$ are equisatisfiable.

Equisatisfiable formulas are not necessarily equivalent.

Equisatisfiability

Definition. Formulas ϕ and ψ are **equisatisfiable** if ϕ is satisfiable if and only if ψ is.

Example. $(x \oplus y) \land (\neg y \oplus z)$ and $x \oplus z$ are equisatisfiable.

Equisatisfiable formulas are not necessarily equivalent.

Question. *Why aren't the above formulas equivalent?*

A Trick using Equisatisfiability

$$\mathsf{one}_{\leq}(x_1, x_2, \ldots, x_6)$$

`and`

$$\mathsf{one}_{\leq}(x_1, x_2, y) \land \mathsf{one}_{\leq}(\neg y, x_3, x_4, z) \land \mathsf{one}_{\leq}(\neg z, x_5, x_6)$$

`are equisatisfiable.`

A Trick using Equisatisfiability

A Trick using Equisatisfiability

$$x_1 \oplus \ldots \oplus x_n$$

`and`

$$(x_1 \oplus x_2 \oplus y) \land (\neg y \oplus x_3 \oplus x_4 \oplus z) \land (\neg z \oplus x_5 \oplus x_6)$$

`are equisatisfiable.`

A Trick using Equisatisfiability

$$x_1 \oplus \ldots \oplus x_n$$

and

$$(x_1 \oplus x_2 \oplus y) \wedge (\neg y \oplus x_3 \oplus x_4 \oplus z) \wedge (\neg z \oplus x_5 \oplus x_6)$$

are equisatisfiable.

This gives can be used to create a CNF formula with linearly many clauses.

Sudoku Solver

The Rules

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Every row has the numbers 1 through 9.

Every column has the numbers 1 through 9.

Every 3×3 box has the number 1 through 9.

Variables

Variables

$x_{i,j,v}$ ≡ "the (i,j)-square has the number v".

Variables

$x_{i,j,v}$ $\equiv$ "the (i,j)–square has the number v".

Example. $\neg(x_{0,0,3} \wedge x_{0,0,5})$ represents "the top–left corner cannot have both 3 and 5.

Variables

$x_{i,j,v} \quad \equiv$ "the (i, j)–square has the number v".

Example. $\neg(x_{0,0,3} \wedge x_{0,0,5})$ represents "the top–left corner cannot have both 3 and 5.

Question. *How many variables will the final formula have?*

The Rules as a CNF

We need clauses for:

1. Every square has a one number
2. The rows are well-formed
3. The columns are well-formed
4. The boxes are well-formed

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Example: Rows

Example: Rows

`Row` i `must have the number` v `:=`

Example: Rows

`Row` i `must have the number` v `:=`

$$\mathsf{one}_{\geq}(x_{i,1,v}, \ldots, x_{i,9,v}) = x_{i,1,v} \vee x_{i,2,v} \vee \ldots \vee x_{i,9,v} \quad (R_{i,v})$$

Example: Rows

Row i must have the number v :=

$$\mathsf{one}_{\geq}(x_{i,1,v}, \ldots, x_{i,9,v}) = x_{i,1,v} \lor x_{i,2,v} \lor \ldots \lor x_{i,9,v} \quad (R_{i,v})$$

Row i must have at most one v :=

Example: Rows

`Row` i `must have the number` v `:=`

$$\text{one}_{\geq}(x_{i,1,v}, \ldots, x_{i,9,v}) = x_{i,1,v} \vee x_{i,2,v} \vee \ldots \vee x_{i,9,v} \quad (R_{i,v})$$

`Row` i `must have at most one` v `:=`

$$\text{one}_{\leq}(x_{i,1,v}, x_{i,2,v}, \ldots, x_{i,9,v}) \quad (S_{i,v})$$

Example: Rows

Row i must have the number v :=

$$\text{one}_{\geq}(x_{i,1,v}, \ldots, x_{i,9,v}) = x_{i,1,v} \vee x_{i,2,v} \vee \ldots \vee x_{i,9,v} \quad (R_{i,v})$$

Row i must have at most one v :=

$$\text{one}_{\leq}(x_{i,1,v}, x_{i,2,v}, \ldots, x_{i,9,v}) \quad (S_{i,v})$$

The rows are well-formed :=

Example: Rows

Row i must have the number v :=

$$\mathsf{one}_{\geq}(x_{i,1,v}, \ldots, x_{i,9,v}) = x_{i,1,v} \vee x_{i,2,v} \vee \ldots \vee x_{i,9,v} \quad (R_{i,v})$$

Row i must have at most one v :=

$$\mathsf{one}_{\leq}(x_{i,1,v}, x_{i,2,v}, \ldots, x_{i,9,v}) \quad (S_{i,v})$$

The rows are well-formed :=

$$\bigwedge_{1 \leq i,v \leq n} R_{i,v} \wedge S_{i,v} \equiv \bigwedge_{1 \leq i,v \leq n} \mathsf{one}(x_{i,1,v}, \ldots, x_{i,9,v})$$

Example: Rows

Columns and Boxes are similar.

Row i must have the number v :=

$$\mathsf{one}_{\geq}(x_{i,1,v}, \ldots, x_{i,9,v}) = x_{i,1,v} \vee x_{i,2,v} \vee \ldots \vee x_{i,9,v} \quad (R_{i,v})$$

Row i must have at most one v :=

$$\mathsf{one}_{\leq}(x_{i,1,v}, x_{i,2,v}, \ldots, x_{i,9,v}) \quad (S_{i,v})$$

The rows are well-formed :=

$$\bigwedge_{1 \leq i,v \leq n} R_{i,v} \wedge S_{i,v} \equiv \bigwedge_{1 \leq i,v \leq n} \mathsf{one}(x_{i,1,v}, \ldots, x_{i,9,v})$$

Adding the Board

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Given a board, we need to add clauses which express that a given position already has a value.

<u>Example.</u> $x_{0,0,5} \wedge x_{0,1,3} \wedge x_{0,4,7} \wedge \ldots$

PySat

```
>>> from pysat.solvers import Glucose3
>>>
>>> g = Glucose3()
>>> g.add_clause([-1, 2])
>>> g.add_clause([-2, 3])
>>> print(g.solve())
>>> print(g.get_model())
...
True
[-1, -2, -3]
```

PySat is an interface for working with a number of different SAT solvers in Python.

Clauses are represented as lists of nonzero integers.

demo