SOLVING HARD PROBLEMS QUICKLY USING SAT SOLVERS

Martin Hořeňovský

Researcher @ Locksley.CZ

SOLVING HARD PROBLEMS QUICKLY USING SAT SOLVERS

Martin Hořeňovský

Researcher @ Locksley.CZ

ABOUT THIS TALK

This talk is about solving real world problems using SAT solvers.

This talk is about solving real world problems using SAT solvers.

SAT solvers will be used as a black box and we will not cover any of the theory behind them.

We will start by going over the boolean satisfaction (SAT) problem.

We will start by going over the boolean satisfaction (SAT) problem.

Then we will learn how to drive a SAT solver from C++.

We will start by going over the boolean satisfaction (SAT) problem.

Then we will learn how to drive a SAT solver from C++.

Then we will apply our newly gained knowledge to two practical examples, Sudoku and Master Key Systems.

INTRODUCTION TO SAT

The boolean satisfaction problem (SAT) is about checking whether a logical formula is *satisfiable*.

The boolean satisfaction problem (SAT) is about checking whether a logical formula is *satisfiable*.

A formula is *satisfiable* if we can assign values to its variables so that the whole formula is true.

```
if (A || B || (!A && !C)) {
    create_new_widget();
} else {
    reuse_old_widget();
}
```

```
if (A || B || (!A && !C)) {
    create_new_widget();
} else {
    reuse_old_widget();
}
```

if (A || B || (!A && !C))

```
if (A || B || (!A && !C)) {
    create_new_widget();
} else {
    reuse_old_widget();
}
```

`if (A || B || (!A && !C))`

$$A \vee B \vee (\neg A \wedge \neg C)$$

LOGICAL OPERATORS

LOGICAL OPERATORS

Negation, also known as NOT [`!A`]

LOGICAL OPERATORS

Negation, also known as NOT [`!A`]

$$\neg \alpha$$

LOGICAL OPERATORS

Negation, also known as NOT [`!A`]

$$\neg\alpha$$

Disjunction, also known as OR [`A || B`]

LOGICAL OPERATORS

Negation, also known as NOT [`!A`]

$$\neg\alpha$$

Disjunction, also known as OR [`A || B`]

$$\alpha \vee \beta$$

LOGICAL OPERATORS

Negation, also known as NOT [`!A`]

$$\neg \alpha$$

Disjunction, also known as OR [`A || B`]

$$\alpha \vee \beta$$

Conjunction, also known as AND [`A && B`]

LOGICAL OPERATORS

Negation, also known as NOT [`!A`]

$$\neg\alpha$$

Disjunction, also known as OR [`A || B`]

$$\alpha \vee \beta$$

Conjunction, also known as AND [`A && B`]

$$\alpha \wedge \beta$$

IMPLICATION

IMPLICATION

$$\alpha \implies \beta$$

IMPLICATION

$$\alpha \implies \beta$$

α	β	$\alpha \implies \beta$
1	1	1
1	0	0
0	1	1
0	0	1

EQUIVALENCE

EQUIVALENCE

$$\alpha \iff \beta$$

EQUIVALENCE

$$\alpha \iff \beta$$

α	β	$\alpha \iff \beta$
1	1	1
1	0	0
0	1	0
0	0	1

All the practical examples in this talk can be formulated using just these logical operators.

All the practical examples in this talk can be formulated using just these logical operators.

There is a small problem; SAT solvers do not accept arbitrary logical formulae.

All the practical examples in this talk can be formulated using just these logical operators.

There is a small problem; SAT solvers do not accept arbitrary logical formulae.

They only accept logical formulae in the *Conjunctive Normal Form* (CNF).

Conjunctive Normal Form (CNF) means that the formula is a *conjunction* of *disjunctive* clauses.

Conjunctive Normal Form (CNF) means that the formula is a *conjunction* of *disjunctive* clauses.

In other words, the formula is an AND of many ORs.

$$A \vee B \vee (\neg A \wedge \neg C)$$

$$A \lor B \lor (\neg A \land \neg C)$$

How do we convert this to CNF?

CONVERSIONS

CONVERSIONS

Every formula can be converted into an *equivalent* CNF formula.

CONVERSIONS

Every formula can be converted into an *equivalent* CNF formula.

It helps if you know De Morgan's laws, distributive laws and some simple identities.

Original clause	Equivalent clause
$\neg\neg\alpha$	α
$\neg(\alpha \wedge \beta)$	$\neg\alpha \vee \neg\beta$
$\neg(\alpha \vee \beta)$	$\neg\alpha \wedge \neg\beta$
$(\alpha \wedge \beta) \vee \gamma$	$(\alpha \vee \gamma) \wedge (\beta \vee \gamma)$
$(\alpha \vee \beta) \wedge \gamma$	$(\alpha \wedge \gamma) \vee (\beta \wedge \gamma)$
$\alpha \implies \beta$	$\neg\alpha \vee \beta$
$\alpha \iff \beta$	$(\alpha \implies \beta) \wedge (\alpha \impliedby \beta)$

$$A \lor B \lor (\neg A \land \neg C)$$

$$A \lor B \lor (\neg A \land \neg C)$$

$$\gamma \lor (\alpha \land \beta)$$

$$A \vee B \vee (\neg A \wedge \neg C)$$

$$\gamma \vee (\alpha \wedge \beta)$$

$$(\gamma \vee \alpha) \wedge (\gamma \vee \beta)$$

$$A \lor B \lor (\neg A \land \neg C)$$

$$\gamma \lor (\alpha \land \beta)$$

$$(\gamma \lor \alpha) \land (\gamma \lor \beta)$$

$$(A \lor B \lor \neg A) \land (A \lor B \lor \neg C)$$

$$A \lor B \lor (\neg A \land \neg C)$$

$$\gamma \lor (\alpha \land \beta)$$

$$(\gamma \lor \alpha) \land (\gamma \lor \beta)$$

$$(A \lor B \lor \neg A) \land (A \lor B \lor \neg C)$$

$$(A \lor B \lor \neg C)$$

That's all we need to know about (CNF-)SAT.

That's all we need to know about (CNF-)SAT.

At least for now.

HOW TO DRIVE SAT SOLVER FROM C++

We will be using MiniSat's C++ interface.

We will be using MiniSat's C++ interface.

There is a CMake-integrated fork at
https://github.com/master-keying/minisat

We will be using MiniSat's C++ interface.

There is a CMake-integrated fork at https://github.com/master-keying/minisat

It is also in vcpkg as "minisat-master-keying".

MiniSat's interface is based around 4 basic types,

MiniSat's interface is based around 4 basic types,

- `Solver` - The solver itself

MiniSat's interface is based around 4 basic types,

- `Solver` - The solver itself
- `Vec` - A relocating implementation of `std::vector`

MiniSat's interface is based around 4 basic types,

- `Solver` - The solver itself
- `Vec` - A relocating implementation of `std::vector`

and 2 vocabulary types

MiniSat's interface is based around 4 basic types,

- `Solver` - The solver itself
- `Vec` - A relocating implementation of `std::vector`

and 2 vocabulary types

- `Var` - The representation of a logic *variable*

MiniSat's interface is based around 4 basic types,

- `Solver` - The solver itself
- `Vec` - A relocating implementation of `std::vector`

and 2 vocabulary types

- `Var` - The representation of a logic *variable*
- `Lit` - The concrete *literal* of a variable in a clause

$$(A \lor B \lor \neg A) \land (A \lor B \lor \neg C)$$

$$(A \vee B \vee \neg A) \wedge (A \vee B \vee \neg C)$$

3 variables, A, B, and C

$$(A \lor B \lor \neg A) \land (A \lor B \lor \neg C)$$

3 variables, A, B, and C

4 literals, A, B, $\neg A$, and $\neg C$.

Let's solve the formula

$$(A \vee B \vee \neg A) \wedge (A \vee B \vee \neg C)$$

Let's solve the formula

$$(A \lor B \lor \neg A) \land (A \lor B \lor \neg C)$$

```
#include <minisat/core/Solver.h>
#include <iostream>

int main() {
    using Minisat::mkLit; using Minisat::lbool;

    Minisat::Solver solver;

    auto A = solver.newVar();
    auto B = solver.newVar();
    auto C = solver.newVar();

    solver.addClause( mkLit(A),  mkLit(B),  ~mkLit(A));
    solver.addClause( mkLit(A),  mkLit(B),  ~mkLit(C));
```

... and then retrieve the results

... and then retrieve the results

```
    auto sat = solver.solve();
    if (sat) {
        std::cout << "SAT\n"
            << "Model found:\n"
            << "A := " << (solver.modelValue(A) == l_True) << '\n'
            << "B := " << (solver.modelValue(B) == l_True) << '\n'
            << "C := " << (solver.modelValue(C) == l_True) << '\n';
    } else {
        std::cout << "UNSAT\n";
        return 1;
    }
}
```

So what solution did Minisat find?

So what solution did Minisat find?

```
$ ./example-1
SAT
Model found:
A := 0
B := 0
C := 0
```

Now we know enough to make a Sudoku solver.

HOW TO CONVERT SUDOKU TO SAT

Sudoku is a puzzle where you put numbers 1-9 onto a 9x9 grid, split into 9 3x3 boxes

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

Some of the numbers are prefilled and we have to fill in the rest, following some simple 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

Some of the numbers are prefilled and we have to fill in the rest, following some simple 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

1. Each row contains all of the numbers 1-9

Some of the numbers are prefilled and we have to fill in the rest, following some simple 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

1. Each row contains all of the numbers 1-9
2. Each column contains all of the numbers 1-9

5.3

Some of the numbers are prefilled and we have to fill in the rest, following some simple 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

1. Each row contains all of the numbers 1-9
2. Each column contains all of the numbers 1-9
3. Each 3x3 box contains all of the numbers 1-9

When translating these rules into SAT, we have to start by defining the variables.

When translating these rules into SAT, we have to start by defining the variables.

The natural thing to do would be to assign each position a variable that can have values 1-9.

When translating these rules into SAT, we have to start by defining the variables.

The natural thing to do would be to assign each position a variable that can have values 1-9.

In SAT, variable can have 2 values, "true", or "false".

The solution is to have a variable per each position
and each possible value.

The solution is to have a variable per each position *and* each possible value.

Let's denote these variables as $x_{r,c}^{v}$

The solution is to have a variable per each position *and* each possible value.

Let's denote these variables as $x_{r,c}^{v}$

If the variable $x_{r,c}^{v}$ is set to true, the r-th row and c-th column contains number v.

1. EACH ROW CONTAINS ALL OF THE NUMBERS 1-9

1. EACH ROW CONTAINS ALL OF THE NUMBERS 1-9

$$\forall (r, v) \in (rows \times values) : x^v_{r,1} \vee x^v_{r,2} \vee \ldots \vee x^v_{r,9}$$

1. EACH ROW CONTAINS ALL OF THE NUMBERS 1-9

$$\forall (r, v) \in (rows \times values) : x^v_{r,1} \lor x^v_{r,2} \lor \ldots \lor x^v_{r,9}$$

$$\forall (r, v) \in (rows \times values) : \bigvee_{i=1}^{9} x^v_{r,i}$$

2. EACH COL CONTAINS ALL OF THE NUMBERS 1-9

2. EACH COL CONTAINS ALL OF THE NUMBERS 1-9

$$\forall (c, v) \in (columns \times values) : x^v_{1,c} \vee x^v_{2,c} \vee \ldots \vee x^v_{9,c}$$

$$\forall (c, v) \in (columns \times values) : \bigvee_{i=1}^{9} x^v_{i,c}$$

3. EACH BOX CONTAINS ALL OF THE NUMBERS 1-9

3. EACH BOX CONTAINS ALL OF THE NUMBERS 1-9

$$\forall (b, v) \in (boxes \times values) : x^v_{br_1, bc_1} \vee x^v_{br_1, bc_2} \vee \ldots \vee x^v_{br_3, bc_3}$$

$$\forall (b, v) \in (boxes \times values) : \bigvee_{(r,c) \in b} x^v_{r,c}$$

We expressed the Sudoku rules as a set of clauses.

We expressed the Sudoku rules as a set of clauses.

But an important set of clauses is missing.

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

5.10

As humans, we assume that each position can contain only a single number.

As humans, we assume that each position can contain only a single number.

This assumption was lost when we split each position into multiple different variables.

As humans, we assume that each position can contain only a single number.

This assumption was lost when we split each position into multiple different variables.

We need to add it back.

4. EACH POSITION CONTAINS EXACTLY ONE NUMBER

4. EACH POSITION CONTAINS EXACTLY ONE NUMBER

$$\forall (r, c) \in (rows \times columns) : \text{exactly-one}(x^1_{r,c}, \ldots, x^9_{r,c})$$

4. EACH POSITION CONTAINS EXACTLY ONE NUMBER

$$\forall (r, c) \in (rows \times columns) : \text{exactly-one}(x^1_{r,c}, \ldots, x^9_{r,c})$$

The exactly-one helper adds a set of clauses that allows only one of the literals to be true.

4. EACH POSITION CONTAINS EXACTLY ONE NUMBER

$$\forall(r, c) \in (rows \times columns) : \text{exactly-one}(x^1_{r,c}, \ldots, x^9_{r,c})$$

The exactly-one helper adds a set of clauses that allows only one of the literals to be true.

Let's take a look at how it works.

We cannot directly limit the number of true literals.

We cannot directly limit the number of true literals.

But we can place lower and upper limits on them.

We cannot directly limit the number of true literals.

But we can place lower and upper limits on them.

In other words, *exactly one* literal is true when *at least* one is true **and** *at most* one is true.

Making *at least one* literal true is simple:

Making *at least one* literal true is simple:

$$\bigvee_{lit \in literals} lit$$

Forcing *at most one* literal to be true is based on a simple observation

Forcing *at most one* literal to be true is based on a simple observation

At most one literal is true when *there is no pair of literals where both literals are true at the same time.*

Forcing *at most one* literal to be true is based on a simple observation

At most one literal is true when *there is no pair of literals where both literals are true at the same time.*

$$\forall l_1 \in literals, \\ l_2 \in literals, \\ l_1 \neq l_2 : \neg\left(l_1 \wedge l_2\right)$$

Let's write us a C++ Sudoku solver.

Let's write us a C++ Sudoku solver.

All the code in this section can be found at https://github.com/horenmar/sudoku-example

First we need to figure out addressing variables.

First we need to figure out addressing variables.

SAT solvers see variables as integers in range 0..N.

First we need to figure out addressing variables.

SAT solvers see variables as integers in range 0..N.

Luckily, we can easily map $x^v_{r,c}$ into an integer as `r * 9 * 9 + c * 9 + v`

First we need to figure out addressing variables.

SAT solvers see variables as integers in range 0..N.

Luckily, we can easily map $x^v_{r,c}$ into an integer as
`r * 9 * 9 + c * 9 + v`

```
Minisat::Var toVar(int row, int column, int value) {
    return row * columns * values + column * values + value;
}
```

Before we can start adding clauses, we need to allocate all variables

Before we can start adding clauses, we need to allocate all variables

```
void Solver::init_variables() {
    for (int r = 0; r < rows; ++r) {
        for (int c = 0; c < columns; ++c) {
            for (int v = 0; v < values; ++v) {
                static_cast<void>(solver.newVar());
            }
        }
    }
}
```

1. EACH ROW CONTAINS ALL OF THE NUMBERS 1-9

1. EACH ROW CONTAINS ALL OF THE NUMBERS 1-9

1. EACH ROW CONTAINS ALL OF THE NUMBERS 1-9

```
for (int row = 0; row < rows; ++row) {





}
```

1. EACH ROW CONTAINS ALL OF THE NUMBERS 1-9

```
for (int row = 0; row < rows; ++row) {
    for (int value = 0; value < values; ++value) {





    }
}
```

1. EACH ROW CONTAINS ALL OF THE NUMBERS 1-9

```
for (int row = 0; row < rows; ++row) {
    for (int value = 0; value < values; ++value) {
        Minisat::vec<Minisat::Lit> literals;




    }
}
```

1. EACH ROW CONTAINS ALL OF THE NUMBERS 1-9

```
for (int row = 0; row < rows; ++row) {
    for (int value = 0; value < values; ++value) {
        Minisat::vec<Minisat::Lit> literals;
        for (int col = 0; col < columns; ++col) {
            literals.push(Minisat::mkLit(toVar(row, col, value)));
        }

    }
}
```

1. EACH ROW CONTAINS ALL OF THE NUMBERS 1-9

```
for (int row = 0; row < rows; ++row) {
    for (int value = 0; value < values; ++value) {
        Minisat::vec<Minisat::Lit> literals;
        for (int col = 0; col < columns; ++col) {
            literals.push(Minisat::mkLit(toVar(row, col, value)));
        }
        solver.addClause(literals);
    }
}
```

2. EACH COL CONTAINS ALL OF THE NUMBERS 1-9

2. EACH COL CONTAINS ALL OF THE NUMBERS 1-9

```
for (int col = 0; col < columns; ++col) {
    for (int value = 0; value < values; ++value) {
        Minisat::vec<Minisat::Lit> literals;
        for (int row = 0; row < rows; ++row) {
            literals.push(Minisat::mkLit(toVar(row, col, value)));
        }
        solver.addClause(literals);
    }
}
```

3. EACH BOX CONTAINS ALL OF THE NUMBERS 1-9

3. EACH BOX CONTAINS ALL OF THE NUMBERS 1-9

```
for (int value = 0; value < values; ++value) {
    for (int r : {0, 3, 6}) {
        for (int c : {0, 3, 6}) {
            Minisat::vec<Minisat::Lit> literals;
            for (int rr : {0, 1, 2}) {
                for (int cc : {0, 1, 2}) {
                    literals.push(Minisat::mkLit(
                        toVar(r + rr, c + cc, value)
                    ));
                }
            }
            solver.addClause(literals);
        }
    }
}
```

4. EACH POSITION CONTAINS EXACTLY ONE NUMBER

4. EACH POSITION CONTAINS EXACTLY ONE NUMBER

```
for (int row = 0; row < rows; ++row) {
    for (int col = 0; col < columns; ++col) {
        Minisat::vec<Minisat::Lit> literals;
        for (int value = 0; value < values; ++value) {
            literals.push(Minisat::mkLit(toVar(row, col, value)));
        }
        exactly_one(literals);
    }
}
```

```cpp
void Solver::exactly_one(Minisat::vec<Minisat::Lit> const& lits) {
    // At least one
    solver.addClause(lits);

    // At most one
    for (size_t i = 0; i < lits.size(); ++i) {
        for (size_t j = i + 1; j < lits.size(); ++j) {
            solver.addClause(~lits[i], ~lits[j]);
        }
    }
}
```

We have a model of Sudoku as a SAT instance.

We have a model of Sudoku as a SAT instance.

Now we need to insert an actual instance of the puzzle, and then extract the solution.

Inserting an instance is easy enough, each prefiled square gets an unary clause:

Inserting an instance is easy enough, each prefiled square gets an unary clause:

```
bool Solver::apply_board(board const& b) {
    for (int row = 0; row < rows; ++row) {
        for (int col = 0; col < columns; ++col) {
            auto value = b[row][col];
            if (value != 0) {
                solver.addClause(
                    Minisat::mkLit(toVar(row, col, value - 1))
                );
            }
        }
    }
    return ret;
}
```

Extracting a solution is similarly simple, we just need to check which variable for a given square is "true".

Extracting a solution is similarly simple, we just need to check which variable for a given square is "true".

```
board Solver::get_solution() const {
   board b(rows, std::vector<int>(columns));
   for (int row = 0; row < rows; ++row) {
      for (int col = 0; col < columns; ++col) {
         for (int val = 0; val < values; ++val) {
            if (solver.modelValue(toVar(row, col, val)).isTrue()) {
               b[row][col] = val + 1;
               break;
            }
         }
      }
   }
   return b;
}
```

Let's take a look at how our solver performs.

Let's take a look at how our solver performs.

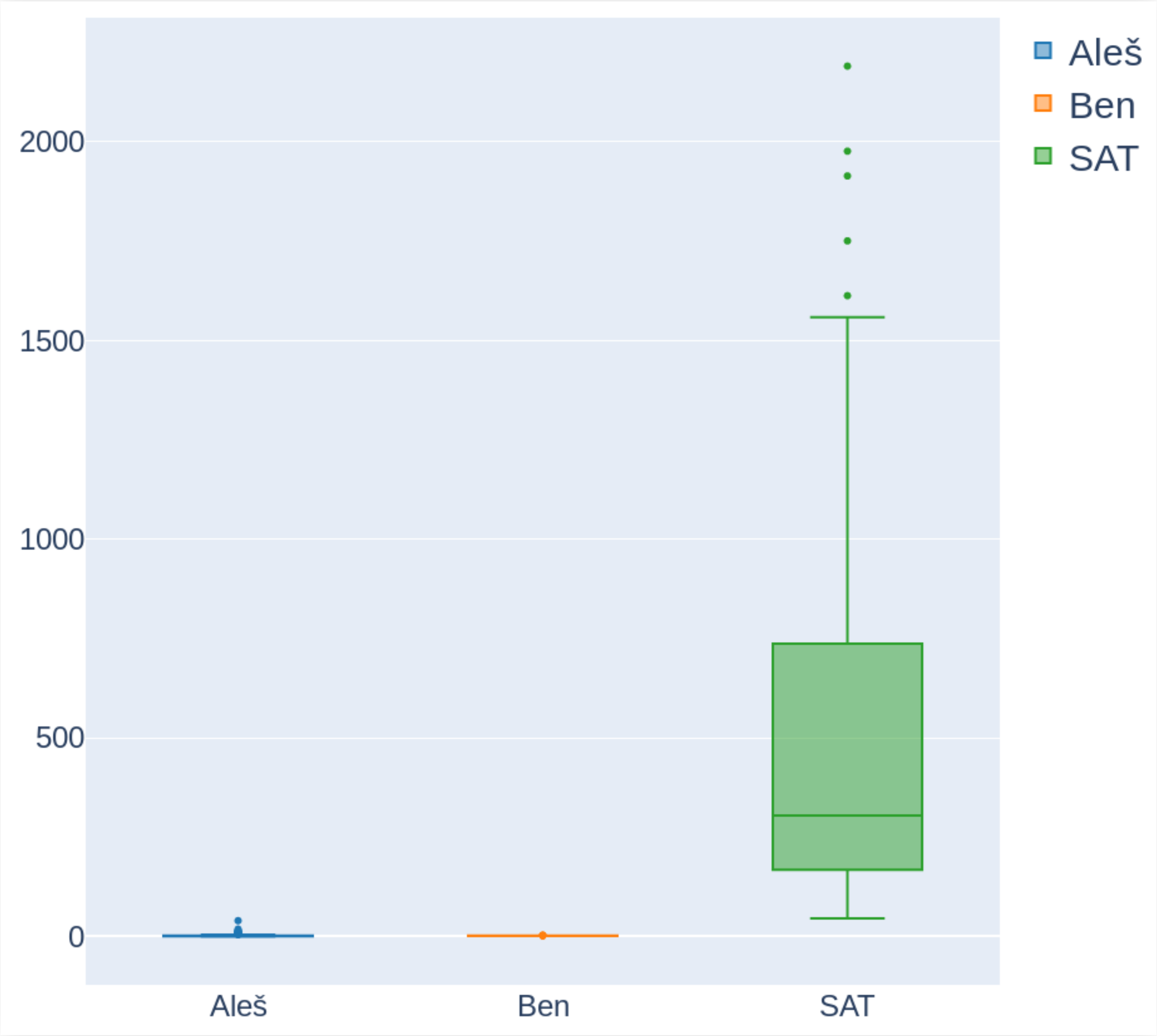
All benchmarks were run on the same machine, and the binaries were compiled with g++ under WSL.

Let's take a look at how our solver performs.

All benchmarks were run on the same machine, and the binaries were compiled with g++ under WSL.

The inputs were 95 "hard" instances of Sudoku.

Runtimes of different solvers [ms]



Counterintuitively, giving a SAT solver less clauses and/or variables can slow it down.

Counterintuitively, giving a SAT solver less clauses and/or variables can slow it down.

Let's see what happens when we encode the sudoku rules differently, and give the solver more information.

1. Each row contains each of the numbers 1-9

1. Each row contains each of the numbers 1-9 **exactly once**

1. Each row contains each of the numbers 1-9 **exactly once**
2. Each column contains each of the numbers 1-9

1. Each row contains each of the numbers 1-9 **exactly once**
2. Each column contains each of the numbers 1-9 **exactly once**

1. Each row contains each of the numbers 1-9 **exactly once**
2. Each column contains each of the numbers 1-9 **exactly once**
3. Each 3x3 box contains each of the numbers 1-9

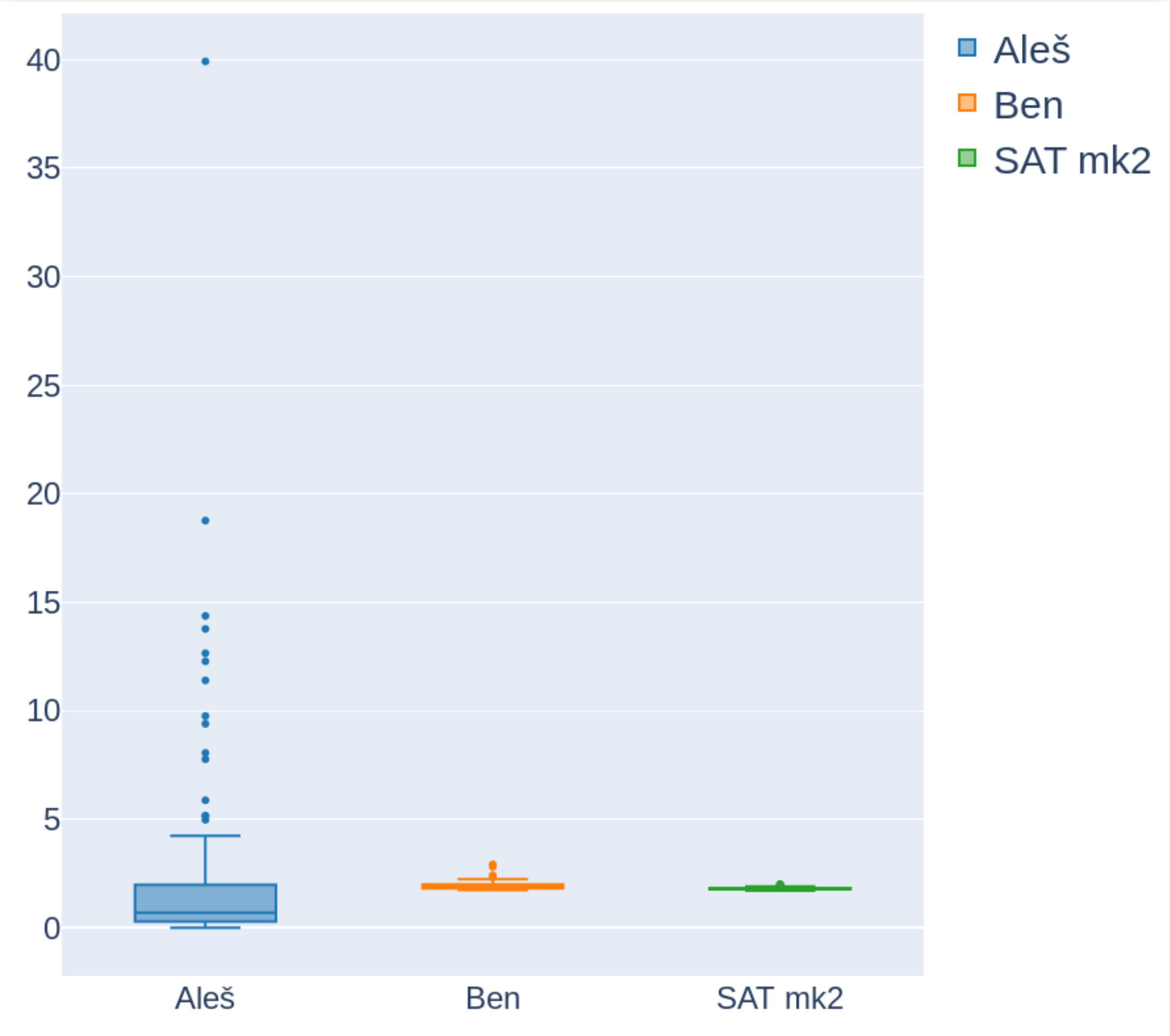
1. Each row contains each of the numbers 1-9 **exactly once**
2. Each column contains each of the numbers 1-9 **exactly once**
3. Each 3x3 box contains each of the numbers 1-9 **exactly once**

```cpp
for (int row = 0; row < rows; ++row) {
    for (int value = 0; value < values; ++value) {
        Minisat::vec<Minisat::Lit> literals;
        for (int col = 0; col < columns; ++col) {
            literals.push(Minisat::mkLit(toVar(row, col, value)));
        }
        solver.addClause(literals);

    }
}
```

```
for (int row = 0; row < rows; ++row) {
    for (int value = 0; value < values; ++value) {
        Minisat::vec<Minisat::Lit> literals;
        for (int col = 0; col < columns; ++col) {
            literals.push(Minisat::mkLit(toVar(row, col, value)));
        }
        solver.addClause(literals);
        exactly_one(literals);
    }
}
```

Runtimes of different solvers [ms]



RECAP

RECAP

Be careful to encode *all* of your assumptions.

RECAP

Be careful to encode *all* of your assumptions.

More clauses does not mean worse performance.

RECAP

Be careful to encode *all* of your assumptions.

More clauses does not mean worse performance.

But it does not mean better performance either.

RECAP

Be careful to encode *all* of your assumptions.

More clauses does not mean worse performance.

But it does not mean better performance either.

Experiment with different encodings.

HOW TO SOLVE A MASTER KEY SYSTEM WITH A SAT SOLVER

Master Key System (MKS) is a set of keys and locks where a key can open more than one lock.

Master Key System (MKS) is a set of keys and locks where a key can open more than one lock.

The relations between keys and locks can be arbitrarily complex, and are described in a lockchart.

The common depiction of a lockchart is a simple table:

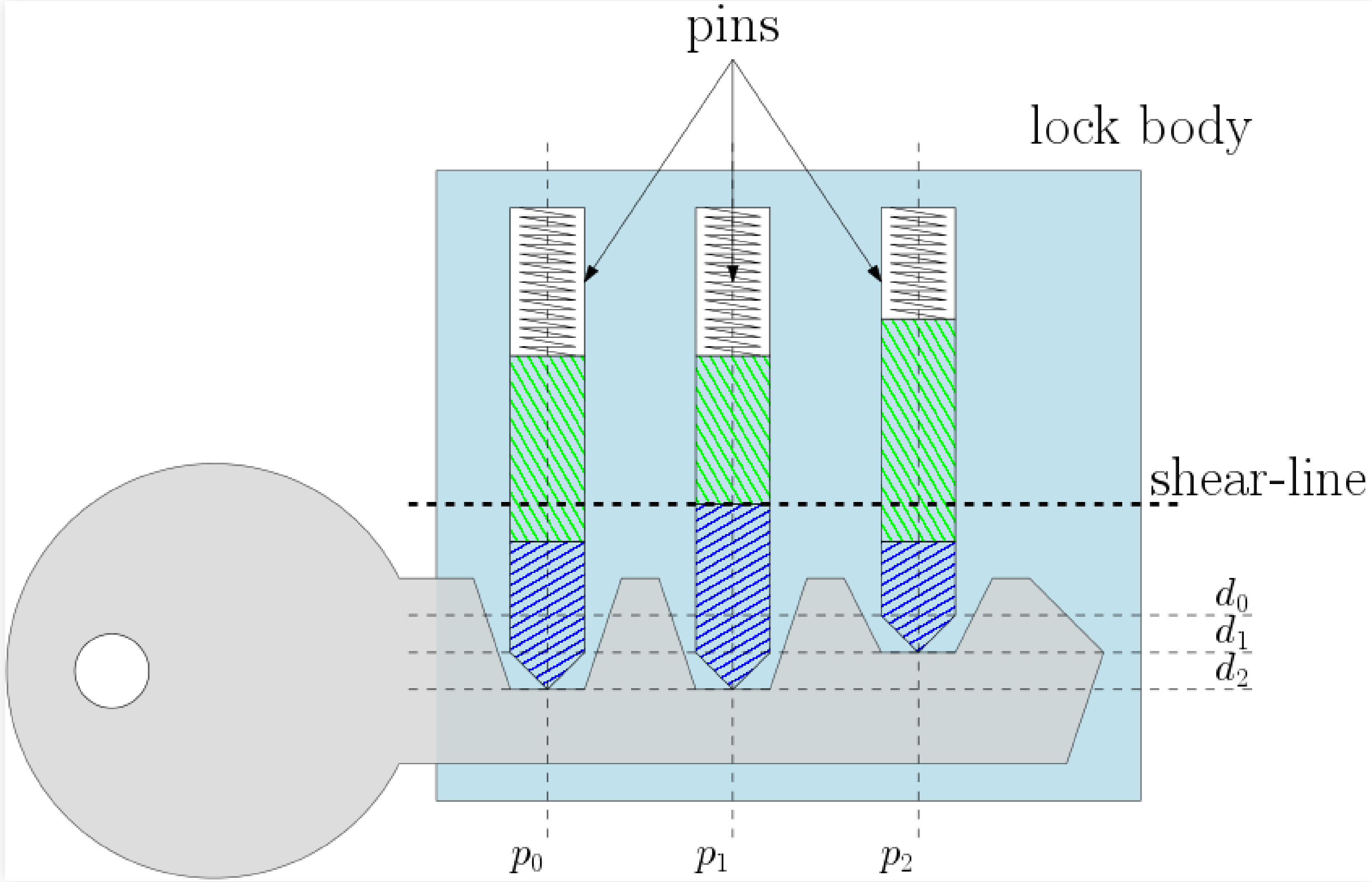
	G	M_1	M_2	K_1	K_2	K_3	K_4	K_5	K_6	K_7	K_8
L_1	■	■		■							
L_2	■	■			■						
L_3	■	■				■					
L_4	■	■					■				
L_5	■		■					■			
L_6	■		■						■		
L_7	■		■							■	
L_8	■		■								■
GL	■	■	■	■	■	■	■	■	■	■	■

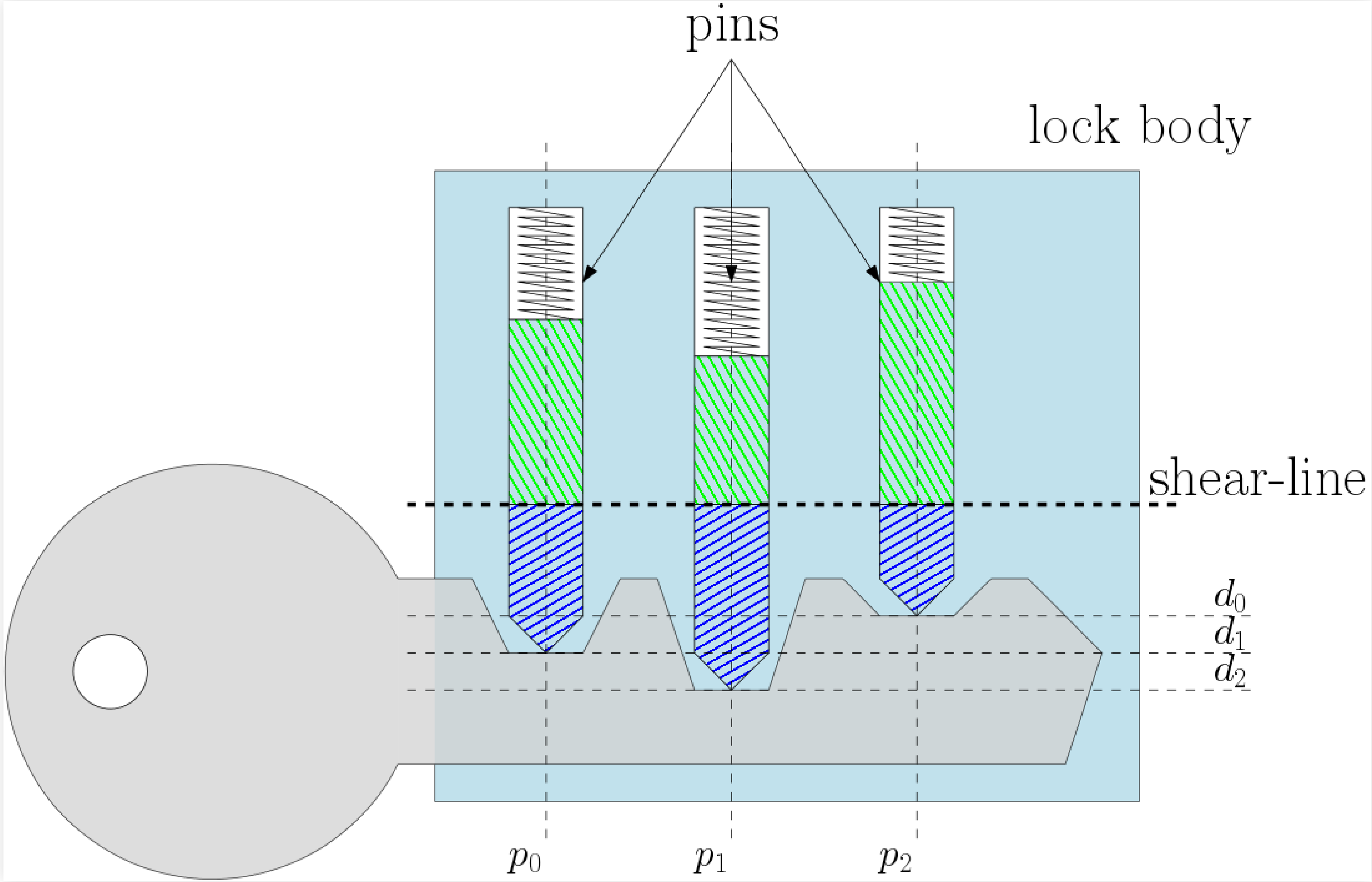
Each MKS has a geometry associated with it.

Each MKS has a geometry associated with it.

The geometry describes the positions, their depths, and the constraints the keys must satisfy.

Let's take a look at the geometry and inner working of a *pin tumbler* lock.




pins
lock body
shear-line
d_0
d_1
d_2
p_0
p_1
p_2

THE RULES

6.7

THE RULES

1. A key has exactly one cutting depth at a position

THE RULES

1. A key has exactly one cutting depth at a position
2. A lock has at least one cutting depth at a position

THE RULES

1. A key has exactly one cutting depth at a position
2. A lock has at least one cutting depth at a position
3. A key must open all locks that the lock-chart specifies it should open

THE RULES

1. A key has exactly one cutting depth at a position
2. A lock has at least one cutting depth at a position
3. A key must open all locks that the lock-chart specifies it should open
4. A key must be blocked in all locks that the lock-chart specifies it should not open

THE RULES

1. A key has exactly one cutting depth at a position
2. A lock has at least one cutting depth at a position
3. A key must open all locks that the lock-chart specifies it should open
4. A key must be blocked in all locks that the lock-chart specifies it should not open
5. A key's cutting must satisfy all constraints

THE VARIABLES

This time we will be using multiple kinds of variables.

THE VARIABLES

This time we will be using multiple kinds of variables.

$key^k_{p,d}$, which is true when key k is cut at depth d in position p

THE VARIABLES

This time we will be using multiple kinds of variables.

$key^k_{p,d}$, which is true when key k is cut at depth d in position p

$lock^l_{p,d}$, which is true when lock l is cut at depth d in position p

1. A KEY HAS EXACTLY ONE CUTTING DEPTH AT A POSITION

1. A KEY HAS EXACTLY ONE CUTTING DEPTH AT A POSITION

$$\forall (k, p) \in (keys \times positions) : \\ \text{exactly-one}(key^k_{p,0}, key^k_{p,1}, \ldots, key^k_{p,d})$$

2. A LOCK MUST HAVE AT LEAST ONE CUTTING DEPTH SELECTED FOR EACH POSITION

2. A LOCK MUST HAVE AT LEAST ONE CUTTING DEPTH SELECTED FOR EACH POSITION

$$\forall (l, p) \in (locks \times positions) : \bigvee_{d \in depths(p)} lock^{l}_{p,d}$$

3. A KEY MUST OPEN ALL LOCKS THAT THE LOCK-CHART SPECIFIES IT SHOULD OPEN

3. A KEY MUST OPEN ALL LOCKS THAT THE LOCK-CHART SPECIFIES IT SHOULD OPEN

A key opens a lock when the lock has the same cutting depths as the key.

3. A KEY MUST OPEN ALL LOCKS THAT THE LOCK-CHART SPECIFIES IT SHOULD OPEN

A key opens a lock when the lock has the same cutting depths as the key.

$$\forall k \in keys,$$
$$\forall l \in \text{opened-by}(k) : \bigwedge_{\substack{p \in positions \\ d \in \text{depths}(p)}} \left(key^{k}_{p,d} \implies lock^{l}_{p,d} \right)$$

4. A KEY MUST BE BLOCKED IN ALL LOCKS THAT THE LOCK-CHART SPECIFIES IT SHOULD NOT OPEN

4. A KEY MUST BE BLOCKED IN ALL LOCKS THAT THE LOCK-CHART SPECIFIES IT SHOULD NOT OPEN

A key is blocked in a lock if, and only if, it does not open the lock.

4. A KEY MUST BE BLOCKED IN ALL LOCKS THAT THE LOCK-CHART SPECIFIES IT SHOULD NOT OPEN

A key is blocked in a lock if, and only if, it does not open the lock.

A key opens a lock when the lock has the same cutting depths as the key.

A key is blocked in a lock when the lock is missing at least one of key's cutting depths.

A key is blocked in a lock when the lock is missing at least one of key's cutting depths.

$$\forall k \in keys,$$
$$\forall l \in \text{blocked-in}(k) : \bigvee_{\substack{p \in positions \\ d \in \text{depths}(p)}} \left(key^k_{p,d} \land \neg\, lock^l_{p,d} \right)$$

We can convert DNF to CNF using distributive laws.

We can convert DNF to CNF using distributive laws.

This creates an exponential number of long clauses.

We can convert DNF to CNF using distributive laws.

This creates an exponential number of long clauses.

We can do better.

At the start of this talk, we talked about converting formulae into equivalent formulae in CNF.

At the start of this talk, we talked about converting formulae into equivalent formulae in CNF.

It is also possible to create *equisatisfiable* formulae.

$\alpha \vee \beta$ and $(\alpha \vee \neg\gamma) \wedge (\gamma \vee \beta)$ are not equivalent,
but they are equisatisfiable.

We will use trick called *Tseytin transformation*.

We will use trick called *Tseytin transformation*.

The idea is to introduce a new variable to represent each inner conjunction, and then disjunct those.

We will use trick called *Tseytin transformation*.

The idea is to introduce a new variable to represent each inner conjunction, and then disjunct those.

Let's call this variable $block_{p,d}^{k,l}$ and define it as

$$block_{p,d}^{k,l} \iff (key_{p,d}^{k} \land \neg lock_{p,d}^{l}).$$

$$block^{k,l}_{p,d} \iff \left(key^{k}_{p,d} \wedge \neg lock^{l}_{p,d}\right)$$

$$block^{k,l}_{p,d} \iff \left(key^k_{p,d} \land \neg lock^l_{p,d}\right)$$

$$\alpha \iff \beta$$

$$block^{k,l}_{p,d} \iff \left(key^{k}_{p,d} \land \neg lock^{l}_{p,d}\right)$$

$$\alpha \iff \beta$$

$$(\alpha \implies \beta) \land (\alpha \impliedby \beta)$$

$$block^{k,l}_{p,d} \iff \left(key^{k}_{p,d} \land \neg lock^{l}_{p,d}\right)$$

$$\alpha \iff \beta$$

$$(\alpha \implies \beta) \land (\alpha \impliedby \beta)$$

$$(\neg\alpha \lor \beta) \land (\alpha \lor \neg\beta)$$

$$block^{k,l}_{p,d} \iff \left(key^{k}_{p,d} \land \neg lock^{l}_{p,d}\right)$$

$$\alpha \iff \beta$$

$$(\alpha \implies \beta) \land (\alpha \impliedby \beta)$$

$$(\neg\alpha \lor \beta) \land (\alpha \lor \neg\beta)$$

The first clause can be distributed out:

The first clause can be distributed out:

$$\left(\neg block^{k,l}_{p,d} \lor \left(key^{k}_{p,d} \land \neg lock^{l}_{p,d}\right)\right)$$

The first clause can be distributed out:

$$\left(\neg block^{k,l}_{p,d} \vee \left(key^{k}_{p,d} \wedge \neg lock^{l}_{p,d}\right)\right)$$

$$\left(\neg block^{k,l}_{p,d} \vee key^{k}_{p,d}\right) \wedge \left(\neg block^{k,l}_{p,d} \vee \neg lock^{l}_{p,d}\right)$$

Second clause is simplified with DeMorgan's laws

Second clause is simplified with DeMorgan's laws

$$\left(block_{p,d}^{k,l} \vee \neg \left(key_{p,d}^{k} \wedge lock_{p,d}^{l}\right)\right)$$

Second clause is simplified with DeMorgan's laws

$$\left(block^{k,l}_{p,d} \vee \neg\left(key^{k}_{p,d} \wedge lock^{l}_{p,d}\right)\right)$$

$$\left(block^{k,l}_{p,d} \vee \neg key^{k}_{p,d} \vee \neg lock^{l}_{p,d}\right)$$

With this, the blocking clauses are simple:

With this, the blocking clauses are simple:

$$\forall k \in keys,$$
$$\forall l \in \text{blocked-in}(k) : \bigvee_{\substack{p \in positions \\ d \in \text{depths}(p)}} block_{p,d}^{k,l}$$

With this, the blocking clauses are simple:

$$\forall k \in \mathit{keys},$$
$$\forall l \in \text{blocked-in}(k) : \bigvee_{\substack{p \in \mathit{positions} \\ d \in \text{depths}(p)}} \mathit{block}^{k,l}_{p,d}$$

We now have a model of a MKS without constraints.

We will explore 2 different constraints:

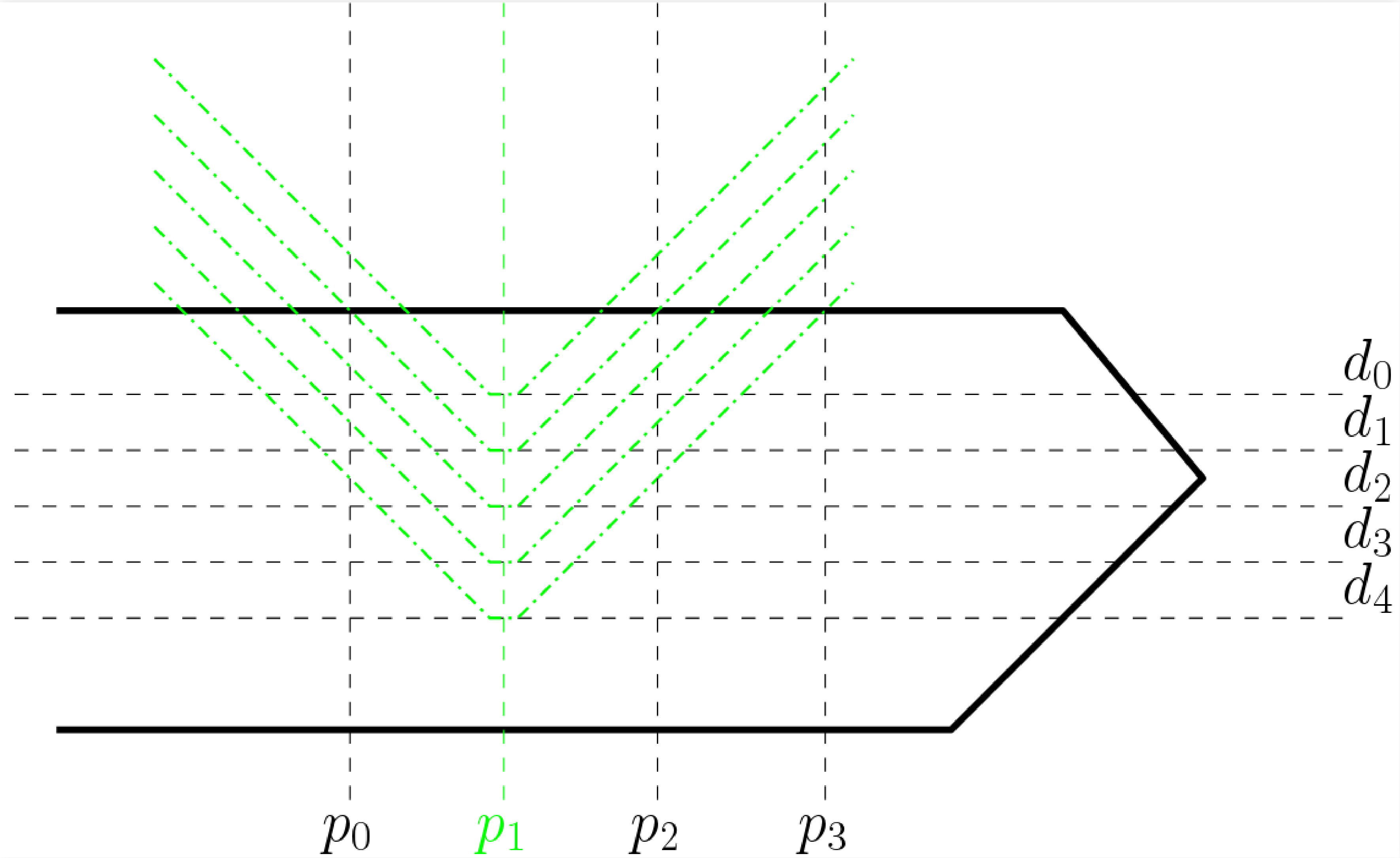
We will explore 2 different constraints:

- *Jump* constraint

We will explore 2 different constraints:

- *Jump* constraint
- *Key cutting hierarchy* constraint

JUMP CONSTRAINT



To manufacture a key, the cutting depths in two adjacent positions cannot differ by more than j.

To manufacture a key, the cutting depths in two adjacent positions cannot differ by more than j.

KEY CUTTING HIERARCHY CONSTRAINT

KEY CUTTING HIERARCHY CONSTRAINT

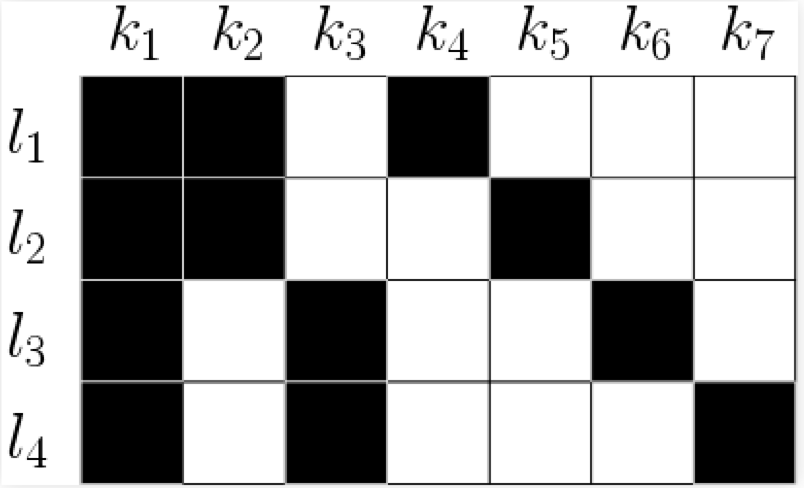
For all keys k_1, k_2, where

$$\text{opened-by}(k_1) \supset \text{opened-by}(k_2), \ldots$$

KEY CUTTING HIERARCHY CONSTRAINT

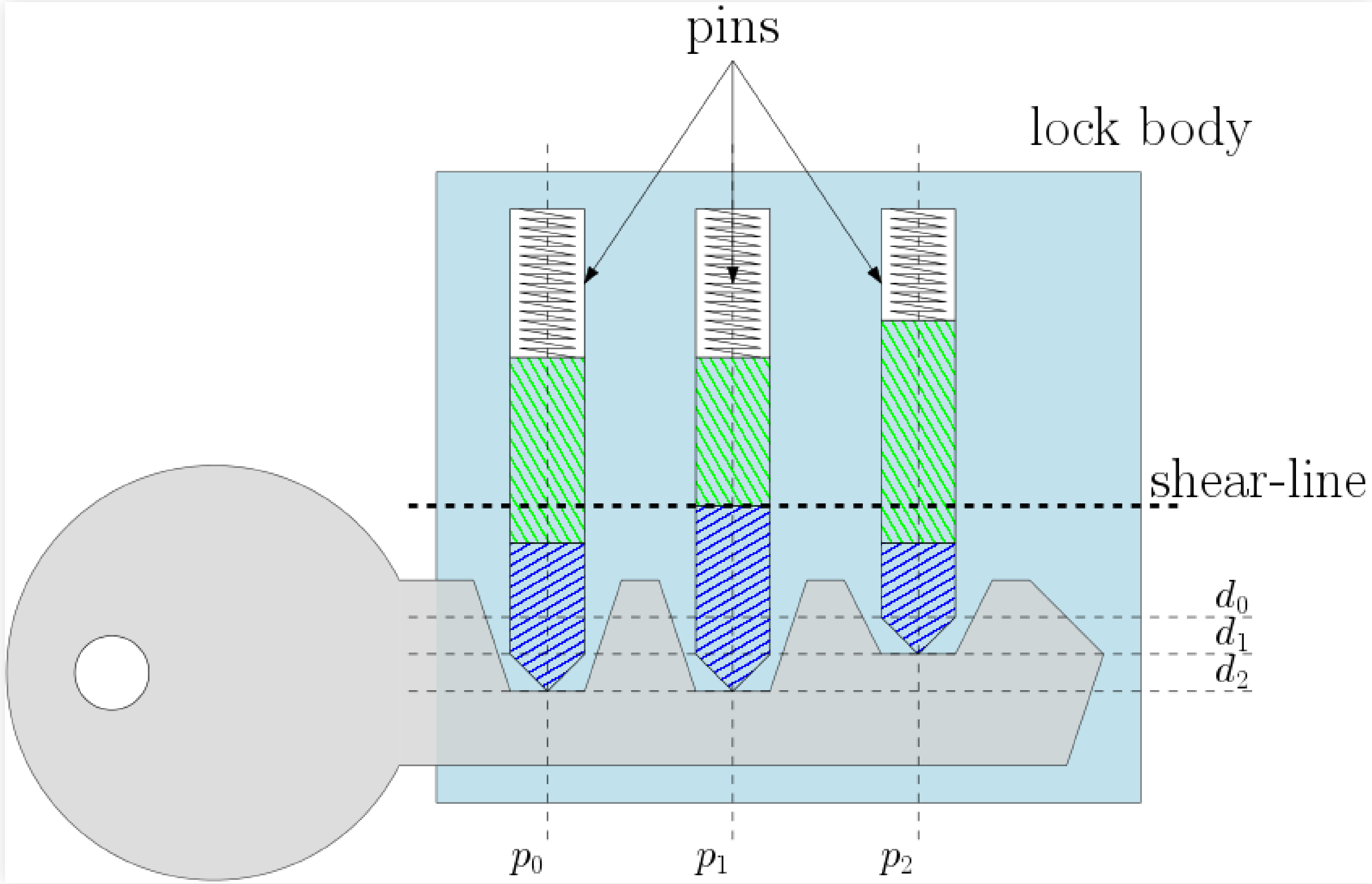
For all keys k_1, k_2, where

$$\text{opened-by}(k_1) \supset \text{opened-by}(k_2), \ldots$$



... for all positions p and depths d_i, d_j, where $i > j$

... for all positions p and depths d_i, d_j, where $i > j$



For all keys k_1, k_2, where $\text{opened-by}(k_1) \supset \text{opened-by}(k_2)$, for all positions p and depths d_i, d_j, where $i > j$:

For all keys k_1, k_2, where $\text{opened-by}(k_1) \supset \text{opened-by}(k_2)$, for all positions p and depths d_i, d_j, where $i > j$:

$$key^{k_1}_{p,d_i} \implies \neg key^{k_2}_{p,d_j}$$

C++ IMPLEMENTATION

C++ IMPLEMENTATION

A variant of the code is implemented in https://github.com/horenmar/mks-example

C++ IMPLEMENTATION

A variant of the code is implemented in
https://github.com/horenmar/mks-example

We will not go over it in this talk.

BENCHMARKS

BENCHMARKS

There are none, sorry.

BENCHMARKS

There are none, sorry.

I do have an anecdote though.

~5 years ago, my university was approached by our local key manufacturer, FAB.

~5 years ago, my university was approached by our local key manufacturer, FAB.

A team spent ~3 years working on a specialized solver.

~5 years ago, my university was approached by our local key manufacturer, FAB.

A team spent ~3 years working on a specialized solver.

It could solve ~80% of test inputs.

Another researcher had the idea to use SAT solvers.

Another researcher had the idea to use SAT solvers.

In 3 months, his solver could solve ~90% of the tests.

Another researcher had the idea to use SAT solvers.

In 3 months, his solver could solve ~90% of the tests.

Its later evolution is currently in production.

RECAP

RECAP

SAT solvers can be used to solve wildly different problems.

RECAP

SAT solvers can be used to solve wildly different problems.

Writing a SAT-based solver is fast and easy.

RECAP

SAT solvers can be used to solve wildly different problems.

Writing a SAT-based solver is fast and easy.

SAT-based solvers can perform surprisingly well.

RECAP

SAT solvers can be used to solve wildly different problems.

Writing a SAT-based solver is fast and easy.

SAT-based solvers can perform surprisingly well.

But specialized solvers will end up faster.

You need to be careful to encode your assumptions when converting a problem to SAT.

You need to be careful to encode your assumptions when converting a problem to SAT.

Adding clauses can speed things up, and so can shortening your clauses.

You need to be careful to encode your assumptions when converting a problem to SAT.

Adding clauses can speed things up, and so can shortening your clauses.

But there are no guarantees.

Experiment with different encodings of the problem.

Experiment with different encodings of the problem.

Experiment with different SAT solvers!

THE END.

QUESTIONS?

- https://github.com/horenmar/sudoku-example
- https://github.com/horenmar/mks-example
- https://github.com/master-keying/minisat
- https://codingnest.com/modern-sat-solvers-fast-neat-and-underused-part-3-of-n/