Redundant Inequalities in Sudoku and Latin Squares

Bart Demoen & María García de la Banda
K.U.Leuven, Belgium and Monash University, Melbourne, Australia

27 June 2012

- Context
- Motivation
- Sudoku
- Latin Squares

- Programming paradigm where given a problem:
  - We first model it
  - And then solve it
- A model is specified by a
  - Set of variables
  - Set of domain constraints (possible values for each variable)
  - Set of (other) constraints
  - Optional: objective function
- Example: Model for Latin Square of size 3
  - 9 variables $x_{ij}, i, j \in 1..3$
  - Each with a domain constraint $x_{ij} \in 1..3$
  - 6 *all_different* constraints:
    - 1 per row (e.g., $all\_different(\{x_{11}, x_{12}, x_{13}\})$)
    - 1 per column (e.g., $all\_different(\{x_{11}, x_{21}, x_{31}\})$)

- Programming paradigm where given a problem:
  - We first model it
  - And then solve it
- A model is specified by a
  - Set of variables
  - Set of domain constraints (possible values for each variable)
  - Set of (other) constraints
  - Optional: objective function
- Example: Model for Latin Square of size 3
  - 9 variables $x_{ij}, i, j \in 1..3$
  - Each with a domain constraint $x_{ij} \in 1..3$
  - 6 *all_different* constraints:
    - 1 per row (e.g., $all\_different(\{x_{11}, x_{12}, x_{13}\})$)
    - 1 per column (e.g., $all\_different(\{x_{11}, x_{21}, x_{31}\})$)

- Programming paradigm where given a problem:
  - We first model it
  - And then solve it
- A model is specified by a
  - Set of variables
  - Set of domain constraints (possible values for each variable)
  - Set of (other) constraints
  - Optional: objective function
- Example: Model for Latin Square of size 3
  - 9 variables $x_{ij}, i, j \in 1..3$
  - Each with a domain constraint $x_{ij} \in 1..3$
  - 6 *all_different* constraints:
    - 1 per row (e.g., $\mathit{all\_different}(\{x_{11}, x_{12}, x_{13}\})$)
    - 1 per column (e.g., $\mathit{all\_different}(\{x_{11}, x_{21}, x_{31}\})$)

X_{11}	X_{12}	X_{13}
X_{21}	X_{22}	X_{23}
X_{31}	X_{32}	X_{33}

- Reasoning about constraint redundancy is useful
- By reasoning I mean determining whether:
  - Constraint c is redundant for set of constraints C
  - Set of constraints C have redundant constraints
- Useful because in Constraint Programming:
  - Adding redundant constraints can speed up the search
  - But adding too many can also slow it down
- Redundancy of equality is easy(er)
  - $X = Y$ & $Y = Z \rightarrow X = Z$ is correct
- Redundancy of inequality is more difficult
  - Is $X \neq Y$ & $Y \neq Z \rightarrow X \neq Z$ correct?
  - Is $X \neq Y$ & $Y \neq Z \rightarrow X = Z$?
  - Is $X \neq Y$ & $Y \neq Z$ & $Z \neq T \rightarrow X \neq T$?

- Reasoning about constraint redundancy is useful
- By reasoning I mean determining whether:
  - Constraint c is redundant for set of constraints C
  - Set of constraints C have redundant constraints
- Useful because in Constraint Programming:
  - Adding redundant constraints can speed up the search
  - But adding too many can also slow it down
- Redundancy of equality is easy(er)
  - $X = Y$ & $Y = Z \rightarrow X = Z$ is correct
- Redundancy of inequality is more difficult
  - Is $X \neq Y$ & $Y \neq Z \rightarrow X \neq Z$ correct?
  - Is $X \neq Y$ & $Y \neq Z \rightarrow X = Z$?
  - Is $X \neq Y$ & $Y \neq Z$ & $Z \neq T \rightarrow X \neq T$?

- Reasoning about constraint redundancy is useful
- By reasoning I mean determining whether:
  - Constraint c is redundant for set of constraints C
  - Set of constraints C have redundant constraints
- Useful because in Constraint Programming:
  - Adding redundant constraints can speed up the search
  - But adding too many can also slow it down
- Redundancy of equality is easy(er)
  - $X = Y \ \& \ Y = Z \rightarrow X = Z$ is correct
- Redundancy of inequality is more difficult
  - Is $X \neq Y \ \& \ Y \neq Z \rightarrow X \neq Z$ correct?
  - Is $X \neq Y \ \& \ Y \neq Z \rightarrow X = Z$?
  - Is $X \neq Y \ \& \ Y \neq Z \ \& \ Z \neq T \rightarrow X \neq T$?

- Reasoning about constraint redundancy is useful
- By reasoning I mean determining whether:
  - Constraint c is redundant for set of constraints C
  - Set of constraints C have redundant constraints
- Useful because in Constraint Programming:
  - Adding redundant constraints can speed up the search
  - But adding too many can also slow it down
- Redundancy of equality is easy(er)
  - $X = Y \ \& \ Y = Z \rightarrow X = Z$ is correct
- Redundancy of inequality is more difficult
  - Is $X \neq Y \ \& \ Y \neq Z \rightarrow X \neq Z$ correct?
  - Is $X \neq Y \ \& \ Y \neq Z \rightarrow X = Z$?
  - Is $X \neq Y \ \& \ Y \neq Z \ \& \ Z \neq T \rightarrow X \neq T$?

- Reasoning about constraint redundancy is useful
- By reasoning I mean determining whether:
  - Constraint c is redundant for set of constraints C
  - Set of constraints C have redundant constraints
- Useful because in Constraint Programming:
  - Adding redundant constraints can speed up the search
  - But adding too many can also slow it down
- Redundancy of equality is easy(er)
  - $X = Y$ & $Y = Z \rightarrow X = Z$ is correct
- Redundancy of inequality is more difficult
  - Is $X \neq Y$ & $Y \neq Z \rightarrow X \neq Z$ correct?
  - Is $X \neq Y$ & $Y \neq Z \rightarrow X = Z$?
  - Is $X \neq Y$ & $Y \neq Z$ & $Z \neq T \rightarrow X \neq T$?

- Reasoning about constraint redundancy is useful
- By reasoning I mean determining whether:
  - Constraint c is redundant for set of constraints C
  - Set of constraints C have redundant constraints
- Useful because in Constraint Programming:
  - Adding redundant constraints can speed up the search
  - But adding too many can also slow it down
- Redundancy of equality is easy(er)
  - $X = Y$ & $Y = Z \rightarrow X = Z$ is correct
- Redundancy of inequality is more difficult
  - Is $X \neq Y$ & $Y \neq Z \rightarrow X \neq Z$ correct?
  - Is $X \neq Y$ & $Y \neq Z \rightarrow X = Z$?
  - Is $X \neq Y$ & $Y \neq Z$ & $Z \neq T \rightarrow X \neq T$?

- Many problems have inequality constraints
- We wanted to learn about redundancy of inequalities using Sudoku
- This drove us to Latin Squares

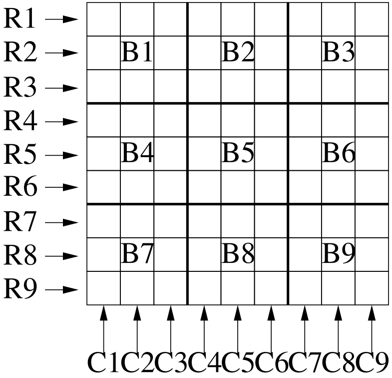
- Many problems have inequality constraints
- We wanted to learn about redundancy of inequalities using Sudoku
- This drove us to Latin Squares

- Many problems have inequality constraints
- We wanted to learn about redundancy of inequalities using Sudoku
- This drove us to Latin Squares

- Modeled using:
  - 81 variables x_{ij} $i, j \in 1..9$
  - 81 domain constraints: x_{ij} is [1..9]
  - 27 *all_different* constraints with 9 variables each
    - one per row, one per column, and one per box
- We call the *all_different* the **BIG** constraints
  - We will see later why
- We let *Sudoku* denote the above problem
  - And any equivalent specification (same solutions)
- Questions:
  - Are any of the BIG constraints redundant?
  - Further: how many can we remove and still be *Sudoku*?
  - Note: the 81 domain constraints are always there

- Modeled using:
  - 81 variables x_{ij} $i, j \in 1..9$
  - 81 domain constraints: x_{ij} is [1..9]
  - 27 *all_different* constraints with 9 variables each
    - one per row, one per column, and one per box
- We call the *all_different* the **BIG** constraints
  - We will see later why
- We let *Sudoku* denote the above problem
  - And any equivalent specification (same solutions)
- Questions:
  - Are any of the BIG constraints redundant?
  - Further: how many can we remove and still be *Sudoku*?
  - Note: the 81 domain constraints are always there

- Modeled using:
  - 81 variables x_{ij} $i, j \in 1..9$
  - 81 domain constraints: x_{ij} is [1..9]
  - 27 *all_different* constraints with 9 variables each
    - one per row, one per column, and one per box
- We call the *all_different* the **BIG** constraints
  - We will see later why
- We let *Sudoku* denote the above problem
  - And any equivalent specification (same solutions)
- Questions:
  - Are any of the BIG constraints redundant?
  - Further: how many can we remove and still be *Sudoku*?
  - Note: the 81 domain constraints are always there

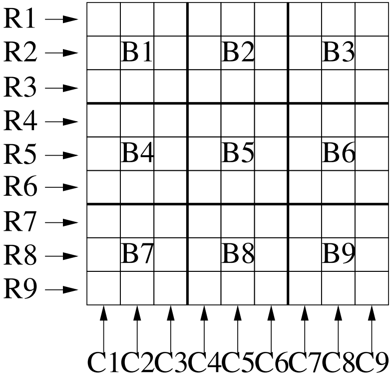


$$\begin{array}{llll} X_{11} & X_{12} & X_{13} & \dots \\ X_{21} & X_{22} & \dots & \\ . & & & \\ . & & & \\ . & & & \\ . & & & \\ . & & & \\ . & & & \\ X_{91} & X_{92} & \dots. & \end{array}$$

CHUTE = 3 boxes horizontally or vertically

Set notation is not clear with 27 elements...

... so we will use pictures for our theroems

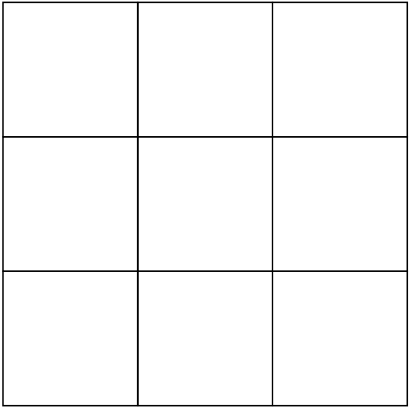


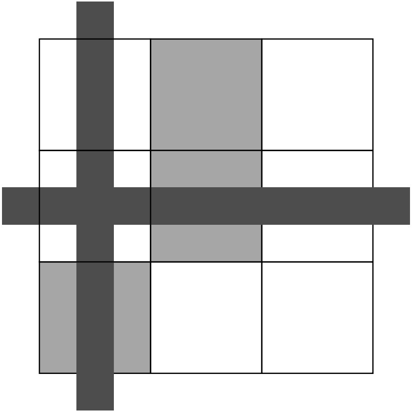
X_{11} X_{12} X_{13} ...
X_{21} X_{22} ...
.
.
.
.
.
.
X_{91} X_{92} ....

CHUTE = 3 boxes horizontally or vertically

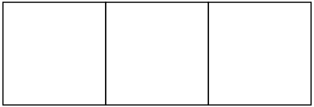
Set notation is not clear with 27 elements...

... so we will use pictures for our theroems



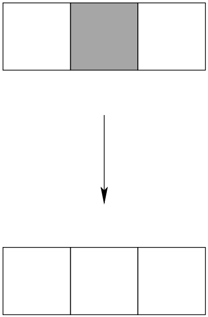


It misses R_5, C_2, B_2, B_5 and B_7



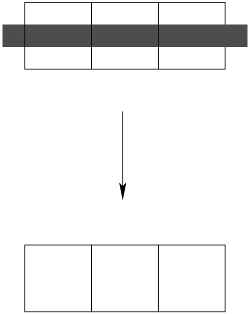
Questions about the pictures ?

If not ... we are ready for the constructive lemmas



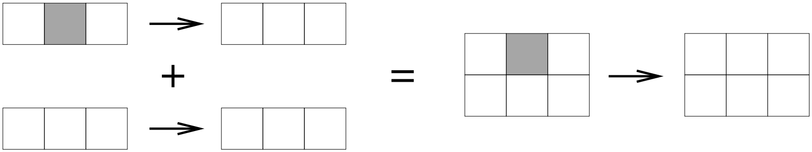
Proof by positioning any $N \in 1..9$ in the chute:

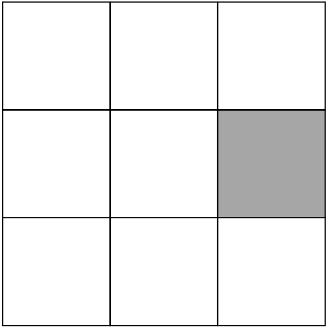
- There must be one N in each row and outer box
- This leaves one in the inner box



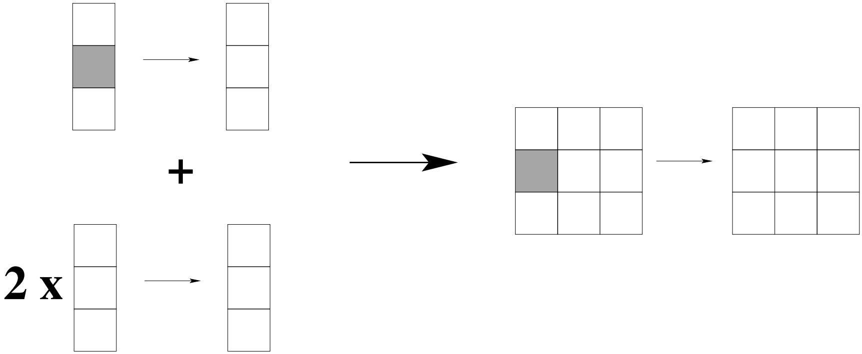
Similar proof

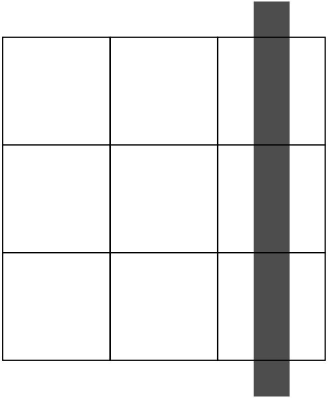
Lemma 1 and 2 can be composed like Lego bricks



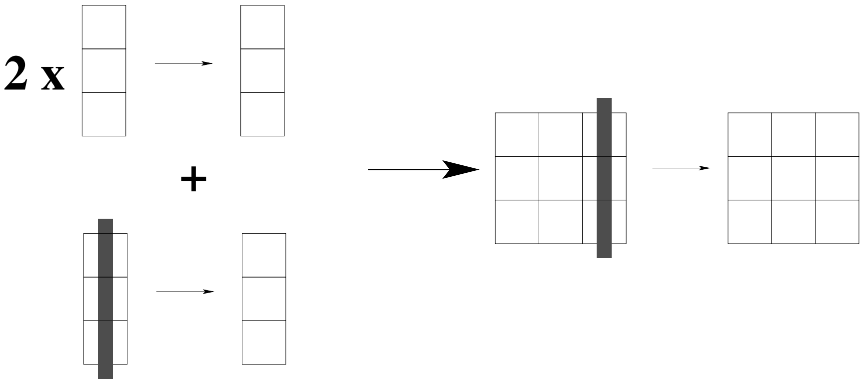


is Sudoku.


2 x
+



is Sudoku.


2 x
+

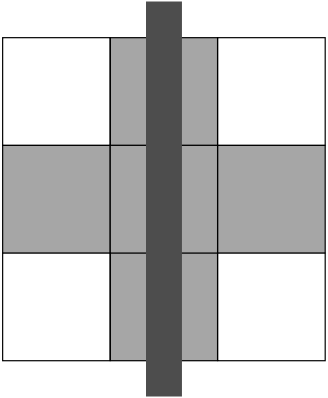
- **Every single BIG is redundant**
  - The first two obtained by composing the lemmas
  - The others by the spatial symmetries of Sudoku
    - Swapping rows in same chute
    - Swapping chutes
    - Rotation
- *Missing*(n): any *Sudoku* subset missing n BIGs
  - Every *Missing*(1) is *Sudoku*
- Can we find larger ns?
- What is the maximum?

In total: 27 Corollaries

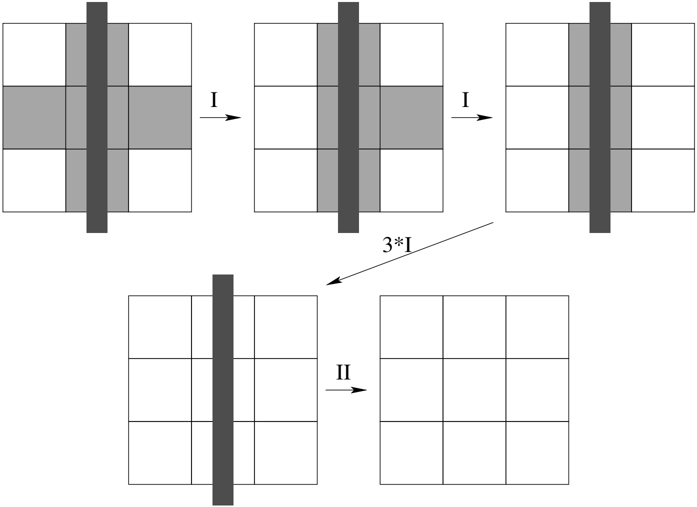
- **Every single BIG is redundant**
  - The first two obtained by composing the lemmas
  - The others by the spatial symmetries of Sudoku
    - Swapping rows in same chute
    - Swapping chutes
    - Rotation
- *Missing*(n): any *Sudoku* subset missing n BIGs
  - Every *Missing*(1) is *Sudoku*
- Can we find larger ns?
- What is the maximum?

- **Every single BIG is redundant**
  - The first two obtained by composing the lemmas
  - The others by the spatial symmetries of Sudoku
    - Swapping rows in same chute
    - Swapping chutes
    - Rotation
- *Missing*(n): any *Sudoku* subset missing n BIGs
  - Every *Missing*(1) is *Sudoku*
- Can we find larger ns?
- What is the maximum?

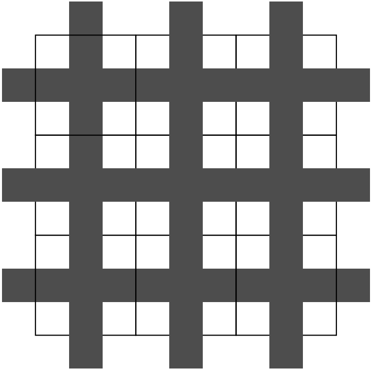
Theorem I:



is Sudoku.


I
I
3*I
II

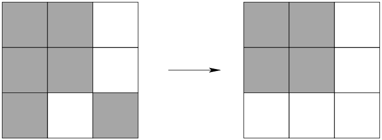
Theorem II:



is Sudoku.

- We would like to classify all *Missing*(6)
  - 296,010 elements (less if we avoid symmetries)
- Plan: computer assisted proofs using Prolog
- Use the constructive lemmas to add new BIGs

- We would like to classify all *Missing*(6)
  - 296,010 elements (less if we avoid symmetries)
- Plan: computer assisted proofs using Prolog
- Use the constructive lemmas to add new BIGs



for each $S \in Missing(n)$ **do**
  $C \leftarrow copy(S)$
  while Lemma I or Lemma II is applicable to C
    apply it to C
  if $|C| = 27$
    then output S *is Sudoku*
    else output S *got stuck in* C

- If stuck:
  - Analyze C manually and turn it into a *negative* lemma
    Proof it is not *Sudoku*
  - Add it to the program
  - Rerun the program until no more negative lemmas needed
- Note:starting with $n = 6$ we derive results for $n \in 2..5$

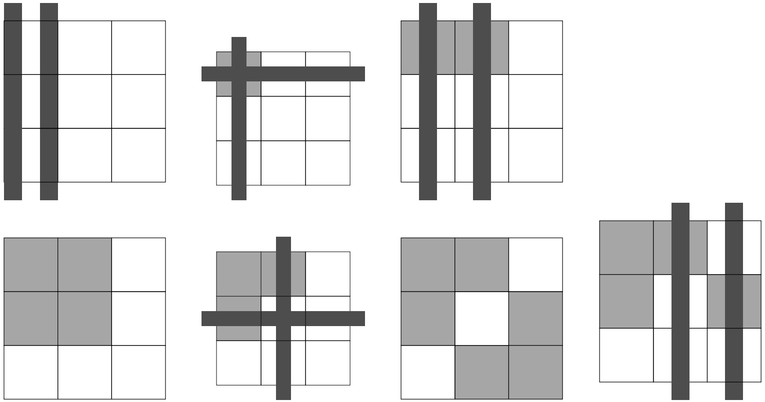
for each $S \in Missing(n)$ **do**
  $C \leftarrow copy(S)$
  while Lemma I or Lemma II is applicable to C
    apply it to C
  if $|C| = 27$
    then output *S is Sudoku*
    else output *S got stuck in C*

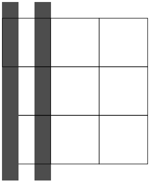
- If stuck:
  - Analyze C manually and turn it into a *negative* lemma
    Proof it is not *Sudoku*
  - Add it to the program
  - Rerun the program until no more negative lemmas needed
- Note:starting with $n = 6$ we derive results for $n \in 2..5$

```
for each S ∈ Missing(n) do
    C ← copy(S)
    while Lemma I or Lemma II is applicable to C
        apply it to C
    if |C| = 27
        then output S is Sudoku
        else output S got stuck in C
```

- If stuck:
  - Analyze C manually and turn it into a *negative* lemma

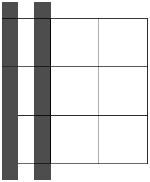
    Proof it is not *Sudoku*
  - Add it to the program
  - Rerun the program until no more negative lemmas needed
- Note:starting with $n = 6$ we derive results for $n \in 2..5$



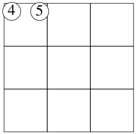


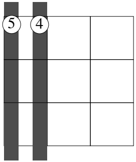
Proof that the above is not *Sudoku*:

Similar for the other six negative Lemmas



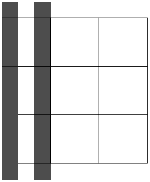
Proof that the above is not *Sudoku*:



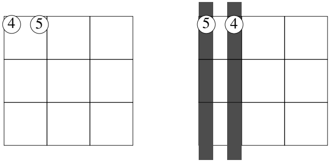


Similar for the other six negative Lemmas

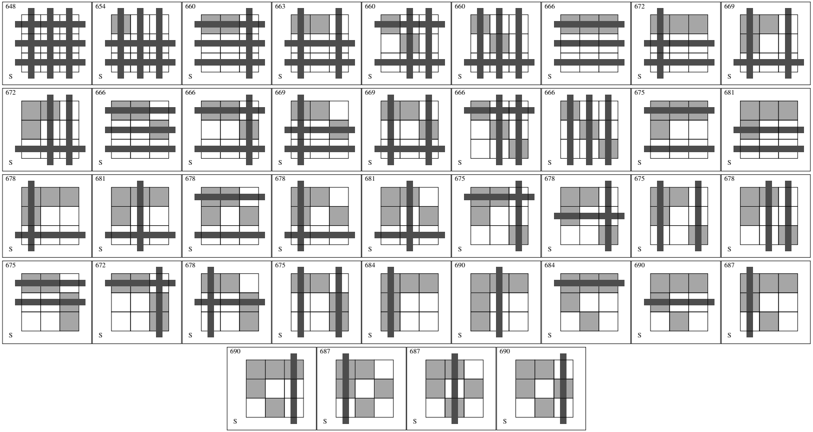
Trivial Negative Lemma



Proof that the above is not *Sudoku*:



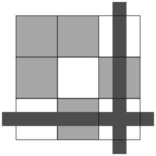
Similar for the other six negative Lemmas



And 71 are not

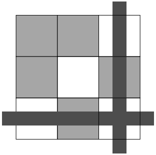
- Use the final program for $n = 7$
- Gets stuck in one new BIG negative lemma:
- **No set of 20 BIGs is Sudoku**, or equivalently
- **No set of 7 BIGs is redundant**
- This settles BIGs completely

- Use the final program for $n = 7$
- Gets stuck in one new BIG negative lemma:



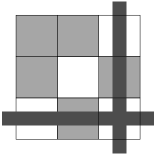
- **No set of 20 BIGs is Sudoku**, or equivalently
- **No set of 7 BIGs is redundant**
- This settles BIGs completely

- Use the final program for $n = 7$
- Gets stuck in one new BIG negative lemma:



- **No set of 20 BIGs is Sudoku**, or equivalently
- **No set of 7 BIGs is redundant**
- This settles BIGs completely

- Use the final program for $n = 7$
- Gets stuck in one new BIG negative lemma:



- **No set of 20 BIGs is Sudoku**, or equivalently
- **No set of 7 BIGs is redundant**
- This settles BIGs completely

- **What about SMALLs?**
- One BIG constraint = quadratic many **small** ones
- Example:
  all_different($\{x_{11}, x_{12}, x_{13}, x_{14}, x_{15}, x_{16}, x_{17}, x_{18}, x_{19}\}$)
  is equivalent to
  $x_{11} \neq x_{12} \wedge x_{11} \neq x_{13} \wedge \ldots \wedge x_{18} \neq x_{19}$
- *Sudoku*: 27 BIGs or 810 different **smalls**
- So, what about small constraints?

- **What about SMALLs?**
- One BIG constraint = quadratic many **small** ones
- Example:
  all_different($\{x_{11}, x_{12}, x_{13}, x_{14}, x_{15}, x_{16}, x_{17}, x_{18}, x_{19}\}$)
  is equivalent to
  $x_{11} \neq x_{12} \wedge x_{11} \neq x_{13} \wedge \ldots \wedge x_{18} \neq x_{19}$
- *Sudoku*: 27 BIGs or 810 different **smalls**
- So, what about small constraints?

- **What about SMALLs?**
- One BIG constraint = quadratic many **small** ones
- Example:
  all_different($\{x_{11}, x_{12}, x_{13}, x_{14}, x_{15}, x_{16}, x_{17}, x_{18}, x_{19}\}$)
  is equivalent to
  $x_{11} \neq x_{12} \wedge x_{11} \neq x_{13} \wedge \ldots \wedge x_{18} \neq x_{19}$
- *Sudoku*: 27 BIGs or 810 different **smalls**
- So, what about small constraints?

- Of the 40 *Sudoku* configurations:
  - The lowest number of smalls is 648 (Theorem II config)
  - The highest is 690
- Take Theorem II configuration (648 smalls)
  - Remove each more small rule
  - Is the result *Sudoku* ?
    - Run on Gordon Royle's puzzles (50,000 minimal *Sudoku*s)
    - Use B-Prolog to find all solutions to each puzzle
    - If more than one solution: not *Sudoku*
- **No such configuration is Sudoku**
- Further: Each of the 40 configs has a subset of 648 smalls that is *Sudoku* and is a local minimal
- *Is 648 a lower bound ?*

- Of the 40 *Sudoku* configurations:
  - The lowest number of smalls is 648 (Theorem II config)
  - The highest is 690
- Take Theorem II configuration (648 smalls)
  - Remove each more small rule
  - Is the result *Sudoku* ?
    - Run on Gordon Royle's puzzles (50,000 minimal *Sudoku*s)
    - Use B-Prolog to find all solutions to each puzzle
    - If more than one solution: not *Sudoku*
- **No such configuration is Sudoku**
- Further: Each of the 40 configs has a subset of 648 smalls that is *Sudoku* and is a local minimal
- *Is 648 a lower bound ?*

- Of the 40 *Sudoku* configurations:
  - The lowest number of smalls is 648 (Theorem II config)
  - The highest is 690
- Take Theorem II configuration (648 smalls)
  - Remove each more small rule
  - Is the result *Sudoku* ?
    - Run on Gordon Royle's puzzles (50,000 minimal *Sudoku*s)
    - Use B-Prolog to find all solutions to each puzzle
    - If more than one solution: not *Sudoku*
- **No such configuration is Sudoku**
- Further: Each of the 40 configs has a subset of 648 smalls that is *Sudoku* and is a local minimal
- *Is 648 a lower bound ?*

- Of the 40 *Sudoku* configurations:
  - The lowest number of smalls is 648 (Theorem II config)
  - The highest is 690
- Take Theorem II configuration (648 smalls)
  - Remove each more small rule
  - Is the result *Sudoku* ?
    - Run on Gordon Royle's puzzles (50,000 minimal *Sudoku*s)
    - Use B-Prolog to find all solutions to each puzzle
    - If more than one solution: not *Sudoku*
- **No such configuration is Sudoku**
- Further: Each of the 40 configs has a subset of 648 smalls that is *Sudoku* and is a local minimal
- *Is 648 a lower bound ?*

- Of the 40 *Sudoku* configurations:
  - The lowest number of smalls is 648 (Theorem II config)
  - The highest is 690
- Take Theorem II configuration (648 smalls)
  - Remove each more small rule
  - Is the result *Sudoku* ?
    - Run on Gordon Royle's puzzles (50,000 minimal *Sudoku*s)
    - Use B-Prolog to find all solutions to each puzzle
    - If more than one solution: not *Sudoku*
- **No such configuration is Sudoku**
- Further: Each of the 40 configs has a subset of 648 smalls that is *Sudoku* and is a local minimal
- *Is 648 a lower bound ?*

- Doing a similar approach to the BIGs is too costly
- AND we would actually like to gain insight
- Plan: try a similar but simpler problem
  - One that has a smaller number of possibilities
  - We can find these automatically
  - And the reason about them
- Latin Squares

- Doing a similar approach to the BIGs is too costly
- AND we would actually like to gain insight
- Plan: try a similar but simpler problem
  - One that has a smaller number of possibilities
  - We can find these automatically
  - And the reason about them
- Latin Squares

- Doing a similar approach to the BIGs is too costly
- AND we would actually like to gain insight
- Plan: try a similar but simpler problem
  - One that has a smaller number of possibilities
  - We can find these automatically
  - And the reason about them
- Latin Squares

LS(N)

- NxN matrix
- Every cell is in 1..N
- Cells in the same column differ (N BIGs)
- Cells in the same row differ (N BIGs)

You can think about it as a simpler form of Sudoku:

LS(9) + the 9 block constraints == Sudoku

First: study BIGs in this context

LS(N)

- NxN matrix
- Every cell is in 1..N
- Cells in the same column differ (N BIGs)
- Cells in the same row differ (N BIGs)

You can think about it as a simpler form of Sudoku:

LS(9) + the 9 block constraints == Sudoku

First: study BIGs in this context

LS(N)

- NxN matrix
- Every cell is in 1..N
- Cells in the same column differ (N BIGs)
- Cells in the same row differ (N BIGs)

You can think about it as a simpler form of Sudoku:

LS(9) + the 9 block constraints == Sudoku

First: study BIGs in this context

- Every single BIG is redundant (as for Sudoku)
- Any set with two or more BIGs is not redundant

- Every single BIG is redundant (as for Sudoku)
- Any set with two or more BIGs is not redundant

- Every single BIG is redundant (as for Sudoku)
- Any set with two or more BIGs is not redundant

1	1	1
1	2	3
1	3	2

1	1	1	1
1	2	3	4
1	3	4	2
1	4	2	3

1	1	1	1	1
1	2	3	4	5
1	3	4	5	2
1	4	5	2	3
1	5	2	3	4

1	**1**	2
2	**3**	3
3	2	1

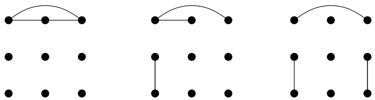
1	**1**	3	2
2	**4**	4	3
3	2	1	4
4	3	2	1

1	**1**	4	3	2
2	**5**	5	4	3
3	2	1	5	4
4	3	2	1	5
5	4	3	2	1

- Analyse LS(2), LS(3) and LS(4) by hand
  - After a Prolog program found the maximal redundant sets
- For LS(2): any small is redundant, more is not
- For LS(3): the only maximal redundant sets are
- For LS(4): pattern I and II are the only ones

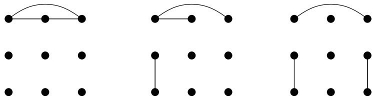
- Analyse LS(2), LS(3) and LS(4) by hand
  - After a Prolog program found the maximal redundant sets
- For LS(2): any small is redundant, more is not
- For LS(3): the only maximal redundant sets are
- For LS(4): pattern I and II are the only ones

- Analyse LS(2), LS(3) and LS(4) by hand
  - After a Prolog program found the maximal redundant sets
- For LS(2): any small is redundant, more is not
- For LS(3): the only maximal redundant sets are



- For LS(4): pattern I and II are the only ones

- Analyse LS(2), LS(3) and LS(4) by hand
  - After a Prolog program found the maximal redundant sets
- For LS(2): any small is redundant, more is not
- For LS(3): the only maximal redundant sets are

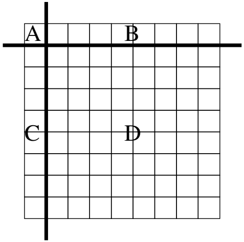


- For LS(4): pattern I and II are the only ones

- Patterns I and II are redundant for any LS(N)
  - Proving pattern I is obvious (same as a BIG)
  - Proved II by dividing the matrix into regions
- Are they the only redundant ones?
- How to prove this?

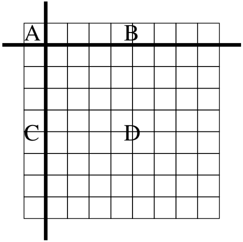
- Patterns I and II are redundant for any LS(N)
  - Proving pattern I is obvious (same as a BIG)
  - Proved II by dividing the matrix into regions
- Are they the only redundant ones?
- How to prove this?

- Patterns I and II are redundant for any LS(N)
  - Proving pattern I is obvious (same as a BIG)
  - Proved II by dividing the matrix into regions



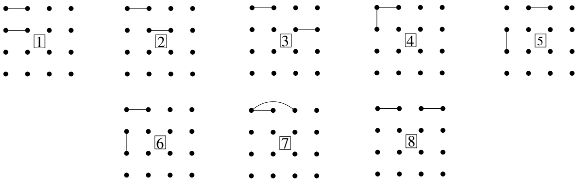
- Are they the only redundant ones?
- How to prove this?

- Patterns I and II are redundant for any LS(N)
  - Proving pattern I is obvious (same as a BIG)
  - Proved II by dividing the matrix into regions



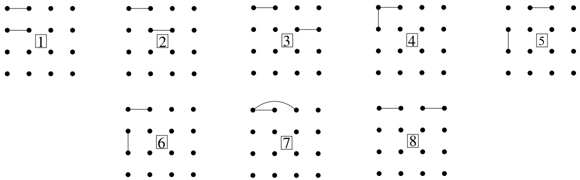
- Are they the only redundant ones?
- How to prove this?

- All possible pairs of inequalities for LS(4)



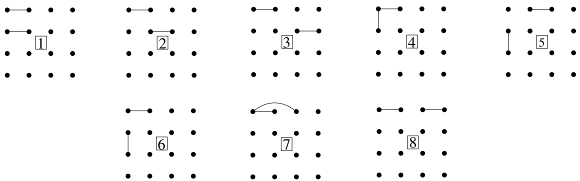
- All possible pairs of inequalities for LS(N)
  - Proved using 3 cases: parallel lines, crossing ones, same line
- 6 is covered by Pattern II, 7 and 8 by Pattern I
- No pair in 1 to 5 is covered by Pattern I or II
  - **Bad pairs**: any subset of LS(N) that mises one of those pairs, has at least one more solution than LS(N), for N>3

- All possible pairs of inequalities for LS(4)



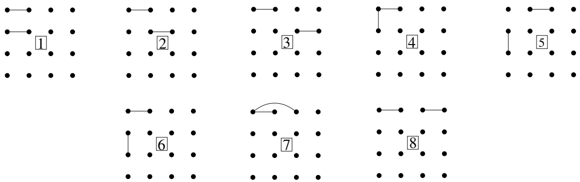
- All possible pairs of inequalities for LS(N)
  - Proved using 3 cases: parallel lines, crossing ones, same line
- 6 is covered by Pattern II, 7 and 8 by Pattern I
- No pair in 1 to 5 is covered by Pattern I or II
  - **Bad pairs**: any subset of LS(N) that mises one of those pairs, has at least one more solution than LS(N), for N>3

- All possible pairs of inequalities for LS(4)

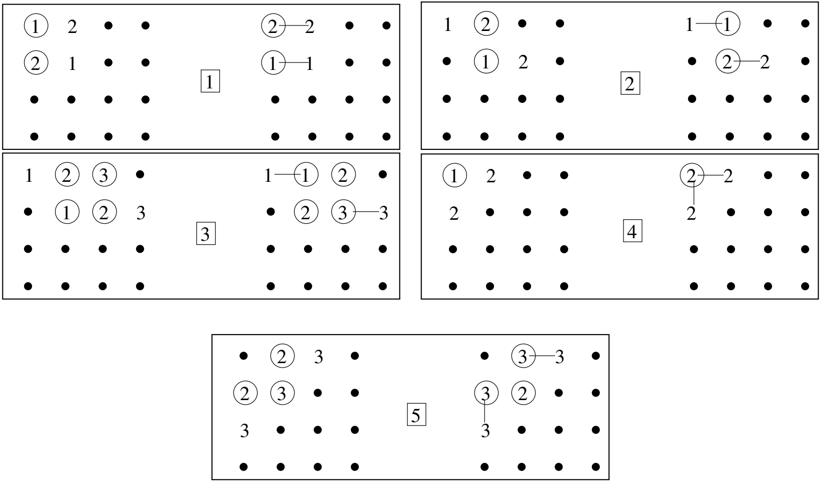


- All possible pairs of inequalities for LS(N)
  - Proved using 3 cases: parallel lines, crossing ones, same line
- 6 is covered by Pattern II, 7 and 8 by Pattern I
- No pair in 1 to 5 is covered by Pattern I or II
  - **Bad pairs**: any subset of LS(N) that mises one of those pairs, has at least one more solution than LS(N), for N>3

- All possible pairs of inequalities for LS(4)



- All possible pairs of inequalities for LS(N)
  - Proved using 3 cases: parallel lines, crossing ones, same line
- 6 is covered by Pattern II, 7 and 8 by Pattern I
- No pair in 1 to 5 is covered by Pattern I or II
  - **Bad pairs**: any subset of LS(N) that mises one of those pairs, has at least one more solution than LS(N), for N>3


1
2
3
4
5

Theorem:

Every set S of inequalities from $LS(N)$ with $N > 3$ either contains a bad pair or is covered by one of the two patterns.

- LS(N,I): same as LS(N) with domain 1..I
- Clearly, LS(N,N-1) has no solutions
- We give the minimum number of smalls that need to be removed from LS(N,I), I<N to be satisfiable (2*(N-I)*N)
- No solution to LS(N,N+1) is stable
- Discuss how many smalls need to be added to become stable again

- LS(N,I): same as LS(N) with domain 1..I
- Clearly, LS(N,N-1) has no solutions
- We give the minimum number of smalls that need to be removed from LS(N,I), I<N to be satisfiable (2*(N-I)*N)
- No solution to LS(N,N+1) is stable
- Discuss how many smalls need to be added to become stable again

- LS(N,I): same as LS(N) with domain 1..I
- Clearly, LS(N,N-1) has no solutions
- We give the minimum number of smalls that need to be removed from LS(N,I), I<N to be satisfiable (2*(N-I)*N)
- No solution to LS(N,N+1) is stable
- Discuss how many smalls need to be added to become stable again

- LS(N,I): same as LS(N) with domain 1..I
- Clearly, LS(N,N-1) has no solutions
- We give the minimum number of smalls that need to be removed from LS(N,I), I<N to be satisfiable (2*(N-I)*N)
- No solution to LS(N,N+1) is stable
- Discuss how many smalls need to be added to become stable again

- LS(N,I): same as LS(N) with domain 1..I
- Clearly, LS(N,N-1) has no solutions
- We give the minimum number of smalls that need to be removed from LS(N,I), I<N to be satisfiable (2*(N-I)*N)
- No solution to LS(N,N+1) is stable
- Discuss how many smalls need to be added to become stable again

- Plan: study 2x2 Sudoku applying what we learned from
  - 3x3 Sudoku (is the conjecture true? no set of 17 inequalities is redundant)
  - Latin Square (does 2x2 Sudobad pairs?)

- Sudoku:
  - Completely characterised redundant BIGs
  - Conjectured smalls
- Latin Square
  - Completely characterised redundant BIGs
  - AND smalls (through bad pairs)
- Started applying the above to 2x2 Sudoku
- Aim: better understand redundancy of smalls

- Sudoku:
  - Completely characterised redundant BIGs
  - Conjectured smalls
- Latin Square
  - Completely characterised redundant BIGs
  - AND smalls (through bad pairs)
- Started applying the above to 2x2 Sudoku
- Aim: better understand redundancy of smalls

- Sudoku:
  - Completely characterised redundant BIGs
  - Conjectured smalls
- Latin Square
  - Completely characterised redundant BIGs
  - AND smalls (through bad pairs)
- Started applying the above to 2x2 Sudoku
- Aim: better understand redundancy of smalls