# Scarab
# SAT-based Constraint Programming System in Scala

Takehide Soh[1]
worked in cooperation with
Daniel Le Berre[2] Stéphanie Roussel[2]
Mutsunori Banbara[1] Naoyuki Tamura[1]

[1]Information Science and Technology Center, Kobe University

[2]CRIL-CNRS, UMR 8188, Université d'Artois

2014/09/05
2019/02/22 (revised)

# Contents of Talk

1. What is SAT?
2. Scarab: SAT-based CP System in Scala
3. Designing Constraint Models in Scarab

# SAT (Boolean satisfiability testing) Problems

- **SAT** is a problem of deciding whether a given Boolean formula is satisfiable or not.

- SAT was the first NP-complete problem [Cook, 1971] and is the most fundamental problem in Computer Science both theoretically and practically.

SAT instances are given in the Conjunctive Normal Form (CNF).

- A **CNF formula** is a conjunction of clauses.
- A **clause** is a disjunction of literals.
- A **literal** is either a Boolean variable or its negation.

# SAT (Boolean satisfiability testing) Problems

- **SAT** is a problem of deciding whether a given Boolean formula is satisfiable or not.

- SAT was the first NP-complete problem [Cook, 1971] and is the most fundamental problem in Computer Science both theoretically and practically.

## Example of an SAT instance (in CNF)

- Let $a, b, c \in \{True, False\}$ be Boolean variables.
- **Question**: the following CNF formula is satisfiable or not?

$$\begin{array}{l} (a \vee b \vee c) \wedge \\ (\neg a \vee \neg b) \wedge \\ (\neg a \vee \neg c) \wedge \\ (\neg b \vee \neg c) \end{array}$$

- **Answer**: Yes.
- There is an assignment $(a, b, c) = (True, False, False)$ satisfying all clauses.

# SAT Solvers

- There are $2^n$ combinations for assignments.
- We cannot solve any SAT instances even for small $n$ (e.g. $n = 100$)?

# SAT Solvers

- There are $2^n$ combinations for assignments.
- We cannot solve any SAT instances even for small $n$ (e.g. $n = 100$)?

**SAT solver** is a program of deciding whether a given SAT instance is satisfiable (`SAT`) or unsatisfiable (`UNSAT`).

Most of all SAT solvers return a satisfiable assignment when the instance is SAT.

# SAT Solvers

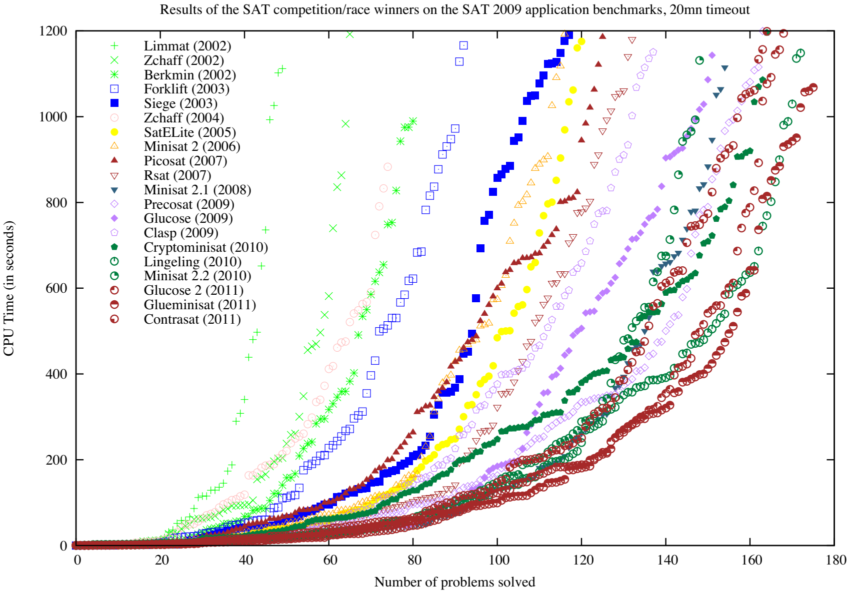
- There are $2^n$ combinations for assignments.
- We cannot solve any SAT instances even for small $n$ (e.g. $n = 100$)?

**SAT solver** is a program of deciding whether a given SAT instance is satisfiable (`SAT`) or unsatisfiable (`UNSAT`).

Most of all SAT solvers return a satisfiable assignment when the instance is SAT.

- Complete SAT solvers originated in **DPLL** [Davis et al., 1962].
- In particular, since around 2000, their performance is improved every year with techniques of Conflict Driven Clause Learning (CDCL), Non-chronological Backtracking, Rapid Restarts, and Variable Selection Heuristics etc.
- Modern SAT solvers can handle instances with more than $10^6$ **variables and** $10^7$ **clauses**.

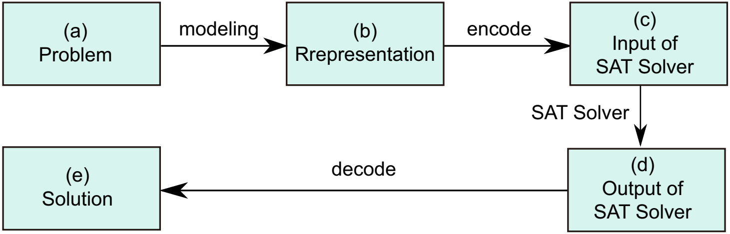
# Progress of SAT Solvers (shown by [Simon 2011])




Cactus Plot shown by [Simon 2011]

# Problem Solving using SAT Solvers

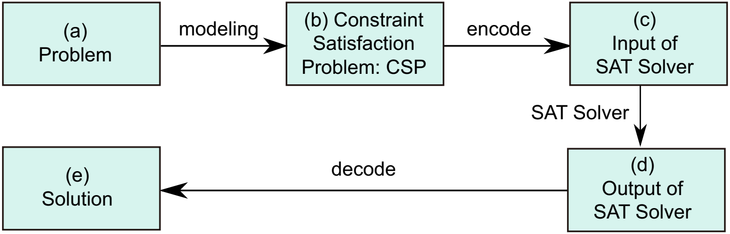
- Thanks to the remarkable progress of SAT solvers, **SAT-based Problem Solving** have been actively studied.




- **SAT-based Systems** are implementations of SAT-based problem solving.
- Many research topics in this field. Among them, the **importance of modeling and encoding** are re-recognized.
- Good modeling/encodings are developed considering the size of solver input and propagations in SAT solvers (and many many trial/errors are necessary!).

# Problem Solving using SAT Solvers

- Thanks to the remarkable progress of SAT solvers, **SAT-based Problem Solving** have been actively studied.




- **SAT-based Systems** are implementations of SAT-based problem solving.
- Many research topics in this field. Among them, the **importance of modeling and encoding** are re-recognized.
- Good modeling/encodings are developed considering the size of solver input and propagations in SAT solvers (and many many trial/errors are necessary!).

# SAT encodings

There have been several methods proposed to encode CSP into SAT.

- *Direct encoding* is the most widely used one [de Kleer, 1989].
- Other encodings:
  - *Multivalued encoding* [Selman et al., 1992]
  - *Support encoding* [Kasif, 1990]
  - *Log encoding* [Iwama and Miyazaki, 1994]
  - *Log-support encoding* [Gavanelli, 2007]

- **Order encoding** is a new encoding showing a good performance for a wide variety of problems [Tamura et al., 2006].
- It is shown that the order encoindg is the only encoding translating tractable CSP to tractable SAT [Petke and Jeavons, 2011].
  - It is first used to encode job-shop scheduling problems by [Crawford and Baker, 1994].
  - It succeeded to solve previously undecided problems in open-shop scheduling, job-shop scheduling, and two-dimensional strip packing.

# Applications of SAT Technology

- **Planning** (SATPLAN, Blackbox) [Kautz and Selman, 1992]
- **Job-shop Scheduling** [Crawford and Baker, 1994]
- **Bounded Model Checking** [Biere et al., 1999]
- **Term Rewriting** (AProVE) [Giesl et al. 2004]
- **Constraint Satisfaction Problem**[Tamura et al., 2006]
  - **Sugar**, SAT-based CSP Solvr, which is the Winner of 2008 and 2009 CSP Solver Competitions in GLOBAL categories.
  - It adopts Order Encoding.
- Others
  - Test Case Generation,
  - Systems Biology,
  - Timetabling,
  - Packing,
  - Puzzle, and more!

# Other News around SAT

- A **SAT solver Sat4j** implemented on Java has been integrated into **Eclipse** for managing plugins dependencies in their update manager.
- **Donald E. Knuth** gave an invited talk about SAT at the International Conference on Theory and Applications of Satisfiability Testing 2012.
  - SAT will be appeared in Volume 4b of **The Art Of Computer Programming**.

## Reference to SAT


- Biere, A., Heule, M., van Maaren, H., and Walsh, T., editors (2009). Handbook of Satisfiability, volume 185 of Frontiers in Artificial Intelligence and Applications (FAIA). IOS Press.
- (in Japanese) Recent Advances in SAT Techniques, Journal of the Japan Society for Artificial Intelligence, Special Issue, 25(1), 2010.

# Contents of Talk

1. What is SAT?
2. Scarab: SAT-based CP System in Scala
3. Designing Constraint Models in Scarab

# Motivation

- Modern fast SAT solvers have promoted the development of **SAT-based systems** for various problems.
- For an intended problem, we usually need to develop a dedicated program that encodes it into SAT.
- It sometimes bothers focusing on **problem modeling** which plays an important role in the system development process.

## In the following

- We introduce the **Scarab** system, which is a prototyping tool for developing SAT-based systems.
- Its features are also introduced through examples of **Graph Coloring**.

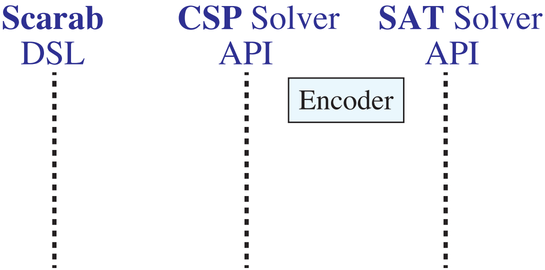
# Scarab

**Scarab** is a prototyping tool for developing SAT-based Constraint Programming (CP) systems.
Its major design principle is to provide an expressive, efficient, customizable, and portable workbench for SAT-based system developers.

# Scarab

**Scarab** is a prototyping tool for developing SAT-based Constraint Programming (CP) systems.
Its major design principle is to provide an expressive, efficient, customizable, and portable workbench for SAT-based system developers.

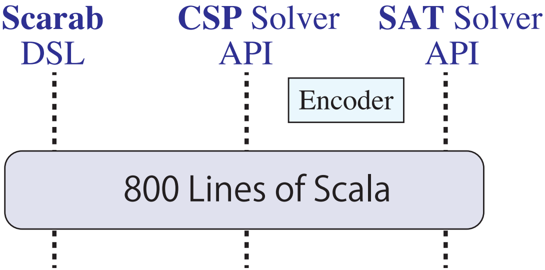
- It consists of the followings:
  1. Scarab DSL: Embedded DSL for Constraint Programming
  2. API of CSP Solver
  3. SAT encoding module
  4. API of SAT solvers



# Scarab

**Scarab** is a prototyping tool for developing SAT-based Constraint Programming (CP) systems.
Its major design principle is to provide an expressive, efficient, customizable, and portable workbench for SAT-based system developers.

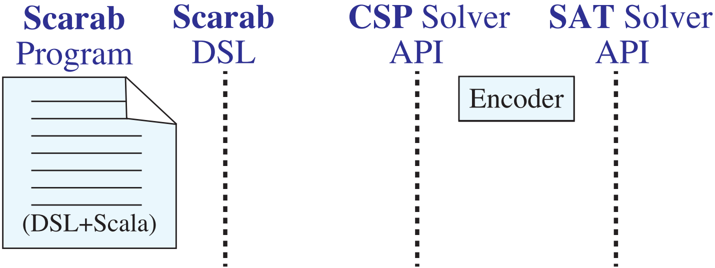
- It consists of the followings:
  1. Scarab DSL: Embedded DSL for Constraint Programming
  2. API of CSP Solver
  3. SAT encoding module
  4. API of SAT solvers



# Scarab

**Scarab** is a prototyping tool for developing SAT-based Constraint Programming (CP) systems.
Its major design principle is to provide an expressive, efficient, customizable, and portable workbench for SAT-based system developers.

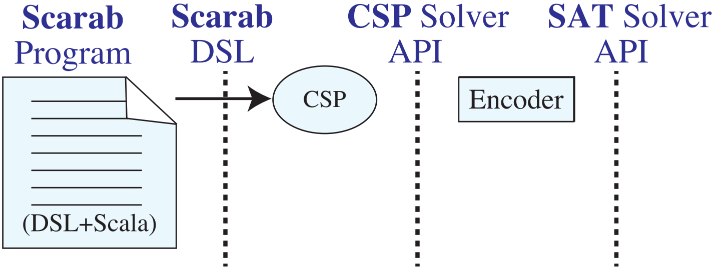
- It consists of the followings:
  1. Scarab DSL: Embedded DSL for Constraint Programming
  2. API of CSP Solver
  3. SAT encoding module
  4. API of SAT solvers



# Scarab

**Scarab** is a prototyping tool for developing SAT-based Constraint Programming (CP) systems.
Its major design principle is to provide an expressive, efficient, customizable, and portable workbench for SAT-based system developers.

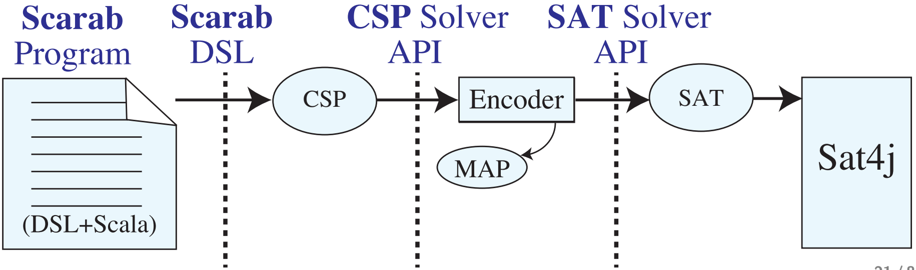
- It consists of the followings:
  1. Scarab DSL: Embedded DSL for Constraint Programming
  2. API of CSP Solver
  3. SAT encoding module
  4. API of SAT solvers



# Scarab

**Scarab** is a prototyping tool for developing SAT-based Constraint Programming (CP) systems.
Its major design principle is to provide an expressive, efficient, customizable, and portable workbench for SAT-based system developers.

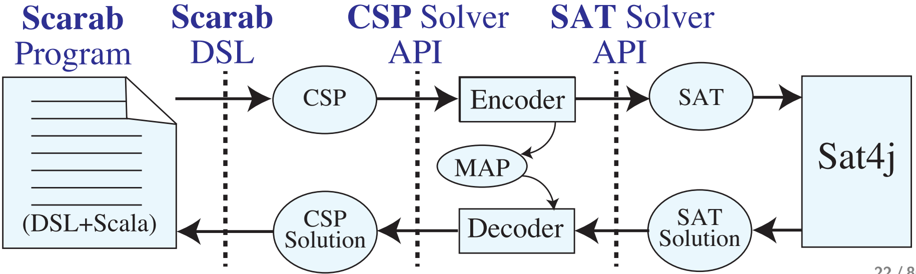
- It consists of the followings:
  1. Scarab DSL: Embedded DSL for Constraint Programming
  2. API of CSP Solver
  3. SAT encoding module
  4. API of SAT solvers



# Scarab

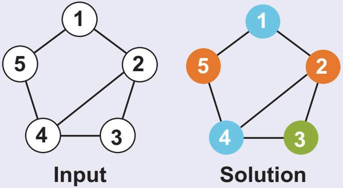
**Scarab** is a prototyping tool for developing SAT-based Constraint Programming (CP) systems.
Its major design principle is to provide an expressive, efficient, customizable, and portable workbench for SAT-based system developers.

- It consists of the followings:
  1. Scarab DSL: Embedded DSL for Constraint Programming
  2. API of CSP Solver
  3. SAT encoding module
  4. API of SAT solvers



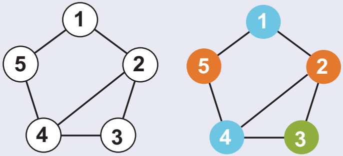
# Example of Scarab Program: GCP.scala

**Graph coloring problem** (GCP) is a problem of finding a coloring of the nodes such that colors of adjacent nodes are different.



# Example of Scarab Program: GCP.scala

**Graph coloring problem** (GCP) is a problem of finding a coloring of the nodes such that colors of adjacent nodes are different.




Input Solution

```
import jp.kobe_u.scarab._ ; import dsl._

val nodes = Seq(1,2,3,4,5)
val edges = Seq((1,2),(1,5),(2,3),(2,4),(3,4),(4,5))
val colors = 3
for (i <- nodes) int('n(i),1,colors)
for ((i,j) <- edges)  add('n(i) !== 'n(j))

if (find) println(solution)
```

# Imports

```
import jp.kobe_u.scarab._ ; import dsl._
```

- This line imports everything necessary and DSL methods provided by Scarab.

- `int(x, lb, ub)` method defines an integer variable.
- `add(c)` method defines a constraint.
- `find` method searches a solution.
- `solution` method returns the solution.
- etc.

# Instance Structure

```scala
val nodes = Seq(1,2,3,4,5)
val edges = Seq((1,2),(1,5),(2,3),(2,4),(3,4),(4,5))
val colors = 3
```

- It defines the given set of nodes and edges as the sequence object in Scala.
- Available number of colors are defined as 3.

# Defining CSP

```
for (i <- nodes) int('n(i),1,3)
```

- It adds an integer variable to the default CSP object by the **int** method.
- `'n` is a notation of symbols in Scala.
- They are automatically converted integer variable (`Var`) objects by an implicit conversion defined in Scarab.

```
for ((i,j) <- edges)  add('n(i) !== 'n(j))
```

- It adds constraints to the default CSP object.
- The following operators can be used to construct constraints:
  - logical operator: `&&`, `||`
  - comparison operator: `===`, `!==`, `<`, `<=`, `>=`, `>`
  - arithmetic operator: `+`, `-`

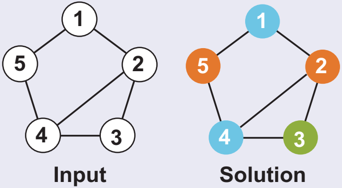
# Solving CSP

```
if (find) println(solution)
```

- The **find** method encodes the CSP to SAT by order encoding, and call Sat4j to compute a solution.
- **solution** returns satisfiable assignment of the CSP.

# We can do more in GCP?

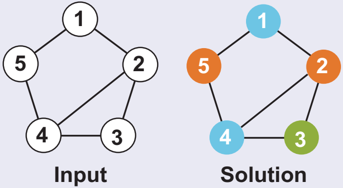
**Graph coloring problem** (GCP) is a problem of finding a coloring of the nodes such that colors of adjacent nodes are different.




- How can we solve optimization version of GCP using Scarab?

# We can do more in GCP?

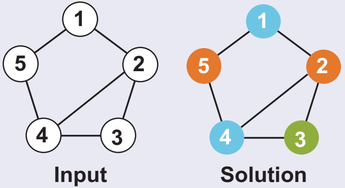
**Graph coloring problem** (GCP) is a problem of finding a coloring of the nodes such that colors of adjacent nodes are different.




- How can we solve optimization version of GCP using Scarab?
- How can we adopt the change of constraints?

# We can do more in GCP?

**Graph coloring problem** (GCP) is a problem of finding a coloring of the nodes such that colors of adjacent nodes are different.




- How can we solve optimization version of GCP using Scarab?
- How can we adopt the change of constraints?
  - Let's consider bandwidth coloring problem!

# Contents of Talk

1. What is SAT?
2. Scarab: SAT-based CP System in Scala
3. Designing Constraint Models in Scarab

# Pandiagonal Latin Square: $PLS(n)$

Place different $n$ numbers into $n \times n$ matrix such that each number appears exactly once for each row, column, diagonally down right, and diagonally up right.

| | | | | |
|---|---|---|---|---|
| 2 | 3 | 5 | 1 | 4 |
| 5 | 1 | 4 | 2 | 3 |
| 4 | 2 | 3 | 5 | 1 |
| 3 | 5 | 1 | 4 | 2 |
| 1 | 4 | 2 | 3 | 5 |

# Pandiagonal Latin Square: $PLS(n)$

Place different $n$ numbers into $n \times n$ matrix such that each number appears exactly once for each row, column, diagonally down right, and diagonally up right.

| | | | | |
|---|---|---|---|---|
| 2 | 3 | 5 | 1 | 4 |
| 5 | 1 | 4 | 2 | 3 |
| 4 | 2 | 3 | 5 | 1 |
| 3 | 5 | 1 | 4 | 2 |
| 1 | 4 | 2 | 3 | 5 |

We can write five SAT-based PLS Solvers **within 35 lines**.

| Name | Modeling | Encoding | Lines |
|---|---|---|---|
| AD1 | alldiff | naive | 17 |
| AD2 | | with Perm. & P. H. Const. | 31 |
| BC1 | Boolean | Pairwise | 22 |
| BC2 | Cardinality | Totalizer [Bailleux '03] | 35 |
| BC3 | | Seq. Counter [Sinz '05] | 27 |

Let's have a look their performance. Note that, in CSP Solver Comp. 2009, **NO CSP solver** (except Sugar) could solve $n > 8$.

# alldiff Model

## Pandiagonal Latin Square $PLS(5)$

| | | | | |
|---|---|---|---|---|
| $x_{11}$ | $x_{12}$ | $x_{13}$ | $x_{14}$ | $x_{15}$ |
| $x_{21}$ | $x_{22}$ | $x_{23}$ | $x_{24}$ | $x_{25}$ |
| $x_{31}$ | $x_{32}$ | $x_{33}$ | $x_{34}$ | $x_{35}$ |
| $x_{41}$ | $x_{42}$ | $x_{43}$ | $x_{44}$ | $x_{45}$ |
| $x_{51}$ | $x_{52}$ | $x_{53}$ | $x_{54}$ | $x_{55}$ |

- $x_{ij} \in \{1, 2, 3, 4, 5\}$

# alldiff Model

## Pandiagonal Latin Square $PLS(5)$

| | | | | |
|---|---|---|---|---|
| $x_{11}$ | $x_{12}$ | $x_{13}$ | $x_{14}$ | $x_{15}$ |
| $x_{21}$ | $x_{22}$ | $x_{23}$ | $x_{24}$ | $x_{25}$ |
| $x_{31}$ | $x_{32}$ | $x_{33}$ | $x_{34}$ | $x_{35}$ |
| $x_{41}$ | $x_{42}$ | $x_{43}$ | $x_{44}$ | $x_{45}$ |
| $x_{51}$ | $x_{52}$ | $x_{53}$ | $x_{54}$ | $x_{55}$ |

- $x_{ij} \in \{1, 2, 3, 4, 5\}$
- alldiff in each row (5 rows)

# alldiff Model

## Pandiagonal Latin Square $PLS(5)$

| | | | | |
|---|---|---|---|---|
| $x_{11}$ | $x_{12}$ | $x_{13}$ | $x_{14}$ | $x_{15}$ |
| $x_{21}$ | $x_{22}$ | $x_{23}$ | $x_{24}$ | $x_{25}$ |
| $x_{31}$ | $x_{32}$ | $x_{33}$ | $x_{34}$ | $x_{35}$ |
| $x_{41}$ | $x_{42}$ | $x_{43}$ | $x_{44}$ | $x_{45}$ |
| $x_{51}$ | $x_{52}$ | $x_{53}$ | $x_{54}$ | $x_{55}$ |

- $x_{ij} \in \{1, 2, 3, 4, 5\}$
- alldiff in each row (5 rows)

# alldiff Model

## Pandiagonal Latin Square $PLS(5)$

| | | | | |
|---|---|---|---|---|
| $x_{11}$ | $x_{12}$ | $x_{13}$ | $x_{14}$ | $x_{15}$ |
| $x_{21}$ | $x_{22}$ | $x_{23}$ | $x_{24}$ | $x_{25}$ |
| $x_{31}$ | $x_{32}$ | $x_{33}$ | $x_{34}$ | $x_{35}$ |
| $x_{41}$ | $x_{42}$ | $x_{43}$ | $x_{44}$ | $x_{45}$ |
| $x_{51}$ | $x_{52}$ | $x_{53}$ | $x_{54}$ | $x_{55}$ |

- $x_{ij} \in \{1, 2, 3, 4, 5\}$
- alldiff in each row (5 rows)
- alldiff in each column (5 columns)

# alldiff Model

## Pandiagonal Latin Square $PLS(5)$

| | | | | |
|---|---|---|---|---|
| $x_{11}$ | $x_{12}$ | $x_{13}$ | $x_{14}$ | $x_{15}$ |
| $x_{21}$ | $x_{22}$ | $x_{23}$ | $x_{24}$ | $x_{25}$ |
| $x_{31}$ | $x_{32}$ | $x_{33}$ | $x_{34}$ | $x_{35}$ |
| $x_{41}$ | $x_{42}$ | $x_{43}$ | $x_{44}$ | $x_{45}$ |
| $x_{51}$ | $x_{52}$ | $x_{53}$ | $x_{54}$ | $x_{55}$ |

- $x_{ij} \in \{1, 2, 3, 4, 5\}$
- alldiff in each row (5 rows)
- alldiff in each column (5 columns)

# alldiff Model

## Pandiagonal Latin Square $PLS(5)$

| | | | | |
|---|---|---|---|---|
| $x_{11}$ | $x_{12}$ | $x_{13}$ | $x_{14}$ | $x_{15}$ |
| $x_{21}$ | $x_{22}$ | $x_{23}$ | $x_{24}$ | $x_{25}$ |
| $x_{31}$ | $x_{32}$ | $x_{33}$ | $x_{34}$ | $x_{35}$ |
| $x_{41}$ | $x_{42}$ | $x_{43}$ | $x_{44}$ | $x_{45}$ |
| $x_{51}$ | $x_{52}$ | $x_{53}$ | $x_{54}$ | $x_{55}$ |

- $x_{ij} \in \{1, 2, 3, 4, 5\}$
- alldiff in each row (5 rows)
- alldiff in each column (5 columns)
- alldiff in each pandiagonal (10 pandiagonals)

# alldiff Model

## Pandiagonal Latin Square $PLS(5)$

| | | | | |
|---|---|---|---|---|
| $x_{11}$ | $x_{12}$ | $x_{13}$ | $x_{14}$ | $x_{15}$ |
| $x_{21}$ | $x_{22}$ | $x_{23}$ | $x_{24}$ | $x_{25}$ |
| $x_{31}$ | $x_{32}$ | $x_{33}$ | $x_{34}$ | $x_{35}$ |
| $x_{41}$ | $x_{42}$ | $x_{43}$ | $x_{44}$ | $x_{45}$ |
| $x_{51}$ | $x_{52}$ | $x_{53}$ | $x_{54}$ | $x_{55}$ |

- $x_{ij} \in \{1, 2, 3, 4, 5\}$
- alldiff in each row (5 rows)
- alldiff in each column (5 columns)
- alldiff in each pandiagonal (10 pandiagonals)

# alldiff Model

## Pandiagonal Latin Square $PLS(5)$

| | | | | |
|---|---|---|---|---|
| $x_{11}$ | $x_{12}$ | $x_{13}$ | $x_{14}$ | $x_{15}$ |
| $x_{21}$ | $x_{22}$ | $x_{23}$ | $x_{24}$ | $x_{25}$ |
| $x_{31}$ | $x_{32}$ | $x_{33}$ | $x_{34}$ | $x_{35}$ |
| $x_{41}$ | $x_{42}$ | $x_{43}$ | $x_{44}$ | $x_{45}$ |
| $x_{51}$ | $x_{52}$ | $x_{53}$ | $x_{54}$ | $x_{55}$ |

- $x_{ij} \in \{1, 2, 3, 4, 5\}$
- alldiff in each row (5 rows)
- alldiff in each column (5 columns)
- alldiff in each pandiagonal (10 pandiagonals)

# alldiff Model

## Pandiagonal Latin Square $PLS(5)$

| | | | | |
|---|---|---|---|---|
| $x_{11}$ | $x_{12}$ | $x_{13}$ | $x_{14}$ | $x_{15}$ |
| $x_{21}$ | $x_{22}$ | $x_{23}$ | $x_{24}$ | $x_{25}$ |
| $x_{31}$ | $x_{32}$ | $x_{33}$ | $x_{34}$ | $x_{35}$ |
| $x_{41}$ | $x_{42}$ | $x_{43}$ | $x_{44}$ | $x_{45}$ |
| $x_{51}$ | $x_{52}$ | $x_{53}$ | $x_{54}$ | $x_{55}$ |

- $x_{ij} \in \{1, 2, 3, 4, 5\}$
- alldiff in each row (5 rows)
- alldiff in each column (5 columns)
- alldiff in each pandiagonal (10 pandiagonals)

# alldiff Model

## Pandiagonal Latin Square $PLS(5)$

| | | | | |
|---|---|---|---|---|
| $x_{11}$ | $x_{12}$ | $x_{13}$ | $x_{14}$ | $x_{15}$ |
| $x_{21}$ | $x_{22}$ | $x_{23}$ | $x_{24}$ | $x_{25}$ |
| $x_{31}$ | $x_{32}$ | $x_{33}$ | $x_{34}$ | $x_{35}$ |
| $x_{41}$ | $x_{42}$ | $x_{43}$ | $x_{44}$ | $x_{45}$ |
| $x_{51}$ | $x_{52}$ | $x_{53}$ | $x_{54}$ | $x_{55}$ |

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| 3 | 4 | 5 | 1 | 2 |
| 5 | 1 | 2 | 3 | 4 |
| 2 | 3 | 4 | 5 | 1 |
| 4 | 5 | 1 | 2 | 3 |

- $x_{ij} \in \{1, 2, 3, 4, 5\}$
- alldiff in each row (5 rows)
- alldiff in each column (5 columns)
- alldiff in each pandiagonal (10 pandiagonals)
- $PLS(5)$ is satisfiable.

# Boolean Cardinality Model

| | | | | |
|---|---|---|---|---|
| $y_{11k}$ | $y_{12k}$ | $y_{13k}$ | $y_{14k}$ | $y_{15k}$ |
| $y_{21k}$ | $y_{22k}$ | $y_{23k}$ | $y_{24k}$ | $y_{25k}$ |
| $y_{31k}$ | $y_{32k}$ | $y_{33k}$ | $y_{34k}$ | $y_{35k}$ |
| $y_{41k}$ | $y_{42k}$ | $y_{43k}$ | $y_{44k}$ | $y_{45k}$ |
| $y_{51k}$ | $y_{52k}$ | $y_{53k}$ | $y_{54k}$ | $y_{55k}$ |

- $y_{ijk} \in \{0, 1\}$ $\qquad$ $y_{ijk} = 1 \Leftrightarrow k$ is placed at $(i, j)$

# Boolean Cardinality Model

| | | | | |
|---|---|---|---|---|
| $y_{11k}$ | $y_{12k}$ | $y_{13k}$ | $y_{14k}$ | $y_{15k}$ |
| $y_{21k}$ | $y_{22k}$ | $y_{23k}$ | $y_{24k}$ | $y_{25k}$ |
| $y_{31k}$ | $y_{32k}$ | $y_{33k}$ | $y_{34k}$ | $y_{35k}$ |
| $y_{41k}$ | $y_{42k}$ | $y_{43k}$ | $y_{44k}$ | $y_{45k}$ |
| $y_{51k}$ | $y_{52k}$ | $y_{53k}$ | $y_{54k}$ | $y_{55k}$ |

- $y_{ijk} \in \{0, 1\}$  $y_{ijk} = 1 \Leftrightarrow k$ is placed at $(i, j)$
- for each value (5 values)
  - for each row (5 rows)  $y_{i1k} + y_{i2k} + y_{i3k} + y_{i4k} + y_{i5k} = 1$

# Boolean Cardinality Model

| | | | | |
|---|---|---|---|---|
| $y_{11k}$ | $y_{12k}$ | $y_{13k}$ | $y_{14k}$ | $y_{15k}$ |
| $y_{21k}$ | $y_{22k}$ | $y_{23k}$ | $y_{24k}$ | $y_{25k}$ |
| $y_{31k}$ | $y_{32k}$ | $y_{33k}$ | $y_{34k}$ | $y_{35k}$ |
| $y_{41k}$ | $y_{42k}$ | $y_{43k}$ | $y_{44k}$ | $y_{45k}$ |
| $y_{51k}$ | $y_{52k}$ | $y_{53k}$ | $y_{54k}$ | $y_{55k}$ |

- $y_{ijk} \in \{0, 1\}$ $\qquad y_{ijk} = 1 \Leftrightarrow k$ is placed at $(i, j)$
- for each value (5 values)
  - for each row (5 rows) $\qquad y_{i1k} + y_{i2k} + y_{i3k} + y_{i4k} + y_{i5k} = 1$

# Boolean Cardinality Model

| | | | | |
|---|---|---|---|---|
| $y_{11k}$ | $y_{12k}$ | $y_{13k}$ | $y_{14k}$ | $y_{15k}$ |
| $y_{21k}$ | $y_{22k}$ | $y_{23k}$ | $y_{24k}$ | $y_{25k}$ |
| $y_{31k}$ | $y_{32k}$ | $y_{33k}$ | $y_{34k}$ | $y_{35k}$ |
| $y_{41k}$ | $y_{42k}$ | $y_{43k}$ | $y_{44k}$ | $y_{45k}$ |
| $y_{51k}$ | $y_{52k}$ | $y_{53k}$ | $y_{54k}$ | $y_{55k}$ |

- $y_{ijk} \in \{0, 1\}$ $\quad$ $y_{ijk} = 1 \Leftrightarrow k$ is placed at $(i, j)$
- for each value (5 values)
  - for each row (5 rows) $\quad$ $y_{i1k} + y_{i2k} + y_{i3k} + y_{i4k} + y_{i5k} = 1$
  - for each column (5 columns) $\quad$ $y_{1jk} + y_{2jk} + y_{3jk} + y_{4jk} + y_{5jk} = 1$

# Boolean Cardinality Model

| | | | | |
|---|---|---|---|---|
| $y_{11k}$ | $y_{12k}$ | $y_{13k}$ | $y_{14k}$ | $y_{15k}$ |
| $y_{21k}$ | $y_{22k}$ | $y_{23k}$ | $y_{24k}$ | $y_{25k}$ |
| $y_{31k}$ | $y_{32k}$ | $y_{33k}$ | $y_{34k}$ | $y_{35k}$ |
| $y_{41k}$ | $y_{42k}$ | $y_{43k}$ | $y_{44k}$ | $y_{45k}$ |
| $y_{51k}$ | $y_{52k}$ | $y_{53k}$ | $y_{54k}$ | $y_{55k}$ |

- $y_{ijk} \in \{0, 1\}$ $\quad$ $y_{ijk} = 1 \Leftrightarrow k$ is placed at $(i, j)$
- for each value (5 values)
  - for each row (5 rows) $\quad$ $y_{i1k} + y_{i2k} + y_{i3k} + y_{i4k} + y_{i5k} = 1$
  - for each column (5 columns) $\quad$ $y_{1jk} + y_{2jk} + y_{3jk} + y_{4jk} + y_{5jk} = 1$

# Boolean Cardinality Model

| | | | | |
|---|---|---|---|---|
| $y_{11k}$ | $y_{12k}$ | $y_{13k}$ | $y_{14k}$ | $y_{15k}$ |
| $y_{21k}$ | $y_{22k}$ | $y_{23k}$ | $y_{24k}$ | $y_{25k}$ |
| $y_{31k}$ | $y_{32k}$ | $y_{33k}$ | $y_{34k}$ | $y_{35k}$ |
| $y_{41k}$ | $y_{42k}$ | $y_{43k}$ | $y_{44k}$ | $y_{45k}$ |
| $y_{51k}$ | $y_{52k}$ | $y_{53k}$ | $y_{54k}$ | $y_{55k}$ |

- $y_{ijk} \in \{0, 1\}$ $\qquad y_{ijk} = 1 \Leftrightarrow k$ is placed at $(i, j)$
- for each value (5 values)
  - for each row (5 rows) $\quad y_{i1k} + y_{i2k} + y_{i3k} + y_{i4k} + y_{i5k} = 1$
  - for each column (5 columns) $\quad y_{1jk} + y_{2jk} + y_{3jk} + y_{4jk} + y_{5jk} = 1$
  - for each pandiagonal (10 pandiagonals) $\quad y_{11k} + y_{22k} + y_{33k} + y_{44k} + y_{55k} = 1$

# Boolean Cardinality Model

| | | | | |
|---|---|---|---|---|
| $y_{11k}$ | $y_{12k}$ | $y_{13k}$ | $y_{14k}$ | $y_{15k}$ |
| $y_{21k}$ | $y_{22k}$ | $y_{23k}$ | $y_{24k}$ | $y_{25k}$ |
| $y_{31k}$ | $y_{32k}$ | $y_{33k}$ | $y_{34k}$ | $y_{35k}$ |
| $y_{41k}$ | $y_{42k}$ | $y_{43k}$ | $y_{44k}$ | $y_{45k}$ |
| $y_{51k}$ | $y_{52k}$ | $y_{53k}$ | $y_{54k}$ | $y_{55k}$ |

- $y_{ijk} \in \{0, 1\}$ $\qquad y_{ijk} = 1 \Leftrightarrow k$ is placed at $(i, j)$
- for each value (5 values)
  - for each row (5 rows) $\quad y_{i1k} + y_{i2k} + y_{i3k} + y_{i4k} + y_{i5k} = 1$
  - for each column (5 columns) $\quad y_{1jk} + y_{2jk} + y_{3jk} + y_{4jk} + y_{5jk} = 1$
  - for each pandiagonal (10 pandiagonals) $\quad y_{11k} + y_{22k} + y_{33k} + y_{44k} + y_{55k} = 1$

# Boolean Cardinality Model

| | | | | |
|---|---|---|---|---|
| $y_{11k}$ | $y_{12k}$ | $y_{13k}$ | $y_{14k}$ | $y_{15k}$ |
| $y_{21k}$ | $y_{22k}$ | $y_{23k}$ | $y_{24k}$ | $y_{25k}$ |
| $y_{31k}$ | $y_{32k}$ | $y_{33k}$ | $y_{34k}$ | $y_{35k}$ |
| $y_{41k}$ | $y_{42k}$ | $y_{43k}$ | $y_{44k}$ | $y_{45k}$ |
| $y_{51k}$ | $y_{52k}$ | $y_{53k}$ | $y_{54k}$ | $y_{55k}$ |

- $y_{ijk} \in \{0, 1\}$ $\quad$ $y_{ijk} = 1 \Leftrightarrow k$ is placed at $(i, j)$
- for each value (5 values)
  - for each row (5 rows) $\quad y_{i1k} + y_{i2k} + y_{i3k} + y_{i4k} + y_{i5k} = 1$
  - for each column (5 columns) $\quad y_{1jk} + y_{2jk} + y_{3jk} + y_{4jk} + y_{5jk} = 1$
  - for each pandiagonal (10 pandiagonals) $\quad y_{11k} + y_{22k} + y_{33k} + y_{44k} + y_{55k} = 1$

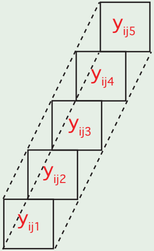
# Boolean Cardinality Model

| | | | | |
|---|---|---|---|---|
| $y_{11k}$ | $y_{12k}$ | $y_{13k}$ | $y_{14k}$ | $y_{15k}$ |
| $y_{21k}$ | $y_{22k}$ | $y_{23k}$ | $y_{24k}$ | $y_{25k}$ |
| $y_{31k}$ | $y_{32k}$ | $y_{33k}$ | $y_{34k}$ | $y_{35k}$ |
| $y_{41k}$ | $y_{42k}$ | $y_{43k}$ | $y_{44k}$ | $y_{45k}$ |
| $y_{51k}$ | $y_{52k}$ | $y_{53k}$ | $y_{54k}$ | $y_{55k}$ |

- $y_{ijk} \in \{0, 1\}$ $\qquad y_{ijk} = 1 \Leftrightarrow k$ is placed at $(i, j)$
- for each value (5 values)
  - for each row (5 rows) $\qquad y_{i1k} + y_{i2k} + y_{i3k} + y_{i4k} + y_{i5k} = 1$
  - for each column (5 columns) $\qquad y_{1jk} + y_{2jk} + y_{3jk} + y_{4jk} + y_{5jk} = 1$
  - for each pandiagonal (10 pandiagonals) $\qquad y_{11k} + y_{22k} + y_{33k} + y_{44k} + y_{55k} = 1$

# Boolean Cardinality Model

| | | | | |
|---|---|---|---|---|
| $y_{11k}$ | $y_{12k}$ | $y_{13k}$ | $y_{14k}$ | $y_{15k}$ |
| $y_{21k}$ | $y_{22k}$ | $y_{23k}$ | $y_{24k}$ | $y_{25k}$ |
| $y_{31k}$ | $y_{32k}$ | $y_{33k}$ | $y_{34k}$ | $y_{35k}$ |
| $y_{41k}$ | $y_{42k}$ | $y_{43k}$ | $y_{44k}$ | $y_{45k}$ |
| $y_{51k}$ | $y_{52k}$ | $y_{53k}$ | $y_{54k}$ | $y_{55k}$ |




- $y_{ijk} \in \{0, 1\}$ $\qquad y_{ijk} = 1 \Leftrightarrow k$ is placed at $(i, j)$
- for each value (5 values)
  - for each row (5 rows) $\qquad y_{i1k} + y_{i2k} + y_{i3k} + y_{i4k} + y_{i5k} = 1$
  - for each column (5 columns) $\qquad y_{1jk} + y_{2jk} + y_{3jk} + y_{4jk} + y_{5jk} = 1$
  - for each pandiagonal (10 pandiagonals) $\qquad y_{11k} + y_{22k} + y_{33k} + y_{44k} + y_{55k} = 1$
- for each $(i, j)$ position (25 positions) $\qquad y_{ij1} + y_{ij2} + y_{ij3} + y_{ij4} + y_{ij5} = 1$

# Experiments

## Comparison on Solving Pandiagonal Latin Square

To show the differences in performance, we compared the following 5 models.

1. AD1: naive alldiff
2. AD2: optimized alldiff
3. BC1: Pairwise
4. BC2: [Bailleux '03]
5. BC3: [Sinz '05]

## Benchmark and Experimental Environment

- Benchmark: Pandiagonal Latin Square ($n = 7$ to $n = 16$)
- CPU: 2.93GHz, Mem: 2GB, Time Limit: 3600 seconds

# Results (CPU Time in Seconds)

| n | SAT/UNSAT | AD1 | AD2 | BC1 | BC2 | BC3 |
|---|---|---|---|---|---|---|
| 7 | SAT | 0.2 | 0.2 | 0.2 | 0.3 | 0.3 |
| 8 | UNSAT | T.O. | 0.5 | 0.3 | 0.3 | 0.3 |
| 9 | UNSAT | T.O. | 0.3 | 0.5 | 0.3 | 0.2 |
| 10 | UNSAT | T.O. | 0.4 | 1.0 | 0.3 | 0.3 |
| 11 | SAT | 0.3 | 0.3 | 2.3 | 0.5 | 0.4 |
| 12 | UNSAT | T.O. | 1.0 | 5.3 | 0.8 | 0.8 |
| 13 | SAT | T.O. | 0.5 | T.O. | T.O. | T.O. |
| 14 | UNSAT | T.O. | 9.7 | 32.4 | 8.2 | 6.8 |
| 15 | UNSAT | T.O. | 388.9 | 322.7 | 194.6 | 155.8 |
| 16 | UNSAT | T.O. | 457.1 | 546.6 | 300.7 | 414.8 |

- Only optimized version of alldiff model (AD2) solved all instances.
- Modeling and encoding have an important role in developing SAT-based systems. Just using SAT solvers is not enough!
- Scarab helps users to focus on them ;)

# Experiments on Hamiltonian Cycle Problem

We evaluate the effectiveness of (1) CEGAR-HCP, (2) Native BC, (3) Implementation on Tightly Integrated System.
We also have (4) a comparison with other specialized methods.

## Machine Spec and Benchmark

- CPU: Intel Xeon 2.93GHz, Memory: 4GB, Time Limit: 500 sec.
- **color04** (119 instances, #nodes: 11 to 10000),
  **knight** (11 instances, 8x8 to 100x100), **tsplib** (9 instances)

## Systems Compared

- CEGAR-HCP (on Scarab)
  **S4J-S** (Seq. Counter), **S4J-N** (Native BC)

# Experiments on Hamiltonian Cycle Problem

We evaluate the effectiveness of (1) CEGAR-HCP, (2) Native BC, (3) Implementation on Tightly Integrated System.
We also have (4) a comparison with other specialized methods.

## Machine Spec and Benchmark

- CPU: Intel Xeon 2.93GHz, Memory: 4GB, Time Limit: 500 sec.
- **color04** (119 instances, #nodes: 11 to 10000),
  **knight** (11 instances, 8x8 to 100x100), **tsplib** (9 instances)

## Systems Compared

- CEGAR-HCP (on Scarab)
  **S4J-S** (Seq. Counter), **S4J-N** (Native BC)
- Eager Method **Velev** (Minisat2.2)

# Experiments on Hamiltonian Cycle Problem

We evaluate the effectiveness of (1) CEGAR-HCP, (2) Native BC, (3) Implementation on Tightly Integrated System.
We also have (4) a comparison with other specialized methods.

## Machine Spec and Benchmark

- CPU: Intel Xeon 2.93GHz, Memory: 4GB, Time Limit: 500 sec.
- **color04** (119 instances, #nodes: 11 to 10000),
  **knight** (11 instances, 8x8 to 100x100), **tsplib** (9 instances)

## Systems Compared

- CEGAR-HCP (on Scarab)
  **S4J-S** (Seq. Counter), **S4J-N** (Native BC)
- Eager Method **Velev** (Minisat2.2)
- Specialized TSP Solver **LKH**
  (It holds all world records of TSP in TSPLIB)

# Experiments on Hamiltonian Cycle Problem

We evaluate the effectiveness of (1) CEGAR-HCP, (2) Native BC, (3) Implementation on Tightly Integrated System.
We also have (4) a comparison with other specialized methods.

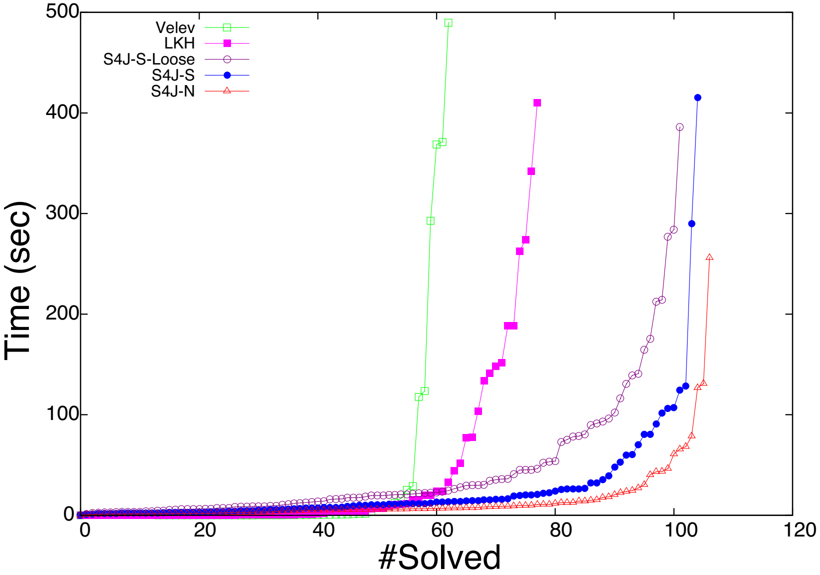
## Machine Spec and Benchmark

- CPU: Intel Xeon 2.93GHz, Memory: 4GB, Time Limit: 500 sec.
- **color04** (119 instances, #nodes: 11 to 10000),
  **knight** (11 instances, 8x8 to 100x100), **tsplib** (9 instances)

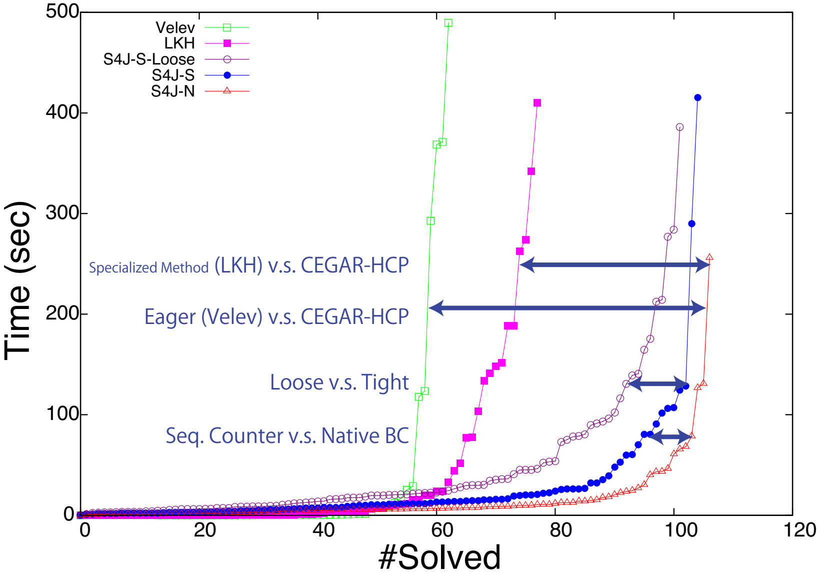
## Systems Compared

- CEGAR-HCP (on Scarab)
  **S4J-S** (Seq. Counter), **S4J-N** (Native BC)
- Eager Method **Velev** (Minisat2.2)
- Specialized TSP Solver **LKH**
  (It holds all world records of TSP in TSPLIB)
- CEGAR-HCP (on loosely integrated system) **S4J-S-Loose**

# Cactus Plot (#Solved–CPU Time)



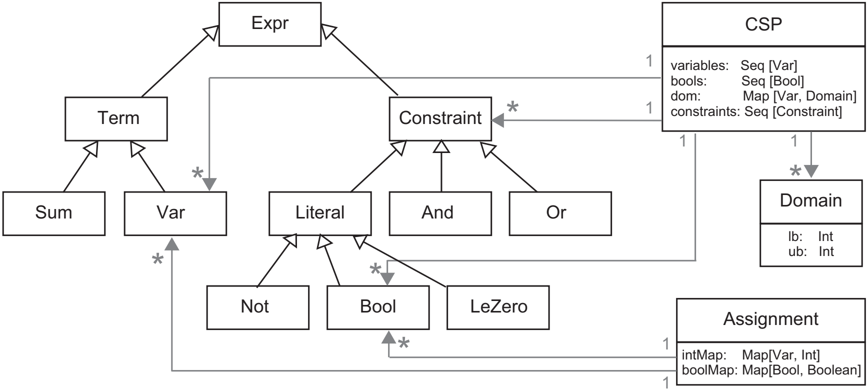
# Cactus Plot (#Solved–CPU Time)



# Features of Scarab

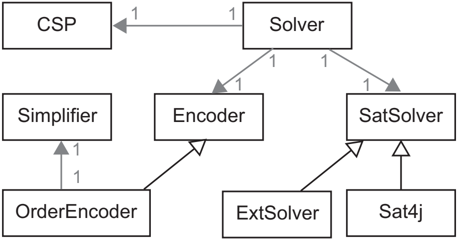
- **Efficiency**
  - Scarab is efficient in the sense that it uses an optimized version of the order encoding for encoding CSP into SAT.
- **Portability**
  - The combination of Scarab and Sat4j enables the development of portable applications on JVM (Java Virtual Machine).
- **Customizability**
  - Scarab is 800 lines long without comments.
  - Core of order encoding module is only 25 lines long.
  - It allows programmers to freely customize Scarab itself.
- **Availability of Advanced SAT Techniques**
  - Thanks to the tight integration to Sat4j, it is available to use several SAT techniques, *e.g.*, incremental SAT solving and native handling constraints.

# Class Diagrams




Class Diagrams for CSPs

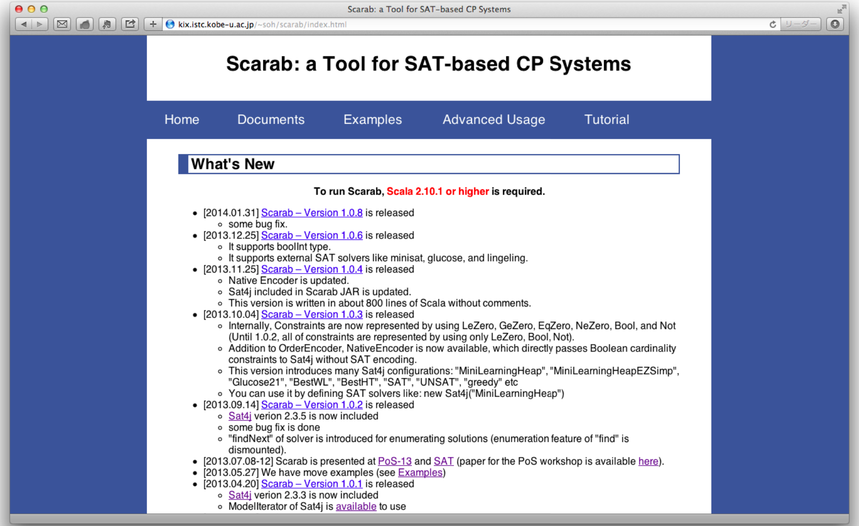
# Class Diagrams




Class Diagrams for Solvers

# Web Page for Scarab

http://kix.istc.kobe-u.ac.jp/~soh/scarab/



Scarab: a Tool for SAT-based CP Systems

kix.istc.kobe-u.ac.jp/~soh/scarab/index.html

# Scarab: a Tool for SAT-based CP Systems

Home | Documents | Examples | Advanced Usage | Tutorial

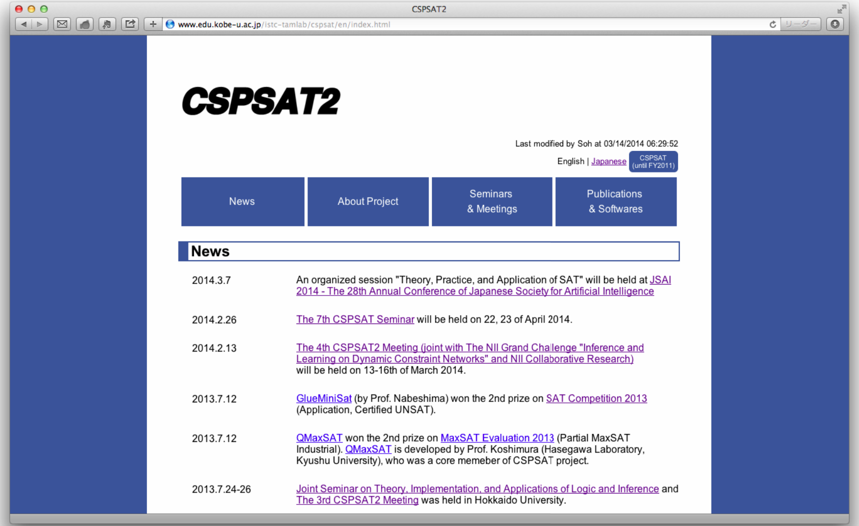
## What's New

**To run Scarab, Scala 2.10.1 or higher is required.**

- [2014.01.31] Scarab – Version 1.0.8 is released
  - some bug fix.
- [2013.12.25] Scarab – Version 1.0.6 is released
  - It supports boolInt type.
  - It supports external SAT solvers like minisat, glucose, and lingeling.
- [2013.11.25] Scarab – Version 1.0.4 is released
  - Native Encoder is updated.
  - Sat4j included in Scarab JAR is updated.
  - This version is written in about 800 lines of Scala without comments.
- [2013.10.04] Scarab – Version 1.0.3 is released
  - Internally, Constraints are now represented by using LeZero, GeZero, EqZero, NeZero, Bool, and Not (Until 1.0.2, all of constraints are represented by using only LeZero, Bool, Not).
  - Addition to OrderEncoder, NativeEncoder is now available, which directly passes Boolean cardinality constraints to Sat4j without SAT encoding.
  - This version introduces many Sat4j configurations: "MiniLearningHeap", "MiniLearningHeapEZSimp", "Glucose21", "BestWL", "BestHT", "SAT", "UNSAT", "greedy" etc
  - You can use it by defining SAT solvers like: new Sat4j("MiniLearningHeap")
- [2013.09.14] Scarab – Version 1.0.2 is released
  - Sat4j verion 2.3.5 is now included
  - some bug fix is done
  - "findNext" of solver is introduced for enumerating solutions (enumeration feature of "find" is dismounted).
- [2013.07.08-12] Scarab is presented at PoS-13 and SAT (paper for the PoS workshop is available here).
- [2013.05.27] We have move examples (see Examples)
- [2013.04.20] Scarab – Version 1.0.1 is released
  - Sat4j verion 2.3.3 is now included
  - ModelIterator of Sat4j is available to use

# Web Page for CSPSAT2

http://www.edu.kobe-u.ac.jp/istc-tamlab/cspsat/en/



# Conclusion

- Introducing Architecture and Features of Scarab
- Using Scarab, we can write various constraint models without developing dedicated encoders, which allows us to focus on problem modeling and encoding.
- **Future Work**
  - Introducing more features from Sat4j
  - Sat4j has various functions of finding MUS, optimization, solution enumeration, handling natively cardinality and pseudo-Boolean constraints.
- URL of Scarab `http://kix.istc.kobe-u.ac.jp/~soh/scarab/`

# References I


Biere, A., Cimatti, A., Clarke, E. M., and Zhu, Y. (1999).
Symbolic model checking without BDDs.
In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 1999), LNCS 1579*, pages 193–207.

Cook, S. A. (1971).
The complexity of theorem-proving procedures.
In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (STOC 1971)*, pages 151–158.

Crawford, J. M. and Baker, A. B. (1994).
Experimental results on the application of satisfiability algorithms to scheduling problems.
In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI 1994)*, pages 1092–1097.

# References II


Davis, M., Logemann, G., and Loveland, D. W. (1962).
A machine program for theorem-proving.
*Communications of the ACM*, 5(7):394–397.

de Kleer, J. (1989).
A comparison of ATMS and CSP techniques.
In *Proceedings of the 11th International Joint Conference on Artificial Intelligence (IJCAI 1989)*, pages 290–296.

Gavanelli, M. (2007).
The log-support encoding of CSP into SAT.
In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP 2007), LNCS 4741*, pages 815–822.

Iwama, K. and Miyazaki, S. (1994).
SAT-variable complexity of hard combinatorial problems.
In *Proceedings of the IFIP 13th World Computer Congress*, pages 253–258.

Kasif, S. (1990).
On the parallel complexity of discrete relaxation in constraint satisfaction networks.
*Artificial Intelligence*, 45(3):275–286.

Kautz, H. A. and Selman, B. (1992).
Planning as satisfiability.
In *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI 1992)*, pages 359–363.

# References IV


Petke, J. and Jeavons, P. (2011).
The order encoding: From tractable csp to tractable sat.
In *Proceedings of the 14th International Conference on Theory and Applications of Satisfiability Testing (SAT 2011), LNCS 6695*, pages 371–372.

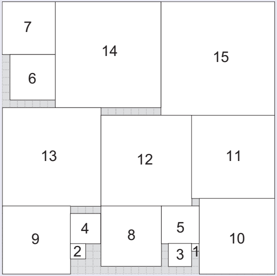
Selman, B., Levesque, H. J., and Mitchell, D. G. (1992).
A new method for solving hard satisfiability problems.
In *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI 1992)*, pages 440–446.

Tamura, N., Taga, A., Kitagawa, S., and Banbara, M. (2006).
Compiling finite linear CSP into SAT.
In *Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming (CP 2006), LNCS 4204*, pages 590–603.

# Example: Square Packing

- **Square Packing** $SP(n, s)$ is a problem of packing a set of squares of sizes $1 \times 1$ to $n \times n$ into an enclosing square of size $s \times s$ without overlapping.

## Example of $SP(15, 36)$




- **Optimum soluiton of** $SP(n, s)$ is the smallest size of the enclosing square having a feasible packing.

# Non-overlapping Constraint Model for $SP(n, s)$

**Integer variables**

- $x_i \in \{0, \ldots, s - i\}$ and $y_i \in \{0, \ldots, s - i\}$
- Each pair $(x_i,\ y_i)$ represents the lower left coordinates of the square $i$.

**Non-overlapping Constraint $(1 \leq i < j \leq n)$**

$$(x_i + i \leq x_j) \vee (x_j + j \leq x_i) \vee (y_i + i \leq y_j) \vee (y_j + j \leq y_i)$$

# Decremental Seach

## Scarab Program for $SP(n, s)$

```
for (i <- 1 to n) { int('x(i),0,s-i) ; int('y(i),0,s-i) }
for (i <- 1 to n; j <- i+1 to n)
  add(('x(i) + i <= 'x(j)) || ('x(j) + j <= 'x(i)) ||
      ('y(i) + i <= 'y(j)) || ('y(j) + j <= 'y(i)))
```

## Searching an Optimum Solution

```
val lb = n; var ub = s; int('m, lb, ub)
for (i <- 1 to n)
  add(('x(i)+i <= 'm) && ('y(i)+i <= 'm))

// Incremental solving
while (lb <= ub && find('m <= ub)) { // using an assumption.
  add('m <= ub)
  ub = solution.intMap('m) - 1
}
```

# Bisection Search

## Bisection Search

```
var lb = n; var ub = s; commit

while (lb < ub) {
  var size = (lb + ub) / 2
  for (i <- 1 to n)
    add(('x(i)+i<=size)&&('y(i)+i<=size))
  if (find) {
    ub = size
    commit // commit current constraints
  } else {
    lb = size + 1
    rollback // rollback to the last commit point
  }
}
```

# Advanced Solving Techniques using Sat4j

- Thanks to the **tight integration to Sat4j**, Scarab provides the functions: Incremental solving and CSP solving with assumptions.
- We explain it using the following program.

```
int('x, 1, 3)
int('y, 1, 3)
add('x === 'y)
find                // first call of find
add('x !== 3)
find                // second call of find

find('y === 3)      // with assumption y = 3
find('x === 1)      // with assumption x = 1
```

# Incremental SAT Solving

```
int('x, 1, 3)
int('y, 1, 3)
add('x === 'y)
find                // first call of find
add('x !== 3)
find                // second call of find
```

- In the first call of **find** method, the whole CSP is encoded and generated SAT clauses are added to Sat4j, then it computes a solution.
- In the second call of **find** method, only the extra constraint $x \neq 3$ is encoded and added to Sat4j, then it computes a solution.
- The learned clauses obtained by the first **find** are kept at the second call.

# CSP Solving under Assumption

```
find('y === 3)    // with assumption y = 3
find('x === 1)    // with assumption x = 1
```

- **find(assumption: Constraint)** method provides CSP solving under assumption given by the specified constraint.
- The constraint of assumption should be encoded to a conjunction of literals (otherwise an exception is raised).
- Then, the literals are passed to Sat4j, then it computes a solution under assumption.
- We can utilize those techniques for optimization and enumeration problems.

# Scarab Program for alldiff Model

```
import jp.kobe_u.scarab._ ; import dsl._

val n = args(0).toInt

for (i <- 1 to n; j <- 1 to n)  int('x(i,j),1,n)
for (i <- 1 to n) {
 add(alldiff((1 to n).map(j => 'x(i,j))))
 add(alldiff((1 to n).map(j => 'x(j,i))))
 add(alldiff((1 to n).map(j => 'x(j,(i+j-1)%n+1))))
 add(alldiff((1 to n).map(j => 'x(j,(i+(j-1)*(n-1))%n+1))))
}

if (find)  println(solution)
```

# Encoding alldiff

- In Scarab, all we have to do for implementing global constraints is just decomposing them into simple arithmetic constraints [Bessiere et al. '09].

**In the case of alldiff($a_1, \ldots, a_n$),**

It is decomposed into pairwise not-equal constraints

$$\bigwedge_{1 \le i < j \le n} (a_i \neq a_j)$$

.

- This (naive) alldiff is enough to just have a feasible constraint model for $PLS(n)$.
- But, one probably want to improve this :)

# Extra Constraints for alldiff$(a_1, \ldots, a_n)$

- In Pandiagonal Latin Square $PLS(n)$, all integer variables $a_1, \ldots, a_n$ have the same domain $\{1, \ldots, n\}$.
- Then, we can add the following extra constraints.
- **Permutation constraints**:

$$\bigwedge_{i=1}^{n} \bigvee_{j=1}^{n} (a_j = i)$$

  - It represents that one of $a_1, \ldots, a_n$ must be assigned to $i$.
- **Pigeon hole constraint**:

$$\neg \bigwedge_{i=1}^{n} (a_i < n) \wedge \neg \bigwedge_{i=1}^{n} (a_i > 1)$$

  - It represents that mutually different $n$ variables cannot be assigned within the interval of the size $n - 1$.

# alldiff (naive)

```
def alldiff(xs: Seq[Var]) =
  And(for (Seq(x, y) <- xs.combinations(2))
      yield x !== y)
```

# alldiff (optimized)

```
def alldiff(xs: Seq[Var]) = {
  val lb = for (x <- xs) yield csp.dom(x).lb
  val ub = for (x <- xs) yield csp.dom(x).ub
  // pigeon hole
  val ph =
    And(Or(for (x <- xs) yield !(x < lb.min+xs.size-1)),
        Or(for (x <- xs) yield !(x > ub.max-xs.size+1)))
  // permutation
  def perm =
    And(for (num <- lb.min to ub.max)
        yield Or(for (x <- xs) yield x === num))
  val extra = if (ub.max-lb.min+1 == xs.size) And(ph,perm)
              else ph

  And(And(for (Seq(x, y) <- xs.combinations(2))
          yield x !== y),extra)
}
```

# Scarab Program for Boolean Cardinality Model

```
import jp.kobe_u.scarab._ ; import dsl._

for (i <- 1 to n; j <- 1 to n; num <- 1 to n)
  int('y(i,j,num),0,1)

for (num <- 1 to n) {
  for (i <- 1 to n) {
   add(BC((1 to n).map(j => 'y(i,j,num)))===1)
   add(BC((1 to n).map(j => 'y(j,i,num)))===1)
   add(BC((1 to n).map(j => 'y(j,(i+j-1)%n+1,num))) === 1)
   add(BC((1 to n).map(j => 'y(j,(i+(j-1)*(n-1))%n+1,num))) === 1)
  }
}

for (i <- 1 to n; j <- 1 to n)
  add(BC((1 to n).map(k => 'y(i,j,k))) === 1)

if (find)  println(solution)
```

# SAT Encoding of Boolean Cardinality in Scarab

- There are several ways for encoding Boolean cardinality.
- In Scarab, we can easily write the following encoding methods by defining your own BC methods.
  - Pairwise
  - Totalizer [Bailleux '03]
  - Sequential Counter [Sinz '05]
- In total, **3 variants of Boolean cardinality model** are obtained.
  - BC1: Pairwise (implemented by 2 lines)
  - BC2: Totalizer [Bailleux '03] (implemented by 15 lines)
  - BC3: Sequential Counter [Sinz '05] (implemented by 7 lines)
- Good point to use Scarab is that we can test those models **without writing dedicated programs**.