Constraint Programming Illustrated on the Subgraph Isomorphism Problem

Christine Solnon
CITI, INSA Lyon / Inria

14th IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition

What is Constraint Programming (CP)?

"Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it."

Eugene C. Freuder

In other words: CP = Model + Search

- Model $\rightsquigarrow$ Define a mathematical model by means of variables and constraints
- Search $\rightsquigarrow$ Use a generic solver to search for a solution

What is the difference with ILP (Integer Linear Programming) or SAT?

- Richer modelling language
  - ILP: Model = Linear inequations + Linear objective function
  - SAT: Model = Boolean formula
  - CP: Model = Conjunction of various kinds of constraints
    $\rightsquigarrow$ Each constraint comes with its own propagation algorithm
    $\rightsquigarrow$ If your favorite constraint does not exist, you can create it!
- Different solving approach: Branch & Propagate (vs Branch & Bound/Cut/Price for ILP or CDCL for SAT)

What is Constraint Programming (CP)?

"Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it."

Eugene C. Freuder

In other words: CP = Model + Search

- Model $\rightsquigarrow$ Define a mathematical model by means of variables and constraints
- Search $\rightsquigarrow$ Use a generic solver to search for a solution

What is the difference with ILP (Integer Linear Programming) or SAT?

- Richer modelling language
  - ILP: Model = Linear inequations + Linear objective function
  - SAT: Model = Boolean formula
  - CP: Model = Conjunction of various kinds of constraints
    $\rightsquigarrow$ Each constraint comes with its own propagation algorithm
    $\rightsquigarrow$ If your favorite constraint does not exist, you can create it!
- Different solving approach: Branch & Propagate (vs Branch & Bound/Cut/Price for ILP or CDCL for SAT)

Overview of the talk

1. **Modelling with CP**
2. Generic CP Solving Algorithms
3. LAD2025: A Constraint-based Solver for the SIP
4. Conclusion

Modelling with CP

CP model = $(X, D, C)[+F]$

- X = Set of variables (unknowns)
- For each variable $x_i \in X$, $D(x_i)$ = domain of x_i
  $\rightsquigarrow$ Set of values that may be assigned to x_i
- C = Constraints (relations between variables of X)
- [Optionally] $F : X \to \mathbb{R}$ = objective function to optimize

Solution of a CP model:

Assignment of a value to every variable of X such that:

- Each variable $x_i \in X$ is assigned to a value that belongs to $D(x_i)$
- Every constraint of C is satisfied
- [Optionally] F is maximized (or minimized)

Remark: A problem may have several different models...

Example: Subgraph Isomorphism Problem (SIP)

Definition of the SIP:

Given $G_p = (V_p, E_p)$ and $G_t = (V_t, E_t)$:
find an injective mapping $f : V_p \to V_t$ such that
$\forall (i,j) \in E_p, (f(i), f(j)) \in E_t$

Example of SIP instance:

$G_p = (V_p, E_p)$ $G_t = (V_t, E_t)$

1

Example: Subgraph Isomorphism Problem (SIP)

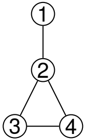
Definition of the SIP:

Given $G_p = (V_p, E_p)$ and $G_t = (V_t, E_t)$:
find an injective mapping $f : V_p \to V_t$ such that
$\forall (i, j) \in E_p, (f(i), f(j)) \in E_t$

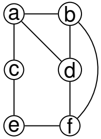
CP model introduced by Ullmann in 1976[1]:

- Variables: $X = \{x_i | i \in V_p\}$
- Domains:
  $\forall i \in V_p, D(x_i) = \{u \in V_t : d^\circ(i) \leq d^\circ(u)\}$
- Constraints:
  - f must be injective: $\forall \{i, j\} \subseteq V_p, x_i \neq x_j$
  - Edge constraints:
    $\forall (i, j) \in E_p, (x_i, x_j) \in E_t$

Example of SIP instance:



$G_p = (V_p, E_p)$



$G_t = (V_t, E_t)$

Domains:

- $D(x_1) = D(x_3) = D(x_4) = \{a, b, c, d, e, f\}$
- $D(x_2) = \{a, b, d, f\}$

1 Ullmann: *An algorithm for subgraph isomorphism*, in J. ACM, 1976

Example: Subgraph Isomorphism Problem (SIP)

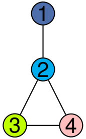
Definition of the SIP:

Given $G_p = (V_p, E_p)$ and $G_t = (V_t, E_t)$:
find an injective mapping $f : V_p \rightarrow V_t$ such that
$\forall (i, j) \in E_p, (f(i), f(j)) \in E_t$

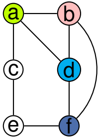
CP model introduced by Ullmann in 1976[1]:

- Variables: $X = \{x_i | i \in V_p\}$
- Domains:
  $\forall i \in V_p, D(x_i) = \{u \in V_t : d^\circ(i) \leq d^\circ(u)\}$
- Constraints:
  - f must be injective: $\forall \{i, j\} \subseteq V_p, x_i \neq x_j$
  - Edge constraints:
    $\forall (i, j) \in E_p, (x_i, x_j) \in E_t$

Example of SIP instance:



$G_p = (V_p, E_p)$



$G_t = (V_t, E_t)$

Domains:

- $D(x_1) = D(x_3) = D(x_4) = \{a, b, c, d, e, f\}$
- $D(x_2) = \{a, b, d, f\}$

Example of solution:

$x_1 = f$, $x_2 = d$, $x_3 = a$, $x_4 = b$

1 Ullmann: *An algorithm for subgraph isomorphism*, in J. ACM, 1976

Example: Subgraph Isomorphism Problem (SIP)

Definition of the SIP:

Given $G_p = (V_p, E_p)$ and $G_t = (V_t, E_t)$:
find an injective mapping $f : V_p \rightarrow V_t$ such that
$\forall (i,j) \in E_p, (f(i), f(j)) \in E_t$

Other CP model with binary variables:

- Variables: $X = \{x_{ij} | i \in V_p, j \in V_t\}$
- Domains: $\forall i \in V_p, \forall j \in V_t, D(x_{ij}) = \{0, 1\}$
- Constraints:
  - f must be injective:
    $\forall i \in V_p, \sum_j x_{ij} = 1 \wedge \forall j \in V_t, \sum_i x_{ij} \leq 1$
  - Edge constraints:
    $\forall (i_1, i_2) \in E_p, \forall (j_1, j_2) \in V_t \times V_t \setminus E_t, x_{i_1 j_1} + x_{i_2 j_2} < 2$

Example of SIP instance:

$G_p = (V_p, E_p)$ $\quad$ $G_t = (V_t, E_t)$

Example of solution:

- $x_{1f} = x_{2d} = x_{3a} = x_{4b} = 1$
- All other variables are set to 0

1 Ullmann: *An algorithm for subgraph isomorphism*, in J. ACM, 1976

CP Languages and Libraries

First CP language introduced by Jean-Louis Laurière in 1976:

ALICE

Extensions of Prolog:

CHIP, Prolog V, Gnu-Prolog, Sicstus Prolog, Picat, ...

Libraries:

- Open source: Choco (Java), PyCSP3 (Python), OR-Tools/Google (C++, Java, Python, ...), etc
- Commercial: CP-Optimizer (IBM)

Modelling languages (accepted by most solvers):

MiniZinc, XCSP3

Example: SIP in PyCSP3 (using a notebook)

Define the pattern graph and the target graph

```
[3]  n_p = 4; n_t = 6
     e_p = [(0,1), (0,3), (1,2), (1,3), (2,3)]
     e_t = [(0,1), (0,4), (1,0), (1,2), (1,3), (2,1), (2,3), (2,5), (3,1), (3,2), (3,4), (4,0), (4,3), (5,1), (5,2)]
     deg_p = [2, 3, 2, 3]
     deg_t = [2, 4, 3, 3, 2, 2]
```

Declare variables and domains

```
[4]  x = VarArray(size=n_p, dom=range(n_t))
     for i in range(n_p) :
       x[i] in {j for j in range(n_t) if deg_p[i] <= deg_t[j]}
```

Declare constraints... and solve!

```
[6]  satisfy(AllDifferent(x),
             [(x[i], x[j]) in e_t for (i, j) in e_p])
     if solve() is SAT:
         print(values(x))
```

```
[5, 1, 3, 2]
```

OK, the model is nice (isn't it?)... but is it efficient?

Comparison of ACE[1] (default solver of PyCSP3) with RI[2] and LAD2025[3] on a subset of instances

Number of instances NOT solved within 100s:

	LV	rand	meshes	all
ACE default	85	30	46	161
ACE tuned	72	22	37	131
RI	152	8	284	324
LAD2025	32	5	1	38

- ACE solves more instances when tuning parameters
- ACE solves more instances than RI on LV and meshes
  But it is less successful on rand instances
- ACE is outperformed by LAD2025

1 Lecoutre: *ACE, a generic constraint solver*, arXiv, 2023

2 Bonnici, Giugno. *On the variable ordering in subgraph isomorphism algorithms*, in IEEE Trans. Bioinform., 2017

3 Solnon: *LAD2025, a Constraint-Based Solver for the Subgraph Isomorphism Problem*, 2025 (submitted)

OK, the model is nice (isn't it?)... but is it efficient?

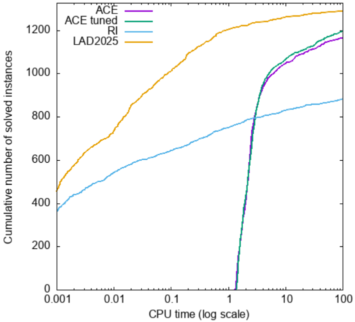
Comparison of ACE[1] (default solver of PyCSP3) with RI[2] and LAD2025[3] on a subset of instances

Number of instances NOT solved within 100s:

	LV	rand	meshes	all
ACE default	85	30	46	161
ACE tuned	72	22	37	131
RI	152	8	284	324
LAD2025	32	5	1	38

- ACE solves more instances when tuning parameters
- ACE solves more instances than RI on LV and meshes
  But it is less successful on rand instances
- ACE is outperformed by LAD2025

What about solving times?



Conclusion:

- ACE solves less instances than dedicated approaches when the time limit is smaller than 3s
- But it solves more instances than RI when the time limit is larger than 3s
- Easy to add application-specific constraints or objective functions

OK, the model is nice (isn't it?)... but is it efficient?

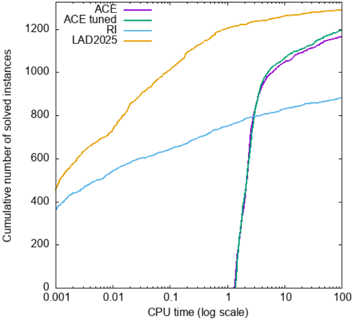
Comparison of ACE[1] (default solver of PyCSP3) with RI[2] and LAD2025[3] on a subset of instances

Number of instances NOT solved within 100s:

	LV	rand	meshes	all
ACE default	85	30	46	161
ACE tuned	72	22	37	131
RI	152	8	284	324
LAD2025	32	5	1	38

- ACE solves more instances when tuning parameters
- ACE solves more instances than RI on LV and meshes
  But it is less successful on rand instances
- ACE is outperformed by LAD2025

What about solving times?



Conclusion:

- ACE solves less instances than dedicated approaches when the time limit is smaller than 3s
- But it solves more instances than RI when the time limit is larger than 3s
- **Easy to add application-specific constraints or objective functions**

Overview of the talk

1. Modelling with CP
2. **Generic CP Solving Algorithms**
3. LAD2025: A Constraint-based Solver for the SIP
4. Conclusion

Branch & Propagate Solving Approach

Branch:

- Choose an unassigned variable x_i
- For each $v \in D(x_i)$, recursively solve a subproblem where v is assigned to x_i

$\leadsto$ Stop when a solution is found; Backtrack when an inconsistency is detected

Propagate constraints at each subproblem:

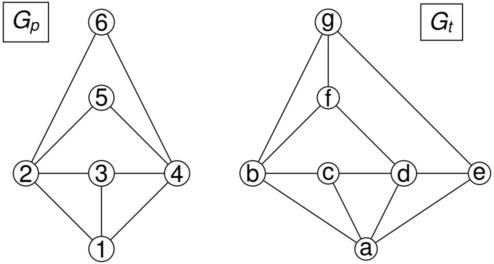
- Goal: Remove inconsistent values from variable domains
  $\leadsto$ Inconsistency detected whenever a domain is wiped out
- Each constraint comes with its own propagation algorithms
  $\leadsto$ Easy to add new kinds of constraints

A first (very simple) propagation: Forward Checking (FC)

Propagate a constraint whenever all its variables but one are assigned

$\rightsquigarrow$ Remove inconsistent values from the domains of unassigned variables

Example of SIP instance:



Initial domains:

- $D(x_1) = D(x_3) = D(x_5) = D(x_6) = \{a, b, c, d, e, f, g\}$
- $D(x_2) = D(x_4) = \{a, b, d\}$

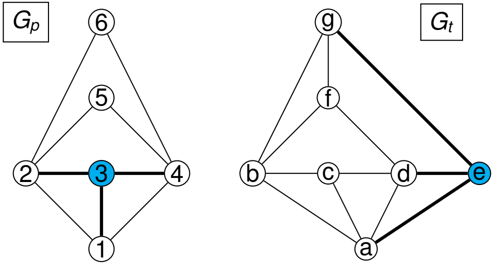
1

A first (very simple) propagation: Forward Checking (FC)

Propagate a constraint whenever all its variables but one are assigned

$\rightsquigarrow$ Remove inconsistent values from the domains of unassigned variables

Example of SIP instance:



Initial domains:

- $D(x_1) = D(x_3) = D(x_5) = D(x_6) = \{a, b, c, d, e, f, g\}$
- $D(x_2) = D(x_4) = \{a, b, d\}$

FC propagation when $x_3 = e$:

- $\forall i \in \{1, 5, 6\}, x_i \neq e$
  $\rightsquigarrow$ Remove e from $D(x_i)$
- $\forall i \in \{1, 2, 4\}, (x_i, e) \in E_t$
  $\rightsquigarrow$ Remove b, c, and f from $D(x_1)$
  $\rightsquigarrow$ Remove b from $D(x_2)$ and $D(x_4)$

This is similar to what is done in VF2[1]

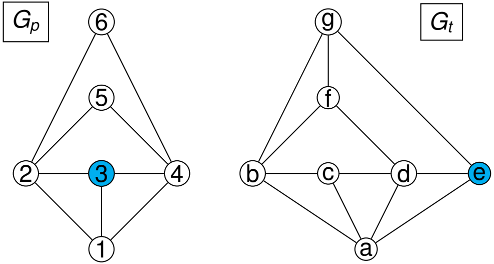
[1] Cordella, Foggia, Sansone, Vento. *A (sub)graph isomorphism algorithm for matching large graphs*, in PAMI, 2004

Stronger propagation: Ensuring Domain Consistency

A constraint c defined over a set $S \subseteq X$ of k variables is Domain Consistent (DC) if:

$\forall x_i \in S, \forall v_i \in D(x_i), \forall x_j \in S \setminus \{x_i\}, \exists v_j \in D(x_j)$ such that c is satisfied by the assignment $x_1 = v_1, ..., x_k = v_k$

Example of SIP instance:



Let's assume that $x_3 = e$.

Domains reduced by FC in this case:

- $D(x_1) = \{a, d, g\}$
- $D(x_2) = \{a, d\}$
- $D(x_4) = \{a, d\}$
- $D(x_5) = D(x_6) = \{a, b, c, d, f, g\}$

Can we do better?

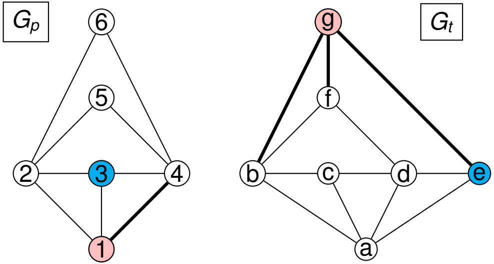
1

Stronger propagation: Ensuring Domain Consistency

A constraint c defined over a set $S \subseteq X$ of k variables is Domain Consistent (DC) if:

$\forall x_i \in S, \forall v_i \in D(x_i), \forall x_j \in S \setminus \{x_i\}, \exists v_j \in D(x_j)$ such that c is satisfied by the assignment $x_1 = v_1, ..., x_k = v_k$

Example of SIP instance:



Let's assume that $x_3 = e$.

Domains reduced by FC in this case:

- $D(x_1) = \{a, d, g\}$
- $D(x_2) = \{a, d\}$
- $D(x_4) = \{a, d\}$
- $D(x_5) = D(x_6) = \{a, b, c, d, f, g\}$

Can we do better?

DC propagation of the constraint "$(x_1, x_4) \in E_t$":

If $x_1 = g$ then x_4 cannot be assigned to a neighbour of g

$\rightsquigarrow$ Remove g from $D(x_1)$

This corresponds to what is done by Ullmann[1]

[1] Ullmann: *An algorithm for subgraph isomorphism*, in J. ACM, 1976

Historical Notes on DC (aka as Arc Consistency)

- DC has been introduced by Ullmann in 1976 for solving the SIP[1]
- AC3 is the first "efficient" algorithm to ensure DC for binary constraints
  - Introduced by Mackworth in 1977[2]
  - Time complexity in $\mathcal{O}(ed^3)$ where e = number of constraints and d = maximum domain size
  - Space complexity in $\mathcal{O}(e)$
- AC4 is introduced by Mohr & Henderson in 1986[3]
  - Space and time complexities in $\mathcal{O}(ed^2)$
- Many (really many) algorithms introduced since then, with different time and space tradeoffs
  $\rightsquigarrow$ See the last volume of TAOCP[4]

---

1 Ullmann. *An algorithm for subgraph isomorphism*, in J. ACM, 1976
2 Mackworth. *Consistency in networks of relations*, in Artificial Intelligence, 1977
3 Mohr & Henderson. *Arc and path consistency revisited*, in Artificial Intelligence, 1986
4 Knuth. *The Art of Computer Programming. Section 7.2.2.3: Constraint Satisfaction*, 2025

Global constraints

What is a global constraint?

Constraint defined over a set of variables (the cardinality of which is not fixed)

Examples of global constraints:

- allDifferent$(x_1, \ldots, x_n) \Leftrightarrow \forall \{x_i, x_j\} \subseteq \{x_1, \ldots, x_n\}, x_i \neq x_j$
- sum$(x_1, \ldots, x_n, s) \Leftrightarrow \sum_{i=1}^{n} x_i = s$
- atLeast$(x_1, \ldots, x_n, k, v) \Leftrightarrow (\sum_{i=1}^{n} [[x_i = v]]) \geq k$ where $[[x_i = v]] = 1$ if $x_i = v$, and 0 otherwise
- ... and many others (see our catalog of 423 global constraints[1])

$\leadsto$ Ease the modelling step

How to propagate a global constraint?

- First possibility: Decompose it into existing constraints and use existing propagators
- Second possibility: Design a dedicated propagator

[1] https://sofdem.github.io/gccat/

Example: Propagation of allDifferent[1]

Suppose that:

- $D(x_1) = \{a, b, d\}$
- $D(x_2) = D(x_3) = \{b, c\}$
- $D(x_4) = \{c, d, e\}$

Propagation of allDifferent(x_1, x_2, x_3, x_4)?

$\rightsquigarrow$ Remove values that cannot belong to a solution

1 Régin. *A filtering algorithm for constraints of difference in CSPs*, in AAAI 1994

2

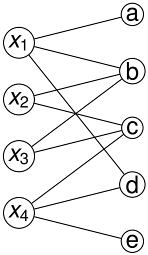
3

4

Example: Propagation of allDifferent[1]

Suppose that:

- $D(x_1) = \{a, b, d\}$
- $D(x_2) = D(x_3) = \{b, c\}$
- $D(x_4) = \{c, d, e\}$



Propagation of allDifferent(x_1, x_2, x_3, x_4)?

$\rightsquigarrow$ Remove values that cannot belong to a solution

Algorithm for propagating allDifferent:

- Build a variable/value bipartite graph
- Add a source r and a sink s
- Search for a maximum flow from r to s
  - $\rightsquigarrow$ Algorithm of Hopcroft & Karp[2] in $\mathcal{O}(p^{5/2})$
  - $\rightsquigarrow$ Inconsistency if $|f| < |X|$
- Build the "final flow graph" and search for SCC
  - $\rightsquigarrow$ Algorithm of Tarjan[3] in $\mathcal{O}(p)$
- Remove i from $D(x_j)$ if SCC$(i) \neq$SCC(x_j)[3]
  - $\rightsquigarrow$ Remove b from $D(x_1)$ and c from $D(x_4)$

1 Régin. *A filtering algorithm for constraints of difference in CSPs*, in AAAI 1994

2 Hopcroft & Karp. *An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs*, in SIAM J. Comput., 1973

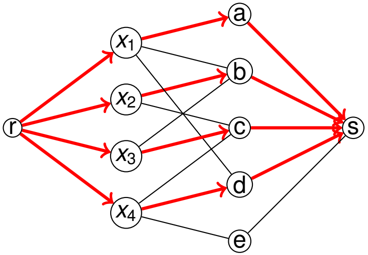
3

4

Example: Propagation of allDifferent[1]

Suppose that:

- $D(x_1) = \{a, b, d\}$
- $D(x_2) = D(x_3) = \{b, c\}$
- $D(x_4) = \{c, d, e\}$



Propagation of allDifferent(x_1, x_2, x_3, x_4)?

$\rightsquigarrow$ Remove values that cannot belong to a solution

Algorithm for propagating allDifferent:

- Build a variable/value bipartite graph
- Add a source r and a sink s
- Search for a maximum flow from r to s
  - $\rightsquigarrow$ Algorithm of Hopcroft & Karp[2] in $\mathcal{O}(p^{5/2})$
  - $\rightsquigarrow$ Inconsistency if $|f| < |X|$
- Build the "final flow graph" and search for SCC
  - $\rightsquigarrow$ Algorithm of Tarjan[3] in $\mathcal{O}(p)$
- Remove i from $D(x_j)$ if SCC$(i) \neq$SCC(x_j)[3]
  - $\rightsquigarrow$ Remove b from $D(x_1)$ and c from $D(x_4)$

1 Régin. *A filtering algorithm for constraints of difference in CSPs*, in AAAI 1994

2 Hopcroft & Karp. *An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs*, in SIAM J. Comput., 1973

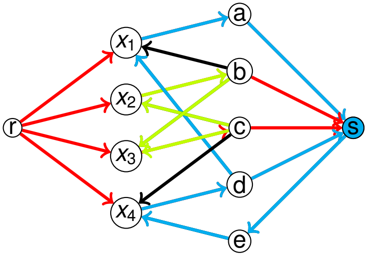
3 Tarjan. *Depth-first search and linear graph algorithms*, in SIAM J. Comput., 1972

4

Example: Propagation of allDifferent[1]

Suppose that:

- $D(x_1) = \{a, b, d\}$
- $D(x_2) = D(x_3) = \{b, c\}$
- $D(x_4) = \{c, d, e\}$



Propagation of allDifferent(x_1, x_2, x_3, x_4)?

$\rightsquigarrow$ Remove values that cannot belong to a solution

Algorithm for propagating allDifferent:

- Build a variable/value bipartite graph
- Add a source r and a sink s
- Search for a maximum flow from r to s
  - $\rightsquigarrow$ Algorithm of Hopcroft & Karp[2] in $\mathcal{O}(p^{5/2})$
  - $\rightsquigarrow$ Inconsistency if $|f| < |X|$
- Build the "final flow graph" and search for SCC
  - $\rightsquigarrow$ Algorithm of Tarjan[3] in $\mathcal{O}(p)$
- Remove i from $D(x_j)$ if SCC(i) $\neq$SCC(x_j)[3]
  - $\rightsquigarrow$ Remove b from $D(x_1)$ and c from $D(x_4)$

1 Régin. *A filtering algorithm for constraints of difference in CSPs*, in AAAI 1994

2 Hopcroft & Karp. *An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs*, in SIAM J. Comput., 1973

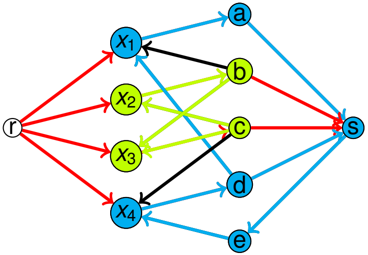
3 Tarjan. *Depth-first search and linear graph algorithms*, in SIAM J. Comput., 1972

4 Berge. *Graphs and Hypergraphs*, 1973

Example: Propagation of allDifferent[1]

Suppose that:

- $D(x_1) = \{a, b, d\}$
- $D(x_2) = D(x_3) = \{b, c\}$
- $D(x_4) = \{c, d, e\}$



Propagation of allDifferent(x_1, x_2, x_3, x_4)?

$\rightsquigarrow$ Remove values that cannot belong to a solution

Algorithm for propagating allDifferent:

- Build a variable/value bipartite graph
- Add a source r and a sink s
- Search for a maximum flow from r to s
  - $\rightsquigarrow$ Algorithm of Hopcroft & Karp[2] in $\mathcal{O}(p^{5/2})$
  - $\rightsquigarrow$ Inconsistency if $|f| < |X|$
- Build the "final flow graph" and search for SCC
  - $\rightsquigarrow$ Algorithm of Tarjan[3] in $\mathcal{O}(p)$
- Remove i from $D(x_j)$ if SCC$(i) \neq$SCC(x_j)[3]
  - $\rightsquigarrow$ Remove b from $D(x_1)$ and c from $D(x_4)$

---

1 Régin. *A filtering algorithm for constraints of difference in CSPs*, in AAAI 1994

2 Hopcroft & Karp. *An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs*, in SIAM J. Comput., 1973

3 Tarjan. *Depth-first search and linear graph algorithms*, in SIAM J. Comput., 1972

4 Berge. *Graphs and Hypergraphs*, 1973

Ordering heuristics

Branch (recall):

- Choose an unassigned variable x_i
- For each $v \in D(x_i)$, recursively solve a subproblem where v is assigned to x_i

$\rightsquigarrow$ Stop when a solution is found; Backtrack when an inconsistency is detected

Classical variable ordering heuristics:

- *deg*: Variable involved in the largest number of constraints $\rightsquigarrow$ Reduce tree depth
- *dom*: Variable with the smallest domain $\rightsquigarrow$ Reduce tree width
- $\frac{dom}{deg}$: Compromise between *dom* and *deg*
- $\frac{dom}{wdeg}$: Each constraint has a weight (incremented on failures)
  $\rightsquigarrow$ divide $|D(x_i)|$ by sum of weights of constraints associated with x_i[1]

---

1 Boussemart et al. *Boosting systematic search by weighting constraints*, in ECAI 2004

2

Ordering heuristics

Branch (recall):

- Choose an unassigned variable x_i
- For each $v \in D(x_i)$, recursively solve a subproblem where v is assigned to x_i

$\rightsquigarrow$ Stop when a solution is found; Backtrack when an inconsistency is detected

Value ordering heuristics:

Choose values that are more likely to belong to solutions
$\rightsquigarrow$ Useless for proving inconsistency of infeasible instances

- Can be learned... but this may be expensive
- Select values in the best found solution (for optimization problems)[2]

---

1 Boussemart et al. *Boosting systematic search by weighting constraints*, in ECAI 2004

2 Demirovic et al. *Solution-based phase saving for CP: A value-selection heuristic to simulate local search*, in CP 2018

Overview of the talk

1 Modelling with CP

2 Generic CP Solving Algorithms

3 LAD2025: A Constraint-based Solver for the SIP

4 Conclusion

Constraint-based Solvers for the SIP

The SIP is straightforward to model with CP...

... and it's easy to add application-dependent constraints or objective functions (related to labels, for example)

... but generic CP solvers suffer from a rather long initialisation process

Constraint-based solvers dedicated to the SIP:

- Combine Branch & Propagate with dedicated data structures, heuristics, propagators, ...
- For example: Ullman[1], LV[2], ILF[3], LAD[4], SND[5], Glasgow[6], PathLAD+[7], ... and LAD2025[8]

---

1 Ullmann. *An algorithm for subgraph isomorphism*, in J. ACM 1976

2 Larrosa and Valiente. *Constraint satisfaction algorithms for graph pattern matching. Mathematical. Structures*, in Comp. Sci. 2002

3 Zampelli et al. *Solving subgraph isomorphism problems with constraint programming*, in Constraints 2010

4 Solnon. *Alldifferent-based filtering for subgraph isomorphism*, in Art. Int. 2010

5 Audemard et al. *Scoring-based neighborhood dominance for the subgraph isomorphism problem*, in CP 2014

6 McCreesh et al. *The glasgow subgraph solver: Using CP to tackle hard SIP variants*, in ICGT 2020

7 Wang et al. *PathLAD+: An Improved Exact Algorithm for Subgraph Isomorphism Problem*, in IJCAI 2023

8 Solnon: *LAD2025, a Constraint-Based Solver for the Subgraph Isomorphism Problem*, 2025 (submitted)

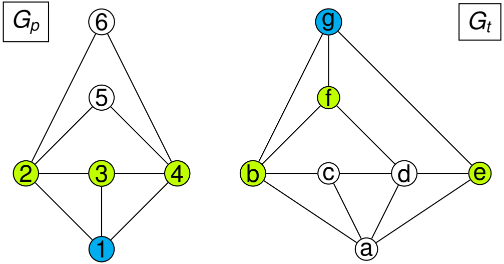
V1 of LAD[1]

⇝ Propagation of Locally All Different (LAD) constraints

Definition of LAD(x_i, u) with $i \in V_p$ and $u \in V_t$:

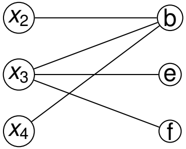
$(x_i = u) \Rightarrow (\forall j \in adj(i), x_j \in adj(u)) \wedge allDifferent(\{x_j \mid j \in adj(i)\})$

Example: Propagation of LAD when $i = 1$ and $u = g$



$D(x_2) = D(x_4) = \{a, b, d\}$
$D(x_1) = D(x_3) = D(x_5) = D(x_6) = V_t$

$(x_1 = g) \Rightarrow x_2, x_3, x_4 \in \{b, e, f\} \wedge allDifferent(x_2, x_3, x_4)$



- Propagating LAD(x_1, g) removes g from $D(x_1)$

1 Solnon. *AllDifferent-based filtering for subgraph isomorphism*, in Artificial Intelligence, 2010

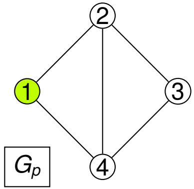
V2 of LAD = PathLAD[1]

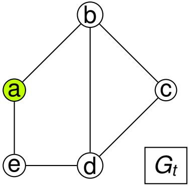
⇝ Propagation of path-based constraints

Definition of path-based constraints[2]:

- Let $p_2(i, G)$ be the number of 2-paths starting from i in G
  $\rightsquigarrow \forall i \in V_p, p_2(i, G_p) \leq p_2(x_i, G_t)$
- Let $p_2(i, j, G)$ be the number of 2-paths between i and j in G
  $\rightsquigarrow \forall i, j \in V_p, p_2(i, j, G_p) \leq p_2(x_i, x_j, G_t)$

Example:





- Propagation of $p_2(1, G_p) \leq p_2(a, G_t)$:
  $p_2(1, G_p) = 4$ and $p_2(a, G_t) = 3$
  $\rightsquigarrow$ Remove a from $D(x_1)$
- Propagation of $p_2(2, 4, G_p) \leq p_2(b, d, G_t)$ when $x_2 = b$:
  $p_2(2, 4, G_p) = 2$ and $p_2(b, d, G_t) = 1$
  $\rightsquigarrow$ Remove d from $D(x_4)$

1 Kotthoff, McCreesh, Solnon. *Portfolios of subgraph isomorphism algorithms*, in LION 2016

2 McCreesh, Prosser. *A parallel, backjumping subgraph isomorphism algorithm using supplemental graphs*, in CP 2015

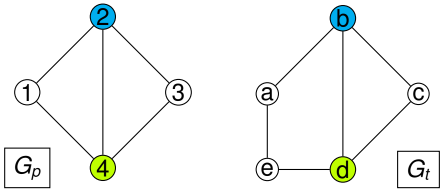
V2 of LAD = PathLAD[1]

⇝ **Propagation of path-based constraints**

Definition of path-based constraints[2]:

- Let $p_2(i, G)$ be the number of 2-paths starting from i in G
  $\rightsquigarrow \forall i \in V_p, p_2(i, G_p) \leq p_2(x_i, G_t)$
- Let $p_2(i, j, G)$ be the number of 2-paths between i and j in G
  $\rightsquigarrow \forall i, j \in V_p, p_2(i, j, G_p) \leq p_2(x_i, x_j, G_t)$

Example:



- Propagation of $p_2(1, G_p) \leq p_2(a, G_t)$:
  $p_2(1, G_p) = 4$ and $p_2(a, G_t) = 3$
  $\rightsquigarrow$ Remove a from $D(x_1)$
- Propagation of $p_2(2, 4, G_p) \leq p_2(b, d, G_t)$ when $x_2 = b$:
  $p_2(2, 4, G_p) = 2$ and $p_2(b, d, G_t) = 1$
  $\rightsquigarrow$ Remove d from $D(x_4)$

1 Kotthoff, McCreesh, Solnon. *Portfolios of subgraph isomorphism algorithms*, in LION 2016

2 McCreesh, Prosser. *A parallel, backjumping subgraph isomorphism algorithm using supplemental graphs*, in CP 2015

V3 of LAD

$\rightsquigarrow$ **Refactoring of PathLAD (mostly based on tips from Knuth[1] :-)**

Main improvements:

- Generalize the use of Sparse Sets[2] to restore states in constant time when backtracking
- Use time-stamps to wipe-out sets in constant time
- Use Tarjan instead of Kosaraju to search for strongly connected components
- Use Ford-Fulkerson instead of Hopcroft-Karp to search for covering matchings[3]
- ...

Was it worth refactoring?

1 Knuth. *The Art of Computer Programming (Fascicule 7: Constraint Satisfaction)*, 2025

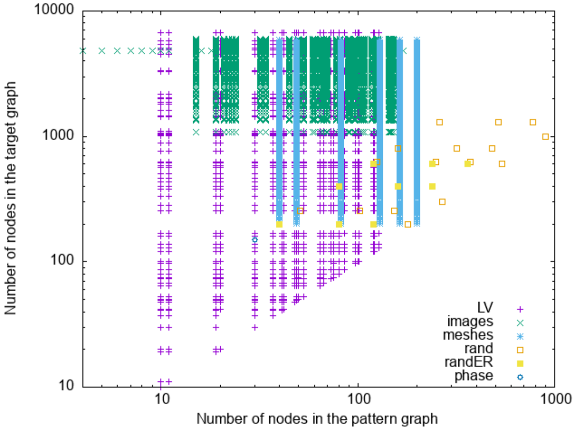
2 Le Clément, Schaus, Solnon, Lecoutre: *Sparse-Sets for Domain Implementation*, in TRICS, 2013

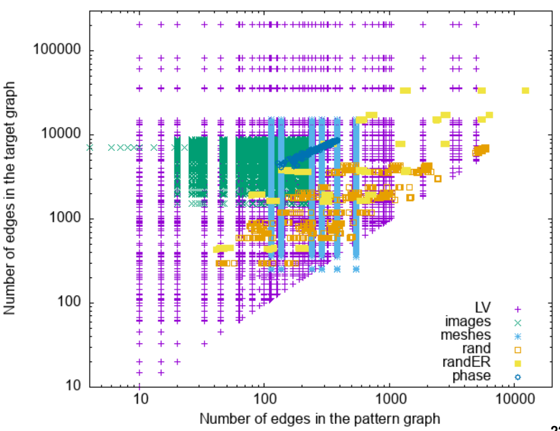
3 Gent, Miguel, Nightingale: *Generalised arc consistency for the alldifferent constraint: An empirical survey* in AI 2008

Experimental Evaluation: Benchmark Description

15,128 instances coming from 8 existing benchmarks

- Instances coming from real applications: Images and Meshes
- Random instances: randER (Erdös-Rényi, including hard instances) and rand (other models)
- Instances generated from the Stanford graph base: LV

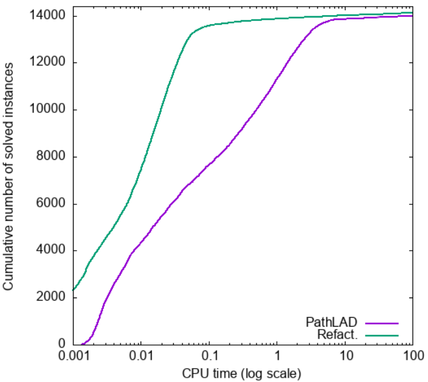


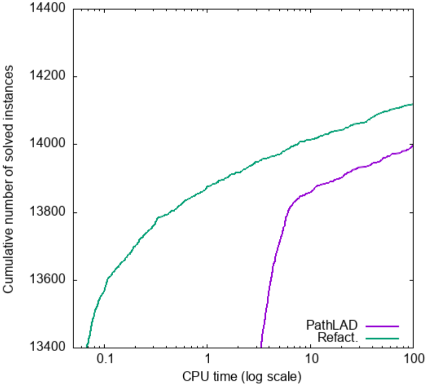


Number of instances NOT solved when the time limit is 100s (average solving time if all solved):

	LV	RandER	Meshes	Rand	Images	All benchmarks
V2 = PathLAD	222	278	51	0 (0.18s)	0 (0.86s)	551
V3 = Refactoring of V2	170	226	44	0 (0.03s)	0 (0.02s)	440

Evolution of the cumulative number of solved instances wrt time:





V4 of LAD

$\rightsquigarrow$ V3 of LAD + Propagation of Clique-based Constraints

Constraints based on maximum cliques[1]:

- Let $\chi(i, G)$ be the order of the largest clique of G that contains i
  $\rightsquigarrow \forall i \in V_p, \chi(i, G_p) \geq \chi(x_i, G_t)$
  $\rightsquigarrow$ Used to filter domains before starting the search

Constraints based on the number of k-cliques (cliques of order k), with $k \in \{3, 4, 5, 6\}$:

- Let $c(i, k, G)$ be the number of k-cliques of G that contain i
  $\rightsquigarrow \forall i \in V_p, c(i, k, G_p) \geq c(x_i, k, G_t)$
  $\rightsquigarrow$ Used to filter domains before starting the search
- Let $c(i, j, k, G)$ be the number of k-cliques of G that contain i and j
  $\rightsquigarrow \forall i, j \in V_p, c(i, j, k, G_p) \geq c(x_i, x_j, k, G_t)$
  $\rightsquigarrow$ Used to tighten LAD constraints during the search

Implementation "details":

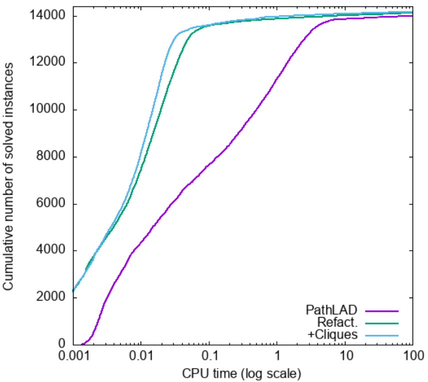
- Use efficient algorithms for computing cliques
- First compute cliques in G_p to derive bounds for the computation of cliques in G_t

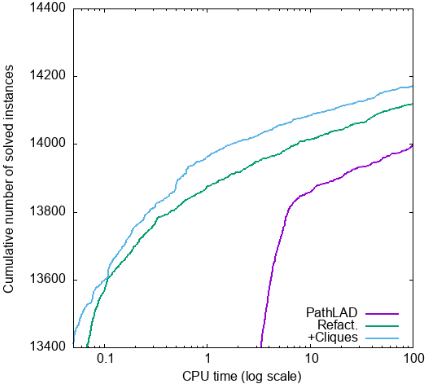
[1] Kraiczy, McCreesh. *Solving graph homomorphism and SIPs faster through clique neighbourhood constraints*, in IJCAI 2021

Number of instances NOT solved when the time limit is 100s (average solving time if all solved):

	LV	RandER	Meshes	Rand		Images		All benchmarks
V3 = Refactoring of V2	170	226	44	0	(0.03s)	0	(0.02s)	440
V4 = V3 + Cliques	142	200	45	0	(0.02s)	0	(0.01s)	387

Evolution of the cumulative number of solved instances wrt time:





V5 of LAD

$\rightsquigarrow$ **V4 of LAD + Variable Ordering Heuristic**

Variable ordering heuristic used in V4: MinDom

- Select the variable x_i that minimizes $|D(x_i)|$
  $\rightsquigarrow$ Break ties by maximizing the degree of the pattern vertex i
  $\rightsquigarrow$ Break further ties randomly

Variable ordering heuristic used in V5: MinDom/wdeg[1]

- Select the variable x_i that minimizes

$$\frac{|D(x_i)|}{\sum_{c \in C(x_i)} w(c)}$$

  where

  - $C(x_i)$ = set of constraints that involve x_i and at least one unassigned variable
  - $w(c)$ = number of times the propagation of c has raised a failure

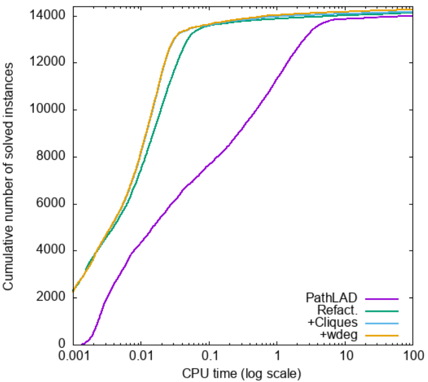
$\rightsquigarrow$ Dynamically identify critical variables that must be assigned first

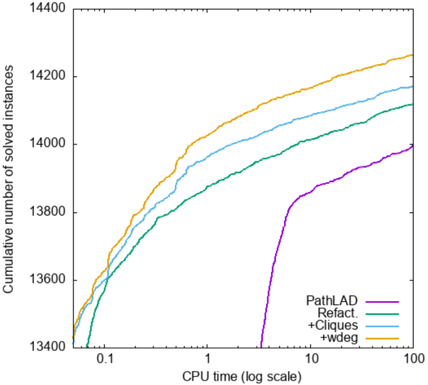
[1] Boussemart et al. *Boosting systematic search by weighting constraints*, in ECAI 2004

Number of instances NOT solved when the time limit is 100s (average solving time if all solved):

	LV	RandER	Meshes	Rand	Images	All benchmarks
V4 = V3 + Cliques	142	200	45	0 (0.02s)	0 (0.01s)	387
V5 = V4 + minDom/wdeg	105	187	1	0 (0.02s)	0 (0.01s)	293

Evolution of the cumulative number of solved instances wrt time:





V6 of LAD

⇝ V5 of LAD + Restarts and Randomized Value Ordering Heuristic

Restarts[1]:

- Restart the search when the number of failures reaches a cutoff limit
- Geometric progression of the cutoff limit from one restart to the next one

⇝ Avoids heavy tails when combined with randomized ordering heuristics

NoGood recording[2]:

- After each restart, add a constraint to forbid the exploration of nodes already explored

Value Ordering Heuristic:

- Degree-biased heuristic[3]: The probability to choose a target vertex is proportional to its degree
- Combined with saving: Give priority to values used in the largest assignment built so far

---

1 Gomes, Selman, Kautz: *Boosting combinatorial search through randomization*, in AAAI 1998

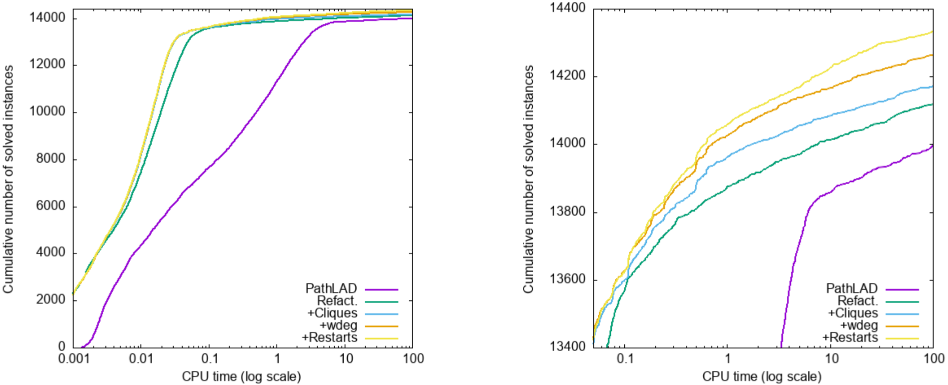
2 Lee, Schulte, Zhu: *Increasing nogoods in restart-based search*, in AAAI 2016

3 Archibald et al: *Sequential and parallel solution-biased search for subgraph algorithms*, in CPAIOR 2019

Number of instances NOT solved when the time limit is 100s (average solving time if all solved):

	LV	RandER	Meshes	Rand	Images	All benchmarks
V5 = V4 + minDom/wdeg	105	187	1	0 (0.02s)	0 (0.01s)	293
V6 = V5 + Restarts	91	133	1	0 (0.02s)	0 (0.01s)	225

Evolution of the cumulative number of solved instances wrt time:



V7 of LAD = LAD2025

⇝ V6 of LAD + Per Instance Propagation Selection

Two complementary propagation algorithms for the subgraph isomorphism problem:

- DC of LAD constraints: Drastically filters domains, but very expensive on dense graphs
- FC of basic edge constraints: Very fast, but filters less values

The "best" propagation algorithm depends on the instance to solve!

Per instance algorithm selection[1]:

Given a portfolio of solvers with complementary performance:

- Learn a model for selecting the best performing solver for each instance
- Use this model to select a solver at run time

Simple rule for selecting the propagation algorithm:

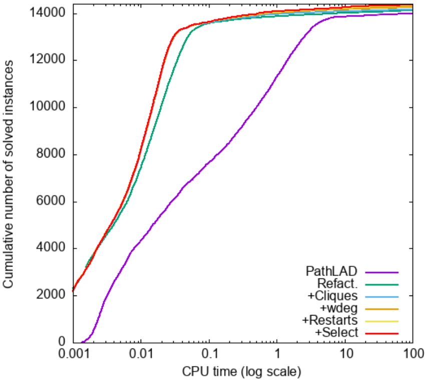
- If the density of G_p and G_t exceeds 0.15: Select FC of edge constraints
- Otherwise: Select DC of LAD constraints

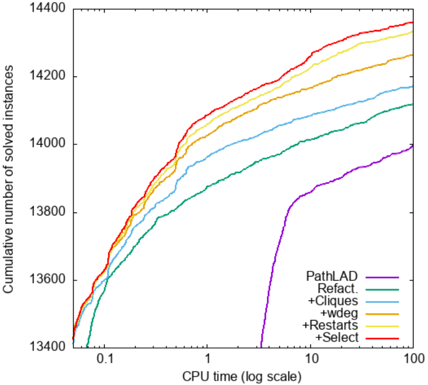
[1] Kotthoff, McCreesh, Solnon: *Portfolios of subgraph isomorphism algorithms*, in LION 2016

Number of instances NOT solved when the time limit is 100s (average solving time if all solved):

	LV	RandER	Meshes	Rand		Images		All benchmarks
V6 = V5 + Restarts	91	133	1	0	(0.02s)	0	(0.01s)	225
V7 = V6 + Select	91	104	1	0	(0.02s)	0	(0.01s)	196

Evolution of the cumulative number of solved instances wrt time:





Experimental comparison with state-of-the-art approaches

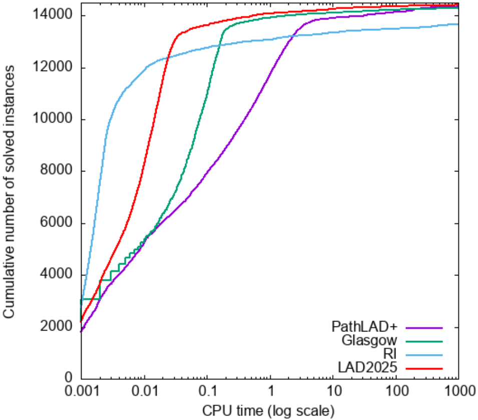
Number of instances NOT solved when the time limit is 1000s (average solving time if all solved):

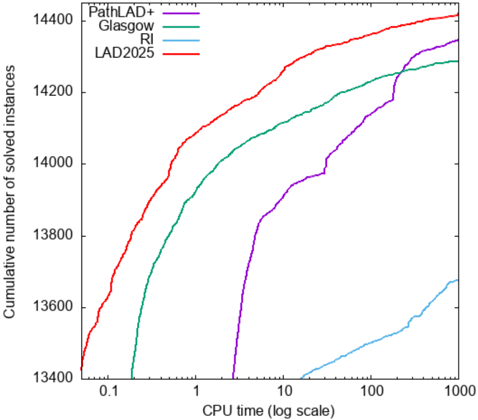
	LV	RandER	Meshes	Rand		Images		All benchmarks
RI[1]	356	180	324	20	-	0	(0.003s)	880
Glasgow[2]	152	86	32	0	(0.042s)	0	(0.080s)	270
PathLAD+[3]	112	89	11	0	(0.198s)	0	(0.842s)	212
LAD2025[4]	72	68	1	0	(0.024s)	0	(0.014s)	141

- RI is the fastest approach on Images, but it is much less successful on all other benchmarks
- LAD2025 is the most successful solver on all other benchmarks

1 Bonnici, Giugno. *On the variable ordering in subgraph isomorphism algorithms*, in IEEE Trans. Bioinform., 2017
2 Kraiczy, McCreesh. *Solving graph homomorphism and SIPs faster through clique neighbourhood constraints*, in IJCAI 2021
3 Wang, Jin, Cai, Lin. *PathLAD+: An Improved Exact Algorithm for Subgraph Isomorphism Problem*, in IJCAI 2023
4 Solnon: *LAD2025, a Constraint-Based Solver for the Subgraph Isomorphism Problem*, 2025 (submitted)

Cumulative Number of Solved Instances wrt Time





- RI is the most successful solver when the time limit is smaller than 0.03s
  ⇝ But it solves 739 less instances than LAD2025 when the time limit is 1000s
- LAD2025 is the most successful solver when the time limit is greater than 0.03s

Overview of the talk

1 Modelling with CP

2 Generic CP Solving Algorithms

3 LAD2025: A Constraint-based Solver for the SIP

4 Conclusion

Conclusion

Modelling and solving the subgraph isomorphism problem with CP

- Very nice and straightforward model
  $\rightsquigarrow$ Application-dependent constraints and/or objective functions may be easily added
- Generic CP solvers are able to solve medium-size instances...
  ...But they are not as efficient as state-of-the-art dedicated approaches

Constraint-based approaches for solving the subgraph isomorphism problem

- Combine classical ingredients of CP solvers
  $\rightsquigarrow$ Constraint propagation, Ordering heuristics, Random restarts, NoGood recording, ...
- With efficient data structures
  $\rightsquigarrow$ Sparse sets, Time stamps, ...
- And dedicated graph invariants
  $\rightsquigarrow$ 2-paths, max cliques, k-cliques

Some further work

- Extension to directed and labelled graphs, and to the induced subgraph isomorphism problem