



Practical verification of MSO properties of graphs of bounded clique-width

Irène Durand (joint work with Bruno Courcelle)

LaBRI, Université de Bordeaux

13 décembre 2010

Cours de Master 2 Informatique, Logique et Langages, INF569, 2010

Objectives

Verify properties of graphs

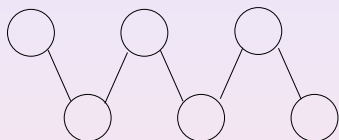
Properties

- ▶ connectedness,
- ▶ k -colorability,
- ▶ existence of cycles
- ▶ existence of paths
- ▶ bounds (cardinality, degree, ...)
- ▶ ...

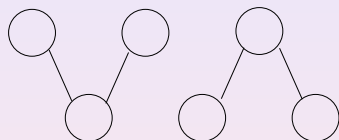
How : using **term automata**

Note that we consider **finite** graphs only

Connectedness



Connected

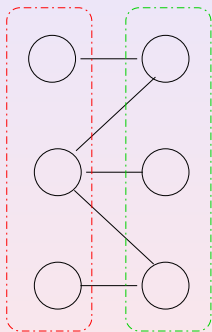


Not connected

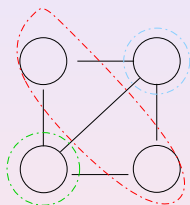
Applications to frequency allocation, scheduling, ...

k-Colorability

colored graph : two vertices connected by an edge do not have the same color

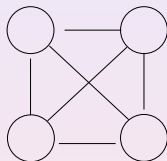


2-colorable
bi-partite



not 2-colorable

3-colorable



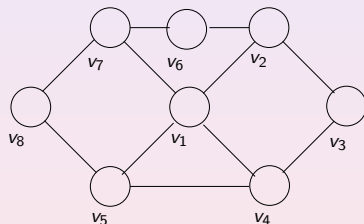
not 3-colorable

Graphs as relational structures

For simplicity, we consider **simple, loop-free undirected** graphs

Extensions are easy

Every graph G can be identified with the **relational structure** $(\mathcal{V}_G, \text{edg}_G)$ where \mathcal{V}_G is the set of vertices and $\text{edg}_G \subseteq \mathcal{V}_G \times \mathcal{V}_G$ the binary symmetric relation that defines edges.



$$\mathcal{V}_G = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\}$$
$$\text{edg}_G = \{(v_1, v_2), (v_1, v_4), (v_1, v_5), (v_1, v_7), (v_2, v_3), (v_2, v_6), (v_3, v_4), (v_4, v_5), (v_5, v_8), (v_6, v_7), (v_7, v_8)\}$$

Expression of graph properties

First order logic (FO) :

▶ Atomic formulas : $x = y$, $edg(x, y)$

▶ Boolean connectives : \wedge, \vee, \neg

▶ Example

$Distance(x, y) \leq 3$:

$x = y \vee edg(x, y) \vee \exists z(edg(x, z) \wedge edg(z, y)) \vee$

$\exists z, t(edg(x, z) \wedge edg(z, t) \wedge edg(t, y))$

▶ quantification on single vertices $x, y \dots$ only

▶ too weak ; can only express "local" properties like having degree or diameter bounded by some fixed integer

▶ k -colorability ($k > 1$) cannot be expressed

Second order logic (SO)

▶ quantifications on relations of arbitrary arity

▶ SO can express most properties of interest in Graph Theory

▶ too complex (many problems are undecidable or do not have a polynomial solution).

Monadic second order logic (MSO)

- ▶ SO formulas that only use quantifications on unary relations (i.e., on sets).
- ▶ can express many useful graph properties like connectedness, k -colorability, planarity...

Example : k -colorability

$$\textit{Stable}(X) : \forall u, v (u \in X \wedge v \in X \Rightarrow \neg \textit{edg}(u, v))$$

$$\textit{Partition}(X_1, \dots, X_m) :$$

$$\forall x (x \in X_1 \vee \dots \vee x \in X_m) \wedge_{i < j} \forall x (x \in X_i \Rightarrow x \notin X_j)$$

k -colorability :

$$\exists X_1, \dots, X_k \textit{Partition}(X_1, \dots, X_k)$$

$$\wedge \textit{Stable}(X_1) \wedge \dots \wedge \textit{Stable}(X_k)$$

Interesting algorithmic consequences

The fundamental theorem

Theorem

[Courcelle (1990) for tree-width,

Courcelle, Makowski, Rotics (2001) for clique-width]

Monadic second-order model checking is *fixed-parameter tractable* for tree-width and clique-width.

- ▶ **Tree-width** and **clique-width** : graph complexity measures based on graph decompositions
- ▶ a **decomposition** produces a **term** representation of the graph
- ▶ the **algorithm** is given by a **term automaton** recognizing the terms denoting graphs satisfying the property
- ▶ How can we find this automaton ?

Representation of graphs by terms

- ▶ depends on the chosen width (here **clique-width**)
- ▶ other widths : tree-width, path-width, boolean-width, ...

Let \mathcal{L} a finite set of labels $\{a, b, c, \dots\}$.

Graphs $G = (\mathcal{V}_G, \text{edg}_G)$ s.t.

each vertex $v \in \mathcal{V}_G$ has a **label**, $\text{label}(v) \in \mathcal{L}$.





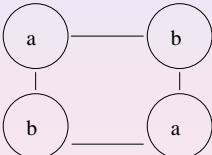
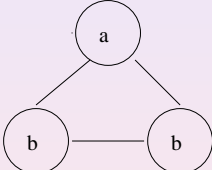
Operations :

- ▶ constant ***a*** denotes a graph with a single vertex labeled by a ,
- ▶ \oplus (binary) : union of disjoint graphs
- ▶ ***add_{a_b}*** (unary) : adds the missing edges between every vertex labeled a and every vertex labeled b ,
- ▶ ***ren_{a_b}*** (unary) : renames a to b

Let $\mathcal{F}_{\mathcal{L}}$ be the set of these operations and constants.

Every term $t \in \mathcal{T}(\mathcal{F}_{\mathcal{L}})$ defines a graph G_t whose vertices are the constants (leaves) of the term t .

Note that, because of the relabeling operations, the labels of the vertices in the graph $G(t)$ may differ from the ones specified in the leaves of the term.

| | | |
|---|---|--|
| $t_0 = a$ | $t_1 = b$ | $t_2 = \oplus(a, b)$ |
|  |  |  |
| $t_3 = \text{add}_{a,b}(t_2)$ | $\text{add}_{a,b}(\oplus(t_2, t_2))$ | $\text{add}_{a,b}(\oplus(a, \text{ren}_{a,b}(t_3)))$ |
|  |  |  |

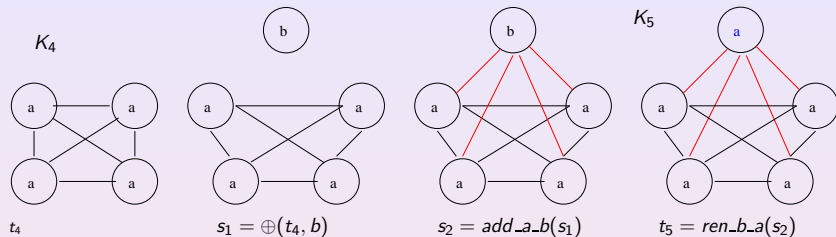
Definition

A graph has **clique-width** at most k if it is defined by some $t \in \mathcal{T}(\mathcal{F}_{\mathcal{L}})$ with $|\mathcal{L}| \leq k$.

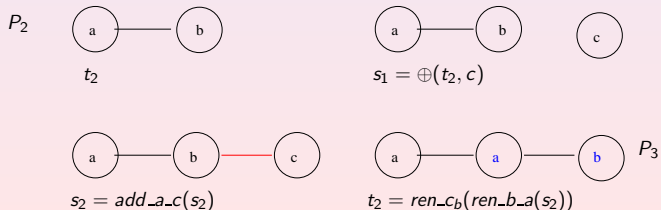
Note that different terms may define identical graphs.

Examples

Clique (K_n) :



Chain (P_n) :



Term automata (Bottom-up)

$\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ with Δ set of **transitions** $f(q_1, \dots, q_n) \rightarrow q$

Automaton 2-STABLE

Signature: a b ren_a_b:1 ren_b_a:1 add_a_b:1 oplus:2*

States: <a> <ab> <error>

Final States: <a> <ab>

Transitions a \rightarrow <a>

add_a_b(<a>) \rightarrow <a>

ren_a_b(<a>) \rightarrow

ren_a_b() \rightarrow

ren_a_b(<ab>) \rightarrow

oplus*(<a>, <a>) \rightarrow <a>

oplus*(<a>,) \rightarrow <ab>

oplus*(<a>, <ab>) \rightarrow <ab>

add_a_b(<ab>) \rightarrow <error>

add_a_b(<error>) \rightarrow <error>

oplus*(<error>, q) \rightarrow <error>

b \rightarrow

add_a_b() \rightarrow

ren_b_a(<a>) \rightarrow <a>

ren_b_a() \rightarrow <a>

ren_b_a(<ab>) \rightarrow <a>

oplus*(,) \rightarrow

oplus*(, <ab>) \rightarrow <ab>

oplus*(<ab>, <ab>) \rightarrow <ab>

ren_a_b(<error>) \rightarrow <error>

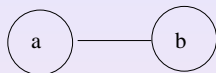
ren_b_a(<error>) \rightarrow <error>

for all q

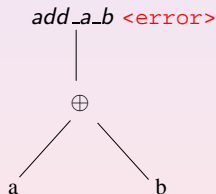
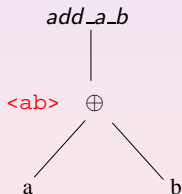
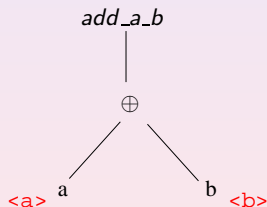
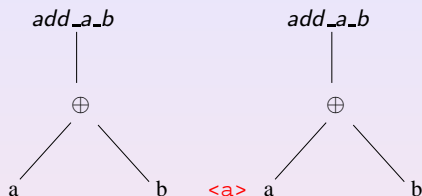
Run of an automaton on a term

The term is recognized when we obtain a final state at the root.

G



$$t_G = \text{add_a_b}(\oplus(a, b))$$



```
add_a_b(ren_a_b(oplus(a,b))) -> add_a_b(ren_a_b(oplus(<a>,b)))
-> add_a_b(ren_a_b(oplus(<a>,<b>))) -> add_a_b(ren_a_b(<ab>))
-> add_a_b(<b>) -> <b>
```

Free set variables $P(X_1, \dots, X_m)$

Each X_i corresponds to a subset of vertices

To express membership of vertices to the X_i , the constants (representing the vertices of the graph) are associated with a bit-vector $k_1 \dots k_m$. $k_i = 1$ iff the vertex belongs to X_i .

$Stable(X_1)$: the subgraph induced by X_1 is a stable

$\mathcal{A}_{Stable(X_1)}$ can be obtained from $\mathcal{A}_{Stable()}$

New signature:

a^0 a^1 b^0 b^1 $ren_a_b:1$ $ren_b_a:1$ $add_a_b:1$ $oplus:2*$

New constant transitions:

$a^0 \rightarrow \#$ $a^1 \rightarrow \langle a \rangle$

$b^0 \rightarrow \#$ $b^1 \rightarrow \langle b \rangle$

New non constant transitions:

$ren_*_*(\#) \rightarrow \#$ $add_*_*(\#) \rightarrow \#$ $oplus(\#,q) \rightarrow q$ for all q

$add_a_b(oplus(oplus(a^1,b^0),a^1)) \rightarrow +$

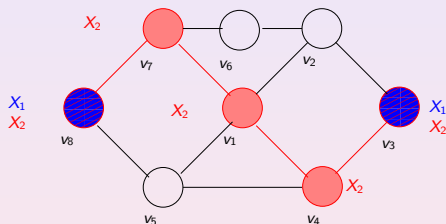
$add_a_b(oplus(oplus(\langle a \rangle,\#),\langle a \rangle)) \rightarrow$

$add_a_b(oplus(\#,\langle a \rangle)) \rightarrow add_a_b(\langle a \rangle) \rightarrow \langle a \rangle$

Example of the $Path(X_1, X_2)$ property

Graph G , X_1 and X_2 two subsets of vertices of G

Predicate $Path(X_1, X_2)$, true when $X_1 \subseteq X_2$, $|X_1| = 2$ and some path in $G[X_2]$ links the two vertices of X_1 .



$$X_1 = \{v_3, v_8\}$$

$$X_2 = \{v_1, v_3, v_4, v_7, v_8\}$$

$$v_8 - v_7 - v_1 - v_4 - v_3$$

$$X_1 = \{v_3, v_8\}$$

$$X_2 = \{v_1, v_3, v_4, v_8\}$$

The following term describes the previous graph with one of the set variables assignment :

```
add_c_d(  
  add_b_d(  
     $\oplus(d^{01},$   
      ren_d_b(  
        add_a_d(  
           $\oplus(d^{00},$   
            add_c_e(  
               $\oplus(\text{add\_a\_b}(\text{add\_b\_c}(\oplus(a^{11}, \oplus(b^{01}, c^{00}))))),$   
                 $\text{add\_a\_b}(\text{add\_b\_e}(\oplus(a^{00}, \oplus(b^{01}, e^{11}))))))))))$ 
```


The $Path(X_1, X_2)$ can be expressed by the following MSO formula :

$$\begin{aligned} & \forall x[x \in X_1 \Rightarrow x \in X_2] \wedge \exists x, y[x \in X_1 \wedge y \in X_1 \wedge x \neq y \wedge \\ & \forall z(z \in X_1 \Rightarrow x = z \vee y = z) \wedge \\ & \forall X_3[x \in X_3 \wedge \forall u, v(u \in X_3 \wedge u \in X_2 \wedge v \in X_2 \wedge \text{edg}(u, v) \Rightarrow v \in X_3) \\ & \Rightarrow y \in X_3]] \end{aligned}$$

of **quantifier-height** 5.

Uppercase variables correspond to **sets of vertices**, and lowercase variables correspond to **individual vertices**.

The problem

Input :

- ▶ an MSO formula $\phi = P(X_1, \dots, X_m)$ expressing a graph property
- ▶ a graph G represented by a term t_G with an assignment to X_1, \dots, X_m

Question :

- ▶ Does G satisfy the graph property expressed by ϕ

Example

$Path(X_1, X_2)$ and the previous graph (with an assignment of the sets variables).

The general solution

1. Transform the MSO formula ϕ into an automaton \mathcal{A}_ϕ
2. Run \mathcal{A}_ϕ on the term t_G representing the graph.

In order to process an MSO formula, we must standardize ϕ .

1. translate it into an equivalent formula
 - ▶ without first-order variables (same quantifier-height)
 - ▶ with existential quantifiers only
 - ▶ with boolean operations only (and, or, negation)
 - ▶ and simple atomic properties like $X = \emptyset$, $Sgl(X)$ (denoting that X is a singleton set), $X_i \subseteq X_j$ for which an automaton is easily computable.
2. standardize the names of set variables.

Standardization of the formula (Example)

$$\begin{aligned} \text{Path}(X_1, X_2) &= X_1 \subseteq X_2 \wedge P_1(X_1, X_2) \\ P_1(X_1, X_2) &= \exists X_3, X_4, P_2(X_1, X_2, X_3, X_4) \\ P_2(X_1, X_2, X_3, X_4) &= \text{Sgl}(X_3) \wedge \text{Sgl}(X_4) \wedge X_3 \subseteq X_1 \wedge X_4 \subseteq X_1 \wedge X_3 \\ &\quad \neq X_4 \wedge |X_1| = 2 \wedge P_4(X_2, X_3, X_4) \\ P_4(X_2, X_3, X_4) &= \neg P_5(X_2, X_3, X_4) \\ P_5(X_2, X_3, X_4) &= \exists X'_1, P_6(X'_1, X_2, X_3, X_4) \\ P_6(X'_1, X_2, X_3, X_4) &= X_3 \subseteq X_5 \wedge \neg X_4 \subseteq X_5 \wedge P_7(X'_1, X_2) \\ P_7(X'_1, X_2) &= \neg P_8(X'_1, X_2) \\ P_8(X'_1, X_2) &= \exists X_3, X_4, P_9(X'_1, X_2, X_3, X_4) \\ P_9(X'_1, X_2, X_3, X_4) &= \text{Sgl}(X_3) \wedge \text{Sgl}(X_4) \wedge X_3 \subseteq X'_1 \wedge X_3 \subseteq X_2 \wedge \\ &\quad X_4 \subseteq X_2 \wedge \text{Edg}(X_3, X_4) \wedge \neg X_4 \subset X'_1 \end{aligned}$$

Note that this translation is here done by hand

Automata for atomic formulas

It is necessary to implement for once the **ad-hoc** constructions for the automata corresponding to atomic formulas

- ▶ $Edg(X_1, X_2)$,
- ▶ $Sgl(X)$,
- ▶ $X_1 \subseteq X_2$,
- ▶ $X_1 = X_2$,
- ▶ ...

Some **variable change** or **homomorphisms** (and inverse homomorphism) may be applied in order to obtain all the desired versions.

Some **variable change** or **inverse homomorphisms** may be applied in order to obtain all the desired versions. These transformations **preserve determinism**.

For instance, from an automaton for a property $P()$, we can easily obtain variants for $P(X)$, $P(\overline{X})$, $P(X_i)$, $P(X_i \cup X_j)$, $P(X_i \cap X_j)$, $P(\dots, X_i, \dots)$.

The general algorithm for computing the automaton

If the formula is **atomic** (or if we already have an automaton for it) then return the corresponding automaton.

Otherwise :

- ▶ **disjunction** $\phi = \phi_1 \vee \phi_2$: **union** of \mathcal{A}_{ϕ_1} and \mathcal{A}_{ϕ_2} .
- ▶ **conjunction** $\phi = \phi_1 \wedge \phi_2$: **intersection** of \mathcal{A}_{ϕ_1} and \mathcal{A}_{ϕ_2} .
- ▶ **negation** $\phi = \neg\phi'$: **complementation** of $\mathcal{A}_{\phi'}$. ($\mathcal{A}_{\phi'}$ must be determinized first).
- ▶ **existential** formula $\exists X_i, P(X_1, \dots, X_m)$: **projection** of $\mathcal{A}_{P(X_1, \dots, X_m)}$ on $(1, \dots, i-1, i+1, m)$. which implies a shift in the indices of variables X_{i+1}, \dots, X_m . **creates nondeterminism**
- ▶ $\phi = P(X_1, \dots, X_j, \dots, X_m)$ does not use X_j : **cylindrification** of the automaton $\mathcal{A}_{P(X'_1, \dots, X'_{m-1})}$ (with $X'_i = X_i$ for $1 \leq i < j$ and $X'_i = X_{i+1}$ for $j \leq i < m$) on the j -th components. **preserves determinism**

Autowrite

- ▶ Lisp software (currently 15000 lines)
- ▶ First designed to check call-by-need properties of term rewriting systems.
- ▶ Implements bottom-up term-automata and most of the well-known operations on such automata
 - ▶ union
 - ▶ intersection
 - ▶ determinization
 - ▶ minimization
 - ▶ complementation
 - ▶ projection
 - ▶ cylindrification
 - ▶ (inverse) homomorphism
 - ▶ ...



```

(setf *p9* (intersection-automata
          (list (setup-singleton-automaton *cwd* 4 3)
                (setup-singleton-automaton *cwd* 4 4)
                (setup-subset-automaton *cwd* 4 3 1)
                (setup-subset-automaton *cwd* 4 3 2)
                (setup-subset-automaton *cwd* 4 4 2)
                (complement-automaton
                 (setup-subset-automaton *cwd* 4 4 1))
                (setup-edge-automaton *cwd* 4 3 4))))
(setf *p8* (project-and-simplify-automaton *p9* '(0 1)))
(setf *p7* (complement-automaton *p8*))
(setf *p7p* (cylindrify-and-simplify-automaton *p7* '(2 3)))
(setf *p6* (intersection-automata
          (list *p7p*
                (setup-subset-automaton *cwd* 4 3 1)
                (complement-automaton
                 (setup-subset-automaton *cwd* 4 4 1)))))
(setf *p5* (vprojection *p6* '(1 2 3)))

```



```

(setf *p5*    ;; blows up for cwd=3
      (ndeterminize-automaton *p5*))
(setf *p5* (nsimplify-automaton *p5*))
(setf *p4* (complement-automaton *p5*))
(setf *p4p* (cylindrify-and-simplify-automaton *p4* 0))
(setf *p3* (intersection-automata
           (list *p4p*
                 (setup-subset-automaton *cwd* 4 3 1)
                 (setup-subset-automaton *cwd* 4 4 1)
                 (complement-automaton
                  (setup-equality-automaton *cwd* 4 3 4))
                  (setup-cardinality-automaton *cwd* 4 1 2))))))
(setf *p2* (intersection-automata
           (list *p3*
                 (setup-singleton-automaton *cwd* 4 3)
                 (setup-singleton-automaton *cwd* 4 4))))
(setf *p1* (project-and-simplify-automaton *p2* '(0 1)))
(setf *p* (intersection-automata
           (list *p1* (setup-subset-automaton *cwd* 2 1 2))))

```

Results for the Path property

| | | |
|---------------------------------|---------|-----|
| <i>cwd</i> | 2 | 3 |
| $\mathcal{A}/\min(\mathcal{A})$ | 25 / 12 | out |

- ▶ **Runs out of memory** for $cwd = 3$, although we know that the **minimal** automaton has 124 states which is still reasonable.
- ▶ The problem comes from **intermediate** steps.
- ▶ The non-deterministic version of $\mathcal{A}_{P_5(x_2, x_3, x_4)}$ has 308 states.
- ▶ Its complementation triggers its determinization which causes the blow up.

Second method : direct construction of the automaton

Observation : intermediate steps induce an exponential blow up although the final automaton is not so big.

Idea : give a direct construction of the automaton.

This method is **not** general.

For each property, one must give a description of the automaton

- ▶ description of the states,
- ▶ description of the transitions rules
- ▶ the state computed at t **encodes** information about the graph G_t defined by the subterm processed so far.
- ▶ the transition function computes the new state (the new information) from the information contained in the states obtained for the subterms.

Direct construction of an automaton for PATH(X1,X2)

Such a description exists for the **path property**

The direct construction works up to $cwd = 4$.

| cwd | 2 | 3 | 4 | 5 |
|---------------------------------|---------|-----------|-------------|-----|
| $\mathcal{A}/\min(\mathcal{A})$ | 25 / 12 | 213 / 124 | 4792 / 2015 | out |

Number of states of the unique minimal automaton :

$$2^{cwd^2/2} < |Q| < 2^{cwd^2+2} \quad \text{For } cwd = 5 : 33554432 < |Q|$$





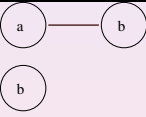
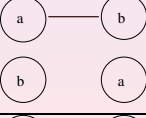
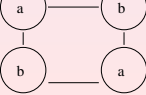
Comment : the automata are simply too **big**!

Experiments with a direct construction

Illustration with connectedness

- ▶ In the automaton for **connectedness**, roughly, the state accessible from a term t contain the set of sets of labels of the connected components of G_t .
- ▶ The **final** states are all the singleton states.

Direct construction for **connectedness**

| Term | Graph | State |
|--------------------------|---|--|
| $t_1 = a$ |  | $[\langle a \rangle]$ |
| $t_2 = b$ |  | $[\langle b \rangle]$ |
| $t_3 = \oplus(t_1, t_2)$ |  | $[\langle a \rangle \langle b \rangle]$ |
| $t_4 = add_a_b(t_3)$ |  | $[\langle ab \rangle]$ |
| $t_5 = \oplus(b, t_4)$ |  | $[\langle b \rangle \langle ab \rangle]$ |
| $t_6 = \oplus(a, t_5)$ |  | $[\langle a \rangle \langle b \rangle \langle ab \rangle]$ |
| $t_7 = add_a_b(t_6)$ |  | $[\langle ab \rangle]$ |

Results for the connectedness property

Number of states : $2^{2^{cwd}-1} + 2^{cwd} - 2$

- ▶ works up to $cwd = 3$
- ▶ runs out of memory

| cwd | 2 | 3 | 4 |
|---------------------------------|--------|----------|-----|
| $\mathcal{A}/\min(\mathcal{A})$ | 10 / 6 | 134 / 56 | out |

For $cwd = 4$: $|Q| = 32782$

Number of states of the **minimal** automaton : $|Q| > 2^{2^{\lfloor cwd/2 \rfloor}}$

Comment : the automata are simply too **big**!

Fly automata

Principle : the transitions are represented by a **function** (in our case a Lisp function); the complete sets of transitions, states and finalstates are **never** computed in extenso.

fly automaton $\mathcal{A} = (\mathcal{F}, final, \delta)$: abstraction of the usual automaton (with stored transitions)

```
(defun fly-path-automaton (cwd)
  (make-fly-automaton
   (setup-signature cwd 2)
   (lambda (root states) ;; f(q1 ... qn) -> q
     (path-transitions-fun root states)))
  (lambda (state)
    (path-final-p state)))

(defclass abstract-automaton (named-object signed-object)
  ((transitions :accessor get-transitions)))

(defclass fly-automaton (abstract-automaton)
  ((finalstates-fun :reader finalstates-fun)))

(defun make-fly-automaton (signature tfun finalstates-fun))
```


Remark : in compilation, one uses **small** automata to process **large** words. Here we use **huge** automata to process **small** terms (say 100 to 100000 nodes).

Connectedness case

Expression of non connectedness :

$$\exists X [\exists x \in X \wedge \exists y \notin X \wedge \forall x, y (\text{Edg}(x, y) \Rightarrow [x \in X \Leftrightarrow y \in X])]$$

Lisp description

```
(defclass connectedness-state (graph-state)
  ((components :type list :reader components)))

(defmethod graph-add-target (a b (co connectedness-state)) ...)
(defmethod graph-ren-target (a b (co connectedness-state)) ...)

(defmethod graph-oplus-target
  ((co1 connectedness-state) (co2 connectedness-state))
  (make-connectedness-state
   (append (components co1) (components co2))))

(defmethod connectedness-transitions-fun
  ((root constant-symbol) (states (eql nil)))
  (let ((port (name-to-port (name root))))
    (make-connectedness-state
     (list (make-port-state (list port)))))))

(defmethod connectedness-transitions-fun
  ((root parity-symbol) (states list))
  (common-transitions-fun root states))
```

Fly automata

- ▶ runs on all our data
- ▶ no limitation on the clique-width to create the automaton
- ▶ limitations come when running the automaton on very deep terms (stack exhaustion)
- ▶ runs in 1mn on a grid 80x80 (cwd=81) connectedness

The implementation of operations on fly automata uses intensively the **functional programming paradigm**.

Operation on fly automata

- ▶ Union
- ▶ Intersection
- ▶ Determinization
- ▶ Complementation
- ▶ Homomorphism

Uses intensively the **functional programming** paradigm

Operations on Fly-automata

```
(defmethod complement-automaton ((f fly-automaton))  
  (make-fly-automaton  
    (get-transitions f)  
    (complement-finalstate-fun f)))
```

```
(defmethod union-automaton  
  ((f1 fly-automaton) (f2 fly-automaton))  
  (make-fly-automaton  
    (lambda (root states)  
      (target-union  
        (apply-transition-function-gft  
          root states (get-transitions f1)  
          (apply-transition-function-gft  
            root states (get-transitions f2))))))  
    (lambda (state)  
      (or (finalstate-p state f1)  
          (finalstate-p state f2)))))
```

Fly-automata versus Table-automata

Table-automata

- ▶ *compiled version* of fly-automata
- ▶ faster for recognizing a term
- ▶ use space for storing the transitions table
- ▶ the space depends on the clique-width

Fly-automata

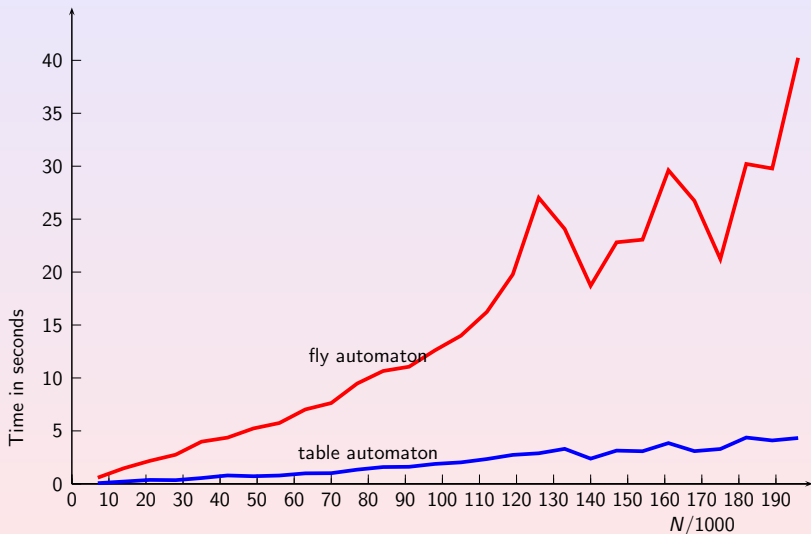
- ▶ use constant space
- ▶ slower for term recognition because of the calls to the transition function
- ▶ the time depends on the clique-width

Use

- ▶ a table-automaton when the transitions table can be computed
- ▶ a fly-automaton otherwise

Experimental results

Connectedness on graphs P_N ($cwd = 3$)



Some properties

Direct constructions of the automata for the following properties.

Polynomial

- ▶ Stable()
- ▶ Partition(X_1, \dots, X_m)
- ▶ k -Cardinality()

non polynomial

- ▶ k -Coloring(C_1, \dots, C_k) compilable up to $cwd = 4$ (for $k = 3$)
- ▶ Connectedness() compilable up to $cwd = 3$
- ▶ Clique() compilable up to $cwd = 4$
- ▶ Path(X_1, X_2) compilable up to $cwd = 4$
- ▶ Forest() (no cycle) not compilable

Some more properties

With the previous properties, using homomorphisms and boolean operations, we obtain automata for

- ▶ k -Colorability() compilable up to $k = 3$ ($cwd = 2$), $k = 2$ ($cwd = 3$)
- ▶ k -Acyclic-Colorability() not compilable (uses Forest)
- ▶ k -Chord-Free-Cycle()
- ▶ k -Max-Degree()
- ▶ Vertex-Cover(X_1) 2^{cwd} states
- ▶ k -Vertex-Cover()

Example of vertex-cover

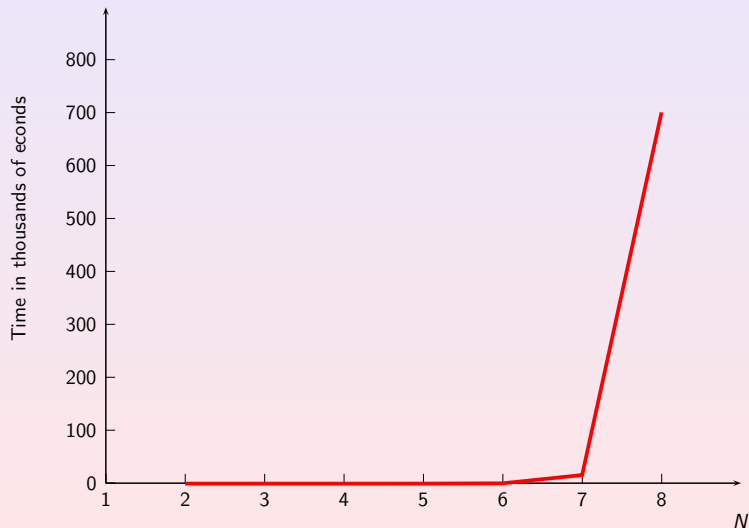
a combination of already defined automata

```
;; Vertex-Cover( $X_1$ ) = Stable( $V-X_1$ )
(defun fly-vertex-cover (cwd)
  (x1-to-cx1 ;; Stable( $V-X_1$ )
    ;; Stable( $X_1$ )
    (fly-subgraph-stable-automaton cwd 1 1)))

(defun fly-k-vertex-cover (k cwd)
  ;; exists  $X_1$  s.t. vertex-cover( $X_1$ ) and card( $X_1$ ) = k
  (vprojection
    (intersection-automaton
      (fly-vertex-cover cwd) ;; Vertex-Cover( $X_1$ )
      (fly-subgraph-cardinality-automaton ;; Card( $X_1$ ) = k
        k cwd 1 1))))
```

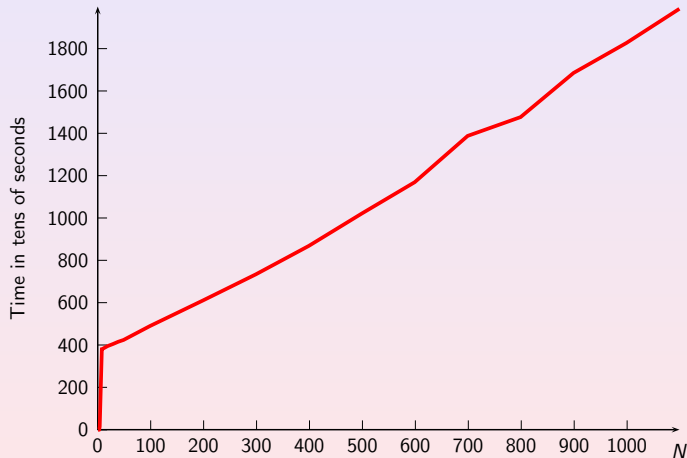
Experimental results

3-colorability on square-grids $N \times N$ (clique-width $N + 1$)



Experimental results

3-colorability on rectangular grids $6 \times N$ (clique-width 8)



Results and future work

| Property | graph | cwd | Time |
|-------------------|-----------|-----|------|
| 4-ac-colorability | petersen | 7 | 17mn |
| 3-colorability | grid 6x33 | 8 | 85mn |

Size of the graphs Limit around 1 000 000 vertices

⇒ terms of size 4 000 000

need to increase stack size because the run of an automaton on a term is **recursive**

- ▶ more graph properties
- ▶ tests on real graphs and random graphs
- ▶ graph decomposition using few labels (**parsing problem**)
- ▶ the concept of **fly-automata** is general and could be applied to other domains