Infinite transducers on terms denoting graphs

Irène Durand and Bruno Courcelle

LaBRI, Université de Bordeaux

June, 2013

European Lisp Symposium, ELS2013

Objectives

What :

Compute information about finite graphs

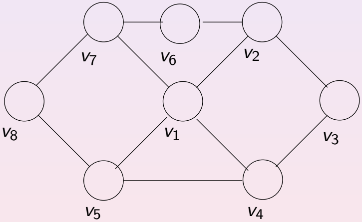
- Verify properties (boolean values)
  - has the graph a proper coloring ?
- Compute non boolean values
  - compute the number of proper colorings ?
  - compute a proper coloring

How :

- represent graphs by terms
- term automata
- term transducers

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 $(\mathcal{V}_G, edg_G)$ where $\mathcal{V}_G$ is the set of vertices and $edg_G \subseteq \mathcal{V}_G \times \mathcal{V}_G$ the binary symmetric relation that defines edges.



$$\begin{aligned}
\mathcal{V}_G &= \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\} \\
edg_G &= \{(v_1, v_2), (v_1, v_4), (v_1, v_5), (v_1, v_7), (v_2, v_3), (v_2, v_6), \\
&\quad (v_3, v_4), (v_4, v_5), (v_5, v_8), (v_6, v_7), (v_7, v_8)\}
\end{aligned}$$

Representation of graphs by terms

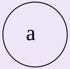
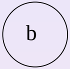
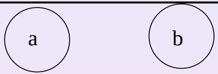
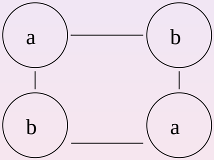
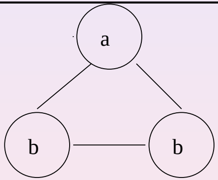
- depends on the chosen decomposition (here clique-width)
- other widths : tree-width, path-width, boolean-width, ...

Let Ports a finite set of port labels (or ports) $\{a, b, c, \ldots\}$.
Graphs $G = (\mathcal{V}_G, edg_G)$ s.t.
each vertex $v \in \mathcal{V}_G$ has a port, $port(v) \in$ Ports.

Operations :

- constant a denotes a graph with a single vertex labeled 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 $\mathcal{F}_{\text{Ports}}$ be the signature containing these operations and constants.
Every cwd-term $t \in \mathcal{T}(\mathcal{F}_{\text{Ports}})$ defines a graph G_t whose vertices are the constants (leaves) of the term t.

$t_0 = a$	$t_1 = b$	$t_2 = \oplus(a, b)$
		
$t_3 = add_{a\_b}(t_2)$	$add_{a\_b}(\oplus(t_2, t_2))$	$add_{a\_b}(\oplus(a, ren_{a\_b}(t_3)))$
		

Definition

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

History of the project

Theorem
[Courcelle (1990) for tree-width,
Courcelle, Makowski, Rotics (2001) for clique-width]
Every monadic second-order definable set of finite graphs of bounded tree-width (or clique-width) has a linear time recognition algorithm.

- the algorithm is given by a term automaton recognizing the terms denoting graphs satisfying the property
- How can we compute such automaton ?

"Courcelle's theorem is a very nice theoretical result but unusable in practice"
The project : make it work

Beginning of the project [2009]

The Autowrite Lisp system [2001–...]
first designed to verify call-by-need properties of term rewriting systems

Implements

- Terms
- Term rewriting systems
- Finite term automata (bottom-up) and operations
  - Emptiness
  - Boolean operations
  - Homomorphisms and inverse homomorphisms
  - Miminization
  - ...

A finite term automaton $\mathcal{A}$ is given by a tuple $(\mathcal{F}, Q, Q^f, \delta)$.
The transition function δ is represented by a table.

`Autograph` : ELS2010

Courcelle's theorem $+$ `Autowrite` $\Longrightarrow$ `Autograph`

Library of automata working on *cwd*-terms
(built with the $\mathcal{F}_{\text{Ports}}$ signature)
verifying graph properties expressed in MSOL (or not)

- connectedness
- k-colorability, k-acyclic-colorability
- forest
- regularity
- ...

First presentation at ELS2010 (lisbon).

- methods for computing the automata (from the formula, direct constructions)
- Some Results

Example : the Stable property

A graph is stable if it has no edge.

```
Automaton 2-STABLE
Signature: a b ren_a_b:1 ren_b_a:1 add_a_b:1 oplus:2*
States: <a> <b> <ab> error
Final States: <a> <b> <ab>

Transitions  a -> <a>
add_a_b(<a>) -> <a>
ren_a_b(<a>) -> <b>
ren_a_b(<b>) -> <b>
ren_a_b(<ab>) -> <b>
oplus*(<a>,<a>) -> <a>
oplus*(<a>,<b>) -> <ab>
oplus*(<a>,<ab>) -> <ab>
add_a_b(<ab>) -> error
add_a_b(error) -> error
oplus*(error,q) -> error for all q

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

ELS2010 : First results

Connectedness property : $|Q| = 2^{2^{cwd}-1} + 2^{cwd} - 2$
For $cwd = 4$: $|Q| = 32782$

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

Stable property : $|Q| = 2^{cwd}$ works up to $cwd = 11$.
Forest property : $|Q| = 3^{3^{cwd}}$ does not work even for $cwd = 2$.

Parallel experimentation done by Frédérique Carrère using `MONA` .

Observation : the automata are simply too big !
$\Longrightarrow$ fly-automata (easy in Lisp not in MONA)

Fly Term automata

A fly term automaton is given by $(\mathcal{F}, \delta, \mathsf{fsp})$ where

- the signature $\mathcal{F}$ may be countably infinite,
- δ is computable transition function
- fsp is the final state predicate

Implementation : the transition function δ is represented by a `Lisp` function
The complete sets of transitions, states and finalstates are never computed in extenso.

Fly automaton for the Stable property

```
(defclass stable-state (graph-state)
  ((ports :type ports :initarg :ports :reader ports)))

(defmethod make-stable-state ((ports port-state))
  (make-instance 'stable-state :ports ports))

(defmethod state-final-p ((s stable-state)) t)

(defun stable-automaton (&optional (cwd 0))
  (make-fly-automaton
   (cwd-signature cwd)
   (lambda (root states)
     (let ((*ports* (port-iota cwd))
           (*neutral-state-final-p* t))
       (stable-transitions-fun root states)))
   :name (format nil "~A-STABLE" cwd)))
```

Fly automaton for the Stable property (continued)

```
(defmethod stable-transitions-fun
    ((root constant-symbol) (arg (eql nil)))
  (let ((port (port-of root)))
    (when (or (not *ports*) (member port *ports*))
      (make-stable-state (make-ports-from-port port)))))

(defmethod stable-transitions-fun ((root abstract-symbol) (arg list))
  (common-transitions-fun root arg))

(defmethod graph-add-target (a b (s stable-state))
  (let ((ports (ports s)))
    (unless (and (ports-member a ports) (ports-member b ports))
      s)))

(defmethod graph-oplus-target ((s1 stable-state) (s2 stable-state))
  (make-stable-state
   (ports-union (ports s1) (ports s2))))

(defmethod graph-ren-target (a b (state stable-state))
  (make-stable-state
   (ports-subst b a (ports state))))
```

Fly automaton for the Stable property

```
AUTOGRAPH> (defparameter *stable* (stable-automaton))
*STABLE*
AUTOGRAPH> *t1*
oplus*(a,oplus*(b,c))
AUTOGRAPH> (recognized-p *t1* *stable*)
T
!<{abc}>
AUTOGRAPH> *t2*
add_a_b(oplus*(a,oplus*(b,c)))
AUTOGRAPH> (recognized-p *t2* *stable*)
NIL
NIL
```

Advantages of fly automata

- when finite, may be compiled to a table-automaton
- solve the space problem for huge finite automata
- yield new perspectives

Fly automata may be infinite in two ways :
infinite signature $\Longrightarrow$ may work on any clique-width

- we are no longer restricted to graphs of bounded *cwd*

infinite set of states $\Longrightarrow$ counting states, attributed states

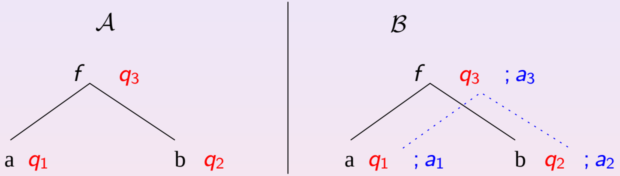
- beyond MSOL
- computation of non boolean values

- we gain in expressing power
- we loose linearity (Complexity issues discussed in CAI2013).

term automata $\longrightarrow$ term transducers

Attributed Fly Automata (deterministic case)

An attributed fly automaton $\mathcal{B}$ is a fly automaton based on a fly automaton $\mathcal{A}$ which in parallel to computing states synthetizes an attribute.



Computation of the attribute :
Function `symbol-fun (f)` which applied to a symbol f returns the function which computes the new attribute from the attributes of the children nodes.

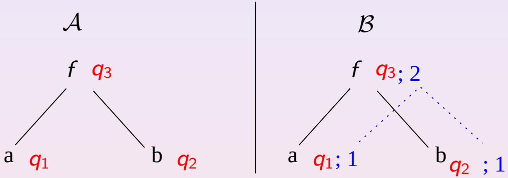
$$a_3 = (\texttt{funcall (symbol-fun f)}\ a_1\ a_2)$$

Example : computing the number of vertices

(so the number of constants)

For all constants `c`, `symbol-fun (c)` is `(lambda () 1)`

For all non constant symbol `f`, `symbol-fun (f)` is `#'+`

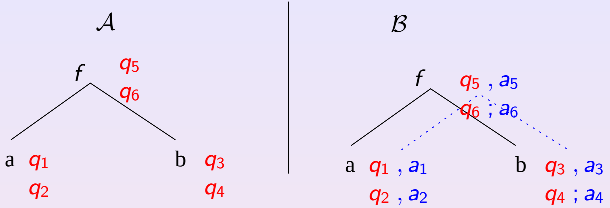


To compute the depth use

```
(lambda (&rest attributes)
  (1+ (reduce #'max attributes :initial-element 0)))
```

instead of `#'+`

Attributed Fly Automata : non deterministic case



$$\begin{cases} f(q_1, q_3) \to q_5 \\ f(q_1, q_4) \to q_6 \\ f(q_2, q_3) \to q_6 \end{cases}$$

$a_5 = (\texttt{funcall (symbol-fun f)}\ a_1\ a_3)$ $\quad a_6 = ?$

There are two ways to access state q_6

$$a_6^1 = (\texttt{funcall (symbol-fun f)}\ a_1\ a_4)$$
$$a_6^2 = (\texttt{funcall (symbol-fun f)}\ a_2\ a_3)$$

$$a_6 = (\texttt{funcall combine-fun}\ a_6^1\ a_6^2)$$

Attribution mechanism

As seen previously the mechanism to attribute an automaton requires two functions :

- symbol-fun (f) for computing attributes
- combine-fun for handling non-determinism

```
(defclass afuns ()
  ((symbol-fun :reader symbol-fun :initarg :symbol-fun)
   (combine-fun :reader combine-fun :initarg :combine-fun)))
```

Counting the number of runs :

```
(defgeneric count-run-symbol-fun (symbol))
(defmethod count-run-symbol-fun ((s abstract-symbol)) #'*)

(defvar *count-afun*
  (make-instance 'afun
                 :symbol-fun  #'count-run-symbol-fun
                 :combine-fun #'+))
```

Transducers

The run of a fly automaton on a term returns a boolean value

"Is the term recognized by the automaton ?"

A transducer is an extension of a fly automaton which may return non boolean values.

It is just a fly automaton equipped with an output function returning a value computed from the accessible final states

The run of an attributed fly automaton gives a set of attributed final states

$$\{[q_1, a_1], \ldots, [q_n, a_n]\}$$

Applying the `combine-fun` to the attributes of the final states

$$(\texttt{funcall combine-fun}\ a'_1, \ldots, a'_m)$$

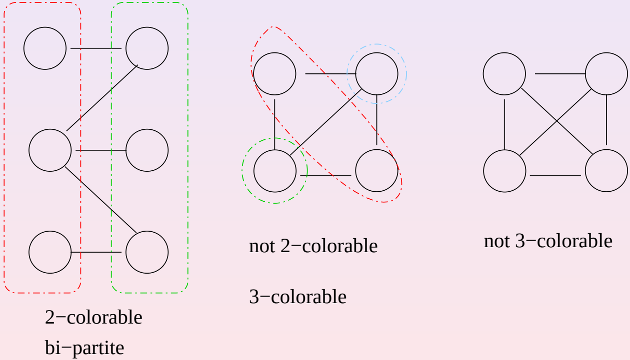
yields the final value.

Application to coloring problems

proper coloring : two vertices connected by an edge do not have the same color

k-coloring : coloring with at most k colors

A graph is k-colorable iff it admits a proper k-coloring.



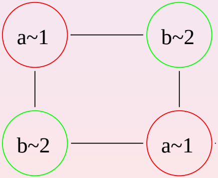
Deciding k-Colorability (NP-complete for $k \geq 3$)

Colored graphs and terms

To deal with colored graphs, we use a modified constant signature.
If we are dealing with k colors then every constant `c` yields k colored constants `c~1`, ..., `c~k`.
In a term, the constant `c~i` means that the corresponding vertex is colored with color `i`.
For instance, the term

```
add_a_b(oplus(a~1,oplus(b~2,oplus(a~1,b~2))))
```

represents the following graph properly 2-colored



Proper coloring automaton

Recognizes graphs with a proper coloring.

```
(defclass colors-state (graph-state)
  ((color-fun :initarg :color-fun :reader color-fun)))

(defgeneric coloring-transitions-fun (root arg))

(defmethod coloring-transitions-fun
    ((root color-constant-symbol) (arg (eql nil)))
  (let ((color (symbol-color root))
        (port (port-of root)))
    (when-correct-port
     port
     (make-colors-state
      (make-color-port-color-fun color port)))))

(defmethod graph-add-target (a b (colors-state colors-state))
  (let ((color-fun (color-fun colors-state)))
    (unless (intersection
             (get-colors a color-fun)
             (get-colors b color-fun))
      colors-state)))
```

Proper coloring automaton (continued)

```
AUTOGRAPH> (defparameter *2-coloring* (coloring-automaton 2))
*2COL*
AUTOGRAPH> *tcol*
add_a_b(oplus*(a~1,oplus*(b~2,oplus*(a~1,b~2))))
AUTOGRAPH> (recognized-p *tcol* *2-coloring*)
T
!<a:1 b:2>
```

Automaton for deciding k-colorability

A graph is k-colorable iff it admits a proper k-coloring.

Automaton level : projection $\Longrightarrow$ non deterministic automaton

Color projection : `color-projection(c~i)` = `c`.

```
2-coloring-automaton
; deterministic
 a~1 -> !<a:1>
 a~2 -> !<a:2>
 b~1 -> !<b:1>
 b~2 -> !<b:2>
```

```
2-colorability-automaton
; non deterministic
 a -> {!<a:2> !<a:1>}
 b -> {!<b:2> !<b:1>}
```

Other transitions identical.

```
AUTOGRAPH> (defparameter *2-colorability*
             (color-projection-automaton *2-coloring* 2))
*2-COLORABILITY*
AUTOGRAPH> *t*
add_a_b(oplus*(a,oplus*(b,oplus*(a,b))))
AUTOGRAPH> (recognized-p *t* *2-colorability*)
T
{!<a:2 b:1> !<a:1 b:2>}
```

Counting the number of proper colorings

```
AUTOGRAPH> (defparameter *2-coloring-counting* ;; count runs
             (attribute-automaton *2-coloring* *count-afun*))
*2-COLORING-COUNTING*
AUTOGRAPH> (defparameter *count-2-colorings*
             (color-projection-automaton      ;; color projection
              *2-coloring-counting* 2))
*COUNT-2-COLORINGS*
AUTOGRAPH> (compute-final-target *t* *count-2-colorings*)
{[!<a:1 b:2>,1] [!<a:2 b:1>,1]}
AUTOGRAPH> (compute-final-value *t* *count-2-colorings*)
2
T
AUTOGRAPH> (with-time
            (compute-final-value
             (petersen)
             (color-projection-automaton
              (attribute-automaton (coloring-automaton 4) *count-afun*)
 in 5.532sec
12960
```

Computing colorings

The coloring of a graph can be described by a color assignment to constant positions.
Positions in a term being represented by Dewey words in $\{0, 1\}^*$ (empty position : `E`)

```
AUTOGRAPH> (defparameter *2-coloring-assigning*
             (attribute-automaton
              *2-coloring* *assignment-afun*))
*2-coloring-assigning*
AUTOGRAPH> (defparameter *computing-2-colorings*
             (color-projection-automaton
              *2-coloring-assigning* 2))
*COMPUTING-2-COLORINGS*
AUTOGRAPH> (compute-final-value *t* *COMPUTING-2-COLORINGS*)
(([0.0:1] [0.1.0:1] [0.1.1.0:2] [0.1.1.1:2])
 ([0.0:2] [0.1.0:2] [0.1.1.0:1] [0.1.1.1:1]))
T
```

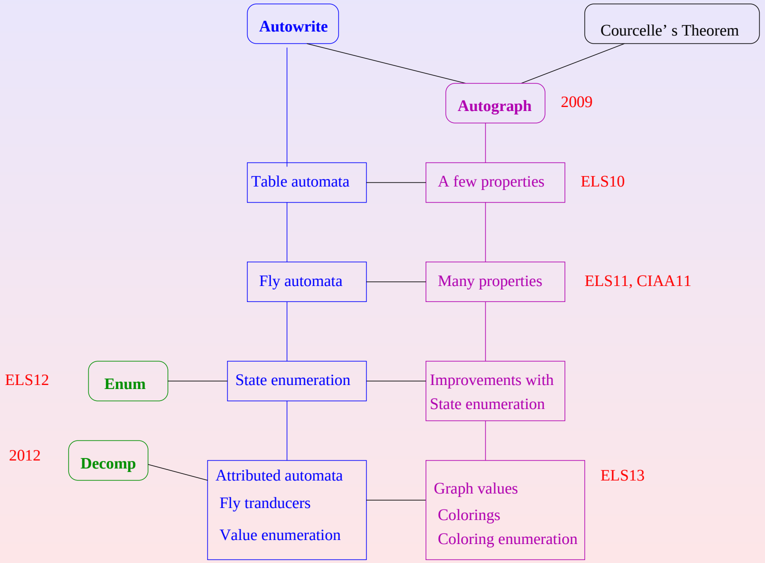
Enumerating values

The set of proper colorings of a graph is generally of exponential size.
We do not necessarily need all of them.

The enumeration mechanism presented at ELS12 is just what we need.

```
AUTOGRAPH> (defparameter *e*
             (final-value-enumerator
              (petersen)
              *computing-4-colorings*))
*E*
AUTOGRAPH> (call-enumerator *e*)
(([0.0.0.0:3] [0.0.0.1.0.0.0.0.0:4]
  [0.0.0.1.0.0.0.0.1.0.0.0.0:3]
  [0.0.0.1.0.0.0.0.1.0.0.0.1.0.0.0.0.0.0.0.0.0.0.0.0:2]
  [0.0.0.1.0.0.0.0.1.0.0.0.1.0.0.0.0.0.0.0.0.0.0.0.1:1]
  [0.0.0.1.0.0.0.0.1.0.0.0.1.0.0.0.0.0.0.0.0.0.0.1:1]
  [0.0.0.1.0.0.0.0.1.0.0.0.1.0.0.0.0.0.0.0.0.0.1:1]
  [0.0.0.1.0.0.0.0.1.0.0.0.1.0.0.0.0.0.0.0.0.1:2]
  [0.0.0.1.0.0.0.0.1.0.0.0.1.0.0.0.0.0.0.0.1:2]
  [0.0.0.1.0.0.0.0.1.0.0.0.1.0.0.0.0.0.0.1:4]))
```

Summary



Autowrite
Courcelle' s Theorem
Autograph
2009
Table automata
A few properties
ELS10
Fly automata
Many properties
ELS11, CIAA11
ELS12
Enum
State enumeration
Improvements with State enumeration
2012
Decomp
Attributed automata
Fly tranducers
Value enumeration
Graph values
Colorings
Coloring enumeration
ELS13

Future work

Short-term

- tests
- tests on real graphs and random graphs
- improve our graph decomposition system (parsing problem NP-Complete)

Long-term

- dags
- parallelism
- apply fly automata to other domains