Implementing huge term automata

Irène A. Durand
idurand@labri.fr
LaBRI, CNRS, Université de Bordeaux, Talence, France

Abstract: We address the concrete problem of implementing bottom-up term automata and in particular huge ones. An automaton which has so many transitions that they cannot be stored in a transition table is represented be a fly-automaton in which the transition function is represented by a (Lisp) function. Fly-automata have been implemented inside the Autowrite software (entirely written in Common Lisp) and experiments have been done in the domain of graph model checking.
Key Words: Tree automata, Lisp, graphs

1 Introduction

The Autowrite[1] software entirely written in Common Lisp was first designed to check call-by-need properties of term rewriting systems [Dur02]. For this purpose, it implements term (tree) automata. In the first implementation, just the emptiness problem (does the automaton recognizes the empty language) was used and implemented. In subsequent versions [Dur05], the implementation was continued in order to provide a substantial library of operations on term automata. The next natural step was to try to solve concrete problems using this library and to test its limits. The following famous theorem connects the problem of verifying graph properties with term automata.

Theorem 1. *Monadic second-order model checking is* fixed-parameter tractable *for tree-width [Courcelle (1990)] and clique-width [Courcelle, Makowski, Rotics (2001)].*

Tree-width and *clique-width* are graph complexity measures based on graph decompositions. A *decomposition* produces a term representation of the graph. For a graph property expressed in monadic second order logic (MSO), the *algorithm* verifying the property takes the form of a term automaton which recognizes the terms denoting graphs satisfying the property. In [CD10], we have given two methods for finding such an automaton given a graph property. The first one is totally general; it computes the automaton directly from the MSO formula; but it is not practically usable because the intermediate automata that are computed along the construction can be very big even if the final one is not. The second method is very specific: it is a direct construction of the automaton; one must describe the states and the transitions of the automaton. Although the direct construction avoids the bigger intermediate automata, we are still faced with

[1] `http://dept-info.labri.fr/~idurand/autowrite/`

the hugeness of the automata. For instance, one can show that an automaton recognizing graphs which are acyclic has 3^{3^k} states where k is the clique-width (see Section 3) of the graph. Even for $k = 2$, with which not very many interesting graphs can be expressed, it is unlikely that we could store the transition table of such an automaton.

The solution to this last problem is to use *fly-automata*. In a fly-automaton, the transition function is represented, not by a table (that would use too much space), but by a (Lisp) function. No space is then required to store the transition table. In addition, fly-automata are more general than finite bottom-up term automata; they can be infinite in two ways: they can work on an infinite (countable) signature. they can have an infinite (countable) number of states. This concept was easily translated into Lisp and integrated to Autowrite.

The purpose of this article is

- to present in detail the concept of fly-automaton,
- to explain how automata and especially fly-automata are implemented in Autowrite,
- to present some experiments done with these automata for the verification of properties of graphs of bounded clique-width.

2 Preliminaries: terms

We recall some basic definitions concerning terms. The formal definitions can be found in the on-line book [CDG+02]. We consider a finite signature $\mathcal{F}$ (set of symbols with fixed arity). We denote by $\mathcal{F}_n$ the subset of symbols of $\mathcal{F}$ with arity n. So $\mathcal{F} = \bigcup_n \mathcal{F}_n$. $\mathcal{T}(\mathcal{F})$ denotes the set of (ground) terms built upon a signature $\mathcal{F}$.

Example 1. Let $\mathcal{F}$ be a signature containing the symbols $\{a, b, add_{a\_b}, rel_{a\_b}, rel_{b\_a}, \oplus\}$ with

$$\begin{array}{|l|}\hline \mathsf{arity}(a) = \mathsf{arity}(b) = 0 \quad \mathsf{arity}(\oplus) = 2 \\ \mathsf{arity}(add_{a\_b}) = \mathsf{arity}(rel_{a\_b}) = \mathsf{arity}(rel_{b\_a}) = 1 \\ \hline \end{array}$$

We shall see in Section 3 that this signature is suitable to write terms representing graphs of clique-width at most 2.

Example 2. t_1, t_2, t_3 and t_4 are terms built with the signature $\mathcal{F}$ of Example 1.

$$\begin{array}{|l|}\hline t_1 = \oplus(a, b) \\ t_2 = add_{a\_b}(\oplus(a, \oplus(a, b))) \\ t_3 = add_{a\_b}(\oplus(add_{a\_b}(\oplus(a, b)), add_{a\_b}(\oplus(a, b)))) \\ t_4 = add_{a\_b}(\oplus(a, rel_{a\_b}(add_{a\_b}(\oplus(a, b))))) \\ \hline \end{array}$$

We shall see in Table 1 their associated graphs.

3 Application domain

All this work will be illustrated through the problem of verifying properties of graphs of bounded clique-width. We present here the connection between graphs and terms and the connection between graph properties and term automata.

3.1 Graphs as a logical structure

We consider finite, simple, loop-free, undirected graphs (extensions are easy)[2]. Every graph can be identified with the relational structure $\langle \mathcal{V}_G, edg_G \rangle$ where $\mathcal{V}_G$ is the set of vertices and edg_G the binary symmetric relation that describes edges: $edg_G \subseteq \mathcal{V}_G \times \mathcal{V}_G$ and $(x, y) \in edg_G$ if and only if there exists an edge between x and y.

Properties of a graph G can be expressed by sentences of relevant logical languages. For instance, G *is complete* can be expressed by $\forall x, \forall y, edg_G(x, y)$ or G *is stable* by $\forall x, \forall y, \neg edg_G(x, y)$ Monadic Second order Logic is suitable for expressing many graph properties like k-colorability, acyclicity, ...

3.2 Term representation of graphs of bounded clique-width

Definition 2. Let $\mathcal{L}$ be a finite set of vertex labels and let us consider graphs G such that each vertex $v \in \mathcal{V}_G$ has a label $label(v) \in \mathcal{L}$. The operations on graphs are $\oplus$[3], the union of disjoint graphs, the unary edge addition add_{a_b} that adds the missing edges between every vertex labeled a to every vertex labeled b, the unary relabeling rel_{a_b} that renames a to b (with $a \neq b$ in both cases). A constant term a denotes a graph with a single vertex labeled by a and no edge.

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 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.

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

4 Term automata

We recall some basic definitions concerning term automata. Again, much more information can be found in the on-line book [CDG$^+$02].

[2] We consider such graphs for simplicity of the presentation but we can work as well with directed graphs, loops, labeled vertices and edges

[3] `oplus` will be used instead of $\oplus$ inside the software `Autowrite`

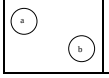
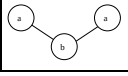
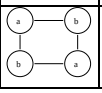
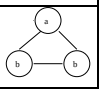
t_1	t_2	t_3	t_4
			

Table 1: The graphs corresponding to the terms of Example 2

4.1 Finite bottom-up term automata

Definition 3. A (finite bottom-up) *term automaton*[4] is a quadruple $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ consisting of a finite signature $\mathcal{F}$, a finite set Q of states, disjoint from $\mathcal{F}$, a subset $Q_f \subseteq Q$ of final states, and a set of transitions rules Δ. Every transition is of the form $f(q_1, \ldots, q_n) \to q$ with $f \in \mathcal{F}$, $\mathsf{arity}(f) = n$ and $q_1, \ldots, q_n, q \in Q$.

Term automata recognize *regular* term languages[TW68]. The class of regular term languages is closed under the Boolean operations (union, intersection, complementation) on languages which have their counterpart on automata. For all details on terms, term languages and term automata, the reader should refer to [CDG$^+$02].

To distinguish these automata from the fly-automata defined in subsection 4.2 and as we only deal with terms in this paper we shall refer to the previously defined term automata as *table-automata*.

Example 3. Figure 1 shows an example of a table-automaton. It recognizes terms representing graphs of clique-width 2 which are stable (do not contain edges). State `<a>` (resp. `<b>`) means that we have found at least a vertex labeled `a` (resp. `b`). State `<ab>` means that we have at least a vertex labeled `a` and at least a vertex labeled `b` but no edge. State `error` means that we have found at least an edge so that the graph is not stable. Note that when we are in the state `<ab>`, an `add_a_b` operation creates at least an edge so we reach the `<error>` state.

Run of an automaton on a term

The *run* of an automaton on a term labels the nodes of the term with the state(s) reached at the corresponding subterm. The run goes from bottom to top starting at the leaves.

Recognition of a term by an automaton

A term is *recognized* by the automaton when a final state is obtained at its root. Figure 2 shows in a graphical way the run of the automaton `2-STABLE` on a term representing a graph of clique-width 2. Below we show a successful run of the automaton on a term representing a stable graph.

[4] Term automata are frequently called tree automata, but it is not a good idea to identify trees, which are particular graphs, with terms.

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

Figure 1: A table-automaton recognizing terms representing stable graphs

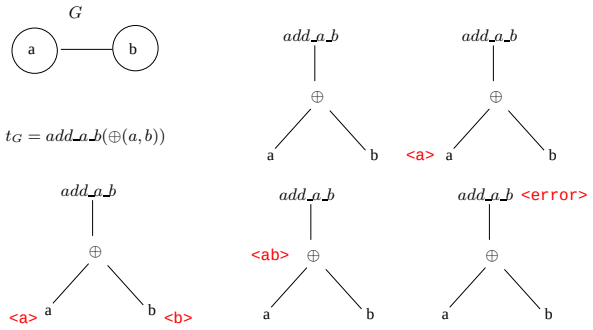


Figure 2: Graphical representation of an (unsuccessful) run of the automaton on a term

```
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>
```

4.2 Fly term automata

Definition 4. A *fly term automaton* (*fly-automaton* for short) is a triple $\mathcal{A} = (\mathcal{F}, \delta, \mathsf{fs})$ where

- $\mathcal{F}$ is a countable signature of symbols with a fixed arity,
- δ is a transition function,

$$\begin{array}{rl} \delta : \bigcup_n \mathcal{F}_n \times Q^n \to Q \\ f\, q_1 \ldots q_n & \mapsto q \end{array}$$

where Q is a countable set of states, disjoint from $\mathcal{F}$,

– fs is the final state function $\mathsf{fs} : Q \rightarrow Boolean$
which indicates whether a state is final or not.

Note that, both the signature $\mathcal{F}$ and the set of states Q may be infinite. A fly-automaton is *finite* if both its signature and set of states are finite.

Theorem 5. *Fly-automata are closed under Boolean operations, homomorphisms and inverse-homomorphisms.*

We shall call *basic* fly-automata that are built from scratch in order to distinguish them from the ones that are obtained by combinations of existing automata using the operations cited in the above theorem. We call the later *composed* fly-automata.

4.3 Relations between fly and table-automata

When a fly-automaton $(\mathcal{F}, \delta, \mathsf{fs})$ is finite, it can be compiled into a table-automaton $(\mathcal{F}, Q, Q_f, \Delta)$. The transition table Δ can be computed from δ starting from the constant transitions and then saturating the table with transitions involving new accessible states until no new state is computed. The set of (accessible) states Q is obtained during the construction of the transitions table. The set of final states Q_f is obtained by removing the non final states (according to the final states function fs) from the set of states.

A table-automaton is a particular case of a fly-automaton. It can be seen as a compiled version of a fly-automaton whose transition function δ is described by the transitions table Δ and whose final state function fs verifies to membership to Q_f. It follows that the automata operations defined for fly-automaton will work for table-automata.

Table-automata are faster for recognizing a term but they use space for storing the transitions table. Fly-automata use a much smaller space (the space corresponding to the code of the transition function) but are slower for term recognition because of the calls to the transition function. A table-automaton should be used when the transition table can be computed and a fly-automaton otherwise.

5 Implementation of term automata

5.1 Representation of states and sets of states

States for table-automata

For table-automata, the *principle* that each state of an automaton is represented by a single Common Lisp object has been in effect since the beginning of Autowrite. It is then very fast to compare objects: just compare the references. This is achieved using hash-consing techniques. Often we need to represent *sets* of states of an automaton. For fly-automata, we shall use *containers* of ordered states. Each state has an internal unique number which allows us to order states in the containers. Operations on containers (equality, union, intersection, addition of a state, ...) can then use algorithms on sorted lists which are faster.

States fly-automata

For fly-automata however, states are not stored in the representation. For basic fly-automata, they are created on the fly by calls to the transition function. It follows that the previously set out principle is not necessarily applicable.

For composed automata, the states returned by the transition function are constructed from the ones returned from the transition functions of the combined automata. For operations like determinization, inverse-homomorphisms, sets of states are involved. If a state is not represented by a unique object, comparisons of states may become very costly when states become more and more complicated. In that case, we shall have no space problem but we may get a time problem. A solution is, to apply the same principle as for table-automata, that is to say, to represent each state by a unique object. But for this we shall have to maintain a table to store the binding between some description of a state and the unique corresponding state. This table could be reset between runs of the automaton on a term. But it may happen that so many states are created by one single run that we get a space problem. In some cases, a compromise must be found between the to techniques.

5.2 Automata

The implementation of table-automata was partially discussed in [CD10]. Although the implementation of table-automata benefits from the use of Lisp, it could as well be programmed in any other general purpose programming language. The implementation of fly-automata however is much more interesting because of its use of the functional paradigm to represent and combine transition functions and its use of the object system to deal uniformly with fly or table automata.

The abstract class *abstract-automaton* generalizes the two notions of table-automaton and fly-automaton. An abstract automaton has a signature $\mathcal{F}$ and transitions.

```
(defclass abstract-automaton (named-object signature-mixin)
  ((transitions :initarg :transitions :accessor get-transitions)))
```

The concrete class `table-automaton` contains the automata whose transitions are represented by a table (`table-transitions`) and whose final states are represented by a set of states.

```
(defclass table-automaton (abstract-automaton)
  ((finalstates ...)
   ...))
```

The concrete class `fly-automaton` contains the fly-automata $(\mathcal{F}, \delta, \mathsf{fs})$ whose transitions are represented by a function (`fly-transitions`) and which have a final state function to decide whether a reached state is final.

```
(defclass fly-automaton (abstract-automaton)
  ((finalstates-fun ...)
   ...))
```

5.3 Transitions

The abstract class *abstract-transitions* generalizes the two notions of transitions: table-transitions and fly-transitions.

```
(defclass abstract-transitions () ())

(defgeneric transitions-fun (transitions)
  (:documentation
   "the transition function to be applied to
    a symbol of arity n and a list of n states"))
```

The transition function applied to a symbol f of arity n and a list of n states $q_1, \ldots, q_n$ returns what we call a *target*. The target can be `NIL` if the transition is undefined, a single state q if the transition is deterministic or a set of states $\{q'_1, \ldots, q'_{n'}\}$ otherwise.

```
(defgeneric apply-transition-function (root states transitions)
  (:documentation "computes the target of ROOT(STATES)
                   with TRANSITIONS"))

(defmethod apply-transition-function
    ((root arity-symbol) (states list)
     (transitions abstract-transitions))
  (funcall (transitions-fun transitions) root states))
```

When we have recursively computed the targets $T_1, \ldots, T_p$ for all the arguments $t_1, \ldots, t_n$ of a term $f(t_1, \ldots, t_n)$, we may compute the target for t with the operation `apply-transitions-function-gft (root targets transitions)` which applies the transition function to the elements of the cartesian product of the targets (a target which is a single state q being assimilated with the singleton $\{q\}$.

```
(defgeneric apply-transition-function-gft (root targets transitions)
  (:documentation "computes the target of ROOT(TARGETS)
                   with TRANSITIONS"))

(defmethod apply-transition-function-gft
    ((root arity-symbol) (targets list)
     (tr abstract-transitions))
  (do ((newargs (targets-product targets) (cdr newargs))
       (target nil))
      ((null newargs) target)
    (let ((cvalue
           (apply-transition-function root (car newargs) tr)))
      (when cvalue
        (setf target (target-union cvalue target))
        (when (typep target 'ordered-container)
          (assert (cdr (contents target))))
        target))))
```

The `compute-states-ra` operation implements the run of the transitions on a term $t = f(t_1, \ldots, t_n)$ given by its root f and list of arguments $(t_1, \ldots, t_n)$. It

computes recursively the targets $T_1, \ldots, T_n$ of the arguments $t_1, \ldots, t_n$ respectively and applies `apply-transition-function-gft` with f and the computed targets $T_1, \ldots, T_n$.

```
(defgeneric compute-target-ra (root args transitions)
  (:documentation
   "computes the target ROOT(args) with the TRANSITIONS"))

(defmethod compute-target-ra
    ((state abstract-state) (args (eql nil))
     (transitions abstract-transitions))
  (declare (ignore transitions))
  (declare (ignore args))
  state)

(defmethod compute-target-ra
    ((root arity-symbol) (args list)
     (transitions abstract-transitions))
  (let ((targets
         (mapcar
          (lambda (arg)
            (compute-target-ra (root arg) (arg arg) transitions))
          args)))
    (apply-transition-function-gft root targets transitions)))
```

6 Implementation of automata operations

The main operations that are implemented on all automata are:

- run of an automaton $\mathcal{A}$ on a term t,
- recognition of a term t by an automaton $\mathcal{A}$,
- decision of emptiness for $\mathcal{A}$ ($\mathcal{L}(\mathcal{A}) = \varnothing$?),
- completion, determinization, complementation of an automaton $\mathcal{A}$,
- union, intersection of two (or more) automata,
- homomorphism and inverse homomorphism on an automaton $\mathcal{A}$ induced by an homomorphism (inverse homomorphism) on the constant signature $\mathcal{F}_0$.

For table-automata, we have also implemented

- reduction (removal of inaccessible states),
- minimization.

but this is not discussed in this paper.

Some high level operations can be implemented at the level of abstract automata. This is the case for the run of an automaton, the recognition of a term.

```
(defgeneric compute-target (term automaton)
  (:documentation
   "computes the target (NIL, q or {q1,...,qk})
```

```
    of TERM with AUTOMATON"))

(defmethod compute-target ((term  term) (a abstract-automaton))
  (compute-target-ra-and-signal (root term) (arg term) a))
```

For instance, the run of an automaton $\mathcal{A}$ on a term $t = f(t_1, \ldots, t_n)$ is achieved by a call to by the operation `compute-target` on t and $\mathcal{A}$ which returns the target accessible from t using $\mathcal{A}$. When no state is accessible, the target is `NIL` otherwise when the computation is deterministic, the target is a single state otherwise it is a set (`container`) of states. A target is *final* if it is not `NIL`, if it is a single final state q or if it contains a final state q. A term is *recognized* when it reaches a final target.

```
(defgeneric recognized-p (term automaton)
  (:documentation "true if TERM is recongized by AUTOMATON"))

(defmethod recognized-p ((term term) (a abstract-automaton))
  (let ((target (compute-target term a)))
    (values
      (finaltarget-p target a)
      target)))
```

The decision of emptiness is also done at the level of `abstract-automaton` because it involves running an automaton and not creating new ones.

Determinization, Complementation, Union, Intersection, Homomorphism and inverse homomorphism can all be implemented for fly-automata. We shall detail some of these constructions further.

Because a table-automaton can always be transformed into a fly-automaton and a finite fly-automaton back to a table automaton we get the corresponding operations for table-automata for free once we have implemented them for fly-automata. However, for efficiency reasons, it might be interesting to implement some of these operations at the level of `table-automaton`. For instance, the complementation which consists in inverting non final and final states is easily performed directly on a table-automaton.

Implementing operations directly at the level of `table-automaton` has the drawback that it depends on the representation chosen for the transitions table. Whenever, we would want to change this representation we would have to re-implement these operations. The only advantage is a gain in efficiency.

Some operations on table-automata may give a blow-up in terms of the size of the transition table (determinization, intersection). In this case, the solution is to omit to compile the resulting operation back to a table-automaton.

It is though possible to deal uniformly with table and fly-automata.

6.0.1 Creation of a fly-automaton

To create a fly-automaton one should provide a signature, a transition function and a final state function:

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

6.0.2 Complementation of a fly-automaton

For a deterministic and complete automaton, the complementation consists just in complementing the final state function. The signature and the transitions remain the same.

```
(defmethod complement-automaton ((f fly-automaton))
  (let ((d (determinize-automaton (complete-automaton f))))
    (make-fly-automaton
     (signature f)
     (transitions-fun (get-transitions d))
     (lambda (state)
      (not (finalstate-p state d))))))
```

6.0.3 Determinization of a fly-automaton

If an automaton $\mathcal{A} = (\mathcal{F}, \delta, \mathsf{fs})$ is not deterministic, its transition function returns sets of states $\{q_1, \ldots, q_p\}$. The determinized version of $\mathcal{A}$ is an automaton $d(\mathcal{A}) = (\mathcal{F}, \delta', \mathsf{fs}')$. If Q is the domain of δ (the set of states of $\mathcal{A}$), let $d(Q)$ denote the set of states of $d(\mathcal{A})$. Each subset $\{q_1, \ldots, q_p\}$ of Q yields a state $[q_1, \ldots, q_p]$ in $d(Q)$. δ' is defined by with

$$\begin{array}{c} \delta' : \bigcup_n \mathcal{F}_n \times d(Q)^n \rightarrow d(Q) \\ f, S_1, \ldots, S_n \quad \mapsto S \end{array}$$

with $q \in S$ if and only if $\exists q_1, \ldots, q_b \in S_1 \times \ldots S_n$ such that $q \in \delta(f, q_1, \ldots, q_n)$. This is easily translated into Lisp:

```
(defmethod det-transitions-fun ((transitions fly-transitions))
  (lambda (root states)
    (let ((target
           (apply-transition-function-gft
            root (mapcar
                  (lambda (state)
                    (container state))
                  states)
            transitions)))
      (when target
        (make-gstate target)))))
```

And fs' is defined by

$$\begin{array}{c} \mathsf{fs} : d(Q) \rightarrow Boolean \\ S \quad \mapsto \exists q \in S \text{ such that } \mathsf{fs}(q) \end{array}$$

which translates into Lisp:

```
(defmethod det-finalstates-fun ((f fly-automaton))
  (lambda (gstate)
   (some
    (lambda (state) (finalstate-p state f))
     (contents (container gstate)))))
```

The new final state fs' calls the final state function fs of $\mathcal{A}$.

It is then obvious to determinize a fly-automaton:

```
(defmethod determinize-automaton ((f fly-automaton))
  (make-fly-automaton
    (signature f)
    (det-transitions-fun (get-transitions f))
    (det-finalstate-fun f)))
```

The following function returns a fly-automaton which recognizes stable graphs. If `cwd>0`, the automaton is finite and works on graphs of clique-width less or equal than `cwd`. If `cwd>0`, the automaton is infinite (by its infinite signature) and works on graphs of arbitrary clique-width.

```
(defun fly-stable-automaton (&optional (cwd 0))
  (make-fly-automaton
   (setup-signature cwd)
   (lambda (root states)
     (stable-transitions-fun root states))
   (lambda (state)
    (stable-finalstate-fun root states))))
```

The call `(fly-stable-automaton 2)` returns a finite fly-automaton whose compiled version is shown in Example 3.

The main task for defining a fly-automaton is to describe the states and the transition function (in the previous example `stable-transitions-fun`). This task is described in [CD10] and detailed examples can be found in [Cou09].

6.0.4 Other operations

The other operations (completion, union, intersection) are implemented in the same style. The transition function of union and intersection automata is a function which calls the respective functions of the composed automata.

7 Experiments

Most of our experiments have been run in the domain of verifying graph properties as described in 3.

7.1 Fly versus table-automata

In order to compare running time of a fly-automaton and of the corresponding table-automaton, we must choose a property and a clique-width for which the automaton is compilable.

This is the case for the *connectedness property*. We have a direct construction of an automaton verifying whether a graph is connected. The corresponding table automaton

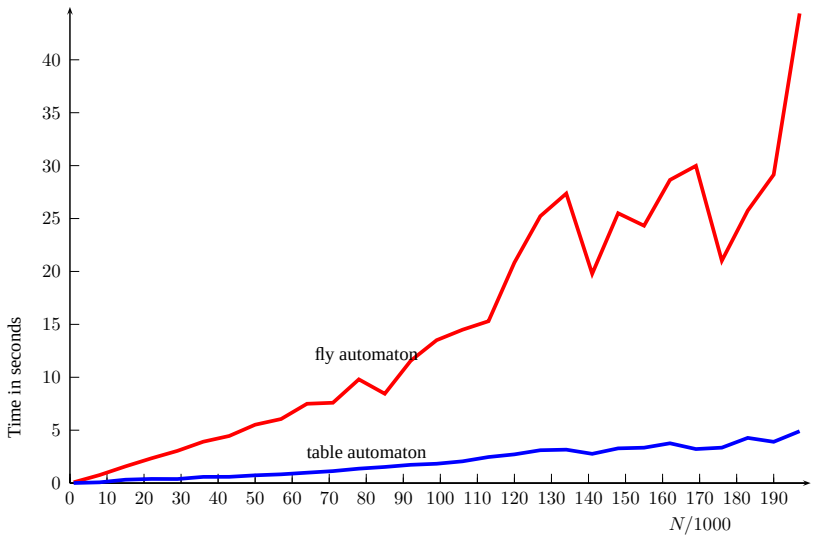


Figure 3: Connectedness on graphs P_N ($cwd = 3$)

has $2^{2^{cwd}-1} + 2^{cwd} - 2$ states. It is compilable up to $cwd = 3$. For $cwd = 4$, which gives $|Q| = 32782$, we run out of memory. It is possible to show that the number of states of the minimal automaton is $|Q| > 2^{2^{\lfloor cwd/2 \rfloor}}$. So there is no hope of having a table-automaton for this property and $cwd > 3$. The P_n[5] graphs have clique-width 3.

We could then compare the computation time with the fly-automaton to the one with the table-automaton, with increasing values of n. The size of a term representing a graph P_n is $5n + 1$ and its depth is $4n - 3$.

Figure 3 shows that the computation time is roughly linear with respect to n and that the slope of the line is steeper for the fly-automaton.

7.2 Verification of properties

We have direct constructions of the automata for the following properties.

1. Polynomial
   - Stable()
   - Partition$(X_1, \ldots, X_m)$
   - k-Cardinality()
2. non polynomial
   - k-Coloring$(C_1, \ldots, C_k)$ compilable up to $cwd = 4$ (for $k = 3$)
   - Connectedness() compilable up to $cwd = 3$

[5] A P_n graph is a chain of n vertices

- Clique() compilable up to $cwd = 4$
- Path(X_1, X_2) compilable up to $cwd = 4$
- Acyclic() not compilable

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 Acyclic)
- k-Chord-Free-Cycle()
- k-Max-Degre()
- Vertex-Cover(X_1) 2^{cwd} states
- k-Vertex-Cover()

The Vertex-Cover property can be expressed by a combination of already defined automata.

```
;; Vertex-Cover(X1) = Stable(V-X1)
(defun fly-vertex-cover (cwd)
  (x1-to-cx1 ;; Stable(V-X1)
    ;; Stable(X1)
    (fly-subgraph-stable-automaton cwd 1 1)))

(defun fly-k-vertex-cover (k cwd)
  ;; exists X1 s.t. vertex-cover(X1) and card(X1) = k
 (vprojection
  (intersection-automaton
   (fly-vertex-cover cwd) ;; Vertex-Cover(X1)
   (fly-subgraph-cardinality-automaton ;; Card(X1) = k
     k cwd 1 1))))
```

Many problems that where unthinkable to solve with table-automata could be solved with fly-automata. For very difficult (NP-complete) problems we still reach time or space limitations. Figure 4 shows the running time of a fly-automaton verifying 3-colorability on rectangular grids $6 \times N$ (clique-width 8).

8 Conclusion and perspectives

We can not think about a better language than Lisp to implement fly-automata whose transition function is represented by a function. Verifying graph properties on graphs of bounded clique-width is a perfect application field to test our implementation. In the near future, we plan to implement more graph properties and to run tests on real and random graphs.

In this paper, we did not address the problem of finding terms representing a graph, that is, to find a clique-width decomposition of the graph. In some cases, the graph of

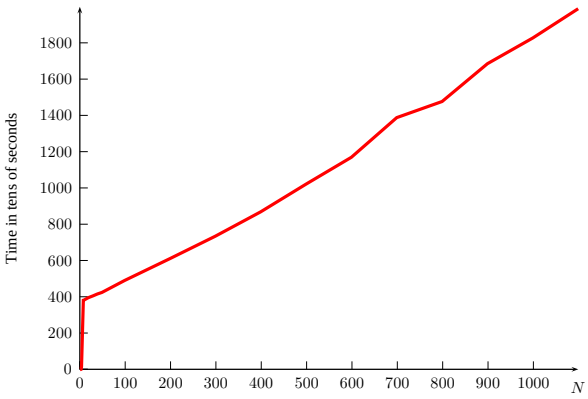


Figure 4: 3-colorability on rectangular grids $6 \times N$

interest comes with a "natural decomposition" from which the clique decomposition of bounded clique-width is easy to obtain but for the general case the known algorithms are not practically usable. This problem, known as the *parsing problem* and has been studied so far only from a very theoretical point of view. It was shown to be NP-complete in [FRRS06]. [Oum08] gives polynomial approximated solutions to solve this problem. More can be found in [Cou09]. The concept of fly-automata is general and could be applied to other domains where big automata are needed.

References

[CD10] Bruno Courcelle and Irne Durand. Verifying monadic second order graph properties with tree automata. In *Proceedings of the 3rd European Lisp Symposium*, pages 7–21, May 2010.

[CDG+02] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. 2002. Draft, available from `http://tata.gforge.inria.fr`.

[Cou09] Bruno Courcelle. Graph structure and monadic second-order logic. Available at `http://www.labri.fr/perso/courcell/Book/CourGGBook.pdf` To be published by Cambridge University Press, 2009.

[Dur02] Irène Durand. Autowrite: A tool for checking properties of term rewriting systems. In *Proceedings of the 13th International Conference on Rewriting Techniques and Applications*, volume 2378 of *Lecture Notes in Computer Science*, pages 371–375, Copenhagen, 2002. Springer-Verlag.

[Dur05] Irène Durand. Autowrite: A tool for term rewrite systems and tree automata. *Electronics Notes in Theorical Computer Science*, 124:29–49, 2005.

[FRRS06] M. Fellows, F. Rosamond, U. Rotics, and S. Szeider. Clique-width minimization is NP-hard. In *Proceedings of the 38th Annual ACM Symposium on Theory of Computing*, pages 354–362, Seattle, 2006.

[Oum08] Sang-Il Oum. Approximating rank-width and clique-width quickly. *ACM Trans. Algorithms*, 5(1):1–20, 2008.

[TW68] J.W. Thatcher and J.B. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2:57–81, 1968.