The TRAG software

by Irène Durand, and
Michael Raskin for the on-line version.

And how to use it.

By Bruno Courcelle

idurand@labri.fr, courcell@labri.fr, raskin@mccme.ru

Second version, July 9th , 2018. Do not distribute.

Introduction

The main purpose of this note is to make usable the notion of *fly-automaton* for the verification of monadic second-order (MS) graph properties [3,4].
Graphs must be defined by *clique-width terms*, hence, somehow, decomposed.
TRAG offers tools for decomposing graphs and defining them by clique-width terms [1].
Some *direct computations* are also implemented : chromatic polynomial, connectedness, acyclicity.
The tools handle *incidence graphs and MS2 properties*, i.e. those defined by MS sentences using edge set quantifications [2].

Main references

[1] B. Courcelle, From tree-decompositions to clique-width terms, *Discrete Applied Mathematics*, In press. https://hal.archives-ouvertes.fr/hal-01398972

[2] B. Courcelle, Fly-automata for checking MSO2 graph properties. *Discrete Applied Mathematics*, **245** (2018) 236-252.
See https://arxiv.org/abs/1511.08605# or https://hal.archives-ouvertes.fr/hal-01234622

[3] B. Courcelle, I. Durand, Automata for the verification of monadic second-order graph properties, J. Applied Logic **10** (2012) 368-409. See : http://hal.archives-ouvertes.fr/hal-00611853/fr/ .

[4] B. Courcelle, I. Durand, Computation by fly-automata beyond monadic second-order logic, Theoretical Computer Science, **619** (2016) 32-67, See http://hal.archives-ouvertes.fr/hal-00828211

Additional references are in the appendix.

Summary :

1 Easy examples

Open http://trag.labri.fr

Example 1 : In menu **`Graphs`**, select **`Standard Graphs`** and choose **`Gnxn`**
Choose *n = 4* and **`Accept`**
In same menu **`Display graph`**.
In **`Graph decomposition`**, select **`Heuristic cwd-decomposition`**.
You get the top part of a term of width 6.
Then try **`Clique-width decomposition`**.
You the top part of a term, of exact (optimal) width 5.
In the menu **`Automata`**, select **`color automata`** and choose **`Kcolorability.`**
Choose *K = 2* and **`Accept`**
The automaton *2colorability* is the current one.
In same menu, select **`Application`** and **`Recognize term`**
« Recognized » means that the graph is 2-colorable.
By selecting **`count colorings`** and with 2 colors, you get 2 colorings.
If you **`count colorings`** with 3 colors, you get 7812 colorings.

In the menu **graph properties**, you can ask for **Graph hamiltonian.** The answer is *yes*.

Example 2 : Let us try now the grid **G5x5** (**Clique-width decomposition** will take some time).
It is not Hamiltonian (so are all grids with odd number of vertices)
and you get this answer.

Example 3 : Let us try now the computed graph ; Petersen.
In **Graph decomposition**, select **Heuristic cwd-decomposition**.
You get 6 as an upper-bound to clique-width.
Clique-width decomposition gives the exact value 5.
In the menu **Automata**, select **color automata** and choose **Kacyclic colorability.**
Choose *K=3*, **Accept,** and in **Application,** select **Recognize.**
The answer is « not recognized ».
With *K = 4*, the answer is is « recognized ». Petersen's graph is 4-acyclically colorable, but not 3-acyclically colorable.
If you select select **Enum recognize**, the answer is quicker.

2 How to input graphs

Parallel edges get fused. Hence, only simple graphs are faithfully handled. (However, see **Incidence graphs** below).

Unoriented graphs :
Loops are not recognized.
Graphs can be input from the menu **Graphs** : **Input nonoriented** as a union of paths.

Example : **Paths** 1 2 3 7 5 ; 1 3 6 10 9 (8 vertices, 8 edges).
Renumber vertices make them into an interval from 1 to some *n*. The renumbering preserves the relative order.

Standard graphs : Paths, cycles, cliques , square grids **Gnxn** , rectangular grids **Gnxm**,
Computed graphs : Petersen, Grunbaum, McGee
Load description in **DIMACS format** (ascii file).
Example : Vertices from 1 to 8, 10 edges.
c insert here your comments, the name of the graph
p edge 8 10
e 1 2

e 1 4
e 1 7
e 1 8
e 2 3
e 2 6
e 2 8
e 3 5

Graph defined by a term (see below).

Graphs can be vizualized at each step of the description by the command **`Display graph`** Planar graphs are planarly displayed (by breaking interesting symmetries is some cases).

Graphs can be modified by **`Add path`** or **`Remove path`**. One can add a cycle : *e.g.* 1 4 6 1. Each time do not forget to push **`Accept`**
Adding a loop by **`Add path`** : 1 1 yields an error on display. Adding edge 1 2 if it exists already has no effect.

Oriented graphs.
They can have loops (but 2 loops at a same vertex get fused into one).
They can be input from the menu **`Graphs`** :
`Input oriented` defines a graph as a union of oriented paths, by **`Paths.`**
Paths are oriented as they are inserted within the oriented graph description.
Example : 1 2 3 7 5 ; 1 3 6 10 9 (8 vertices, 8 edges).
Inputing the path 1 1 yields a loop on vertex 1.

`Renumber vertices` make them into an interval from 1 to *n,* the number of vertices.

Load in **`DIMACS format`** for oriented graphs.
Example : Vertices from 1 to 5, 6 arcs (oriented edges).
c insert here your comments, the name of the graph
p arc 5 6
a 1 2
a 1 4
a 1 5
a 5 1
a 2 3
a 2 5

The current graph can be modified by **`Add path`** or **`Remove path`**. Added paths

are oriented. One can add a directed cycle by **`Add path`** : 1 4 6 1.

A graph can defined by a term (see below).

An unoriented graph can be given a **`Random orientation.`** To the opposite, one can **`Erase orientation`**. Two opposite directed edges become a single undirected one.
Loops cause errors and should be removed beforehand by **`Remove path`**.

Naming and saving graphs.

The current graph can be saved in DIMACS format by the command **`Download DIMACS`**. Vertices must be consecutively numbered.

The menu **`Named objects`** permits to name the current graph and to retrieve it by its name. Several graphs can be named during a session. They are lost when closing the session. But the current graph can be saved, see 8 below.

3 Decompositions

TRAG uses clique-width terms as inputs to automata.

Note on syntax (see examples below) :

`oplus(.,.)` denotes disjoint union : .
Labels are letters `a,b,c` … possibly tagged : `a^0, a^1`,... (see incidence graphs),
`ren_d_a` relabels d into a,
`add_a->c` creates oriented edges from `a`-labelled vertices to `c`-labelled ones,
`add_a_c` creates unoriented edges between a-labelled vertices and c-labelled ones,
`c[6]` creates vertex number 6 with label `c`.

The menu **`Graph decomposition`** offers the following algorithms for a graph previously specified :
`Heuristic cwd-decomposition` for an oriented or unoriented graph : it yields a term and displays its width that bounds the exact clique-width.
`Clique-width decomposition`, yields for an unoriented graph, the (exact) clique-width and one corresponding term.
`Try specific clique-width` tries to find a term with the specified number of

labels. In both cases, he input graph must have vertices consecutively numbered from 1. Use **Renumber vertice**s if necessary.

CAUTION : These last two algorithms work for graphs with, say, no more than 20 vertices and 40 edges, and « small » clique-width, say 6 (otherwise the servor may crash).
They are based on reductions to SAT problems solved by GLUCOSE (see reference [5]). The reduction is due to Heule and Szeider [6].
For larger graphs, use rather **Heuristic cwd-decomposition.**

Heuristic twd-decomposition, for an unoriented graph, produces a tree-decomposition, expressed by normal trees, cf. [1]. See Appendix for an example.
Heuristic cwd-twd-decomposition for an unoriented graph, produces a clique-width term constructed from a tree-decomposition (cf. [1]).
To process oriented graphs, one removes orientation. Tree-width is insensible to orientation and parallel edges.

The menu **terms** helps to understand the produced clique-width terms.
It gives several numbers : **clique-width**, **depth**, **size. Draw** shows the graph with the « final » labelling of vertices.
The term shows the nullaries that specify vertices. *Example* : `c[6]` to specify vertex 6 with « initial » label `c`.

Show multiterm splits a term into subterms, for example, the term

```
     t = add_c->b(oplus(c[6],ren_d_a(ren_c_a(add_a-
>d(add_d->c(
     oplus(d[1],add_b->a(add_c-
>a(oplus(a[4],oplus(b[7],c[3])))))))))))
```

is *t= t0* and splitted as follows :

```
     t0 = add_c->b(oplus(c[6],t1)).
     t1 = ren_d_a(ren_c_a(add_a->d(add_d-
>c(oplus(d[1],t2))))).
     t2 = add_b->a(add_c->a(oplus(a[4],t3))).
     t3 = oplus(b[7],c[3]).
```

This description shows the corresponding decomposition of the graph and the way vertices are created (say vertex 6 with label `c` by the nullary `c[6]`)

Download multiterm saves this sequence of terms into a file.

Use **Input multiterm** or **input term** to import a term. It must be copy-

pasted in the relevant box.
Then, the command **Term to graph** converts the term into a graph, that can be displayed.
Some **Standard terms** and **Computed terms** are precomputed for standard and computed unoriented graphs (see above).

For **Annotations** see [3]. They are used in coloring algorithms (see below).

For incidence graphs, see below.

If a graph is given by a term t (commands **Input term** or **Input multiterm**) an automaton checks whether t is well-written and does not use edge addition operations for oriented *and* unoriented edges (it does not use `add_a->b` *and* `add_c_d`).

4 Automata based computations

The system allows general fly-automata computations, cf. [3,4].
Some automata concerning *coloring problems* are implemented.
In all cases, a clique-width term must have been computed as the « current term ».

When an automaton is selected via the menu **Automata**, one must go to **Application** and select :
Recognize : says *yes* or *no*, whether the graph has the considered property. The automaton is run « deterministically ».
Enum recognize : says *yes* or *no.* The automaton enumerates the set of states reached at the root and stops as soon as an accepting one is found. This algorithm may give a quick answer in case of *Yes*. Because of the preprocessing, it may take more time than the previous one.
Compute # : Computes the number of accepting runs
Compute Sp : Computes the spectrum : See Appendix
Compute MSp : Computes the multi-spectrum : See Appendix
Compute SAT : Computes the set of satisfying tuples. See Appendix

Some commands are not available for certain automata.

Color automata :
They only concern unoriented graphs (orientations can be removed, see 2 above).
Coloring : One specifies a number of colors.
Applicable functions are **Recognize, Compute #, Compute Sp, Compute Msp, Compute SAT**

K-coloring : checks if the graph is K-colorable ; one has to choose the value K = the number of allowed colors.
Applicable functions are **Recognize, Enum recognize,Compute #,**
K-coloring with symmetries and annotations :
Same as **K-coloring** but it *requires an annotated term.* (Use the **Annotation** command, see above. Implements an algorithm by M.Raskin [7]).
counting colorings : gives the number of *K*-colorings ; one has to choose the value *K*.
K-acyclic coloring : checks if the graph is *K*-acyclically-colorable ; one has to choose the value *K*. (*Acyclic coloring* means that the union of any two color classes induces a forest).
Show one coloring provides for example, a 3-coloring of the cycle `C5` is :

```
[C1:(1) C2:(2 4) C3:(3 5) ]
```

showing that color `C1` is used for vertex 1, color `C2` for vertices 2 and 4 and color `C3` for vertices 3 and 5.
Show all colorings yields the list of all such words representing all colorings.

Looking for a term of clique-width at most k recognized by a chosen automaton.

The command **Emptyness** indicates the existence. **Show nonemptyness witness** yields some term of given maximal clique-width.
CAUTION : To be used for « small » bounds on clique-width and « small » automata.

5 Direct computations

The menu **Graph direct computations** gives the **Chromatic polynomial**, and its value (the **chromatic value**) for a given number of colors.
Permits to verify that the previously computed numbers of colorings are correct.
From the menu **Graph Properties** one can check **Hamiltonicity** and **connectedness** for oriented or unoriented graphs, and **acyclicity** for unoriented graphs only.

One need not request a decomposition.

6 Incidence graphs [2]

The Incidence graph *Inc(G)* of a graph *G* is bipartite. Its vertex set is $V_G \cup E_G$. The

part V_G of $V_G \cup E_G$. is known by a bit. See below a worked example.

If G is oriented, its edges are $x \longrightarrow e$ and e $\longrightarrow y$ whenever e is an edge from x to y.
If G is unoriented and loop-free, *Inc(G)* has an unoriented edge $x - e$ if e is an edge with an end x.
Edge-representing vertices have degree 2 in both cases.
Multiples edges in G can be distinguished in *Inc(G)* by distinct edge-representing vertices.

A bipartite graph $H=(V \cup E,inc)$, where $inc \subseteq V \times E \cup E \times V$ is defined as an inc-graph iff it is is as follows :

either $inc \subseteq V \times E$ and each e in E has degree exactly 2,

or $inc \subseteq V \times E \cup E \times V$ and each e in E has indegree 1 and outdegree 1.
In the former case, $H = Inc(G)$ for a loop-free unoriented graph G.
In the latter one, $H = Inc(G)$ for an oriented graph G that may have loops.
In both cases, G may have parallel edges.
Certain of these inc-graphs cannot be defined from graphs G (by the **Graphs** menu) because G has no parallel edges. They must be defined directly by terms, in case one needs to deal with parallel edges and oriented loops.
See the term t below for example.

For the graph `C5`, the command **Optimal incidence cwd-decomposition** yields the term t :

```
add_c_d(oplus(ren_c_a(add_a_b(oplus(oplus(add_a_d(oplus(
a^0[1],d^1[7])),
ren_b_c(add_a_b(oplus(a^0[2],ren_a_c(add_a_c(oplus(
oplus(add_b_c(oplus(c^0[3],b^1[8])),
add_c_d(oplus(c^0[4],d^1[10]))),a^1[9])))))),b^1[6]))),
c^0[5])
```

The tag ^0 in labels `a^0, c^0` indicates that the corresponding vertices : 1, 2, … 5 are vertices of the original graph.
The tag ^1 in labels `a^1, b^1, d^1` indicates that the corresponding vertices : 6,7, … 10 represent edges of the original graph.

Here is a correct *incidence term,* not produced by **Optimal incidence cwd-decomposition,** because of one parallel edge added :

```
add_c_d(oplus(ren_c_a(add_a_b(oplus(oplus(add_a_d(oplus(
a^0[1],
```

```
oplus(d^1[7], d^1[11])),
ren_b_c(add_a_b(oplus(a^0[2],ren_a_c(add_a_c(oplus(
oplus(add_b_c(oplus(c^0[3],b^1[8])),
add_c_d(oplus(c^0[4],d^1[10]))),a^1[9])))))),b^1[6]))),
c^0[5])
```

The modified subterm is underlined.

In the menu **Terms** the command **Term to incidence graph** determines, from a clique-width term, the incidence graph of the graph it defines. It works for oriented and unoriented graphs. The command **Incidence term to graph** determines from a correct incidence term (that can be uploaded by **Input term**) a graph where parallel edges get fused.

7 MS2 properties

Hamiltonicity (looking for a Hamiltonian cycle ; an MS2 property that is not MS expressible and is NP-complete) is implemented in different ways for oriented and unoriented graphs.

For an oriented graph, one must request
Heuristic incidence cwd-decomposition
and then, one chooses the automaton **Oriented hamiltonian**.

For an unoriented graph, one requests
Heuristic incidence cwd-decomposition or
Optimal incidence cwd-decomposition
and then, one chooses the automaton **Unoriented hamiltonian**.

8 General commands

Menu **Sesssion**
Reset,
Cancel a computation (if it takes too much time).
Export
Saves in an ascii file with extension .json the current graph, term, automaton and computed value (« recognized » or not, number of runs etc.)
Import
Retrieves the file to restaure the session.
Force to server
Restores from your computer a session interrupted for some reason.

9 How to build automata (Advanced constructions).

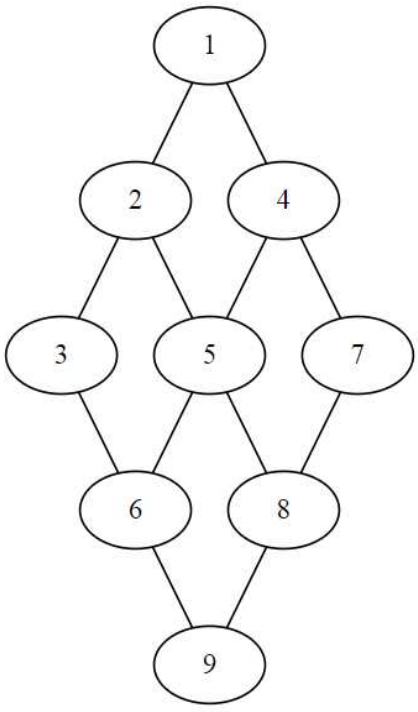
Automata are built by TRAG from existing **`basic automata`** by operations reflecting logical operations (cf [3] for thorough description).
The submenu **`Formula automata`** of **`Automata`** permits an automatic construction of a fly-automaton (working for any clique-width term) from a formula. There are two variants, for oriented and unoriented graphs.

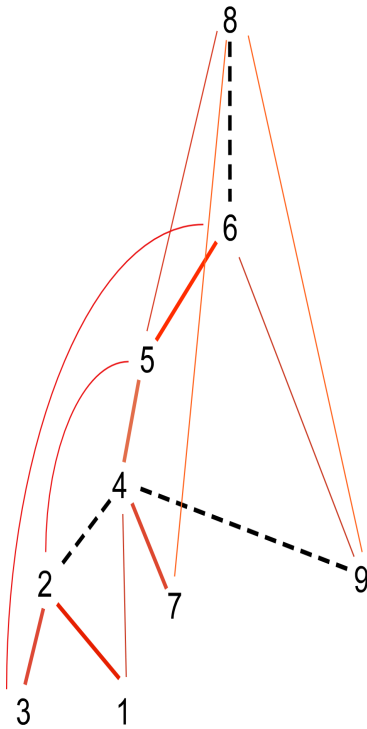
The syntax of formulas is based on that of TPTP (see http://www.cs.miami.edu/~tptp/). It is described in the Appendix, where examples are given.

10 Appendix

Normal tree-decompositions

Here is the example of the grid 3x3.





We obtain the following word that describes a normal tree-decomposition.
((8 6) (6 5) (5 4) (4 2) (4 9) (4 7) (2 3) (2 1))

(8:NIL 6:(8) 5:(6 8):(6 8) 4:(5 6 8):(5) 2:(4 5 6):(5) 9:(6 8):(6 8) 7:(4 8):(4 8) 3:(2 6):
(2 6) 1:(2 4):(2 4))

The first part :
((8 6) (6 5) (5 4) (4 2) (4 9) (4 7) (2 3) (2 1)) indicates that the tree has nodes 1 to 9 (the vertices), 8 is the root, 5 is a son of 8, 4 has sons 2,7 and 9, 2 has sons 1 and 3, etc.

The box of a node *u* contains always *u* (it is a vertex of the graph), its neighbours above it on the tree and some other nodes.
The second part, (8:NIL 6:(8) 5:(6 8):(6 8) 4:(5 6 8):(5) 2:(4 5 6)..... shows that the boxes are as follows :

Node *u*	Box of *u*	Neighbours of *u* in its box
8, the root	8	
6	6,8	
5	5,6,8	6,8
4	4,5,6,8	5
2	2,4,5,6	5
7	7,4,8	4,8
9	9,6,8	6,8
3	3,2,6	2,6
1	1,2,4	2,4

The edges of the graph are in red (thin or thick). The edges of the tree are thick, either red or dotted and black.

Spectrum, Multispectrum and SAT

The *spectrum* of a property $P(X_1,...,X_p)$ is the set of p-tuples of cardinalities of the p-tuples of sets $(X_1,...,X_p)$ that satisfy P.
The *multispectrum* of $P(X_1,...,X_p)$ is the multiset of p-tuples of cardinalities of the p-tuples of sets $(X_1,...,X_p)$ that satisfy P. Hence, each p-tuple of integers is given with the number of p-tuples of sets that define it.
The satisfying set (query answer) of a property $P(X_1,...,X_p)$ is the set of p-tuples of sets $(X_1,...,X_p)$ that satisfy P.

We give small examples.
The term `add_a_b(oplus(oplus(a[1],oplus(b[2],b[4])),a[3]))` defines the 4-cycle `C4`

It has 18 3-colorings (by **`Compute #`**).
The answer to **`Compute Sp`** is the string
#(1 2 1) #(2 1 1) #(2 2 0) #(0 2 2) #(1 1 2) #(2 0 2)
It shows 2 types of 3-colorings (6 with the permutations of colors).

The answer to **`Compute MSp`** shows :
#(1 2 1):4 #(2 1 1):4 #(2 2 0):2 #(0 2 2):2 #(1 1 2):4 #(2 0 2):2
indicating the numbers of colorings for each 3-tuple of possible cardinalities of color classes.
We have 18 = 4 + 4 +2 +2 +4 +2.

The answer to **`Compute SAT`** shows :

<[0.0.0:3] [0.0.1.0:1] [0.0.1.1:2] [0.1:3]>
<[0.0.0:3] [0.0.1.0:2] [0.0.1.1:1] [0.1:3]>
<[0.0.0:3] [0.0.1.0:1] [0.0.1.1:1] [0.1:3]>
<[0.0.0:3] [0.0.1.0:2] [0.0.1.1:2] [0.1:3]>
… and 14 more.

The sequence [0.0.0:3] [0.0.1.0:1] [0.0.1.1:2] [0.1:3] indicates that the vertex at position 0.0.0 in Dewey notation in the used term is in the third set, i.e. has color 3. The vertex at position 0.0.1.0 has color 1. Etc. For colorings, the commands **`Show one coloring`** and **`Show all colorings`** are easier to read as they indicate the vertex number rather than the corresponding Dewey word.

MS formulas in the syntax of TPTP.

Because of the translation of formulas into automata, we will express universal quantification in terms of existential quantification and negation.
All variables are of the form `X, Y, Z33, Uprime` *etc* and denote sets. Individual variables are handled as set variables denoting singletons.

Ascii coding of logical connectives :

?	there exists
!	for all
&	and
|	or
~	not
<=>	equivalent

=> implies
<= implied by
= equals
!= not equal to

Blank spaces are not significant. Atomic formulas must start with a lowercase letter, and variables by an uppercase one.

Existential quantification is written `?[X1,...,Xp]: (...)`
Universal quantification is written `![X1,...,Xp]: (...)`

The atomic formulas relative to sets are :

`X=Y,`
`X!=Y,`
`subset(X,Y)` meaning that X is a subset of Y,
`singleton(X)` meaning that X is singleton,
`empty(X)` meaning that X is empty
`card(i,X)` meaning that X has cardinality exactly i, and i is a fixed integer.

The atomic formulas relative to basic graph structure are :

`someedge(X,Y) : X,Y` are disjoint and there is an edge between some vertex of `X` and some vertex of `Y ;` when dealing with oriented graphs, this edge is directed from `X` to `Y`.

`alledges(X,Y) : X,Y` are disjoint and there is an edge between each vertex of `X` and each vertex of `Y ;` when dealing with oriented graphs, these edges are directed from `X` to `Y`.

`edge(X,Y) : path(X,Y)` : `X` consists of two vertices and there is an unoriented path between them, all vertices of which are in `Y` (which implies that `X` $\subseteq$ `Y`).

`stable(X)` : there are no edges between any two vertices, and no loop incident with a vertex in `X`,

Set variables can be replaced by set terms with obvious meanings :

`intersection(X,Y)`
`union(X,Y),`
`difference(X,Y)` denoting X-Y,
`xor(X,Y)` denoting the symmetric difference of X and Y,
`complement(X).`

Hence, we can write : `alledges(union(X,Y),difference(U,Z)).`

When considering incidence structures, the variable `Edges` denotes the edges in the considered incidence graph. It must be put in the set of declared variables, say for use in the formula `subset(Edges,X)` meaning that all edges are in `X`.

Two constants are `empty` and `universal` denoting the empty set and the full domain. Hence, `path(X,universal)` means that there is a path between the two vertices of `X` (see below for `path`).

More atomic formulas can be used :

`path(X,Y)` : `X` consists of two vertices and there is an unoriented path between them, all vertices of which are in `Y` (which implies that `X` $\subseteq$ `Y`).

`oripath(X1,X2,Y)` : `X1,X2` are singletons and there is an oriented path from `X1` to `X2`, all vertices of which are in `Y`.

`connected(X)` : `X` induces a connected graph,

That a graph is regular is not an MS property but can be checked by a fly-automaton [4]. A corresponding command will be added.

Multi-formulas : A formula is split into subformulas for readability and smaller size. Each component, including the main one, usually put first, must be named with a possibly empty list of arguments. See Examples 2 and 4.

The submenu **`Formula Automata`** of **`Automata`**
has 4 commands for oriented or nonoriented graphs and possibly, multi-formulas.

For multiformulas, the command **`Expand`** opens a large window to paste the formula. Then, **`Accept`,** the automaton is (hopefully) constructed.

Example 1 : A formula expressing that an unoriented graph is complete.

```
~?[X1,X2]: (singleton(X1) & singleton(X2) &
~subset(X1,X2) & ~edge(X1,X2))
```

Example 2 : A multi-formula expressing that the graph is bipartite :

```
bipartite( ) : ?[X,Y]:(st(X)& st(Y)&(~ ?[U] :
(singleton(U) & ((subset(U,X)& subset(U,Y)) |
 ( ~ subset(U,X) & ~ subset(U,Y)))) )).
st(X) : ~?[U,V]: ( subset(U,X) & subset(V,X) &
```

```
edge(U,V)).
```

Here, the stability predicate is defined as `st(X)` and used also as `st(Y)` in the main formula, via a substitution of variable.

Example 3 : Good 3-colorings of oriented graphs :

This property says that there is a coloring with colors 1,2,3 such that all edges are directed from 1 to 2, 2 to 3 and 3 to 1. Hence, the colors of ends determine the directions of edges.

```
    ?[X,Y] : (empty(intersection(X,Y)) & ![U,V]:  (
    edge(U,V) => ((subset(U,X) & subset(V,Y)) |
(subset(U,Y) & ~subset(V,X) & ~subset(V,Y)) |
(~subset(U,X) & ~subset(U,Y) &    subset(V,X)))))
```

Verification : `t`he directed cycles with 3 and 6 edges are recognized but the one with 5 edges is not.

Example 4 : the existence of a path between two vertices.

This example concerns unoriented graphs. We consider the property *path2(x_1,x_2,Y)* simpler to write than `path(X,Y)` meaning that :

x_1 and x_2 are distinct vertices and there is an unoriented path between x_1 and x_2,

all vertices of which are in *Y* (which implies that x_1 and x_2 are in *Y*).

x_1 and x_2 will be handled as singleton sets `X1` and `X2`.

Let the auxiliary property `closed(U,Y)` mean that : $U \subseteq Y$ and for all vertices *u* in *U* and *v* in *Y*, if *u* in *U* and *v* in `edge(u,v)` holds, then v is in U

equivalently : $U \subseteq Y$ and there does not exist *u,v* that are neighbours with *u* is in *U* and *v* in *Y-U*.

Then, `path2(x1,x2,Y)` is equivalent to :

x2 is not x1 and, for every U such that `closed(U,Y)` holds, if x1 is in U, then x2 is in U.

The auxiliary formula `q(Z1,Z2,W)` means : for every U such that `closed(U,W)` holds, if Z1 is included in U, then Z2 is included in U.

Hence, we can define `path2(X1,X2,Y)` by the following multi-formula consisting of three formulas :

```
    path2(X1,X2,Y): (singleton(X1) & singleton(X2) &
subset(X1,Y) & subset(X2,Y) &  ~subset(X1,X2) &
q(X1,X2,Y) ).
    q(Z1,Z2,W): ~?[U]: ( subset(Z1,U) & ~subset(Z2,U)&
closed(U,W)  ).
    closed(U,W): subset(U,W)&  ~?[U1,W1]: (subset(U1,U)&
subset(W1,W)& ~subset(W1,U) & edge(U1,W1) ).
```

Additional references

[5] G. Audemard and Laurent Simon, On the Glucose SAT solver. *International Journal on Artificial Intelligence Tools* 27 (2018) 1-25.

[6] M. Heule and S. Szeider, A SAT approach to clique-width. *ACM Trans. Comput. Log.*16 (2015) 24:1-24:27.

[7] M. Raskin, Enumerating colourings via clique-width and colour renaming, Slides.