Set Graphs VI
Logic Programming and Bisimulation

Agostino Dovier

Università di Udine
Dip. di Matematica e Informatica

TORINO, June 18, 2014

Introduction

- Several forms of graph Equivalence are used in computer science
- Graph/Subgraph isomorphism are central notions in complexity theory
- Graph (DFA) minimization is a key notion in Hardware definition
- Graph/Sugraph bisimulation is used in concurrency theory, temporal logic, model checking, web databases, and, of course, in hyper-set theory
- We focus on the graph bisimulation problem and consider its encoding(s) in logic programming paradigms.

Sets Basics

Set equality: The *extensionality principle* (E)

$$\forall z\Big((z \in x \leftrightarrow z \in y) \rightarrow x = y\Big) \qquad (E)$$

Sets Basics

Set equality: The *extensionality principle* (E)

$$\forall z\Big((z \in x \leftrightarrow z \in y) \rightarrow x = y\Big) \qquad (E)$$

Well-foundedness of $\in$: The *foundation axiom* (FA):

$$\forall x\left(x \neq \emptyset \rightarrow (\exists y \in x)(x \cap y = \emptyset)\right) \qquad (FA)$$

that ensures that a set cannot contain an infinite descending chain $x_0 \ni x_1 \ni x_2 \ni \cdots$ of elements.

Sets Basics

Set equality: The *extensionality principle* (E)

$$\forall z\Big((z \in x \leftrightarrow z \in y) \to x = y\Big) \qquad (E)$$

Well-foundedness of $\in$: The *foundation axiom* (FA):

$$\forall x \Big(x \neq \emptyset \to (\exists y \in x)(x \cap y = \emptyset)\Big) \qquad (FA)$$

that ensures that a set cannot contain an infinite descending chain $x_0 \ni x_1 \ni x_2 \ni \cdots$ of elements.

In particular, if x is s.t. $x = \{x\}$ then x is not empty, its unique element y is x itself, and $x \cap y = \{y\} \neq \emptyset$ contradicting the axiom.

Sets as graphs

An *accessible pointed graph (apg)* $\langle G, \nu \rangle$ is a directed graph $G = \langle N, E \rangle$ together with a distinguished node $\nu \in N$ (the *point*) such that all the nodes in N are reachable from ν.

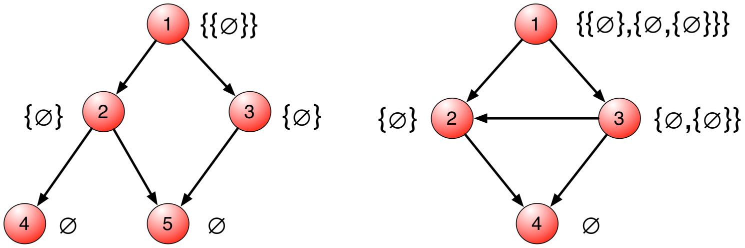
Intuitively, an edge $a \longrightarrow b$ means that the set "represented by b" is an element of the set "represented by a".

$$a \longrightarrow b \quad a \longrightarrow\!\!\!> b \quad a \ni\!\!\!- b \quad a \ni b$$

The above idea can be used to *decorate* an apg, namely, assigning a (possibly non-well founded) set to each of the nodes.

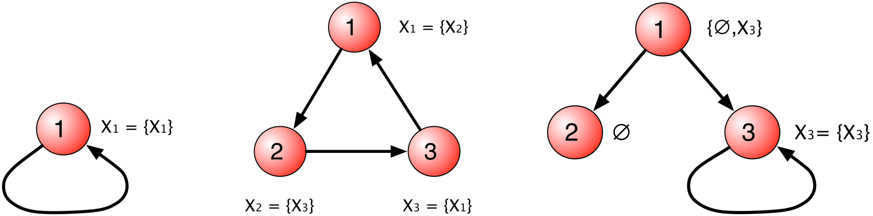
Sinks, i.e., nodes without outgoing edges have no elements and are therefore decorated as the empty set $\emptyset$.

Sets as graphs



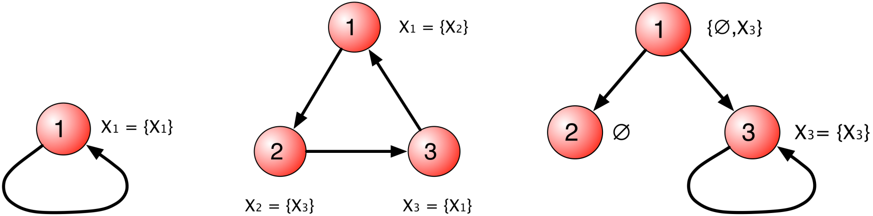
Cyclic graphs and hypersets

If the graph contains cycles, interpreting edges as membership implies that the set that decorates the graph is no longer well-founded.
Non well-founded sets are often referred to as *hypersets*.



Cyclic graphs and hypersets

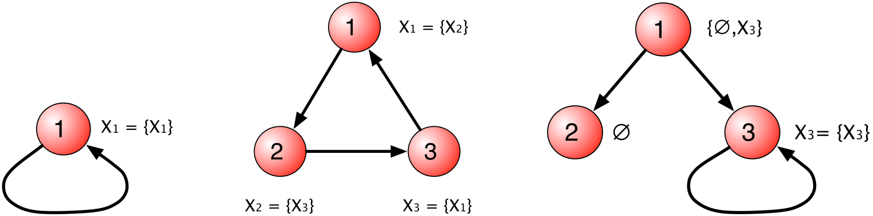
If the graph contains cycles, interpreting edges as membership implies that the set that decorates the graph is no longer well-founded.
Non well-founded sets are often referred to as *hypersets*.



Anti Foundation Axiom (AFA) states that every apg has a unique decoration.

Cyclic graphs and hypersets

If the graph contains cycles, interpreting edges as membership implies that the set that decorates the graph is no longer well-founded.
Non well-founded sets are often referred to as *hypersets*.



Anti Foundation Axiom (AFA) states that every apg has a unique decoration.
Two apgs denote the same hyperset if and only if their decoration is the same.

Applying extensionality axiom (E) for verifying equality would lead to a circular argument.

Bisimulation

Let $G_1 = \langle N_1, E_1 \rangle$ and $G_2 = \langle N_2, E_2 \rangle$ be two graphs, a *bisimulation* between G_1 and G_2 is a relation $b \subseteq N_1 \times N_2$ such that:

1. $u_1 \; b \; u_2 \wedge \langle u_1, v_1 \rangle \in E_1 \Rightarrow \exists v_2 \in N_2 (v_1 \; b \; v_2 \wedge \langle u_2, v_2 \rangle \in E_2)$
2. $u_1 \; b \; u_2 \wedge \langle u_2, v_2 \rangle \in E_2 \Rightarrow \exists v_1 \in N_1 (v_1 \; b \; v_2 \wedge \langle u_1, v_1 \rangle \in E_1)$.

In case G_1 and G_2 are apgs pointed in ν_1 and ν_2, respectively, it is also required that $\nu_1 \; b \; \nu_2$.

Bisimulation

Let $G_1 = \langle N_1, E_1 \rangle$ and $G_2 = \langle N_2, E_2 \rangle$ be two graphs, a *bisimulation* between G_1 and G_2 is a relation $b \subseteq N_1 \times N_2$ such that:

1. $u_1 \, b \, u_2 \wedge \langle u_1, v_1 \rangle \in E_1 \Rightarrow \exists v_2 \in N_2 (v_1 \, b \, v_2 \wedge \langle u_2, v_2 \rangle \in E_2)$
2. $u_1 \, b \, u_2 \wedge \langle u_2, v_2 \rangle \in E_2 \Rightarrow \exists v_1 \in N_1 (v_1 \, b \, v_2 \wedge \langle u_1, v_1 \rangle \in E_1)$.

In case G_1 and G_2 are apgs pointed in ν_1 and ν_2, respectively, it is also required that $\nu_1 \, b \, \nu_2$.
If there is a bisimulation between G_1 and G_2 then the two graphs are bisimilar.

Bisimulation

Let $G_1 = \langle N_1, E_1 \rangle$ and $G_2 = \langle N_2, E_2 \rangle$ be two graphs, a *bisimulation* between G_1 and G_2 is a relation $b \subseteq N_1 \times N_2$ such that:

1. $u_1 \, b \, u_2 \wedge \langle u_1, v_1 \rangle \in E_1 \Rightarrow \exists v_2 \in N_2 (v_1 \, b \, v_2 \wedge \langle u_2, v_2 \rangle \in E_2)$
2. $u_1 \, b \, u_2 \wedge \langle u_2, v_2 \rangle \in E_2 \Rightarrow \exists v_1 \in N_1 (v_1 \, b \, v_2 \wedge \langle u_1, v_1 \rangle \in E_1)$.

In case G_1 and G_2 are apgs pointed in ν_1 and ν_2, respectively, it is also required that $\nu_1 \, b \, \nu_2$.

If there is a bisimulation between G_1 and G_2 then the two graphs are bisimilar.

If they are bisimilar, they represent the same set (their point is decorated by the same set).

Bisimulation

A complexity summary

- If b is required to be a bijective function then it is a *graph isomorphism*.
- Establishing whether two graphs are isomorphic is an NP-problem neither proved to be NP-complete nor in P.
- Establishing whether G_1 is isomorphic to a subgraph of G_2 (subgraph isomorphism) is NP-complete.
- Establishing whether G_1 is bisimilar to a subgraph of G_2 (subgraph bisimulation) is NP-complete.

Bisimulation

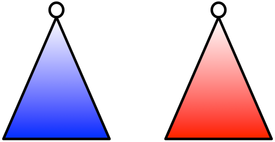
A complexity summary

- If b is required to be a bijective function then it is a *graph isomorphism*.
- Establishing whether two graphs are isomorphic is an NP-problem neither proved to be NP-complete nor in P.
- Establishing whether G_1 is isomorphic to a subgraph of G_2 (subgraph isomorphism) is NP-complete.
- Establishing whether G_1 is bisimilar to a subgraph of G_2 (subgraph bisimulation) is NP-complete.
- Instead, establishing whether G_1 is bisimilar to G_2 is in P: $O(|E_1 + E_2| \log |N_1 + N_2|)$.

Bisimulation

In case G_1 and G_2 are the same graph $G = \langle N, E \rangle$, a *bisimulation on* G is a bisimulation between G and G.

It is immediate to see that there is a bisimulation between two apg's $\langle G_1, \nu_1 \rangle$ and $\langle G_2, \nu_2 \rangle$ if and only if there is a bisimulation b on the graph $G = \langle \{\nu\} \cup N_1 \cup N_2, \{(\nu, \nu_1), (\nu, \nu_2)\} \cup E_1 \cup E_2 \rangle$ such that $\nu_1 \; b \; \nu_2$

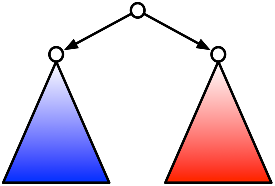


We can focus on the bisimulations on a single graph; we are interested in computing the *maximum bisimulation*: it is unique, it is an equivalence relation, and it contains all other bisimulations on G.

Bisimulation

In case G_1 and G_2 are the same graph $G = \langle N, E \rangle$, a *bisimulation on* G is a bisimulation between G and G.

It is immediate to see that there is a bisimulation between two apg's $\langle G_1, \nu_1 \rangle$ and $\langle G_2, \nu_2 \rangle$ if and only if there is a bisimulation b on the graph $G = \langle \{\nu\} \cup N_1 \cup N_2, \{(\nu, \nu_1), (\nu, \nu_2)\} \cup E_1 \cup E_2 \rangle$ such that $\nu_1 \; b \; \nu_2$

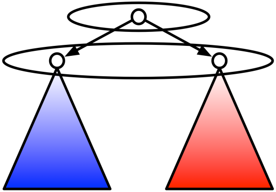


We can focus on the bisimulations on a single graph; we are interested in computing the *maximum bisimulation*: it is unique, it is an equivalence relation, and it contains all other bisimulations on G.

Bisimulation

In case G_1 and G_2 are the same graph $G = \langle N, E \rangle$, a *bisimulation on* G is a bisimulation between G and G.

It is immediate to see that there is a bisimulation between two apg's $\langle G_1, \nu_1 \rangle$ and $\langle G_2, \nu_2 \rangle$ if and only if there is a bisimulation b on the graph $G = \langle \{\nu\} \cup N_1 \cup N_2, \{(\nu, \nu_1), (\nu, \nu_2)\} \cup E_1 \cup E_2 \rangle$ such that $\nu_1 \; b \; \nu_2$



We can focus on the bisimulations on a single graph; we are interested in computing the *maximum bisimulation*: it is unique, it is an equivalence relation, and it contains all other bisimulations on G.

Bisimulation

Therefore, we might restrict our search to bisimulations on G that are *reflexive and symmetric relations* on N such that:

$$\begin{array}{c}\forall u_1, u_2, v_1 \in N \left(u_1 \; b \; u_2 \wedge \langle u_1, v_1\rangle \in E \Rightarrow\right. \\ \left.(\exists v_2 \in N)(v_1 \; b \; v_2 \wedge \langle u_2, v_2\rangle \in E)\right)\end{array} \tag{1}$$

The symmetric requirement makes the second case of the definition of bisimulation superfluous. We will use the following logical rewriting in some encodings:

$$\begin{array}{l}\neg\exists u_1, u_2, v_1 \in N \Big(u_1 \; b \; u_2 \wedge \langle u_1, v_1\rangle \in E \wedge \\ \qquad \neg((\exists v_2 \in N)\,(v_1 \; b \; v_2 \wedge \langle u_2, v_2\rangle \in E))\Big)\end{array} \tag{1'}$$

Bisimulation

Another characterization of the maximum bisimulation is based on the notion of *stability*. Given a set N, a partition P of N is a collection of non-empty disjoint sets (blocks) $B_1, B_2, \ldots$ such that $\bigcup_i B_i = N$. Let E be a relation on the set N, with E^{-1} we denote its inverse relation.
A partition P of N is said to be *stable* with respect to E if and only if

$$(\forall B_1 \in P)(\forall B_2 \in P)(B_1 \subseteq E^{-1}(B_2) \vee B_1 \cap E^{-1}(B_2) = \emptyset) \qquad (2)$$

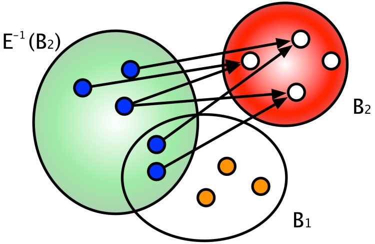
which is in turn equivalent to state that there do not exist two blocks $B_1 \in P$ and $B_2 \in P$ such that:

$$(\exists x \in B_1)(\exists y \in B_1)\,(x \in E^{-1}(B_2) \wedge y \notin E^{-1}(B_2)) \qquad (2')$$

Bisimulation

Maximum fixpoint

A class B_2 of P *splits* a class B_1 of P if B_1 is replaced in P by $B_1 \cap E^{-1}(B_2)$ and $B_1 \setminus E^{-1}(B_2)$ (both not empty)



Starting from the partition $P = \{N\}$, after at most $|N| - 1$ split operations a procedure halts determining the *coarsest stable partition (CSP)* w.r.t. E. The CSP "corresponds" to the maximum bisimulation.

Paige and Tarjan showed us the way for fast implementations (1987).

Encoding

apg's are represented by

- facts `node(1).` `node(2).` `node(3).` ... for nodes
- facts `edge(u,v).` where `u` and `v` are nodes, for edges
- node 1 is the point of the apg

`http://clp.dimi.uniud.it`

Prolog

$\forall u_1, u_2, v_1 \in N \left(u_1 \; b \; u_2 \wedge \langle u_1, v_1 \rangle \in E \Rightarrow (\exists v_2 \in N)(v_1 \; b \; v_2 \wedge \langle u_2, v_2 \rangle \in E)\right)$

$\forall \Rightarrow$ recursion on list. (Generate & Test; B is reflexive and symmetric)

```
bis(B) :-  bis(B,B).          % Recursively analyze B

bis([],_).
bis([ (U1,U2) |RB],B) :-      %%% if U1 bis U2
       successors(U1,SU1),    %%% Collect the successors SU1 of U1
       successors(U2,SU2),    %%% Collect the successors SU2 of U2
       allbis(SU1,SU2,B),     %%% Then recursively consider SU1
       bis(RB,B).

allbis([],_,_).
allbis([V1 | SU1],SU2,B) :-   %%% If V1 is a successor of U1
       member(V2,SU2),        %%% there is a V2 successor of U2
       member( (V1,V2),B),    %%% such that V1 bis V2
       allbis(SU1,SU2,B).

successors(X,SX) :- findall(Y,edge(X,Y),SX).
```

CLP(FD)

$$\forall u_1, u_2, v_1 \in N \left(u_1 \; b \; u_2 \wedge \langle u_1, v_1 \rangle \in E \Rightarrow (\exists v_2 \in N)(v_1 \; b \; v_2 \wedge \langle u_2, v_2 \rangle \in E)\right)$$

$\forall \Rightarrow$ recursion on list.

```
bis :- size(N), M is N*N,                    %%% Define the N * N Boolean
       length(B,M), domain(B,0,1),           %%% Matrix B
       constraint(B,N), Max #= sum(B),       %%% Max is the number of pairs
       labeling([maximize(Max),ffc,down],B).  %%% in the bisimulation

constraint(B,N) :- reflexivity(N,B), symmetry(1,2,N,B), morphism(N,B).

morphism(N,B) :-
       findall( (X,Y),edge(X,Y),EDGES),
       foreach( E in EDGES,  U2 in 1..N, morphismcheck(E,U2,N,B)).

morphismcheck( (U1,V1),U2,N,B) :-
       access(U1,U2,B,N,BU1U2),             % Flag BU1U2 stands for (U1 B U2)
       successors(U2, SuccU2),              % Collect all edges (U2,V2)
       collectlist(SuccU2,V1,N,B,BLIST),% BLIST contains all flags BV1V2
       BU1U2  #=< sum(BLIST).    % If (U1 B U2) there is V2 s.t. (V1 B V2)
```

ASP

$$\neg\exists u_1, u_2, v_1 \in N \left(u_1 \, b \, u_2 \wedge \langle u_1, v_1 \rangle \in E \wedge \neg((\exists v_2 \in N)\,(v_1 \, b \, v_2 \wedge \langle u_2, v_2 \rangle \in E)) \right)$$

$\forall = \neg\exists \Rightarrow$: ASP constraints

```
%% Reflexivity and Symmetry
bis(I,I)   :- node(I).
bis(I,J)   :- node(I;J), bis(J,I).
%%% Nondeterministic choice
{bis(I,J)} :- node(I;J).
%%% Morphism requirement (1')
:-  node(U1;U2;V1), bis(U1,U2), edge(U1,V1),  not one_son_bis(V1,U2).
one_son_bis(V1,U2) :-  node(V1;U2;V2), edge(U2,V2), bis(V1,V2).

%% Minimization  (max bisimulation)
non_rep_node(A) :- node(A), bis(A,B), B < A.
rep_node(A)     :- node(A),  not non_rep_node(A).
rep_nodes(N)    :- N=#sum[rep_node(A)].
#minimize [rep_nodes(N)=N].
```

co-LP

$$\forall u_1, u_2, v_1 \in N \left(u_1\ b\ u_2 \wedge \langle u_1, v_1 \rangle \in E \Rightarrow (\exists v_2 \in N)(v_1\ b\ v_2 \wedge \langle u_2, v_2 \rangle \in E)\right)$$

co-LP semantics is based on the greatest fixpoint
(for coinductive predicates)

```
bis(U,V) :-
            successors(U,SU),
            successors(V,SV),
            allbis(SU,SV),
            allbis(SV,SU).
allbis([],_ ).
allbis([U|R],SV ) :-
            member(V,SV),
            bis(U,V),
            allbis(R,SV).
```

`member` and `successors` are inductive.
No need of extra code for “maximization”

{log} and (main predicate of) Prolog

$(\forall B_1 \in P)(\forall B_2 \in P)(B_1 \subseteq E^{-1}(B_2) \vee B_1 \cap E^{-1}(B_2) = \emptyset)$

```
stable(P) :-
       forall(B1 in P, forall(B2 in P, stablecond(B1,B2) ) ).
stablecond(B1,B2) :-
       edgeinv(B2,InvB2) &
       (subset(B1,InvB2) or  disj(B1,InvB2)).
edgeinv(A,B) :-
       B = {X : exists(Y,(Y in A & edge(X,Y)))}.
```

```
stablecond(B1,B2) :-  edgeinv(B2,InvB2),
       (subseteq(B1,InvB2)  ;  emptyintersection(B1,InvB2)).
```

CLP(FD)

$(\forall B_1 \in P)(\forall B_2 \in P)(B_1 \subseteq E^{-1}(B_2) \vee B_1 \cap E^{-1}(B_2) = \emptyset)$

```
stability(B,N) :-
    foreach( I in 1..N, J in 1..N, stability_cond(I,J,B,N)).

stability_cond(I,J,B,N) :-         % Blocks BI and BJ are considered
    inclusion(1,N,I,J,B, Cincl), % Nodes  in 1..N are analyzed
    emptyintersection(1,N,I,J,B,Cempty), % Cincl and Cempty are reified
    Cincl + Cempty #> 0.           % OR condition

inclusion(X,N,_,_,_, 1) :- X>N,!.
inclusion(X,N,I,J,B, Cout) :-   % Node X is considered
    alledges(X,B,J,Flags),      % Flags stores existence of edge (X,Y) with
    LocFlag #=  ((B[X] #= I) #=> (Flags #> 0)), %% Inclusion check:
    X1 is X+1,                  % If X in BI then X in E-1(BJ)
    inclusion(X1,N,I,J,B,Ctemp), % Recursive call
    Cout #= Ctemp*LocFlag.      % AND condition (forall nodes it should hol

alledges(X,B,J,Flags) :-        % Collect the successors of X
    successors(X,OutgoingX),    % And use them for assigning the Flags var
    alledgesaux(OutgoingX,B,J,Flags).
alledgesaux([],_,_,0).
alledgesaux([Y|R],B,J,Flags) :- % The Flags variable is created
```

ASP

$(\exists x \in B_1)(\exists y \in B_1)\,(x \in E^{-1}(B_2) \wedge y \notin E^{-1}(B_2))$

```
blk(I)      :- node(I).
%%% Function assigning nodes to blocks
1{inblock(A,B):blk(B)}1 :- node(A).
%%% STABILITY (2')
:- blk(B1;B2), node(X;Y), X != Y, inblock(X,B1), inblock(Y,B1),
       connected(X,B2),   not connected(Y,B2).
connected(Y,B) :- edge(Y,Z),blk(B),inblock(Z,B).
%% Basic symmetry-breaking rules (optional)
:- node(A), internal(A), inblock(A,1).
internal(X) :- edge(X,Y).
leaf(X)      :-node(X), not internal(X).
non_empty_block(B) :- node(A), blk(B), inblock(A,B).
empty_block(B) :- blk(B), not non_empty_block(B).
:- blk(B1;B2), 1 < B1, B1 < B2, empty_block(B1), non_empty_block(B2).
%% Minimization
number_blocks(N)   :- N=#sum[non_empty_block(B)].
#minimize [number_blocks(N)=N].
```

```
stable_comp(Final, Nclasses) :-
       findall(X,node(X),Nodes),
       initialize(Nodes, Initial),
       maxfixpoint(Initial, 2, Final, Nclasses). % start with "2"
%%% maxfixpoint procedure. If possible, split, else stop.
maxfixpoint(AssIn, I, AssOut, C) :-
       split(I,AssIn,AssMid),!,
       I1 is I+1,
       maxfixpoint(AssMid, I1, AssOut, C).
%%% When stop, simply compute the number of classes used
maxfixpoint(Stable,C,Stable,C1) :-
       count_classes(C,Stable,C1).
%%% Split operation.
%%% First locate a block that can be split. Then find the splitter
split(MaxBlock,AssIn,AssMid) :-
        between(1,MaxBlock,I),
        findall(X,member(X-I,AssIn),BI),
        BI = [_, _ | _], %% BI might be split (not empty, not singleton)
        %%% Find potential splitters BJ (and remove duplicates)
        findall(Q,(member(V-Q,AssIn),edge(W,V),member(W,BI)),SP),
        sort(SP,SPS), member(J,SPS),
        findall(Z,(member(Y-J,AssIn),edge(Z,Y)),BJinv),
        my_delete(BI,BJinv,[D|ELTA]), %%% The difference is computed when
        MaxBlock1 is MaxBlock + 1,
        update(AssIn,AssMid,MaxBlock1,[D|ELTA]).
```

Benchmarks

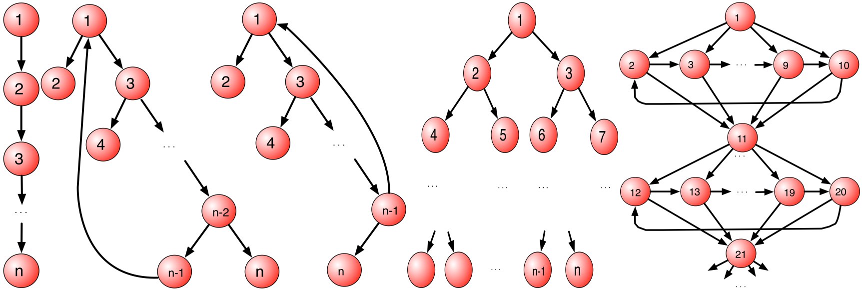
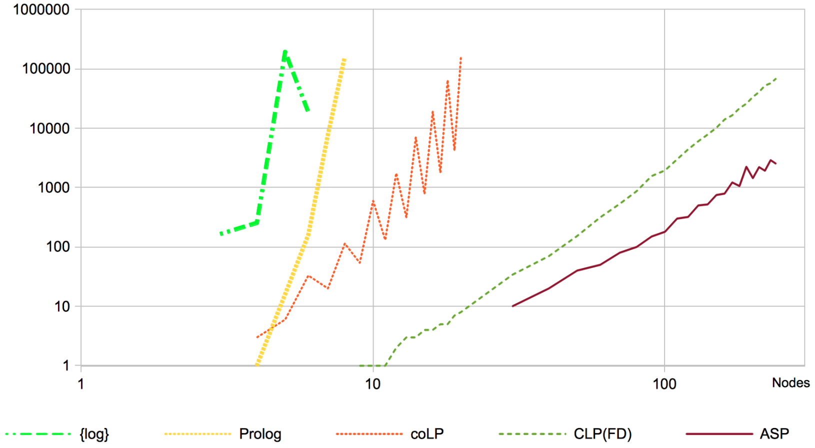


Figure : From left to right, the graphs G_1, G_2 (n odd), G_2 (n even), G_3, and G_5 used in the experiments. G_4 is the complete graph (not reported).

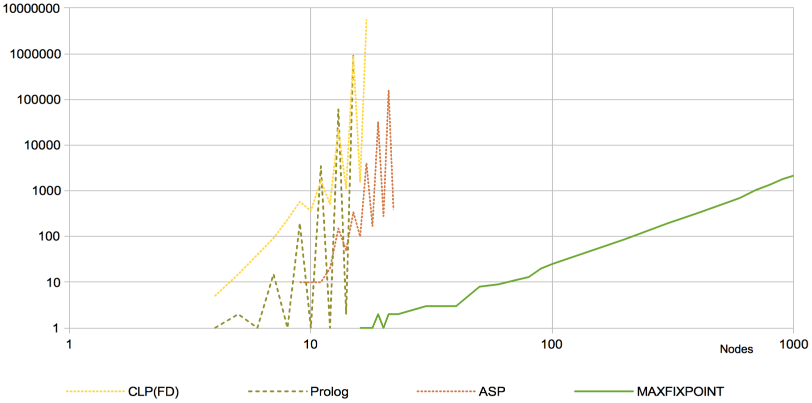
Summary of results

Direct encoding



Summary of results

Coarsest stable paritition



Conclusions

- Prolog generate & test is useless
- CLP constraint & generate introduces too many constraints for nested quantifiers
- ASP generate & test allows clear code and good running time
- These results can be inherited by the encoding of other (similar) graph properties

Conclusions

- Prolog generate & test is useless
- CLP constraint & generate introduces too many constraints for nested quantifiers
- ASP generate & test allows clear code and good running time
- These results can be inherited by the encoding of other (similar) graph properties
- Theoretical algorithmic results can be implemented in Prolog (with a great speed-up w.r.t. declarative approach)!