An introduction to logic program

Zhang Mingyi
Guizhou Academy of Sciences

Zhangmingyi045@aliyun.com

Outline

1. Why logic programming is interesting?

   As well known, when we deal with knowledge, in particular, it is needed to represent them by some suitable formal systems, and then to solve a problem or make a decision concerning them. Usually, we apply a declarative logic programming language, that is，so-called Logic Program (LP) and its extended form —— Answer Set Program（ASP）.

2. How are they defined

   For basic programs?

   For Extended Programs (with default negation or non-monotonic) ?

3. How can they be used for problem solving?

1. Why logic programming is interesting?

In this survey

- Provide a declarative programming language

- Logic programming——a method for representing declarative knowledge.

- Answer Set Logic Programming —— a form of declarative logic programming oriented towards different combinational research problems. It has been applied to:

– developing a decision support system of a Space shuttle;

– graph-theoretic problems arising in zoology and linguistics;

– Represent incomplete knowledge by introducing negation as failure and represent acceptable sets of believes in term of answer sets (also called stable models) .

(1) Using relational database to represent knowledge

- Example 1. Database DB_1

- We represent the parenthood relation among several members of Britain's royal family: the Queen, the prince and Princess of Wales, and their children, as a relational database:

Relation Mother

Parent	Child
Elizabeth	Charles
Diana	William
Diana	Harry

Relation Father

Parent	Child
Charles	William
Charles	Harry

(2) Using first-order formulas to represent knowledge

We represent the database DB_1 by the theory T_1

Axioms

- Domain closure assumption

$\forall x(x=\text{Elizabeth} \vee x=\text{Charles} \vee x=\text{Dianna} \vee x=\text{William} \vee x=\text{Harry})$ (1)

- Unique name assumption

Elizabeth $\neq$ Charles, Elizabeth $\neq$ Dianna, …, William $\neq$ Harry (2)

- Alternatively we can represent it as a first-order theory T_1:

$\forall xy[\text{Mother}(x,y) \equiv$
$(x=\text{Elizabeth} \wedge y=\text{Charles}) \vee (x=\text{Diana} \wedge y=\text{William}) \vee (x=\text{Diana} \wedge y=\text{Harry})]$
(3)

$\forall xy[\text{Father}(x,y) \equiv (x=\text{Charles} \wedge y=\text{william}) \vee (x=\text{Charles} \wedge y=\text{Harry})]$

I. Properties of T_1

- Problem 1. Prove that T_1 is consistent.
- Problem 2. Prove T_1 is complete.
- Problem 3. Prove that any atomic sentence in the language of T_1 other than an equality is provable in T_1 if and only if it belongs to DB_1.

Would it be true without axiom (1) in the axiom set of T_1 ? Without axiom (2)?

II. Extending T_1

- T_1 can be further extended by adding axioms defining some new predicates in terms of Mother and Father, for instance

  - $\forall xy(\text{Parent}(x,y) \equiv \text{Mother}(x,y) \lor \text{Father}(x,y))$ (4)
  - $\forall x(\text{Childless}(x) \equiv \neg \exists y(\text{Parent}(x,y))$ (5)
  - $\forall xy(\text{Grandparent}(x,y) \equiv \exists z(\text{Parent}(x,z) \land \text{Parent}(z,y))$ (6)
- We can also add the predicate Male and the corresponding axioms for extending T_1
  - $\forall xy(\text{Father}(x,y) \rightarrow \text{Male}(x))$;
  - $\forall xy(\text{Mother}(x,y) \rightarrow \neg \text{Male}(x)$ (7)

III. Is the extended T_1 also complete?

- Note that the theory obtained at this stage is incomplete, since it only give a sufficient condition and a necessary condition for the definition of the predicate “Male” respectively. For instance we can not decide which of Male(William) and $\neg$Male(William) is true, i.e.

- Problem 4. Show that the sentence Male(William) and Male(Harry) are neither provable nor refutable from the axioms introduced above.

2. How are logic programs defined?

(1) Logic Program (without default negation or monotonic)

A logic program: a set of rules with the form

Head $\leftarrow$ Body,

where "$\leftarrow$" means "if".

Head: consequence of a rule.

Bady: antecedent of a rule.

- If the body is empty then $\leftarrow$ can be dropped and the rule is called a *fact*.
- Fact : Any first-order formula can be reduced to an equivalent CNF (Conjunctive Normal Form) and to DNF (Disjunctive Normal Form)

I. *Basic Program*

Syntax

Firstly we have a nonempty set **A** of symbols, called atoms, which are also called **positive literals**. A **negative literal** is an atom preceded the classical negation symbol $\neg$**.**

- A literal is a possible or negative literal. The literals A and $\neg$A are said to be complementary. The set of all literals will be denoted by $\mathrm{Lit}_{\mathbf{A}}$ (simply Lit).

- A set of literals is **inconsistent** if it contains a complementary pair, and **consistent** otherwise.

What is a basic program?

- Basic rule: $L_0 \leftarrow L_1, \ldots, L_k$ " ,"means logical "and"
- Basic program: a set of basic rules.

How to represent declarative knowledge by logic programs?

Example 1 (cont.)

Define "Mother" and "Father" using facts:

Farther(Charles, William)
Father(Charles, Harry)
Mother(Elizabeth, Charles)
Mother(Diana, William)
Mother(Diana, Harry).

How to define derivative notions

Define new notions using rules:

$Parent(x,y) \leftarrow Mather(x,y)$

$Parent(x,y) \leftarrow Father(x,y)$

$\neg Childless(x) \leftarrow Parent(x,y)$

$Grandparent(x,y) \leftarrow Parent(x,z), Parent(z,y)$

$Male(x) \leftarrow Father(x,y)$

$\neg\ Male(x) \leftarrow Mother(x,y)$

An example of recursive definition:

$Ancestor(x,y) \leftarrow Parent(x,y),$

$Ancestor(x,y) \leftarrow Ancestor(x,z), Ancestor(z,y)$

Problem 5. Is It always possible to derive a negative fact, e.g. $\neg$Ancestor(Elizabeth, Elizabeth), from the previous axioms and rules?

What is derivable from a program?

Prolog (stand for Programming in Logic)----the name of a family of logic programming systems, which express a body of knowledge as a logic program such that they can be sometimes used to answer queries on the basis of this knowledge.

Note that a rule with variables are viewed as a "schemata" that represent their ground instances. So, we can only consider rules without variables.

What are derived from a logic program?

*The notion of a **consequence** (member of an answer set)*

***---**using (part of) **answer sets** to represent **solutions** of a problem*

Consequence operator

- Let X be a set of literals. We say that X is *logically closed* if it is c*onsistent* or equal to Lit. X is *closed* under a basic program Π if Head$\in$X whenever Body$\subseteq$X for every rule Head$\leftarrow$ Body in Π.

- The notion of a **consequence operator**

By Cn(Π) we denote the smallest set of literals which is both logically closed and closed under Π. The elements of Cn(Π) are called the consequences and Cn is called a consequence operator. Π is consistent if Cn(Π) is consistent, and inconsistent otherwise.

- X is *supported* by Π if, for each liter L$\in$X, there is a rule Head$\leftarrow$Body in Π such that Head=L and Body $\subseteq$X (intuitively, it provides a “reason” for including L in X).

Properties of a consequence operator

- Proposition 1. For any basic program Π, $Cn(\Pi)$ always exists.
- Proposition 2. If Π is consistent, then $Cn(\Pi)$ is the smallest set of literals closed under Π; if Π is inconsistent, then $Cn(\Pi)=Lit$
- Proposition 3. For any consistent basic program Π, $Cn(\Pi)$ is supported by Π

- Proposition 4. If $\Pi_1 \subseteq \Pi_2$ then $Cn(\Pi_1) \subseteq Cn(\Pi_2)$;

furthermore $Cn(\Pi) = \cup_{n \geq 0} T_\Pi{}^n(\emptyset)$, where

$$T_\Pi(X) = \begin{cases} \{Head \mid Head \leftarrow Body \in \Pi,\ Body \subseteq X\}, & \text{if } X \text{ is consistent} \\ Lit, & \text{otherwise} \end{cases}$$

The above proposition gives the process of bottom-up evaluation for the set of consequences.

- Corollary If a basic program Π is consistent then each consequence of Π is a head literal of Π.

Bottom-up evaluation

Example 2 Let **A=**{p,q,r,s} and Π contains

$$\left\{\begin{array}{l} p, \\ \neg q, \\ r \leftarrow p, q\ , \\ \neg r \leftarrow p, \neg q\ , \\ s \leftarrow r\ \ , \\ s \leftarrow p, s\ \ , \\ \neg s \leftarrow p, \neg q, \neg r \end{array}\right.$$

Then

$T_\Pi{}^0\ (\varnothing)=\varnothing$

$T_\Pi{}^1\ (\varnothing)=\{p,\ \neg q\}$

$T_\Pi{}^2\ (\varnothing)=\{p,\ \neg q,\ \neg r\}$

$T_\Pi{}^3(\varnothing)=\{p, \neg q,\ \neg r,\ \neg s\}$

and Cn(Π)={p,¬q, ¬r, ¬s}.

For Example 1?

How to determine whether a literal is a solution of a given problem?

- **Resolution in the propositional logic**
- **Idea**: to determine whether a literal is a consequence of Π it is sufficient to find out a rule (all rules) with head L and show all literals of its body are consequences of Π.

- Iteratively apply this method until all (some) literals to be shown are in Head(Π) (in Body(Π)\ (Head(Π)), where Head(Π) is the set of all head literals of Π and Body(Π) is the set of all body literals of Π, Body(Π)\ (Head(Π)) is the difference between Body (Π) and (Head(Π)), .

SLD calculus

- **Method:**

The *SLD* (Selection-Linear-Definite) calculus----

Let G (goal) be a set of finite literals and L a literal. Rule (S) and Rule(F) represent rules for proving success and failure respectively, where S---Success and F---Failure .

Axiom $\models \varnothing$

Rule (S) $\dfrac{\models G \cup B}{\models G \cup \{L\}}$ if $B \in$ Bodies(L), which is the set of bodies of all rules with head L

The sign $\models$ expresses "success"

Rule (F) $\qquad \dfrac{\Dashv G \cup B}{\Dashv G \cup \{L\}}$ $\quad$ if all $B \in \text{Bodies}(L)$

The sign $\Dashv$ expresses “failure”

Note that the number of premises equals to the cardinality of Bodies(L) (in particular,it can be zero).

If $\models G$ **i**s derivable (obtained by finite application of Rule(S)) in the SLD calculus for Π, then we say G **succeeds** relative to Π.

If $\Dashv G$ then we say G **fails** relative to Π.

Is the SDL calculus is sound and complete?

Proposition 5. For a basic program Π, no goal both succeeds and fails relative to Π.

Proposition 6 （Soundness） For any basic program Π and any literal L

1. if {L} succeeds relative to Π then L is a consequence of Π.
2. If Π is consistent and {L} fails relative to Π then L is not a consequence of Π.

Proposition 7 (Completeness) For any basic program Π and any consequence L of Π, {L} succeeds relative to Π.

- Example 2 (cont.) $\neg$r succeeds relative to Π and r fails relative to Π since

- $\models \varnothing$
-
- $\models \{p\}$ $\qquad\qquad$ $\Dashv \{q\}$
-
- $\models \{p, \neg q\}$ $\qquad\qquad$ $\Dashv \{p, q\}$
-
- $\models \{\neg r\}$ $\qquad\qquad$ $\Dashv \{r\}$

Restrictive SDL resolution

Note that (1) the SDL calculus may be unsound if the program fails. For instance, if Π is inconsistent and L is not a head literal of Π then L is a consequence of Π that fails: $\dashv$ {L} can be derived by one application of Rule(F) to the empty set of premises; (2) the failure rule is generally incomplete even for consistent program. For instance, {p} does not fail relative to {p$\leftarrow$p}.

For a special class of basic programs we have a solution strategy---the restrictive SDL resolution (Breadth-First).

II. Definite Program

Definite Rule or **Program**---does not contain the negative symbol $\neg$.

Fact: Any basic program can be reduced to a definite program in the way:

Let $\mathbf{A}' = \{A' \mid A \in \mathbf{A}\}$, for any $A \in \mathbf{A}$, Norm(A)=A, Norm($\neg$A)=A′,

Norm(X)={Norm(L)$\mid$ L$\in$X} (X is a set of literals on **A**)
Norm(Head$\leftarrow$Body)=Norrm(Head)$\leftarrow$Norm(Body)
Norm(Π)={Norm(R)$\mid$ R$\in\Pi$}.

For a set of atoms (goal) $\{A_1, \ldots, A_n\}$ and a definite rule
$B \leftarrow C_1,\ldots,C_m$ if $B=A_i$, then we have a resolvent (new goal)
$\{A_1, \ldots, A_{i-1}, C_1,\ldots,C_m, A_{i+1},\ldots,A_n\}$-----*SLD Resolution rule.*

- We call A the atom resolved upon. Form Propositions 6 and 7 it is easy to get the following result:

- Proposition 8 (soundness and completeness) For any set of atoms (goal) $\{A_1, \ldots, A_n\}$ and any definite program $\prod$, $\{A_1, \ldots, A_n\} \subseteq Cn(\Pi)$ iff there is a infer process, which products the empty set of atoms (goal) by applying (finite times) SLD resolution starting from $\{A_1, \ldots, A_n\}$.

- *Restrictive SLD resolution*---- For every solution we always choice the leftmost atom in a goal $\{A_1, \ldots, A_n\}$ as the atom resolved upon and resolve it with a rule in $\prod$ according to the order on Π. If it can not resolve with any rule, then choice the next atom as the atom resolve upon. This is the linear solution.

Example 3. Consider program Π

$p(a,b) \leftarrow q(a, b), p(b,b),$

$p(b,b) \leftarrow q(b,a), p(a,b),$

$q(a,b),$

$p(a,a),$

$p(b,b)\}$

and goal is $p(a,b)$.

We have a restrictive SLD resolution (Breadth-First)

$p(a,b)$ $p(a,b) \leftarrow q(a,b), p(b,b)$

$q(a,b), p(b,b)$ $q(a,b)$

$p(b,b)$ $p(b,b)$

$\square$ (empty goal)

- Similarly we have Depth-First SLD resolution which always choice the rightmost atom in a goal as the atom resolved upon. Unfortunately it is not complete.

Problem 6 Show that the Depth-first SLD resolution is incomplete for Example 3.

Negation as failure

(2) Extended Program
(with default negation or non-monotonic)

The symbol "not"----Negation as failure ("not" means "is not believed")
or Default negation
Why to introduce "not"?
In the notation of a logic programming, the predicate of Mother seems to be

Mother(Elizabeth, Charles)
Mother(Diana, William)
Mother(Diana, Harry)

But we can not determine whether Diana is mother of Elizabeth.
How to do this in a relational database?

CWA

- **Closed World Assumption** (CWA): roughly, if an atom A couldn't be derived from a database, we can conclude $\neg$A. So, we should conclude $\neg$Mother(Diana, Elizabeth) by applying CWA. This can be done by adding a rule with "not" (default negation or negation as failure) to the above program.

Mother(Elizabeth, Charles)
Mother(Diana, William)
Mother(Diana, Harry)
$\neg$Mother(x,y)$\leftarrow$ not Mother(x,y).

Non-monotonicity

- The final rule tell us that, for any individual x and y under consideration, we can conclude $\neg$Mother(x,y) if the program gives no evidence that Mother(x,y). It expresses the CWA for the predicate Mother. It allows us to infer $\neg$Mothe(Diana,Elizabeth).

- Note that a rule with not in its body makes a program "nonmonotonic " (i.e. adding new facts invalidates a conclusion that could be obtained earlier.)

- For instance, after deleting the rule Mather(Diana, Harry), we can get a conclusion $\neg$Mother(Diana,Elizabeth). If we now put it back in, then this conclusion will be retracted. Usually, we call such a conclusion a **belief**.

What does mean “not”?

- Adding “not” makes us to completely define a notion when information is incomplete. Intuitively, the presence of not L in the body of a rule limits the applicability of the rule to the case when the program as a whole provides no possibility for deriving L.

Syntax

- A rule element is a literal possibly preceded by the negation as failure symbol *not*. A rule is an ordered pair Head$\leftarrow$ Body, whose first member Head is a literal and whose second member Body is a finite set of rule elements. For any set X of literals we will denote the set {not L$\mid$ L$\in$X} by not (X). Then any rule can be represented as

$$\text{Head} \leftarrow \text{Pos} \cup \text{not(Neg)}$$

Where Pos and Neg are sets of literals.

How to define an Extended Program

- for some finite sets of literals Pos, Neg, the rule with the head L_0 and the body $\{L_1,\ldots,L_m$, not L_{m+1}, …,not $L_n\}$ will be also written as

$$L_0 \leftarrow L_1,\ldots,L_m, \text{not } L_{m+1}, \ldots \text{not } L_n$$

An extended program is a set of rules. For instance

p,
q←p, not r (*)
q←r, not p
r←p, not s

What are derivable ?

is a program. This program does not contain the classical negation symbol $\neg$; the syntax of rules allows us to insert this symbol in front of the atoms p, q, r, s anywhere in the program.

What are derived from an extended program?

- ***Ansvwer set***

We would like to generalize the definition of Cn(Π) from Section 1 to arbitrary programs.

Difference between basic and extended rules

- Intuitively, the presence of a rule element not L in the body of a rule limits the applicability of the rule to the case when the program as a whole provides no possibilities for deriving L. For instance, rules in (*) differ from the basic rules

$$
\begin{array}{ll}
p, & \\
q \leftarrow p & \quad (**) \\
q \leftarrow r & \\
r \leftarrow p &
\end{array}
$$

in that

- the second rule allows us to derive q from p only if r can not be derived

- the third rule allows us to derive q from r only if p can not be derived

- the last rule allows us to derive r from p only if s can not be derived

What does “not” block?

This informal description of how the symbol not blocks the application of program rules, is circular, because it characterizes the process of applying rules in terms of what can be derived using these rules.

Nevertheless, for any set X of literals that description makes it possible to test the claim that rules allow us to derive the elements of X and nothing else.

Reducing extended programs into basic ones

- Take, for instance, X to be {p,r}.If p and r are indeed derivable and the other literals are not， then the second rule of (*) is blocked in view of the presence of not r in its body and the third rule is blocked by the presence of not p， the other two rules are no blocked. Then the effect of (*) rules is the same as the effect of

    p,

    $r \leftarrow p$ (***)

----the subset of (**) obtained by deleting its second and third rules.

This is a basic program. The set of its consequences is {p, r}, which is exactly the set X that we initially assumed to be the set of derivable literals. This fact confirms that {p, r} was a good guess.

- Generally, there can be several good guesses about the result of application of a given set of rules.

Guesses

- Consider, for instance, the program

  $p \leftarrow \text{not } q$

  $q \leftarrow \text{not } p$ (I)

  $r \leftarrow p$

  $r \leftarrow q$

- There are two reasonable conjectures about what can be derived using these rules. One is that we can derive p and r but not q .If so then (I) has the same meaning as the basic program p

r←p (II)

r←q

The set of consequences of this program is indeed {p r}.

The other possibility is that q and r can be derived but not p. In this case (I) has the same meaning as the basic program

q

r←p (III)

r←q

whose consequences are indeed q and r.

- This example leads us to the view that negation as failure can make the rules of a program non deterministic. There can be several correct ways to organize the process of deriving literals using the rules of a program that contains negation as failure. Each of them produces a different set of literals; these sets will be called the answer sets for the program.

- A consequence of a program is a literal that is guaranteed to be produced no matter which derivation process is selected—a literal that belongs to all answer sets. For instance, the only answer set for (*) program is {p,r} so that the consequences of this program are p and r; the answer sets for (I) are {p,r} and {q,r}, so that its only consequence is {r}.

Reduction

- In order to give the definition of an answer set we need a general description of the process of reducing an arbitrary program to a basic program that was used above to obtain (***) from (*), and (II), (III) from (I).

Answer set

- Let Π be a program and X a set of literals. The reduct of Π relative to X is the basic program obtained from Π by

  - deleting each rule $Head \leftarrow Pos \cup$ not Neg such that $Neg \cap X \neq \emptyset$ and
  - replacing each remaining rule $Head \leftarrow Pos \cup$ not Neg by $Head \leftarrow Pos$.

- This program will be denoted by Π^X . We say that X is **an answer set** for Π if $Cn(\Pi^X)=X$.

Do answer sets exist?

- It is clear that every answer set is logically closed. We have seen that a program can have one or several answer sets .Some programs have no answer sets for instance,

  $p \leftarrow not\ p$

- A *consequence* of a program is a literal that belongs to *all its answer sets*. Alternatively, the consequences of a program can be characterized as the literals that belong to all its consistent answer sets. It is clear that the set of consequences is logically closed.

Consequence operator

- If a program Π is basic then its reduct relative to any set of literals is Π. It follows that the only answer set for a basic program Π is $Cn(\Pi)$ so that the new definition of a consequence applied to a basic program is equivalent to the one given in Section 1.

- For the set of consequences of a program we will use the same notation Cn as in the basic case.

Nonmonotonicity of Cn

- On programs with negation as failure, the consequence operator is nonmonotone.
- For instance, the set of consequences of {p←not q} is {p}; if we add q to this program as another rule, the set of consequences will be {q}.
- In this sense logic programming with negation as failure is a nonmonotonic formalism.

- Problem 7 Find all answer sets for the program

  $p \leftarrow not\ q$

  $q \leftarrow not\ p$

  $r \leftarrow not\ r$

  $r \leftarrow p$

- Problem 8 Find all answer sets for the program

$$p_{n+1} \leftarrow not\ p_n \quad (n \geq 0)$$

- Problem 9 Find all answer sets for the program

$$p_n \leftarrow not\ p_{n+1} \quad (n \geq 0)$$

Properties of Cn

- Proposition 8 If X and Y are answer sets for a program Π and $X \subseteq Y$ then $X=Y$.

- Corollary 1 Every program Π satisfies exactly one of the following conditions:
  - has no answer sets,
  - the only answer set for Π is Lit,
  - has an answer set and all its answer sets are consistent.

Consistency and Closure

- The consistency of a program is defined as it was defined for basic programs. A program is consistent if the set of its consequences is consistent and inconsistent otherwise. In the first two cases listed in the statement of the corollary 1 is inconsistent and $Cn(\Pi)=Lit$. In the third case is consistent.

- The definition of closure under a program given in Section is extended to arbitrary programs as follows. A set X of literals is closed under a program Π if for every rule $Head \leftarrow Pos \cup not\ Neg$ in Π, $Head \in X$ whenever $Pos \subseteq X$ and $Neg \cap X = \emptyset$.

Properties of answer sets

Proposition 9 Every answer set for a program Π is closed under Π.

- **The set of consequences of Π however is not necessarily closed under Π. This can be illustrated by program (I). Note that the set of consequences is the meet of all answer sets.**
- **We say that a set X of literals is supported by Π if for each literal $L \in X$ there exists a rule $Head \leftarrow Pos \cup not\ Neg$ in Π such that $Head=L$, $Pos \subseteq X$, $Neg \cap X = \emptyset$.**

- Proposition 3 can be generalized to arbitrary programs in the following way:

Proposition 10 Any consistent answer set for a program Π is supported by Π.

- As in the case of basic programs a head literal of a program Π is the head of a rule of Π.

Corollary 2 Any element of any consistent answer set for a program Π is a head literal of Π.

Corollary 3 If a program Π is consistent then every consequence of is a head literal of Π.

- As in the case of basic programs a program is head consistent if the set of its head literals is consistent. It is easy to see that any answer set of a head-consistent basic program is consistent. This can be generalized to arbitrary programs as follows:

- Proposition 11 If a program Π is head consistent then every answer set for Π is consistent.

- This proposition tells us that a head consistent program can not belong to the second of the three groups listed in the corollary 1 to Proposition 8.

- It is possible, however, that a head-consistent program belongs to the first group so that such a program can be inconsistent.

For instance, $p \leftarrow not\ p$ is a head consistent program without answer sets. An additional condition needed to guarantee the existence of an answer set, called “order-consistency” is discussed in future.

3. Application to Problem Solution

- The art of answer set programming is based on the possibility of representing the collection of sets that we are interested in as the collection of stable models of a formula.

- This is often achieved by conjoining a choice formula, which provides an approximation from above for the collection of sets that we want to describe, with formulas of a special syntactic form, called **constrains**, that eliminate the unsuitable stable models.

(1) Graph coloring

- Method---answer set programming
  - represent problem such that solutions are (part of) answer sets
  - Commonly used method: generate and test

(1) Graph coloring

Point coloring of a graph G----an assignment of colors to the points of G so that no two adjacent points have the same color (or a function f from its set of vertices to {1,…,n} such that $f(x)\neq f(y)$ for any adjacent vertices x and y).

Some notions for graph coloring

- n-coloring ---- n colors are used
- Chromatic number $\chi(G)$ ---- the minimal n for which G has an n-coloring
- n-chromatic ---- $\chi(G)=n$
- n-colorable ----- $\chi(G) \le n$
- Density $\omega(G)$ of G ---the number of points in a maximum clique (complete subgraph) of G.

Theorem For any graph G, $\chi(G) \ge \omega(G)$.

- For instance, for $G=C_{2n+1}$ ($n \ge 2$), $\chi(G) > \omega(G)$; for $G=K_p$ (p-complete graph), $\chi(G)= \omega(G)$.

- Example *Description of graph*:

  node(v_1), … , node(v_n), edge(v_i,v_j)…

- *Generate* (r—red, b-blue, g-green)

  col(X,r)$\leftarrow$node(X), not col(X,b), not col(X,g)

  col(X,b)$\leftarrow$node(X),not col(X,r),not col(X,g)

  col(X,g)$\leftarrow$node(X), not col(X,r), not col(X,b)

- *Test*

  $\leftarrow$edge(X,Y), col(X,Z), col(Y,Z)

Answer sets contain solution to problem!

I. G contains only one node A. We have

node(A)

col(A,r)←node(A), not col(A,b), not col(A,g)

col(A,b)←node(A),not col(A,r),not col(A,g)

col(A,g)←node(A), not col(A,r), not col(A,b).

- Clearly it is not needed to testing whether constrains are satisfied since there is no any constrain if we do not allow a pseudo-circle. So, it has 3 answer set { col(A,r)}, { col(A,b)} and { col(A,b)}.

II. G contains just two nodes A and B.

Case 1. G has no edges. It is similar to I.

Case 2. G has one edge. We have

node(A)
node(B)
edge(A,B)
col(A,r)←node(A), not col(A,b), not col(A,g)
col(A,b)←node(A),not col(A,r),not col(A,g)

col(A,g)←node(A), not col(A,r), not col(A,b).
col(B,r)←node(B), not col(B,b), not col(B,g)
col(B,b)←node(B),not col(B,r),not col(B,g)
col(B,g)←node(A), not col(A,r), not col(A,b).
←edge(A,B), col(A,Z), col(B,Z)   (Z∈{g, r, b}

It is easy to find out 6 answer sets:

{col(A,g), col(B,r)}, {col(A,g), col(B,b)}, {col(A,r), col(B,g)}
{col(A,r), col(B,b)}, {col(A,b), col(B,g)} {col(A,b), col(B,r)}

- III. G is a 4-complete graph (K4). We have

  node(X) $\quad (X \in \{A, B, C, D\})$
  edge(X,Y) $\quad (X, Y \in \{A, B, C, D\},\ X \neq Y)$
  col(X,r) $\leftarrow$ node(X), not col(X,b), not col(X,g)
  col(X,b) $\leftarrow$ node(X),not col(X,r),not col(X,g) $\quad (X \in \{A, B, C, D\})$
  col(X,g) $\leftarrow$ node(X), not col(X,r), not col(X,b)
  $\leftarrow$ edge(X,Y), col(X,Z), col(Y,Z)
  $\quad (X, Y \in \{A, B, C, D\},\ X \neq Y,\ Z \in \{g, r, b\})$

- It is easy to verify that there is no answer set since any two nodes in G are adjacent. So, the nodes in G should be colored by different colors, i.e. the minimal number of colors for coloring of G is 4.

(2) The block word

- The block world consists of several blocks 1,…,n, placed on the table such that they form a tower or several towers. For instance, if n=2, then the blocks words can be in 3 states:

```
 1          2
 2          1              1    2
 -------    ---------      ------------
```

- Predicate on(x,y)---$x \in \{1,\ldots,n\}$, $y \in \{1,\ldots,n, \text{table}\}$, $x \neq y$

- Generate

  $1 \leq \{on(x,y) \mid y \in \{1,\ldots,n, table\} - \{x\}\}c \leq 1 \quad (1 \leq x \leq n)$

  ---choice rule (allowing us to choice arbitrarily, for each block x, a unique position

  $s(x) \leftarrow on\ (x, table) \quad (1 \leq x \leq n)$

  $s(x) \leftarrow s(y), on\ (x, table) \quad (1 \leq x, y \leq n: x \neq y)$

  ----- s is the auxiliary recursively defined predicate. s(x) expresses x is supported by the table, tjat is to say, belongs to a tower of blocks that rests on the table.

- Test

  $\leftarrow 2 \leq \{on(x,y) \mid x \in \{1,\ldots,n\} - \{y\}\} \quad (1 \leq y \leq n)$

  ----no allow two blocks on the same block

  $\leftarrow not\ s(x) \quad (1 \leq x \leq n)$

  --- no allow circular configurations “floating in space” (e.g. on(1,2) and on (2,1)

Cardinality expressions

- Horn Formulae
- A horn formula is a conjunction of several (0 or more) implications of the form $F \rightarrow A$, where F is a conjunction of several (0 or more) atoms and A is an atom.
- **Proposition 1** For a Horn formula F, the minimal model of F is the only stable model of F.

Notations

- Cardinality expressions
- For any nonnegative integer l (lower bound) and u(upper bound) and formulas F1,...Fn,

I. $k \leq \{F_1,\ldots,F_n\}$ stands for the disjunction

$$\vee_{I \subseteq \{1,\ldots,n\},\ |I| = k} \wedge_{i \in I} F_i$$

For instance, $2 \leq \{F_1,F_2,F_3\}$ stands for

$(F_1 \wedge F_2) \vee (F_1 \wedge F_3) \vee (F_2 \wedge F_3)$,

II. $\{F_1,\ldots,F_n\} \leq p$ stands for

$\neg(p \leq \{F_1,\ldots,F_n\})$.

III. $k \leq \{F_1,\ldots,F_n\} \leq p$ stands for

$k \leq \{F_1,\ldots,F_n\} \wedge \{F_1,\ldots,F_n\} \leq p$.

- **Proposition 2** Any set of atoms
- - satisfies (I) iff it satisfies at least l of the formulas F1,...Fn ;
- - satisfies (II) iff it satisfies at most u of the formulas F1,...Fn ;

Choice formula

IV. For nay finite set Z of atoms, Z^c (choice formula) stands for

$$\wedge_{A \in Z} (A \vee \neg A)$$

Proposition 3 For any finite set Z, a set of atoms X is an answer set of Z^c iff $X \subseteq Z$.

- For instance, $\{p.q\}^c$ is $(p\vee\neg p) \wedge(q\vee\neg q)$. and $\{p.q\}^c$ has 4 answer Sets, each one of which is a subset of $\{p,q\}$.
-