BDD - Cours 9

Introduction à Datalog

Celine Kuttler

24 octobre 2017

Histoire

Datalog est une machine pour construire des nouveaux faits.
Autrement dit : une base de données déductive.

- 1977s : invention de Prolog, dans le contexte des bases de données. Idée : ajouter du calcul récursif aux requêtes relationnelles.
- 1980s : programmation logique populaire pour l'intelligence artificielle. présence industrielle forte, mais pas encore de killer app pour les requêtes récursives.
- 1990s : niches...
- depuis environ 2007 : renaissance de Datalog pour le web.
- nous travaillerons avec DES (`des.sourceforge.net`) de Fernando Saenz-Perez
- puis en fin de semestre nous travaillerons avec un outil en Prolog pour la normalisation des BDDs (support pédagogique du livre d'Elmasri et Navathe).

Hypothèse du monde clos

Un fait est considéré faux s'il n'est ni inclus dans la base de données des faits, ni démontrable en temps fini. On suppose qu'il n'y a pas de monde extérieur qui pourrait contenir des éléments inconnus au programme.
Cette *hypothèse du monde clos* motive des contraintes syntaxiques, qui s'appellent des règles de sécurité.

Contenu

Exemple 1 : rien que des faits

```
fete.
femme(mia).
femme(jody).
femme(yolanda).
happy(yolanda).
joueAirGuitar(jody).
```

Requêtes :

- tuple trouvé ou non.
- requête avec variables : présence de femmes ? noms des femmes ?
- notion de but : toute chose qui peut être prouvée

Résumé de la syntaxe (1)

- constantes
  - nombres (entiers et réels)
  - séquences de caractères alphanumériques, _ inclus, qui commencent avec une minuscule
- prédicats $p(a_1, a_2, \ldots, a_n)$. Le prédicat p prend n arguments, qui sont des variables ou constantes. Le nom du prédicat doit commencer avec une minuscule.
- variables
  - X, Y (séquences qui commencent avec une majuscule),
  - _ (variable anonyme, tiret bas)

Comment construire des nouveaux faits ?

Modus Ponens

Du fait A, en combinaison avec la règle $A \Rightarrow B$, on déduit B.

Une base de connaissances contient :

- des faits et
- des règles.

Exemple 1 : la fête

- Fait : C'est la fête.
  F
- Règle : Quand c'est la fête, il y a de la musique.
  $F \Rightarrow M$
- Conclusion : Il y a de la musique !
  M

Une première règle en Datalog

```
fete.
musique :- fete.
```

Comment lire une règle Datalog ?

En notation de logique propositionnelle, la règle Datalog

$$B :- A.$$

se lit comme implication dans l'autre direction,

$$A \Rightarrow B$$

Exemple 2 : règles et leurs interprétations

```
ecoute2LaMusique(yolanda) :- happy(yolanda).
joueAirGuitar(mia) :- ecoute2LaMusique(mia).
joueAirGuitar(yolanda) :-
                        ecoute2LaMusique(yolanda).
```

Interprétation

- Quand Yoland est contente, elle écoute de la musique.
- Quand Mia écoute de la musique, elle joue de l'AirGuitar.
- Quand Yolanda écoute de la musique, elle joue de l'AirGuitar.

Exemple 2 : Requêtes, concernant des faits déduits

```
fete.femme(mia).  ecoute2LaMusique(mia).
femme(yolanda).happy(yolanda).

ecoute2LaMusique(yolanda) :- happy(yolanda).
joueAirGuitar(mia) :- ecoute2LaMusique(mia).
joueAirGuitar(yolanda) :-
                          ecoute2LaMusique(yolanda).
```

Requêtes

- Est-ce que Mia joue de l'AirGuitar ?
- Est-ce que Yolanda joue de l'AirGuitar ?
- Qui joue de l'AirGuitar ?

Règles avec plusieurs conditions

En notation de logique propositionnelle, la règle Datalog

$$Z :- A1, A2, ..., An.$$

se lit comme implication dans la direction inverse,

$$A1 \text{ and } A2 \text{ and } ... \text{ and } An \Longrightarrow Z$$

- dans la tête d'une règle, toujours un seul fait
- le corps de la règle est une conjonction de n faits
- si toutes les conditions sont satisfaites, on peut déduire le nouveau fait en tête, Z
- on appelle les A_i des buts. Chaque chose qui est a prouver en Datalog est un but.

Exemple 3 : Conjonction logique (et)

```
happy(vincent).
ecoute2LaMusique(paul).

joueAirGuitar(vincent) :-
        ecoute2LaMusique(vincent), happy(vincent).
```

Syntaxe

La virgule exprime la conjonction en Datalog.

Interprétation

Vincent joue de l'AirGuitar, quand il est content et qu'il écoute de la musique.

Requêtes

- Est-ce que Vincent joue de l'AirGuitar ?
- Comment demander s'il existe une femme qui joue de l'AirGuitar ? [mot-clé réservé (answer)]

Exemple 4 : Disjonction logique

Paul joue de l'AirGuitar, quand il est content, ou qu'il écoute de la musique.

Syntaxe

Pour exprimer un OU logique en Datalog, on écrit deux règles.

```
ecoute2LaMusique(paul).

joueAirGuitar(paul) :- happy(paul).

joueAirGuitar(paul) :- ecoute2LaMusique(paul).
```

Requête

- Est-ce que Paul joue de l'AirGuitar ?

Résumé de la syntaxe (2)

Comment exprimer les opérateurs logiques en Datalog :

- conjonction : virgule
- disjonction : écrire deux règles
- implication $B \ :- \ A_1, \ldots, A_n \ .$
  - le corps est une conjonction de n faits
  - dans la tête, un seul fait
  - si toutes ses conditions sont satisfaites, on peut déduire le nouveau fait B

Catégories de règles

Nous pouvons distinguer 3 catégories de règles, de plus en plus expressives :

- règles avec faits simples, c.a.d. constantes (quand c'est la fête, il y a de la musique)
- règles avec prédicats + constantes
- règles avec prédicats + variables

Règles avec variables

Exemple :

- $avoir\_coffre\_fort(X) :- millionaire(X).$
- Signification : Tous les millionnaires ont un coffre-fort.
- En logique :
  $\forall x : millionnaire(x) \Rightarrow avoir\_coffre\_fort(x)$
- Fait : Balthazar Picsou est un millionnaire.
  $millionnaire(bp)$
- Lorsqu'on pose la requête
  $avoir\_coffre\_fort(bp)$, Datalog conclut du programme, que Balthazar Picsou a un coffre-fort, et répond oui.

Base : bars

Schémas

sert(bar, biere ,prix)
frequente(personne, bar)
aime(personne,biere)

```
sert(mcevans,lachouffe,3.5).
sert(mcevans,leffe,2.5).
sert(omnia,lachouffe,4.5) .
sert(taverneflamande,lachti,1.9).

frequente(timoleon,mcevans).

aime(timoleon,lachouffe).
```

Exemple de règle avec variables apparaissant uniquement dans corps

```
content(X) :-   aime(X, Beer) , frequente(X, Bar),
                sert(Bar, Beer, _).
```

En français

Si quelqu'un aime une certaine bière, et fréquente un bar qui vend cette bière, alors il est content.

Les variables qui apparaissent uniquement dans le corps

sont quantifiees existellement !

Les variables qui apparaissent dans deux relations

sont partagées : meme nom, meme valeur.

Questions auxquelles Datalog sait répondre

Exemples

- Timoléon est-il content ?
- Qui est content ?

```
sert(mcevans,lachouffe,3.5).
sert(mcevans,leffe,2.5).
sert(omnia,lachouffe,4.5)
sert(taverneflamande,lachti,1.9)

frequente(timoleon,mcevans).

aime(timoleon,lachouffe).

content(X) :- aime(X,Beer), frequente(X,Bar),
    sert(Bar,Beer,_).
```

Exemple de règle avec comparaison

Syntaxe

comparaisons *A op B*, ou *A* et *B* sont des constantes ou variables, et *op* un opérateur de comparaison

```
bonmarche(B)  :-  sert(B, _ ,P), P =< 2.0 .
```

Interprétation de la règle

Un bar est bon marché, s'il sert une bière à moins de deux euros.

Astuce

L'occurrence de la variable P dans le prédicat *sert* permet une liaison de cette variable. Uniquement après cette liaison, la variable peut être comparée.

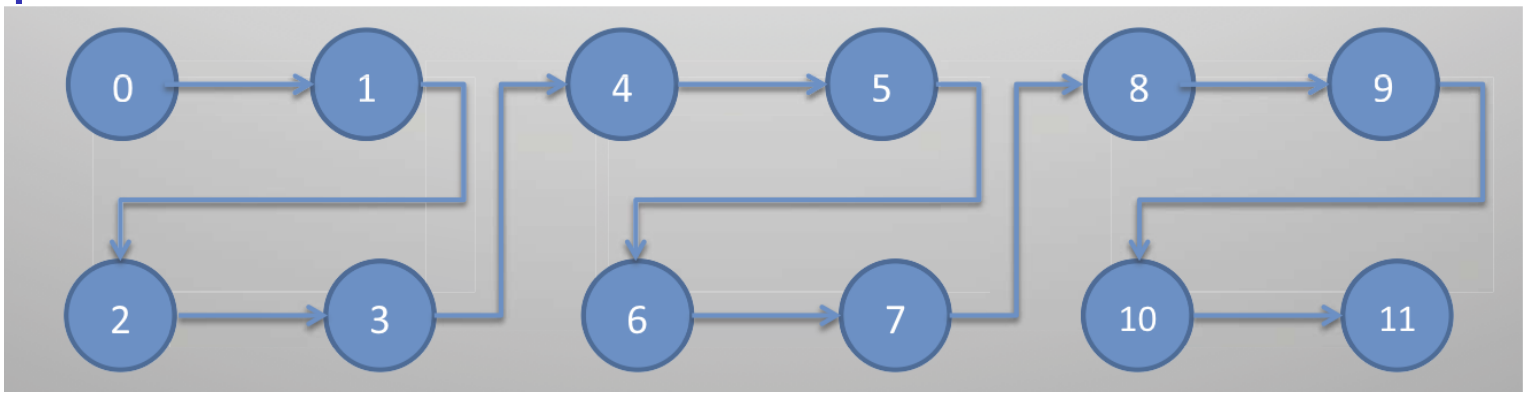
Signification des règles Datalog

Datalog est une machine pour construire des nouveaux faits.

Première approximation de la sémantique pour règles avec variables, non-récursives.

- prends les valeurs de variables qui rendent le corps de la règle vrai (il faut rendre vrai chacun des sous-buts)
- considère les valeurs que peuvent prendre les variables de la tête
- ajoute le tuple crée dans l'étape précédente, à la relation en tête de règle

Graphe



Le prédicat $e/2$ exprime un lien direct entre deux sommets (e pour anglais : edge) :

```
e(0,1). e(1,2). e(2,3).
e(3,4). e(4,5). e(5,6).
e(6,7). e(7,8). e(8,9).
e(9,10). e(10,11).
```

But : tester l'existence d'un chemin entre deux noeuds

- Requête pour tester l'existence des chemin de longueur fixe, p.ex. 2 et 3, entre deux noeuds.
- Peut-on poser les requêtes correspondantes en SQL ?

Chemins dans un graphe

- Comment tester si deux sommets (edge) sont connectés, quelque soit la longueur du chemin ?
- Définir un prédicat $p/2$, qui exprime un chemin (path) entre deux sommets passant par un nombre arbitraire de liens.

```
p(X,Y) :- e(X,Y).

p(X,Z) :- e(X,Y), p(Y,Z).
```

Datalog et l'algèbre relationnelle

- Nous montrons comment exprimer, en Datalog, des requêtes sur une base de données relationnelle, du type SELECT-FROM-WHERE, ou des requêtes en algèbre relationnelle
- les mêmes opérations qu'on peut définir en algèbre relationnelle, ou SQL de base, peuvent être définies en Datalog
- Nous supposons un nom de relation (symbole de prédicat) r en Datalog, pour chaque tableau R d'une base de données relationnelle

Exemple : la boutique en DES

- articles(*aid* : *int*, *anom* : *string*, *acoul* : *string*)
- fournisseurs(*fid* : *int*, *fnom* : *string*, *fad* : *string*)
- catalogue(*fid* : *int*, *aid* : *int*, *prix* : *float*)
- déclaration de types (create table) + ajoutà la EDB (inserts) :

```
:-type(articles(aid:int,anom:string,acoul:string)).
:-type(fournisseurs(fid:int,fnom:string,fad:string)).
:-type(catalogue(fid:int,aid:int,prix:float)).

articles(1,'Left Handed Toaster Cover','rouge').
articles(2,'Smoke Shifter End','noir').
...
fournisseurs(1,'kiventout','59 rue du Chti,
 F-75001 Paris').
fournisseurs(2,'Big Red Tool and Die','4 My Way,
 Bermuda Shorts, OR 90305, USA').
...
catalogue(1,1,36.10).
catalogue(1,2,42.30).
```

Projection $\pi_{Acoul}(Articles)$

Afficher toutes les couleurs d'articles

```
couleur(C) :- articles(_,_,C).
```

Requête et résultat avec l'outil DES :

```
DES> couleur(X)
{
   couleur(argente),   couleur(cyan),
   couleur(magenta),   couleur(noir),
   couleur(opaque),   couleur(rouge),
   couleur(superjaune),   couleur(vert)
}
Info: 8 tuples computed.
```

en SQL :

```
CREATE VIEW couleur AS
SELECT acoul FROM articles
```

Selection σ

Afficher tous les noms d'articles verts. Afficher tous les noms d'articles rouges.

```
art_vert(Anom)
  :- articles(_,Anom,'vert').
art_rouge(Anom)
  :- articles(_,Anom,Acoul),Acoul='rouge'.

art_vert(X).
art_rouge(X).
```

```
CREATE VIEW art_rouge(anom) as
 SELECT anom FROM articles WHERE acoul='rouge';
CREATE VIEW art_vert(anom) as
 SELECT anom FROM articles WHERE acoul='vert';
select anom from art_rouge;
select anom from art_vert;
```

Intersection

articles existant en rouge et en vert

```
rouge_et_vert(X) :- art_rouge(X), art_vert(X).
```

en SQL :

```
CREATE VIEW rouge_et_vert AS
(
art_rouge INTERSECT art_vert
);

select * from rouge_et_vert;
```

Différence

les articles existant en rouge, mais pas en vert

```
rouge_pas_vert(X) :-
    art_rouge(X), not(art_vert(X)).
```

en SQL :

```
CREATE VIEW rouge_pas_vert AS
(art_rouge MINUS art_vert)
```

Union

les articles rouges ou verts

```
rouge_ou_vert(X) :-  art_rouge(X).
rouge_ou_vert(X) :-  art_vert(X).
```

en SQL :

```
CREATE VIEW rouge_ou_vert AS
    (art_rouge UNION art_vert)
```

Produit cartésien

toutes les combinaisons de noms de fournisseurs et noms d'articles.

```
cart(Anom,Fnom) :-
    articles(_,Anom,_), fournisseurs(_,Fnom,_).
```

Requête et résultat :

```
DES> cart(X,Y)
{   cart('7 Segment Display','Alien Aircaft Inc.'),
    cart('7 Segment Display','Autolux'),
    ...
    cart('Smoke Shifter End','Vendrien'),
    cart('Smoke Shifter End','kiventout')}
Info: 54 tuples computed.
```

```
CREATE VIEW cart AS
SELECT anom, fnom FROM articles, fournisseurs
```

Jointure

Qui vend quel article a quel prix ?

```
quivendquoi(Anom, Acoul, Fid, Prix)
  :- articles(Aid, Anom, Acoul),
     catalogue(Fid, Aid, Prix).
```

Variable Aid partagée entre les deux relations. Ou bien, test d'égalité entre deux variables :

```
quivendquoi(Anom, Acoul, Fid, Prix)
  :- articles(A_aid, Anom, Acoul),
     catalogue(Fid, C_aid, Prix), A_aid=C_aid.
```

en SQL :

```
CREATE VIEW quivendquoi AS
SELECT anom, acoul, fid, prix
FROM articles c join catalogue a using (aid);

select * from quivendquoi;
```

Quantification existentielle

Articles offerts par au moins un fournisseur

```
vendu(Anom) :-
    catalogue(_,Aid,_),articles(Aid,Anom,_).
```

```
SELECT anom FROM articles a WHERE exists
  (select * from catalogue c where c.aid=a.aid)
```

Quantification universelle

Elle peut s'exprimer, puisque nous avons aussi bien la quantification existentielle que la négation. Nous verrons plus tard la stratification, une contrainte syntaxique, imposée en Datalog pour utiliser la négation.

Fonctionnalités supplémentaires de DES

Les opérations montrées jusqu'ici peuvent être faites avec n'importe quel DATALOG. Pour augmenter son attractivité, DES offre :

- fonctions d'agrégation : count, min,max,avg,sum
  - Versions avec 1,2 ou 3 arguments pour différents contextes.
- group by - having
- différentes jointures

Chemin le plus court en Datalog [source : F. Saenz-Perez]

```
path(X,Y,1) :- edge(X,Y).

path(X,Y,L) :-
   path(X,Z,L0),
   edge(Z,Y),
   count(edge(A,B),Max),
   L0<Max,          % assure la terminaison
   L is L0+1.       % cree L avec valeur L0+1

shortest_paths(X,Y,L) :-
    min(path(X,Y,Z),Z,L).   %  L: min pour Z
```

Le prédicat count/2, compte le nombre de résultats pour la requête (premier paramètre) et associe ce nombre au second paramètre. La condition $L0 < Max$ assure la terminaison (elle interdit de boucler infiniment dans un cycle du graphe). Variables de min/3 : une fonction, Z le paramètre de cette fonction pour lequel on veut obtenir le minimum, qui est associé a L

Requêtes récursives en SQL

- requêtes récursives dans le standard SQL depuis la quatrième révision SQL :99 (nom alternatif : SQL3)
- en Postgres depuis version 8.4 (2009)

```
WITH [RECURSIVE] with_query [, ...]
SELECT ...
```

syntaxe pour with_query :

```
query_name [ (column_name [,...]) ]
AS (SELECT ...)
```

- supposez la table *edge*(*origin*, *destination*) pour représenter le graphe

```
CREATE OR REPLACE VIEW
shortest_paths(Origin,Destination,Length) AS
WITH RECURSIVE
path(Origin,Destination,Length) AS
    (SELECT e.*,1 FROM edge)
  UNION
    (SELECT
        path.Origin,edge.Destination,path.Length+1
     FROM path,edge
     WHERE path.Destination=edge.Origin
           and path.Length <
            (SELECT COUNT(*) FROM Edge)
     )
SELECT Origin,Destination,MIN(Length)
FROM path
GROUP BY Origin,Destination;

% requete en SQL
SELECT * FROM shortest_paths;
```

Contenu

Extension vs intension

Prédicats

- extension : les prédicats dont les relations sont enregistrées dans la base, comme faits.
- intention : des prédicats définis par des règles (c.a.d. en tête)

Terminologie

- EDB (extensional database) : collection de relations extensionelles
- IDB (intensional database) : collection de relations intensionnelles

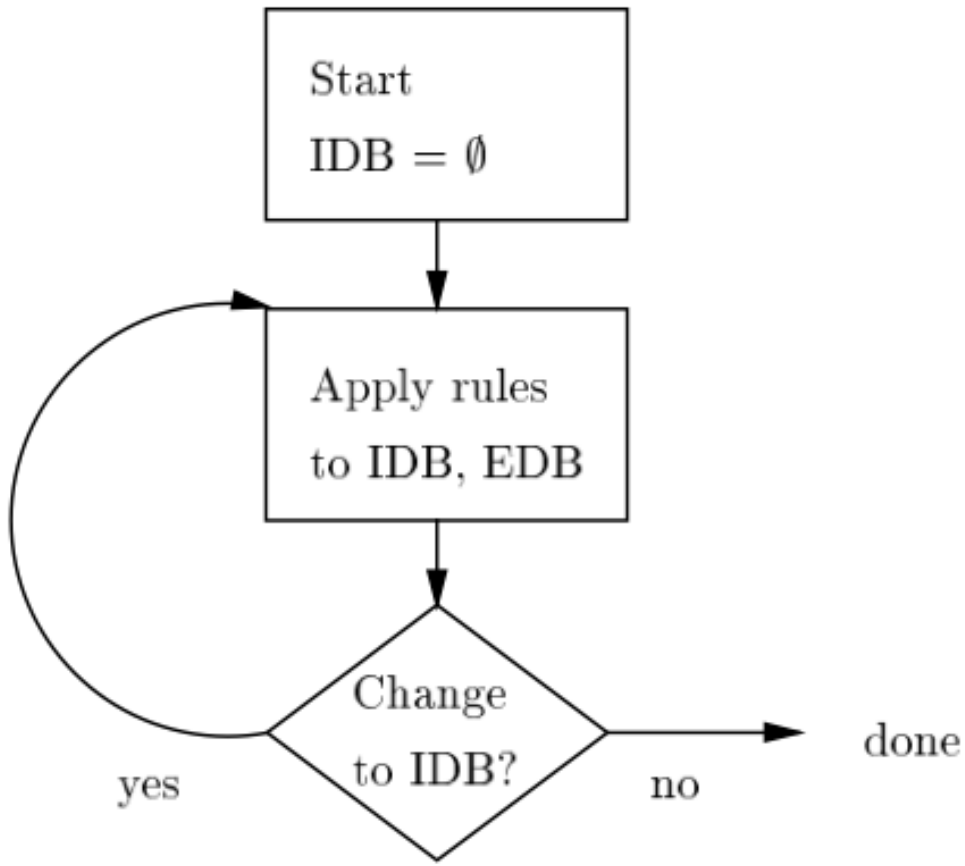
Exemple

Quels prédicats du schéma *bar* sont extensionnels, quels intensionnels ? Et pour les graphes et les chemins ?

Sémantique des points fixes

Idée de l'algorithme :

Iterative Fixed-Point Evaluates Recursive Rules



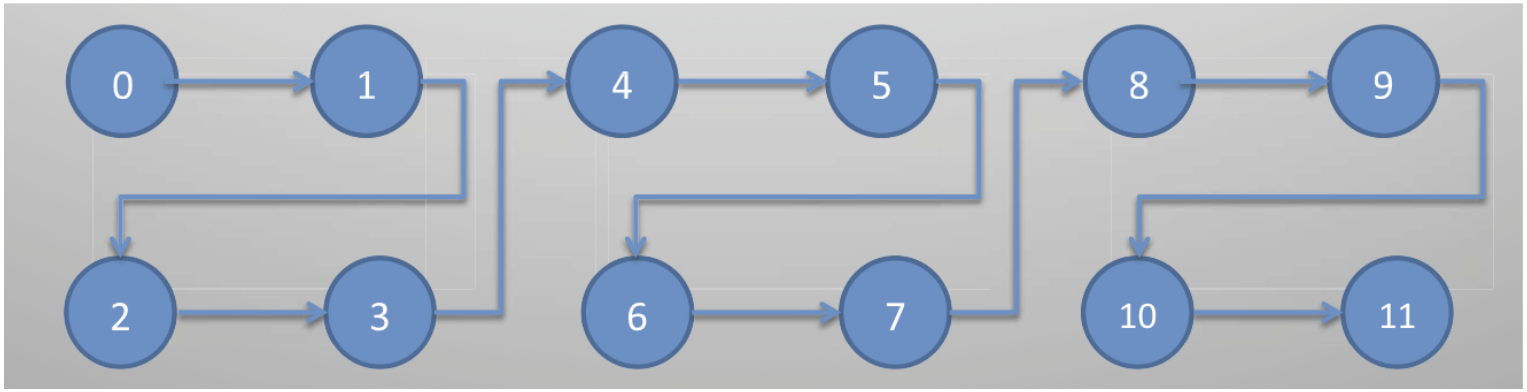
[source : Ullmann]

Sémantique des points fixes

Verbalisation de l'algorithme :

1. Suppose que tous les prédicats de la IDB sont vides. Uniquement les faits de la EDB sont présents initialement.
2. Construction de relations IDB de plus en plus grandes :
   - Applique les règles aux tuples dans la EDB, et ajoute les tuples obtenus aux relations IDB.
   - Utilise les tuples ajoutés à la IDB dans l'étape précédente pour ajouter encore plus de tuples à la IDB, avec des nouvelles applications de règles.
3. Continue à appliquer les règles, jusqu'à ce que cela n'ajoute plus de nouveaux tuples. On a atteint un point fixe. Si les règles sont sûres, il n'y aura qu'un nombre fini de tuples satisfaisant les corps des règles, et donc, le point fixe sera atteint avec un nombre borné de répétitions.

Exemple pour la sémantique : graphe



Le prédicat $e/2$ exprime un lien direct entre deux sommets :

```
e(0,1). e(1,2). e(2,3). e(3,4). e(4,5). e(5,6).
e(6,7). e(7,8). e(8,9). e(9,10). e(10,11).
```

Le prédicat $p/2$ exprime un chemin (path) entre deux sommets passant par un nombre arbitraire de liens :

```
p(X,Y) :- e(X,Y).

p(X,Z) :- e(X,Y), p(Y,Z).
```

Exemple : que peut-on déduire du programme ?

```
p(X,Y) :- e(X,Y).

p(X,Z) :- e(X,Y),p(Y,Z).

ok :- p(0,11).
```

EDB :

```
e(10,11).
e(9,10).
e(8,9).
e(7,8).
e(6,7).
e(5,6).
e(4,5).
e(3,4).
e(2,3)
e(1,2).
e(0,1).
```

IDB :

Nous montrons qu'il existe un chemin de 0 à 11 (on peut deduire plus du programme).

- instantiations des règles, puis déduction de nouveaux tuples dans la IDB, utilisant des tuples existants.
- au tableau ...

Exemple : que peut-on déduire du programme ?

```
p(X,Y) :- e(X,Y).

p(X,Z) :- e(X,Y),p(Y,Z).

ok :- p(0,11).
```

EDB :

```
e(10,11).
e(9,10).
e(8,9).
e(7,8).
e(6,7).
e(5,6).
e(4,5).
e(3,4).
e(2,3)
e(1,2).
e(0,1).
```

IDB :

```
p(10,11).
p(9,11).
p(8,11).
p(7,11).
p(6,11).
p(5,11).
p(4,11).
p(3,11).
p(2,11).
p(1,11).
p(0,11).
ok.
```

- pas 1 : règle 1 avec $e(10, 11)$, ajout de $p(10, 11)$ à la IDB.
- puis : règle 2 avec le prochain lien, et le dernier tuple ajouté à la IDB. injection d'un tuple supplémentaire la IDB. **répète.**
- règle 3 : quand possible.

(pas encore saturée !)

Comment expliquer l'absence de boucle infinie ?

```
a(X) :- b(X).                    % test.dl

b(X) :- a(X).
```

- DES rend le résultat rapidement pour la requête $a(4)$
- pourtant, la recherche de preuve devrait continuer à l'infini (d'après ce que nous avons vu jusqu'à présent)

Mémoisation

Technique d'optimisation de code visant à réduire le temps d'exécution. Datalog mémorise les faits qu'il a déjà prouvés, ou tenté de prouver. Évite de répéter le même calcul deux fois. Cette technique est absente en Prolog.

Commande DES pour voir ce qui est mémorisé : *list_et*

Motivation pour la sûreté

Sources de problèmes

Mauvaise utilisation des prédicats prédéfinis (comparaisons, négation . . .), et des variables.

- Le résultat d'une requête doit être une relation finie
- Certains types de buts (sous-requetes) génèrent un nombre infinis de lignes, alors qu'il est impossible que la table d'une relation R soit de taille illimitée
- il faut éliminer maximum les warnings **unsafe**

Règles correctes, ou sûres

Une règle est sûre, quand

chaque variable, et notamment,

- dans la tête
- dans une négation
- dans une comparaison

apparaît également sous forme positive dans le corps de la règle, dans un prédicat défini dans le même programme.

Le résultat d'une requête est toujours de taille finie.

Exemples de violations :

```
p(X) :- q(Y).
celibataire(X) :- not(marie(X,Y)).
celibataire(X) :- personne(X), not(marie(X,Y)).
homme(X) :- not(femme(X)).
```

Mauvais exemple avec variable

```
c(X) :- d.
d.
```

Permet de prouver un nombre infini de faits.

Mauvais exemples avec comparaison et arithmétique

```
% insecure1.dl
p(X) :- X>10.
```

```
% insecure2.dl

 c(Y) :- c(X), Y+1=X.
```

Requêtes closes vs requêtes avec variables.

Contenu

Mauvais exemples avec négation

```
homme(X) :- not(femme(X)).
celibataire(X) :- not(marie(X,Y)).
celibataire(X) :- personne(X), not(marie(X,Y)).
```

Chaque variable d'une règle doit apparaître dans le corps, sous forme positive, dans un prédicat défini par l'utilisateur.

Négation par l'échec

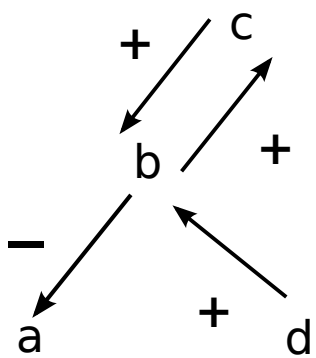
```
a :- not(b).
b :- c.
```

Si, avec le programme suivant, on pose la requête *a*, Datalog doit tester si *not*(*b*) est vrai. Pour cela, Datalog cherche une preuve pour *b*. Ceci échoue, puisque le fait *c* n'est pas donné. Datalog conclut que *not*(*b*) est vrai. Et par conséquent *a* est vrai.

Graphe de dépendances

Dépendences mutuelles des prédicats définis par l'utilisateur.

```
a :- not(b).                    % negation.dl
b :- c,d.
c :- b.
c.
```




```
DES>/pdg
Nodes: [d/0,a/0,b/0,c/0]
Arcs : [a/0−b/0,c/0+b/0,b/0+d/0,b/0+c/0]
```

- $x + y$: regle avec tête x ,et dans le corps y
- x–y : règle avec tête x , dans le corps $not(y)$

Négation et récursion

- en combinaison avec la récursion, la négation ouvre un nouveau niveau de complexité
- ce phénomène est connu et craint depuis l'antiquité
- il mène à des paradoxes logiques
- il peut ne pas exister de solution unique et minimale
- malgré la sûreté des règles

Le paradoxe du barbier

- dans un village, deux catégories d'hommes
  - se rasent eux-mêmes
  - ne se rasent pas eux-mêmes
- le barbier rase tous ceux qui ne rasent pas eux-mêmes

```
homme(barbier).                                    %barbier.dl
homme(maire).

rase(barbier,H) :-   homme(H), not(rase(H,H)).
```

Paradoxe : si le barbier...

- ne se rase pas lui-même : il doit donc être rasé par le barbier !
- se rase lui même : alors, il ne fait pas appel au barbier. Mais, pourtant, c'est lui le barbier !

Datalog : valeur logique undefined. warning : not stratisfiable.

Le problème de la négation avec récursion

```
p(X) :- r(X), not(q(X)).
q(X) :- r(X), not(p(X)).
```

Plusieurs modèles minimaux, supposant validité de $r(a)$:

- soit, $p(a)$ est valide, et q vide
- ou, $q(a)$ est valide, et p vide
- Absence de solution unique. On ne peut pas dire si $p(a)$ est valide, ou $q(a)$.
- La sémantique n'est pas claire. Aucune garantie pour la validité des résultats.

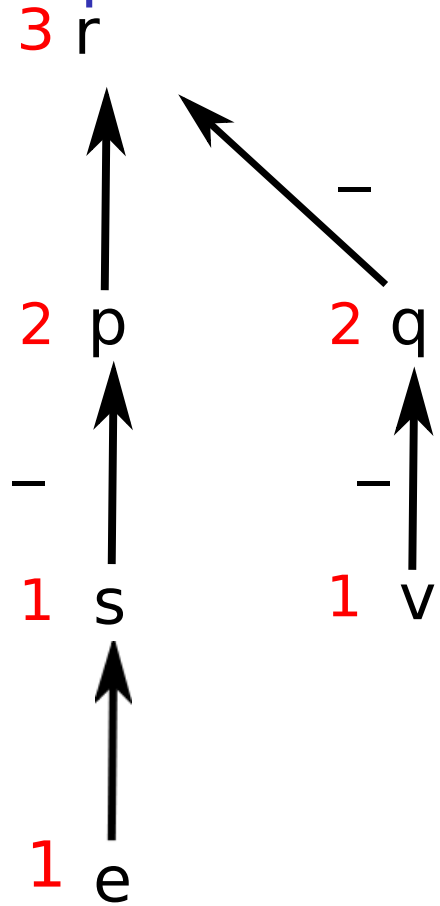
Programme stratifié

1. **Graphe de dépendances** décrit les dépendances mutuelles des prédicats
2. **Datalog stratifié** : interdit la négation dans les cycles , dans les graphes de dépendances
3. La sémantique est claire pour les programmes stratifiables. Ils peuvent être executés sans aucun risque.

Calcul des strates

- initialisation : tous les prédicats ont la valeur 1
- pour chaque prédicat, on considère les chemins terminant dans ce prédicat.
- la valeur augmente de 1 par flèche négative, dans un chemin de dépendances terminant dans ce prédicat
- le strate d'un prédicat est la valeur maximale dans un chemin terminant dans ce prédicat.

Exemple : calcul des strates




```
r(X) :- p(X), not(q(X)).
p(X) :- not(s(X)).
s(X) :- e(X).
q(X) :- not(v(X)).
e(999). v(999).
```

Exemple

Ce programme est stratifié :

- Strate 1 : e, s et v
- Strate 2 : p et q
- Strate 3 : r

Affichage des strates :

```
DES> /strata
[(e/1,1),(s/1,1),(v/1,1),(p/1,2),(q/1,2),
  (r/1,3)]
```

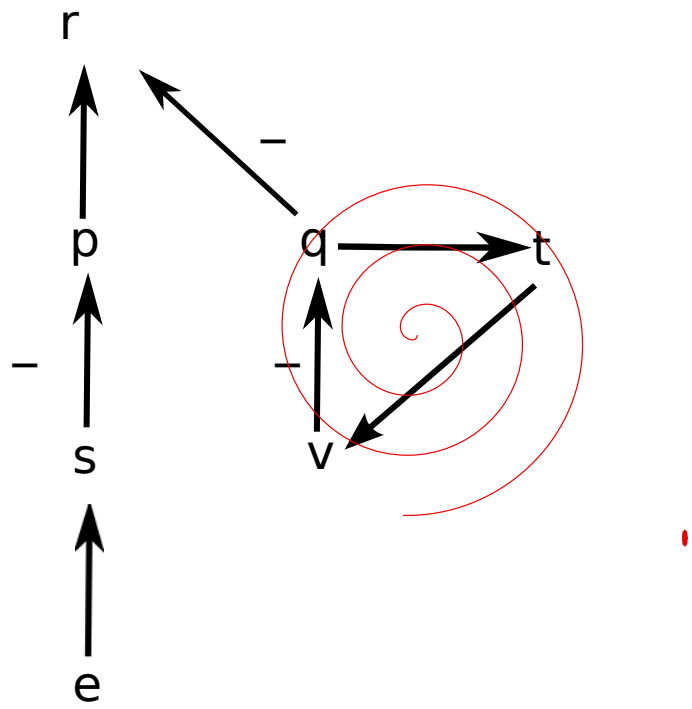
Autrement dit

- strate 1 : ces prédicats ne dépendent d'aucun prédicat sous négation
- strate 2 : dépend de prédicat négatif du strate 1
- strate 3 : dépend de prédicat négatif du strate 2
- ...
- strate n : dépend de prédicat IDB négatif du strate $n-1$

Programmes non stratifiables

Un cycle incluant une négation mène a des valeurs infinies. Un tel programme n'est pas stratisfiable.

Programme avec cycle negatif : non stratifiable




```
r(X) :- p(X) , not(q(X)).        % stratum1.dl
p(X) :- not(s(X)) .
s(X) :- e(X).
q(X) :- not(v(X)) .
v(X) :- t(X).
t(X) :- q(X).
```

Commandes DES

```
DES> \list_et    % information memorisee
DES> \safe       % transformation sure: ON ou OFF
DES> \pdg        % predicate dependency graph
DES> \strata     % strates
```

Le vendeur qui vend tout

```
ne_vend_pas(Fid,Aid):-
    fournisseurs(Fid,_,_),
    articles(Aid,_,_),
    not(catalogue(Fid,Aid,_)).

n_a_pas_tout(Fid) :-
    fournisseurs(Fid,_,_),
    articles(Aid,_,_),
    ne_vend_pas(Fid,Aid).

qui_a_tout(Fnom) :-
    fournisseurs(Fid,Fnom,_),
    not(n_a_pas_tout(Fid)).
```

Contenu

Différences entre Datalog et Prolog

Prolog est vu au S6 dans l'UE Logique.

- Tout ce qui peut être écrit en Datalog (a l'exception des constructions spécifiques de DES pour le groupage et l'aggregation), peut être écrit en Prolog.
- Prolog permet d'écrire des faits plus complexes que Datalog (distinction entre symboles de fonction et prédicats). Prolog permet du filtrage de motif sur les termes complexes. Par exemple, des listes sont des termes complexes. On peut donc en Prolog écrire des fonctions sur des listes, ce qui n'est pas possible en Datalog.
- En Prolog, on dessine manuellement des arbres pour expliquer le comportement d'un programme. Ces arbres s'appellent des arbres de résolution.

Différences entre Datalog et Prolog

Prolog est vu au S6 dans l'UE Logique.

- Prolog, contrairement a Datalog, n'a pas de sémantique formelle. Nous avons vu en survol une des trois sémantiques de Datalog, aujourd'hui.
- En Prolog, il n'y a pas de stratification. En Datalog, la stratification rend l'utilisation de la négation sure.
- Pour comprendre la négation en Prolog, il faut s'imaginer de couper des branches dans des arbres de preuve (voir UE Logique du S6).

Différences entre Datalog et Prolog

Prolog est vu au S6 dans l'UE Logique.

- Prolog n'a pas de message d'erreurs permettant de détecter des mauvais usages du langage. Il faut maîtriser des heuristiques syntaxiques complexes pour contourner des bugs. Ces derniers mènent a la non-terminaison de programmes. La non-terminaison peut etre visualisee par des branches infinies dans l'arbre de résolution, qui ne peut que etre obtenu manuellement...
- En Prolog, seulement certaines des variantes syntaxiques admises d'un programme se comportent correctement. Par exemple, en Prolog, pour la simple recherche dans un arbre (plus simple qu'un graphe), une seule variante fonctionne sur quatre.
- Toutefois, Prolog a des avantages. Les listes permettent d'ecrire des programmes puissants. Nous allons en utiliser un en fin de semestre, en TP.