Lecture Five: Creating a Knowledge Base

301315 Knowledge Representation and Reasoning

Principles for Creating a Knowledge Base

The collection of statements about the world we choose to give the agent is called the **knowledge base**.

When creating a knowledge base, it's important to

- model the domain with relations that ensure a high degree of elaboration tolerance;
- know the difference between knowledge representation of closed vs. open domains;
    - Can we assume our information about a relation is complete?
    - If we can't, what kinds of assumptions can we make?

Principles for Creating a Knowledge Base

- represent commonsense knowledge along with expert knowledge;
- exploit recursion and hierarchical structure.

Representing Family Relations - Basic Information

We start with a simply family relation program, called *family1.lp*:

```
father(john,sam).
mother(alice,sam).
gender_of(john,male).
gender_of(alice,female).
gender_of(sam,male).
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
child(X,Y) :- parent(Y,X).
```

Run *clingo*, what can we get?

Representing Family Relations - Basic Information

```
yan$ clingo family1.lp
clingo version 5.4.0
Reading from family1.lp
Solving...
Answer: 1
father(john,sam) parent(john,sam) parent(alice,sam)
mother(alice,sam) child(sam,john) child(sam,alice)
gender_of(john,male) gender_of(alice,female)
gender_of(sam,male)
SATISFIABLE

Models       : 1
Calls        : 1
```

Representing Family Relations - New Knowledge

Suppose John and Alice had a baby boy named Bill. Let's add Bill to our list of persons, and add facts about his mother and father.

```
father(john,bill).
mother(alice,bill).
gender_of(bill,male).
```

We also add new relation `brother(X,Y)`.

Representing Family Relations - New Knowledge

```
brother(X,Y) :- gender_of(X,male),
                father(F,X),
                father(F,Y),
                mother(M,X),
                mother(M,Y).
```

Let's call this new program `family2.lp`

However, if we run `family2.lp` under *clingo*, we will get one answer set which contains fact `brother(bill,bill)` - the agent thinks one can be his own brother!

Representing Family Relations - New Knowledge

How do you fix this problem?

```
brother(X,Y) :- gender_of(X,male),
                father(F,X),
                father(F,Y),
                mother(M,X),
                mother(M,Y),
                X!=Y.
```

Run this modified program using *clingo*, we will get rid of this issue.

Representing Family Relations - Representing Negative Information

If we ask whether Alice is Bill's father or if Sam is Bill's father, we would want our agent to answer "no", but we haven't taught it to do so yet.

Let's tell our agent that females can't be children's father and that a person can have only one father.

```
-father(X,Y) :- gender_of(X,female).
-father(X,Y) :- father(Z,Y),
                X != Z.
```

Adding these two rules, we call the new program *family3.lp*.

Representing Family Relations - Representing Negative Information

Now if we run `family3.lp` under *clingo*, what do we get?

```
yan$ clingo family3.lp
clingo version 5.4.0
Reading from family3.lp
family3.lp:21:1-37: error: unsafe variables in:
  (-father(X,Y)):-[#inc_base];gender_of(X,female).
family3.lp:21:11-12: note: 'Y' is unsafe

family3.lp:22:1-23:24: error: unsafe variables in:
  (-father(X,Y)):-[#inc_base];father(Z,Y);X!=Z.
family3.lp:22:9-10: note: 'X' is unsafe

*** ERROR: (clingo): grounding stopped because of errors
UNKNOWN
```

Representing Family Relations - Safety

- In a rule, a variable X is called **unsafe** if X only occurs in the head, and/or in the negative body of the rule (i.e., starting with *not*).
- Note that a variable *only* occurs in = or !=, not anywhere else in a rule's body, is still unsafe.

Example

axiom

```
    -father(X,Y) :- gender_of(X,female).
    -father(X,Y) :- father(Z,Y),
                    X != Z.
```

Variable `Y` in the first rule and variable `X` in the second rule are *unsafe*.

Representing Family Relations - Safety

- We can fix all unsafe variables by adding a new predicate `person(_)` into the program, to form a new program *family4.lp*:

```
person(john).
...
-father(X,Y) :- gender_of(X,female),person(Y).
-father(X,Y) :- father(Z,Y),person(X),X!=Z.
```

Representing Family Relations - Closed World Assumption (CWA)

- Generally speaking, the *Closed World Assumption (CWA)* states that if an agent does not know whether a fact is true or not, then the agent would assume this fact is false.
- CWA has been used in our commonsense reasoning all the time.
- In database query answering, CWA is applied explicitly.

Why is CWA important?

Representing Family Relations - Closed World Assumption (CWA)

- Let's add a new person to our program named Bob, i.e., add `person(bob)` into the program, to form *family5.lp*.
- What can we assume about John being Bob's father?
- What does our agent assume?

If we query $\Pi_{family5.lp} \models father(john, bob)$, i.e., is John the father of Bob?

In *clingo*, this is just done by checking if $father(john, bob)$ in *all* answer set(s) of program *family5.lp*.

Representing Family Relations - Closed World Assumption (CWA)

Under CWA, if we did not tell the agent that John was Bob's dad, it should assume that he is not. We do so by adding the closed world assumption for predicate `father`:

```
-father(X,Y) :- not father(X,Y).
```

Representing Family Relations - Closed World Assumption (CWA)

10 min classroom exercise

Consider a program Π consisting of the following rules:

$$\begin{array}{l} r(a). \\ r(b). \\ p(X) \leftarrow not\ q(X). \\ q(X) \leftarrow not\ p(X). \end{array}$$

(a) Compute all answer sets of program Π.

(b) Add CWA rules for predicates p and q into Π, to form a new program Π'.

(c) Compute all answer sets of Π'.

Defining Orphans - The Program

We have the following knowledge:

- We have a list of people.
- We have a *complete* list of children.
- For each child, we have the names of their parents.
- We have a complete record of deaths of people in our KB.

Defining Orphans - The Program

How can we teach our program the notion of orphan?

For someone to be considered an orphan, both their mother and their father have to be dead.
Program *orphans.lp*:

```
%% Facts

child(mary).
child(bob).
father(mike,mary).
father(rich,bob).
mother(kathy,mary).
mother(patty,bob).

dead(rich).
dead(patty).
```

Defining Orphans

```
%% Rules for Closed World Assumption

-child(X) :- not child(X).
-father(F,C) :- child(C),
                not father(F,C).
-mother(M,C) :- child(C),
                not mother(M,C).
-dead(X) :- not dead(X).
-orphan(X) :- not orphan(X).
```

Defining Orphans

```
%% P's parents are dead if both the father and
%% the mother of P are dead:

parents_dead(P) :- father(F,P),
                   mother(M,P),
                   dead(F),
                   dead(M).

%% P is considered an orphan if P is a child
%% and both parents are dead:

orphan(P) :- child(P),
             parents_dead(P).
```

Defining Orphans - Some Notes about the Program

- We could also define predicate -parents_dead using CWA:

  ```
  -parents_dead(X) :- not parents_dead(X).
  ```

  But this is not necessary, as it can be directly defined in terms of predicate -orphan:

  ```
  -parents_dead(X) :- -orphan(X).
  ```

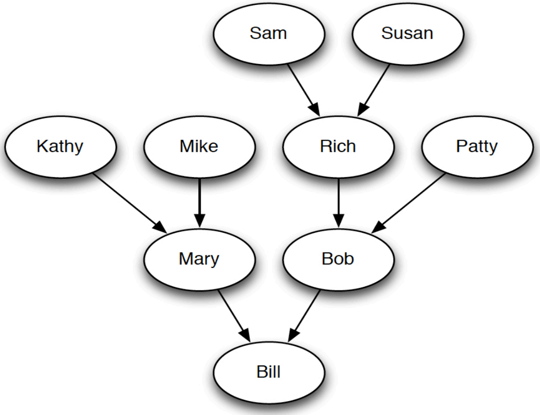
Defining Orphans - Some Notes about the Program

- If we add information that violates the notions of completeness that we outlined, the program may not answer intelligently. For example, what can we conclude given a new child, Perry, whose mother is Patty? What does the program conclude?

```
child(perry).
mother(patty,perry).
```

Defining Ancestors - The Family Tree

Our task: Given a complete family tree starting at some given ancestors, define the notion of ancestor.



Defining Ancestors - Exploiting Recursion

- Recursion - If A is an ancestor of B, B is an ancestor of C, then A is an ancestor of C.
- To represent this recursion, we first need to represent a *base*, e.g., “if A is a parent of B, then A is an ancestor of B”.
- Then we need to represent a *recursion* rule as above.

Defining Ancestors - The Program *ancestor.lp*

```
%% Facts

father(bob,bill).
father(rich,bob).
father(mike,mary).
father(sam,rich).
mother(mary,bill).
mother(patty,bob).
mother(kathy,mary).
mother(susan,rich).

%% Closed World Assumption

-father(F,C) :- not father(F,C).
-mother(M,C) :- not mother(M,C).
```

Defining Ancestors - The Program *ancestor.lp*

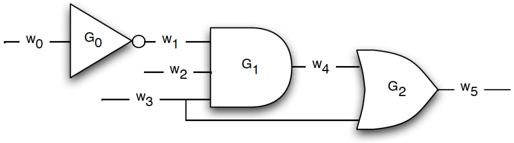
```
%% Rules

parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
-parent(X,Y) :- not parent(X,Y).   %% CWA

ancestor(X,Y) :- parent(X,Y).      %% Base
ancestor(X,Y) :- parent(Z,Y),      %% Recursion rule
                 ancestor(X,Z).
-ancestor(X,Y) :- not ancestor(X,Y) %% CWA
```

Describing an Electrical Circuit - The Problem

How do we describe an electrical circuit?



G_0: "not-gate"
G_1: "and-gate"
G_2: "or-gate"

What are the objects and relations that we are trying to represent?

Describing an Electrical Circuit - Choosing a Representation

- Objects: three gates G_i ($i = 0, 1, 2$), and six wires w_j ($j = 0, 1, 2, 3, 4, 5$)
- Relations: (a) object properties; and (b) connections between these objects

Describing an Electrical Circuit - The Program *circuit.lp*

```
%% Types of gates

type(g0,not_gate).
input(g0,w0).
output(g0,w1).

type(g1,and_gate).
input(g1,w1).
input(g1,w2).
input(g1,w3).
output(g1,w4).

type(g2,or_gate).
input(g2,w4).
input(g2,w3).
output(g2,w5).
```

Describing an Electrical Circuit - The Program *circuit.lp*

```
%% Example circuit input:
val(w0,1).    % The value of the signal on wire 0 is 1
val(w2,0).    % The value of the signal on wire 2 is 0
val(w3,1).    % The value of the signal on wire 3 is 1

%% A NOT gate flips the value of the signal:
opposite(0,1).
opposite(1,0).

val(W1,V1) :- output(G,W1),
              type(G,not_gate),
              input(G,W0),
              val(W0,V0),
              opposite(V1,V0).
```

Describing an Electrical Circuit - The Program *circuit.lp*

```
%% The output of an AND gate is 0 if at least one
%% input is 0:
val(W1,0) :- output(G,W1),
             type(G,and_gate),
             input(G,W0),
             val(W0,0).

%% It is 1 otherwise:
val(W1,1) :- output(G,W1),
             type(G,and_gate),
             -val(W1,0).
```

Describing an Electrical Circuit - The Program *circuit.lp*

```
%% The output of an OR gate is 1 if at least one
%% input is 1:
val(W1,1) :- output(G,W1), type(G,or_gate),
             input(G,W0), val(W0,1).

%% It is 0 otherwise:
val(W1,0) :- output(G,W1),
             type(G,or_gate), -val(W1,1).

 %% CWA
-val(W,V) :- not val(W,V).

%% A wire can have only one signal value
-val(W,V1) :- val(W,V2),
              V1 != V2.
```

Hierarchical Information and Inheritance - An Example

Example

Consider how we could represent the following information:

- The Narwhal is a submarine.
- A submarine is a vehicle.
- Submarines are black.
- The Narwhal is a part of the U.S. Navy.

Note that there is a lot that is implicit in this specification.

Hierarchical Information and Inheritance - An Example

```
sub(narwhal).
vehicle(X) :- sub(X).
black(X) :- sub(X).
part_of(narwhal,us_navy).
```

- Is the Narwhal a car?
- Is it red?
- Every time we wanted to add a new vehicle or color, we would have to add a couple lines to express negative information.

Hierarchical Information and Inheritance - An Example

Add new negative rules:

```
-car(X) :- sub(X).
-sub(X) :- car(X).
-red(X) :- black(X).
-black(X) :- red(X).
```

Or add new CWA rules:

```
-car(X) :- not car(X).
-sub(X) :- not sub(X).
-red(X) :- not red(X).
-black(X) :- not black(X).
```

Hierarchical Information and Inheritance - Representing Hierarchical Information

- **Inheritance hierarchy** can be used to significantly simplify our knowledge representation.
- Inheritance hierarchy is a tree-like structures of classes and subclasses.
- Objects in a subclass will *inherit* all properties of the class that the underlying subclass belongs to.

Hierarchical Information and Inheritance - Representing Hierarchical Information

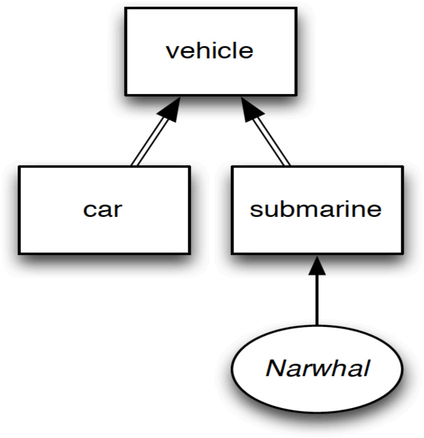


Figure: Hierarchy structure of vehicle.

Hierarchical Information and Inheritance - Representing Hierarchical Information

We introduce new predicates to define *class* and *subclass*, and other related concepts.

```
is_subclass(sub,vehicle).
is_subclass(car,vehicle).
is_subclass(vehicle,machine).

%% Subclass Relation:
subclass(C1,C2) :- is_subclass(C1,C2).

subclass(C1,C2) :- is_subclass(C1,C3),
                   subclass(C3,C2).

-subclass(C1,C2) :- not subclass(C1,C2).
```

Hierarchical Information and Inheritance - Representing Hierarchical Information

```
is_a(narwhal,sub).

%% Class Membership:
member(X,C) :- is_a(X,C).
member(X,C) :- is_a(X,C0),
               subclass(C0,C).
siblings(C1,C2) :- is_subclass(C1,C),
                   is_subclass(C2,C),
                   C1 != C2.
-member(X,C2) :- member(X,C1),
                 siblings(C1,C2),
                 C1 != C2.
```

Hierarchical Information and Inheritance - Representing Hierarchical Information

```
%% Submarines are black:
has_color(X,black) :- member(X,sub).

%% An object can only have one color.
-has_color(X,C2) :- has_color(X,C1),
                    C1 != C2.

%% The Narwhal is part of the U.S. Navy
part_of(narwhal,us_navy).

%% Other properties:
used_for_travel(X) :- member(X,vehicle).
-alive(X) :- member(X,machine).
```

Tutorial and Lab Exercises

1. Review section “Representing Family Relations” carefully, write a complete ASP program *family5.lp* and run it under *clingo*, and list all new information derived from the answer set of *family5.lp*.
2. Consider the following program Π:

   $$\begin{array}{l} r(a). \\ r(b). \\ s(c). \\ p(X,Y) \leftarrow not\ q(X,Y). \\ q(X,Y) \leftarrow not\ p(X,Y). \end{array}$$

   Add Closed World Assumption rules into Π, to form a new program Π', so that we can always derive negative information for all these predicates. Also transform your Π' into *clingo* syntax and run it under *clingo*.

Tutorial and Lab Exercises

3. Modify the existing vehicle hierarchy described in section "Hierarchical Information and Inheritance", to add the new subclasses and member shown in the following figure. Add some property of water vehicles to your program and make sure the Narwhal inherits this property and Abby does not.

Tutorial and Lab Exercises

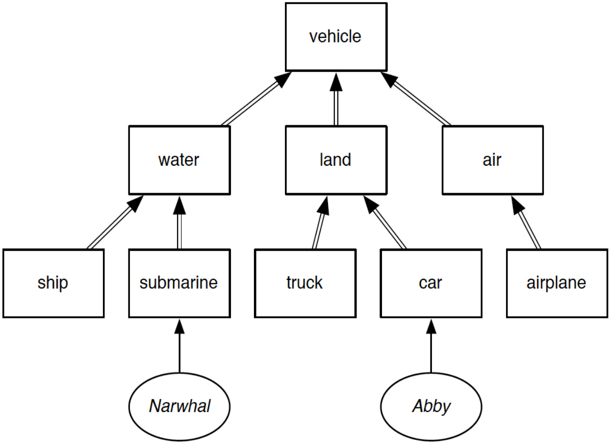


Figure: Subclasses of vehicle.