Table of Contents I

Reading

- Read Chapter 4 in *Knowledge Representation, Reasoning and the Design of Intelligent Agents* by Gelfond and Kahl.

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

Back to the Family Example

Let's implement the family example from Chapter 1 in SPARC:

```
sorts
#person = {john, sam, alice}.
#gender = {male, female}.

predicates
father(#person,#person).
mother(#person,#person).
gender_of(#person,#gender).
parent(#person,#person).
child(#person,#person).
```

Back to the Family Example, cont.

```
rules

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).
```

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.

```
sorts
  #person = {john, alice, sam, bill}.
predicates
  ...
rules
  ...
  father(john,bill).
  mother(alice,bill).
  gender_of(bill,male).
```

Let's add relation $brother(X, Y)$.

What's wrong with this?

Let's add `brother(#person,#person)` to our list of predicates and then add

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

Our agent thinks that you can be your own brother.

Let's add `X!=Y` to our premises.

Hidden Knowledge

A very large part of our knowledge is so deeply engrained in us that we do not normally think about it.

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 father children and that a person can have only one father.

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

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

Safety

If we were using straight ASP (without sorts), we would have run into a problem here. Our solver would have complained about the second rule being unsafe.

Broadly, a rule is **unsafe** if it contains an unsafe variable. An unsafe variable is one that does not occur in a literal in the body that is neither built-in nor preceded by default negation.

To make variables safe, we normally add sort information into the rule, so we end up having to add a sort such as person whether we use SPARC or not, and we have to add it to every rule that might be unsafe.

The New Guy

- Let's add a new person to our program named Bob.
- What can we assume about John being Bob's father?
- What does our agent assume?

Adding the CWA for *father*

If we wanted our agent to assume that if we did not tell it that John was Bob's dad, it should assume that he is not, we can add the closed world assumption for *father*.

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

Orphans Example

What do we know?

- 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.

What Is an Orphan?

How can we teach our program the notion of orphan?

There are two definitions in the dictionary. Let's pick the one that says that for someone to be considered an orphan, both their mother and their father have to be dead.

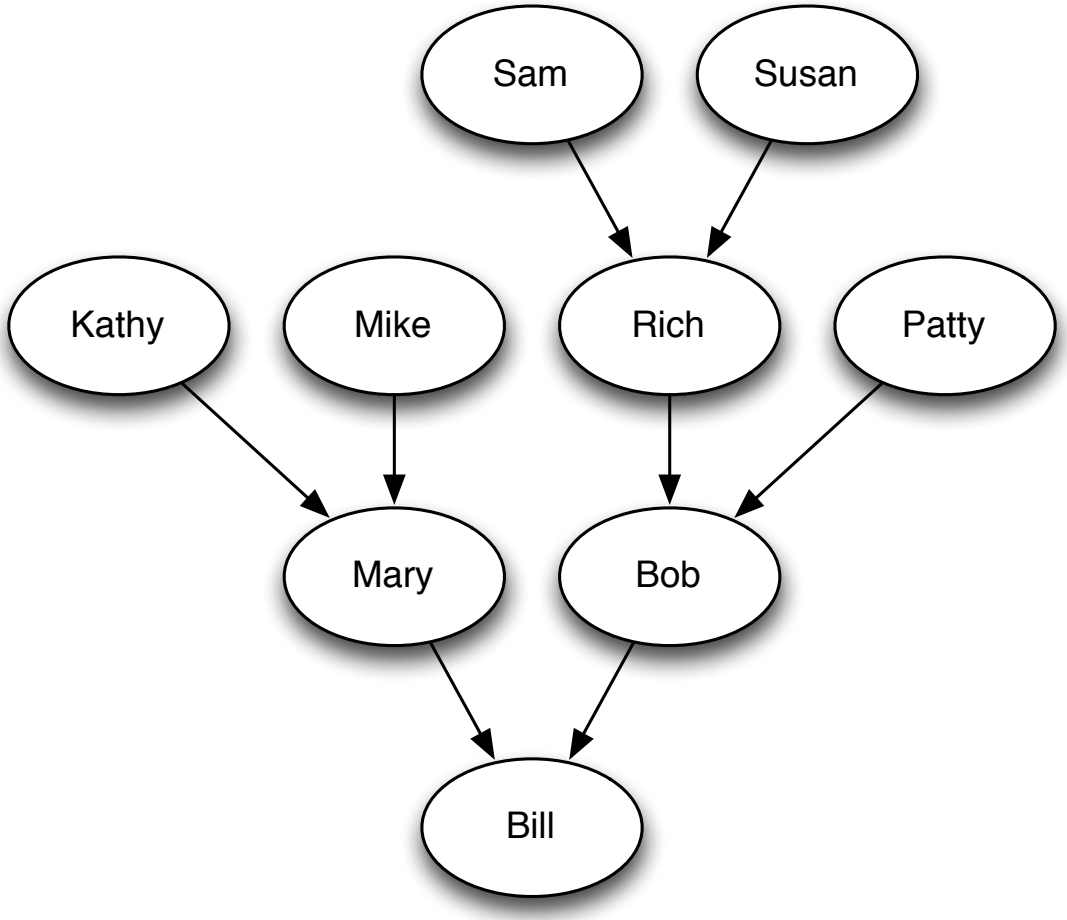
The program is at
`http://pages.suddenlink.net/ykahl/s_orphans.txt`

Some Notes on the Program

- Note that we are not defining `-parents_dead(P)`. This predicate was defined for readability of the code, and is not meant to be used beyond its limited purpose.
- 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?

Defining Ancestors

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

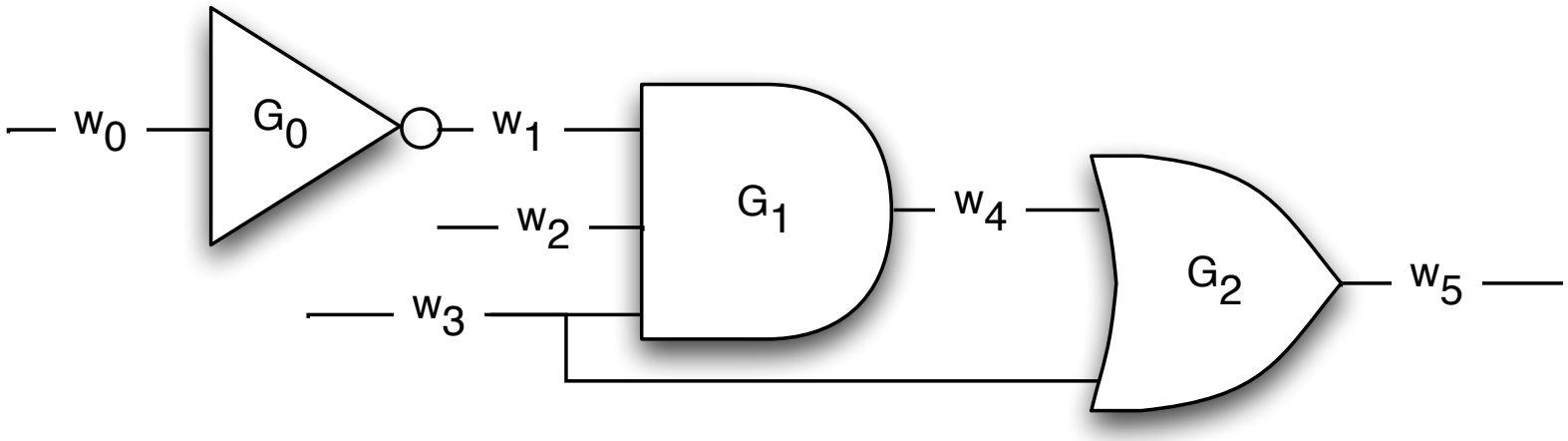


Exploiting Recursion

- Define the base case.
- Define the rest.
- Define when someone is not an ancestor. Is the CWA justified in this domain?

The program is at
`http://pages.suddenlink.net/ykahl/s_ancestors.txt`

How Do We Describe an Electrical Circuit?



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

Choosing a Representation

Choice 1: Gate+Inputs+Output makes a unit
Problem:

- Gates can have different numbers of inputs.
- Output wires of one gate can be input wires of another, but if we store the wire with the gate, we can't specify this relationship easily.

Choice 2: The objects are gates and wires. The connections between them are the relations.
`http://pages.suddenlink.net/ykahl/s_ec.txt`

Predicting the Output Values of Gates

- Add another sort called `signal` to denote the value of the current on a particular wire.
- Define relation `val` that gives the value of the signal for a given wire.
- `val` can be used to record facts about inputs, as well as to compute the output of a gate given the inputs.
- Define `val` for each type of gate.
- Add the CWA for `val` so that we know when a wire does not have a given value.
- Bonus rule: Check for inconsistency of input that assigns 0 and 1 to the same wire.

(See program.)

Evaluating Our Representation

- Is it readable?
- Is it easy to add new gates and connections?
- Can we add commonsense knowledge?

Example: Adding Commonsense Knowledge

- Assume the system has a sensor that tells it the actual value of the output wire of a gate by setting the value of predicate `sensor_val` for that wire. Then, if the sensor value does not match the predicted value, the gate must be defective.
- Let's define predicates `defective(#gate)` and `needs_replacing(#gate)`.

Example: Describing a Graph

(See program connected.sp.)

Hierarchical Information and Inheritance

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.

A Possible Solution

```
sorts
  #submarine = {narwhal}.
  #branch = {us_navy}.
predicates
  sub(#submarine).
  vehicle(#submarine).
  black(#submarine).
  part_of(#submarine, #branch).
rules
  sub(narwhal).
  vehicle(X) :- sub(X).
  black(X) :- sub(X).
  part_of(narwhal,us_navy).
```

- Is the *Narwhal* a car?
- Is it red?
- Even if we didn't have to worry about sorts, every time we wanted to add a new vehicle or color, we would have to add a couple lines to express negative information such as

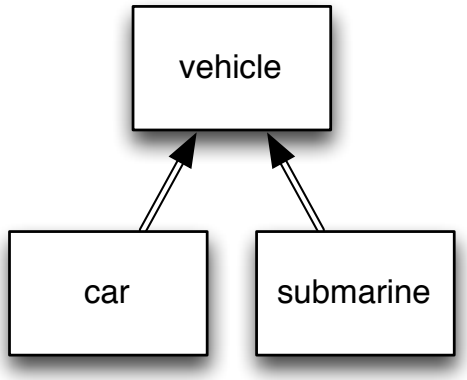
```
-car(X) :- sub(X).
-sub(X) :- car(X).
-red(X) :- black(X).
-black(X) :- red(X).
```

- We can do better.

Representing Hierarchical Information

- Humans are good at organizing the world into tree-like structures of classes and subclasses.
- For example, we recognize "submarine" as a class of things, members of which have some common properties.
- Because "submarine" is a subclass of "vehicle", objects that are submarines will *inherit* properties of vehicles.
- An **inheritance hierarchy** is a collection of classes organized in a tree formed by the subclass relation.

Subclasses in the Expanded Submarine Story



These are the classes in our expanded story.
In our representation, we will not exclude the possibility of there being other subclasses that we haven't mentioned. Therefore, we cannot conclude that a vehicle is either a car or a submarine; it could belong to some other class we haven't mentioned.

Representing Hierarchical Information: Reification

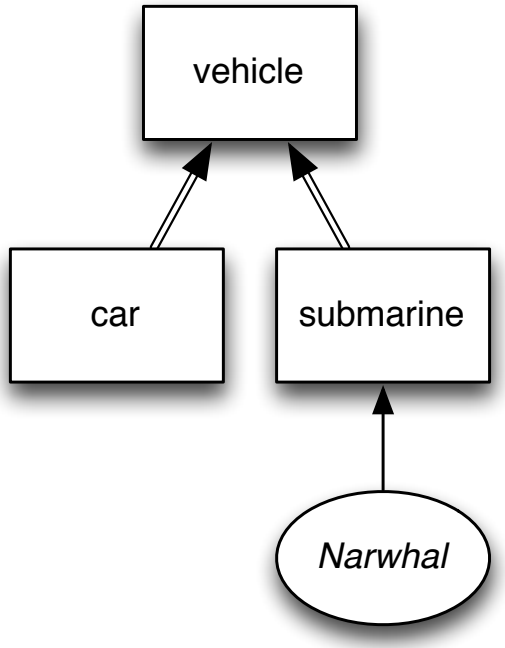
- Identify the *implicit* classes relevant to our story and *make them objects of our domain.*
- **reification** — the process of taking an implicit concept and making it an explicit object that we can reason about. I think of it as a jump to higher-order reasoning.
- Instead of just talking about a submarine called the *Narwhal*, we speak of a whole class of objects.

Representing Hierarchical Information: Classes and Subclasses

- Introduce a new sort — *class*.
- Introduce relation *is_subclass*(C_1, C_2) corresponding to the subclass links in the hierarchy.
- Define the *subclass* relation as transitive closure of *is_subclass*.

(See program s_hierarchy.sp at `http://pages.suddenlink.net/ykahl`.)

Representing Hierarchical Information: Adding Objects



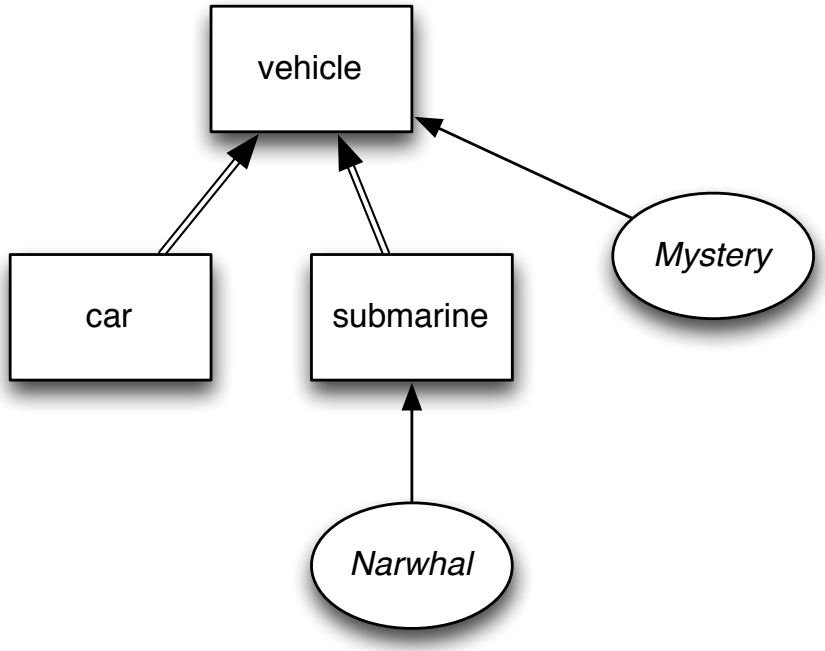
- Adding objects implies adding a new sort and a new type of link.
- The link is represented by the $is\_a(X, C)$ relation, where X is an object and C is a class.

Representing Hierarchical Information: Defining Membership

- An object is a member of a class via an *is_a* link.
- It is also a member of all classes that its direct class is a subclass of.

```
is_a(narwhal,sub).
member(X,C) :- is_a(X,C).
member(X,C) :- is_a(X,C0),
               subclass(C0,C).
```

Representing Hierarchical Information: Adding an Assumption



- Do we know which classes an object is not a member of?
- Often it is reasonable to assume that *children of a class in a hierarchy are disjoint;* e.g., cars are not submarines and vice versa.

Representing Hierarchical Information: Sibling Classes are Disjoint

If sibling classes are disjoint, then we know that if an object is a member of one sibling class, it is not a member of the other.

```
siblings(C1,C2) :- is_subclass(C1,C),
                   is_subclass(C2,C),
                   C1 != C2.
-member(X,C2) :- member(X,C1),
                 siblings(C1,C2),
                 C1 != C2.
```

What about colors?

- Add a sort called *color*.
- We can say that members of the submarine class are black.
- And we can add a rule stating that things of one color are not of another color.
- This negative information can now be computed for all members of the class.

Expanding the Knowledge Base

- How can we add a new color?
- How about a new class?
- A new property?

Comparing the Programs

- The first program started out much shorter, but with the expansion of the story, it soon lost this advantage.
- The second is much more general and elaboration tolerant.
- The challenge, as usual, is predicting how much the program might change and balancing compactness with elaboration tolerance.