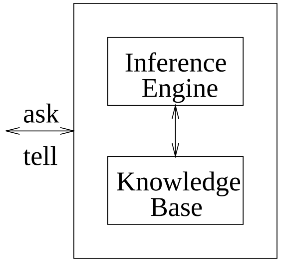
Knowledge Bases and Knowledge Engineering

1 Knowledge Bases

Our agent needs to represent and reason with large quantities of knowledge about the world. This knowledge is kept in a *knowledge base*. This contains a set of statements about the world, expressed in some logic/knowledge representation language. Reasoning with this knowledge is done by an *inference engine*. An inference engine is a program that draws conclusions from the knowledge in the knowledge base.

Any system that contains both a knowledge base and an inference engine may be called a *knowledge-based system* (KBS). Such a system will offer (at least) two methods in its public interface: `tell` and `ask`. `tell` is used to insert a new wff into the knowledge base; `ask` is used to query the knowledge base.



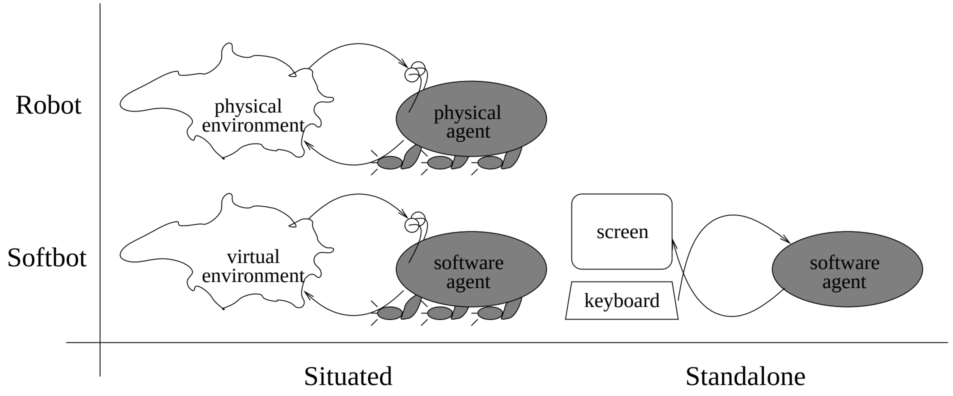
Knowledge-Based System

A *database system* also has methods that are seemingly equivalent to `tell` and `ask` (e.g. `INSERT` and `SELECT` in SQL respectively). While there are no clear-cut differences between KBS and database systems, it may help us to understand KBS better if we try to draw some distinctions:

- One difference is in the expressiveness of what can be `tell`ed. A typical relational database, for example, can only store data as rows in tables. Logically, it can only store wffs that are atoms whose arguments are all constant symbols. A KBS is less restrictive in the wffs it can store.
- Perhaps more importantly, there is a difference in the way that database systems and KBS respond to queries that they have been `ask`ed. In both cases, the answers must follow from what has been `tell`ed, but, in the case of KBS, extended chains of reasoning might be used to answer the queries. The answer may only be implicit in what was `tell`ed.

(One should note at this point that there is a large body of research into systems called 'deductive databases'. These systems blur the distinction between database systems and KBS.)

KBS play two roles in AI, depending on the kind of agents we are building. In lecture 2, we listed two kinds of agents: robots and softbots. But, for softbots, there was a special case where the agent is not really 'situated' in a world; rather, it is a standalone, usually interactive, system:

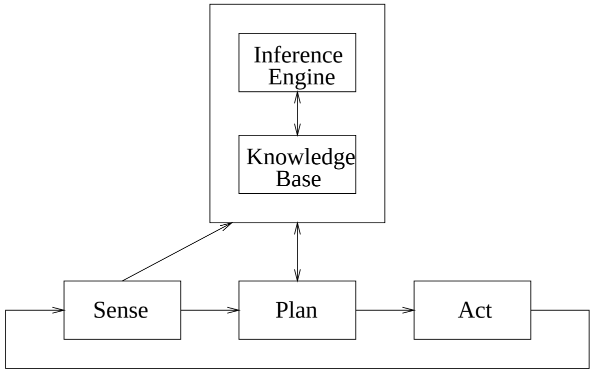


The first two of these make use of KBS differently from the third:

'Situated' agents: Robots are 'situated' agents and many softbots are 'situated' agents: they must act in rich worlds. (They differ in that robots have bodies and so they operate in physical environments, whereas softbots are pure software systems that operate in virtual environments such as the Internet.)

For 'situated' agents, the main purpose of the knowledge base is to hold background knowledge about the world in which they operate. There could be quite a lot of this knowledge. Much of it could be quite general-purpose. A lot of it would be what we might call commonsense knowledge. (The knowledge base might also be the place where we store the operators, the goal, the current state of the world and/or the current state of the plan we are building.)

The KBS is updated (using the `tell` method) primarily as a result of processing sensory inputs. The KBS is interrogated (using the `ask` method) by the planning algorithm. For example, suppose the algorithm needs to find an operator whose effects can achieve a so-far unachieved precondition. When we looked at planning algorithms (such as POP), we assumed that this was simply a matter of using *unification* to see whether one of an operator's effects matched a step's precondition. But, in general, unification may not be sufficient. An amount of reasoning may be necessary before we can determine that the operator is relevant to the unachieved precondition. This is why, in general, this part of the algorithm is done by querying the KBS.

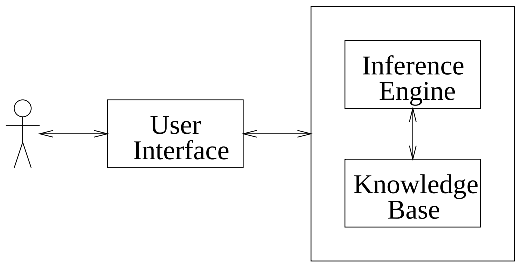


(An alternative architecture in which the planning algorithm and the inference engine are combined into a single module is also possible.)

'Standalone' agents: 'Standalone' agents are ones that are not embedded in rich worlds in which they must act. Rather, they receive suitably encoded descriptions of problems that need solving and deliver suitably encoded descriptions of the solutions.

This means that a 'standalone' agent is really little more than a KBS. It isn't, in general, planning sequences of actions that it will execute for itself in its environment.

Most commonly, a human user is updating and interrogating the KBS, via a user interface, much as s/he might interact with a database system. (Less commonly, some other system, e.g. a non-AI system, might be supplying data to and receiving data from the KBS.)



In 'standalone' agents, the knowledge base is less likely to contain lots of general-purpose knowledge. Rather, it will probably contain lots of knowledge about some specific problem domain. For example, it might contain knowledge about symptoms of meningitis, progression of the condition, and treatment of the condition. A human user might be able to use the system to aid in meningitis diagnosis and treatment. Or the knowledge base might contain knowledge about kitchen appliances. A human user might be able to use the system to aid in the layout of a new kitchen.

We are going to devote a considerable portion of the remaining part of this course to 'standalone' agents. They offer a practical, commercial form of AI, and are therefore worthy of study.

But before that, we will continue to talk more generally about KBS, covering especially the topics of knowledge representation and reasoning.

2 Knowledge Engineering

A *domain* is some aspect of the world. As we've discussed, 'standalone' agents tend to be domain-specific: they focus on one (or a very small number) of aspects of the world, e.g. meningitis diagnosis. 'Situated' agents, on the other hand, tend to work in richer environments and must therefore represent knowledge about many domains, e.g. the properties of physical objects, time, and beliefs about other agents.

We refer to the process of building a knowledge base for one or more domains as *knowledge engineering*. It is carried out by AI experts, *knowledge engineers*, in conjunction with people who are familiar with the domain, *domain experts* (e.g. medical consultants). The knowledge engineers must become acquainted with the domain, usually by interviewing the domain experts, and this part of knowledge engineering is known as *knowledge elicitation* or *knowledge acquisition*. Then, formal representations can be written and placed into the knowledge base.

The process of knowledge engineering involves such steps as:

Ontological engineering

Decide what to talk about. The knowledge engineer must first determine what is relevant and irrelevant to the system s/he wishes to build. What kinds of things exist in this domain? E.g. is time relevant? Are events, states and processes relevant or are we interested only in physical objects? Which physical objects are we interested in? Are all objects atomic or can they have subparts? Etc.

Decide on a vocabulary of predicate symbols, function symbols and constant symbols. For example, should some relationship be represented as a function symbol or as a predicate symbol? For example, to represent *"The Sun is yellow"*, we could choose between

- *yellow*(*sun*),
- *colour*(*sun*, *yellow*)
- *propertyOf*(*sun*, *colour*, *yellow*)
- *equals*(*colourOf*(*sun*), *yellow*)

and any number of further options.

Question. *Give some advantages/disadvantages for these options.*

Axiomatisation

Encode general knowledge. Write wffs to capture relationships between concepts.

Encode specific knowledge. Write wffs to capture specific problem instances.

We'll look at an example that uses these four steps in the next lecture. But, in the current lecture, we'll take a much simpler example domain so that we can practice writing wffs.

3 An Example Domain

Before we start on the example, let's check our ability to convert two simple English sentences into FOPL.

Exercise. *Which are the correct translations into FOPL?*

- *Every person is happy.*

$$\forall x(\textit{person}(x) \wedge \textit{happy}(x))$$
$$\forall x(\textit{person}(x) \Rightarrow \textit{happy}(x))$$

- *Some person is happy.*

$$\exists x(\textit{person}(x) \wedge \textit{happy}(x))$$
$$\exists x(\textit{person}(x) \Rightarrow \textit{happy}(x))$$

Now we'll encode a few facts about the domain of family relationships. We'll use the following 'key' for the unary predicate symbols *male* and *female*, the binary predicate symbols *parent*, *child*, *grandparent*, *sibling* and *equals*, and the unary function symbols *father* and *mother*:

male(x)	:	x is male
female(x)	:	x is female
parent(x, y)	:	x is a parent of y
child(x, y)	:	x is a child of y
grandparent(x, y)	:	x is a grandparent of y
sibling(x, y)	:	x is a sibling of y
equals(x, y)	:	x is equal to y
father(x)	:	the father of x
mother(x)	:	the mother of x

(For tidiness, we'll allow ourselves to write $x = y$ using $=$ as an infix predicate symbol in place of the more cumbersome and less familiar but syntactically more accurate *equals*(x, y).)

Here are some facts we'll translate into logic.

- Male and female are disjoint sets.
- Parent and child are inverse relationships.
- A grandparent is a parent of one's parent.
- One's mother is one's female parent.
- A sibling is another child of one's parents.

And so on!

In fact, strictly we can't just use *equals* (or $=$) without also tackling the domain of equality. We need to write wffs to capture the following facts:

- Equality is reflexive

$$\forall x\,(x = x)$$

- Equality is symmetric

$$\forall x \forall y(x = y \Rightarrow y = x)$$

- Equality is transitive

$$\forall x \forall y \forall z((x = y \wedge y = z) \Rightarrow x = z)$$

- Equality is substitutive

$$\forall x \forall y((\textit{male}(x) \wedge x = y) \Rightarrow \textit{male}(y))$$
$$\forall x \forall y((\textit{female}(x) \wedge x = y) \Rightarrow \textit{female}(y))$$

and so on for all other predicate symbols and function symbols

(There is an alternative to doing this, and this is to revise our semantics for FOPL. Within a revised semantic, we can give the *equals* predicate symbol a special, fixed interpretation. This would remove the need to specify the reflexivity, symmetry, transitivity and substitutivity properties.)

4 What is a good set of wffs

The statements that are in the knowledge base initially are sometimes called *axioms*. They are our basic facts about the domain. It is from these that the inference engine derives other facts (sometimes called *theorems*). The question is: how do we know when we've got a good set of axioms in our knowledge base?

Well, obviously, we need to write *'enough'* wffs. We want to keep writing axioms until all true facts about our domain (that we deem relevant) follow from the axioms, and only facts that are true in the domain follow from the axioms.

But mathematicians often try to write a *minimal* set of axioms. They try to make it the case that they never write redundant axioms. A redundant axiom would be one that could be derived from the other axioms. They aim for a minimal set from which all other facts about the domain (that they deem relevant) can be derived.

Some knowledge engineers strive for similar elegance when representing knowledge about a domain. However, this is not strictly necessary: our goal is not elegance. Efficiency of question-answering is much more important. And here the issue becomes less clear-cut. On the one hand, the more facts about the domain that we represent explicitly (including redundant ones), then the less reasoning we need to do when answering questions, which should improve efficiency. On the other hand, the more axioms in the knowledge base (including redundant ones), then the more 'stuff' there is that the inference engine has to plough through, which may worsen efficiency. Some compromise is needed.

Exercise

In addition to the predicate symbols and function symbols used earlier, we'll now also use the following:

brother(x, y)	:	x is a brother of y
sister(x, y)	:	x is a sister of y
aunt(x, y)	:	x is an aunt of y
uncle(x, y)	:	x is an uncle of y
cousin(x, y)	:	x is a (first) cousin of y
ancestor(x, y)	:	x is an ancestor of y

Now translate the following statements into FOPL:

1. Your brother is your male sibling.
2. Your sister is your female sibling.
3. An aunt is a sister of a parent.
4. An uncle is a brother of a parent.
5. Cousins are children of aunts or uncles.
6. One's ancestors are one's parents or ancestor's of one's parents. (A recursive definition!)