Reasoning Systems

In this part of the module we take a broad look at a variety of techniques for creating reasoning systems.

These are traditional AI techniques that are successfully used in modern applications.

Topics:

- Rule-based systems (this lecture)
- Reasoning about uncertainty: certainty factors in RBS
- Reasoning about uncertainty: fuzzy logic
- Case-based reasoning

 CSCU9T6 1

Rule-Based Systems

We begin by meeting one of the oldest kinds of reasoning: *rule based systems*.

These are also known as *expert systems*.

Idea: to capture the knowledge of a human expert in a specialized domain and embody it within a software system.

The knowledge is stored as rules of the form
IF condition THEN action

For example: If income < 1000 THEN deny-mortgage

Rule based systems were invented in the early 1970's and are still in use today. Let's take a brief look at their history…

 CSCU9T6 2

Rule-Based Systems: early history

Early AI (1956-late 1960s): great expectations, followed by disappointment and disillusion. Early efforts were too ambitious, attempting to tackle large, difficult problems.

Early 1970s: realisation that it would be better to focus on specific, restricted problem domains – this led to the first great successes of AI:

DENDRAL (Buchanan *et al*, 1969) – an "expert system" for analysing chemical (intended for studying the soil on Mars)

MYCIN (Feigenbaum *et al*, 1972) – system for diagnosis of infectious blood diseases. Introduced certainty factors for handling uncertainty

PROSPECTOR (Duda *et al*, 1979) – geological analysis of rocks and minerals. Used Bayesian methods to deal with uncertainty

 CSC9UT6 3

Rule-Based Systems Today

Modern rule-based systems are still based on many of the ideas of the early pioneers, but are often integrated within more complex systems.

Some modern examples of rule based systems:

The NHS Direct adviser

The Automobile Diagnosis adviser

Ikea online assistant - an RBS with a chatbox interface

American Express Authorizer's Assistant

- developed in 1988, but still in use today
- processes credit requests, deciding whether to authorise or deny
- very large: around 35,000 rules, including 3,000 "business rules"

 CSCU9T6 4

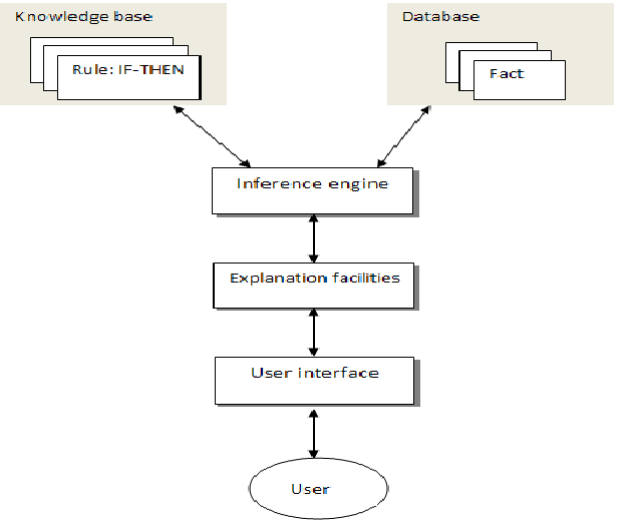
Architecture of a rule based system

A typical rule-based system consists of the following components:

- **knowledge base**: contains the rules embodying expert knowledge about the problem domain
- **database**: contains the set of known facts about the problem currently being solved
- **inference engine**: carries out the reasoning process by linking the rules with the known facts to find a solution
- **explanation facilities**: provides information to user about the reasoning steps that are being followed
- **user interface**: communication between the user and the system

 CSCU9T6 5

Architecture of a rule-based system



 CSCU9T6 6

Knowledge engineering

"....an engineering discipline that involves integrating knowledge into computer systems in order to solve complex problems normally requiring a high level of human expertise." – Feigenbaum and McCorduck

- Knowlege engineering is the process used to create a RBS:

  Assessing the problem and **selecting** an appropriate task for the RBS

  Interviewing the domain expert to find out how the problem is solved

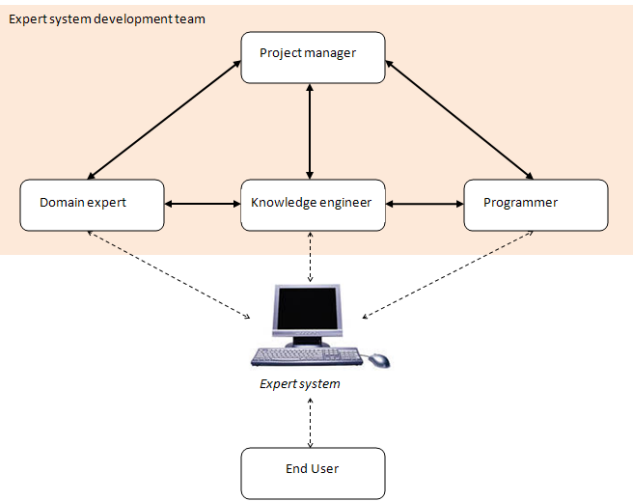
  Representing the domain knowledge as facts or rules

  Choosing appropriate development software such as a RBS "shell" and **encoding** the knowledge within that software

  Testing, **revising**, and **integrating** the system in the workplace

 CSCU9T6 7

Knowlege engineering team structure



 CSCU9T6 8

Expert system shells

Modern RBS are usually implemented by customizing an off-the-shelf expert system "shell".

These are also known as "rule engines".

The shell provides the inference engine, explanation facility, and infrastructure for populating the knowledge base and database. The shell provides interfaces for developers and (maybe) for end users.

The RBS is built by populating the knowledge base and database with suitable rules and facts, and (if necessary) creating suitable user interfaces.

Examples of shells: JESS "the rule engine for the Java platform"; CLIPS; Drools, RulesEngine, **e2glite** (which we will look at next)

E2glite: building a knowledge base

The knowledge base consists of a set of If…Then… rules. Here is an example of a rule:

```
RULE [Is the car out of petrol?]
If [the petrol tank] = "empty"
Then [the recommended action] = "refuel the car"
```

Because it is made up of rules the knowledge base is sometimes called the *rule base*.

Anatomy of a rule

Here is the rule we just saw:

```
RULE [Is the car out of petrol?]
If [the petrol tank] = "empty"
Then [the recommended action] = "refuel the car"
```

The part of the rule after the **If** is called the *premise* or *antecedent*. It contains what will be "subgoals".

The part of the rule after the **Then** is called the *consequent* or *conclusion*.

The *name* of this rule is **[Is the car out of petrol?]**

Using and and or in rules

The body of a rule may contain several subgoals, combined with **and** or **or**. Here is an example using **and**:

```
RULE [Is the petrol tank empty?]
If [the result of trying the starter] = "the engine tumbles normally" and
   [a petrol smell] = "not present when trying the starter"
Then [the petrol tank] = "empty"
```


Using and and or in rules

Here is an example of a rule using **or**:

```
RULE [Is the battery dead?]
If [the result of switching on the headlights]
= "nothing happens" or
   [the result of trying the starter] =
"nothing happens"
Then [the recommended action] = "recharge or
replace the battery"
```

Note: **and** and **or** may not be used together in the same rule.

Attributes

Attributes are somewhat like variables in Java, in that they are used to store values for future use. The values may be boolean (true/false), text strings, or numeric.

Attributes do not need to be declared and can hold multiple values at the same time.

Attribute names are enclosed in square brackets, may contain spaces, and are not case-sensitive.

Here are some examples of attribute names:

```
[a petrol smell]
[the recommended action]
[the result of trying the starter]
```


Giving values to attributes

Attributes may be given a value in the *conclusion* of a rule.

For example the conclusion of the rule `[Is the car out of petrol?]` gives a value to `[the recommended action]` :

```
[the recommended action] = "refuel the car"
```

Attributes may also be given a value via prompts for input from the user. We shall look at prompts later.

Comparing attributes

Attributes may be compared with each other or with literal values in the *premise* of a rule using various relational operators.

Examples of comparisons:

`[the petrol tank] = "empty"` (equals)

`[the age of the battery in months] < 24` (less than)

`[the petrol tank] ! "empty"` (not equal)

`[the manufacturer] : "BMW" "Opel" " Toyota"`

(equals any of. Used with multi-valued attributes only.)

User input

The value of an attribute may be obtained by input from a user. User input is obtained via *prompts*. Here is an example:

```
PROMPT [the result of trying the starter] Choice
"What happens when you turn the key to try to
start the car?"
"the engine tumbles normally"
"the engine tumbles slowly"
"nothing happens"
```

This is a `Choice` prompt: it displays a question followed by a drop-down list of response choices. The response is placed in the attribute `[the result of trying the starter]`
You will learn about other kinds of prompts in the practical.

The top level goal

The aim of a consultation with an e2glite RBS is to find a value (or values) for some goal attribute.

In the automobile diagnosis system, the goal attribute is `[the recommended action]`.

Goal attributes are specified via a `GOAL` command, e.g.:

```
GOAL [the recommended action]
```

A rule base must contain one or more `GOAL` commands.

If convenient, we may provide a default value for a goal attribute. This value will be returned if the system is unable to find any other value using the rules. For example:

```
DEFAULT [the recommended action] = "Scrap the car!"
```


Structure of a rule base

A typical rule base will consist of

- a set of rules, followed by…
- a set of prompts for user input, followed by…
- one or more goals, with optional default values

Comments may be used to separate the sections and to document the code. Comments are written in lines beginning with the command **REM**, for example:

```
REM*******************************************
REM                THE GOAL SECTION
REM The goal is to find the recommended action
REM*******************************************
```

 CSCU9T6 19

Inference in a Rule-Based KBS

The inference process used in a rule-based system is deductive inference. Recall that this means that the rules of logic are used to deduce new knowledge from existing knowledge and rules.

There are various different approaches to managing deductive inference. The most significant distinction is between

- Forward Chaining
- Backward Chaining

 CSCU9T6 20

Forward Chaining

Suppose we have a set of rules of the form

```
If A and B Then C
```

Then we may apply such rules to reason *forwards* from the known facts in our database, to establish new conclusions (i.e. facts) to add to the database. Specifically, if `A` and `B` are facts in our database, we can apply the rule above, and add `C` to the database. This is also called data-driven reasoning.

The problem with this approach is controlling it. At any given stage there may be *many* different rules which may be chosen to apply to extend the body of knowledge. Identifying them all is a potential problem (*matching*). Deciding which one to use is another (*conflict resolution*).

Forward Chaining (2)

Alternative approaches are:

- Use the first rule found.
- Attach priorities to rules.
- Use the most specific rule possible.
- Use *heuristics* to decide which rule will be most advantageous.
- Investigate in parallel the results of using different rules.

All approaches must deal with the problem of ***combinatorial explosion***.

Forward chaining is useful in RBS where no specific goal is being explored. It was used in the DENDRAL system for soil analysis.

Backward Chaining

We can think of our rules as:

```
C If A and B
```

We can then think of applying such rules in a *goal-driven* manner. If we have a specific goal in view (e.g., we are trying to confirm the diagnosis of some disease), then:

To justify `C`, apply the rule to generate *subgoals* `A` and `B`.

Then attempt to justify the subgoals in the same manner, by generating further subgoals, until ...

... all subgoals are facts present in the database.

In general, this also requires some *control*:

- Where alternative rules exist, the order may matter.
- Many dead-ends may be tried before the solution is found.

Backward chaining was used in the MYCIN diagnosis system

CLIPS is a well-known rule-based system shell that uses forward chaining. It provides a language for representing facts and rules, and an inference engine to do the processing.

The language is based on the AI language LISP.

The inference engine works reasonably efficiently by using the *Rete Algorithm* for the *matching* process.

To ease the *conflict resolution* problem, CLIPS has a mechanism for prioritising rules.

JESS ("Java Expert System Shell") is an implementation of CLIPS in Java. Being in Java it provides an interface between CLIPS and Java, and gives the potential for incorporating a KBS in a Java program.

The e2glite shell probably uses a combination of forward and backward chaining.

Background reading

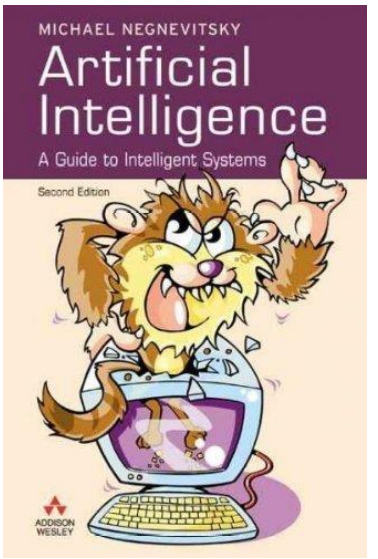
Artificial Intelligence: A Guide to Intelligent Systems,

3rd edition (2011),

Michael Negnevitsky,

ISBN: 1408225743,

Addison Wesley.



 CSCU9T6 25