DSS DEVELOPMENT

Business face a various of problems and therefore, variety of decisions to be taken

Example,

i. Allocation problem,
ii. Classification problem,
iii. Prediction problem,
iv. Problems with huge amount of data (especially for present and future),
v. Network-related problems,
vi. Complex problems with uncertainty, .

Note: These problems can be solved using different techniques.

For example, for allocation problems, Linear Programming (LP) algorithms can be applied;

For prediction problem, regression of different types depending on the type of problem can be applied;

For classification problem, decision tree or some other techniques are applicable;

Problems with huge amount of data can be handled by data warehousing techniques;

For network-related problems, networking methods or methods like Petri net can be applied.

Thus, different kinds of problems demand different kinds of analytical techniques

DECISION SUPPORT TOOLS

If a tool supports decision-making then that tool proves to be a decision support tool.

Examples

- Decision Tree
- Linear programming
- Predicate Logic
- Fuzzy theory & Fuzzy logic
- Network tools
- Markov Chain
- Case based reasoning
- Simulation etc

i)Decision Tree

A decision tree is a graphical tool for decision-making. Here a manager uses graphics to study alternative solutions available.

A decision tree is a graphical representation of the various alternatives available to solve a given problem to determine the most effective course of action.

This technique can be used for many different project management cases.

For example: Should we upgrade the software that we are using in our organization? Should we build a prototype for our new project? Should we select the low-price contractor? Should we select the low-budget project?

A decision tree is a **mathematical model** used to help managers make decisions.

- A decision tree uses **estimates** and **probabilities** to calculate likely outcomes.
- A decision tree helps to decide whether the net gain from a decision is worthwhile.
- Used to calculate the expected monitory value(EMV)

Decision tree analysis implementation steps

1. List all the decisions and prepare a decision tree for a project management situation.
2. Assign the probability of occurrence for all the risks.
3. Assign the impact of a risk as a monetary value.
4. Calculate the Expected Monetary Value (EMV) for each decision path.

The EMV a three-step process:

1. Determine the probability (P) an outcome will occur.
2. Determine the monetary value or impact (I) of the outcome.
3. Multiply P x I to calculate the EMV.

If a scenario presents multiple potential outcomes, calculate the EMV for each potential outcome and add them together to get the overall EMV.

Example 1

As a project manager you need to decide which sub-contractor is appropriate for your projects critical path activities. While selecting a sub-contractor, you should take into consideration the costs and delivery dates.

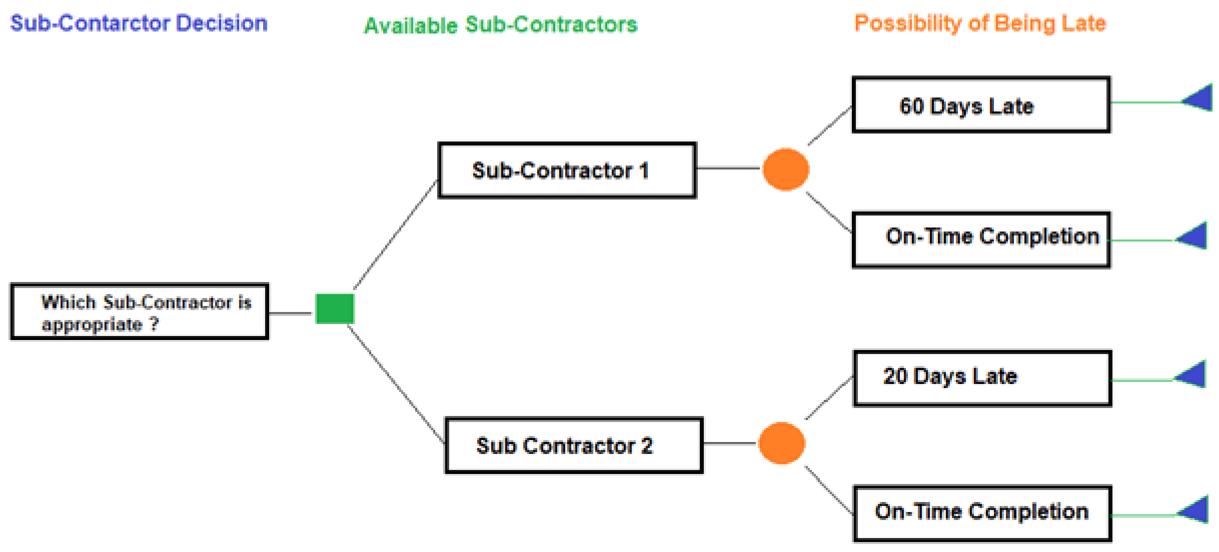
• Sub-contractor 1 bids $250,000. You estimate that there is a 30% possibility of completing 60 days late. As per your contract with the client, you must pay a delay penalty of $5,000 per calendar day for every day you deliver late.

• Sub-contractor 2 bids $320,000. You estimate that there is a 10% possibility of completing 20 days late. As per your contract with the client, you must pay a delay penalty of $5,000 per calendar day for every day you deliver late.

You need to determine which sub-contractor is appropriate for your projects critical path activities. Both sub-contractors promise successful delivery and high-quality work.

Solution

Step 1: List decisions and prepare a decision tree for a project management situation.



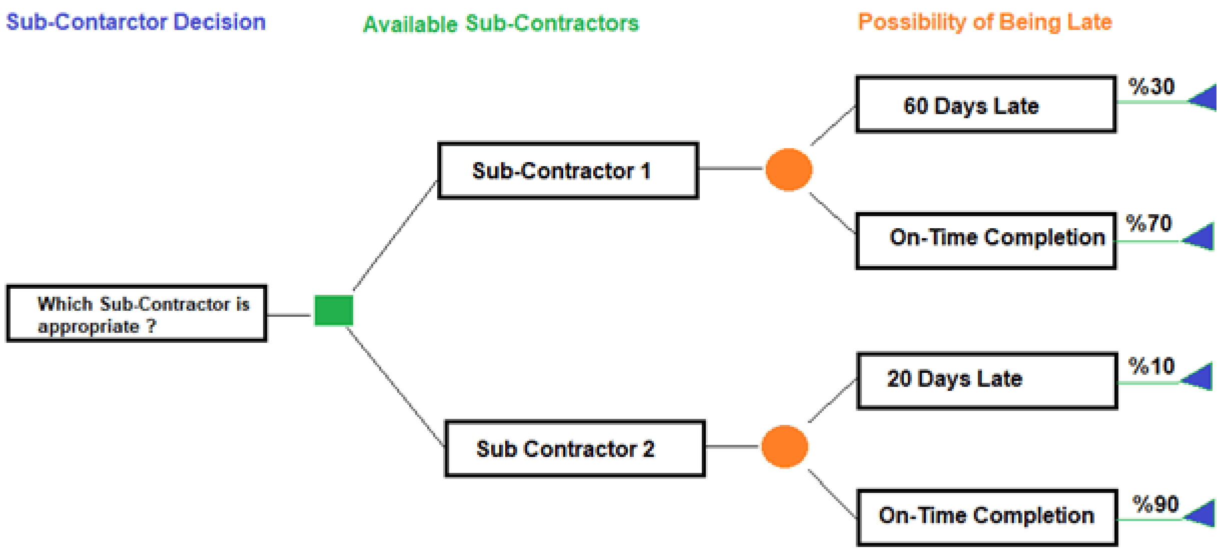
Step 2: Assign the probability of occurrence for the risks.

In this example, the possibility of being late for Sub-contractor 1 is 30% and for Sub-contractor 2 is 10 %. This means that the possibility of completing on-time for Sub-contractor 1 is 70% and for Sub-contractor 2 is 90 %.

In Figure 2 below the probability of occurrence for the risks are assigned.

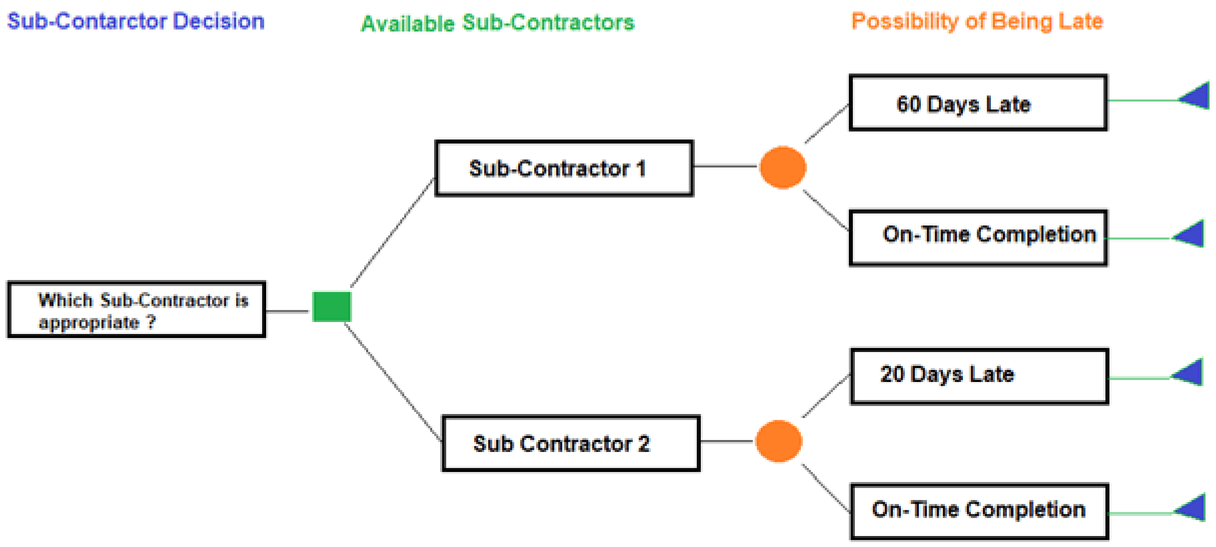
Step 2: Assign the probability of occurrence for the risks.

In this example, the possibility of being late for Sub-contractor 1 is 30% and for Sub-contractor 2 is 10 %. This means that the possibility of completing on-time for Sub-contractor 1 is 70% and for Sub-contractor 2 is 90 %. In Figure 2 below the probability of occurrence for the risks are assigned.



Step 3: Assign the impact of a risk as a monetary value.

In Step 3 we are calculating the value of the project for each path, beginning on the left-hand side with the first decision and cumulating the values to the final branch tip on the right side as if each of the decisions was taken and each case occurred. Figure 3 below shows the value of each path.



As shown in the figure, path values are calculated by the formulas given below.

Sub-Contractor 1
Path value of completing on-time = Bid Value = $ 250,000
Path value of being late = Bid Value + Penalty = $ 250,000 + 60 x $5,000 = $ 550,000

Sub-Contractor 2
Path value of completing on-time = Bid Value = $ 320,000
Path value of being late = Bid Value + Penalty = $ 320,000 + 20 x $5,000 = $ 420,000

Step 4: Calculate The Expected Monetary Value (EMV) for each decision path.

In Step 4, we are calculating the value of each node including both possibility nodes and decision nodes. We begin with the path values at the far right-hand end of the tree and then proceeding from the right to the left calculate the value of each node. This calculation is called "folding back" the tree.
The Expected Monetary Value (EMV) of each node will be calculated by multiplying Probability and Impact. Figure 4 below shows The Expected Monetary Value (EMV) of each path.

As shown in the figure, The Expected Monetary Value (EMV) of each path is below.

Sub-Contractor 1

EMV = %30 x $ 550,000 + %70 x $ 250,000 = $ 340,000

Sub-Contractor 2

EMV = %10 x $ 420,000 + %90 x $ 320,000 = $ 330,000
In this simple example Expected Monetary Values (EMV) are very close. Now we are selecting Contractor 2 because of low cost and low possibility of being late.

Linear Programming:

Linear programming, mathematical modeling technique in which a linear function is maximized or minimized when subjected to various constraints.

This technique has been useful for guiding quantitative decisions in business planning, in industrial engineering, and to a lesser extent in the social and physical sciences.

Linear programming is a method of optimizing operations with some constraints. The main objective of linear programming is to maximize or minimize the numerical value

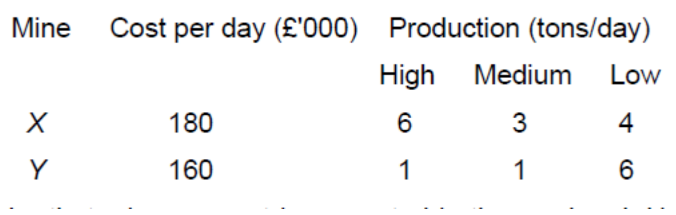
LP model. A model consisting of linear relationships representing a firm's objective and resource constraints

LP Applications

- Scheduling school buses to minimize total distance traveled
- Allocating police patrol units to high crime areas in order to minimize response time to 911 calls
- Scheduling tellers at banks so that needs are met during each hour of the day while minimizing the total cost of labor
- Selecting the product mix in a factory to make best use of machine- and labor-hours available while maximizing the firm's profit
- Picking blends of raw materials in feed mills to produce finished feed combinations at minimum costs
- Determining the distribution system that will minimize total shipping cost
- Developing a production schedule that will satisfy future demands for a firm's product and at the same time minimize total production and inventory costs

Example: Two Mines

The Two Mines Company own two different mines that produce an ore which, after being crushed, is graded into three classes: high, medium and low-grade. The company has contracted to provide a smelting plant with 12 tons of high-grade, 8 tons of medium-grade and 24 tons of low-grade ore per week. The two mines have different operating characteristics as detailed below.



Mine	Cost per day (£'000)	Production (tons/day)		
		High	Medium	Low
X	180	6	3	4
Y	160	1	1	6

Consider that mines cannot be operated in the weekend. How many days per week should each mine be operated to fulfill the smelting plant contract?

Solution
What we have is a verbal description of the Two Mines problem.
What we need to do is to translate that verbal description into an *equivalent* mathematical description.
In dealing with problems of this kind we often do best to consider them in the order:

- Variables
- Constraints
- Objective

This process is often called *formulating* the problem (or more strictly formulating a mathematical representation of the problem).

Variables
These represent the "decisions that have to be made" or the "unknowns".
We have two decision variables in this problem:
x = number of days per week mine X is operated
y = number of days per week mine Y is operated
Note here that $x \geq 0$ and $y \geq 0$.

Constraints
It is best to first put each constraint into words and then express it in a mathematical form.
ore production constraints - balance the amount produced with the quantity required under the smelting plant contract

Ore

High $6x + 1y \geq 12$

Medium $3x + 1y \geq 8$

Low $4x + 6y \geq 24$

Days per week constraint - we cannot work more than a certain maximum number of days a week e.g. for a 5 day week we have
$x \leq 5$ $y \leq 5$

Objective
The objective is to minimize cost which is given by
180x + 160y

Complete mathematical representation of the problem:
Minimize $180x + 160y$

Subject to

$6x + y \geq 12$ $3x + y \geq 8$ $4x + 6y \geq 24$ $x \leq 5$ $y \leq 5$ $x, y \geq 0$

Predicate Logic.

A formal system used in logic and mathematics to represent and reason about complex relationships and structures.

Allows facts about the world to be represented as sentences formed from propositional symbols using Logic Symbols.

Predicate logic forms the backbone of logical reasoning in AI, offering a structured approach to problem-solving and decision-making.

Basic Elements of predicate logic/ First-order logic:

$\wedge$ And

$\vee$ Or

$\neg$ Not

$\rightarrow$, = > Implies / Then

$\leftrightarrow$, < = > Equivalent to

- Implies: $\Rightarrow$
- Therefore: $\vdash$

Quantifiers

$\exists$ there exist

$\forall$ For all

Key Components of First-Order Logic / Predicate Logic.

1. **Constants**:

Constants are symbols that represent specific objects in the domain.

Examples: If a, b, and c are constants, they might represent specific individuals like Alice, Bob, and Ann

2. **Variables**:

Variables are symbols that can represent any object in the domain.

Examples: Variables such as x, y, and z can represent any object in the domain.

3. **Predicates**:

Predicates represent properties of objects or relationships between objects.

Examples: P(x) could mean “x is a person”, while Q(x, y) could mean “x is friends with y”.

4. **Functions**:

Definition: Functions map objects to other objects.

Examples: f(x) could represent a function that maps an object x to another object, like "the father of x".

5. **Quantifiers**:

Universal Quantifier (∀): Indicates that a statement applies to all objects in the domain.

Example, ∀x P(x) means "P(x) is true for all x".

Existential Quantifier (∃): Indicates that there exists at least one object in the domain for which the statement is true.

Example, ∃x P(x) means "There exists an x such that P(x) is true".

6. **Logical Connectives**:

These include ∧ (and), ∨ (or), ¬ (not), → (implies), and ↔ (if and only if).

Examples: P(x) ∧ Q(x, y) means "P(x) and Q(x, y) are both true".

7. **Equality**:

States that two objects are the same.

Examples: x = y asserts that x and y refer to the same object.

First-order logic statements can be divided into two parts:

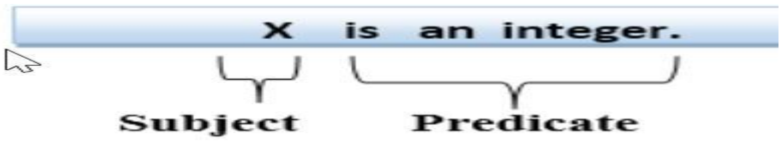
Subject: Subject is the main part of the statement.

Predicate: Defined as a relation, which binds two atoms together in a statement.

Consider the statement: x is an integer.

It consists of two parts, the first part x is the subject of the statement

Second part "is an integer," is known as a predicate.



Quantifiers in First-order logic:

A quantifier is a language element which generates quantification, and quantification specifies the quantity of specimen in the universe of discourse.

These are the symbols that permit to determine or identify the range and scope of the variable in the logical expression.

There are two types of quantifier:

Universal Quantifier, (for all, everyone, everything)

Existential quantifier, (for some, at least one).

Universal Quantifier:

Universal quantifier is a symbol of logical representation, which specifies that the statement within its range is true for everything or every instance of a particular thing.

The Universal quantifier is represented by a symbol ∀,

Note: In universal quantifier we use implication "→".

If x is a variable, then ∀x is read as:

- **For all x**
- **For each x**
- **For every x.**

Example:

All man drink coffee.

Let a variable x which refers to a cat so all x can be represented

∀x man(x) → drink (x, coffee).

Read as follows:

There are all x where x is a man who drink coffee.

Examples

i)All birds fly

The predicate is "**fly(bird)**."

All birds who fly can be represented as follows.
∀x bird(x) →fly(x).

ii) Every man respects his parent.

The predicate is "**respect(x, y)," where x=man, and y= parent**.

Since there is every man so will use ∀, and it will be represented as follows:

∀x man(x) → respects (x, parent).

iii) Some boys play cricket.

The predicate is "**play(x, y)**," where x= boys, and y= cricket. Since there are some boys hence **∃, represented as**:

∃x boys(x) → play(x, cricket).

iv) Not all students like both Mathematics and Science.
The predicate is "**like(x, y)," where x= student, and y= subject**.
Since there are not all students, we use **∀ with negation,**

¬∀ (x) [student(x) → like(x, Mathematics) ∧ like(x, Science)].

v) Only one student failed in Mathematics.

The predicate is "**failed(x, y)," where x= student, and y= subject**.

Since there is only one student who failed in Mathematics, then the representation is a follows.:

∃(x) [student(x) → failed (x, Mathematics) ∧∀ (y) [¬(x==y) ∧ student(y) → ¬failed (x, Mathematics)].

Predicate logic in decision making.

Predicate logic provides a formal basis for logical reasoning.

AI systems can use the rules of predicate logic to perform deductive reasoning, infer new facts from existing knowledge, and make informed decisions.

Prolog Fundamentals (Programming in Logic)

In Prolog, you arrive at solutions by logically inferring one thing from something already known.

Prolog program isn't a sequence of actions it's a **collection of facts** together with **rules** for drawing conclusions from those facts.

Prolog is therefore what is known as a **declarative language.**

Prolog includes **an inference engine**, which is a process for reasoning logically about information. The inference engine includes a pattern matcher, which retrieves stored (known) information by matching answers to questions. Prolog tries to infer that a hypothesis is true (in other words, answer a question) by questioning the set of information already known to be true.

For example, the following sentences are transformed into predicate logic syntax:

Natural Language:	**Predicate Logic:**
A car is fun. A rose is red.	fun(car). red(rose).
Bill likes a car if the car is fun.	likes(bill, Car) if fun(Car).

Sections of a Prolog Program

Program includes four basic program sections. These are the **clauses** section, the **predicates** section, the **domains** section, and the **goal** section.

The **clauses** section is the heart of a Visual Prolog program; this is where you put the facts and rules that Visual Prolog will operate on when trying to satisfy the program's goal.

The **predicates** section is where you declare your predicates and the domains (types) of the arguments to your predicates. (You don't need to declare Visual Prolog's built-in predicates.)

The **domains** section is where you declare any domains you're using that aren't Visual Prolog's standard domains. (You don't need to declare standard domains.)

The **goal** section is where you put the starting goal for a Visual Prolog program.

N.B: To execute the GOAL, you should activate the menu item **Project | Test Goal**, or press the **Ctrl+G** keys

Sentences: Facts and Rules (Facts, rules, and queries)

A Prolog programmer defines ***objects*** and ***relations,*** then defines ***rules*** about when these relations are true. For example, the sentence

Bill likes dogs.

shows a relation between the objects *Bill* and *dogs*; the relation is *likes*. Here is a rule that defines when the sentence Bill likes dogs. is true:

Bill likes dogs **if** the dogs are nice.

Facts: What Is Known

In Prolog, a relation between objects is called a *predicate*.

Fact: Socrates is a man.

man(socrates).

Examples of facts expressing "likes" relations in natural language:

Bill likes Cindy.
Cindy likes Bill.
Bill likes dogs.

Here are the same facts, written in Prolog syntax:

```
likes(bill, cindy).
likes(cindy, bill).
likes(bill, dogs).
```

Facts can also express properties of objects as well as relations; in natural language "Kermit is green" and "Caitlin is a girl."

Example of Prolog facts that express the same properties:

```
green(kermit).
girl(caitlin).
```

Rules: What You Can Infer from Given Facts

Rules enable you to infer facts from other facts. Examples of some rules concerning a "likes" relation:

Cindy likes everything that Bill likes.
Caitlin likes everything that is green.

Given these rules, you can infer from the previous facts some of the things that Cindy and Caitlin like:

Cindy likes Cindy.
Caitlin likes Kermit.

To encode these same rules into Prolog, you only need to change the syntax a little, like this:

```
likes(cindy, Something):- likes(bill, Something).
likes(caitlin, Something):- green(Something).
```

Note: The :- symbol is simply pronounced "if", and serves to separate the two parts of a rule: the head and the body.

You can also think of a rule as a procedure. In other words, these rules

```
likes(cindy, Something):- likes(bill, Something)
likes(caitlin, Something):- green(Something).
```

Queries* (querying the Prolog system.*)

Once we give Prolog a set of facts, we can ask questions concerning these facts; this is known as *querying the Prolog system*.

In natural language, we ask you:

Does Bill like Cindy?

In Prolog syntax, we ask Prolog:

```
likes(bill, cindy).
```

Given this query, Prolog would answer

```
yes
```

Putting Facts, Rules, and Queries Together

Suppose you have the following facts and rules:

A fast car is fun.
A big car is nice.
A little car is practical.
Bill likes a car if the car is fun.

When you read these facts, you can deduce that Bill likes a fast car. In the same way, Prolog will come to the same conclusion.

Example demonstrating how Prolog uses rules to answer queries.

Program 1: Look at the *facts* and *rules* in this portion

```
likes(ellen, tennis).
likes(john, football).
likes(tom, baseball).
likes(eric, swimming).
likes(mark, tennis).
likes(bill, Activity):- likes(tom, Activity).
```

The last line in Program 1 is a rule:

```
likes(bill, Activity):- likes(tom, Activity).
```

This rule corresponds to the natural language statement

Bill likes an activity if Tom likes that activity.

In this rule, the head is likes(bill, Activity), and the body is likes(tom, Activity).

Notice that there is no fact in this example about Bill liking baseball. For Prolog to discover if Bill likes baseball, you can give the query

```
likes(bill, baseball).
```

When attempting to find a solution to this query, Prolog will use the rule:

```
likes(bill, Activity):- likes(tom, Activity).
```

Clauses.

There are only two types of phrases that make up the Prolog language; a phrase can be either a *fact* or a *rule*. These phrases are known in Prolog as *clauses*. The heart of a Prolog program is made up of clauses.

Program 1:

```
PREDICATES
    nondeterm likes(symbol,symbol)

CLAUSES
    likes(ellen,tennis).
    likes(john,football).
    likes(tom,baseball).
    likes(eric,swimming).
    likes(mark,tennis).
    likes(bill,Activity):-
    likes(tom, Activity).

GOAL
    likes(bill, baseball).
```

The system replies in the Dialog window

```
yes
```

It has combined the rule

```
likes(bill, Activity):- likes(tom, Activity).
```

with the fact

```
likes(tom, baseball).
```

to decide that

```
likes(bill, baseball).
```

Exercise Try the query:

```
likes(bill, tennis).
```

The system replies

```
no
```

Visual Prolog replies no to the latest query ("Does Bill like tennis?") because

i) There is no fact that says Bill likes tennis.

ii) Bill's relationship with tennis can't be inferred using the given rule and the available facts.

Overview

1. A Prolog program is made up of two types of phrases (also known as *clauses*): facts and rules.

   Facts are relations or properties that you, the programmer, know to be true.

   Rules are dependent relations; they allow Prolog to infer one piece of information from another. A rule becomes true if a given set of conditions is proven to be true. Each rule depends upon proving its conditions to be true.

2. In Prolog, all rules have two parts: a head and a body separated by the special :- symbol

   The *head* is the fact that would be true if some number of conditions were true. This is also known as the conclusion or the dependent relation.

   The *body* is the set of conditions that must be true so that Prolog can prove that the head of the rule is true.

3. Facts and rules are really the same, except that a fact has no explicit body. The fact simply behaves as if it had a body that was always true.

4. Once you give Prolog a set of facts and/or rules, you can proceed to ask questions concerning these; this is known as *querying the Prolog system*. Prolog always looks for a solution by starting at the top of the facts and/or rules, and keeps looking until it reaches the bottom.

5. Prolog's inference engine takes the conditions of a rule (the body of the rule) and looks through its list of known facts and rules, trying to satisfy the conditions. Once all the conditions have been met, the dependent relation (the head of the rule) is found to be true. If all the conditions can't be matched with known facts, the rule doesn't conclude anything.

Clauses (Facts and Rules) Examples

A fact represents one single instance of either a property of an object or a relation between objects.

A rule is a property or relation known to be true when some set of other relations is known. The relations are separated by commas,

Examples of Rules

Example shows a rule that can be used to conclude whether a menu item is suitable for Diane.

Diane is a vegetarian and eats only what her doctor tells her to eat.

Given a menu and the preceding rule, you can conclude if Diane can order a particular item on the menu.

To do this, you must check to see if the item on the menu matches the constraints given.

a. Is Food_on_menu a vegetable?

b. Is Food_on_menu on the doctor's list?

c. Conclusion: If both answers are yes, Diane can order Food_on_menu.

Prolog, a relationship represented by a rules

```
diane_can_eat(Food_on_menu):-
    vegetable(Food_on_menu),
    on_doctor_list(Food_on_menu).
```

Note comma after vegetable(Food_on_menu) introduces a conjunction of several goals, and is simply read as "and"; both vegetable(Food_on_menu) **and** on_doctor_list(Food_on_menu) must be true, for diane_can_eat(Food_on_menu) to be true.

Suppose you want to make a Prolog fact that is true if *Person1* is the parent of *Person2*. simply state the Prolog fact

```
parent(paul, samantha).
```

Means Paul is the parent of Samantha

Suppose you know that *Person1* is the parent of *Person2* if *Person1* is the father of *Person2* or if *Person1* is the mother of *Person2,*

After stating these conditions in natural language, it would be fairly simple to code this into a Prolog rule by writing a rule that states the relationships.

```
parent(Person1, Person2):- father(Person1, Person2).
parent(Person1, Person2):- mother(Person1, Person2).
```

These Prolog rules simply state that

Person1 is the parent of Person2 **if** Person1 is the father of Person2.
Person1 is the parent of Person2 **if** Person1 is the mother of Person2.

Example 2

A person can buy a car if the person likes the car and the car is for sale.

This natural language relationship can be conveyed in Prolog with the following rule:

```
can_buy(Name, Model):-
    person(Name),
    car(Model),
    likes(Name, Model),
for_sale(Model).
```

This rule shows the following relationship:

Name can_buy Model **if**
    Name is a person **and**
    Model is a car **and**
    Name likes Model **and**
    Model is for sale.

Note: This Prolog rule will succeed if all four conditions in the body of the rule succeed.

Program

Here is a program designed to find solutions to this car-buying problem:

```
/* Program ch00e02.pro */

PREDICATES
    nondeterm can_buy(symbol, symbol)
    nondeterm person(symbol)
    nondeterm car(symbol)
    likes(symbol, symbol)
    for_sale(symbol)

CLAUSES
    can_buy(X,Y):-
    person(X),
    car(Y),
    likes(X,Y),
    for_sale(Y).

    person(kelly).
    person(judy).
    person(ellen).
    person(mark).

    car(lemon).
    car(hot_rod).

    likes(kelly, hot_rod).
    likes(judy, pizza).
    likes(ellen, tennis).
    likes(mark, tennis).

    for_sale(pizza).
    for_sale(lemon).
    for_sale(hot_rod).
```

What can Judy and Kelly buy? Who can buy the hot rod? You can try the following goals:

```
can_buy(Who, What).
can_buy(judy, What).
can_buy(kelly, What).
can_buy(Who, hot_rod).
```

Experiment! Add other facts and maybe even a rule or two to this Prolog program. Test the new program with queries that you make up. Does Prolog respond in a way you would expect it to?

Exercises

Write natural-language sentences corresponding to the following Visual Prolog rules:

```
eats(Who, What):- food(What), likes(Who, What).

pass_class(Who):- did_homework(Who), good_attendance(Who).

does_not_eat(toby, Stuff):- food(Stuff), greasy(Stuff).

owns(Who, What):- bought(Who, What).
```

Write Visual Prolog rules that convey the meaning of these natural-language sentences:

a. A person is hungry if that person's stomach is empty.
b. Everybody likes a job if it's fun and it pays well.
c. Sally likes french fries if they're cooked.
d. Everybody owns a car who buys one, pays for it, and keeps it.

KNOWLEDGE BASED SYSTEMS AI & EXPERT SYSTEMS

A knowledge-based system is a computer program that reasons and uses a knowledge base to solve complex problems.
An *expert system* is a computer program that uses artificial intelligence (AI) technologies to simulate the judgment and behavior of a human expert.

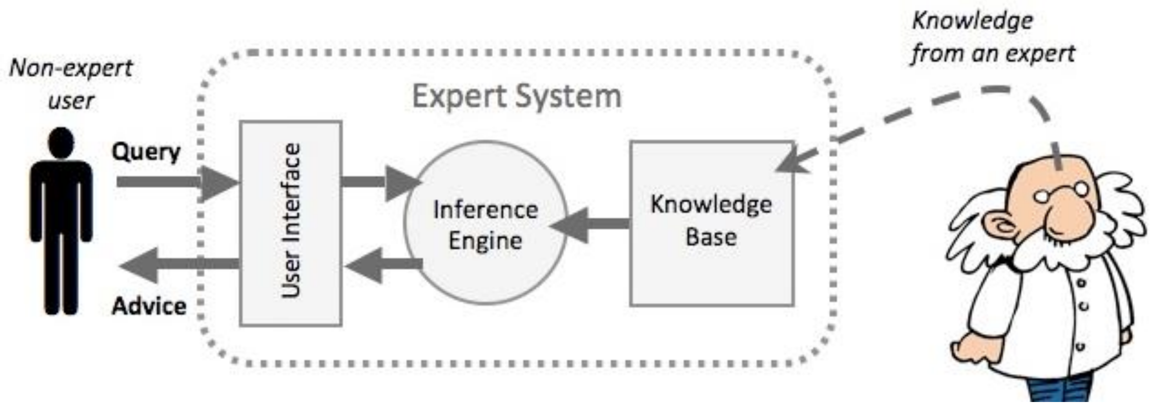
Components of a KBS

Consist of **user interface, Knowledge base and inference engine**

A **user interface** - This is the system that allows a **non-expert user** to **query** (question) the expert system, and to **receive advice**. The user-interface is designed to be a **simple** to use as possible.

A **knowledge base** - This is a **collection of facts and rules**. The knowledge base is created from **information** provided by **human experts**

An **inference engine** - This act like a **search engine**, examining the knowledge base for information that **matches** the user's **query**



The **non-expert user** queries the expert system. This is done by **asking a question**, or by **answering questions** asked by the expert system.

The **inference engine** uses the query to **search** the **knowledge base** and then provides an answer or some **advice** to the user.

Application of Expert Systems

i)Medical diagnosis (the knowledge base would contain medical information, the symptoms of the patient would be used as the query, and the advice would be a diagnose of the patient’s illness)
ii)Playing **strategy games** like **chess** against a computer (the knowledge base would contain strategies and moves, the player's moves would be used as the query, and the output would be the computer's 'expert' moves)
iii)Providing **financial advice** - whether to invest in a business, etc. (the knowledge base would contain data about the performance of financial markets and businesses in the past)
iv) Helping to **identify items** such as plants / animals / rocks / etc. (the knowledge base would contain characteristics of every item, the details of an unknown item would be used as the query, and the advice would be a likely identification)
v)Helping to **discover locations to drill for water / oil** (the knowledge base would contain characteristics of likely rock formations where oil / water could be found, the details of a particular location would be used as the query, and the advice would be the likelihood of finding oil / water there)
vi) Helping to **diagnose car engine problems** (like medical diagnosis, but for cars!)

KNOWLEDGE ACQUISITION, VALIDATION & REPRESENTATION

Knowledge representations schemes

a) Rules
b) Predicate Logic
c) Semantic nets
d) Frames
e) Decision Trees

Rules knowledge representation

Formalization often used to specify recommendations, give directives or strategy, make use of nested if statements

Example

Assume: Knowledge base consisting of facts and rules, a rule interpreter to match the rule conditions against facts and means for executing the rules.

Rules:

R1: IF: Raining,Outside(x),Has_Umbrella(x)

THEN: Uses_Umbrella(x)

R2: IF: Raining,Outside(x)

NOT Has_Umbrella(x)

THEN: Wet(x)

R3: IF: Wet(x)

THEN: Gets_Cold(x)

R4: IF: Sunny,Outside(x)

THEN: Gets_Sun_burn(x)

Initial facts: Raining, Outside(John)

<u>Correct:</u>

- Only one rule, R2 matches the facts with [x→John], hence add Wet(John)
- Facts after first cycle:
  Raining, Outside(John), Wet(John)
- Now R3 matches facts, hence add
  Gets_Cold(John)
- Facts after second cycle:
  Raining, Outside(John), Wet(John), Gets_Cold(John)

Predicate Logic / first order logic

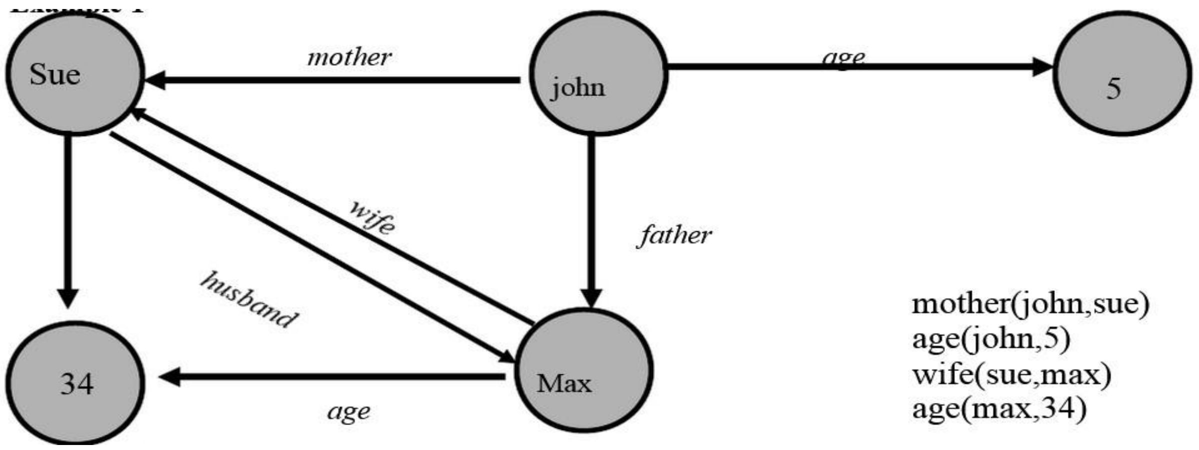
Predicate logic allows one to represent complex facts about the world
Predicate logic is a well understood formal language, with well-defined syntax, semantics and rules of inference.
Allows facts about the world to be represented as sentences formed from propositional symbols:

Semantic networks

Semantic networks, represent knowledge in the form of graphical networks.

Semantic nets store knowledge in the form of a graph, with nodes representing objects in the world, and arcs representing relationships between the objects.

A semantic network uses directed graphs to represent knowledge. A directed graph is made of vertices (nodes) and edges (arcs). Example.



mother(john,sue)
age(john,5)
wife(sue,max)
age(max,34)

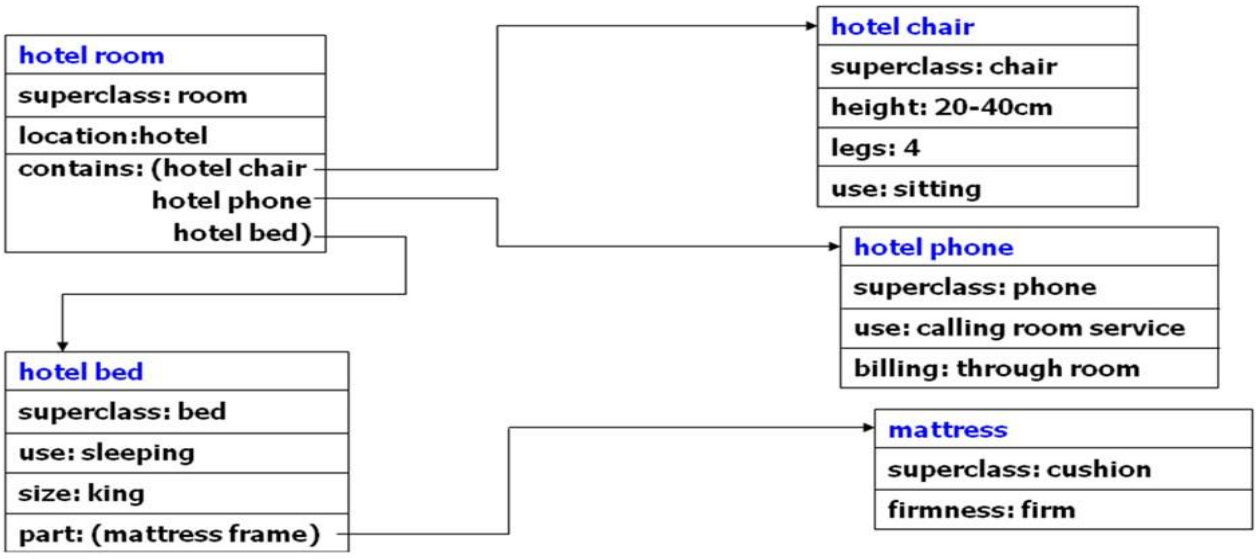
Frames

Frames knowledge representation technique used in AI to model concepts, objects, and their attributes

Example: Part of the Frame Description of a Hotel Room

Example

Part of the Frame Description of a Hotel Room



KNOWLEDGE ACQUISITION

Knowledge acquisition is the process of extracting knowledge (facts, procedures, rules) from human experts, books, documents, sensors or computer files and converting it into a form that can be stored and manipulated by the computer for purposes of problem solving.

Methods of knowledge elicitation

Face to face interview with experts – the experts are interviewed by knowledge engineers.

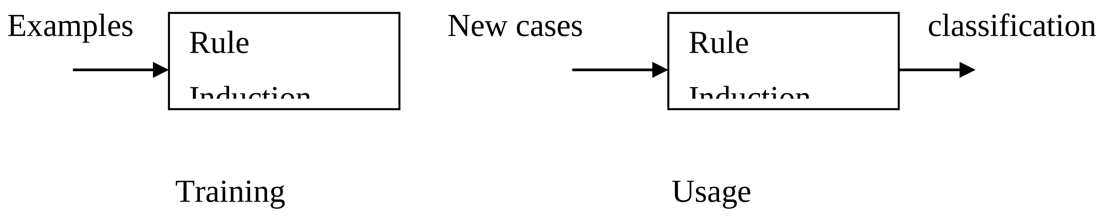
Protocol analysis – this is a documentation of how the expert behaves and process information during problem solving. Usually the experts think aloud.

Observation – the experts are observed at work.

Questionnaires – these are questions that are sent to experts for responses.

Analysis of documented knowledge – this is extraction of knowledge from sources such as books, journals, articles, magazines, mass media materials.

Rule induction (computer aided knowledge acquisition) – rule induction can be viewed as a system that accepts examples and develops classification rules.



INFERENCE

Inference is the process of drawing a conclusion from a given evidence. or arriving at a decision through reasoning.

Rule-based Reasoning

i. Forward-chaining
ii. Backward chaining.

Forward chaining.

A data-driven inference technique that starts with the available data and applies rules to infer new data until a goal is reached.

The forward-chaining algorithm starts from known facts, triggers all rules whose premises are satisfied and adds their conclusion to the known facts.

Algorithm for Forward Chaining

i)Start with Known Facts: The inference engine begins with the known facts in the knowledge base.

ii)Apply Rules: It looks for rules whose conditions are satisfied by the known facts.

iii)Infer New Facts: When a rule is applied, new facts are inferred and added to the knowledge base.

iv)Repeat: This process is repeated until no more rules can be applied or a specified goal is achieved.

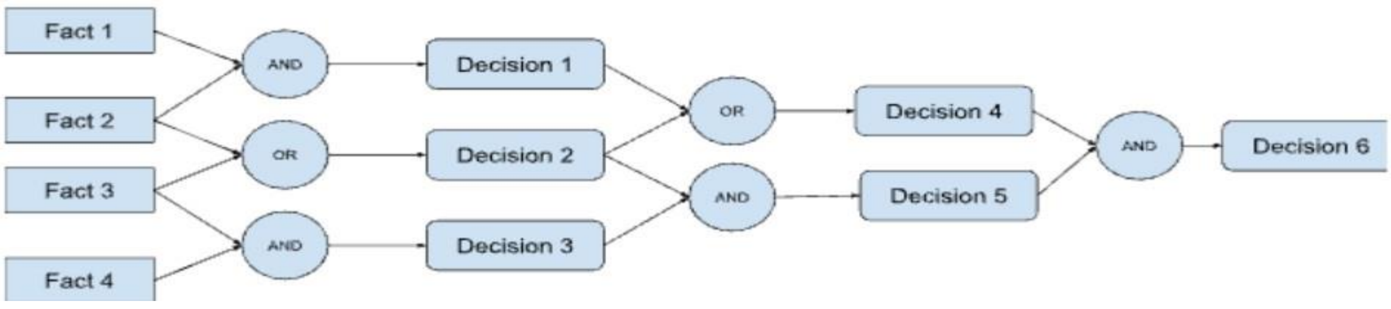
Example of Forward Chaining

Consider a medical diagnosis system where rules are used to diagnose diseases based on symptoms:

- **Fact**: The patient has a fever.
- **Rule**: If a patient has a fever and a rash, they might have measles.

Starting with the known fact (fever), the system checks for other symptoms (rash). If the patient also has a rash, the system infers the possibility of measles.

Illustration of forward chaining.



Backward Chaining

Backward chaining is a goal-driven inference technique that starts with the goal and works backward to determine which facts must be true to achieve that goal.

This method is ideal for situations where the goal is clearly defined, and the path to reach it needs to be established.

How Backward Chaining Works

i) Start with a Goal: The inference engine begins with the goal or hypothesis it wants to prove.

ii) Identify Rules: It looks for rules that could conclude the goal.

iii) Check Conditions: For each rule, it checks if the conditions are met, which may involve proving additional sub-goals.

4. **Recursive Process**: This process is recursive, working backward through the rule set until the initial facts are reached or the goal is deemed unattainable.

Example of Backward Chaining

In a troubleshooting system for network issues:

Goal: Determine why the network is down.

Rule: If the router is malfunctioning, the network will be down.

The system starts with the goal (network down) and works backward to check if the router is malfunctioning, verifying the necessary conditions to confirm the hypothesis.

IMPLEMENTING KBS & EXPERT SYSTEMS

CASE STUDY EXAMPLES - Simple Expert Systems

The expert systems given below are very basic, they should give you an on how to develop expert systems They can all be quickly and easily implemented using Crystal

CASE STUDY 1: A car trouble diagnostic system

Knowledge Acquisition

The first task is knowledge acquisition. The solutions for this expert system are based wholly on knowledge on automotive systems from the internet and a local Jua Kali mechanic.

The basic items that were identified to be needed in order to get a vehicle to start are a combustion chamber, some sort of mechanism to turn the engine, air and fuel to burn, and something to ignite the air fuel mixture. All the solutions in this illustration deal with how these elements come together in order to make a vehicle start. Below is an introduction to the basic systems to be considered.

Battery: This is the part of a vehicle that stores the power that is required to turn the engine and create a spark.

Battery Cables: This is a set of wires that carry the power from the battery to the starter and the rest of the engine. These cables usually fail due to corrosion, which interferes with the energy flow from the battery.

Starter: This is a mechanical device, an electric motor that uses power from the battery to rotate the engine.

Coil: An electronic component that takes the twelve volts coming from the battery and converts it to a much larger voltage.

Coil Wire: A wire which caries the voltage from the coil to the distribution or computer controlled ignition points, which then distribute the pulse to the correct spark plug wire.

Spark Plug Wires: A set of wires that caries the electronic pulse from the distributor or ignition points to the appropriate spark plug.

Spark Plugs: A set of electronic components constructed of insulators and conductors. These spark plugs create a short between a spark point and a conductor. This short creates a spark that ignites the air fuel mixture.

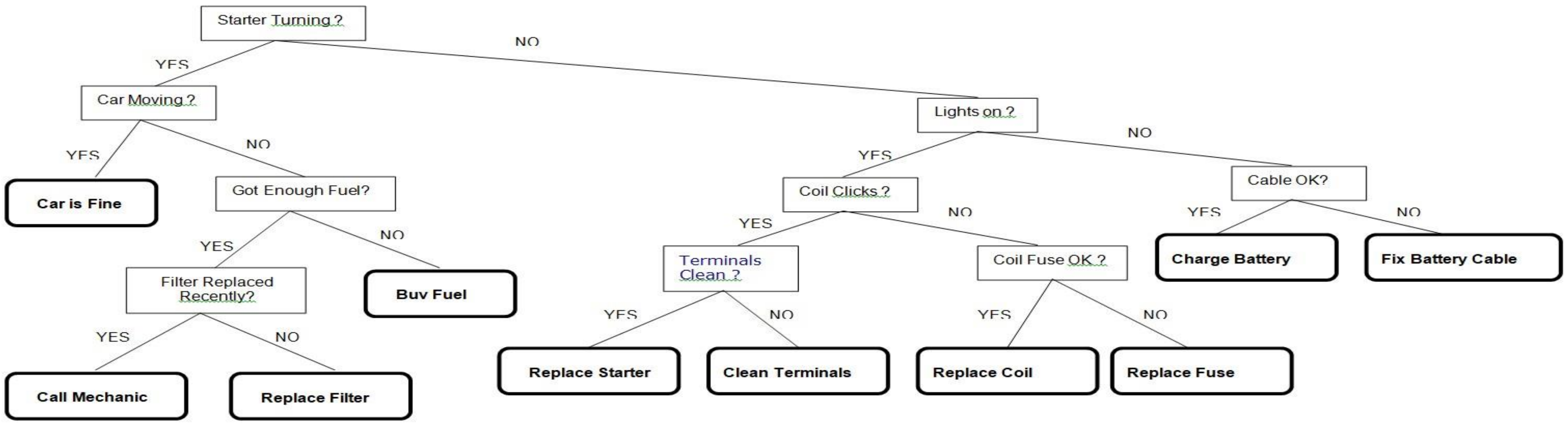
Fuel: Also referred to as petrol.

Fuel Filter: A filtering device located somewhere between the fuel tank and engine. Used to eliminate impurities from the fuel.

Knowledge Representation

The Second step is to represent the acquired knowledge. This involved coming up with rules that would later be encoded into the knowledge base of the expert system. The hard part is to decide at what point should problems be included in the solution space and which should be dropped. A decision to be made is to limit the solution space to problems that can be fixed without any special knowledge of how a car works.

Below, is a decision tree used to represent the reasoning used in the system.



Solution to the car trouble diagnostic expert system

Rule to infe, sSuppose that we have the following rules:

```
1. IF engine_getting_petrol
   AND engine_turns_over
   THEN problem_with_spark_plugs
2. IF NOT engine_turns_over
   AND NOT lights_come_on
   THEN problem_with_battery
3. IF NOT engine_turns_over
   AND lights_come_on
   THEN problem_with_starter
4. IF petrol_in_fuel_tank
   THEN engine_getting_petrol
```

Our problem is to work out what's wrong with our car given some observable symptoms. There are three possible problems with the car: problem_with_spark_plugs, problem_with_battery, problem_with_starter. We'll assume that we have been provided with no initial facts about the observable symptoms.

In the simplest goal-directed system we would try to prove each hypothesised problem (with the car) in turn. First the system would try to prove "problem_with_spark_plugs". Rule 1 is potentially useful, so the system would set the new goals of proving "engine_getting_petrol" and "engine_turns_over". Trying to prove the first of these, rule 4 can be used, with new goal of proving "petrol_in_fuel_tank" There are no rules which conclude this (and the system doesn't already know the answer), so the system will ask the user:

```
Is it true that there's petrol in the fuel tank?
```

Let's say that the answer is yes. This answer would be recorded, so that the user doesn't get asked the same question again. Anyway, the system now has proved that the engine is getting petrol, so now wants to find out if the engine turns over. As the system doesn't yet know whether this is the case, and as there are no rules which conclude this, the user will be asked:

```
Is it true that the engine turns over?
```

Lets say this time the answer is no. There are no other rules which can be used to prove "problem_with_spark_plugs" so the system will conclude that this is not the solution to the problem, and will consider the next hypothesis: problem_with_battery. It is true that the engine does not turn over (the user has just said that), so all it has to prove is that the lights don't come one. It will ask the user

```
Is it true that the lights come on?
```

Suppose the answer is no. It has now proved that the problem is with the battery. Some systems might stop there, but usually there might be more than one solution, (e.g., more than one fault with the car), or it will be uncertain which of various solutions is the right one. So usually all hypotheses are considered. It will try to prove "problem_with_starter", but given the existing data (the lights come on) the proof will fail, so the system will conclude that the problem is with the battery. A complete interaction with our very simple system might be:

```
System: Is it true that there's petrol in the fuel tank?
User: Yes.
System: Is it true that the engine turns over?
User: No.
System Is it true that the lights come on?
User: No.
System: I conclude that there is a problem with battery.
```

Note that in general, solving problems using backward chaining involves *searching* through all the possible ways of proving the hypothesis, systematically checking each of them. A common way of doing this search is the same as in Prolog (programming language) - depth first search with backtracking.

SIMPLE EXPERT SYSTEM 2: A MEDICAL DIAGNOSIS SYSTEM

Knowledge Acquisition Process

For this process, two medical practitioners – one Doctor working for an organization, and one in the private sector here in Mombasa - with a view to finding out: (1) Illnesses that have nearly similar signs and symptoms to Malaria, and (2) The signs and symptoms for each of these illnesses. The interviews were face-to-face. Other information was obtained from the internet.

The ailments covered are listed below, and the respective signs and symptoms also given.

	Malaria	**Malaria + RTI**	**RTI**	**Typhoid**	**Meningitis**
Fever	✓	✓	✓	✓	✓
Chills (and sweating)	✓	✓			
Coughing		✓	✓		
Headache	✓	✓		✓	✓
Severe Headache				✓	✓
Nausea (and vomiting)	✓	✓			✓
Body Malaise	✓	✓		✓	
Abdominal discomfort				✓	
Diarrhoea/Constipation				✓	

Loss of Appetite				✓	
Stiff neck					✓
Photophobia (sensitive to light)					✓

Note: RTI = Respiratory Track Infection (specifically the common cold, also referred to as the upper respiratory infection).

Note: Cells shaded in green indicate YES for the given sign/symptom.

Rule-based Knowledge Representation

R1	IF fever THEN **patient_ill**
R2	IF patient_ill AND coughing THEN **respiratory_tract_infection**
R3	IF patient_ill AND headache AND chills_sweat AND nausea AND body_malaise THEN [**malaria**]
R4	IF patient_ill AND respiratory_track_infection AND NOT malaria THEN [**common_cold**]
R5	IF malaria AND respiratory_track_infection THEN [**malaria_and_respiratory_tract**]
R6	IF patient_ill AND NOT chills_sweat AND headache AND severe_headache THEN **non_malaria**
R7	IF non_malaria AND nausea AND stiff_neck AND photophobia THEN [**meningitis**]
R8	IF non_malaria AND body_malaise AND diarrhoea_constipation AND appetite_loss AND NOT stiff_neck AND NOT photophobia THEN [**typhoid**]
R9	IF patient_ill AND headache AND severe_headache AND body_malaise AND abdominal_discomfort AND stiff_neck THEN [**unknown_illness_1**]
R10	IF patient_ill AND headache AND NOT severe_headache AND body_malaise AND NOT nausea THEN [**unknown_illness_2**]

R11	IF patient_ill AND headache AND NOT severe_headache AND NOT body_malaise THEN [**unknown_illness_3**]
R12	IF patient_ill AND NOT headache AND NOT common_cold THEN [**unknown_illness_4**]
R13	IF patient_ill AND headache AND severe_headache AND NOT body_malaise AND nausea AND NOT stiff_neck THEN [**unknown_illness_5**]
R14	IF patient_ill AND headache AND severe_headache AND NOT body_malaise AND NOT nausea THEN [**unknown_illness_6**]