

Teacher/Lecturer Notes

Aim

This unit is designed to develop knowledge and understanding of knowledge domains, search methods and expert system shells. Problem-solving skills, in the context of artificial intelligence, are extended as are practical abilities in the use of specialised software, in particular expert system shells.

Status of the Learning and Teaching Pack

These materials are for guidance only. The mandatory content of this unit is detailed in the unit specification of the Arrangements for Advanced Higher Computing.

Target Audience

While entry is at the discretion of the centre, students would normally be expected to have attained one of the following (or possess equivalent experience):

- Artificial Intelligence (H) unit
- Computing course at Higher level
- Higher Grade Computing Studies

Pre-knowledge and Skills

As well as the computing knowledge and skills that would be derived from one of the above courses of study, it is expected that students will have the ability to make use of on-line help facilities, software documentation and manuals. Students should also have developed independent study skills.

Progression

This unit builds on the introduction to artificial intelligence, knowledge representation and processing in the optional unit Artificial Intelligence at Higher level in Computing. The knowledge and skills acquired in this unit may provide the basis for a project for the Advanced Higher course.

Hardware and Software Requirements

Each student will need access for approximately 15 to 20 hours to a system capable of running a declarative language and expert system tools. The student notes refer to implementations in Prolog and the Appendices to Chapter 2 specifically use exercises and sample programs written in Prolog. The sample program listings can be found in Appendix 2 of these Teacher/Lecturer Notes. If you wish students to use a different declarative language, these exercises will first need to be translated into the chosen language. The choice of expert system shells has been left to the discretion of the centre.

There are many versions of Prolog available. Demonstration versions are often available either on the Internet or direct from the manufacturer. These allow limited use of the language and usually do not allow the user to save object code. Details of sources for versions of Prolog and possible expert system shells are listed in Appendix 1 of these Teacher/Lecturer Notes.

Learning and Teaching Approaches

The materials in this pack are intended for student use with limited teacher or lecturer input. It has been recognised that students undertaking Advanced Higher courses are young adults who will soon be moving into or are already in Further or Higher Education. As a consequence of studying at Advanced Higher, students will develop personal study and learning skills, in addition to knowledge and understanding of the course content.

The materials are therefore in the form of a supported self-study pack with Self Assessment Questions embedded in the text of each of the three chapters, and answers to these Self Assessment Questions at the end of each chapter.

The teacher or lecturer will be able to set the context for the topics to be studied and to provide clarification where the student is having difficulty grasping an idea.

The use of Prolog or another declarative language is not mandatory. However, as suggested in the support notes section of the National Unit Specification for Computing, Prolog has been used to enhance much of the work for Outcome 2, by making an explicit link between the processes related to problem solving and possible computer-based solutions.

Pathway Through the Unit

It is suggested that the materials are used in chapter order as each chapter matches the equivalent Outcome in the Arrangements. Some of the practical work concerned with expert system shells, especially becoming familiar with the use of each shell, could be introduced while covering the more theoretical aspects of Chapters 1 and 2.

References

There is no single text that covers all the content of the three Outcomes for this unit but it would be useful to have available a number of texts for student reference. The most useful are listed in Appendix 1.

In addition, there are a number of Internet sites, often linked to universities, that provide material not only relevant to the content of this unit but also that might be found useful by a student doing a project based upon artificial intelligence. A few of these are listed in Appendix 1.

Appendix 1

Books, Software Resources and Web Sites

The following books are relevant to the content of this unit:

Title	Author	Publisher	ISBN
<i>Artificial Intelligence</i>	Elaine Rich and Kevin Knight	McGraw Hill	0071008942
<i>Prolog – Programming for Artificial Intelligence</i>	Ivan Bratko	Addison-Wesley	0201416069
<i>An Introduction to Artificial Intelligence</i>	Janet Finlay and Alan Dix	UCL Press	1857283996
<i>Expert Systems – Concepts and Examples</i>	J L Alty and M J Coombs	NCC Publications	0850123992
<i>Introduction to Programming in Prolog</i>	Danny Crookes	Prentice Hall	0137101465

The following books also provide some coverage of the content of this unit:

Title	Author	Publisher	ISBN
<i>Expert Systems – Design and Development</i>	John Durkin	Macmillan	0023309709
<i>Introduction to Artificial Intelligence & Expert Systems</i>	Dan W Patterson	Prentice Hall	0134771001
<i>A Guide to Expert Systems</i>	Donald A Waterman	Addison-Wesley	0201083132
<i>Expert Systems – Principles and Programming</i> [includes CD-ROM of Clips]	Joseph Giarratano and Gary Riley	PWS Publishing Company	0534950531

<i>Artificial Intelligence and the Design of Expert Systems</i>	George F Luger and William A Stubblefield	Benjamin/Cummings	0805301399
<i>Introduction to Expert Systems</i>	Peter Jackson	Addison-Wesley	0201142236
<i>Artificial Intelligence – A Modern Approach</i>	Stuart Russell and Peter Norvig	Prentice Hall	0133601242
<i>A Prolog Primer</i>	Jean B Rogers	Addison-Wesley	0201064677
<i>Programming in Prolog</i>	W Clocksin and C Mellish	Springer Verlag	0387175393

The following software is available:

Software	Source	Comment
LPA Prolog Flex Flint LPA MacProlog	Logic Programming Associates Ltd, Studio 4, Royal Victoria Patriotic Building, Trinity Road, London SW18 3SX www.lpa.co.uk	A Windows version of Prolog that can have the Flex expert system shell added to it. Flint adds uncertainty to Flex. The company also do a version for Apple Mac.
Inter•Modeller	Parallel Logic Programming Ltd 99 Cooks Close, Bristol BS32 0BB www.parlog.com	Available for Mac and Windows, this allows various representation methods to be used. The replacement for Primex.
Prolog for Windows (SWI)	www.cse.unsw.edu.au/~timm/pub/lang/prolog	A public domain version of Prolog.
CLIPS	www.ghg.net/clips/download/executables/examples/ www.ghg.net/clips/download/documentation http://web.ukonline.co.uk/julian.smart/wxclips/	An artificial intelligence programming language. The manuals are not very helpful but Giarratano and Riley (see above) contains a CD-ROM of the language and a very much better introduction.

There are a large number of Internet sites that contain material on artificial intelligence. Unfortunately, a number seem to change address regularly but, by using a search engine, new sites can be identified. The following have been found useful:

Address	Comment
http://ai.iit.nrc.ca/subjects/ai_subjects.html	Canadian National Research Council articles on most aspects of artificial intelligence.
www.cs.berkeley.edu/~russell/ai.html	Berkeley site on artificial intelligence.
http://home.clara.net/ml/ai/	A site that it is being developed with artificial intelligence articles.
www.ai.sri.com/~leclerc/	Useful for information on vision systems.
www.activmedia.com/robots/robocup.html	Information on a robot competition.
www.cs.cmu.edu/	Contains information on fuzzy logic systems.
www.henneman.com/ www.kbs.ai.uiuc.edu/ www.aiinc.ca/applications/applications.html	Various sites with information on expert systems.
www.yahoo.co.uk/science/computer_science/artificial_intelligence	General site for AI.

Appendix 2

Listings of LPA Prolog Programs

routes.pl

```

route :- retractall(gone_along(_)), write('From '), read(X), write('To '), read(Y),
        can_get_to(X, Y).
towns([dundee, perth, arbroath, montrose, forfar, carnoustie, brechin,
kirriemuir, crieff, dunkeld, pitlochry, blairgowrie, glamis, petterden,
monifieth, muirdrum, ballinluig]).
is_town(X) :- towns(Y), in(X, Y).
in(X, [X|_]).
in(X, [_|_]) :- in(X, Y).

on(dundee, [a90, a930, a92]).
on(monifieth, [a930]).
on(carnoustie, [a930]).
on(muirdrum, [a930, a92]).
on(arbroath, [a92, a933]).
on(montrose, [a92, a935]).
on(brechin, [a935, a933, a90]).
on(forfar, [a94, a90, a926]).
on(petterden, [a90, a928]).
on(glamis, [a928, a94]).
on(kirriemuir, [a926, a928]).
on(blairgowrie, [a926, a924, a93, a923]).
on(perth, [a90, a85, a9, a93, a94]).
on(dunkeld, [a9, a822, a923]).
on(pitlochry, [a9, a924]).
on(ballinluig, [a9]).
on(crieff, [a85, a822]).

connect(X,Y) :- is_town(X), is_town(Y), X\==Y, on(X, X1), on(Y, Y1), in(Z1, X1),
in(Z1, Y1), not(gone_along(Z1)), assert(gone_along(Z1)),
printout([Z1, ' ', X, ' ', Y]), nl.
can_get_to(X, X) :- printout(['Do not be silly.']), nl.
can_get_to(X, Y) :- connect(X, Y).
can_get_to(X, Y) :- connect(X, X1), can_get_to(X1, Y).
printout([]).
printout([X|Y]) :- write(X), printout(Y).

```

chemical.pl

```

has_elements(water, [hydrogen, oxygen]).
has_elements(ammonia, [nitrogen, hydrogen]).
has_elements(methane, [carbon, hydrogen]).
has_elements(carbon_dioxide, [carbon, oxygen]).
has_elements(alcohol, [carbon, hydrogen, oxygen]).

```

hanoi.pl

```

/*
Tower of Hanoi - program for Higher AI
*/
hanoi(N) :- move(N, left, middle, right).
move(1, A, _, C) :- inform(A, C), !.
move(N, A, B, C) :- N1 is N-1, move(N1, A, C, B),
inform(A, C), move(N1, B, A, C).
inform(Loc1, Loc2) :- write('Move a disk from '),
write(Loc1), write(' to '), write(Loc2), nl.

```

supplies.pl

feeds(station, t1).	generator(station).
feeds(station, t2).	transformer(t1).
feeds(station, t3).	transformer(t2).
feeds(t1, t4).	transformer(t3).
feeds(t1, t5).	transformer(t4).
feeds(t3, t6).	transformer(t5).
feeds(t4, c1).	transformer(t6).
feeds(t4, c2).	consumer(c1).
feeds(t4, c3).	consumer(c2).
feeds(t5, c4).	consumer(c3).
feeds(t5, c5).	consumer(c4).
feeds(t2, c6).	consumer(c5).
feeds(t2, c7).	consumer(c6).
feeds(t6, c8).	consumer(c7).
feeds(t6, c9).	consumer(c8).
feeds(t3, c10).	consumer(c9).
	consumer(c10).

company.pl

```
works_for(finance_manager, director).
works_for(administration_manager, director).
works_for(production_manager, director).
works_for(finance_officer, finance_manager).
works_for(clerk, finance_officer).
works_for(cashier, finance_officer).
works_for(receptionist, administration_manager).
works_for(typist, administration_manager).
works_for(telephonist, administration_manager).
works_for(sales_officer, production_manager).
works_for(stores_manager, production_manager).
works_for(publicity_officer, production_manager).
```

credit.pl

```
has_credit_card(chris, visa).
has_credit_card(chris, diners).
has_credit_card(joe, shell).
has_credit_card(sam, mastercard).
has_credit_card(sam, citibank).
account_empty(chris, diners).
account_empty(sam, mastercard).
account_empty(chris, visa).
likes_shopping(P) :- has_credit_card(P,C),
not (account_empty(P,C)), write(P),
write(' can shop with the '), write(C),
write(' credit card. '), nl.
```

hobbies.pl

```
is_liked_by([golf,chess,squash], jim).
is_liked_by([rugby,football,watching_tv], bill).
is_liked_by([tennis,chess,golf], george).
is_liked_by([squash,tennis,rugby], david).
is_liked_by([golf,squash,football], alex).
is_liked_by([football,snooker,tennis], mike).
is_liked_by([snooker,watching_tv,golf], tom).
is_liked_by([chess,rugby,squash], john).
```

```
likes(Boy,Hobby) :- is_liked_by(X, Boy),
member_of(Hobby, X).
member_of(H, [H|_]).
member_of(H, [_|T]) :- member_of(H, T).
```

family.pl

```

male(andrew).
male(colin).
male(graham).
male(edward).
male(kenneth).
male(iain).
male(james).
male(michael).
male(quintin).
female(barbara).
female(fiona).
female(deborah).
female(helen).
female(lesley).
female(natalie).
female(ophelia).
female(patricia).
father(andrew, colin).
father(andrew, deborah).
father(andrew, edward).
father(colin, helen).
father(colin, iain).
father(graham, james).
father(kenneth, michael).
father(kenneth, natalie).
father(kenneth, ophelia).
father(james, patricia).
father(james, quintin).
mother(barbara, colin).
mother(barbara, deborah).
mother(barbara, edward).
mother(fiona, helen).
mother(fiona, iain).
mother(deborah, james).
mother(helen, michael).
mother(helen, natalie).
mother(helen, ophelia).
mother(lesley, patricia).
mother(lesley, quintin).

```

queens.pl

```

/*
  Program to solve N queens on a chessboard.
                          Advanced Higher Artificial Intelligence
*/
nqueens(N) :- makelist(N, L), Diagonal is N*2-1 , makelist(Diagonal, LL),
              placeN(N, board([], L, L, LL, LL), Final), write(Final), nl.
placeN(_, board(D, [], [], D1, D2), board(D, [], [], D1, D2)) :- !.
placeN(N, Board1, Result) :- place_a_queen(N, Board1, Board2) ,
                             placeN(N, Board2, Result).
place_a_queen(N, board(Queens, Rows, Columns, Diag1, Diag2),
              board([q(R, C)|Queens], NewR, NewC, NewD1, NewD2)) :-
  nextrow(R, Rows, NewR), findandremove(C, Columns, NewC), D1 is N+C-R,
  findandremove(D1, Diag1, NewD1), D2 is R+C-1,
  findandremove(D2, Diag2, NewD2).
findandremove(X, [X|Rest], Rest).
findandremove(X, [Y|Rest], [Y|Tail]) :- findandremove(X, Rest, Tail).
makelist(1, [1]).
makelist(N, [N|Rest]) :- N1 is N-1, makelist(N1, Rest).

nextrow(Row, [Row|Rest], Rest).

```

Student Notes

Introduction

This unit is designed to develop your knowledge and understanding of knowledge domains, search methods and expert system shells. There are three outcomes in this unit:

1. Analyse the processes related to problem solving in artificial intelligence.
2. Analyse the representation of knowledge domains and methods of searching these.
3. Critically appraise features of expert system shells.

What Will I Learn?

By completing this unit, you will increase your knowledge of the processes and methods used in artificial intelligence and you will be able to apply these to new problem areas. You will also increase your practical skills in the use of expert system shells and be able to identify the characteristics and features of different shells.

The Study Materials

The study materials have been written as three chapters corresponding to each of the outcomes.

In Chapter 1, you will consider a number of game and problem situations and analyse the methods used to solve these. You will consider the appropriateness of these methods and evaluate your solutions.

In Chapter 2, you will look at four methods for representing knowledge domains and a variety of search techniques that are used to match the knowledge and conditions to solve problems. You will also consider the use of these methods in three of the principal applications of artificial intelligence – natural language processing, computer vision and robotics.

In Chapter 3, you will study what is currently the most successful application of artificial intelligence, namely, expert systems. Through applying two different expert system shells to a range of problems, you will learn the determining characteristics of shells that support uncertainty and those that do not.

Instead of being taught the content of this unit, you will be expected to work through the materials with only some assistance from your teacher or lecturer. Throughout the three chapters, there are Self Assessment Questions. These have been included as a substitute for the questions that a teacher might have asked you to determine whether you understood. Their purpose is to show you whether or not you understand the work being covered.

The Self Assessment Questions are numbered sequentially through each chapter and answers are given at the end of each chapter. When you reach Self Assessment Questions, you should write out your answers, referring to your notes if necessary and then check your answers. There's no point in cheating as you're only kidding yourself!

If you answer the Self Assessment Questions successfully then move ahead, perhaps making notes of any problem areas that you had.

If you find that your answers differ from those included at the end of the chapter then you should return to the start of the material that you've just covered and find out at which point you first failed to understand. You then have to decide on the remedial action to take – revise earlier work; consult alternative textbooks or materials; consult your teacher or lecturer. Of course, you may decide to do all three. Aim for 100% understanding every time.

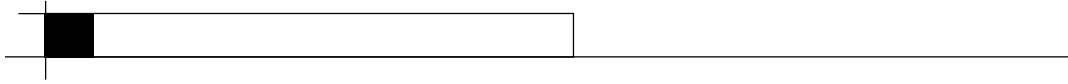
How Will I Be Assessed?

There are three assessments.

The first assessment is to produce a solution to a problem using the knowledge representation methods and search techniques of Outcomes 1 and 2. Hence you will take this assessment after you have covered the work of these two outcomes.

The second assessment is a written test on the theoretical knowledge of Outcome 2.

The third and final assessment will require you to implement the solution to a problem using two expert system shells. One version should be implemented using an expert system shell that supports uncertainty and the other using a shell that does not. You will then critically describe the two expert system shells and also compare your two implementations using specific criteria.



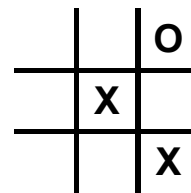
Introduction

One of the first areas of artificial intelligence to be explored by computer scientists was games playing. Before computer scientists could begin to program a computer to play the game 'intelligently', they had to analyse how problem situations were solved, how strategies to win were devised and how both the game situation and the strategy could be represented.

A starting point was an analysis of their own human approach to the game.

Have you ever stopped to think how you solve problems or play games?

What is your strategy when playing noughts and crosses? Similarly, think about strategies for playing other games such as chess and draughts or for solving puzzles – including jigsaw puzzles.

**Self Assessment Questions**

1. For the game of noughts and crosses (sometimes called tic-tac-toe), consider the answers to each of the following questions:
 - i) Do you prefer to play first or second?
 - ii) If you are playing first where do you play? Why?
 - iii) How do you decide where to place a nought or a cross?
 - iv) Is your first priority to obtain a winning row yourself or is it to stop your opponent getting a row?
2. What are your strategies for solving jigsaw puzzles? You should try to think of at least five strategies.

Simple Problems

We are going to analyse the ways in which strategies can be developed for solving simple problems and the ways of representing the processes in solving the problem when:

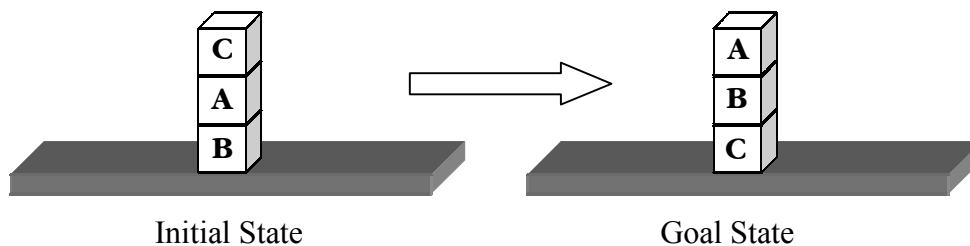
- making moves in a board game;
- solving geometric pattern matching problems;
- solving brainteasers;
- making moves in moving block problems.

The reason that we look at these apparently rather simple situations is that we can quickly grasp the problem idea and so we are able to concentrate on the methods that are being applied. Even so some of the problems are more complex than they appear at first glance.

State Space

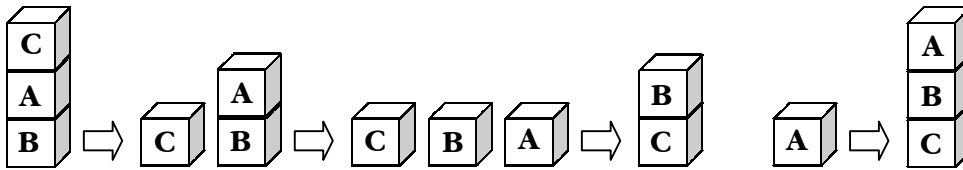
The first formal method of representing the processes in solving problems that we will consider is called a **state space**. The state space describes the initial state of the problem and the legal changes that produce new states. Let's look at an example and see how this works.

We have three blocks A, B and C in a stack and we require to rearrange them to produce a particular stack, e.g.

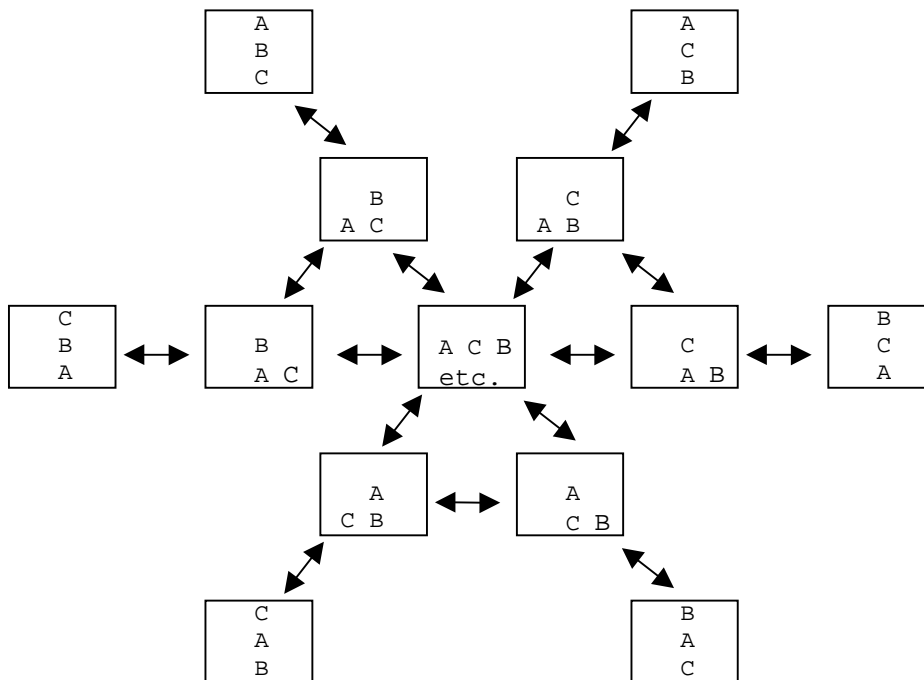


We can only move one block at a time by placing it on the table or on top of another block, e.g. we could place block C on the table and then place A on top of C.

It should be clear that we can place the top block on the table then place the second block on the table so forming three stacks each of one block and then assemble a stack in whatever order we require. This gives rise to the following simple solution for the arrangement above:



This can be extended to cover all the possible stacks of three blocks (there are six possible orders of the three blocks). [Note: the order in the centre is only one of the orders that can occur when there are three stacks each with one block.]



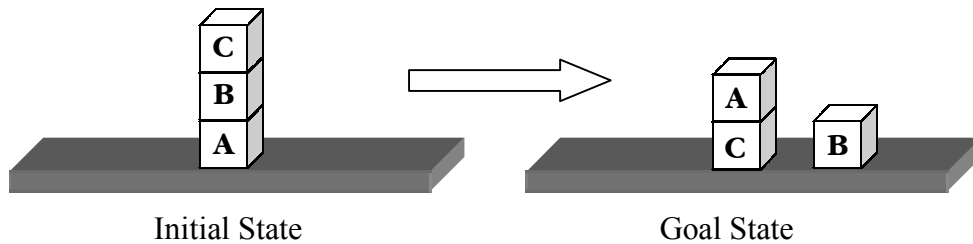
Notice that there are shortcuts and that, when we move from one arrangement to another, it is not always necessary to go through the stage of three separate stacks.

This is an example of **symbolic representation**. The nodes represent the different problem states while the connecting arrows represent valid transitions between states.

The whole process of describing the initial state, the goal state and constraints on the system is called **problem abstraction**. A constraint in the blocks problem is that a block cannot be moved from underneath another block.

Self Assessment Question

- Using the diagram on the previous page show the stages in achieving:



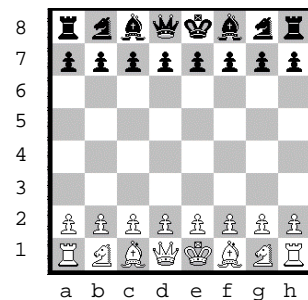
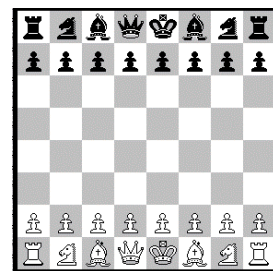
Chess Representation

There are limitations to the usefulness of symbolic representation.

Consider the game of chess. Every state could be represented by a chess board with the pieces shown. In this case it is estimated that we would need 10^{120} diagrams! Clearly neither humans nor computers could handle this number of different positions. A situation where a vast number of possible combinations has to be considered is termed a **combinatorial explosion**.

If you don't know how to play chess then you should familiarise yourself with the names of the pieces and some of the simple rules.

One solution is to describe possible moves and the constraints upon them by a system of rules.



For example, the white pawn on a2 can advance to a4 if both a3 and a4 are empty and this could be expressed by the rule:

white pawn a2 to a4 if a3 empty and a4 empty

We start with the representation of the initial state and generate new states by applying the rules. This representation method is appropriate for chess and similar situations where the possibilities are well structured and obey strict rules.

Stages in Problem Solving

There are four stages in designing the solution to a problem like those above:

1. Produce a precise problem description.
2. Analyse the problem.
3. Isolate and represent the knowledge that is necessary to solve the problem.
4. Select and apply the best problem solving method.

Self Assessment Questions

4. For the game of chess discussed above, describe how you could employ the four stages of problem solving.
5. You have a 4 litre jug and a 3 litre jug (both have no measuring marks) and a supply of water. How can you measure exactly 2 litres of water into the 4 litre jug? (You are allowed to throw water away.)

Write down your own solution to this water jug problem.

Test and evaluate your solution by asking questions such as:

Does it really work?

How good is this solution?

Is there an alternative way that this could have been solved?

Would less water have been wasted using an alternative method?

Water Jugs Problem

The 'water jugs problem' is one of the classic puzzles of artificial intelligence and it can be analysed by a variety of representation methods providing a useful comparison between them. The two jugs can be represented by an ordered pair (a,b) where a = 0, 1, 2, 3 or 4, the number of litres in the larger jug, while b = 0, 1, 2 or 3, the number of litres in the smaller jug. Hence our initial state is (0,0) and the goal state is (2,0).

(i) ***Production Rules***

Let us first look at a set of **production rules**. Each of these rules describes a possible change that we can make to the contents of the two jugs.

	<i>Condition</i>	<i>New State</i>	<i>Description</i>
1	(a,b) if $a < 4$	(4,b)	Fill the 4l jug
2	(a,b) if $b < 3$	(a,3)	Fill the 3l jug
3	(a,b) if $a > 0$	(0,b)	Empty the 4l jug on the ground
4	(a,b) if $b > 0$	(a,0)	Empty the 3l jug on the ground
5	(a,b) if $a+b \geq 4$ and $b > 0$	(4,b-4+a)	Pour water from the 3l jug to fill the 4l jug
6	(a,b) if $a+b \geq 3$ and $a > 0$	(a-3+b,3)	Pour water from the 4l jug to fill the 3l jug
7	(a,b) if $a+b \leq 4$ and $b > 0$	(a+b,0)	Empty the 3l jug into the 4l jug
8	(a,b) if $a+b \leq 3$ and $a > 0$	(0,a+b)	Empty the 4l jug into the 3l jug

You may have to try some values in these rules to understand them, especially rules 5 and 6.

Often in artificial intelligence it is easy to lose sight of the fact that we are considering situations with a view to computerising the process. It should be obvious that it will be straightforward to represent these production rules by program statements of the form:

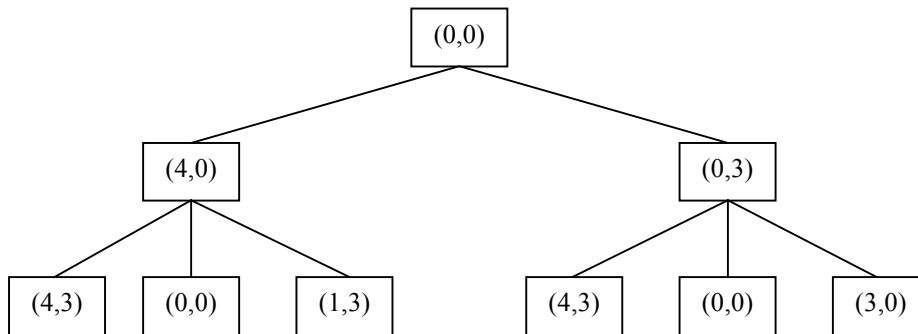
if $a < 4$ then $a = 4$

Self Assessment Question

6. For the solution given in the answer to Self Assessment Question 5, list the rules from the table above that apply at each stage of the solution.

(ii) *Search Tree*

Another representation that we could use is a **search tree** where at each level we list all the possible states that can be produced from the state at the level above. These branches are generated by applying the production rules above. Here is the start of the search tree for the water jugs problem:



Self Assessment Question

7. Copy the search tree above and add another level.

What are the problems in continuing to add levels to this tree?

What else do you notice about the entries in the tree?

Take one branch of the tree (you should recognise which one to use) and extend it so that you obtain the goal state that we require with 2 litres in the larger jug.

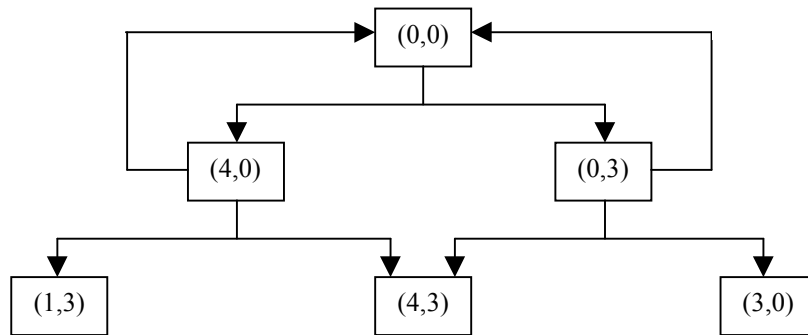
(iii) *Search Graphs*

Self Assessment Question 7 should have shown you some of the problems with search trees, namely:

- the tree becomes very large very quickly and most of it need never be searched for the goal state;
- nodes on the tree are repeated on each path by simply reversing the previous change, e.g. $(0,0) \rightarrow (4,0) \rightarrow (0,0)$;
- nodes on the tree can be generated by different paths but the actual path used does not matter, e.g. $(4,3)$ can be reached by filling the 4l jug then the 3l jug or vice versa.

Again it is fairly straightforward to represent a search tree in a program, but due to the rapid increase in size (another example of a combinatorial explosion), it is usual only to generate promising branches of the tree that are going to be searched.

Another method, called a **search graph**, solves the duplication problems of search trees. Here is the start of the search graph for the water jugs problem:



Self Assessment Question

8. Copy the search graph above and develop the bottom three nodes to another level.

A Variation on the Water Jugs Problem

Again the nodes are generated by applying the production rules, but now we have to check each time to see if that node has already been generated. This extra processing may not be worth the effort when compared to the processing involved in dealing with duplicate nodes.

Let's turn our water into wine and consider another problem!

Self Assessment Question

9. There are three jugs on a table. The largest one is full and contains 8 litres of wine. The other jugs are empty but can hold 5 litres and 3 litres respectively. The problem is to divide the wine so that there are 4 litres of wine in the 8 litre jug and 4 litres of wine in the 5 litre jug.

This time there is a limited supply of wine and obviously it cannot be thrown away.

Using one of the strategies discussed above, produce a solution to the problem where the first step is to fill the 5 litre jug.

Now produce a second solution where the first step is to fill the 3 litre jug.

Test the two solutions by checking that the three jugs always hold 8 litres among them.

Compare these two solutions for efficiency.

Evaluation

When we evaluate solutions we have to consider:

- the **functionality** of the solution:
 - does it do what it is meant to do, i.e. does it resolve the original problem?
 - is a solution by other means better?
- the **quality** of the solution:
 - is the solution efficient?
 - is the solution valid?
- **enhancements** to the solution:
 - can the solution be improved?
 - can the solution be generalised?
- the **consequences** of the solution:
 - are there moral or ethical implications in applying the solution?
 - are there legal implications?

Let us now put all these ideas together in a single problem. This example will also be used to describe what you require to include in a report on a problem of this type.

Self Assessment Question

10. A farmer wishes to move a fox, a goose and a bag of grain across a river. The farmer owns a boat but it is only big enough for himself and one other object. How can he move the fox, the goose and the bag of grain to the other side of the river when he cannot leave the fox and goose alone together as the fox will kill the goose, and he cannot leave the goose and the bag of grain alone as the goose will eat the grain.

Solve this problem and write a report recording the process at each of the following stages:

- i) State the problem in your own words.
- ii) State the boundaries of the problem.
- iii) Design a solution using a search tree.
- iv) State and test the solution or solutions that you find.
- v) Evaluate the solution with regard to the functionality of your solution(s), the quality of your solution(s), enhancements to your solution(s) and the consequences of your solution(s).

Heuristics and the ‘Eight Puzzle’

Another favourite problem of artificial intelligence is the ‘eight puzzle’ in which eight numbered tiles can slide horizontally and vertically in a 3×3 frame. One cell is empty and any adjacent tile can be moved into the space. Instead of thinking of the eight tiles moving around, it is sometimes simpler to think of the empty cell moving around.

4		3
2	1	5
7	8	6

Initial State

The puzzle starts with an initial position of the tiles and a stated goal position. The object is to manoeuvre the tiles to reach the goal position.

We have seen that a large number of alternatives quickly appear when the processes of problem abstraction and symbolic representation are applied to simple, contained problems with distinct goals.

1	2	3
4	5	6
7	8	

Goal State

When humans solve problems, they seem to reach solutions much more quickly. What we do is apply a **heuristic**. A heuristic is basically a rule of thumb, an educated guess, an intuitive judgement, experience, expertise or, in many cases, simply common sense that can be applied to solve a problem.

A heuristic cuts down the number of alternative paths that need to be tried. There are no rules for making up a heuristic. You simply have to try one in a particular problem situation and if it does not work, you discard it and try another. Let us consider how we might apply heuristics to our 'eight puzzle'.

Self Assessment Question

11. i) Use the initial and goal states shown on the previous page. We could start by drawing a search tree but probably what you have already done is to look at the problem and to solve it in your head. If you have not done this yet then do it now and count how many moves it will take you.

For each move the search tree will add another level, and for every situation at one level there will be two, three or four possible moves (depending on whether the empty square is a corner, a centre side or the middle square) though one of these possibilities will be reversing the previous move and so can be discarded. You should be able to appreciate that the tree will grow rapidly. What is this called?

Of course, our chosen problem has a solution within a few moves, but an initial state such as that shown on the right together with the same goal state will not be solved in so few moves – resulting in many more levels in the search tree. It is clearly impractical to generate all the levels of the search tree and so we must use a heuristic to select favourable branches to explore further and so discard other less favourable branches.

	8	7
6	5	4
3	2	1

Self Assessment Question 11 continued

ii) Here is a possible heuristic:

Count the number of tiles out of place and select the move that minimises this value.

This **evaluation function** will have a value of 0 when the goal state is reached and so we can hope that, by selecting the state at each stage that moves us towards this goal, we should progress along favourable branches of our search tree.

Use this heuristic to evaluate the possible first moves of our original 'eight puzzle'. What problem did you find?

iii) This first heuristic fails to take account of how far away each tile is from its goal position. Here is a second heuristic:

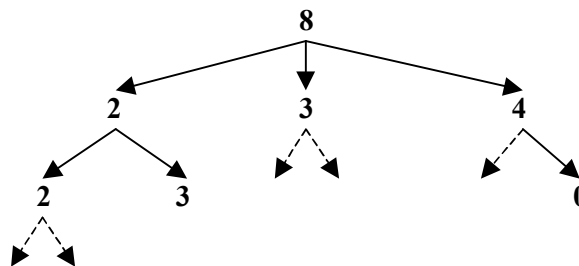
Calculate the sum of the horizontal and vertical distances of all the tiles that are out of position and select the move that minimises this value.

[For the initial state, tile 1 is one unit horizontally and one unit vertically out of position and so adds 2 to the evaluation function while tile 4 is only one unit vertically out of position and so adds 1 to the evaluation function.]

Use this heuristic to draw the favourable branches of the search tree and so find the solution to the puzzle.

Hill Climbing

The method that we have been using to select the branch to follow is to evaluate some function for each branch and then to select the branch that gives the 'best result'. In the above example the 'best result' is the value nearest to zero.



However, though this method seems very useful it has a number of drawbacks and may not produce the required solution. For example if the nodes of the search tree produce values for the evaluation function as shown then assuming low values to be best we would follow the path:

$8 \rightarrow 2 \rightarrow 2 \rightarrow \dots$

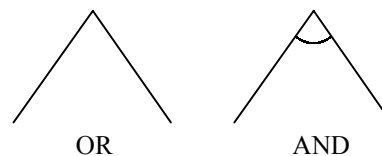
but fail to reach the solution by the path $8 \rightarrow 4 \rightarrow 0$.

In another situation high values of the evaluation function may be desirable and we would then select the highest value at each stage as the 'best result'. This method is known as **steepest-ascent hill climbing**. [We can think of the heuristic for the 'eight puzzle' as 'steepest-descent downhill skiing'.]

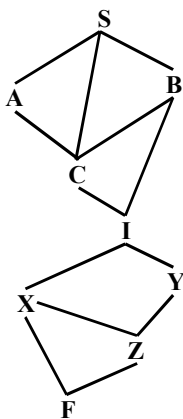
In the search trees and graphs that we have studied so far each branch is an OR branch. In other words we can go down one branch *or* down one of the others.

AND/OR Graphs

Another method of symbolic representation called an **AND/OR graph** is used for problems that can be split into independent sub-problems. OR branches are shown as before but AND branches are shown with an arc joining the branch lines.



An example of a problem that splits into independent sub-problems is one of finding routes between start and finish points that must go through some intermediate point or points. You may have come across Autoroute Express or similar software that can be used to plan road journeys.



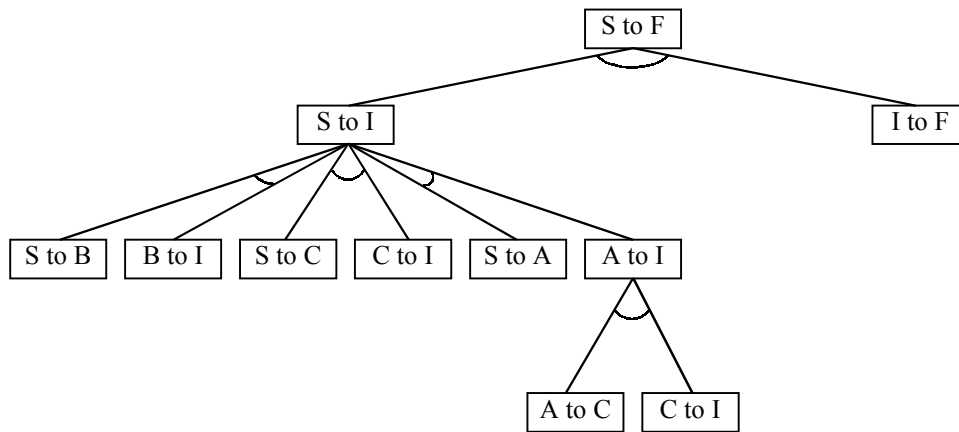
Typically, you enter the start and finish points and also places that you want to go through on the journey. You also enter details about the speed that you want to travel at and the sort of route that you want to follow, e.g. scenic or motorway. The program then searches for routes and displays the shortest route; the fastest route; the route using motorways most; the route using B roads most and other possibilities that you have selected.

Here is a system of roads between our start (S), our finish (F) and going through an intermediate point (I).

We can split the problem of finding routes from S to F into the sub-problems of finding routes from S to I and from I to F. Similarly, routes from S to I can be split into the sub-problems of finding routes from S to A and from A to I and so on.

Each road would be allocated certain properties such as its distance and its quality which would be combined to assign a 'cost' for that section of the route. We would then search for the route with a particular minimum 'cost' such as the shortest distance or the fastest route or the route on the best quality roads.

Here is part of the AND/OR graph for this route finding problem:



Self Assessment Question

12. Copy and complete the above AND/OR graph.

Conclusion

In this chapter we have looked at various methods of representing problem situations so that we can find solutions by searching through the possibilities that have been represented.

It is important to remind ourselves that developing artificial intelligence solutions requires these representations to be converted into data and instructions for computer solution. This course introduces the necessary concepts.

In Prolog the possibilities that have been represented by search trees, search graphs and AND/OR graphs would become **lists** while the

production rules and heuristic evaluation functions would become **rules**. Lists in Prolog are the equivalent of arrays in procedural languages.

You may have access to *Prolog – Programming for Artificial Intelligence*, by Ivan Bratko, which covers a number of the problems that we have discussed above. After discussion and analysis of the problem, the author produces a Prolog program to solve the problem.

You will find it a valuable exercise to try to follow some of these programs though you will have to read some of the early chapters to extend your knowledge of Prolog. You can also enter some of the programs as they are written in a standard dialect of Prolog.

Chapter 1: Key Points

- The analysis of solutions to even the simplest problems shows that most solutions are not trivial. A situation where a vast number of possible combinations has to be considered is termed a **combinatorial explosion**.
- Before a problem can be solved you must have a clear understanding of the problem. This means that you need to know the starting state and the final goal. Any constraints on the solution have to be identified. This is known as **problem abstraction**.
- **Symbolic representation** is a useful way of illustrating the state space. Representing the starting state, the final goal and valid intermediate states in a diagram is a clear and unambiguous way to solve a problem.
- The four stages in designing a solution are:
 - produce a precise problem description
 - analyse the problem
 - isolate and represent the necessary knowledge
 - select and apply the best problem solving method.
- **State spaces** can be represented by search trees, search graphs and AND/OR graphs.
- In many cases, people use **heuristics** to solve problems. This means that they bring their past experience and knowledge gained in similar situations to the current problem. A good heuristic should reduce the time needed to find a solution.
- Heuristics are frequently based upon an evaluation function that calculates a value for each state and then a minimum or maximum value is used to determine the branch to follow.

Chapter 1: Self Assessment Question – Solutions

1.
 - i) First as this always allows you an extra counter before your opponent plays and hence more win possibilities.
 - ii) The centre square as this opens up four win lines.
 - iii) Here is a possible strategy written in order of priority:
 on the first play take the centre otherwise take a corner;
 complete a winning line;
 stop an opponent's winning line;
 play to give yourself two winning lines;
 play to prevent the opponent having a play to give two winning lines;
 play to give one winning line;
 play in a corner square;
 play in a centre side square.
 - iv) This is answered by the priority of the strategy in iii).

2. Pick out the corner pieces (two straight edges).
 Pick out the frame pieces (one straight edge).
 Arrange the corner pieces and the frame pieces according to the picture.
 Sort frame pieces into types: two incut holes next to the straight edge; two extrusions next to the straight edge; pieces with one incut and one extrusion.
 Choose all the pieces of one object in the picture that has a dominant colour, e.g. a red brick building.
 Pick out the edge pieces of this object, i.e. a bit of red brick and blue sky or grass.
 There are other equally acceptable strategies.

3. Using the diagram, start at the extreme left, move to the centre and then down diagonally right, i.e. place C on the table; place B on the table; place A on top of C.

4.
 - (1) Precise problem description: by stating that we were dealing with chess we bring in to the problem all the precise details of chess, i.e. the board size, the scope of each piece, valid moves and the desired solution – checkmate.
 - (2) Analyse the problem: we played the game and recognised the large number of possible positions. We realised that the piece and its position must be represented and that possible moves had to be described.

- (3) Isolate and represent the necessary knowledge: The initial state is shown in the diagrams while a rule can be produced to describe the goal state, i.e. checkmate. The pieces, their positions and possible moves must be represented. This can be done by stating the piece and its position and producing rules like the one given.
- (4) Select and apply the best problem solving method: we would need to test every possible move from a given position for several moves by both players. To determine the best move we would require a function to assign a value to each move and resultant position and then select the move with the highest value.

- 5. Fill the 3 litre jug. Pour this into the 4 litre jug. Fill the 3 litre jug. Pour water from the 3 litre jug into the 4 litre jug to fill it. Empty the 4 litre jug on the ground. Pour the water from the 3 litre jug into the 4 litre jug. The 4 litre jug now contains 2 litres of water.

Yes, it works.

It is difficult to judge the quality of this solution – see the rest of this answer.

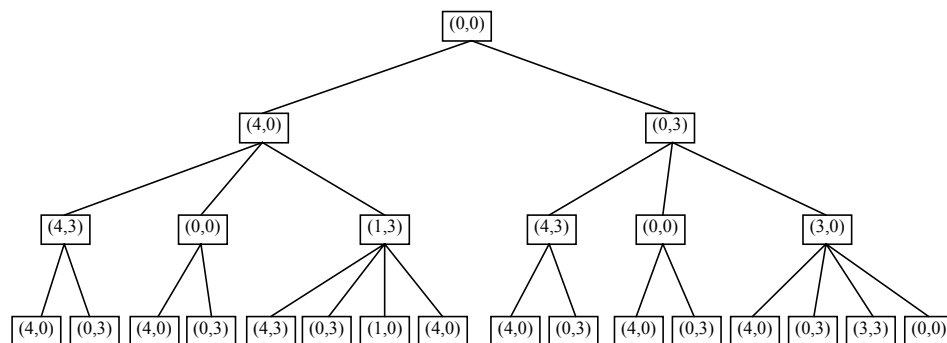
We could have produced all the possible changes that we can make and used each of these from the initial state to generate every possible combination until the goal state was reached. This would have generated exhaustively every possible solution.

Once the last stage had produced every possible solution then we could check each for the amount of wasted water.

It is unlikely that there is a better solution than the one that we produced but how do we know?

- 6. The rules are used in the order: 2, 7, 2, 5, 3, 7.

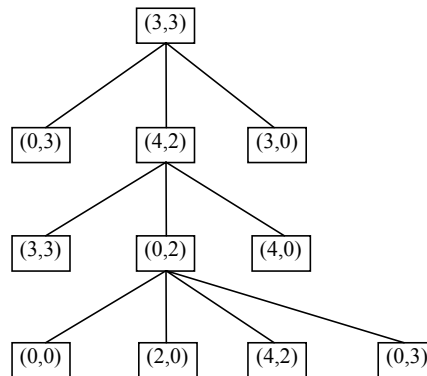
- 7.



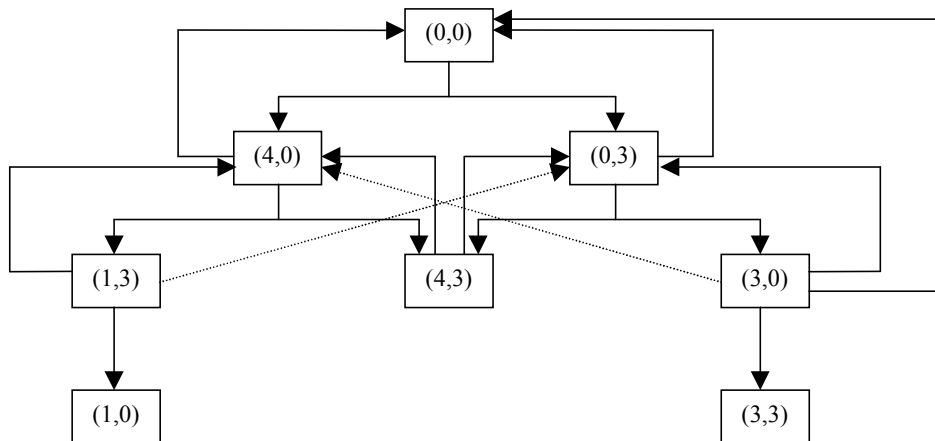
There are many more states at each level and hence the diagram becomes cumbersome and crowded.

Entries are repeated and at each level we are including the reverse of the change that produced the level above.

We continue the (3,3) branch but only showing branches from the solution at each level.



8.



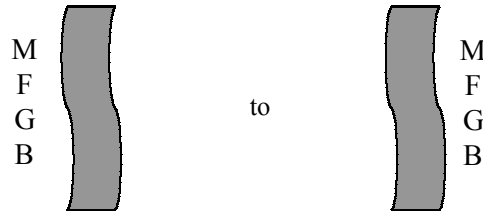
9. We can represent a state for this problem by an ordered triple (x,y,z) where x is the amount of wine in the 8l jug, y is the amount of wine in the 5l jug and z is the amount of wine in the 3l jug. The starting state is $(8,0,0)$ and the goal state is $(4,4,0)$.

Here is the sequence when we fill the 5l jug first: $(8,0,0) \rightarrow (3,5,0) \rightarrow (3,2,3) \rightarrow (6,2,0) \rightarrow (6,0,2) \rightarrow (1,5,2) \rightarrow (1,4,3) \rightarrow (4,4,0)$.

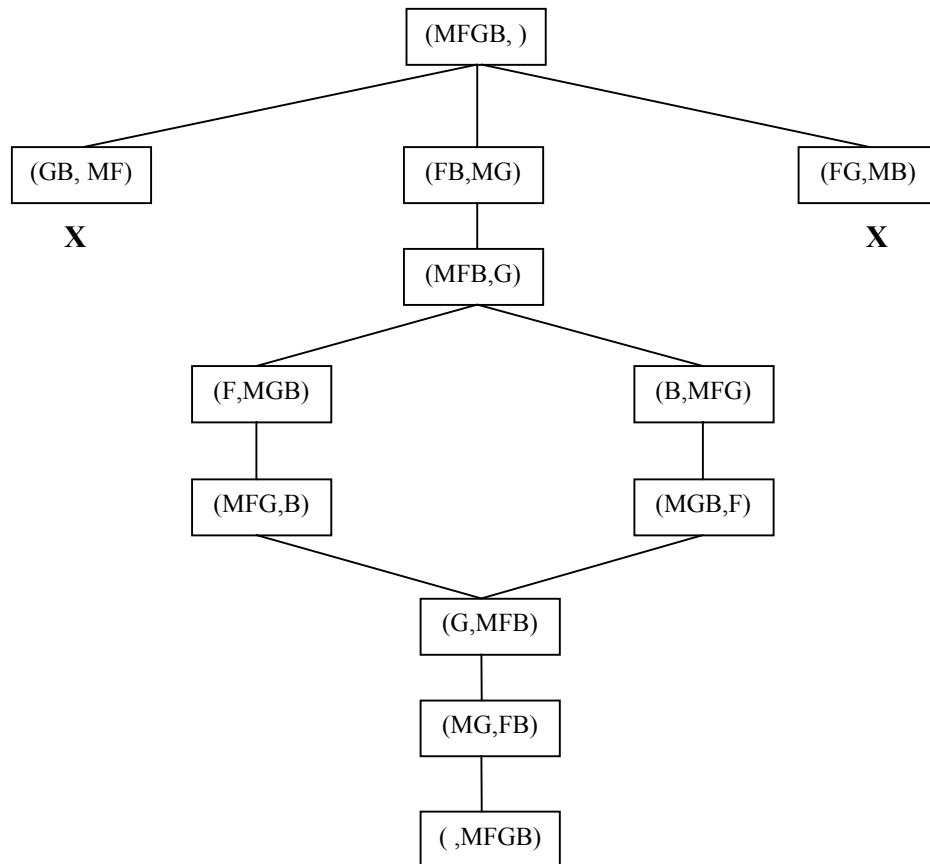
Here is the sequence when we fill the 3l jug first: $(8,0,0) \rightarrow (5,0,3) \rightarrow (0,5,3) \rightarrow (3,5,0)$ and continue from step 2 above.

These solution paths could be represented by a search tree or a search graph. The first solution is more efficient as it takes seven pourings while the second solution requires two pourings to reach the state where the first solution is used.

10. i) A farmer (M = man) must move a fox (F), a goose (G) and a bag of grain (B = bag) across a river. The initial and goal states are:



- ii) He must never leave the fox and goose alone on one bank or the goose and the bag of grain alone on one bank. His boat can only carry himself and one other object.
- iii) The search tree is shown below. Ordered pairs such as (FB, MG) indicate that after a crossing the fox and the bag of grain are on the left bank while the farmer and the goose are on the right bank.



- iv) The left-hand solution is: The farmer crosses with the goose; returns alone; crosses with the bag of grain; returns with the goose; crosses with the fox; returns alone; crosses with the goose. This gives seven crossings. You should also test the right-hand solution.
- v) The solution satisfies the original problem and succeeds in moving the farmer, the fox, the goose and the bag of grain to the other bank without loss. It would be difficult to prove that there is no better solution.

The solution is certainly valid and appears to be efficient though again this would be difficult to prove.

As stated above it would be difficult to prove that there is no better solution but years of attempting to find one by large numbers of people have failed to find a better solution. Generalisations of this solution would be that a solution should isolate the ‘most awkward’ member (in this case, the goose as it can interact with both of the other members).

There are no moral, ethical or legal consequences of this solution.

- 11. i) A solution in seven moves: move 1; move 2; move 4; move 1; move 2; move 5; move 6.
- ii) The original state evaluates to 5.

4		3
2	1	5
7	8	6

The possible second states are shown below and evaluate to 5, 5 and 6 respectively as moving one tile does not place any tile in its correct position and hence the evaluation is either not reduced or made worse. Hence we could choose either the left-hand or middle moves.

4	1	3
2		5
7	8	6

	4	3
2	1	5
7	8	6

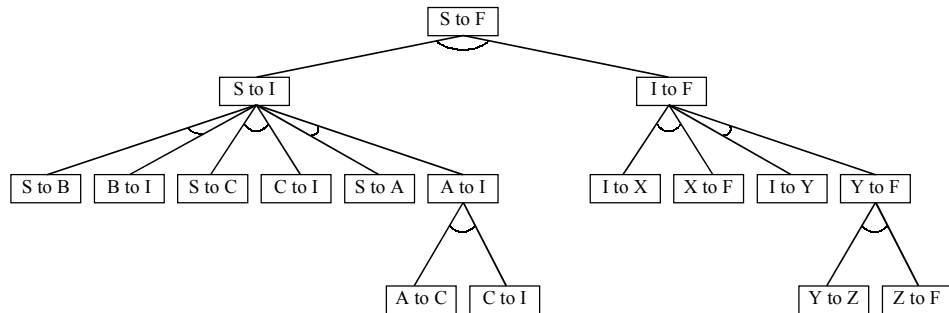
4	3	
2	1	5
7	8	6

- iii) Using the second heuristic, the initial state evaluates to 2 (tile 1) + 2 (tile 2) + 1 (tile 4) + 1 (tile 5) + 1 (tile 6) = 7 .

The second states above evaluate to 6 , 8 and 8 respectively and hence we would select the left-hand move. If you follow through the solution you will see that the evaluation function changes as follows: move 2 (value = 5); move 4 (value = 4); move 1 (value = 3); move 2 (value = 2); move 5 (value = 1); move 6 (value = 0).

Did you spot that instead of moving tile 2 from the position in the left-hand diagram, you could move tile 5 and still get a value of 5 . If you then moved tile 6, the value would decrease to 4 (the bottom right corner is now the empty square) but then any move would result in a greater value. We need to add to our heuristic that if two nodes are equal, we follow one path until either we reach a solution or the evaluation increases and we backtrack to the other node.

12.



Introduction

In Chapter 1 we looked at some simple puzzles and how we might represent the processes involved in solving the problem. The intention was to devise general methods that would apply to a variety of problems and variations of these problems. For example, if we have a method to solve the water jugs problem when the jugs are 4 litre and 3 litre, then we should be able to apply the same method when the jugs are 5 litre and 2 litre.

In all these situations we were representing the state space of the problem as new states were generated. Problem abstraction took the initial state, the goal state and valid intermediate states and represented these using a number of symbolic methods.

We now look at representing a constant domain of knowledge rather than changing states – **knowledge representation**. As before it is important to keep in mind that our methods should be easy to convert into data and instructions for a computer to process.

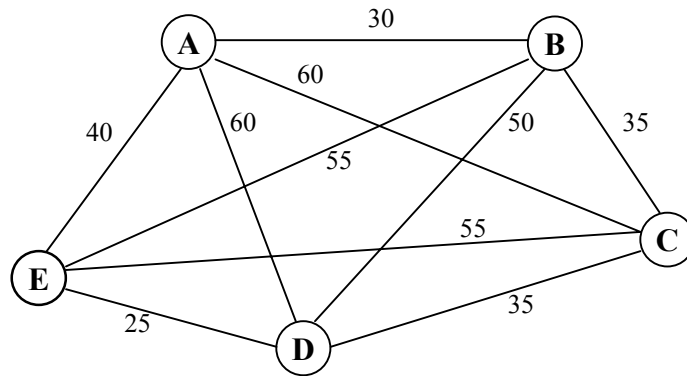
We also have to be able to search this knowledge domain to find the answers to questions. You will have met some of these ideas before if you studied Prolog and expert systems in the Higher Computing Unit – Artificial Intelligence. Also, many of the ideas developed in Chapter 1 carry over to knowledge representation.

Section 1: Methods for Representing Knowledge Domains

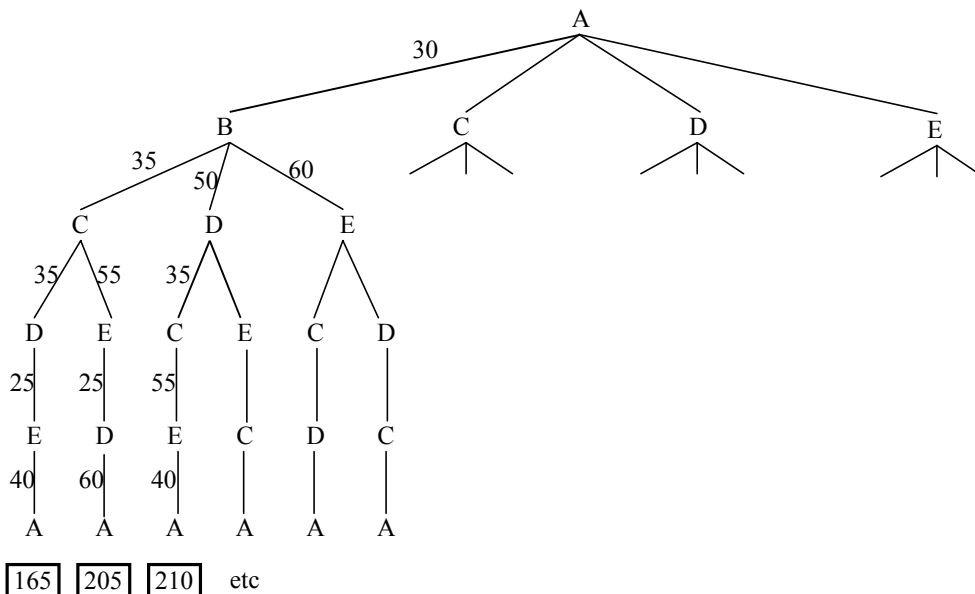
We are now going to look at four methods for representing knowledge. The first of these we have met already in the simple puzzles of Chapter 1.

Search Trees

Here is a simple map of the roads linking five towns that we will call A, B, C, D and E. The numbers on each line indicate the distance between that pair of towns. This is another classic problem in artificial intelligence, where a travelling salesman starts and finishes his visits to the other towns at town A and the task is to find the shortest route for the salesman to travel.



We can now construct a search tree of all the routes starting and finishing at A and, for each branch of the tree, add up the distance travelled.



The figures 165, 205 and 210 are the totals for the first three routes.

Self Assessment Questions

1. By inspecting the search tree for 5 towns, write down the number of possible routes that start at A and finish at A.

If we extended the problem to 6 towns, how many routes would there be?

If the problem is extended to 50 towns, which phrase would describe the situation that arises?

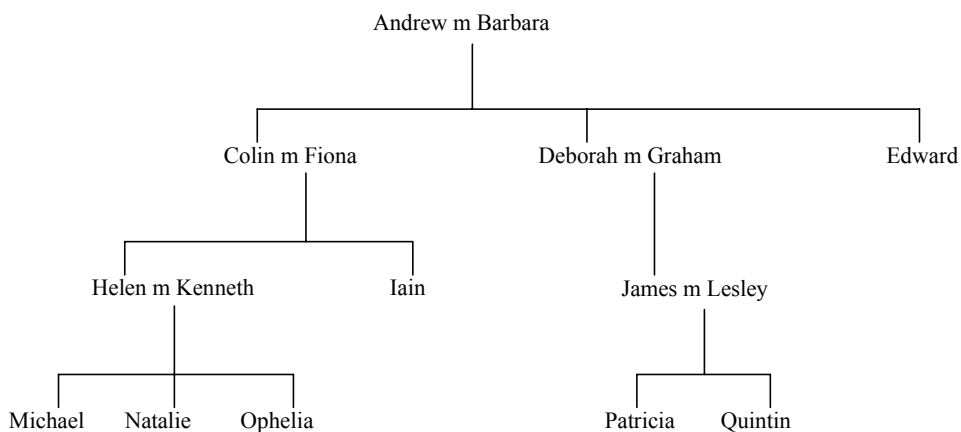
2. One method to reduce the number of search paths is to always select the town nearest to the current town that has still not been visited. This is called the 'nearest neighbour' heuristic and results in only one path being tested.

Apply the 'nearest neighbour' heuristic to the problem of finding the shortest route between 5 towns starting and finishing at A.

Search Trees and Prolog

We mentioned in Chapter 1 that it is important when representing state spaces or, as now, knowledge to bear in mind that we must eventually computerise the process. Prolog is one declarative language used in the representation and processing of knowledge. Let us look now at how we code a search tree in Prolog.

When you studied Prolog, you almost certainly looked at a family tree of the form:



Here Colin, Deborah and Edward are the children of Andrew and Barbara. Colin then married Fiona and Deborah married Graham. The family tree then continues for another two generations. This family tree can be coded in Prolog using facts of the form:

male(andrew).	similarly for the other males
female(barbara).	similarly for the other females
father(andrew,colin).	similarly for the other father and son or daughter pairs.
mother(barbara,colin).	similarly for the other mother and son or daughter pairs.

We can then define rules for limited connections between nodes of the tree. For example, we can define rules for 'parent', 'offspring', 'grandparent', 'brother' and 'sister'. Here are the rules for 'parent' and 'offspring':

parent(A, B) :- father(A, B).	A is the parent of B if A is the father of B
parent(A, B) :- mother(A, B).	or A is the mother of B. A rule that contains OR is normally written as two clauses. Some versions of Prolog use 'if' instead of ':-'.
offspring(A, B) :- parent(B, A).	

However, if we wish to find out if two nodes are connected over any number of levels of the search tree then we have a more complex problem. For example, to answer the questions, 'Is Andrew an ancestor of Ophelia?' or 'Is Michael a descendant of Edward?' would require us to determine whether there is a connecting path between our two objects. From an inspection of the family tree above, Andrew is an ancestor of Ophelia while Michael is not a descendant of Edward.

To code this in Prolog we require to define rules using **recursion**. Here are 'ancestor' and 'descendant' rules:

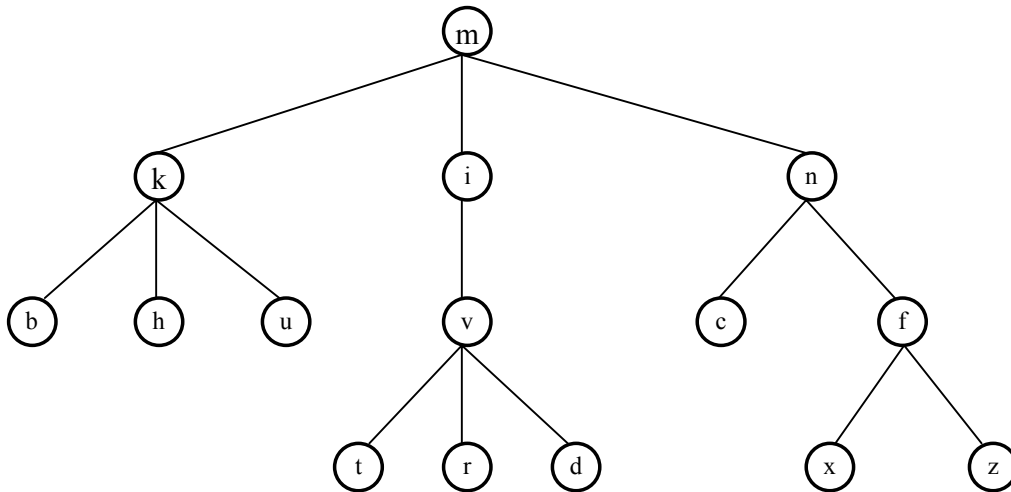
ancestor(A, B) :- parent(A, B).
ancestor(A, B) :- parent (A, X), ancestor(X, B).

A recursive rule has two parts – the special or terminating case and the general or recursive case. The general case reads as A is an ancestor of B if A is the parent of some X and X is an ancestor of B. Some versions of Prolog use 'and' instead of ','.

descendant(A, B) :- offspring(A, B).
descendant(A, B) :- offspring(A, X), descendant(X, B).

When the 'ancestor' rule is satisfied then there exists a connecting path from A down to B. When the 'descendant' rule is satisfied then there exists a connecting path from A up to B.

These rules can be generalised to produce rules that find connecting paths in any search tree. For example, if we have a search tree of the form:



We can check for connections down the tree by using a recursive rule of the form:

```

connection_down(A, B) :- direct_down(A, B).
connection_down(A, B) :- direct_down(A, X), connection_down(X, B).
  
```

Self Assessment Question

- Write the recursive rule for 'connection_up'.

If you are not very sure about recursion and forming recursive rules then you could work through Appendix 1 or your teacher/lecturer may give you alternative materials on recursion.

Problems with Search Trees and the 'Nearest Neighbour' Heuristic

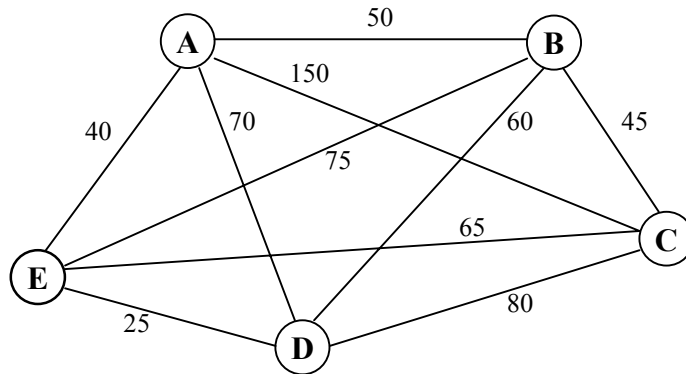
Search trees are only useful when a limited number of combinations are possible. If we are using a computer program to generate the branches of the tree then we would only generate one path at a time together with its evaluation and only store the best path found so far.

As with any heuristic, we must remember that it may not find the best solution. The 'nearest neighbour' heuristic is particularly prone to this problem as it takes no account of later values when the nearest

neighbour is selected. Let us consider this for the travelling salesman example.

So far we have only considered the distance between the towns, but other factors would influence our choice of route between two towns if we wanted to minimise the salesman's travelling time. These would include the quality of the roads, the speed limit, the amount of traffic likely to be encountered, the use of ferries or toll bridges. You can probably think of a local journey where it is faster to use motorways or A roads than a shorter route on B roads or crowded streets.

If we include these factors then each link between two towns can be given a 'cost' in terms of the time, distance and extra costs of the route and we can produce a revised map of these 'costs'.



Self Assessment Question

4. Use the 'nearest neighbour' heuristic to find a route between the 5 towns starting and finishing at A and calculate the 'cost' of this route.

How does this route's 'cost' compare with the 'cost' of the route round the perimeter $A \rightarrow E \rightarrow D \rightarrow C \rightarrow B \rightarrow A$?

Logic

Logic is a formal mathematical discipline that specifies a system of rules to define the process of reasoning. Logic is another method of representing knowledge domains. Representing knowledge using formal logic is a very attractive concept for artificial intelligence programmers as deductions can be made using the rules of reasoning.

You will have come across some of the following ideas in other work but here we are going to use the standard symbols of logic:

\rightarrow implies	\wedge and	\forall for all
\neg not	\vee or	\exists there exists

We can then write statements of the form:

$\exists x: \text{mother}(x, \text{Mary})$	There exists x such that x is the mother of Mary.
$\exists y: \neg \text{mother}(y, \text{Mary})$	There exists y such that y is not the mother of Mary.
$\forall x \forall y: \text{parent}(x, y) \rightarrow \text{father}(x, y) \vee \text{mother}(x, y)$	For all x and y , if x is the parent of y then x is the father of y or x is the mother of y .
$\forall x \forall y \forall z: \text{parent}(x, y) \wedge \text{parent}(x, z) \rightarrow \text{sibling}(y, z)$	For all x, y and z , if x is the parent of y and x is the parent of z then y and z are siblings.

Logic and Prolog

This is very similar to the methods that you have used to express facts and rules in Prolog. Prolog is based on one type of formal logic called **first-order predicate calculus**.

The one logic statement above that is not easily expressed in Prolog is the second one where we state that there exists y such that y is not the mother of Mary. This logical negation cannot be expressed clearly in Prolog and is one of the major drawbacks in the use of Prolog.

Once we have the knowledge represented by logic statements, we use backward reasoning to find the answer to questions. That is, we start with the question and decide what we require to satisfy it and whether this is contained in the knowledge base. For example, use the numbered logic statements (written in Prolog form):

1	<code>father(zeus, ares).</code>	Zeus is the father of Ares.
2	<code>father(zeus, hebe).</code>	Zeus is the father of Hebe.
3	<code>parent(X, Y) :- father(X, Y).</code>	X is the parent of Y if X is the father of Y
4	<code>sibling(X, Y) :- parent(Z, X), parent(Z, Y).</code>	X is the sibling of Y if Z is the parent of X and Z is the parent of Y .

to answer the question, 'Are Ares and Hebe siblings?'.

Using 4 we look for a parent for Ares; using 3 we look for a father for Ares; using 1 we find that Zeus is a father. Using 4 we look for a parent for Hebe; using 3 we look for a father for Hebe; using 2 we find that Zeus is a father. Hence Zeus is a father and so a parent for both and so Ares and Hebe are siblings.

Self Assessment Questions

5. Consider the following sentences:

James likes all types of books.

Thrillers are a type of book.

Biographies are a type of book.

Anything that takes over three hours to read is a book.

Jennifer took six hours to read *Treasure Island*.

Susan likes the same books as Jennifer.

Represent this knowledge as logic clauses. You will probably find it easier to write them in Prolog rather than using formal logic.

- i) Explain how the fact that James likes biographies can be deduced.
- ii) Explain how the fact that James likes *Treasure Island* can be deduced.
- iii) Explain how the question, 'What books does Susan like?' would be answered. What problem do you see with using logic to represent knowledge?

6. Consider the following facts:

Fraser only likes easy courses.

Intermediate 2 and Higher courses are hard.

Intermediate 1 courses are easy.

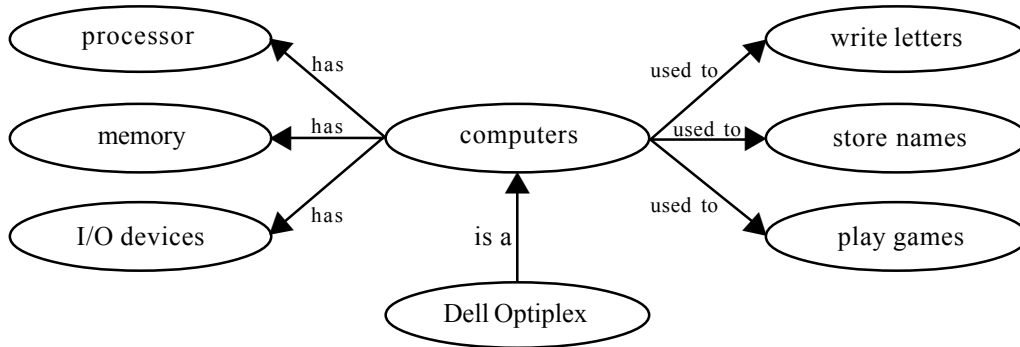
Computing Studies is an Intermediate 1 course.

Represent this knowledge as logic clauses and then show how the solution to the question, 'What course should Fraser take?' would be found.

Semantic Nets

Again, this is another method of representing knowledge domains. Semantic nets are used when objects belong to a larger class and so each object can inherit the properties of the larger class without the need to list every property for every object. As the name suggests, the knowledge is most easily described by using a diagram.

Computers are a class of objects that all have a processor, memory and input and output devices. Also computers can be used to write letters, to store names and to play games. The Dell Optiplex is a computer and so it inherits all the properties of computers. This would be shown by the following diagram:



Semantic Nets and Prolog

Prolog can be used to code knowledge that has been represented in a semantic net that includes the idea of inheritance. This is done by using facts to represent the basic information and rules to represent the information that is inherited. The above example could be translated into Prolog as follows:

```

has(computer, processor).
has(computer, memory).
has(computer, i/o_devices).
used_for(computer, write_letters).
used_for(computer, store_names).
used_for(computer, play_games).
is_a(dell_optiplex, computer).
  
```

```

has(dell_optiplex, X) :- has(computer, X).
used_for(dell_optiplex, X) :- used_for (computer, X).
  
```

These are the rules of inheritance

The two inheritance rules are rather specific and if we had another make of computer, e.g. a Compaq Proliant, then we would have to duplicate these rules by replacing dell_optiplex with compaq_proliant in addition to adding the 'is_a(compaq_proliant, computer).' fact. However we can generalise the two inheritance rules above:

```

has(Y, X) :- is_a(Y, computer), has(computer, X).
used_for(Y, X) :- is_a(Y, computer), used_for(computer, X).
  
```

We can now add as many makes of computer as we wish and they will all inherit the properties of the class computer.

If in the same knowledge base, we also had a class called 'printer', properties attached to this class such as 'has(printer, buffer).' and 'has(printer, ink_cartridge).' and members of the class such as 'is_a(hp_laserjet, printer)' then we would need another inheritance rule for "printer" of the form:

has(Y, X) :- is_a(Y, printer), has(printer, X).

By now you should have guessed that this is inefficient and that a more general rule is possible. Here is a fully generalised inheritance rule:

has(Y, X) :- is_a(Y, Z), has(Z, X).

Self Assessment Questions

7. Represent the following knowledge using a semantic net.

All expert systems have expertise, symbolic reasoning, handle difficult problem domains and examine their own reasoning.

Expert systems can be categorised into several application types of which interpretation, diagnosis, design and monitoring are four such categories.

Examples of interpretation systems are Dendral and Puff. Examples of diagnostic systems are Mycin, Prospector and Ace. An example of a design system is Palladio. Examples of monitoring systems are Reactor and Anna.

8. Represent the following knowledge using a semantic net.

All vehicles have certain common features. They all have wheels, an engine, a driver and need to have vehicle excise duty paid. There are, of course, other common features.

Vehicles can be split into three categories, private cars and vans, public service vehicles and heavy goods vehicles.

Private cars and vans have seats for up to eight passengers and have only four wheels. Examples of private cars are the Ford Mondeo and the Vauxhall Astra. Public service vehicles have seats for nine passengers or more and have a specially qualified driver. Examples of public service vehicles are coaches and minibuses. A Ford Transit is an example of a

Self Assessment Question 8 continued

minibus. Heavy goods vehicles have a load capacity over three tons, have more than four wheels and have a driver with a HGV licence. An example of a heavy goods vehicle is an articulated lorry.

9. Convert the information from the semantic net for Self Assessment Question 8 to Prolog statements. If you have access to a version of Prolog then you should enter the program and test it. You will require a recursive rule to allow connections between the levels for the 'is_a' property, e.g. a Transit is a minibus is a PSV is a vehicle. You will also require an inheritance rule for the 'has' property.

Frames

This is the final method of knowledge representation that we will look at.

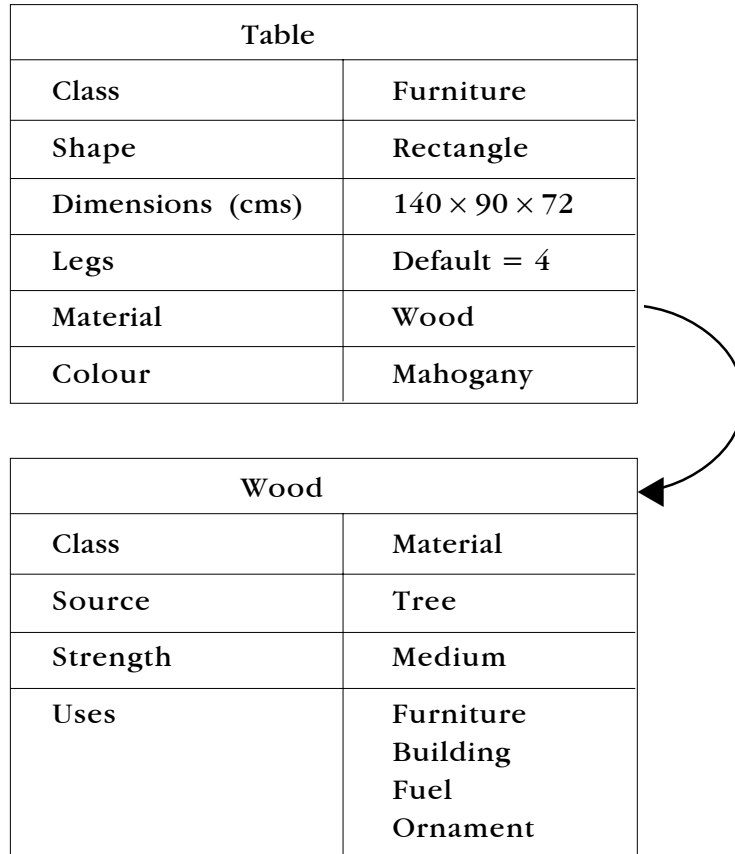
Research has shown that humans organise information in broad conceptual structures and fill in particular details as needed. For example, if you hear or see mention of a wooden table your brain will recall what it knows about a table (four legs with a solid horizontal, rectangular, square, circular or oval top) and it will also recall the relevant facts about wood (sturdy, brown, can be polished) and you get an even better picture of a wooden table.

The frame system developed in artificial intelligence research mimics this human style of knowledge organisation by creating a data structure consisting of slots that are filled with specific instances of data. For example we could organise the data about a wooden table as follows:

Table	
Class	Furniture
Shape	Rectangle
Dimensions (cms)	140 × 90 × 72
Legs	Default = 4
Material	Wood
Colour	Mahogany

In some cases a slot may hold a default value which will be assumed unless an alternative value is specified. In the Table frame, the default value for the number of legs is 4, but this could be changed (for example a larger table might have 6 legs).

As shown below, the values stored in slots are often pointers to other frames:



Of course, 'Tree' could now link to a frame of biological and natural history knowledge.

This is similar to the hypertext links that you find on the Internet and in CD-ROM based information sources such as an electronic encyclopaedia. For example, one CD-ROM encyclopaedia links 'Impressionism' to 'Barbizon school' to 'Theodore Rousseau' to 'Fontainebleau' to 'Paris' to 'Eiffel Tower'.

Frames allow facts to be retrieved directly and, in a similar way to semantic nets, frames may be linked to form a hierarchy that allows properties to be inherited. For example, the 'Table' would inherit properties from the class 'Furniture', and 'Wood' would inherit

properties from the class 'Materials'. This property inheritance is coded in Prolog in the same way as with semantic nets. The frame properties are handled using lists and so require more advanced Prolog programming.

Self Assessment Questions

10. Represent the following knowledge using frames.

A microcomputer system has a Pentium II processor with a speed of 350 MHz. It has a memory size of 64 Mb and is fitted with a network card so that it can be connected to a network. It has a keyboard and mouse as input devices and a monitor, loudspeakers and printer as output devices. It has a floppy disk drive, a hard disk drive and a CD-ROM drive as backing storage devices.

Printers come in two main types and are used to produce hard copy of computer output. Ink-jet printers are slow, can produce colour and are cheap to buy but expensive to run. Laser printers are fast, produce black and white or colour output and are expensive to buy but cheap to run.

Show the links between your frames.

11. Represent the following knowledge using frames.

Canada is a country consisting of ten provinces of which the westernmost is British Columbia. It occupies the northern part of North America and has an area of almost 10 million square kilometres with a population of 26 million. Languages used are English and French and the currency is the Canadian dollar.

British Columbia is a province of Canada with a land area of 950,000 square kilometres and a population of 3.5 million. Its capital is Victoria but its largest city is Vancouver. Its natural resources are minerals, timber and fisheries.

Minerals are obtained by deep mining or quarrying. They consist of ores that are used directly such as sand and gravel or from which metals such as gold, copper or iron are extracted.

Show the links between your frames.

Conclusion

You should now be aware of how a simple domain of knowledge may be represented by a search tree, logic, semantic net or frame. The representation method that is used in any particular situation depends upon the knowledge domain being represented. Each method has its strengths and weaknesses.

	Strengths	Weaknesses
Search tree	Clear development of links to objects from initial state. Good pictorial representation of search paths especially in 'route' problems.	Not suitable for knowledge that has no obvious source point and development path, e.g. knowledge of a wooden table. Can be difficult to convert to Prolog code. Only useful for small domains of knowledge as there is no mechanism for inheritance and so the tree grows quickly.
Logic	Fast conversion to Prolog code. Efficient and compact representation.	Formal symbols awkward for inexperienced users. No pictorial representation of knowledge structure or search.
Semantic net	Efficient and clear pictorial representation of knowledge that contains property inheritance. Reduces the size of the knowledge base as common properties are stored only once. Searches follow paths through the network.	Diagram can become unwieldy with large knowledge bases.
Frame	Allows the representation of structured knowledge. More complex knowledge is efficiently captured.	Conversion to Prolog requires more advanced features of the language.

Section 2: Search Techniques

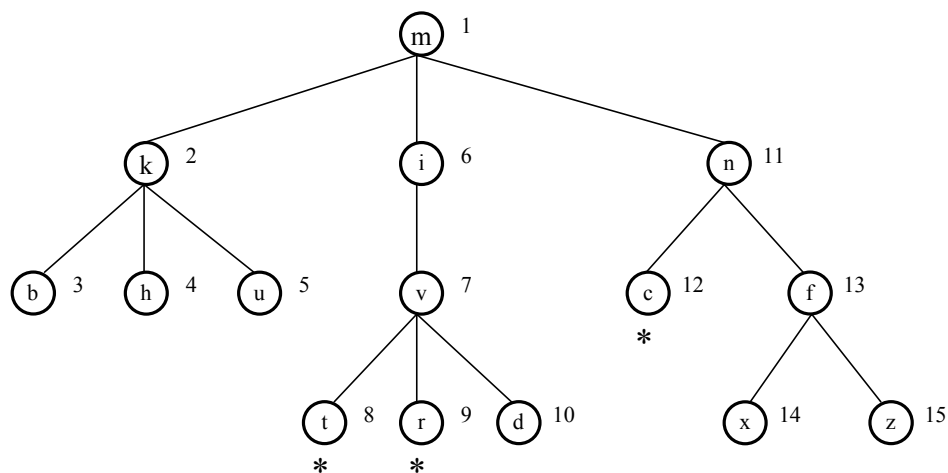
Exhaustive Search Techniques

We have just considered four ways in which simple domains of knowledge can be represented. In this section we are going to investigate how the knowledge domain can be searched. We will first consider two methods of carrying out an exhaustive search. This is a systematic problem-solving technique that tries to find a solution by examining all possible solutions. Clearly in many situations an exhaustive search is a costly and impractical process due to the time and processing power required.

The two methods by which an exhaustive search can be undertaken are called **depth-first** searching and **breadth-first** searching. Let us now look at how these two exhaustive search methods work and compare them.

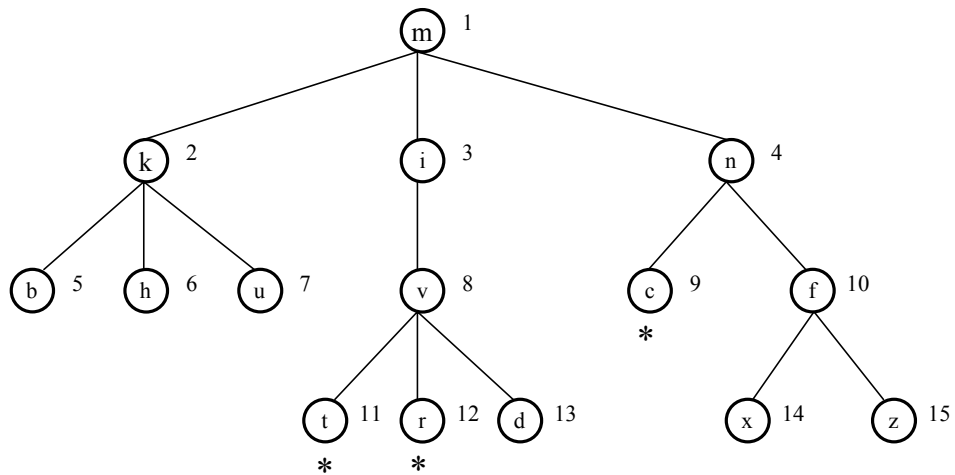
i) *Depth-first Search*

This search starts at the root of the tree (or the start of the clauses if we have converted the knowledge into Prolog) and works down the left-hand branch of the tree considering each node until it reaches a leaf or terminal node. If this is not a goal state then it goes back up and tries the next available path down. This search, as its name suggests, tries to get as deep as possible as fast as possible. The following diagram indicates the order in which the nodes will be tested (goal states are marked with an asterisk):



ii) *Breadth-first Search*

This search starts at the root of the tree and searches all possible nodes at that level, only searching the next level once all nodes at the previous level have been checked. The following diagram indicates the order in which the nodes will be tested:



iii) *Comparison of Depth-first and Breadth-first Search*

- a) The two search methods may encounter different solutions first. For example, depth-first search finds solution 't' after 8 nodes while breadth-first search finds solution 'c' after 9 nodes.

Depth-first always finds the first solution from the left while breadth-first finds the shallowest solution first.

- b) Depth-first search requires less memory as only nodes on the current path are stored. For example, when searching to 't', depth-first search must store nodes 'm', 'i' and 'v' and when the solution is found then the path 'm→i→v→t' is returned as a solution.

Breadth-first search requires more memory as it must store the whole tree generated so far. For example, when searching to 't', breadth-first search must store all the paths used so far [m, k, b], [m, k, h], [m, k, u], [m, i, v], [m, n, c], and [m, n, f] and when the solution is found then the path 'm→i→v→t' is returned as a solution.

- c) By chance, depth-first search may find a solution very quickly if it is deep at the left-hand side of the tree and so it can stop if only one solution is required.

Breadth-first search must search all the previous levels before finding a deep solution.

- d) Depth-first search seems to have all the advantages but ...

Depth-first search may get into an infinite loop if we do not do extra programming to ensure that previously generated states are not searched again. For example, we must eliminate the reverse of the previous change (see the Water Jugs problem in Chapter 1).

Breadth-first search will not get into an infinite loop.

- e) Depth-first search may find a very long solution deep in a left-hand side branch of the tree when a short solution exists elsewhere in the tree.

Breadth-first search is guaranteed to find a solution if it exists and this solution will be the solution with the minimum number of steps.

Humans often use a combination of both techniques to try to get the best of each. A French mathematician, Henri Poincaré, described searching for solutions to mathematical problems as follows:

Imagine that you are in an underground cavern trying to get back to the surface and that there are passages running off in many different directions but you can only see a few feet along each passage. The born mathematician always selects a passage that runs a long way if, indeed, it is not the solution passage.

There are artificial intelligence methods that combine depth-first and breadth-first searching. One such method is called **branch-and-bound**. You may be interested to use a textbook to find out about this and other methods.

Prolog and other knowledge processing languages use the depth-first search strategy. You should be familiar with this from earlier

work that you have done in Prolog where depth-first searching can be seen by using the language's trace facility when it is attempting to satisfy a query.

You should also ask your teacher/lecturer if there is an expert system available which would allow you to investigate the depth-first and/or the breadth-first search methods further.

A problem associated with both depth-first and breadth-first search methods is **combinatorial explosion**. For non-trivial problems, the number of possible solutions that need to be checked can become unacceptably high. We have met this problem in several earlier problems in both Chapter 1 and Chapter 2.

(iv) *Coding Exhaustive Searches in Prolog*

As we have seen it is necessary to store nodes in both types of exhaustive search. In Prolog, when we wish to store data then we use lists. If you are not very sure about lists and forming recursive rules to process them then you could work through Appendix 2 or your teacher/lecturer may give you alternative materials on lists.

For depth-first search the method that we use is to keep a list of the nodes on the current path that we are checking. This list is initialised to the root and so at the start this is the head of the list. The 'successor' of the head is generated and added to the head of the list and further 'successors' of the head are generated and added to the head of the list of nodes. Each time a successor is added, the head is checked and if it is the goal state then this solution is output. If the head has no 'successors' then it is removed from the list (as there are no more branches below this node) until no nodes remain in the list as the root has no more unchecked successors.

Here is the algorithm for depth-first search:

```

1   Initialise the list of nodes to the root
2   WHILE the list is not empty
3     Take the head of the list
4     IF the head has a successor THEN
5       Generate the next node at the next level below the
        head
6       IF this new node is the goal state THEN return the
        list as a solution
7       Add this new node to the head of the list of nodes
8     ELSE
9       Remove the head from the list of nodes
10    ENDIF
11  ENDWHILE
12  IF the list is empty and no solutions have been found THEN
    report no solutions

```

With reference to the search tree used previously to illustrate depth-first search, the list of nodes followed by the numbered step will change as follows:

```

[m] from step 1; ...[k, m] from 6; ...[b, k, m] from 6; ...[k, m] from 9;
[h, k, m] from 6; ...[k, m] from 9; ...[u, k, m] from 6; ...[k, m] from 9;
[m] from 9; ...[i, m] from 6; ...[v, i, m] from 6; ...
[t, v, i, m] from 6 and success reported; ...

```

The only differences to the above algorithm when we do a breadth-first search is that now we store a list of paths rather than just nodes, and in step 7 we add the whole newly generated path to the tail of the list of paths.

```

7   Add this new path to the tail of the list of paths

```

The list of paths will change as follows:

```

[[m]] from step 1;
[[m], [m, k]] from 6;
[[m], [m, k], [m, i]] from 6;
[[m], [m, k], [m, i], [m, n]] from 6;
[[m, k], [m, i], [m, n]] from 9;
[[m, k], [m, i], [m, n], [m, k, b]] from 6;
[[m, k], [m, i], [m, n], [m, k, b], [m, k, h]] from 6;
[[m, k], [m, i], [m, n], [m, k, b], [m, k, h], [m, k, u]] from 6;
[[m, i], [m, n], [m, k, b], [m, k, h], [m, k, u]] from 9; ...

```

The next stage is that all the paths [m, i, _] are generated and added to the end of the list before [m, i] is discarded.

Self Assessment Questions

12. Complete the list of nodes for depth-first search to produce all the solutions.
13. Complete the list of paths for breadth-first search as far as the first solution [m, i, v, t].
14. Consider each of the following problems and explain why each would or would not give rise to a combinatorial explosion.
 - i) Playing chess.
 - ii) Finding the shortest route from your home to school.
 - iii) Using Pythagoras' Theorem to calculate the third side of a right-angled triangle.
 - iv) Building a house.
15. Represent the following knowledge by a search tree. Indicate on the tree the order in which the nodes will be checked by a depth-first search (by marking them d1, d2, d3, etc.) and by a breadth-first search (using b1, b2, b3, etc.).

Parents in a small town have the choice of sending their children to the local state primary school or to a private preparatory school. Once they reach the age for secondary school, pupils at the state primary transfer to the local state secondary school, while pupils at the private preparatory school usually move on to the secondary department of the same school, though some transfer to the state secondary.

Once they complete their school education, pupils from both schools either go straight into work, or go to college or university.

Heuristic Search

Heuristics were introduced in Chapter 1. Applying a heuristic to a problem means that you use past experience and knowledge gained in similar situations to solve the current problem. A good heuristic should reduce the time needed to find a solution as it can be used to limit the search process through eliminating a lot of unproductive searches. But, as we have seen with the ‘nearest neighbour’ heuristic, it may not find the best solution.

The ‘nearest neighbour’ heuristic would be applied to a set of Prolog facts of the form:

distance(A, B, M). e.g. distance(perth, edinburgh, 47).

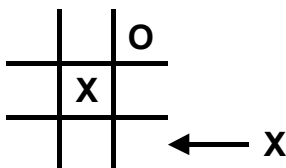
There would also be a list of all the towns not yet visited. This list would be initialised with the set of all the towns, and as each town was visited it would be removed from the list.

Our search would be to satisfy a set of sub-goals of the form:

distance(Here, Next, Distance). Here is the present location, Next is a member of the list of towns not yet visited and Distance is to be minimised.

This is an example of a **heuristic function** as it generates a numerical value that can be used to select the search path. The ‘nearest neighbour’ heuristic function attempts to find a minimum value as its optimum.

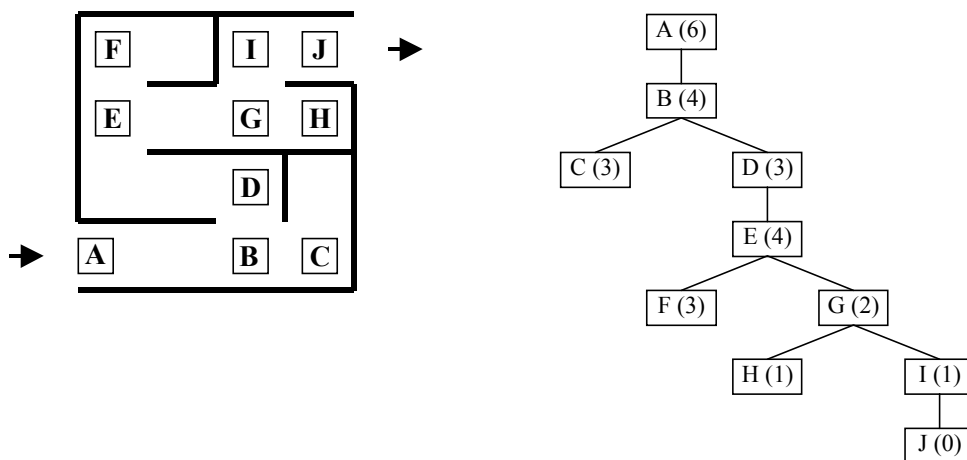
Other heuristic functions attempt to select search paths that maximise the function value. For example, in the game of noughts and crosses (tic-tac-toe), the best move will be the one that produces the maximum value based upon 1 for each line in which we could win plus 2 for each such line in which we have two counters. For example:



Playing an X as indicated would count as 1 for the possible win along the bottom line and 2 for the two counters now on the diagonal, giving a total of 3.

The diagram below shows a maze, and the heuristic function we are going to apply is called the Manhattan block distance – calculated by adding the horizontal and vertical distances from the goal position. [You may know that American cities are built as a grid pattern of streets and so this distance is the number of blocks that you have to walk to get from the current position to the goal position.]

Beside it is drawn the search tree for routes from the entrance to the maze at A to the exit at J. The values in brackets are the Manhattan block distances to J at each of the possible changes of direction.



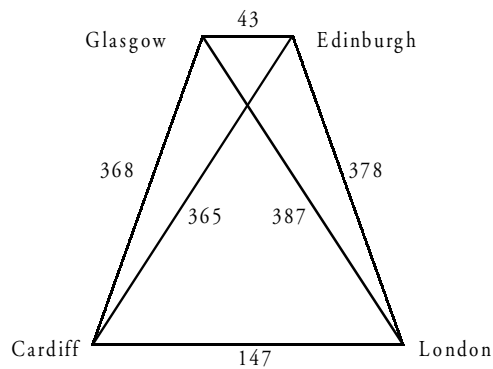
Clearly we wish to select the route that gives the minimum distance to the exit. At B and G our heuristic function does not assist our search as both alternatives evaluate to 3 and 1 respectively. But at E the heuristic function would lead us to select G rather than F.

This is another example of ‘hill climbing’ that we met in Chapter 1, where we select the branch that gives the ‘best result’.

Self Assessment Questions

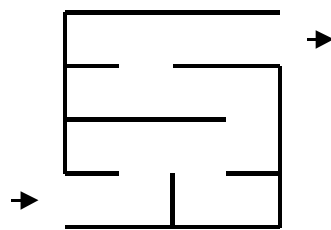
16. Consider the following diagram (not drawn to scale) showing the distances in miles between each of the four cities, Cardiff, Edinburgh, Glasgow and London. You have been asked by a transport company that is based in Edinburgh to plan an efficient route allowing deliveries to be made to warehouses in each of the other cities and returning to Edinburgh.

Draw a search tree showing all the possible routes from Edinburgh to each of the other cities and returning to Edinburgh.



Evaluate the total mileage for each route. Indicate the route that would be produced using an exhaustive search and also the route that would be produced using the 'nearest neighbour' heuristic. Compare the two solutions.

17. Draw the maze below and annotate the points at which alternative routes are possible. Draw the search tree for finding a route through the maze and use the Manhattan block distance to produce a value for each point.



Comment upon the route found by selecting the route with the minimum value of the heuristic function for this maze.

Section 3: Applications of Knowledge Representation and Search Techniques

In this section we will study how the techniques for knowledge representation and searching apply to three of the central areas of artificial intelligence research – **natural language processing, vision systems** and **robotics**. We will only be able to indicate the general methods used, as the coding of the software is beyond the scope of this course.

Natural Language Processing

Natural languages are those that are developed by the world's people in their everyday life (e.g. English, Chinese and Arabic) as opposed to languages that are invented for a special purpose (e.g. morse code, programming languages and Esperanto).

Artificial intelligence research into natural language has been stimulated by:

- i) The need to improve communication between humans and machines (e.g. natural language driven human computer interfaces) and between humans (e.g. automatic language translation systems). These systems may process natural language in a completely different way from the way that humans do it. All that matters is that the end result is that communications are understood.
- ii) The need to understand how humans process language and thus how linguistic skills may be modelled. These models are then used to test theories about linguistics.

Most of the current research uses English as its natural language but, of course, this presents a huge number of problems due to the imprecise and ambiguous nature of English and to the various assumptions that we make subconsciously when we hear or read English. Here are some examples of the problems:

Problem	Example
Language is often imprecise or incomplete in its description.	'There was a bank robbery today.' We have no information where this took place and when during today; whether the robbery was successful; whether people were injured, etc.
Language is often ambiguous due to the same word or phrase having a different meaning depending upon its context.	'The mouse was not working properly.' 'The mouse was caught in the trap.'
Language is often ambiguous due to different interpretations of the same sentence.	'Lance hated Joel because he liked Amy.' Who liked Amy?
We assume other facts that are not included in the sentence.	'The trees are shedding their leaves.' We assume that it is only the deciduous trees and not all trees that are meant and we deduce that the time of year is autumn.
Language has a very large vocabulary that is always developing and expanding.	'The rap artist faxed his agent when he couldn't get him on his mobile.'
Language allows us to express the same idea in different ways without any difference in the final meaning.	'Today is Jim's eighteenth birthday.' 'Jim was born 18 years ago today.'

Natural language processing is difficult to achieve on a computer because of the large vocabulary associated with our normal use of language, the complex grammar associated with any language and the difficulty of representing solutions to the above problems in a way that a computer can process.

If you are not sure what is meant by 'complex grammar' then read the paragraph above again!

Now that we have described the problems, let's do what early researchers in this area did – namely, restrict the problem to a limited vocabulary with a limited grammar.

Firstly, matching words is not enough. Consider the following sentences:

The boy ate the chocolate.
The chocolate ate the boy.

Obviously, for the verb ‘ate’, ‘boy’ is an acceptable subject but ‘chocolate’ is not. Though the words match exactly we must consider the structure to check the meaning. [Note that ‘boy’ may be an acceptable object – ‘The cannibal ate the boy’!]

When we consider the structure, we are considering the rules of the language – its grammar. To define the structure of a sentence, we need to use the technical terms of grammar.

Article	A word that decides the reference of a noun – a, an, the. [The word ‘determiner’ is the correct term used in text books and includes such words as ‘each’ and ‘some’.]
Noun	A naming word – man, tree, table.
Proper noun	A particular name for an individual object, it starts with a capital letter – Fred, Rome, Everest.
Noun phrase	A phrase that consists of a proper noun on its own; an article and a noun; an article, an adjective and a noun, i.e. noun phrase → proper noun noun phrase → article, noun noun phrase → article, adjective, noun.
Verb	A word for actions or states – go, write, felled.
Verb phrase	A phrase that consists of a verb on its own; a verb and a noun phrase, i.e. verb phrase → verb verb phrase → verb, noun phrase.
Adjective	A word used to modify a noun – tall, beautiful, blue.

There are many others, e.g. adverbs, prepositions, conjunctions and pronouns, but the above will be sufficient for our purposes.

In our simplified grammar, a sentence will consist of a noun phrase followed by a verb phrase (i.e. sentence → noun phrase, verb phrase). Hence the following sentences fit our grammar:

- Rome is a city.
- Rome is a beautiful city.
- The tall man wrote a long letter.
- The computer works.

We can list our grammar in a table:

Left-hand side	Rewrites as	Right-hand side
sentence	→	noun phrase, verb phrase
noun phrase	→	proper noun
noun phrase	→	article, noun
noun phrase	→	article, adjective, noun
verb phrase	→	verb
verb phrase	→	verb, noun phrase

As these are rules of grammar, it is no surprise that they are represented in Prolog as rules. These rules are recursive and involve lists. Defining them is beyond this course though you will find such definitions in text books.

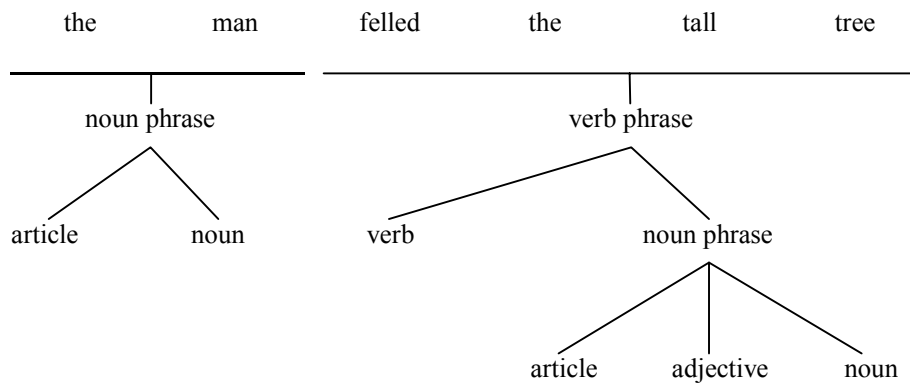
Now that we have defined our grammar, it is time to define the vocabulary and each word's part of speech. We draw up a table of the form:

Left-hand side	Rewrites as	Right-hand side
article	→	a
article	→	an
article	→	the
proper noun	→	Fred
proper noun	→	Rome
proper noun	→	Everest
...		
noun	→	man
noun	→	tree
noun	→	table
...		
adjective	→	tall
adjective	→	beautiful
adjective	→	blue
...		
verb	→	go
verb	→	write
verb	→	felled
...		

To represent the vocabulary in Prolog, we use lists so that the noun list would be:

[man, tree, table, ...]

We can now use our grammar and vocabulary to break a sentence down into its constituent parts of speech – **top-down parsing** – to produce a parse tree. We start with a sentence, split it into a noun phrase and a verb phrase, split the noun phrase, split the verb phrase and continue until we cannot split the parts any further. The end result is:



It is important to understand how we arrive at this parse tree. We match the sentence that we have with the left-hand side of a rule and replace it with the right-hand side until the right-hand side is a single part of speech. We then attempt to match the part of speech to the list of words for that part of speech. If it matches then we move on to the next item in the list. If it does not match then we backtrack and try again.

The sequence is:

- | | |
|-------------------------------------|--|
| sentence → noun phrase, verb phrase | |
| noun phrase → proper noun | Match 'the' to the list of proper nouns [Fred, Rome, Everest,...] – no match, so backtrack. |
| noun phrase → article, noun | Match 'the' to the list of articles [a, an, the,...] – match. Match 'man' to the list of nouns [man, tree, table,...] – match. |
| verb phrase → verb | Match 'felled the tall tree' to the list of verbs – no match, so backtrack. |
| verb phrase → verb, noun phrase | Match 'felled' to the list of verbs – match. |

noun phrase → proper noun	Match 'the tall tree' to the list of proper nouns – no match, so backtrack.
noun phrase → article, noun	Match 'the' to the list of articles [a, an, the,...] – match. Match 'tall tree' to the list of nouns – no match, so backtrack.
noun phrase → article, adjective, noun	Match 'the' to the list of articles [a, an, the,...] – match. Match 'tall' to the list of adjectives – match. Match 'tree' to the list of nouns – match.

If this seems familiar then it should be. This is **depth-first searching** with **backtracking** – the same method that Prolog employs.

Many versions of Prolog come with demonstration files that include a simple parser. Ask your teacher/lecturer if there is any software available which will let you see the implementation of a simple parser.

We have only touched the surface of the considerable amount of research that has taken place into natural language processing. For example, we have not considered the stage after parsing where the meaning of the sentence is determined by analysis of the relationships of the words as parts of speech.

Already we have grammar checkers included in word processing software and programs that can translate between languages. The next few years will see further developments and products arising from natural language processing research becoming available to the general public.

Self Assessment Question

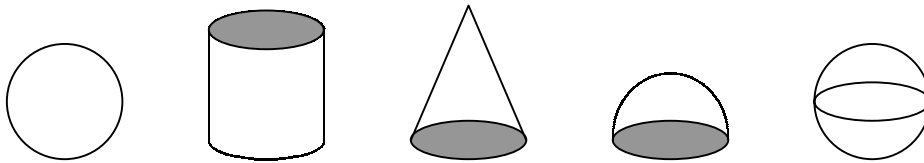
18. Parse each of the following sentences. Assume that all the words are in our lists of vocabulary.
- i) Joan drove the new car.
 - ii) The young child watched the goldfish.
 - iii) The old woman caught the red ball.

Vision Systems

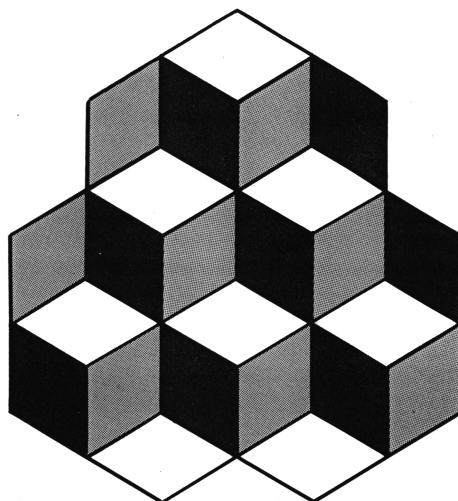
Visual interpretation is something that we take for granted. When we look at a complex scene, we instantly make sense of the shapes; the shading; lighting and colour; the perspective; the distance of each object and recognise objects even when they are partially hidden. All of this happens subconsciously in a fraction of a second.

In addition, our visual ability extends effortlessly to interpret changing scenes such as those that confront car drivers every day.

Artificial intelligence research in the area of vision systems has had to overcome the problems of interpretation of complex scenes from a two-dimensional image and with the problems of ambiguity. For example, if we have a circle, should it be interpreted as the top surface of a cylinder, the bottom face of a cone, the base of a hemisphere or the outline of a sphere?



Another example of ambiguity is provided by the numerous optical illusions that produce two different interpretations depending on how you look at the picture.

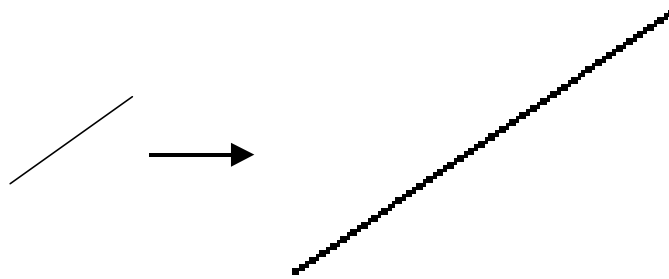


How many cubes do you see?

The first stage in any vision system involves **digitisation**.

The image source may be a digital camera, a video camera, a scanner (e.g. for optical character recognition or scanning a photograph) or may be a satellite photograph. In the case of a video image, though digital video is becoming more common, it is likely that the analogue image will require to be converted, using a video digitiser, to a digital image.

The problems at this stage are connected with the quality of the image. This can be affected by electronic distortion, dust on the lens or poor lighting. In addition, the digital image may not depict the edge of a black rectangle on a white background as a simple transition from black to white but rather as black, a shade of grey and then white. An edge at an angle may not appear as straight edge but rather as a saw tooth line.



The second stage is **signal processing** where the computer performs calculations on each pixel in the picture in order to enhance the quality of the image. This in turn leads to the third stage – **edge and region detection**.

This involves the computer defining and locating objects and areas. The values of adjacent pixels are compared and so edges are detected. In both these stages there are a number of mathematical methods employed to analyse and interpret the pixels.

However, though these are very interesting topics to study, there is no real application of artificial intelligence methods until the fourth and fifth stages of a vision system that involve **object recognition** and **image understanding**.

Object recognition involves matching the pattern of edges and regions to stored patterns to identify individual objects in the whole image. Image understanding is putting together the individual objects, found at the fourth stage, into a comprehensible scene. Only then can our computer react to the scene.

Again let us consider a car driver. The driver can recognise individual objects such as other vehicles, pedestrians, traffic lights, buildings, etc., but only the relative positions of these objects and changes to their positions can determine whether the driver decides to brake sharply, slow down or continue.

A great deal of the research into computer vision has been based on problems that are known as block worlds. These are restricted environments where a variety of three-dimensional objects (cubes, cuboids, pyramids, spheres, etc.) are stacked or placed near each other. The computer vision system uses a camera focused on the scene. It then acquires, processes, analyses, recognises and understands the image.

What we require is a method of describing each object in our image and then to compare this description with a stored set of descriptions to obtain a match for our object.

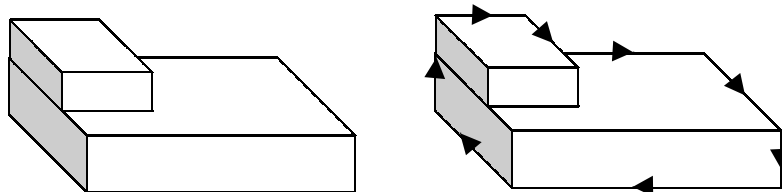
Let us look first at how a three-dimensional object can be described once we have determined its edges, i.e. the lines that form the shape. This is called a **primal sketch**.

The Waltz Algorithm

This is a method of labelling edges according to whether they are concave edges (marked with a +), convex edges (marked with a -) or obscuring edges (marked with an arrow). The direction of the arrow on an obscuring edge indicates that the face in view is to the right while the obscured face is to the left.

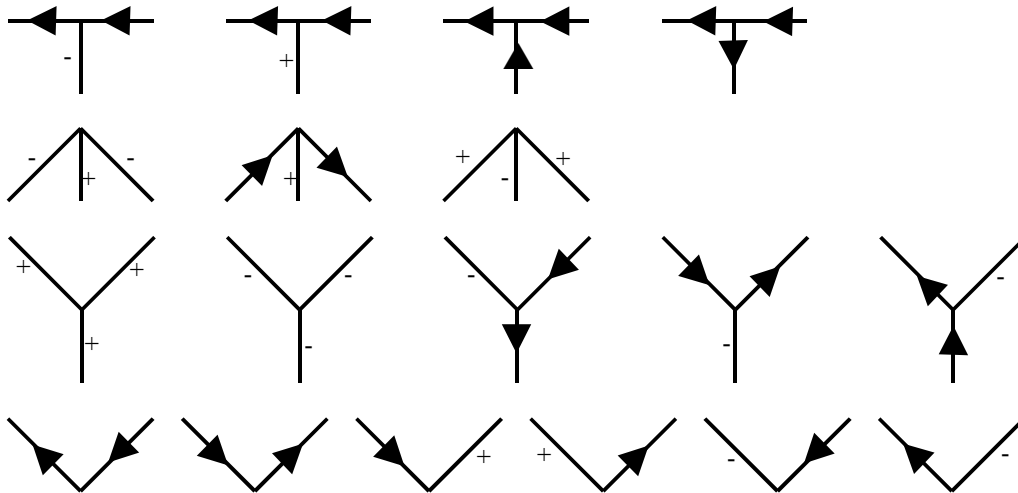
But how do we get a computer to decide what labels to put on which lines?

The Waltz Algorithm starts from outside the object and works inwards. The boundary lines of the object obscure the background and so all of these lines can be marked by a set of clockwise arrows as the face in view will be to the right and the obscured face to the left.



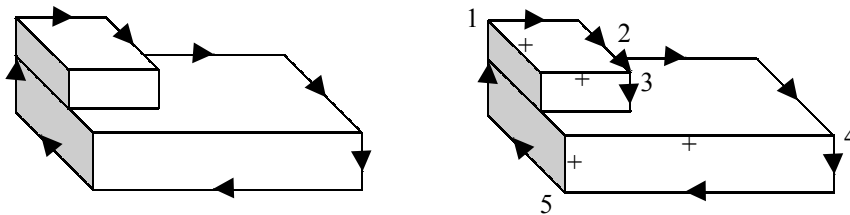
How do we now get the computer to decide on concave or convex edges?

The lines that form the faces meet at vertices where three faces have a common point – hence the name trihedral vertex. There are 18 possible trihedral vertices that can be categorised as T junctions, arrows, forks (Y junctions) and L junctions.



We now inspect the currently labelled lines at each vertex and decide which of the above 18 possible patterns our vertex fits. It may not be possible to categorise a vertex at first, but another vertex will supply information to categorise on a second or later repetition.

Let us see how that works for our shape starting in the top left corner.



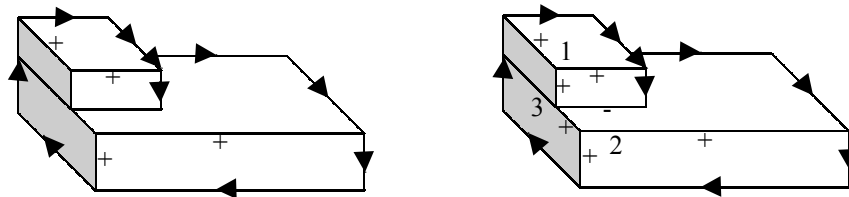
At 1 we have an arrow vertex with two arrowed boundary lines. This must be arrow 2 above and so we place + on the middle line of the arrow.

At 2 we have a T junction with 2 arrows. This must be T junction 4 above and hence we put a third arrow.

At 3 we have an arrow vertex with one arrow. This must be arrow 2 above and so we place an arrow and a +.

At 4 and 5 we have an arrow vertex with two arrows. This must be arrow 2 above and so we place + on the middle line of the arrow.

This completes the outside vertices. We now repeat the process, attempting to categorise the remaining vertices.



At 1 and 2 we have a fork (Y junction) with two +s. This must be fork 1 above and so we place a + on the remaining line.

At 3 we have an arrow vertex with two +s. This must be arrow 3 above and so we place a - on the remaining line.

This completes our diagram.

When successful matches are found, 'recognition' occurs, that is, the computer identifies what it is looking at. This process is continued and the computer builds up a complete picture of the scene by matching each vertex and junction. By linking groups of vertices and using the rules (about particular type(s) of vertex associated with a particular shape) stored in its knowledge base, the system will be able to conclude that it has identified a cube, a cuboid or a pyramid, etc.

You probably realise that, as yet, vision systems have limited applications. The reason for this is the huge amount of information on objects and the processing required to identify them when we look at everyday scenes.

Imagine the complexity in simply extending the scope and trying to get a vision system to understand the scene in a room in your house. There would be a much greater number of objects (TV, video, settee, chairs, table, pictures on wall, plants, books, etc.) so the primal sketch would be very complex. Thus a computer would have far more difficulty in picking out the various objects in the scene. If the vision system is to be used outside there will be further complications as objects like trees, bushes and animals will be very difficult to identify due to curved, irregular surfaces and the lack of hard edges.

There are many current and potential applications.

Manufacturing:

- parts inspection for quality control
- assembly, sorting, locating and packaging
- guidance of robots.

Medical:

- screening X-rays, e.g. for breast cancer
- checking slides from medical tests, e.g. checking tissue samples for cancer
- checking ultrasound images, e.g. babies in the womb.

Military:

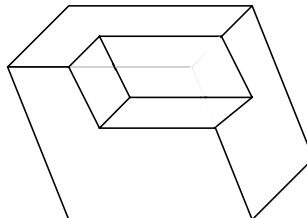
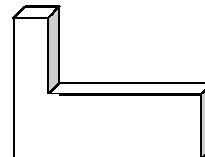
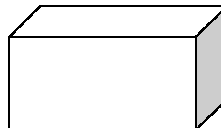
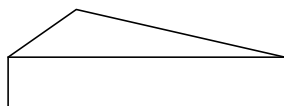
- weapons guidance
- photo and satellite reconnaissance and analysis.

Science:

- analysis of astronomical images, e.g. to find particular types of star
- analysis of satellite weather images.

Self Assessment Question

19. Apply the waltz algorithm to label the lines and determine whether other lines are concave or convex.



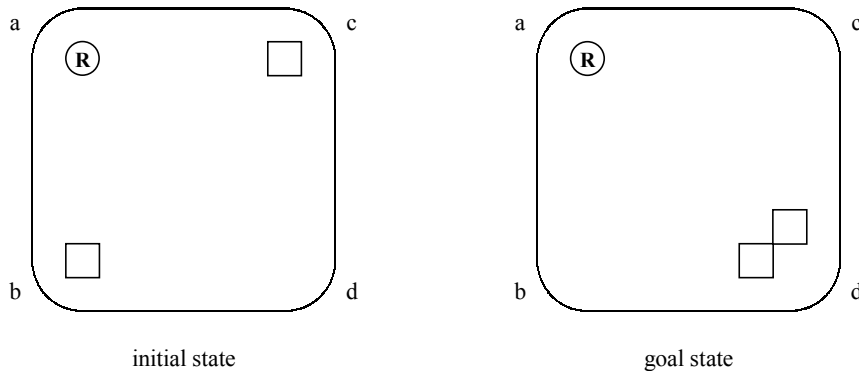
Robotics

Robotics is the third application of knowledge representation and search techniques in artificial intelligence that we will study. Robotics is the branch of artificial intelligence research concerned with enabling computers to react intelligently to objects in the surrounding environment.

An intelligent robot will have sensors providing feedback regarding pressure, light intensity, temperature, liquid levels or other physical characteristics. These sensors provide feedback that must then be processed by the controlling software and it is this software that uses the techniques of artificial intelligence.

These problems resolve into representing the initial state of the robot, the objects that it has to manipulate and the environment that it is in (e.g. the layout of the factory). We then represent the goal state and search the rules for altering the scene to find a sequence of actions that solve the problem. This set of actions is a plan to achieve the goal. Hence this area of research is frequently called **planning**.

An example of a planning problem is one where we have a robot in some initial position, boxes that the robot can move, initial positions and final positions for the boxes and the robot.



The task is to use the robot at 'a' to move the boxes at 'b' and 'c' to 'd' and to return the robot to 'a'.

At first this problem appears very similar to the state space problems that we encountered in Chapter 1, but the differences here are that we must specify preconditions for each action and the resultant state after each action. Also we do not go from the initial state branching through all the possibilities to try to reach the goal state, but rather we start at the goal state and work back to the initial state.

The initial and goal states can be represented by facts of the form:

initial state	goal state
at(a, robot).	at(a, robot).
at(b, box1).	at(d, box1).
at(c, box2).	at(d, box2).

We can move the robot on its own and also move the robot while it carries a box. These can be defined as follows:

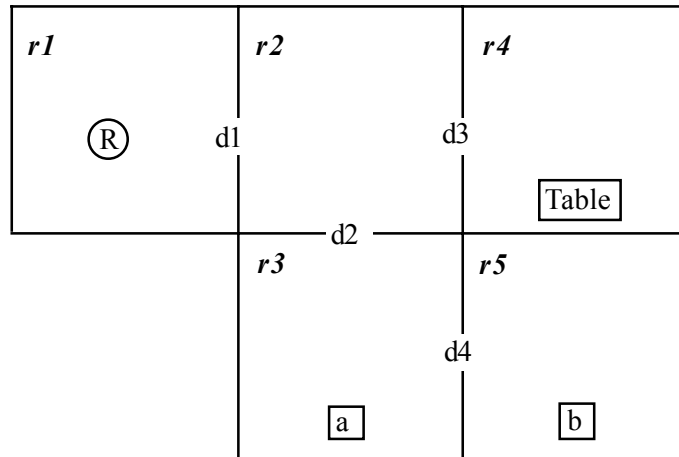
Action	move_robot(X, Y).	carry_box(B, X, Y).
Precondition	at(X, robot).	at(X, B). at(X, robot).
Resultant change to facts in the state.	DELETE at(X, robot). ADD: at(Y, robot).	DELETE at(X, B). at(X, robot). ADD: at(Y, B). at(Y, robot).

Starting at the goal state, we formulate the previous state by a systematic search of the operations. In this case, the two previous states would be:

at(d, robot).	from move_robot(a, d).
at(d, box1).	
at(d, box2).	
at(b, robot).	from carry_box(box1, d, b).
at(b, box1).	
at(d, box2).	

We continue in this way until we reach the initial state and the series of actions can now be reversed to form our plan to achieve the task.

Obviously, our 'world' is rather simple. Here is a more complex 'world' consisting of five rooms (r1 to r5) with some connecting doors (d1 to d4) and two boxes (a and b) that must be placed by the robot on the table in room 4.



The facts and actions that we might use are:

Facts	connects(d1, r1, r2).	in_room(boxa, r3).
Action	go_through_door(D, X, Y).	carry_box(B, D, X, Y).
Precondition	in_room(robot, X). connects(D, X, Y).	in_room(B, X). in_room(robot, X). connects(D, X, Y).
Resultant change to facts in the state	DELETE in_room(robot, X). ADD: in_room(robot, Y).	DELETE in_room(B, X). in_room(robot, X). ADD: in_room(B, Y). in_room(robot, Y).

In industrial applications, static robots are equipped with sensors to allow the controlling computer to detect and react to certain conditions, e.g. a paint-spraying robot may have a light sensor to detect when the object is correctly positioned and so start the paint spray cycle. Mobile robots are equipped with sensors to follow pre-set paths on the factory or warehouse floor and the controlling computer's program has data to determine the available paths at each junction.

Neither of these situations requires artificial intelligence as the environment is carefully managed to be identical on every occasion that the robot carries out a task.

However, if a static robot has objects arriving for processing at different angles or of various types then a vision system linked to artificial intelligence software will allow it to react correctly to the varying situations.

Similarly, if a mobile robot is moving in an unspecified area and carrying out ill defined tasks then, again, the combination of a vision system and artificial intelligence software will allow it to 'map' its environment and to determine the nature of the task and the actions required.

For example, a robot vehicle in a deep mine or on the seabed will not have white lines on a smooth concrete floor to follow and may meet obstructions that it must find its way round and not merely stop until a human supervisor removes the obstruction.

Self Assessment Question

20. In a square room, there is a robot at corner A and three boxes at corners B, C and D. The task is to move all three boxes into corner B and return the robot to corner A.
- i) Draw diagrams of the room at the start and at the end.
 - ii) Produce a simple Prolog representation of the initial and goal states.
 - iii) Define actions, preconditions and resultant changes to move the robot from one corner to another and to carry a box from one corner to another.
 - iv) By working back from the goal state describe a set of the previous three states required.
 - v) Produce a plan to carry out the task. List the action used and the resultant changes at each step of your plan.
 - vi) Discuss how many plans would satisfy the task description. What choices do you have?

Chapter 2: Key Points

- A simple domain of knowledge can be represented by a **search tree**. This is a diagram derived from the state space. The search tree has an initial or starting state, linked through several intermediate states to a goal state.
- **Logic** was one of the first methods of representing knowledge used in artificial intelligence. Logic is the scientific study of the process of reasoning and the system of rules that are used in the reasoning process.
- **Semantic networks** are another way in which knowledge can be represented. Semantic networks use nodes to represent objects or events and these nodes are connected by lines that show relationships. Semantic networks are particularly useful when the knowledge can be classified into groups, and objects in these groups inherit properties.
- A **frame** is a knowledge representation method in which the data structure has components called slots. Frames are used to represent large sets of facts in a structured way in a system that allows facts to be retrieved directly, or inferred using inheritance.
- All of the above **representations can be converted into Prolog facts** and the relationships and inheritance can be defined by **Prolog rules**.
- An **exhaustive search** is a systematic problem-solving technique in which every possible route to a solution is examined. It is a costly and time-consuming method but will always find a solution if one exists.
- A **depth-first search** is an exhaustive search that examines all problem states, by first going to deeper problem states rather than those on the same level, and then backtracking, if required, until a goal state is found.
- A **breadth-first search** is an exhaustive search that examines all states in a search tree one level at a time, from left to right and top to bottom. Beginning with the initial state, a breadth-first search looks at all the states on each level before proceeding to the next lower level.

- The extremely large number of search state possibilities often makes an exhaustive search impractical for non-trivial problems. In most cases, the number of possible combinations that would need to be considered increases very rapidly and will exceed the capacity of the computer system. This is called a **combinatorial explosion**.
- Depth-first and breadth-first searches both have their advantages and disadvantages and both can be coded in Prolog.
- A **heuristic search** uses a hint, a trick, a rule of thumb, or knowledge about the state space to focus or limit the search process in an effort to find the solution faster. A heuristic function often provides a numeric value for a state and the search progresses by selecting the path that maximises or minimises this function.
- **Natural language** refers to the way in which people communicate verbally or in writing with one another in English, French or German, etc. Special artificial intelligence techniques have been developed which allow people, using their own natural language, to communicate with computers. This is very complex due to the vast number of words and the range of acceptable grammars.
- The most powerful natural language programs analyse the syntax and semantics of a sentence that has been input. Pattern matching techniques enable the syntax or grammar of a sentence to be determined by a **parser** that separates a sentence into its parts of speech (noun phrase, verb phrase, article, noun, verb, adjective, etc.). The parser works along with a dictionary of known words. Finally, the meaning of a parsed sentence would be determined (by semantic analysis).
- **Vision** involves five stages – **digitisation, signal processing, edge and region detection, object recognition and image understanding**.

Image acquisition involves capturing a scene on a camera, TV or scanner and converting it into a digitised format for processing by the computer.

Signal processing is carried out by the computer performing calculations on each pixel in the picture.

Edge and region detection involves the computer defining and locating objects and areas. The values of adjacent pixels are compared and so edges are detected. Using this data, the computer produces a primal sketch.

The final stages of the vision process, object recognition and image understanding are the only ones dependent on artificial intelligence. For example, object recognition uses classification, such as the Waltz Algorithm, and pattern matching to identify objects in a scene.

- An artificial intelligence program in a computer controlling a **robot** can give it the intelligence to operate independently in an unrestricted domain.
- **Planning** involves determining the state space, the robot's rules of movement within that space and the preconditions and resultant changes from these movements. The plan is found by working back from the goal state to the initial state in order to eliminate fruitless paths.

Chapter 2: Self Assessment Questions – Solutions

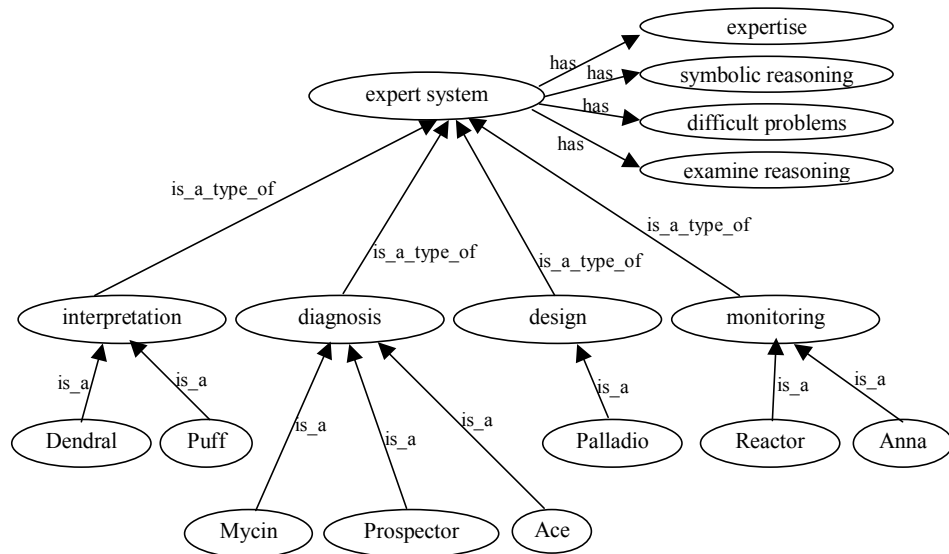
1. 24. There are 4 towns at level 1. Each of these splits to 3 towns which in turn split to 2 towns each (and finally 1 town each). Hence there are $4 \times 3 \times 2 \times 1$ routes. [If you have done permutations in Mathematics then you will recognise this as $4!$ – 4 factorial.]
With 6 towns there would 5 towns at level 1 and so $5 \times 4 \times 3 \times 2 \times 1 = 120$ routes.
With 50 towns we would have a combinatorial explosion.
2. The route would be A to B (30); B to C (35); C to D (35); D to E (25) and E to A (40) giving a total distance of 165 – this is the left-hand branch of the search tree.
3. `connection_up(A, B) :- direct_up(A, B).`
`connection_up(A, B) :- direct_up(A, X), connection_up(X, B).`
4. Using the ‘nearest neighbour’ heuristic, the route would be A to E (40); E to D (25); D to B (60); B to C (45) and C to A (150) giving a total ‘cost’ of 320.
The route round the perimeter would be A to E (40); E to D (25); D to C (80); C to B (45) and B to A (50) giving a total ‘cost’ of 240. This shows that a heuristic, though it will save search time, will not necessarily give the optimum solution.
5.
 - 1 likes(james, X) if book(X)
 - 2 book(thriller)
 - 3 book(biography)
 - 4 book(X) if time_to_read(X, T) and $T > 3$
 - 5 time_to_read(treasure_island, 6)
 - 6 read(jennifer, treasure_island)
 - 7 likes(susan, X) if likes(jennifer, X)
 - i) Using 3 biography is a book; using 1 James likes biography.
 - ii) Using 5 the time to read *Treasure Island* is 6; using 4 *Treasure Island* is a book; using 1 James likes *Treasure Island*.
 - iii) We can only state that Susan likes the same books as Jennifer from 7. We have no knowledge whether Jennifer liked *Treasure Island* – it may have been a set book for English!

The problem with knowledge bases expressed using logic statements is that a large number of statements are required to describe all the knowledge in a particular situation and the knowledge engineer does not know how much to include.

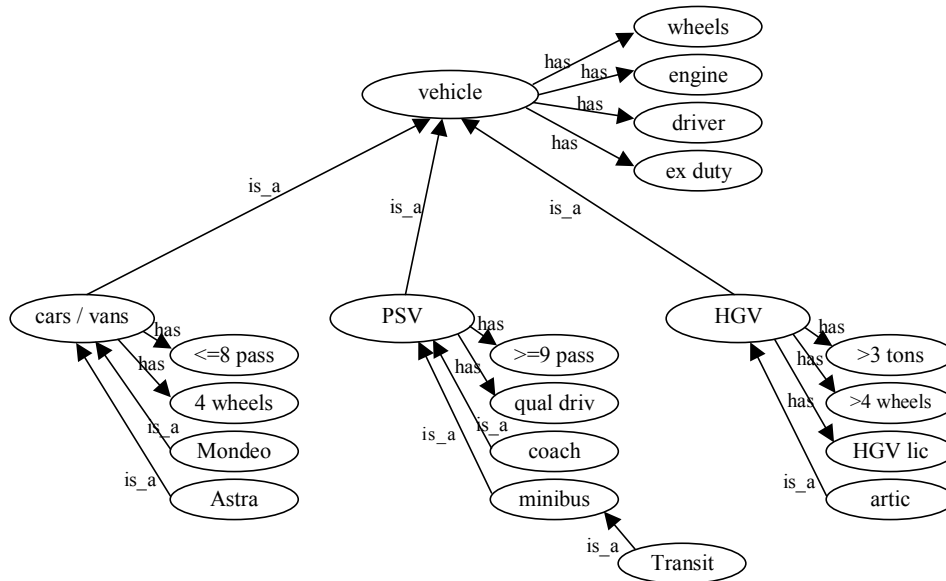
6. 1 likes(fraser, easy)
- 2 course(intermediate_2, hard)
- 3 course(higher, hard)
- 4 course(intermediate_1, easy)
- 5 level(computing_studies, intermediate_1)

This time we will use forward reasoning. Using 1 Fraser likes easy courses; using 4 Intermediate 1 courses are easy; using 5 Computing Studies is an Intermediate 1 course. Hence Fraser should do Computing Studies.

7. The semantic net could be laid out as shown:



8. The semantic net could be laid out as shown:



9. `has(vehicle, wheels).`
`has(vehicle, engine).`
`has(vehicle, driver).`
`has(vehicle, excise_duty).`
`has(car, <=8_passengers).`
`has(car, 4_wheels).`
`has(psv, >=9_passengers).`
`has(psv, qualified_driver).`
`has(hgv, >3_tons).`
`has(hgv, >4_wheels).`
`has(hgv, hgv_licence_driver).`
`is_a(car, vehicle).`
`is_a(psv, vehicle).`
`is_a(hgv, vehicle).`
`is_a(mondeo, car).`
`is_a(astra, car).`
`is_a(coach, psv).`
`is_a(minibus, psv).`
`is_a(transit, minibus).`
`is_a(articulated_lorry, hgv).`

`connect(X, Y) :- is_a(X, Y).`
`connect(X, Y) :- is_a(X, Z), connect(Z, Y).`

`has(Y, X) :- connect(Y, Z), has(Z, X).`

10.

Microcomputer	
Processor	Pentium II
Speed	350 MHz
Memory	64 Mb
Network card	Yes
Input devices	Keyboard Mouse
Output devices	Monitor Loudspeakers Printer
Backing Storage devices	Floppy disk drive Hard disk drive CD-ROM drive

Printer	
Use	Hard copy
Types	Ink-jet Laser

Ink-jet	
Speed	Slow
Colour	Yes
Purchase cost	Low
Running cost	High

Laser	
Speed	Fast
Colour	No
Purchase cost	High
Running cost	Low

11.

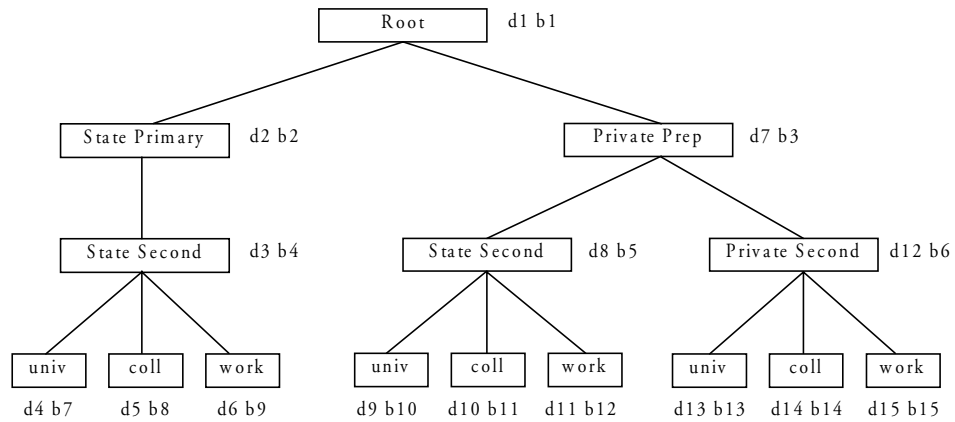
Canada	
Provinces	10
Westernmost	British Columbia
Location	N part of North America
Area	Almost 10 million sq kms
Population	26 million
Languages	English French
Currency	Canadian dollar

British Columbia	
Area	950,000 sq kms
Population	3.5 million
Capital	Victoria
Main city	Vancouver
Resources	Minerals Timber Fisheries

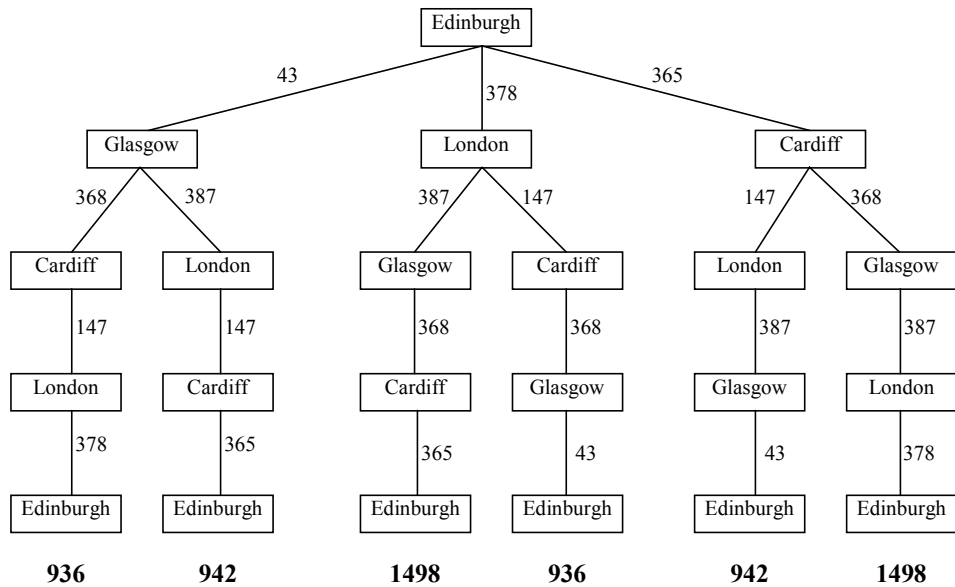
Minerals	
Obtained	Deep mining Quarrying
Used directly	Sand and gravel
Extracted	Gold Copper Iron

12. [v, i, m] from 9; ...[r, v, i, m] from 6 and success reported; ...[v, i, m] from 9; [d, v, i, m] from 6; ...[v, i, m] from 9; ...[i, m] from 9; ...[m] from 9; ...[n, m] from 6; ...[c, n, m] from 6; ...[n, m] from 9; ...[f, n, m] from 6; ...[x, f, n, m] from 6; ...[f, n, m] from 9; ...[z, f, n, m] from 6; ...[f, n, m] from 9; ...[n, m] from 9; ...[m] from 9; ...[] from 9 and ends.
13. [[m, i], [m, n], [m, k, b], [m, k, h], [m, k, u], [m, i, v]] from 6;
 [[m, n], [m, k, b], [m, k, h], [m, k, u], [m, i, v]] from 9;
 [[m, n], [m, k, b], [m, k, h], [m, k, u], [m, i, v], [m, n, c]] from 6;
 [[m, n], [m, k, b], [m, k, h], [m, k, u], [m, i, v], [m, n, c], [m, n, f]] from 6;
 [[m, k, b], [m, k, h], [m, k, u], [m, i, v], [m, n, c], [m, n, f]] from 9;
 [[m, k, h], [m, k, u], [m, i, v], [m, n, c], [m, n, f]] from 9;
 [[m, k, u], [m, i, v], [m, n, c], [m, n, f]] from 9;
 [[m, i, v], [m, n, c], [m, n, f]] from 9;
 [[m, i, v], [m, n, c], [m, n, f], [m, i, v, t]] from 6 and success reported.
14. i) Chess gives rise to a combinatorial explosion because there are a large number of branches available at each of many levels as there is a level for each move.
- ii) This will depend on the length and complexity of the journey that you make and also on whether or not you use a heuristic of the form, 'Take the road that is closest to the direction to the school.' Obviously there are an infinite number of routes if instead of the usual two-mile route, the journey takes in visits to Rome, Tokyo and New York! Another method to cut down the number of possible routes would be to identify a point or points that are required on the route (such as a bridge) and then find a route to that intermediate point.
- iii) There are only two possibilities: finding the hypotenuse using $h^2 = a^2 + b^2$ or finding a short side using $a^2 = h^2 - b^2$.
- iv) The fine detail of building a house, when you consider every nail, brick and piece of wood, would lead to a combinatorial explosion.

15. This is a possible search tree layout:



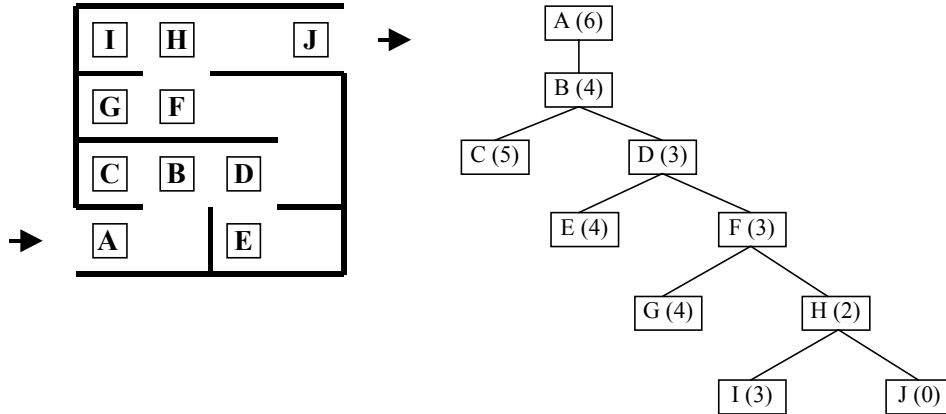
16.



Using an exhaustive search, either depth-first or breadth-first, will require all the branches to be evaluated to ascertain which is the minimum. The best route will be Edinburgh, Glasgow, Cardiff, London and Edinburgh (or its reverse).

Using the 'nearest neighbour' heuristic, the same route will be found by exploring only one branch of the tree.

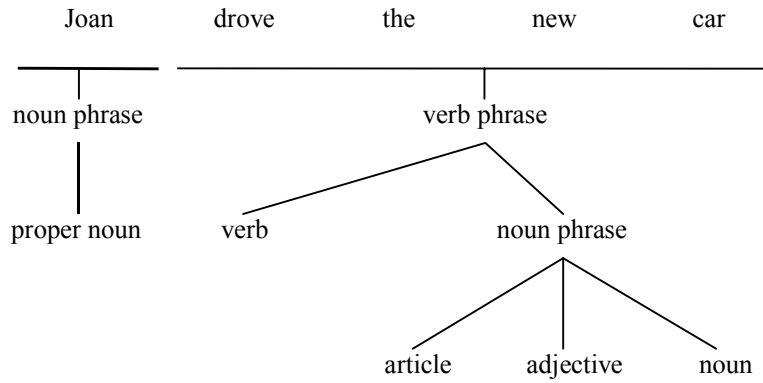
17. The points A to J should be marked and the search tree shown produced.



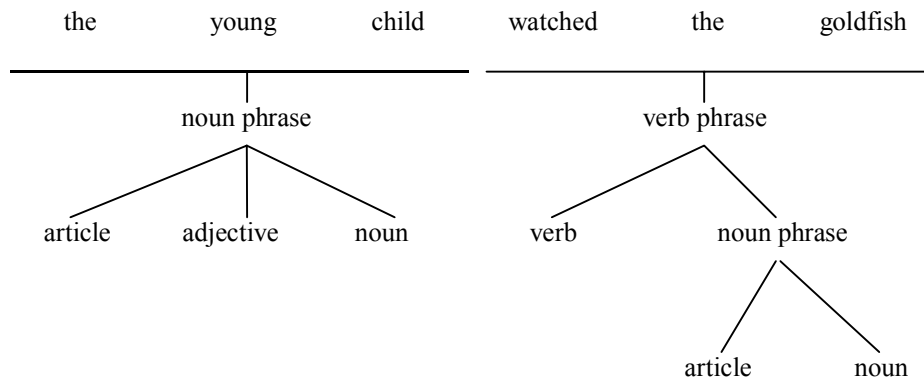
If each time we select the next node to be the one with the lower Manhattan distance then we obtain the best route through this maze.

18. The parse trees would be:

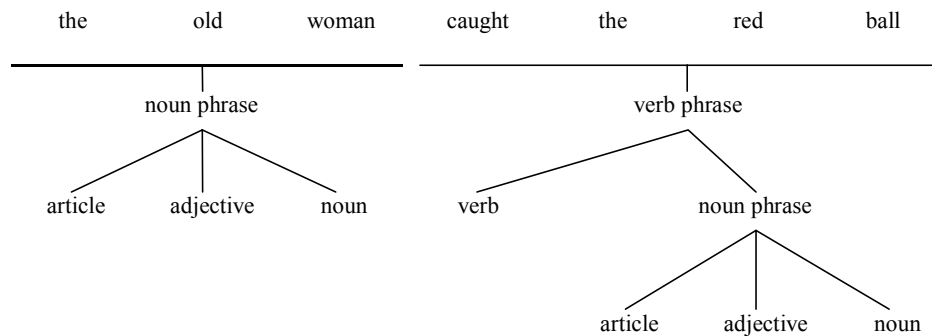
(i)



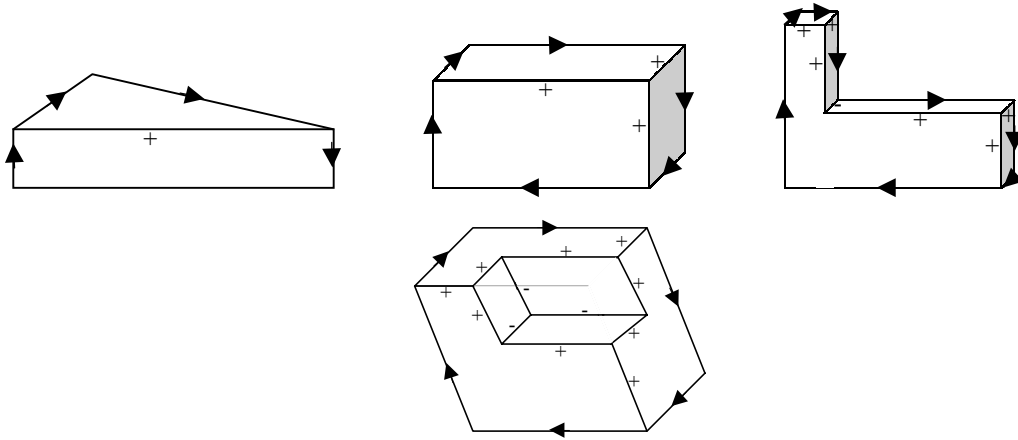
(ii)



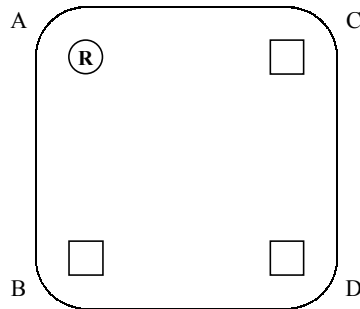
(iii)



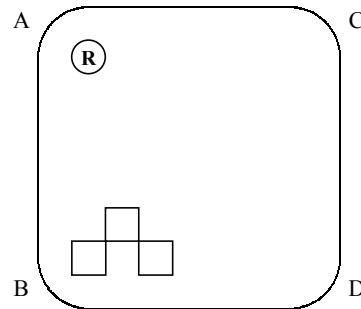
19. The final labels should be:



20. i) As below or similar.



initial state



goal state

ii) at(a, robot).
at(b, box1).
at(c, box2).
at(d, box3).

at(a, robot).
at(b, box1).
at(b, box2).
at(b, box3).

iii) Action move_robot(X, Y).

Precondition at(X, robot).

Result DELETE at(X, robot).
 ADD at(Y, robot).

carry_box(B, X, Y).

at(X, B).
at(X, robot).
DELETE at(X, B).
 at(X, robot).
ADD at(Y, B).
 at(Y, robot).

iv) at(b, robot). from move_robot(a, b).
 at(b, box1).
 at(b, box2).
 at(b, box3).

at(d, robot). from carry_box(box3, b, d).
 at(b, box1).
 at(b, box2).
 at(d, box3).

at(b, robot). from move_robot(d, b).
 at(b, box1).
 at(b, box2).
 at(d, box3).

This is the order if we move box3 last.

v) The complete plan, moving the boxes in the order box2 then box3 to B would be.

move_robot(a, c).
 carry_box(box2, c, b).
 move_robot(b, d).
 carry_box(box3, d, b).
 move_robot(b, a).

vi) We could move the boxes in the order box3 then box2 with the same efficiency.

There is an infinity of possible plans where we move a box back and forth to B or move boxes to other corners apart from the target one. For example, we could move box2 to A then to D then back to C or move box1 from B to another corner. All of these would be inefficient.

Appendix 2.1

Recursion

In Mathematics the number of ways that n objects can be ordered is given by **$n!$ – n factorial** (i.e. if 6 people run a race then there are 6! finishing orders).

$$6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1$$

This can be explained by there being 6 people who can be first, but once there is a winner then there are only 5 possibilities for second place, and then there are only 4 possibilities left for third place and so on.

Notice the pattern:

$$2! = 2 \times 1 = 2 \times 1!$$

$$3! = 3 \times 2 \times 1 = 3 \times 2!$$

$$4! = 4 \times 3 \times 2 \times 1 = 4 \times 3!$$

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 5 \times 4!$$

$$6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 6 \times 5!$$

We can now define the factorial rule recursively by:

$$1! = 1 \quad \{\text{special case to stop recursion}\}$$

$$n! = n \times (n-1)! \quad \{\text{general case}\}$$

We work out this rule by using the general case to reduce the problem to the special case and then to rebuild to the value that we require. For example, to calculate 6!:

$$6! = 6 \times 5! \quad 5! = 5 \times 4! \quad 4! = 4 \times 3! \quad 3! = 3 \times 2! \quad 2! = 2 \times 1!$$

but $1! = 1$ so rebuild

$$2! = 2 \times 1 = 2 \quad 3! = 3 \times 2 = 6 \quad 4! = 4 \times 6 = 24 \quad 5! = 5 \times 24 = 120$$

and finally $6! = 6 \times 120 = 720$.

In Prolog, rules are often defined in this very powerful way. These **recursive rules** always consist of two parts:

the **special case** or **terminating case**

and

the **general case** or **recursive case**.

Let us look at the **'family.pl'** program where we have now defined a number of relationship rules.

We want a new rule **'ancestor'** which in its simplest case is:

A is an ancestor of B if A is the parent of B

i.e.

ancestor(A, B) :- parent(A, B). {special case}

But of course grandparents, great-grandparents, great-great-grandparents ... are all ancestors.

We could try to chain the 'parent' rule in the form:

A is an ancestor of B if A is the parent of Z and Z is the parent of Y and Y is the parent of ... is the parent of B.

But how many times to do we need the 'parent' relationship?

Here is the recursive alternative:

A is an ancestor of B if A is the parent of X and X is an ancestor of B.

i.e.

ancestor(A, B) :- parent(A, X), ancestor(X, B). {general case}

Tasks

1. Load the **'family.pl'** program. Look at the facts and from them produce a family tree on paper. This will allow you to check the results when testing your rules.

Define a rule 'parent' where parent(A, B) means that A is the parent of B.

Add the 'ancestor' rule (remember both parts – the special case and the general case). Use **Options -> Trace** and follow what happens with:

- i) ancestor(X, colin).
- ii) ancestor(X, james).
- iii) ancestor(X, andrew).
- iv) ancestor(X, natalie).

Make sure that you understand what is happening. Try more goals if you wish.

2. Define a rule 'offspring' where:

offspring(A, B) means that A is the offspring of B, i.e. B is the parent of A.

You will need this rule for the next question.

3. Equivalent to 'ancestor' is the relationship 'descendant'. Define a recursive rule for 'descendant(A, B)' to give 'A is a descendant of B'.

[Hints: Think of the special case – opposite of the special case for 'ancestor'. Think of the conditions for the special case – again the opposite of 'ancestor'.]

Save the whole program so far (including 'ancestor', 'offspring' and 'descendant') as 'family.pl'.

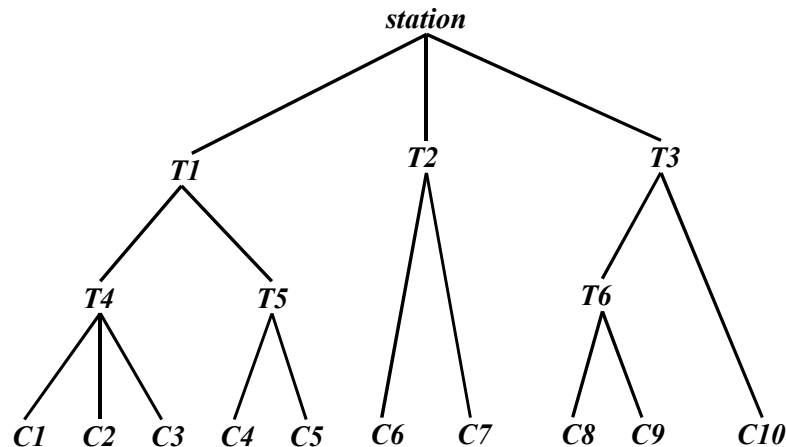
4. Add another generation to the family tree. Make up your own names for the marriage partners and offspring of some of Michael, Natalie, Ophelia, Patricia and Quintin. [Names starting with R, S, T ... will help.]

Test your extended program with goals involving all the rules defined so far. Predict what the results should be and check that the run gives the same results.

Save this program as 'extragen.pl' in your own directory.

A Power Distribution Network

Electricity consumers in a district are supplied with electricity from an electricity generating station. This power is distributed from the station to the various consumers through a system of transformers, as shown below:



C1 to C10 are consumers; T1 to T6 are transformers. Occasionally one of the transformers will malfunction or require servicing. If this should happen, the management would naturally wish to know which consumers will be affected.

The fact that one point in the distribution network directly feeds another point is shown on the diagram by a line connecting the two points. In Prolog, we write:

```
feeds(station, t1).    {and so on for all the connections}
```

We must also distinguish between consumers, transformers and the generating station:

```
generator(station).
transformer(t1).      {and T2 to T6}
consumer(c1).        {and C2 to C10}
```

The predicate 'feeds' refers only to direct connections. But one point can feed or supply another indirectly. For example, T1 supplies T4 and T5 and so consumers C1 to C5.

We want to define a rule:

```
supplies(X, Y)      meaning X supplies Y either directly or indirectly
```

This is a recursive relation.

Tasks

5. Load the file '**supplies.pl**' which contains the facts for the electricity generating network.

Define the recursive rule 'supplies' outlined above.

Decide on the special case (direct connection between points) and then on the general case (indirect connection) where the two points have one or more intermediate points.

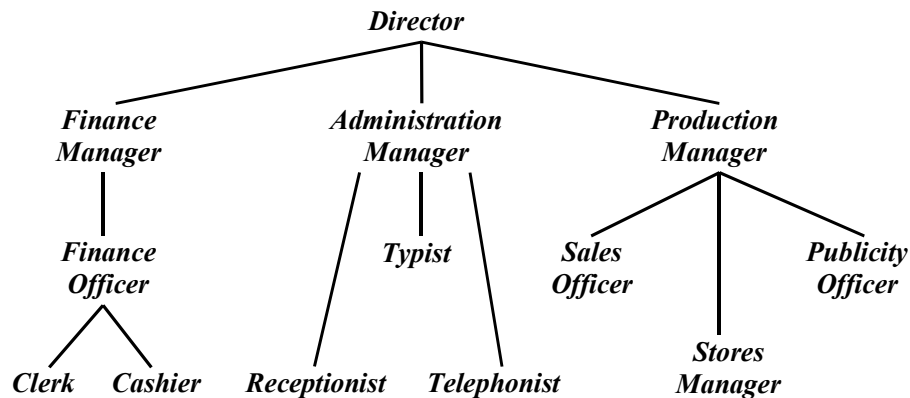
Test your rule with the goal 'supplies(t1, X)'.

Unless you've been very clever, you may find that the list of solutions also contains transformers. Add to the rule 'supplies' so that it will only list consumers.

6. Alternatively, a consumer (C8, say) may report a power loss and the power company wants to find out which transformer supplies this consumer.

Alter the rule 'supplies' so that it will provide solutions to the goal 'supplies(T, c8)'.

7. The diagram shows the management structure for a firm. The fact that one person works for another is indicated in the diagram by one person being directly below the other.



In Prolog, we will use the predicate:

`works_for(A, B)`

Load the file '**company.pl**'.

Define Prolog rules for:

- i) X is the immediate boss of Y.
- ii) X manages Y in the firm (either directly or indirectly).
- iii) X is managed by Y in the firm (either directly or indirectly).

The Order of Clauses

We have now met the representation of facts and the construction of simple, compound and recursive rules in Prolog.

Here are a few potential problems:

Does it matter in which order we write the clauses?

Does it matter in which order we write the sub-goals or rules?

Tasks

8. Load your final version of **'family.pl'** which includes the **'ancestor'** rule.

Now swap the two parts of the **'ancestor'** rule, i.e. place the general case before the terminating case to give:

```
ancestor(A, B):-parent(A, X),ancestor(X, B).
ancestor(A, B):-parent(A, B).
```

Now try the goals given in Task 1.

Also try the goal: `ancestor(X, Y)`.

What has been the effect of reversing the two parts of the rule?

9. Return the **'ancestor'** rule to its original order and then alter the general case by swapping the two conditions to give:

```
ancestor(A, B):-ancestor(X, B),parent(A, X).
```

This now places the recursive call in front of the **'parent'** check.

Now try the goals given in Task 1.

Also try the goal: `ancestor(X, Y)`.

What has been the effect of reversing the two conditions of the rule?

10. What if our **'ancestor'** rule came before all the facts about **'father'** and **'mother'**?

Move the two **'ancestor'** clauses in their original form to the head of the program.

Now try the goals given in Task 1.

Also try the goal: `ancestor(X, Y)`.

What has been the effect of placing the rule first?

These are ‘not’ Problems!

You will have met ‘**not**’ which negates the success of a sub-goal. Strictly speaking ‘not’ succeeds when the sub-goal cannot be proved true.

For example, if we have a set of ‘likes’ facts such as ‘likes(jim, freda).’ then entering a goal such as:

```
not(likes(freda, jim))
```

will give ‘Yes’ if the ‘likes(freda, jim)’ fact does not exist – which does not mean that Freda does not like Jim!

‘not’ is most useful when we want to define a rule where a given fact or rule must not occur as a condition.

Great care is required when we use ‘not’ with variables. For example, if we tried the goal:

```
not(likes(X, Y))
```

how could Prolog give us any answers when it only has the ‘likes’ facts? It cannot come up with all the combinations of names that are not included as ‘likes’ pairs.

Tasks

11. Load the prepared file ‘**credit.pl**’ and inspect the facts and the rule.

Enter the goal: likes_shopping(Who).

12. Now swap the first two conditions of the rule to give:

```
likes_shopping(X) :-  not(account_empty(P, C)),
                      has_credit_card(P, C),
                      write(P), write(' can shop with the '),
                      write(C), write(' credit card. '), nl.
```

Try the goal from Task 11 again. No you haven’t made a mistake!

The rule in Task 11 worked because when Prolog got to the ‘not’ section it had instantiated the variables P and C to values and so it could establish the sub-goal’s success or failure. Now in Task 12, after swapping the parts, Prolog will meet the ‘not’ section without any values for the variables, i.e. the variables are free.

Appendix 2.2

Lists

A list is a simple data structure that contains a sequence of items, e.g.

[ann, barbara, carol, tom, dick, harry]
 [rugby, football, hockey, gymnastics]
 [english, mathematics, french, computing, physics]

Lists are roughly equivalent to arrays in procedural languages. However, you do not dimension their length beforehand; you do not access elements of the list by the place value of the element, and unlike arrays, lists can vary in length.

Lists are in fact more closely allied to the ideas of sets in Mathematics.

Features of Lists

1. [] represents the empty or null list.
2. Non-empty lists can be split into two parts – the **head** and the **tail**.

The head of a list is an element; the tail of a list is always a list.

e.g. [ann, barbara, carol, tom, dick, harry] has as its head **ann** and its tail the list [**barbara, carol, tom, dick, harry**].

Of course, we now have a list [barbara, carol, tom, dick, harry] which has a head **barbara** and tail [**carol, tom, dick, harry**].

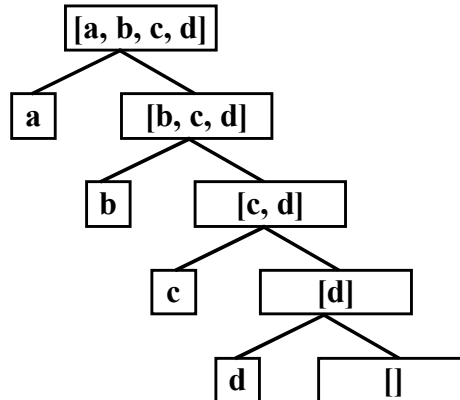
We could continue in this way splitting elements off the head of the list until we are left with the empty list. This is the method by which we can run through the elements of a list.

3. To split the head from the tail of a list we use | in a rule where part of it is of the form [**Head|Tail**] or in shorter form [**H|T**].

For example, with [ann, barbara, carol, tom, dick, harry] and [X|Y] then X is instantiated to **ann** and Y is instantiated to the list [**barbara, carol, tom, dick, harry**].

[To get | use <SHIFT> and the key in the bottom left of the keyboard which also has <\>.]

4. If we start with the list [a, b, c, d] and split off the head then eventually we will be left with the empty list.



Lists have a tree structure as shown on the right.

Because a list is a recursive data structure, we need recursive rules to process a list.

Here is short program to illustrate how to print the elements of a list:

```

write_a_list([]).           {terminating}

write_a_list([H|T]) :- write(H), nl, write_a_list(T). {general}
{nl = new line}
  
```

and we enter goals such as:

```

write_a_list([fred, jim, ann, tess])
  
```

Tasks

1. Type in the above program to write out a list and try it with different goals, i.e. make up your own lists and enter goals of the form `write_a_list([a, b, c, d])`.
2. Enter a goal to write out a list of integers, e.g. `[23, 45, 17]`.

Further Examples Using Lists

If we want to construct a database of boys and which hobbies or sports they like, we could use clauses of the style:

```

is_liked_by(golf, jim).
  
```

But if each likes several hobbies then we will have a lot of these clauses and the database will be clumsy.

Instead we will make a list of hobbies associated with each boy as follows:

```
is_liked_by([golf, chess, squash], jim).
is_liked_by([rugby ,football, watching_tv], bill).
is_liked_by([tennis, chess, golf], george).
is_liked_by([squash, tennis, rugby], david).
is_liked_by([golf, squash, football], alex).
is_liked_by([football, snooker, tennis], mike).
is_liked_by([snooker, watching_tv, golf], tom).
is_liked_by([chess, rugby, squash], john).
```

```
likes(Boy, Hobby) :- is_liked_by(X, Boy), member_of(Hobby, X).
```

```
member_of(H, [H|_]).
member_of(H, [_|T]) :- member_of(H, T).
```

Notes

1. The 'is_liked_by' clauses list the hobbies liked by each boy (the first three for each boy in a poll).
2. The 'likes' rule means:

Boy likes Hobby if there is a list for that Boy and Hobby is a member of that list.
3. 'member_of' is defined recursively. The terminating case is when the item being tested (H) is at the head of the list. The general case is when the item being tested (H) is in the tail of the list.

The rule peels off the head of the list until the item is found.
[NOTE: it is not necessary to mention the empty list as H cannot be a member of the empty list and so the goal would fail => answer No].

Tasks

3. Load the prepared file **'hobbies.pl'**. Try the following goals (maybe try **Options** -> **Trace** with one or more):
- i) Does George like chess?
 - ii) Which boys like squash?
 - iii) Does John like football?
 - iv) What does David like?
 - v) Do Mike and George both like tennis?
 - vi) Which hobby or hobbies do Jim, Alex and Tom all like?
 - vii) Who likes chess and watching TV?
 - viii) Does Tom like either snooker or tennis?

4. Load the prepared file **'chemical.pl'**. Additional clauses are required to define:
- a) 'member_of' – see above.
 - b) 'component_of' which is required to state that an element *x* is a component of compound *y* if *y* has a list of constituent elements and *x* is a member of this list.

Define and add these rules. Answer the following queries:

- i) Which compounds have carbon as a component?
 - ii) Which compounds have both carbon and oxygen as components?
 - iii) What are the components of methane?
5. Tickets have been sold for a lucky draw. Most people have bought more than one ticket and lists of the ticket numbers bought by each person are available.

Write a Prolog program, using lists, which could be used to find the name of the person who has bought a winning ticket.

6. A group of Sixth Year girls decided to celebrate leaving school by going on holiday together during the summer. The following was overheard in the Prefects' Room:

'Last year's school trip went to Spain,' said Allison, 'I really enjoyed it there but I like Majorca too. I've been there twice with my family and we also went to Portugal, it was fantastic.'

‘The best holiday I’ve ever had was in Greece,’ said Elaine, ‘but Spain was OK, I wouldn’t mind going there again.’

Claire, who had been sitting listening to all this, interrupted, ‘I haven’t been abroad before but I watch all the holiday programmes on TV and I would really like to go to Greece or perhaps Italy or even Ibiza.’

Jennifer, the Languages Department’s star pupil advised, ‘Perhaps we should go to France or Germany – at least some of us could communicate with the locals! I could try out my Spanish now that I’ve finished my module.’

Finally, Vicky, the Head Girl, said, ‘We’ll have to leave this discussion until later, that was the bell, wasn’t it? However, if we are stating preferences, I would go for Portugal or Italy. I’ve never been to either of them but I’ve read about them and they both sound terrific.’

Construct Prolog clauses (using lists of course!) from the knowledge contained in the above extract. You will need some other clauses as well as you proceed with the question. Enter your program and save it as ‘**holiday.pl**’.

Use it to answer the following queries:

- i) Which girls would like to go to Italy?
- ii) Which girls would like to go to either Spain or Portugal?
- iii) Where would Jennifer like to go?

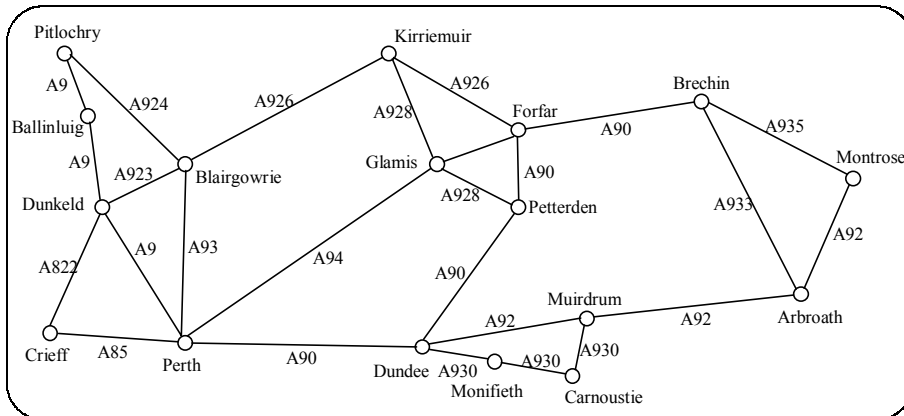
Complex Prolog Programs

The Artificial Intelligence Unit in Advanced Higher does not require you to write programs in Prolog or any other declarative language. However, you will realise that further, more complex language features allow the construction of much larger, less trivial and more useful programs than those you may have studied at Higher and earlier in this Appendix.

Here are some tasks using larger Prolog programs that nevertheless you should be able to follow and in which you should recognise certain features.

Tasks

7. Load the prepared file 'routes.pl'. This program contains the knowledge contained on the road map shown below and processes this knowledge to give a route between any pair of towns.



Type: ?- route.

Enter the following pairs of towns followed by a <full stop> on each run of the program and check the results with the map, e.g.

From |: perth. and press <Enter>.

- i) From perth to montrose (lower case of course).
 - ii) From kirriemuir to dundee.
 - iii) From pitlochry to arbroath.
 - iv) From perth to perth (this is not a mistake!).
 - v) Try other pairs.
8. Inspect the listing of the 'routes.pl' program. Here is a brief explanation of the new terms:

assert adds facts to the internal database of Prolog.

retractall removes all the facts in the internal database. In this program it is used to ensure that there are no facts at the beginning of the run.

The program contains a list of towns and 'on' facts which pair each town with a list of roads. The rules are:

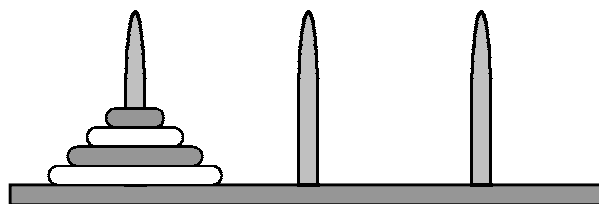
route	removes all previous 'gone_along' facts, requests the user input 'From' and 'To' and then starts the 'can_get_to' rule.
is_town	checks that the user input is two valid towns.
in	equivalent to 'member_of'
connect	is satisfied if two towns are connected by the same road. It then checks that this road has not been used before, adds the road to the internal database and prints the connection to the screen.
can_get_to	a recursive rule to search for the links between two non-adjacent towns.
printout	a recursive rule to print the items in the list given by the last line of the 'connect' rule.

The program works by using the 'connect' rule to add facts about the roads used and print out the route once the 'can_get_to' recursion has found a set of connections between the two towns.

9. Load the prepared file 'hanoi.pl'. This program solves the classic Towers of Hanoi puzzle. Look at the program listing.

The puzzle starts with a number of discs stacked on one pole (the left pole), with two more empty poles. The discs are of diminishing diameters.

The object is to move all the discs to another pole (the right pole) so that they are in their original order. You move the discs one at a time but at no time can a larger disc be placed on top of a smaller disc.



A simple recursive strategy for solving the puzzle is:

With one disc, move it directly from left to right – terminating case.

With N discs, follow the three general steps: Move N-1 discs to the middle pole;

Move the last disc (largest and Nth) directly over to the right pole;

Move the N-1 discs from the middle pole to the right pole.

The predicates used are:

`hanoi` has one argument (N the number of disks) and starts the move routine.

`move` the recursive rule to move as described above.

`inform` displays the sequence of moves.

Construct a pile of 3 different coins and follow the instructions given by the goal:

`hanoi(3)`

Try other goals, e.g. `hanoi(5)`.

10. Load the prepared file '**queens.pl**'. This solves the puzzle of placing N chess queens on a N × N chessboard so that no two queens attack each other. It is obvious that no two queens can be on the same row, column or diagonal.

Run the program with the goal:

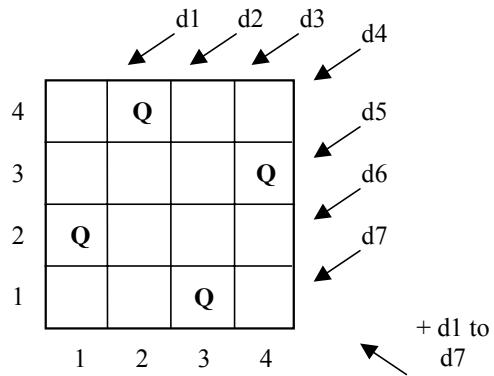
`nqueens(4)`

The result is the output of the list of lists 'board', e.g.

<code>board([q(1,2), q(2,4), q(3,1), q(4,3)],</code>	<code>[],</code>	<code>[],</code>	<code>[7,4,1],</code>	<code>[7,4,1]</code>
	<code>free</code>	<code>free</code>	<code>diags</code>	<code>diags</code>
<code>coords of queens</code>	<code>rows</code>	<code>cols</code>	<code>NE-SW</code>	<code>SE-NW</code>

The entries in this list are the coordinate pairs for the 4 queens and the free lists for rows, columns, diagonals (SW to NE) and diagonals (SE to NW).

Here is a diagram to help:



Try other goals – e.g. nqueens(8), nqueens(5) and nqueens(3).

Explain the result for nqueens(3).

Inspect the listing – oh well!

Briefly, this is another recursive situation where we start with an empty board and place a queen. We then add a queen by finding an empty row on which to place it and removing this row from the list of empty rows available for later queens. Similarly we find an empty column and an empty diagonal and remove the values used from their respective lists.

Examples 9 and 10 are concerned with puzzle situations where the general solution pattern can be extended through recursion to any number of discs or queens. All our programs are just the bare bones of the solution method coded in Prolog.

The next two examples demonstrate the use of Prolog together with a user interface. To construct the user interface, software designers have added a number of extra graphics, windows handling and event handling (clicking the mouse is an event) commands to Prolog.

This is a common extension of programming languages when they are used to write programs for modern systems, e.g. Turbo Pascal became Turbo Pascal for Windows.

Tasks

11. Load the prepared file '**reversi.pl**'. In this program the computer plays a reasonably intelligent strategy for the game of Reversi (also called Othello).

You, the player, play by clicking on a square where you want to place a blue counter. You must play to capture one or more white pieces by placing your counter to form a line of white counters with a blue counter at each end. The white counters in between then become blue, e.g.

B W W W and you add a counter at the right-hand end to give

B W W W B the white counters now become blue to give

B B B B B

Of course, this can be done horizontally, vertically and diagonally and one counter can capture in several directions simultaneously. To win, you either eliminate your opponent by capturing all his counters or by having the greater number of counters after the grid has been filled.

To run the program type the goal:

?- reversi.

12. Load the prepared file '**salesman.pl**'. This program finds the minimum mileage necessary for a travelling salesman to travel between towns in Britain. To run the program type:

?- salesman.

Select up to 7 towns by using <SHIFT> and click with the mouse. Once you have selected the towns, you have a choice of:

exhaustive search	all combinations of routes are checked and hence the shortest route will definitely be found but, of course, the search may take some time.
-------------------	---

heuristic search a simple rule of thumb is applied to find a short route. The search works by always going to the nearest town still not visited. This may not result in the shortest route as you will find if you compare the total mileages given by each search.

Unravelling a route is removing crossovers. Try the program with different numbers of towns and different spreads. With too many towns the exhaustive search may take a great deal of time – click <Stop> if necessary.

List Processing Rules

Tasks

13. Here is the shell of a list processing program:

```
member_of(H, [H|_]).
member_of(H, [_|T]) :- member_of(H, T).
```

With this program, we can enter goals such as:

```
member_of(6, [12,3,5,16,6,29])
```

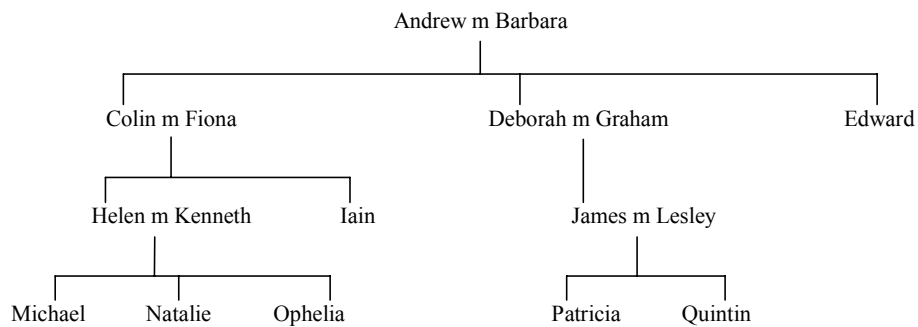
Define the following rules (you'll do well to do them all!) and use test goals to check your rules:

- a) first(X, Y) X is the first member of the list Y.
- b) butfirst(X, Y) X is the list obtained by removing the head of the list Y.
- c) count(X, Y) Y is the length of the list X.
- d) sum(X, Y) Y is the result of summing the list X.
- e) reverse(X, Y) Y is the list obtained by reversing the elements of the list X.
- f) append(X, Y, Z) Z is the list obtained by adding the list X to the end of the list Y, i.e. append X to Y to give Z.

Appendix 2.3

Solutions to Appendix 2.1 – Recursion

1. The family tree is:



parent(A, B) :- father(A, B).

parent(A, B) :- mother(A, B).

- i) X=andrew, X=barbara.
 - ii) X=graham, X=deborah, X=andrew, X=barbara.
 - iii) No solution.
 - iv) X=kenneth, X=helen, X=andrew, X=colin, X=barbara,
X=fiona.
2. offspring(A, B) :- parent(B, A).
3. descendant(A, B) :- offspring(A, B).
descendant(A, B) :- offspring(A, X), descendant(X, B).
4. Extra generations are added and the test goals give the expected results.
5. supplies(X, Y) :- feeds(X, Y), consumer(Y).
supplies(X, Y) :- feeds(X, A), supplies(A, Y), consumer(Y).
6. supplies(X, Y) :- feeds(X, Y), transformer(X).
supplies(X, Y) :- feeds(X, A), supplies(A, Y), transformer(X).
- i) boss(A, B) :- works_for(B, A).
 - ii) under(A, B) :- works_for(A, B).
under(A, B) :- works_for(A, X), under(X, B).
 - iii) over(A, B) :- boss(A, B).
over(A, B) :- boss(A, X), over(X, B).

8. No effect.
9. 'Local Stack Full' message as the recursive calls happen endlessly.
10. No effect.
11. Who=joe, Who=sam.
12. 'Control Error' message.

Appendix 2.4

Solutions to Appendix 2.2 – Lists

1. Enter goals of the form – `write_a_list([list of items])`.
2. `write_a_list([23, 45, 17])`.
3.

i)	<code>likes(george, chess)</code> .	Yes.
ii)	<code>likes(B, tennis)</code> .	B=george, B=david, B=tom.
iii)	<code>likes(john, football)</code> .	No.
iv)	<code>likes(david, H)</code> .	H=squash, H=tennis, H=rugby.
v)	<code>likes(mike, tennis), likes(george, tennis)</code> .	Yes.
vi)	<code>likes(jim, X), likes(alex, X), likes(tom, X)</code> .	No solution.
vii)	<code>likes(B, chess), likes(B, watching-tv)</code> .	No solution.
viii)	<code>likes(tom, snooker); likes(tom, tennis)</code> .	Yes.
4.

a)	<code>member_of(H, [H _])</code> . <code>member_of(H, [_ T]) :- member_of(H, T)</code> .	
b)	<code>component_of(Element, Compound) :-</code> <code>has_elements(Compound, X),</code> <code>member_of(Element, X)</code> .	
i)	<code>component_of(carbon, Y)</code> . Y=methane, Y=carbon_dioxide, Y=alcohol	
ii)	<code>component_of(carbon, X); component_of(oxygen, X)</code> . X=methane, X=carbon_dioxide, X=alcohol, X=water, X=carbon_dioxide, X=alcohol	
iii)	<code>component_of(X, methane)</code> . X=carbon, X=hydrogen	
5. Same as other programs in this section.
6.


```
preferred_by([spain, majorca, portugal], allison).
preferred_by([greece, spain], elaine).
preferred_by([greece, italy, ibiza], claire).
preferred_by([france, germany, spain], jennifer).
preferred_by([portugal, italy], vicky).
```

`like_to_go_to(Who, Where) :- preferred_by(X, Who), member_of(Where, X).`

```
member_of(H, [H | _]).
member_of(H, [_ | T]) :- member_of(H, T).
```

- i) `like_to_go_to(Who, italy)` – `Who=claire`, `Who=vicky`.
 - ii) `like_to_go_to(Who,spain)`; `like_to_go_to(Who, portugal)`
`Who=allison`, `Who=elaine`, `Who=jennifer`, `Who=allison`,
`Who=vicky`.
 - iii) `like_to_go_to(jennifer, Where)` – `Where=france`,
`Where=germany`, `Where=spain`.
10. `nqueens(3)` gives an infinite loop message because 3 queens cannot be arranged on a 3×3 board without attacks.

13. a) `first(H, [H | _])`.
- b) `butfirst(T, [_ | T])`.
- c) `count([], 0)`.
`count([_ | T], X) :- count(T, N), X = N+1`.

Recursion as the length of `[_ | T]` is one more than the length of `T` and eventually `T` can be reduced to `[]` which has length 0.

- d) `sum([], 0)`.
`sum([H | T], S) :- sum(T, N), S = H+N`.

Recursion again as the sum of `[H | T]` is H more than the sum achieved for `T` and `T` can be reduced to `[]` which has sum 0.

- e) `reverse([X], [X])`.
`reverse([H | T], L) :- reverse(T, L1), append([H], L1, L)`.

A trick question as you need ‘append’ defined!

- f) `append(X, [], X)`.
`append(L, [H | T], [H | L1]) :- append(L, T, L1)`.

Introduction

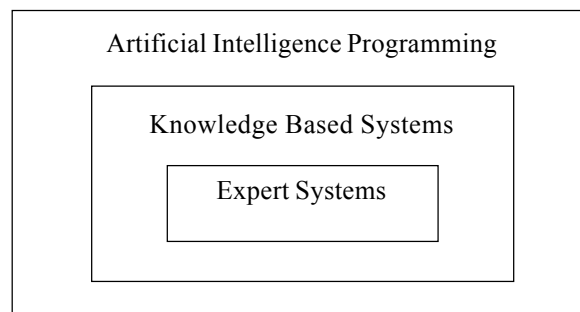
In this chapter we are going to study expert systems. You may already have met some of the ideas if you studied expert systems in the Higher Computing Unit – Artificial Intelligence.

In particular, you are going to critically appraise the features of expert system shells by describing these features and by comparing your implementations of expert systems using different shells.

Expert Systems

These result from attempts to automate the knowledge and reasoning of human experts.

This application area of artificial intelligence has been a significant and growing aspect of research since about 1965.



Expert systems are used by:

- doctors to help diagnose specialist diseases such as cancer, kidney disease, viral or skin infections.
- mining companies to advise on the most likely places to explore for minerals.
- the DSS to assist staff who are dealing with a client's claim.
- supermarket chains to analyse customers' shopping patterns and to identify trends.
- financial advisers to help advise clients on the best type of investment.

- electricity distribution companies to advise what to do in the event of a power shortage emergency.
- legal departments of international companies to advise on the effects of laws in all the countries in which they operate.
- electronics companies to identify faults in circuit boards and components.

Definition

An expert system is a computer system that has stored in it some of the expert knowledge of a group of people, such as doctors, geologists, analytical chemists or stock market traders.

Given a problem in its specialist field through user input of data, an expert system is expected to provide advice along with an explanation to the user of its line of reasoning.

Some expert systems are intended for use as interactive advisers by experts in the same field; others are used by less highly qualified or non-expert personnel. Expert systems must be able to handle uncertain and incomplete information.

Expert systems operate in interactive mode, allowing the user to supply information in a flexible way and providing conclusions as soon as these can be inferred from the knowledge base.

In some systems the knowledge base is 'fixed' when it is entered into the system and remains the same throughout. In other systems the knowledge base can alter as the system can 'learn' by altering existing knowledge or acquiring new knowledge.

Creating an Expert System

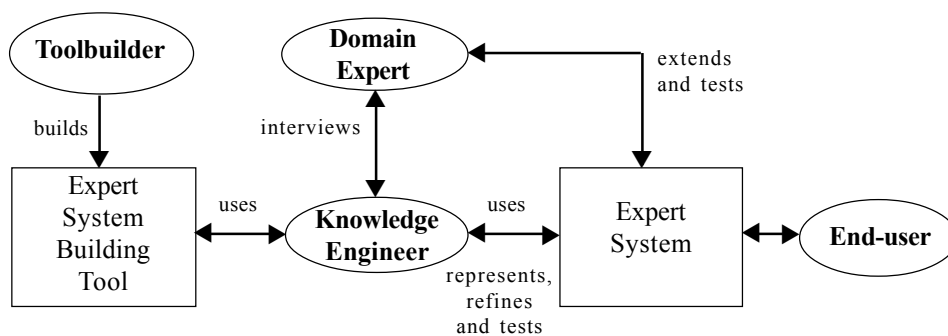
There are three stages in creating an expert system:

- 1) knowledge acquisition
- 2) knowledge representation
- 3) system validation.

1) *Knowledge Acquisition*

This involves:

- i) the **domain expert** – a human with the knowledge.
- ii) the **knowledge engineer** – a programmer/analyst who decides how the knowledge will be represented and stored, tests the system and decides how it will be accessed by users.
- iii) the **end-user** – who may be expert to a greater or lesser degree.



The **domain expert** will have:

- personal experience of problems to be solved
- personal expertise in how to solve the problems
- personal knowledge of the reasons for selecting certain methods.

The **knowledge engineer** will have no specialist knowledge of the domain and will have to learn from the domain experts, prior to the start of the project and during the acquisition of knowledge.

Some of the difficulties with this are:

- the expert may be unable to express his/her knowledge in a systematic and structured manner.
- the expert may not be aware of the significance of his/her knowledge.
- the expertise may be vague and imprecise ('in most cases', 'usually').
- some experts may not wish to co-operate ('lose my job to a machine!').
- the knowledge domain may have to be restricted (e.g. medicine is too large a knowledge domain).
- the system needs to be updated (e.g. new treatments and drugs).
- the system needs to be able to justify its inferences.
- high development costs (time and money).
- the expert is too busy.

In addition, the expert system has no common sense and so it must be provided with all the knowledge. For example, you can give a human instructions to cross a road along the lines of: 'Stop at the kerb.', 'Look for vehicles.', 'Wait until there are none.', 'Cross the road'. But an expert system would need a definition of kerb, vehicle and road and it would still wait forever because there was a car parked ten metres away!

2) ***Knowledge Representation***

We have looked at this area in considerable detail in Chapter 2. There are a large number of software tools that can be used to build the knowledge base and the instructions to infer conclusions from this knowledge base.

Some possibilities are:

- **expert system shell** – you only have to add the knowledge base.
- **declarative language** – Lisp or Prolog, designed for this type of task.
- **procedural language** – you have to program all parts of the system.

3) ***System Validation***

During this phase the system is tested and problems are identified by comparing the expert system's advice with that provided by the domain experts. If there are differences then these could be due to:

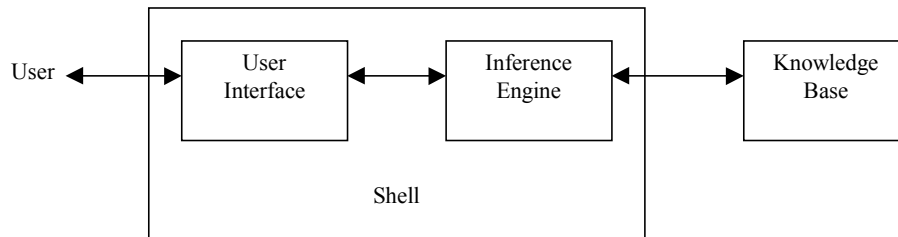
- the knowledge not being represented correctly or being incomplete.
- the knowledge engineer misunderstanding information given by the domain experts.
- the knowledge not being processed correctly to produce appropriate advice.

Testing will continue until the expert system consistently agrees with the domain experts.

Self Assessment Questions

1. Give reasons why an expert in a certain field might use and appreciate help from a computerised expert system.
2. Describe each of the stages involved in creating an expert system.

Components of an Expert System



There are three main components: a knowledge base, an inference engine and the user interface.

In addition there will be **working memory** where the user's responses to questions and the deductions for the current query are stored and then discarded when a new query is started.

It is convenient to view the inference engine and the user interface as one module, called an **expert system shell**. This can be supplied as a common part of any expert system regardless of the knowledge domain.

1) A **knowledge base** consists of:

- facts that define classifications of objects and relationships between objects
- rules for manipulating the facts.

As we have seen in Chapter 2 there are a number of methods employed to represent knowledge:

- search trees
- logic
- semantic nets
- frames.

But knowledge cannot always be expressed categorically. That is, answers to questions are not always true or false but rather of the form highly likely, likely, unlikely and highly unlikely.

So that a computer can process **uncertainty** these human expressions must be changed to numerical measures of the certainty or uncertainty of the answer. In statistical probability, the measures of certainty vary from 0 (impossible) to 1 (certain), e.g. the probability or **certainty factor** of getting a head on the toss of a coin is expressed as 0.5.

To reflect this uncertainty in the real world, expert systems have been developed to allow the knowledge engineer to enter certainty factors when creating the expert system; to allow the user to enter certainty factors when responding to the system's questions; and also to produce certainty factors for each of its deductions or elements of advice.

Different expert systems use different ranges as a measure of uncertainty. For example, MYCIN – an expert system designed to diagnose bacterial infections – uses the range 0 to 1, while Prospector – an expert system designed to aid geologists in their search for ore deposits – uses the range -5 to +5.

The main difficulty with certainty factors lies in the fact that neither the human experts, when constructing the knowledge base, nor the users when consulting the expert system, think accurately in terms of probabilities. You just have to think of football commentators trying to predict match results. The statistics based upon previous performance could be used to forecast a likely outcome but there is always the possibility of a freak result.

2) **An inference engine** provides a control structure for applying rules and consists of:

- an interpreter that decides on the meaning of the user's input
- a scheduler that decides on which rule to implement next.

This program hunts through the rules within the knowledge base looking for a rule that most closely matches the query it has been given. The inference program may ask for further information from the user and will finally provide a conclusion.

If certainty factors are used then the inference engine will combine certainty factors that are allocated to the conclusions of rules when the knowledge base is constructed, with the certainty factors for the data entered by the user. The inference engine will then produce a certainty factor for each conclusion or item of advice. The details of how these are combined do not concern us here but are based upon the laws of statistical probability.

There are two techniques used when searching through the rules in the knowledge base:

i) *forward chaining*

This involves reasoning forward from the facts given, using the rules to generate more facts until the desired end is reached.

ii) *backward chaining*

This involves reasoning backward from the desired end point to determine what starting facts are required and then to check that these facts exist and are true.

A simple example should help to make the difference between these two methods clear.

When a patient sees a doctor, the patient will describe symptoms such as a sore throat, cough, shortness of breath, the time that the illness has been present, loss of appetite, sleeping badly and so on. Using these symptoms and observation the doctor will reason forward to the possible cause such as flu.

However, if the illness cannot be easily diagnosed or there is uncertainty as to the cause of the illness then the doctor will refer the patient to a specialist. The specialist will conjecture a possible cause, such as a viral infection of the lungs, and then use tests, such as biopsy and X-rays to check whether the necessary conditions for this virus are present. The specialist is reasoning backward to find facts which support the illness hypothesised.

There are advantages and disadvantages to both forward and backward chaining.

Advantages and Disadvantages of Forward Chaining

Advantages

Works well when the problem naturally starts by gathering data and then finding what information can be inferred from it.

Can provide a considerable amount of information from a small amount of initial data.

Suitable for problems where an expert system is required for planning, monitoring, control or interpretation. XCON, used to configure large computer systems, is an example of a forward chaining expert system.

Disadvantages

The inference engine generates new information without knowing what is important and relevant to the problem solution.

Forward chaining systems may ask for a lot of user input much of which will be irrelevant to the final conclusion.

Advantages and Disadvantages of Backward Chaining

Advantages

Works well when the problem naturally starts by forming a hypothesis and then checking whether the hypothesis can be proved.

Remains focused on a given goal and so only asks questions that are relevant to proving the hypothesis and only uses parts of the knowledge base that are relevant to the hypothesis.

Suitable for problems where an expert system is required for diagnostics, prescription or debugging. MYCIN is an example of a backward chaining expert system.

Disadvantages

Backward chaining systems may follow a particular path of reasoning and search even though it eventually proves fruitless.

Choosing Between Forward and Backward Chaining

It is important to analyse the problem and knowledge domain to determine which inference method is appropriate. This will also eliminate the use of certain expert system shells as their inference methods are not appropriate.

However, as our example of a patient seeing a doctor and a specialist indicates, there are many real life situations where forward and backward chaining are used in combination. Some expert systems also use a combination of forward and backward chaining.

3) **A user interface** provides:

- a means by which knowledge engineers can input and edit the knowledge base
- a means for users to query the system and receive information and explanation of its reasoning.

Sometimes called the **explanatory interface**, the **user interface** allows communication between the user and the system to make consultation and the resultant advice and justification as easy as possible.

The common features of Human Computer Interface are used, namely:

menus graphics help pages question and answer
WIMP (Windows, Icons, Mouse, Pull-Down Menus).

Justification of Reasoning

A distinguishing feature of an expert system is its ability to explain its decisions and advice given. Users can ask two types of question asking for justification:

Why questions – Why are you interested in this information?

The user will ask 'Why?' when the system's query seems irrelevant or it will require additional effort or cost on the part of the user. From the answer the user can judge whether the extra effort or cost to gain the information required by the system is worth it.

Suppose that the system has asked:

Is Z true?

and the user responds:

Why?

The inference engine will provide an explanation along the lines:

I can use Z to investigate W by rule R(z), and
 I can use W to investigate T by rule R(w), and
 I can use T to investigate K by rule R(t), and

 I can use C to investigate A by rule R(c), and
 A was your original question.

How questions – How did you reach this conclusion?

The user may wish to know how a certain conclusion was reached. The inference engine responds by listing the rules and sub-goals that were satisfied for it to reach its conclusion.

The inference engine might respond:

andrew is a carnivore
 was derived by rule 3 from
 andrew is a mammal
 was derived by rule 1 from
 andrew has hair
 was told

and
 andrew eats meat
 was told.

Self Assessment Questions

3. Name four methods that are used to represent a domain of knowledge in a knowledge base. Which method would you choose for each of the systems below? Give reasons for your choice.
- i) An expert system to identify soya bean diseases that uses knowledge about disease symptoms and plant environment.
 - ii) An expert system to help battlefield commanders to deploy military units which consist of men, equipment, stores, support personnel, etc.
 - iii) An expert system to isolate and diagnose faults in electronic equipment.

Each of these expert systems exists – called PLANTds, AMUID and FOREST.

4. An insurance company uses an expert system to assist in processing insurance claims. A claim is received from the driver of a car involved in a collision. Describe two examples of uncertainty or incompleteness in this situation.
5. In MYCIN, the certainty factor was on a scale of 0 (impossible) to 1 (always true). Other expert systems use ranges such as -5 to +5, or 0 to 100.

Find out what range of values is used to express uncertainty in the expert system shell that you will be using.

6. Describe each of the inference methods: forward chaining and backward chaining.
7. Give reasons why you think expert systems should be programmed to explain their reasoning.
8. Describe the circumstances in which a user would want to make use of:
- i) the 'Why' justification feature
 - ii) the 'How' justification feature.

Fuzzy Logic

As we have seen, many situations are not clear cut, true or false, and certainty factors allow the expert system builder and the user to enter facts and rules that are neither 100% true or false. There are other situations where we use language descriptions that are not strictly true or false and yet we still cannot apply certainty factors. These situations occur where a particular object may be a member of a property set with varying degrees of possibility.

As usual a few examples and re-reading the abstract definition above will help to make this idea clearer.

Consider the statement, 'James is tall'. We could decide if James belongs to the set 'tall' by defining 'tall' as being over 6 feet. In that case, if James is 6 foot 2 inches then 'James is tall' is true. But if James is a professional basketball player then James is short. The definition of 'tall' is fuzzy. [It does not make sense here to use a certainty factor, e.g. 'James is tall (0.6)', as his height does not vary from situation to situation but rather the interpretation of the word 'tall' varies.]

Consider the statement, 'Joanne is a child'. Again we could use a legal definition and state that at a certain age, e.g. 16 years old, children become adult and so Joanne moves from the set of children to the set of adults. But we all know 12-year-olds who are going on 25 and, also, 18-years-olds who behave as if they are going on 10!

Properties of objects such as tall and short, hot and cold and young and old have fuzzy definitions. Expert systems have been devised that use fuzzy set membership and combine these properties using fuzzy logic to generate output.

A detailed treatment of these ideas is beyond the scope of this course, but these systems are frequently used in control applications. Fuzzy logic has been applied to controlling household devices such as refrigerators and washing machines and to controlling trains and mobile robots.

Self Assessment Question

9. Decide what certainty factor, on a scale of 0 to 1, you would apply to the following rules. Describe any difficulties you encounter particularly where you have to apply fuzzy logic.
- i) If a stone is colourless and clear, then there is a 60% chance that it will be identified as a diamond.
 - ii) If a person has a high temperature, then he/she should stay in bed.
 - iii) If a student attends school regularly, then he/she will pass an exam.

Expert System Shells

An expert system shell consists of:

- an inference engine
- a user interface.

The user interface allows both:

- the entry of the knowledge base of facts and rules
- the system to be interrogated through a query language.

As we have seen, expert systems can be implemented using both declarative and procedural languages. By comparison, expert system shells have advantages and disadvantages:

Advantages

- fast development and prototyping (testing of first system).
- user only has to provide knowledge about the problem domain.
- knowledge is easily entered (WP style) – suitable for non-expert computer users.
- interfaces to other software (e.g spreadsheets – Excel, databases – Access) allowing use of existing data.
- cheap medium for distributing knowledge (package is self-contained requiring no further software).

Disadvantages

- inflexible interface (the nature of the knowledge may not fit the data format allowed by the shell – hence it is normal to search for a suitable shell for each domain).
- the implementation at run-time may be disappointing as the inference engine may not be suitable for the structure of the knowledge.
- often not suitable for large applications.

Comparing Expert System Shells

Your teacher/lecturer should have two expert system shells available for you to use. One of these should deal with uncertainty while the other does not. You must make yourself familiar with the use of these shells through inspecting sample files that are supplied with the shells and also through reading the available documentation and help screens.

Self Assessment Questions

10. For each of the expert system shells, investigate and write notes on:
 - i) the method of building the knowledge base
 - ii) the search techniques
 - iii) the quality of the user interface
 - iv) the quality of the justification facilities.
11. For the expert system shell that supports uncertainty, write notes on the range of certainty factors, how these are combined and what effect this has on the use of the expert system.

Organising Knowledge for Use in Expert System Shells

A common method used to organise knowledge is to draw up a table where each row of the table refers to one object and each column is a property of the objects being considered.

For example, if we were developing an expert system about holiday destinations then each row would be a possible destination (Monte Carlo, Ibiza, Scottish Highlands, etc.) and each column would be a feature of the destinations (cost, sunshine, sea, sports, etc.).

We can then develop rules for determining a suitable object or group of objects by combining the properties for each object. For example, Nice would be suggested as a suitable destination if you could afford an expensive holiday, and liked the sun, the sea and water sports.

Here is an example of converting knowledge statements into a table prior to implementing the knowledge using an expert system shell:

Swimming and badminton are indoor sports while cycling, tennis and hockey are outdoor sports. Swimming and cycling are individual sports, badminton and tennis both require an opponent and hockey requires two teams of eleven players.

All of these sports require some specialist equipment before you can start participating. Swimming costumes are quite cheap. Badminton and tennis rackets and hockey sticks are a bit more expensive. Cycling is the most expensive to start.

This knowledge could be expressed in the following table:

Sport	Indoor/Outdoor	Numbers involved	Initial cost
Swimming	Indoor	One	Low
Badminton	Indoor	Two	Middle
Cycling	Outdoor	One	High
Tennis	Outdoor	Two	Middle
Hockey	Outdoor	Twenty-two	Middle

The words used for certain properties are descriptive and not precise so that another knowledge engineer might use other words. In the above example, low, middle and high have been used to describe the initial

cost but we might have used cheap, not too expensive and expensive as our descriptive categories.

Self Assessment Question

12. An expert system could be used to train people in first aid. The following paragraphs outline a limited domain of knowledge about the treatment of burns.

Burns are divided into three categories.

Superficial burns involve only the outer layer of the skin, and are characterised by redness, swelling and tenderness.

Partial-thickness burns require medical treatment. The skin will look raw and blisters will form. Full-thickness burns are where all the layers of the skin are damaged and maybe the nerves, muscle and fat below. The skin will appear pale, waxy and sometimes charred. These always need immediate medical treatment.

To treat a superficial burn, flood the injured area with cold water for about ten minutes and cover the area with a sterile, non-fluffy dressing. A polythene bag or cling film makes a good temporary covering.

To treat partial-thickness or full-thickness burns, lay the casualty down and flood the injured area with cold water for over ten minutes. While doing this arrange for removal of the casualty to hospital. Cover the area with a sterile, non-fluffy dressing or a plastic bag or cling film. Treat the casualty for shock while waiting for an ambulance.

- i) State the problem in your own words.
- ii) State the boundaries of the problem.
- iii) Draw up a table to collate the knowledge described in the paragraphs above.
- iv) Implement the knowledge using an expert system shell without uncertainty.
- v) Implement the knowledge using an expert system shell that supports uncertainty. You will have to assign certainty factors to the classification of the burn and to the need for hospitalisation and treatment of shock.

Self Assessment Question 12 (continued)

- vi) Write a brief report comparing the two implementations using the following criteria:
- the ease of building the knowledge base;
 - the quality of the user interface;
 - the quality of the justification facilities;
 - the effect of the search techniques used by the shells;
 - the effect of uncertainty.

Chapter 3: Key Points

- An **expert system** is a computer system that has stored in it some of the expert knowledge of a group of people. Expert systems are used in a large number of different situations.
- There are three stages in creating an expert system: **knowledge acquisition**, **knowledge representation** and **system validation**.
- Knowledge acquisition involves the **domain expert**, the **knowledge engineer** and the **end-user**.
- The expert system can be created using an **expert system shell**, a **declarative language** or a **procedural language**.
- The main components of an expert system are: a **knowledge base**, an **inference engine** and a **user interface**. An **expert system shell** consists of a pre-programmed inference engine and user interface to which the user adds a knowledge base.
- The knowledge base can be represented using **search trees**, **logic**, **semantic nets** or **frames** and may involve **uncertainty** expressed by probability-based certainty factors.
- The inference engine can be based upon **forward chaining** or **backward chaining**. The more appropriate inference method to use depends upon the type of problem involved.
- The user interface not only allows the user and the system to interact but will also include **justification of reasoning** facilities where users can ask '**why**' and '**how**' questions.
- **Fuzzy logic** applies to situations where language definitions are not precise.

Chapter 3: Self Assessment Questions – Solutions

1. The expert may not wish to remember lists of facts that would be available from the expert system. The expert can get a 'second opinion' from the system. The system may save hours of analytical work to arrive at the deduction. The system may prevent mistakes when the expert is tired.
2. Descriptions of: knowledge acquisition, knowledge representation and system validation – see p.112 and following pages.
3. Search trees, logic, semantic nets and frames.
 - i) Logic; diseases are of the cause and effect type, i.e. there are clear connections (and probabilities) between certain symptoms and the disease concerned which can be expressed as rules, e.g. if leaves turn brown then disease is root fungus ($P=0.8$).
 - ii) Semantic net; there is a complex relationship between the components of the unit and properties inherited, e.g. men cannot advance to fight unless there are supplies of ammunition, medical personnel, etc. and each of these require further conditions, e.g. supplies of ammunition require lorries which in turn require petrol, drivers, repair teams, etc.
 - iii) Search tree; the fault can be diagnosed by answering a series of questions of the type, 'Does the power light come on?' that take the search down branches of the tree to further questions based on that response. [Logic could also be used; as in i) there are clear causal rules and probabilities linking observations and faults, e.g. if no power on main board then power supply unit faulty ($P=0.95$).]
4. Examples of uncertainty or incompleteness: details of the movement of the two vehicles – speeds, braking, times of movement, e.g. pulling out at a junction; details of injuries and damage – how serious, have extras been added; road conditions – wet, icy, low sun.
5. This will depend upon the expert system shell in use.

6. Forward chaining is where you start with all the known facts and use the rules to produce new facts until the solution that you seek is found.

Backward chaining is where you start with the hypothesis and generate all the facts that are required to satisfy it. You then check that these required facts are a subset of the known facts.

7. Only through explanation can faulty systems be detected and the knowledge base or inference engine be rectified. Where deductions are probability-based then an explanation can provide the user with an idea of what data influenced the system most strongly. Where the human expert and the system disagree on their conclusions then the explanation facility allows the point of divergence and the reason for divergence to be identified. Human experts have to be accountable and explain their reasoning, e.g. doctor to patient or expert to a court.
8.
 - i) The user wants to know why the system is asking for this information. The answer may be expensive and time consuming to find (e.g. medical test, deep drilling to find rock composition) and may not add much to the expert system's reasoning (e.g. it may distinguish between two responses but the user can do this, once the two responses are output).
 - ii) The user is uncertain of the advice and wants to know what combination of facts and rules gave rise to the outcome. It will also be used extensively during testing of the system to ensure that the correct rules are firing to produce the correct responses.
9.
 - i) The value of 60% appears to be based on some measure of past experience, perhaps based on actual data recorded by a diamond prospector. A 60% chance would be expressed as a certainty factor of 0.6.
 - ii) Staying in bed is the best advice, but there is a degree of fuzziness about this – how high is a 'high' temperature? Would lying on the settee be all right, or is the temperature so high that the patient should be admitted to hospital? A fairly high certainty factor of (say) 0.8 to 1 might be used.

- iii) There is a degree of fuzziness in the rule. How often is 'regular'? Once a week? The condition about attending school regularly is not sufficient to guarantee success in exams. Further conditions are needed – a student needs to pay attention in class, spend time studying, do homework conscientiously. A certainty factor of approximately 0.5 might be appropriate, indicating that neither success nor failure is more likely as a consequence of the condition being satisfied.

10 and 11. You should get your answers checked by your teacher or lecturer to ensure that you have provided correct and sufficient details.

- 12. i) The problem is to develop an expert system to advise untrained people on the treatment of burns.
- ii) The boundaries of the problem are that the system will only deal with the three categories of burns. It will not take other factors such as the extent of the burns, the nature of the burn (scald, chemical, electrical) or the area of the body affected (mouth, nose, eyes) into account.

iii)

Type	Signs	Treatment	First Aid
Superficial	redness, swelling, tenderness		cold water for ten minutes, sterile, non-fluffy dressing or plastic bag or cling film
Partial-thickness	raw skin, blisters	hospital	lay down, call ambulance, cold water for over ten minutes, sterile, non-fluffy dressing or plastic bag or cling film, treat for shock
Full-thickness	all skin layers damaged, pale, waxy skin, maybe charred	immediate hospital	lay down, call ambulance, cold water for over ten minutes, sterile, non-fluffy dressing or plastic bag or cling film, treat for shock

- iv) You should fully test your expert system.
- v) Again you should fully test your expert system. The user should be asked to enter a certainty factor in answer to questions about the severity of the burn and should be given certainty factors for the need to call an ambulance and to treat for shock.
- vi) You should get your teacher or lecturer to check your answer here to ensure that you have provided sufficient correct comparisons.