PLANNING and SEARCH

Artificial Intelligence 2

1986/87

Mike Uschold

"The study of how to make computers do things which, at the moment, people are better."

by Elaine Rich [[Rich 83]]

"The effective computational deployment of knowledge."

by Henry Thompson

"The study of mental facilities through the use of computational models. The ultimate goal of AI research is to build a person."

by Charniak and McDermott [[Charniak 85]]

"The attempt to liberate tools/machines from absolute dependence on human control."

anonymous [1]

Unfortunately, these definitions are not overly enlightening. In particular, they leave unanswered the following philosophical questions:

- What is intelligence?
- How can we know whether a machine is intelligent?

Some of our common sense views of what makes people intelligent include:

- Ability to remember.
- Can think through a problem to find a solution, given some rules and guidelines to follow(*e.g.* Changing the brakes on your car using a manual, or solving maths problems).
- Can reason from first principles to solve problems for which no 'cookbook' answers exist (*e.g.* Combine common sense with the information in an auto manual to solve a problem never encountered before and not discussed directly in the manual).
- Can *learn* new things easily.
- Creativity, *i.e.* can think of novel solutions to problems; be inventive

Computers are extremely good at 'remembering' things, possessing a great capacity for storing and retrieving information. However, we do not think of a computer as having intelligence because of this ability. Current AI computer programs are able to solve certain types of problems, especially when there are well defined rules for solving them (*e.g.* diagnosing the problem with a malefunctioning bicycle). Some programs have been written which can do very rudimentary forms of learning. Reasoning from first principles is a very difficult task which is being actively researched. However, making computers which are in any real sense *creative* is considerably beyond the current state of the art. No one has seriously considered how to even begin to tackle this problem. The achievements to date for the field are much more meager. In fact, AI programs can be highly successful if they are able to reproduce the intelligence of a normal 5 year old. (*e.g.* in language understanding, recognising objects with sight).

The problem of deciding whether or not a machine can be considered intelligent was given some thought by Alan Turing who suggested the now famous *Turing test* to answer just this question. The test may be summarised as follows:

> A person is in a room with no communication with the outside world except via computer terminal. S/he conducts a conversation with another 'party' which claims to be a person, but which may in fact be a computer. If the person cannot tell whether or not s/he is conversing with a person or a computer, then the computer is said to have passed the turing test.

[1] I read this from the AI Digest, 1986. I forget the author's name.

Chapter 1

General Introduction to Artificial Intelligence

1.1 Preface

There are a variety of AI textbooks around which present basic material on problem solving, search and planning. However most are geared for third and fourth year undergraduates, or postgraduates. Examples include: [[Charniak 85], [Rich 83], [Winston 84], [Nillson 80] [Barr 81a], [Barr 81b], [Cohen 81], & [Pearl 84]]). These notes differ from these texts in two important ways:

1. I give more elaborate treatment to the basic concepts than is typically found in these more advanced texts.
2. I usually avoid abstract formal treatment of the methods and techniques presented. Instead, I attempt to convey sound intuitions using plenty of examples.

In the large, I present the basic issues problem solving somewhat differently than the other texts. In particular, I place more emphasis on separating the issues of representation and search. However, many of the important points I stress are borrowed. These notes are a synthesis of many points of view.

1.2 What is Artificial Intelligence?

Perhaps the first and most obvious question to address is: "What is Artificial Intelligence?". Unfortunately, there is no accepted definition among the workers in this field. In fact, people's ideas differ rather substantially. This can be seen from the following selection of definitions. The first is probably the most often quoted and universally accepted definition. There are two rather different viewpoints reflected in these definitions:

Cognitive Science - Psychology: The goal is to understand the process by which humans exhibit intelligent behaviour, and to replicate this on computers.

Engineering: The goal is to build machines which exhibit intelligent behaviour, full stop. It is of no consequence whether or not the techniques used bear any similarity to those employed by the human mind.

The first three reflect the engineering view and the fourth reflects the cognitive science view. The fifth reflects yet another view, that of machine liberation which, as far as I know, (and fortunately, perhaps) is not a widely held view.

"The science of making machines do things that would require intelligence if done by humans."

by Marvin Minsky

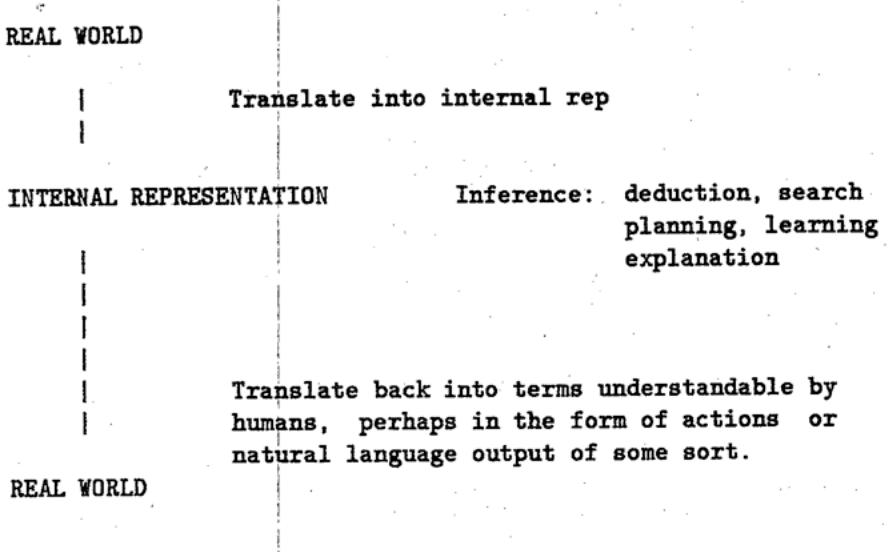


Figure 1.1: Structure of an AI Program

No program is likely to pass this test in the near future. Many decades of research are likely to be necessary. Interestingly, however there is a very famous counselling program called ELIZA [[?]] with which many people conducted intensely personal conversations. It was so human-like that they were able almost to *forget* that they were actually talking to a computer. Some even got psychologically hooked on it, using it everyday to help deal with their problems. Although this sounds very impressive, it turned out that this program only used very straightforward computer techniques. They were so simple in fact, that anyone who understood them would be most reluctant to admit that the program had intelligence in any useful sense of the word. This program will be examined in some detail in the Natural Language module.

1.3 What is an AI Program?

This is another very slippery question, which I shall treat somewhat briefly. The obvious and hence unsatisfying answer is: "any program which exhibits intelligent behaviour". This however, gives no insights into how people go about building intelligent programs, or what they consist of. The overall structure of an AI program is illustrated in figure 1.3. You must encode the relevant information and knowledge in a form suitable for computer processing. This is referred to as an *internal representation.* This done, the information must be processed, and then made available to the user of the program. Thus, it will have to be retranslated back to a form understandable by humans.

At the highest level, this structure resembles that of any computer program. It is in the nature of the internal representation and how it is processed which sets AI programs apart. Any intelligent behaviour whether exhibited by humans, monkeys, or machines requires knowledge as the key ingredient. Therefore, to reproduce such behaviour will indeed require, (as Henry Thompson so succinctly stated on page 3) "the effective computational deployment of knowledge". The builder of any AI program will have to consider two major issues: [2].

1. knowledge representation
2. knowledge application (*i.e. reasoning*)

The distinction between representing and applying knowledge is actually quite important. We all know people who evidently have great stores of knowledge, yet they seem to lack sufficient common sense, discipline, or indeed *intelligence* to put this knowledge to use. The ability of putting two and two together, has indeed escaped them. Truly intelligent people not only know a great deal, but they also know how to reason with this knowledge to derive new information (*e.g.* Sherlock Holmes). Equally important, is their ability to direct their reasoning fruitfully thus enabling the speedy analysis of a situation, or solution of a problem. Our friend Sherlock would never have gotten anywhere if he simply went on making logical deductions

[2]Such people are widely (and pretentiously?) known as *Knowledge Engineers* (especially in California).

willy nilly deriving all sorts of useless information. Simply knowing how to make inferences is not enough, you must know when to make them! This is discussed further in section 1.3.2.

1.3.1 Knowledge Representation

How can one *represent* the knowledge in the computer? One must first consider what sorts of things are required to achieve the tasks at hand. Then appropriate formalisms must be found which can encode these things in a way which enables a computer to process them. It is useful to suggest a classification of the types of knowledge that computer programs typically have. There are two basic categories, or levels:

1. OBJECT KNOWLEDGE There are two types:
   - FACTUAL KNOWLEDGE: This includes the objects and relationships in the domain. This includes simple facts and figures, and is often representable using simple data structures.
     (*e.g.* See figure 1.2).
   - INFERENCE KNOWLEDGE: indicating how to reason in the domain. This includes deriving new facts about, and relationships between objects in the domain. These usually can be cast in the form of rules.
     (*e.g.* See figure 1.3)
2. META-KNOWLEDGE: This knowledge *about* knowledge. This includes abstract generalisations about the object level knowledge. An example of this type of knowledge would be a rule like the following. When diagnosing bicycle faults, first consider all the rules which have to do with problems which are easily and cheaply fixed. Meta-level knowledge is often used as *control* knowledge, guiding the making of inferences at the object level.

Here we are primarily concerned with the two types of object-level knowledge. The theories and uses of meta-level knowledge are ill understood, and constitute an active area of current research in AI. Early AI programs typically have the control knowledge implicitly embedded in the program. Recently, people have recognised the usefulness of encoding control knowledge explicitly in their programs. This is natural extension of the first main tenant of AI programs, which is to make the knowledge explicit. This issue is not discussed except briefly in the Expert Systems module.

The most important thing to realise is that any representation is simply a *stylised* version of the objects or information in the real world. We represent the game configuration for noughts and crosses as a nine digit number in base 3. We encode blank as 0, x as 2, and o as 1. The first three digits represent the contents of the top row, the next three the middle row, etc. In doing this, we have characterised all the *essential* information about a particular game configuration, but not all of it by any means. We indicate nothing about the people who are playing, nothing about the weight of the board, or what it is made of et cetera. A program to play this game would of course need to represent a game playing strategy in addition. The program would work roughly as follows: You look at the current game configuration (in the real world). You translate this into the appropriate 9-digit base three number and enter it into the computer (or, even better, suppose the computer was hooked up to a camera which could automatically generate the internal representation for the board position.) The machine churns away processing the internal representation it has for the board, the rules of the game, and the strategy, and out comes a suggested move. The details of how this may be achieved for this game are discussed in section 1.4.

In the blocks world, we represent the state of the world as a set of predicate calculus assertions. [3] In particular, we only represent the relations *on* and *clear*. Nothing about colour or size. There are other things which you may want to reason about in this domain. For example, consider the *above* relation. Your program ought to have the ability to conclude that block 'a' is *above* block 'c' since it's *on* a block (*i.e.* 'b') which is *above* block 'c'. You may well wish to use the fact that *above* is a transitive relation. Transitivity means in this case, if X is above Y, and Y is above Z, then X is above Z. [4] You might want to make explicit the fact that the table is always clear. You must consider what you want to *do* with the blocks. You will presumably want to move them from place to place. How shall you represent this? You might have a *move(X, Y, Z)* command which moves block X from Y to Z. Alternatively, you might wish to break this action down into smaller parts. This could include commands such as *unstack(X)*, *stack(X,Y)*, *pickup(X)* and *putdown(X)*. Are there many agents capable of stacking and unstacking blocks, or perhaps only a single robot arm. How many blocks can be held by an agent at one time? If there is a single agent capable of holding no more than one block at a time, then it will be necessary to keep track

[3] See appendix ?? if you are unfamiliar with predicate logic.

[4] In general, the relation *foo* is said to be transitive if for all X, Y, and Z, foo(X,Y) and foo(Y,Z) implies that foo(X,Z). Many relations are transitive. Other examples include *ancestor*, *taller than*, and *heavier than*. Examples of relations which are not transitive are: *father of*, *can consistently defeat in a game of squash*. Can you think of others?

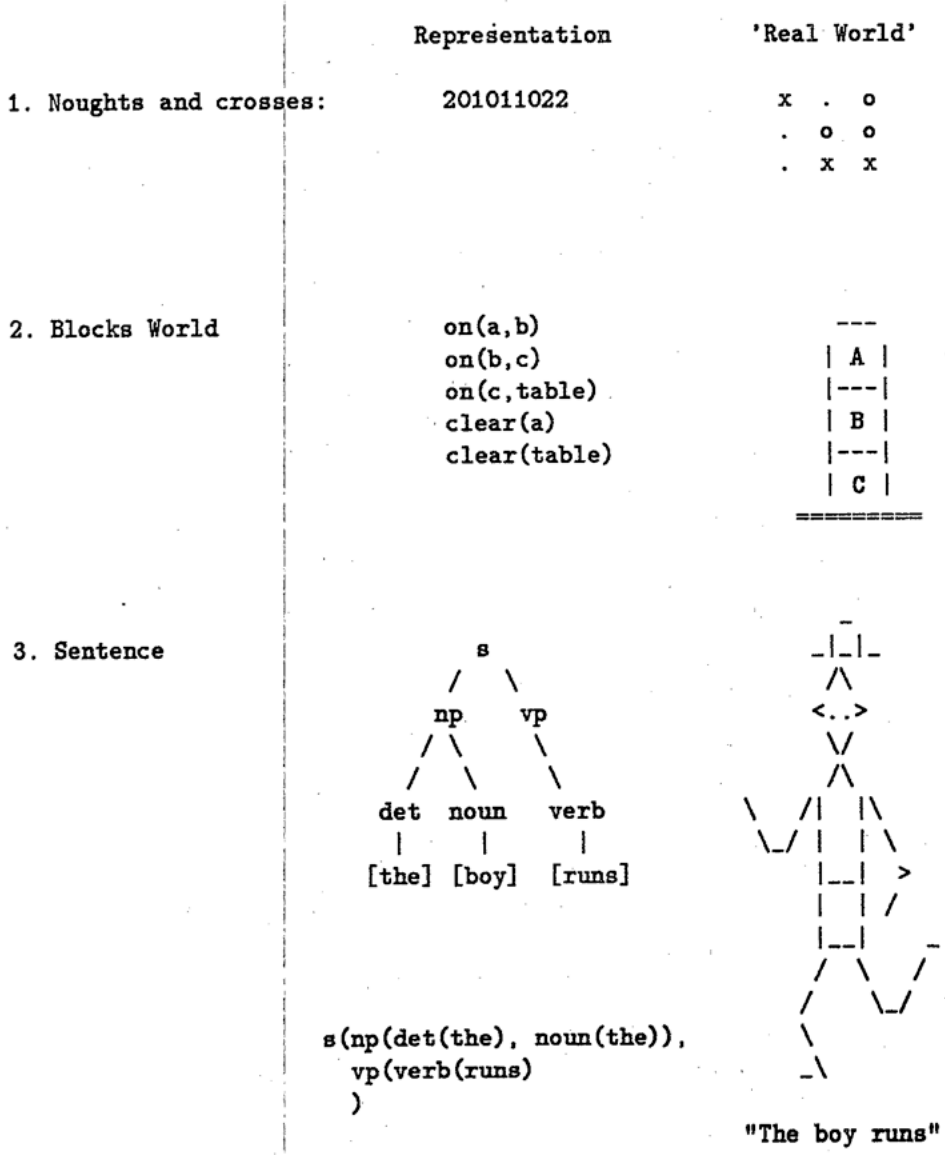


Figure 1.2: Examples of Representations I

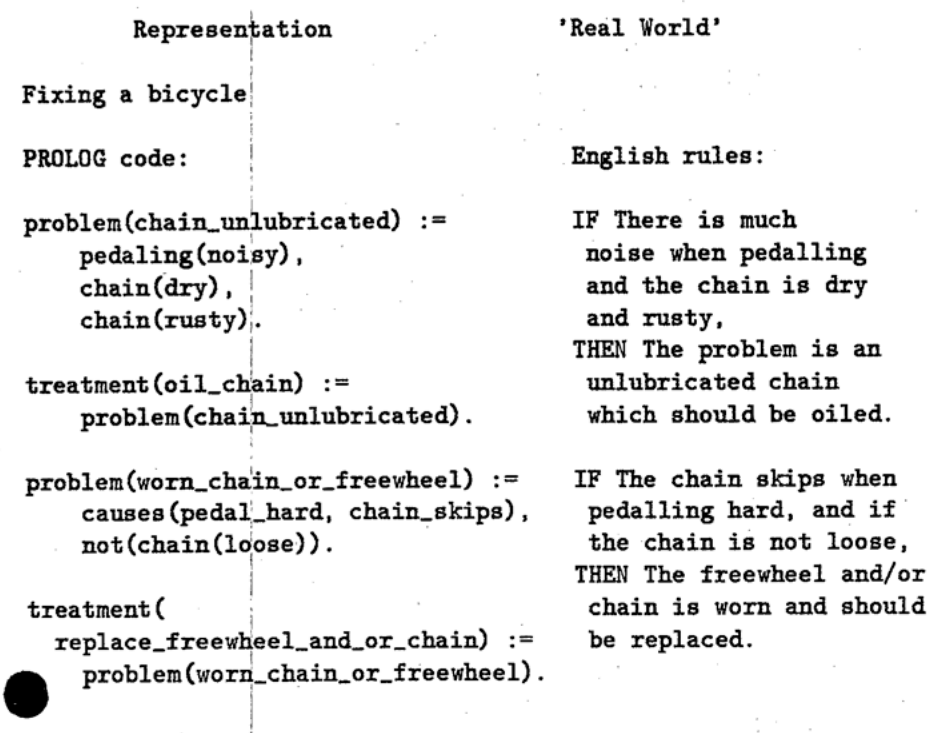


Figure 1.3: Examples of Representations II

of whether or not the robot's arm is empty in order to determine what actions the robot might take to accomplish a task. For instance, if it is holding block 'a', then it can only perform the action *putdown(a)*. If the robot arm is empty, then it may be possible to perform *unstack(X)*, *stack(X,Y)*, or *pickup(X)*. Whether it is possible depends, of course on what the current configuration of blocks is, and what the values of X and Y are.

In the third example, we have chosed to represent only the syntactic structure of the sentence: "The boy runs". There is nothing about the boy's name, who his girlfriend is, how fast he is running etc.

Finally, consider the representation for knowledge about how to diagnose and repair bicycles. The example illustrates one formalism which has found extensive application in a variety of AI programs (*i.e. production rules*). Figure 1.3 shows the Prolog code in addition to it's English translation. A complete program for diagnosing and treating bicycle faults would need many more rules to cover the whole range of possible faults. Additionally, it would require some representation for all of the parts of the bicycle, and their relationships to each other. For example, they may be grouped into separate subsystems such as the transmission, the brakes, etc. There are many ways one could represent this information.

There is one critical assumption implicit in the AI program model presented in figure 1.3 about the nature of the internal representation: *The meaning of each bit in the representation must be unambiguous.* That is to say, it must be possible to translate any portion of the internal representation into it's real world interpretation. Ambiguity arises in a number of ways:

Referential ambiguity For example, pronouns in natural language must refer to a specific person or object. This will usually be available from context. One could use a specific name such as 'Dave', but there might be many Dave's. Generate a symbol: *dave-1* to remove all ambiguity. These are also called *instances* or *tokens*.

Word sense ambiguity For example, define *caught(X,Y)* to mean *X caught Y*. But *caught* is ambiguous. Catch a cold, or catch physical object. A solution: is to use two predicates *catch_object* and *catch_illness*. This would not always be a problem, for instance if we limited our discussion to health problems and knew a priori that there would never be an occasion to be catching objects.

This critical assumption cannot be overemphasised. If there is any sort of ambiguity, then the whole system is potentially unreliable and it's usefulness is severly limited. Another way to express this is to say that there must be some consistent

interpretation (in the world) of an internal representation. Recall, the computer is only manipulating symbols; it has no notion of meaning. It is up to you the system designer to attach consistent meanings to the bits in the representation. Only then will the translation of the internal representation back into the real world be of any value. This may be summarised from a negative perspective as the GIGO phenomenon (Garbage In, Garbage Out). A knowledge representation formalism for which a mathematically precise mechanism for attaching meaning to every possible structure constructable using the formalism is said to have a *formal semantics*. It is far easier to create complex formalisms for representing all sorts of specialised knowledge, than it is to give them even semi-formally consistent semantic interpretation. Workers in the knowledge representation subfield are constantly struggling with bugs in their formalism. They think that they just solved the latest snag, and then someone comes along and points out a glaring inconsistency, or shortcoming. It should be clear from this discussion that there are many issues which need to be considered when finding representations. The task of finding the right representation for a given problem is rarely straightforward. A much more detailed discussion of various knowledge representation formalisms is deferred to the Expert Systems module. See ?? for an illuminating discussion of these issues.

1.3.2 Reasoning

So far, we have been discussing representation issues primarily, and have only mentioned inference in passing. Referring to figure 1.3, we have a 3 stage process in using an AI program. You must first translate all the relevant information into some internal representation, process the representation internally, and then retranslate back into real world terms. It is this middle stage that we are now considering. What techniques and/or methods can effectively apply the knowledge in order to intelligently solve the tasks at hand. There are two major aspects in reasoning:

Inference This consists of deriving new information from old.

Control This determines when to derive what new information.

Any and all intelligent action consists in some form or the other of concluding via some reasoning process something that was not already known. After all, if you know everything already, there are no problems, and intelligence is never called for! As a simple example of inference, consider the following. You know that whenever it snows, it takes much longer to get to work than normally. Suppose when you wake up one morning, you look out the window and see 6 inches of new snow. You can now *infer* a new piece of information from what you already know, namely that it will take you a long time to get to work that morning. This simple inference is an example of logical deduction and is necessarily valid. Other common forms of inference need not be valid. For instance, you know that when it's raining, folk usually get wet. Someone walks in from outside drenched, and you make the perfectly sensible inference that is is raining. However reasonable it may be, this is not necessarily a correct inference to draw, for there could have been a fire hydrant nearby which just sprung a leak, or any number of other possibilities. This is an example of common sense reasoning. It is distinguished from logical deduction in that, we are willing to withdraw conclusions and change what we know about the world. Because the amount of information, or knowledge can decrease as well as increase, reasoning of this sort is called *non-monotonic reasoning* and constitutes a significant area of study. [5] It's not easy to assess the 'reasonableness' of an inference. If it is only tentative, then how can we use that information to make still other uncertain inferences? Not surprisingly, getting computers to perform this sort of reasoning is very difficult, and only very limited progress has been made. There are three major types of inference: deduction, abduction, and induction. See figure 1.4 for examples.

Deductive reasoning is always guaranteed to make correct inferences, *i.e.* it will never derive false information. This type of reasoning is what we generally mean when we use the word 'logical' in everyday speech. Abductive and inductive reasoning on the other hand is uncertain. Abduction is type of reasoning which is used to generate explanations. Referring to the example in figure 1.4: having noticed that I am slurring my speech, the inference that I am drunk is drawn as an attempt to explain the slurred speech. Yet, I may not be drunk at all, I could have burnt my tongue. Finally, induction is a type of reasoning which generalises. That is, if a pattern is detected from a number of individual observations, make a general rule. The reader should identify the type of inference made in the raining example.

Knowing *what* inferences are possible to make is only part of the process of reasoning. You must also know *which* ones to make, given a choice of many. Different control strategies determine the choice of inferences to draw which in turn dictates how to go about solving problems which may normally be solved in any number of ways. Consider the bicycle example in figure 1.3. One could imagine a program which asked the user for all the details of what the current state of the bicycle is and what the problems are etc. Once this information was available, the program could look in its set of rules to see which ones

[5] This is AI jargon, other names for the same thing are *reasoning with uncertainty*, and *default reasoning*.

Deduction	From:	a. I'm drunk b. IF I'm drunk, THEN I slur my speech
(deduce)	Conclude:	c. I slur my speech
Abduction	From:	a. I slur my speech b. IF I'm drunk, THEN I slur my speech
(explain)	Conclude:	c. I am drunk
Induction	From:	a. Joe slurs his speech when drunk b. Nigel slurs his speech when drunk
(generalise)	Conclude:	c. IF a man is drunk, THEN he slurs his speech

Figure 1.4: Three Common Types of Inference

were relevant thus finding the solution to the problem. Different strategies could be employed however. The system might allow the user to tell the system everything possible about the state of the bicycle, and then begin to use that information in conjunction with its set of rules and make conclusions. Alternatively, the system may ask a few general questions first to get some clues or 'hunches' as to what may be wrong. The system then directs the questioning selectively in order to verify or deny these hunches. In other words, it attempts to make specific inferences (*i.e.* to confirm or deny a hunch), rather than simply inferring whatever it can.

As another example illustrating alternate control strategies, consider high school geometry problems. Suppose you are given some diagram containing line segments, and angles with only some of the sizes indicated. You are required to show two particular angles are equivalent. One approach would be to start with the given diagram; try to think of all the axioms and theorems which can be applied; and keep applying them in hopes of getting the answer. Another approach would be to start by looking at the desired result and to find axioms and/or theorems which would give the answer if only they applied. Suppose you find one; call it theorem **A**. Unless it is an easy problem, this theorem won't apply to the original diagram. You continue your attempt to solve the problem by asking yourself what would have to be true in order for theorem **A** to apply. You then set yourself to the task of showing that those things are true. If you can do this, then theorem **A** will apply and the original problem will be solved.

The first strategy in the geometry example above is often called *forward reasoning* because you start with what is given in the beginning and work forward by applying axioms and theorems until you get to the desired end. Conversely, the second approach is called *backward reasoning* since you start with what needs to be true at the end, and work backwards trying to apply rules until the rules that you wish to use are applicable in the original diagram.

In the bicycle example, the first approach described is akin to forward reasoning. The idea is to start with the current state of affairs, and apply the rules to see what conclusions can be drawn. The second approach is somewhat mixed. It starts out in a forward mode, asking a few questions and making some tentative conclusions. Then, it switches to a backward reasoning strategy by trying to verify a hunch.

Note that the choice of whether to reason forward or backward is often not clear. Forward reasoning, since it infers whatever it can, may get stuck inferring lots of useless information. In pathological cases, it can go on inferring new things ad infinitum. Yet a forward reasoning system has the advantage of knowing everything that it can, so that if any queries are made, the answer will be immediately forthcoming. When reasoning backwards, it may take longer to process queries, but advantageously, inferences are only made when necessary.

1.3.3 What is different about an AI program?

The reader may be wondering at this point how what we are talking about is different from what we might call conventional programs. What about highly complex programs written to help send spacecraft to the moon? Surely these embody tremendous amounts of knowledge of astronomy, aerodynamics, Newtonian mechanics etc. There are certainly no hard and fast criteria by which one can unequivocably decide whether a given program is intelligent. However, there is fairly unanimous agreement that any respectable AI program has the following two characteristics. Furthermore, 'conventional'

programs typically do not.

1. There is some explicit representation of *knowledge* in the program.
2. The basis for programming is to manipulate *symbols* rather than numbers.

Conventional programs do indeed embody considerable amounts of knowledge, however, it is not made explicit, instead remaining deeply buried in the code. For example, the laws of physics are in the form of equations which are tightly embedded into the code. Two unfortunate consequences of this are that the knowledge:

1. is difficult to modify *and*
2. cannot be used to explain the program's reasoning

The knowledge used is not readily accessible to the programmers, or indeed to the program itself. Unless the code is extensively documented, the assumptions that go into it are unavailable for examination or modification. If new laws of physics were to be discovered, or more realistically, if some of the assumptions were to change, a major coding effort could be necessary. Any intelligent entity ought to be able to explain the reasoning used in making conclusions. Only if this knowledge is made explicit can this be possible. Consider again the bicycle example with knowledge encoded as rules. The rules which were used to reach conclusions can be cited as reasons for the answers given. The role of knowledge in a program is discussed in greater detail in section 1.4 where we attempt to define an 'AI technique'.

1.3.4 Symbolic Computation

The essence of symbolic computation is that the primary activity one engages in when programming is to manipulate *symbols* rather than numbers. Clearly, programs which compute trajectories for satellites are primarily crunching numbers. On the other hand, people have found that when writing AI programs the primary activity is in defining, characterising, and manipulating symbols. In the bicycle example, there are symbols for various objects such as chains, tires, grease, etc. These symbols would ordinarily be in the form of strings of letters or words. However, each symbol has a certain meaning and may only be used in certain contexts. For example you might use the symbol *chain(x)* to denote the fact that the object x is a chain. Another symbol, *replace-chain*, stands for an action, namely that the chain should be replaced. It would be inappropriate to ever compare these two symbols with each other. It should be clear that in this example there is not much to be gained by making numerical calculations. They will not contribute in the large to the problem of diagnosing and repairing bicycle faults.

The fact that AI programs tend to manipulate symbols rather than numbers is unarguable. However, one must be careful not to carelessly inject any causal relationships here. In no way can it be said that all programs based on symbolic computation are AI programs. If anything, the causal relation works the other way. The sorts of operations that need to be performed to build intelligent programs are more easily accomplished by manipulating symbols rather than numbers. As a result, many new programming languages were developed to respond to this need. The most well known and extensively used symbolic programming languages are Lisp and Prolog. It should be stressed that programming languages are simply tools, which programmers have at their disposal. Like any other sort of tool, some are appropriate for some types of job, and not for others. Although perfectly possible, it would be most awkward to write heavy number crunching programs in Prolog. Similarly, it would be a nuisance to use conventional programming languages like Fortran and Pascal for many AI applications.

Newell and Simon [[Newell 81]] have attempted to characterise what would be the necessary ingredients that any intelligent artefact, be it an animal, human, or machine, must have. Symbolic computation is central to their whole thesis. Their characterisation is embodied in what they call a Physical Symbol System (PSS) which they define as follows:

> "A physical symbol system consists of a set of entities, called symbols, which are physical patterns that can occur as components of another type of entity called an expression (or symbol structure). Thus, a symbol structure is composed of a number of instances (or tokens) of symbols related in some physical way (such as one token being next to another). At any instant of time, the system will contain a collection of these symbol structures. Besides these structures, the system also contains a collection of processes that operate on expressions to produce other expressions: processes of creation, modification, reproduction and destruction. A physical symbol system is a machine that produces through time an evolving collection of symbol structures. Such a system exists in a world of objects wider than just these symbolic expressions themselves."

In other words, a symbol system is a set of abstract entities (symbol types) that can be combined to form complex expressions (symbol structures). Examples of symbol systems include the English language, computer programming languages, and mathematical logic.

A Physical Symbol System is a machine that can represent symbol structures and manipulate them lawfully. That is, expressions are created, modified, and destroyed (*e.g.* people, computers).

The symbols themselves are arbitrary, *i.e.* just names with no special significance. There are four things about symbols which need to be distinguished:

1. The symbols themselves, (*i.e.* the signifiers).
   (*e.g.* the predicate symbol: 'apple(X)')
2. The conceptual content of the symbols.
   (appleness, *i.e.* a sweet fruit which grows on trees etc...)
3. The thing in the real world which is signified.
   (a particular apple)
4. The implementation of the symbols. How are they manipulated?
   (computer implementation)

Having defined Physical Symbol System, they go on to suggest what they call the Physical Symbol System Hypothesis: "A PSS has the necessary and sufficient means for intelligent action." This hypothesis consists of two parts:

1. Any PSS is capable of intelligent action.
2. Anything which is capable of intelligent action is a PSS.

The first of these can be empirically tested (but not proved) by building intelligent programs using PSS's (*i.e.* computers) and corresponds to the AI as engineering viewpoint mentioned above.

The second of these can be tested by studying intelligent agents (eg people, monkeys, whales) and showing that the mechanisms used to produce intelligence are indeed those of a PSS. This corresponds to the AI as cognitive science viewpoint.

1.3.5 Closing Remarks

An AI program represents knowledge explicitly, and is based on symbolic rather than numeric computation. This is not to say, however, that any program having these characteristics will necessarily be intelligent, far from it! Nor is it the case that AI programs may not do number crunching. Some applications, such as vision programs do indeed require considerable amounts of numerical computation in addition to symbolic programming.

It is also not the case that AI programs are *better* than conventional programs. AI programming techniques have been developed in response to the specific need of creating intelligent computers. There are still zillions of tasks which are most effectively performed using conventional techniques. It's simply a matter of selecting the right tool for the right job.

1.4 What is an AI technique?

This is a condensed version of a discussion presented in [[Rich 83] pp 5-13]. Knowledge is the key to intelligent behaviour, but there are serious problems with getting this knowledge into a computer and using it. Knowledge is:

1. voluminous
2. difficult to accurately characterise
3. constantly changing

We informally define an AI technique to be a method which exploits knowledge and has the following characteristics:

It captures generalisations That is to say, it must be possible to represent a general case which can be *instantiated* to a wide range of specific instances. This alleviates the need for excessive storage requirements.

Knowledge is clearly expressed It must be understandable to those who provide the knowledge, or else it will be difficult to check it for accuracy.

Easily modifiable This is partly a consequence of the previous characteristic. Can apply to a wide range of situations

In order to illustrate these points, we consider how we might build a computer program which can play noughts and crosses. We shall consider three different methods for achieving this task.

1.4.1 Method 1

Number the squares on the board as per the following diagram. We can represent the configuration of the game as a nine digit number in base three, where the digits 0, 1, and 2 signify *blank*, *x*, and *o* respectively. See figure 1.2 for an example. For each of the 3^9 (about twenty thousand) possible board positions, there is a best move which will result in another configuration which can again be represented as a base three number (this time with one more non-zero digit). The first method is to record explicitly the game configuration which corresponds to the best move from every possible position. This information can be stored in an array with 19683 elements, one for each configuration. The number associated with the 'before' configuration can be used as an index into the array. The value stored in each array element corresponds to the 'after' configuration.

To play the game, simply encode the current board position as a number in base 3. Use this to index into the array to find the board position which should result from the move from the current position. It will be encoded as a number in base 3 (easily decoded). The actual move is found by detecting the difference between the current and new position.

1	2	3
4	5	6
7	8	9

Comments: Uses too much space. Not easily modifiable, if the rules changed slightly, you would have to create an entirely new table. The strategy is not easily understandable from the representation used. This approach will not extend to other problems without doing everything all over again for the new problem.

1.4.2 Method 2

Number each square on the board as in the previous method, however this time, encode the values of each element as follows: blank=2, x=3, and o=5. This simplifies the computation of whether or not it is possible for either player to win. Quite simply, X can win if the product of the numbers in a row, column, or diagonal is 18, and O can win if the product comes to 50. We proceed by encoding three sub-procedures to be used by the main game playing algorithm:

Make2 : Tries to make two in a row (or column or diagonal). Try the center first, then the noncorners.

Posswin(p) : Returns 0 if player 'p' cannot win on the next move, otherwise, it returns the number of the winning move.

Go(n) : Make a move on square 'n'

We then encode an explicit strategy for playing the game. This consists of a decision procedure for where to go on each move depending on certain board characteristics. (NB: We assume that X moves 1st) For example:

Move 1(X): Go(1) (*i.e.* upper left corner)
Move 4(O): If Posswin(X) is not 0, then go(Posswin(X))
Else: go(Make2)

The second move encapsulates the strategy of first preventing the opponent from winning on the next move, otherwise try to get two in a row. *Comments*: Much more efficient on space, but much slower. Much easier to see the strategy and modify it. We still cannot generalise this to another domain, without manually encoding a new strategy for the new problem.

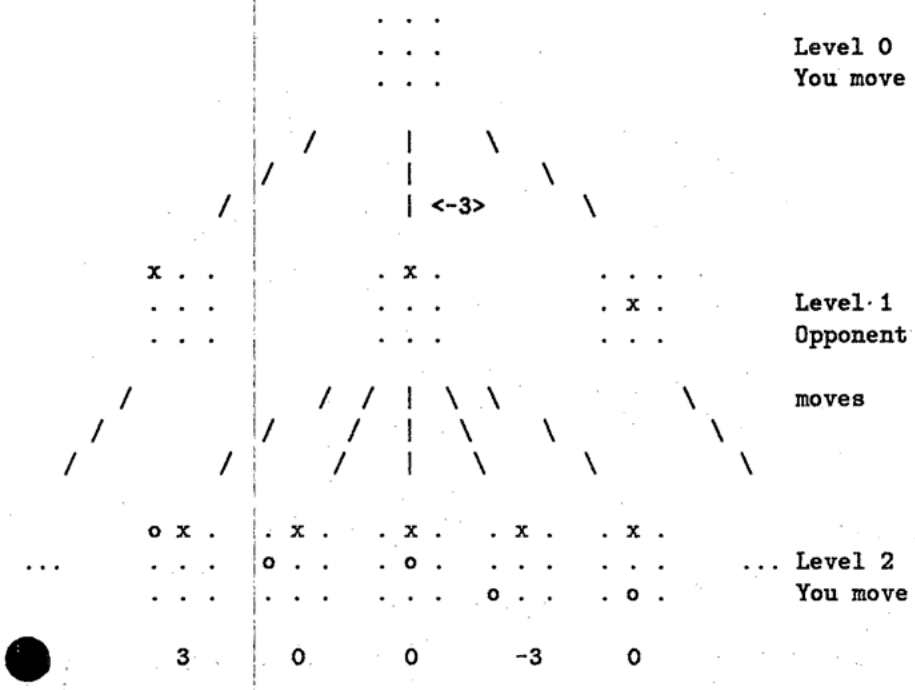


Figure 1.5: Partial game tree for noughts and crosses

1.4.3 Method 3

We describe informally a procedure for finding the best move from any given game configuration. First, find all possible moves from the starting configuration. Then, choose the move which corresponds to the resulting configuration which looks most favourable. We need a procedure for measuring 'favourableness' which enables the assignment of a number to each possible resulting configuration. This number estimates how likely it is that you will win from that configuration. Unless the configuration corresponds to a definite win or loss, it may be difficult to find such numbers which accurately assess the game situation (more on this in section ??). It would certainly help if we could look further ahead than a single move. The idea is to look ahead as many moves as resources permit, thus computing a large number of possible resulting game configurations. This is most conveniently represented in a *game tree* (see figure 1.5). [6] We start by assigning numbers to each leaf node at the bottom of the tree as in the figure. When doing this, you might take into account such things as whether the configuration forces your opponent to make a move, or who has certain corners. The key observation which shows us how to analyse this tree, is that *you* will attempt to MAXimise your chances for winning (*i.e.* pick the move with the highest number), while your *opponent* will try to MINimise your chances of winning (*i.e.* pick the move which has the lowest number). For example (see figure 1.5), if you were to make the move with 'x' in the middle of the top row, then your opponent would put their 'o' in the lower left corner since this corresponds to the lowest number and is thus his/her best move. We have now predicted what the outcome of that move ought to be, so we assign to it the corresponding value (*i.e.* -3). Similarly, we assign numbers for the other two moves and then pick the one which has the highest number (the details of the other two moves are not in the figure)..

This procedure is known as MINI-MAX search and is discussed in greater detail in section ??. Finding procedures for assigning numbers to evaluate goodness of game configurations is part of the study of *heuristics* which is discussed in detail in section ??.

Comments: This is much less efficient than methods 1 and 2, but for large problems, the other methods are less feasible. There would be insufficient space for the first method, and for the second method, it would be difficult to characterise a strategy fully. The major difference in this method is its potential for extensibility in other domains. In fact, it has been applied with some success to such games as chess, draughts, go, and many others. The way to create the game tree depends

[6] Consider configurations which are symmetric as being identical. Thus, instead of nine starting moves, there in fact only three (side, corner, and center).

on the rules of the game. The way to assign the number which assesses how good the game configuration is will also depend on the game. In chess, for example, you must take into account not only the value of the pieces on the board, but their relative positions as well. Other than this, the way to search the tree, (*i.e.* the MINI-MAX procedure) is unchanged. This latter method is a good example of an AI technique.

1.5 Key Subfields in AI

1.5.1 Major Subfields

Knowledge Representation and Inference

This is the heart of AI. Any sort of intelligence that one can imagine will certainly require the existence and application of knowledge. We humans know an awful lot, but we don't know very well how that knowledge is stored in our brains. Neither do we understand much about the mechanisms of accessing the knowledge and making inferences from that knowledge. This very large and significant subfield concerns itself with these issues.

Planning and Search

This subfield used to go under the single label: *Problem Solving*. Solving problems invariably concerns itself with creating some *plan* of action to solve the problem. Creating this plan invariably requires some sort of implicit or explicit *searching* through all the possible approaches to solving the problem. This is another 'core AI' subfield which is relevant to most of the rest of AI.

Vision

How do we see objects? More importantly, how to we recognise what it is that we see?

Natural Language Processing

The goal here is to build computers which humans can converse with in their native languages, rather than some highly formal computer language. Most work has concentrated on English, but there are a number of other languages under study, including, Japanese, Chinese, Italian, French, and Italian.

Expert Systems

This is to a large extent, applied AI. Expert systems have been defined to be computer programs which can perform some task which is difficult enough to require genuine human expertise (as opposed to simply requiring some intelligence). The range of applications to date is extremely wide, from medical diagnosis, to mineral prospecting to nuclear power plant monitoring systems.

1.5.2 Other Subfields

There are many other more esoteric and/or more specialised subfields in AI. This is by no means a universally accepted breakdown of the field, but close enough for our purposes. It happens to reflect the choice of the organisers of the most recent International Joint Conference on Artificial Intelligence in summer 1985. These are summarised below:

Philosophical foundations

What is the nature of intelligence? What is knowledge? What are the philosophical/moral implications of building computers which are even smarter than humans? Will computers ever think? Will computers ever achieve self-consciousness?

Cognitive Modelling

What are the actual mechanisms which are responsible for producing natural intelligence in people and/or other animals. This overlaps considerably with psychology and places much less emphasis on actual computing.

AI Architectures

This is anything but a coherent subfield. However, people are studying a variety of architectures for building AI systems. Many have been built in an ad hoc fashion, in pursuit of solving a specific problem and are identified only after the fact.

Robotics

Building robots which can perform motor tasks like going through a room or picking up an object form an assembly line, or hoover your carpet. This field is to some extent away from mainstream AI and overlaps considerably with physics, applied maths, mechanical engineering etc. On the other hand, successful robots of the future will need to apply technology from many other areas in AI including planning and search, vision, knowledge representation and inference

Automatic Programming

How can we get computers to program themselves? Start with high level specifications for what the program is to do, and eventually write the program. This draws heavily on the theory of programming from computer science, and is very closely related to the planning subfield. Writing a program to achieve a variety of tasks which may interfere with each other will involve a considerable amount of very careful planning. Finding adequate formalisms for specifying programs and transforming this into a correct program draws considerably from the knowledge representation and inference subfield.

Knowledge Acquisition and Learning

This area concerns itself with acquiring the knowledge which is needed for AI programs. There are a number of techniques which have been tried which vary from labour-intensive approaches where knowledge engineers interview the experts and try to formalise the expertise, to automated approaches where a knowledge base is created automatically by noticing patterns in a data base of actual cases.

AI in Education

How can computers be used to teach? Other related issues include building intelligent interfaces to existing computer systems, building models of the users which can be used to expedite learning, etc.

Logic Programming

Using logic based formalisms directly for programming. The best example of this is PROLOG.

Theorem Proving

How can theorems be proven automatically? This is closely tied in with logic, and is the basis for implementing languages like PROLOG

1.6 Summary

In this chapter, we have given a general introduction to the field of Artificial Intelligence. A number of definitions were offered, a the most popular of which is roughly, "the art of building computer programs which perform tasks which require

intelligence". We noted that the primary ingredient of all intelligent behaviour is *knowledge*. Thus, the central tasks, faced by builders of AI programs are how to represent and apply knowledge. Intelligence requires not only the existence of great stores of information (*i.e.* knowledge about the world) , but also the ability to *reason* with this knowledge. The two crucial components of the reasoning powers of a truly intelligent entity are:

- **inference:** the ability to derive new information from old (*e.g.* logical deduction)
- **control strategy:** knowing *what* information should be derived *when*.

We identified two characteristics which tend to be unique for AI programs:

1. Knowledge is represented explicitly.
2. The basis for programming is in the manipulation of symbols rather than numbers.

Representing the knowledge explicitly in a program enables easy modification because the assumptions behind the code are evident. It also becomes feasible for the program to justify it's reasoning, or decisions. The importance of symbolic computation is evidenced by the new style of programming which arose as a response to the needs of people who wanted to build intelligent programs. It has even been postulated (but not proven) that any 'physical symbol system' is capable of exhibiting general intelligent behaviour. Finally, we presented a very brief survey of the major and minor subfields of study within Artificial Intelligence.

In the remainder of these notes, we shall only consider one of these subfields: Planning & Search. This is an outgrowth of the more general field of Problem Solving which was a major preoccupation of AI researchers in the early days. We begin by considering conceptual frameworks for solving problems.

In doing so, we motivate the need for search. In chapter ?? we explore a wide variety of search techniques. In section ?? present some formal methods of searching state space representations. It is here that we introduce the use of heuristics in problem solving, which is a general technique applicable in non state space representations as well. In section ?? we address the issue of search as applied to the problem reduction representation. Finally, in section ?? we will discuss search techniques for game playing. We will see that the problem of finding game playing strategies can naturally be cast into the problem reduction framework.

In chapter ?? we introduce *planning*, as it is currently viewed in AI. In particular, we discuss some of the limitations of simple state space search as it is used to create plans.

1.7 Further Reading

Every AI text introduces the subject in a different way. I have distilled what I deem to be most useful from a number of sources. The model of an AI computer program is borrowed from chapter one of [[Charniak 85]]. The brief discussion of ambiguity is from chapter six of the same text. A more extended discussion of physical symbol systems may be found in [[Nilsson 81]]. The discussion about what constitutes an AI technique is a condensed version of the introductory chapter of [[Rich 83]].

Chapter 2

Problem Solving Paradigms

Early workers in AI attempted to write programs to solve simple problems, puzzles and games which required a certain amount of skill for humans. By looking at some simple problems, and intuitive approaches to solving them, they began to characterise some general approaches to solving problems. Two issues of fundamental importance emerged:

- Choosing Representations
- Search

The representation issue is primarily to do with finding the appropriate perspective from which to view a problem. Viewed correctly, an otherwise difficult problem can appear trivial. Defining a representation for a problem consists of three tasks:

1. finding structures which can describe the problem
2. defining rules for manipulating these structures for the purpose of finding a solution.
3. identifying in terms of these structures what constitutes the solution to the problem

In this chapter, we explore in detail the issue of finding representations for solving problems. In particular, we examine two problem solving paradigms which have been used extensively by AI workers. A problem solving paradigm can be thought of as a framework for representing problems, or equivalently, a set of rules for formulating a problem in such a way that the method of finding a solution is well defined. In other words, a problem solving paradigm provides a structured way for achieving the three tasks listed above. We shall study two such paradigms, *state space search*, and *problem reduction*.

The search issue concerns itself with finding the appropriate strategy for attacking problems, once formulated. Although essentially different in nature, the issues of finding representations, and search are not at all independent. By carefully formulating a problem, the amount of search can be drastically reduced. Indeed, as we will see, some problems when viewed correctly, are in fact trivial. That is to say, there is no search required! This issue is treated in detail in chapter ??.

2.1 State Space Representation

2.1.1 Formal Description

This highly celebrated approach to problem solving is best described in the following quote [[?], p7]:

> "We postulate some kind of space in which treasures are hidden. We build symbol structures (nodes) [1] that model this space, and 'move' operators that alter these symbol structures, taking us from one node to another. In this metaphor, solving a problem consists in searching the model of the space (selectively), moving from one node to another along links that connect them until a treasure is encountered."

[1] *Node* is a term used by mathematicians and computer scientists when describing data structures such as trees and graphs.

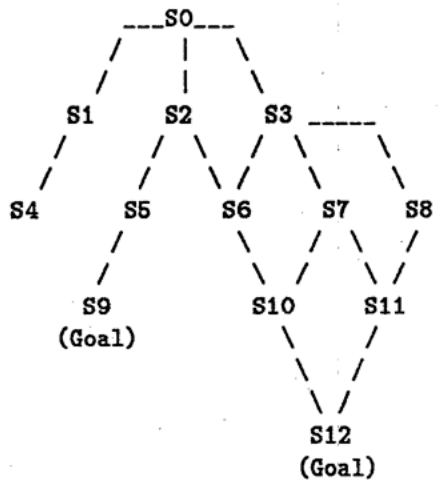


Figure 2.1: A Generic Search Space

Depending on the specific requirements of the individual problem, the solution may be the treasure itself, or it may be the path to the treasure consisting of a sequence of move operators. The important thing is to be able to recognise that a treasure has been encountered.

This space which is being searched is called a *search space* (see figure 2.1.1). A node, *Si*, is referred to as a state, which can in principle be any sort of object at all. For the time being it is convenient to restrict the usage of the term state to refer to a world state. By that we mean some partial description of the world which nevertheless is a complete characterisation containing all the information relevant to the problem. For example, in the noughts and crosses example, a state would consist of a complete description of the state of the game, (*i.e.* the locations of all the x's and o's). Other interpretations for a state will be considered briefly in section 2.3.3, and in considerable detail in chapter ??. A node at which a treasure is present is called a goal state node, or simply, a goal state.

A link joining state *Si* to *Sj* indicates that there is an operator which may legally be applied to *Si* which transforms the problem state into that represented by *Sj*. Operators are often called *actions* because they correspond to real actions being performed by agents whose effects are to change the state of the world. New things become true, and things which previously were so, no longer are.

In order to solve a problem using the state space paradigm, you must first formulate it by performing the following tasks:

Represent States: Design a structure for representing a world state. It must contain all of the information relevant to the problem.

Initial State: Give a description of the initial state using this structure.

Operators: Characterise all operators which can transform one state into another.

Goal State Recogniser: Devise a test which can recognise whether or not a goal state has been reached.

Once formulation is complete, the framework is set, but we still have to solve the problem. In the state space paradigm, this consists of finding a sequence of operators which, when applied, transform the initial state into a goal state. Graphically, this operator sequence corresponds to a path of arcs through the search space. Finding the solution path requires search.

We shall now illustrate by way of two example problems how the state space paradigm works. In what follows, we shall concentrate primarily on issues of representation. We will also see how much this impinges on the resultant search required. A more formal discussion of search techniques is deferred until chapter ??.

2.1.2 The Monkey and Bananas Problem

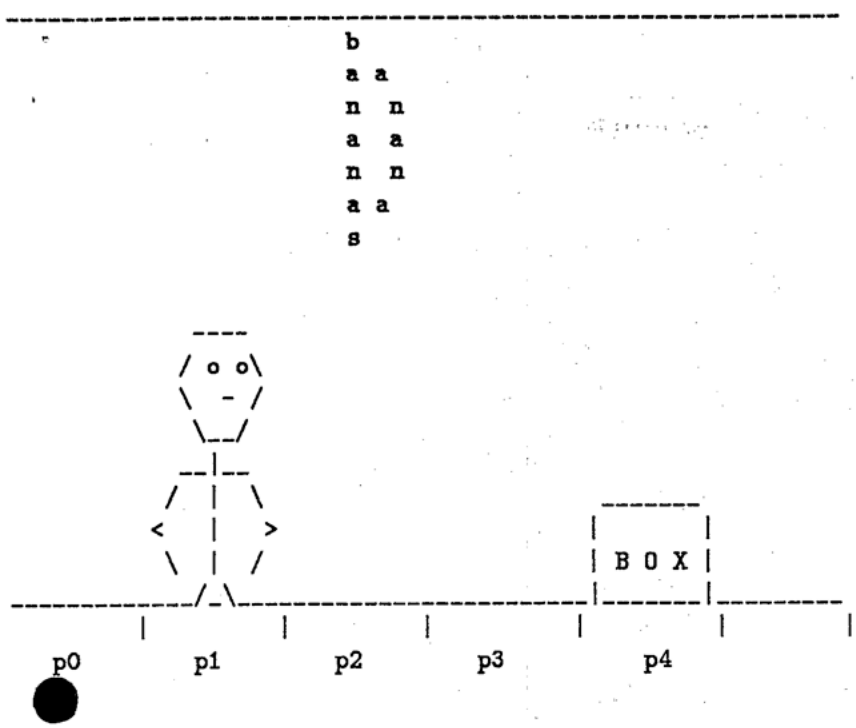


Figure 2.2: Monkey and Bananas Problem

This problem is a long time favourite for illustrating state space search. Consider the scenario depicted in figure 2.2. The monkey wants the bananas, but cannot reach them while standing on the floor. The monkey may move the box from place to place, and also has the ability to climb on to the box (which would give it the necessary height to reach the bananas). For us, there is really no problem here. It's obvious what the monkey should do. But does the monkey know? More to the point, how could we represent this problem to a computer and how might we tell it to go about finding a solution?

We might use predicate logic assertions to represent the state of affairs at any point in time. Figure 2.3 shows what the initial state and goal state recogniser look like.

Note that we are not interested in a complete description of the goal state. It suffices to test for the single assertion. In particular, it does not include the location of the monkey and/or the box after the monkey has the bananas.

The operators correspond to the actions which the monkey is capable of performing. These are:

- *climb_on*: climbs on to the box from the floor
- *push_box(X,Y)*: push the box from place X, to place Y
- *go(X,Y)*: go from place X to place Y
- *grab_bananas*: grabs the bananas.

Note that the second two operators are actually characterising many possible actions depending on the values of X and Y. This is analogous to a procedure in a computer program. Operators in this form (*i.e.* with variables) are referred to as *operator schema*. The operators need to be further formalised. In particular, it is necessary to restrict the application of these operators to only those states from which these actions would be possible. For instance, it would not be possible for the monkey to push the box unless it was in the same place as the box. Furthermore, it is necessary to model how the state of the world changes as a result of the monkey performing one of the actions. One concise way to represent this is to summarise the preconditions and effects of each action in terms of the predicate calculus assertions which affect or are affected by the action. The operators can be completely characterised by three lists, the preconditions, the add list, and the delete list (see figure 2.3). The preconditions are all the assertions which must be true before the action may be performed. The add list consists of all the assertions which become true as a direct result of the action. The delete list, conversely, is the list of assertions which are no longer true as a direct result of the action. For instance, pushing the box from p4 to p3 results in the deletion of the assertion *at(box,p4)*, and the addition of the assertion *at(box,p3)*.

Initial State	*Goal State Recogniser*
at(monkey,p1) at(box,p4) at(bananas,p2) on(monkey,floor) status(bananas,hanging)	Is the assertion: 'status(bananas,grabbed)' in the database?

Action	*Preconditions*	*Add list*	*Delete list*
climb_on	at(monkey, X) at(box, X)	on(monkey,box)	on(monkey,floor)
push_box(X,Y)	at(monkey, X) at(box, X)	at(monkey,Y) at(box,Y)	at(monkey,X) at(box(X)
go(X,Y)	?	?	?
grab_bananas	?	?	?

Figure 2.3: Problem Formalised

There are other details which you also would need to provide the computer. These include such things as the fact that p1-p5 are places. This could be done by putting five assertions in the database of the form: *place(P)*. You would want to restrict the types of things that can go in certain argument positions in the predicates. Thus, the predicate *at(X,Y)* must have a place as its second argument. For example, it won't try matching *at(monkey, box)*. The reason for doing this is that when the computer goes about solving the problem, it will have less work to do.

Search Strategy: Finding a solution.

At this point, the representational issues have been settled. The problem has been formulated. We have an initial state, some actions which can change the world state, and a test for recognising a goal state. The search space for this problem consists of all the possible world states which can result from the monkey performing any of its actions. That is, let the monkey move about, pushing the box from here to there, climbing on the box, grabbing the bananas, etc. It is important to realise that the search space itself, will never exist on the computer explicitly. Rather, it is in the definitions of the initial state, actions, and goal states which implicitly define it. It is not searched by generating the whole thing and looking at it. Instead, it is generated on the fly, as necessary. Once a path is found to a goal state, computation stops. See figure 2.4 for an illustration of a partial search space for the monkey and bananas problem.

One thing that is clear from this figure, is that there are indeed a myriad of choices to be made in finding a path to a solution. How to make these choices effectively is a control issue, and is the essence of search. The key question is in determining which actions should you apply and when? At any point, there may be a number of possibilities. How can we decide? The obvious and naive thing to do is to use a *forward reasoning* strategy, randomly applying any action it can. This could lead to pushing the box back and forth till the cows come home. Preferably, there would be some way to determine which of the choices are likely to lead to a solution. How to do this is not always clear.

An alternate method would be to reason backwards from the goal state. For the monkey to have the bananas, it must have performed the action: *grab_bananas*. This is because it does not have them to begin with, and no other action has on its add list anything about the bananas being grabbed. The conditions which must be true before it can grab them are that it is under the bananas and standing on the box. We could then turn our attention to the slightly smaller problem of putting the box and the monkey under the bananas. It turns out that with our current problem formulation, this approach is not is not entirely straightforward to implement. A suitable technique for implementing this strategy is called *means ends analysis*. It is described in detail in section ??.

In this example, it seems much more sensible to reason backwards. This is not surprising, since you are using information about the goal state to solve the problem rather than somewhat arbitrarily acting from the initial state in hopes of getting to the goal state.

EXERCISE 321 Represent the preconditions and effects for the actions *go(X,Y)* and *grab_bananas*.

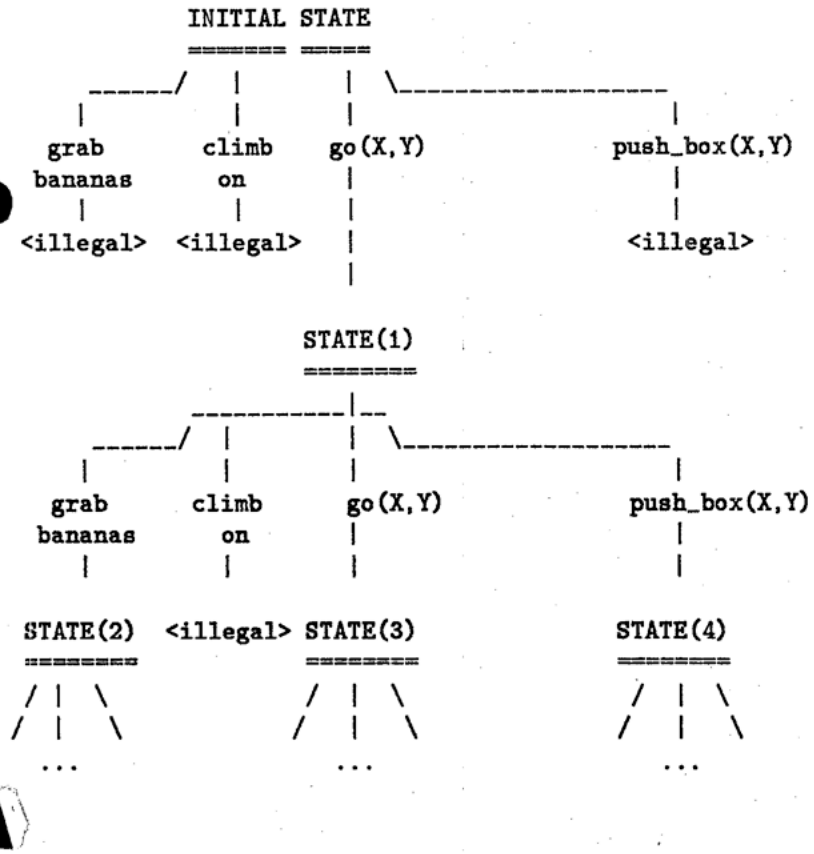


Figure 2.4: Partial Search Space

EXERCISE 322 Suppose our poor monkey climbs on the box and is not under the bananas. According to our formalisation of the problem, the game is over, for it has no way to get off. Is a *climb-off* action required? Why or why not? What are the advantages and disadvantages of adding this action?

2.1.3 8-Queens Problem

This problem can be succinctly stated as follows:

> Place 8 queens on a chessboard so that no queen is attacking any other queen.

Recall, the four tasks required for solving a problem using the state space paradigm are: first,determine a structure for representing states; then give the initial state; thirdly, define operators which transform states; and finally, define a test which can recognise goal states.

In this case, a world state consists of some configuration of up to eight queens on a chessboard. It is easy enough to draw a picture of a chessboard with queens on various squares, however we must find a representation suitable for the computer. To this end, let each square be identified with an index into a 64 element array called 'board'. Let the value of each array element be 'empty' if there is no queen on it, and 'queen' if there is. The array itself may be one or two dimensional depending on which of the following choices is made:

1. Number the squares on the board 1 to 64. 1-8 is the first row, 9-16 is the second row, etc.
2. Identify each square as an ordered pair (i,j) where i is the row number, and j is the column number.

The remaining three tasks can be achieved in a variety of ways, depending on how one views the problem, and decides to attack it. I will present several different approaches to solving this problem which are increasingly more intelligent. We will then see how well these approaches map onto our paradigm. Some of the differences will affect the way we formulate the problem which can have a significant effect on the amount of search required to solve it. Other differences among the increasingly intelligent approaches described below will be purely search control issues which are taken into account only after the problem has been formulated. Details of the search issues are deferred until chapter ??.

Trial and Error-'Dumb'

The most naive approach which a child might use is to randomly place the queens down on the board and then look to see if that placement constitutes a solution. If not, then move one queen one space in any direction and check again to see if a solution has been found. Continue this process until a solution is found. This can be formulated as follows:

Initial state : The eight queens are randomly placed on the board.

Operators : The only action possible is to move a queen from one square to another. The, a following single operator schema is sufficient: *move(X,Y)*. This is interpreted as *move the queen on square X to square Y*. The necessary preconditions are that there was a queen on square *X*, and that there was no queen on square *Y*. The second precondition encodes an implicit assumption that only one queen is allowed on a square. It must not be left out!

Goal State Recogniser A given chessboard configuration constitutes a goal state if and only if there are exactly eight queens on the board *and* no two queens are on the same row, column, or diagonal. I leave this as an exercise.

The search space consists of four and a half million nodes, corresponding to all possible combinations of eight queens on a chessboard. While this formulation is correct, it does not in fact capture the trial and error aspect. This is a search control issue, not a representational one. The method is not guaranteed to find a solution, because there is no rhyme or reason behind generating the next move. You could easily encounter the same positions over and over again. There are too many to store each one and do a simple check for duplication.

Trial and Error-'Smart'

We can make a considerable improvement on the last method if we choose which queen to move somewhat more carefully. If we choose a queen that is currently under attack and move it to a square where it is no longer being attacked, it would seem that an answer would be found more quickly. This choice, which seems like a control issue, can actually be embedded into the representation itself. We need only we can alter the preconditions of the *move(X, Y)* schema above by requiring the queen on square X to be currently under attack and that square Y is currently not under attack.

In doing this, we prevent a great number of moves which were allowed in the previous formulation. Thus, the search space has been reduced considerably. It is still a trial and error method because there is no procedure which says which of the several possible moves should be tried at any point. Also, as before, there are no guarantees that a solution will indeed be found. A position might be reached from which no move is possible. The reader should verify this to himself or herself. It may or may not be possible to recover from such a position by reconsidering previous moves.

Systematic-'Dumb'

We would like a method which is guaranteed to find a solution if one exists. This can be done if all the possible placements of queens are generated systematically. For example, we might number the squares on the board 1 to 64. Start by placing seven queens on squares 1 through 7. Take the eighth queen and generate the first 57 possible board configurations by placing it on square number 8, then 9, ... up to 64. Each time check to see if a solution has been found. At this point, all the possible board configurations have been tested which have squares 1-7 covered. Now, move the queen on square 7 to square 8. We now try all possible placements of the last queen which yield new configurations. The case where it's on square 8 has already been tested, so we begin with square 9 , then 10, ... to 64. These are the next 56 possibilities (assuming that no solution has yet been found, which, of course, it will not have). We continue this way until all possibilities have been tried, or until a solution has been found. [2]

It is not so easy to see how to reformulate the problem to get this behaviour. The initial state is with all queens on the first row. There are never any choices to be made, the next state is always predetermined. It would be very difficult indeed to structure the operator preconditions so as to guarantee the right move for the right situation.

This method has the advantage of guaranteeing that a solution will be found. Even so, there are still 4.5 million possibilities which need to be checked. The majority of these can readily be identified as being losers by we 'intelligent' folk without even thinking about it. A systematic, yet unintelligent approach like this is commonly referred to as a *brute force* method. In spite of necessarily being able to find a solution, this approach might actually prove worse than the previous trial and error method since the latter has a smaller search space. This would be especially likely, if the latter method was equipped with some capacity to recover from a dead end. What we need is some combination of the above two methods, *i.e.* a systematic approach which chooses moves 'intelligently'.

Systematic-'Smart'

Let's make the observation that there is never any reason to knowingly have two queens on the same row. Instead of placing all the queens on at once and checking to see if it is a solution, we begin by placing the first queen on some arbitrary square in the first row (*i.e.* squares numbered 1-8). Proceed by placing the next queen in a free square on the second row (9-16). A square is *free* if it is not under attack by any queen on the board so far. We solve the problem by placing one queen at a time on the board, one per row. If at any point, we get to a row which has no free spaces, then go back to a row where there was another option and try that one instead. Continue this process as before until the eighth queen is on the board, or until there are no more choices which have not been tried. Note the similarity between this approach and PROLOG backtracking.

The new problem formulation is summarised below:

Initial state : There are no queens on the board (*i.e.* Every element in the array has the value 'empty').

Operators : The only possible action is to put a single queen somewhere on the chessboard. This is formalised by the operator schema: *put(X)*. This is interpreted as placing a queen on square X. The preconditions are that square X is not currently under attack.

[2] The study of the principles underlying this process of enumerating all the combinations of possible queen placements is a subfield of mathematics called *combinatorics*.

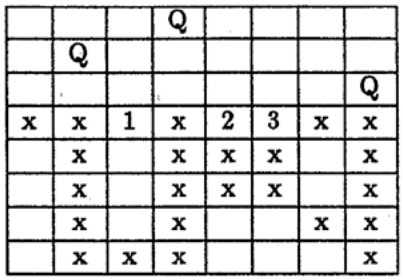


Figure 2.5: 8-Queens Problem

Goal State Recogniser The goal state is reached if all eight queens are on the board.

This relatively minor change in point of view, has a rather major impact. This new formulation has a number of advantages. First, the search space is dramatically smaller. Thousands, perhaps millions of states are avoided. Furthermore, we save a great deal of computation in the goal recognition phase. The test is now trivial. We avoid much redundant computation which was necessary when the entire chessboard had to be checked each time for attacking queens. The work is done in the formulation of the operator. You only have to check for problems with each new queen as it is placed on the board. A solution is guaranteed as soon as the last queen is in place!

The method of search, is still undetermined by this formulation. There are actually two aspects to this method. The first, namely placing the queens one by one only on to free squares, is a representational issue which ultimately changes the entire problem formulation. The second, which says how to recover from a dead end is purely a search control issue. It ensures that all possibilities are checked, thus guaranteeing that any existing solutions will be found.

As in the smart trial and error method, dead ends are possible. The method described above is known as as *backtracking*. However, it is still a brute force method enumerating all the possibilities. We are still basing decisions about what to do next on an arbitrary selection procedure (perhaps left to right) instead of on the basis of how promising a move might be. Even so, by looking at the problem in from a different viewpoint, we have effected considerable savings.

Heuristic Method

The final technique that we will discuss is a straightforward, and in fact fairly minor variation on the previous method of placing queens one at a time, row by row. The difference will be in how we choose among the options each time a new queen is being put down. Instead, of choosing an arbitrary order, and applying it systematically, we will use some sensible guidelines for selecting a square to place each successive queen onto the board. Suppose we have already placed the first 3 queens as indicated in figure 2.5. There are 3 choices for the placement of the next queen (assuming that we do the rows in order). How can we select the one which is most likely to lead to a solution quickly, thus avoiding the costly backtracking? A good rule of thumb, is to make the choice which will leave the most number of options open for later queen placements. This could be formalised in a number of ways. For example, consider the move *1*. If the queen was to go here, there would be a total of 8 free spaces remaining on the whole chessboard. The other two choices, *2*, and *3* would leave 9 and 12 respectively. Thus, the best move is option *3*. This number is a measure, or an *estimate* of how good a choice is.

This method is certainly a good deal more *intelligent* than the previous ones. In the jargon of Artificial Intelligence, it is called *heuristic search*. A heuristic is simply a rule of thumb, some hint about the problem domain which will help solve the problem quickly. The procedure for measuring how promising a possible move is likely to be is called a heuristic evaluation function. It is a function (in the mathematical sense) because a specific number is associated with each situation.

EXCERCISE 340: Determine goal test algorithms for each of the two representations suggested for the chessboard (*i.e.* the one and two dimensional array). Which is better? Why?

EXERCISE 341 Identify any differences in the problem formulation which are required for the heuristic search approach as compared with that used for the smart systematic method? If there are any, how does the change affect the search space? If not, explain why.

EXERCISE 342: Think of other ways to formalise the general heuristic of keeping your options open for the 8-queens

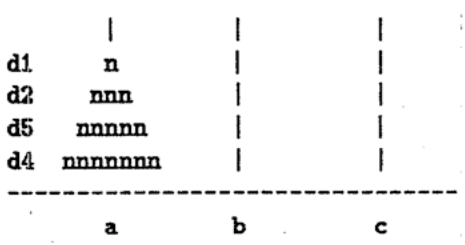


Figure 2.6: Towers of Hanoi

problem. In other words, what other heuristic evaluation functions will act as sensible measures of how good a particular choice of queen placement is?

2.1.4 Towers of Hanoi

There are four discs stacked on tower a as in figure 2.6. The problem is to transfer all the discs to tower c under the proviso that at no time can there ever be a larger disc on a smaller one. Tower *b* may be used to store discs temporarily. I leave this as an exercise.

EXERCISE - Formulate the Towers of Hanoi problem using the state space paradigm. Draw a tree diagram of part of the search space. Only go a few levels down and ignore redundancy (*e.g.* don't consider undoing the last move).

2.1.5 Summary

We have described the state space paradigm for solving problems. In order to formulate a problem in this paradigm, you have to:

1. Design a structure for representing a world state.
2. Give a description of the Initial State using this structure.
3. Characterise all operators which can transform one state into another.
4. Devise a test which can recognise whether or not a given state constitutes a solution.

Two example problems have been discussed in detail, and alternate formulations considered. Note the dramatically different nature of the world states for the two examples. In the case of the monkey and bananas the nodes are simply sets of assertions which describe the whereabouts of the monkey and the boxes etc. In the case of the 8-queens problem, the node corresponds to a particular placement of some or all of the queens on the board. In this case, a world state is represented as a 64 element array. This is not to say that it is not possible characterise the state using assertions as before, only that it was not the natural thing to do.

Note also that there are in general many possible ways to formulate a problem even using the same state-space paradigm. We saw this in the 8-queens problem where we had different initial states, different operators, and even different tests for goal states. In the next section, we consider another problem solving paradigm, *problem reduction.*

2.2 Problem Reduction

2.2.1 Formal Description

Another quite different approach to solving a problem is to break it up into subproblems which are hopefully easier to solve. Solving the larger problem then *reduces* to solving all of these smaller subproblems. Of course, it may be necessary to further reduce the subproblems into still smaller subproblems. This process stops when the individual subproblem are so trivial that

they can't really be viewed as problems at all. We shall refer to these as *primitive problems*. They correspond to primitive operations in the problem domain which are immediately achievable. The overall problem is considered *solved* when all of the subproblems are reduced to primitive problems.

It is equivalent to speak of goals, rather than problems. The goal corresponding to a problem is simply: to solve that problem. Thus, we can speak of reducing goals to subgoals, instead of reducing problems to subproblems, and of achieving goals rather than of solving problems. I shall use both terminology interchangeably.

Generally there are two ways to reduce a goal:

1. By enumerating a number of alternative subgoals *any* of which if achieved achieves the original goal. The alternatives can be thought of as methods for achieving the original goal.
2. By enumerating a number of subgoals *all* of which must be achieved to achieve the original goal.

To formulate a problem using this paradigm, the following tasks are required:

1. Main Goal: Define the main goal.
2. Constraints: Identify any constraints which must be obeyed. These can affect how goals may be achieved.
3. Rules for reducing the problem: Define rules for reducing goals into subgoals. This is generally the hardest part. For their may be various types of subgoals which reduce differently. It must be possible to foresee all possible types of goal so that all can be reduced.
4. Primitive Subgoals: Define the primitive subgoals.

A convenient way to represent this problem formulation is to use an And/Or tree. When used in this context, it may be referred to as a *goal tree*. The root node of a goal tree is the main or top level goal, the branches correspond to subgoals. The *or* nodes are appropriate when any of the subgoals will suffice; the *and* nodes appropriate when all the subgoals need to be achieved to achieve the parent goal. In general, a goal tree will have some sort of constraint(s) associated with it. Finally, there are leaf nodes. These correspond to the primitive subgoals.

The nature of a solution is a rather different than for the state space paradigm. Instead of a simple path which leads from an initial state to a goal state, represented as a sequence of operators, we have a goal tree with *and* nodes and *or* nodes to analyse. A solution is no longer a simple path, rather it is a subtree. A goal tree is said to be solved if it's root node is solved. There are three types of node, each having different criteria for being solved. This is summarised as follows:

- An *And* node is solved when all of its successors are solved.
- An *Or* node is solved when at least one of its successors is solved.
- A *Leaf* node is solved if it obeys all the constraints.

The general case of a goal tree is illustrated in figure 2.7. An example solution subtree for that tree is found in figure 2.8. The nature of the search is also quite different. It consists of traversing the goal tree, keeping track of whether the nodes are *and* nodes or *or* nodes, and also minding the constraints. This is discussed in some detail in section ??. This will become more clear by considering some examples.

2.2.2 Enjoying Your Holiday

Your problem is that Mike wants to enjoy his holiday, but he only has £300 to spend. He has limited himself to two alternatives: to go cross-country skiing in Norway, or to go the Canary Islands and get a suntan. In either case, he will need to consider transport, and accommodation. The transport options may be combinations of car, train, ferry, and plane. Accommodation would be rather cheap in Norway if he stayed in mountain huts, or expensive if he stayed in hotels. Similarly, the Canaries offers a choice between pensions and hotels (unless, of course he happens to have a friend there!)

We will address each of the four tasks enumerated above. The main goal is for Mike to enjoy his holiday The only constraint is the £300 limit on spending. There are two alternate methods for achieving the top level goal: going to Norway,

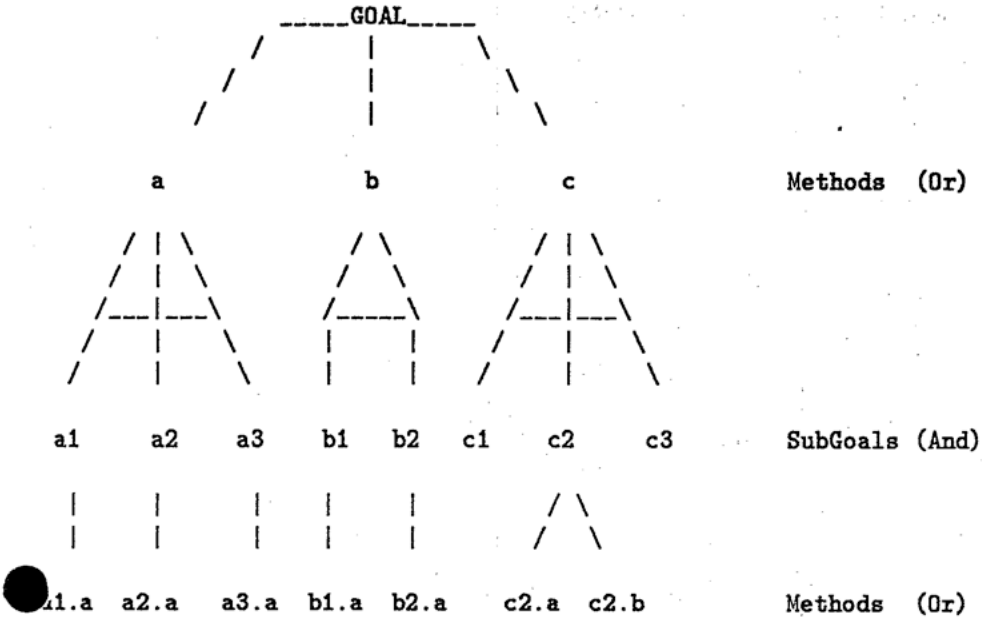


Figure 2.7: A Generic Goal Tree

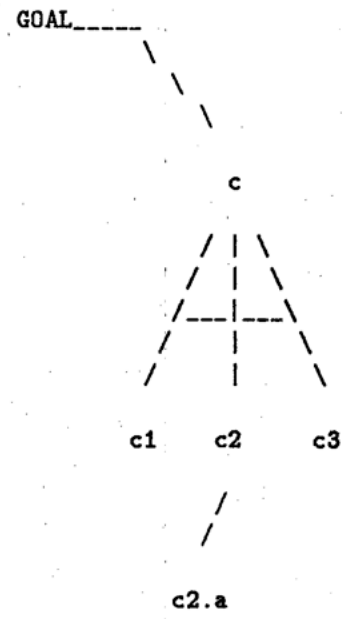
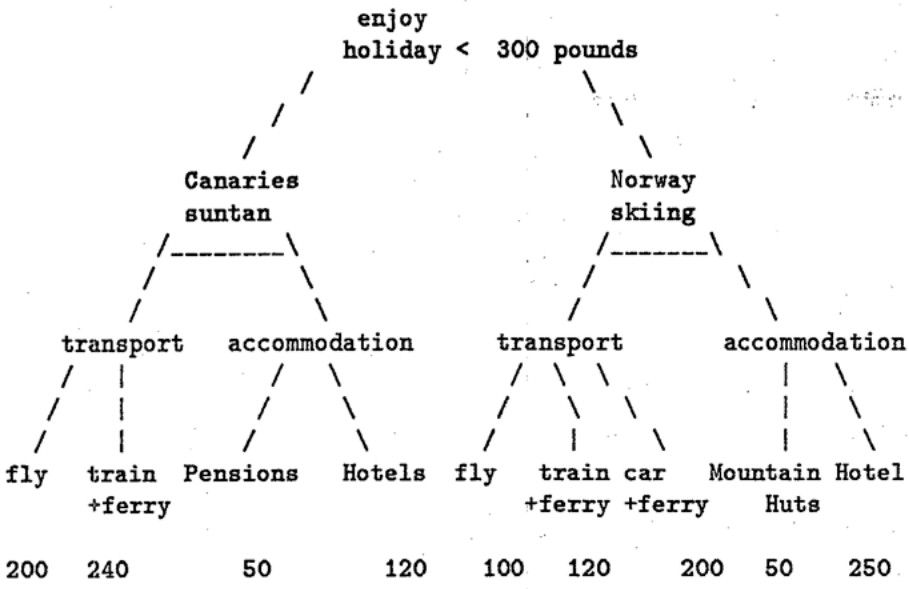


Figure 2.8: A Solution Subtree



Main goal : To enjoy your holiday.

Constraints : You must spend no more than £300

Rules for Reducing Goals into Subgoals : There are no regular rules to follow per se. Each subgoal reduction is mentioned explicitly in the problem description. See goal tree above.

Primitive Subgoals : There are two classes of primitive subgoals corresponding to the two types of decisions that need to be made. The primitive subgoals for transportation are *fly to destination, take train and ferry to destination*, and *Take car and ferry*. For accommodation, the primitives are *Stay in pensions, Stay in Mountain huts*, and *Stay in Hotels.*

Figure 2.9: Goal Tree: How Can Mike Enjoy his Holiday

or the Canaries. Each of these, further reduces into two subgoals: make arrangements for transport and for accommodation. However, the options available if going to the Canaries differ from those available if going to Norway. For a problem such as this one has to make choices as to which actions and/or sugboals are to be taken as primitives. Conceivably, one could further reduce the problem of flying to Norway by worrying about how to get to and from the airports, etc. Similarly, the problem of staying in hotels involves dealing with incompetent clerks, stolen goods, etc. However, since this is not in the problem description, we ignore these details. Instead, we assume that Mike knows how to deal with the vagaries of flying, riding trains, sorting out hassles with hotel clerks etc. These are not perceived as a problems, and thus are considered to be immediately achievable. Therefore, we shall take them to be the primitives for our problem. For simplicity, we ignore other relevant considerations such as food, entertainment, etc. This discussion is summarised in figure 2.9.

Once formulated, however, the problem is still not solved! We have not considered at all how a solution may be found. One thing is sure some, search will be required. There are many possibilities which must be examined. At the risk of repeating ourselves, we continue to emphasise that the issues of representation and control although related, are quite different. We saw this already with regards to the state space paradigm. This problem is easy enough to solve by inspection. One possible solution subtree for the holiday example is shown in figure 2.10. Note that the cost is within the stated constraint. We will take a closer look at control strategies for searching goal trees in section ??.

2.2.3 Counterfeit Coin Problem

This problem (from [Pearl 84] p23) is stated as follows:

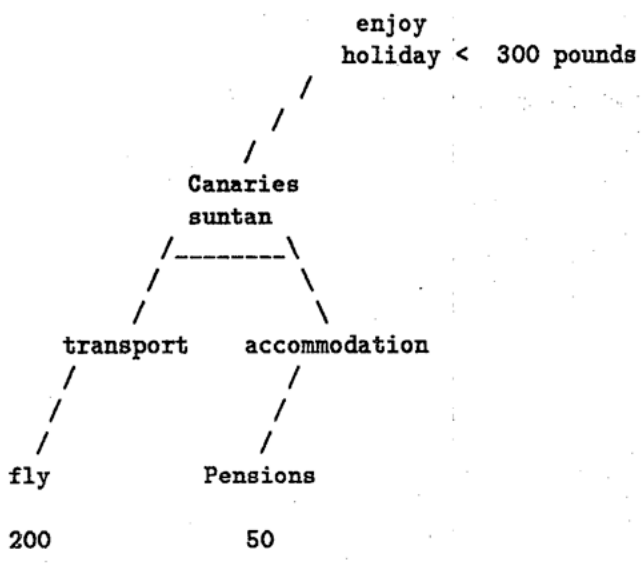


Figure 2.10: An Example Solution Subtree

"We are given 12 coins, one of which is known to be heavier or lighter than the rest. Using a two-pan scale, we must find the counterfeit coin and determine whether it is light or heavy in no more than three tests."

The main problem, or goal is to determine which is the bad coin and whether it is heavy or light. The constraint is that no more than three weighings are allowed. The real effort formulating this in the problem reduction framework is to identify subproblems, and how they come about. There are a number of ways to begin attacking this problem. We could just pick two coins and hope to get lucky by identifying one of the bad coins right away. A safer approach might be to pick six coins, and put three of each side of the scale. If the scale balances, then we know that all six are good and that the bad coin is one of the other six. If not, then the bad coin is one of the six on the scale. In any event we have reduced the original problem from twelve to six coins. We shall refer to this as the 6-coin problem, and of the original as the 12-coin problem. Similarly, if we were to start with weighing four coins (*i.e.* 2 in each pan), then we will have reduced the original problem to either an 8-coin or 4-coin problem depending on whether or not the scale balanced respectively. We now see how to reduce the problem into smaller subproblems.

Our final task is to identify the problem primitives. A problem can be regarded as primitive if it is trivial, or otherwise sufficiently well understood that we are not concerned with the details. In this case, the problem is only trivial if we know how to identify the problem coin, and whether or not it is heavy or light. Eventually we will get to the 2-coin problem which entails being left with two coins one of which is heavier or lighter than the others. Can this be regarded as a primitive, or do we need to go to the 1-coin problem? We proceed by putting one in each pan and noting which is the heavier of the two. For this discussion, assume that the heavier coin (*Hcoin*) was in the left pan, and the lighter one (*Lcoin*) in the right pan. We now know that either *Hcoin* is heavy or *Lcoin* is light and that the other ten are all good. We leave *Hcoin* on the left pan and weigh it against one of the known good coins (*Gcoin*). If the scale tips left again, then *Hcoin* can't be the good coin, since that would imply two coins lighter than it and we know there is only one bad coin. Thus, *Hcoin* is positively identified as the counterfeit coin and it is heavy. If the scale balances, then *Lcoin* is the counterfeit and it's light. The scale couldn't possibly tip right, since that would imply that *Lcoin* is lighter than *Hcoin* which is lighter than *Gcoin* which would mean there were three coins with three distinct weights. But we know there are 11 good coins which must weigh the same amount. This analysis shows that once we reduce the problem to the 2-coin subproblem, there is exactly one more weighing required to identify the coin. We can thus opt to view this as a primitive subproblem.

Figure 2.11 shows part of the goal tree representation of this problem. The root node is an *or* node indicating that the problem can be attacked by any one of the methods. This in not to say that all will work given the three test constraint, but that only one is necessary. The other methods need not be considered. By contrast, the nodes on the second level are *and* nodes. For each of these methods, there are three possible outcomes, *all* of which must be pursued, in order to guarantee that only three weighings are necessary. We cannot assume that we will get lucky. Each of these outcomes in turn correspond to *or* nodes as the top level goal node. The details of the formulation for this problem follow:

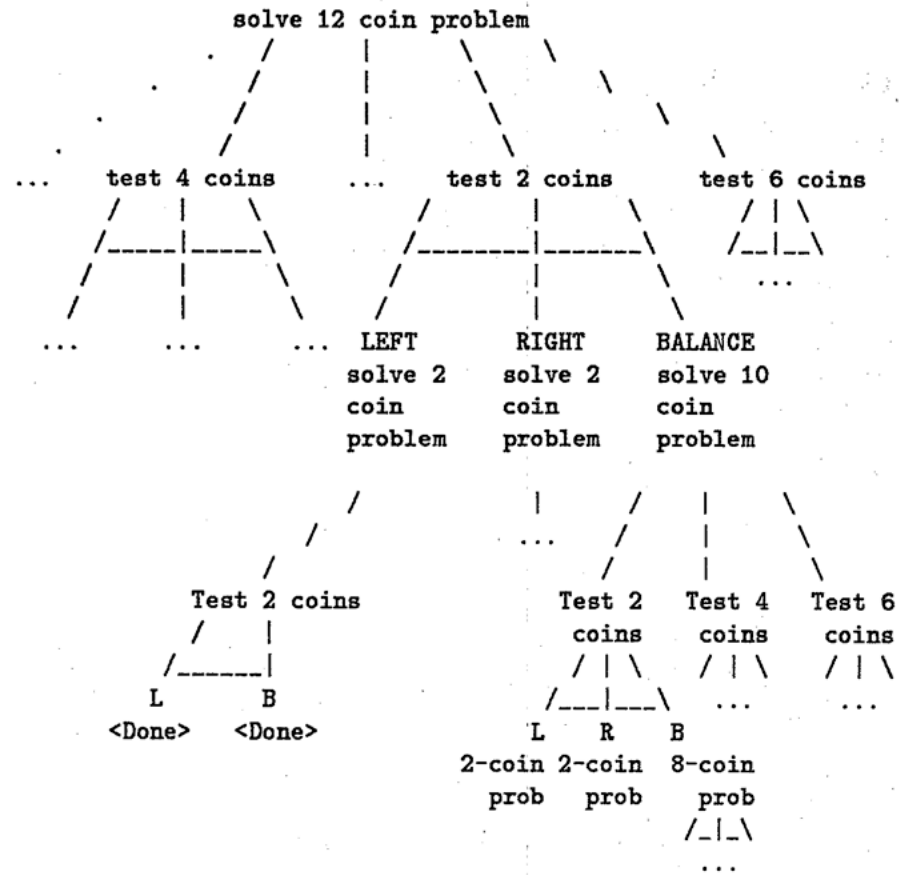


Figure 2.11: Counterfeit Coin Problem

1. **Main goal** : To solve 12-coin problem
2. **Constraints** : You are allowed only three weighings
3. **Rules for Reducing Goals into Subgoals** : This is left as an exercise.
4. **Primitive Subgoals** : We can regard the 2-coin problem as a primitive problem. See text.

Exercise 454: Draw the missing portion of the goal tree in figure 2.11 which corresponds to a solution to the counterfeit coin problem.

Exercise 455: Suppose you wish to help your friend write a computer program to generate the goal tree for the counterfeit coin problem (see figure 2.11). Your job is to describe formal rules for breaking up the counterfeit coin problem up into subproblems. You must make them as specific as possible to make your friend's job easy.

Exercise 456: Write the computer program for your friend.

2.2.4 Towers of Hanoi

We saw this problem in the section 2.1.4, page 25) as an example which could be solved using the state space paradigm. It's easy enough to represent the problem, but hard to see how it might be readily solved, other than some form of brute force enumeration of all possibilities.

This problem can also be formulated as a goal tree using the problem reduction paradigm (see figure 2.12). The key observation to make is that the problem decomposes into simpler problems of the same type. In particular, to move four discs from a to *c*, we need to do three things:

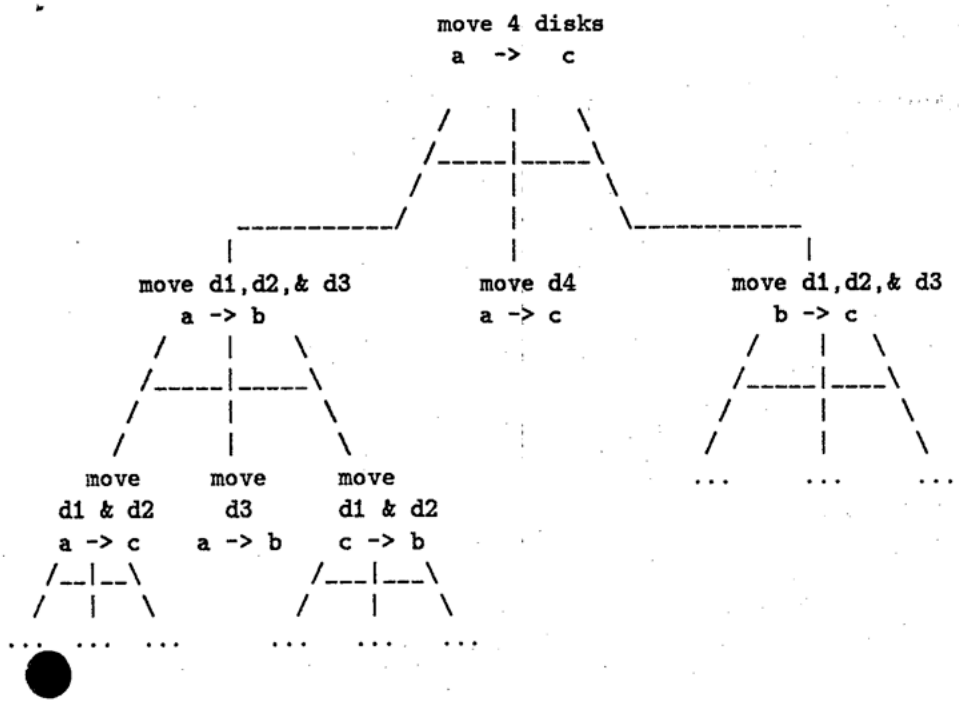


Figure 2.12: Towers of Hanoi: A clever solution

1. move discs 1-3 from a to *b*
2. move disk 4 from a to *c*
3. move discs 1-3 from *b* to *c*

The second step is trivial. The first and third steps are simpler versions of the *exact* same problem (*i.e.* with 3 discs instead of 4). These steps further decompose in the exact same way leaving two instances of the 2-disc problem which is easily solved by moving individual discs. The goal tree is illustrated in figure 2.12.

A solution to the problem is found by reading left to right the moves found on the leaf nodes of the completed version of this tree. This contrasts to the previous examples where a solution consisted of a subtree much smaller than the whole goal tree. Because there are only *and* nodes, the only solution subtree is the whole tree. As soon as the goal tree is created, the solution is found. This means that *there is no search required!!.* All the other examples we have seen, for both the state space and problem reduction paradigms have required search after the problem has been formulated. This is an extreme example of how the correct representation for a problem can make it's solution easier.

The details of the formulation for this problem follow:

Main goal : To solve the 4-disc problem

Constraints : A disc may never be placed on a disc smaller than itself.

Rules for Reducing Goals into Subgoals : The general form of the problem can be completely described using the notation *problem(N,From,To,Scratch).* This means move N discs from tower 'From' to tower 'To' using tower 'Scratch' as a placeholder. Using this terminology, *problem(N,From,To,Scratch)* decomposes into the following three subproblems in the order shown:

1. *problem(N-1,From,Scratch,To)*
2. *problem(1,From,To,Scratch)*
3. *problem(N,Scratch,To,From)*

Primitive Subgoals : Solving the single disc problem is the only primitive subgoal.

EXERCISE - Write a simple recursive PROLOG program to solve the Towers of Hanoi problem for n disks. Beware of trying it out on too large of n, the combinatorial explosion will get you!

2.2.5 Games

We saw an example of a game tree in figure 1.5, page 13. There is a strong similarity with this and the state space representation. In fact, it's virtually identical. We have all four ingredients:

States: These correspond to board positions.
(*e.g.* for noughts and crosses, a state could be a 9 digit number in base 3.

Initial State: The current board position.
(*e.g.* 000,000,000 for the beginning of the game)

Operators: These are the legal moves of the game.

Goal State Recogniser: Corresponds to a winning positions.
(*e.g.* Three in a row, column, or diagonal of the same type)

Indeed it is fair to say that the state space representation accurately describes the playing of a game. In spite of this similarity, we can *not* use the state space paradigm as described in section 2.1 for solving the problem of finding winning game-playing strategies. The reason is that the nature of a solution is entirely different. In that paradigm, a solution corresponds to simply finding a *path* to a goal state, or simply to find a goal state. In games, this is not adequate. Many goal states (winning positions) could be achieved if your opponent made lots of stupid moves. Getting to a goal state is not enough. You have to guarantee that you can get to a winning position no matter what your opponent may do. In game trees, the nodes do in fact correspond to explicit board positions. However, the 'problem' from any board position is not simply to find a path to a winning position (goal state). The goal at a node is to find a winning strategy from that position. The subgoals consist of finding winning strategies for each of the board positions which result from possible moves from some position. An *and* node corresponds to a position from which the opponent may move. This requires checking out *all* the possible moves that the opponent may make, since it is uncertain what the move will be. An *or* node corresponds to a position from which it is *your* turn. You have many possible moves, but you can say beforehand which one you would take from that position. Thus, only one of those moves need be pursued. The problem reduction formulation of the problem of finding a winning strategy for a game is:

Main Goal: To find a winning strategy.

Constraints: Board positions which correspond to wins for the opponent are not allowed.

Rules for Reducing goals into subgoals: These are the move operators, the same as defined in the state space representation. The new subgoals correspond to finding a winning strategy from the new game position.

Primitive Subgoals: These correspond to board positions for which the outcome of the game is decided.

Note that the counterfeit coin problem can be thought of as a game, with *chance* as the opponent. It is infeasible to search entire game trees for complex games such as chess, or even draughts. Various approaches for searching game trees are described in detail in section ??.

2.2.6 Summary

We have described the problem reduction paradigm for solving problems. This is an alternative approach to the state space paradigm. In the latter formulation, we view the world as changing from state to state according to some predefined transformation operators. A solution consists of finding a path from the initial state to some goal state. By contrast, the problem reduction approach works by reducing problems into simpler problems, or equivalently, reducing goals to subgoals. The subgoals continue to be reduced until they can be trivially achieved in which case they are called primitive. An and/or goal tree is a convenient structure for representing the problem. There are usually some constraints associated with the problem. A solution consists of a subtree which satisfies the constraints, and abides by the and/or restrictions on the nodes of the tree. To formulate a problem in this paradigm, you have to:

1. Define the main goal.
2. Identify any constraints.
3. Define rules for reducing goals into subgoals.
4. Define the primitive subgoals.

We have ignored the issue of searching goal trees to find solutions. This is deferred until section ??. We now compare the two problem solving paradigms in some detail.

2.3 State Space versus Problem Reduction

It is important to realise that these are two *approaches* to solving problems, not two types of problems. We do not speak of a problem as being a state space type or a problem reduction type. Instead we speak of choosing an approach to solving a problem. These approaches correspond to alternate points of view. Some problems can be viewed more naturally using a state space representation, others fit the problem reduction framework better. Still other problems quite naturally fit into either framework. That these paradigms are *not* mutually exclusive has been demonstrated by the Towers of Hanoi example. In this case, the problem reduction paradigm was found to be clearly superior to the state space approach This is because in the former, there was no search! In this section, we shall discuss what problem characteristics are relevant in choosing g[redacted] representations for problems. In particular, we discuss criteria for determining which paradigm is likely to be more appropriate for a particular problem. Finally, we briefly examine the formal relationship between the two paradigms.

2.3.1 Criteria and Guidelines for Choosing Good Representations

The purpose of this section is to discuss some general guidelines which will help you decide for a particular problem what the most appropriate representation is. It is important to note that there are two levels of choices to be made when selecting representations. The higher level decision involves determining the general approach to solving the problem, (*e.g.* state space or problem reduction or perhaps another more specialised ad hoc approach). Once this decision has been made, there are still many lower level decisions to make about the actual data structures to use. These, as we have seen, can be extremely important. If the problem fits the state space framework, you must decide how to represent a state, and characterise the operators which allow movement between states. In section 2.1.3 saw how different choices for representing states and operators for the 8-queens problem had considerable impact on the ease with which the problem could be solved. If the problem reduction paradigm is being used, the rules for reducing goals to subgoals must be carefully defined. The primitives must be chosen.

First, we will consider a variety of general characteristics of problems and then briefly discuss some general guidelines which may help in formulating problems.

[redacted]lem Characteristics

Nature Of A Solution—This is perhaps he first thing one should consider when attempting to solve problems. Solutions come in many shapes and flavours. It may be a particular configuration which is desired (*e.g.* 8-queens). In this case, how you stumble on the answer is irrelevant. In other problems, such as the monkey and bananas we already know what the final configuration is. The problem is to find a means of achieving that configuration. We must find a sequence of actions, which are guaranteed to achieve the goal state. We can view this as a *path* from the initial state to the goal state. *Any* path will suffice. In the traveling salesperson problem [3] most paths are of no interest, rather we are looking for the *best* path. Solutions need not be characterised by states or paths betweens states at all. In the counterfeit coin problem, for example, the solution is a *strategy*. That is, a solution to the problem is a somewhat complex recipe or decision procedure for finding the bad coin. No matter how the scale may tip, this procedure must guarantee that no more than three weighings are necessary. The solution to playing noughts and crosses is likewise, a strategy.

[3] This is a famous problem in which a salesperson needs to visit N cities, and wishes to minimise the total distance travelled.

Commitment To Solution Steps?—In the process of solving the problem various steps will be taken. Unless you're very lucky or clever or both, some steps won't seem to lead to a solution. Sometimes these steps make no difference whatever and may be ignored. Theorem proving is an example of this type. If you are attempting to prove something is true from some set of facts and you go down a wrong track, it matters not, you simply continue searching elsewhere for the solution. In the case of the 8-Queens problem, a step which leads to a dead end has to be undone before you can continue the attempt to solve the problem. This is done by backtracking. In the worst case, as in games, steps are irrecoverable. If you make a bad move in a chess game, you have to live with it. The way to deal with this sort of problem is to plan ahead.

Is Problem Decomposable?—Does the solution to the problem seem to be obtainable by solving a number of separate subproblems? We have seen this in several examples including the holiday problem, counterfeit coin etc. It is not always cut and dry. A problem may be *partially* decomposable. For example, consider the problem of removing furniture from a room. For the most part, this decomposes into removing each bit individually. However the painting on the wall behind the piano must be removed after the piano. In this case some of the subproblems interact with each other. Some simple blocks world problems are in a similar category.

The Role Of Knowledge—Problems such as chess, don't require much knowledge except for efficiency. You need only supply the rules of the game and you can play it. Other problems, by their very nature require extensive amounts of knowledge. A newspaper scanner, for example, would be of no use whatever without a sizable store of knowledge,

Is Universe Predictable?—A problem is greatly simplified if you can know with certainty the future effects of your actions (*e.g.* 8-queens). In games, on the other hand, you cannot predict what your opponent will do. The counterfeit coin problem is like a game; the opponent is chance. It is not possible *a priori* to know what the outcome of a weighing will be. Some games are worse than others; bridge, for example, has a large element of chance since you don't know what cards your opponents (or partner) are holding. Similarly for problems involving robot planning, you cannot predict with certainty all the relevant events. A robot on an assembly line may not notice that the conveyor belt has stopped.

The Role Of Time—For some problems, the sequencing of events is crucial. The Towers of Hanoi problem is an such an example. The timing of the actions is crucial. In the 8-queens problem on the other hand, time is not important. The queens can be placed on the board in any order at all.

Selecting a Representation

The two most important questions which must be considered when choosing between the state space and problem reduction formulations are:

1. What is the nature of the solution?
2. How does one most naturally proceed in searching for a solution?

Generally speaking, use state space representation if:

- The solution can be characterised as either a single state, or as a path from an initial state to a final state. A *state* is a partial description of the world, as discussed in section 2.1.
- One can easily define operators which can transform states into other states, thereby enabling the search for the solution to take place.

Use Problem Reduction if:

- The solution itself can be viewed as a strategy of some sort. A strategy is potentially more complex than a path from an initial to a goal state. Typically, it needs to accommodate various possibilities which are unknowable in advance. Consider the solution to the counterfeit coin problem. It is a decision procedure which says what to to whatever the outcomes of individual weighings may be. For the holiday example, the strategy consists of making all the necessary decisions. In the Towers of Hanoi problem, the answer is in form of a simple strategy of when to move which disc from where to where.
- It is easy to reduce the main problem into subproblems which further reduce until finally, the subproblems are trivially solved.

What if the problem seems to fit both frameworks, as in the Towers of Hanoi? There are some further questions to consider which can be of help in deciding, but often it's just a case of using your instincts. Some of these questions are:

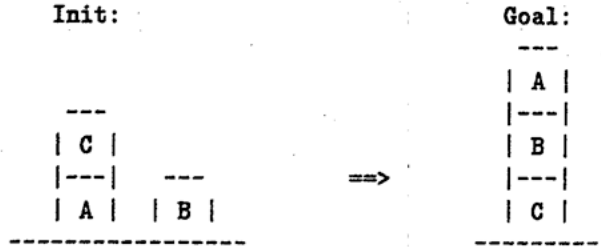
- Which representation seems more 'natural'? Did you have to fight to force the problem into one of the paradigms? If so, it is probably not a good choice.
- Is there interaction between subgoals? Will working on one subgoal affect whether or not another subgoal can be achieved? If so, then we say the problem is only partially decomposable, at best. Problem reduction doesn't work well in this case; it may be wiser to stick with the state space representation if possible. This issue will be discussed at length in chapter ?? where we consider specialised techniques to cope with subgoal interaction.
- How much search is required to solve the problem, once formulated? It is not always easy to quantify an answer to this question, but often, one can get a fairly good idea of how bad things look. We have already seen in the 8-queens problem how different formulations within the single state space paradigm had dramatic effects on the amount of search required. Similarly, the search required to find a solution in the problem reduction paradigm may differ considerably from that required by a state space formulation of the same problem. This was the deciding factor in the Towers of Hanoi problem. In real life, choices are rarely so clear cut.

After an overall framework for problem solving is selected, there are still many details which need to be decided on. It is important to exploit any regularities in the structure of a problem (*e.g.* symmetry). Things which are distinct in fact, may not need to be represented differently. One example of this is in generating moves for noughts and crosses. There are in fact 9 distinct opening moves possible. However, there are only three moves which are important to distinguish (corner, center, side). This choice can significantly reduce the amount of search required. We saw other examples of this phenomenon in the 8-queens , and Towers of Hanoi problems. By noticing certain regularities in the structure of these problems, we were able to reformulate the problems in such a way as to reduce search. Other than this broad hint, there is unfortunately, very little in the way of general theory which is helpful here. The knack for choosing good ways to view problems and defining appropriate data structures etc comes with experience.

Even worse, what if the problem does not seem to fit either paradigm well? Many real world problems are like this. Problems which require vast amounts of knowledge to solve, such as understanding natural language, common sense reasoning, diagnosing illnesses tend to be especially difficult unless the knowledge is well structured and easy to use. Often, however, when asked how they solve problems, human experts find they cannot say why. Just intuition, they say, to the distress of the keen knowledge engineer. These problems require more advanced and/or specialised techniques. Some of these techniques will be discussed in chapter ?? on planning. Other techniques are being developed in other subfields in Artificial Intelligence such Expert Systems, Knowledge Representation, Common Sense Reasoning, etc.

It should be stressed that the issue of representation is a thorny one. Selecting the appropriate point of view for a particular problem is still very much an ill-understood human skill. It will be a long time indeed before computers will be put to the this task.

Exercise 483 Attempt to formalise the following blocks world problem in both the state space and problem reduction paradigms. Comment on which you think is more appropriate.



Exercise 484 Formulate the Monkey and Bananas problem (section 2.1.2, page 18) in the problem reduction paradigm. Comment on the advantages and/or disadvantages of this approach as compared with the state space formulation given in the text.

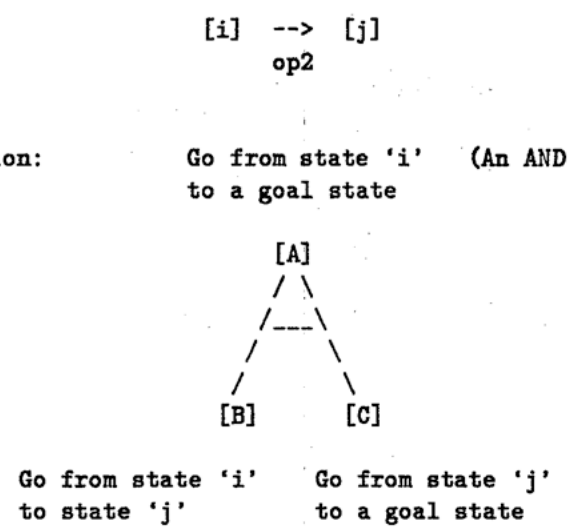


Figure 2.13: State Space to Problem Reduction

2.3.2 State Space to Problem Reduction

If we can solve the same problem using two different approaches, perhaps there is some sort of relationship between these two frameworks. In fact, there is a strong relationship between these two representations. We can convert from one to the other if we follow a certain set of rules. A more detailed discussion of this may be found in [[Barr 81a] pp 74-83, 36-42].

Consider the state space representation for using *op2* to transform state *i* into state *j*. When we are at state *i*, the implicit problem is: "how to get from state *i* to the goal state". In the problem reduction representation, the problem (or goal) is made explicit. In this example, to go from state *i* to some goal state involves the two subgoals:

1. go from node *i* to node *j*
2. go from node *j* to the goal state.

In a state space representation, operators are applied which transform states into new states. In problem reduction, goals reduce to subgoals with no operators being applied, and no new states being explicitly created. When the subgoals reduce no more, we say they are primitive and can be achieved by applying operators directly. In this example, subgoal 1 (node *B*) is a primitive subgoal, and subgoal 2 (node *C* needs to be further reduced. Recall that the primitive subgoals correspond to the primitive operations in the problem domain (see section 2.2.1, page 25). For this abstract example, the primitive operation is *op2*; for the Towers of Hanoi problem, the primitive operation is moving a single disc from one tower to the next.

The graphical representations for these two equivalent formulations is found in figure 2.13. In the state space version, a node is a state of the world, and an arc is an operator. Each node in the problem reduction formulation, corresponds to a subproblem, and each arc corresponds to a decomposition rule. The rules for decomposing problems form the fundamental basis for the problem reduction formulation.

2.3.3 Problem Reduction to State Space

This is considerably more complicated, and I choose not to discuss it in detail. The interested reader is referred to [[Charniak 85] pp 270-281] for an elaborate treatment of this. In doing this conversion, the notion of 'state' will be different from how we have been viewing it. We have viewed a state as some description of the world (*e.g.* the 8-puzzle, monkey and bananas). A broader interpretation is to consider it as a state in the problem solving. Assuming that the steps taken are reasonable ones which will lead to a solution, we can view the state as a *partial solution*. It can contain virtually any sort of information at all (including world state information). Thus, we are not suggesting an alternate view of a state, but rather a more general one. We can reasonably view world states as partial solutions as well, if we also keep track of how

we got there. A partial solution represents the choices made so far, in the attempt to solve the problem, as well as a record of what further choices need to be made. In this more general view, operators correspond to steps taken in attempting to solve the problem. Applying an operator corresponds to making another choice, the state is transformed into one which has this additional choice incorporated into the partial solution. A solution consists of a state where no more choices need to be made.

Very concisely, the conversion is as follows:

1. A state is a partial solution of the goal tree, *i.e.* a subtree which potentially constitutes a solution.
2. The initial state is the root node with no arcs (*i.e.* no choices made).
3. The operators correspond to making choices (*i.e.* adding one more node to the potential solution subtree. The choice to be made is which node of the partial solution to expand. One can add another successor to a node on the subtree which corresponds to an *and* node on the original goal tree. Alternatively, one can add a successor to a leaf node of the partial solution subtree which corresponds to an *or* node on the original goal tree. In the case of *and* nodes, a record must be kept so that all successor nodes eventually are added to the partial solution.
4. A candidate subtree is a solution if it is a solution to the original goal tree according to the rules set forth in section 2.2.1. (*I.e.* if the leaf nodes all correspond to primitive subproblems, the and/or relations are obeyed, and if no other constraints are violated).

2.4 Summary

We have presented two major problem solving paradigms, state space search, and problem reduction. Many problems can readily be formulated in one or the other paradigms. Some fit into both, while still others into neither. We have discussed what sorts of problem characteristics are most suited for each paradigm. We have seen that there are two related issues in formulating problems:

- How to view the problem?
  - What sort of structures can describe the problem?
  - What constitutes a solution?
- Define rules for manipulating the representation (for the purpose of finding a solution).

Often there are many possible ways to look the same problem. In the case of the monkey and bananas, we can view the problem as one of finding a path from the initial state to some goal state. The *path* takes us from different configurations of the problem's 'world' which we call states. From any state, there are a number of actions which may be possible, some of which will lead to a solution and others which will not. Alternatively one could start with an overall problem, or goal of getting the bananas and break it up into smaller subproblems each of which are hopefully easier to solve on their own. In this case, the subproblems would be to have the monkey on the box, and for the box to be under the bananas. Each of these problems further decompose into even smaller problems until the solution is hopefully apparent. We call the former approach *State Space* and the latter *Problem Reduction.*

In some cases,the viewpoint matters little. More typically, however there are great advantages in choosing the 'correct' viewpoint (*e.g.* one may be more 'natural' often resulting in greater efficiency; in extreme cases, as in the Towers of Hanoi, the whole problem may fall apart). The actual choice of representation for a specific problem can be a rather difficult problem in its own right. There may be no obvious way to formalise the description of the problem, or perhaps there are several options, some of which may be better or worse than others. Whatever representation is chosen, you must define rules for manipulating it. If the representation is well chosen then the rules for manipulating it should be fairly obvious. If on the other hand, a poor representation is chosen, it may be very difficult or awkward to define operations which will enable the problem to be solved.

The very nature of a solution may be dramatically different for different representations. We saw this in the noughts and crosses example. In one case, a solution was a huge linear array which explicitly encoded the appropriate move for every conceivable board position. The solution to the problem of finding a winning strategy is to find an appropriate array of moves. Note that this is no easy task. If on the other hand, we systematically search a game tree it becomes much easier to

detect a solution. We will study game trees in detail in section ??. It is of primary importance to be able to characterise precisely what a solution is or you have no hope of finding it.

Often, it is more natural to view a problem in one way than another. It is a good rule of thumb to stick with the representation that seems most natural. A good sign that you have poorly chosen a representation is that it becomes extremely awkward to represent some features of the problem. Often, there are tradeoffs. One choice may make detecting solutions very simple, but the manipulation of the representation may be awkward, or vice versa. There are no clear cut answers.

In section 1.3, we said there were there are two major tasks which were to be required of any one wanting to build intelligent computer programs.

Representation The facts and knowledge about the world which pertain to the problem at hand must be encoded in a form suitable for a computer.

Reasoning Solving problems consists of putting knowledge to use. It consists of

Inference Knowing *how* to derive new information from old.

Control Knowing *when* to make *which* inferences.

It is worth pointing out here, the close analogy between reasoning as discussed in section 1.3.2, and problem solving as discussed here. We said that reasoning consists of inference and control. Here, the 'inference rules' are the rules for manipulating the representations, and the 'control' is the search strategy.

In this chapter, we have discussed in great detail how facts and knowledge about simple problems can be represented. We have also defined ways for manipulating the structures in the representation. For example, in the state space paradigm, the operators define mechanisms for manipulating the states. In the problem reduction paradigm, the rules for reducing goals to subgoals define mechanisms for creating and manipulating a goal tree.

What remains, is the issue of control. Once a representation is chosen, the nature of a solution is characterised and rules are defined for manipulating it, a problem is still not solved. The control issue, in the context of solving simple problems of the sort we have described above, amounts to finding strategies which determine just when an how to manipulate the structures when *searching* for a solution.

2.5 Further Reading

The state space representation I have presented here is encompassed by what Nillson [[Nillson 80]] calls a production system. A *decomposable* production system as discussed in this text encompasses the problem reduction paradigm. A brief discussion of the state space representation is found in [[Barr 81a], pp 32-36]. What Charniak and McDermott [[Charniak 85] pp 255-267] mean by a *search problem* is exactly what we have called the state space problem solving paradigm, although they do not speak of it as such.

Problem reduction and goal trees are discussed in the following places:

ich 83] pp 87-94

rr 81a] pp 74-83, 36-42

arl 84] pp 14-31

iak 85] pp 270-281

Most of the insights regarding the relationship between the state space and problem reduction representations were obtained from chapter one of [[Pearl 84]] (very terse) and chapter five of [[Charniak 85] pp 274-278] (very readable). A much less intuitively appealing discussion of how to convert representations back and forth is found in [[Barr 81a], pp 41-42]. I am indebted to [[Charniak 85]] for pointing out the close link between problem reduction and game trees.

The discussion on problem characteristics is a condensed version of what is found in [[Rich 83] pp 37-48].

2.6 Answers to Selected Exercises

8-Queens 340 For the linear array, it's not immediately obvious how to perform this check. For example, without looking at a numbered diagram, how might you determine whether a queen on square 35 is attacking a queen on square 57. It could be done of course, but not without a bit of a hassle. Hint: Use arithmetic modulo 8.

With each square as an ordered pair of integers: one is the row number and one is the column number, it's fairly obvious how to test for attacking queens. For any two squares (ordered pairs) if the row or column numbers are identical, or if the difference between the row numbers and the difference between the column numbers is identical in absolute value, then queens placed on these two squares would be attacking each other. We have simplified the calculation necessary to check for attacking queens. We say this latter choice of representation is more *natural* for the purposes of checking for attacking queens.

455 There are N possible methods of attacking the 2N-coin problem. They are: test 2*I coins coins by placing I coins in each pan where I = 1, 2, ..., N. There are three possible results of each test, namely that the balance tips left, right, or not at all. If it balances, then the problem reduces to a (2*N-2*I)-coin problem. If not, then it reduces to a 2*I-coin problem.

341 There is no difference, the change is purely a control issue having to do with how to make choices, not in determining what choices are possible. The search space is exactly the same, the savings result from searching it more judiciously by avoiding looking in the wrong place as much as possible.

342 Count the number of free spaces that are removed by the selection of a particular square. So, the choices in the diagram evaluate to 6, 5, and 2 respectively. From this point of view, low scores are desirable. This is really equivalent to measuring the number of free spaces. Only the point of view is different. But, note that it is a different function returning different values which must be compared in different ways.

483 We could create a *move* operator for stacking and unstacking blocks and simply try every move and eventually stumble on the goal state. This would be state space search. On the other hand, we could reduce the goal into two subgoals and work on each independently in hopes of finding the solution. This problem, however is pathologically bad. No matter which goal you achieve first, in order to achieve the other, you have to undo the previous goal! This is called *subgaol interaction* and causes serious problems. When this can be detected, you are better off sticking with the state space representation.

122: This is true because the estimated cost of getting to the nearest goal state from a goal state is 0 (you are already there!)

Answers 763 Backwards search can be summarised as follows. We simply flip the initial and goal states. Then, we try to apply the actions in reverse (*i.e. unapply* them) to see what the state of affairs would have to have been before applying that action to result in the state where you currently 'are'. If the actual initial state is ever reached, then the problem is essentially solved. All we need to do is keep track of the list of actions that we applied backwards and reverse the order.

Monkey and Bananas: Not everything is known about the goal state is known explicitly. The only certain thing is that the assertion: *status(bananas, grabbed)* is in part of the world state description.

8-Queens: Impossible, because you don't know what the goal state is!

8-puzzle: Won't make any real difference because the search space branches equally rapidly from either direction, in general.

Missionaries and Cannibals: Search space is symmetric, as in 8-puzzle, thus no difference is had.

Blocks World: Should work fine. The branching rate will often be less effecting considerable savings.

Bibliography

[Barr 81a] A. Barr and E. Feigenbaum. *The Handbook of AI.* Volume I, HeurisTech Press, 1981.

[Barr 81b] A. Barr and E. Feigenbaum. *The Handbook of AI.* Volume II, HeurisTech Press, 1981.

[Charniak 85] E. Charniak and D. McDermott. *Introduction to Artificial Intelligence.* Addison Wesley, 1985.

[Cohen 81] P. Cohen and E. Feigenbaum. *The Handbook of AI.* Volume III, HeurisTech Press, 1981.

[Newell 81] A. Newell and H. Simon. *Computer Science as Empirical Enquiry*, page ? ?, 1981.

[Nillson 80] N.J. Nillson. *Principles of AI.* Tioga Publishing Company, 1980.

[Pearl 84] J. Pearl. *HEURISTICS - Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley, 1984.

[Rich 83] E. Rich. *Artificial Intelligence.* McGraw-Hill, Inc., 1983.

[Winston 84] P.H. Winston. *Artificial Intelligence.* Addison Wesely, 1984. (2nd Edition).

Contents