CHAPTER 4

Horn Clause Problem-Solving

When logic is used to express problems and problem-solving methods, proof procedures behave as problem-solvers. We shall argue that Horn clause inference subsumes many of the alternative models of problem-solving developed in artificial intelligence.

In this chapter we compare Horn clause inference both with the path-finding model of the Graph Traverser [Doran and Michie 1966] and the General Problem Solver [Newell and Simon 1963] and with the and-or tree model of problem-reduction [Gelernter 1963], [Nilsson 1971]. In the next chapter we compare Horn clause inference with problem-solving regarded as execution of programs. In subsequent chapters we investigate both the use of non-Horn clauses in problem-solving (Chapters 7 and 8) as well as more global problem-solving strategies (Chapter 9).

The close relationship between problem-reduction and top-down inference has been observed by several authors, including [Kowalski and Kuehner 1971], [Loveland and Stickel 1973], [Pople 1973], [Van der Brug and Minker 1975]. Moreover it is already implicit in the Logic Theorist [1963], The General Problem-Solver and the Geometry Theorem Proving Machine [Gelernter 1963].

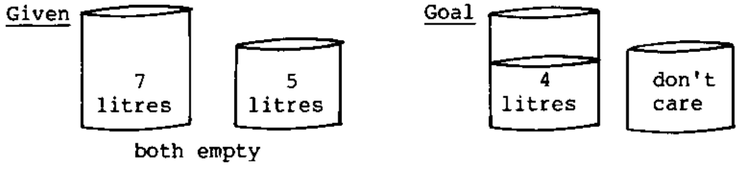
Path-finding

It is possible to express any problem as a path-finding problem.

> Given an initial state A, a goal state Z, and operators which transform one state into another, the problem is to find a path from A to Z.

The water containers problem

The water-containers problem can be formulated naturally as a path-finding problem.



Given both a seven and a five litre container, initially empty, the goal is to find a sequence of actions which leaves four litres of liquid in the seven litre container. There are three kinds of actions which can alter the state of the containers:

(1) A container can be filled.

(2) A container can be emptied.

(3) Liquid can be poured from one container into the other, until the first is empty or the second is full.

The water-containers problem has a simple Horn clause formulation. Interpret

> State(u,v) as expressing that there is a state in which the 7 litre container contains u litres of liquid and the 5 litre container contains v litres.

Assume that the relations

$$x + y = z \quad \text{and} \quad x \leq y$$

are already defined (by infinitely many variable-free assertions, for example).

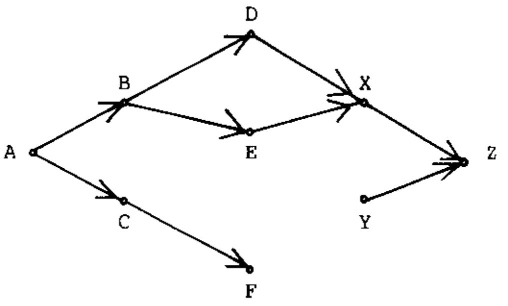
```
WC1        State(Ø,Ø) <-
WC2     <- State(4,y)
WC3        State(7,y) <- State(x,y)
WC4        State(x,5) <- State(x,y)
WC5        State(Ø,y) <- State(x,y)
WC6        State(x,Ø) <- State(x,y)
WC7        State(Ø,y) <- State(u,v), u+v = y, y ≤ 5
WC8        State(x,Ø) <- State(u,v), u+v = x, x ≤ 7
WC9        State(7,y) <- State(u,v), u+v = w, 7+y = w
WC1Ø       State(x,5) <- State(u,v), u+v = w, 5+x = w
```

Clauses WC1 and WC2 express the given and the goal states respectively. WC3 and WC4 define the action of filling a container. WC5 and WC6 define emptying a container. WC7 and WC8 define pouring from one container into another until the first is empty. WC9 and WC1Ø define pouring from one into another until the second is full.

Before investigating the top-down and bottom-up search spaces, it is useful to define the graph-representation of search spaces. First we shall consider a simplified version of the path-finding problem and its Horn clause formulation.

A simplified path-finding problem

Suppose the problem is to find a path from node A to node Z in the following graph.



The problem can be formulated with a one-place predicate

Go(x)

which expresses that it is possible to go to node x. Later in the chapter we shall compare this formulation with the one (suggested by semantic networks) which employs a two-place predicate

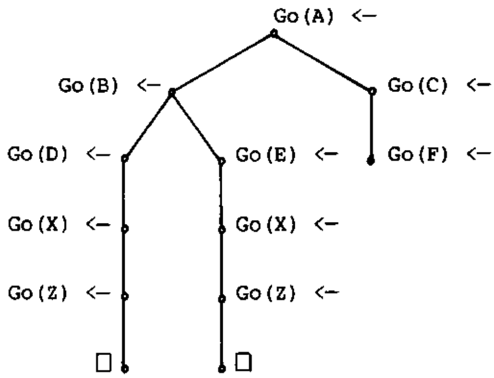
Go* (x,y)

expressing that it is possible to go from node x to node y.

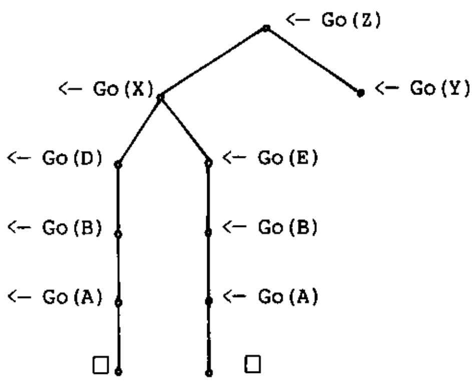
Go(A) <–
Go(B) <– Go(A)
Go(D) <– Go(B)
Go(E) <– Go(B)
Go(Z) <– Go(X)
Go(Z) <– Go(Y)

<– Go(Z)
Go(C) <– Go(A)
Go(F) <– Go(C)
Go(X) <– Go(D)
Go(X) <– Go(E)

In this formulation the clauses which describe the graph behave as path-finding procedures which connect adjacent nodes. The top-down and bottom-up search spaces are both trees.



Bottom-up search space

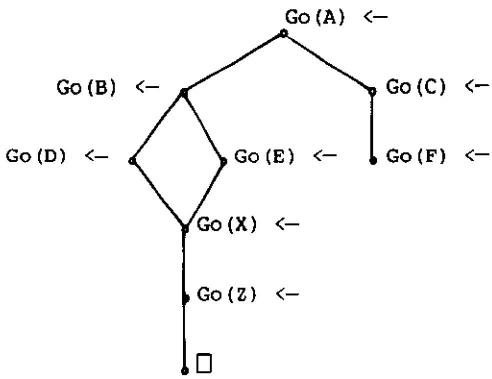


Top-down search space

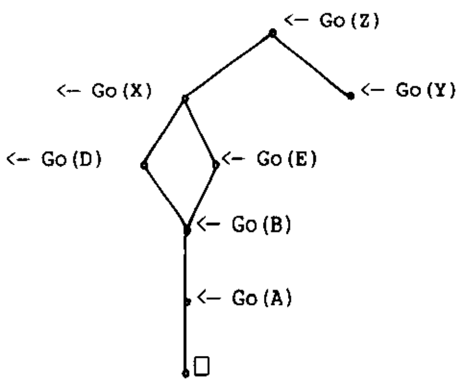
In both search spaces there is a one-to-one correspondence between refutations and solution paths. Both search spaces, however, contain undesirable redundancies. The bottom-up search space derives the assertion Go(X) <- in two different ways and then redundantly uses it twice in the same way to obtain two refutations. The top-down search space derives the goal statement <- Go(B) in two different ways and then redundantly solves it twice in the same way. These redundancies can be eliminated by representing the search spaces as graphs rather than as trees.

Graph-representation of search spaces

The graph-representation of a search space is obtained from the tree-representation by identifying nodes which have the same label. Thus no clause occurs in the graph-representation more than once.



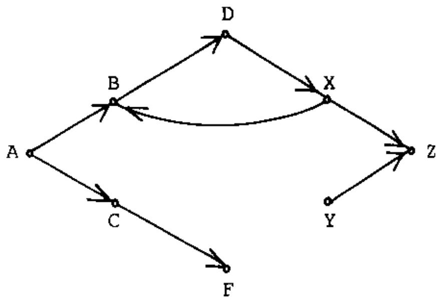
Graph-representation of the bottom-up search space

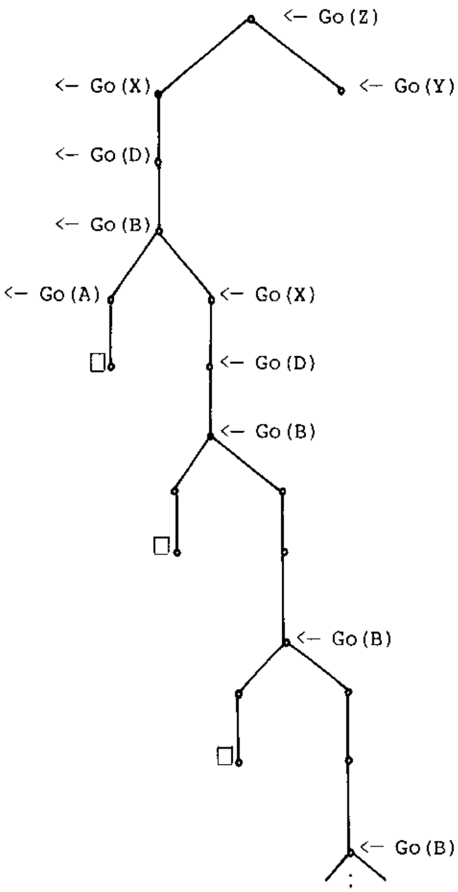


Graph-representation of the top-down search space

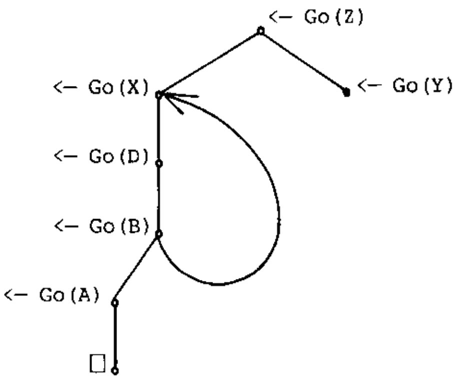
Use of the graph-representation suggests that whenever a search strategy generates a clause in the search space, it checks whether the clause has been generated before. If it has, then only one occurrence of the clause is retained. Generally, the new occurrence is deleted.

The graph-representation can turn an infinite search space into a finite one. The top-down search space for the problem of finding a path from A to Z in the following graph is a simple example.





Infinite top-down search space in the tree representation



Finite top-down search space in the graph-representation

The Search Spaces for the Water Containers Problem

We can now exhibit the graph representations of the search spaces for the water containers problem. In order to avoid complicating the appearance of the search spaces, arcs which lead to nodes labelled by clauses which already occur elsewhere in the search space are not always shown.

The top-down search space is more complicated than the bottom-up search space. Notice, however, that the matching substitutions which are generated in the first step of both branches of the top-down search space determine that if the goal

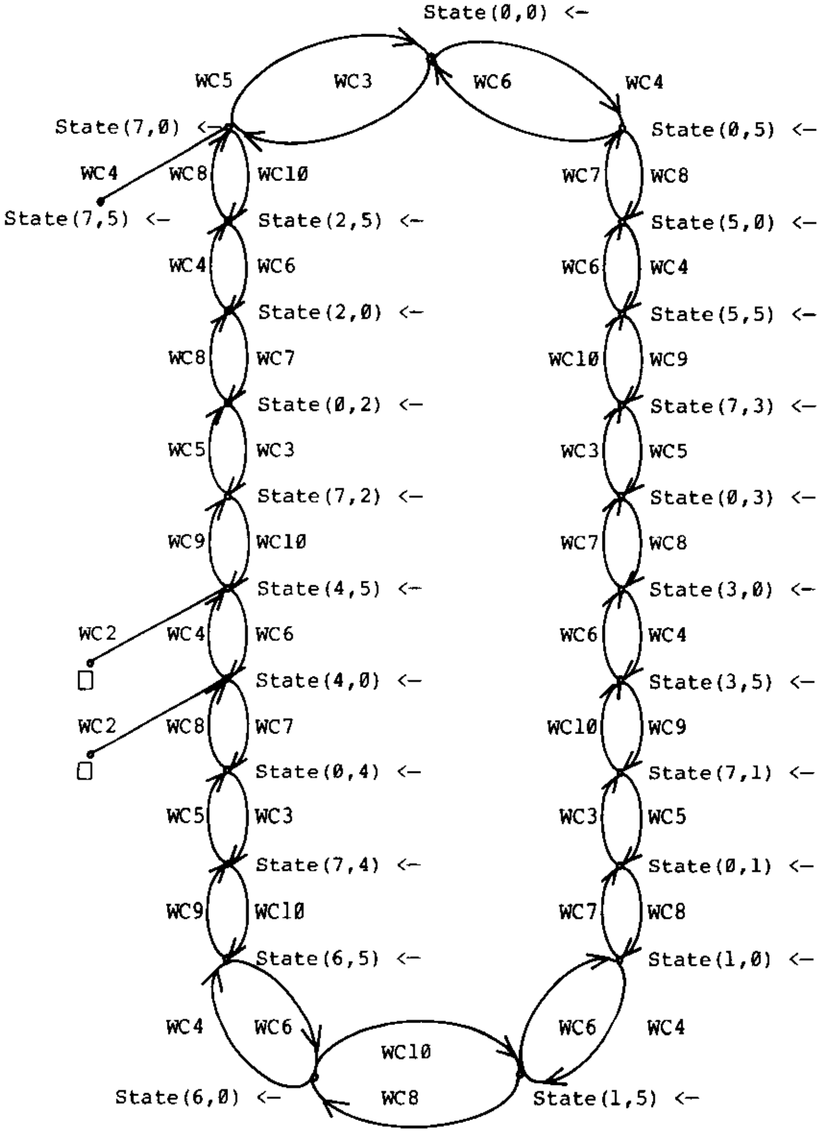
<- State(4,x)

has a solution, then x must be either 0 or 5.

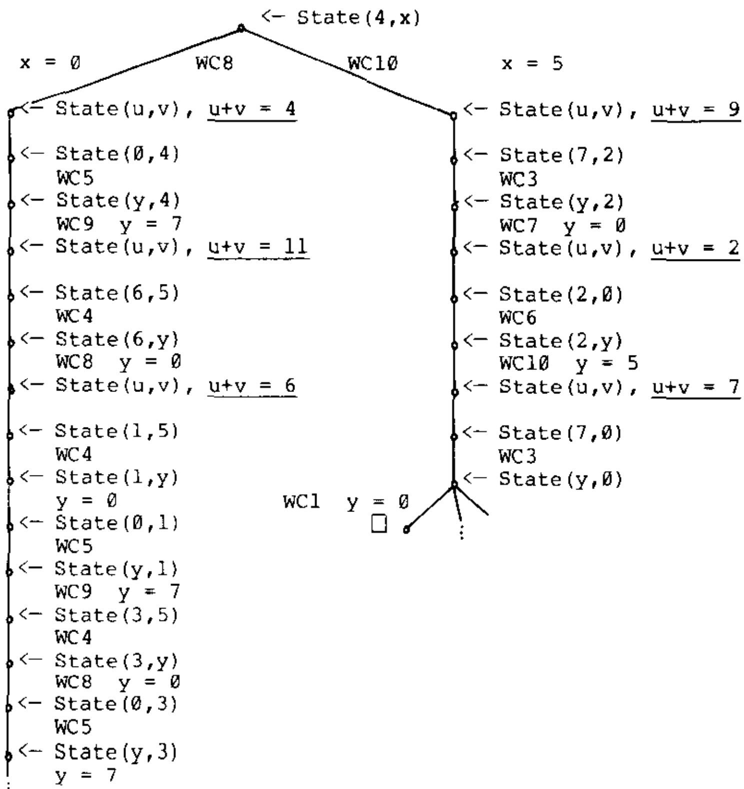
Generally speaking, the conclusions of clauses WC3-10 will not match any goal state which cannot have at least one container either full or empty. For this reason, in the clause

<- State(u,v), u+v = 9

it is easier to select the second goal which generates pairs of integers adding up to 9, and to reject those yielding impossible goal states than it is to solve the subgoals in the other sequence.



<u>Bottom-up search space for the containers problem</u>



Top-down search space for the containers problem

Search strategies for path-finding

The path-finding model of problem-solving is concerned more with the development of search strategies than it is with the structure of search spaces and the representation of information. Given the task of finding a path in a graph, the search problem becomes one of devising intelligent strategies for searching the graph.

Most search strategies for path-finding employ some form of guidance by evaluation functions. Given a search space, an evaluation function f applied to nodes in the space produces real numbers as values. The value f(N) of a node N is intended to measure the usefulness of continuing the search from that node. The greater the value of the node the more promising it is to apply operators to it. heuristic search strategy, guided by the evaluation function, always searches from the node of currently greatest value.

Breadth-first and depth-first search can be regarded as special cases of heuristic search. In depth-first search, the value of a node is its distance from the start node. In breadth-first search, it is the inverse of its distance from the start node. In both cases, the distance between two nodes is measured simply by the number of arcs contained in the currently shortest path connecting the nodes.

In a typical path-finding problem, a node in the search space represents a state of some collection of objects. If there are n objects, a state can be represented by the n-tuple consisting of the individual states of the objects. In the water containers problem, for example, there are two objects which can be in one of the eight states 0-7. Such state-space path-finding problems can easily be represented with Horn clauses by using a predicate

$$State(x_1, x_2, \ldots, x_m)$$

which expresses that the state in which

the 1st individual is in state x_1
the 2nd individual is in state x_2
.
.
.
the mth individual is in state x_m

is possible.

Special evaluation functions are useful for such state-space problems. In the simplest case, given a node

$$N = State(s_1, s_2, \ldots, s_m)$$

(which is either an assertion or a goal, depending on the direction of the search space) and searching for a node

$$T = State(t_1, t_2, \ldots, t_m)$$

the distance between N and T might be estimated by the sum of the distances between the individual states.

$$dist(t_1, s_1) + dist(t_2, s_2) + \ldots + dist(t_m, s_m)$$

The value of a node is greater the smaller its estimated distance to T. More sophisticated evaluation functions might estimate overall distance by a weighted sum of individual distances or by a more complex function of individual distances (such as the square root of the weighted sum of the squares of the distances).

In many path-finding problems, costs are associated with nodes or arcs of the graph and the problem is to find the least costly path connecting the given and goal nodes. In the water-containers problem, for example, it might be required to find the shortest solution. In such cases, the greater the cost of reaching a node the smaller is its value. Both evaluation function guided search strategies [Nilsson 1971] and branch-and-bound [Lawler and Wood 1966] are useful for such problems.

It is not always possible or desirable to use a numerical-valued evaluation function to guide the search strategy. It may be possible, none the less, to define a merit ordering among nodes in the search space. The search strategy, guided by the merit ordering, always searches from a node having the greatest merit.

Since a top-down refutation can be regarded as a path from an initial set of goals to the empty clause, the problem of finding a refutation in a top-down Horn clause search space can be regarded as a path-finding problem and the theory of heuristic search can be applied. However, it must be modified when applied to bottom-up search spaces where solutions are more naturally regarded as trees or graphs [Kowalski 1972]. Even in the case of top-down search spaces the heuristic search path-finding model of problem-solving does not address the important problem of selecting subgoals. These deficiencies are remedied by the problem-reduction model of problem-solving and its associated and-or tree representation.

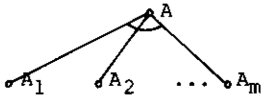
The and-or tree representation of problem-reduction

In the problem-reduction model of problem-solving the task is to find a solution to an initially given problem, using a given collection of assertions and procedures to reduce problems to subproblems. The task is accomplished by repeatedly applying procedures to unsolved problems, replacing them by subproblems, until the initial problem has eventually been replaced by the empty set of subproblems.

In the and-or tree representation of problem-reduction, nodes of the tree are labelled by problems:

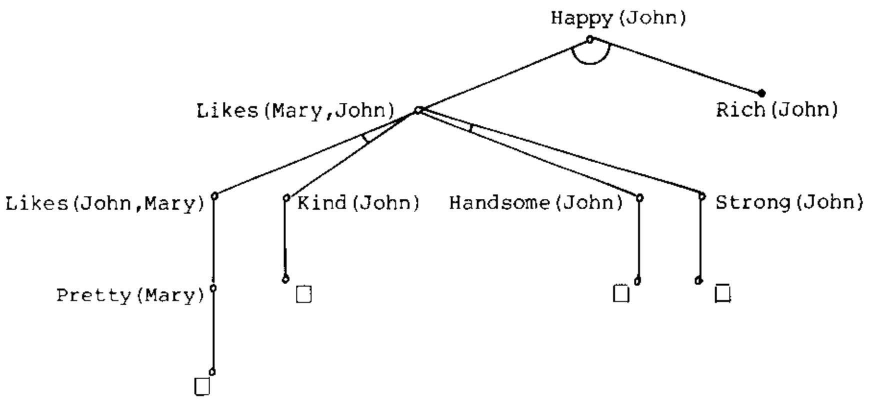
(1) The root node is labelled by the initial problem.

(2) If a problem A labels a node and a procedure reduces A to the subproblems $A_1, A_2, \ldots, A_m$ then the node is connected by a bundle of directed arcs to nodes labelled by the individual subproblems. The bundle itself may be labelled by the procedure.



(3) If the problem A labelling a node matches an assertion, then it is connected by a single arc to a node labelled by the empty collection of subproblems.

The figure below illustrates both the and-or tree representation and the Horn clause representation for a simple problem-reduction task.



<u>Initial Problem</u> <- Happy(John)

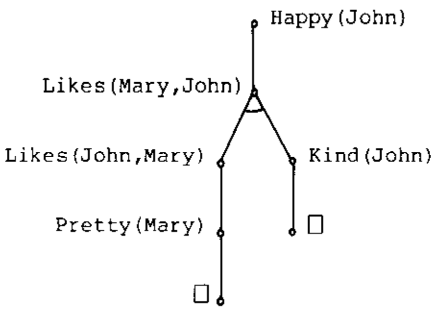
<u>Procedures</u>

Happy(John) <- Rich(John)
Happy(John) <- Likes(Mary,John)
Likes(Mary,John) <- Likes(John,Mary), Kind(John)
Likes(Mary,John) <- Handsome(John), Strong(John)
Likes(John,Mary) <- Pretty(Mary)

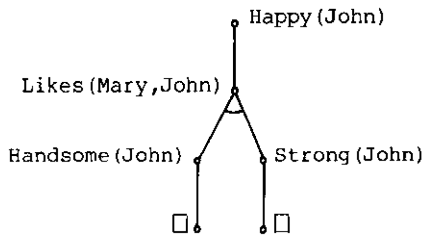
<u>Assertions</u>

Pretty(Mary) <-
Kind(John) <-
Handsome(John) <-
Strong(John) <-

The problem has two solutions which can be represented as subtrees of the and-or tree:

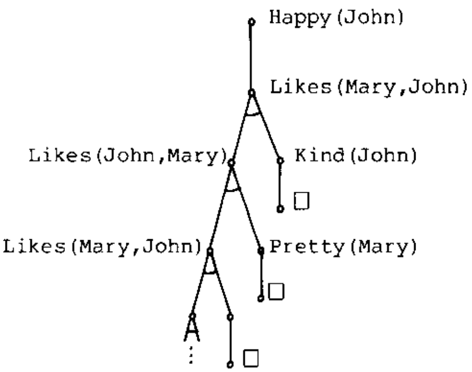


<u>one solution</u>

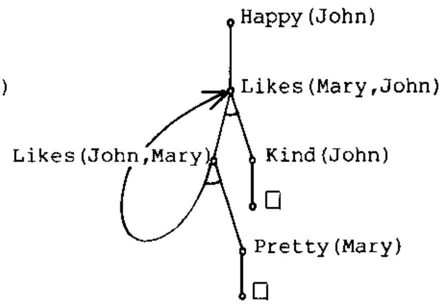


<u>the other solution</u>

The <u>and-or graph representation</u> is obtained from the and-or tree representation by identifying all nodes which are labelled by the same subproblem. In the example below, the and-or graph representation turns an infinite and-or tree search space into a finite one. The problem has no solution.



<u>and-or tree representation</u>



<u>and-or graph representation</u>

<u>Initial Problem</u>	<— Happy(John)
<u>Procedures</u>	Happy(John) <— Likes(Mary,John)
	Likes(Mary,John) <— Likes(John,Mary), Kind(John)
	Likes(John,Mary) <— Likes(Mary,John), Pretty(Mary)
<u>Assertions</u>	Pretty(Mary) <—
	Kind(John) <—

Both the and-or tree and and-or graph representations of problem-reduction focus attention on the structure of the search space and on search strategies. However, they ignore both the structure of the problems which label the nodes of the search space and the connection between problems in the form of shared variables. The Horn clause model of problem-reduction represents problems by atomic formulae and makes explicit (in the form of matching substitutions) the information which is generated when a procedure or assertion is applied to a problem.

The problem-solving interpretation of Horn clauses

The problem-solving interpretation of Horn clauses is basically the top-down interpretation.

The atoms in a denial $\leftarrow A_1,\ldots,A_m$ are interpreted as problems, or goals, to be solved. If the denial contains the variables $x_1,\ldots,x_k$ then it is interpreted as stating the goal:

> Find $x_1,\ldots,x_k$
> which solve the problems $A_1,\ldots,A_m$.

and is called a goal statement.

An implication $A \leftarrow A_1,\ldots,A_m$ is interpreted as a problem-solving method, or procedure:

> To solve a problem of the form A,
> solve the subproblems $A_1,\ldots,A_m$.

Given a problem B which matches A, the procedure reduces the solution of B to the solution of the subproblems

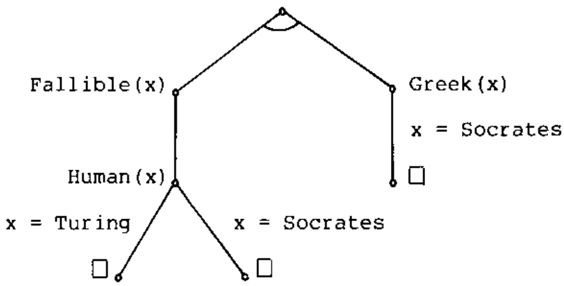
$$A_1\theta,\ldots,A_m\theta$$

where θ is the matching substitution. We say both that the procedure matches A and that it applies to A.

An assertion $A \leftarrow$ is interpreted as a procedure which solves problems directly without reducing them to further subproblems.

The empty clause $\square$ is interpreted as the empty goal statement.

The and-or tree and and-or graph representations can be extended to Horn clause problem-reduction in general. It is necessary to represent the contribution of a procedure to the values of the variables in the problem to which the procedure is applied. In the extended and-or tree representation, each bundle of arcs is labelled by that part of the matching substitution (called the output component) which affects variables in the problem under consideration. The figure below illustrates the extended and-or tree representation for the fallible Greek problem of Chapter 1.



In general, the substitution Θ which matches a problem B with a procedure $A \leftarrow A_1,\ldots,A_m$ can be decomposed into two parts $\Theta = \Theta_i \cup \Theta_\emptyset$.

(1) One part Θ_i affects variables in the procedure. It passes <u>input</u> from the problem to be solved to the procedure which tries to solve it. Θ_i is called the <u>input component</u> of the matching substitution.

(2) The other part $\Theta_\emptyset$ affects variables in the problem to be solved. It passes <u>output</u> from the procedure to the problem whose solution is being attempted. $\Theta_\emptyset$ is called the <u>output component</u> of the matching substitution.

Thus the procedure reduces the problem B to the collection of subproblems

$$A_1\Theta_i,\ldots,A_m\Theta_i$$

whereas the output component $\Theta_\emptyset$ is the procedure's contribution to finding the values of the variables in B.

When the matching substitution makes a variable, say x, in the problem identical to a variable, say y, in the procedure, then it is useful to treat the substitution as transmitting input and to include y = x in the input component of the matching substitution.

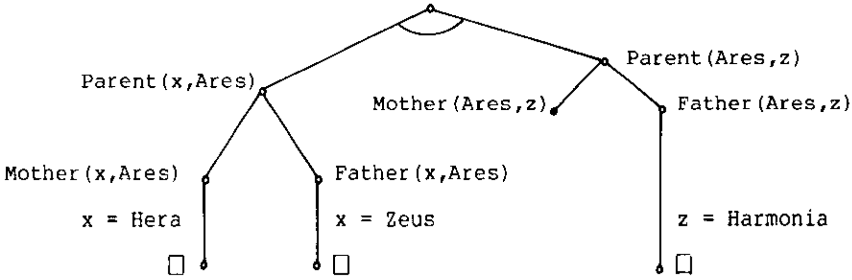
<u>Splitting and independent subgoals</u>

An important characteristic of the and-or tree representation is that it explicitly exhibits the <u>splitting</u> of a goal statement into <u>separate</u> subgoals. Splitting is especially useful when the subgoals share no variables. Subgoals which share no variables are <u>independent</u> and can be solved by different problem-solvers working independently.

In the family relationships example the two subgoals in the initial goal statement

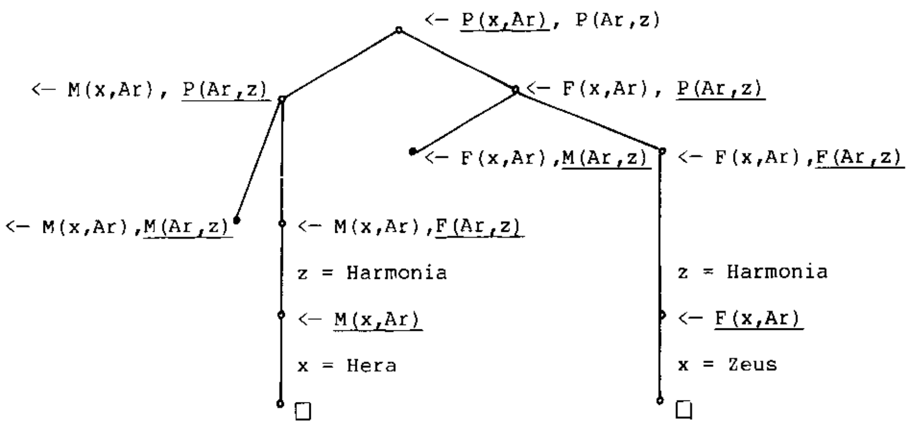
<- Parent(x,Ares), Parent(Ares,z)

share no variables and are independent.



Any solution to the problem of finding an x which is a parent of Ares is compatible with any solution to the problem of finding a z which is a child of Ares. Problem-solvers could work on the separate problems simultaneously without danger of interfering with one another.

Top-down search spaces whose nodes are labelled by goal statements contain redundancies when subgoals are independent. This is illustrated by the goal statement search space for the previous problem. The same abbreviations are used as in the previous chapter.



Here the subgoal of finding a child of Ares is redundantly considered twice, once in the context of the goal statement <- M(x,Ar), <u>P(Ar,z)</u> and again in the context of the goal statement <- F(x,Ar), <u>P(Ar,z)</u>. In the and-or tree search space the subgoal is represented only once.

More generally, given an initial goal statement <- A, B, n ways of solving A and m ways of solving B, the goal statement top-down search space contains n*m branches, whereas the and-or tree contains only n+m.

Dependent subgoals

The extended and-or tree representation does not specify the relationship between the solution of a goal statement and the solution of its separate subgoals. In particular, the problem-solving interpretation leaves open the possibility that a goal statement

<- $A_1,\ldots,A_m$

might be solved by

(1) independently solving the separate subgoals, obtaining associated substitutions $\theta_1,\ldots,\theta_m$ which solve the subgoals and then

(2) combining the separate substitutions to obtain a solution of the goal statement itself.

If the subgoals are independent then it suffices to combine the separate substitutions by taking their union. If they are dependent then it is necessary to combine them by finding a most general common instance of the substitutions. For example, the combined substitution for the independent subgoals in the goal statement

<- Parent(x,Ares), Parent(Ares,z)

is simply the union

{x = Hera, z = Harmonia}

of the individual substitutions. But the combined substitution for the dependent subgoals

<- Ø < y, Even(y)

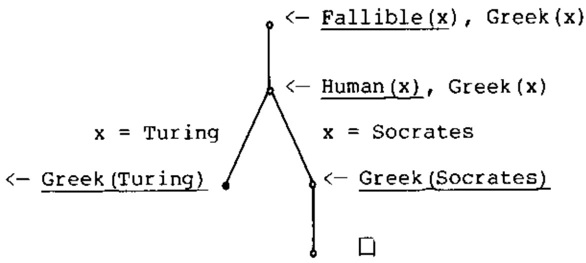
given the separate substitutions

{y = s(y')} and {y = s(s(Ø))},

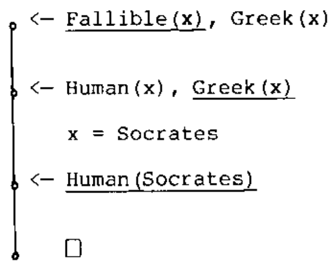
is obtained by matching the two values for y giving

{y = s(s(Ø))}.

Top-down goal-statement search spaces make explicit both the dependencies among sub-goals and the effect on the size of the search space of solving different subgoals in different sequences. The and-or tree search space for the problem of the fallible Greek, for example, is independent of the order in which the top level goals are solved. The goal statement search spaces, however, are quite different. Solving goals in one sequence we obtain a search space containing alternative branches, whereas solving them in a different sequence generates a search space consisting only of the solution itself. Notice that, as in the extended and-or tree representation, it is useful to label arcs by the output component of the matching substitution.



<u>One top-down search space</u>



<u>Another search space</u>

For the remainder of the book we shall use goal statement search spaces (in preference to extended and-or tree spaces), because they make it easier to show the effect of the subgoal selection strategy on the size of the search space. In practice, computer implementations of Horn clause problem-solving systems use a representation which combines features of both and-or tree and goal-statement spaces.

The goal statement search spaces for the fallible Greek problem illustrate a general principle. When subgoals are dependent, <u>select one to which the fewest procedures apply</u>. The aim is to minimise the overall size of the search space by locally minimising the number of alternative branches which emanate from any node.

<u>Finding versus showing</u>

Logic does not distinguish between procedures which <u>show</u> that a given relationship holds and procedures which <u>find</u> individuals for which it holds. Thus the grandparent procedure, for example, is able not only to show that one individual is grandparent of another but also to find both grandparents and grandchildren.

The difference between showing and finding is indicated by the presence or absence of variables. In general, the more variables a problem contains, the more finding there is to be done.

Any procedure which applies to a showing problem P(t) also applies to the corresponding finding problem P(x). Thus the search space for a finding problem is generally larger than it is for a showing problem. This suggests the principle of _selecting a subgoal which involves least finding and most showing_. This principle is subsumed by the one which selects the subgoal to which fewest procedures apply, but it is easier to apply. It requires only an analysis of the subgoals under consideration rather than an analysis of all the matching procedures as well.

Applying these principles to the grandparent procedure

Grandparent(x,y) <— Parent(x,z), Parent(z,y)

results in the selection of different subgoals depending on the form of the problem to be solved:

(1) Given x, to find grandchildren y of x, first find children z of x, then find children y of z.

(2) Given y, to find grandparents x of y, first find parents z of y, then find parents x of z.

(3) Given both x and y, to show x is grandparent of y, compare the number n of children of x with the number m (two) of parents of y.
If $n < m$, first find children z of x then show they are parents of y.
If $n > m$, first find parents z of y and then show they are children of x.
If $n = m$, it doesn't matter which of the two subgoals is selected first.

(4) Given neither x nor y, to find individuals in the grandparent relationship, it doesn't matter which subgoal is selected first.

The principle of preference for subgoals to which fewest procedures apply has two aspects. On one hand, it is a _principle of procrastination_, which delays as long as possible the selection of explosive subgoals that can be solved in many ways. On the other hand, it is a _principle of eager consideration_ of subgoals which can be solved in few ways.

The principle of procrastination can lead to smaller searches in two ways. When subgoals share variables, delaying the selection of a finding problem (which can be solved in many ways) can turn it into a more manageable showing problem which can be solved in fewer ways. Finding the values of variables may be done more efficiently by selecting other, less explosive, dependent subgoals. Whether subgoals are dependent or not, it may be possible to postpone the consideration of explosive subproblems until after the initial problem has been solved by alternative methods. By then, whether or not the explosive subproblem has been instantiated it can be ignored.

The principle of eager consideration is of particular utility when a subgoal can be solved in at most one way. To solve a goal statement, all its subgoals have to be solved. Therefore, if a goal statement contains an unsolvable subgoal, which matches no procedure, then the selection and recognition of the unsolvable subgoal demonstrates the unsolvability of the goal statement as a whole; hence we avoid the unnecessary consideration of other subgoals in the same goal statement. When only a single procedure matches a given subgoal, then it must be applied sooner or later, if the goal statement has a solution. Early consideration has the advantage that any information in the form of values for variables can be obtained as soon as possible and communicated to other dependent subgoals. Moreover, if the procedure eventually fails to solve the subgoal, then consideration of other more explosive subgoals in the same goal statement may be avoided.

The number of procedures (including assertions) which apply to a given subgoal is only a local approximation to the total number of ways the subgoal can be solved. It can be misleading in some cases. Better approximations can be obtained by employing look-ahead techniques similar to the mini-max methods discussed later in this chapter.

The effect of different strategies for selecting subgoals on the size of the search space is more pronounced when composite terms, constructed by means of function symbols, are involved. The effect of composite terms on the selection of subgoals will be investigated in the next chapter.

Lemmas, duplicate subgoals and loops

Many features of the extended and-or graph representation can be incorporated into the top-down goal statement representation by generating lemmas which record the solution of solved subgoals. When a subgoal is solved, an assertion can be generated which solves the subgoal directly in one step. Such assertions are lemmas, which are found by top-down deduction but could have been generated bottom-up. Thus a lemma which has been generated when a subgoal is solved in the context of one goal statement can be used to solve the same subgoal directly when it arises again in the context of another goal statement.

To achieve the problem-solving power of and-or graphs, negative lemmas also need to be generated when a subgoal is recognised as unsolvable. Negative lemmas can be used to recognise that the same subgoal is unsolvable when it arises again in another context.

The generation of positive lemmas was first described by Loveland [1969] for the top-down model-elimination proof procedure. Both positive and negative lemma generation are incorporated into the top-down parsing procedure for context-free grammars devised by Earley [1970]. An equivalent of lemma generation in Horn clause problem-solving has been proposed by Warren [unpublished] as an extension of the Earley parsing procedure.

The simple case, where duplicate subgoals occur in the same goal statement, can be dealt with directly - simply by deleting all but one of the duplicate occurrences. Such merging of duplicate atoms in the same clause is a special case of the factoring rule described in Chapter 7.

It is also a special case of the rule for deleting redundant subgoals, described in Chapter 9.

Perhaps the most important case of duplicate subgoals arises when a goal occurs as its own subgoal. This is one of the situations that leads to loops and to infinite search spaces. Given a goal B and a matching procedure

$$A \leftarrow A_1, A_2, \ldots, A_m$$

each of the goals $A_1\theta, A_2\theta, \ldots, A_m\theta$ where θ is the matching substitution is a <u>subgoal</u> of B. Moreover, any subgoal of a subgoal of B is also a <u>subgoal</u> of B. Thus one goal is subgoal of another if they both occur on the same branch of the and-or tree search space.

Loop detection procedures, which test whether a goal occurs as its own subgoal, are a feature of Loveland's model elimination procedure and of Sl-resolution. More general loop detection strategies, which test whether a goal subsumes a subgoal, have been investigated by Derek Brough [1979] and have been incorporated into a Horn clause problem-solving system implemented at Imperial College.

Search strategies for problem-reduction spaces

Search strategies for and-or trees and graphs are extensions of those for path-finding. They differ primarily because they combine the evaluation of procedures with the selection of subgoals.

The mini-max and alpha-beta strategies [see Nilsson 71] are commonly employed when and-or trees represent game playing problems. Individual subgoals represent states of the game. Alternative procedures which apply to a given subgoal represent the problem-solver's alternative moves for the state represented by the subgoal. The bundle of subgoals which results from the application of a procedure represents the states associated with all the opponent's alternative responses to the problem solver's move.

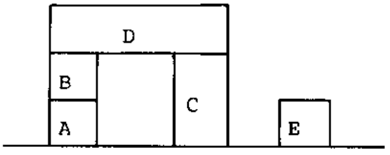
The value of a move (represented by a procedure) for the problem-solver is only as great as the opponent's strongest response. Thus the value of applying a procedure is the minimum of the values of the subgoals in the bundle associated with the procedure. The value of an individual state of the game (represented by a subgoal) on the other hand, is as great as the problem-solver's best move. Hence the value of a subgoal is the maximum of the values of the procedures which apply to the subgoal.

Given an initial evaluation of subgoals, mini-max evaluation looks ahead into the search space and provides a revised, more accurate evaluation of subgoals. It can be used not only for game playing but for problem-reduction in general. An appropriately modified version of mini-max evaluation can be used specifically to improve the criterion for selecting subgoals. A general method for using 'look-ahead' to improve evaluation functions for clausal theorem-proving has been developed for the connection graph proof procedure [Kowalski 1974a] presented in Chapter 8.

For many problem-reduction applications it is more appropriate to use some form of depth-first search. This is efficient to implement because only one branch of the top-down search space is considered at any time. When no untried procedure applies to the selected subgoal in the goal statement at the end of the branch, the search strategy backtracks to the next-to-last node of the branch and tries to solve the selected subgoal there in an alternative way. For this reason depth-first search is also called backtracking.

Although backtracking is effective in many cases it can be distressingly unintelligent in others. Both successful and unsuccessful applications of backtracking are illustrated by the arch recognition problem.

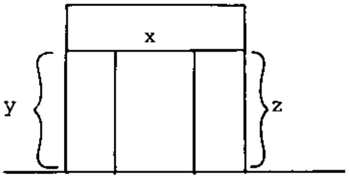
Consider, for example, the problem of recognising an arch in the following scene:



It is convenient to name an arch by means of a function symbol which collects together the immediate constituents of the arch. We let the term

a(y,x,z)

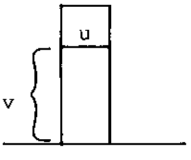
name the arch



which consists of block x on top of left tower y and right tower z. A tower can be named by using a function symbol which combines the block on top of the tower with the subtower beneath it. We let the term

t(u,v)

name the tower



which consists of block u on top of tower v. Thus t(B,A) names the tower comprising block B on top of block A; a(t(B,A),D,C) names the arch in the scene above. The scene and the definitions of arch and tower are represented by clauses A1-12.

```
A1      Arch(a(y,x,z)) <- Block(x), Tower(y),
                          Tower(z), On(x,y), On(x,z)

A2      Tower(x)       <- Block(x)
A3      Tower(t(x,y))  <- Block(x), Tower(y), On(x,y)

A4      On(x, t(y,z))  <- On(x,y)

A5      Block(A) <-
A6      Block(B) <-
A7      Block(C) <-
A8      Block(D) <-
A9      Block(E) <-

A10     On(B,A) <-
A11     On(D,B) <-
A12     On(D,C) <-
```

Clause A4 reduces the problem of determining whether a block is on a tower to that of determining whether the block is on the block which is on top of the tower.

The definition of arch A1 is unsatisfactory for several reasons (see exercise 5). The problems which arise with backtracking, however, are independent of them.

Consider the problem

<- Arch(a(t(B,A), D, C))

of recognising the arch in which block D is both on the tower B on A and on the tower C. Using A1 and solving subproblems in any sequence, the top-down search space consists of just the single path which solves the problem. No search strategy, including backtracking, behaves unintelligently.

Suppose, however, that the problem is to find an arch in the scene

<- Arch(w).

Assume that subproblems are selected and procedures are applied in the order in which they are written. Because such strategies are especially easy to implement, they are incorporated in many computer-based problem-solving systems. The initial problem quickly reduces to an unsolvable goal statement.

```
                    <- Arch(w)
          A1
w = a(y,x,z)
                    <- Block(x), Tower(y), Tower(z), On(x,y), On(x,z)
       x = A
          A5
                    <- Tower(y), Tower(z), On(A,y), On(A,z)
          A2
                    <- Block(y), Tower(z), On(A,y), On(A,z)
          A5
       y = A
                    <- Tower(z), On(A,A), On(A,z)
          A2
                    <- Block(z), On(A,A), On(A,z)
          A5
       z = A
                    <- On(A,A), On(A,A)

                       unsolvable
```

The simple depth-first strategy backtracks to the previous node and searches for another block z. But changing z does not affect the unsolvability of On(x,y) so long as x and y are both A. The backtracker goes into an infinite loop, trying a potentially infinite sequence of towers z which do not affect the unsolvability of the subproblem On(x,y), where x and y are A.

Backtracking can be made more intelligent if, when generating an unsolvable subgoal, it analyses the substitutions which cause the failure (in this case x=A and y=A), and backtracks to a node where it can undo them (in this case to the goal statement containing the selected subgoal Block(y)). Efficiency can be improved by preserving intermediate solved subgoals. The backtracker can be made more intelligent still by analysing the failure, not only to identify the subgoal whose solution should be undone, but also to determine how it should be done [Schmidt et al 1978]. In this example, when the subgoal On(x,y) with x=A and y=A is recognised as unsolvable, the assertion On(B,A) <- can be identified as the nearest match. The search strategy can then backtrack to the goal statement containing the selected subgoal Block(x) with substitution x=A and test whether Block(x) with x=B can be solved. Such goal-directed intelligent backtracking has the spirit of Sussman's [1975] model of problem-solving. Instead of carefully evaluating subgoals and alternative procedures, the problem-solver picks them arbitrarily. If they fail, he analyses the mistake in order to find a better method of solution.

Notice, however, that the effect of solving subgoals in an arbitrary sequence and backtracking intelligently when things go wrong can be achieved more directly by selecting the correct subgoals in the first place. In this example, it suffices to select the subgoals

On(x,y) and On(x,z)

before the others in the definition A1 of the arch. Similarly, the subgoal

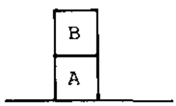
On(x,y)

should be selected first in the definition A3 of tower. It is necessary, moreover, to try the assertions A10-12, which define the location of blocks resting on blocks, before the procedure A4, which defines the location of blocks on towers.

```
                 <- Arch(w)
w = a(y,x,z)
                 <- Block(x), Tower(y), Tower(z), On(x,y), On(x,z)
       x = B
       y = A
                 <- Block(B), Tower(A), Tower(z), On(B,z)
       z = A
                 <- Block(B), Tower(A), Tower(A)

                 <- Tower(A)

                 <- Block(A)

                    □
```

Here the duplicate subgoal Tower(A) has been deleted to avoid redundancy. Notice that the first solution finds the pathological arch:

a(A, B, A)



Backtracking is employed in both the PLANNER [Hewitt 1969] programming language and the PROLOG [Colmerauer et al 1972] [Roussel 1975] top-down, Horn clause programming system. The inefficiencies of backtracking in PLANNER led to the development of CONNIVER [Sussman and McDermott 1972a, 1972b], a PLANNER-like programming language in which the programmer writes both problem-solving procedures and search strategies. In PROLOG, the problem-solver provides the backtracking search strategy but the programmer can control the extent of backtracking.

Various problem-solvers incorporating intelligent backtracking have been designed and implemented by Sussman and his colleagues [Sussman 1975], [Stallman and Sussman 1977], [Doyle 1978]. Intelligent Horn clause backtracking problem-solvers have also been investigated by Cox and Pietrzykowski [1976], [Cox 1978] and by Bruynooghe [1978]. Limited intelligent backtracking strategies have also been implemented in various Horn clause systems at Imperial College.

Bi-directional problem-solving

The Horn clauses which describe a typical problem-solving task can be classified into three kinds:

(1) general-purpose procedures (including assertions), which describe the problem-domain,

(2) problem-specific assertions, which express the hypotheses of the problem to be solved, and

(3) a goal statement, which expresses the problem itself.

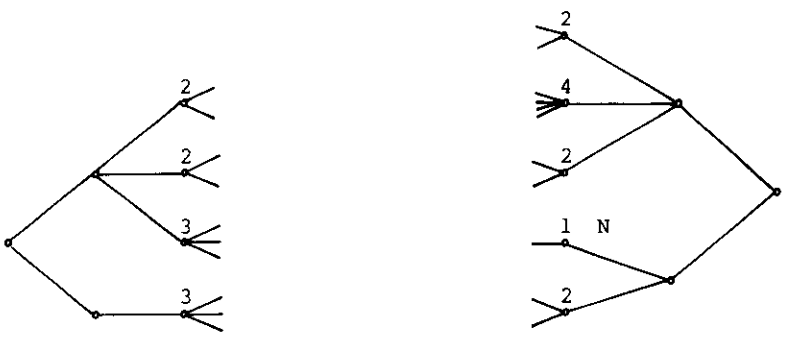
Problem-specific assertions can be absent from a given task description. But when they are present, it may be useful to combine top-down reasoning (from the problem to be solved) with bottom-up reasoning (from the hypotheses of the problem). However, it is important in this case to avoid bottom-up reasoning from assertions which are part of the general description of the problem-domain. This restricted use of bottom-up reasoning combined with top-down reasoning is a characteristic feature of Bledsoe's theorem-proving system [1971].

The majority of bottom-up proof procedures, however, do not distinguish betwen different types of assertions. As a result, they generally lead to combinatorially explosive behaviour, generating assertions which follow from the general description of the problem-domain, in addition to assertions which follow from the assumptions of the particular problem at hand.

A useful criterion for combining problem-specific bottom-up reasoning with top-down reasoning is a variation of the one proposed by Pohl [1972] for path-finding problems:

> At every step choose the direction of inference which gives rise to the least number of alternatives.

In the top-down direction, the number of alternatives is the smallest number of procedures which match the selected subgoal in a goal statement. In the bottom-up direction, it is the smallest number of assertions which can be derived from any assertion. The Pohl criterion is illustrated for a path-finding problem below.



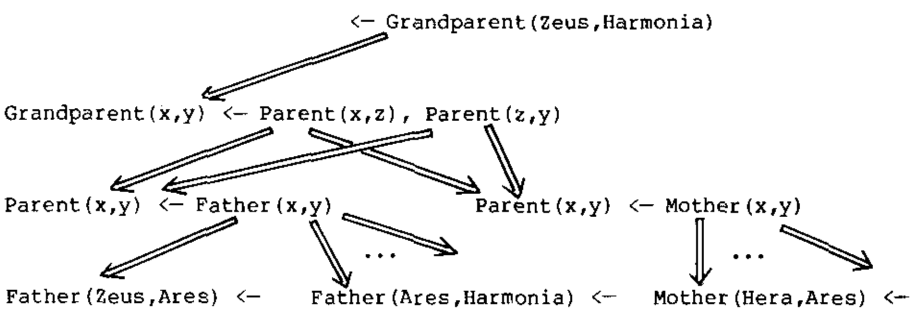
<u>The search space generated in one direction</u> <u>The search space generated in the other direction</u>

The number next to each node indicates the number of successor nodes. The Pohl criterion selects the direction associated with generating the successor of N. Given the previous formulation of the path-finding problem, bi-directional path-finding is accomplished by combining top-down and bottom-up reasoning.

A notation for describing bi-directional problem-solving

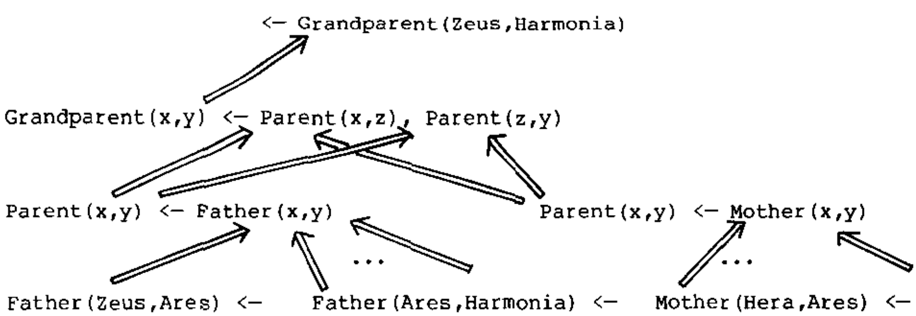
The distinction between top-down and bottom-up inference can be pictured using arrows to indicate the direction of reasoning. For every pair of matching atoms in the initial set of clauses (of which one is a condition and the other a conclusion) an arrow is directed from one atom to the other.

For top-down inference, arrows are directed from conditions to conclusions. For the grandparent problem, we obtain the following graph.



Reasoning is guided by the direction of the arrows. It starts with the initial goal statement, is transferred within procedures from conclusions to conditions and ends with the assertions.

For bottom-up inference, arrows are directed from conclusions to conditions.



Reasoning begins with the assertions, is transferred within procedures from conditions to conclusions, and ends with the goal statement.

The grandparent definition can also be used in a combined top-down, bottom-up manner. Different combinations can be represented by using numbers to indicate sequencing. For simplicity, we show only the notation associated with the grandparent definition. The combination of directions

```
      3
      ⇑
Grandparent(x,y) <- Parent(x,z), Parent(z,y)
                          ⇑            ⇓
                          1            2
```

represents the algorithm which

1) waits until x is asserted to be parent of z, then
2) finds a child y of z, and finally
3) asserts that x is grandparent of y.

The combination indicated by

```
      1
      ⇓
Grandparent(x,y) <- Parent(x,z), Parent(z,y)
                          ⇑            ⇓
                          2            3
```

1) responds to the problem of showing that x is grandparent of y,
2) by waiting until x is asserted to be parent of z, and then
3) attempting to show that z is parent of y.

The arrow notation can also be used for non-Horn clauses. In Chapter 8 it is used to control the behaviour of the connection graph proof procedure.

Another formulation of the path-finding problem

The effectiveness of a problem-solving strategy (such as bi-directional reasoning) depends on the problem-formulation rather than on the problem itself. This is shown by comparing the previous formulation of the path-finding problem with the one suggested by the representation of semantic networks.

In this representation we employ a predicate Go*(x,y) which expresses that it is possible to go from node x to node y. Assertions describe the arcs in the initial graph. The following assertions describe the graph at the beginning of the chapter.

```
Go*(A,B) <-          Go*(D,X) <-
Go*(A,C) <-          Go*(E,X) <-
Go*(B,D) <-          Go*(X,Z) <-
Go*(B,E) <-          Go*(Y,Z) <-
Go*(C,F) <-
```

In addition to the assertions, a single procedure is necessary for path-finding

```
Go*(x,y) <- Go*(x,z), Go*(z,y).
```

The problem of finding a path from A to Z is described by a single goal statement

```
<- Go*(A,Z).
```

Here the assertions are specific to the graph, whereas the path-finding procedure is general-purpose. However, only the goal statement is specific to the particular path in the graph. Bottom-up inference generates assertions about paths which are unmotivated by the particular path to be found. Both forward and backward search, as well as bi-directional search, can be accomplished by top-down inference alone. The direction of search depends on the choice of subgoal in the path-finding procedure. Selecting Go*(x,z) before Go*(z,y) is forward search. Selecting the two subgoals in parallel or timesharing between them gives rise to bi-directional search.

The path-finding problem can be formulated in different ways; the same problem-solving behaviour can be obtained from different formulations by applying different problem-solving strategies. Even the specific behaviour determined by the bi-directional path-finding strategy which at every step chooses the direction which grows least rapidly can be accomplished with both formulations. In the first formulation it is obtained by applying the Pohl criterion for combining top-down and bottom-up inference. In the second formulation it is accomplished by top-down inference alone, applying the strategy of selecting the subgoal to which fewest procedures (including assertions) apply.

Other aspects of problem-solving

Problem-solving can be classified into three main stages.

1) The first stage identifies the problem-domain and formulates problem-solving procedures.
2) The second stage applies the procedures to the solution of problems.
3) The third stage improves the problem problem-solving strategies and procedures.

This chapter has been restricted to a discussion of the second stage. It has not considered the other stages which are concerned with learning. In this respect we have followed the advice of McCarthy [1968] and Minsky [1968] to explore the adequacy of the representation language before dealing with the problems of formulating and improving the representation of the problem domain.

In the next chapter we investigate the interpretation of the Horn clause subset of logic as a programming language. This unifies problem-solving with programming. The first stage of problem-solving is the initial stage of problem formulation and specification. The second stage runs the specification as a program, and the third identifies inefficiencies and remedies them by improving the procedures and tailoring the problem-solving strategies to the problems to be solved.

In subsequent chapters we investigate the role of non-Horn clauses in problem-solving and the use of global problem-solving strategies. In the last chapter we compare the interpretation of logic as a model for problem-solving with the role of logic in philosophy as a model for representing beliefs and formalising arguments.

However, nowhere in this book do we investigate the problems of learning. Nor do we investigate such important strategies as problem-solving by example and by analogy.

Exercises

1) a) Express the arrow-inversion problem by means of Horn clauses without function symbols:

> Given three arrows in a row D U D, pointed down, up, down respectively, the goal is to reach the state D D D in which all arrows point down. The only action possible is to invert a pair of adjacent arrows, changing both their directions simultaneously.

Hint : Let State(x,y,z) express that there is a possible state in which the first, second and third arrows point in directions x, y and z respectively.

b) Show that the problem is unsolvable by generating the graph representation of the top-down search space and showing that it contains no solutions.

c) Describe how the clausal formulation of the problem can be modified in order to

i) invert adjacent arrows only when they have opposite directions,
ii) add an action which interchanges adjacent arrows,
iii) deal with a row of four arrows instead of three.

2) a) Express the farmer, wolf, goat and cabbage problem by means of Horn clauses:

> The farmer, wolf, goat and cabbage are all on the north bank of a river and the problem is to transfer them to the south bank. The farmer has a boat which he can row taking at most one passenger at a time. The goat cannot be left with the wolf unless the farmer is present. The cabbage,

which counts as a passenger, cannot be left with the goat unless the farmer is present.

b) Compare the graph representations of both the top-down and bottom-up search spaces.

c) Can you find useful evaluation functions to guide the search for a solution?

3) Given the two different representations of the path-finding problem, compare the problem-solving strategies needed

a) to recognise that there is no path from A to B if there is no arc leading from A or no arc leading to B and

b) to show that it is possible to go from A to A.

4) Let sequences be characterised by means of two relations

Item(i,j,k) which holds when i_j = k i.e.
the j-th element in the sequence i is k and
Length(i,u) which holds when the length of sequence i is u.

Thus the sequence

A: $a_1, a_2, \ldots, a_n$

can be characterised by means of the assertions:

Item(A,1,a_1) <–
Item(A,2,a_2) <–
.
.
.
Item(A,n,a_n) <–
Length(A,n) <–

Assume that Plus(x,y,z) holds when x+y = z.

a) Define by means of Horn clauses the relation Sum(x,v) which holds when v is the sum of the numbers in the sequence x.

b) Use the clauses of part (a) to find top-down the sum of the numbers in the sequence B: 3,4,1Ø.

c) Can Sum(x,v) be defined in such a manner that, given x to find v, the search space contains only the solution?

5) a) List all the solutions to the problem

<– Arch(w)

implied by the definition of arch and the description of the scene given by clauses A1-12.

b) Reformulate the definition of arch and tower by means of Horn clauses in order to eliminate as many pathological arches and towers as possible. (This problem can be solved more easily later using negation as failure, investigated in Chapter 11.)

6) Consider the problem

```
<- Numb(u), Numb(v), u > v
```

given the clauses

```
Numb(0) <-
Numb(s(x)) <- Numb(x)
s(x) > 0 <-
s(x) > s(y) <- x > y.
```

Analyse the behaviour of the backtracking search strategy for solving the problem. Assume that the solution of subgoals is attempted in the order in which they are written and that alternative clauses also are tried in the order given.