CHAPTER 9

Global Problem-Solving Strategies

In this chapter we investigate problem-solving strategies which deal with problems as a whole rather than with subproblems individually. Goal transformation deals with the combination of goals in goal statements, whereas analysis of differences deals with the effect of procedures on the difference between goals and assertions.

Goal transformation consists of a number of related strategies which are concerned with the logical relationships among subgoals. It includes deletion of redundant subgoals, which are implied by other subgoals, addition of implicit subgoals which are easier and more useful to solve than those which are explicitly given, rejection of inconsistent subgoals, which are mutually incompatible, and rejection of subgoals which are contradicted by an example.

The techniques of goal transformation are similar to those of program transformation developed for recursion equations by Burstall and Darlington [1977]. Program transformation transforms programs before problems are given, whereas goal transformation transforms goals during the course of attempting to solve them. Goal transformation techniques have also been used in robot plan-formation, mathematical programming and geometry theorem-proving.

Analysis of the differences between goals and assertions involves an even more global approach to problem-solving. It attempts to identify both procedures which reduce differences as well as those which increase them or leave them invariant. Preference can be given to procedures which reduce differences over those which do not. Goals can be rejected as unsolvable if it can be demonstrated that no procedure reduces differences at all.

The techniques of difference analysis are similar to ones used in program-proving. Demonstrating that programs reduce differences is involved in proving program termination, whereas demonstrating that programs leave properties invariant is used for proving program properties. The strategy of selecting procedures for their effectiveness at reducing differences is the basis, moreover, of the General Problem Solver developed by Newell, Shaw and Simon [1963].

Although the methods we describe can also be applied to non-Horn clauses, we shall simplify matters by limiting attention to top-down problem-solving by Horn clauses alone. Moreover, we shall not concern ourselves with the heuristics which would be needed for the effective utilisation of these methods.

Deletion of redundant subgoals

A subgoal can be deleted from a goal statement, if the assumption that the other subgoals have a solution implies that the redundant subgoal has a compatible solution as well. According to this criterion, assuming the transitivity of the $\leq$ relation,

$$x \leq y \leftarrow x \leq z,\ z \leq y$$

the goal statement

$$\leftarrow r \leq s,\ s \leq t,\ r \leq t$$

contains the redundant subgoal $r \leq t$. For, assuming that the other subgoals have a solution, it follows that the assertions

$$r' \leq s' \leftarrow$$
$$s' \leq t' \leftarrow$$

hold for appropriate instances r', s' and t' of the terms r, s and t respectively. But those assertions together with the transitivity of $\leq$ imply the assertion

$$r' \leq t' \leftarrow$$

which expresses that the third subgoal is compatibly solvable also. It is unnecessary to solve the redundant subgoal explicitly. It suffices to know that any solution of the other subgoals guarantees the existence of a compatible solution of the third subgoal as well.

The transitivity clause does not need to be part of the program or even a logical consequence of it. To justify deletion of the redundant subgoal, it suffices that transitivity be a property of the program. This is the case, for example, if the $\leq$ relation is defined by the clauses

$$0 \leq y \leftarrow$$
$$s(x) \leq s(y) \leftarrow x \leq y.$$

A statement is a property of a Horn clause program P, if it is consistent with P and together with P implies no variable-free assertions not already implied by P. A program property, therefore, adds no solutions to those which can be obtained by the program itself.

Deleting a duplicate occurrence of a subgoal is a special case of deleting a redundant subgoal, since any one occurrence of a subgoal implies any other occurrence. Thus the goal statement

$$\leftarrow P,\ Q,\ P$$

for example, can be replaced by

$$\leftarrow P,\ Q.$$

Addition of surrogate subgoals

Although it is often useful to delete redundant subgoals, it is sometimes beneficial to add them instead.

The strategy of deriving additional subgoals is common in mathematical programming, where subgoals are regarded as constraints to be satisfied. A surrogate constraint, whose solution is implied by the solution of the original constraints, can be added and then solved before the others. This is useful if the surrogate constraint is easier to solve and aids the solution of the original constraints by determining the values of some of their variables.

Consider, for example, the initial collection of two constraints involving the variables x and y:

```
<- x+y = 2, x-y = 0
```

A sequential, top-down problem-solver would generate pairs of numbers satisfying one of the constraints and then test whether they solve the other. A more intelligent problem-solver, programmed by Warren in PROLOG, coroutines between the two subgoals solving them simultaneously by successive approximation. The program, a general-purpose, Horn clause problem-solver, always selects a subgoal which contains fewest variables at the top-most level.

The normal, mathematical problem-solving method, however, derives and solves a surrogate constraint instead. It assumes that the original constraints have a solution and concludes (by adding the two equations together) that the additional constraint

```
<- 2*x = 2
```

must also be satisfied by the same solution. The new constraint is redundant in the new goal statement

```
<- x+y = 2, x-y = 0, 2*x = 2
```

but it can be solved without any search. Moreover, once it has been solved, the remaining instantiated original constraints

```
<- 1+y = 2, 1-y = 0
```

can then be solved without search as well. In fact, it suffices to solve just one of the remaining constraints, because the other constraint is now redundant.

The strategy of surrogate subgoals is useful for plan-formation problems. Consider, for example, the problem of finding a state w in which the robot is in the room and next to the box

```
<- In(Rob,room,w), Nextto(Rob,box,w).
```

Assuming that the box is not in the room initially and that the robot is more mobile than the box, it is useful to derive the surrogate subgoal

```
<- In(box,room,w)
```

from the original subgoals using the program properties

```
In(x,y,w) <- In(z,y,w), Nextto(x,z,w)
Nextto(x,z,w) <- Nextto(z,x,w).
```

If the surrogate subgoal is added to the original goal statement

```
<- In(Rob,room,w), Nextto(Rob,box,w), In(box,room,w)
```

and is selected for solution before the others, then the simplest solution (where the robot pushes the box into the room) finds a state w which directly solves the remaining subgoals.

Rejection of inconsistent goal statements

An entire goal statement can be rejected as unsolvable, if the assumption that it can be solved leads to contradiction.

A simple case is the one in which a goal statement is subsumed by a program property or an integrity constraint. The goal statement

```
G    <- On(A,B,w), Clear(B,w), On(B,C,w)
```

for example, is subsumed by the clause

```
C    <- On(x,y,z), Clear(y,z)
```

which expresses that nothing is clear and has something on it at the same time. In general, one clause C_1 subsumes another C_2 if all the conditions and conclusions of some instance of C_1 are contained among the conditions and conclusions of C_2. The subsuming clause is more general than the subsumed clause and possibly has fewer conditions or fewer conclusions. In the example above, the instance of the subsuming clause C (in which x = A, y = B and z = w) contains one fewer condition than the subsumed clause G.

A clause can be deleted from a set of clauses if it is subsumed by another clause in the same set. Deletion of the subsumed clause does not affect the consistency (or inconsistency) of the set of clauses as a whole. A thorough discussion of the completeness of deleting subsumed clauses is contained in the book by Loveland [1978].

The strategy of deleting a subsumed goal statement can be regarded as a special case of deleting an inconsistent one. In the preceding example, the assumption

```
On(A,B,s) <-
Clear(B,s) <-
On(B,C,s) <-
```

that there exists a solution w = s of the goal statement G is inconsistent with C.

Rejection of an inconsistent goal statement, however, is more general than deletion of a subsumed one. It can involve an arbitrary amount of

deduction. The database query

<— Teaches(John,y)

for example, is not subsumed by any of the clauses

T1 Teacher(x) <— Teaches(x,y)
T2 <— Teacher(x), Student(x)
T3 Student(John) <—

but is unsolvable because the assumption that it is solvable, namely

Teaches(John,A) <—

say, is inconsistent with T1-3.

Similar strategies for rejecting queries which are inconsistent with type information have been developed by McSkimin and Minker [1977] who augment a resolution theorem-prover with a semantic network which stores and processes type information. Subsumption of unsolvable goal statements is also a feature of plan-formation systems developed by Dawson and Siklossy [1977], Hewitt [1975] and, more generally, of the logic programming system developed by Robinson and Sibert [1978].

Generalising the use of diagrams in geometry

In order to justify the addition or deletion of a redundant subgoal, it is necessary that the assumptions used to derive the subgoal be properties of the procedures which can be used to solve it. In order to justify rejection of an inconsistent goal statement, however, a weaker condition suffices: The assumptions A used to derive inconsistency need only be consistent with the procedures P.

For, suppose that

(i) P is consistent with A,
(ii) G* expresses that the goal statement G is solvable,
(iii) G* is inconsistent with P and A, but
(iv) P solves G.

Then, since P solves G, P implies G* and therefore P together with A implies G*. But then, since P is consistent with A, G* is consistent with P and A , contradicting (iii). It follows that

if P is consistent with A, but
G* is inconsistent with P and A, then
P does not solve G.

The use of diagrams to reject unsolvable subgoals in Gelernter's Geometry Theorem Proving Machine [1963] can be regarded as a case of using assumptions which are consistent with the problem-solving procedures to reject inconsistent goal statements. The axioms of geometry function as procedures and the description of the diagram functions as the additional assumptions. The use of a diagram is justified, provided its description is consistent with the general axioms

of geometry and with the particular hypotheses of the theorem to be proved. Gelernter estimated that the use of diagrams reduced the size of search spaces on the average to 1/200 their original size. The argument above shows that the use of examples to recognise the unsolvability of problems need not be restricted to geometry. Examples can be used to recognise and reject unsolvable subgoals in any problem-domain.

Goals as generalised solutions

It is sometimes useful not to solve subgoals explicitly but to regard them instead as standing for the general class of all their solutions.

Consider, for example, an initial goal statement

```
<- G(x)
```

which eventually reduces to the subgoal

```
<- x > 0.
```

Instead of generating an arbitrary positive number x as an explicit solution, it is more informative to report that any positive number is a solution. This can be effected by regarding the subgoal x > 0 as a generalised solution which stands for the class of all its individual solutions.

Solving subgoals by generalised solutions is a feature of Bledsoe's approach to theorem-proving [1971, 1977]. To be effective, it needs to be combined with goal transformation. Given a goal statement

```
<- x > 0, x > 1, G(x)
```

for example, deletion of the redundant subgoal is necessary to transform the goal statement to the new one:

```
<- x > 1, G(x)
```

Given

```
<- x < 0, x > 1, G(x)
```

on the other hand, rejection of inconsistent subgoals is necessary to recognise that the goal statement is unsolvable.

Treating certain kinds of goals as generalised solutions is also useful for database queries, and is a feature both of Darlington's [1969] resolution information retrieval system and of McSkimin and Minker's [1977] semantic network theorem-prover. Given the query

```
Who teaches programming?
<- Teaches(x,programming)
```

and the general rule

```
All professors teach programming.
Teaches(x,programming) <- Professor(x)
```

it is better to regard the resulting subgoal as a generalised solution

```
<- Professor(x)
```

than it is to report one or more of the answers which qualify as solutions as a result of the assertions

```
Professor(Mary) <-
Professor(John) <-
Professor(Bob) <- .
```

Goal transformation and the information explosion

It is a characteristic of human problem-solving that the assimilation of additional information generally improves problem-solving efficiency. This contrasts with the simple model of problem-solving in which all knowledge is used as problem-solving procedures. Additional information only increases the size of the search space and makes problems harder to solve (except in those cases where only one solution is required and the non-determinism$_1$ doesn't matter). In the goal transformation model, however, additional information can be used to transform goal statements and to reduce the size of the search space.

Loop detection by analysis of differences

Like goal transformation, analysis of differences adds to the possiblities of recognising that a procedure goes into a loop.

Consider, for example, the procedure

```
Numb(x) <- Numb(s(x))
```

given the goal

```
<- Numb(s(s(∅)))
```

and the assertion

```
Numb(∅) <- .
```

Repeated top-down execution of the procedure gives rise to the non-terminating, infinite sequence of subgoals:

```
<- Numb(s(s(∅)))
<- Numb(s(s(s(∅))))
      .
      .
      .
```

In this case the connection graph proof procedure avoids the loop, because the procedure call Numb(s(x)) has only a pseudo-link to the head of the procedure. It follows that the procedure is unusable and can be

deleted from the graph. If the assertion Numb(∅) <- is replaced by the assertion

Numb(s(∅)) <-

however, application of the procedure gives rise to the same infinite loop, but the procedure can no longer be deleted, because its procedure call has an additional non-pseudo-link to the new assertion. The loop can be avoided in all these cases, though, if it can be recognised that application of the procedure cannot reduce the difference between the goal and the assertion. The goal differs from the assertion in that it contains a greater number of occurrences of the function symbol s. Application of the procedure only increases the difference by generating subgoals which contain even more occurrences of s.

The global nature of difference analysis becomes apparent if the assertion is replaced by the new assertion

Numb(s(s(s(s(∅))))) <- .

Now, application of the procedure reduces the difference between the goal and the assertion and eventually solves the problem.

<- Numb(s(s(∅)))

<- Numb(s(s(s(∅))))

<- Numb(s(s(s(s(∅)))))

□

A procedure might be needed for a solution even if it increases the difference between the goal and the assertions. Given, for example, the goal

<- Numb(s(s(s(∅))))

and the assertion

Numb(∅) <-

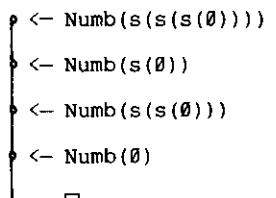
the procedure

Numb(s(s(x))) <- Numb(x)

decreases the difference, whereas the procedure

Numb(x) <- Numb(s(x))

increases it. But both procedures are necessary to solve the problem.



In the preceding examples the application of a procedure which increases differences either generates a loop or else is essential for a solution. More often, increasing differences neither contributes to a solution nor prevents its being found. Such is the case with the pair of procedures

Numb(s(x)) <- Numb(x)
Numb(x) <- Numb(s(x)).

If one of them unnecessarily increases differences, the other can be used to restore them to their previous state. Indeed using one procedure after the other simply generates the kind of loop which can be avoided in the connection graph proof procedure by deleting links whose resolvents are tautologies.

In all of these examples, the difference between subgoals and assertions can be measured simply by the number of occurrences of the function symbol s. In other cases the characterisation of differences is more complicated.

The factorial example

The definition of factorial is a more realistic example. The non-clausal sentence

Times(s(x),u,v) -> [Fact(x,u) <-> Fact(s(x),v)]

gives rise to two Horn clause procedures:

(1) Fact(s(x),v) <- Fact(x,u), Times(s(x),u,v)
(2) Fact(x,u) <- Fact(s(x),v), Times(s(x),u,v)

Given the assertion

Fact(∅,s(∅)) <-

there is no goal for which the second procedure is necessary. However, given the assertion

Fact(1∅,3628800) <-

instead, the second procedure is necessary for solving the problem

<- Fact(s(∅),x)

and the first procedure is unnecessary. Here the natural number n is used as an abbreviation for the term

$$\underbrace{s(s(s(\ldots(\emptyset)\ldots)))}_{n \text{ times}}$$

containing n occurrences of the function symbol s.

More generally, it may be useful to have several assertions, e.g.

```
Fact(Ø,1) <-
Fact(1Ø,3628800) <-
```

and, using analysis of differences, to apply the procedure which most quickly narrows the gap between the problem and the assertions, using (1), for example, for the problem

```
<- Fact(3,x)
```

and using (2) for

```
<- Fact(8,x).
```

Notice that the last example is a case of "don't care" non-determinism$_1$. There are several ways of finding the factorial, all of which lead to the same result. It doesn't matter which method is chosen. But, if backtracking is used, then it does matter (for the sake of efficiency) that only one method is tried.

Invariant properties of procedures

The unsolvability of a problem can be detected not only by analysing the effect of procedures on differences but also by analysing the properties which procedures leave invariant. A problem can be recognised as unsolvable if it can be shown that it differs from the assertions in a property which is not affected by the procedures. A typical property of this kind is parity.

Suppose we are given the clauses

```
Even(8) <-
Even(s(s(x))) <- Even(x)
Even(x) <- Even(s(s(x)))
<- Even(17)
```

By analysis of differences, the second procedure can be rejected as useless. Used alone it only increases differences. Used together with the other procedure it only generates loops. By analysis of invariants the first procedure can also be rejected. It reduces a problem of a given parity to a subproblem of the same parity. No matter how many times the procedure is used it cannot change the parity of the original problem. Since the original problem has an odd number of occurrences of "s" and the assertion has an even number, the procedure cannot be used to solve the problem. Here parity can be determined by counting occurrences of the

function symbol "s". In more realistic cases the invariant property is more complex.

Such is the case in the following example, where the invariant property is another form of parity. Given a sequence of six arrows (or coins) each of which can face up or down, the problem is to transform them from one state to another - for example, from

U U U D D D to U U D D U U

There is only one action available: it is possible simultaneously to change the direction of two adjacent arrows.

A simple n-tuple representation in which

$$State(d_1, d_2, d_3, d_4, d_5, d_6)$$

expresses that

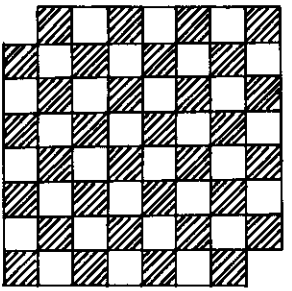
the first arrow can have direction d_1,
the second arrow can have direction d_2,
and in general
the i-th arrow can have direction d_i

simultaneously, is the following.

```
State(U,U,U,D,D,D) <-

<- State(U,U,D,D,U,U)

State(x,y,z,u,v,w) <- State(x',y',z,u,v,w), Opp(x,x'), Opp(y,y')

State(x,y,z,u,v,w) <- State(x,y',z',u,v,w), Opp(y,y'), Opp(z,z')

State(x,y,z,u,v,w) <- State(x,y,z',u',v,w), Opp(z,z'), Opp(u,u')

State(x,y,z,u,v,w) <- State(x,y,z,u',v',w), Opp(u,u'), Opp(v,v')

State(x,y,z,u,v,w) <- State(x,y,z,u,v',w'), Opp(v,v'), Opp(w,w')

Opp(U,D) <-

Opp(D,U) <-
```

The problem is unsolvable, because, whereas the procedures leave invariant the parity of the number of arrows in either direction, in the assertion there is an odd number of arrows in both directions and in the goal there is an even number. To show that the procedures leave parity invariant it is necessary to consider the two cases: Either the two inverted arrows have the same direction before inversion or they have different directions. If they have the same direction, then inversion increases the number of arrows in one direction by two and decreases the number in the other by two, but leaves the parity the same. If they have different directions, then inversion leaves the number of arrows in both directions unchanged and therefore does not affect the parity. In both cases parity is an invariant property of the procedures.

The mutilated checker board problem is similar. Given a checker-board with two opposite corners removed,



the problem is to cover it with dominoes, each one of which covers two adjacent squares. Since adjacent squares have different colours, the procedures leave invariant the difference between the number of uncovered squares of different colours. The problem is unsolvable, therefore, because in the goal state the difference is zero, but in the initial state it is two.

There is an obvious relationship between proving that logic procedures leave a property invariant and proving a property of a flowchart program using invariants. In both cases the objective is to show that if a property holds at the beginning of a repetitive process then it holds at the end. This is done by showing that if it holds at the beginning of one step of the process then it holds at the end of the step. The desired result then holds by induction.

Exercises

1) Suppose y is a function of x in the relation F(x,y), i.e.

$$y = z \leftarrow F(x,y),\ F(x,z)$$

where the only clause defining equality is

$$x = x \leftarrow .$$

Show how goal transformation can be used to eliminate redundancy when a goal statement contains a pair of subgoals of the form

$$F(r,s) \text{ and } F(r,t)$$

where r, s and t are terms.

2) Show that "goal transformation" can be used to justify transforming the clause

$$\text{Tower}(t(x,y)) \leftarrow \text{Block}(x),\ \text{Tower}(y),\ \text{On}(x,y)$$

into the clause

Tower(t(x,y)) <- Tower(y), On(x,y).

What property of the On relation is needed for the transformation?

3) In Chapter 6, the precondition Diff(x,z) can be eliminated from the definition of the action trans(x,y,z) and its use can be replaced by that of the integrity constraint

<- Holds(on(x,x), w)

instead. Compare the problem-solving behaviour needed for these two alternative formulations of the plan-formation task.

4) Analyse the English sentence

S1 Reject stealing as a way of having something if you also want to be virtuous.

as a recommendation concerning the use of the procedure

Have(u,x) <- Steal(u,x)

applied to goal statements containing two subgoals of the form

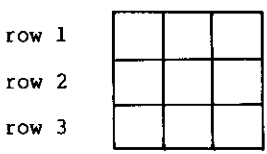
Have(r,s) and Virtuous(r).

Can the notions of goal transformation be used to establish a logical relationship between the sentences S1, S2 and S3?

S2 Do not steal if you want to be virtuous.
S3 Anyone who steals is not virtuous.

5) Discuss the formalisation of the following problems and the problem-solving strategies needed to solve them intelligently.

a) Find an assignment of digits 1,2,3,...,9 to the cells of a 3 by 3 matrix such that:



(i) Exactly one digit is assigned to each cell.

(ii) No digit is assigned to more than one cell.

(ii) The three digit number in row 3 is the sum of the three

digit numbers in rows 1 and 2.

(iv) If the digit i is assigned to a cell then the digit i+1 is assigned to a cell which is horizontally or vertically adjacent.

b) Find an assignment of digits 1,2,3,...,9 to letters in the names such that:

```
 DONALD
+GERALD
--------
 ROBERT
```

(i) Exactly one digit is assigned to each letter.

(ii) No digit is assigned to more than one letter.

(iii) The 6 digit number assigned to the word "ROBERT" is the sum of the 6 digit numbers assigned to "DONALD" and "GERALD".

(iv) 5 is assigned to "D".