Solving Rubik's Cube

Prolog In Action

by Dennis Merritt

(Originally published in PC AI magazine)

Introduction

This article describes a Prolog program which solves Rubik's cube. The program illustrates many of the knowledge engineering problems encountered in building expert systems, such as knowledge representation, state space manipulation and the codification of expertise. Performance - that is, speed and efficiency - is a key issue and affects most of the design decisions in the program.

The program is a classic example of a "dumb" expert system. It unscrambles cubes as fast or faster than a human expert. However, it does not have the intelligence to discover how to solve Rubik's cube. It simply lists, in a form that the machine can follow, a collection of the rules that an expert would use to solve the cube. The rules are based on the expertise contained in Unscrambling the Cube by Black & Taylor.

Because of the large knowledge base involved in a program like this, a blind search strategy that picks rules at random simply will not work. For this reason, such programs must use heuristic programming to direct the watch. Using various heuristic (intelligent) rules, the search space can be drastically reduced so that the problem can be solved in a reasonable amount of time.

This problem, like many other AI problems, resolves itself into three main design decisons. First, we must decide how to represent the current state of the cube. Then, we must decide how to represent the things that can be done to the cube; that is, we must represent the rules of manipulation. Finally, we must decide how to figure out which of the rules to apply in any one given situation. Each of these factors figures heavily in the program design.

Prolog was chosen as the language in which to code this problem because of its tremendous power and expressiveness. These will be illustrated in the discussion of the program design. Like every language, Prolog has its strengths and its weaknesses; this example will show some of both.

Stating the Problem

Rubik's cube is a deceptively simple-looking puzzle. It is a cube with nine tiles on each face. In its solved state, each of the sides is made up of tiles of the same color, with a different color for each side. Each of the tiles is actually part of a small cube, called a "cubie." Each face of the cube (made up of nine cubies) can be rotated. The mechanical genius of the puzzle is that the same cubie can be rotated from multiple sides. A corner cubie can move with three sides; an edge cubie moves with two sides.

The basic goal is to take a cube whose sides have been randomly rotated and figure out how to get it back to the initial solved state. The problem is, there are an astronomical number of possible ways to manipulate the cube, and only a small proportion of these actually lead to the solved state. To reach a solution using a blind search algorithm is not feasible; even on the largest machines it takes a prohibitively large amount of time. A human expert can unscramble the cube in well less than a minute; obviously, the human expert does not use blind search. The difficulty with solving the cube revolves around the fact that if you move one cubie, you have to move seven other cubies as well (the center one doesn't really go anywhere). This is not a big problem in the early stages of unscrambling the cube, but once a number of tiles are positioned correctly, new rotations tend to destroy the solved parts of the cube. The experienced cube solver knows some complex sequences of moves

which can be used to manipulate a small portion of the cube without disturbing the other portions. For example, this program knows a specific 14-move sequence that can be used to twist two corner pieces without disturbing any other pieces.

Prolog Predicates

Predicates are the basic computational building blocks of prolog programs. They are called predicates since they simply succeed or fail. The ":-" symbol is often read as "if." Thus, a prolog predicate can be read as: "The first term is true if the following terms are true." The arguments are used to pass information between predicates through the magic of unification.

Each predicate can contain multiple clauses. When one prolog predicate calls another, Prolog finds the clauses that makes the goal statement true, or else fails. In the case of "move" there are two clauses. If the calling pattern has a "+M" pattern, then the first clause will be used. If the calling pattern has a "-M", then the second clause will be used.

The "move" predicate can be paraphrased as saying: 1) A positive move relationship exists between the old and new states if a "mov" relationship exists between them; or 2) similarly, a negative move relationship exists.

Prolog uses unification to match the patterns of "mov" in the predicate "move" with the actual "mov" predicates stored in the program. If "OldState" has a cube as a value, then "NewState" will have the rotated state of that cube as a value.

Representing the Cube

The core of any AI program is knowledge representation. In this program, the core is the representation of the cube and its fundamental rotations. The cube lends itself to at least two obvious representation strategies. It can either be viewed as 54 separate tiles or as 28 different cubies (called "pieces" in the program). Eight of the pieces have one tile (the part of the piece that is visible), 12 have two tiles, and 8 have three tiles. Since much of the intelligence in the program is based on locating pieces and their positions on the cube, a representation which preserves the piece identity is preferred. However, there are also brute-force search predicates which need a representation which can be manipulated fast. For these predicates a simple flat structure of tiles is best.

The next decision is whether to use flat Prolog data structures (terms) with each tile represented as an argument of the term, or to use Prolog lists with each tile represented as an element of the list. Lists are much better for any predicates which might want to search for specific pieces, but they are slower to manipulate as a single entity. Flat data structures (terms) are more difficult to tear apart argument by argument, but are much more efficient to handle as a whole. Based on the conflicting design constraints of speed and accessibility, the program actually uses two different notations. One is designed for speed, using flat data structures and tiles; the other is a list of pieces designed for use by the analysis predicates.

The cube is then represented by either the structure:

```
cube(X1, X2, X3, X4, ........, X53, X54)
```

where each X represents a tile, or by the list:

```
[p(X1),p(X2),...p(X7,X8,X9),...
       p(X31,X32),p(X33,X34),...]
```

where each p(..) represents a piece. A piece has one, two, or three arguments, representing its visible faces. Pieces with one argument are center pieces; pieces with two arguments are edge pieces; pieces with three

arguments are corner pieces. (Upper case letters signify variables in Prolog; square brackets signify a list.) The tiles are each represented by an uppercase letter representing the side of the cube the tile should reside on when the cube is in its solved state. The six sides are front, back, top, bottom, right, and left. Quotes are used to indicate that the tiles are constants, not variables. Using the constants, the solved state (or goal state of the program) is stored as the Prolog fact "ghoul" ("ghoul" is used instead of "goal" due to a long-gone taboo regarding reserved words):

```
ghoul( cube('F', 'R', 'U', 'B', ............))
```

Unification in Prolog

Unification is an extremely powerful feature of prolog, and is used heavily in the Rubik's cube program. Unification is basically a pattern matching feature built into the language. If two patterns containing variables can be matched, it determines the values of the variables that will make the patterns match. The variables can contain simple or complex Prolog terms.

Unification can be used explicitly by attempting to unify two terms using the "=" operator. It is more often used implicitly through Prolog's calling method. When a predicate is refered to in a Prolog goal (or query), Prolog attempts to unify the arguments of that predicate with arguments of the goal.

A simple example of unification will illustrate how it is used in the program. The predicate

```
swap(foo(X, Y), foo(Y, X)).
```

can be used to swap the arguments of "foo". It simply states that the predicate "swap" is true if the arguments of "foo" are reversed. The scope of the two variables is such that the two X's must be equivalent (or unify), as must the two Y's.

The following interpreter session illustrates how "swap" might be used. The second line is the interpreters response to the query line.

```
?- swap (foo(1, 2), Z).
Z= foo (2, 1).
```

By matching the query pattern against the predicate "swap", Prolog determines that the two unify if the variable Z is equal to the Prolog structure "foo (2, 1)." In this way we have used the predicate "swap" to generate a "foo" with reversed arguments. The same predicate can also be used in the opposite direction:

```
?- swap (Z, foo (3, 4)).
Z= foo (4, 3).
```

Manipulating the Cube

Having decided on two representations, it is necessary to change quickly from one to the other. Unification has exactly the power we need to transform between one notation of the cube and the other. A predicate "pieces" takes the flat structure and converts it to a list, or vice versa.

```
pieces( cube(X1, X2, .......X54),
        [p(X1), ...... p(X7, X8, X9),.....]).
```

If Z is a variable containing a cube in structure notation, then the query

```
?- pieces(Z,Y).
```

will bind the variable Y to the same cube in list notation. It can also be used the other way. The following query can be used to get the goal state in list notation in the variable PieceState:

```
?- ghoul(FlatState), pieces(FlatState, PieceState).
  FlatState = cube('F', 'R', 'V', 'B', .....).
  PieceState = [p('F'), p('R'), .... p('R', 'U'),
       ......p('B', 'R', 'F'), ......].
```

The first goal unifies "FlatState" with the initial cube we saw earlier. "pieces" is then used to generate "PieceState" from "FlatState". Unification also gives us the most efficient way to rotate a cube. Each rotation is just a predicate which maps one arrangement of tiles to another. The first argument is the name of the rotation, while the second and third arguments represent a clockwise turn of the side. For example, the rotation of the upper side is represented by:

```
mov(u, cube(X1, ...X6, X7, X8, X9, ...),
cube(X1, ...X6, X20, X19, X21, ...))
```

We can apply this rotation to the top of the goal cube:

```
?- ghoul(State), mov(u, State, NewState).
```

The variable "NewState" would now have a solved cube with the upper side rotated clockwise. Since these can be used in either direction, we can write a higher level predicate that will make either type of move based on a sign attached to the move.

```
move(+M, OldState, NewState):-
  mov(M, OldState, NewState).
move(-M, OldState, NewState):-
  mov(M, NewState, OldState).
```

Having now built the basic rotations, it is necessary to represent the complex sequences of moves that unscramble the cube. In this case the list notation is best. For example, a sequence which rotates three corner pieces is represented by:

```
seq(tc3, [+r, -u, -l, +u, -r, -u, +l, +u]).
```

The sequence can be applied to a cube using a recursive list predicate, 'move_list':

```
move_list( [ ], X, X).
move_list( [Move|T], X, Z):-
  move(Move, X, Y),
  move_list(T, Y, Z).
```

At this point we have a very efficient representation of the cube and a means of rotating it. We next need to apply some expertise to the search for a solution.

Unification with Variables

The use of unification to determine whether or not a cube is in the desired partially solved state is very powerful. However, it is difficult to understand by just reading the code. Here is the basic technique for a simple puzzle which might be trying to get some numbers in order.

Suppose the goal of a puzzle was:

```
ghoul (puzzle(1, 2, 3, 4, 5)).
```

and the state was:

```
state (puzzle(2, 4, 1, 5, 3)).
```

The initial criteria for the puzzle would be:

```
crit (puzzle(_, _, _, _, _)).
```

where each of the "_'s" indicates a variable whose value we don't really care about.

To solve for the first piece in this simple puzzle, the criteria would first be set:

```
crit (puzzle(1, _, _, _, _)).
```

Now, this can be used to test a given solution. The following unifications are key.

```
?- puzzle(1, _, _, _, _) = puzzle(2, 4, 1, 5, 3).
no
```

This fails since the first argument is not a 1.

```
?- puzzle(1, _, _, _, _) = puzzle(1, 4, 2, 5, 3).
yes
```

This succeeds since the first argument is a 1. The other four arguments can be anything, since they are unified with variables. The next stage would set criteria:

```
crit (puzzle(1, 2, _, _, _)).
```

This would only unify with a state which had the first two peoces in place. Thus, if State and Crit were two variables bound to "puzzle" data structures, a search routine for solutions to this puzzle could end its search with the test:

```
..., State = Crit.
```

Then when the state matches the criteria, the predicate will succeed. Otherwise, the last unification will fail, causing the predicate to backtrack and resume whatever searching it was doing. This gives a very rapid way of testing potential solutions.

High Level Rules

The most obvious rule for solving Rubik's cube is to attack it one piece at a time. The placing of pieces in the solved cube is done in stages. In Black & Taylor's book they recognize six different stages which build the cube up from the left side to the right. Two examples of stages are, "put the left side edge pieces in place" and "put the right side corner pieces in place." The particular knowledge necessary to solve each stage is stored in predicates, which are then used by another predicate, "stage," to set up and solve each stage. Each stage has a plan of pieces it tries to solve for. These are stored in the predicate "pln," which contains the stage number and a list of pieces. For example, stage 5 looks for the four edge pieces on the right side:

```
pln(5, [p('R', 'U'), p('F', 'R'),
        p('R', 'D'), p('B', 'R')]).
```

Each stage will also use a search routine which tries various combinations of rotations to position a particular target piece. Different rotations are useful for different stages, and these are stored in a predicate similar to "pln". The predicate is "cnd" which contains the candidate rotations for the stage.

For example, the first stage (left edge pieces) can be solved using just the simple rotations of the right, upper, and front faces. The last stage (right corner pieces) requires the use of powerful sequences which exchange and twist corner pieces without disturbing the rest of the cube. These have names such as corner-twister 3 and tricorner 1. They are selected from Black and Taylor's book. These two examples are representative:

```
cnd(l, [r, u, f]).
cnd(6, [u, tc1, tc3, ct1, ct3]).
```

The "stage" predicate drives each of the stages. It basically initializes the stage, and then calls a general purpose routine to improve the cube's state. The initialization of the stage includes setting up the data structures that hold the plan for the stage and the candidate moves. "Stage" also reorients the cube to take advantage of symmetries and/or make for better displays.

Improving the State

The predicate "stage" calls the main working predicates which search for the rotations to put a given piece in place, and update all of the appropriate data structures. The representation of the partially solved cube is a key design issue for this portion of the program. How do the search routines know when the target piece is successfully in place? There are predicates in the program which search through a cube in list- piece notation (rather than tile notation) and determine where a piece is, or conversely, which piece is in a given position. These predicates are useful for many portions of the program but are too slow to be used for testing whether a given search has been successful or not. This is true since they not only have to check for the new piece being placed, but they also have to insure that none of the previously placed pieces have moved.

Unification is again the answer. So far, there are two "cube" terms used in the program. One represents the final solved state of the cube, and the other represents the current state of the cube. We introduce a third "cube," referred to as the criteria, which is used to denote which tiles are currently in place, and the tiles of the cubie which is currently being positioned.

Initially all of the arguments of this third cube are variables. This structure will unify with any cube. As pieces are put in place, the variables representing tiles of the criteria cube are unified with the corresponding tiles of the solved cube. In this case, the criteria cube will only unify with a cube that has those corresponding tiles in place. As the program attempts to place each piece, it binds another piece in the criteria. For example, as the program attempts to position the sixth piece, the "improve" predicate first binds the sixth piece in the criteria with the solved state. At this point, the first six pieces will have bound values the same as the solved state. The remaining tiles will be represented by unbound variables which unify with anything. The criteria cube will then successfully unify with any cube that has the first six pieces in place.

The Search

Now that we have a plan of attack on the cube, and a means of representing the current state, and the criteria for testing if a given piece is in place, we can institute a very fast search routine. The core routine to the Rubik's cube program is a predicate "rotate".

```
rotate([], State, State).
rotate(Moves, State, Crit):-
  rotate(PriorMoves, State, NextState),
  get_move(ThisMove, NextState, Crit),
  append(PriorMoves, [ThisMove], Moves).
```

The variable Moves is unbound at calling, and contains the list of moves necessary to position the piece after the search has succeeded. State is the current state of the cube, and Crit is the criteria for this stage of the solution. Crit has all of the pieces found so far bound, as well as the one additional piece for this search. "rotate" searches for a sequence of moves which will put the new piece in place without disturbing the existing pieces.

"rotate" illustrates the tremendous power and compactness of Prolog code. At the same time it illustrates the difficulty of understanding some Prolog code. Prolog's power derives from the built-in backtracking execution and unification. Both of these features help to eliminate many of the standard programming structures normally used. Thus, a predicate like "rotate" has a fraction of the code it would take in another language (and executes fast as well), but it requires a good understanding of the underlying execution behavior of Prolog to understand it.

"rotate" does a breadth-first search, as can be seen by the fact that it calls itself recursively before it calls the move generation predicate "get-move". Since the application of moves and testing is so fast, and the depth of search is never great, intermediate results are not saved as in a normal breadth-first search. Instead, they are just recalculated each time.

"append" is a predicate which can be used to build lists. In this case it takes ThisMove and appends it to the end of the list PriorMoves, generating a new list, Moves.

The candidate moves for a given stage are stored in a predicate "cand" (the actual program is a little more complex) which is maintained by the "stage" predicate. For stage one, it would look like:

```
cand(r).
cand(u).
cand(f).
```

"get_move" is called with the Move unbound, and the second and third arguments bound to the current state and a criteria respectively. If the call to "move" fails (because it does not rotate the cube into a position which unifies with the criteria), then "cand" backtracks, generating another possible move. When all of the positive moves fail, then "get_move" tries again with negative moves.

```
get_move(+Move, State, Crit):-
  cand(Move),
  mov(Move, State, Crit).
get_move(-Move, State, Crit):-
  cand(Move),
  mov(Move, Crit, State).
```

The efficiencies in "rotate" show the rationale behind the early design decisions of cube representation. "get_move" is called with the State and the Crit. If it generates a move which unifies with Crit, it succeeds; otherwise it fails and backtracks. All of this testing and analysis is done automatically by Prolog's pattern-matching call mechanism (unification).

The entire logic of the breadth-first search also happens automatically due to the backtracking behavior of Prolog. If get-Move fails to find a move which reaches the criteria, "rotate" backtracks into the recursive call to "rotate". Since the recursive call to "rotate" uses NextState as the criteria, and NextState is unbound, the recursive call will succeed in generating PriorMoves and a modified state.

Now "get_move" tries again with this new state to see if a single move will reach the criteria. This process repeats through as many levels of depth as is necessary to find a sequence of moves which reach the criteria. In practice, any more than a three-deep search begins to get tedious. 'The design of the program is such that it does not require more than a three-deep search to find and position any given piece.

Recursion and Backtracking in Prolog

"move_list" is a typical example of a Prolog recursive predicate. Each time "move_list" is called, the second argument, X, is rotated by "move" into the argument Y. Y is now recursively fed into "move_list". In each level of recursion, the third argument, Z, is passed down unchanged. In fact, Z is not bound to any value at the call to "move_list". When the list is empty, the first clause of move_list takes effect. It simply equates the second and third argument. This ripples back up through the layers of recursion, setting Z to the cube after all of the rotations have been applied.

Prolog is unusual among programming languages in that it has bi- directional execution. This means that after a given statement is executed, control might go to either the next or the previous statement. This decision is based on whether the call succeeded or failed. If a predicate has multiple clauses, and it is called from backtracking, it will attempt to find another clause which succeeds. If it does, then forward execution resumes again.

This mechanism is clear in a predicate such as "get_move". Each time it is called on backtracking it generates a new move. If there are no more new moves it fails and causes the calling statement to pass control to the statement before it.

More Heuristics

The program as described so far almost works. However, it turns out there are a few situations that will cause the search routines to dig too deep for a solution. These situations drastically affect the performance.

It was necessary to add more intelligence to the program to recognize situations that will not be easily unscrambled by the search routine, and to correct them before calling "rotate." One of the problems occurs when positioning pieces on the left side. If the piece to be positioned is currently on the right side, then a few simple moves will put it in place on the left side. However, if the piece is already on the left side, but in the wrong position, then it will have to be moved to the right and back to the left. This longer sequence of moves takes longer to search for, so one of the extra heuristics looks for this situation.

The heuristics analyze the cube, test for this condition, and blindly move the piece to the right if it occurs. Then the normal search routine gets it back into its proper place. There are a couple of situations like this which are covered by the heurisitics.

It is tempting to think of adding more and more of these heuristics to straighten out the cube with less searching. There is a tradeoff, however, and that is it takes time to apply the heuristics, and the search routine is really fast. So a heuristic is only worthwhile when the search is slow. I feel the current program may be improved by additional heuristics, but the search will still be the core of the program.