6 Three Meta-Interpreters: Prolog in Prolog, EXSHELL, and a Planner

Chapter Objectives

Prolog's meta-predicates used to build three meta-interpreters
- Prolog in Prolog
- An expert system shell: `exshell`
- A planner in Prolog

The Prolog in Prolog interpreter:
- Left-to-right and depth-first search
- Solves for a goal look first for facts, then rules, then ask user

`exshell` performed, using a set of `solve` predicates:
- Goal-driven, depth-first search
- Answers `how` (rule stack) and `why` (proof tree)
- Pruned search paths using the Stanford certainty factor algebra

The Prolog planner
- Uses an add and delete list to generate new states
- Performs depth-first and left-to-right search for a plan

Chapter Contents

6.1 An Introduction to Meta-Interpreters: Prolog in Prolog

Meta Interpreters

In both Lisp and Prolog, it is easy to write programs that manipulate expressions written in that language's syntax. We call such programs *meta-interpreters*. In an example we will explore throughout this book, an *expert system shell* interprets a set of rules and facts that describe a particular problem. Although the rules of a problem situation are written in the syntax of the underlying language, the meta-interpreter redefines their semantics. The "tools" for supporting the design of a meta-interpreter in Prolog were the *meta predicates* presented in Chapter 5.

In this chapter we present three examples of meta-interpreters. As our first example, we define the semantics of pure Prolog using the Prolog language itself. This is not only an elegant statement of Prolog semantics, but also will serve as a starting point for more complex meta-interpreters. `solve` takes as its argument a Prolog goal and processes it according to the semantics of Prolog:

```
solve(true) :-!.
solve(not A) :- not(solve(A)).
solve((A, B)) :-!, solve(A), solve(B).
solve(A) :- clause(A, B), solve(B).
```

The first `solve` predicate checks to see if its argument is a fact and true. The second checks to determine whether the argument of `solve` is false and makes the appropriate change. The third `solve` predicate sees if the argument of the `solve` predicate is the `and` of two predicates and then calls `solve` on the first followed by calling `solve` on the second. Actually, the third `solve` can handle any number of `and`ed goals, calling `solve` on the first and then calling `solve` on the set of `and`ed remaining goals. Finally, when the three previous attempts have failed, `solve`, using the `clause` metapredicate, finds a rule whose head is the goal and then calls `solve` on the body of that rule. `solve` implements the same left-to-right, depth-first, goal-directed search as the built-in Prolog interpreter.

If we assume the following simple set of assertions,

```
p(X, Y) :- q(X), r(Y).
q(X) :- s(X).
r(X) :- t(X).
s(a).
t(b).
t(c).
```

`solve` has the behavior we would expect of Prolog:

```
?- solve(p(a, b)).
Yes
?- solve(p(X, Y)).
X = a, Y = b;
X = a, Y = c;
No
?- solve(p(f, g)).
no
```

The ability easily to write meta-interpreters for a language has certain theoretical advantages. For example, McCarthy wrote a simple Lisp meta-interpreter as part of a proof that the Lisp language is Turing complete (McCarthy 1960). From a more practical standpoint, we can use meta-interpreters to extend or modify the semantics of the underlying language to better fit our application. This is the programming methodology of *meta-linguistic abstraction*, the creation of a high-level language that is designed to help solve a specific problem.

For example, we can extend the standard Prolog semantics so as to ask the user about the truth-value of any goal that does not succeed (using the four `solve` predicates above) in the knowledge base. We do this by adding the following clauses to the end of the previous definitions of `solve`:

```
solve(A) :- askuser(A).
askuser(A) :- write(A),
     write('? Enter true if the goal is true, false
          otherwise'), nl.
     read(true).
```

Because we add this definition to the end of the other `solve` rules, it is called only if all of these earlier `solve` rules fail. `solve` then calls `askuser` to query the user for the truth value of the goal (`A`). `askuser` prints the goal and instructions for answering. `read(true)` attempts to unify the user's input with the term `true`, failing if the user enters `false` (or anything that does not unify with `true`). In this way we have changed the semantics of `solve` and extended the behavior of Prolog. An example, using the simple knowledge base defined above, illustrates the behavior of the augmented definition of `solve`:

```
?- solve(p(f, g)).
s(f)? Enter true if the goal is true, false
otherwise
true.
t(g)? Enter true if the goal is true, false
otherwise
true.
yes
```

Another extension to the meta-interpreter allows it to respond to "why" queries. When the interpreter asks the user a question, the user can respond with `why`; the appropriate response to this query is the current rule that the program is trying to solve. We implement this by storing the stack of rules in the current line of reasoning as the second parameter to `solve`. Whenever `solve` calls `clause` to solve a goal, it places the selected rule on the stack. Thus, the rule stack records the chain of rules from the top-level goal to the current subgoal.

Because the user may now enter two valid responses to a query, `askuser` calls `respond`, which either succeeds if the user enters `true` (as before) or prints the top rule on the stack if the user enters `why`. `respond` and `askuser` are mutually recursive, so that after printing the answer to a `why` query, `respond` calls `askuser` to query the user about the goal again. Note, however, that `respond` calls `askuser` with the tail of the rule stack. Thus, a series of `why` queries will simply chain back up the rule stack until the stack is empty – the search is at the root node of the tree – letting the user trace the entire line of reasoning.

```
solve(true, _) :-!.
solve(not(A), Rules) :- not(solve(A, Rules)).
solve((A, B), Rules) :- !,
     solve(A, Rules), solve(B, Rules).
solve(A, Rules) :-
     clause(A, B), solve(B, [(A :- B) | Rules]).
solve(A, Rules) :- askuser(A, Rules).
askuser(A, Rules) :-
     write(A),
     write('? Enter true if goal is true,
          false otherwise'),nl,
     read(Answer), respond(Answer, A, Rules).
```

```
respond(true, _, _).
respond(why, A, [Rule | Rules]) :-
     write(Rule), nl,
     askuser(A, Rules).
respond(why, A, [ ]) :- askuser(A, [ ]).
```

For example, suppose we run `solve` on the simple database introduced earlier in the section:

```
?- solve(p(f, g), [ ]).
s(f)? Enter true if goal is true, false otherwise
why.
q(f) :- s(f)
s(f)? Enter true if goal is true, false otherwise
why.
p(f,g) :- (q(f), r(g))
s(f)? Enter true if goal is true, false otherwise
true.
t(g)? Enter true if goal is true, false otherwise
true.
yes
```

Note how successive `why` queries actually trace back up the full line of reasoning.

A further useful extension to the `solve` predicate constructs a proof tree for any successful goal. The ability to build proof trees provides expert system shells with the means of responding to "how" queries; it is also important to any algorithm, such as explanation-based learning (Chapter 7), that reasons about the results of a problem solver.

We may modify the pure Prolog interpreter to build a proof tree recursively for a goal as it solves that goal. In the definition that follows, the proof is returned as the second parameter of the `solve` predicate. The proof of the atom `true` is that atom; this halts the recursion. In solving a goal `A` using a rule `A :- B`, we construct the proof of `B` and return the structure (`A :- ProofB`). In solving a conjunction of two goals, `A` and `B`, we simply conjoin the proof trees for each goal: (`ProofA, ProofB`).

The definition of a meta-interpreter that supports the construction of the proof trees is:

```
solve(true, true) :-!.
solve(not(A), not ProofA) :-
     not(solve(A, ProofA)).
solve((A, B),(ProofA, ProofB)) :-
     solve(A, ProofA), solve(B, ProofB).
solve(A, (A :- ProofB)) :-
     clause(A, B), solve(B, ProofB).
solve(A, (A :- given)) :-
askuser(A).
```

```
askuser(A, Proof) :-
     write(A),
     write('enter true if goal is true,
          false otherwise'),
     read(true).
```

Running this on our simple database gives the results:

```
?- solve(p(a, b), Proof).
Proof = p(a, b) :-
       ((q(a) :-
            (s(a) :-
                 true)),
       r(b) :-
            (t(b) :-
                 true)))
```

In the next section, we use these same techniques to implement an expert system shell. `exshell` uses a knowledge base in the form of rules to solve problems. It asks the user for needed information, keeps a record of case-specific data, responds to `how` and `why` queries, and implements the Stanford certainty factor algebra (Luger 2009, Section 9.2.1). Although this program, `exshell`, is much more complex than the Prolog meta-interpreters discussed above, it is just an extension of this methodology. Its heart is a `solve` predicate implementing a back-chaining search.

6.2 A Shell for a Rule-Based Expert System

EXSHELL

In this section we present the key predicates used in the design of an interpreter for a goal-driven, rule-based expert system. At the end of this section, we demonstrate the performance of `exshell` using an automotive diagnostic knowledge base. If the reader would prefer to read through this trace before examining `exshell`'s key predicates, we encourage looking ahead.

An `exshell` knowledge base consists of rules and specifications of queries that can be made to the user. Rules are represented using a two-parameter `rule` predicate of the form `rule(R, CF)`. The first parameter is an assertion to the knowledge base, written using standard Prolog syntax. Assertions may be Prolog rules, of the form (`G :- P`), where `G` is the head of the rule and `P` is the conjunctive pattern under which `G` is true. The first argument to the rule predicate may also be a Prolog fact. `CF` is the confidence the designer has in the rule's conclusions. `exshell` implements the certainty factor algebra of MYCIN, (Luger 2009, Section 9.2.1), and we include a brief overview of the Stanford algebra here. Certainty factors (`CF`s) range from +100, a fact that is true, to –100, something that is known to be false. If the `CF` is around 0, the truth value is unknown. Typical rules from a knowledge base for diagnosing automotive failures are:

```
rule((bad_component(starter) :-
     (bad_system(starter_system),
          lights(come_on))), 50).
rule(fix(starter, 'replace starter'),100).
```

Thts first rule states that if the bad system is shown to be the starter system and the lights come on, then conclude that the bad component is the starter, with a certainty of 50. Because this rule contains the symbol `:-` it must be surrounded by parentheses. The second rule asserts the fact that we may fix a broken starter by replacing it, with a certainty factor of 100. `exshell` uses the `rule` predicate to retrieve those rules that conclude about a given goal, just as the simpler versions of `solve` in Section 6.1 used the built-in `clause` predicate to retrieve rules from the global Prolog database.

`exshell` supports user queries for unknown data. However, because we do not want the interpreter to ask for every unsolved goal, we allow the programmer to specify exactly what information may be obtained from asking. We do this with the `askable` predicate:

```
askable(car_starts).
```

Here `askable` specifies that the interpreter may ask the user for the truth of the `car_starts` goal when nothing is known or can be concluded about that goal.

In addition to the programmer-defined knowledge base of rules and askables, `exshell` maintains its own record of case-specific data. Because the shell asks the user for information, it needs to remember what it has been told; this prevents the program from asking the same question twice during a consultation (decidedly non-expert behavior!).

The heart of the `exshell` meta-interpreter is a predicate of four arguments called, surprisingly, `solve`. The first of these arguments is the goal to be solved. On successfully solving the goal, `exshell` binds the second argument to the (accumulated) confidence in the goal as computed from the knowledge base. The third argument is the rule stack, used in responding to `why` queries, and the fourth is the cutoff threshold for the certainty factor algebra. This allows pruning of the search space if the confidence falls below a specified threshold.

In attempting to satisfy a goal, `G`, `solve` first tries to match `G` with any facts that it already has obtained from the user. We represent known facts using the two-parameter `known(A, CF)` predicate. For example, `known(car_starts, 85)` indicates that the user has already told us that the car starts, with a confidence of `85`. If the goal is unknown, `solve` attempts to solve the goal using its knowledge base. It handles the negation of a goal by solving the goal and multiplying the confidence in that goal by –1. It solves conjunctive goals in left-to-right order. If `G` is a positive literal, `solve` tries any rule whose head matches `G`. If this fails, solve queries the user. On obtaining the user's confidence in a goal, `solve` asserts this information to the database using a `known` predicate.

```
% Case 1: truth value of goal is already known
solve(Goal, CF, _, Threshold) :
     known(Goal, CF), !,
     above_threshold(CF, Threshold).
```

```
% Case 2: negated goal
solve(not(Goal), CF, Rules, Threshold) :-!,
     invert_threshold(Threshold, New_threshold),
     solve(Goal, CF_goal, Rules, New_threshold),
     negate_cf(CF_goal, CF).
% Case 3: conjunctive goals
solve((Goal_1,Goal_2), CF, Rules, Threshold) :- !,
     solve(Goal_1, CF_1, Rules, Threshold),
     above_threshold(CF_1, Threshold),
     solve(Goal_2, CF_2, Rules, Threshold),
     above_threshold(CF_2, Threshold),
     and_cf(CF_1, CF_2, CF).
% Case 4: back chain on a rule in knowledge base
solve(Goal, CF, Rules, Threshold) :-
     rule((Goal :- (Premise)), CF_rule),
     solve(Premise, CF_premise, [rule((Goal :-
            Premise), CF_rule)|Rules], Threshold),
     rule_cf(CF_rule, CF_premise, CF),
     above_threshold(CF, Threshold).
% Case 5: fact assertion in knowledge base
solve(Goal, CF, _, Threshold) :-
     rule(Goal, CF),
     above_threshold(CF, Threshold).
% Case 6: ask user
solve(Goal, CF, Rules, Threshold) :-
     askable(Goal),
     askuser(Goal, CF, Rules), !,
     assert(known(Goal, CF)),
     above_threshold(CF, Threshold).
```

We start a consultation using a two-argument version of `solve`. The first argument is the top-level goal in the knowledge base, and the second is a variable that will be bound to the confidence in the goal's truth as inferred from the knowledge base. `solve/2` (`solve` with arity of 2) prints a set of instructions to the user, calls `retractall(known(_,_))` to clean up any residual information from previous uses of `exshell`, and calls `solve/4` initialized with appropriate values:

```
solve(Goal, CF) :-
     print_instructions,
     retractall(known(_, _)),
     solve(Goal, CF, [ ], 20).
```

`print_instructions` gives allowable responses to an exshell query:

```
print_instructions :- nl,
      write('Response must be either:'), nl,
      write('Confidence in truth of query.'), nl,
```

```
        write('A number between –100 and 100.'), nl,
        write('why.'), nl,
        write('how(X), where X is a goal'), nl.
```

The next set of predicates computes certainty factors. Again, `exshell` uses a form of the Stanford certainty factor algebra. Briefly, the certainty factor of the `and` of two goals is the minimum of the certainty factors of the individual goals; the certainty factor of the negation of a fact is –1 times the certainty of that fact. Confidence in a fact concluded using a rule equals the certainty of the premise multiplied by the certainty factor in the rule. `above_threshold` determines whether the value of a certainty factor is too low given a particular `threshold`. `exshell` uses the `threshold` value to prune a goal if its certainty gets too low.

Note that we define `above_threshold` separately for negative and positive values of the threshold. A positive `threshold` enables us to prune if the goal's confidence is less than `threshold`. However, a negative `threshold` indicates that we are trying to prove a goal false. Thus for negative goals, we prune search if the value of the goal's confidence is greater than the `threshold`. `invert_threshold` is called to multiply `threshold` by –1.

```
and_cf(A, B, A) :- A = < B.

and_cf(A, B, B) :- B < A.

negate_cf(CF, Negated_CF) :-
     Negated_CF is - 1 * CF.

rule_cf(CF_rule, CF_premise,CF) :-
     CF is (CF_rule * CF_premise/100).

above_threshold(CF, T) :-
     T > = 0, CF >= T.

above_threshold(CF, T) :-
     T < 0, CF =< T.

invert_threshold(Threshold, New_threshold) :-
     New_threshold is –1 * Threshold.
```

`askuser` writes out a query and reads the user's `Answer`; the `respond` predicates take the appropriate action for each user input.

```
askuser(Goal, CF, Rules) :-
     nl, write('User query:'),
     write(Goal), nl, write('?'),
     read(Answer),
     respond(Answer, Goal, CF, Rules).
```

The user can respond to this query with a CF between 100 and –100, for confidence in the goal's truth, `why` to ask why the question was asked, or `how(X)` to inquire how result `X` was established. The response to `why` is the rule currently on top of the rule stack. As with our implementation of Prolog in Prolog in Section 6.1, successive `why` queries will pop back up the rule stack, enabling the user to reconstruct the entire line of reasoning. If the user answer matches `how(X)`, `respond` calls `build_proof` to

build a proof tree for `X` and `write_proof` to print that proof in a readable form. There is a "catchall" `respond` for unknown input values.

```
% Case 1: user enters a valid confidence factor
respond(CF, _, CF, _) :-
     number(CF),
     CF =< 100, CF >= –100.
% Case 2: user enters a why query
respond(why, Goal, CF, [Rule | Rules]) :-
     write_rule(Rule),
     askuser(Goal, CF, Rules).
respond(why, Goal, CF, [ ]) :-
     write('Back to top of rule stack.'),
     askuser(Goal, CF, [ ]).
% Case 3: user enters a how query. Build/print proof
respond(how(X), Goal, CF, Rules) :-
     build_proof(X, CF_X, Proof), !,
     write(X), write('concluded with certainty'),
     write(CF_X), nl, nl,
     write('The proof is '), nl, nl,
     write_proof(Proof, 0), nl, nl,
     askuser(Goal, CF, Rules).
% User enters how query, could not build proof
respond(how(X), Goal, CF, Rules) :-
     write('The truth of '), write(X), nl,
     write('is not yet known.'), nl,
     askuser(Goal, CF, Rules).
% Case 4: User presents unrecognized input
respond(_, Goal, CF, Rules) :-
     write('Unrecognized response.'), nl,
     askuser(Goal, CF, Rules).
```

`build_proof` is parallel to `solve/4`, but `build_proof` does not ask the user for unknowns, as these were already saved as part of the case-specific data. `build_proof` constructs a proof tree as it proves the goal.

```
build_proof(Goal, CF, (Goal, CF :- given)) :-
     known(Goal, CF), !.
build_proof(not Goal, CF, not Proof) :- !,
     build_proof(Goal, CF_goal, Proof),
     negate_cf(CF_goal, CF).
build_proof((Goal_1, Goal_2), CF,
          (Proof_1, Proof_2)) :- !,
     build_proof(Goal_1, CF_1, Proof_1),
     build_proof(Goal_2, CF_2, Proof_2),
     and_cf(CF_1, CF_2, CF).
```

```
build_proof(Goal, CF, (Goal, CF :- Proof)) :-
     rule((Goal :- Premise), CF_rule),
     build_proof(Premise, CF_premise, Proof),
     rule_cf(CF_rule, CF_premise, CF).
build_proof(Goal, CF, (Goal, CF :- fact)) :-
     rule(Goal, CF).
```

The final predicates create a user interface. The interface requires the bulk of the code! First, we define a predicate `write_rule:`

```
write_rule(rule((Goal :- (Premise)), CF)) :-
     write(Goal), write(':-'), nl,
     write_premise(Premise), nl,
     write('CF = '), write(CF), nl.
write_rule(rule(Goal, CF)) :-
     write(Goal), nl, write('CF = '), write(CF), nl.
```

`write_premise` writes the conjuncts of a rule premise:

```
write_premise((Premise_1, Premise_2)) :- !,
     write_premise(Premise_1),
     write_premise(Premise_2).
write_premise(not Premise) :- !,
     write(''), write(not), write(''),
     write(Premise), nl.
write_premise(Premise) :-
     write(''), write(Premise), nl.
```

`write_proof` prints proof, using indents to show the tree's structure:

```
write_proof((Goal, CF :- given), Level) :-
     indent(Level), write(Goal), write(' CF= '),
     write(CF),write(' given by the user'), nl, !.
write_proof((Goal, CF :- fact), Level) :-
     indent(Level), write(Goal), write(' CF = '),
     write(CF),
     write(' was a fact of knowledge base'), nl, !.
write_proof((Goal, CF :- Proof), Level) :-
     indent(Level), write(Goal), write(' CF = '),
     write(CF), write(' :-'), nl, New_level is
          Level + 1,write_proof(Proof,New_level), !.
write_proof(not Proof, Level) :-
     indent(Level), write((not)), nl,
     New_level is Level + 1,
     write_proof(Proof, New_level), !.
write_proof((Proof_1, Proof_2),Level) :-
     write_proof(Proof_1, Level),
     write_proof(Proof_2, Level), !.

indent(0).
```

```
indent(l) :-
     write(''), l_new is l – 1, indent(l_new).
```

As an illustration of the behavior of `exshell`, consider the following sample knowledge base for diagnosing car problems. The top-level goal is fix/1. The knowledge base decomposes the problem solution into finding the `bad_system`, finding the `bad_component` within that system, and finally linking the diagnosis to `Advice` for its solution. Note that the knowledge base is incomplete; there are sets of symptoms that it cannot diagnose. In this case, `exshell` simply fails. Extending the knowledge base to some of these cases and adding a rule that succeeds if all other rules fail are interesting challenges and left as exercises. The following set of rules is segmented to show reasoning on each level of the search tree presented in Figure 6.1. The top segment, `rule((fix(Advice),` is at the root of the tree:

```
rule((fix(Advice) :-                % Top-level query
     (bad_component(X), fix(X,Advice))), 100).
rule((bad_component(starter) :-
     (bad_system(starter_system),
          lights(come_on))), 50).
rule((bad_component(battery) :-
     (bad_system(starter_system),
          not lights(come_on))), 90).
rule((bad_component(timing) :-
     (bad_system(ignition_system),
          not tuned_recently)), 80).
rule((bad_component(plugs) :-
     (bad_system(ignition_system),
          plugs(dirty))), 90).
rule((bad_component(ignition_wires) :-
     (bad_system(ignition_system),
          not plugs(dirty), tuned_recently)), 80).
rule((bad_system(starter_system) :-
     (not car_starts, not turns_over)), 90).
rule((bad_system(ignition_system) :-
     (not car_starts, turns_over, gas_in_carb)),80).
rule((bad_system(ignition_system) :-
     (runs(rough), gas_in_carb)), 80).
rule((bad_system(ignition_system) :-
     (car_starts, runs(dies), gas_in_carb)), 60).

rule(fix(starter, 'replace starter'), 100).
rule(fix(battery, 'replace/recharge battery'), 100).
rule(fix(timing, 'get the timing adjusted'), 100).
rule(fix(plugs, 'replace spark plugs'), 100).
rule(fix(ignition_wires, 'check ignition'),100).

askable(car_starts).       % May ask user about goal
```

```
askable(turns_over).
askable(lights(_)).
askable(runs(_)).
askable(gas_in_carb).
askable(tuned_recently).
askable(plugs(_)).
```

Next we demonstrate, `exshell` using this knowledge base. Figure 6.1 presents the trace and the search space: solid lines are searched, dotted lines are not searched, and bold lines indicate the solution.

```
?- solve(fix(X), CF).
Response must be either:
    A confidence in the truth of the query.
    This is a number between –100 and 100.
    why.
    how(X), where X is a goal
User query:car_starts
? –100.
User query:turns_over
? 85.
User query:gas_in_carb
? 75.
User query:tuned_recently
? –90.
    X = 'get the timing adjusted' CF = 48.0
```

We now run the problem again using `how` and `why` queries. Compare the responses with the corresponding subtrees and search paths of Figure 6.1:

?- solve(fix(X), CF).

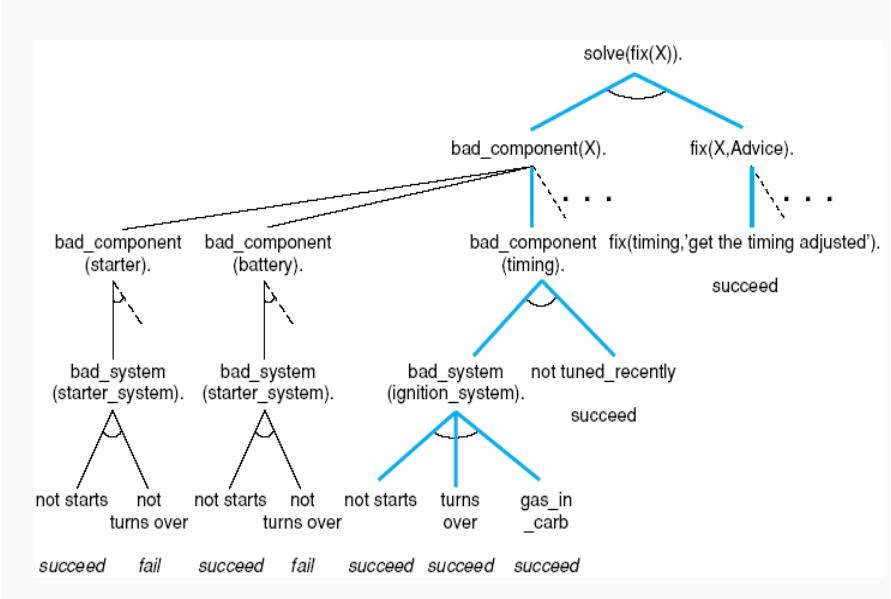

Figure 6.1. The graph searched in an automotive diagnosis consultation; dashed lines are branches not examined, bold lines indicate the solution.

```
Response must be either:
     A confidence in the truth of the query.
     This is a number between –100 and 100.
     why.
     how(X), where X is a goal
User query:car_starts
? –100.
User query:turns_over
? why.
bad_system(starter_system):-
     not car_starts
     not turns_over
     CF = 90
User query:turns_over
? why.
bad_component(starter):-
     bad_system(starter_system)
     lights(come_on)
     CF = 50
User query:turns_over
? why.
fix(_0):-
     bad_component(starter)
     fix(starter,_0)
     CF = 100
User query:turns_over
? why.
Back to top of rule stack.
User query:turns_over
? 85.
User query:gas_in_carb
? 75.
User query:tuned_recently
? why.
bad_component(timing):-
     bad_system(ignition_system)
     not tuned_recently
     CF = 80
User query:tuned_recently
? how(bad_system(ignition_system)).
bad_system(ignition_system) was concluded with
          certainty 60.0
The proof is
bad_system(ignition_system) CF= 60.0 :-
     not car_starts CF = –100 was given by the user
     turns_over CF = 85 was given by the user
     gas_in_carb CF = 75 was given by the user
User query:tuned_recently
? –90.
X = 'get the timing adjusted' CF = 48.0
```

6.3 A Prolog Planner

For the third meta-interpreter of Chapter 6 we present a predicate calculus-based planning algorithm. In many ways this approach to planning is similar to early work in planning at SRI-International (Fikes and Nilsson 1971, Fikes et al. 1972). Our planner is predicate calculus based in that the PC representation is used to describe both the states of the planning world (the state descriptions) as well as the rules for changing the state of the world. In this section we create a Prolog version of that algorithm.

We represent the states of the world, including the `start` and `goal`, as lists of predicates that have interpretations as states of the world. Thus, the `start` and `goal` states are each described as a list of predicates:

```
start = [handempty, ontable(b), ontable(c), on(a,b),
          clear(c), clear(a)]

goal = [handempty, ontable(a), ontable(b), on(c,b),
          clear(a), clear(c)]
```

These states are seen, with a portion of the search space, in Figure 6.2.

The moves in this blocks world are described using an *add* and *delete* list. The add and delete list describes how the list of predicates describing a new state of the solution is created from the list describing the previous state: some predicates are added to the state list and others are deleted. The `move` predicates for state change have three arguments. First is the `move` predicate name with its arguments. The second argument is the list of preconditions: the predicates that must be true of the description of the present state of the world for the `move` rule to be applied to that state. The third argument is the list containing the add and delete predicates: the predicates that are added to and/or deleted from the state of the world to create the new state of the world that results from applying the `move` rule. Notice how useful the ADT set operators of union, intersection, set difference, etc., are in manipulating the preconditions and the predicates in the add and delete list.

Four of the moves within this blocks world may be described:

```
move(pickup(X), [handempty, clear(X), on(X,Y)],
   [del(handempty), del(clear(X)), del(on(X,Y)),
        add(clear(Y)), add(holding(X))]).

move(pickup(X), [handempty, clear(X), ontable(X)],
     [del(handempty), del(clear(X)),
        del(ontable(X)), add(holding(X))]).

move(putdown(X), [holding(X)],
     [del(holding(X)), add(ontable(X)),
        add(clear(X)), add(handempty)]).

move(stack(X,Y), [holding(X), clear(Y)],
     [del(holding(X)), del(clear(Y)),
        add(handempty),add(on(X,Y)),add(clear(X))]).
```

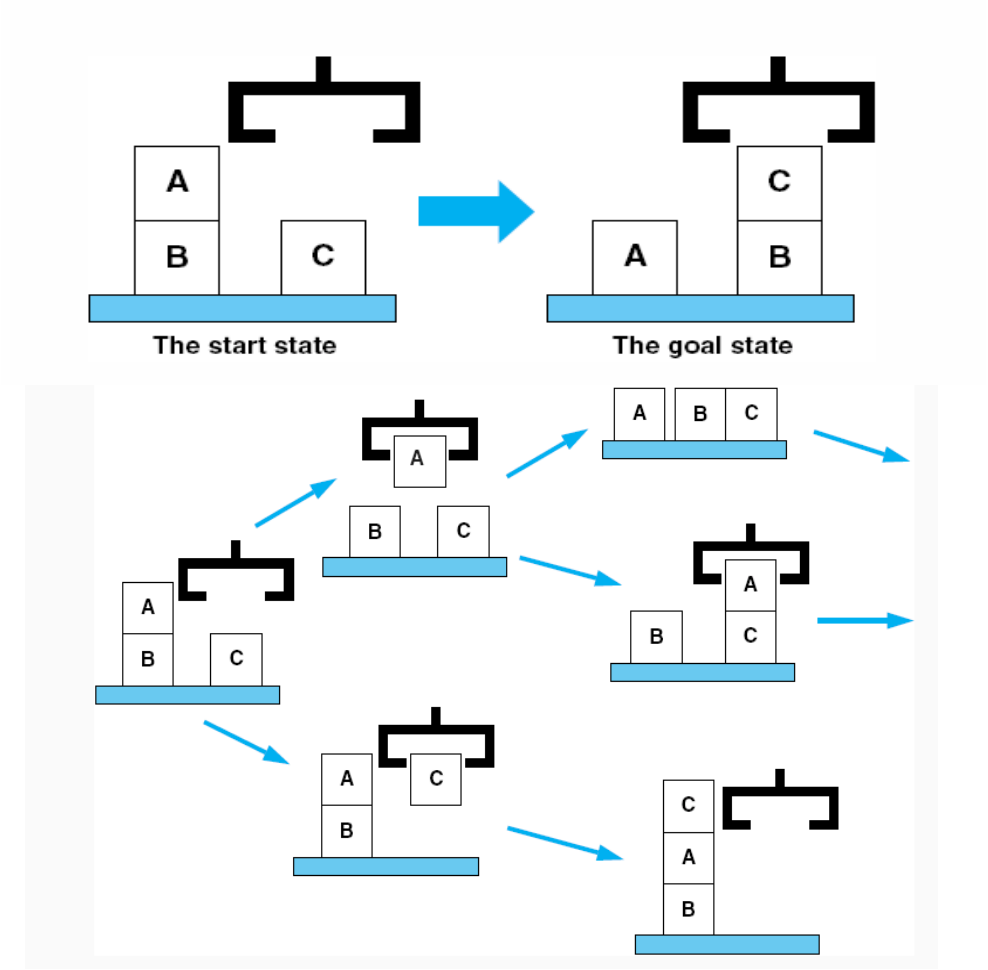

Figure 6.2. The start and goal states along with the initial portion of the search space for the blocks world planner.

Finally, we present the recursive controller for the plan generation. The first plan predicate gives the successful termination conditions (goal state description) for the plan when the `Goal` is produced. The final plan predicate states that after exhaustive search, no plan is possible. The recursive plan generator:

1. Searches for a `move` relationship.

2. Checks, using the `subset` operator, whether the state's `Preconditions` are met.

3. The `change_state` predicate produces a new `Child_state` using the add and delete list.
`member_stack` makes sure the new state has not been visited before.

4. The `stack` operator pushes the new `Child_state` onto the `New_move_stack`.

5. The `stack` operator pushes the original `Name` state onto the `New_been_stack`.

6. The recursive `plan` call searches for the next state using the `Child_state` and an updated `New_move_stack` and `Been_stack`.

A number of supporting utilities, built on the stack and set ADTs of Section 3.3 are included. Of course, the search being stack-based, is depth-first with backtracking and terminates with the first path found to a goal. It is left as an exercise to build other search strategies for planning, e.g., breadth-first and best-first planners.

```
plan(State, Goal, _, Move_stack) :-
     equal_set(State, Goal),
     write('moves are'), nl,
     reverse_print_stack(Move_stack).
plan(State, Goal, Been_stack, Move_stack) :-
     move(Name, Preconditions, Actions),
     conditions_met(Preconditions, State),
     change_state(State, Actions, Child_state),
     not(member_stack(Child_state, Been_stack)),
     stack(Name, Been_stack, New_been_stack),
     stack(Child_state, Move_stack, New_move_stack),
     plan(Child_state, Goal, New_been_stack,
          New_move_stack), !.
plan(_, _, _) :-
     write('No plan possible with these moves!').
conditions_met(P, S) :-
     subset(P, S).
change_state(S, [ ], S).
change_state(S, [add(P) | T], S_new) :-
     change_state(S, T, S2),
     add_if_not_in_set(P, S2, S_new), !.
change_state(S, [del(P) | T], S_new) :-
     change_state(S, T, S2),
     delete_if_in_set(P, S2, S_new), !.
reverse_print_stack(S) :-
     empty_stack(S).
reverse_print_stack(S) :-
     stack(E, Rest, S),
     reverse_print_stack(Rest), write(E), nl.
```

Finally, we create a `go` predicate to initialize the arguments for `plan`, as well as a `test` predicate to demonstrate an easy method to save repeated creation of the same input string.

```
go(Start, Goal) :-
     empty_stack(Move_stack),
     empty_stack(Been_stack),
     stack(Start, Been_stack, New_been_stack),
     plan(Start, Goal, New_been_stack, Move_stack).

test :-
go(
     [handempty, ontable(b), ontable(c),
          on(a,b), clear(c), clear(a)],
     [handempty, ontable(a), ontable(b), on(c,b),
          clear(a), clear(c)]
  ).
```

In Chapter 7 we present two machine learning algorithms in Prolog, *version space search* and *explanation based learning*.

Exercises

1. Extend the meta-interpreter for Prolog in Prolog (Section 6.1) to include `or` and the cut.

2. Further complete the rules used with the `exshell` cars example in the text. You might add several new sets of rules for the transmission, cooling system, and brakes.

3. Create a knowledge base for a new domain for the expert system `exshell`.

4. `exshell` currently allows the user to respond to queries by entering a confidence in the query's truth, a why query, or a how query. Extend the `respond` predicate to allow the user to answer with `y` if the query is true, n if it is false. These responses correspond to having certainty factors of 100 and -100.

5. As currently designed, if `exshell` cannot solve a goal using the rule base, it fails. Extend `exshell` so if it cannot prove a goal using the rules, and if it is not `askable`, it will call that goal as a Prolog query. Adding this option requires changes to both the `solve` and `build_proof` predicates.

6. Add a predicate that that exshell does not just fail if it cannot find a solution recommendation. This could be a `solve` predicate at the very end of all `solve` predicates that prints out some message about the state of the problem solving, perhaps by binding `X`, and linking it to some `Advice`, and then succeeds. This an important consideration, guaranteeing that `exshell` terminates gracefully.

7. Finish the code for the planner of Section 6.3. Add code for a situation that requires a new set of moves and has new objects in the domain, such as adding pyramids or spheres that cannot be stacked on.

8. Add appropriate predicates and ADTs to `plan` to implement a breadth-first search controller for the planner of Section 6.3.

9. Design a best-first search controller for the planner of Section 6.3. Add heuristics to the search of your planning algorithm. Can you specify a heuristic that is admissible (Luger 2009, Section 4.3)?