4 Depth-, Breadth-, and Best-First Search Using the Production System Design Pattern

Chapter Objectives

A production system was defined and examples given:
Production rule sets
Control strategies
A production system written in Prolog was presented:
A rule set and control strategy for the Farmer Wolf, Goat, and Cabbage problem
Search strategies for the production system created using Abstract Data Types
Depth-first search and the stack operators
Breadth-first search and the queue operators
Best first search and the priority queue operators
Sets were used for the closed list in all searches

Chapter Contents

4.1 Production System Search in Prolog
4.2 A Production System Solution to the Farmer, Wolf, Goat, Cabbage Problem
4.3 Designing Alternative Search Strategies

4.1 Production System Search in Prolog

The Production System

The *production system* (Luger 2009, Section 6.2) is a model of computation that has proved particularly important in AI, both for implementing search algorithms and for modeling human problem solving behavior. A production system provides pattern-directed control of a problem-solving process and consists of a set of *production rules*, a *working memory*, and a *recognize–act* control cycle.

A *production system* is defined by:

The set of production rules. These are often simply called *productions*. A production is a *condition–action* pair and defines a single chunk of problem-solving knowledge. The *condition part* of the rule is a pattern that determines when that rule may be applied by matching data in the working memory. The *action part* of the rule defines the associated problem-solving step.

Working memory contains a description of the *current state of the world* in a reasoning process. This description is a pattern that, in *data-driven reasoning*, is matched against the condition part of a production to select appropriate problem-solving actions. The actions of production rules are specifically designed to alter the contents of working memory, leading to the next phase of the recognize-act cycle.

The recognize–act cycle. The control structure for a production system is simple: *working memory* is initialized with the beginning problem description. The current state of the problem solving is maintained as a set of patterns in working memory. These patterns are matched against the conditions of the

production rules; this produces a subset of the production rules, called the *conflict set*, whose conditions match the patterns in working memory. One of the productions in the conflict set is then selected (*conflict resolution*) and the production is *fired*. After the selected production rule is fired, the control cycle repeats with the modified working memory. The process terminates when the contents of working memory do not match any rule conditions.

Conflict resolution chooses a rule from the conflict set for firing. Conflict resolution strategies may be simple, such as selecting the first rule whose condition matches the state of the world, or may involve complex rule selection heuristics. The *pure* production system model has no mechanism for recovering from dead ends in the search; it simply continues until no more productions are enabled and halts. Many practical implementations of production systems allow backtracking to a previous state of working memory in such situations. A schematic drawing of a production system is presented in Figure 4.1.

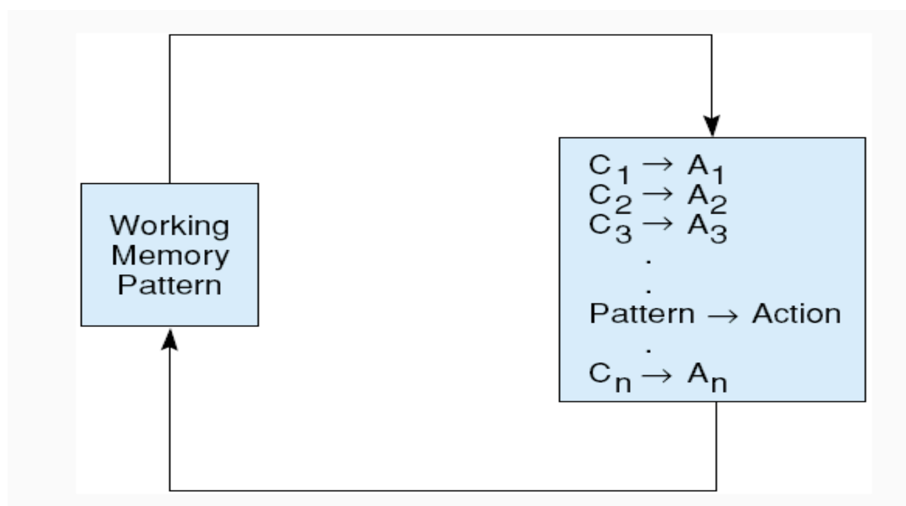

Figure 4.1. The production system. Control loops from the working memory through the production rules until no rule matches a working memory pattern.

Example 4.1: The Knight's Tour Revisited

The 3 x 3 knight's tour problem may be solved with a production system, Figure 4.1. Each move can be represented as a rule whose condition is the location of the knight on a particular square and whose action moves the knight to another square. Sixteen productions, presented in Table 4.1, represent all possible moves of the knight.

We next specify a recursive procedure to implement a control algorithm for the production system. We will use the recursive path algorithm of Section 3.1, where the third argument of the path predicate is the list of already visited states. Because `path(Z, Z, L)` will unify only with predicates whose first two arguments are identical, such as `path(3, 3, _)` or `path(5, 5, _)`, it defines the desired terminating condition. If `path(X, X, L)` does not succeed we look at the production rules for a next state and then recur.

RULE #	CONDITION	ACTION
1	knight on square 1	move knight to square 8
2	knight on square 1	move knight to square 6
3	knight on square 2	move knight to square 9
4	knight on square 2	move knight to square 7
5	knight on square 3	move knight to square 4
6	knight on square 3	move knight to square 8
7	knight on square 4	move knight to square 9
8	knight on square 4	move knight to square 3
9	knight on square 6	move knight to square 1
10	knight on square 6	move knight to square 7
11	knight on square 7	move knight to square 2
12	knight on square 7	move knight to square 6
13	knight on square 8	move knight to square 3
14	knight on square 8	move knight to square 1
15	knight on square 9	move knight to square 2
16	knight on square 9	move knight to square 4

Table 4.1. Production rules for the 3 x 3 knight tour problem.

The general recursive path definition is given by two predicate calculus formulas:

```
path(Z, Z, L).
path(X, Y, L) :-
     move(X, Z), not(member(Z, L)),
     path(Z, Y, [Z | L]).
```

Working memory, represented by the parameters of the recursive `path` predicate, contains both the current board state and the goal state. The control regime applies rules until the current state equals the goal state and then halts. A simple conflict resolution scheme would fire the first rule that did not cause the search to loop. Because the search may lead to dead ends (from which every possible move leads to a previously visited state and thus a loop), the control regime must also allow backtracking; an execution of this production system that determines whether a path exists from square 1 to square 2 is charted in Table 4.2.

Production systems are capable of generating infinite loops when searching a state space graph. These loops are particularly difficult to spot in a production system because the rules can fire in any order. That is, looping may appear in the execution of the system, but it cannot easily be found from a syntactic inspection of the rule set.

LOOP	CURRENT	GOAL	CONFLICT RULES	USE RULE
0	1	2	1, 2	1
1	8	2	13, 14	13
2	3	2	5, 6	5
3	4	2	7, 8	7
4	9	2	15, 16	15
5	2	2	No Rules Match	Halt

Table 4.2. The iterations of the production system finding a path from square 1 to square 2.

For example, with the "move" rules of the knight's tour problem ordered as in Table 4.1 and a conflict resolution strategy of selecting the first match, the pattern `move(2, X)` would match with `move(2, 9)`, indicating a move to square 9. On the next iteration, the pattern `move(9, X)` would match with `move(9, 2)`, taking the search back to square 2, causing a loop. The `not(member(Z, L))` will check the list of visited states. The actual conflict resolution strategy was therefore: *select the first matching move that leads to an unvisited state*. In a production system, the proper place for recording such case-specific data as a list of previously visited states is not a global closed list but within the working memory itself, as we see in the next sections where the parameters of the `path` call make up the content of working memory.

4.2 A Production System Solution to the FWGC Problem

Example 4.2: The Farmer, Wolf, Goat, and Cabbage Problem

In Section 4.1 we described the production system and demonstrated a simple depth-first search for the restricted Knight's Tour problem. In this section we write a production system solution to the farmer, wolf, goat, and cabbage (FWGC) problem. In Section 4.3 we use the simple abstract data types created in Chapter 3 to create depth-, breadth-, and best-first solutions for production system problems. The FWGC problem is stated as follows:

> A farmer with his wolf, goat, and cabbage come to the edge of a river they wish to cross. There is a boat at the river's edge, but, of course, only the farmer can row. The boat also can carry only two things (including the rower) at a time. If the wolf is ever left alone with the goat, the wolf will eat the goat; similarly, if the goat is left alone with the cabbage, the goat will eat the cabbage. Devise a sequence of crossings of the river so that all four characters arrive safely on the other side of the river.

We now create a production system solution to this problem. First, we observe that the problem may be represented as a search through a graph. To do this we consider the possible moves that might be available at any time in the solution process. Some of these moves are eventually ruled out because they produce states that are unsafe (something will be eaten).

For the moment, suppose that all states are safe, and simply consider the graph of possible states. We often take this approach to problem solving,

relaxing various constraints so that we can see the general structure of the search problem. After we have described the full graph then it is often straightforward to add constraints that prohibit parts of the graph – the "illegal" states – from further exploration. The boat can be used in four ways: to carry the farmer and wolf, the farmer and goat, the farmer and cabbage, or the farmer alone. A state of the world is some combination of the characters on the two banks. Several states of the search are represented in Figure 4.2. States of the world may be represented using the predicate, `state(F, W, G, C)`, with the location of the farmer as first parameter, location of the wolf as second parameter, the goat as third, and the cabbage as fourth. We assume that the river runs "north to south" and that the characters are on either the east, `e`, or west, `w`, bank. Thus, `state(w, w, w, w)` has all characters on the west bank to start the problem.

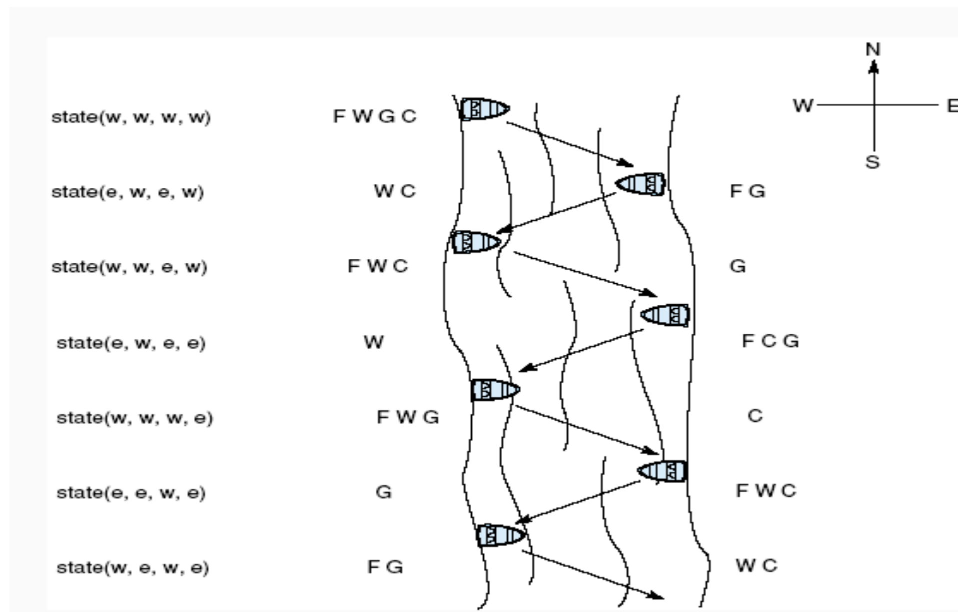

Figure 4.2. State representation and sample crossings of the F, W, G, C problem.

It must be pointed out that these choices are conventions that have been arbitrarily chosen by the authors. Indeed, as researchers in AI continually point out, the selection of an appropriate representation is often the most critical aspect of problem solving. These conventions are selected to fit the predicate calculus representation in Prolog. Different states of the world are created by different crossings of the river, represented by changes in the values of the parameters of the `state` predicate, as in Figure 4.2. Other representations are certainly possible.

We next describe a general graph for this river-crossing problem. For the time being, we ignore the fact that some states are unsafe. In Figure 4.3 we see the beginning of the graph of possible moves back and forth across the river. Since the farmer always rows, it is not necessary to have a separate representation for the location of the boat. Figure 4.3 represents part of the graph that is to be searched for a solution path.

The recursive path call described in Section 4.1 provides the control mechanism for the production system search. The production rules change state in the search. We define these *if… then…* rules in Prolog form. We take a direct approach here requiring that the pattern for the present state and the pattern for the next state both be placed in the head of the Horn clause, or to the left of :-. These are the arguments to the move predicate.

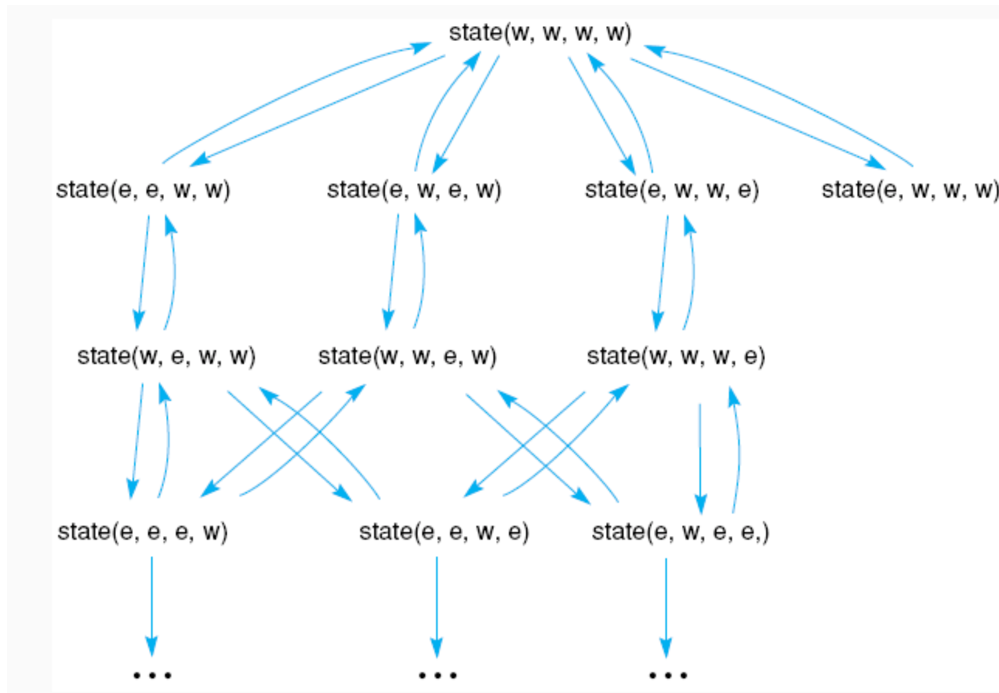

Figure 4.3. The beginning portion of the state space graph in the FWGC problem, including unsafe states.

The constraints that the production rule requires to fire and return the next state are placed to the right of :-. As shown in the following example, these conditions are expressed as unification constraints. The first rule is for the farmer to take the wolf across the river. This rule must account for both the transfer from east to west and the transfer from west to east, and it must not be applicable when the farmer and wolf are on opposite sides of the river. Thus, it must transform `state(e, e, G, C)` to `state(w, w, G, C)` and `state(w, w, G, C)` to `state(e, e, G, C)`. It must also fail for `state(e, w, G, C)` and `state(w, e, G, C)`. The variables `G` and `C` represent the fact that the third and fourth parameters can be bound to either `e` or `w`. Whatever their values, they remain the same after the move of the farmer and wolf to the other side of the river. Some of the states produced may indeed be "unsafe."

The following rule operates only when the farmer and wolf are in the same location and takes them to the opposite side of the river. Note that the goat and cabbage do not change their present location, whatever it might be.

```
move(state(X, X, G, C), state(Y, Y, G, C)) :-
     opp(X, Y).
```

```
opp(e, w).
opp(w, e).
```

This rule fires when a state (the present location in the graph) is presented to the first parameter of `move` in which the farmer and wolf are at the same location. When the rule fires, a new state, the second parameter of `move`, is produced with the value of `X` opposite, `opp`, the value of `Y`. Two conditions are satisfied to produce the new state: first, that the values of the first two parameters are the same and, second, that both of their new locations are opposite their old.

The first condition was checked implicitly in the unification process, in that `move` is not matched unless the first two parameters are the same. This test may be done explicitly by using the following rule:

```
move(state(F, W, G, C), state(Z, Z, G, C)) :-
    F = W, opp(F, Z).
```

This equivalent move rule first tests whether `F` and `W` are the same and, only if they are (on the same side of the river), assigns the opposite value of `F` to `Z`. Note that Prolog can do "assignment" by the binding of variable values in unification. Bindings are shared by all occurrences of a variable in a clause, and the scope of a variable is limited to the clause in which it occurs.

Pattern matching, a powerful tool in AI programming, is especially important in pruning search. States that do not fit the patterns in the rule are automatically pruned. In this sense, the first version of the `move` rule offers a more efficient representation because unification does not even consider the state predicate unless its first two parameters are identical.

Next, we create a predicate to test whether each new state is safe, so that nothing is eaten in the process of crossing the river. Again, unification plays an important role in this definition. Any state where the second and third parameters are the same and opposite the first parameter is `unsafe`: the wolf eats the goat. Alternatively, if the third and fourth parameters are the same and opposite the first parameter, the state is `unsafe`: the goat eats the cabbage. These `unsafe` situations may be represented with the following rules.

```
unsafe(state(X, Y, Y, C)) :- opp(X, Y).
unsafe(state(X, W, Y, Y)) :- opp(X, Y).
```

Several points should be mentioned. First, if a state is to be not unsafe (i.e., safe), according to the definition of `not` in Prolog, neither of these unsafe predicates can be true. Thus, neither of these predicates can unify with the current state or, if they do unify, their conditions are not satisfied. Second, `not` in Prolog is not exactly equivalent to the logical ~ of the first-order predicate calculus; `not` is rather "negation by failure of its opposite." The reader should test a number of states to verify that `unsafe` does what it is intended to do. Now, `not unsafe` is added to the previous production rule:

```
move(state(X, X, G, C), state(Y, Y, G, C)) :-
     opp(X, Y), not(unsafe(state(Y, Y, G, C))).
```

The `not unsafe` test calls `unsafe`, as mentioned above, to see whether the generated `state` is an acceptable new state in the search. When all criteria are met, including the check in the `path` algorithm that the new state is not a member of the visited-state list, `path` is (recursively) called on this state to go deeper into the graph. When `path` is called, the new state is added to the visited-state list.

In a similar fashion, we can create the three other production rules to represent the farmer taking the goat, cabbage, and himself across the river. We have added a `writelist` command to each production rule to print a trace of the current rule. The `reverse_print_stack` command is used in the terminating condition of `path` to print out the final solution path.

Finally, we add a fifth "pseudorule" that always fires, because no conditions are placed on it, when all previous rules have failed; it indicates that the `path` call is backtracking from the current state, and then that rule itself fails. This pseudorule is added to assist the user in seeing what is going on as the production system is running. We now present the full production system program in Prolog to solve the farmer, wolf, goat, and cabbage problem. The Prolog predicates `unsafe`, `writelist`, and the ADT stack predicates of Section 3.3.1, must also be included:

```
move(state(X, X, G, C), state(Y, Y, G, C)) :-
     opp(X, Y), not(unsafe(state(Y, Y, G, C))),
     writelist([`try farmer - wolf', Y, Y, G, C]).
move(state(X, W, X, C), state(Y, W, Y, C)) :-
     opp(X, Y), not(unsafe(state(Y, W, Y, C))),
     writelist([`try farmer - goat', Y, W, Y, C]).
move(state(X, W, G, X), state(Y, W, G, Y)) :-
     opp(X, Y), not(unsafe(state(Y, W, G, Y))),
     writelist([`try farmer - cabbage', Y, W, G, Y]).
move(state(X, W, G, C), state(Y, W, G, C)) :-
     opp(X, Y), not(unsafe(state(Y, W, G, C))),
     writelist([`try farmer by self', Y, W, G, C]).

move(state(F, W, G, C), state(F, W, G, C)) :-
     writelist([ `BACKTRACK at:', F, W, G, C]), fail.

path(Goal, Goal, Been_stack) :-
     write(`Solution Path Is: ' ), nl,
     reverse_print_stack(Been_stack).

path(State, Goal, Been_stack) :-
     move(State, Next_state),
     not(member_stack(Next_state, Been_stack)),
     stack(Next_state, Been_stack, New_been_stack),
     path(Next_state, Goal, New_been_stack), !.
opp(e, w).
opp(w, e).
```

The code is called by requesting `go`, which initializes the recursive `path` call. To make running the program easier, we can create a predicate, called `test`, that simplifies the input:

```
go(Start, Goal) :-
     empty_stack(Empty_been_stack),
     stack(Start, Empty_been_stack, Been_stack),
     path(Start, Goal, Been_stack).
test :- go(state(w,w,w,w), state(e,e,e,e)).
```

The algorithm backtracks from states that allow no further progress. You may also use `trace` to monitor the various variable bindings local to each call of `path`. It may also be noted that this program is a general program for moving the four creatures from any (legal) position on the banks to any other (legal) position, including asking for a path from the goal back to the start state. Other interesting features of production systems, including the fact that different orderings of the rules can produce different searches through the graph, are presented in the exercises. A partial trace of the execution of the F, W, G, C program, showing only rules actually used to generate new states, is presented next:

```
?- test.
try farmer takes goat e w e w
try farmer takes self w w e w
try farmer takes wolf e e e w
try farmer takes goat w e w w
try farmer takes cabbage e e w e
try farmer takes wolf w w w e
try farmer takes goat e w e e
    BACKTRACK from e,w,e,e
    BACKTRACK from w,w,w,e
try farmer takes self w e w e
try farmer takes goat e e e e
Solution Path Is:
state(w,w,w,w)
state(e,w,e,w)
state(w,w,e,w)
state(e,e,e,w)
state(w,e,w,w)
state(e,e,w,e)
state(w,e,w,e)
state(e,e,e,e)
```

In summary, this Prolog program implements a production system solution to the farmer, wolf, goat, and cabbage problem. The `move` rules make up the content of the production memory. The working memory is represented by the arguments of the `path` call. The production system control mechanism is defined by the recursive `path` call. Finally, the ordering of

rules for generation of children from each state (conflict resolution) is determined by the order in which the rules are placed in the production memory. We next present depth-, breadth-, and best-first search algorithms for production system based graph search.

4.3 Designing Alternative Search Strategies

As the previous subsection demonstrated, Prolog itself uses depth-first search with backtracking. We now show how alternative search strategies can be implemented in Prolog. Our implementations of depth-first, breadth-first, and best-first search use *open* and *closed* lists to record states in the search. The open list contains all potential next states in the search. How the open list is maintained, as a stack, as a queue, or as a priority queue, determines which particular state is next, that is, search is in either depth-first, breadth-first, or as best-first modes. The closed set keeps track of all the states that have been previously visited, and is used primarily to preventing looping in the graph as well as to keep track of the current path through the space. The details of how the open and closed data structures organize a search space can be found in Luger (2009, Chapter 3 and 4). When search fails at any point we do not backtrack. Instead, open and closed are updated within the `path` call and the search continues with these revised values. The cut is used to keep Prolog from storing the old versions of the open and closed lists.

Depth-first Search

Because the values of variables are restored when recursion backtracks, the list of visited states in the depth-first path algorithm of Section 4.2 records states only if they are on the current path to the goal. Although the testing each "new" state for membership in this list prevents loops, it still allows branches of the space to be reexamined if they are reached along paths generated earlier but abandoned at that time as unfruitful. A more efficient implementation keeps track of all the states that have ever been encountered. This more complete collection of states made up the members of the set we call *closed* (see Luger 2009, Chapter 3), and `Closed_set` in the following algorithm.

`Closed_set` holds all states on the current path plus the states that were rejected when the algorithm determined they had no usable children; thus, it no longer represents the path from the start to the current state. To capture this path information, we create the ordered pair `[State, Parent]` to keep track of each state and its parent; the `Start` state is represented by `[Start, nil]`. These state–parent pairs will be used to re-create the solution path from the `Closed_set`.

We now present a shell structure for depth-first search in Prolog, keeping track of both open and closed and checking each new state to be sure it was not previously visited. `path` has three arguments, the `Open_stack`, `Closed_set`, maintained as a set, and the `Goal` state. The current state, `State`, is the next state on the `Open_stack`. The stack and set operators are found in Section 3.3.

Search starts by calling a `go` predicate that initializes the `path` call. Note that `go` places the `Start` state with the `nil` parent, `[Start, nil]`, alone on `Open_stack`; the `Closed_set` is empty:

```
go(Start, Goal) :-
     empty_stack(Empty_open),
     stack([Start, nil], Empty_open, Open_stack),
     empty_set(Closed_set),
     path(Open_stack, Closed_set, Goal).
```

The three-argument `path` call is:

```
path(Open_stack, _, _) :-
     empty_stack(Open_stack),
     write('No solution found with these rules').
path(Open_stack, Closed_set, Goal) :-
     stack([State, Parent], _, Open_stack),
     State = Goal,
     write(`A Solution is Found!'), nl,
     printsolution([State, Parent], Closed_set).
path(Open_stack, Closed_set, Goal) :-
     stack([State, Parent], Rest_open_stack,
          Open_stack),
     get_children(State, Rest_open_stack, Closed_set,
          Children),
     add_list_to_stack(Children, Rest_open_stack,
          New_open_stack),
union([[State, Parent]], Closed_set,
          New_closed_set),
path(New_open_stack, New_closed_set, Goal), !.

get_children(State, Rest_open_stack, Closed_set,
          Children) :-
bagof(Child, moves(State, Rest_open_stack,
          Closed_set, Child), Children).

moves(State, Rest_open_stack, Closed_set, [Next,
          State]) :-
    move(State, Next),
    not(unsafe(Next)),     % test depends on problem
    not(member_stack([Next,_], Rest_open_stack)),
    not(member_set([Next,_], Closed_set)).
```

We assume a set of `move` rules appropriate to the problem, and, if necessary, an `unsafe` predicate:

```
move(Present_state, Next_state) :- …     % test rules
move(Present_state, Next_state) :- …
…
```

The first `path` call terminates search when the `Open_stack` is empty, which means there are no more states on the open list to continue the search. This usually indicates that the graph has been exhaustively searched. The second `path` call terminates and prints out the solution path when the solution is found. Since the states of the graph search are maintained as

[State, Parent] pairs, printsolution will go to the Closed_set and recursively rebuild the solution path. Note that the solution is printed from start to goal.

```
printsolution([State, nil], _) :- write(State), nl.
printsolution([State, Parent], Closed_set) :-
    member_set([Parent, Grandparent], Closed_set),
    printsolution([Parent, Grandparent], Closed_set),
    write(State), nl.
```

The third path call uses bagof, a Prolog built-in predicate standard to most interpreters. bagof lets us gather all the unifications of a pattern into a single list. The second parameter to bagof is the pattern predicate to be matched in the database. The first parameter specifies the components of the second parameter that we wish to collect. For example, we may be interested in the values bound to a single variable of a predicate. All bindings of the first parameter resulting from these matches are collected in a list, the *bag*, and bound to the third parameter.

In this program, bagof collects the states reached by firing *all* of the enabled production rules. Of course, this is necessary to gather all descendants of a particular state so that we can add them, in proper order, to open. The second argument of bagof, a new predicate named moves, calls the move predicates to generate all the states that may be reached using the production rules. The arguments to moves are the present state, the open list, the closed set, and a variable that is the state reached by a good move. Before returning this state, moves checks that the new state, Next, is not a member of either rest_open_stack, open once the present state is removed, or closed_set. bagof calls moves and collects all the states that meet these conditions. The third argument of bagof represents the new states that are to be placed on the Open_stack.

For some Prolog interpreters, bagof fails when no matches exist for the second argument and thus the third argument, List, is empty. This can be remedied by substituting (bagof(X, moves(S, T, C, X), List); List = [ ]) for the current calls to bagof in the code.

Finally, because the states of the search are represented as state–parent pairs, member check predicates, e.g., member_set, must be revised to reflect the structure of pattern matching. We test if a state–parent pair is identical to the first element of the list of state–parent pairs and then recur if it isn't:

```
member_set([State, Parent], [[State, Parent]|_]).
member_set(X, [_|T]) :- member_set(X, T).
```

Breadth-first Search

We now present the *shell* of an algorithm for breadth-first search using explicit open and closed lists. This algorithm is called by:

```
go(Start, Goal) :-
    empty_queue(Empty_open_queue),
    enqueue([Start, nil], Empty_open_queue,
        Open_queue),
    empty_set(Closed_set),
    path(Open_queue, Closed_set, Goal).
```

`Start` and `Goal` have their obvious values. The shell can be used with the move rules and unsafe predicates for any search problem. Again we create the ordered pair `[State, Parent]`, as we did with depth-first search, to keep track of each state and its parent; the start state is represented by `[Start, nil]`. This will be used by `printsolution` to re-create the solution path from the `Closed_set`. The first parameter of `path` is the `Open_queue`, the second is the `Closed_set`, and the third is the `Goal`. *Don't care* variables, those whose values are not used in a clause, are written as "_".

```
path(Open_queue, _, _) :-
     empty_queue(Open_queue),
     write('Graph searched, no solution found.').
path(Open_queue, Closed_set, Goal) :-
     dequeue([State, Parent], Open_queue, _),
     State = Goal,
     write('Solution path is: '), nl,
     printsolution([State, Parent], Closed_set).
path(Open_queue, Closed_set, Goal) :-
     dequeue([State, Parent], Open_queue,
          Rest_open_queue),
get_children(State, Rest_open_queue,
          Closed_set, Children),
     add_list_to_queue(Children, Rest_open_queue,
          New_open_queue),
     union([[State, Parent]], Closed_set,
          New_closed_set),
     path(New_open_queue, New_closed_set, Goal), !.
get_children(State, Rest_open_queue, Closed_set,
          Children) :-
     bagof(Child, moves(State, Rest_open_queue,
          Closed_set, Child), Children).
moves(State, Rest_open_queue, Closed_set, [Next,
          State]) :-
     move(State, Next),
     not(unsafe(Next)),     %test depends on problem
     not(member_queue([Next,_], Rest_open_queue)),
     not(member_set([Next,_], Closed_set)).
```

This algorithm is a shell in that no `move` rules are given. These must be supplied to fit the specific problem domain, such as the FWGC problem of Section 4.2. The queue and set operators are found in Section 3.3.

The first `path` termination condition is defined for the case that `path` is called with its first argument, `Open_queue`, empty. This happens only when no more states in the graph remain to be searched and the solution has not been found. A solution is found in the second path predicate when the head of the `open_queue` and the `Goal` state are identical. When

`path` does not terminate, the third call, with `bagof` and `moves` predicates, gathers all the children of the current state and maintains the queue. (The detailed actions of these two predicates were described in Section 4.3.2.) In order to recreate the solution path, we saved each state as a state–parent pair, `[State, Parent]`. The start state has the parent `nil`. As noted in Section 4.3.1, the state–parent pair representation makes necessary a slightly more complex pattern matching in the `member`, `moves`, and `printsolution` predicates.

Best-first Search

Our shell for best-first search is a modification of the breadth-first algorithm in which the open queue is replaced by a priority queue, ordered by heuristic merit, which supplies the current state for each new call to `path`. In our algorithm, we attach a heuristic measure permanently to each new state on open and use this measure for ordering the states on open. We also retain the parent of each state. This information is used by `printsolution`, as in depth- and breadth-first search, to build the solution path once the goal is found.

To keep track of all required search information, each state is represented as a list of five elements: the state description, the parent of the state, an integer giving the depth in the graph of the state's discovery, an integer giving the heuristic measure of the state, and the integer sum of the third and fourth elements. The first and second elements are found in the usual way; the third is determined by adding one to the depth of its parent; the fourth is determined by the heuristic measure of the particular problem. The fifth element, used for ordering the states on the `open_pq`, is `f(n) = g(n) + h(n)`. A justification for using this approach to order states for heuristic search, usually referred to as the *A Algorithm*, is presented in Luger (2009, Chapter 4).

As before, the `move` rules are not specified; they are defined to fit the specific problem. The ADT operators for *set* and *priority queue* are presented in Section 3.3. `heuristic`, also specific to each problem, is a measure applied to each state to determine its heuristic weight, the value of the fourth parameter in its descriptive list.

This best-first search algorithm has two termination conditions and is called by:

```
go(Start, Goal) :-
     empty_set(Closed_set),
     empty_pq(Open),
     heuristic(Start, Goal, H),
     insert_pq([Start, nil, 0, H, H], Open, Open_pq),
     path(Open_pq, Closed_set, Goal).
```

`nil` is the parent of `Start` and `H` its heuristic evaluation. The code for best-first search is:

```
path(Open_pq, _,_) :-
     empty_pq(Open_pq),
     write('Graph searched, no solution found.').
```

```
path(Open_pq, Closed_set, Goal) :-
     dequeue_pq([State, Parent, _, _, _], Open_pq,_),
     State = Goal,
     write('The solution path is: '), nl,
     printsolution([State, Parent, _, _, _],
          Closed_set).
path(Open_pq, Closed_set, Goal) :-
     dequeue_pq([State, Parent, D, H, S], Open_pq,
          Rest_open_pq),
     get_children([State, Parent, D, H, S],
          Rest_open_pq, Closed_set, Children, Goal),
     insert_list_pq(Children, Rest_open_pq,
          New_open_pq),
     union([[State, Parent, D, H, S]], Closed_set,
          New_closed_set),
     path(New_open_pq, New_closed_set, Goal), !.
```

`get_children` is a predicate that generates all the children of `State`. It uses `bagof` and `moves` predicates as in the previous searches, with details earlier this Section. A set of `move` rules, a `safe` check for legal moves, and a `heuristic` must be specifically defined for each application. The `member` check must be specifically designed for five element lists.

```
get_children([State,_,D,_, _],Rest_open_pq,
          Closed_set,Children,Goal)  :-
     bagof(Child, moves([State, _, D, _, _],
          Rest_open_pq, Closed_set, Child,Goal),
          Children).
moves([State, _, Depth, _, _], Rest_open_pq,
          Closed_set,[Next,State,New_D,H,S], Goal) :-
     move(State, Next),
     not(unsafe(Next)),        % application specific
     not(member_pq([Next, _, _, _, _],Rest_open_pq)),
     not(member_set([Next, _, _, _, _],Closed_set)),
     New_D is Depth + 1,
     heuristic(Next, Goal, H), % application specific
     S is New_D + H.
```

`printsolution` prints the solution path, recursively finding state–parent pairs by matching the first two elements in the state description with the first two elements of the five element lists that make up the `Closed_set`.

```
printsolution([State, nil, _, _, _], _) :-
     write(State), nl.
printsolution([State, Parent, _, _, _], Closed_set):-
     member_set([Parent, Grandparent, _, _, _],
          Closed_set),
     printsolution([Parent, Grandparent, _, _, _],
          Closed_set),
     write(State), nl.
```

In Chapter 5 we further generalize the approach taken so far in that we present a set of built-in Prolog meta-predicates, predicates like `bagof`, that explicitly manipulate other Prolog predicates. This will set the stage for creating meta-interpreters in Chapter 6.

Exercises

1. Take the `path` algorithm presented for the knight's tour problem in the text. Rewrite the `path` call in the recursive code to the following form:

```
path(X, Y) :- path(X, W), move(W, Y).
```

Examine the trace of this execution and describe what is happening.

2. Write the Prolog code for the farmer, wolf, goat, and cabbage problem, Section 4.2:

   A. Execute this code and draw a graph of the search space.
   B. Alter the rule ordering to produce alternative solution paths.
   C. Use the shell in the text to produce a breadth-first problem.
   D. Describe a heuristic that might be appropriate for this problem.
   E. Build the heuristic search solution.

3. Do A - E as in Exercise 2 to create a production system solution for the Missionary and Cannibal problem. Hint: you may want the `is` operator, see Section 5.3.

> Three missionaries and three cannibals come to the bank of a river they wish to cross. There is a boat that will hold only two, and any of the group is able to row. If there are ever more missionaries than cannibals on any side of the river the cannibals will get converted. Devise a series of moves to get all the people across the river with no conversions.

4. Use and extend your code to check alternative forms of the missionary and cannibal problem—for example, when there are four missionaries and four cannibals and the boat holds only two. What if the boat can hold three? Try to generalize solutions for the whole class of missionary and cannibal problems.

5. Write a production system Prolog program to solve the full 8 x 8 Knight's Tour problem. Do tasks A - E as described in Exercise 2.

6. Do A - E as in Exercise 2 above for the Water Jugs problem:

> There are two jugs, one holding 3 and the other 5 gallons of water. A number of things can be done with the jugs: they can be filled, emptied, and dumped one into the other either until the poured-into jug is full or until the poured-out-of jug is empty. Devise a sequence of actions that will produce 4 gallons of water in the larger jug. (Hint: use only integers.)