3 Abstract Data Types and Search

Chapter Objectives

Prolog's graph search representations were described and built:
    Lists
A recursive tree-walk algorithm
The cut predicate, !, for Prolog was presented:
    Controls the interpreter's backtracking
Limits variable instantiations, and thus
    May prevent the interpreter from computing good solutions
Demonstrated procedural abstraction and information hiding with Abstract Data Types
The stack operators
    The queue operators
    The priority queue operators
    Sets

Chapter Contents

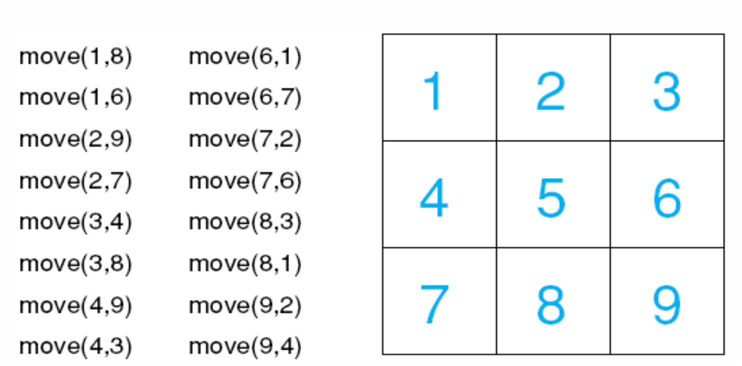
3.1 Introduction

Recursion-Based Graph Search

We next introduce the 3 x 3 knight's tour problem, create a predicate calculus based representation of problem states, and a recursive search of its state space. The chess knight can move on a restricted board as on any regular chessboard: two squares one direction (horizontally or vertically) and one in the other (vertically or horizontally). Thus, the knight can go from square 1 to either square 6 or 8 or from square 9 to either 2 or 4. We ask if the knight can generate a sequence on legal moves from one square to another on this restricted chessboard, from square 1 to 9, for example. The knight's moves are represented in Prolog using `move` facts, Figure 3.1.

The `path` predicate defines an algorithm for a path between its two arguments, the present state, `X`, and the goal that it wants to achieve, `Y`. To do this it first tests whether it is where it wants to be, `path(Z, Z)`, and if not, looks for a state, `W`, to move to

The Prolog search defined by `path` is a recursive, depth-first, left-to-right, tree walk. As shown in Section 2.3, `assert` is a built-in Prolog predicate that always succeeds and has the side effect of placing its argument in the database of specifications. The `been` predicate is used to record each state as it is visited and then `not(been(X))` determines, with each new state found whether that state has been previously visited, thus avoiding looping within the search space.



```
move(1,8)    move(6,1)
move(1,6)    move(6,7)
move(2,9)    move(7,2)
move(2,7)    move(7,6)
move(3,4)    move(8,3)
move(3,8)    move(8,1)
move(4,9)    move(9,2)
move(4,3)    move(9,4)
```

Figure 3.1. The 3 x 3 chessboard and set of legal moves expressed as Prolog facts.

```
path(Z, Z).
path(X, Y) :-
     move(X, W), not(been(W)), assert(been(W)),
     path(W, Y).
```

This use of the `been` predicate violates good programming practice in that it uses global side-effects to control search. `been(3)`, when asserted into the database, is a fact available to any other predicate and, as such, has global extension. We created `been` to modify the program execution.

A more sophisticated method for control of search is to create a list that keeps track of visited states. We create this list and make it the third argument of the `path` predicate. As each new state is encountered, the `member` predicate, Section 2.3, checks whether or not it is already a visited state. If the state is not a member of the list we put it on this list in the order in which it was encountered, the most recent state encountered the head of the list. If the new state is on the list of already visited states, the `path` predicate backtracks looking for new non-visited states. This approach remedies the problems of using global `been(W)`. The following clauses implement depth-first left-to right graph search with backtracking.

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

The third parameter of `path` is the variable representing the list of visited states. When a new state not already on the list of visited states `L`, it is placed on the front of the state list `[Z | L]` for the next `path` call. It should be noted that all the parameters of `path` are local and their current values depend on where they are called in the graph search. Each successful recursive call adds a state to this list. If all continuations from a certain state fail, then that particular `path` call fails and the interpreter backs up to the parent call. Thus, states are added to and deleted from this state list as the backtracking search moves through the graph.

When the `path` call finally succeeds, the first two parameters are identical and the third parameter is the list of states visited, in reverse order. Thus we can print out the steps of the solution. The call to the Prolog interpreter `path(X,Y,[X])`, where `X` and `Y` are replaced by numbers between 1 and 9, finds a path from state `X` to state `Y`, if the path exists. The third parameter initializes the path list with the starting state `X`. Note that there is no typing distinction in Prolog: the first two parameters are any representation of states in the problem space and the third is a list of states. Unification makes this generalization of pattern matching across data types possible. Thus, `path` is a general depth-first search algorithm that may be used with any graph. In Chapter 4 we use this algorithm to implement a solution to the farmer, wolf, goat, and cabbage problem, with different state specifications of state in the call to `path`.

We now present a solution for the 3 x 3 knight's tour. (It is an exercise to solve the full 8 x 8 knight's tour problem in Prolog. We refer to the two parts of the `path` algorithm by number:

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

?- path(1, 3, [1]).
   path(1, 3, [1]) attempts to match 1. fail 1<>3.
   path(1, 3, [1]) matches 2. X=1, Y=3, L=[1]
        move(1, Z) matches, Z=6,
        not(member(6,[1]))=true,
        call path(6, 3, [6,1]
   path(6, 3, [6,1]) trys to match 1. fail 6<>3.
   path(6, 3, [6,1]) calls 2. X=6, Y=3, L=[6, 1].
        move(6, Z) matches, Z=7,
        not(member(7, [6,1]))=true,
        call path(7, 3, [7,6,1])
   path(7, 3, [7,6,1]) trys to match 1. fail 7<>3.
   path(7, 3, [7,6,1]) in 2: X=7, Y=3, L=[7,6,1].
        move(7, Z) matches Z=6,
        not(member(6, [7,6,1])) fails,   backtrack!
        move(7, Z) matches, Z = 2,
        not(member(2, [7,6,1])) is true
        call path(2, 3, [2,7,6,1])
   path(2, 3, [2,7,6,1]) attempts 1, fail, 2 <> 3.
   path matches 2, X in 2: Y is 3, L is [2, 7, 6, 1]
        move(2, Z) matches, Z=7,
        not(member(…)) fails, backtrack!
        move(2, Z) matches Z=9,
        not(member(…)) is true,
        call path(9, 3, [9,2,7,6,1])
   path(9, 3, [9,2,7,6,1]) fails 1, 9<>3.
```

```
path matches 2, X=9, Y=3, L=[9,2,7,6,1]
     move(9, Z) matches Z = 4,
     not(member(…)) is true,
     call path(4, 3, [4,9,2,7,6,1])
path(4, 3, [4,9,2,7,6,1])fails 1, 4<>3.
path matches 2, X=4, Y=3, L is [4,9,2,7,6,1]
     move(4, Z) matches Z = 3,
     not(member(…)) true,
     call path(3, 3, [3,4,9,2,7,6,1])
path(3, 3, [3,4,9,2,7,6,1]) attempts 1, true, 3=3
```

The recursive `path` call then returns `yes` for each of its calls.

In summary, the recursive `path` call is a *shell* or general control structure for search in a graph: in `path(X, Y, L)`, `X` is the present state; `Y` is the goal state. When `X` and `Y` are identical, the recursion terminates. `L` is the list of states on the current path towards state `Y`, and as each new state `Z` is found with the call `move(X, Z)` it is placed on front of the list: `[Z | L]`. The state list is checked, using `not(member(Z, L))`, to be sure the path does not loop.

In Chapter 4, we generalize this approach creating a closed list that retains all the states visited and does not remove visited states when the path predicate backtracks from a state that has no "useful" children. The difference between the state list `L` in the `path` call above and the closed set in Chapter 4 is that closed records all states visited, while the state list L keeps track of only the present path of states.

3.2 Using `cut` to Control Search in Prolog

The predicate *cut* is represented by an exclamation point, !. The syntax for cut is that of a goal with no arguments. Cut has several side effects: first, when originally encountered it always succeeds, and second, if it is "failed back to" in backtracking, it causes the entire goal in which it is contained to fail. For a simple example of the effect of the cut, we create a two-move `path` call from the knight's tour example that we just presented. Consider the predicate path2:

```
path2(X, Y) :- move(X, Z), move(Z, Y).
```

There is a two-move path between X and Y if there exists an intermediate state Z between them. We assume part of the knight's tour database:

```
move(1, 6).
move(1, 8).
move(6, 7).
move(6, 1).
move(8, 3).
move(8, 1).
```

The interpreter finds all the two-move paths from 1; there are four:

```
?- path2(1, W).
W = 7
```

```
;
W = 1
;
W = 3
;
W = 1
;
no
```

When `path2` is altered with cut, only two answers result:

```
path2(X, Y) :- move(X, Z), !, move(Z, Y)
?- path2(1, W).
W = 7
;
W = 1
;
no
```

The `no` response happens because variable `Z` takes on only one value (the first value it is bound to), namely 6. Once the first subgoal succeeds, `Z` is bound to 6 and the cut is encountered. This prohibits further backtracking using the first `move` subgoal and allowing any further bindings for the variable `Z`.

There are several justifications for the use of cut in Prolog programming. First, as this example demonstrated, it allows the programmer to control precisely the shape of the search tree. When further (exhaustive) search is not required, the tree can be explicitly pruned at that point. This allows Prolog code to have the flavor of function calling: when one set of values (bindings) is "returned" by a Prolog predicate (or set of predicates) and the cut is encountered, the interpreter does not search for any other unifications. Thus, if that set of values does not lead to a solution then no further values are attempted. Of course, in the context of the mathematical foundations of the predicate calculus itself, cut may prevent the computation of possible interpretations of the particular predicate calculus and as a result eliminate a possible answer or *model*, (Luger 2009, Sections 2.3, 14.3).

A second use of the cut controls recursive calls. For example, in the `path` call:

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

The addition of cut means that (at most) one solution to the graph search is produced. This single solution is produced because further solutions occur after the clause `path(Z, Z, L)` is satisfied. If the user asks for more solutions, `path(Z, Z, L)` fails, and the second `path` call is reinvoked to continue the (exhaustive) search of the graph. When the cut is placed after the recursive `path` call, the call cannot be reentered (backed into) for further search.

Important side effects of the cut are to make the program run faster and to conserve memory locations. When cut is used within a predicate, the pointers in memory needed for backtracking to predicates to the left of the cut are not created. This is, of course, because they will never be needed. Thus, cut produces the desired solution, and only the desired solution, with a more efficient use of memory.

The cut can also be used with recursion to reinitialize the path call for further search within the graph. This will be demonstrated with the general search algorithms presented in Chapter 4. For this purpose we also need to develop several abstract data types.

3.3 Abstract Data Types (ADTs) in Prolog

Programming, in almost any environment, is enhanced by creating procedural abstractions and by hiding information. Because the *set*, *stack*, *queue*, and *priority queue* data structures are important support constructs for graph search algorithms, a major component of AI problem solving, we build them in Prolog in the present section. We will use these ADTs in the design of the Prolog search algorithms presented in Chapter 4.

Since lists, recursion, and pattern matching, as emphasized throughout this book, are the primary tools for building and searching graph structures. These are the pieces with which we build our ADTs. All list handling and recursive processing that define the ADT are "hidden" within the ADT abstraction, quite different than the normal static data structure.

The ADT Stack

A *stack* is a linear structure with access at one end only. Thus all elements must be added to, *pushed*, and removed, *popped*, from the structure at that access end. The stack is sometimes referred to as a last-in-first-out (LIFO) data structure. We will see its use with depth-first search in Chapter 4. The operators that we will define for a stack are:

1. Test whether the stack is empty.
2. *Push* an element onto the stack.
3. *Pop* or remove, the top element from the stack.
4. *Peek* (often called *Top*) to see the top element on the stack without popping it.
5. `Member_stack`, checks whether an element is in the stack.
6. `Add_list_to stack`, adds a list of elements to the stack.

Operators 5 and 6 may be built from 1–4.

We now build these operators in Prolog, using the list primitives:

1. `empty_stack([ ]).`

This predicate can be used either to test a stack to see whether it is empty or to generate a new empty stack.

2–4. `stack(Top, Stack, [Top | Stack]).`

This predicate performs the push, pop, and peek predicates depending on the variable bindings of its arguments. For instance, push produces a new stack as the third argument when the first two arguments are bound. Likewise, pop produces the top element of the stack when the third argument is bound to the stack. The second argument will then be bound to the new stack, once the top element is popped. Finally, if we keep the stack as the third argument, the first argument lets us peek at its top element.

```
5. member_stack(Element, Stack) :-
       member(Element, Stack).
```

This allows us to determine whether an element is a member of the stack. Of course, the same result could be produced by creating a recursive call that peeked at the next element of the stack and then, if this element did not match Element, popped the stack. This would continue until the empty stack predicate was true.

```
6. add_list_to_stack(List, Stack, Result) :-
       append(List, Stack, Result).
```

`List` is added to `Stack` to produce `Result`, a new stack. Of course, the same result could be obtained by popping `List` (until empty) and pushing each element onto a temporary stack. We would then pop the temporary stack and push each element onto the `Stack` until `empty_stack` is true for the temporary stack. `append` is described in detail in Chapter 10.

A final predicate for printing a stack in reverse order is `reverse_print_stack`. This is very useful when a stack has, in reversed order, the current path from the start state to the present state of the graph search. We will see several examples of this in Chapter 4.

```
reverse_print_stack(S) :-
    empty_stack(S).
reverse_print_stack(S) :-
    stack(E, Rest, S),
    reverse_print_stack(Rest),
    write(E), nl.
```

The ADT Queue

A *queue* is a first-in-first-out (FIFO) data structure. It is often characterized as a list where elements are taken off or *dequeued* from one end and elements are added to or *enqueued* at the other end. The queue is used for defining breadth-first search in Chapter 4. The queue operators are:

```
1. empty_queue([ ]).
```

This predicate tests whether a queue is empty or initializes a new empty queue.

```
2. enqueue(E, [ ], [E]).
   enqueue(E, [H | T], [H | Tnew]) :-
       enqueue(E, T, Tnew).
```

This recursive predicate adds the element `E` to a queue, the second argument. The new augmented queue is the third argument.

```
3. dequeue(E, [E | T], T).
```

This predicate produces a new queue, the third argument, which is the result of taking the next element, the first argument, off the original queue, the second argument.

```
4. dequeue(E, [E | T], _).
```

This predicate lets us peek at the next element, `E`, of the queue.

5. `member_queue(Element, Queue) :-`
   `member(Element, Queue).`

This tests whether `Element` is a member of `Queue`.

6. `add_list_to_queue(List, Queue, Newqueue) :-`
   `append(Queue, List, Newqueue).`

This predicate enqueues an entire list of elements. Of course, 5 and 6 can be created using 1–4; `append` is presented in Chapter 10.

The ADT Priority Queue

A *priority queue* orders the elements of a regular queue so that each new element added to the priority queue is placed in its sorted order, with the "best" element first. The *dequeue* operator removes the "best" sorted element from the priority queue. We will use the priority queue in the design of the best-first search algorithm in Chapter 4.

Because the priority queue is a sorted queue, many of its operators are the same as the queue operators, in particular, `empty_queue`, `member_queue`, and `dequeue` (the "best" of the sorted elements will be next for the `dequeue`). `enqueue` in a priority queue is the `insert_pq` operator, as each new item is placed in its proper sorted order.

```
insert_pq(State, [ ], [State]) :- !.
insert_pq(State, [H | Tail], [State, H | Tail]) :-
     precedes(State, H).
insert_pq(State, [H | T], [H | Tnew]) :-
     insert_pq(State, T, Tnew).
precedes(X, Y) :- X < Y.      % < depends on problem
```

The first argument of this predicate is the new element that is to be inserted. The second argument is the previous priority queue, and the third argument is the augmented priority queue. The `precedes` predicate checks that the order of elements is preserved. Another priority queue operator is `insert_list_pq`. This predicate is used to merge an unsorted list or set of elements into the priority queue, as is necessary when adding the children of a state to the priority queue for best-first search, Chapter 4. `insert_list_pq` uses `insert_pq` to put each individual new item into the priority queue:

```
insert_list_pq([ ], L, L).
insert_list_pq([State | Tail], L, New_L) :-
     insert_pq(State, L, L2),
     insert_list_pq(Tail, L2, New_L).
```

The ADT Set

Finally, we describe the ADT `set`. A *set* is a collection of elements with no element repeated. Sets can be used for collecting all the children of a state or for maintaining the set of all states visited while executing a search algorithm.

In Prolog a set of elements, e.g., {a,b}, may be represented as a list, `[a,b]`, with the order of the list not important. The set operators include `empty_set`, `member_set`, `delete_if_in`, and `add_if_not_in`. We also include the traditional operators for combining and comparing sets, including `union`, `intersection`, `set_difference`, `subset`, and `equal_set`.

```
empty_set([ ]).
member_set(E, S) :-
     member(E, S).
delete_if_in_set(E, [ ], [ ]).
delete_if_in_set(E, [E | T], T) :- !.
delete_if_in_set(E, [H | T], [H | T_new]) :-
     delete_if_in_set(E, T, T_new), !.
add_if_not_in_set(X, S, S) :-
     member(X, S), !.
add_if_not_in_set(X, S, [X | S]).
union([ ], S, S).
union([H | T], S, S_new) :-
     union(T, S, S2),
     add_if_not_in_set(H, S2, S_new),!.
subset([ ], _).
subset([H | T], S) :-
     member_set(H, S),
     subset(T, S).
intersection([ ], _, [ ]).
intersection([H | T], S, [H | S_new]) :-
     member_set(H, S),
     intersection(T, S, S_new), !.
intersection([_ | T], S, S_new) :-
     intersection(T, S, S_new), !.
set_difference([ ], _, [ ]).
set_difference([H | T], S, T_new) :-
     member_set(H, S),
     set_difference(T, S, T_new), !.
set_difference([H | T], S, [H | T_new]) :-
     set_difference(T, S, T_new), !.
equal_set(S1, S2) :-
     subset(S1, S2),
     subset(S2, S1).
```

In Chapters 4 and 5 we use many of these abstract data types to build more complex graph search algorithms and meta-interpreters in Prolog. For example, the stack and queue ADTs are used to build the "open" list that organizes depth-first and breadth-first search. The set ADTs coordinate the "closed" list that helps prevent cycles in a search.

Exercises

1. Write Prolog code to solve the full 8 X 8 knight's tour problem. This will require a lot of effort if all the `move(X, Y)` facts on the full chessboard are itemized. It is much better to create a set of eight predicates that capture the general rules for moving the knight on the full chessboard. You will also have to create a new representation for the squares on the board. Hint: consider a predicate containing the two element order pair, for example, `state(Row, Column)`.

2. Take the path algorithm presented for the knight's tour problem in the text. Rewrite the `path` call of the recursive code given in Section 3.1 to the following form:

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

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

3. Create a three step path predicate for the knight's tour:

```
path3(X, Y) :- move(X, Z), move(Z, W), move (W, Y).
```

Create a tree that demonstrates the full search for the `path3` predicate. Place the cut operator between the second and third `move` predicates. Show how this prunes the tree. Next place the cut between the first and second `move` predicates and again demonstrate how the tree is pruned. Finally, put two cuts within the `path3` predicate and show how this prunes the search.

4. Write and test the ADTs presented in Section 3.3. `trace` will let you monitor the Prolog environment as the ADTs execute.