State Space Search in Prolog

Introduction

□ One of the most powerful approaches to problem solving in AI
□ Represent problem as
  – a set of states
  – a start state
  – a set of legal moves between states
  – a set of goal states or a goal condition
□ Convenient to represent as a directed graph
Block Stacking

State space

start

goal
Sliding Tiles Problem
8-Puzzle

6 8 7
3 5 1
4 2 -  

1 2 3
4 5 6
7 8 -  

start state  
goal state

362,880 possible states!

A fragment of state space
Route Finding

Get from Euston to Waterloo

Other Examples

<table>
<thead>
<tr>
<th>Problem</th>
<th>Nodes/States</th>
<th>Arcs/Moves</th>
</tr>
</thead>
<tbody>
<tr>
<td>chess</td>
<td>board positions</td>
<td>legal moves</td>
</tr>
<tr>
<td>transport scheduling</td>
<td>candidate schedules</td>
<td>operators modifying schedules</td>
</tr>
</tbody>
</table>
State space search using Prolog

- Represent each arc by a relation:
  
  \[ s(X, Y) \quad \text{or} \quad move(X, Y) \]

  meaning there is a legal move from node X to node Y or state X to state Y. Can say Y is a successor state to X.

- Remember to include moves in both directions.

- Assume a predicate
  
  \[ \text{goal}(X) \]

  that is true if X is a goal node.

The solve predicate

- Want predicate solve(Node, List) that takes a start node Node and returns a list of nodes List on the route to a goal node, if such a route exists.

- Two cases
  
  - Node is already a goal node
  
  - if a route exists to a goal node from a successor of Node, then tacking Node onto this route gives a solution.
Writing this in Prolog

□ solve(Node, [Node]) :-
    goal(Node).

□ solve(Node, [Node|Sol]) :-
    s(Node, Successor),
    solve(Successor, Sol).

□ Note: instead s(Node, Successor) we usually write
    something like: move(State, NextState)

An example

s(a,b).
s(a,c).
s(b,d).
s(b,e).
s(c,f).
e tc...
goal(j).
goal(f).

?- solve(a,Sol).
Execution Trace - 1

\[
\text{solve}(\text{a}, \text{Sol}) \quad \text{by rule 1} \\
\text{goal}(\text{a}) \quad \text{no} \\
\text{s}(\text{a}, \text{N1}) \\
\text{solve}(\text{N1}, \text{Sol1}) \quad \text{by rule 2} \\
\text{Matching: } \text{a} = \text{N} \\
\text{Sol} = [\text{N} \mid \text{Sol1}] \\
\text{This forces: } \text{N} = \text{a} \\
\text{Sol} = [\text{a} \mid \text{Sol1}] \\
\text{solve}(\text{N}, [\text{N}]) : \text{- rule 1} \\
\text{goal}(\text{N}). \\
\text{solve}(\text{N}, [\text{N}]) : \text{- rule 2} \\
\text{s}(\text{N}, \text{N1}), \text{solve}(\text{N1}, \text{Sol1}).
\]

Execution Trace - 2

\[
\text{solve}(\text{b}, \text{Sol1}) \quad \text{by rule 1} \\
\text{goal}(\text{b}) \quad \text{no} \\
\text{s}(\text{b}, \text{N1}') \\
\text{solve}(\text{N1}', \text{Sol1}') \quad \text{by rule 2} \\
\text{Matching: } \text{b} = \text{N}' \\
\text{Sol1} = [\text{N}' \mid \text{Sol1}'] \\
\text{This forces: } \text{N}' = \text{b} \\
\text{Sol1} = [\text{b} \mid \text{Sol1}'] \\
\text{solve}(\text{N}', [\text{N}']) : \text{- rule 1} \\
\text{goal}(\text{N'}). \\
\text{solve}(\text{N}', [\text{N}' \mid \text{Sol1}']) : \text{- rule 2} \\
\text{s}(\text{N}', \text{N1''}), \text{solve}(\text{N1''}, \text{Sol1'}). \\
\text{Matching: } \text{N1'} = \text{d} \\
\text{solve}(\text{d}, \text{Sol1}'') \\
\text{Matching: } \text{N1'} = \text{e} \\
\text{solve}(\text{e}, \text{Sol1}') \\
\text{eventually......no} \\
\text{eventually......yes}
\]
Search path

visits nodes in order: a, b, d, h, e, i, j

depth-first search

First attempt in Prolog

% Naive approach
move(a,b).
move(a,c).
move(b,d).
move(b,e).
move(c,e).
move(c,g).
move(d,h).
move(e,i).
move(e,j).
move(f,k).
goal(j).
goal(f).

solve(State, [State]) :-
goal(State).
solve(State, [State|Sol]) :-
move(State, NewState),
solve(NewState, Sol).

1 ?- solve(a, Sol).
Sol = [a, b, c, j] ;
Sol = [a, c, f] ;
false.

2 ?- solve(d, Sol).
false.

3 ?-
Circular paths

Deal with this by recording where we’ve been

Our Prolog code goes into infinite loop with circular path

% Naive approach

solve(State, [State]) :-
    goal(State).
solve(State, [State|Sol]) :-
    move(State, NewState),
    solve(NewState, Sol).

move(a,b).
moved(a,c).
mov(b,d).
mov(b,e).
mov(c,f).
mov(c,g).
mov(d,h).
mov(e,i).
mov(e,j).
mov(f,k).

start(a).
goal(f).
goal(j).
goal(f).

1 ?- solve(a, Sol).
ERROR: Out of global stack
2 ?- 
Prolog code to prevent looping

No looping, Sol not instantiated until solve/3 succeeds. Sol is then Path.

solve(State, Path, Sol) :-
goal(State).

solve(State, Path, Sol) :-
move(State, NewState),
not(member(NewState, Path)),
solve(NewState, [State|Path], Sol).

solve(X, Soln) :-
solve(X, [], Sol),
reverse(Sol, Soln).

Prolog code to prevent looping

solve(N,Sol):- solve(N,[],Sol).
solve(N, Path,[N|Path]):- goal(N).
solve(N,Path,Sol):-
s(N,N1),
not (member(N1,Path)),
solve(N1,[N|Path],Sol).
Execution Trace - 1

```
solve(a,Sol) by rule 1

goal(a)
  no

s(a,N1)
  not (member(N1,[]))
solve(N1, [a], Sol)

[] = Path
a = N
Sol = Sol

N1 = b

not (member(b,[]))
solve(b, [a], Sol)

... etc.
```

Execution Trace - 2

```
solve(N, Sol):- solve(N, [ ], Sol). 1
solve(N, Path, [N | path]):- goal(N). 2
solve(N, Path, Sol):=
s(N,N1),
  not (member(N1,Path)),
solve(N1, [N | Path], Sol). 3

solve(j, [a, b, e], Sol)
by rule 2

Matching:
[a, b, e] = Path
j = N
Sol = [N | Path]
This forces:
Sol = [j, a, b, e]

goal(j)
yes
```

Page 11
Lab Work

□ Use Prolog state space search code outlined here to find solution paths for
  - farmer, wolf goat and cabbage problem
  - missionaries and canibals problem

□ You only need to write the correct code for s(N,N1) which we write as move(N,N1).

□ Hint: Instead of
  move(a,b).
  use code like
  move(state(X,X,G,C), state(Y,Y,G,C)) :-
    opp(X,Y), not(unsafe(Y,Y,G,C)).