



Logic Programming: Search Strategies

Alan Smaill

Nov 7, 2013

- ▶ Problem representation
- ▶ Search
 - ▶ Depth First
 - ▶ Iterative Deepening
 - ▶ Breadth First
- ▶ AND/OR (alternating/game tree) search

Many classical AI/CS problems can be formulated as **search** problems.

Examples:

- ▶ Graph searching
- ▶ Blocks world
- ▶ Missionaries and cannibals
- ▶ Planning (e.g. robotics)

Given by:

- ▶ Set of states s_1, s_2, \dots
- ▶ **Goal** predicate $goal(X)$
- ▶ **Step** predicate $s(X, Y)$ that says we can go from state X to state Y
- ▶ A **start state** (or states)

- ▶ A **solution** is a path leading from the S to a goal state G satisfying $goal(G)$.

Example: Blocks world



Take configuration of blocks as a list of three towers, each tower being a list of blocks in a tower from top to bottom.



`[[c,b,a], [], [c]]`

Example: Blocks world



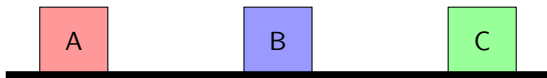
Move a block from top of a tower to top of another tower:



$[[b, a], [], [c])$

Example: Blocks world

Next move:

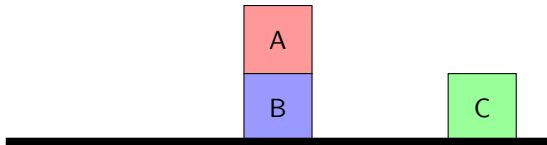


$[[a], [b], [c]]$

Example: Blocks world



Then —



$[[], [a, b], [c])$

- ▶ State is a list of stacks of blocks:

`[[a,b,c], [], []]`

- ▶ Transitions move a block from the top of one stack to the top of another:

`s([[A|As],Bs,Cs), [As,[A|Bs],Cs]).`

`s([[A|As],Bs,Cs), [As,Bs,[A|Cs]]).`

...

- ▶ Can specify particular goal position:

`goal([], [], [a,b,c]).`

Think of the graph generated by these declarations.

$s(a, b)$.

$s(b, c)$.

$s(c, a)$.

$s(c, f(d))$.

$s(f(N), f(g(N)))$.

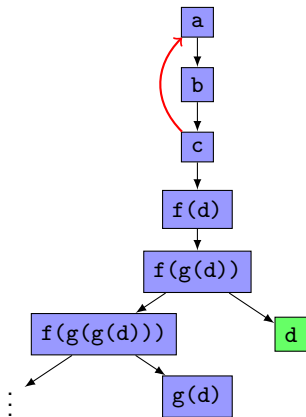
$s(f(g(X)), X)$.

In this case:

▶ the graph is infinite

▶ there is a loop near the top of the graph

$goal(d)$.



We can already see in the blocks world example and in the abstract search space that it is easy to follow actions around in **cycles**, and not find the goal, even if there is a path to the goal.

There are two main approaches to deal with this:

- ▶ remember where you've been; OR ...
- ▶ work with depth bound

Solution 1: remember where you've been

```
% dfs( PathSoFar, CurrentNode, PathToGoal )

dfs_noloop(Path,Node, [Node|Path]) :-
    goal(Node).

dfs_noloop(Path,Node,Path1) :-
    s(Node,Node1),
    \+ member(Node1,Path),
    dfs_noloop([Node|Path],Node1,Path1).
```

Compare the graph from the abstract search space.

Depth First Search has similar problems to Prolog proof search:

- ▶ We may miss solutions because state space is infinite;
- ▶ Even if state space is finite, may wind up finding “easy” solution only after a long exploration of pointless part of search space

- ▶ Keep track of depth, stop if bound exceeded
 - ▶ Note: does not avoid loops (can do this too)

```
dfs_bound(_,Node,[Node]) :-  
    goal(Node).
```

```
dfs_bound(N,Node,[Node|Path]) :-  
    N > 0,  
    s(Node,Node1),  
    M is N-1,  
    dfs_bound(M,Node1,Path)
```

Problem 3: what is a good bound?

- ▶ In general, we just don't know in advance:
 - ▶ Too low? –
Might miss solutions
 - ▶ Too high? – Might spend a long time searching pointlessly

Use the following with some small start value for N

```
dfs_id(N,Node,Path) :-  
    dfs_bound(N,Node,Path)  
    ;  
    M is N+1,  
    dfs_id(M,Node,Path).
```

NB: if there is no solution, this will not terminate.

Keep track of all possible solutions, try shortest ones first;
do this by maintaining a “queue” of solutions

```
bfs([[Node|Path] | _], [Node|Path]) :-  
    goal(Node).
```

```
bfs([Path|Paths], S) :-  
    extend(Path, NewPaths),  
    append(Paths, NewPaths, Paths1),  
    bfs(Paths1, S).
```

```
bfs_start(N, P) :- bfs([[N]], P).
```

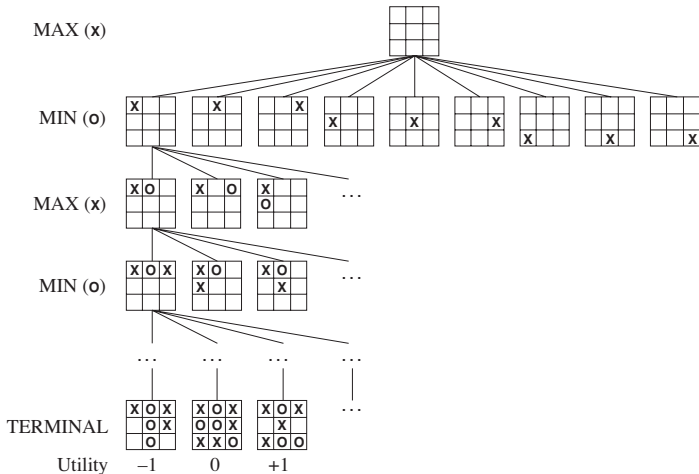
```
extend([Node|Path],NewPaths) :-
    bagof([NewNode,Node|Path],
        (s(Node,NewNode),
         \+ (member(NewNode,[Node|Path]))),
        NewPaths),
    !.
%% if there are no next steps,
%% bagof will fail and we'll fall through.

extend(_Path,[]).
```

- ▶ Concatenating new paths to end of list is slow
- ▶ Avoid this using difference lists?

- ▶ So far we've considered graph search problems
 - ▶ Just want to find some path from start to end
- ▶ Other problems have more structure
 - ▶ e.g. 2-player games
- ▶ AND/OR search is a useful abstraction

Example: Noughts and Crosses



- ▶ `or(S, Nodes)`
 - ▶ S is an OR node with possible next states Nodes
 - ▶ “Our move”
- ▶ `and(S, Nodes)`
 - ▶ S is an AND node with possible next states Nodes
 - ▶ “Opponent moves”
- ▶ `goal(S)`
 - ▶ S is a “win” for us

Example: A simple game

```
and(a, [b, c]).  
or(b, [d, a]).  
or(c, [d, e]).  
goal(e).
```

What is the graph here?


```
andor(Node) :- goal(Node).  
andor(Node) :-  
    or(Node,Nodes),  
    member(Node1,Nodes),  
    andor(Node1).  
andor(Node) :-  
    and(Node,Nodes),  
    solveall(Nodes).  
  
solveall(Nodes) :- ...
```

- ▶ For each AND state, we need solutions for all possible next states
- ▶ For each OR state, we just need one choice
- ▶ A “solution” is thus a tree, or strategy
 - ▶ Can adapt previous program to produce solution tree;
 - ▶ Can also incorporate iterative deepening, loop avoidance, BFS.
 - ▶ heuristic measures of “good” positions leads to algorithms like MiniMax.

- ▶ Bratko, Prolog Programming for Artificial Intelligence
 - ▶ ch. 8 (difference lists), ch. 11 (DFS/BFS)
 - ▶ also Ch. 12 (BestFS), 13 (AND/OR)