Logic Programming

Lecture 7: Search Strategies:
Problem representations
Depth-first, breadth-first, and AND/OR search

Outline for today

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

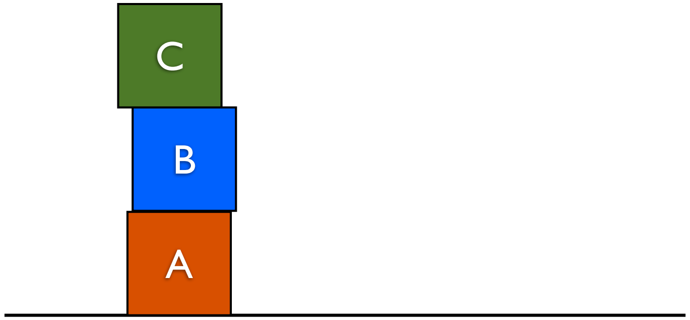
Search problems

- Many classical (AI/CS) problems can be formulated as **search problems**
- Examples:
  - Graph searching
  - Blocks world
  - Missionaries and cannibals
  - Planning (e.g. robotics)

Search spaces

- Set of states s_1,s_2,...
- **Goal predicate** `goal(X)`
- **Step** predicate `s(X,Y)` that says we can go from state `X` to state `Y`
- A **solution** is a path leading from some **start state** `S` to a goal state `G` satisfying `goal(G)`.

Example: Blocks world

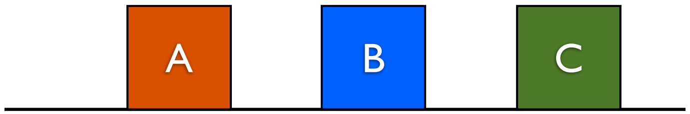


`[[c,b,a],[],[]]`

Example: Blocks world

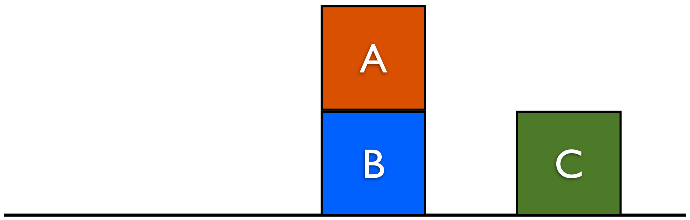
`[[b,a],[],[c]]`

Example: Blocks world



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

Example: Blocks world



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

Representation in Prolog

- State is a list of stacks of blocks.

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

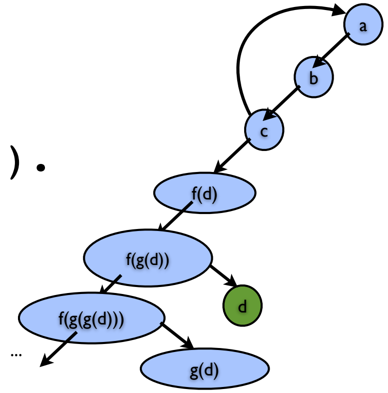
- Transitions move a block from the top of one stack to the top of another

```
s([[A|As],Bs,Cs], [As,[A|Bs],Cs]).
s([[A|As],Bs,Cs], [As,Bs,[A|Cs]]).
...
goal([[],[],[a,b,c]]).
```


An abstract problem space

```
s(a,b).
s(b,c).
s(c,a).
s(c,f(d)).
s(f(N),f(g(N))).
s(f(g(X)),X).

goal(d).
```



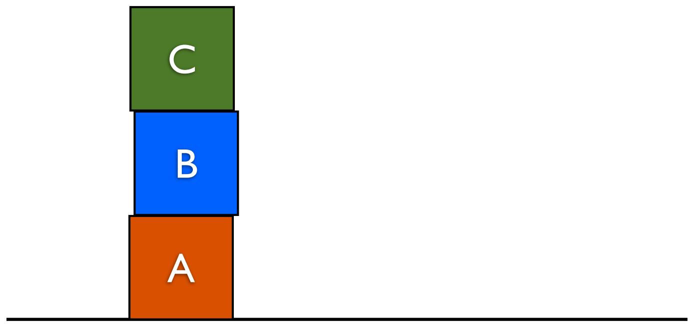
Depth-first search

- `dfs(Node,Path)`
  - `Path` is a path to a goal starting from `Node`

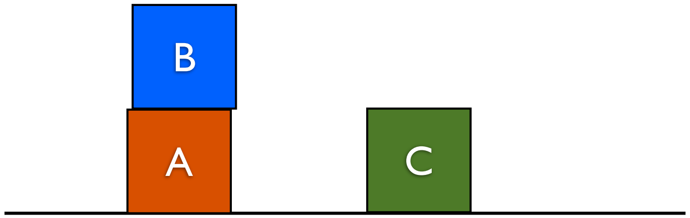
```
dfs(S,[S])   :- goal(S).
dfs(S,[S|P]) :- s(S,T),
                dfs(T,P).
```

- This should look familiar

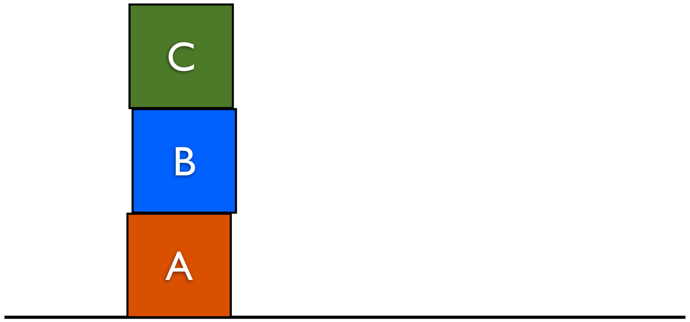
Problem 1: Cycles



Problem 1: Cycles



Problem 1: Cycles

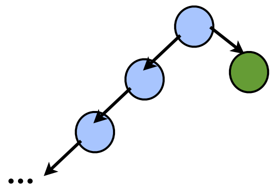


Solution 1: Remember where you've been

- Avoid cycles by avoiding previously visited states

```
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).
```


Problem 2: Infinite state space



- DFS 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

Solution 2: Depth bounding

- 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?

- Don't know this in advance, in general
- Too low?
  - Might miss solutions
- Too high?
  - Might spend a long time searching pointlessly

Solution 3: Iterative deepening

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


Breadth-first search

- Keep track of all possible solutions, try shortest ones first
  - Maintain 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).
```

Extending paths

```
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,[]).
```


Problem: Speed

- Concatenating new paths to end of list is slow
- Avoid this using difference lists?
  - Will revisit next week

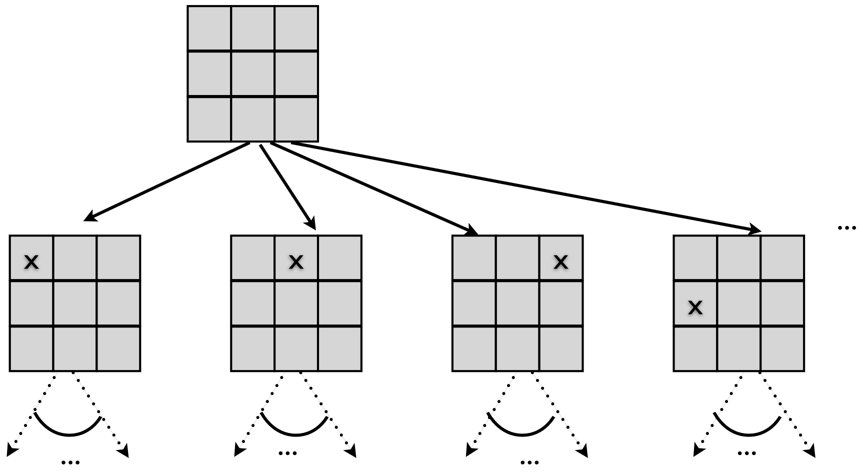
AND/OR search

- 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

AND/OR search

- Search space has 2 kinds of states:
  - OR: "we get to choose next state"
  - AND: "opponent gets to choose"
    - we need to be able to handle any opponent move

Example: Tic tac toe



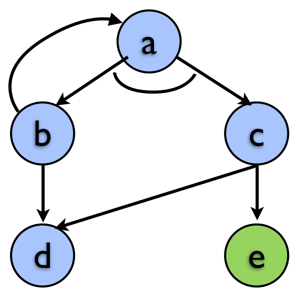
Representation

- `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).
```



Basic idea

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

Solutions

- 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 - min/max

- Further reading:
  - Bratko, *Prolog Programming for Artificial Intelligence*
  - ch. 8 (difference lists), ch. 11 (DFS/BFS)
  - also Ch. 12 (BestFS), 13 (AND/OR)
- Next time:
  - Higher-order logic programming