Implementing Search in Prolog

- How to represent the problem
- Uninformed Search
  - depth first
  - breadth first
  - iterative deepening search
- Informed Search
  - Hill climbing
  - Graph Search
    - which can do depth first, breadth first, best first, Algorithm A, Algorithm A*, etc.
Representing the Problem

• Represent the problem space in terms of these predicates:
  – goal/1
  – start/1
  – arc/3, arc/2

• goal(S) is true iff S is a goal state.
• start(S) is true iff S is an initial state.
• arc(S₁,S₂,N) is true iff there is an operator of cost N that will take us from state S₁ to state S₂.
• arc(S₁,S₂) :- arc(S₁,S₂,_)
Eight Puzzle Example

- Represent a state as a list of the eight tiles and o for blank.
- E.g., \([1,2,3,4,o,5,6,7,8]\) for goal([1,2,3,4,o,5,6,7,8]).

```
1 2 3
4 0 5
6 7 8
```

Missionaries and Cannibals

There are 3 missionaries, 3 cannibals, and 1 boat that can carry up to two people on one side of a river.

- **Goal:** Move all the missionaries and cannibals across the river.
- **Constraint:** Missionaries can never be outnumbered by cannibals on either side of river, or else the missionaries are killed.
- **State:** configuration of missionaries and cannibals and boat on each side of river.
- **Operators:** Move boat containing some set of occupants across the river (in either direction) to the other side.
# Missionaries and Cannibals Solution

<table>
<thead>
<tr>
<th>Far side</th>
<th>Near side</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Initial setup:</td>
<td>MMMCCC B</td>
</tr>
<tr>
<td>1 Two cannibals cross over:</td>
<td>MMMC B CC</td>
</tr>
<tr>
<td>2 One comes back:</td>
<td>MMMCC B C</td>
</tr>
<tr>
<td>3 Two cannibals go over again:</td>
<td>MMM B CCC</td>
</tr>
<tr>
<td>4 One comes back:</td>
<td>MMMC B CC</td>
</tr>
<tr>
<td>5 Two missionaries cross:</td>
<td>MC B MMMCC</td>
</tr>
<tr>
<td>6 A missionary &amp; cannibal return:</td>
<td>MMCC B MC</td>
</tr>
<tr>
<td>7 Two missionaries cross again:</td>
<td>CC B MMM</td>
</tr>
<tr>
<td>8 A cannibal returns:</td>
<td>CCC B MMM</td>
</tr>
<tr>
<td>9 Two cannibals cross:</td>
<td>C B MMMCC</td>
</tr>
<tr>
<td>10 One returns:</td>
<td>CC B MMM</td>
</tr>
<tr>
<td>11 And brings over the third:</td>
<td>- B MMMCCC</td>
</tr>
</tbody>
</table>

---

## Missionaries and Cannibals

% Represent a state as
% [ML,CL,MR,CL,B]

start([3,3,0,0,left]).
goal([0,0,3,3,X]).

% eight possible moves…
arc([ML,CL,MR,CR,left],
    [ML2,CL,MR2,CR,right]):-
% one M & one C row right
   MR2 is MR+1,
   ML2 is ML-1,
   CR2 is CR+1,
   CL2 is CL-1,
   legal(ML2,CL2,MR2,CR2).

arc([ML,CL,MR,CR,left],
    [ML2,CL,MR2,CR,right]):-
% two Ms row right
   MR2 is MR+2,
   ML2 is ML-2,
   legal(ML2,CL2,MR2,CR2).

% is this state a legal one?
legal(ML,CL,MR,CL) :-
   ML>0, CL>0, MR>0, CL>0,
   ML>=CL, MR>=CR.
Depth First Search (1)

%% this is surely the simplest possible DFS.
dfs(S,[S]) :- goal(S).
dfs(S,[S|Rest]) :-
    arc(S,S2),
    dfs(S2,Rest).

Depth First Search (2)

%% this is surely the simplest possible DFS:-
:- ensure_loaded(showPath).
dfs :- dfs(Path), showPath(Path).
dfs(Path) :- start(S), dfs(S,Path).
dfs(S,[S]) :- goal(S).
dfs(S,[S|Rest]) :-
    arc(S,S2),
    dfs(S2,Rest).
Showpath

:- use_module(library(lists)).

%%% Print a search path
showPath(Path) :-
    Path=[First|_],
lst(Path,Last),
nl, write('A solution from '),
showState(First),
write(' to '),
showState(Last),
nl,
foreach1(member(S,Path),(write('  '), showState(S), nl)).

% call Action for each way to prove P.
foreach1(_,_).

%%% once(P) execute's P just once.
once(P) :- call(P), !.

showState(S) :- writeState(S) -> true| write(S).

Depth First Search which avoids loops

/* this version of DFS avoids loops by keeping track of the path as it explores. It's also tail recursive! */
:- ensure_loaded(library(lists)).

dfs(S,Path) :-
    dfs1(S,[S],ThePath),
    reverse(ThePath,Path).

dfs1(S,Path,Path) :- goal(S).
dfs1(S,SoFar,Path) :-
ar(S,S2),
\+(member(S2,SoFar)),
dfs1(S2,[S2|SoFar], Path).
Breadth First Search

bfs :- start(S), bfs(S).

bfs(S) :-
    empty_queue(Q1),
    queue_head(S,Q1,Q2),
    bfs1(Q2).

bfs1(Q) :-
    queue_head(S,_,Q),
    arc(S,G),
    goal(G).

bfs1(Q1) :-
    queue_head(S,Q2,Q1),
    findall(X,arc(S,X), Xs),
    queue_last_list(Xs,Q2,Q3),
    bfs1(Q3).

:- use_module(library(queues)).

bfs(S,Path) :-
    empty_queue(Q1),
    queue_head([S],Q1,Q2),
    bfs1(Q2,Path).

bfs1(Q,[G,S|Tail]) :-
    queue_head([S|Tail],_,Q),
    arc(S,G),
    goal(G).

bfs1(Q1,Solution) :-
    queue_head([S|Tail],Q2,Q1),
    findall([Succ,S|Tail],
            (arc(S,Succ), \+member(Succ,Tail)),
            NewPaths),
    queue_last_list(NewPaths,Q2,Q3),
    bfs1(Q3,Solution).
Note on Queues

- use_module(library(queues))
- empty_queue(?Q)
  - Is true if Q is a queue with no elements.
- queue_head(?Head, ?Q1, ?Q2)
  - Q1 and Q2 are the same queues except that Q2 has Head inserted in the front. Can be used to insert or delete from the head of a Queue.
- queue_last(?Last, ?Q1, ?Q2)
  - Q2 is like Q1 but have Last as the last element in the queue. Can be used to insert or delete from the end of a Queue.
- list_queue(+List, ?Q)
  - Q is the queue representation of the elements in list List.
- Note: Queues are represented as a pair (L,Hole) where list L ends with a variable unified with Hole.

More on Queues

enqueue(X,Qin,Qout) :-
  queue_last(X,Qin,Qout).

dequeue(X,Qin,Qout) :-
  queue_head(X,Qout,Qin).
Iterative Deepening

\[ \text{id}(S, \text{Path}) : - \]
\[ \text{from}(\text{Limit}, 1, 5), \]
\[ \text{id}1(S, 0, \text{Limit}, \text{Path}). \]

\[ \text{id}1(S, \text{Depth}, \text{Limit}, [S]) : - \]
\[ \text{Depth} < \text{Limit}, \]
\[ \text{goal}(S). \]

\[ \text{id}1(S, \text{Depth}, \text{Limit}, [S|\text{Rest}]) : - \]
\[ \text{Depth} < \text{Limit}, \]
\[ \text{Depth2} \text{ is} \text{Depth} + 1, \]
\[ \text{arc}(S, S2), \]
\[ \text{id}1(S2, \text{Depth2}, \text{Limit}, \text{Rest}). \]

Informed Search

- For informed searching we’ll assume a heuristic function \( h(+S,?D) \) that relates a state to an estimate of the distance to a goal.
- Hill climbing
- Best first search
- General graph search which can be used for
  - depth first search
  - breadth first search
  - best first search
  - Algorithm A
  - Algorithm A*
Hill Climbing

hc(Path) :- start(S), hc(S,Path).

hc(S,[S]) :- goal(S), !.

hc(S,[S|Path]) :-
    h(S,H),
    findall(HSS-SS,
        (arc(S,SS),h(SS,Dist)),
        L),
    keysort(L,[BestD-BestSS|_]),
    H>BestD -> hc(BestSS,Path)
    ; (debug("Local max:~p\n", [S]), fail).

Best First Search

:- ensure_loaded(showPath).
:- ensure_loaded(debug).
:- use_module(library(lists)).

/* best first search is like dfs but we chose as the next node to expand the one that seems closest to the goal using the heuristic function h(?S,-D) */

bestfs :- bestfs(Path), showPath(Path).

bestfs(Path) :- start(S), bestfs(S,Path).

bestfs(S,[S]) :- goal(S), !.

bestfs(S,[S|Path]) :-
    findall(Dist-SS,
        (arc(S,SS), h(SS,Dist)),
        L),
    keysort(L,SortedL),
    member(_,NextS,SortedL),
    bestfs(NextS,Path).
Graph Search

The graph is represented by a collection of facts of the form: node(S,Parent,Arcs,G,H) where
- S is a term representing a state in the graph.
- Parent is a term representing S’s immediate parent on the best known path from an initial state to S.
- Arcs is either nil (no arcs recorded, i.e. S is in the set open) or a list of terms C-S2 which represents an arc from S to S2 of cost C.
- G is the cost of the best known path from the state state to S.
- H is the heuristic estimate of the cost of the best path from S to the nearest goal state.

Graph Search

In order to use gs, you must define the following predicates:
- goal(S) true if S is a term which represents the goal state.
- arc(S1,S2,C) true iff there is an arc from state S1 to S2 with cost C.
- h(S,H) is the heuristic function as defined above.
- f(G,H,F) F is the metric used to select which nodes to expand next. G and H are as defined above. Default is "f(G,H,F) :- F is G+H.".
- start(S) (optional) S is the state to start searching from.
gs(Start, Solution) :-
    retractall(node(_,_,_,_,_)),
    addState(Start, Start, 0, 0),
    gSearch(Path),
    reverse(Path, Solution).

gSearch(Solution) :-
    select(State),
    (goal(State) ->
        collect_path(State, Solution)
    | (expand(State), gSearch(Solution))).

select(State) :-
    % find open state with minimal F value.
    findall(F-S, (node(S,P,nil,G,H), f(G,H,F)), OpenList),
    keysort(OpenList, [X-State|Rest]).

expand(State) :-
    dbug("Expanding state ~p.~n", [State]),
    retract(node(State,_,_,_,_)),
    findall(ArcCost-Kid, (arc(State,Kid,ArcCost), add_arc(State,Kid,G,ArcCost)), Arcs),
    assert(node(State,_,A arcs,G,H)).

collect_path(Start, [Start]) :-
    node(Start, Start, _,_,_).
collect_path(S, [S|Path]) :-
    node(S, Parent, Arcs, G, H),
    collect_path(Parent, Path).

add arc(Parent, Child, ParentG, ArcCost) :-
    % Child is a new state, add to the graph.
    (! node(Child,_,_,_,_)),
    G is ParentG+ArcCost,
    h(Child,H),
    dbug("Adding state ~p with parent ~p and cost ~p-n", [Parent,Child,G]),
    assert(node(Child,Parent,nil,G,H)), !.

add arc(Parent, Child, ParentG, ArcCost) :-
    % Child state is already in the graph.
    % update cost if the new path better.
    node(Child,_,CurrentParent,Arcs,CurrentG,H),
    NewG is ParentG+ArcCost,
    CurrentG> NewG, !,
    dbug("Updating ~p 's cost thru ~p to ~p-n", [State,Parent,NewG]),
    retract(node(Child,_,_,_,_)),
    assert(node(Child,Parent,Arcs,NewG,H)),
    % better way to get to any grandKids?
    foreach(member(ArcCost-Kid, Arcs),
        NewCostToChild is NewG+ArcCost,
        update(Child,State,NewCostToChild)).
add arc(_,_,_,_).

Note on Sorting

- sort(+L1,?L2)
  - Elements of the list L1 are sorted into the standard order and identical elements are merged, yielding the list L2.
    | ?- sort([f,s,foo(2),3,1],L).
    L = [1,3,f,s,foo(2)] ?

- keysort(+L1,?L2)
  - List L1 must consist of items of the form Key-Value. These items are sorted into order w.r.t. Key, yielding the list L2. No merging takes place.
    | ?- keysort([3-bob,9-mary,4-alex,1-sue],L).
    L = [1-sue,3-bob,4-alex,9-mary] ?
  - Example:
    youngestPerson(P) :-
        findall(Age-Person,(person(Person),age(Person,Age)),L),
        keysort(L,[_-P|_].