STATE-SPACE SEARCH

Consider the following rudimentary method for problem-solving:

find(X) :- generate(X), test(X).

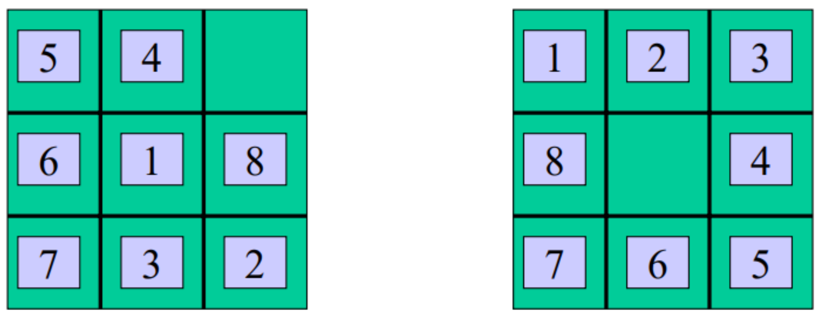
generate(X) 'guesses' alternate solutions via backtracking

test(X) verifies that guess is correct

How much intelligence should be put in the generator, if random guess, then very inefficient

e.g. To sort a list, randomly change the list order

Example State Space: 8-puzzle



Start Goal

- States: integer location for each tile AND …
- Operators: move empty square up, down, left, right
- Goal Test: goal state as given

Depth-first search is built into Prolog

solve_dfs(State,History,Moves) ←
Moves is a sequence of moves to reach a desired final state from the current *State*, where *History* contains the states visited previously.

```
solve_dfs(State,History,[ ]) ←
    final_state(State).
solve_dfs(State,History,[Move|Moves]) ←
    move(State,Move),
    update(State,Move,State1),
    legal(State1),
    not member(State1,History),
    solve_dfs(State1,[State1|History],Moves).
```

Testing the framework

```
test_dfs(Problem,Moves) ←
    initial_state(Problem,State), solve_dfs(State,[State],Moves).
```

Wolf/Goat/Cabbage Problem - Farmer has boat that can carry one of W,G,C across river. He must get all three from left to right side without leaving W with G or G with C

Note that representation is important wgc()

States for the wolf, goat and cabbage problem are a structure *wgc(Boat,Left,Right)*, where *Boat* is the bank on which the boat currently is, *Left* is the list of occupants on the left bank of the river, and *Right* is the list of occupants on the right bank.

```
initial_state(wgc,wgc(left,[wolf,goat,cabbage],[ ])).      - all on left bank
final_state(wgc(right,[ ],[wolf,goat,cabbage])).           - all on right bank

move(wgc(left,L,R),Cargo) ← member(Cargo,L).
move(wgc(right,L,R),Cargo) ← member(Cargo,R).
move(wgc(B,L,R),alone).

update(wgc(B,L,R),Cargo,wgc(B1,L1,R1)) ←
    update_boat(B,B1), update_banks(Cargo,B,L,R,L1,R1).

update_boat(left,right).
update_boat(right,left).

update_banks(alone,B,L,R,L,R).
update_banks(Cargo,left,L,R,L1,R1) ←
    select(Cargo,L,L1), insert(Cargo,R,R1).
update_banks(Cargo,right,L,R,L1,R1) ←
    select(Cargo,R,R1), insert(Cargo,L,L1).

insert(X,[Y|Ys],[X,Y|Ys]) ←
    precedes(X,Y).
insert(X,[Y|Ys],[Y|Zs]) ←                  } Keeps list sorted
    precedes(Y,X), insert(X,Ys,Zs).
insert(X,[ ],[X]).

precedes(wolf,X).
precedes(X,cabbage).

legal(wgc(left,L,R)) ← not illegal(R).
legal(wgc(right,L,R)) ← not illegal(L).

illegal(Bank) ← member(wolf,Bank), member(goat,Bank).
illegal(Bank) ← member(goat,Bank), member(cabbage,Bank).

select(X,Xs,Ys) ← Ys is list after removing X
```

Hill-Climbing

solve_hill_climb(State,History,Moves) ←
Moves is the sequence of moves to reach a desired final state from the current *State*, where *History* is a list of the states visited previously.

```
solve_hill_climb(State,History,[ ]) ←
    final_state(State).
solve_hill_climb(State,History,[Move|Moves]) ←
    hill_climb(State,Move),
    update(State,Move,State1),
    legal(State1),
    not member(State1,History),
    solve_hill_climb(State1,[State1|History],Moves).

hill_climb(State,Move) ←
    findall(M,move(State,M),Moves),
    evaluate_and_order(Moves,State,[ ],MVs),
    member((Move,Value),MVs).
```

evaluate_and_order(Moves,State,SoFar,OrderedMVs) ←
All the *Moves* from the current *State* are evaluated and ordered as *OrderedMVs*. *SoFar* is an accumulator for partial computations.

```
evaluate_and_order([Move|Moves],State,MVs,OrderedMVs) ←
    update(State,Move,State1),
    value(State1,Value),
    insert((Move,Value),MVs,MVs1),
    evaluate_and_order(Moves,State,MVs1,OrderedMVs).
evaluate_and_order([ ],State,MVs,MVs).

insert(MV,[ ],[MV]).
insert((M,V),[(M1,V1)|MVs],[(M,V),(M1,V1)|MVs]) ←
    V ≥ V1.
insert((M,V),[(M1,V1)|MVs],[(M1,V1)|MVs1]) ←
    V < V1, insert((M,V),MVs,MVs1).
```

Testing the framework

```
test_hill_climb(Problem,Moves) ←
    initial_state(Problem,State),
    solve_hill_climb(State,[State],Moves).
```

Hill-climbing search is simple modification of depth-first that requires an evaluation function

Best-first search can be similarly implemented using state(State,Path, Value) where the state, path and heuristic value are remembered for all paths