Negation as failure

- Negation in Prolog: `not/1` or `\+/1`.
- Prolog assumes the closed world paradigm. The negation is therefore different from logical negation:

```
?- member(X,[1,2,3]).

X = 1 ;
X = 2 ;
X = 3 ;
No

?- not(member(X,[1,2,3])).
No

?- not(not(member(X,[1,2,3]))).
```


Negation as failure

- Negation in Prolog: `not/1` or `\+/1`.
- Prolog assumes the closed world paradigm. The negation is therefore different from logical negation:

```
?- member(X,[1,2,3]).

X = 1 ;
X = 2 ;
X = 3 ;
No

?- not(member(X,[1,2,3])).
No

?- not(not(member(X,[1,2,3]))).

X = _G332 ;
No
```

  - `not/1` fails upon resatisfaction (a goal can fail in only one way).
  - `not/1` does not bind variables.

Negation in case selections

```
positive(X) :- X > 0.
negative(X) :- X < 0.

sign(X,+) :- positive(X).
sign(X,-) :- negative(X).
sign(X,0).
```

```
?- sign(1,X).

X = + ;
```


Negation in case selections

```
positive(X) :- X > 0.
negative(X) :- X < 0.

sign(X,+) :- positive(X).
sign(X,-) :- negative(X).
sign(X,0).
```

```
?- sign(1,X).

X = + ;
X = 0 ;
No
```

Negation in case selections

```
positive(X) :- X > 0.
negative(X) :- X < 0.

sign(X,+) :- positive(X).
sign(X,-) :- negative(X).
sign(X,0).

sign1(X,+) :- positive(X).
sign1(X,-) :- negative(X).
sign1(X,0) :- not(positive(X)), not(negative(X)).


?- sign(1,X).

X = + ;
X = 0 ;
No

?- sign1(1,X).

X = + ;
No
```


Modifying the search space

The `!/0` predicate (pronounced "cut") does not allow backtracking over it. All attempts to redo goals to the left of the cut fail.

- Prunes the proof trees (improves efficiency); controversial control facility (not a Horn clause)
- Green cut: increases efficiency

```
gamble(X) :- gotmoney(X),!.
gamble(X) :- gotcredit(X), not(gotmoney(X)).
```

- Red cut: modifies the behaviour of the program

```
gamble(X) :- gotmoney(X),!.
gamble(X) :- gotcredit(X).
```

- Succeed once:

```
member(X,[X,_]).                        memberchk(X,[X,_]) :- !.
member(X,[_,Y]) :- member(X,Y).         memberchk(X,[_,Y]) :- memberchk(X,Y).
```


Modifying the search space (cont'd)

- Succeed once (cont'd):

```
fact1(1,1).
fact1(N,R) :- N1 is N-1, fact1(N1,R1), R is N*R1.

fact2(1,1).
fact2(N,R) :- N>1, N1 is N-1, fact2(N1,R1), R is N*R1.

fact3(1,1) :- !.
fact3(N,R) :- N1 is N-1, fact3(N1,R1), R is N*R1.
```

- Fail goal now
  - An apparently useless predicate: `fail/0` always fails.

Modifying the search space (cont'd)

- Succeed once (cont'd):

```
fact1(1,1).
fact1(N,R) :- N1 is N-1, fact1(N1,R1), R is N*R1.

fact2(1,1).
fact2(N,R) :- N>1, N1 is N-1, fact2(N1,R1), R is N*R1.

fact3(1,1) :- !.
fact3(N,R) :- N1 is N-1, fact3(N1,R1), R is N*R1.
```

- Fail goal now
  - An apparently useless predicate: `fail/0` always fails.

```
not(P) :- P, !, fail.
not(P).
```

Modifying the search space (cont'd)

- Succeed once (cont'd):

```
fact1(1,1).
fact1(N,R) :- N1 is N-1, fact1(N1,R1), R is N*R1.

fact2(1,1).
fact2(N,R) :- N>1, N1 is N-1, fact2(N1,R1), R is N*R1.

fact3(1,1) :- !.
fact3(N,R) :- N1 is N-1, fact3(N1,R1), R is N*R1.
```

- Fail goal now
  - An apparently useless predicate: `fail/0` always fails.

```
not(P) :- P, !, fail.
not(P).
```

  - Another useful predicate: `call/1`.
    - `call(P)` behaves as if P were passed as a goal to the interpreter.

```
not(P) :- call(P), !, fail.
not(P).
```


State space search

- The concept of state space search is widely used in AI.
  - The idea is that a problem can be solved by examining the steps which might be taken towards its solution.
  - Each action takes the solver to a new state.
  - The solution to such a problem is a list of steps leading from the initial state to a goal state.
- The classical example is the Farmer who needs to transport a Goat, a Wolf and some Cabbage across a river one at a time. The Wolf will eat the Goat if left unsupervised. Likewise the Goat will eat the Cabbage.
  - In this case, a state is described by the positions of the Farmer, Goat, Wolf, and Cabbage. The solver can move between states by making a legal move (which does not result in something being eaten).

State space search: The formulation

- The general form of a state space search problem:

  Input :

  1. The start state.
  2. One (or more) goal states or final states.
  3. The state transition function, or how to get from one state to another.

  Output : a list of moves or state transitions that lead from the initial state to one of the final states.

State space search (cont'd)

- There are two approaches to a state space search: depth-first and breadth-first.
- When given some query, Prolog itself performs a no-frills depth-first search (called backtracking) in order to answer the given query:
  - A state in this space is a set of facts that are inferred to be true.
  - Prolog generates a new state by inferring new facts.
  - The initial state is the empty state.
  - A goal state is any state that contains the given query.

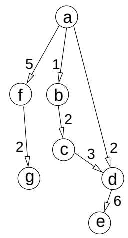
  The only difference between what Prolog does and our formulation is that Prolog does not explain how it reached the goal state; it just states whether a goal state is reachable or not.
- So, the search itself is done by Prolog, we just have to provide a way to report the list of moves.

```
search(Final,Final,[]).
search(Current,Final,[M|Result]) :-
    move(Current,SomeState,M),
    search(SomeState,Final,Result).
```

A simple state space search problem

- Finding a path in a directed, acyclic graph:
  - A state is a vertex of the graph.

```
distance(a,f,5).
distance(f,g,2).
distance(a,b,1).
distance(a,d,2).
distance(b,c,2).
distance(c,d,3).
distance(d,e,6).
move(A,B,to(A,B)) :- distance(A,B,_).
```




```
?- search(a,e,R).

R = [to(a,b),to(b,c),to(c,d),to(d,e)] ;
R = [to(a,d),to(d,e)] ;
No
?- search(e,a,R).
No
```


Searching a state space, revised

- Often, the search space contains cycles. Then, Prolog search strategy may fail to produce a solution.

```
move(A,B,to(A,B)) :- distance(A,B,_).
move(A,B,to(A,B)) :- distance(B,A,_).

?- search(a,e,R).
ERROR: Out of local stack
```


Searching a state space, revised

- Often, the search space contains cycles. Then, Prolog search strategy may fail to produce a solution.

```
move(A,B,to(A,B)) :- distance(A,B,_).
move(A,B,to(A,B)) :- distance(B,A,_).

?- search(a,e,R).
ERROR: Out of local stack
```


- We can help Prolog by using the generate and test technique:
  - We keep track of the previously visited states.
  - Then, we generate a new state (as before), but we also check that we haven't been in that state already. We proceed forward only if the test succeeds.

```
search(Initial,Final,Result) :-
    search(Initial,Final,[Initial],Result).

search(Final,Final,_,[]).
search(Crt,Final,Visited,[M|Result]) :-
    move(Crt,AState,M),          % generate
    not(member(AState,Visited)), % test
    search(AState,Final,[AState|Visited],
           Result).
```

```
?- search(a,e,R).
R = [to(a, b), to(b, c),
     to(c, d), to(d, e)] ;
R = [to(a, d), to(d, e)] ;
No
```


The problem-dependent part

- In order to solve a specific state space search problem, all you have to do is:
  - Establish what is a state for your problem and how will you represent it in Prolog.
  - Establish your state transition function. That is, define the `move/3` predicate.
    - ∗ Such a predicate should receive a state, and return another state together with the move that generates it.
    - ∗ Upon resatisfaction, a new state should be returned.
    - ∗ If no new state is directly accessible from the current one, `move/3` should fail.

LIMITATIONS

- The predicate `search/3` works on any finite search space.
  - It exploits the fact that Prolog performs by itself a depth-first search.
    - Since the depth-first search is not guaranteed to terminate on an infinite search space, neither is `search/3`.
  - It is possible to implement a breadth-first search in Prolog.
    - However, this cannot take advantage of the search strategy which is built in the Prolog interpreter (in fact, it sidesteps it altogether).
    - Such an implementation is thus a bit more complicated and exceeds the scope of this course (but if you are really curious, contact me).

ON GOATS, WOLVES, AND CABBAGE

```
% A state: [Boat,Cabbage,Goat,Wolf]
% Moving around.  We use the ''generate and test'' paradigm:
move(A,B,M) :- move_attempt(A,B,M), legal(B).

% first, attempt to move the Cabbage, then the Goat, then the Wolf:
move_attempt([B,B,G,W],[B1,B1,G,W], moved(cabbage,B,B1)) :- opposite(B,B1).
move_attempt([G,B,G,W],[G1,B,G1,W], moved(goat,G,G1)) :- opposite(G,G1).
move_attempt([W,B,G,W],[W1,B,G,W1], moved(wolf,W,W1)) :- opposite(W,W1).
%... eventually, move the empty boat:
move_attempt([X,C,G,W],[Y,C,G,W], moved(nothing,X,Y)) :- opposite(X,Y).

% By the way, the two shores are opposite:
opposite(south,north).      opposite(north,south).

% Make sure that nothing gets eaten:
legal(State) :- not(conflict(State)).

% we cannot allow the Cabbage and the Goat on the
% same shore unsupervised...
conflict([B,C,C,W]) :- opposite(C,B).
% ... nor the Goat and the Wolf...
conflict([B,C,W,W]) :- opposite(W,B).
% ... but anything else is fine.
```


ON GOATS, WOLVES, AND CABBAGE (CONT'D)

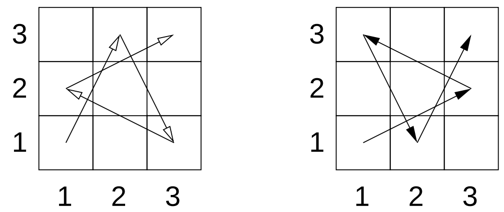
```
?- search([north,north,north,north],
          [south,south,south,south], R).

R = [moved(goat, north, south),
     moved(nothing, south, north),
     moved(cabbage, north, south),
     moved(goat, south, north),
     moved(wolf, north, south),
     moved(nothing, south, north),
     moved(goat, north, south)] ;

R = [moved(goat, north, south),
     moved(nothing, south, north),
     moved(wolf, north, south),
     moved(goat, south, north),
     moved(cabbage, north, south),
     moved(nothing, south, north),
     moved(goat, north, south)] ;

No
```


ON KNIGHTS AND THEIR TOURS

```
% The board size is given by the predicate size/1
size(3).

% The position of the Knight is represented by the structure -(X,Y) (or X-Y),
% where X and Y are the coordinates of the square where the Knight is located.
% We represent a move by the position it generates.

% We use, again, the generate and test technique:
move(A,B,B) :- move_attempt(A,B), inside(B).

% There are 8 possible moves in the middle of the board:
move_attempt(I-J, K-L) :- K is I+1, L is J-2.
move_attempt(I-J, K-L) :- K is I+1, L is J+2.
move_attempt(I-J, K-L) :- K is I+2, L is J+1.
move_attempt(I-J, K-L) :- K is I+2, L is J-1.
move_attempt(I-J, K-L) :- K is I-1, L is J+2.
move_attempt(I-J, K-L) :- K is I-1, L is J-2.
move_attempt(I-J, K-L) :- K is I-2, L is J+1.
move_attempt(I-J, K-L) :- K is I-2, L is J-1.

% However, if the Knight is somwhere close to board's
% margins, some moves might fall out of the board...
inside(A-B) :- size(Max), A > 0, A =< Max, B > 0, B =< Max.
```

ON KNIGHTS AND THEIR TOURS (CONT'D)

```
?- search(1-1,3-3,R).

R = [2-3, 3-1, 1-2, 3-3] ;

R = [3-2, 1-3, 2-1, 3-3] ;

No
```



VARIATIONS ON A SEARCH THEME

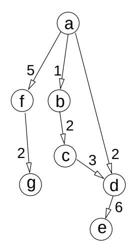
- Since our `search/3` predicate generates all the possible solutions, we can use it within another generate and test process:
- On a 4×4 board, a Knight moves from one square S to another square D. For a given N, find all the paths between S and D in which the Knight does not make more than N moves.

```
search_shorter(S,D,N,Result) :- search(S,D,Result),        % generate
                                length(Result,L), L =< N.   % test
% length([],0).
% length([_|T],L) :- length(T,L1), L is L1+1.

?- search_shorter(1-1,4-3,5,R).

R = [2-3, 3-1, 4-3] ;                  R = [3-2, 2-4, 4-3] ;
R = [2-3, 3-1, 1-2, 2-4, 4-3] ;        R = [3-2, 2-4, 1-2, 3-1, 4-3] ;
R = [2-3, 4-4, 3-2, 2-4, 4-3] ;        R = [3-2, 1-3, 3-4, 2-2, 4-3] ;
R = [2-3, 4-2, 3-4, 2-2, 4-3] ;        No
R = [3-2, 4-4, 2-3, 3-1, 4-3] ;

?- search_shorter(1-1,4-3,4,R).

R = [2-3, 3-1, 4-3] ;          R = [3-2, 2-4, 4-3] ;          No
```


VARIATIONS ON A SEARCH THEME (CONT'D)

- Given some integer n and two vertices A and B, is there a path from A to B of weight smaller than n?

```
distance(a,f,5).
distance(f,g,2).
distance(a,b,1).
distance(a,d,2).
distance(b,c,2).
distance(c,d,3).
distance(d,e,6).
move(A,B,to(A,B,C)) :- distance(A,B,C).
move(A,B,to(A,B,C)) :- distance(B,A,C).
weight([],0).
weight([to(_,_,C)|P],W) :- weight(P,W1), W is W1+C.

smaller(A,B,N,Result) :- search(A,B,Result), weight(Result,W), W =< N.
```




```
?- smaller(a,e,12,R).
R = [to(a, b, 1), to(b, c, 2), to(c, d, 3), to(d, e, 6)] ;
R = [to(a, d, 2), to(d, e, 6)] ;
No
?- smaller(a,e,10,R).
R = [to(a, d, 2), to(d, e, 6)] ;
No
```