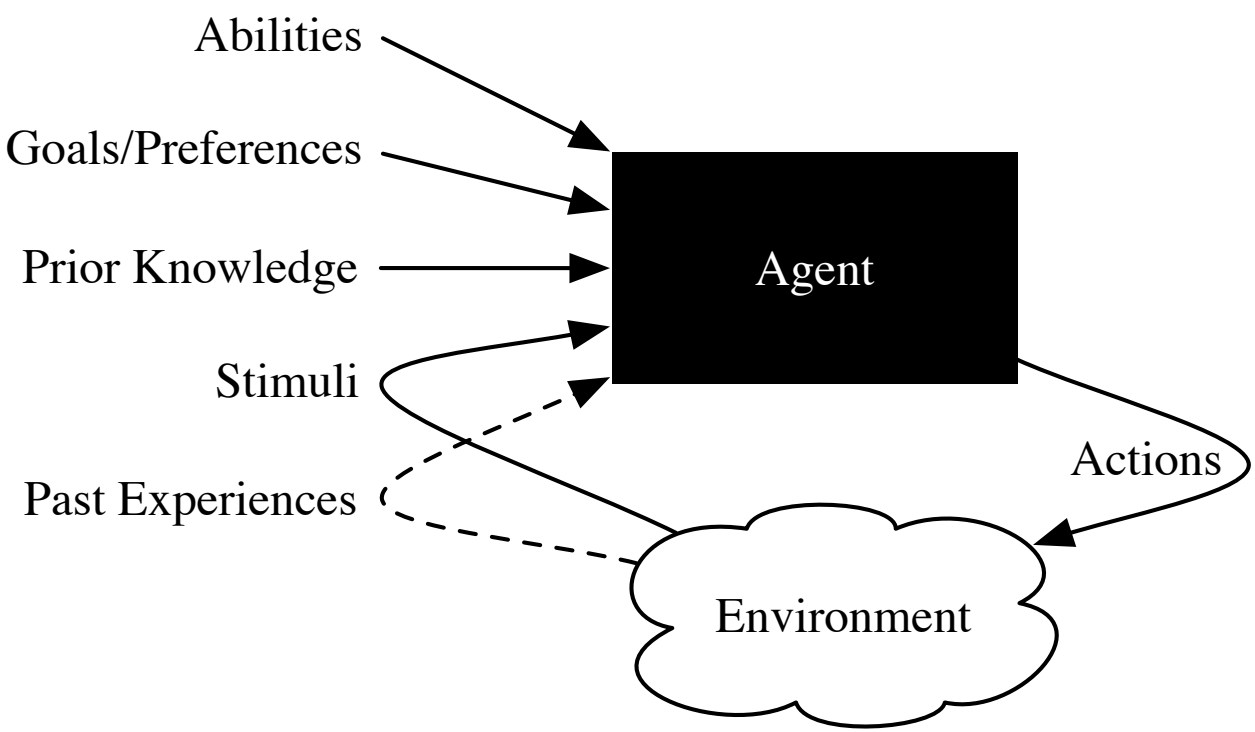
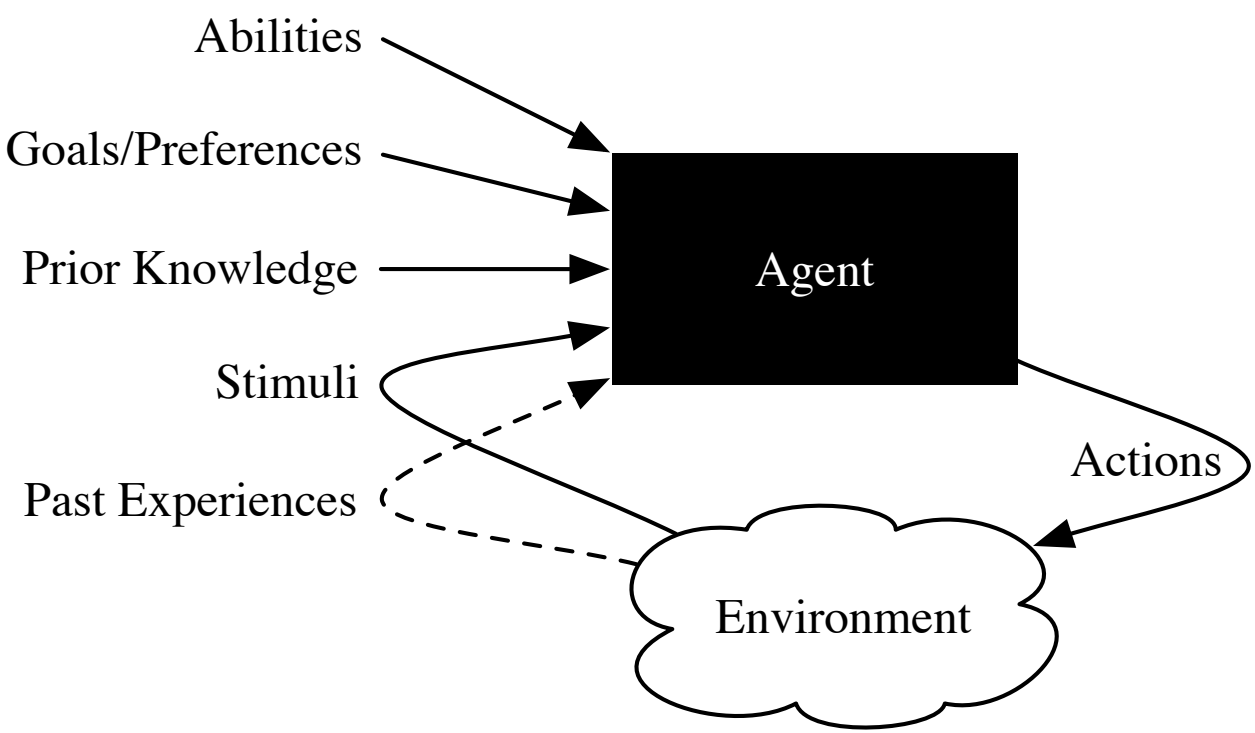
Fsm exercise: solution

```
accept(_,Final,Q,[]) :-

accept(Trans,Final,Q,[H|T]) :-
```



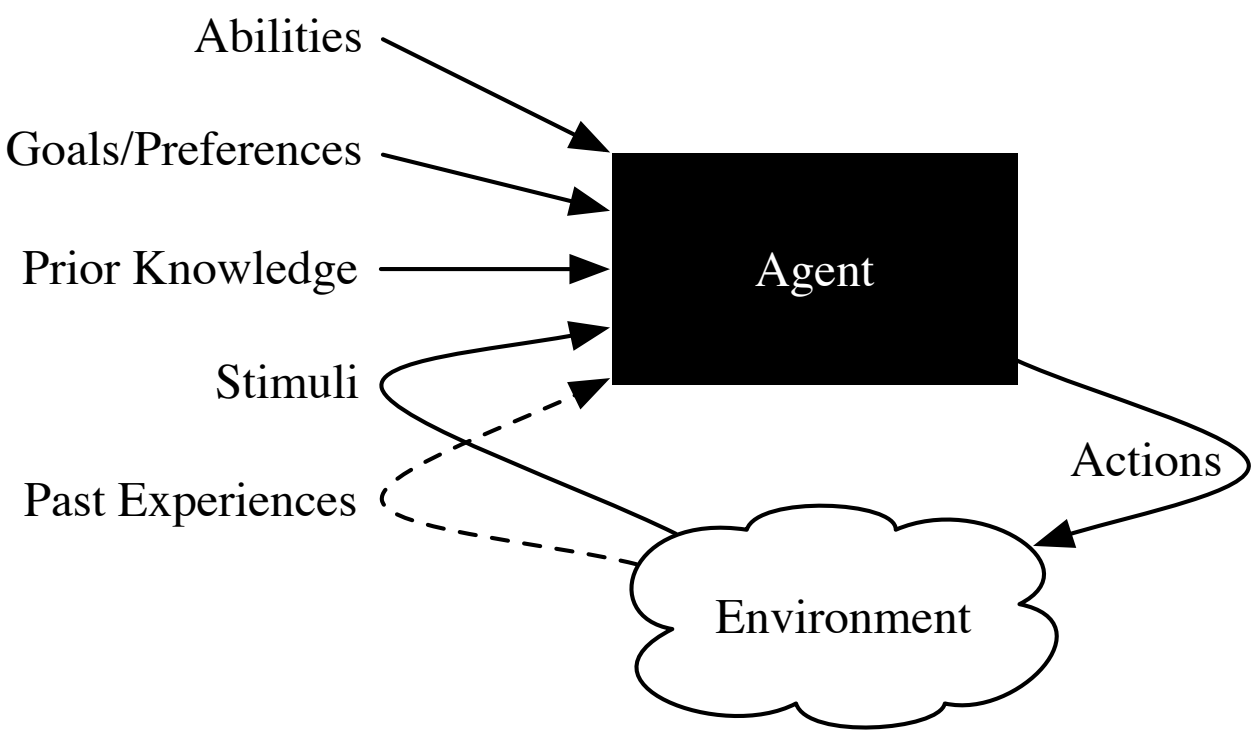
Fsm exercise: solution

```
accept(_,Final,Q,[]) :- member(Q,Final).

accept(Trans,Final,Q,[H|T]) :-


member(X,[X|_]).
member(X,[_|L]):- member(X,L).
```



Fsm exercise: solution

```
accept(_,Final,Q,[]) :- member(Q,Final).

accept(Trans,Final,Q,[H|T]) :-
                        member([Q,H,Qn],Trans),
                        accept(Trans,Final,Qn,T).
member(X,[X|_]).
member(X,[_|L]):- member(X,L).
```



Search (in Prolog)

Given `goal`, `arc`

```
search(Node) :- goal(Node).

search(Node) :- arc(Node,Next), search(Next).
```

Search (in Prolog)

Given `goal`, `arc`

```
search(Node) :- goal(Node).

search(Node) :- arc(Node,Next), search(Next).
```

Example: `accept(Trans,Final,Q0,String)`

`Node as [Q,UnseenString]`

Search (in Prolog)

Given `goal`, `arc`

```
search(Node) :- goal(Node).

search(Node) :- arc(Node,Next), search(Next).
```

Example: `accept(Trans,Final,Q0,String)`

`Node as [Q,UnseenString]`

```
goal(Q,[],Final) :- member(Q,Final).
```

Search (in Prolog)

Given `goal`, `arc`

```
search(Node) :- goal(Node).

search(Node) :- arc(Node,Next), search(Next).
```

Example: `accept(Trans,Final,Q0,String)`

`Node as [Q,UnseenString]`

```
goal(Q,[],Final) :- member(Q,Final).

arc([Q,[H|T]],[Qn,T],Trans) :-
                        member([Q,H,Qn],Trans).
```

Search (in Prolog)

Given goal, arc

```
search(Node) :- goal(Node).

search(Node) :- arc(Node,Next), search(Next).
```

Example: accept(Trans,Final,Q0,String)

Node as [Q,UnseenString]

```
goal(Q,[],Final) :- member(Q,Final).

arc([Q,[H|T]],[Qn,T],Trans) :-
                        member([Q,H,Qn],Trans).

search(Q,S,F,_) :- goal(Q,S,F).

search(Q,S,F,T) :- arc([Q,S],[Qn,Sn],T),
                        search(Qn,Sn,F,T).
```

Search (in Prolog)

Given goal, arc

```
search(Node) :- goal(Node).

search(Node) :- arc(Node,Next), search(Next).
```

Example: accept(Trans,Final,Q0,String)

Node as [Q,UnseenString]

```
goal(Q,[],Final) :- member(Q,Final).

arc([Q,[H|T]],[Qn,T],Trans) :-
                        member([Q,H,Qn],Trans).

search(Q,S,F,_) :- goal(Q,S,F).

search(Q,S,F,T) :- arc([Q,S],[Qn,Sn],T),
                   search(Qn,Sn,F,T).

accept(T,F,Q,S) :- search(Q,S,F,T).
```

Prolog as search

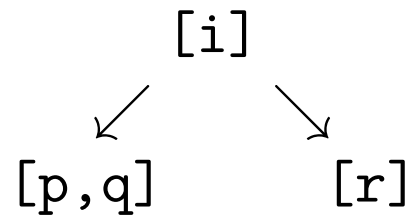
```
i :- p,q.

i :- r.

p.

r.

_____________

| ?- i.
```

Prolog as search

```
i :- p,q.                          [i]

i :- r.

p.

r.

_____________

| ?- i.                    StartNode = [i]
```

Prolog as search

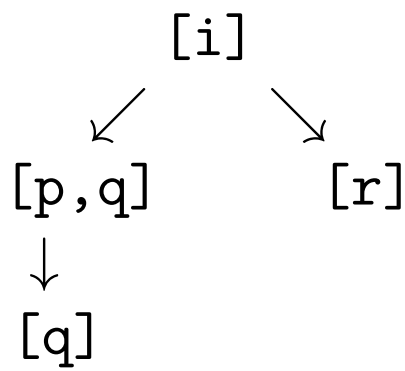
```
i :- p,q.

i :- r.

p.

r.

_____________

| ?- i.
```




```
StartNode = [i]
```

Prolog as search

```
i :- p,q.

i :- r.

p.

r.
```

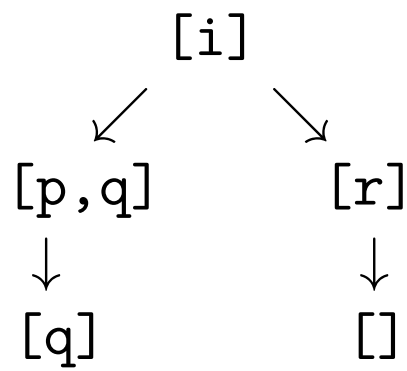


```
| ?- i.
```

```
StartNode = [i]
```

Prolog as search

```
i :- p,q.

i :- r.

p.

r.
```

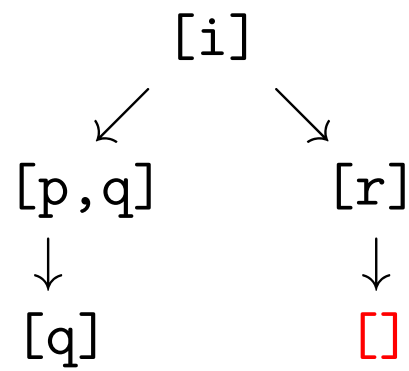


```
| ?- i.
```

StartNode = [i]

Prolog as search

```
i :- p,q.

i :- r.

p.

r.
```

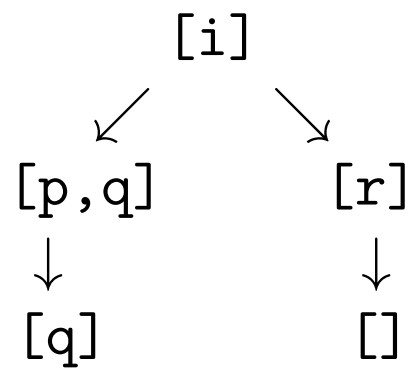


```
| ?- i.

yes
```

```
StartNode = [i]

goal([]).
```

Prolog as search

```
i :- p,q.

i :- r.

p.

r.
```



```
| ?- i.

yes
```

```
StartNode = [i]

goal([]).
```

```
prove(Node) :- goal(Node) .
prove(Node) :- arc(Node,Next), prove(Next).
```

KB and arc

```
i :- p,q.

i :- r.

p.

r.
```

KB and arc

```
i :- p,q.
```

[i,p,q]

```
i :- r.
```

[i,r]

```
p.
```

[p]

```
r.
```

[r]

KB and arc

```
i :- p,q.                    [i,p,q]

i :- r.                      [i,r]

p.                           [p]

r.                           [r]

                KB = [[i,p,q],[i,r],[p],[r]]
```

KB and arc

```
i :- p,q.                    [i,p,q]

i :- r.                       [i,r]

p.                             [p]

r.                             [r]

                KB = [[i,p,q],[i,r],[p],[r]]

arc(Node1,Node2,KB) :- ??
```

KB and arc

```
i :- p,q.                    [i,p,q]

i :- r.                       [i,r]

p.                             [p]

r.                             [r]

                  KB = [[i,p,q],[i,r],[p],[r]]
```

```
arc([H|T],N,KB) :- member([H|B],KB), append(B,T,N).
```

KB and arc

```
i :- p,q.                    [i,p,q]

i :- r.                       [i,r]

p.                             [p]

r.                             [r]

                  KB = [[i,p,q],[i,r],[p],[r]]
```

```
arc([H|T],N,KB) :- member([H|B],KB), append(B,T,N).

prove(Node,KB) :- goal(Node) ;
                  arc(Node,Next,KB), prove(Next,KB).
```

Non-termination

```
i :- p,q.                                   [i]

i :- r.

p :- i.

r.
```

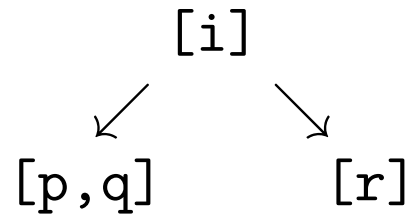
```
| ?- i.
```

```
prove([],_).
prove([H|T],KB) :- member([H|B],KB), append(B,T,Next),
                   prove(Next,KB).
```

Non-termination

```
i :- p,q.

i :- r.

p :- i.

r.
```

```
| ?- i.
```

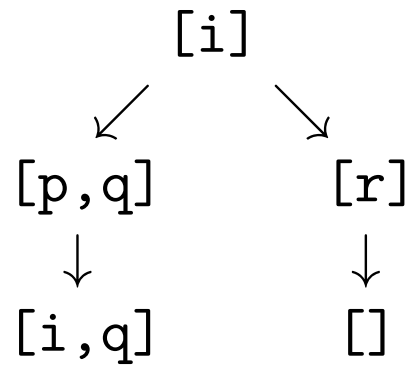



```
prove([],_).

prove([H|T],KB) :- member([H|B],KB), append(B,T,Next),
                   prove(Next,KB).
```

Non-termination

```
i :- p,q.

i :- r.

p :- i.

r.
```

```
| ?- i.
```

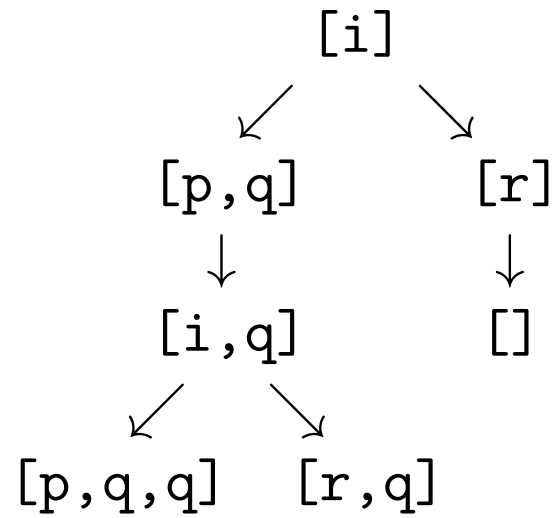



```
prove([],_).

prove([H|T],KB) :- member([H|B],KB), append(B,T,Next),
                   prove(Next,KB).
```

Non-termination

```
i :- p,q.

i :- r.

p :- i.

r.
```

```
| ?- i.
```

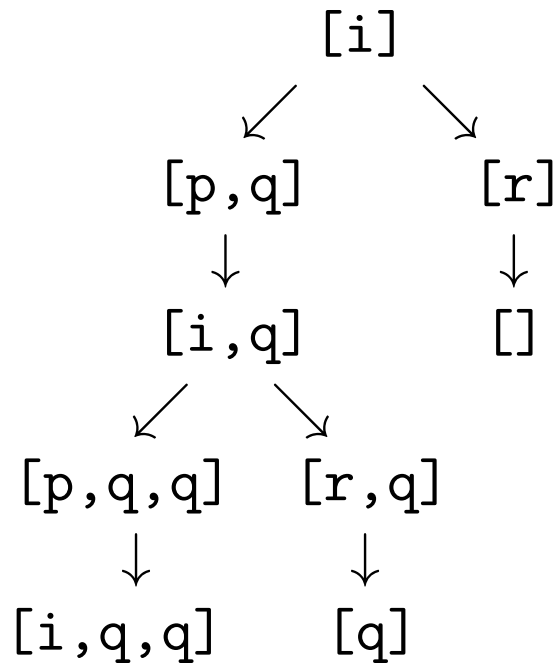



```
prove([],_).

prove([H|T],KB) :- member([H|B],KB), append(B,T,Next),
                   prove(Next,KB).
```

Non-termination

```
i :- p,q.

i :- r.

p :- i.

r.
```

```
| ?- i.
```

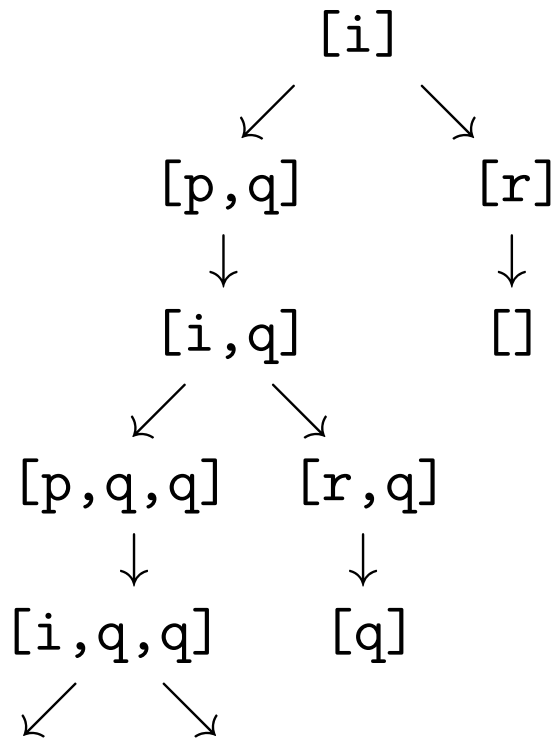



```
prove([],_).

prove([H|T],KB) :- member([H|B],KB), append(B,T,Next),
                   prove(Next,KB).
```

Non-termination

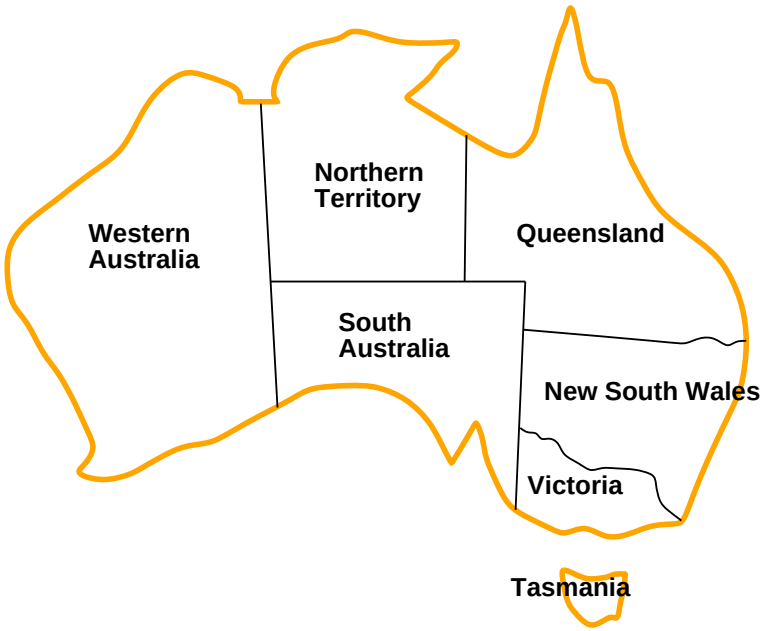
```
i :- p,q.

i :- r.

p :- i.

r.
```

```
| ?- i.
```

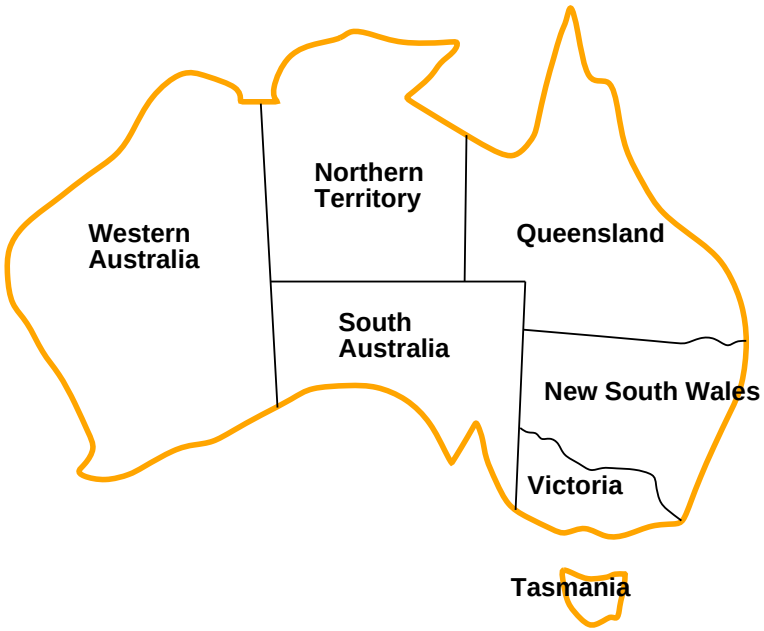


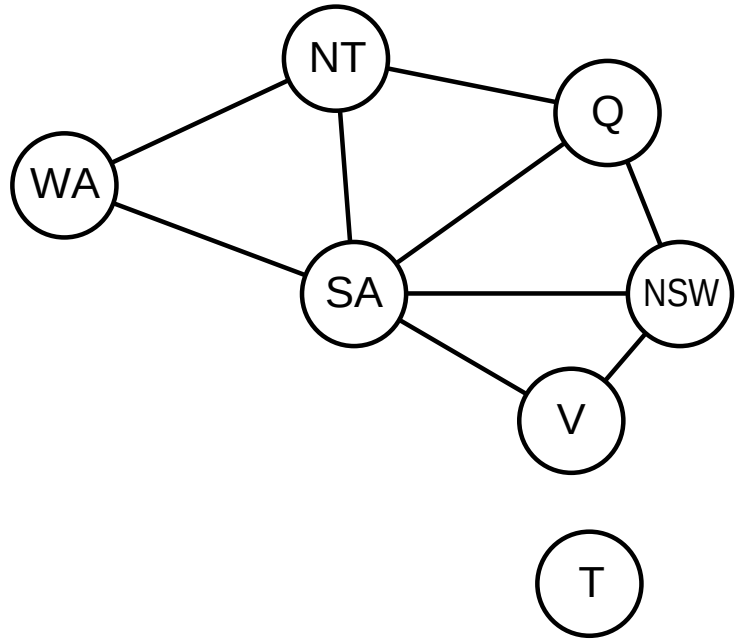

```
prove([],_).

prove([H|T],KB) :- member([H|B],KB), append(B,T,Next),
                   prove(Next,KB).
```

Graph modeling



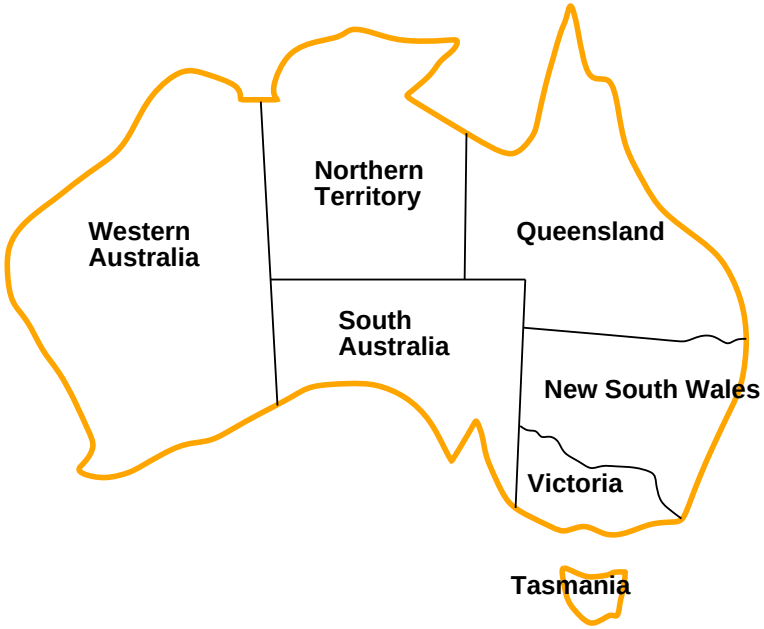
Graph modeling

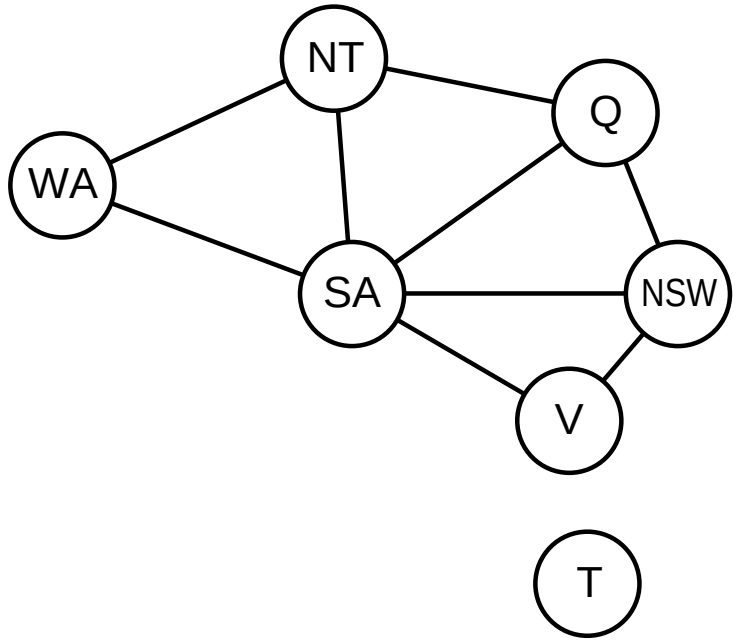




Russell & Norvig

Graph modeling





Russell & Norvig

```
arc(wa,nt).   arc(nt,q).   arc(q,nsw).
arc(wa,sa).   arc(nt,sa).  arc(sa,q).
arc(sa,nsw).  arc(sa,v).   arc(v,nsw).

arc2(X,Y) :- arc(X,Y) ; arc(Y,X).
```

Non-termination (due to poor choices)

```
i :- p,q.                                   [i]

i :- r.

p :- i.

r.
```

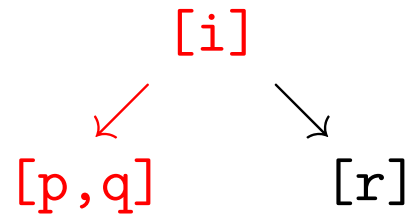
```
| ?- i.
```

```
prove([],_).
prove([H|T],KB) :- member([H|B],KB), append(B,T,Next),
                   prove(Next,KB).

| ?- prove([i],[[i,p,q],[i,r],[p,i],[r]]).
```

Non-termination (due to poor choices)

```
i :- p,q.

i :- r.

p :- i.

r.
```

```
| ?- i.
```

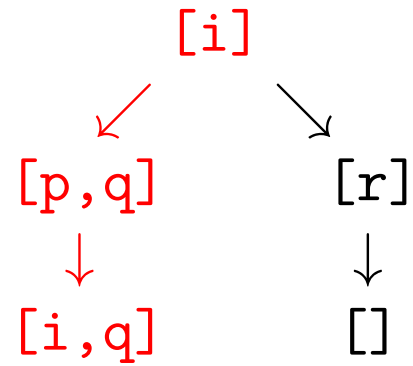



```
prove([],_).

prove([H|T],KB) :- member([H|B],KB), append(B,T,Next),
                   prove(Next,KB).

| ?- prove([i],[[i,p,q],[i,r],[p,i],[r]]).
```

Non-termination (due to poor choices)

```
i :- p,q.

i :- r.

p :- i.

r.
```

```
| ?- i.
```

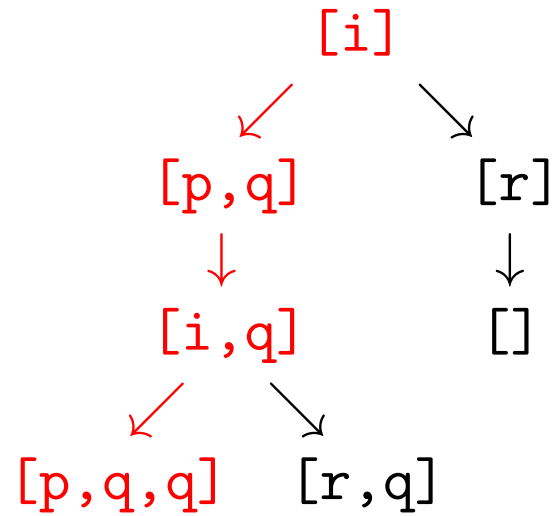



```
prove([],_).

prove([H|T],KB) :- member([H|B],KB), append(B,T,Next),
                   prove(Next,KB).

| ?- prove([i],[[i,p,q],[i,r],[p,i],[r]]).
```

Non-termination (due to poor choices)

```
i :- p,q.

i :- r.

p :- i.

r.
```

```
| ?- i.
```

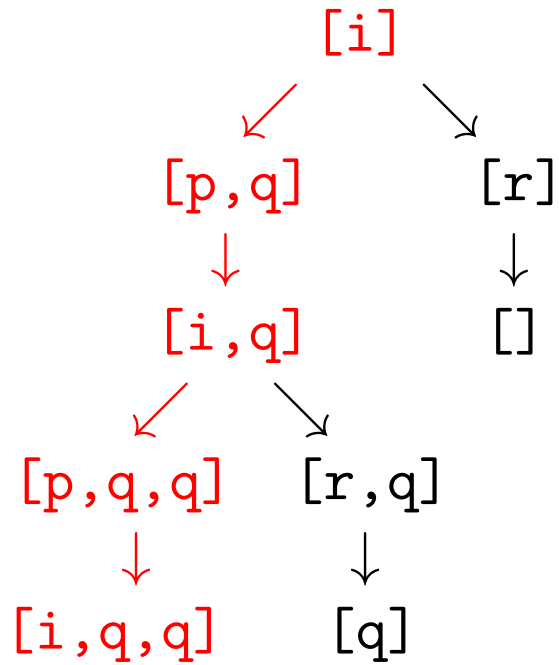



```
prove([],_).

prove([H|T],KB) :- member([H|B],KB), append(B,T,Next),
                   prove(Next,KB).

| ?- prove([i],[[i,p,q],[i,r],[p,i],[r]]).
```

Non-termination (due to poor choices)

```
i :- p,q.

i :- r.

p :- i.

r.
```

```
| ?- i.
```

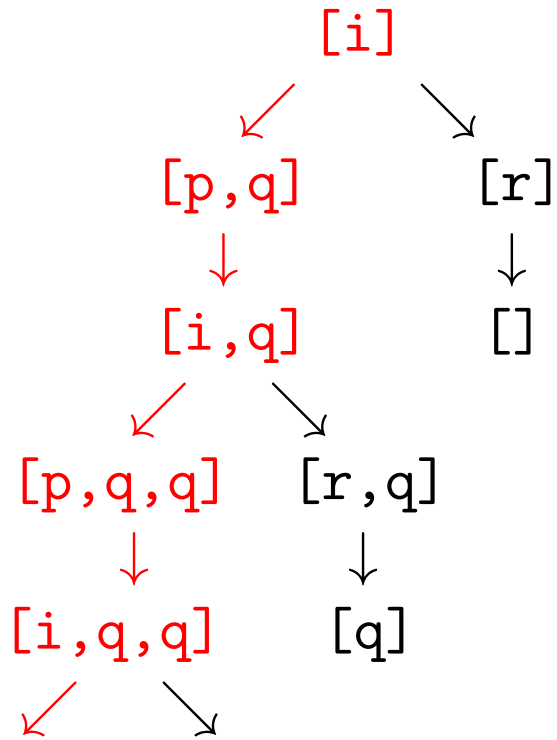



```
prove([],_).

prove([H|T],KB) :- member([H|B],KB), append(B,T,Next),
                   prove(Next,KB).

| ?- prove([i],[[i,p,q],[i,r],[p,i],[r]]).
```

Non-termination (due to poor choices)

```
i :- p,q.

i :- r.

p :- i.

r.
```

```
| ?- i.
```




```
prove([],_).

prove([H|T],KB) :- member([H|B],KB), append(B,T,Next),
                   prove(Next,KB).

| ?- prove([i],[[i,p,q],[i,r],[p,i],[r]]).
```

Determinization (eliminate choice)

A fsm `[Trans, Final, Q0]` such that

for all `[Q,X,Qn]` *and* `[Q,X,Qn′]` *in* `Trans`, $\texttt{Qn} = \texttt{Qn}'$

is a *deterministic finite automaton* (DFA).

Determinization (eliminate choice)

A fsm `[Trans, Final, Q0]` such that

for all `[Q,X,Qn]` *and* `[Q,X,Qn′]` *in* `Trans`, $\texttt{Qn} = \texttt{Qn}'$

is a *deterministic finite automaton* (DFA).

Fact. Every fsm has a DFA accepting the same language.

Determinization (eliminate choice)

A fsm `[Trans, Final, Q0]` such that

for all `[Q,X,Qn]` *and* `[Q,X,Qn′]` *in* `Trans`, $\texttt{Qn} = \texttt{Qn}'$

is a *deterministic finite automaton* (DFA).

Fact. Every fsm has a DFA accepting the same language.

Proof: Subset (powerset) construction

Apply to `arc,goal`, contra `Trans,Final`:

```
arcD(NodeList,NextList) :-
        setof(Next, arcLN(NodeList,Next), NextList).

arcLN(NodeList,Next) :- member(Node,NodeList),
                        arc(Node,Next).
```

Determinization (eliminate choice)

A fsm `[Trans, Final, Q0]` such that

for all `[Q,X,Qn]` *and* `[Q,X,Qn′]` *in* `Trans`, $\texttt{Qn} = \texttt{Qn}'$

is a *deterministic finite automaton* (DFA).

Fact. Every fsm has a DFA accepting the same language.

Proof: Subset (powerset) construction

Apply to `arc,goal`, contra `Trans,Final`:

```
arcD(NodeList,NextList) :-
        setof(Next, arcLN(NodeList,Next), NextList).

arcLN(NodeList,Next) :- member(Node,NodeList),
                        arc(Node,Next).

goalD(NodeList):-member(Node,NodeList),goal(Node).
```

Determinization (eliminate choice)

A fsm `[Trans, Final, Q0]` such that

for all `[Q,X,Qn]` *and* `[Q,X,Qn']` *in* `Trans`, $\texttt{Qn} = \texttt{Qn}'$

is a *deterministic finite automaton* (DFA).

Fact. Every fsm has a DFA accepting the same language.

Proof: Subset (powerset) construction

Apply to `arc,goal`, contra `Trans,Final`:

```
arcD(NodeList,NextList) :-
        setof(Next, arcLN(NodeList,Next), NextList).

arcLN(NodeList,Next) :- member(Node,NodeList),
                        arc(Node,Next).

goalD(NodeList):-member(Node,NodeList),goal(Node).

searchD(NL) :- goalD(NL);
               (arcD(NL,NL2), searchD(NL2)).
```

Determinization (eliminate choice)

A fsm `[Trans, Final, Q0]` such that

for all `[Q,X,Qn]` *and* `[Q,X,Qn′]` *in* `Trans`, $\texttt{Qn} = \texttt{Qn}'$

is a *deterministic finite automaton* (DFA).

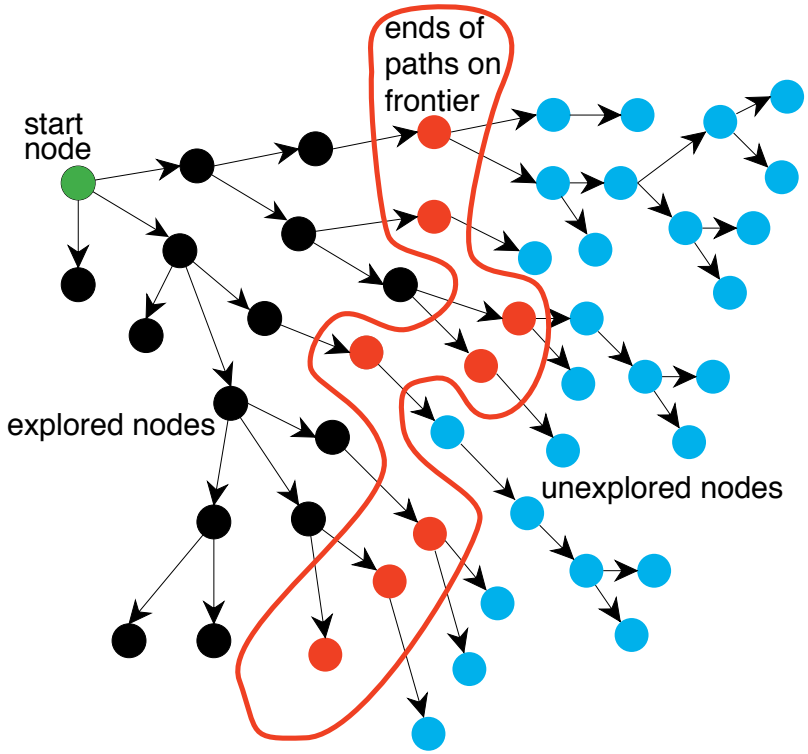
Fact. Every fsm has a DFA accepting the same language.

Proof: Subset (powerset) construction

Apply to `arc,goal`, contra `Trans,Final`:

```
arcD(NodeList,NextList) :-
      setof(Next, arcLN(NodeList,Next), NextList).

arcLN(NodeList,Next) :- member(Node,NodeList),
                        arc(Node,Next).

goalD(NodeList):-member(Node,NodeList),goal(Node).

searchD(NL) :- goalD(NL);
               (arcD(NL,NL2), searchD(NL2)).

search(Node) :- searchD([Node]).
```

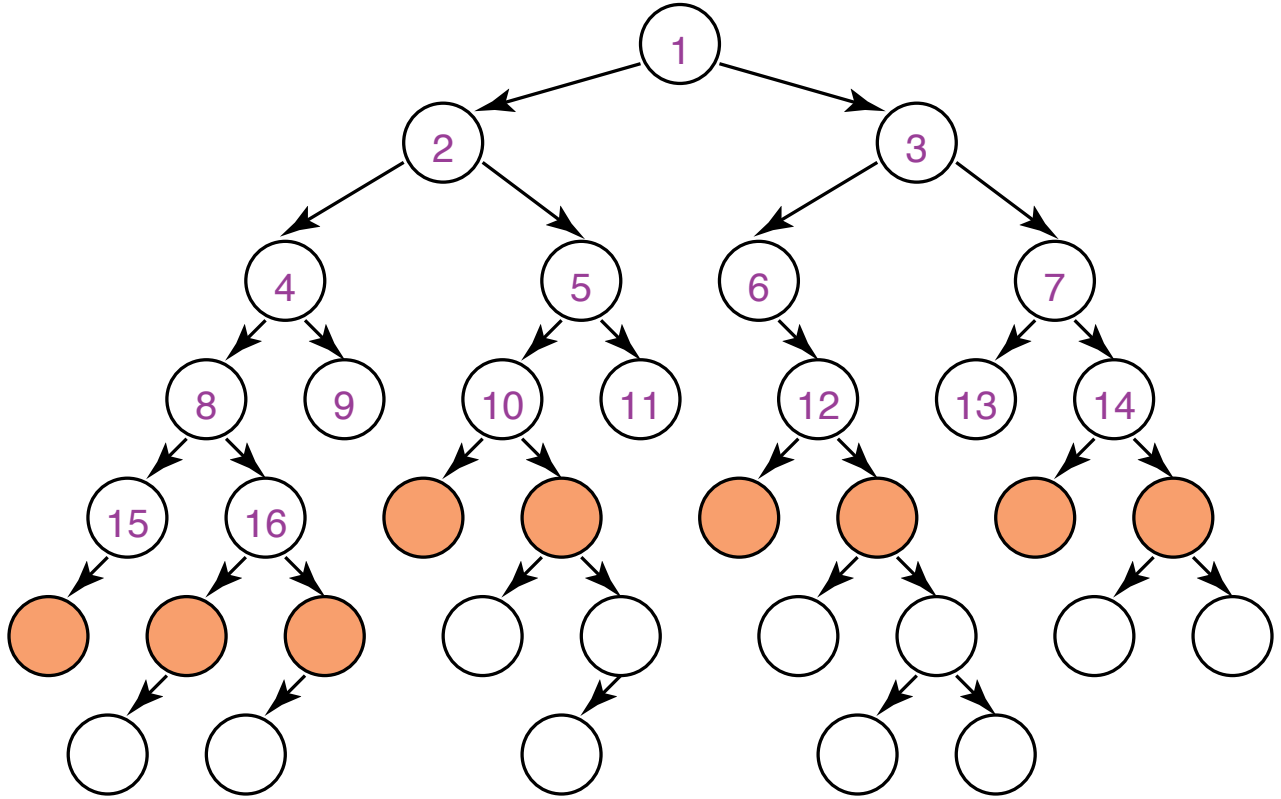
Frontier search



Poole & Mackworth

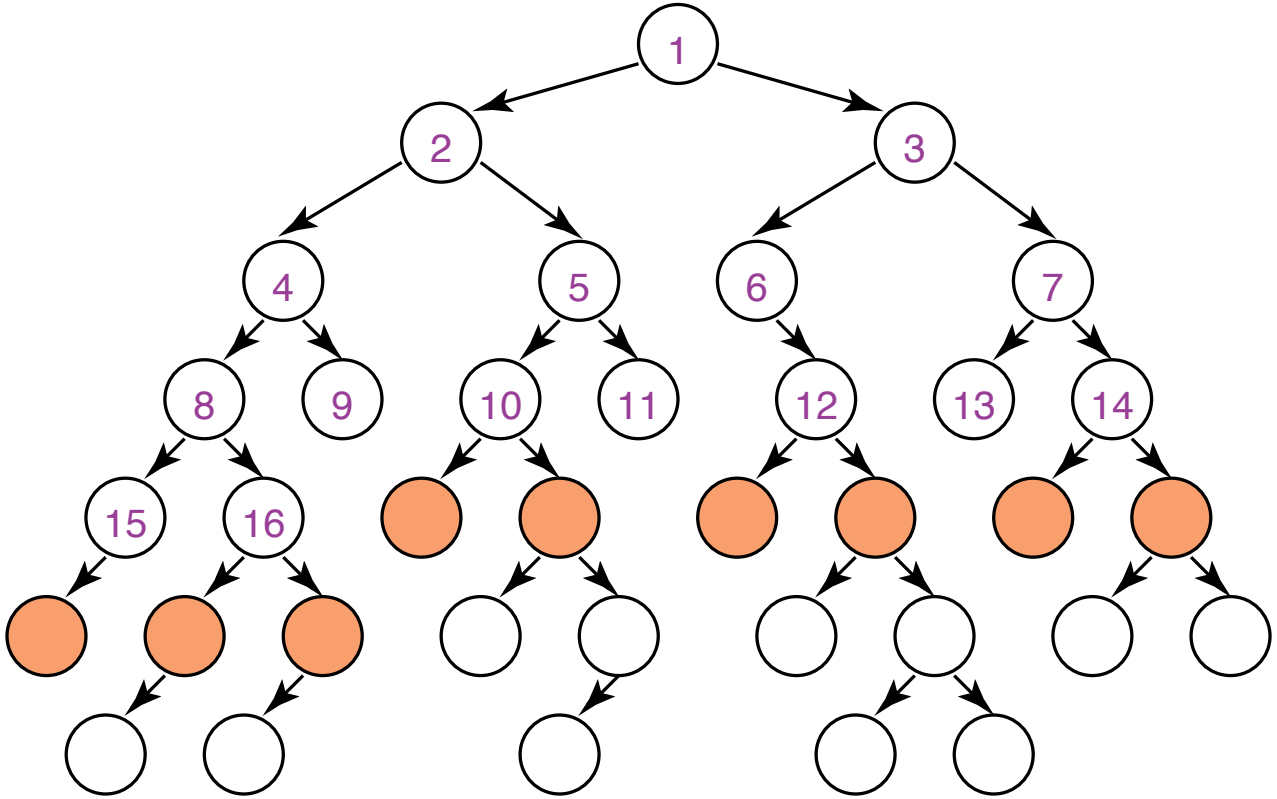
```
search(Node) :- frontierSearch([Node]).

frontierSearch([Node|_]) :- goal(Node).

frontierSearch([Node|Rest]) :-
            findall(Next, arc(Node,Next), Children),
            add2frontier(Children,Rest,NewFrontier),
            frontierSearch(NewFrontier).
```

Breadth-first: queue (FIFO)



[1] $\rightsquigarrow$ [2,3] $\rightsquigarrow$ [3,4,5] $\rightsquigarrow$ [4,5,6,7] $\rightsquigarrow \cdots$

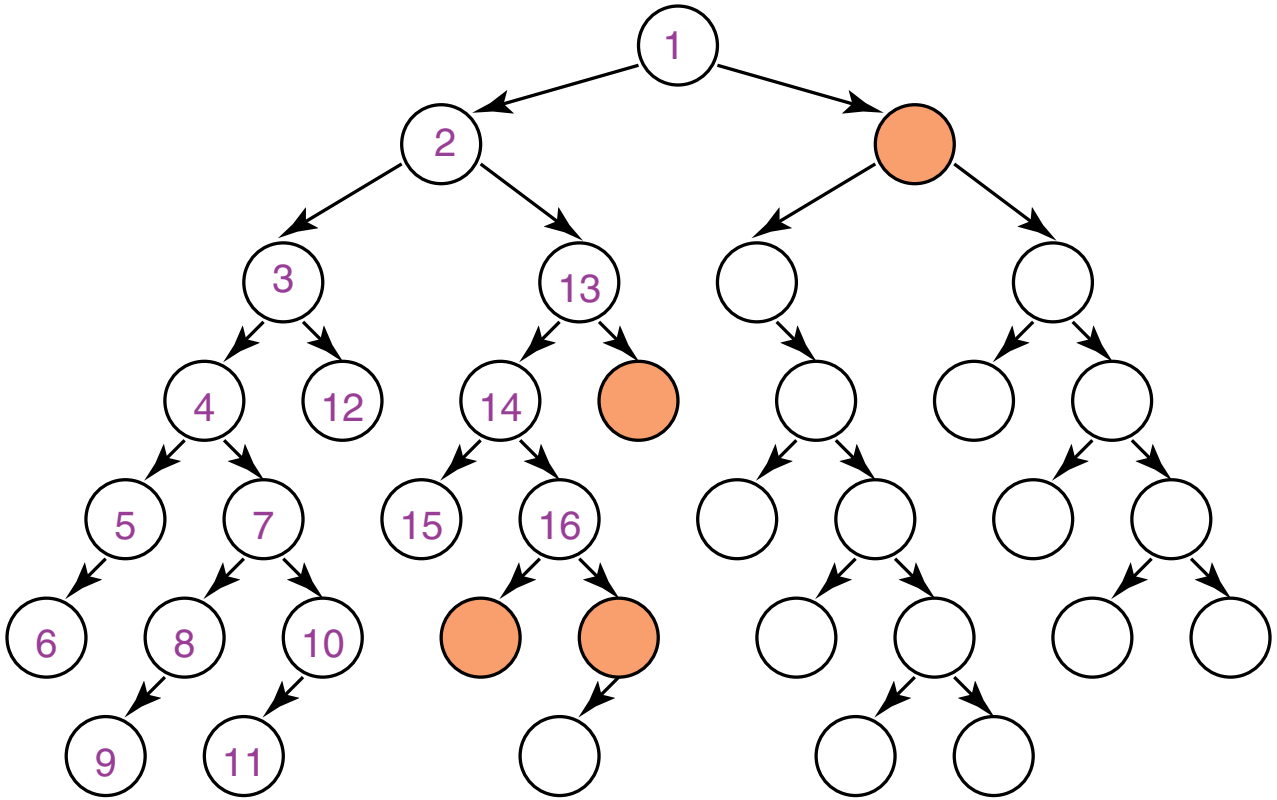
Breadth-first: queue (FIFO)



[1] $\rightsquigarrow$ [2,3] $\rightsquigarrow$ [3,4,5] $\rightsquigarrow$ [4,5,6,7] $\rightsquigarrow \cdots$

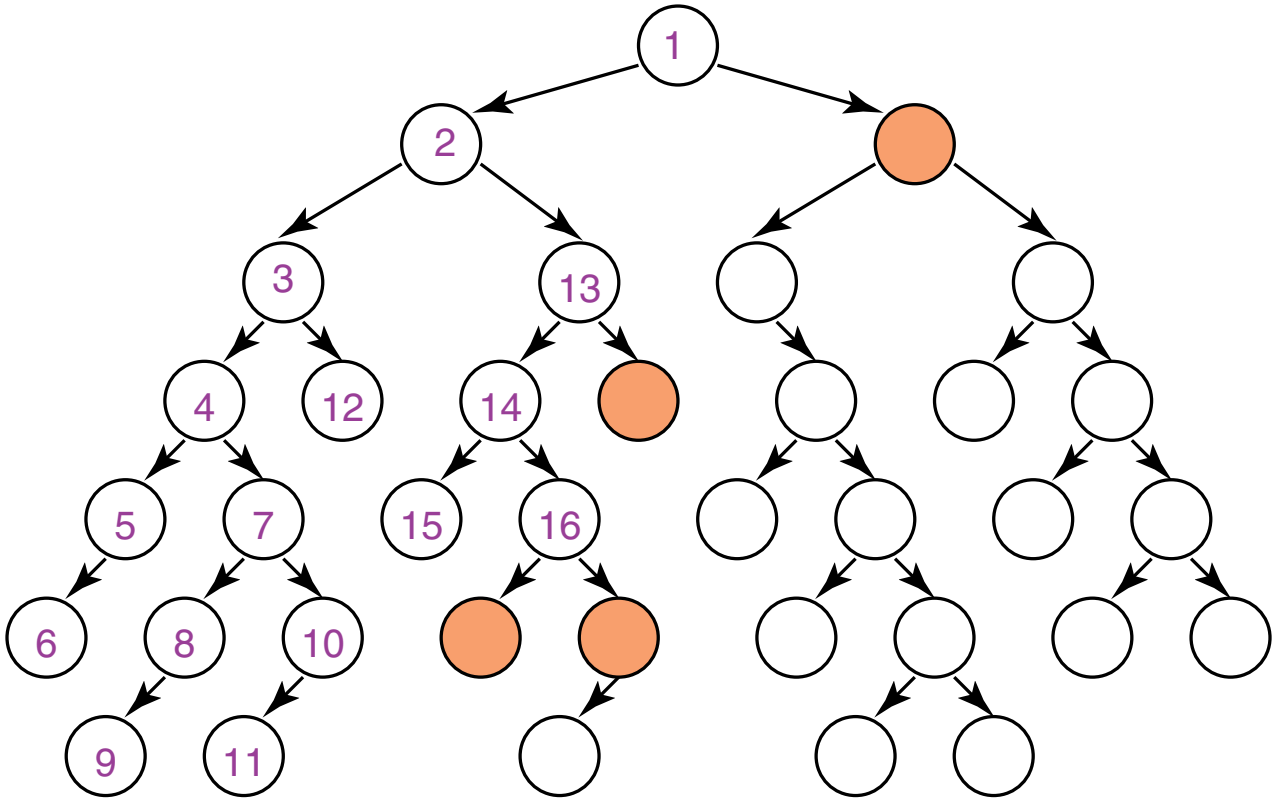
```
add2frontier(Children,[],Children).

add2frontier(Children,[H|T],[H|More]) :-
                        add2frontier(Children,T,More).
```

Depth-first: stack (LIFO)



$[1] \rightsquigarrow [2,\bullet] \rightsquigarrow [3,13,\bullet] \rightsquigarrow [4,12,13,\bullet] \rightsquigarrow \cdots$

Depth-first: stack (LIFO)



[1] $\rightsquigarrow$ [2,•] $\rightsquigarrow$ [3,13,•] $\rightsquigarrow$ [4,12,13,•] $\rightsquigarrow \cdots$

```
add2frontier([],Rest,Rest).

add2frontier([H|T],Rest,[H|TRest]) :-
                        add2frontier(T,Rest,TRest).
```

If-then-else and cut !

```
i :- p,!,q.

i :- r.

p.

r.

_____________

| ?- i.
```

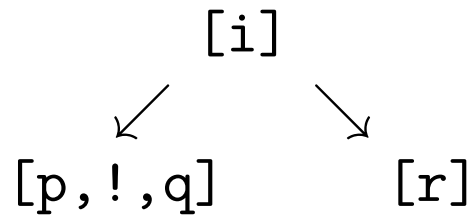
If-then-else and cut !

```
i :- p,!,q.                    [i]

i :- r.

p.

r.

-----------------

| ?- i.
```

If-then-else and cut !

```
i :- p,!,q.

i :- r.

p.

r.
```

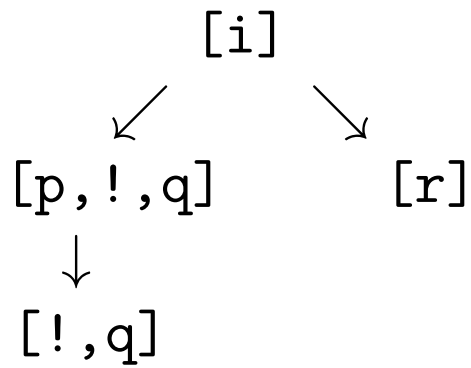
```
| ?- i.
```



If-then-else and cut !

```
i :- p,!,q.

i :- r.

p.

r.
```

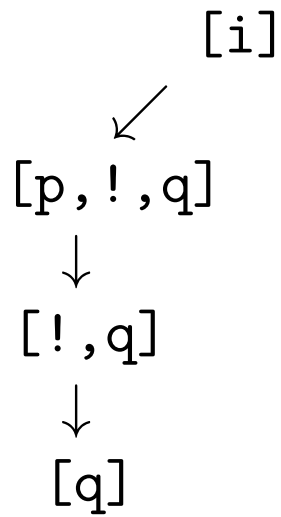
```
| ?- i.
```



Cut ! is true but destroys backtracking.

If-then-else and cut !

```
i :- p,!,q.

i :- r.

p.

r.
```

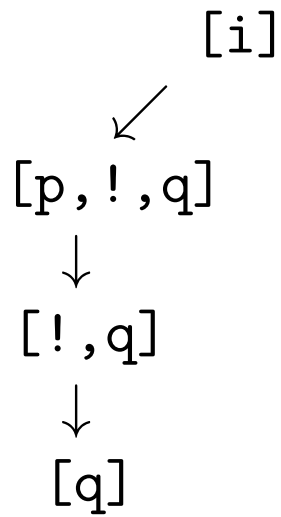


```
| ?- i.
```

Cut ! is true but destroys backtracking.

If-then-else and cut !

```
i :- p,!,q.

i :- r.

p.

r.
```



```
| ?- i.

no
```

Cut ! is true but destroys backtracking.

Review: Depth-first as frontier search

```
prove([],_).       % goal([]).
prove(Node,KB) :- arc(Node,Next,KB), prove(Next,KB).
```

Review: Depth-first as frontier search

```
prove([],_).      % goal([]).
prove(Node,KB) :- arc(Node,Next,KB), prove(Next,KB).
```

```
fs([[]|_],_).

fs([Node|More],KB) :- findall(X,arc(Node,X,KB),L),
                      append(L,More,NewFrontier),
                      fs(NewFrontier,KB).
```

Review: Depth-first as frontier search

```
prove([],_).      % goal([]).
prove(Node,KB) :- arc(Node,Next,KB), prove(Next,KB).
```

```
fs([[]|_],_).

fs([Node|More],KB) :- findall(X,arc(Node,X,KB),L),
                      append(L,More,NewFrontier),
                      fs(NewFrontier,KB).
```

Cut?

Tracking the frontier

```
[[i]]
```

```
i :- p,!,q.
```

```
i :- r.
```

```
p.
```

```
r.
```

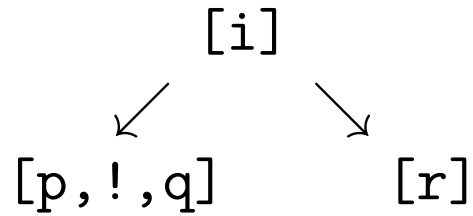
```
| ?- i.
```

```
[i]
```

Tracking the frontier

`[[i]]` $\rightsquigarrow$ `[[p,!,q],[r]]`

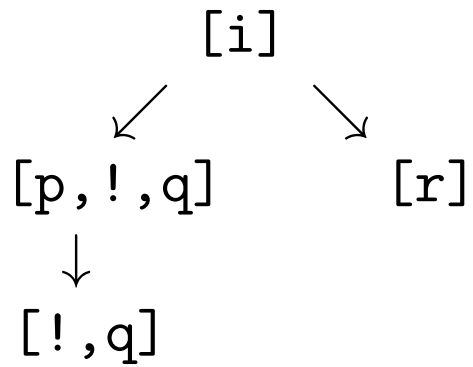
```
i :- p,!,q.

i :- r.

p.

r.

---------------

| ?- i.
```



Tracking the frontier

`[[i]]` $\rightsquigarrow$ `[[p,!,q],[r]]` $\rightsquigarrow$ `[[!,q],[r]]`

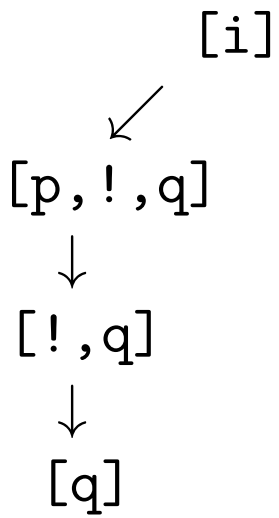
```
i :- p,!,q.

i :- r.

p.

r.

-------------

| ?- i.
```



Tracking the frontier

`[[i]]` $\rightsquigarrow$ `[[p,!,q],[r]]` $\rightsquigarrow$ `[[!,q],[r]]` $\rightsquigarrow$ `[[q]]`

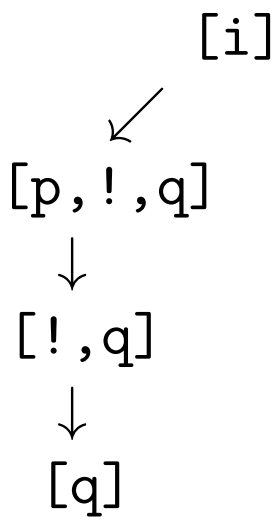
```
i :- p,!,q.

i :- r.

p.

r.

-----------------

| ?- i.
```



Tracking the frontier

`[[i]]` $\rightsquigarrow$ `[[p,!,q],[r]]` $\rightsquigarrow$ `[[!,q],[r]]` $\rightsquigarrow$ `[[q]]` $\rightsquigarrow$ `[]`

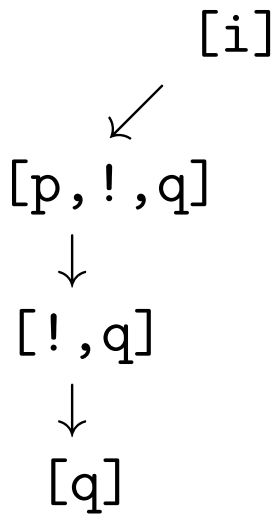
```
i :- p,!,q.

i :- r.

p.

r.

-----------------

| ?- i.
```



Tracking the frontier

`[[i]]` $\rightsquigarrow$ `[[p,!,q],[r]]` $\rightsquigarrow$ `[[!,q],[r]]` $\rightsquigarrow$ `[[q]]` $\rightsquigarrow$ `[]`

```
i :- p,!,q.

i :- r.

p.

r.
```



```
| ?- i.

no
```

Cut via frontier depth-first search

```
fs([[]|_],_).


fs([Node|More],KB) :-
                        findall(X,arc(Node,X,KB),L),
                        append(L,More,NewFrontier),
                        fs(NewFrontier,KB).
```

Cut via frontier depth-first search

```
fs([[]|_],_).

fs([[cut|T]|_],KB)) :- fs([T],KB).

fs([Node|More],KB) :-
                        findall(X,arc(Node,X,KB),L),
                        append(L,More,NewFrontier),
                        fs(NewFrontier,KB).
```

Cut via frontier depth-first search

```
fs([[]|_],_).

fs([[cut|T]|_],KB)) :- fs([T],KB).

fs([Node|More],KB) :-  Node = [H|_], H\== cut,
                       findall(X,arc(Node,X,KB),L),
                       append(L,More,NewFrontier),
                       fs(NewFrontier,KB).
```

Cut via frontier depth-first search

```
fs([[]|_],_).

fs([[cut|T]|_],KB)) :- fs([T],KB).

fs([Node|More],KB) :-  Node = [H|_], H\== cut,
                       findall(X,arc(Node,X,KB),L),
                       append(L,More,NewFrontier),
                       fs(NewFrontier,KB).


if(p,q,r) :- (p,!,q); r.       % contra (p,q);r
```

Cut via frontier depth-first search

```
fs([[]|_],_).

fs([[cut|T]|_],KB)) :- fs([T],KB).

fs([Node|More],KB) :-  Node = [H|_], H\== cut,
                       findall(X,arc(Node,X,KB),L),
                       append(L,More,NewFrontier),
                       fs(NewFrontier,KB).


if(p,q,r) :- (p,!,q); r.       % contra (p,q);r

negation-as-failure(p) :- (p,!,fail); true.
```