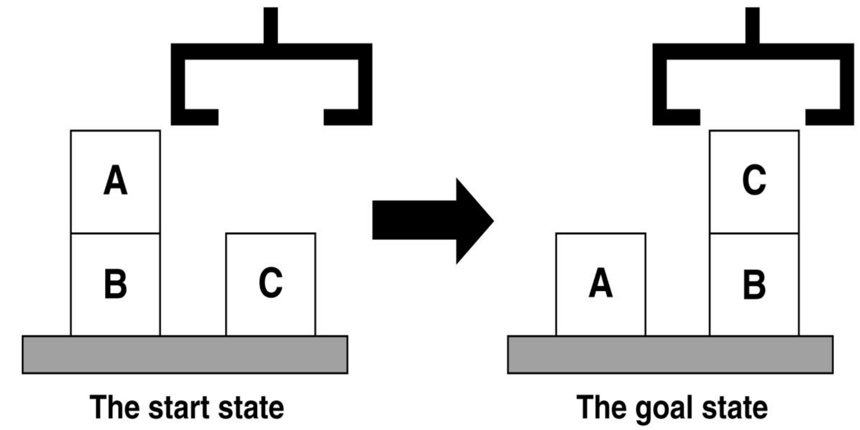
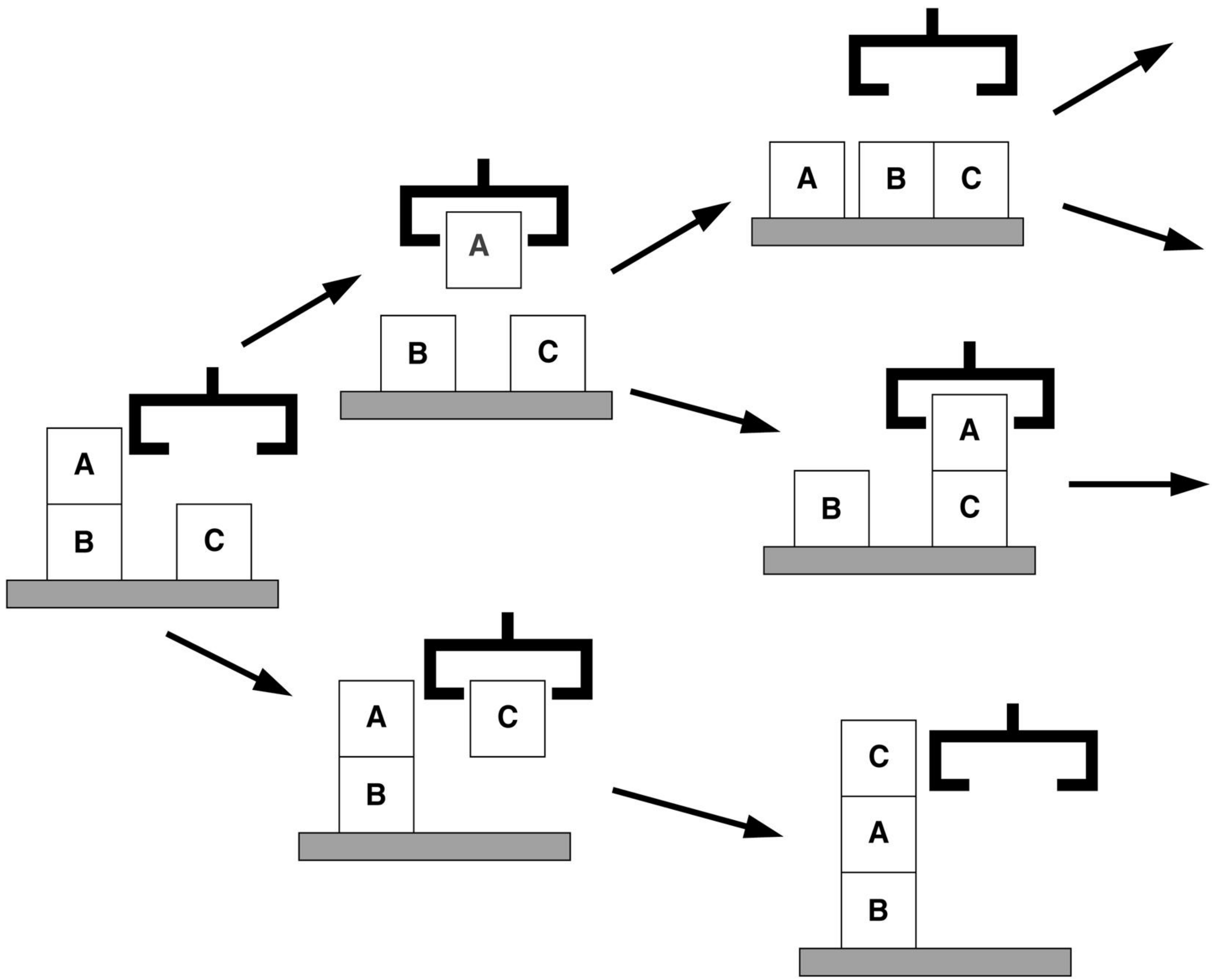
Declarative Programming

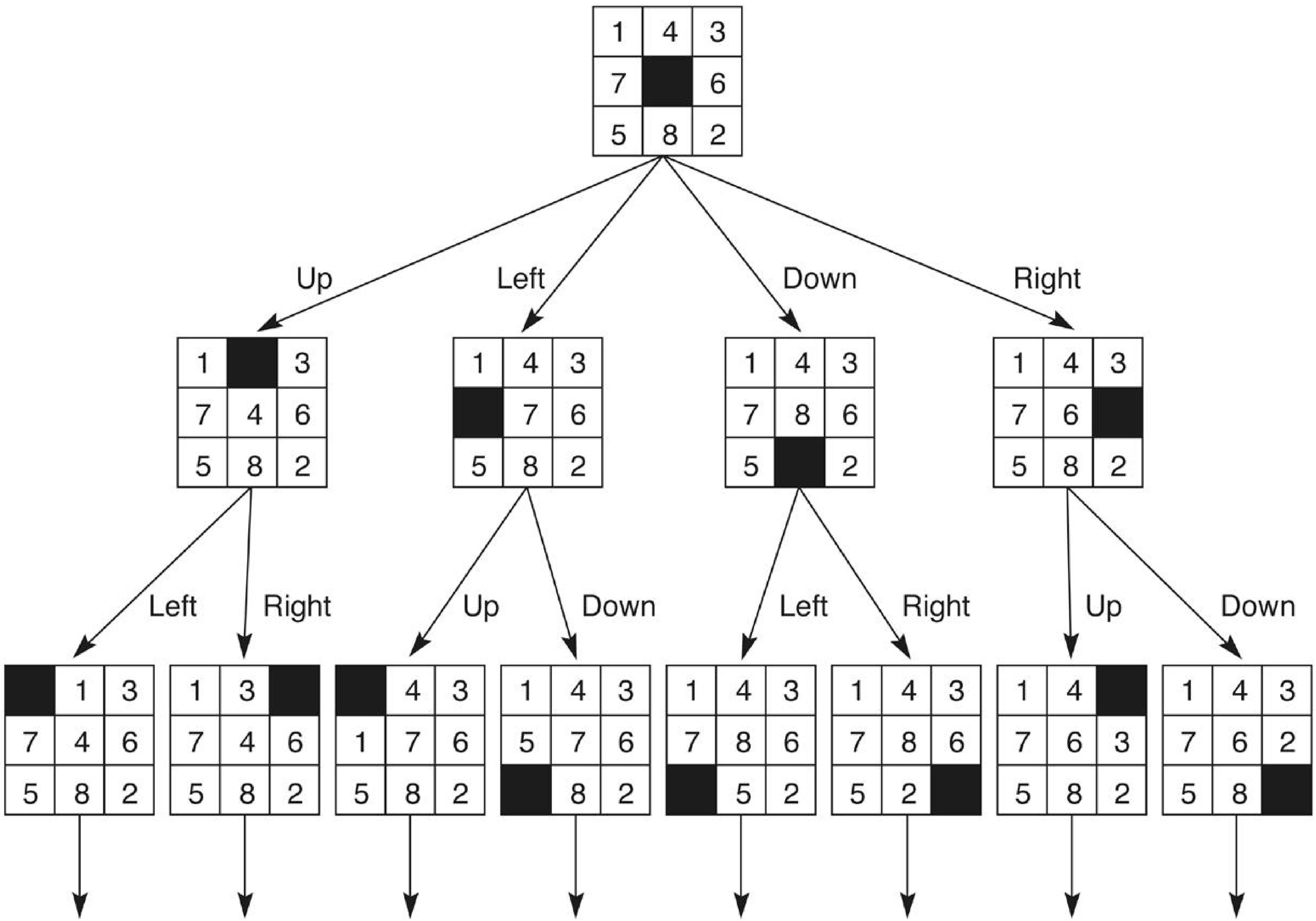
4: blind and informed search of state space, proving as search process

State space search: *blocks world*





State space search:
8-puzzle



State space search:
graph representation

state space

state=node, state transition=arc

goal nodes and start nodes

cost associated with arcs between nodes

solution

path from start to goal node

optimal if cost over path is minimal

search algorithms

completeness: will a solution always be found if there is one?

optimality: will highest-quality solution be found when there are several?

efficiency: runtime and memory requirements

blind vs informed: does quality of partial solutions steer process?

State space search:
Prolog skeleton for search algorithms

reached, but untested states

goal state for which `goal(Goal)` succeeds

succeeds if the goal state Goal can be reached from a state on the Agenda

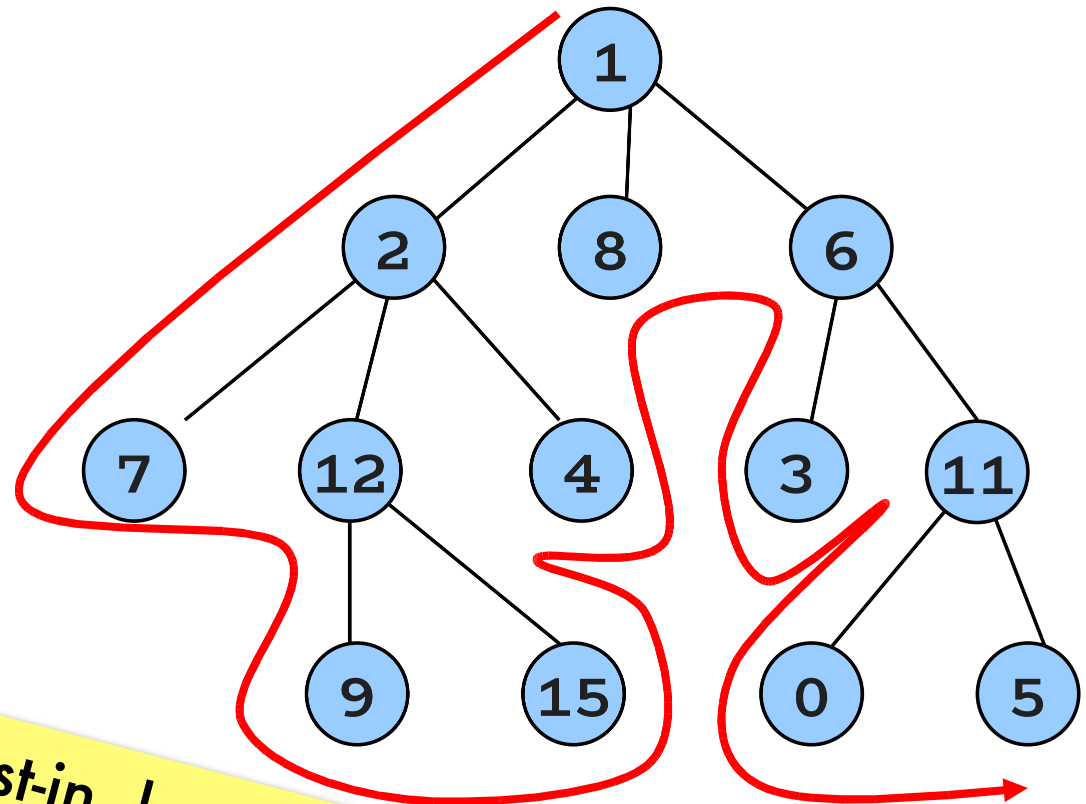
```
search(Agenda,Goal):-
  next(Agenda,Goal,Rest),
  goal(Goal).

search(Agenda,Goal):-
  next(Agenda,Current,Rest),
  children(Current,Children),
  add(Children,Rest,NewAgenda),
  search(NewAgenda,Goal).
```

selects a candidate state from the Agenda

expands the current state

State space search: *depth-first search*

```
arc(1,2). arc(1,8). arc(1,6).
arc(2,7). arc(2,12). arc(2,4).
arc(12,9). arc(12,15). arc(6,3).
arc(6,11). arc(11,0). arc(11,5).
```



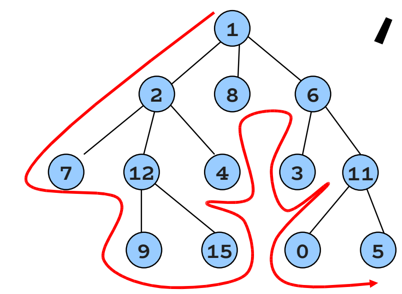
next/3 implemented by taking first element of list

first-in, last-out
agenda treated as a stack

```
search_df([Goal|Rest],Goal):-
  goal(Goal).

search_df([Current|Rest],Goal):-
  children(Current,Children),
  append(Children,Rest,NewAgenda),
  search_df(NewAgenda,Goal).

children(Node,Children):-
  findall(C,arc(Node,C),Children).
```

add/3 implemented by prepending children of first element on agenda to the remainder of the agenda

State space search: *depth-first search with paths*



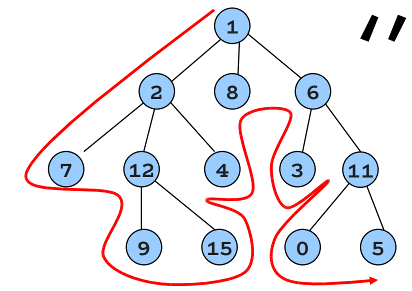
keep path to node on agenda, rather than node

only requires a change to children/3
AND
way search_df/2 is called

```
children([Node|RestOfPath],Children):-
  findall([Child,Node|RestOfPath],arc(Node,Child),Children).
```

```
?- search_df([[initial_node]],PathToGoal).
```

State space search:
depth-first search with loop detection



keep list of
visited nodes

```
search_df_loop([Goal|Rest],Visited,Goal):-
  goal(Goal).
search_df_loop([Current|Rest],Visited,Goal):-
  children(Current,Children),
  add_df(Children,Rest,Visited,NewAgenda),
  search_df_loop(NewAgenda, [Current|Visited],Goal).
```

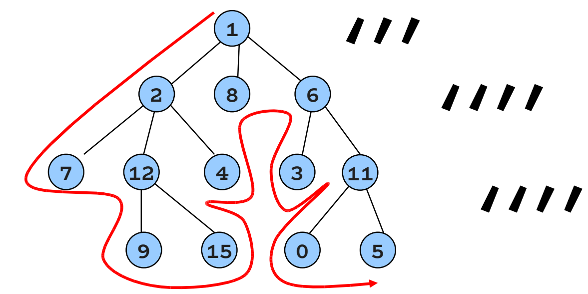
add current node to list of visited nodes

```
add_df([],Agenda,Visited,Agenda).
add_df([Child|Rest],OldAgenda,Visited, [Child|NewAgenda]):-
  not(element(Child,OldAgenda)),
  not(element(Child,Visited)),
  add_df(Rest,OldAgenda,Visited,NewAgenda).
add_df([Child|Rest],OldAgenda,Visited,NewAgenda):-
  element(Child,OldAgenda),
  add_df(Rest,OldAgenda,Visited,NewAgenda).
add_df([Child|Rest],OldAgenda,Visited,NewAgenda):-
  element(Child,Visited),
  add_df(Rest,OldAgenda,Visited,NewAgenda).
```

do not add a child if it's already on the agenda

do not add already visited children

State space search:
depth-first search using Prolog stack



vanilla

```
search_df(Goal,Goal):-
   goal(Goal).
search_df(CurrentNode,Goal):-
   arc(CurrentNode,Child),
   search_df(Child,Goal).
```

use Prolog call stack as agenda

might loop on cycles

depth bounded

```
search_bd(Depth,Goal,Goal):-
   goal(Goal).
search_bd(Depth, CurrentNode, Goal):-
   Depth>0,
   NewDepth is Depth-1,
   arc(CurrentNode, Child),
   search_bd(NewDepth, Child, Goal).
```

```
?- search_df(10,initial_node,Goal).
```

do not exceed depth threshold while searching

always halts, but no solutions beyond threshold

iterative deepening

```
search_id(CurrentNode, Goal):-
   search_id(1,CurrentNode,Goal).
search_id(Depth,CurrentNode,Goal):-
   search_bd(Depth,CurrentNode,Goal).
search_id(Depth,CurrentNode,Goal):-
   NewDepth is Depth+1,
   search_id(NewDepth,CurrentNode,Goal)
```

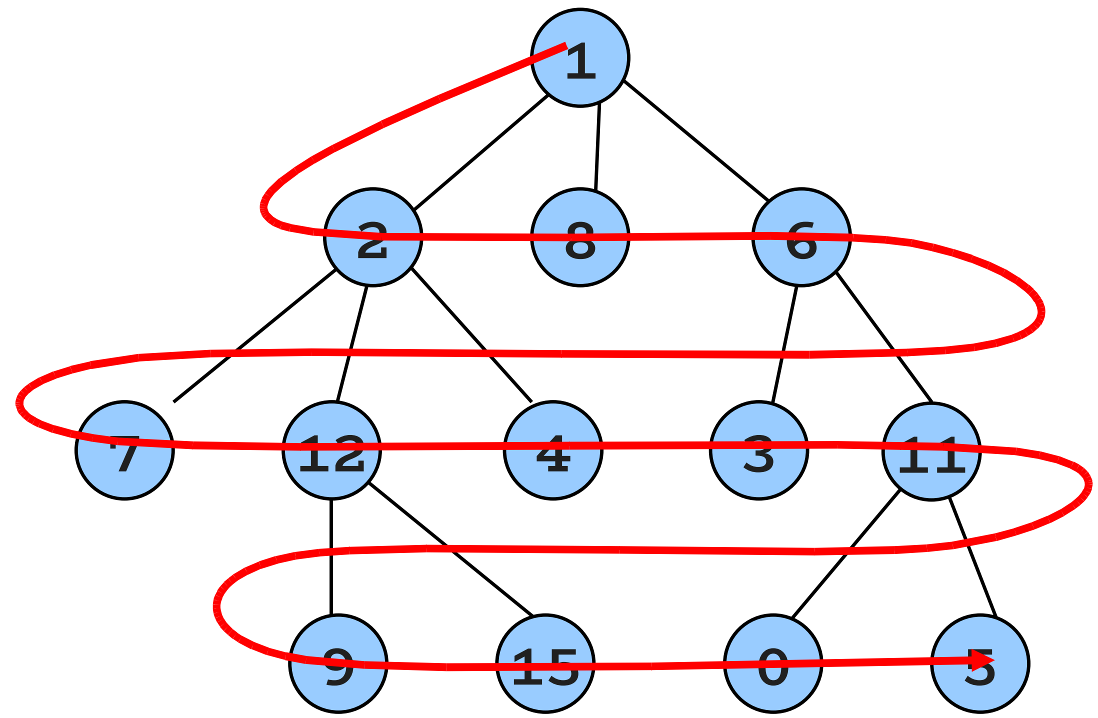
less memory than bfs

increase depth bound on each iteration

complete and solutions on, but upper parts of search space

not that bad for full trees: number of nodes at a single level is smaller than all nodes above it

State space search: *breadth-first search*



next/3 implemented by taking first element of list

```
search_bf([Goal|Rest],Goal):-
  goal(Goal).
search_bf([Current|Rest],Goal):-
  children(Current,Children),
  append(Rest,Children,NewAgenda),
  search_bf(NewAgenda,Goal).


children(Node,Children):-
  findall(C,arc(Node,C),Children).
```

first-in, first-out
agenda treated as a queue

add/3 implemented by appending children of first element on agenda to the remainder of the agenda

State space search:
dfs vs bfs

spirals away from start node,
candidate paths to be remembered
grows exponentially with depth

l=depth-limit
b=branching factor of search space
d=depth of search space
m=depth of shortest path solution

	breadth-first	depth-first	depth-limited	iterative deepening
time	b^d	b^m	b^l	b^d
space	b^d	bm	bl	bd
shortest solution path	√			√
complete	√		√ if $l \geq d$	√

might be second child of root node

State space search:
water jugs problem

[The Art of Prolog, Sterling and Shapiro]

20L

5L 8L

operations

- fill a jug from the pool
- empty a jug into the pool
- pour one jug into another until one poured from is empty or the one poured into is full

goal

4L in a jug

State space search: *implementing the search*

[The Art of Prolog, Sterling and Shapiro]

as a generic algorithm for state space problems

visited states

sequence of transitions to reach goal from current state

```
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).

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

until now, we only had unnamed arcs

multiple named transitions out of a state

State space search:
encoding water jugs problem

[The Art of Prolog, Sterling and Shapiro]

starting and goal states

```
initial_state(jugs,jugs(0,0)).
final_state(jugs(4,V2)).
final_state(jugs(V1,4)).
```

possible transitions out of a state

```
move(jugs(V1,V2),fill(1)).
move(jugs(V1,V2),fill(2)).
move(jugs(V1,V2),empty(1)) :- V1>0.
move(jugs(V1,V2),empty(2)) :- V2>0.
move(jugs(V1,V2),transfer(2,1)).
move(jugs(V1,V2),transfer(1,2)).
```

empty first jug (1), but only if it still contains water (C1)

State space search:
encoding water jugs problem

[The Art of Prolog, Sterling and Shapiro]

states a transition can lead to

```
update(jugs(V1,V2),fill(1),jugs(C1,V2)) :-
  capacity(1,C1).
update(jugs(V1,V2),fill(2),jugs(V1,C2)) :-
  capacity(2,C2).
update(jugs(V1,V2),empty(1),jugs(0,V2)).
update(jugs(V1,V2),empty(2),jugs(V1,0)).
update(jugs(V1,V2),transfer(2,1),jugs(W1,W2)) :-
  capacity(1,C1),
  Liquid is V1 + V2,
  Excess is Liquid - C1,
  adjust(Liquid,Excess,W1,W2).
update(jugs(V1,V2),transfer(1,2),jugs(W1,W2)) :-
  capacity(2,C2),
  Liquid is V1 + V2,
  Excess is Liquid - C2,
  adjust(Liquid,Excess,W2,W1).
```

a jug can be filled up to its capacity from the pool

the first jug will contain 0L after emptying it

the first jug can be poured in the second

```
adjust(Liquid, Excess,Liquid,0) :- Excess =< 0.
adjust(Liquid,Excess,V,Excess) :-
  Excess > 0,
  V is Liquid - Excess.
```

```
capacity(j1,8).
capacity(j2,5).
legal(jugs(C1,C2)).
```

Proving as a search process:
df agenda-based meta-interpreter

true: empty conjunctions
single term: singleton conjunction

```
prove(true):- !.
prove((A,B)):-
  !,
  clause(A,C),
  conj_append(C,B,D),
  prove(D).
prove(A):-
  clause(A,B),
  prove(B).
```

instead of
prove((A,B)) :-
prove(A),prove(B)

```
conj_append(true,Ys,Ys).
conj_append(X,Ys,(X,Ys)):-
  not(X=true),
  not(X=(One,TheOther)).
conj_append((X,Xs),Ys,(X,Zs)):-
  conj_append(Xs,Ys,Zs).
```

depth-first

```
prove_df_a(Goal) :-
  prove_df_a([Goal]).
prove_df_a([true|Agenda]).
prove_df_a([(A,B)|Agenda]) :-
  !,
  findall(D,(clause(A,C),conj_append(C,B,D)),Children),
  append(Children,Agenda,NewAgenda),
  prove_df_a(NewAgenda).
prove_df_a([A|Agenda]) :-
  findall(B,clause(A,B),Children),
  append(Children,Agenda,NewAgenda),
  prove_df_a(NewAgenda).
```

swapping arguments of append/3 turns this into a breadth-first meta-interpreter!

Proving as a search process: *bf agenda-based meta-interpreter*

This time with answer substitution.

```
foo(X) :- bar(X).
```

problem:
findall(Term,Goal,List)
creates new variables in the instantiation of Term for the unbound variables in answers to Goal

```
?- findall(Body,clause(foo(Z),Body),Bodies).
Bodies = [bar(_G336)].
```

trick:
store a(Literals,OriginalGoal) on agenda where OriginalGoal is a copy of the Goal

breadth-first

```
prove_bf(Goal):-
  prove_bf_a([a(Goal,Goal)],Goal).
prove_bf_a([a(true,Goal)|Agenda],Goal).
prove_bf_a([a((A,B),G)|Agenda],Goal):-!,
  findall(a(D,G),(clause(A,C),conj_append(C,B,D)),Children),
  append(Agenda,Children,NewAgenda),
prove_bf_a(NewAgenda,Goal).
prove_bf_a([a(A,G)|Agenda],Goal):-
  findall(a(B,G),clause(A,B),Children),
  append(Agenda,Children,NewAgenda),
  prove_bf_a(NewAgenda,Goal).
```

Goal will be instantiated with the correct answer substitutions

Proving as a search process: *forward vs backward chaining of if-then rules*

backward chaining	forward chaining
from head to body	from body to head
search starts from where we want to be towards where we are	search starts from where we are to where we want to be
e.g. Prolog query answering	e.g. model construction

what's more efficient depends on structure of search space (cf. discussion on practical uses of var)

Proving as a search process:
forward chaining - bottom-up model construction

model of clauses defined by cl/1

grounds literal from head

no more violated clauses (note the !)

grounds literal from body

```
model(M):- model([],M).
model(M0,M):-
  is_violated(Head,M0),!,
  disj_element(L,Head),
  model([L|M0],M).
model(M,M).

is_violated(H,M) :-
  cl((H:-B)),
  satisfied_body(B,M),
  not(satisfied_head(H,M)).
```

add a literal from the head of a violated clause to the current model

a violated clause:
body is true in the current model,
but the head not

Proving as a search process:
forward chaining - auxiliaries

body is a conjunction of literals

```
satisfied_body(true,M).
satisfied_body(A,M) :-
  element(A,M).
satisfied_body((A,B),M) :-
  element(A,M),
  satisfied_body(B,M).
```

single disjunct

```
disj_element(X,X):-
  not(X=false),
  not(X=(One;TheOther)).
disj_element(X,(X;Ys)).
disj_element(X,(Y;Ys)):-
  disj_element(X,Ys).
```

false = empty disjunction

, and ; are right-associative operators:
a;b;c=;(a,;(b,c))

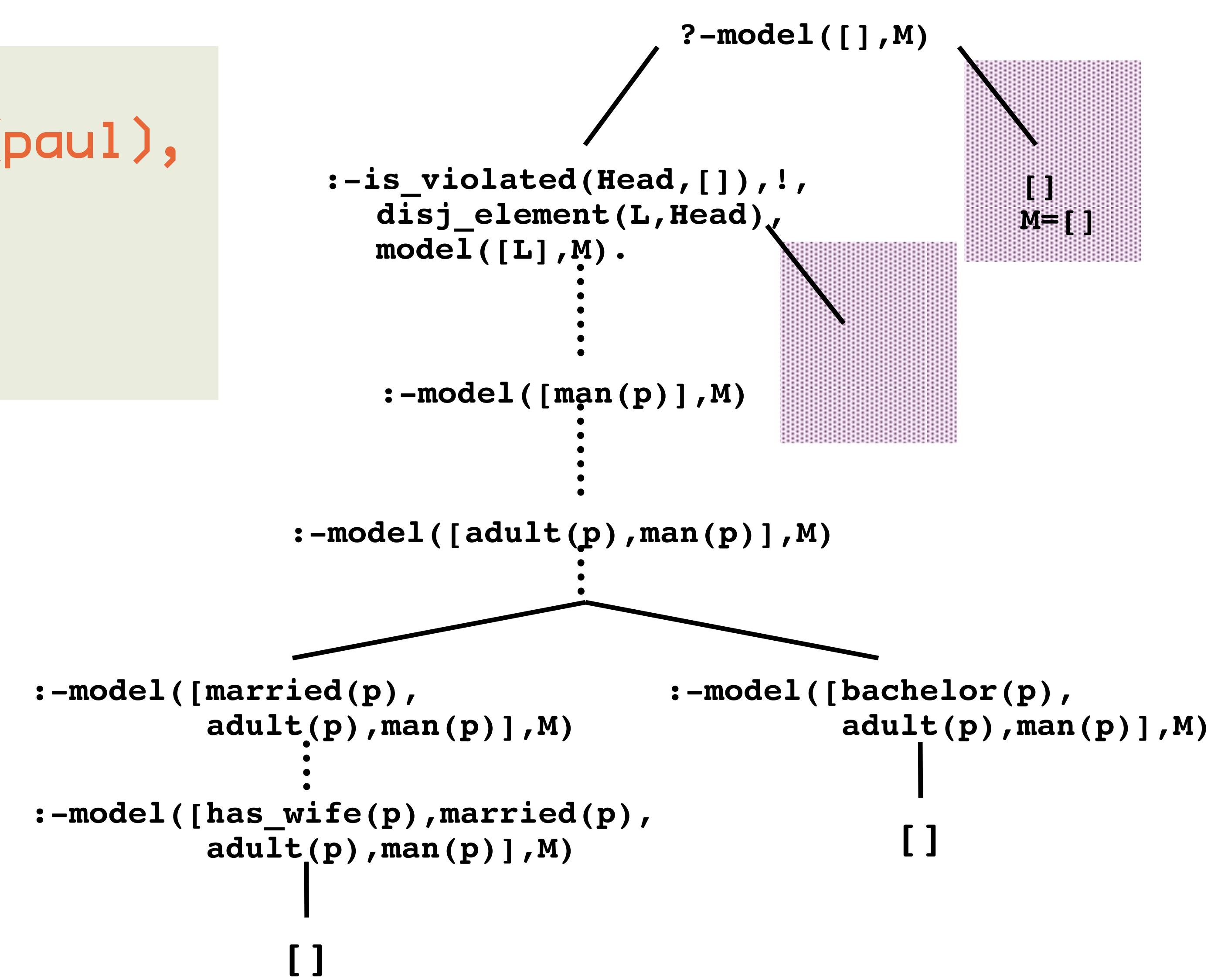
```
satisfied_head(A,M):-
  element(A,M).
satisfied_head((A;B),M) :-
  element(A,M).
satisfied_head((A;B),M) :-
  satisfied_head(B,M).
```

Proving as a search process:
forward chaining - example

```
cl((married(X);bachelor(X):-man(X),adult(X)).
cl((has_wife(X):-married(X),man(X))).
cl((man(paul):-true)).
cl((adult(paul):-true)).
```

```
?- model(M)
M = [has_wife(paul),married(paul),
     adult(paul),man(paul)];
M = [bachelor(paul),
     adult(paul),
     man(paul)]
```

two minimal models as there is a disjunction in the head



Proving as a search process:
forward chaining - range-restricted clauses

Our simple forward chainer cannot construct a model for following clauses:

```
cl((man(X);woman(X):-true)).
cl((false:-man(maria))).
cl((false:-woman(peter))).
```

an unground man(X) will be added to the model, which leads to the second clause being violated –which cannot be solved as it has an empty head

works only for clauses for which grounding the body also grounds the head

add literal to first clause, to enumerate possible values of X

```
cl((man(X);woman(X):-person(X))).
cl((person(maria):-true)).
cl((person(peter):-true)).
cl((false:-man(maria))).
cl((false:-woman(peter))).
```

range-restricted clause:
all variables in head also occur in body

can be ensured by adding predicates that quantify over each variable's domain

```
?- model(M)
M = [man(peter),person(peter),woman(maria),person(maria)]
```

Proving as a search process:
forward chaining - subsets of infinite models

```
cl((append([],Y,Y):-list(Y))).
cl((append([X|Xs],Ys,[X|Zs]):-thing(X),append(Xs,Ys,Zs))).
cl((list([]):-true)).
cl((list([X|Y]):-thing(X),list(Y))).
cl((thing(a):-true)).
cl((thing(b):-true)).
cl((thing(c):-true)).
```

range-restricted version of append/3

```
model_d(D,M):-
  model_d(D,[],M).

model_d(0,M,M).
model_d(D,M0,M):-
  D>0,
  D1 is D-1,
  findall(H,is_violated(H,M0),Heads),
  satisfy_clauses(Heads,M0,M1),
  model_d(D1,M1,M).

satisfy_clauses([],M,M).
satisfy_clauses([H|Hs],M0,M):-
  disj_element(L,H),
  satisfy_clauses(Hs,[L|M0],M).
```

depth-bounded construction of submodel