Production Systems:

Forward and Backward Chaining

Motivations

- A production system consists of (1) a set of production rules, (2) the working memory and (3) the control structure.
- Previously, you saw exhaustive control structure. Check all rules, one by one.
- This lecture considers forward chaining and backward chaining for control structures.

Objectives

1. 8-puzzle as production system
2. Forward chaining
3. Backward chaining
4. Comparison
5. Prolog implementation of the farmer, wolf, goat and cabbage problem

3 components of the production system

1. A set of production rules
   - knowledge base
   - condition/action pairs
   - $p(x) \rightarrow q(x)$
2. Working memory
   - utility routines
   - data structures, queues, stacks
3. Control structure
   - recursive pattern-driven search

8-puzzle as a production system

Start state:

2	8	3
1	6	4
7		5

Goal state:

1	2	3
8		4
7	6	5

1. **Production set:**

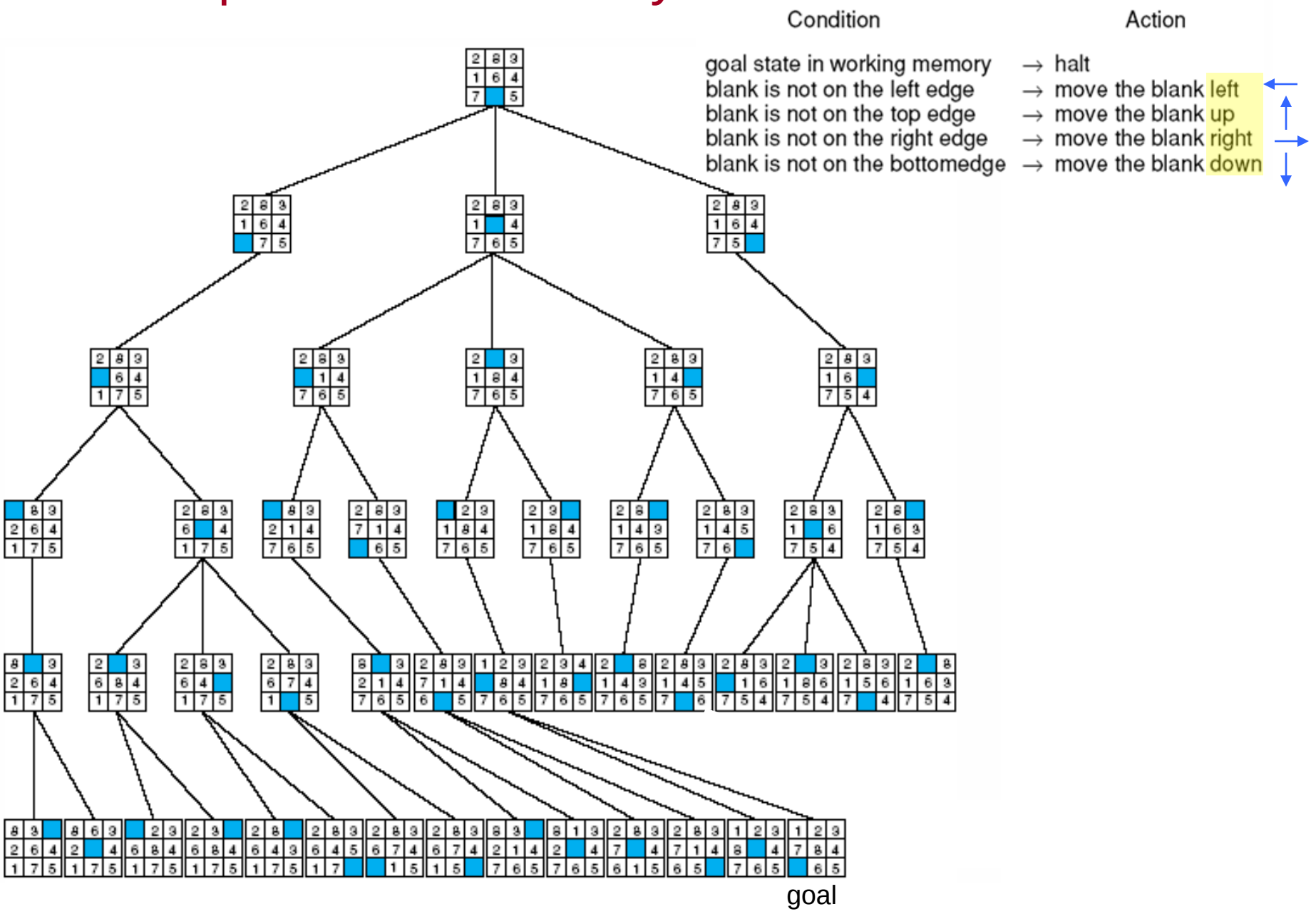
Condition		Action
goal state in working memory	→	halt
blank is not on the left edge	→	move the blank left
blank is not on the top edge	→	move the blank up
blank is not on the right edge	→	move the blank right
blank is not on the bottomedge	→	move the blank down

2. **Working memory is the present board state and goal state.**

3. **Control regime:**

1. Try each production in order.
2. Do not allow loops.
3. Stop when goal is found.

8-puzzle searched by a production system



Advantages of production systems

- Separation of Knowledge and Control
- A Natural Mapping onto State Space Search
- Pattern-Directed Control
- Opportunities for Heuristic Control of Search
- Tracing and Explanation
- Programming Language Independence
- A Plausible Model of Human Problem-Solving

Data-driven search: forward chaining

1

Production set:

1. $p \wedge q \rightarrow goal$
2. $r \wedge s \rightarrow p$
3. $w \wedge r \rightarrow q$
4. $t \wedge u \rightarrow q$
5. $v \rightarrow s$
6. $start \rightarrow v \wedge r \wedge q$

Trace of execution:

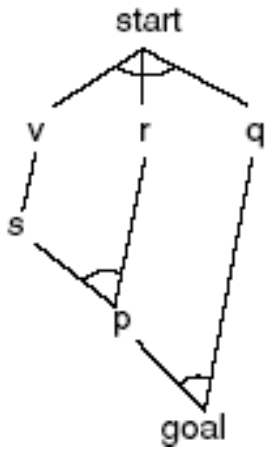
2

Iteration #	Working memory	Conflict set	Rule fired
0	start	6	6
1	start, v, r, q	6, 5	5
2	start, v, r, q, s	6, 5, 2	2
3	start, v, r, q, s, p	6, 5, 2, 1	1
4	start, v, r, q, s, p, goal	6, 5, 2, 1	halt

3 Fire the last rule in the set.

Space searched by execution:

Working memory contains true states.



Forward chaining
Direction of search

Goal-driven search: backward chaining

Production set:

1. p ∧ q → goal
2. r ∧ s → p
3. w ∧ r → p
4. t ∧ u → q
5. v → s
6. start → v ∧ r ∧ q

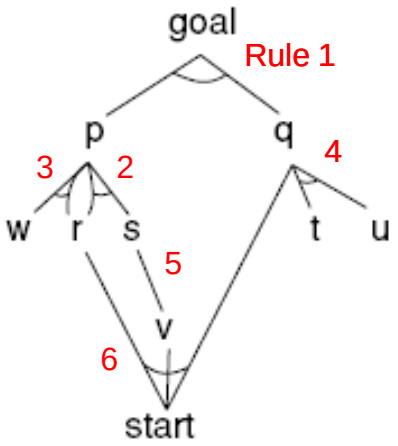
Trace of execution:

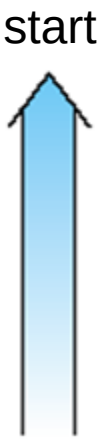
subgoals — oldest untried rule

Iteration #	Working memory	Conflict set	Rule fired
0	goal	1	1
1	goal, p, q	1, 2, 3, 4	2
2	goal, p, q, r, s	1, 2, 3, 4, 5	3
3	goal, p, q, r, s, w	1, 2, 3, 4, 5	4
4	goal, p, q, r, s, w, t, u	1, 2, 3, 4, 5	5
5	goal, p, q, r, s, w, t, u, v	1, 2, 3, 4, 5, 6	6
6	goal, p, q, r, s, w, t, u, v, start	1, 2, 3, 4, 5, 6	halt

Working memory contains goal and sub-goal states waiting to be satisfied (shown true).

Space searched by execution:





start

goal

Backward chaining
Direction of search

Forward chaining vs. backward chaining

- Data-driven, forward chaining
  - Starts with the initial given data and search for the goal.
  - At each iteration, new conclusion (RHS) becomes the pattern to look for next
  - Working memory contains true sentences (RHS's).
  - Stop when the goal is reached.
- Goal-driven is the reverse.
  - Starts with the goal and try to search for the initial given data.
  - At each iteration, new premise (LHS) becomes the new subgoals, the pattern to look for next
  - working memory contains subgoals (LHS's) to be satisfied.
  - Stop when all the premises (subgoals) of fired productions are reached.
- Sense of the arrow is in reality reversed.
- Both repeatedly pick the next rule to fire.

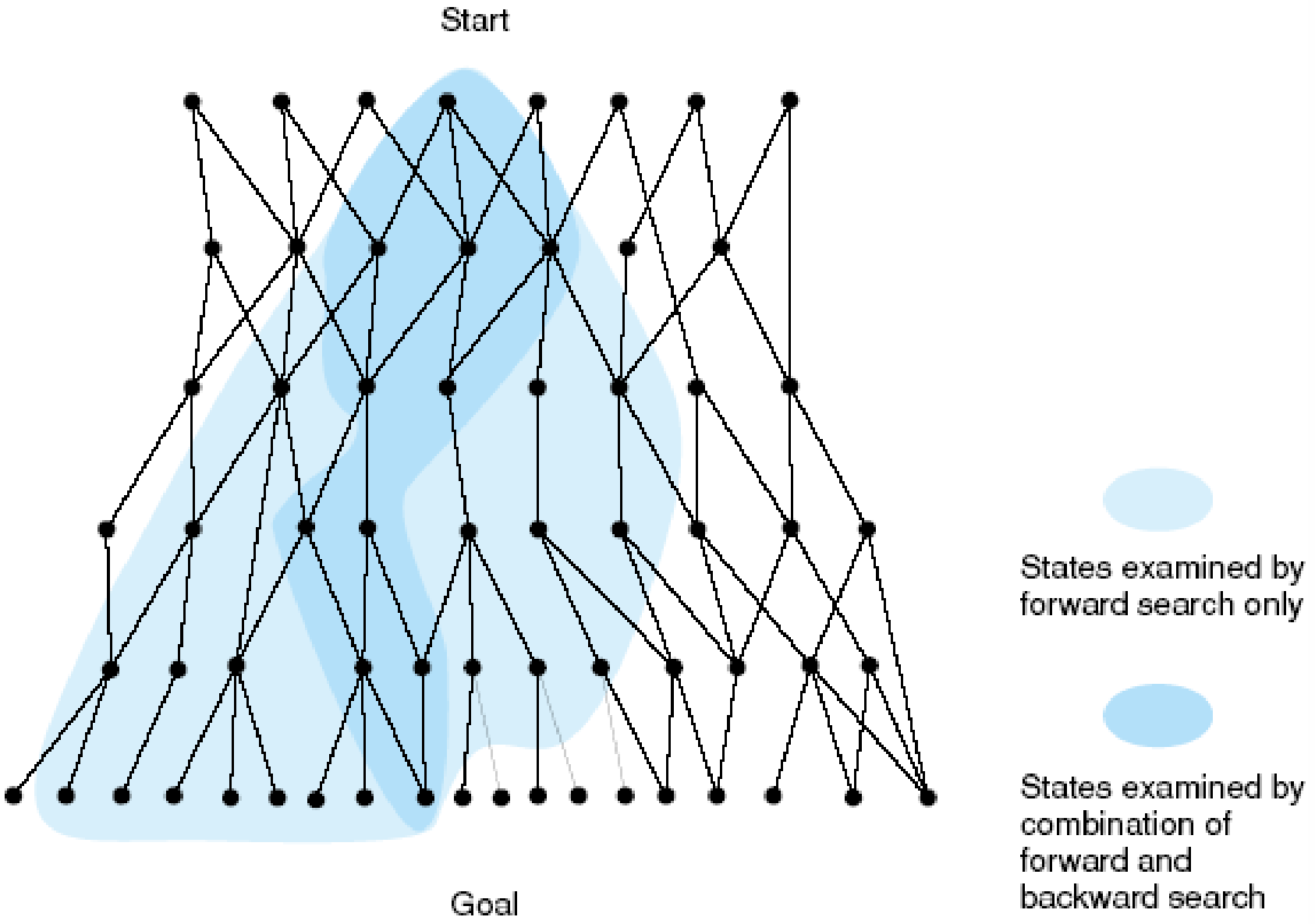
condition → action
premise → conclusion

	Forward chaining	**Backward chaining**
Starts with	premise	conclusion
Search for	conclusion	premise
Working memory	true statements	subgoals to be proved
Stopping criteria	goal is reached	Initial data are reached
	Data-driven	**Goal-driven**

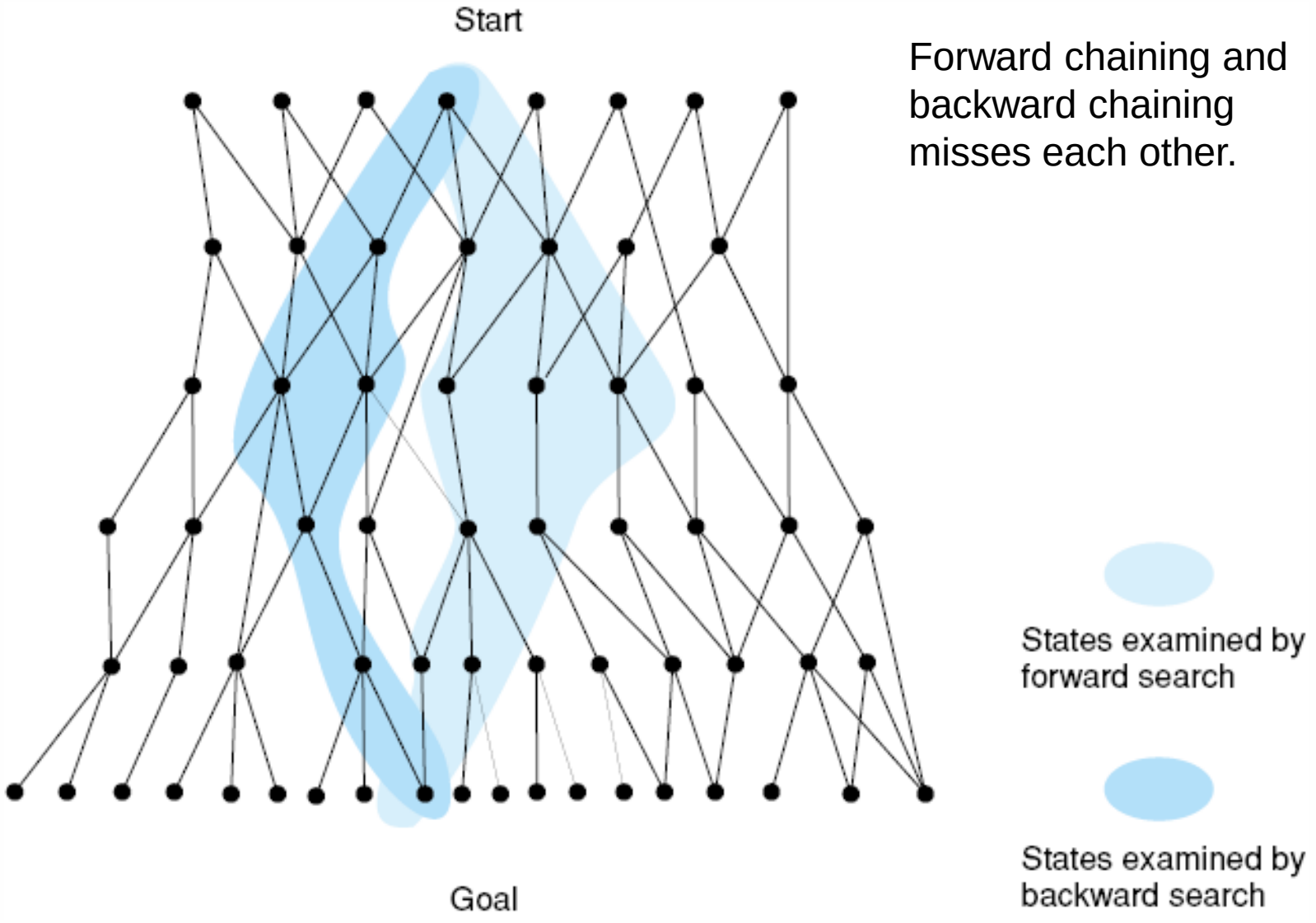
Combining forward- and backward-chaining

- Begin with data and search forward until the number of states becomes unmanageably large.
- Switch to goal-directed search to use subgoals to guide state selection.

Bidirectional search better than unidirectional search

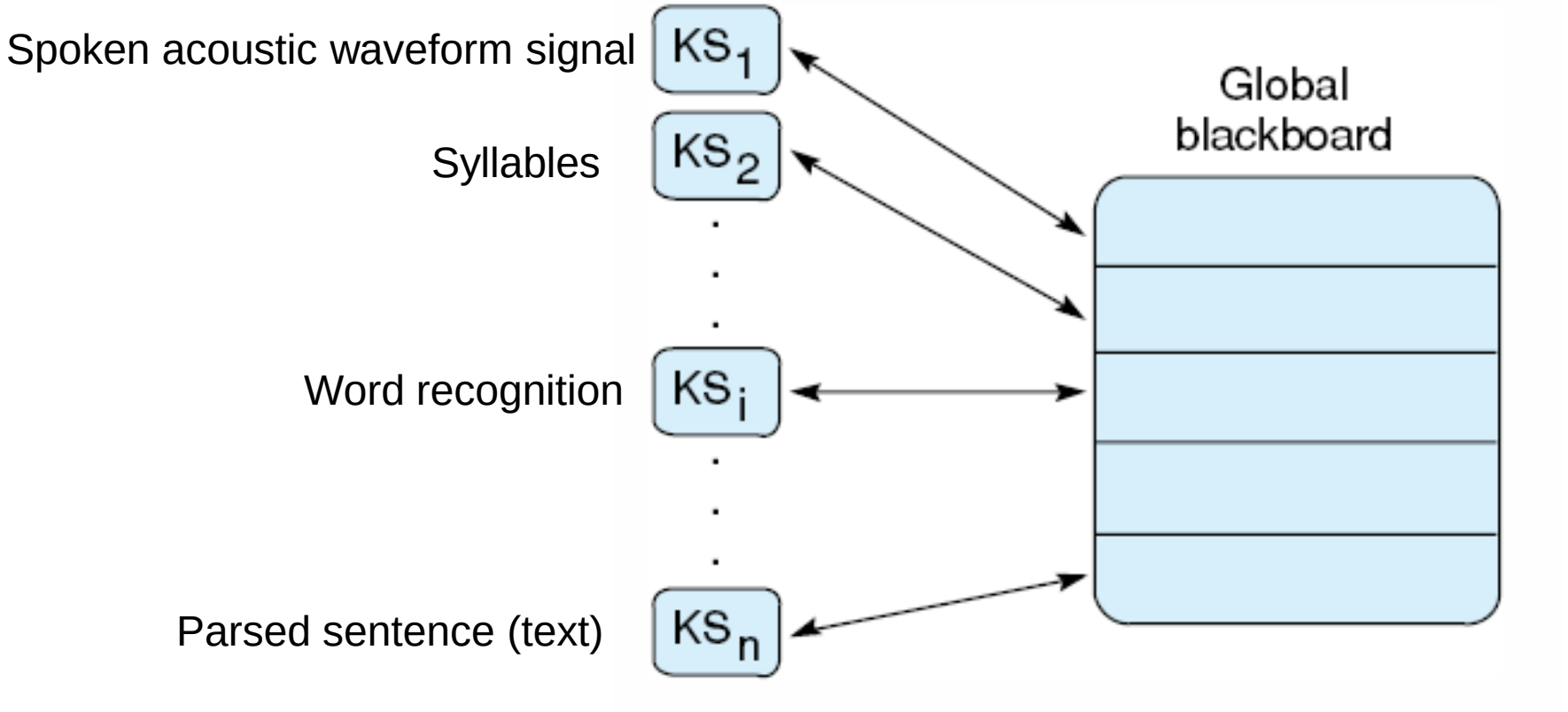


Bidirectional search worse than unidirectional search



Blackboard architecture for language understanding

- Layered production systems
- Separate productions into modules
- Each module (PS) is an agent -- knowledge source
- A single global structure -- blackboard



Conclusion

- Data-driven, forward chaining
  - conditions first, then actions
  - working memory contains true statements describing the current environment
- Goal-driven, backward chaining
  - actions first, then conditions (subgoals)
  - working memory contains subgoals to be shown as true
- Mixed approach
  - Start with data and go forward until frontier is too big
  - Then start with goal and go backward
  - Try to connect the two in the middle of the state space
- Prolog implementation of production systems requires infinite loop detection.

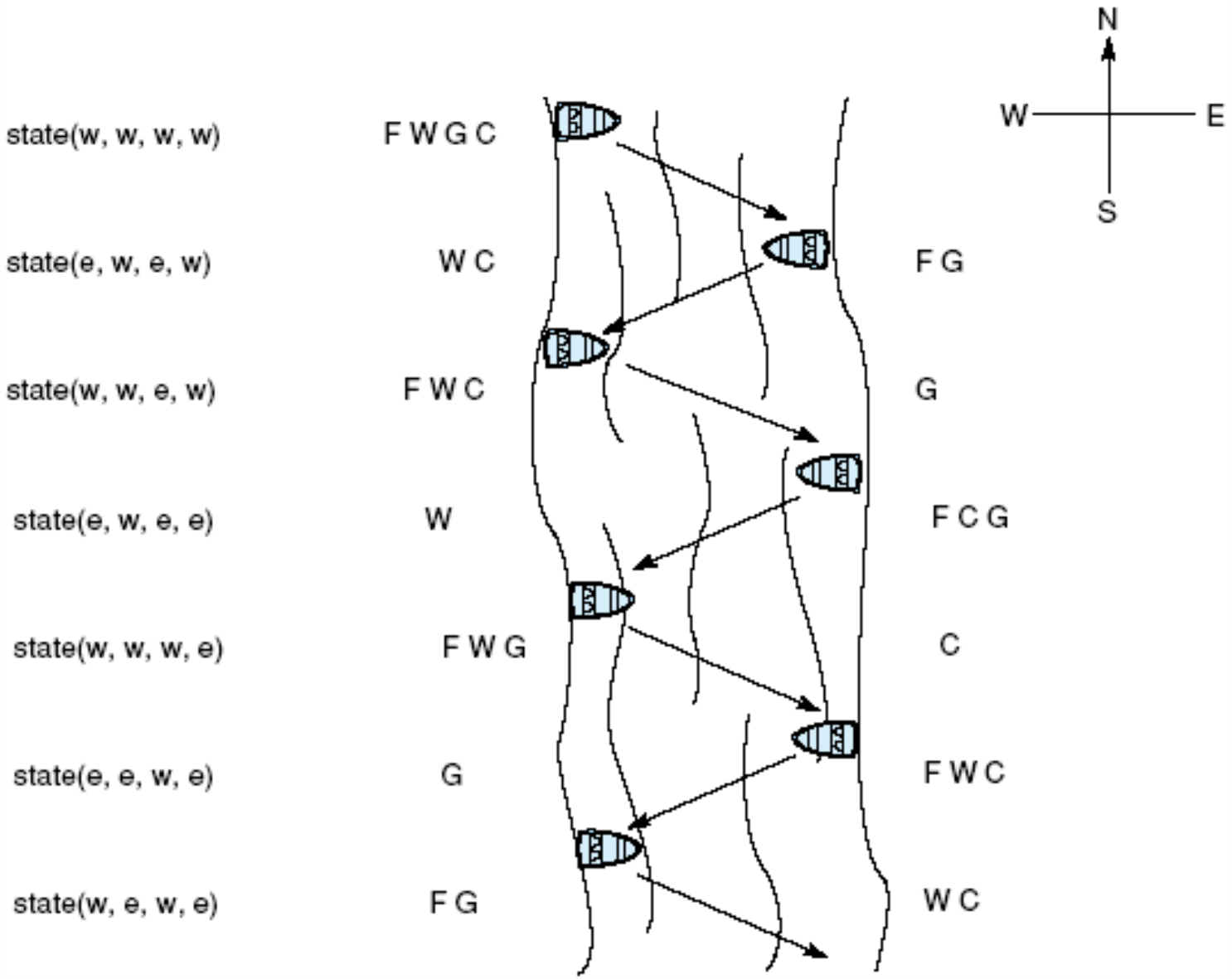
- Break for 10 minutes.
- But you can start lab if you like.

Farmer, wolf, goat, and cabbage problem as a production system

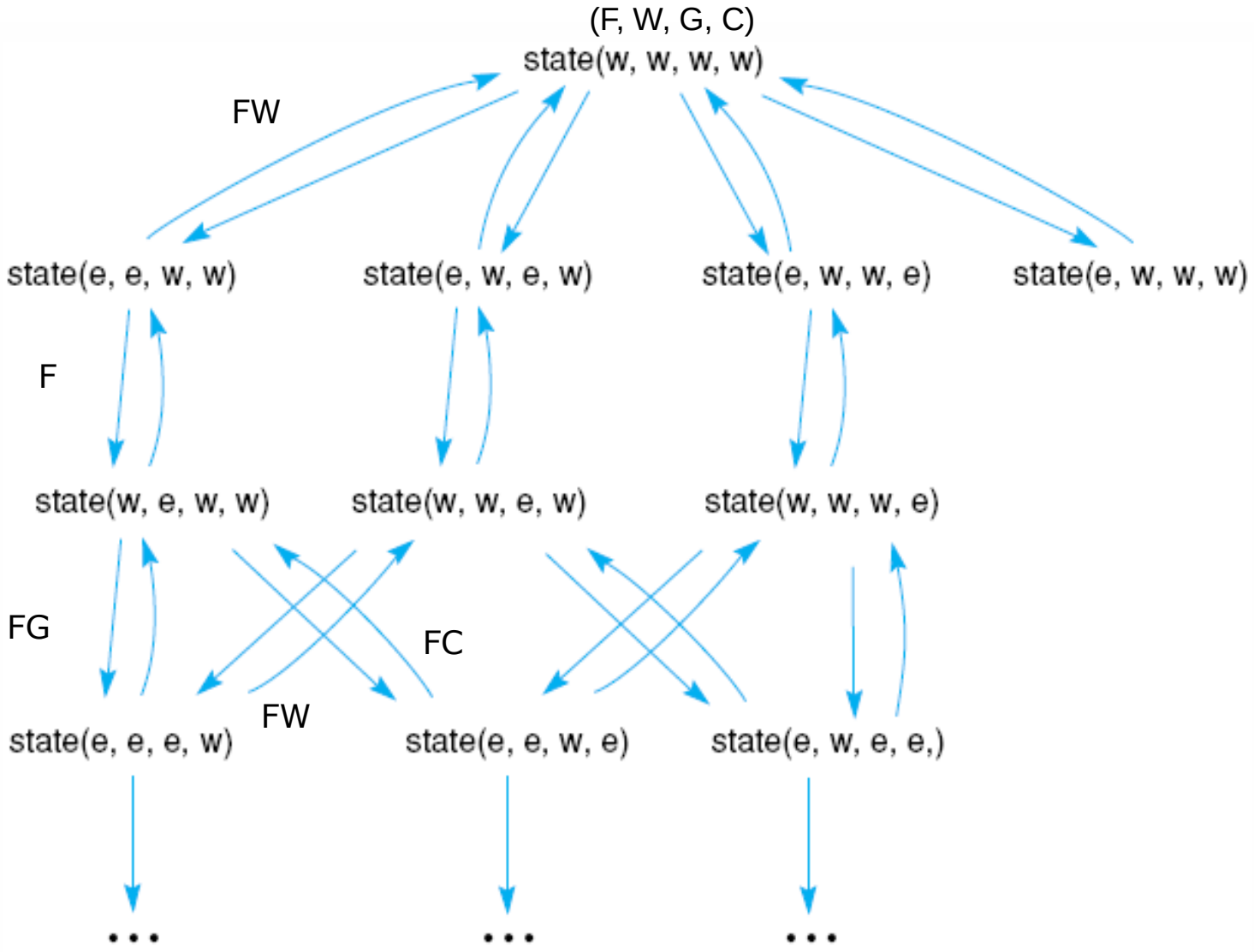
- Farmer, wolf, goat, and cabbage come to the edge of a river.
- A boat at the river's edge. Only the farmer can row. The boat can carry only two things, including the rower, at a time.
- If the wolf is ever left alone with the goat, the wolf will eat the goat.
- If the goat is left alone with the cabbage, the goat will eat the cabbage.
- Devise a sequence of crossings of the river so that all four characters arrives safely on the other side of the river.

- Representation
  - Predicate state(F, W, G, C) describes the location of Farmer, Wolf, Goat, and Cabbage.
  - Possible locations are e for east, w for west, bank for each of the 4 variables.
  - Initial state is state(w, w, w, w)
  - Goal state is state(e, e, e, e)

Sample solution



State space graph including unsafe states



Prolog implementation

- Use list to represent a state or configuration [Farmer, Wolf, Goat, Cabbage]
- Initially, the game begins
  - everyone is on the West bank
  - [w,w,w,w]
- Farmer takes the wolf across
  - [e,e,w,w]
  - Goat eats the cabbage
- Final configuration
  - everyone is on the East bank
  - [e,e,e,e]

Make a move

- In each move, the farmer crosses the river with either the wolf, the goat, the cabbage, or nothing.
- Each move can be represented with a corresponding atom
  1. wolf
  2. goat
  3. cabbage
  4. nothing
- Each move transforms one configuration into another.
- move([w,w,w,w], wolf, [e,e,w,w]).
- move(Config, Move, NextConfig).
  - Config and NextConfig are lists
  - Move is a variable whose value is one of the 4 possible atoms.

8 possible movements

- When the wolf and the farmer move, the goat and the cabbage do not change position.

1. move([w,w,Goat,Cabbage],wolf,[e,e,Goat,Cabbage]).

- Farmer and the wolf can go from East to West also.

2. move([e,e,Goat,Cabbage],wolf,[w,w,Goat,Cabbage]).

- Farmer takes goat

3. move([w,Wolf,w,Cabbage],goat,[e,Wolf,e,Cabbage]).
4. move([e,Wolf,e,Cabbage],goat,[w,Wolf,w,Cabbage]).

- Farmer takes cabbage

5. move([w,Wolf,Goat,w],cabbage,[e,Wolf,Goat,e]).
6. move([e,Wolf,Goat,e],cabbage,[w,Wolf,Goat,w]).

- Farmer takes nothing

7. move([w,Wolf,Goat,Cabbage],nothing,[e,Wolf,Goat,Cabbage]).
8. move([e,Wolf,Goat,Cabbage],nothing,[w,Wolf,Goat,Cabbage]).

Safety

- We need a predicate to decide whether a configuration is safe or not.
- safe([Man,Wolf,Goat,Cabbage])
- Today's lab

Implementation so far

- Use list to represent a configuration (state)
- Make a move
  - source configuration
  - farmer takes a belonging or nothing across the river from east to west or west to east
  - destination configuration
- Predicate safe returns safety status of a configuration

- Still, how to search for the sequence of moves for the solution?

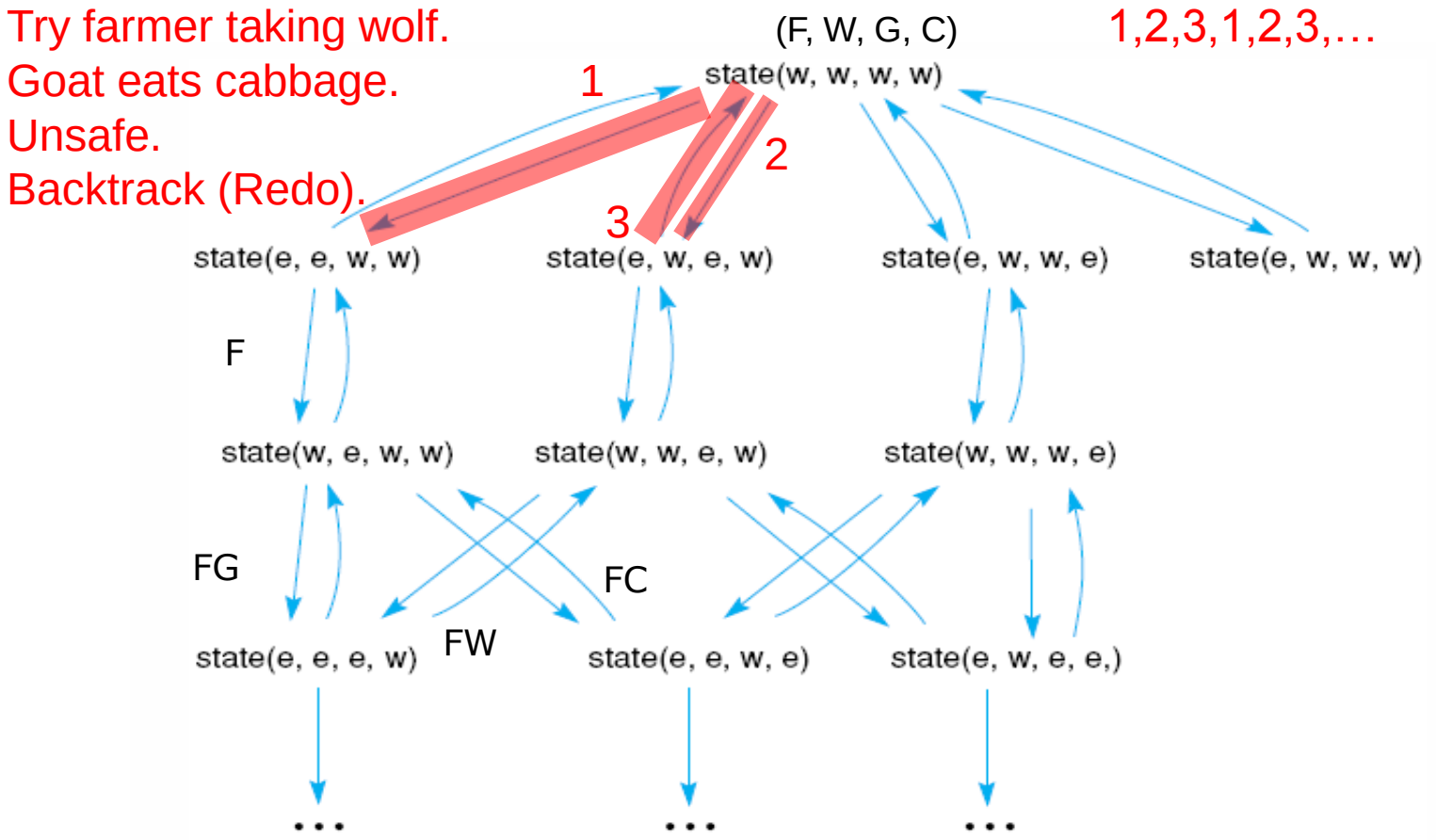
Search for the solution recursively

- solve([w,w,w,w], S)
- From initial configuration, search for a solution S.
- S is a list containing the sequence of belongings taken by the farmer.
- Trivial case
  - solve([e,e,e,e], []).
  - If everything is on the east bank, stop recursion.
- Recursive case
  - solve(Config, [NextMove | OtherMoves]) :-

```
move(Config,NextMove,NextConfig),
                safe(NextConfig),
                solve( NextConfig, OtherMoves ).
```

Infinite loop

- solve([w,w,w,w], S) would cause an infinite loop
  - ERROR: Out of local stack
  - Never reached the trivial case in the recursion



Main program

- farmer(S) :- length(S,7), solve([w,w,w,w], S), !.
- Search for a solution recursively.
- Limit the length of the solution to 7 to avoid infinite loop.
- Stop as soon as the first solution is found.
- ?- farmer(X).

- Be careful with the order of the predicates
- farmer(S) :- solve([w,w,w,w], S), length(S,7), !.
- ERROR: Out of local stack