Course 7

Simbolic (classical) AI problems.

How to solve them?

5 steps in modeling an AI problem

Step 1	Understand the difference between the general problem and an instance of it
Step 2	Decide what a state is and appreciate the dimension of the search space
Step 3	Find a proper representation of a state
Step 4	Represent the transitions
Step 5	Choose a control strategy and apply it

Toy problems – examples

- **The 8-puzzle problem**

  On a 3x3 board there are 8 square pieces that can move upwards, downwards, to the right and to the left. At each move, just one piece is moved in a neighbouring position and in the limits of the table. There is an initial configuration of the pieces on the table and a final one that has to be reached. What is the sequence of moves that brings the table from the initial configuration into the final one?

Toy problems

- **Missionaries and cannibals**

  3 missionaries and 3 cannibals have to cross a river in a boat having 2 places. If at any moment during crossing, on one boarder or the other, the cannibals exceeds in number the missionaries, then missionaries are in darger of being eaten. How can all 6 men cross the river safely?

Toy problems

- **Sentence generation problem**

  Suppose we have a grammar (a set symbols called terminals, a set of symbols called non-terminals, a collection of production rules, and a start symbol). How could one produce a sentence belonging to the language generated by the grammar?

Toy problems

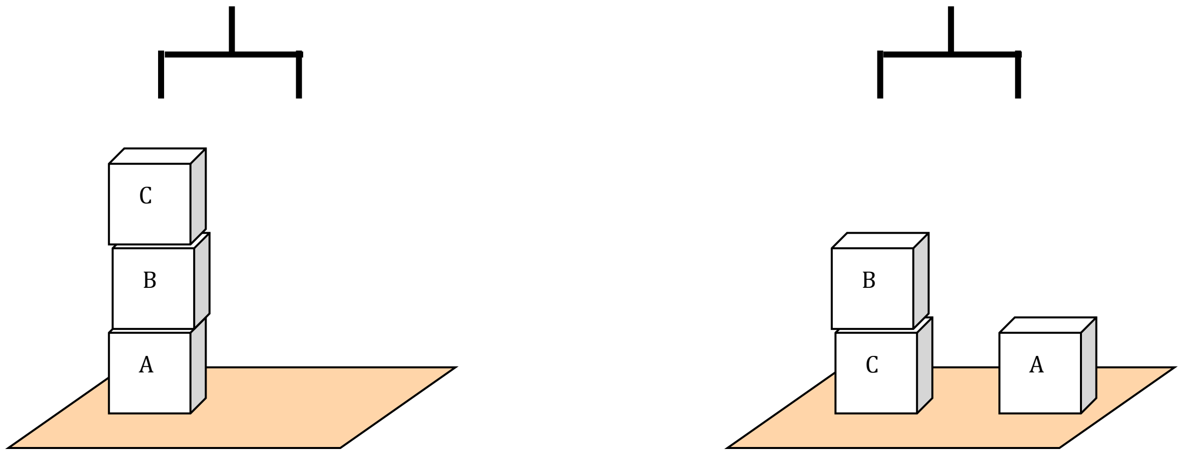
- **The monkey and the banana**

  In a cage there are: a monkey, a banana hanged by the ceiling at a height the monkey cannot reach, and, in a corner, a box. After a number of unseccessful tries, the monkey goes to the box, puts it under the banana, climbs on the box and catches the banana. How did the monkey find the solution?

Toy problems

- **The cubs world**

A robot hand has to move a stack of cubs from an initial arrangement into a given final arrangement.



Initial state

Final state

Step 1

Problem or instance of a problem?

- 8-puzzle
  - formulated as an instance
- Missionaries and cannibals
  - formulated as an instance
- Sentence generation
  - formulated as a problem
- Monkey and banana
  - formulated as an instance
- Cubs world
  - formulated as an instance
- Other examples:
  - The chess game
  - Driving a car...

Step 1

An instance of the problem of sentence generation

- Let G1 = {N1, T1, S, P1} be a grammar, such that:

N1 = {S, NP, VP, N, V} – a set of non-terminals, meaning: sentence, nominal group, verbal group, noun, verb;

T1 = {*cat*, *mouse*, *catches*} – a set of terminals (words);

S = the start symbol of the grammar; choosing it has the significance that a sequence of terminals covered by this category is wanted;

P1 = {S:= NP VP,

NP:= DET N,

VP:= V NP,

N := *cat*,

N := *mouse*,

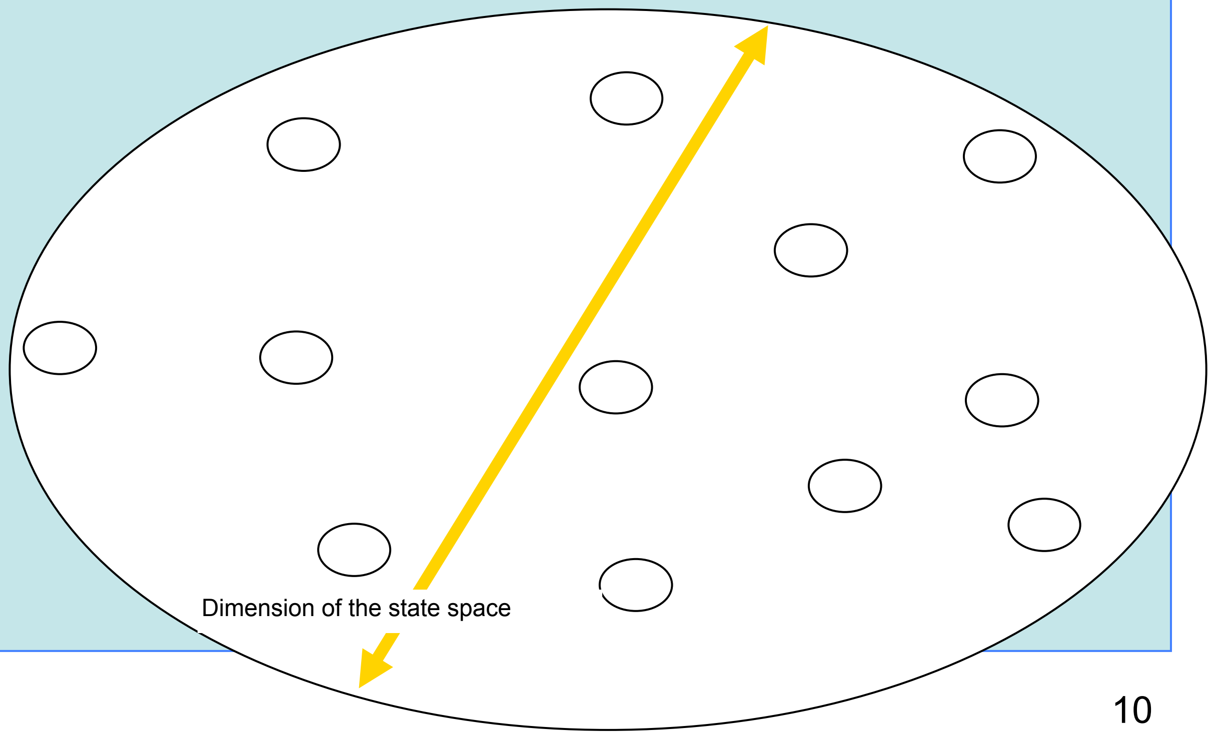
V := *catches,*

DET := the} – a list of production rules.

Step 2

The problem's space

- States, dimension of the state space



Step 2

Dimension of the state space

- Chess game: *10^{120}*

Step 2

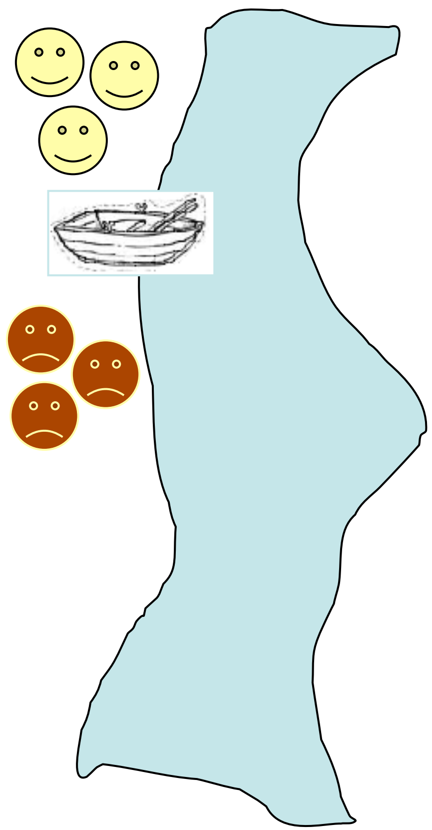
Dimension of the state space

- Chess game: 10^{120}
- 8-puzzle: 9!

Step 2

Dimension of the state space

- Chess game: 10^{120}
- 8-puzzle: 9!
- Missionaries and cannibals:

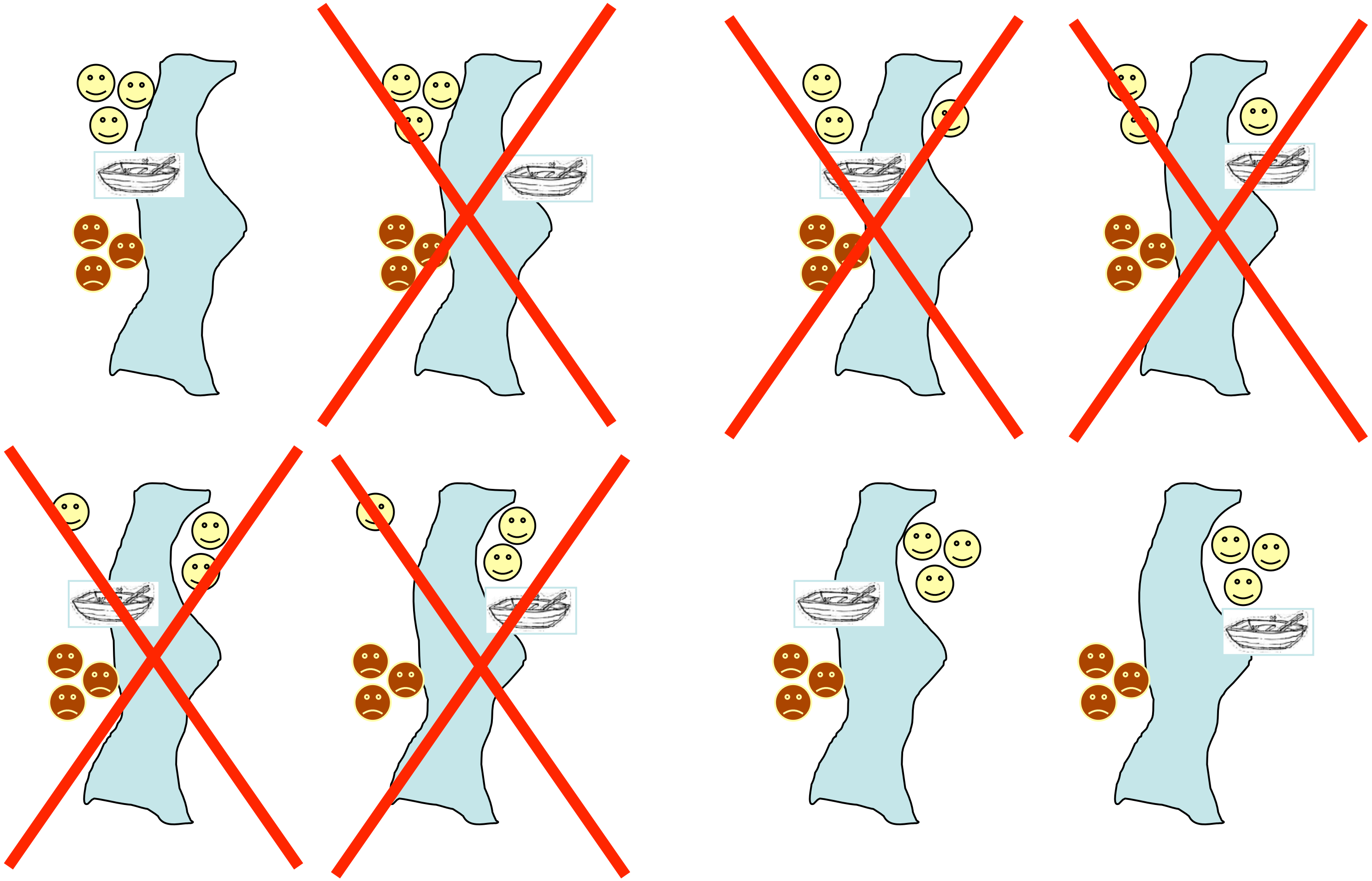


Step 2

States: missionaries and cannibals

3 cannibals on the left side

crossed states: never reached

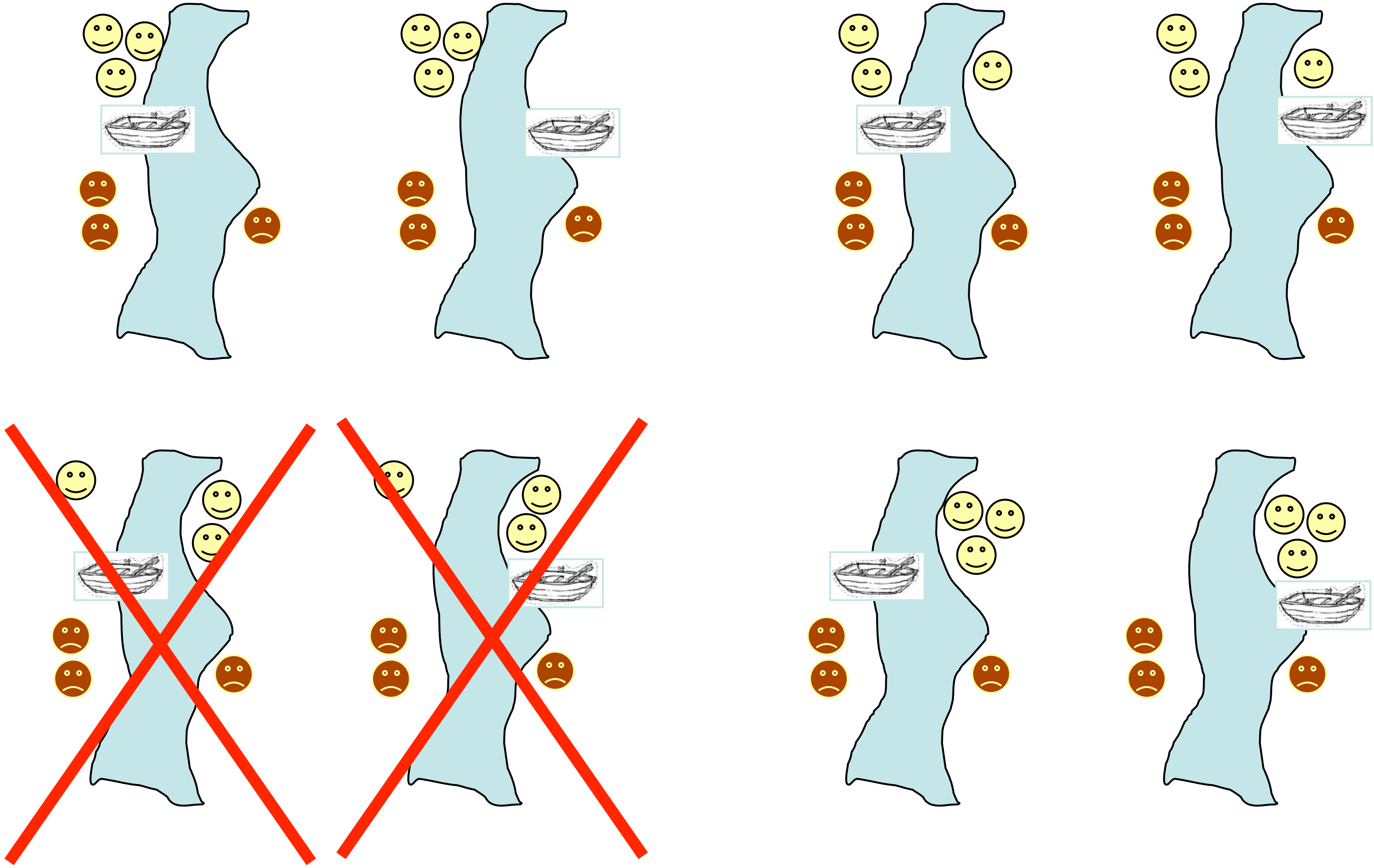


Step 2

States: missionaries and cannibals

2 cannibals on the left side

crossed states: never reached

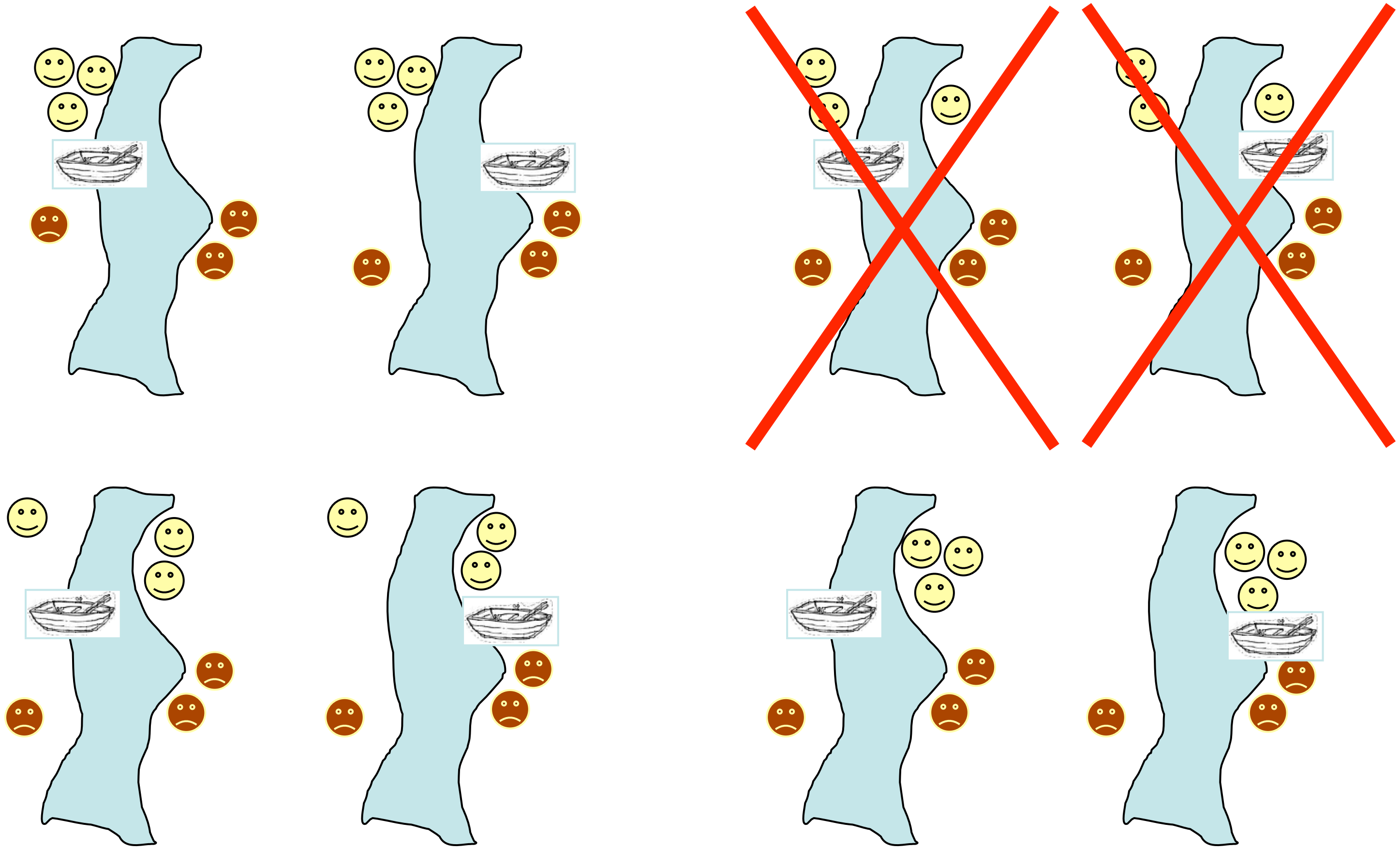


Step 2

States: missionaries and cannibals

1 cannibal on the left side

crossed states: never reached

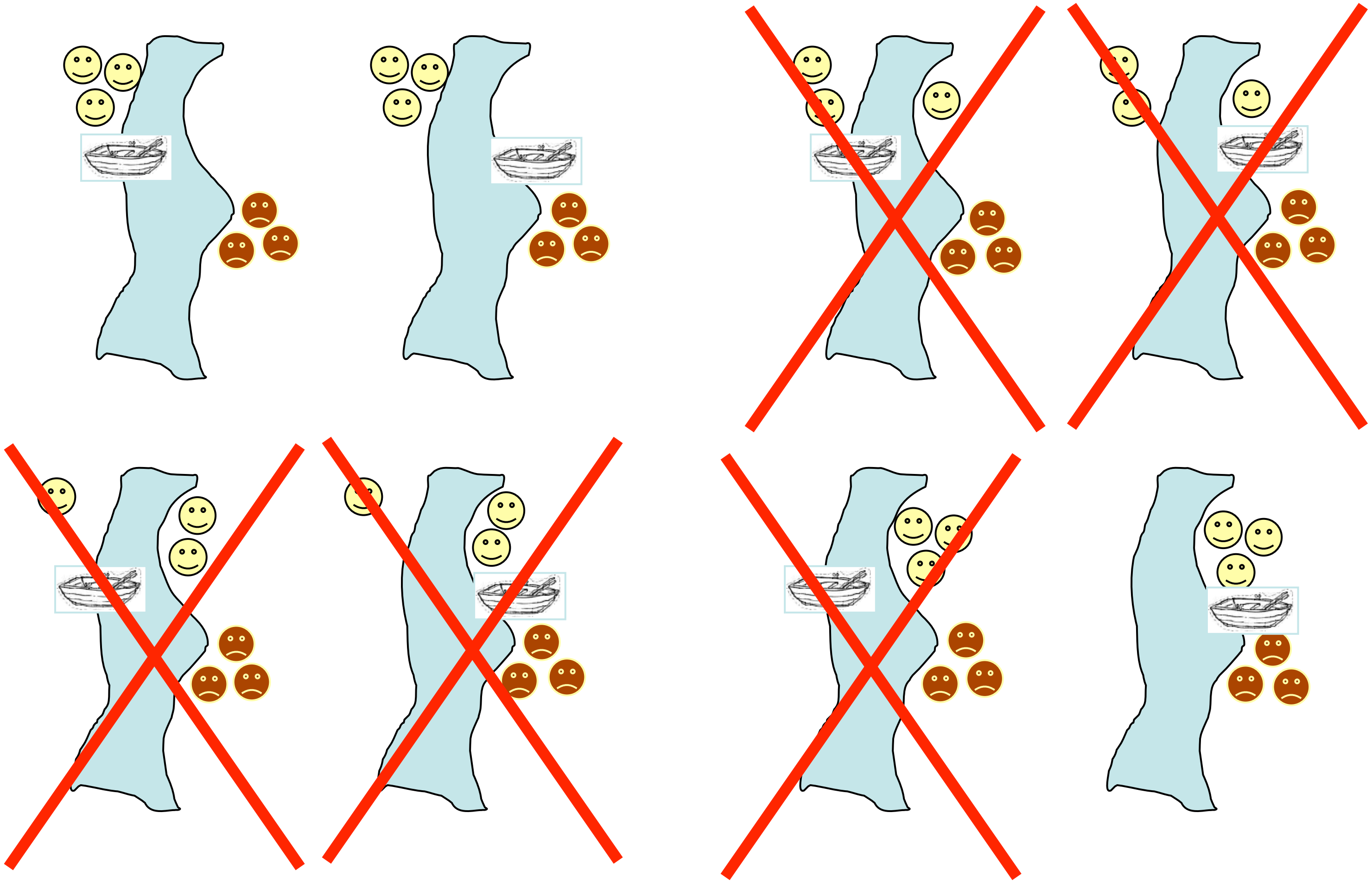


Step 2

States: missionaries and cannibals

no cannibals on the left side

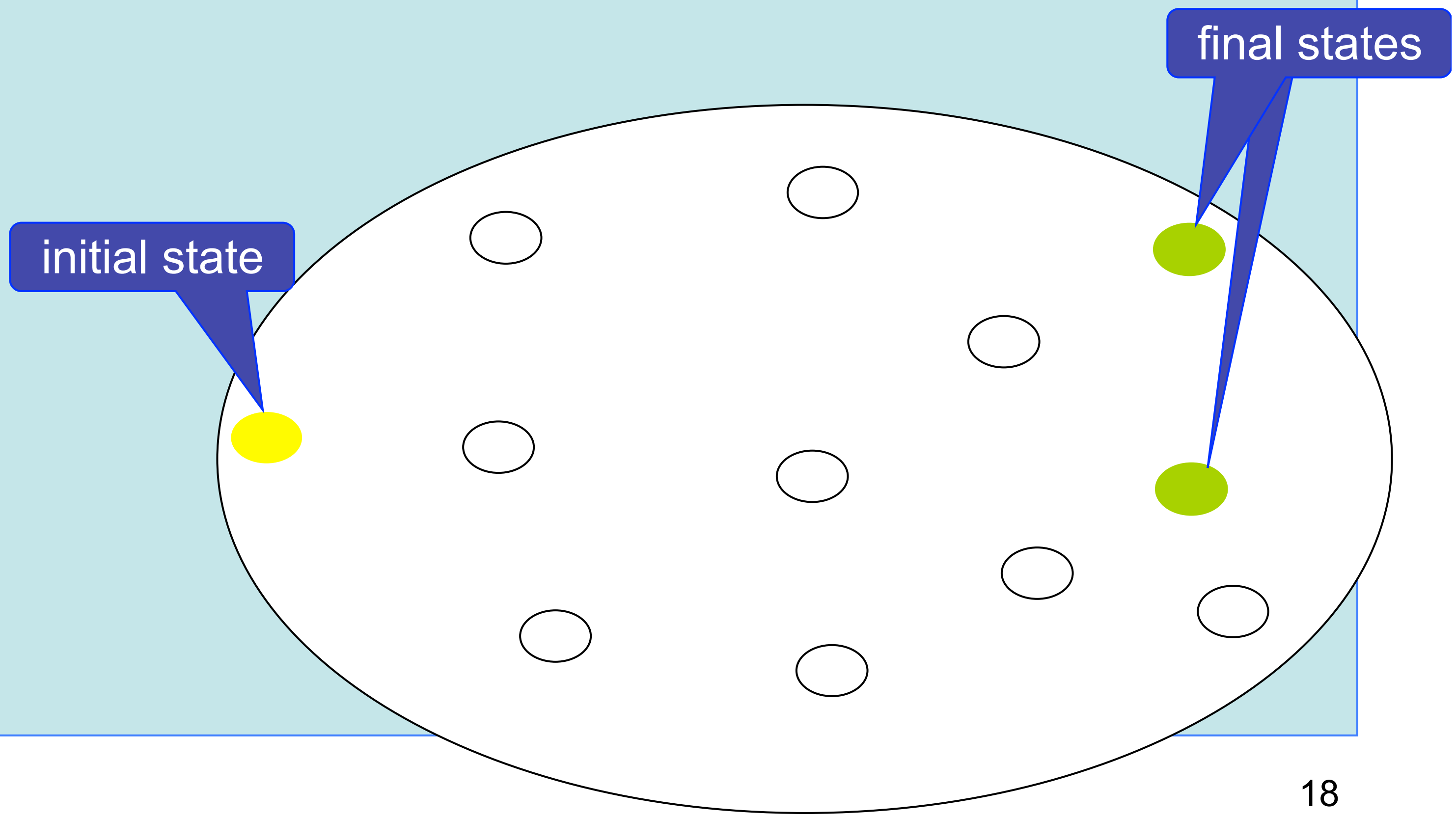
crossed states: never reached



Step 2

Seeing a problem in the state space

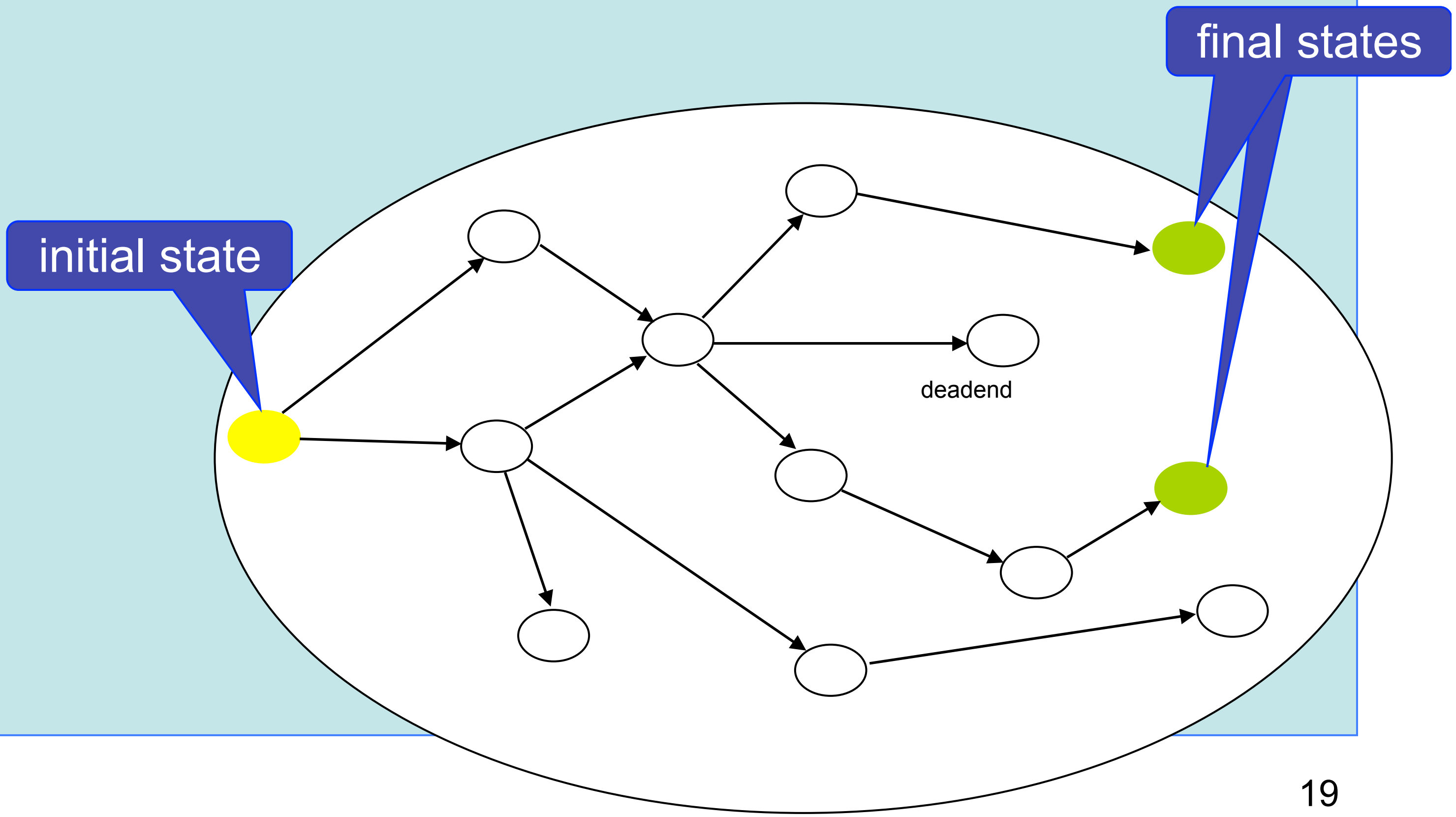
Initial and final states



Step 2

Seeing a problem in the state space

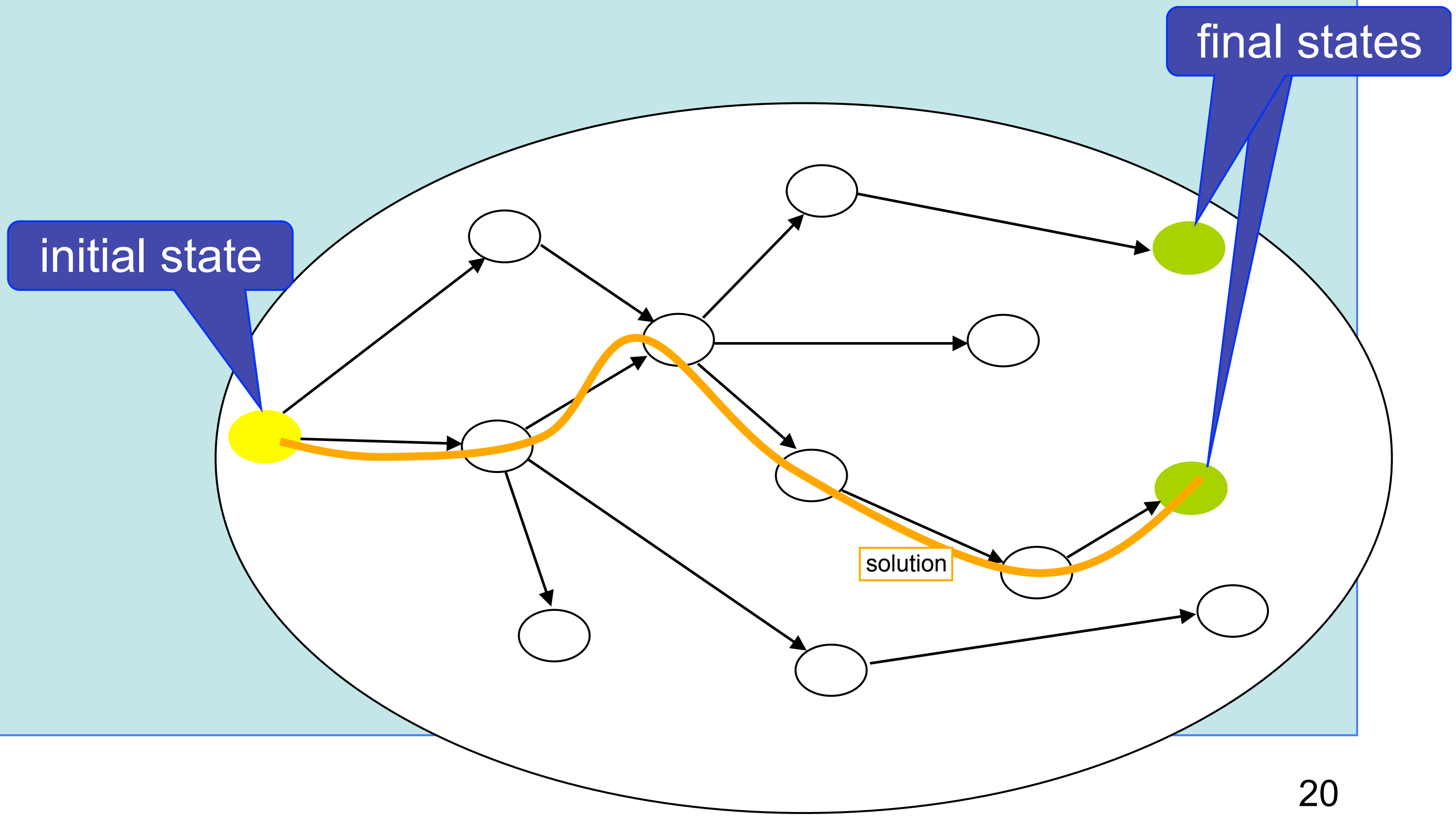
Transitions



Step 2

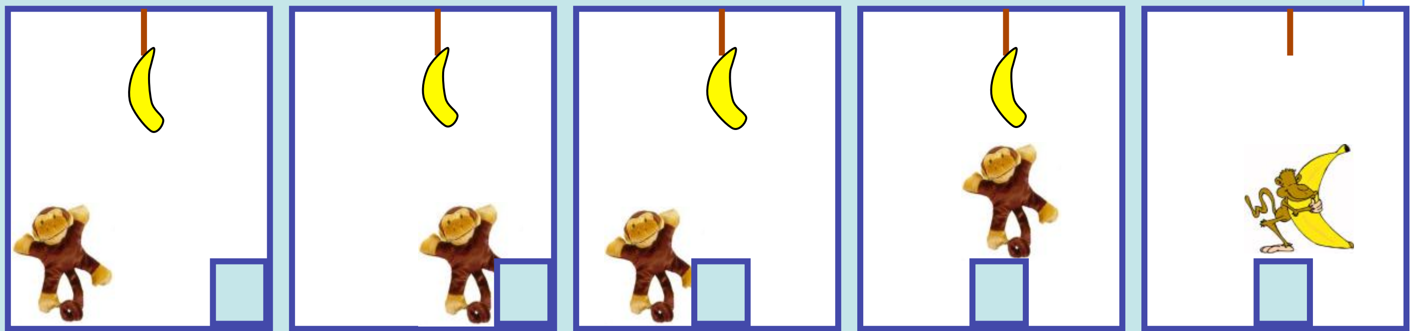
Seeing a problem in the state space

Solution = a sequence of transitions from the initial state to one final state



Monkey and banana

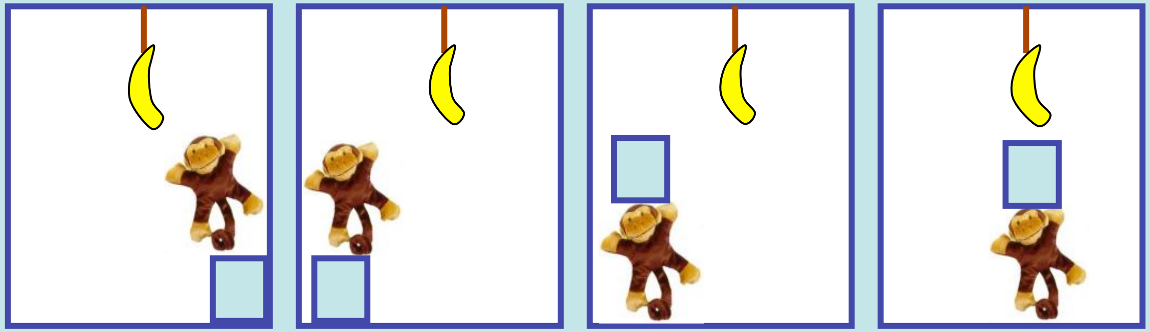
Solution = a sequence of transitions (states)



Step 2

Monkey and banana

Other possible states



Step 2

Step 3

How to represent a state?

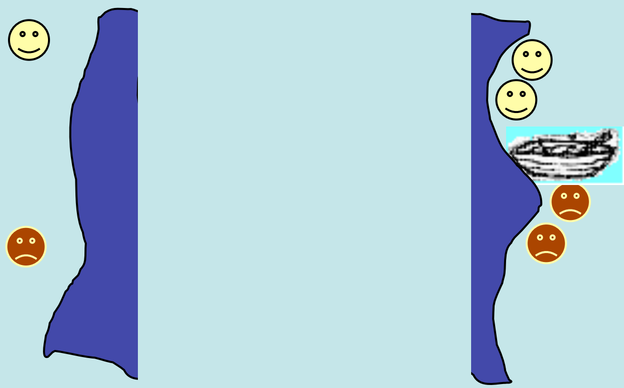
8-puzzle

a 3x3 matrix

Step 3

How to represent a state?

Missionaries and cannibals



a 3 positions vector: (c, m, b)

Step 3

How to represent a state?

Sentence generation

Pentru instanţa de problemă G1 = {N1, T1, S1, P1}

N1 = {S, NP, VP, N, V}

T1 = {*cat*, *mouse*, *catches, the*}

S1 = S => we want to generate sentences

P1 = {

S:= NP VP,

NP:= DET N,

VP:= V NP,

N := *cat*,

N := *mouse*,

V := *catches,*

DET := *the*

}

a sequence of symbols

States while production: S | NP VP | DET N VP | the N VP | the cat VP | the cat V NP | the cat catches NP | the cat catches DET N | the cat catches the N | the cat catches the cat

Step 3

How to represent a state?

Monkey and banana

Relation Monkey-Box:

- **MoBo-far**= Monkey is far from Box
- **MoBo-near** = Monkey is near Box
- **MoBo-on** = Monkey is on Box
- **MoBo-under** = Monkey is under Box

Relation Box-Banana:

- **BoBa-lateral** = Box is lateral from Banana
- **BoBa-under** = Box is under Banana

Relation Monkey-Banana:

- **MoBa-far** = Monkey is far from Banana
- **MoBa-near** = Monkey is near Banana
- **MoBa-holds** = Monkey holds Banana

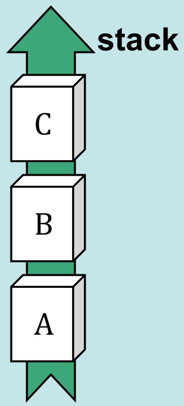
Initial state: **MoBo-far**, **BoBa-lateral**, **MoBa-far**.
Final state: **MoBa-holds**

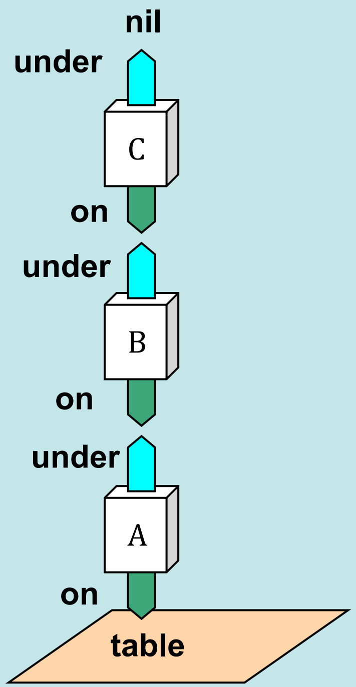
a collection of predicates

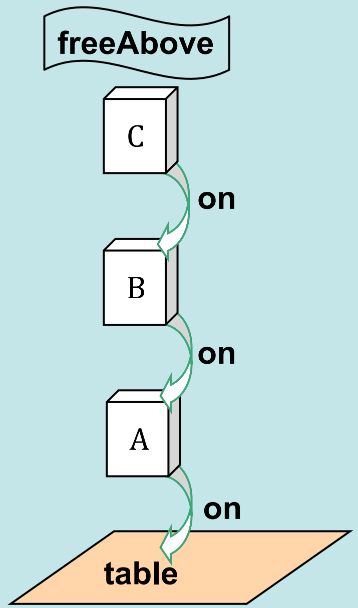
Step 3

How to represent a state?

Cubs world







globaly defined ordering

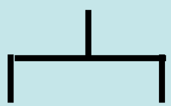
ordering defined through local vicinities

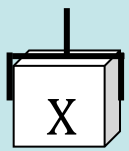
ordering defined through relations between objects

Step 3

How to represent a state?

Cubs world – representing the configuration of the hand





handEmpty

handHolds(X)

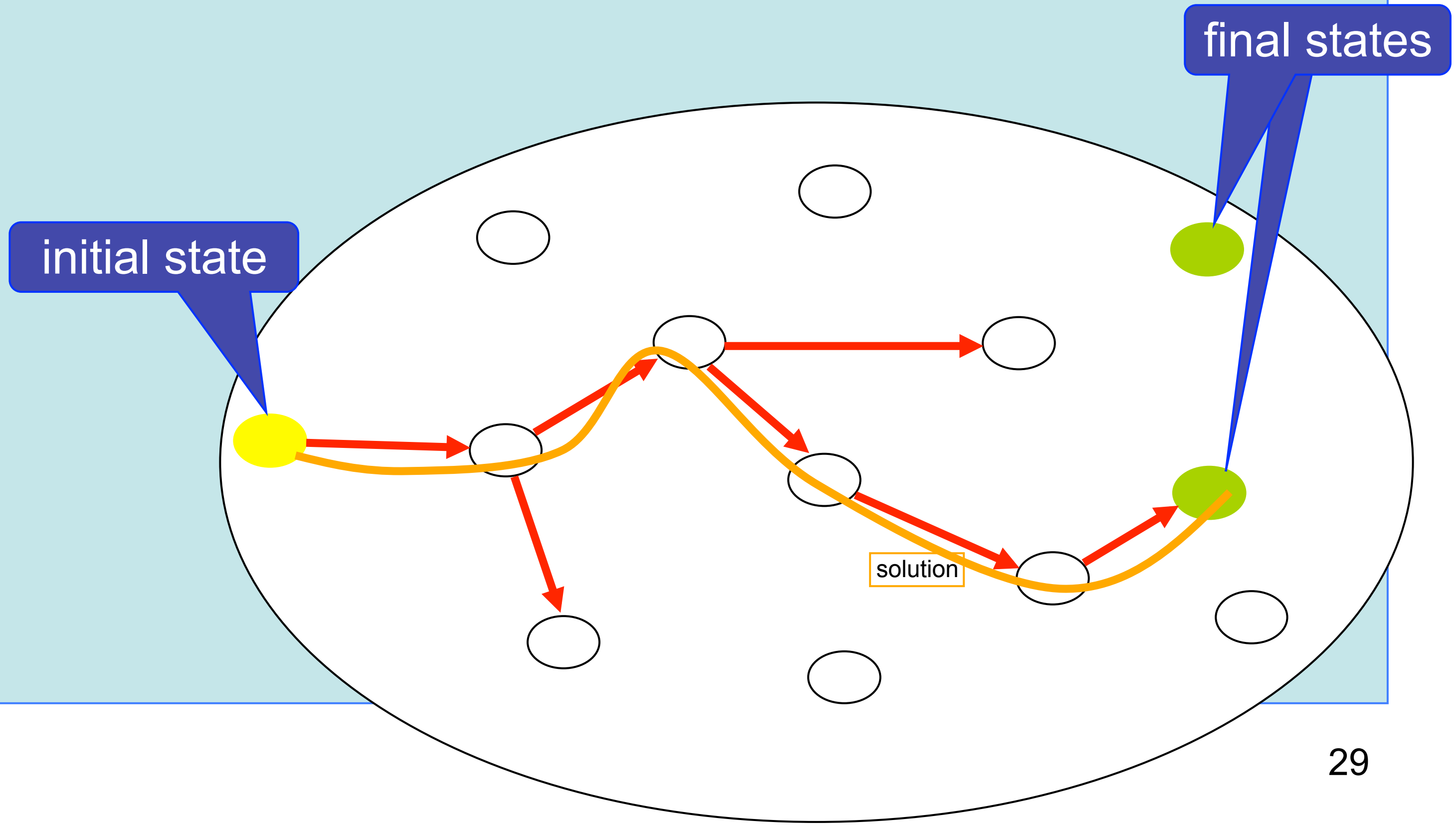
a collection of predicates

How to represent transitions?

Step 4

Two ways to navigate in the state space:

– states exist and have only to be visited

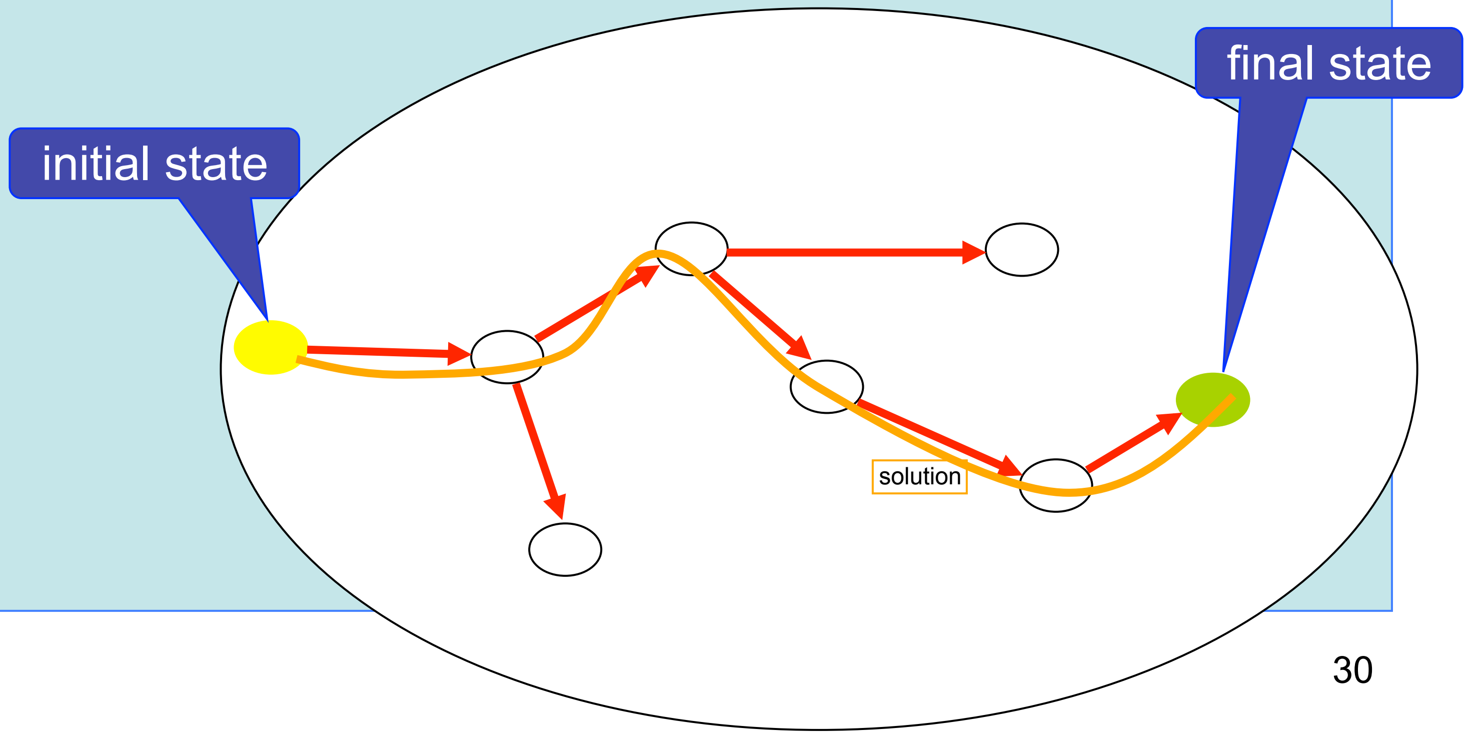


How to represent transitions?

Step 4

Two ways to navigate in the state space:

- states are generated only the moment they are visited

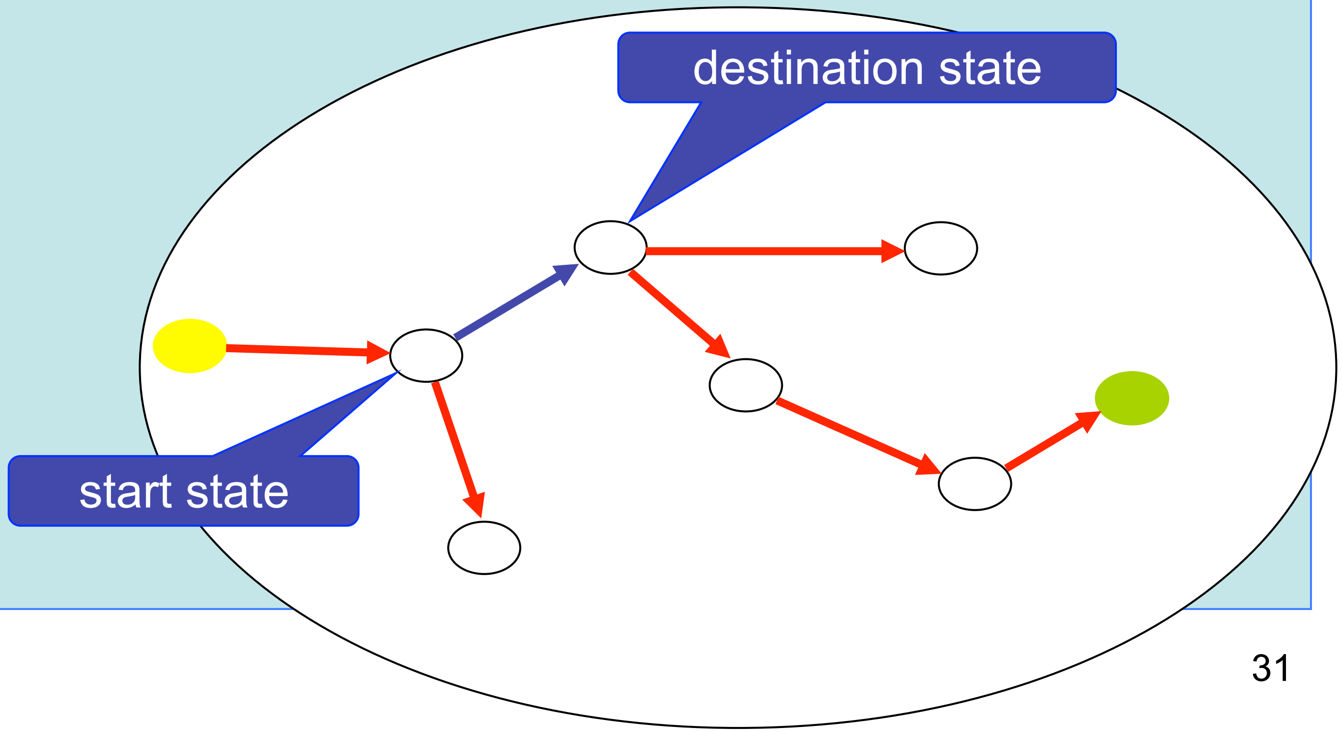


How to represent transitions?

Step 4

An operator verifies conditions and initiates actions

```
if <conditions> then <actions>
```



How to represent transitions?

Step 4

Chess:

Rule pawn-on-A-double-step
IF
    pawn in position (A,2) AND
    position (A,3) is free AND
    position (A,4) is free
THEN
    move pawn from position (A,2) onto position (A,4)

8 rules of this kind...

How to represent transitions?

Step 4

Chess:

Rule pawn-on-(X)-double-step

IF

pawn in position (X,2) AND

position (X,3) is free AND

position (X,4) is free

THEN

move pawn from position (X,2) onto position (X,4)

Only one rule of this kind!

... or two, if corresponding rule for the other player is also considered.

How to represent transitions?

Step 4

8-puzzle:

Regula move-piece-1-upwards

IF

piece 1 is not tight to the upper side of the border AND

the position above is free

THEN

change the piece with the position above it

8 rules of this kind!

x 4 directions ➔ 32 rules in all

How to represent transitions?

Step 4

8-puzzle:

Rule move-blac-upwards
IF
    blanc is not tight to the upper side of the border
THEN
    change the blanc with the piece above it

Only one rule of this kind!

x 4 directions ➔ 4 rules in all

How to represent transitions?

Step 4

Monkey and banana:

Being far from the box, the monkey gets closer to the box:
getCloser-MoBo:
IF {MoBo-far} THEN DELETE{MoBo-far}, ADD{MoBo-near}

Being by the box, the monkey draws away from it:
drawAway-MoBo:
IF {MoBo-near} THEN DELETE{MoBo-near}, ADD{MoBo-far}

Being by the box and far from banana, the monkey pulls the box under banana:
pullUnder-MoBoBa:
IF {MoBo-near, BoBa-lateral} THEN DELETE{BoBa-lateral}, ADD{BoBa-under}

Being by the box and under banana, monkey pushes the box away from banana: **pushLateral-MoBoBa**;
Being by the box, monkey climbs on it: **climbOn-MoBo**;
Being on the box, monkey gets down from it: **getDown-MoBo**;
Being by the box, monkey puts the box on its head: **putOnHead-MoBo**;
With the box on its head, monkey gets down the box from the head: **getDownFromHead-MoBo**;
Being on the box and under banana, monkey grasps banana: **grasp-MoBa**.

How to represent transitions?

Step 4

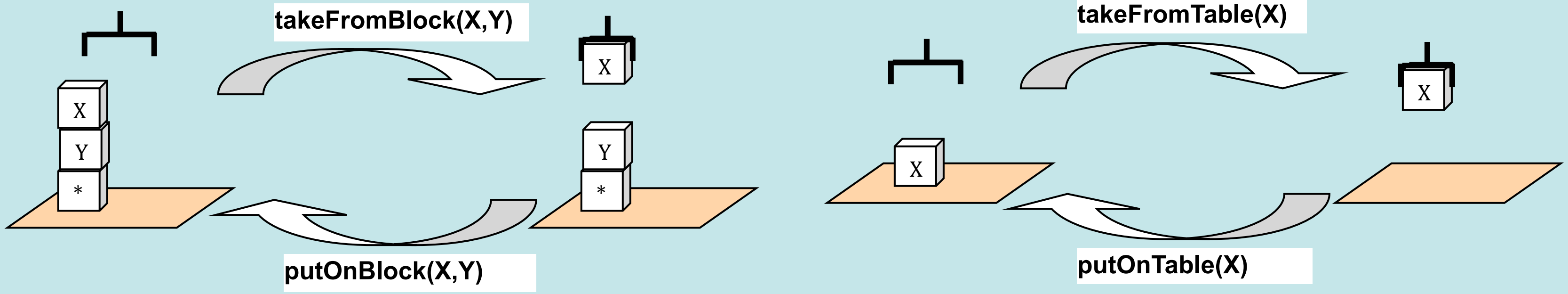
STRIPS rules systems

- States represented as sets of predicates (features)
- Rules:
  - if <preconditions-list>
  - then <delete-list> <add-list>

Step 4

How to represent transitions?

STRIPS rules systems in the cubs world



How to represent transitions?

Step 4

STRIPS rules systems in the cubs world

takeFromBlock(X,Y):

IF {on(X, Y), freeAbove(X), handEmpty} THEN DELETE{on(X, Y), freeAbove(X), handEmpty} ADD{freeAbove(Y), handHolds(X)}

takeFromTable(X):

IF {on(X, table), freeAbove(X), handEmpty} THEN DELETE{on(X, table), freeAbove(X), handEmpty} ADD{handHolds(X)}

putOnBlock(X,Y):

IF {handHolds(X), freeAbove(Y)} THEN DELETE{handHolds(X), freeAbove(Y)} ADD{on(X, Y), freeAbove(X), handEmpty}

putOnTable(X):

IF {handHolds(X)} THEN DELETE{handHolds(X)} ADD{on(X, table), freeAbove(X), handEmpty}

Step 5

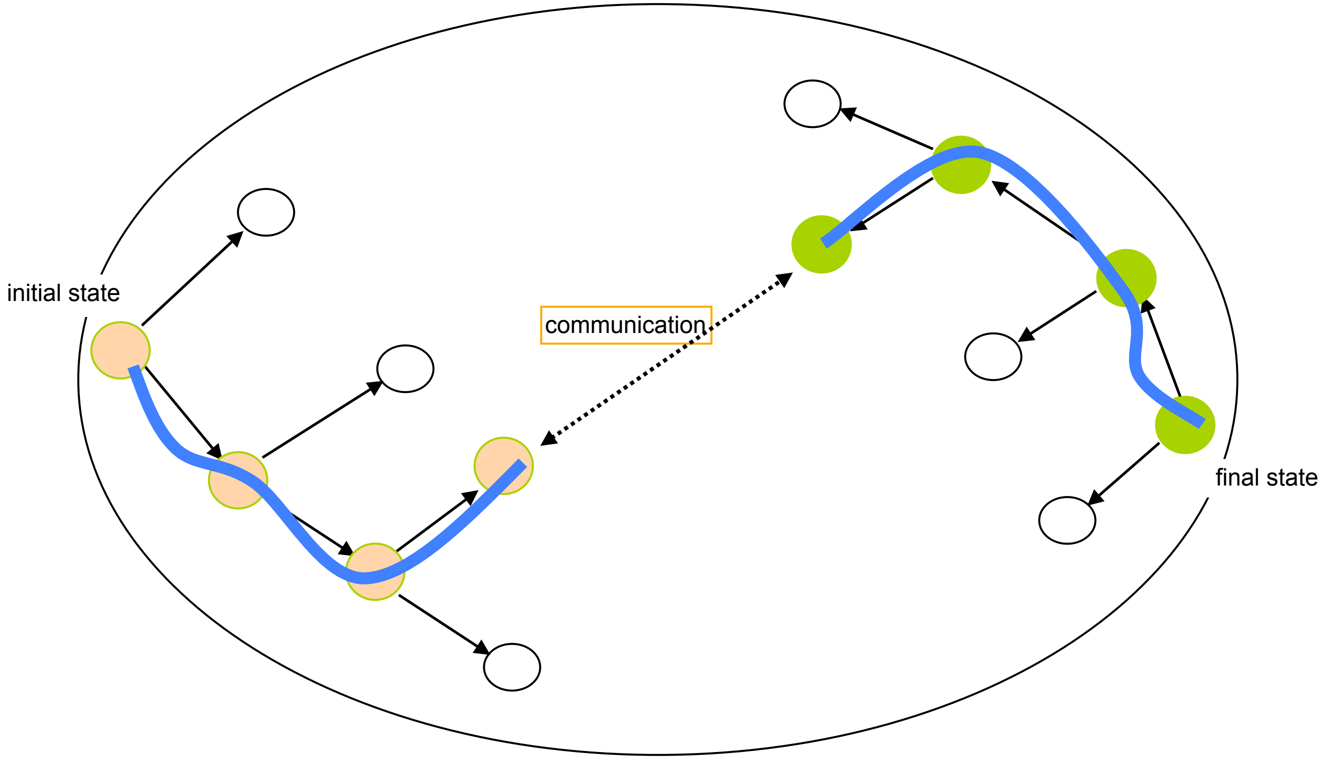
In search for the solution

- Algorithms and search heuristics in the state space
  - irrevocable strategies
    - *hill-climbing* (ascensional)
  - tentative strategies
    - *backtracking* (annealing ascensional)
  - exhaustive strategies (*brute-force*)
    - *generate-and-test*
    - *depth-first*
    - *breadth-first*
    - *best-first*

Step 5

In search for the solution

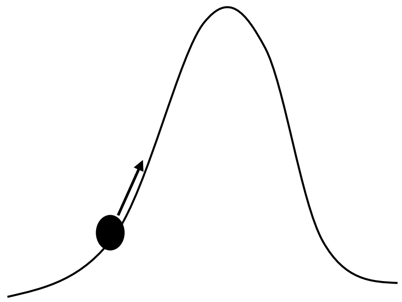
- Synchronous bidirectional search



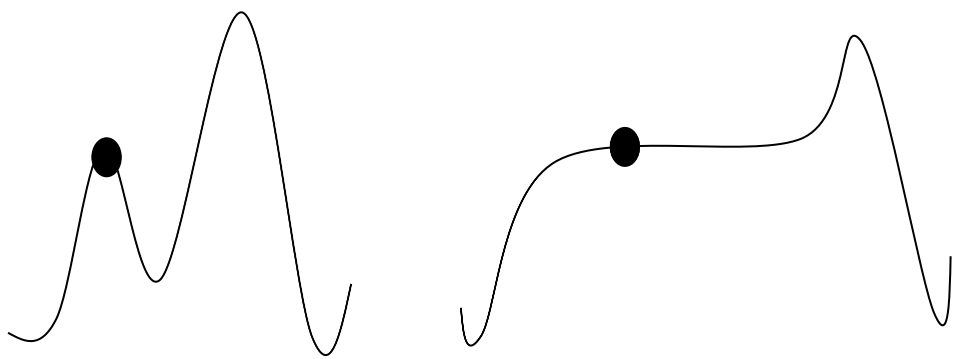
Step 5

Irrevocable strategies: *hill-climbing*

- There is no way back
  - a function approximates the closeness to the solution at any step



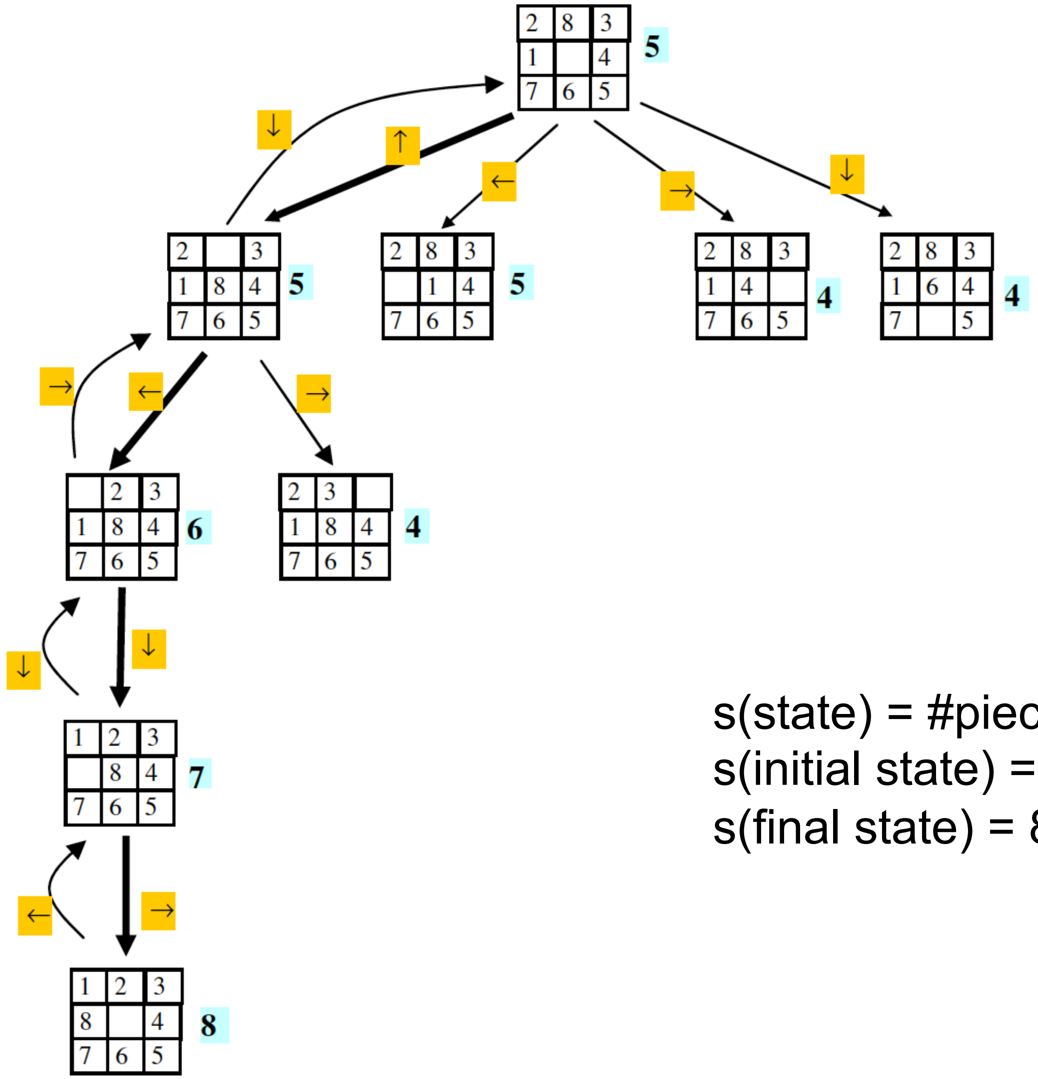
  - dangers: local maxima, plateaux

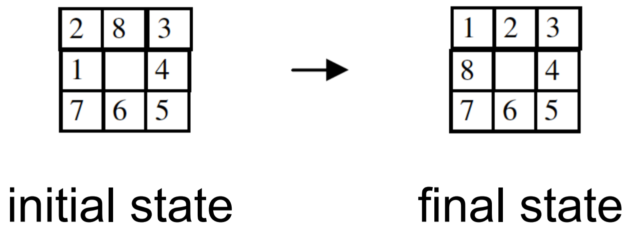


Hill climbing

```
procedure hill-climbing(initial-state)
begin
  current-state <- initial-state;
  while(current-state) {
   if (current-state is a final state) return current-state;
   all-new-neighbour-states <- set of all states that can be
obtained from current-state by operators possible to be applied
here;
   all-new-neighbour-states <- all-new-neighbour-states minus all
states already visited;
   sort all-new-neighbour-states in the descending order of their
cost values;
   all-new-neighbour-states <- all-new-neighbour-states minus all
states having a lower cost then current-state;
   if (all-new-neighbour-states ≠ ∅) current-state <- the first
state ranked in all-new-neighbour-states);
   else current-state = nil;
   }
  return FAIL;
end
```

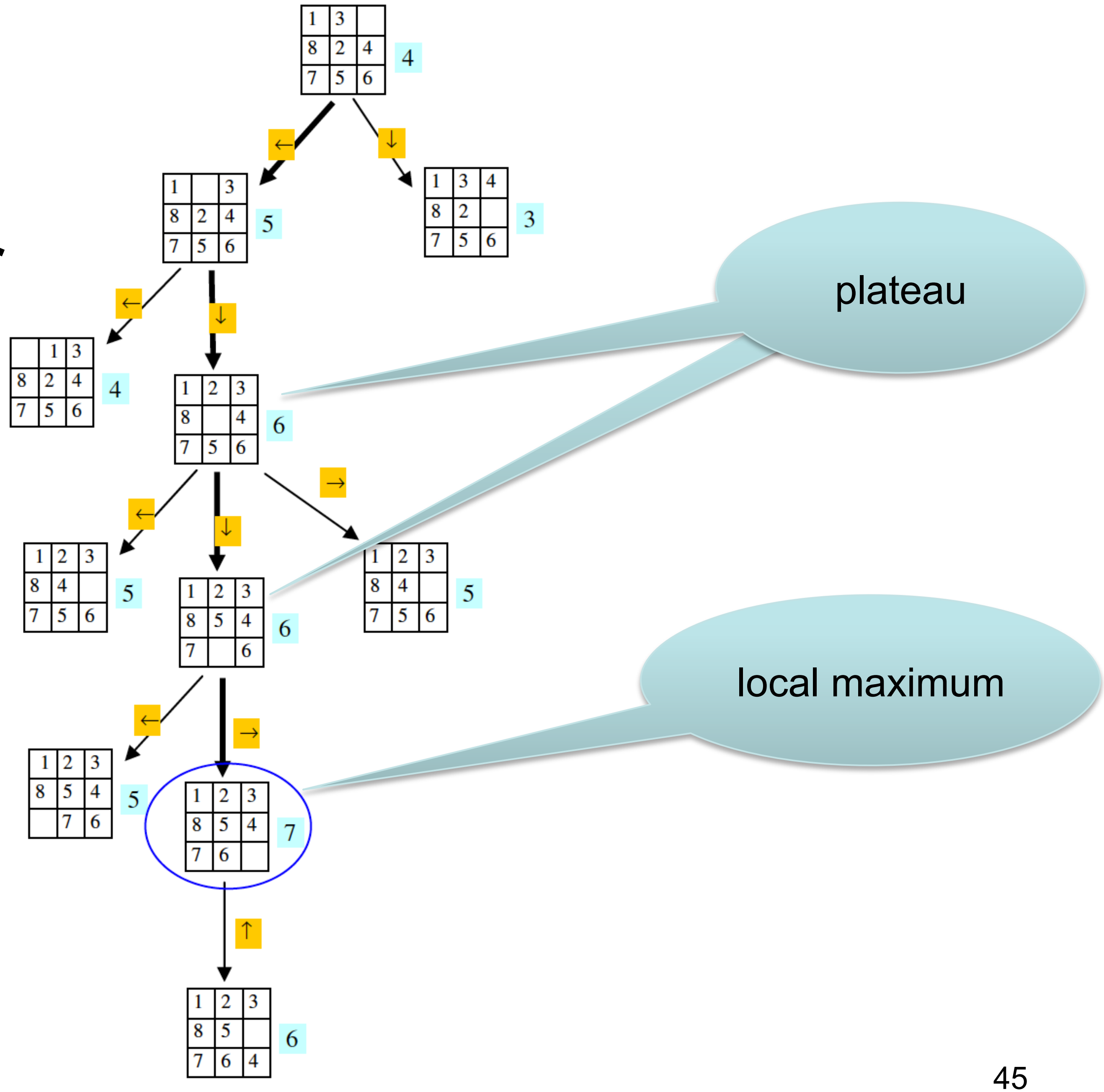
8-puzzle: a happy ending search using *hill-climbing*





s(state) = #pieces in final positions
s(initial state) = 5
s(final state) = 8

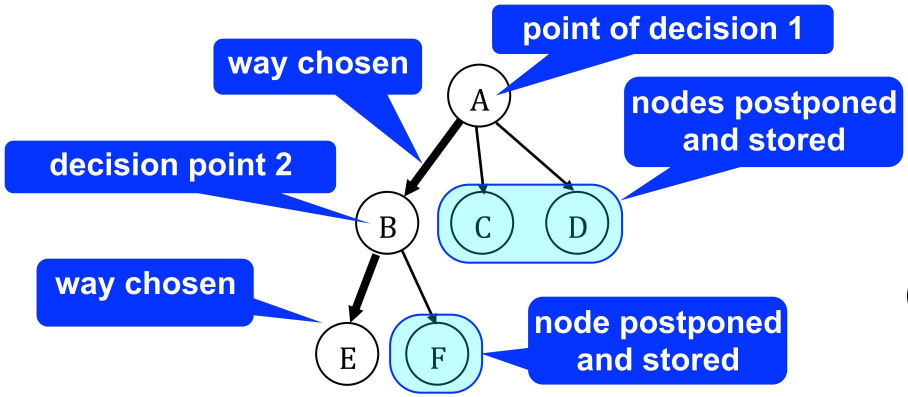
8-puzzle: search interrupted by a local maximum



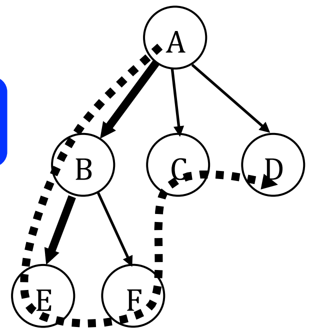
Step 5

Tentative strategies: *backtracking*

- If a state has no successors ”take the track back”
  - needed: a memory which stores at any step the neighbouring unvisited states



a. In any point in which a choice is made, the unexplored yet states are stored



b. When the storing space is represented as a stack, the visiting is performed in the depth-first order

Pasul 5

Backtracking hill-climbing

```
procedure backtracking-hill-climbing(initial-state)
begin
  heap <- initial-state  ° ∅;
  solution <- ∅;
  while(heap) {
   current-state <- first(heap);
   heap <- rest(heap);
   solution <- solution ° current-state;
   if (current-state e stare finală) return solution;
   all-new-neighbour-states <- setul stărilor ce pot fi
obţinute din current-state prin operatorii aplicabili ei;
   all-new-neighbour-states <- all-new-neighbour-states \
toate stările deja vizitate;
   heap <- heap ° all-new-neighbour-states;
   sort heap descendent după valorile funcției cost;
   heap <- heap \ stările de valori mai mici decât a lui
current-state;
   }
  return FAIL;
end
```

Systematic search methods (*brute-force*)

Step 5

- **Depth-first search – DFS**
  – memory is a stack

```
function depthFirstSearch(root)
begin
  stack <- push(root, ∅); solution <- ∅;
  while (stack not empty)
   { node <- pop(stack);
     if goal(node) then {
       solution <- node ° ∅; ancestor <- node.father;
       while (ancestor) {
         solution <- solution ° ancestor;
         ancestor <- ancestor ° father;
         }
       return solution;
     }
     else push(node's successors, stack);
   }
  return FAIL;
end
```

Systematic search methods (*brute-force*)

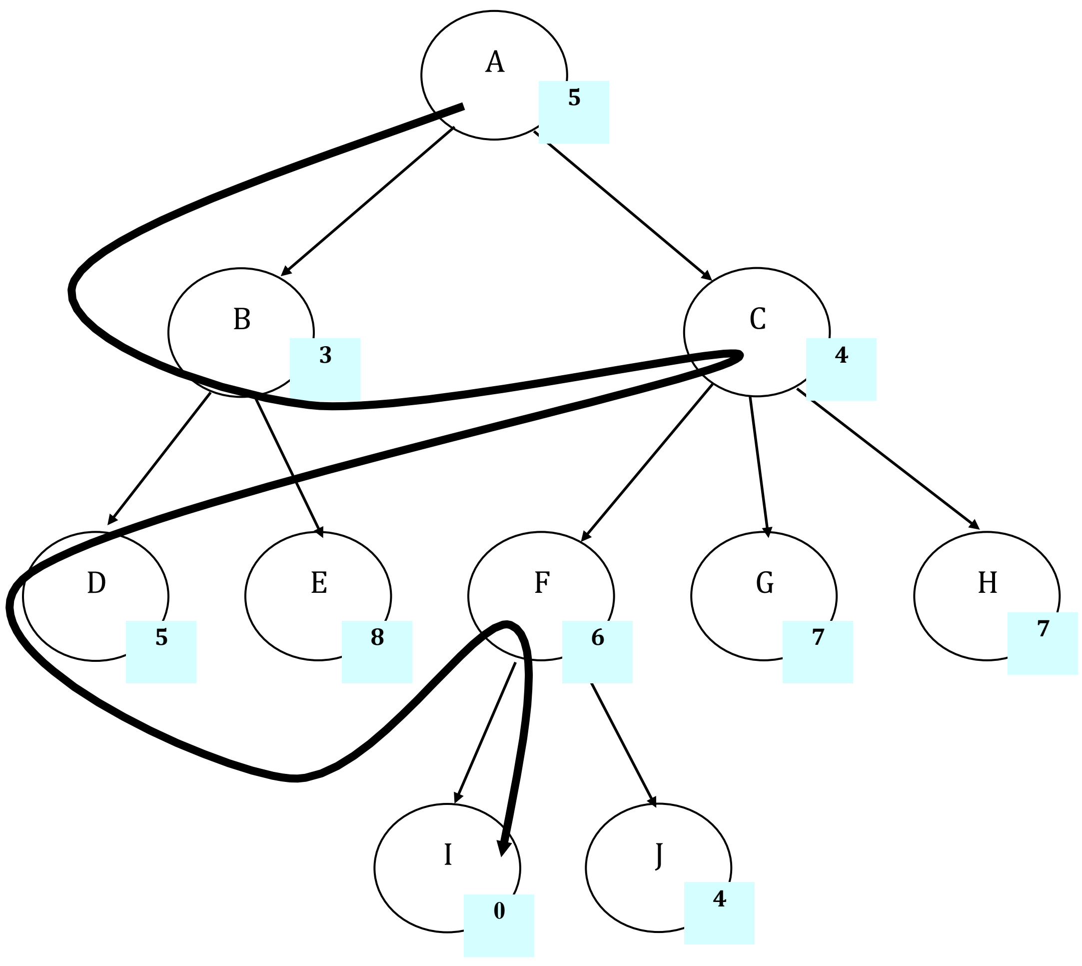
Step 5

- **Breadth-first search – BFS**
  - memory is a queue

```
function breadthFirstSearch(root)
begin
  queue <- in(root, ∅); solution <- ∅;
  while (queue not empty)
   { node <- out(queue);
     if goal(node) then {
       solution <- node ° ∅; ancestor <- node.father;
       while (ancestor) {
         solution <- solution ° ancestor;
         ancestor <- ancestor ° father;
         }
       return solution;
     }
     else in(node's successors, queue);
   }
  return FAIL;
end
```

Step 5

Example: *best-first search*



Systematic search methods (*brute-force*)

Step 5

- **Best-first search**
  - memory is a list; nodes got scores by a cost function

```
function bestFirstSearch(root)
begin
  list <- include(root, ∅);
  while (list not empty)
   { node <- get-first(list);
     if goal(node) then {
       solution <- node ° ∅; ancestor <- node.father;
       while (ancestor) {
          solution <- solution ° ancestor;
          ancestor <- ancestor ° father;
          }
       return solution;
     else
      { for each successor of node {
                 list <- successor ° list); }
        sort list descending;
      }
   }
  return FAIL;
end
```