Overview

I Artificial Intelligence

II Problem Solving

3. Solving Problems by Searching

4. Search in Compex Environments

5. Adversarial Search and Games

6. Constraint Satisfaction Problems

III Knowledge, Reasoning, Planning

IV Uncertain Knowledge and Reasoning

V Machine Learning

VI Communicating, Perceiving, and Acting

VII Conclusions

4. Search in Compex Environments

- Local Search and Optimiziation Problems
- Local Search in Continuous Spaces
- Search with Nondeterministic Agents
- Search in Partially Observable Environments
- Online Search Agents and Unknown Environments

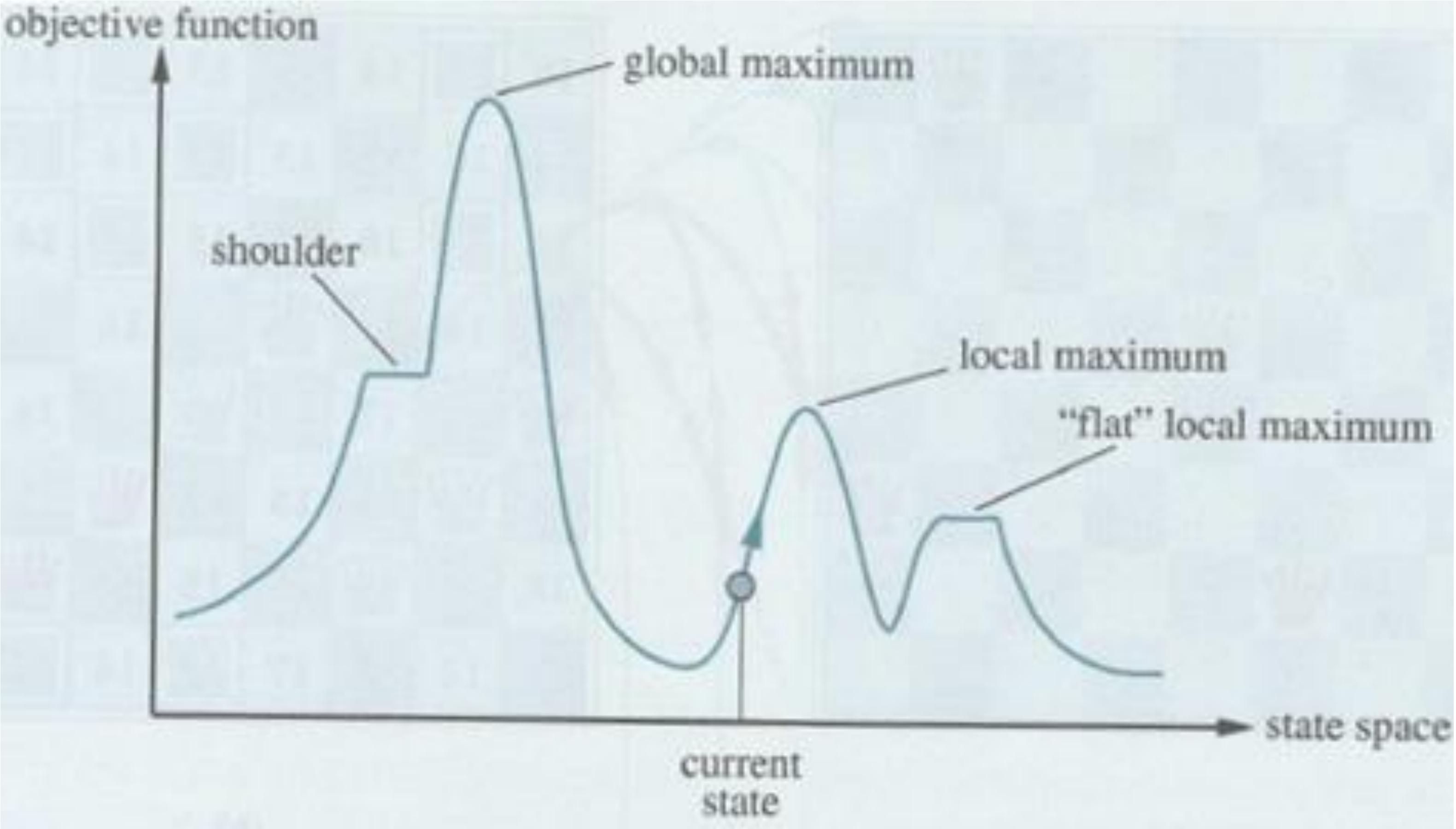
Local Search algorithms

- Local search algorithms are useful, if only the solution is relevant, not the path to the solution.
  - Example: 8-queens-problem
  - Other Examples: VLSI-Layout, factory floor layout, job shop scheduling, automatic programming, telecommunication network optimization, crop planning, portfolio managment
- Idea: Start with a **complete-state formulation** and **improve it by stepwise variations**
- Algorithms:
  - **Hill-Climbing**
  - **Simulated Annealing**
  - **Local Beam Search**
  - **Genetic Algorithms**

Hill Climbing Search

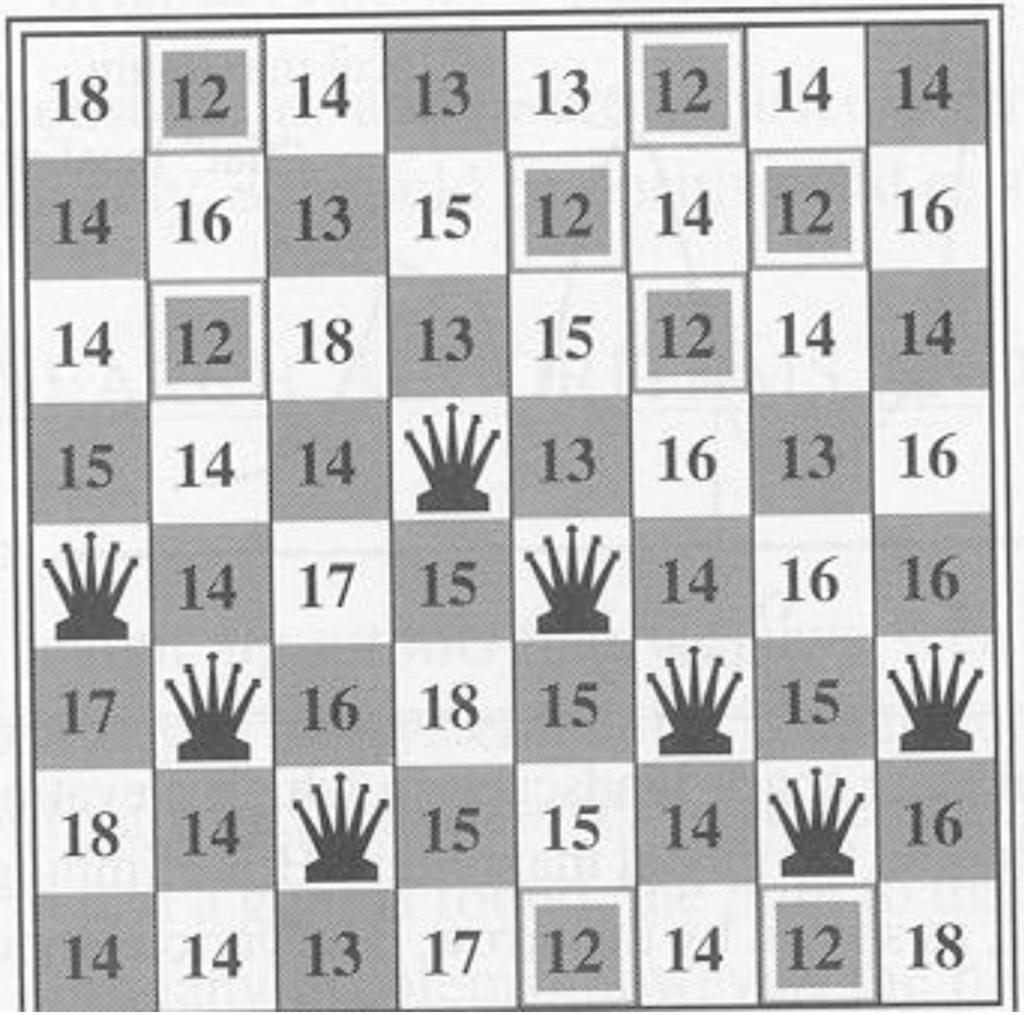
- Choose alway the highest-value sucessor node and return the current node, if no improvement is possible
    - No search tree, no backtracking → very efficient
    - Problems: Local maximas, plateaus, shoulders, flat local maximas

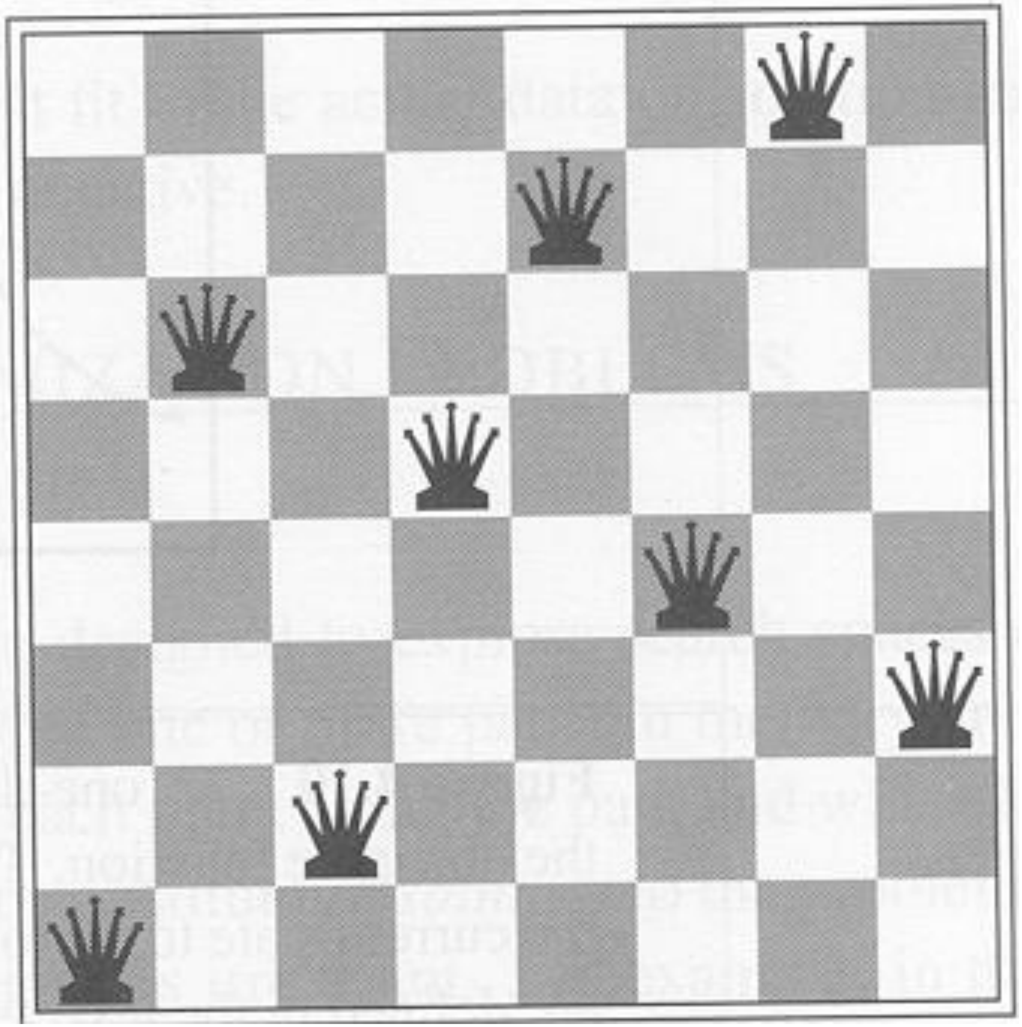
```
function HILL-CLIMBING(problem) returns a state that is a local maximum
  current ← problem.INITIAL
  while true do
      neighbor ← a highest-valued successor state of current
      if VALUE(neighbor) ≤ VALUE(current) then return current
      current ← neighbor
```



Hill-Climing: Example from 8-queens problem

- Left: In each step, a queen is moved to reduce the total number of conflicts (e.g. to 12)
- Right: Local minima, where hill-climbing gets stuck





Improvements for Hill-Climbing

- **Sideways move**: Limited; to pass shoulders but to avoid infinite loops in e.g. flat maxima
- **Stochastic hill-climbing**: Random selection of all improvements
  - **First-choice hill-climbing**: Generate sucessor-nodes and take the first improvement
- **Random-restart hill-climbing**: Repeat hill-climbing with randomly generated initial states

- **Simulated Annealing:** Allow worsening with a low probability
- **Local beam search:** Simultaneous search an several paths

Simulated Annealing

- Idea: Allow „down-hill“ steps to overcome local maxima
  - Model: Gradually cooling of hot material (e.g. metallurgy), freezing of water etc.
- Problem: Control of down-hill steps
- Solution:
  - Random factor for choosing steps: if it improves the solution, choose it with 100%, if not, choose it with a probability depending on the degree of worsening
  - Terminate, by lowering the probability of down-hill steps continuously

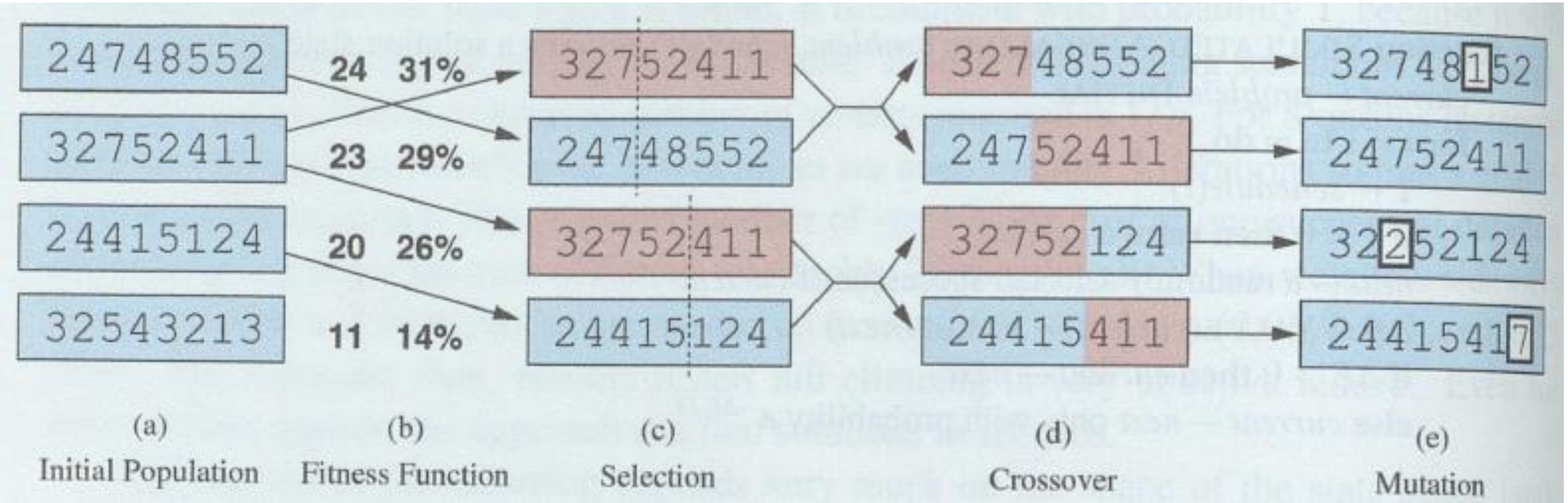
```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
   current ← problem.INITIAL
   for t = 1 to ∞ do
      T ← schedule(t)
      if T = 0 then return current
      next ← a randomly selected successor of current
      ΔE ← VALUE(current) – VALUE(next)
      if ΔE > 0 then current ← next
      else current ← next only with probability e^(−ΔE/T)
```

Local Beam Search

- **Idea:** Keep a set of k nodes („beam“) instead of one node. From one iteration to the next, the k best sucessors are chosen.
- **Difference to random restart hill-climbing**: Passing of information among parallel searches and concentration of search in promising regions.
- **Problem:** Concentration in a small region of the space (e.g. around a high, but local maxima).
- Improvement: **Stochastic generation of sucessors** nodes similar to stochastic hill-climbing for generation of diversity.

Evolutionary Algorithms

- Variant of stochastic beam search, which can **combine useful solution-parts** (blocks)
- Main steps (data structure is pool of solutions called „**population**“)
  - Repeat until **termination criterium**
    1. **Select:** Select two solutions in population according to fitness
    2. **Recombine:** Generate new solution from both parents
    3. **Repair** (optionally): Repair (improve) new solution
    4. **Mutate:** Modify new solution randomly by mutations
  - **Choose best solutions** (from new population)



Variations of Evolutionary Algorithms

- **Size of population**
- **Representation of individuals**
  - Genetic algorithms: String over an alphabet (in biology DNA over alphabet ACGT)
  - Evolution strategies: Indiviual is a sequence of real numbers
  - Genetic programming: Individual is a computer program
- Mixing p parents: p = 1: Stochastic beam search; p = 2: Standard case; p > 2: Possible
- **Selection process**: Different functions usually depending on fitness
- **Recombination procedure**: Standard is to (randomly) select a crossover point
- **Mutation rate:** Determines, how often an offstring have random mutations
- **Makeup of next generation**: Keep top-scoring parents? Discard individuals below threshhold?

Examples from 8-queens and Assignment

N1 to A1
N2 to A2
N3 to A3
N4 to A4
-5

N1 to A2
N2 to A1
N3 to A4
N4 to A3
-3

N1 to A2
N2 to A3
N3 to A1
N4 to A4
-2

Parent Generation

N1 to A1
N2 to A2
N3 to A4
N4 to A3
-3

N1 to A2
N2 to A3
N3 to A4
N4 to A3

Repair

N1 to A2
N2 to A3
N3 to A4
N4 to A1
0

Child Generation

Genetic Algorithm

```
function GENETIC-ALGORITHM(population, fitness) returns an individual
  repeat
      weights ← WEIGHTED-BY(population, fitness)
      population2 ← empty list
      for i = 1 to SIZE(population) do
          parent1, parent2 ← WEIGHTED-RANDOM-CHOICES(population, weights, 2)
          child ← REPRODUCE(parent1, parent2)
          if (small random probability) then child ← MUTATE(child)
          add child to population2
      population ← population2
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to fitness

function REPRODUCE(parent1, parent2) returns an individual
  n ← LENGTH(parent1)
  c ← random number from 1 to n
  return APPEND(SUBSTRING(parent1, 1, c), SUBSTRING(parent2, c + 1, n))
```

- **Population:** Ordered list of individuals
- **Weight:** List of fitness values for each individual
- **Fitness:** Function to compute weights

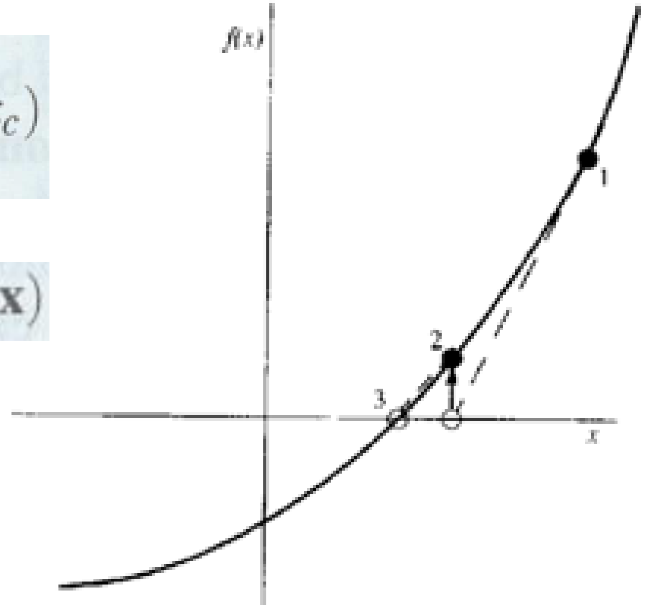
Local Search in Continuous Spaces

- Example problem: Place 3 airports in Romania with minimum square distance to the cities
  - Input: Coordinates of cities C_i (maybe with population weights), whose next airput is i
  - Output: Coordinate (x,y) of the three airports (x_1, y_1; x_2, y_2; x_3 y_3)
  - Optimization criteria: $$f(\mathbf{x}) = f(x_1,y_1,x_2,y_2,x_3,y_3) = \sum_{i=1} \sum_{c \in C_i} (x_i - x_c)^2 + (y_i - y_c)^2$$

Solution Approaches

- **Discretization of neighborhoods**
  - Moving an airport in x or y-direction with a constant d
  - With 6 variables 12 sucessors per state
  - Application of any search algorithms
- **Local gradient search**
  - Derivation of goal function for each variable $$\frac{\partial f}{\partial x_1} = 2 \sum_{c \in C_1} (x_1 - x_c)$$
  - Gradient of goal function is a vector indicating the size and direction of the steepest slope (α = step seize) $$\mathbf{x} \leftarrow \mathbf{x} + \alpha \nabla f(\mathbf{x})$$
  - Newton-Raphson algorithm often effective, also for matrices
- **Linear programming**
  - Constraints for goal function must be linear (e.g. airports should not be placed in mountains)
  - Goal function must be linear too (e.g. sum of distances)

Newton-Raphson for solving equations of the form g(x) = 0
$x \leftarrow x - g(x)/g'(x)$

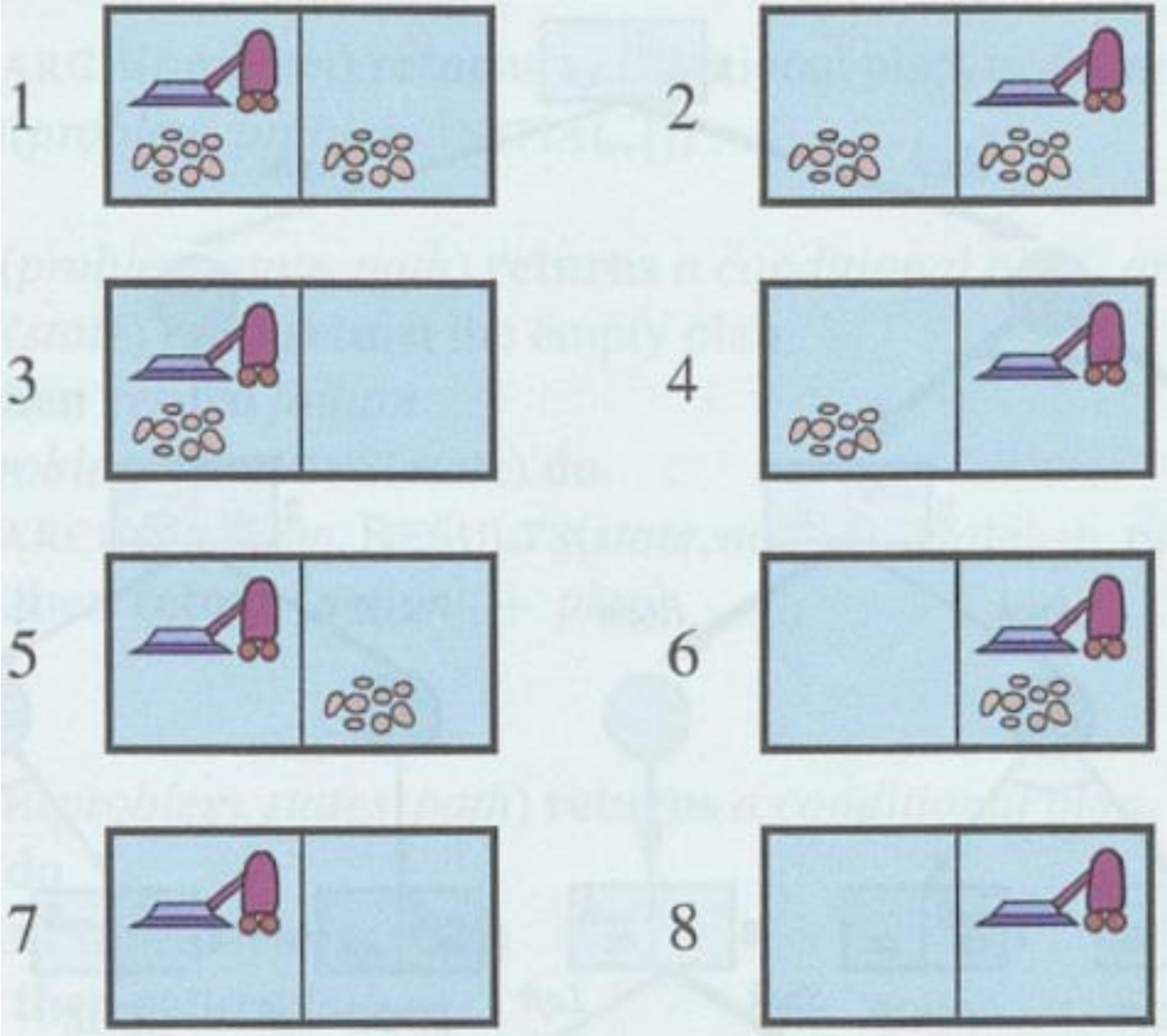


Search with Nondeterministic Actions

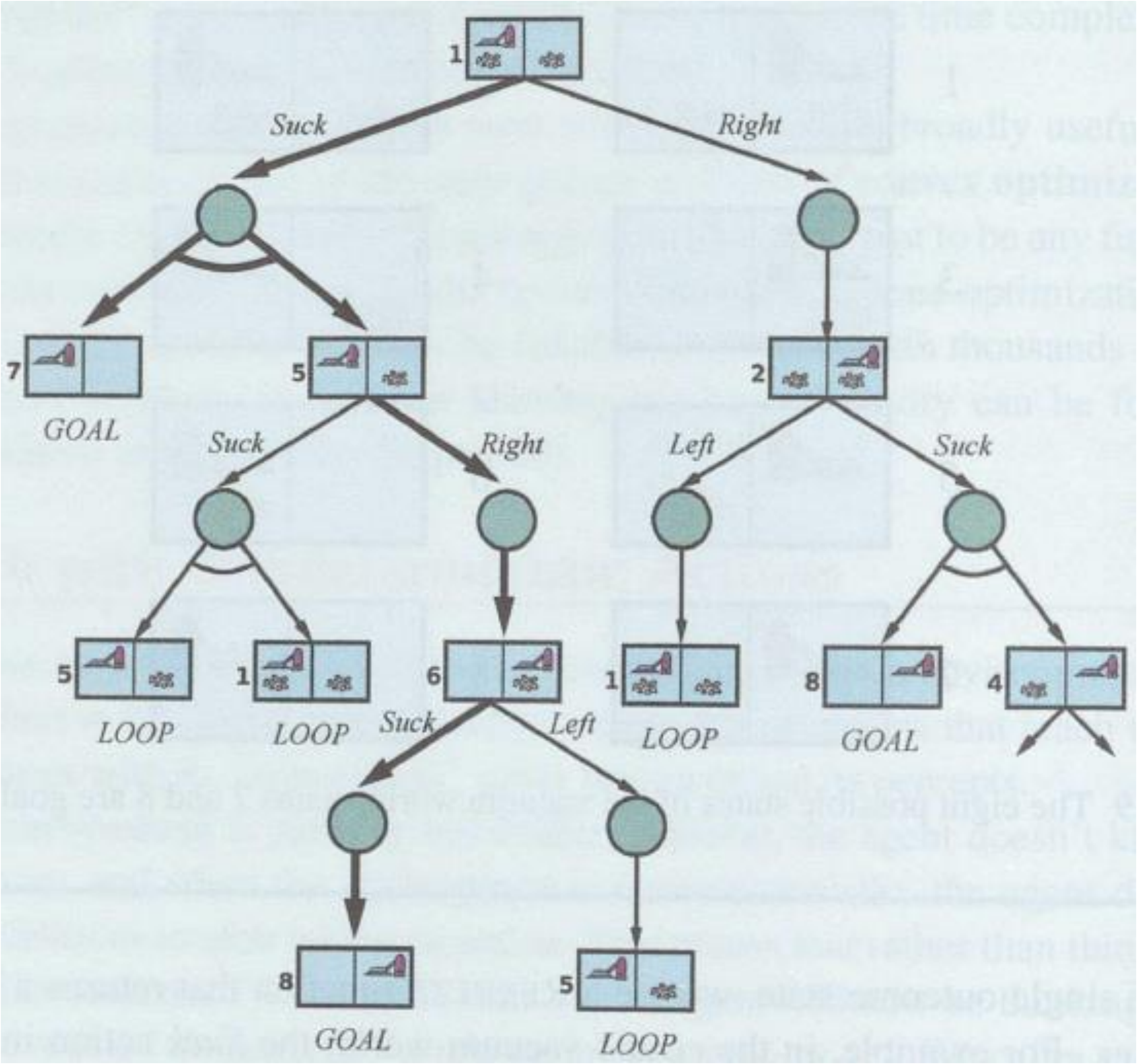
- Example problem:
  - Vacuum world with erratic actions:
    - Suck of a dirty square makes it clean and sometimes cleans the adjacent square too
    - Suck of a clean square sometimes deposits dirt on it
- Solution in state 1
  - Deterministic: No plan available
  - Conditional plan:
    1. Suck
    2. if state = 5

       then [Right, Suck]

       else [ ]



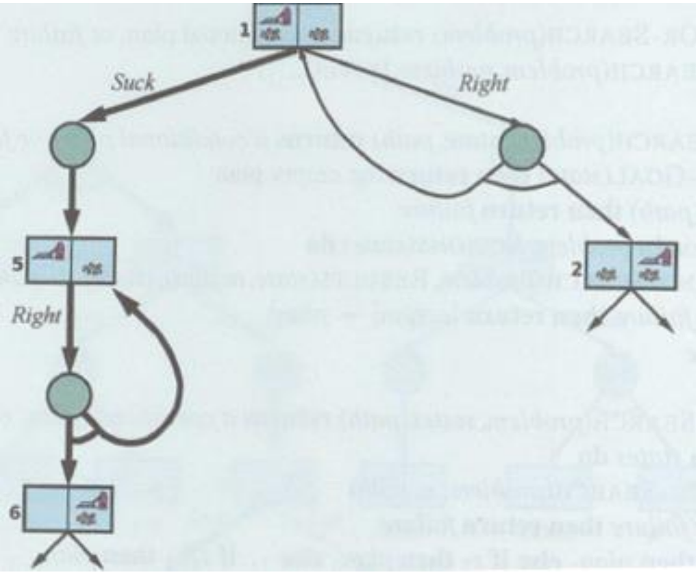
Solution: And-Or-Search Tree



- OR-Nodes: rectangles
- AND-Nodes: circles
  - All results must be handled
  - The state must be checked
- Solution is a subtree of the complete search tree that has …
  - a goal node at every leaf
  - specifies one action for each of its OR-nodes
  - includes every outcome branch at each of its AND-nodes
- Bold arrows: Solution

And-Or Graph Search and Cycles

- Different search algorithms possible, e.g. depth-first, breadth-first, best-first
- Next slide: Recursive depth-first algorithm
- Key aspect: dealing with cycles (arising often in nondetermistic problems)
  - Finding a plan avoiding cycles
  - Keep trying an indetermistic action until the desired outcome occurs
    - Example in a slippery vacuum world, where movements may fail, i.e. the agent may stay in the same location:
      - [Suck,   while State = 5 do Right,   Suck]
      - [Suck, L_1: Right, **if** State = unchanged **then** L_1 **else** Suck]
    - Might result in infinite loop
    - Try a limited number of repetitions (like inserting an electronic card)



Conditional And-Or-Search Algorithm

Solution is a conditional plan considering every nonterministic outcome

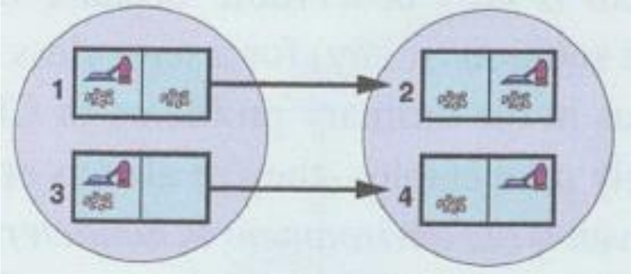
```
function AND-OR-SEARCH(problem) returns a conditional plan, or failure
   return OR-SEARCH(problem, problem.INITIAL, [])

function OR-SEARCH(problem, state, path) returns a conditional plan, or failure
   if problem.IS-GOAL(state) then return the empty plan
   if IS-CYCLE(path) then return failure
   for each action in problem.ACTIONS(state) do
      plan ← AND-SEARCH(problem, RESULTS(state, action), [state] + path])
      if plan ≠ failure then return [action] + plan]
   return failure

function AND-SEARCH(problem, states, path) returns a conditional plan, or failure
   for each s_i in states do
      plan_i ← OR-SEARCH(problem, s_i, path)
      if plan_i = failure then return failure
   return [if s_1 then plan_1 else if s_2 then plan_2 else ...if s_{n-1} then plan_{n-1} else plan_n]
```

Search in Partial Observable Environments

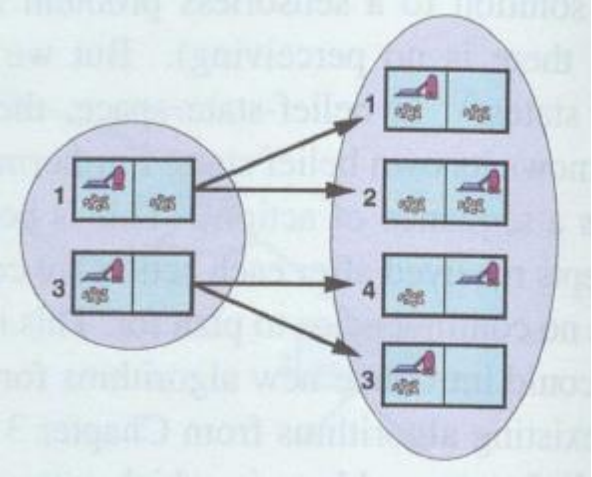
- Example problem: Vacuum world without or with partial sensor information
- Solution approaches: **Search in belief states** and update of belief states
  - **Belief state:** All possible states compatible with the current information
    - Should include the physical state

Search without Observation

- Sensorless (conformant) problem
- Advantages: Sensors are expensive and often not reliable
- Examples:
  - Producing a base state for machines (e.g. restart a computer)
  - Infections: Prescribing broadband antibiotics
  - Sensorless vacuum world:
    - Effect of the deterministic action „right“ in a belief state:



    - Effect of the indeterministic action „right“ in a belief state:

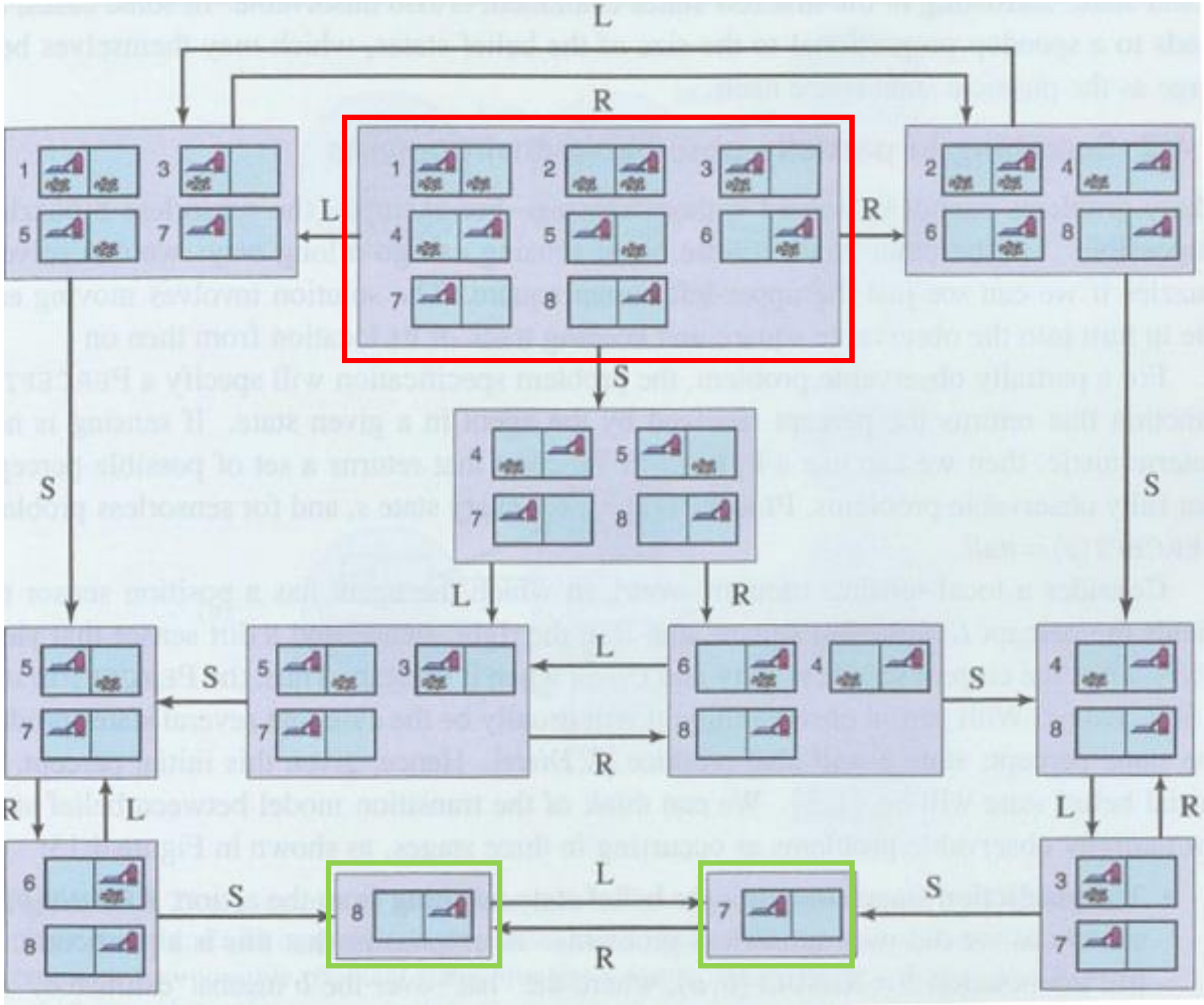


Search State for Sensorlos Planning in Vacuum World

… with deterministic actions

- Start state in red
- goal states in green

- Possible solutions:
  [right, suck, left, suck]
  [left, suck, right, suck]



Construction of Search Space in Belief State (1)

- **Belief state:** Set of all possible physical states
  - With N physical states, there are 2^N belief states
- **Initial state:** Without knowledge all 2^N belief states
- **Actions:**
  - Assumption: Illegal actions have no effect in environment
    - New belief state b' = union of all actions a in every physical state p of the current belief state b

$$\text{ACTIONS}(b) = \bigcup_{s \in b} \text{ACTIONS}_P(s)$$

  - Assumption: Illegal actions might cause (great) damage
    - It is safer to allow only actions, which are legal in all the states (computed by intersection)

Construction of Search Space in Belief State (2)

- **Transition model:**
  - For deterministic actions: the new belief state b has one result state s for each of the possible actions a: set b‘ = Result (b, a) = {s‘ : s‘ = Result$_p$ (s,a) und s $\in$ b}
  - For inderministic action: the new belief state may have several result states for each of the possible actions: set b‘ = Result (b, a) = {s‘ : s‘$\in$ Result$_p$ (s,a) und s $\in$ b} = $\bigcup_{s \in b} \text{RESULTS}_P(s,a)$
  - The size of the result set may decrease for deterministic actions and may increase for interdeterministic actions.
- **Goal test:** All physical states p in belief state b should satisfy the goal test
- **Action costs:** Could be difficult, if the same action has different costs if applied to different states, otherwise straightforward.

Search Algorithms in Sensorless Planning

- Apply **ordinary search algorithms** in belief state representation
  - Special treatment of supersets: if a superset {1, 3, 5, 7} is solved, all subsets, e.g. (e.g. {5,7}) are solved too.
  - The representation of one belief state is already exponential
    - Compacter representation of belief states by using logic instead of enumaration, e.g. after action [right]: „not in left cell“
  - Very large space 2^N instead of N, and even N is often too large for e.g. breadth-first search or A* search.
- **Incremental belief state search**
  - Search a solution for just one physical state in the belief state
  - Check, whether this plan works also for the other physical states
  - If not, search for another plan in the first physical state etc.
  - Is efficient to find out, whether a problem has no solution

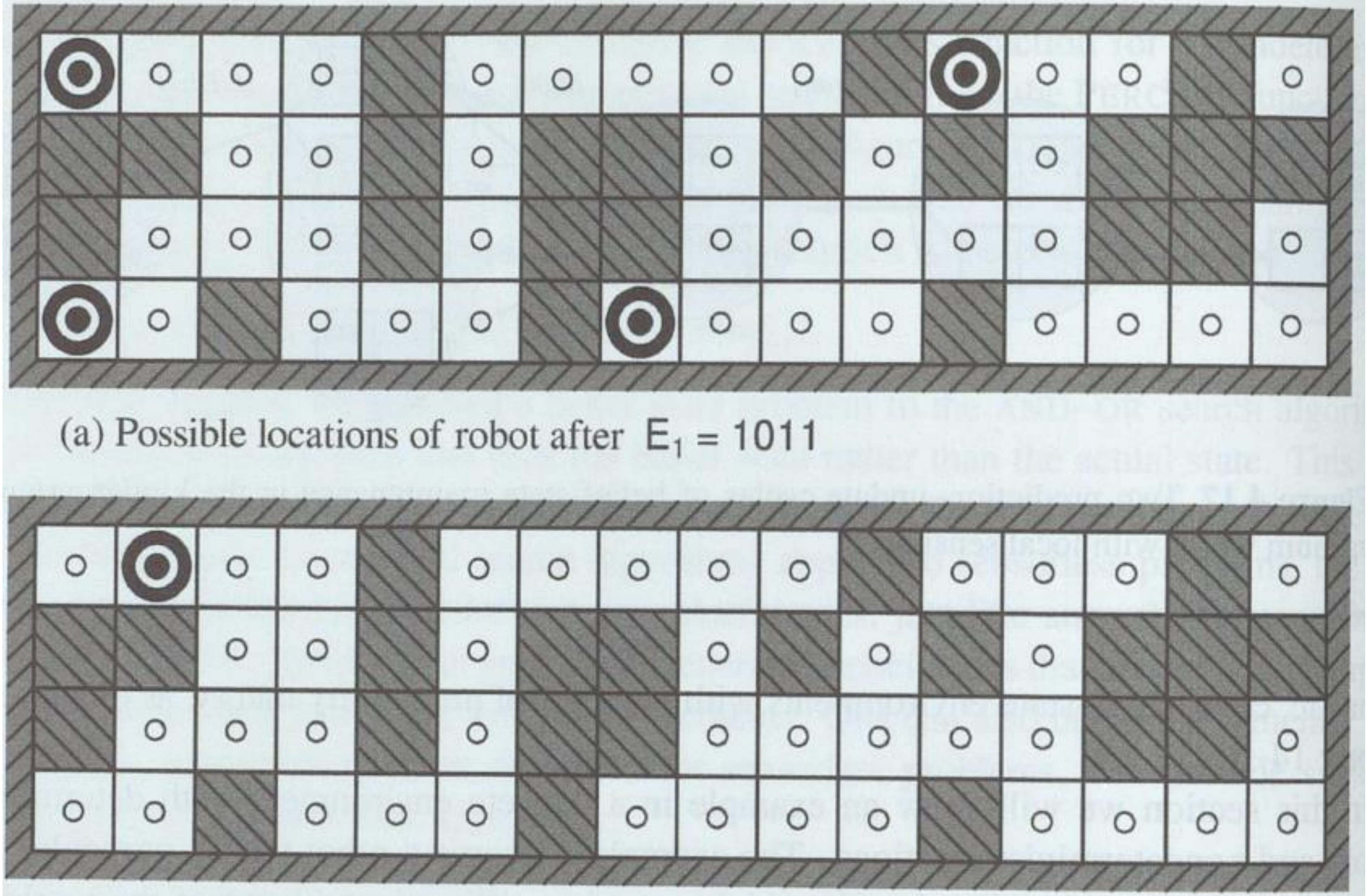
Searching in Partially Observable Environments

- Sensorless planning often impossible (e.g. 8-puzzle)
- A little bit of sensing often useful (e.g. one square in 8-puzzle sufficient)

- Algorithm aspects
  - Prediction step similar to sensorloss planning
    - But results in possible percepts, that could be observed in predicted belief state
  - Update step computes for each possible percept new belief state from percept
    - For deterministic sensing, belief states for different possible percepts are disjoint
  - AND-OR search algorithm applicable
    - Solution is a conditional plan; agent tests condition and execute appropriate branch
    - Agent updates its belief state after each action (and percept)
    - Probabilistic information for indeterministic actions would be useful (treated later)

Example: Robot Navigation with non-deterministic Actions

- A robot has 4 sonar sensors telling him, whether there is a wall, e.g. 1011 means North: Wall; East: No wall; South and West: Wall.
- Robot has 4 action to move in each direction.
    - However, move is non-deterministically, so the robot may end on any adjacent field.



(a) Possible locations of robot after $E_1 = 1011$

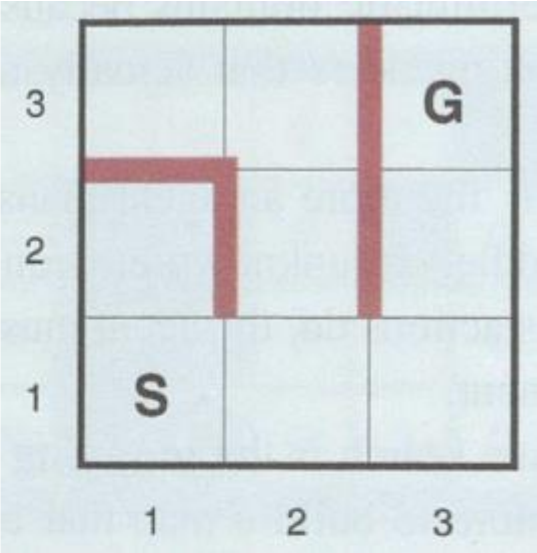
(b) Possible locations of robot after $E_1 = 1011$, $E_2 = 1010$

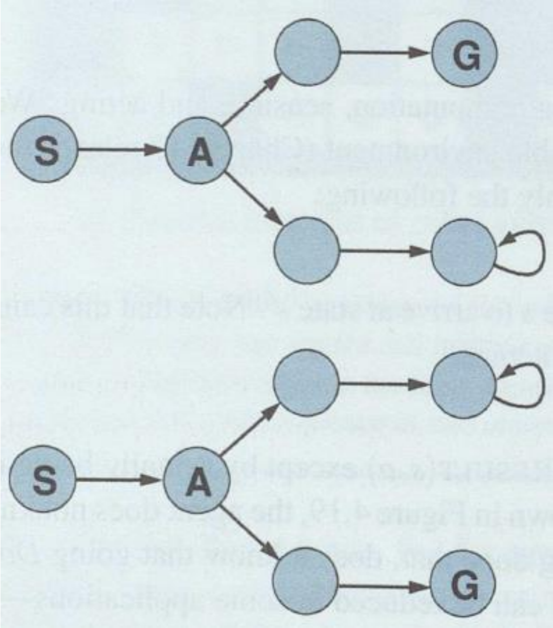
Online-Search

- Offline-Search: Solution is computed before execution
- Online-Search: Computation and Execution overlap
- Necessary in unknown environments (but goal and action with costs are known)
- Problems: Irreversible action leading in dead ends (e.g. robot drops off a cliff, one-way streets)

Solution Approaches for Online-Search (1)

- **Online-Depth-First Search:**
  - In each state, follow the action given by depth-first search
  - If no action possible, go back to last branch (must be stored) and try another action
  - In maze (right), the agent would walk: (1,1) → (1,2) → (1,1) → (2,1) → (2,2) → (2,3) → (1,3) → (2,3) → (2,2) → (2,1) → (3,1) → (3,2) → (3,3)
  - Indefinite paths would prevent the agent from finding the goal (right below)
- Online variant of **iterative deepening depth-first search** would avoid indefinite paths and would find „shallow" goals much faster.





Solution Approaches for Online-Search (2)

- **Hill-climbing search** (does not explore the environment, might get stuck)
- Hill-climbing with random restarts and random walk (very slow)
- Augmenting hill-climbing with memory: For each state, an heuristic cost-estimate to reach the goal is stored and updated with more experience
- **Real-time A* (LRTA*):** Agent builts a map in its result table:
  - The estimated cost to reach a goal through neighbor state s‘ is cost to get to s‘ + estimated cost to goal, i.e.: c(s, a, s‘) + H (s‘).
  - Example: cost increase in red circle to escape plateau
  - Prefer untried states u by H(u) = 0