Course 16 :198 :440 : Introduction To Artificial Intelligence
Lecture 4

Solving Problems by Searching: Beyond Classical Search

Abdeslam Boularias

Wednesday, February 5, 2025

Outline

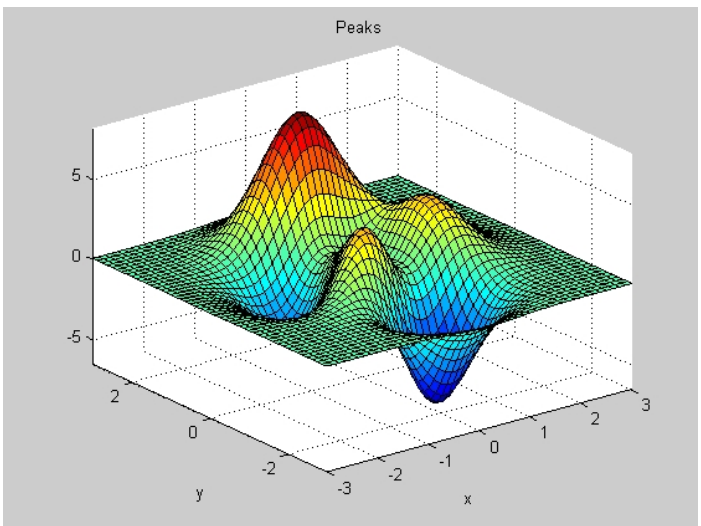
We relax the simplifying assumptions of the previous lectures, and we get closer to real-world applications

1. Local search
2. Continuous state spaces
3. Stochastic actions
4. Partial observations

Local search

- So far, we have focused on searching for the best path (sequence of state-actions) that leads to a goal.
- In many problems, such as the 8-queens problem, we do not care about the path to the goal. We only care about finding the goal.
- If paths are not relevant, one can start with an initial state and move only to **neighbors** of that state.

Local search is suitable for **optimization problems**, where we aim to find the best state according to a given **objective function**.



Local search

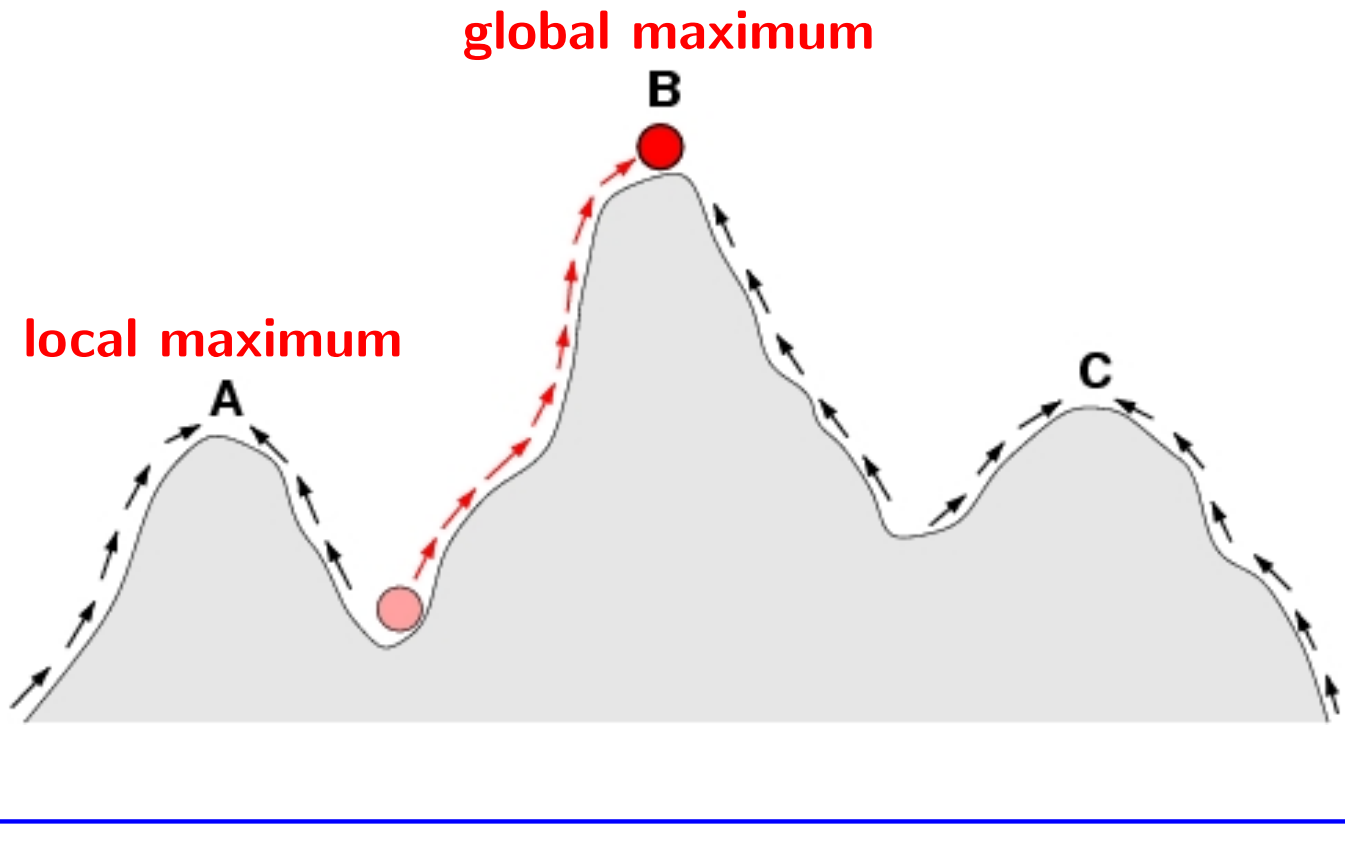
- So far, we have focused on searching for the best path (sequence of state-actions) that leads to a goal.
- In many problems, such as the 8-queens problem, we do not care about the path to the goal. We only care about finding the goal.
- If paths are not relevant, one can start with an initial state and move only to **neighbors** of that state.

Local search is suitable for **optimization problems**, where we aim to find the best state according to a given **objective function**.

Local search algorithms have two key advantages :

1. They use little memory as they usually do not need to remember paths.
2. They can often find reasonable solutions in large (or infinite) state spaces where systematic search is unsuitable.

Local search



$$\text{Optimum} = \begin{cases} \text{Maximum} & \text{if the function is an objective function (reward),} \\ \text{Minimum} & \text{if the function is a cost function (penalty).} \end{cases}$$

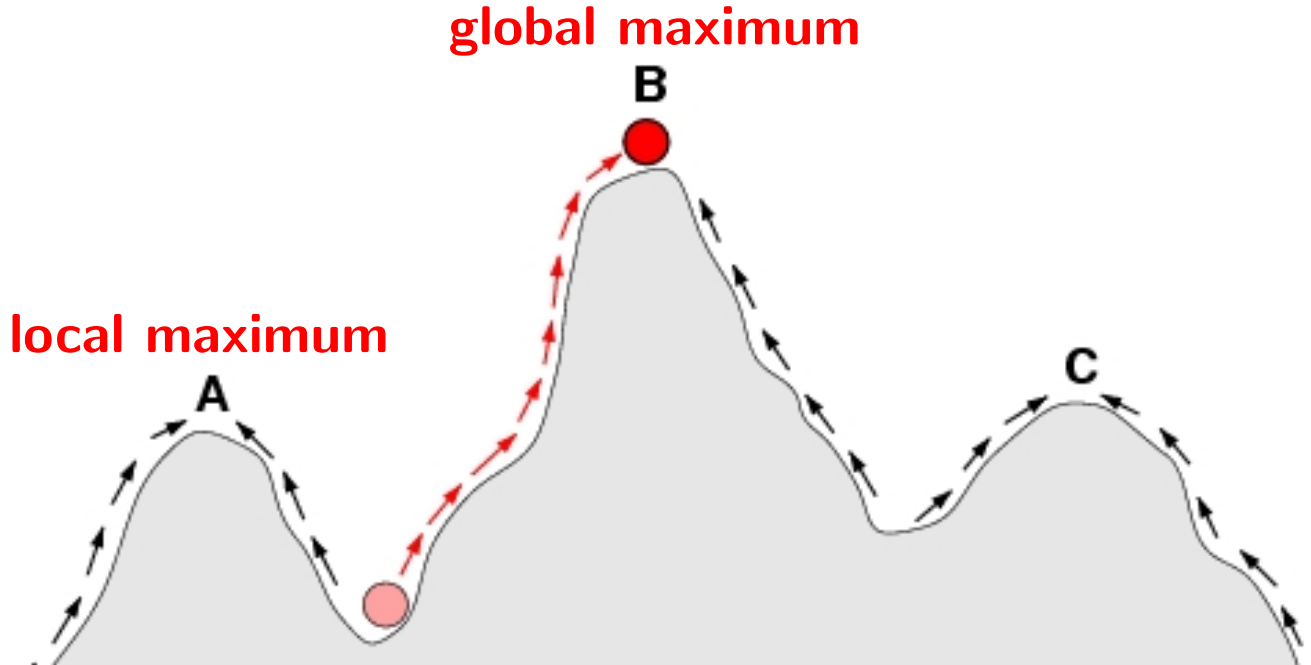
Every global optimum is a local optimum, the inverse is not always correct.

Hill-climbing search (a.k.a. greedy local search)

Continually move in the direction of increasing value (uphill).

function HILL-CLIMBING($problem$) **returns** a state that is a local maximum

$current \leftarrow$ MAKE-NODE($problem$.INITIAL-STATE)
loop do
  $neighbor \leftarrow$ a highest-valued successor of $current$
  if neighbor.VALUE $\leq$ current.VALUE **then return** $current$.STATE
  $current \leftarrow neighbor$

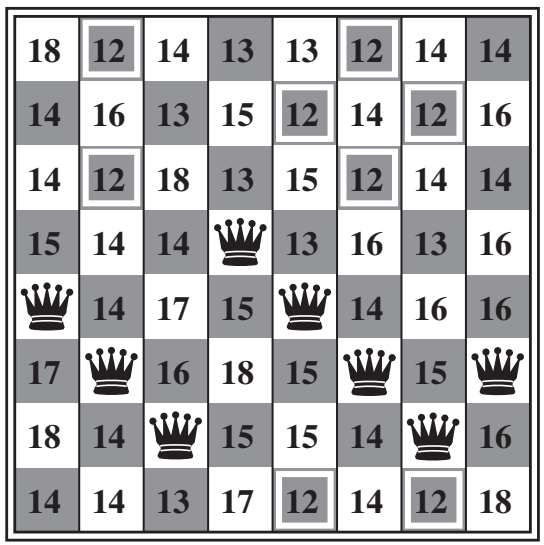


An example of hill-climbing search

Cost function : number of pairs of attacking queens.

function HILL-CLIMBING($problem$) **returns** a state that is a local maximum

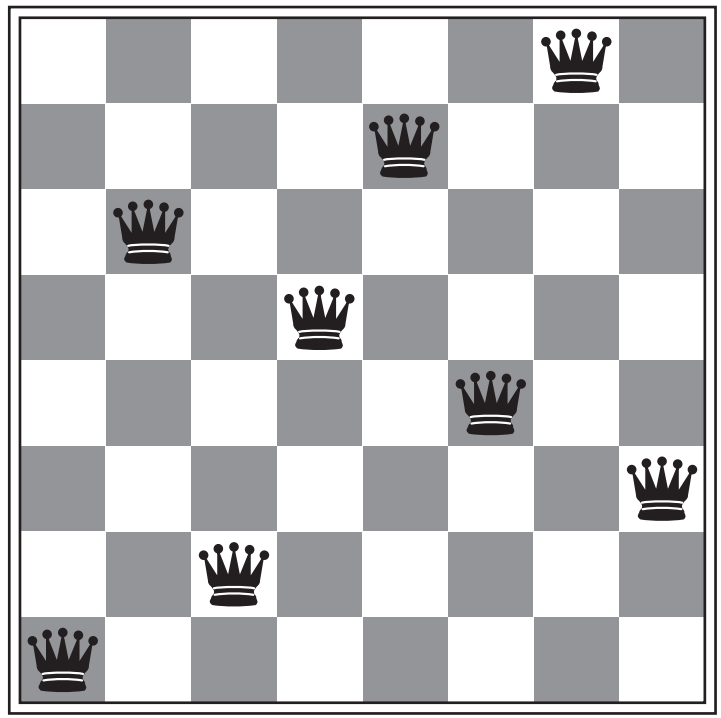
$current \leftarrow$ MAKE-NODE($problem$.INITIAL-STATE)
loop do
$neighbor \leftarrow$ a highest-valued successor of $current$
if neighbor.VALUE $\leq$ current.VALUE **then return** $current$.STATE
$current \leftarrow neighbor$



A state with cost function equal to 17. Numbers indicate the value of moving each queen vertically.

An example of hill-climbing search

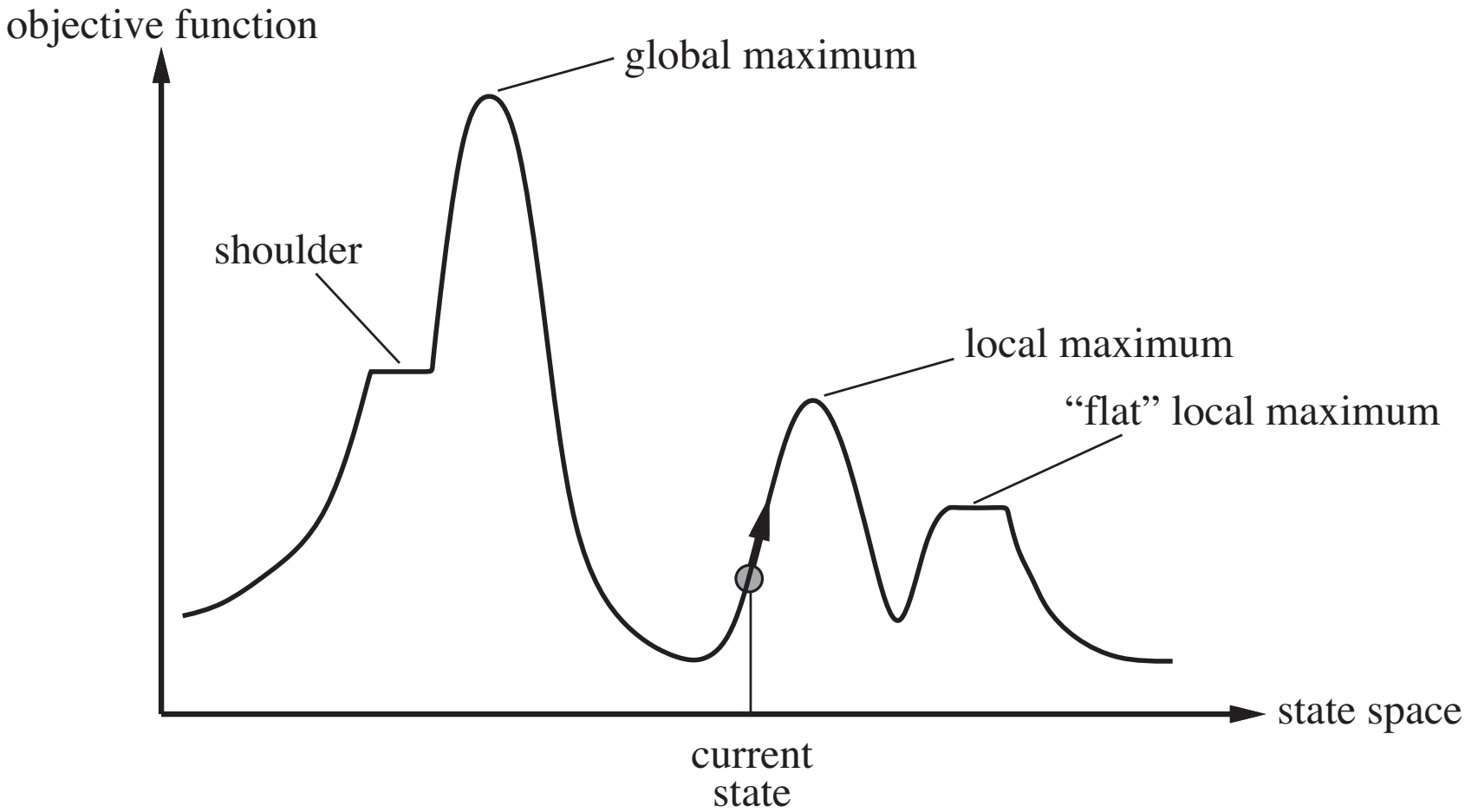
Cost function : number of pairs of attacking queens.



Local minimum with cost function equal to 1. It takes just five steps to reach this state from the previous one. Not bad !

Hill-climbing : the trouble with plateaus

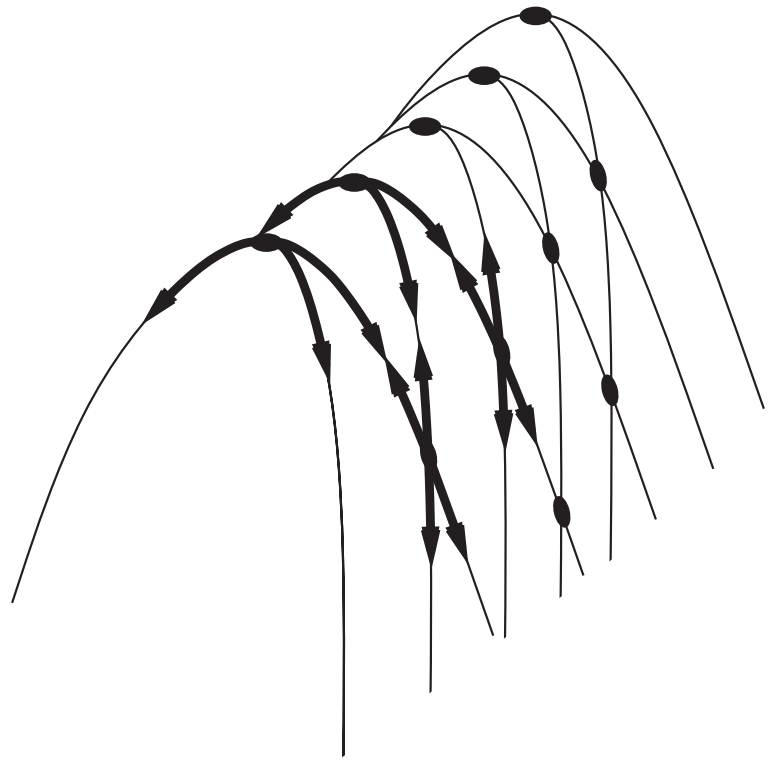
Plateau : a flat area of the state-space landscape.



Hill-climbing gets stuck in plateaus.

Hill-climbing : the trouble with ridges

Ridges : a sequence of local maxima that is very difficult for greedy algorithms to navigate.



Sequence of local maxima that are not directly connected to each other.

Hill-climbing : how to avoid shoulders ?

Starting from a random initial 8-queens state, hill-climbing gets stuck in a local maximum 86% of the time (with an average of 3 steps) and finds the global maximum only 14% (with an average of 4 steps).

Sideways moves

- To avoid staying stuck in a shoulder plateau, we allow hill-climbing to move to neighbor states that have the same value as the current one.
- However, infinite loops may occur. We avoid infinite loops by limiting the number of sideways moves.

By allowing up to 100 sideways moves in the 8-queens problem,

- We increase the percentage of problem instances solved by hill climbing from 14% to 94%.
- However, each successful instance is solved in 21 steps (on average) and failures (local maxima) are reached after 64 steps (on average).

Variants of hill-climbing

Stochastic hill-climbing

- Chooses randomly among the uphill moves, with probabilities that vary with the steepness of the moves.
- Converges slowly, but explores better the state space, and may find better solutions.

First-choice hill-climbing

- Generates successors randomly until finding one that is better than the current state.
- Efficient in high dimensions.

Random-restart hill-climbing

- Repeats the hill-climbing from a randomly generated initial state, until a solution is found.
- Guaranteed to eventually find a solution if the state space is finite.
- The expected number of restarts is $\frac{1}{p}$ if hill-climbing succeeds with probability p at each iteration.

Example of random-restart hill-climbing in the 8-queens problem

- Without sideways moves : $p = 0.14$, we need about 7 iterations on average to find a solution. The average number of steps is 22.
- With sideways moves : $p = 0.94$, we need about 1.06 iterations on average to find a solution. The average number of steps is 68.

Simulated annealing

- Simulated annealing is a widely used technique for solving combinatorial optimization problems (such as VLSI, factory scheduling, and structural optimization).
- Motivated by an analogy to annealing in solids :
  1. Heat the metal to a high temperature to increase the size of its crystals and reduce their defects
  2. Cool it down slowly until the particles arrange themselves in the ground state of the solid. Ground state is a minimum energy state of the solid.

Simulated annealing

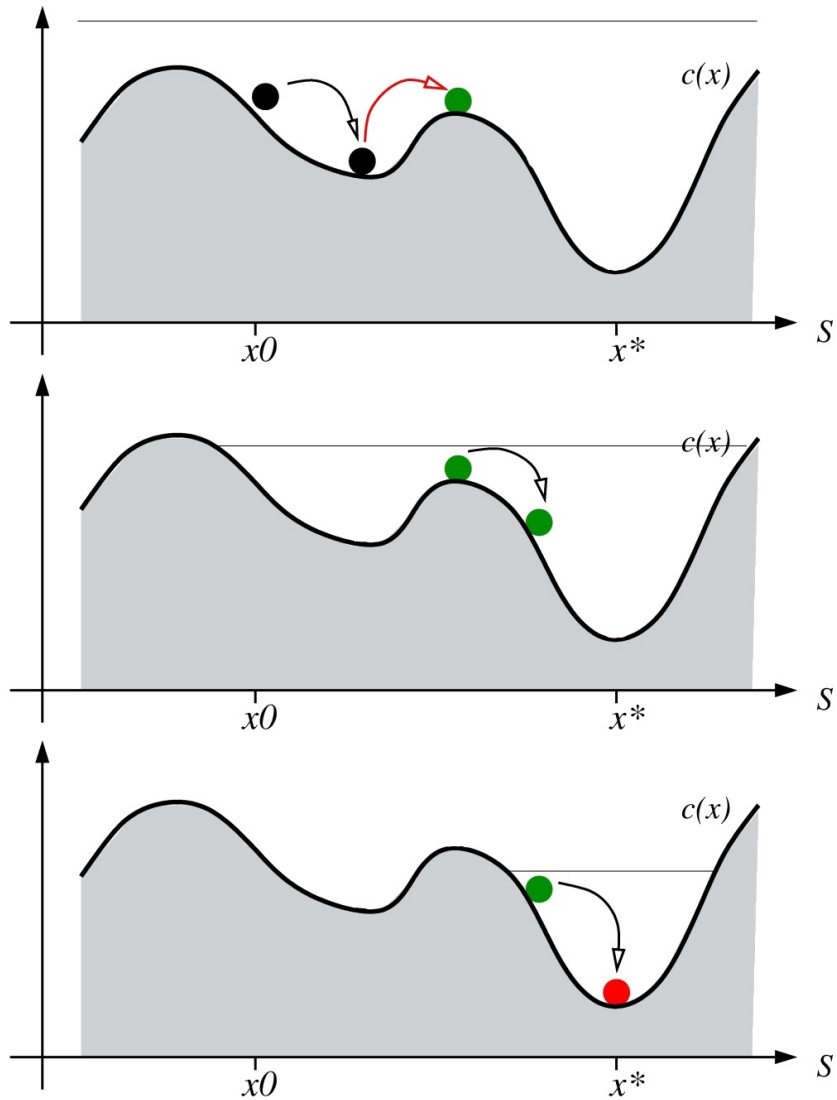
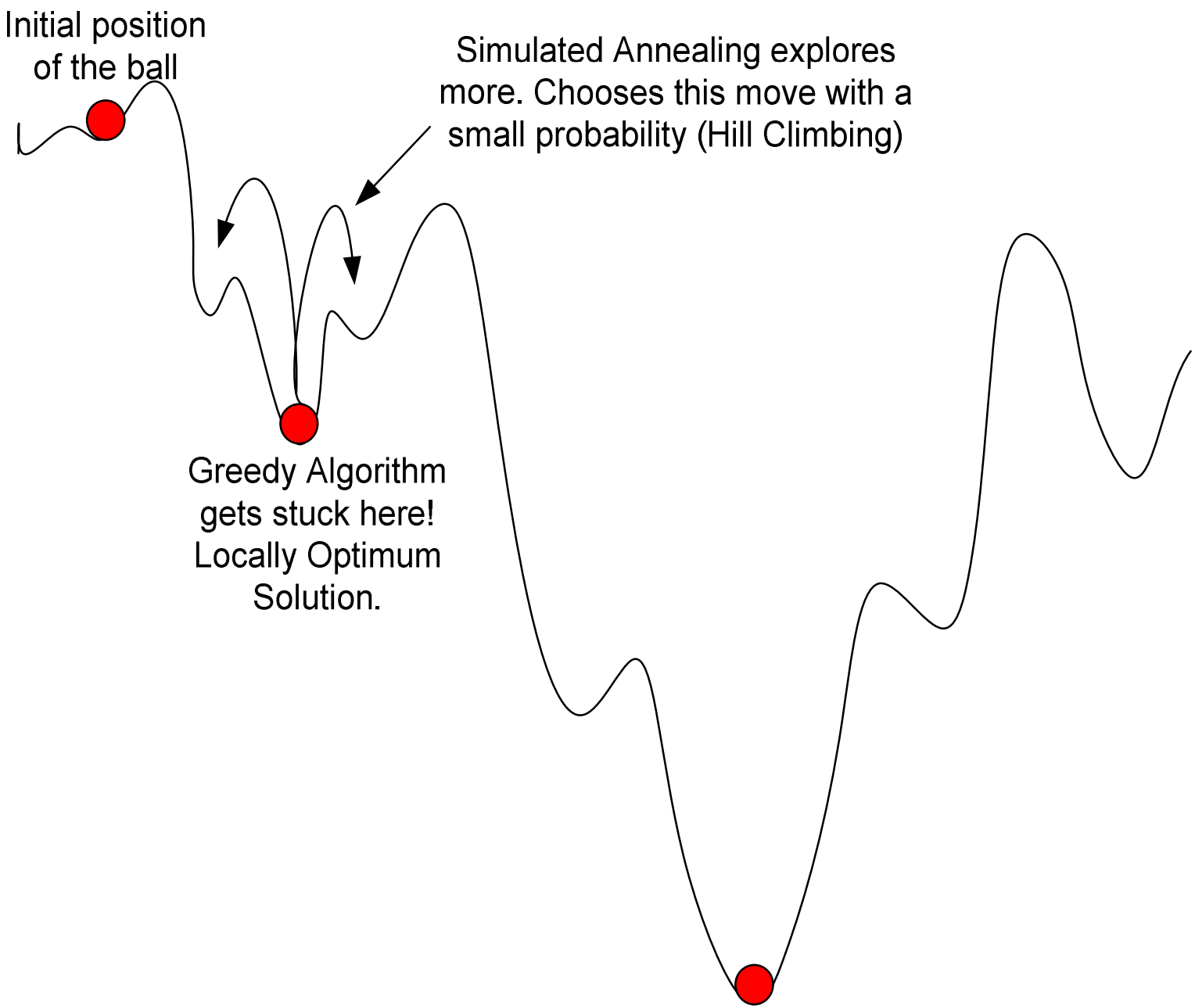


image from cmbi.ru.nl

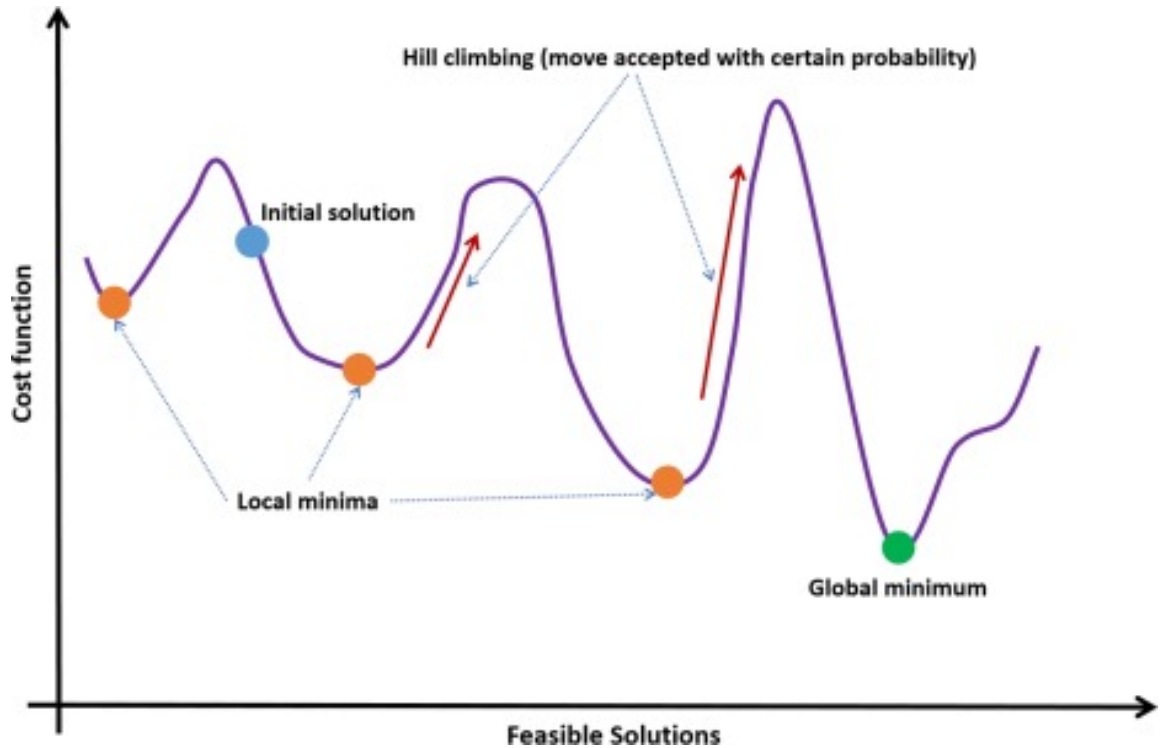
Simulated annealing



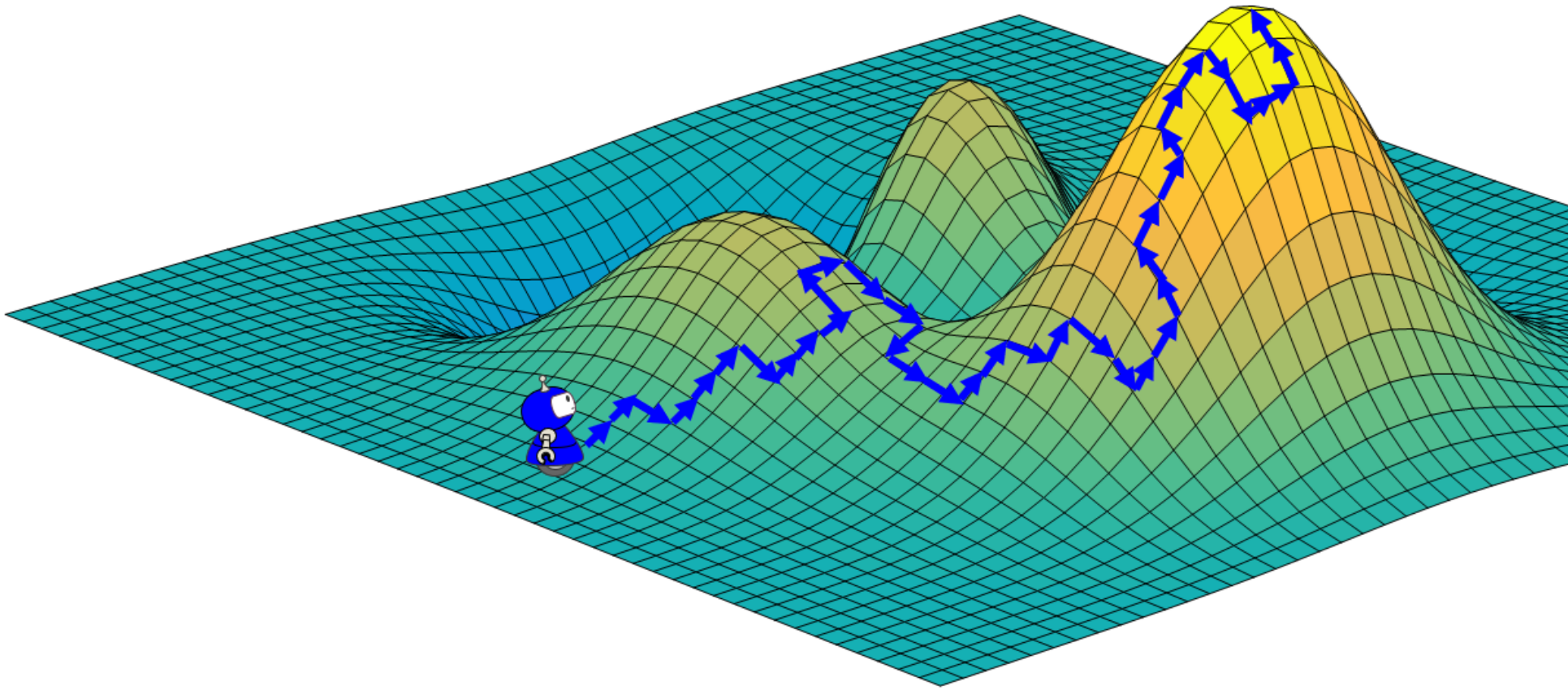
Simulated annealing

1. Step 1 : Initialize the search with a randomly chosen state, and set the temperature T to a high value.
2. Step 2 : Move in a random direction for a fixed distance.
3. Step 3 : Calculate the value of the objective function in the new state and compare it to the value in the old state.
4. Step 4 : If the value has increased, then stay in the new state. Else, stay in the new state with a probability that is proportional to the change in value, otherwise move back to the old state.
5. Step 5 : Decrease the temperature and go back to step 2.

Simulated annealing



Simulated annealing



Simulated annealing

function SIMULATED-ANNEALING($problem$, $schedule$) **returns** a solution state
  inputs: $problem$, a problem
    $schedule$, a mapping from time to "temperature"

  $current \leftarrow$ MAKE-NODE($problem$.INITIAL-STATE)
  for $t = 1$ **to** ∞ **do**
    $T \leftarrow schedule(t)$
    if $T = 0$ **then return** $current$
    $next \leftarrow$ a randomly selected successor of $current$
    $\Delta E \leftarrow next$.VALUE $-$ $current$.VALUE
    if $\Delta E > 0$ **then** $current \leftarrow next$
    else $current \leftarrow next$ only with probability $e^{\Delta E/T}$

Example of temperature function (schedule) : $T = \frac{100}{t^2}$. The key thing is that the temperature starts high and decreases over time.

Local beam search

- Start with k randomly chosen initial states.
- At each step, generate all the successors of the k states, and retain the k best ones.
- Stop when a goal state is reached or when no further local improvement is possible.

- Local beam search seems similar to running hill-climbing k times, but it is different.
- In a local beam search, the k search paths are not independent. The successors of the k states are compared with each other.
- States with many "good" successors dominate local beam searches.

Stochastic local beam search

- Local beam search suffers from a lack of diversity among the k paths. The search often ends up concentrated in a small region.
- Stochastic local beam search overcomes this problem by randomly generating the successors and keeping them with probabilities proportional to their values.

Genetic Algorithms (GA)

- Similar to local beam search, except that successors are generated by combining two parent states.
- Inspired from biological evolution by natural selection.
- A **population** is a set of **individuals**, each individual is a state represented as a string over a finite alphabet (DNA analogy).
- A **fitness** function is used to select individuals that will reproduce.
- Fitness function is a term used in GAs to describe objective functions.

Example : the 8-queens problem

1. State : the positions of 8 queens, each in a column of 8 squares. Each state is described by a sequence of $8 \times \log_2 8 = 24$ bits.
2. Fitness : number of pairs of queens that are not attacking each other (maximum $= 28$).

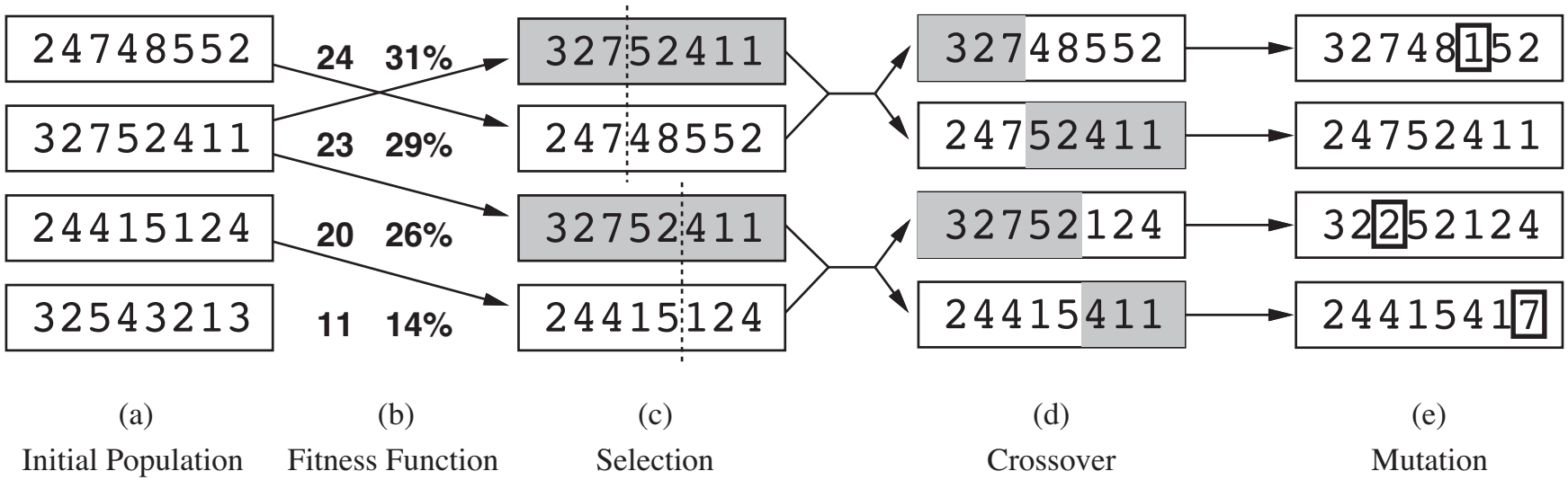
Genetic Algorithms (GA)

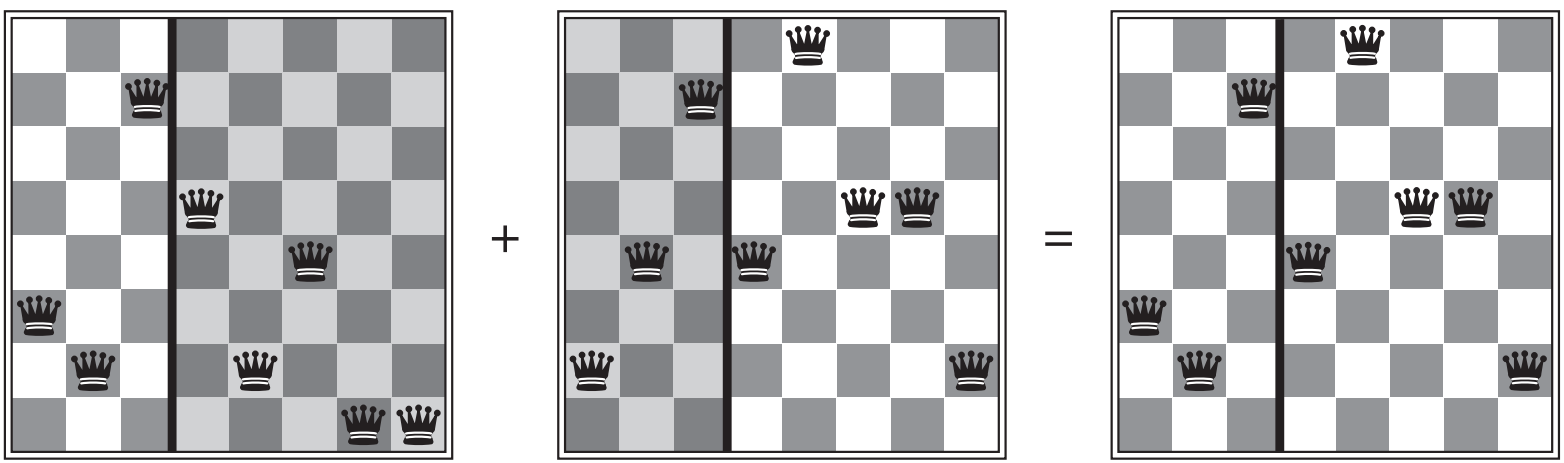
A genetic algorithm consists in the following steps :

Start with a population of k individuals

1. **Selection** : randomly draw k individuals from the population with probabilities proportional to their fitness. The same individual can be selected several times.
2. **Crossover** : let (s_1, s_2) be a selected couple of individuals, and let n be the size of the alphabet, and $s[i]$ be the i-th letter in the code of s.
   - Uniformly sample a random number $i \in [1, n]$.
   - Create new individual s'_1 that has the code $[s_1[1], s_1[2], \ldots, s_1[i], s_2[i+1], s_2[i+2], \ldots, s_2[n]]$
   - Create new individual s'_2 that has the code $[s_2[1], s_2[2], \ldots, s_2[i], s_1[i+1], s_1[i+2], \ldots, s_1[n]]$
3. **Mutation** : each letter in the codes of the new individuals is changed to a random value with probability ϵ.

Genetic Algorithms (GA)



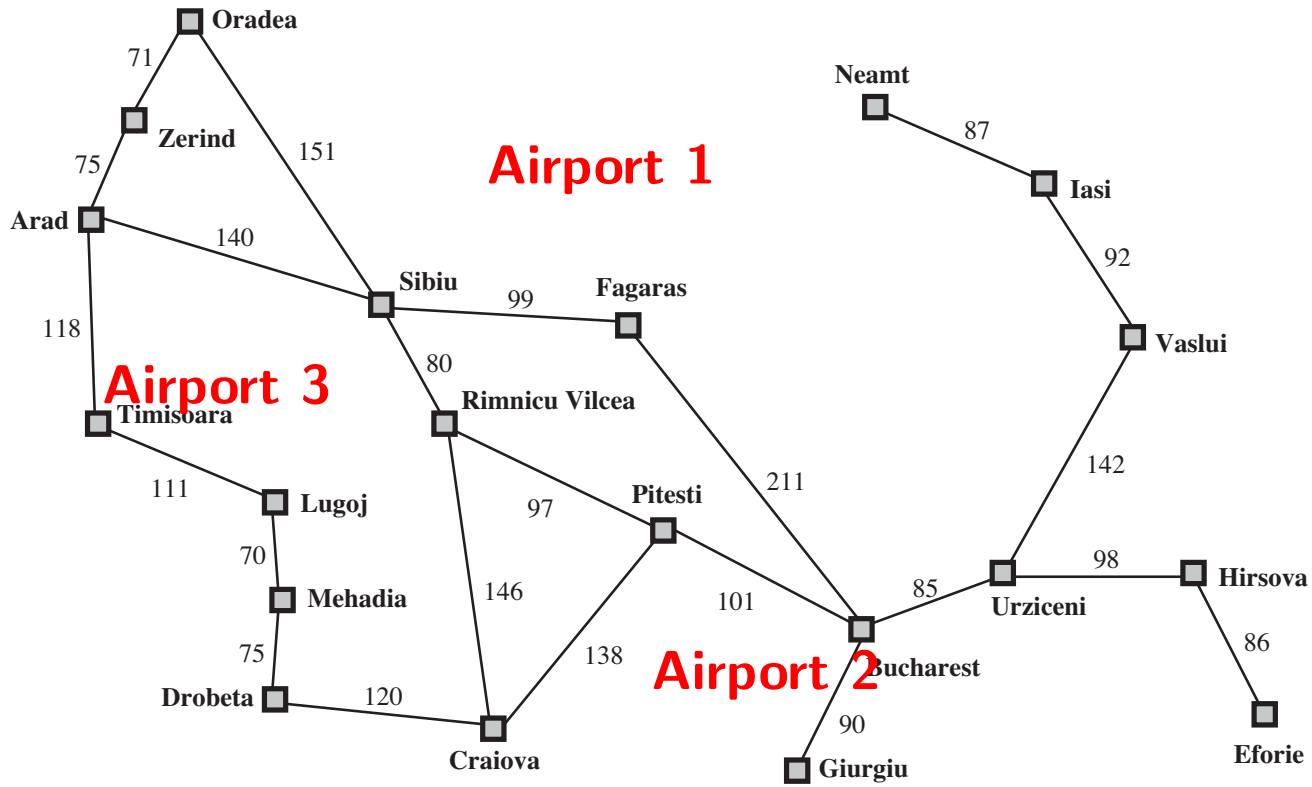


Optimization in continuous spaces

In most real-world applications, states are continuous variables.

Optimizing the locations of new airports

- Suppose we want to construct three new airports in a given country.
- Objective : minimize the distance from each city to its nearest airport.



Optimizing the locations of new airports

- Suppose we want to construct three new airports in a given country.
- Objective : minimize the distance from each city to its nearest airport.

Let C_i denote the cities that have airport i as their nearest airport. Let (x_i, y_i) be the coordinates of airport i and (x_c, y_c) be the coordinates of city c. The objective is to minimize function f

$$f(x_1, y_1, x_2, y_2, x_3, y_3) = \sum_{i=1}^{3} \sum_{c \in C_i} (x_i - x_c)^2 + (y_i - y_c)^2.$$

Gradient descent

A common approach for solving optimization problems consists in computing the gradient of the objective

$$\nabla f = (\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y_3}),$$

and applying the update rule : $x \leftarrow x - \alpha \nabla f$.
In our example,

$$\frac{\partial f}{\partial x_i} = 2 \sum_{c \in C_i} (x_i - x_c).$$

How can we choose the step-size parameter α ?

Newton's method

- In 1669, Newton proposed a method for finding a zero of a given function g that consists in iteratively applying the update rule

$$x \leftarrow x - \frac{g(x)}{g'(x)}$$

- In optimization, we search for a point where the derivative ∇f is zero. Using Newton's method, we can write the update rule as

$$x \leftarrow x - \frac{f'(x)}{f''(x)}.$$

- If f is multivariate, then we can use H_f, the Hessian matrix of second derivatives

$$x \leftarrow x - H_f^{-1}(x)\nabla f(x).$$

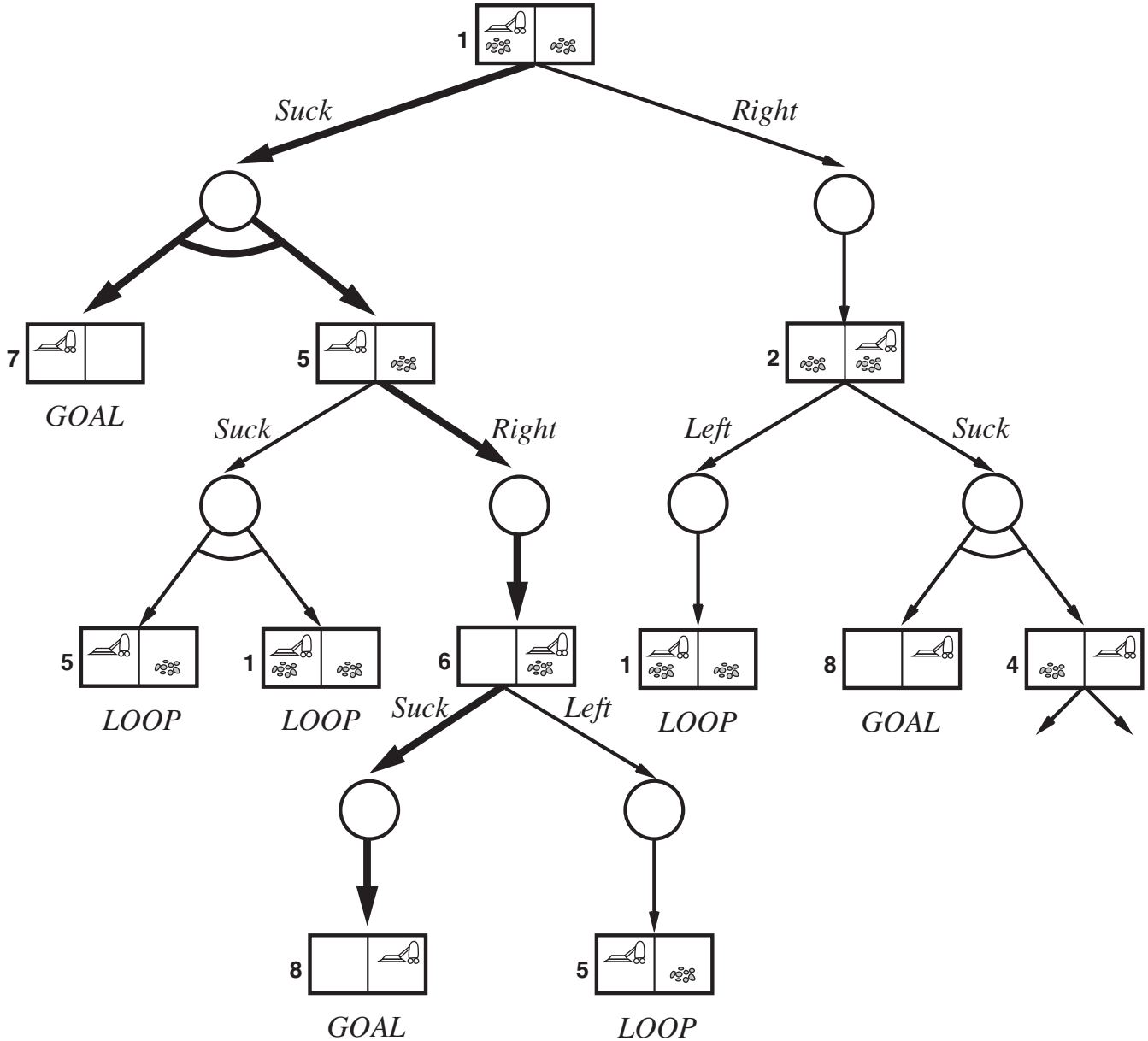
Searching with non-deterministic actions

- So far, we have assumed that the actions are deterministic.
- In the real-world, things do not always go as expected.
- To account for different possible outcomes, we need to come up with a **contingency plan** instead of a single path of actions.

Example : the erratic vacuum world

- We reconsider the vacuum world but we assume that the *Suck* action has a non-deterministic effect :
  - When applied to a dirty square, it cleans the square, but sometimes it cleans an adjacent square too.
  - When applied to a clean square, it may deposit dirt on the square.

Contingency AND-OR plan

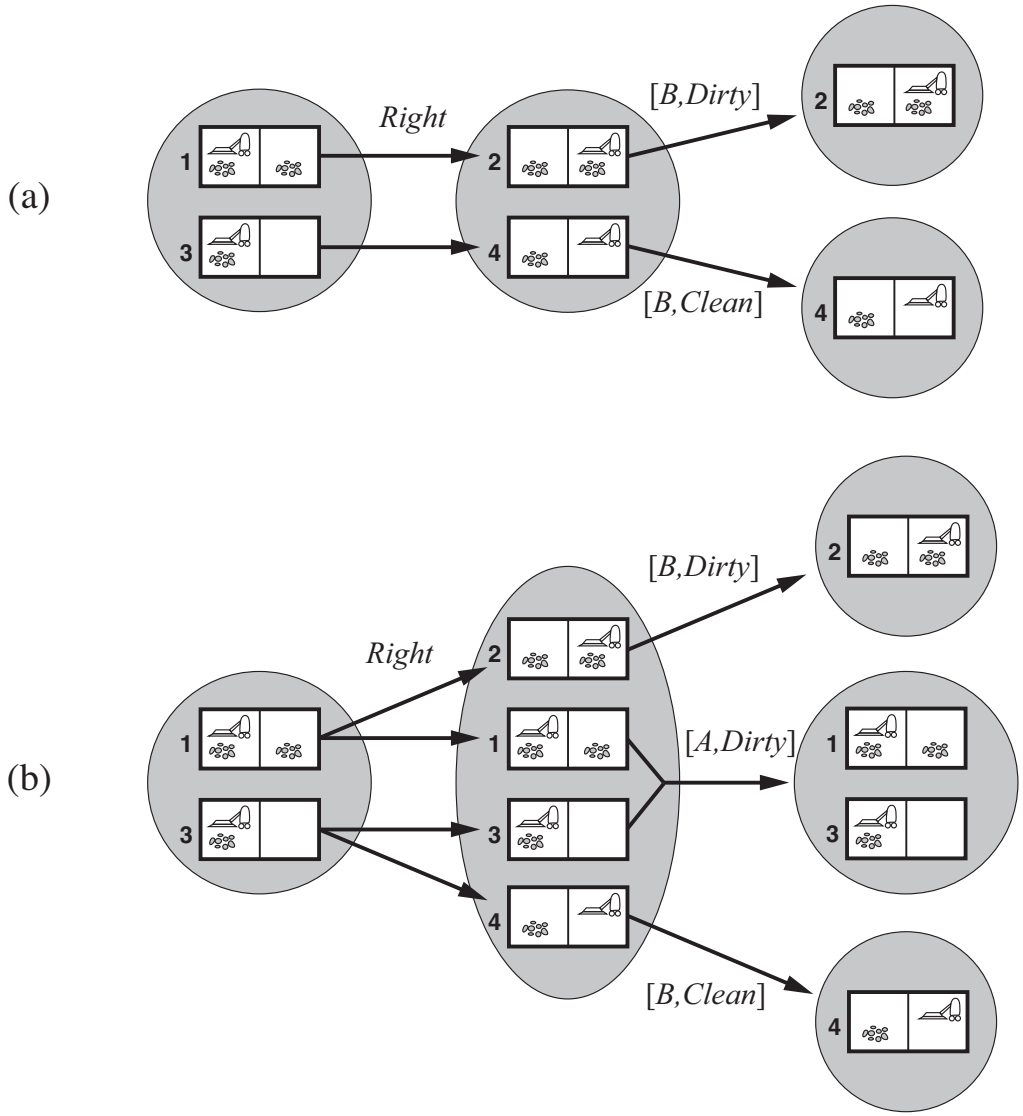


We can use the usual tree search algorithms for finding contingency AND-OR plans.

Searching with partial observations

- So far, we have assumed that the agent knows exactly the state of its environment.
- In reality, an agent receives partial, possibly noised, observations. Therefore, the state can only be estimated.
- In this case, the agent needs to remember all its history of actions and observations in order to track the state.
- The solution here is also a contingency AND-OR plan where state nodes are replaced by observations.

Example of searching with partial observations



In (a), we assume that the robot can observe only its current square.

In (b), we also assume that the floor is slippery, and deplacement actions are stochastic.

Partial observations

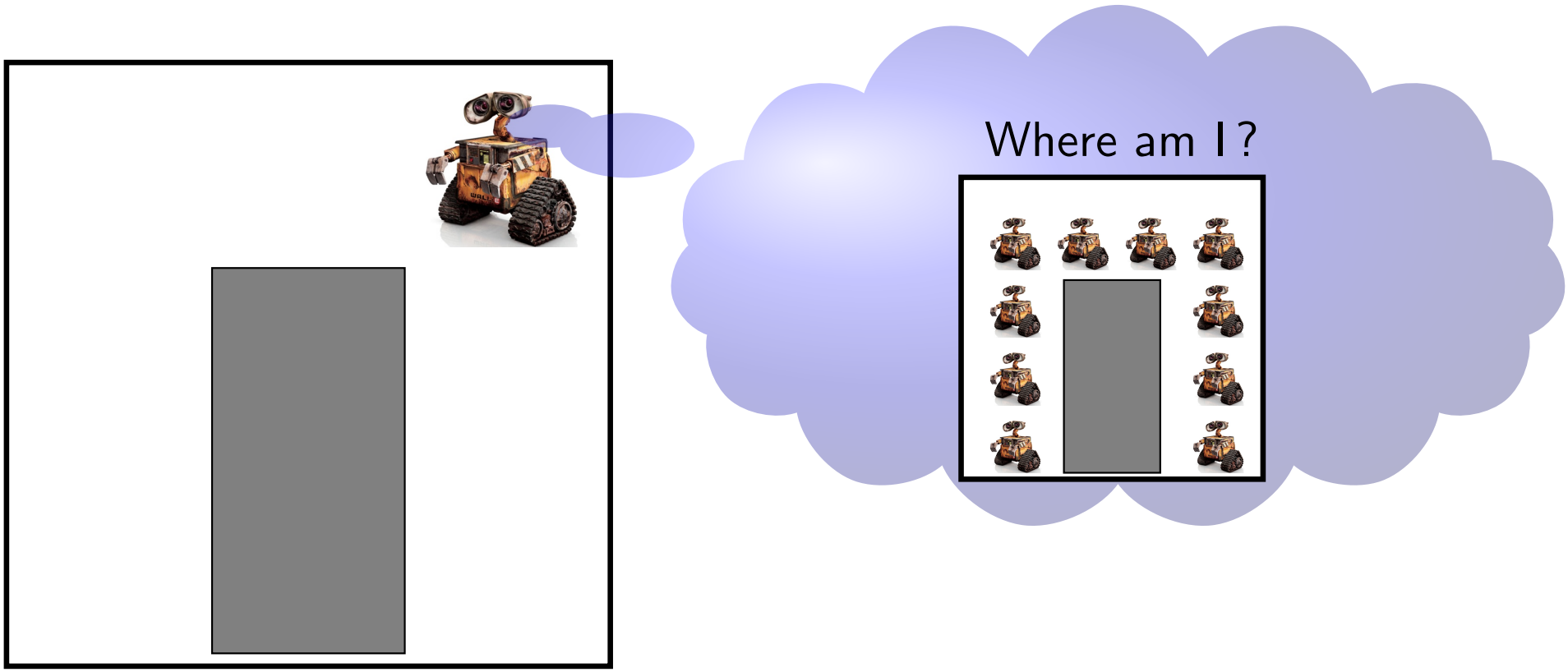


FIGURE: Initial belief state.

Partial observations

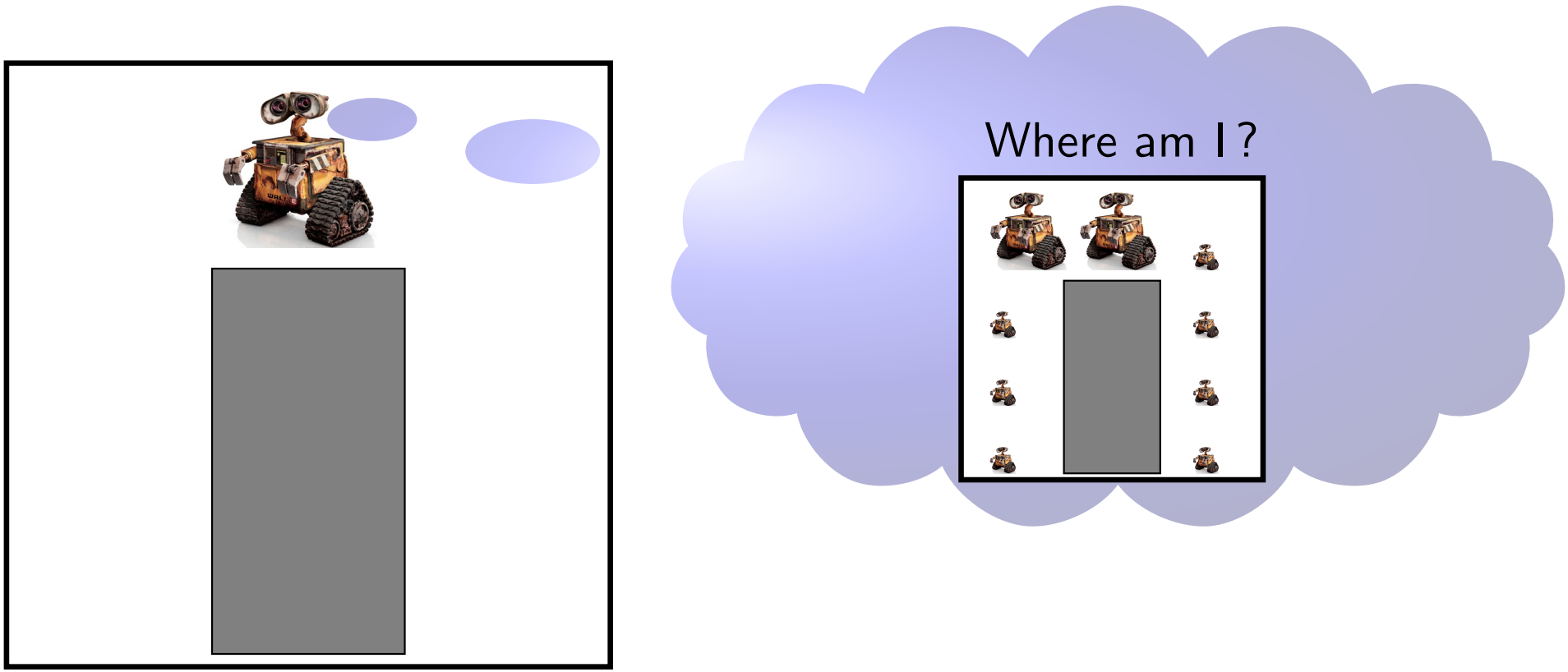


FIGURE: Belief state after moving left.

Partial observations

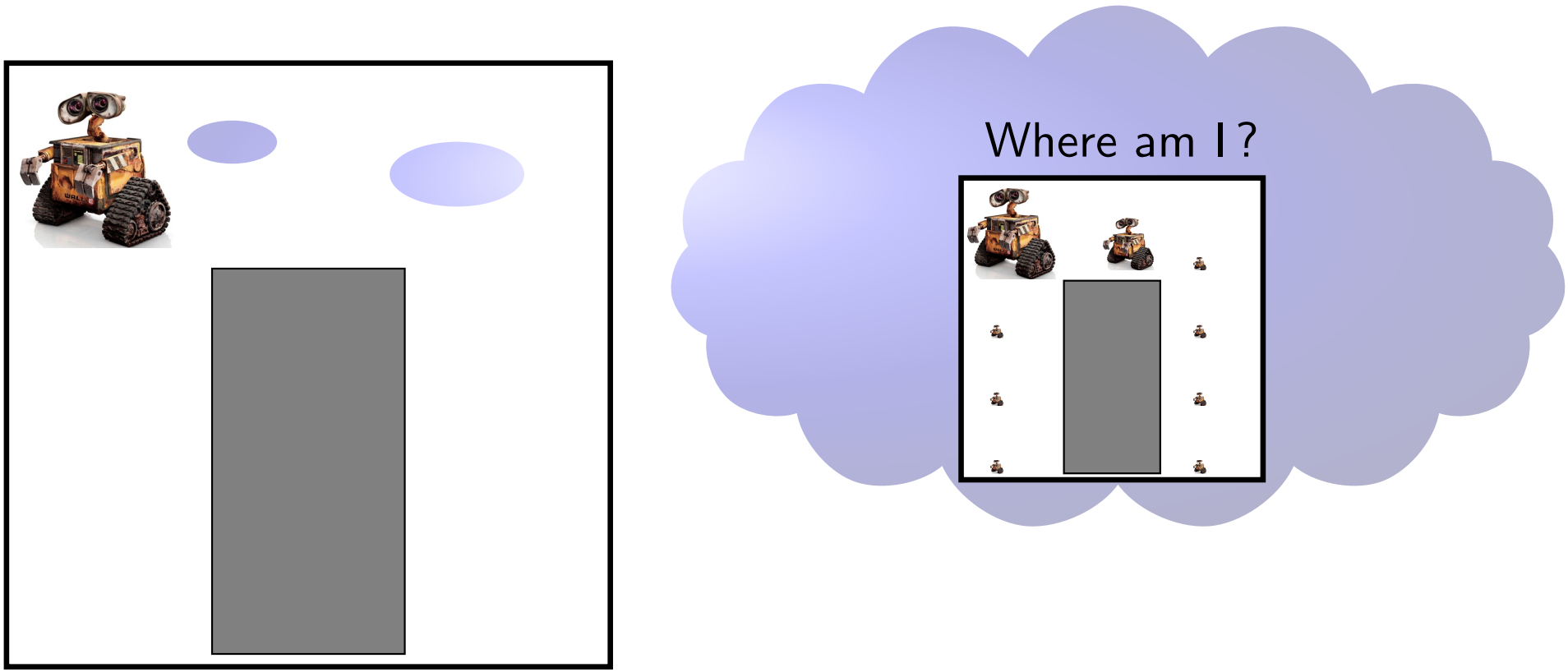


FIGURE: Belief state after moving left twice.