Chapter 4

Beyond Classical Search

CS361 Artificial Intelligence
Dr. Khaled Wassif
Spring 2023

(This is the instructor's notes, and the student must read the textbook for complete material.)

Chapter Outline

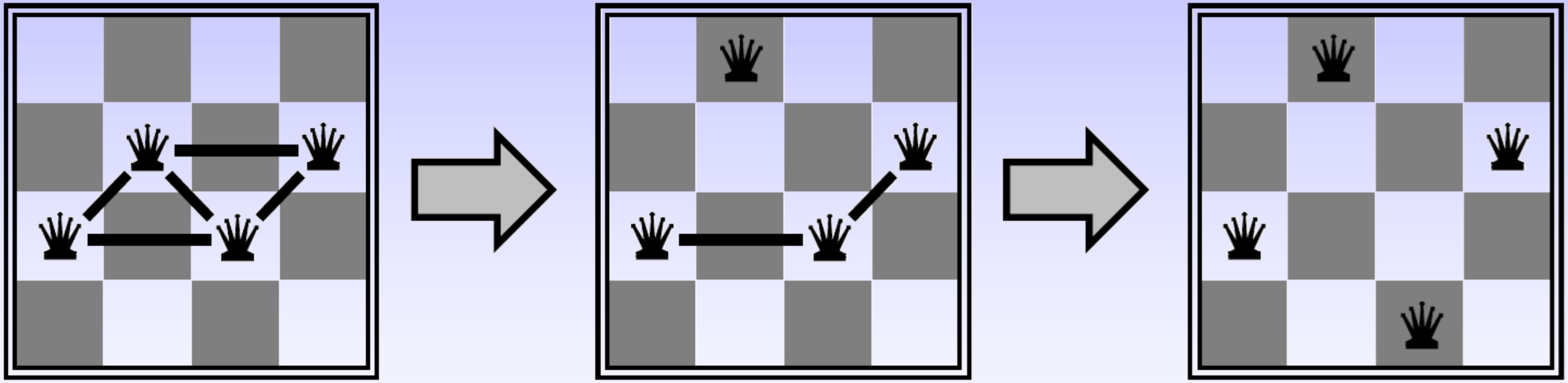
- Local Search Algorithms
  - Hill-climbing search
  - Simulated annealing search
  - Local beam search
  - Genetic algorithms (briefly)
- Searching with Nondeterministic Actions
- Searching with No Observation
- Searching with Partial Observations

Local Search Algorithms

- In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution.
  - State space is set of "complete" configurations of a problem.
  - Target is to find a configuration that satisfy constraints.
  - For example, in *n*-queens problem, what matters is the final queens configuration, not the order in which they are added.
- In such cases, we can use **local search algorithms**.
  - Keep a single "current" state and try to improve it.
- Useful for solving pure optimization problems, in which the aim is to find the best state according to an objective function.

Example: *n*-queens

- Put n queens on an $n \times n$ board with no two queens on the same row, column, or diagonal.
  – Move a queen to reduce number of conflicts

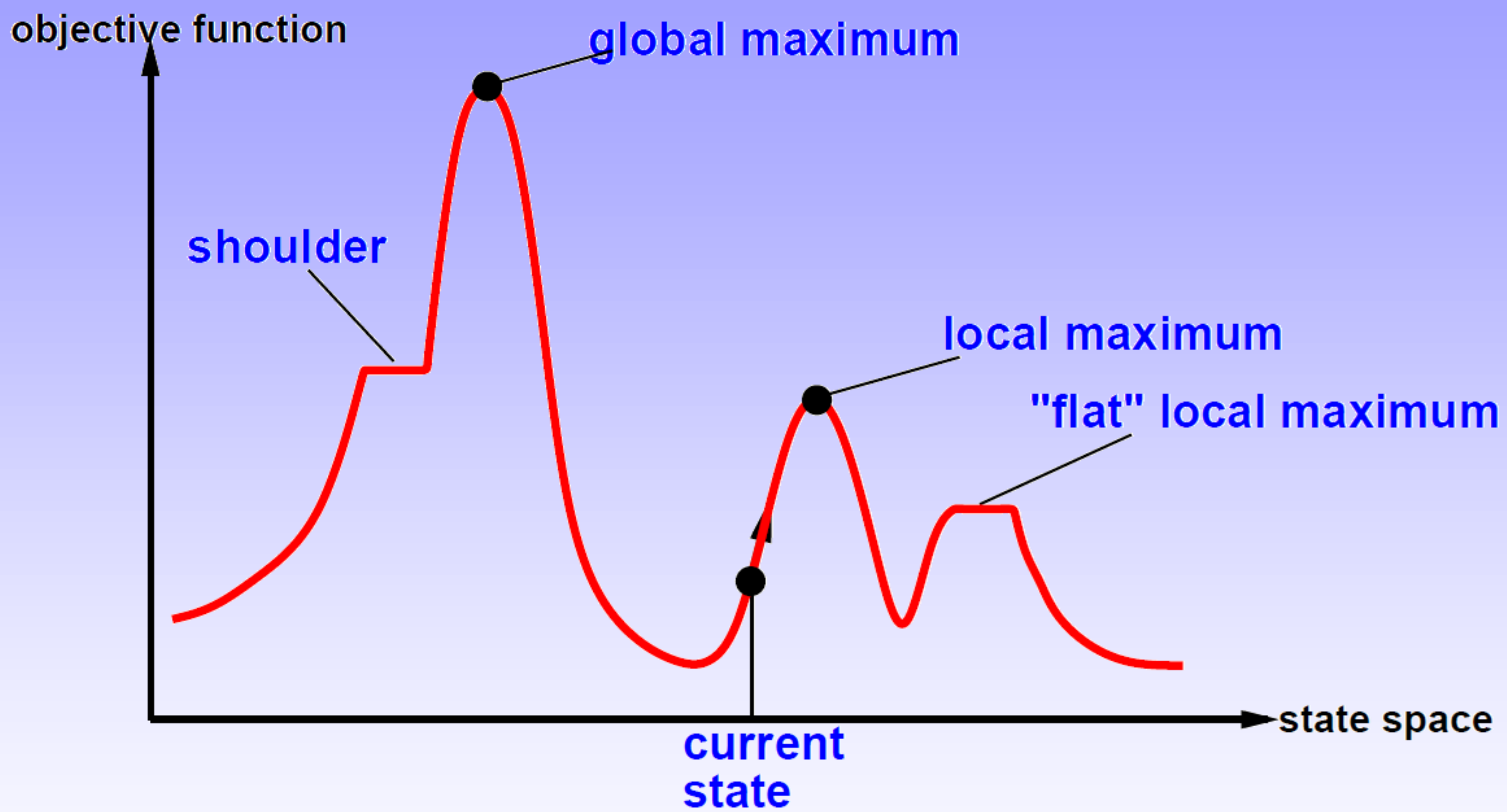


Initial state … Improve it … using local transformations

Local Search Algorithms

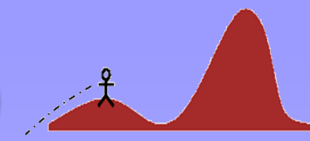
- To understand local search, it is helpful to consider the **state-space landscape** that contains the following:
  - "location" defined by the state
  - "elevation" defined by a heuristic cost or objective function
    - » If elevation corresponds to cost, then the aim is to find the lowest valley—***a global minimum***.
    - » If elevation corresponds to an objective function, then the aim is to find the highest peak—***a global maximum***.
- Local search algorithms explore this landscape.
  - A complete local search algorithm always finds a goal if one exists; an optimal algorithm always finds a global minimum/maximum.

State-space Landscape



- Problem: depending on initial state, can get stuck in local maxima.

Hill-climbing Search (HCS)

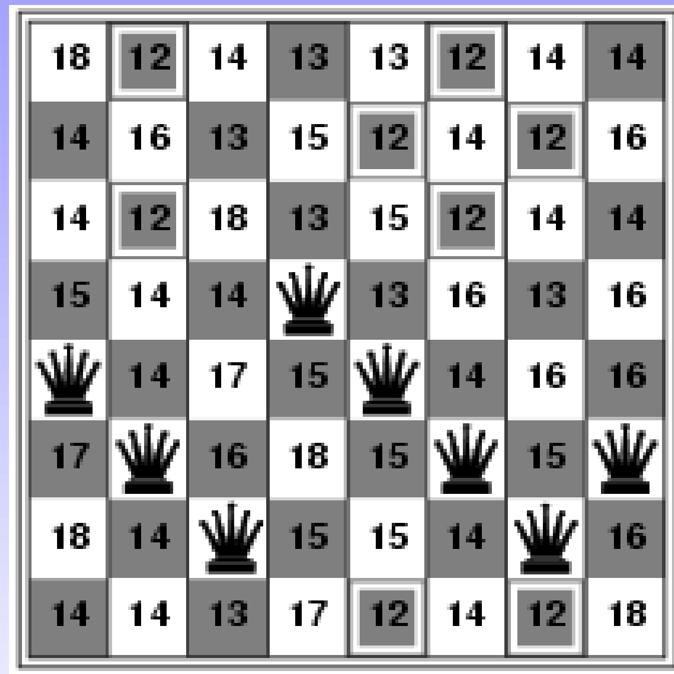


- "Like climbing Everest in thick fog with forgetfulness"

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

  current ← MAKE-NODE(problem.INITIAL-STATE)
  loop do
     neighbor ← a highest-valued successor of current
     if neighbor.VALUE ≤ current.VALUE then return current.STATE
     current ← neighbor
```

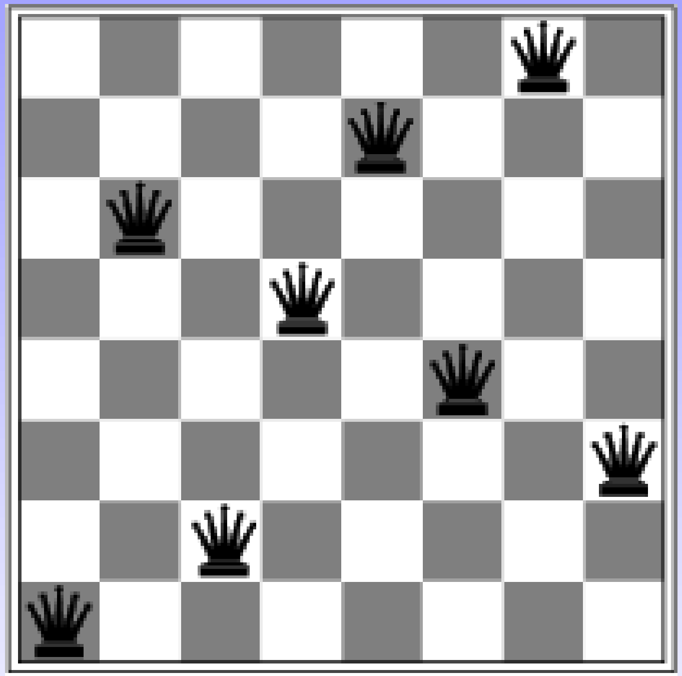
- Loop that continually moves in the direction of increasing value and terminates when it reaches a "peak" where no neighbor has a higher value.
  - Does not maintain a search tree
  - So, the current node need only record the state and the value of the objective function.

Hill-climbing Search: 8-queens Problem



- Heuristic cost function *h* = number of queen-pairs that are attacking each other, either directly or indirectly.
- *h* = 17 for the above state and best next is *h* = 12

Hill-climbing Search: 8-queens Problem

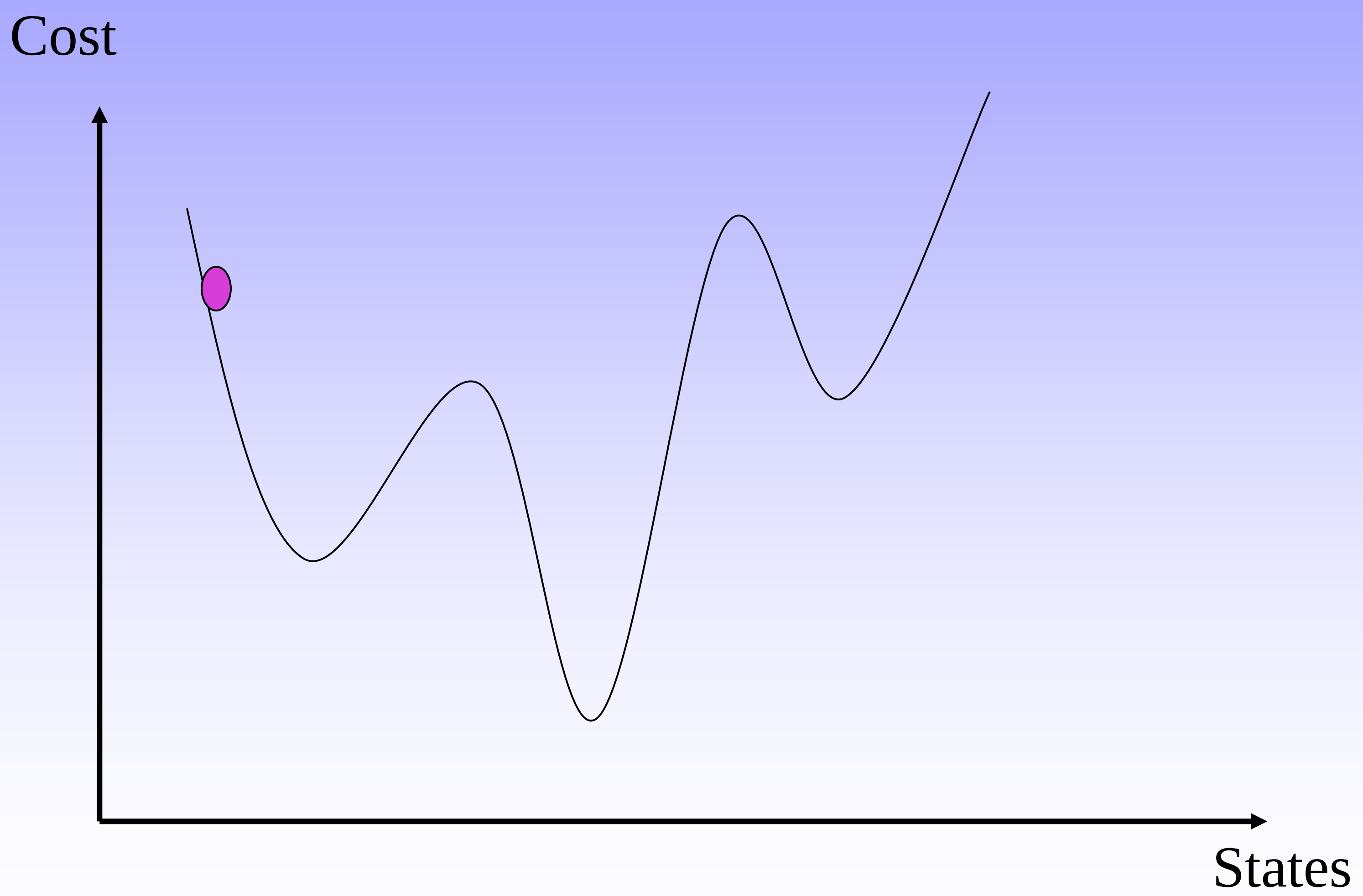


- A local minimum state with h = 1 but every successor has a higher cost.
- Therefore, hill climbing is sometimes called **greedy local search**.

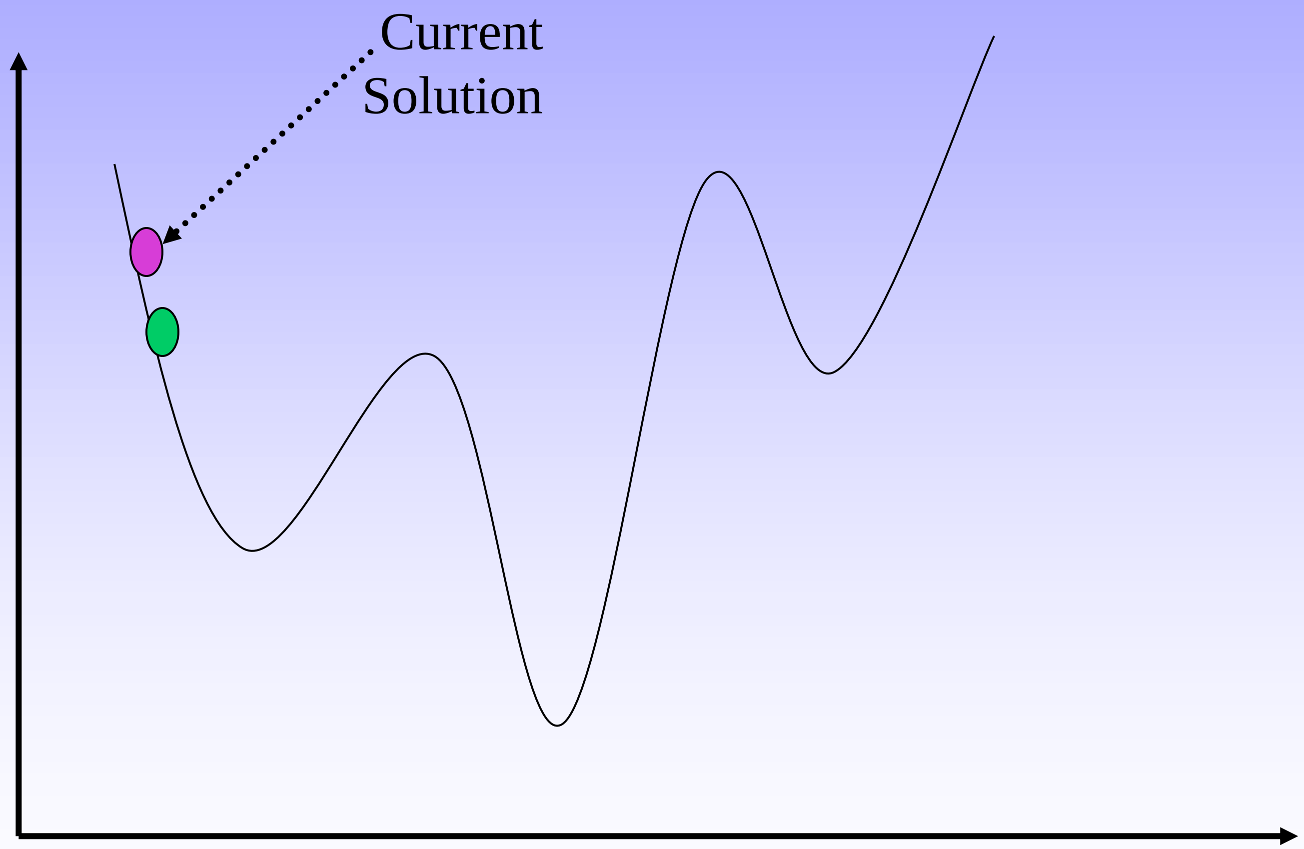
Hill-climbing Search

- HCS often makes rapid progress toward a solution because it is usually quite easy to improve a bad state.
- But, HCS frequently gets stuck when reaches a point at which no progress is being made for the following reasons:
  - **Local maxima**: a local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum.
  - **Ridges**: result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.
  - **Plateaux**: can be a flat local maximum with no uphill exit exists, or shoulder, from which progress is possible.

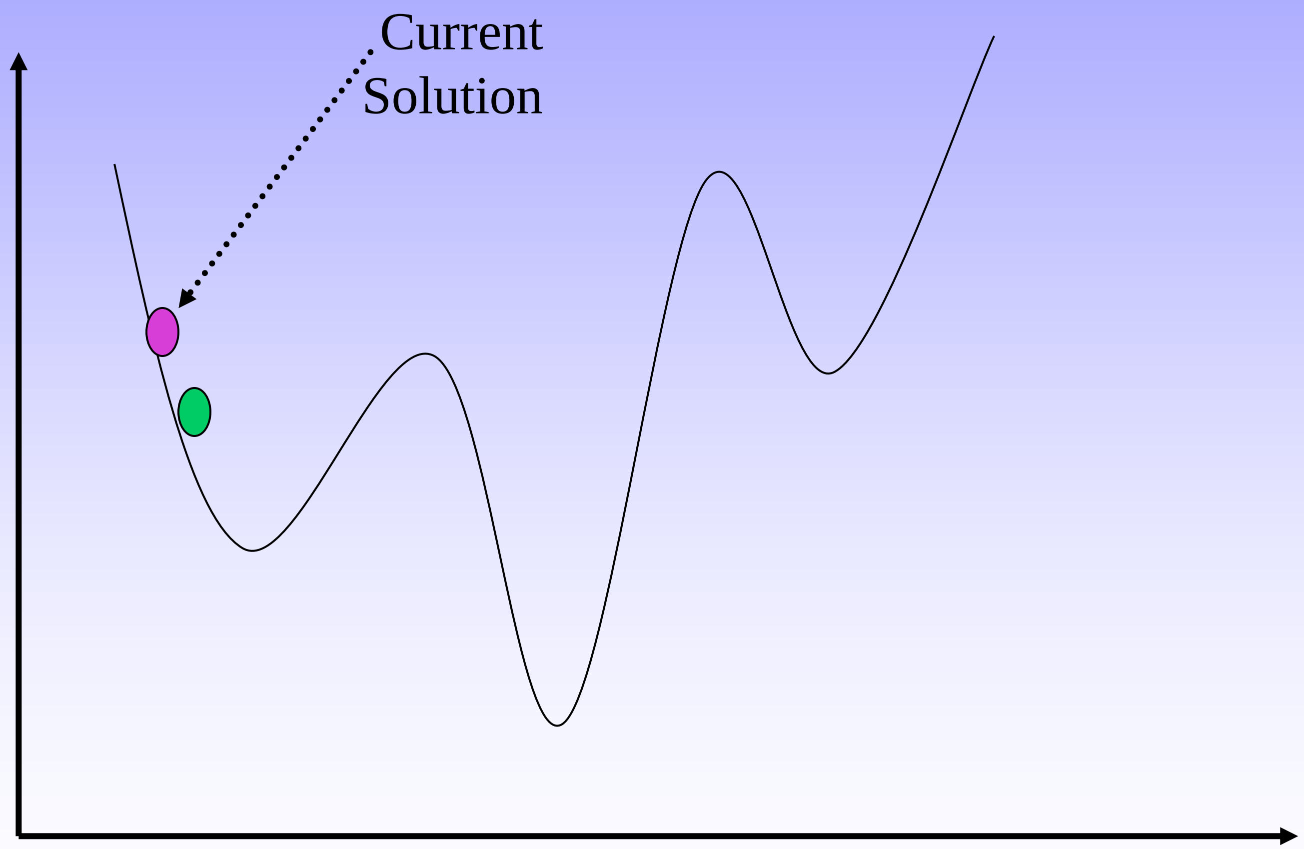
Hill-climbing Search



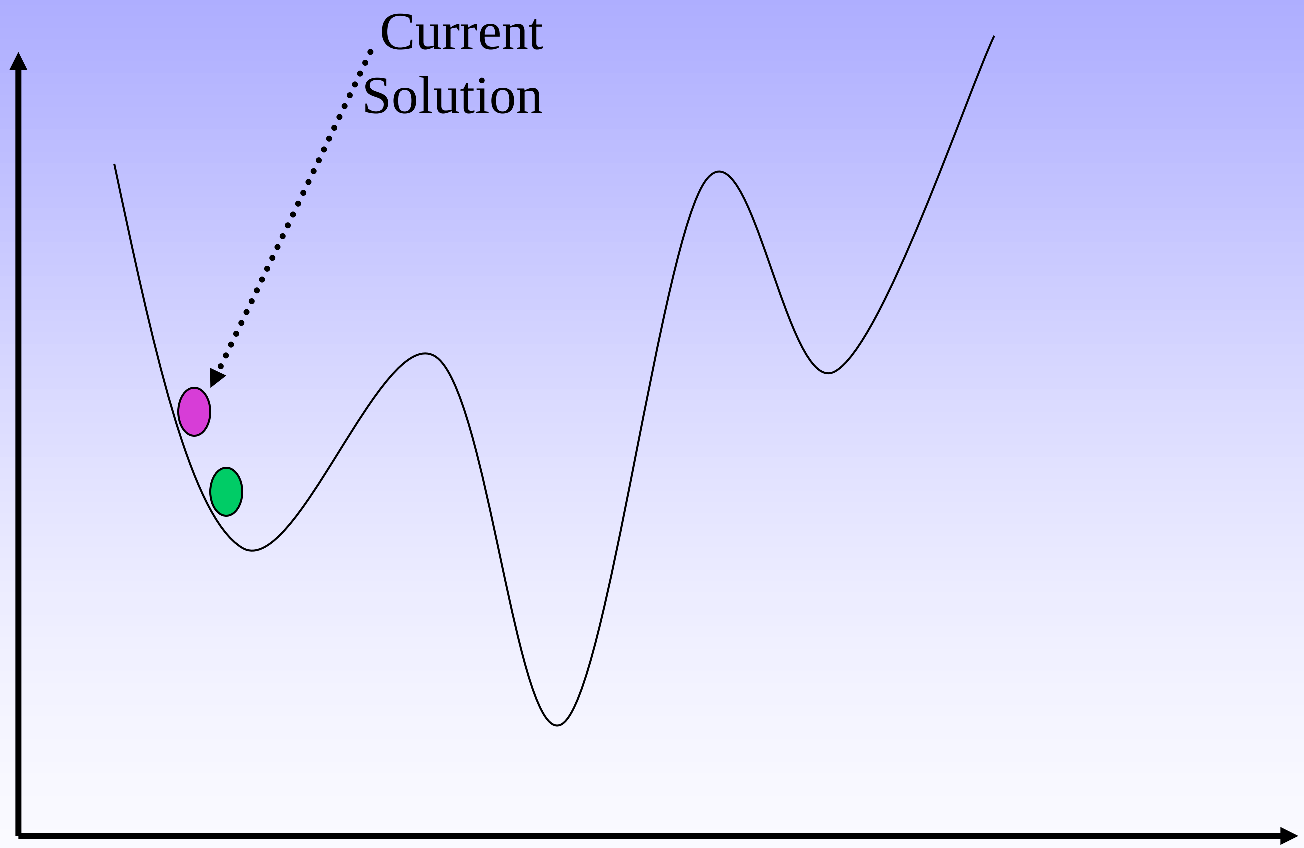
Hill-climbing Search



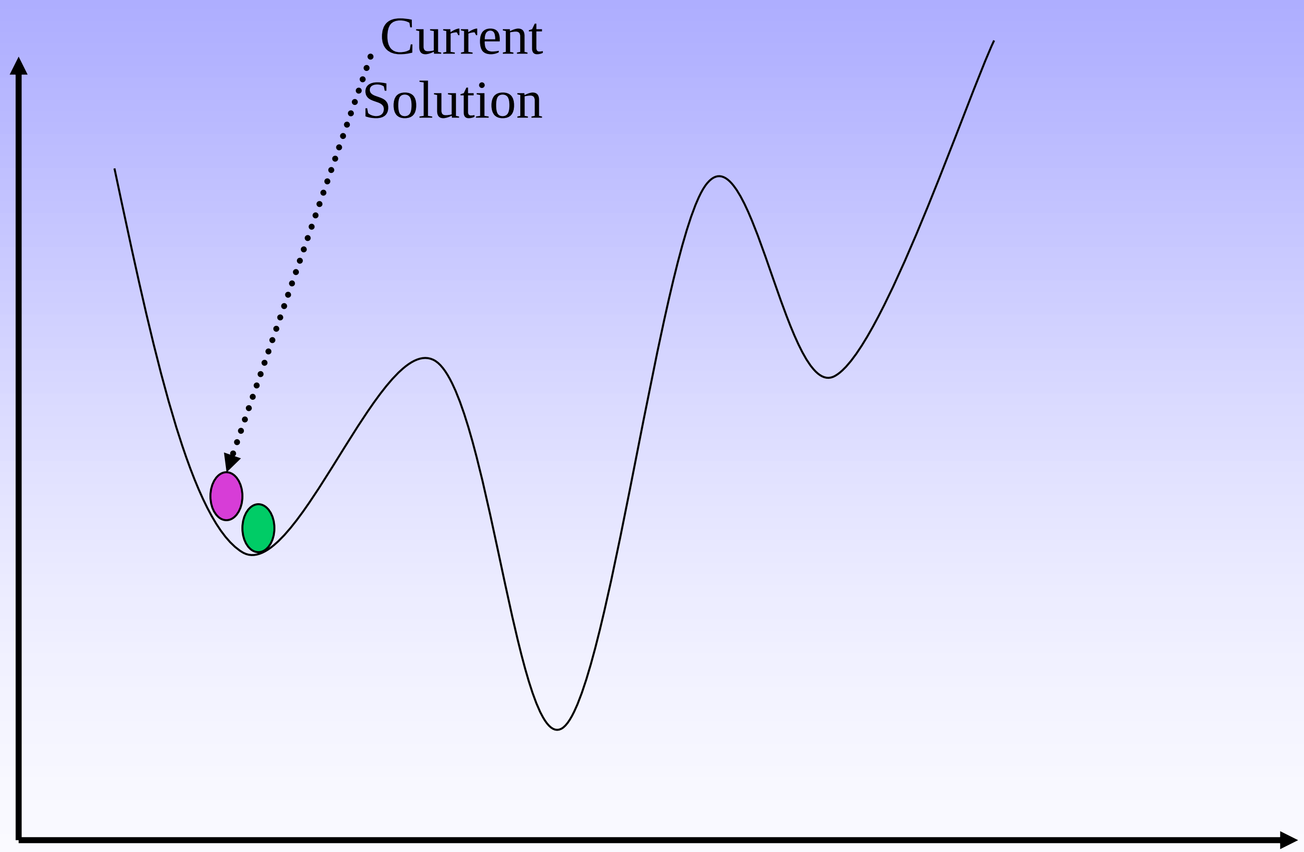
Hill-climbing Search



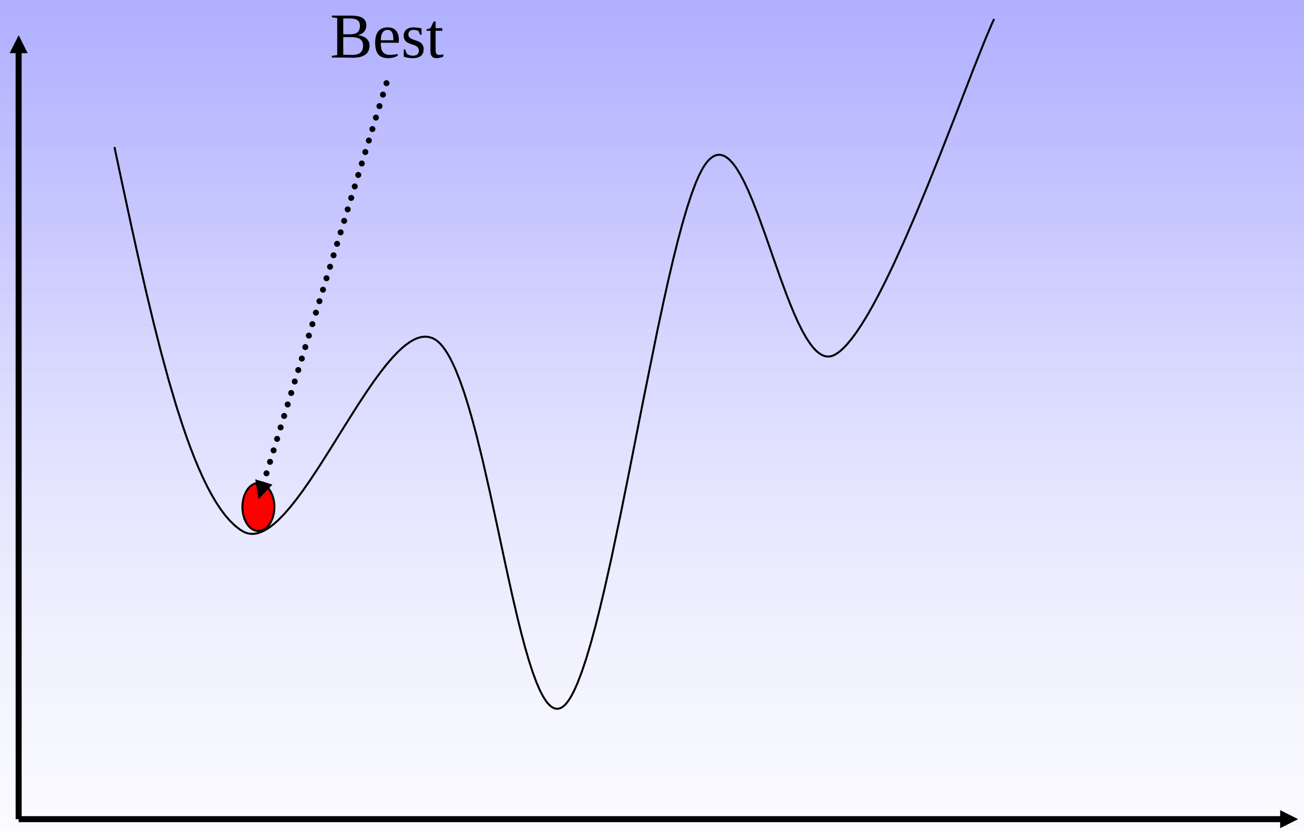
Hill-climbing Search



Hill-climbing Search



Hill-climbing Search



Simulated Annealing Search

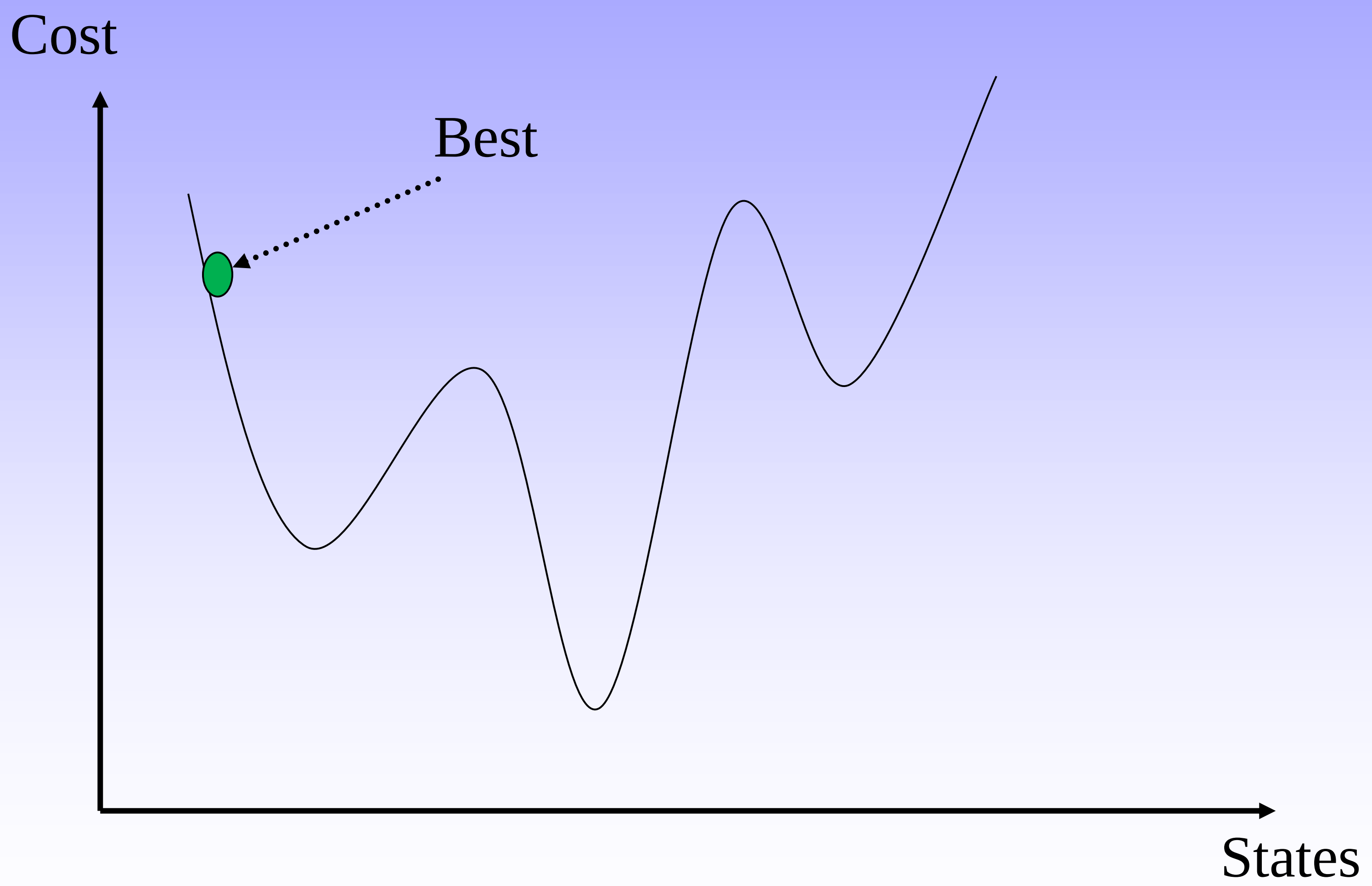
- HCS algorithm is incomplete and can get stuck on a local maximum because it *never* makes downhill moves toward states with lower value (or higher cost).
- Key idea: escape local maxima by allowing some "bad" moves but **gradually decrease** their size and frequency.
  - Instead of picking the best move, it picks a *random* move.
  - Take some downhill steps to escape the local maximum.
  - If the move improves the situation, it is always accepted.
  - Else, it accepts the move with some probability less than 1.
- Physical analogy with annealing process to harden metals:
  - Heating them to a high temp. and then gradually cooling them
  - The heuristic value is the energy, E
  - Temperature parameter, T, controls speed of convergence.

Simulated Annealing Search

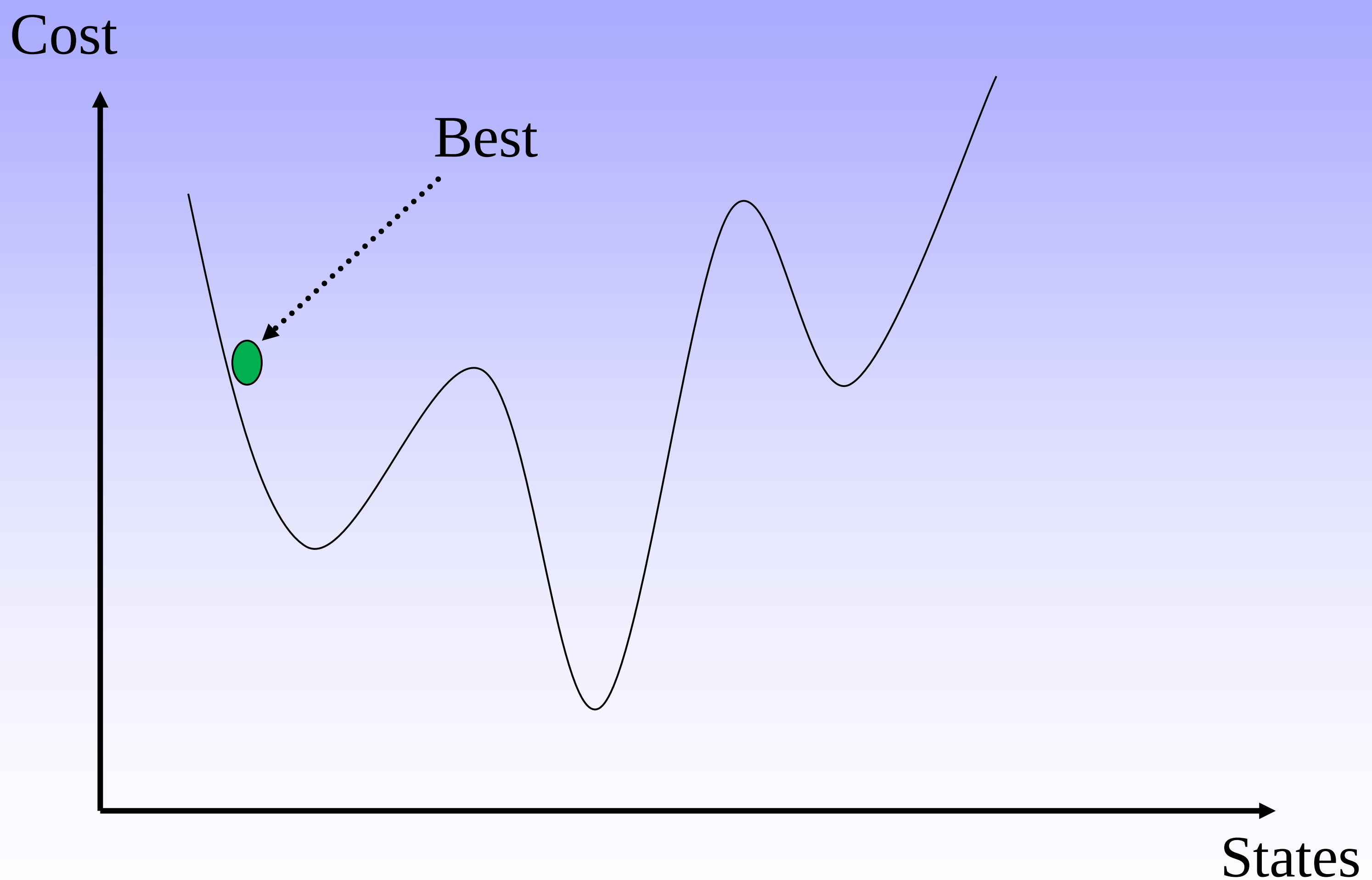
```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"

  current ← MAKE-NODE(problem.INITIAL-STATE)
  for t = 1 to ∞ do
      T ← schedule(t)
      if T = 0 then return current
      next ← a randomly selected successor of current
      ΔE ← next.VALUE – current.VALUE
      if ΔE > 0 then current ← next
      else current ← next only with probability e^(ΔE/T)
```

- Can prove: If T decreases slowly enough, then simulated annealing search will find a global optimum with probability approaching 1.
- Widely used in VLSI layout, airline scheduling, etc.

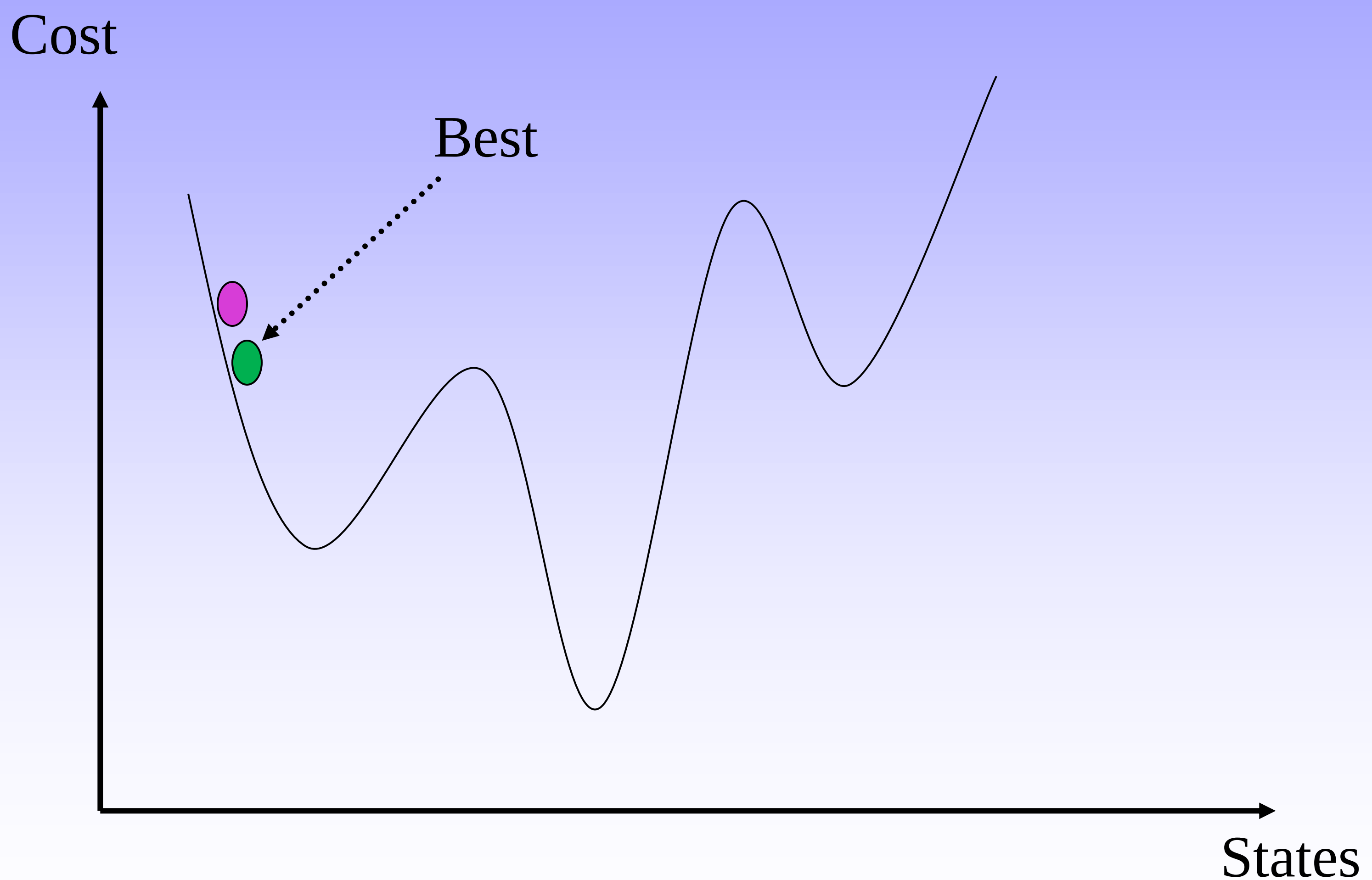
Simulated Annealing Search



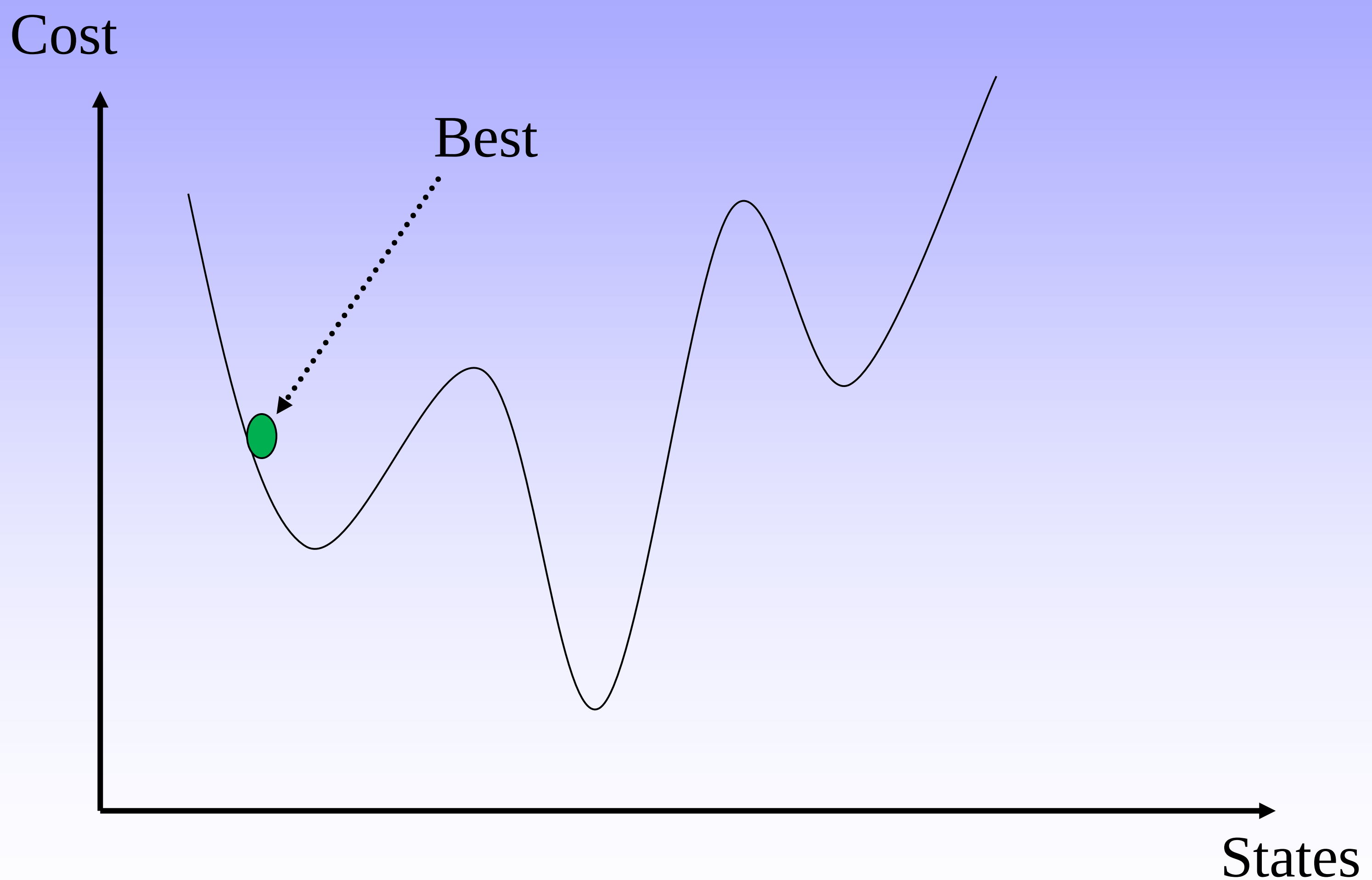
Simulated Annealing Search



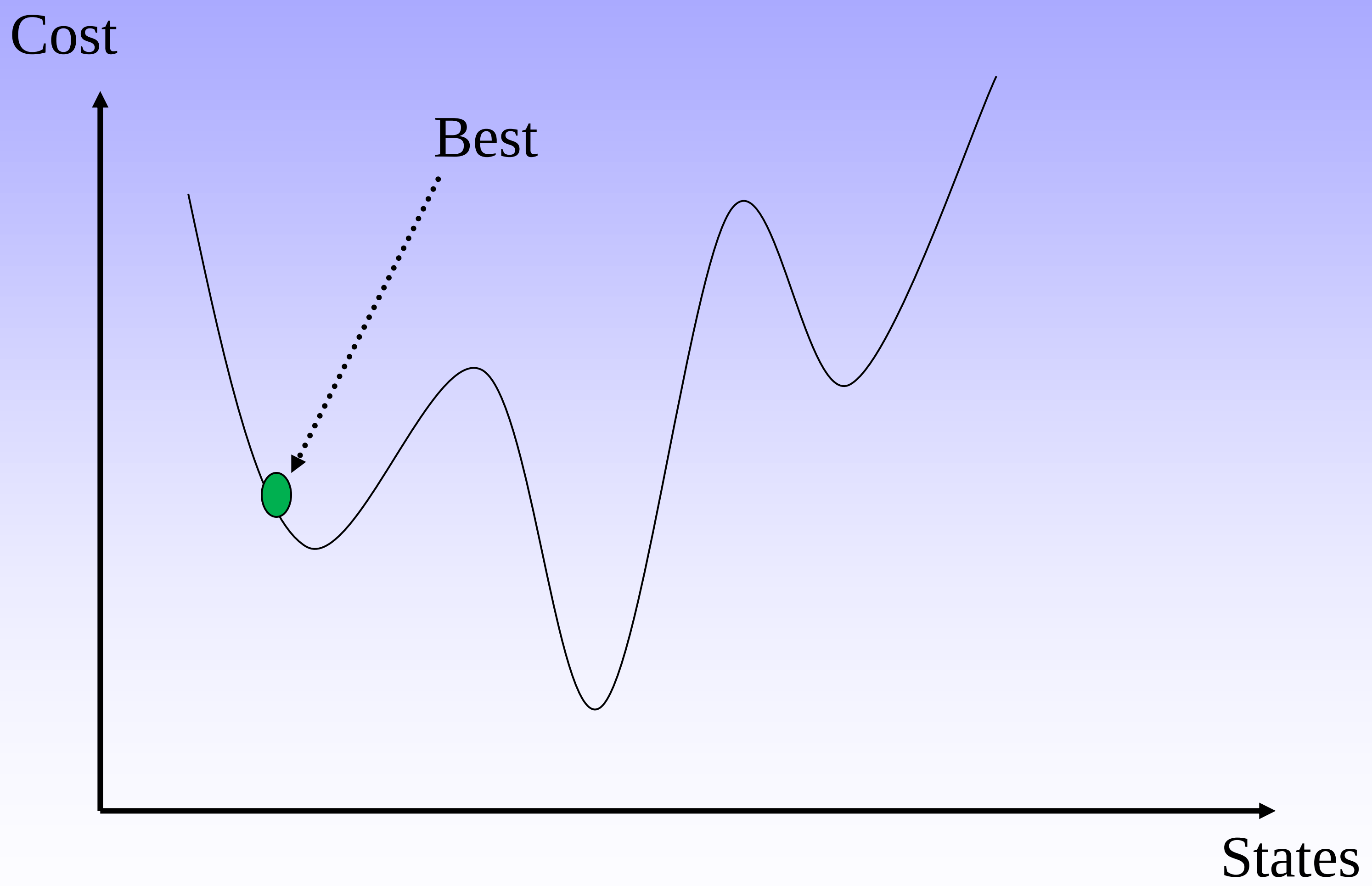
Simulated Annealing Search



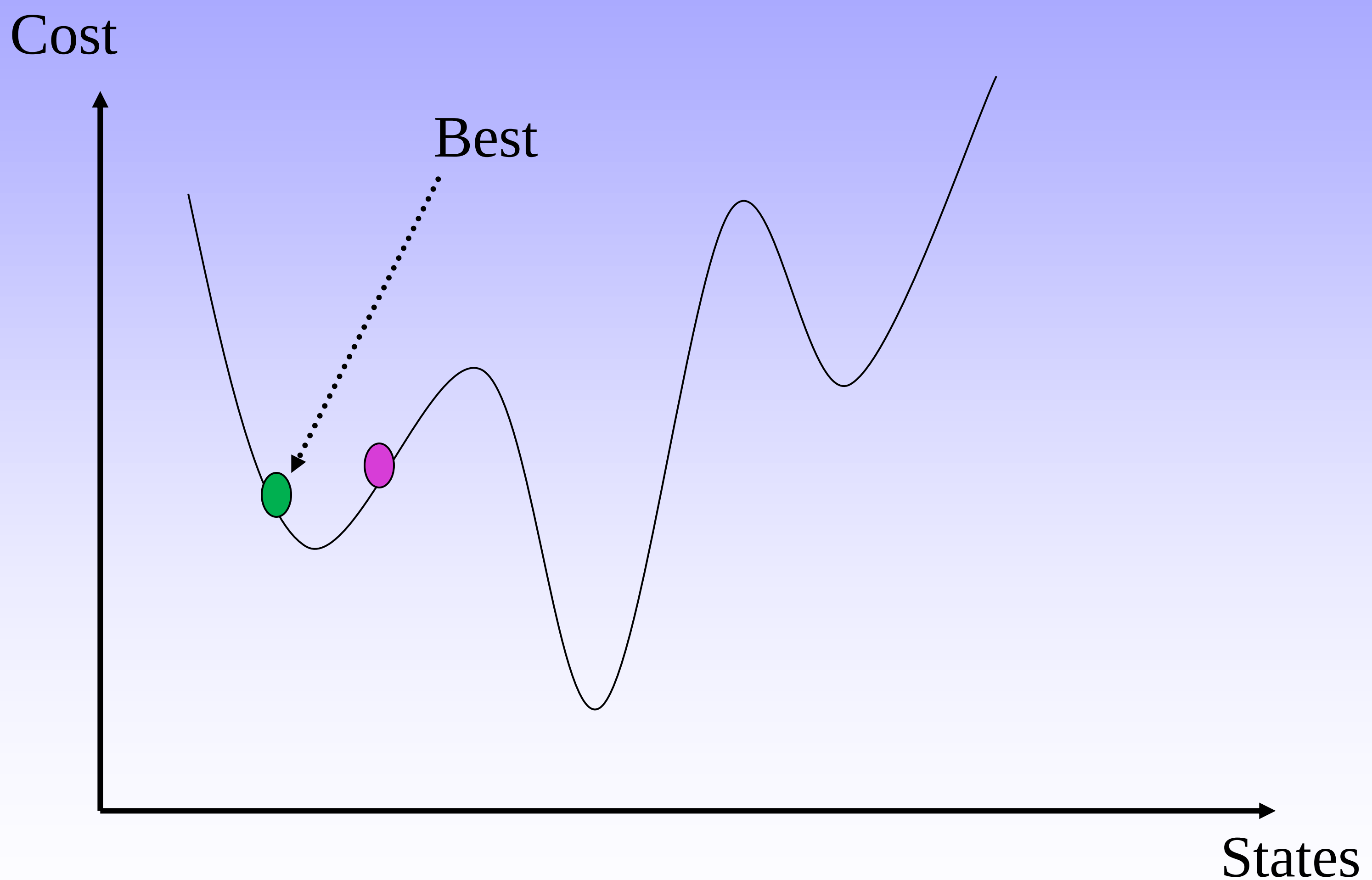
Simulated Annealing Search



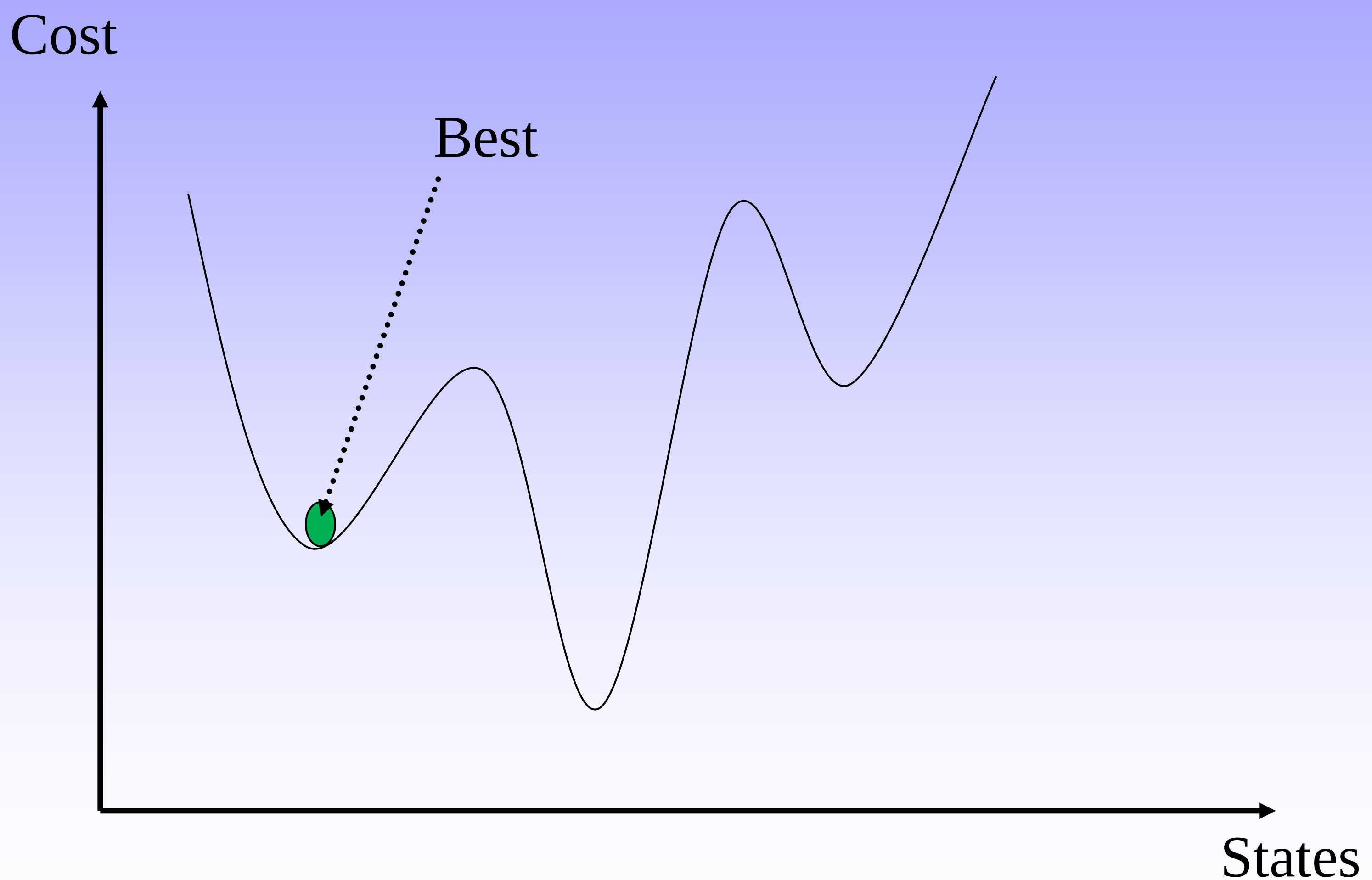
Simulated Annealing Search



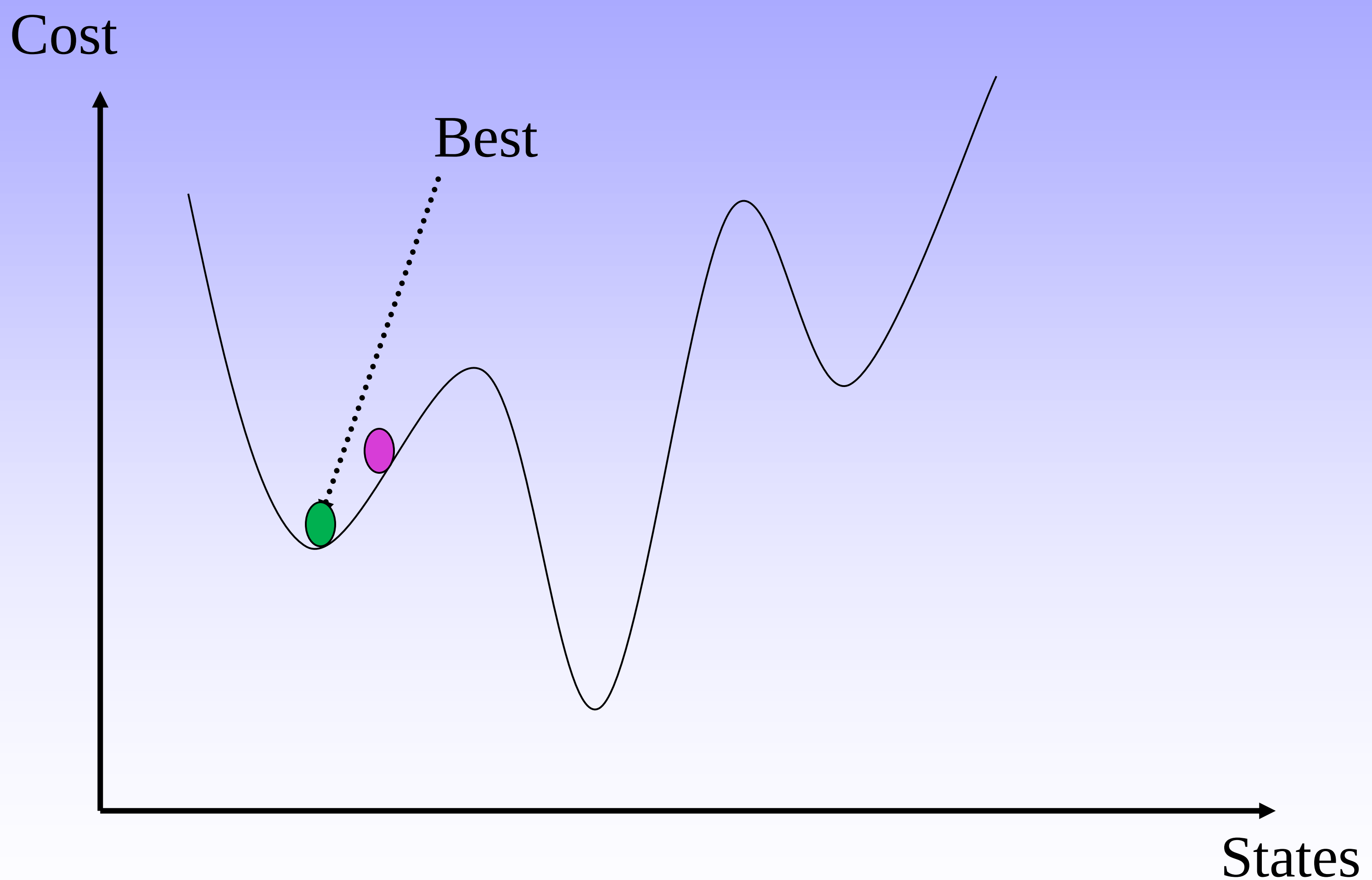
Simulated Annealing Search



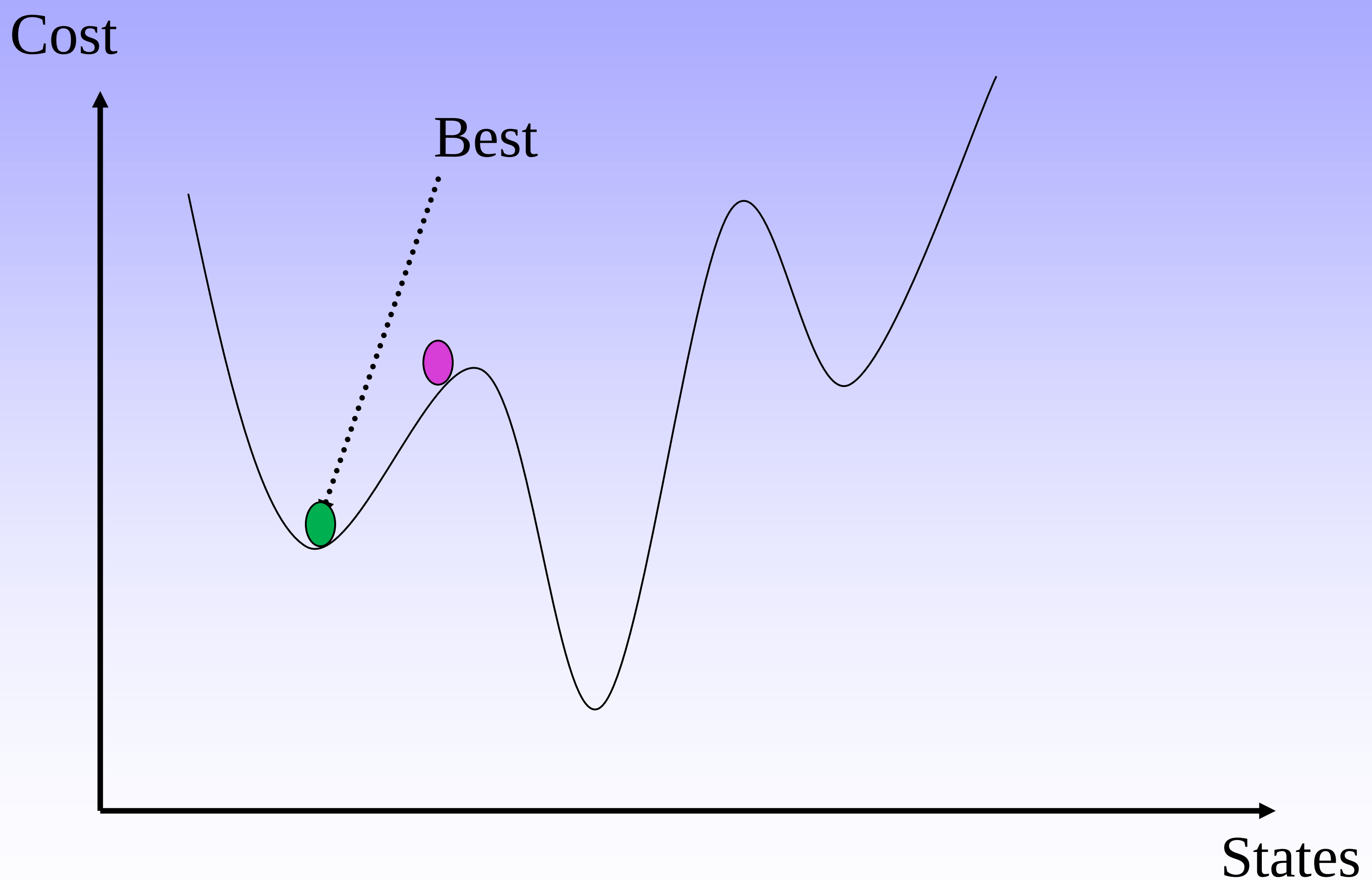
Simulated Annealing Search



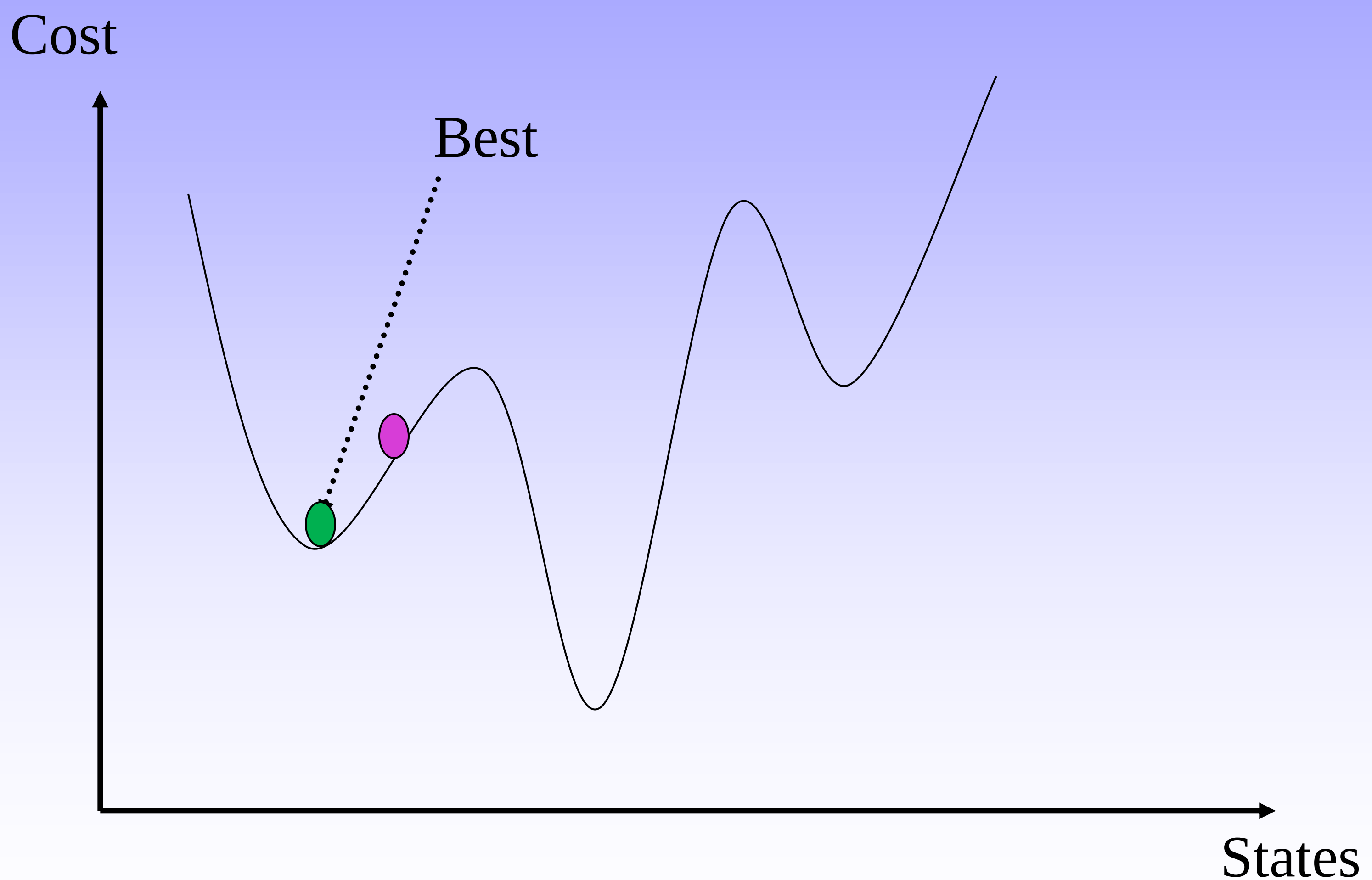
Simulated Annealing Search



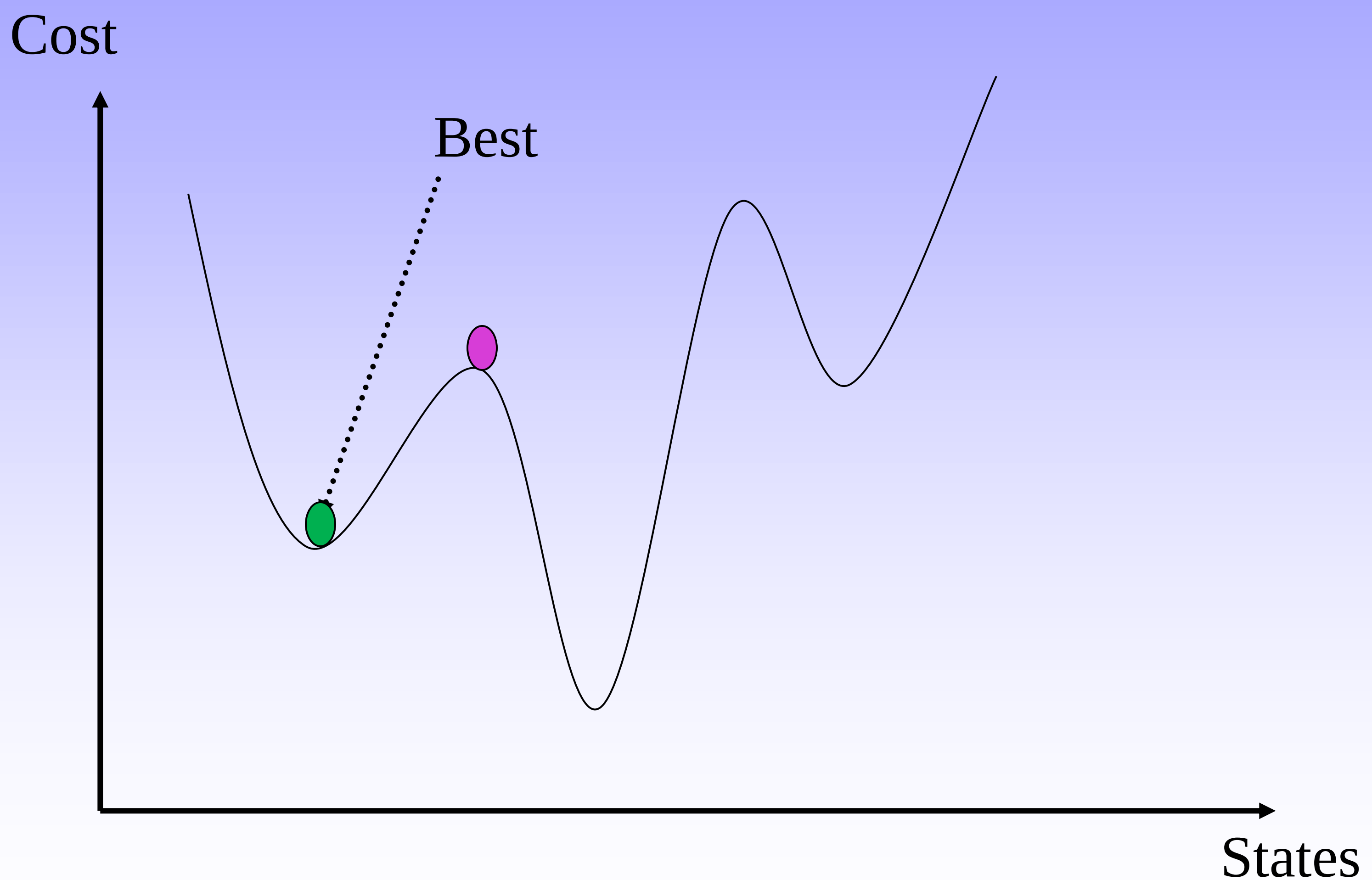
Simulated Annealing Search



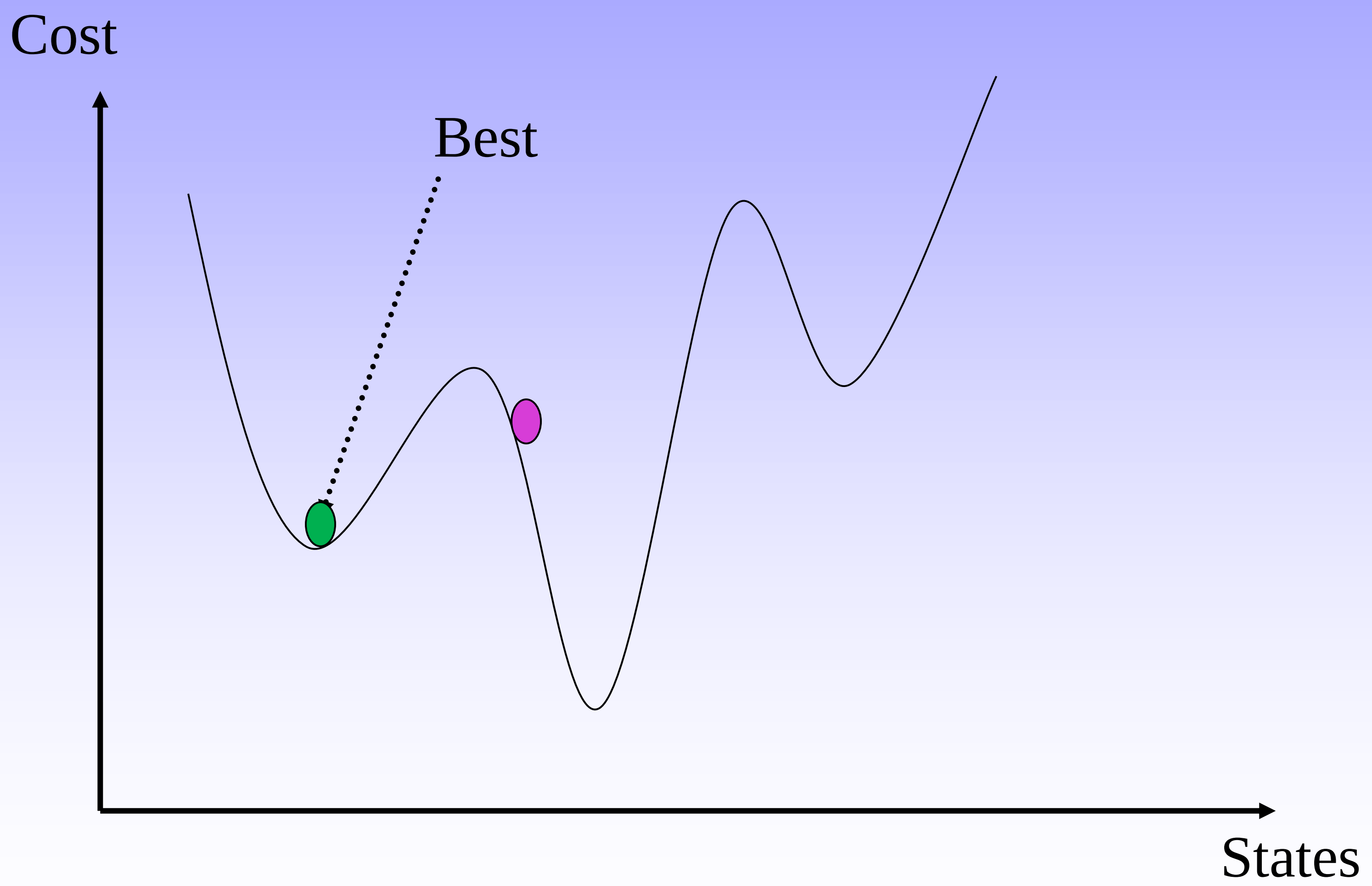
Simulated Annealing Search



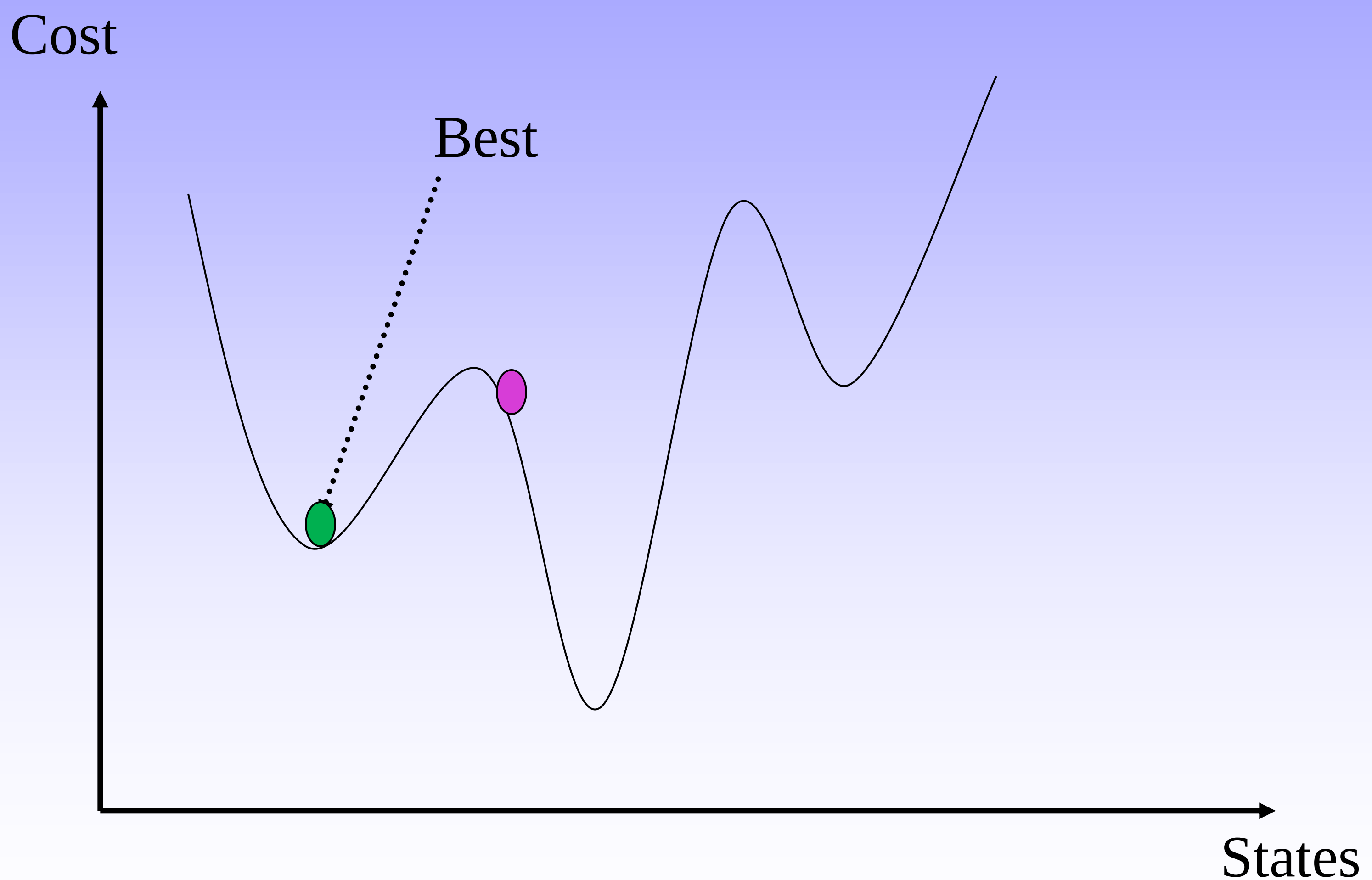
Simulated Annealing Search



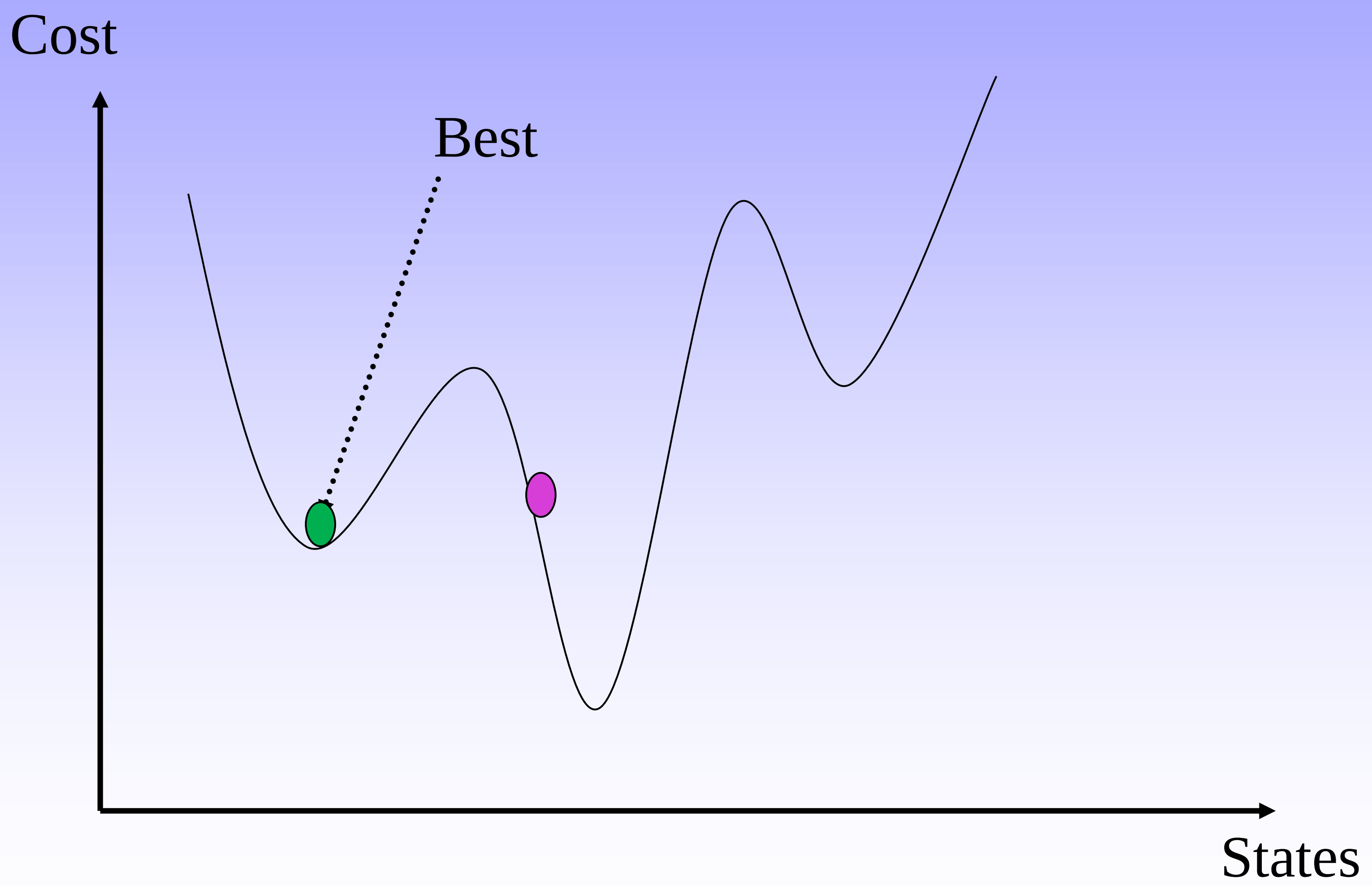
Simulated Annealing Search



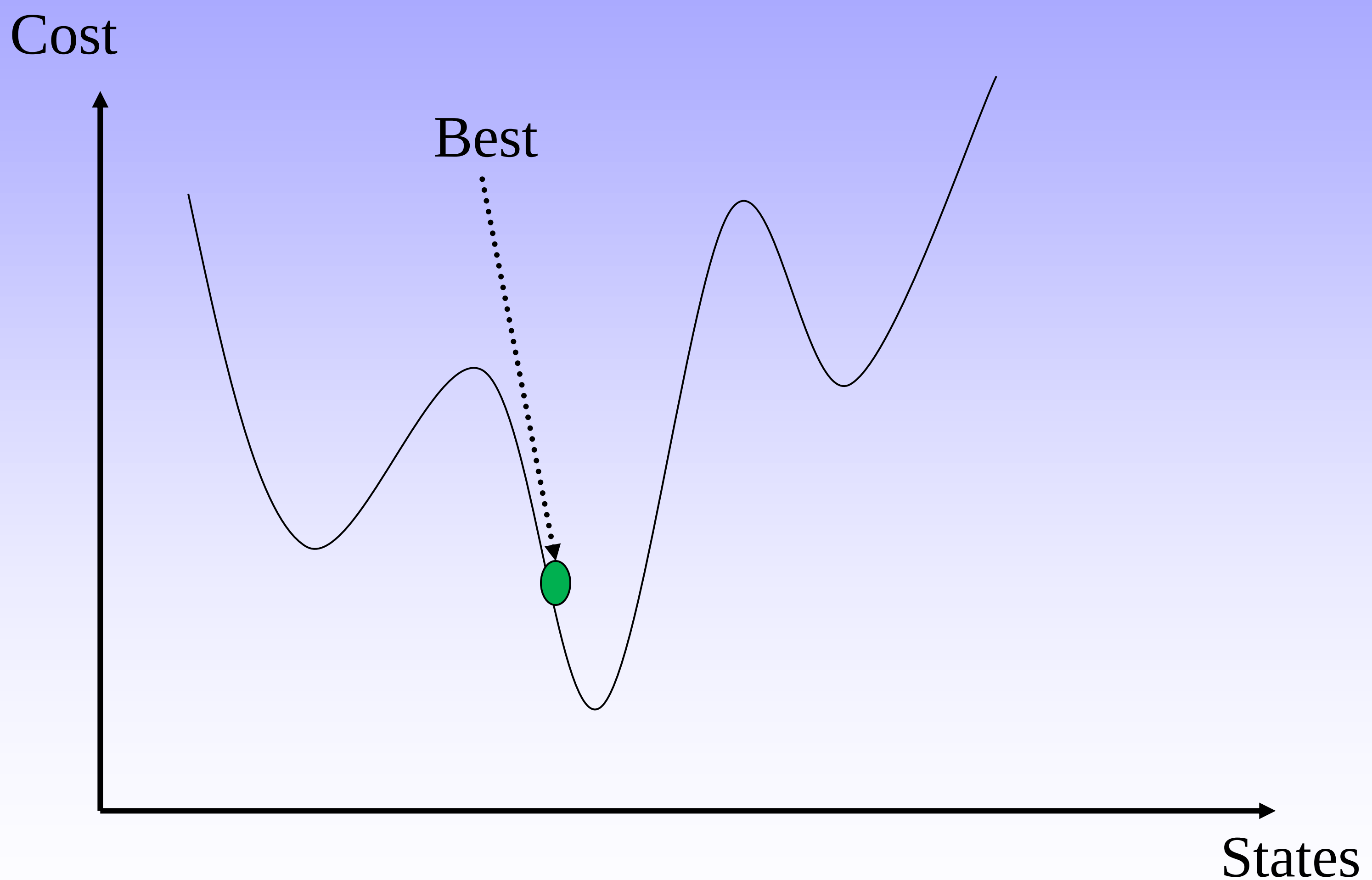
Simulated Annealing Search



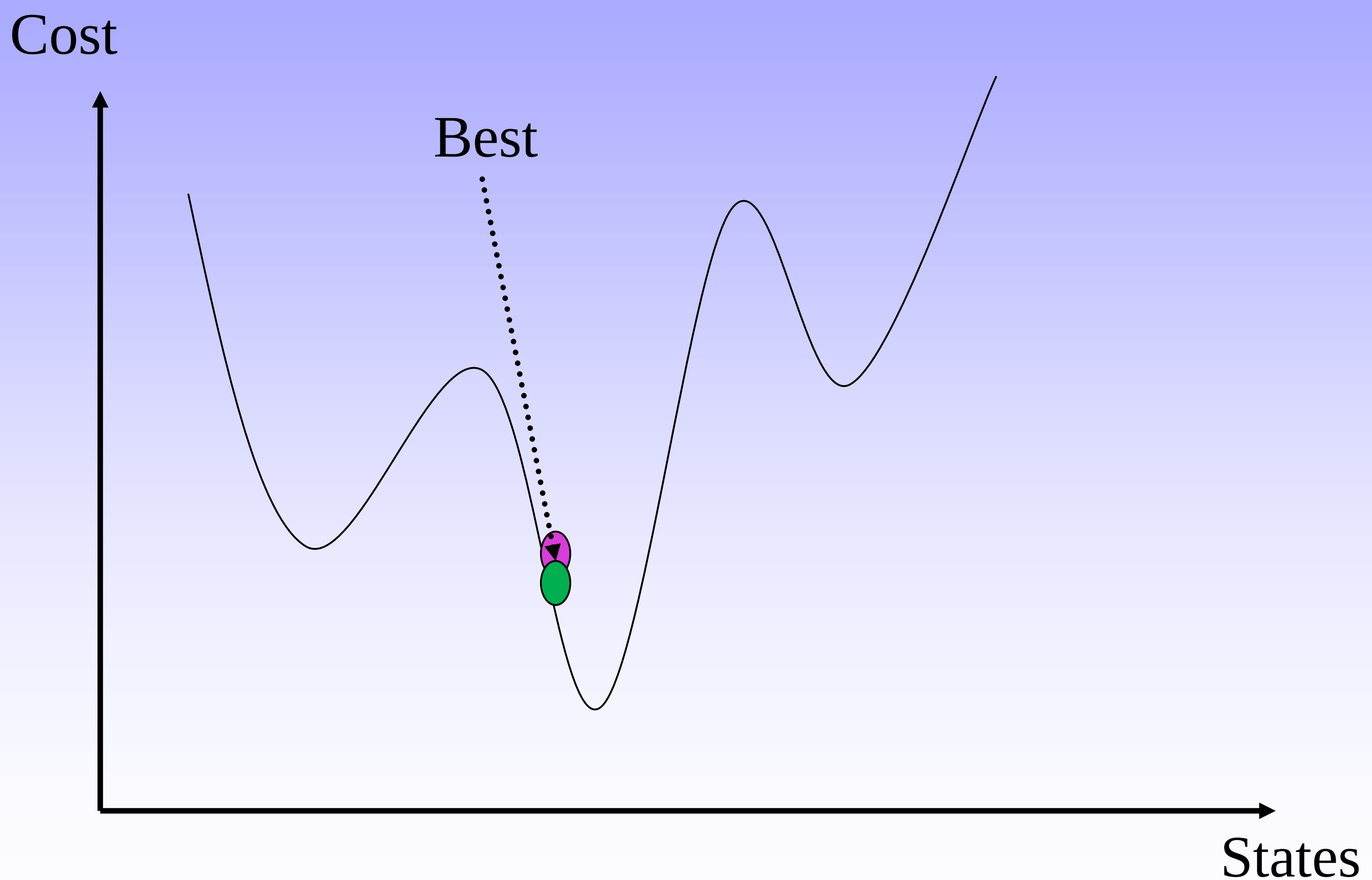
Simulated Annealing Search



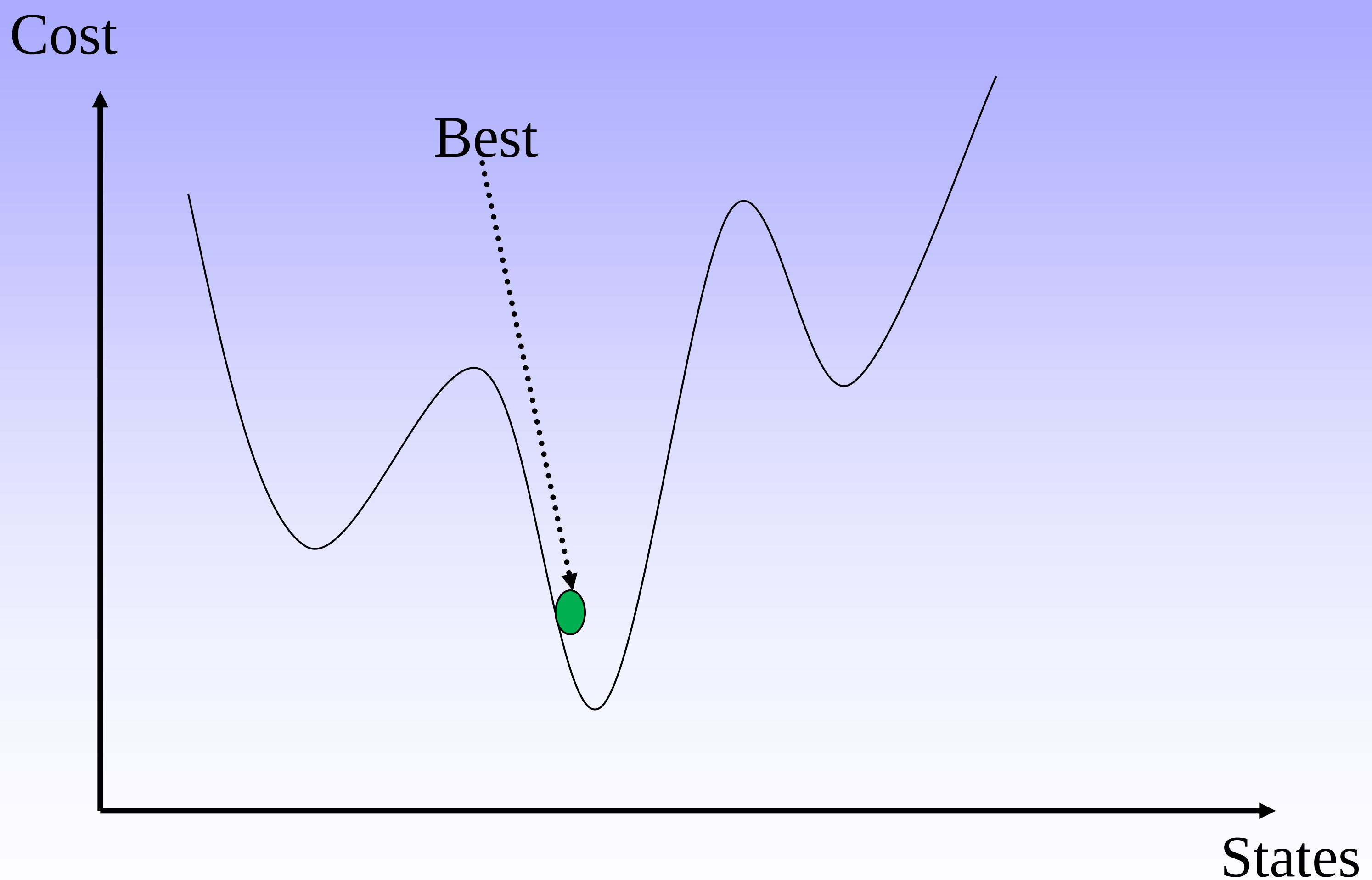
Simulated Annealing Search



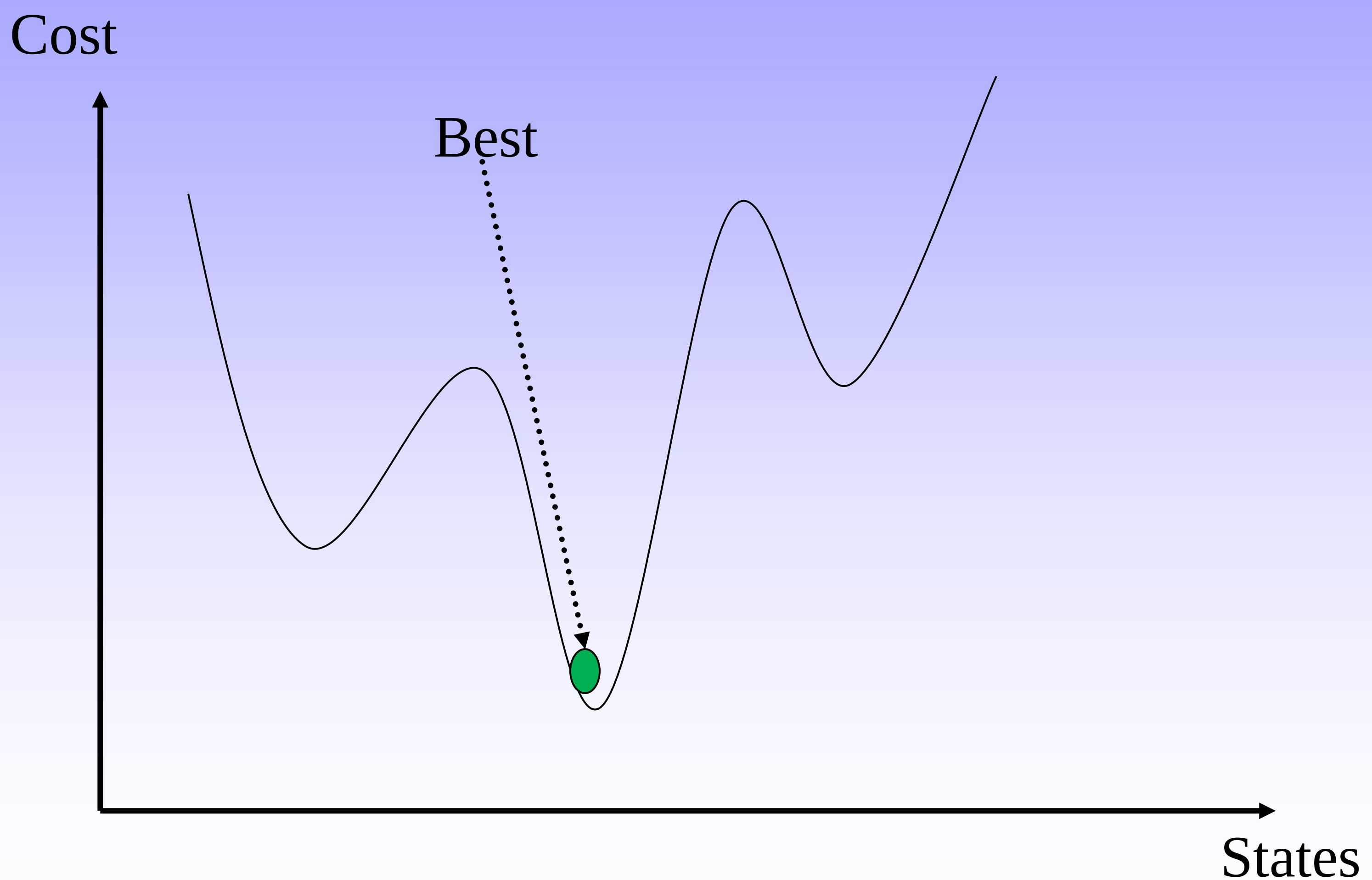
Simulated Annealing Search



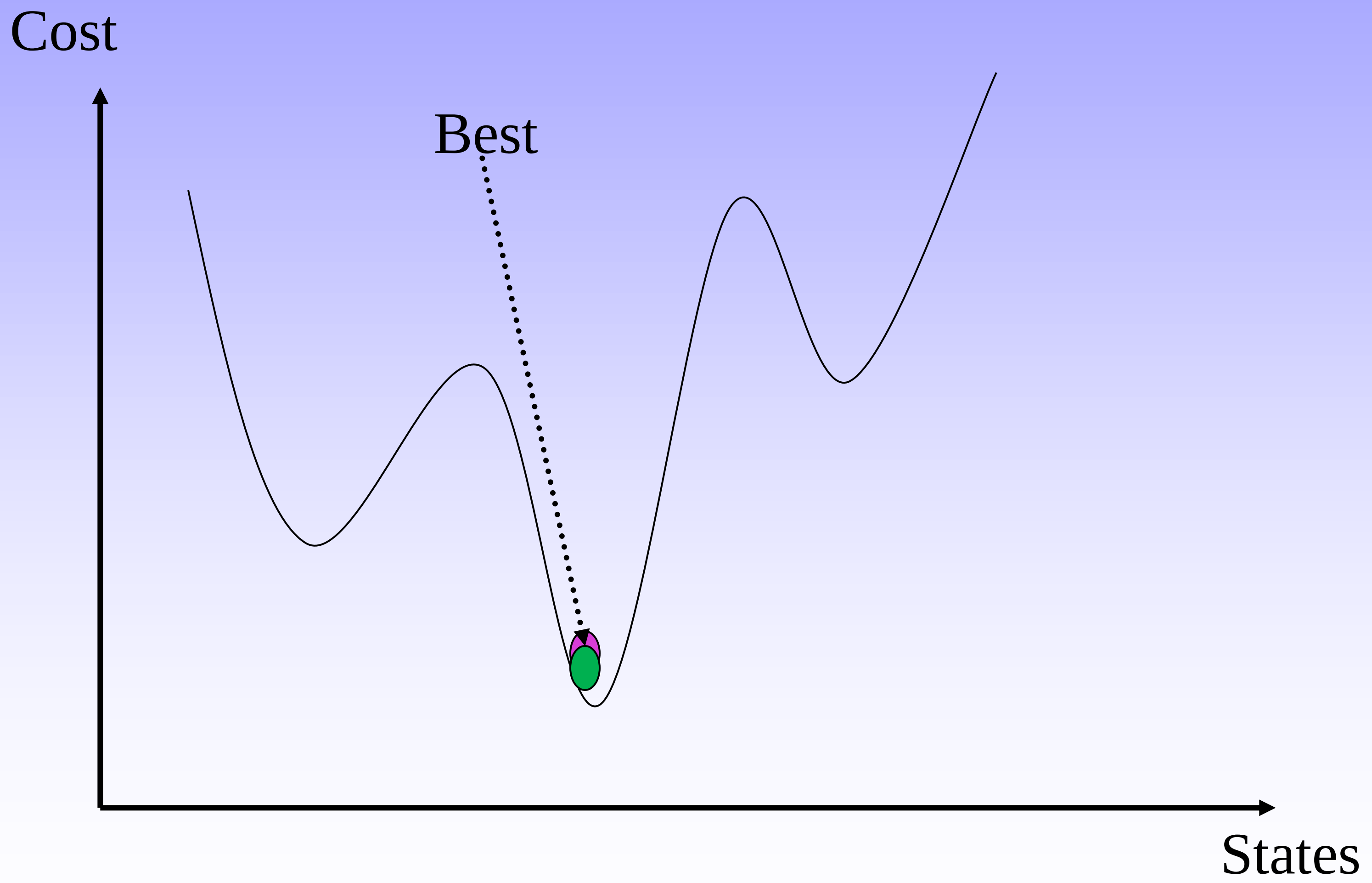
Simulated Annealing Search



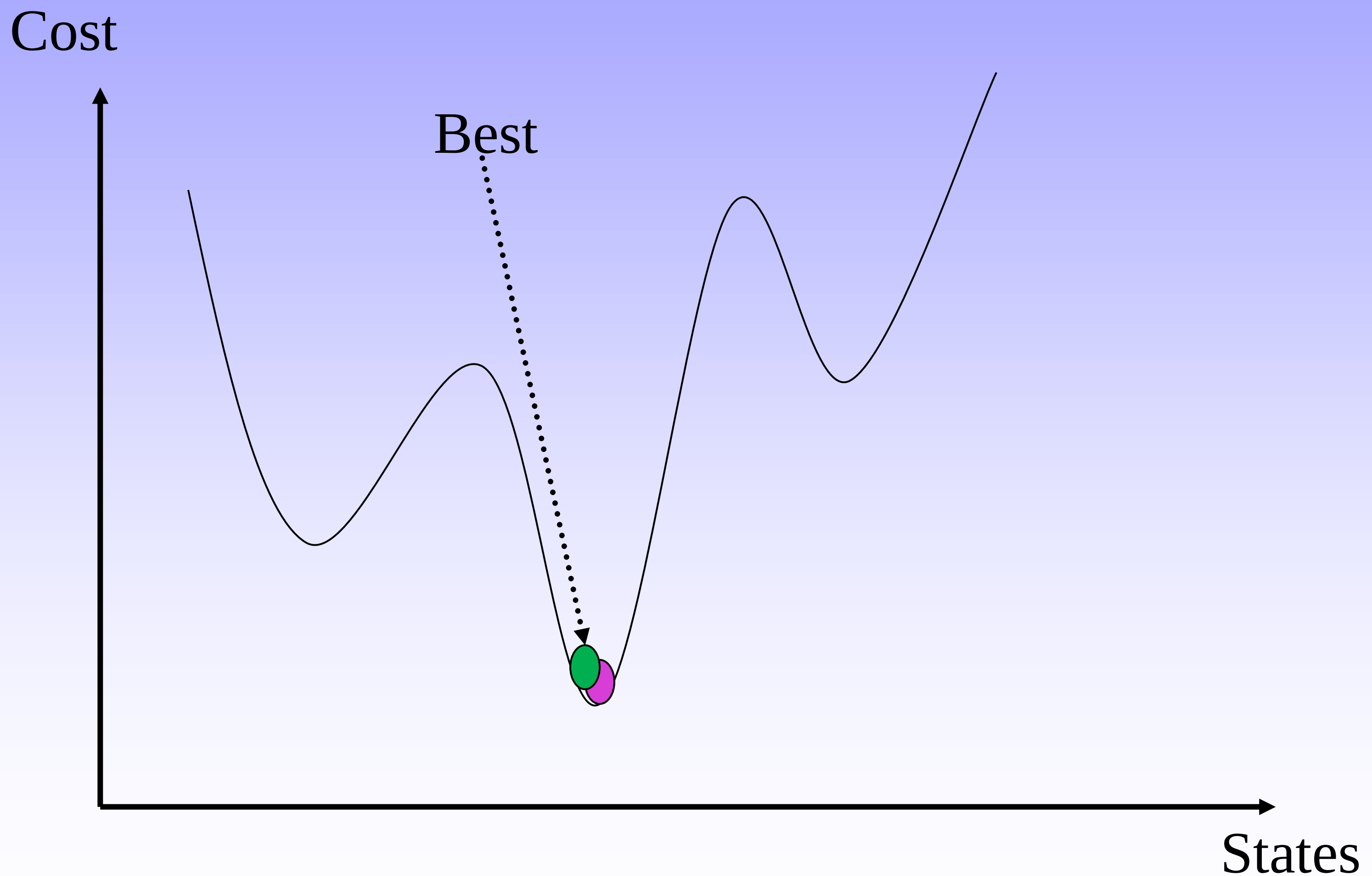
Simulated Annealing Search



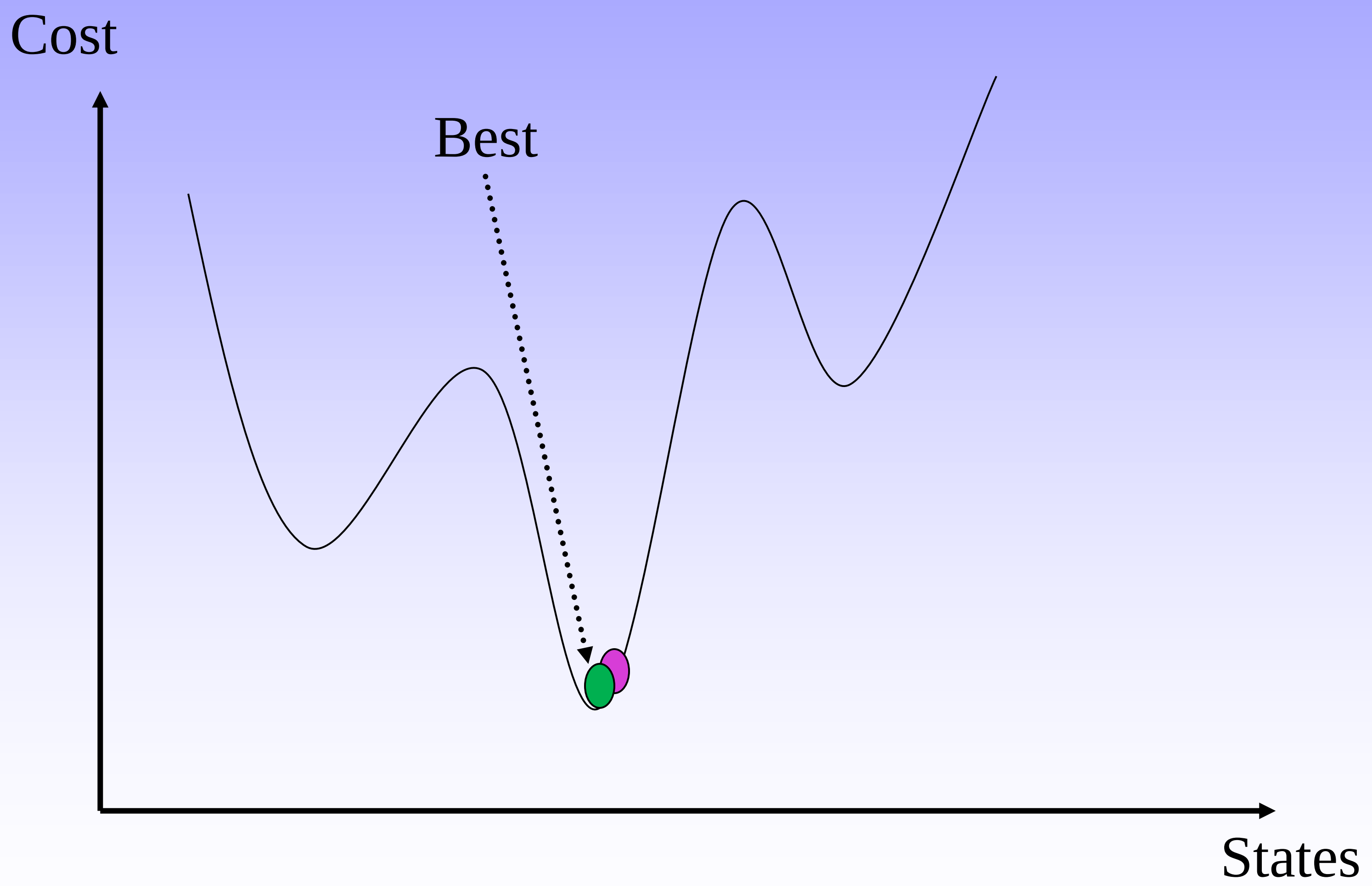
Simulated Annealing Search



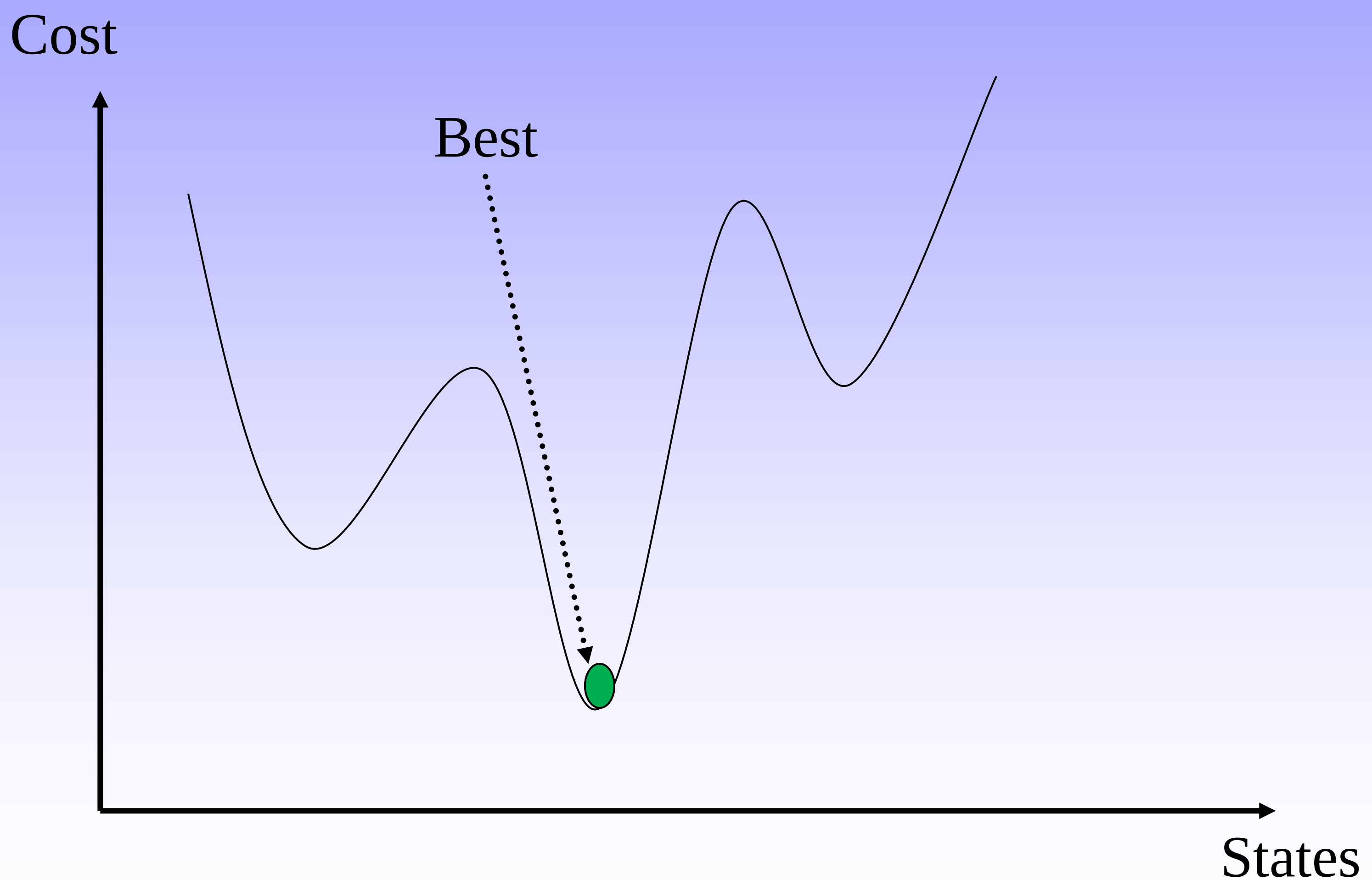
Simulated Annealing Search



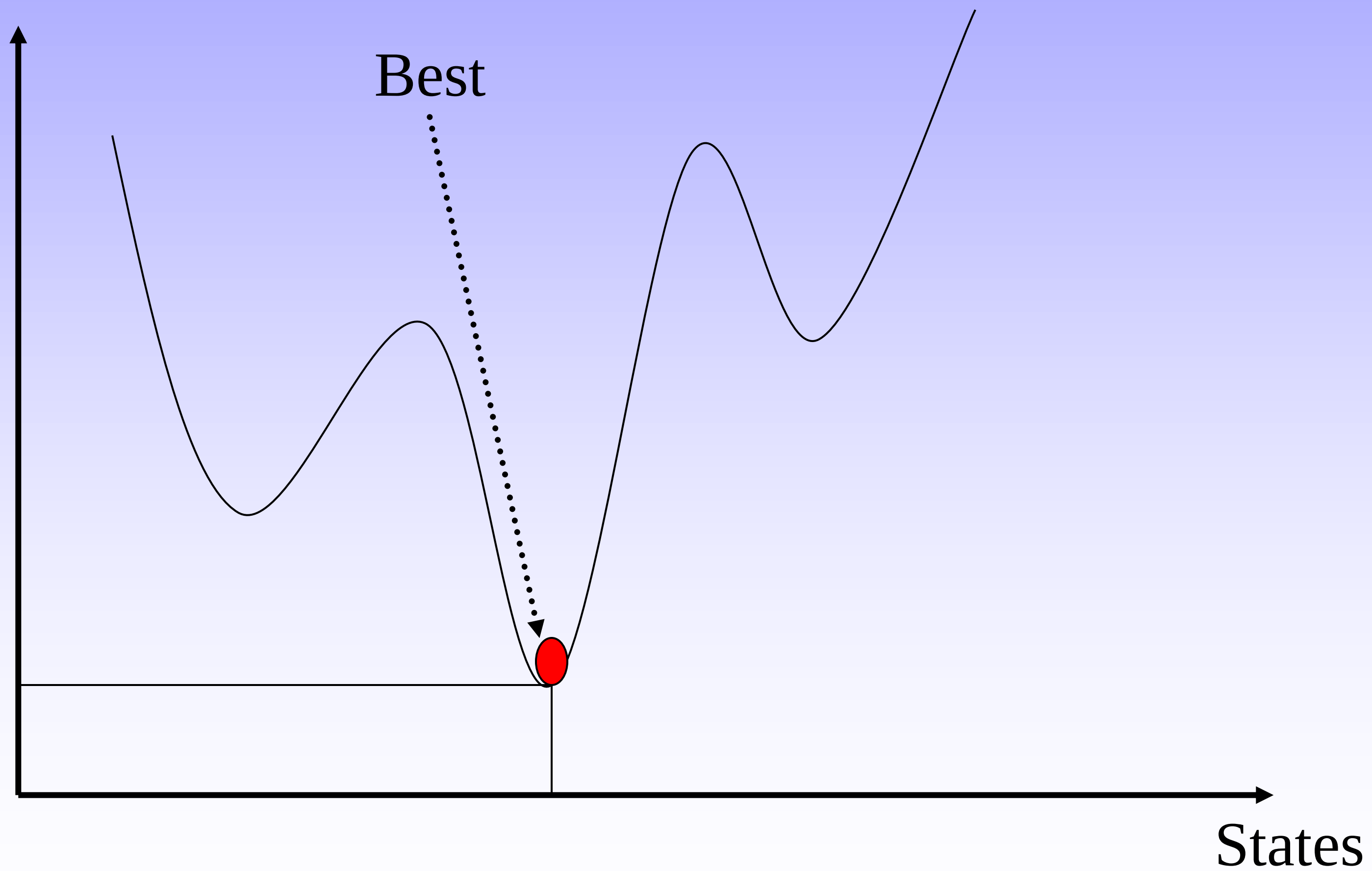
Simulated Annealing Search



Simulated Annealing Search



Simulated Annealing Search



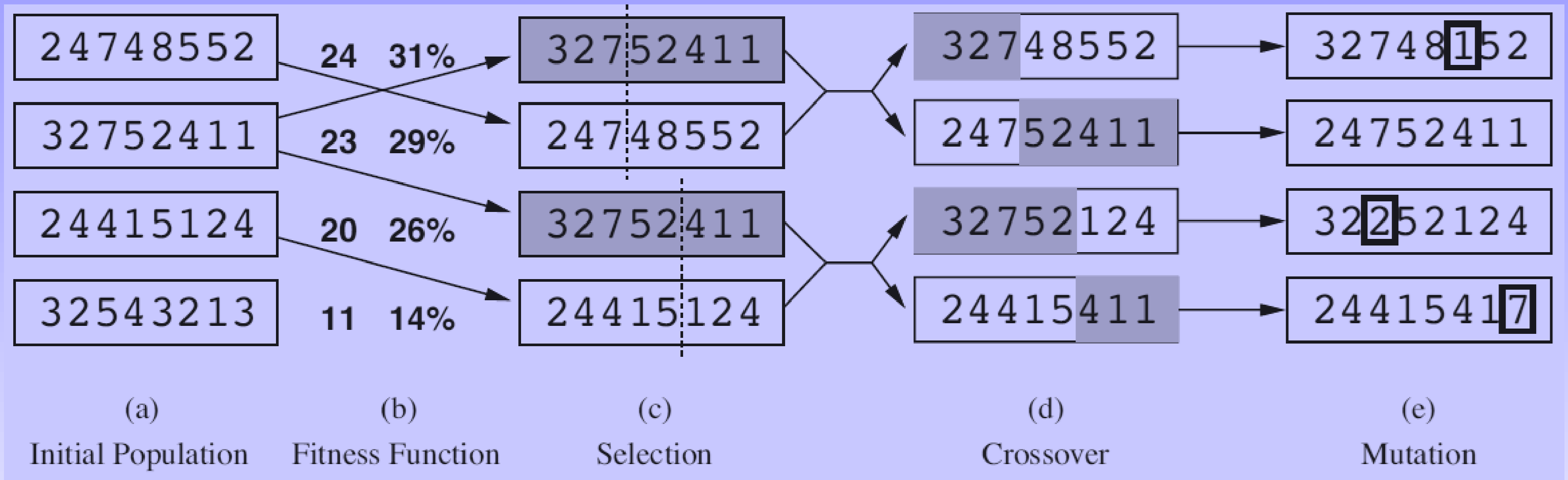
Local Beam Search (LBS)

- Idea: keep track of *k* states rather than just one
- Start with *k* randomly generated states:
  - At each iteration, all the successors of all *k* states are generated.
  - If any one is a goal state, stop; else select the *k* best successors from the complete list and repeat.
- Useful information can be passed among the parallel search threads.
  - LBS quickly abandons unfruitful searches and moves its resources to where the most progress is being made.
- Problem: LBS can suffer from a lack of diversity among the *k* states, making the search an expensive version of hill climbing.

Genetic Algorithms (GAs)

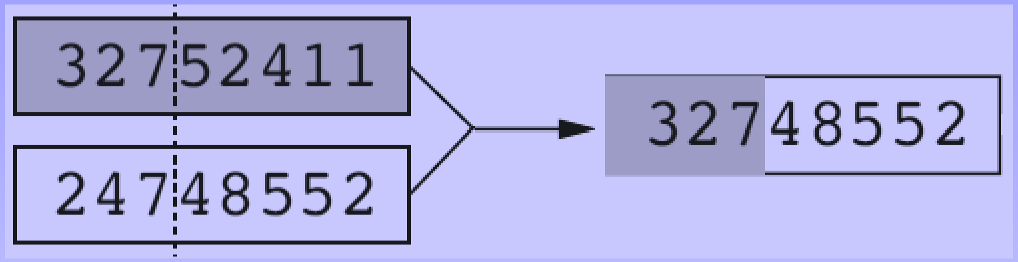
- A successor state is generated by combining two parent states rather than by modifying a single state.
- Like beam searches, start with *k* randomly generated states (called **population**).
  - Each state, or **individual**, is represented as a string over a finite alphabet (most commonly, a string of 0s and 1s).
  - Each state is rated by the objective function (or **fitness function** in GA terminology).
    - » A fitness function should return higher values for better states.
    - » The probability of selecting a state for reproducing is directly proportional to the fitness score.
- Produce the next generation of states by selection, crossover, and mutation.

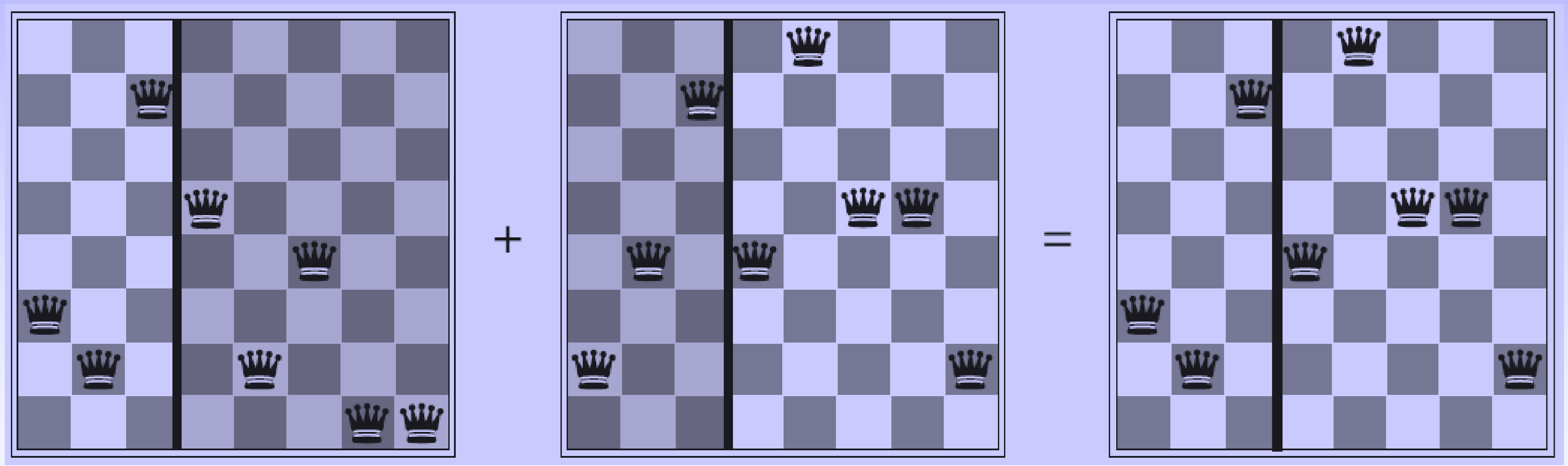
Genetic Algorithms: 8-queens Problem



- <u>Fitness function</u>: number of non-attacking pairs of queens:
  - Min rate = 0, and Max rate = (8 × 7)/2 = 28
  - 24/(24+23+20+11) = 31%
  - 23/(24+23+20+11) = 29% etc.

Genetic Algorithms: 8-queens Problem





- The shaded columns are lost in the crossover step and the unshaded columns are retained.

Genetic Algorithms

- Two pairs are selected at random for reproduction, in accordance with the probabilities.
  - There are many variants of this **selection** rule.
  - The method of *culling*, in which all individuals below a given threshold are discarded, can be shown to converge faster than the random version.
- For each pair to be mated, a **crossover** point is chosen randomly from the positions in the string.
  - The offspring themselves are created by crossing over the parent strings at the crossover point.
- Finally, each location is subject to random **mutation** with a small independent probability.
  - In 8-queens problem, this corresponds to choosing a queen at random and moving it to a random square in its column.

Genetic Algorithms

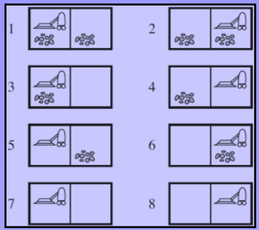
```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
     new_population ← empty set
     for i = 1 to SIZE(population) do
        x ← RANDOM-SELECTION(population, FITNESS-FN)
        y ← RANDOM-SELECTION(population, FITNESS-FN)
        child ← REPRODUCE(x, y)
        if (small random probability) then child ← MUTATE(child)
        add child to new_population
     population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN
```

```
function REPRODUCE(x, y) returns an individual
  inputs: x, y, parent individuals

  n ← LENGTH(x); c ← random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```

Searching with Nondeterministic Actions

- In case of fully observable and deterministic environment, the agent knows the effects of each action and its percepts provide no new information.
  - Therefore, the agent can calculate exactly which state results from any sequence of actions.
- But, when the environment is nondeterministic, percepts tell the agent which of the possible outcomes of its actions has actually occurred.
  - So, the future percepts cannot be determined in advance and the agent's future actions will depend on those future percepts.
  - Thus, a problem solution is not a sequence but a **contingency plan** that specifies what to do depending on what percepts are received.

Erratic Vacuum World



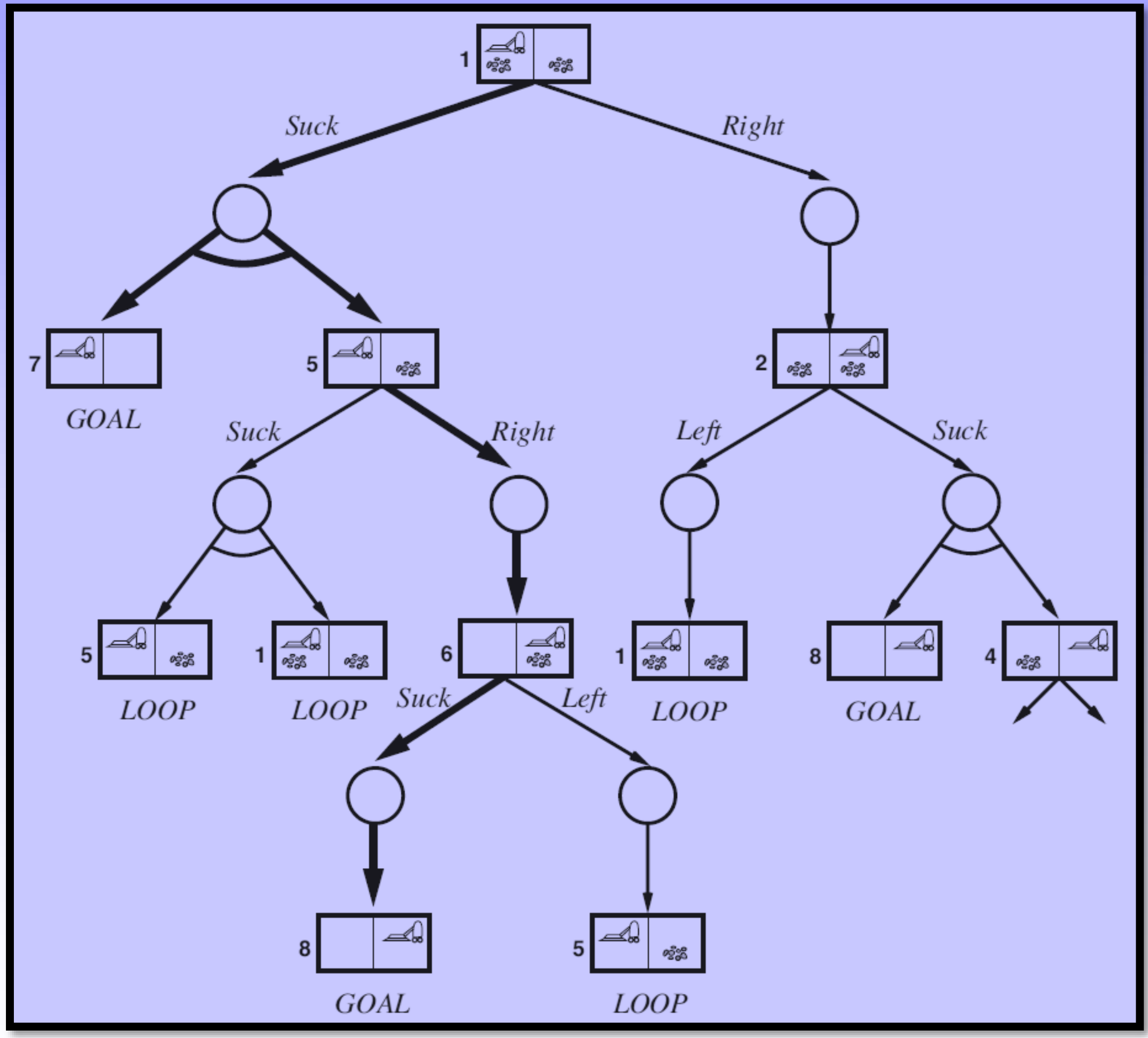
- It is a powerful but erratic vacuum cleaner, such that the *Suck* action works as follows:
  - When applied to a dirty square the action cleans the square and sometimes cleans up dirt in an adjacent square, too.
  - When applied to a clean square the action sometimes deposits dirt on the carpet.
- In this case, we use a RESULTS function that returns a *set* of possible outcome States:
  - The *Suck* action in state 1 leads to a state in the set {5, 7}
- Need also to generalize the notion of a problem **solution**:
  - If we start in state 1, we need a contingency plan such as the following:

    [*Suck*, **if** State = 5 **then** [*Right*, *Suck*] **else** [ ]] .

Searching with Nondeterministic Actions

- Consequently, solutions for nondeterministic problems can contain nested **if–then–else** (or case) statements.
  - Means that the solutions are ***trees*** rather than sequences.
  - This allows the selection of actions based on contingencies arising during execution.
- In the search tree of a deterministic environment, the only branching is introduced by the agent's own choices in each state – we call these nodes **OR nodes**.
- Otherwise, in a nondeterministic environment, branching is also presented by the *environment's* choice of outcome for each action – we call these nodes **AND nodes**.
  - These two kinds of nodes alternate, leading to an **AND–OR tree**.

Erratic Vacuum World: AND–OR Tree



Searching with Nondeterministic Actions

- A solution for an AND–OR search problem is a subtree that:

  (1) has a goal node at every leaf

  (2) specifies one action at each of its OR nodes

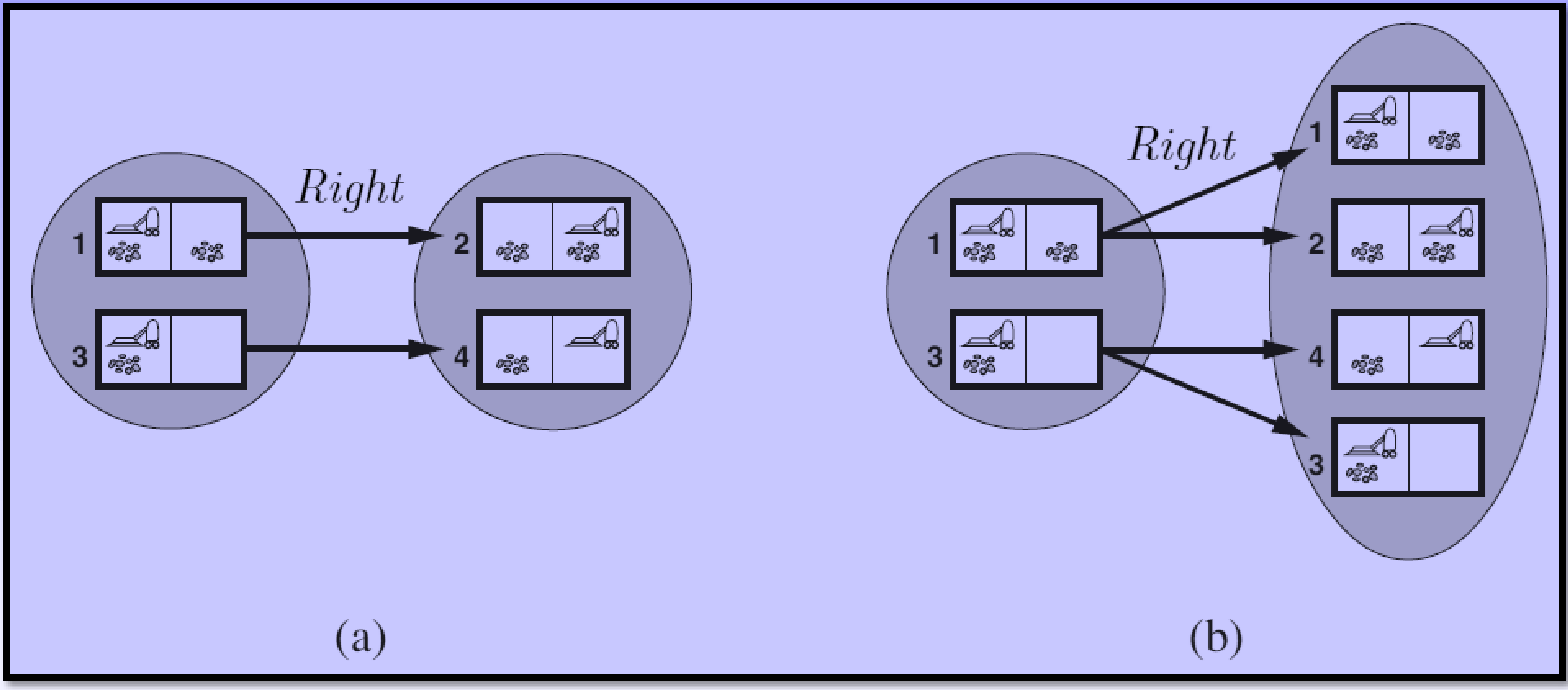
  (3) includes every outcome branch at each of its AND nodes

- AND–OR trees can be explored by depth-first, breadth-first or best-first methods.
  - The concept of a heuristic function must be modified to estimate cost of a contingent solution rather than a sequence.
  - But the notion of admissibility carries over and there is an analog of the A^* algorithm for finding optimal solutions.
- This type of interleaving of search and execution is also useful for exploration problems and for game playing.

Searching with No Observation

- When the agent's percepts provide *no information at all*, we have what is called a **sensorless** problem.
  - Sensorless agents can be useful, primarily because they *don't* rely on sensors working properly.
  - Many ingenious methods have been developed for orienting parts correctly from an unknown initial position.
- To solve sensorless problems, we search in the space of **belief states** rather than physical states.
  - A belief state represents the agent's current belief about the possible physical states it might be in, given the sequence of actions and percepts up to that point.
- In belief-state space, the problem is fully observable and the solution (if any) is always a sequence of actions.

Sensorless Vacuum World



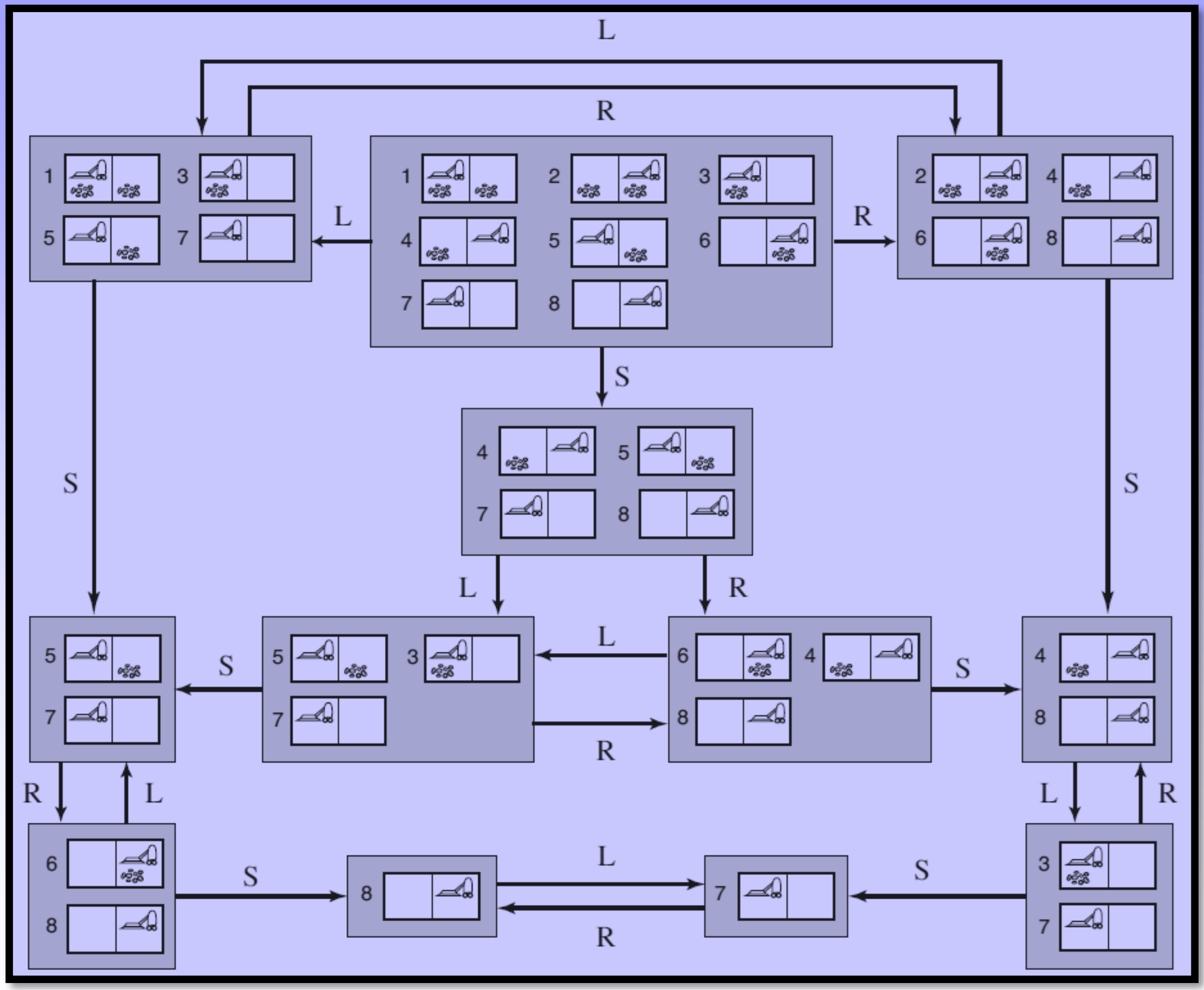
(a) Predicting the next belief state with a deterministic action, *Right*.

(b) Prediction for the same belief state and action in slippery vacuum.

Searching with No Observation

- The entire belief-state space contains every possible set of physical states.
  - If a problem has N states, then the sensorless problem has up to 2^N states, although many may be unreachable from the initial state.
- Typically, the initial state of a problem is the set of all physical states of that problem.
- The agent doesn't know which physical state in the belief state is the right one.
  - It might get to any of the physical states resulting from applying a certain action to one of the states in the belief state.
- A belief state satisfies the goal only if *all* the physical states in it satisfy the problem GOAL-TEST.

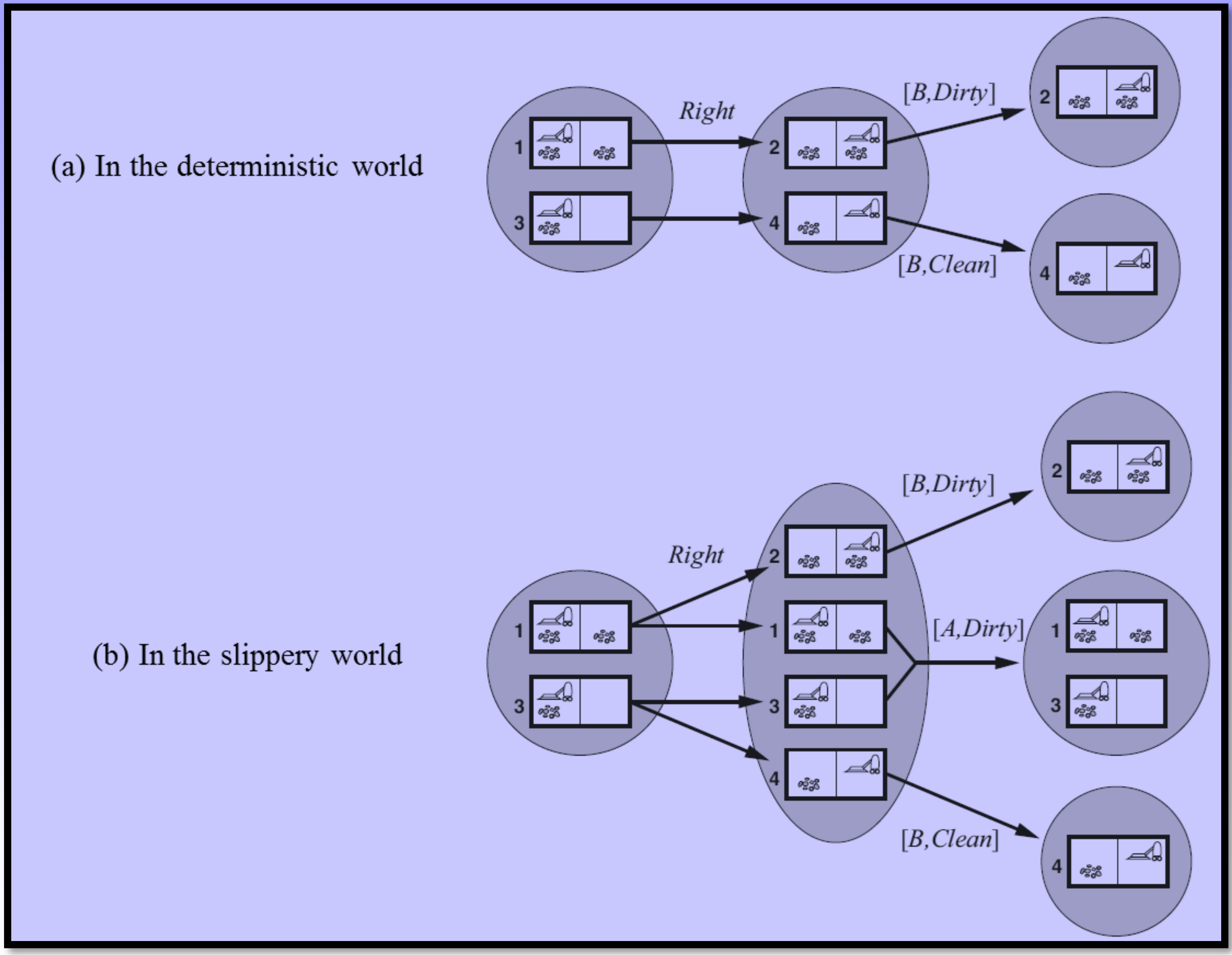
Sensorless Vacuum: Belief-state Space



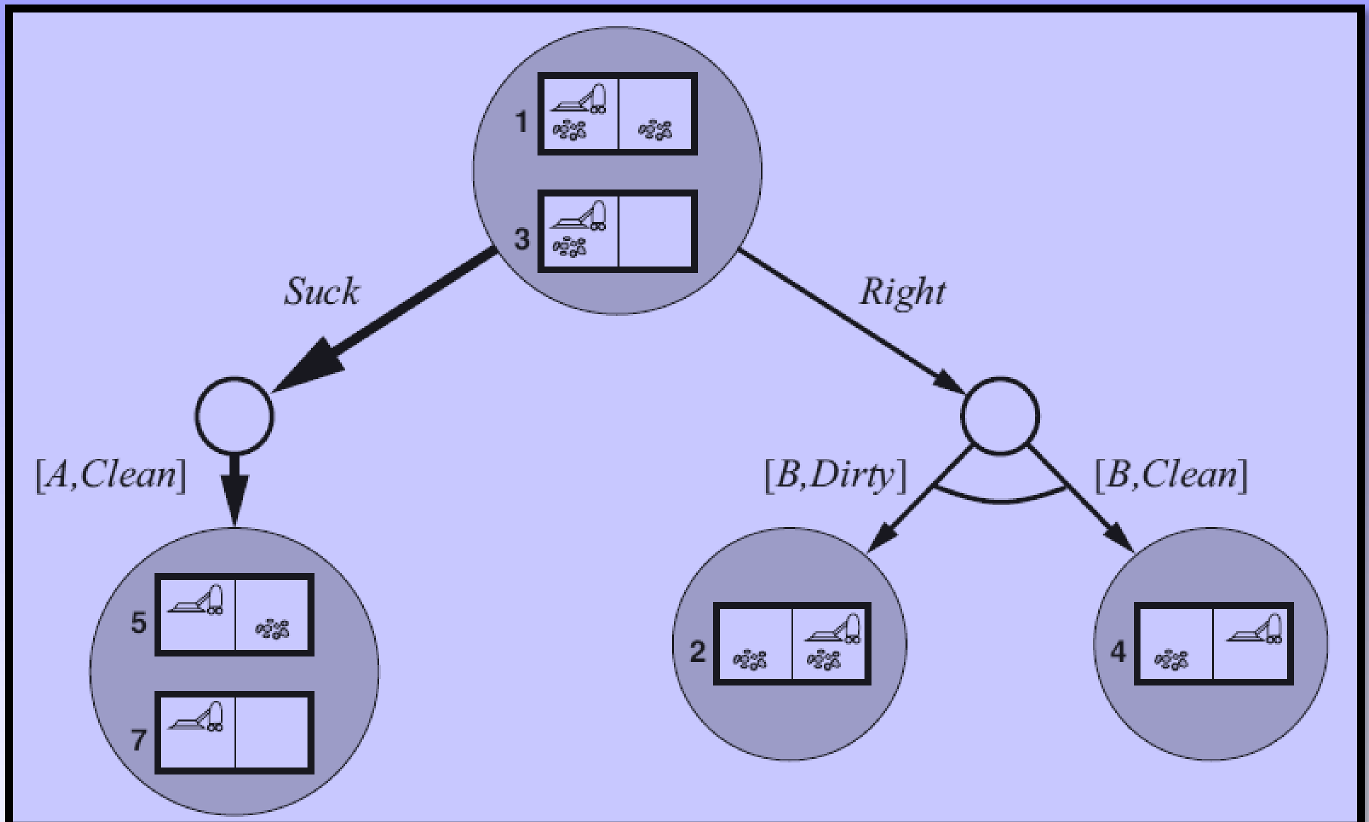
Searching with Partial Observations

- When observations are partial, the agent's percepts do not suffice to pin down the exact state.
  - It is the case that several states could have produced any given percept.
- The belief-state space of a problem is constructed from the underlying physical problem just as for sensorless problem, but the transition model is more complicated.
  - Partial observations can only help reduce uncertainty compared to the sensorless case.
- Example: Local-sensing vacuum world
  - The agent has a position sensor and a local dirt sensor.
  - But it has no sensor capable of detecting dirt in other squares.

Local-sensing Vacuum World



Local-sensing Vacuum: AND–OR Tree



Assuming an initial percept [*A*, *Dirty*] then the solution is the conditional plan as follow:

[*Suck*, *Right*, **if** *Dirty* **then** *Suck* **else** [ ]] .