# Beyond Classical Search
# Chapter 4

## Some real world problems

- Suppose you had to solve VLSI layout problems (minimize distance between components, unused space, etc.)…
- schedule airlines
- schedule workers with specific skills sets to do tasks that have resource and ordering constraints

## Common to all such problems

Characteristics of these problems are:

- The path to the goal is irrelevant -- all you care about is the final configuration
- These are often optimization problems in which you find the best state according to an objective function
- These problems are examples of local search problems

## Iterative Improvement Algorithms

- Hill climbing
- Simulated annealing
- Genetic algorithms

## Iterative Improvement Algorithms

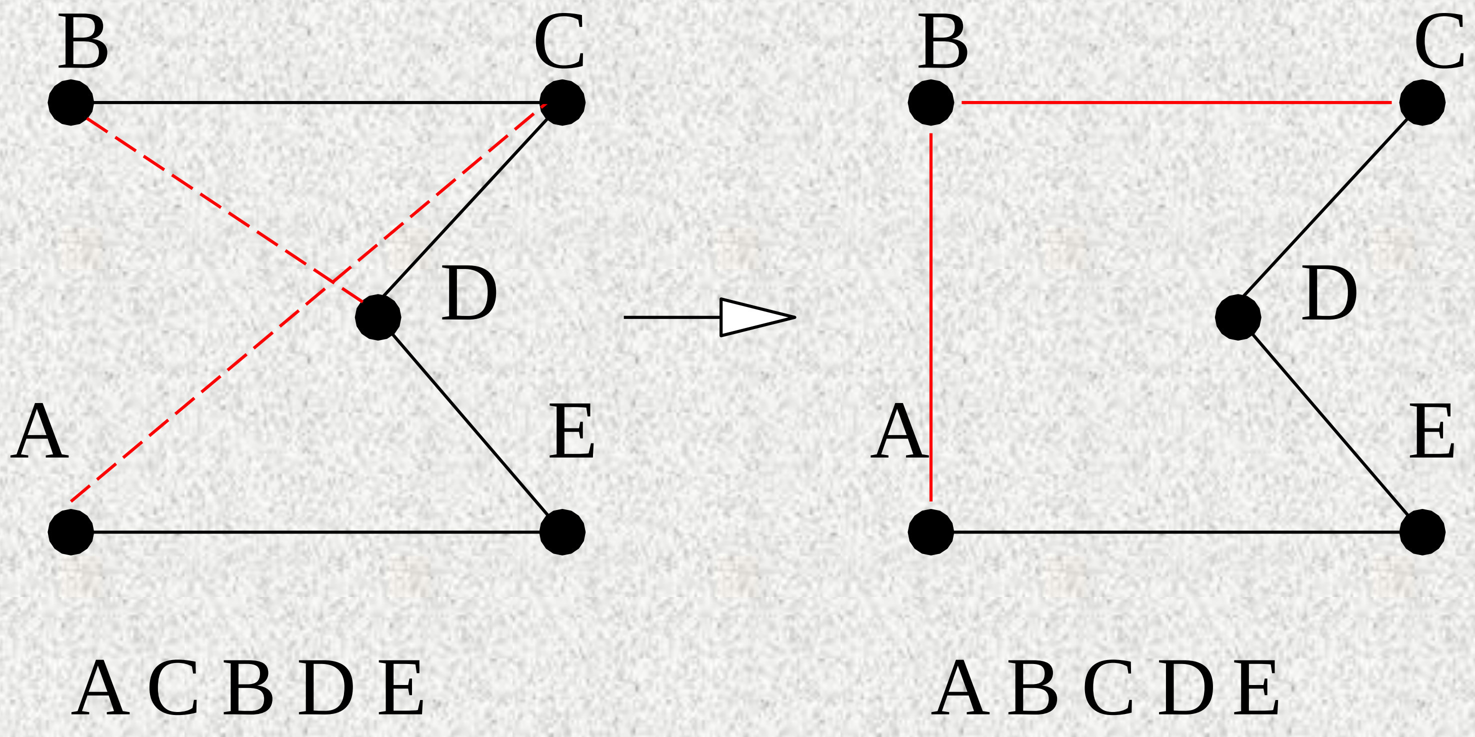
- In the most problems the solution is the path. For example, the solution to the 8-puzzle is a series of movements for the "blank tile." The solution to the traveling in Romania problem is a sequence of cities to get to Bucharest.
- For many optimization problems, solution path is irrelevant
  - Just want to reach goal state

## Iterative Improvement Algorithms

- State space / search space
  - Set of “complete” configurations
  - Want to find optimal configuration (or at least one that satisfies goal constraints)
- For these cases, use iterative improvement algorithm
  - Keep a single current state
  - Try to improve it
- Constant memory: The space  complexity is constant!

# Example: Travelling Salesperson Problem

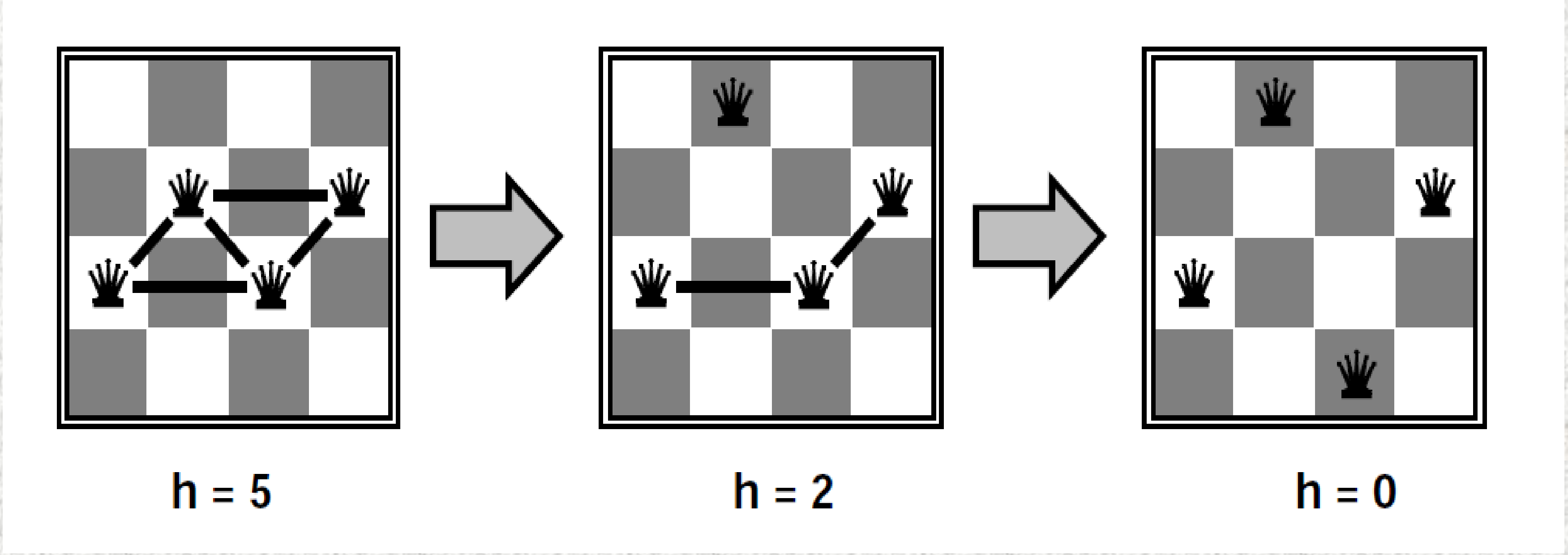
Start with any complete tour, perform pairwise exchanges



Variants of this approach get within 1% of optimal very quickly with thousands of cities.

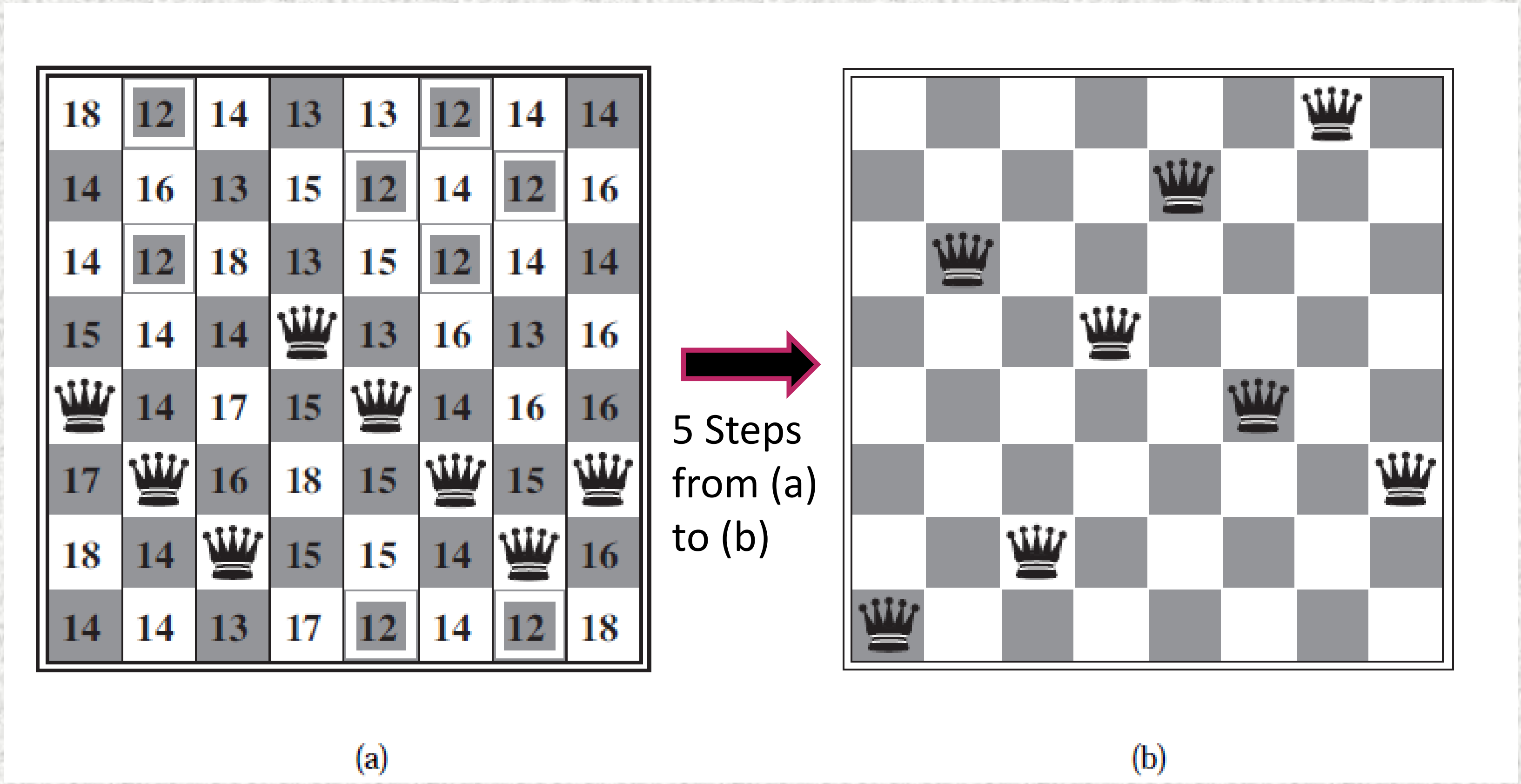
# Example: *n*-queens

Put *n* queens on an *n* × *n* board with no two queens on the same row, column, or diagonal. Move a queen to reduce the number of conflicts (*h*).



Almost always solves *n*-queens problems almost instantaneously for very large *n*, e.g., *n* = 1 million.

# Example: *n*-queens (cont'd)



(a) shows the value of *h* for each possible successor obtained by moving a queen within its column. The marked squares show the best moves.

(b) shows a local minimum: the state has h = 1 but every successor has higher cost. A Local Minima

# Hill-climbing (or gradient ascent/descent)

```
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
```

The hill-climbing search algorithm, which is the most basic local search technique. At each step the current node is replaced by the best neighbor; in this version, that means the neighbor with the highest VALUE, but if a heuristic cost estimate h is used, we would find the neighbor with the lowest h

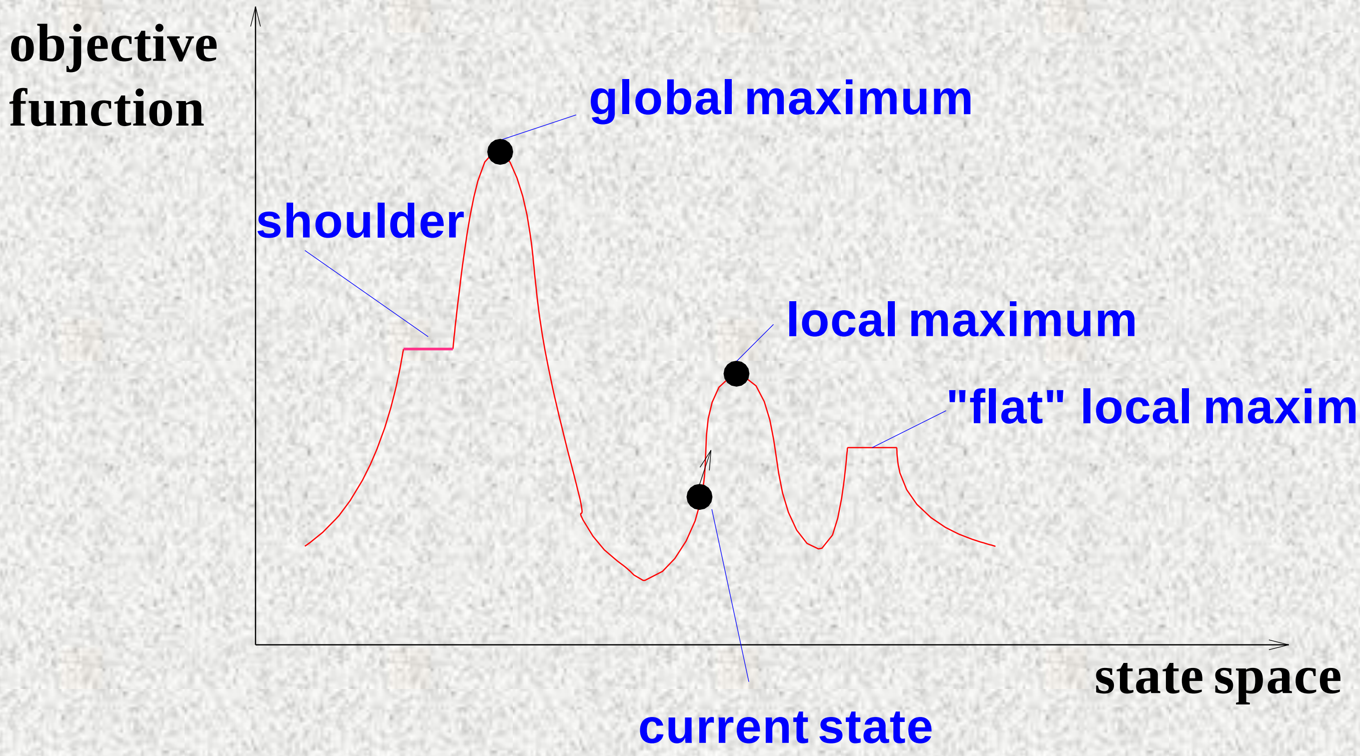
# Hill-climbing (cont’d)

**“Like climbing Everest in thick fog with amnesia.”**

**Problem: depending on initial state, can get stuck on local maxima**

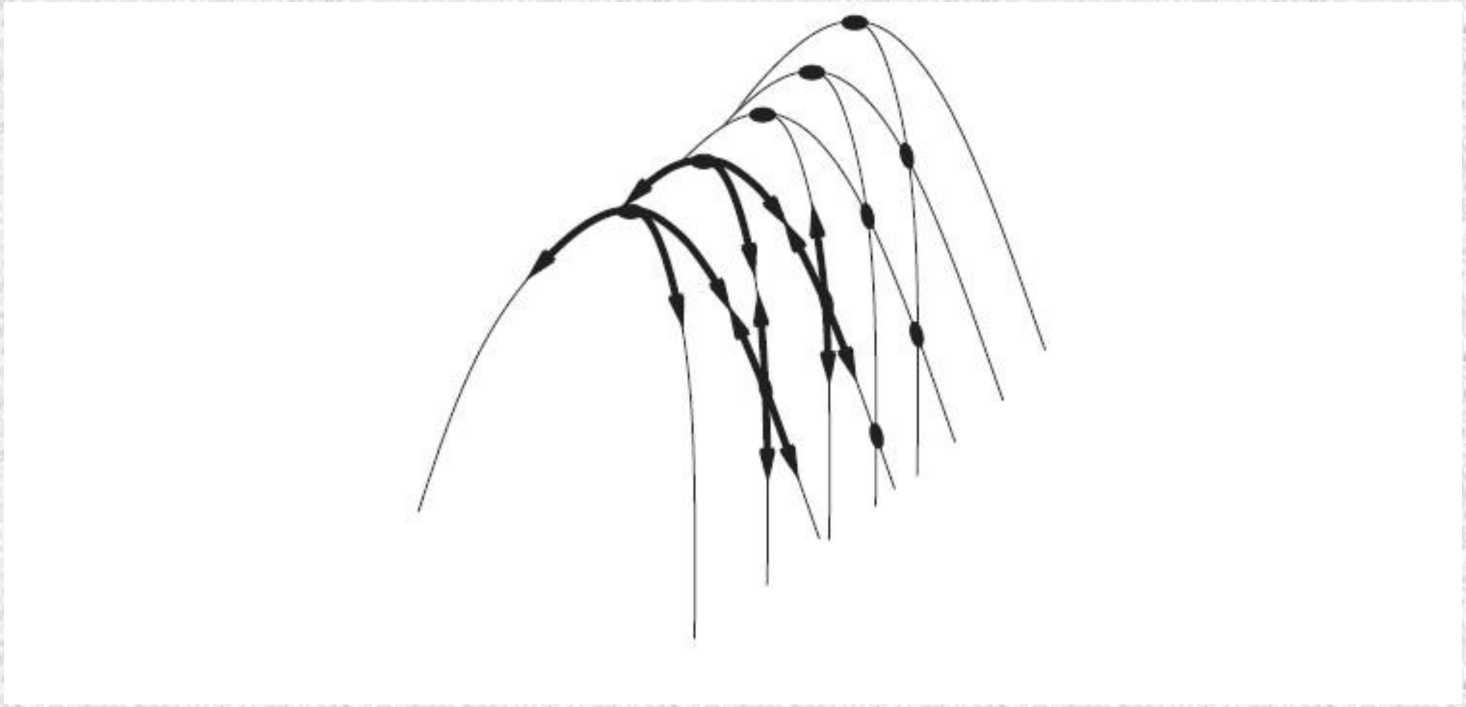
**Random-Restart HC: Overcomes local maxima**

**Random side ways moves: Escapes from shoulders but loops on flat maxima**



## Difficulties with ridges

The “ridge” creates a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available actions point downhill.



## Variation of Hill-Climbing techniques

stochastic: choose randomly from uphill moves. The probability of selection can vary with steepness. This variation converges more slowly than steepest ascent, but in some state landscapes it finds better solutions.

first-choice: generate successors randomly one-by- one until one better than the current state is found. This is a good strategy for states with many (e.g., thousands) of successors.

Random-restart: restart with a randomly generated initial state to escape from local maxima or plateau

Better fix: Instead of picking the best move pick *any move that produces an improvement.* This is called *randomized hill climbing*

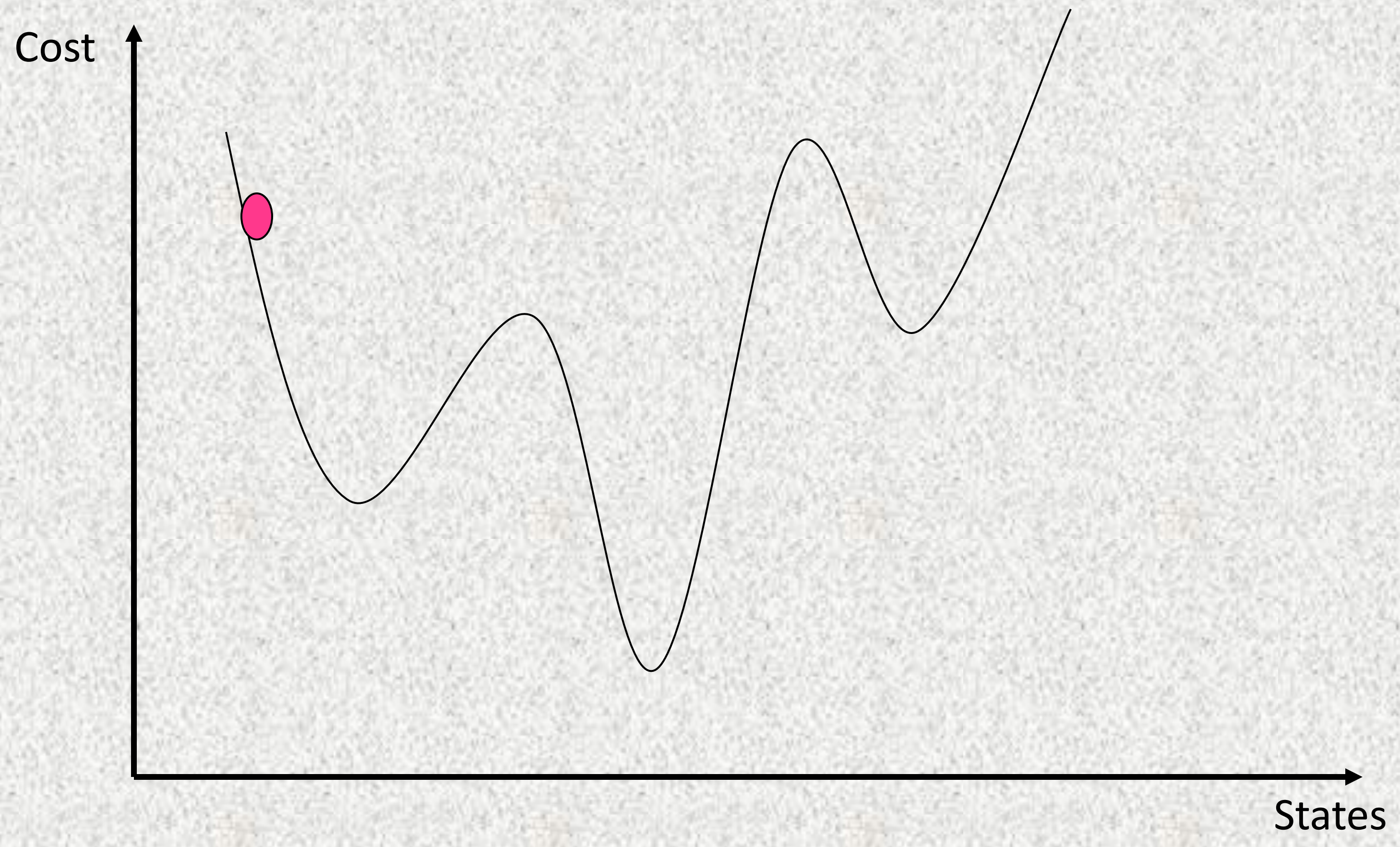
# Random-Restart Hill-Climbing

**Advice:** If at first you don’t succeed, try, try again!
Random-restart hill-climbing conducts a series of hill-climbing searches from randomly generated initial states, stopping when a goal is found.
With probability approaching 1, we will eventually generate a goal state as the initial state.
If each hill-climbing search has a probability *p* of success, then the expected number of restarts required to reach a solution is 1/*p*.

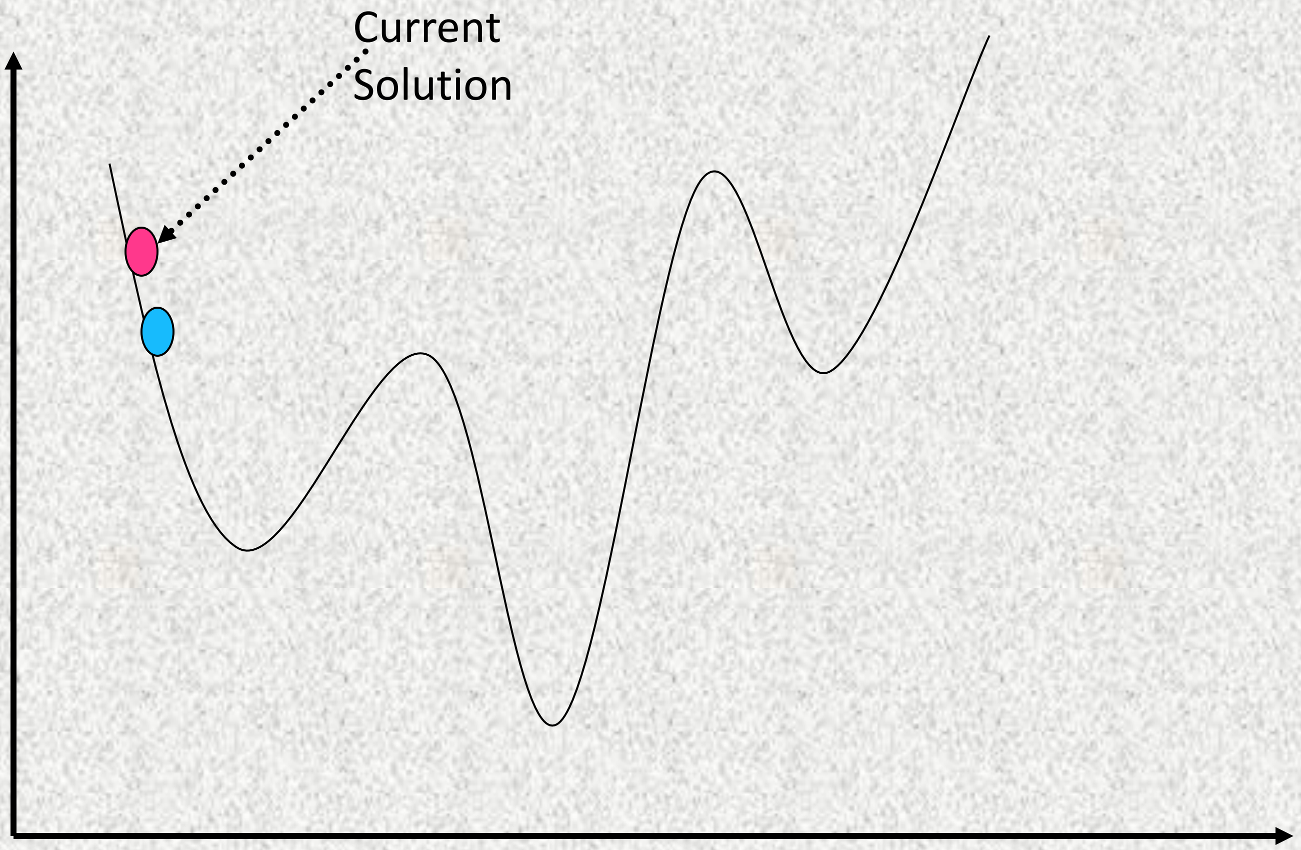
# Random-Restart Hill-Climbing Cont'd.

**Example:** 8-queens As we saw earlier, $p \approx 0.14$.
In this case we need roughly 7 iterations (6 failures and 1 success).
Random restart hill-climbing is **very effective** for 8-queens.
Even for **three million queens**, the approach can find solutions in under a minute.
The success of random-restart hill-climbing depends very much on the **shape** of the state space. There are practical problems with state spaces with very bad shapes.

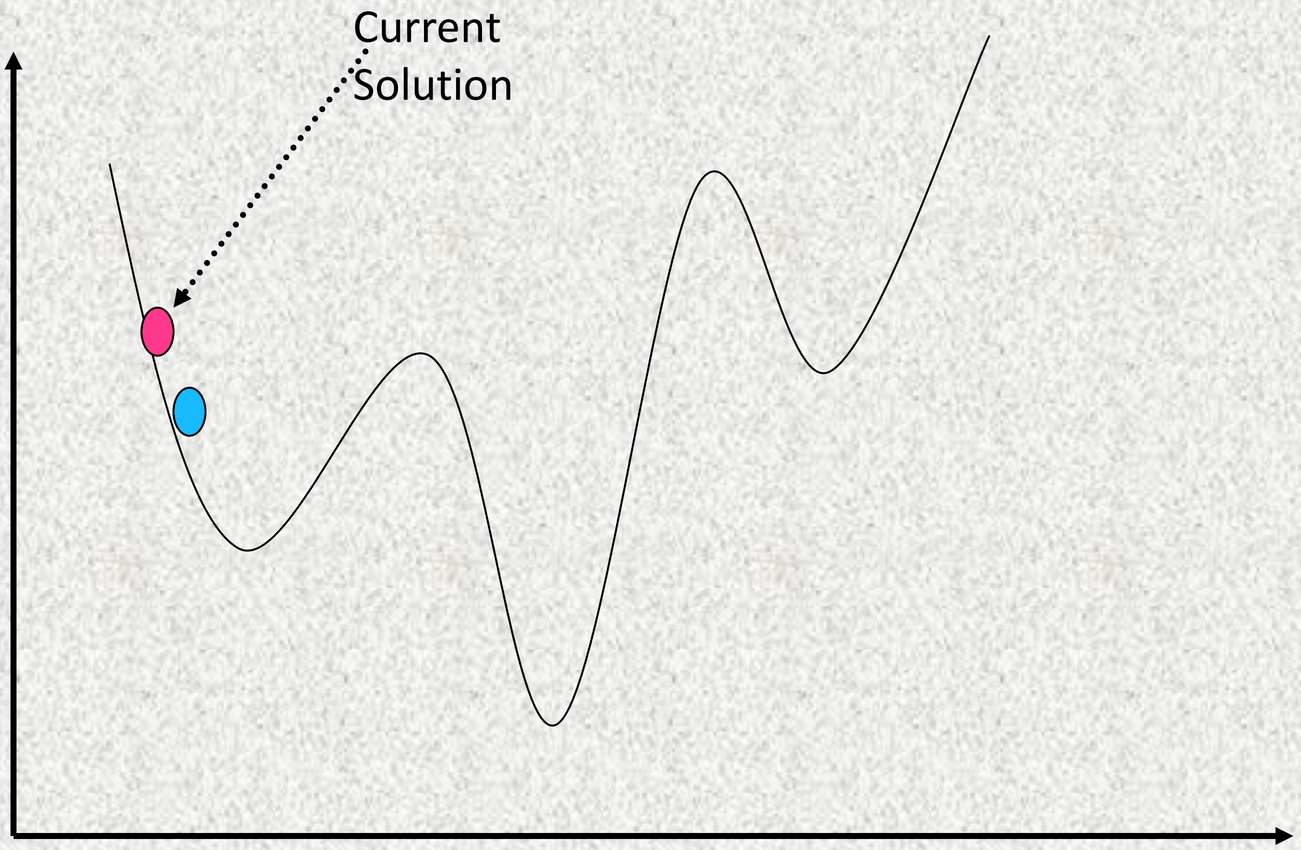
# Hill Climbing



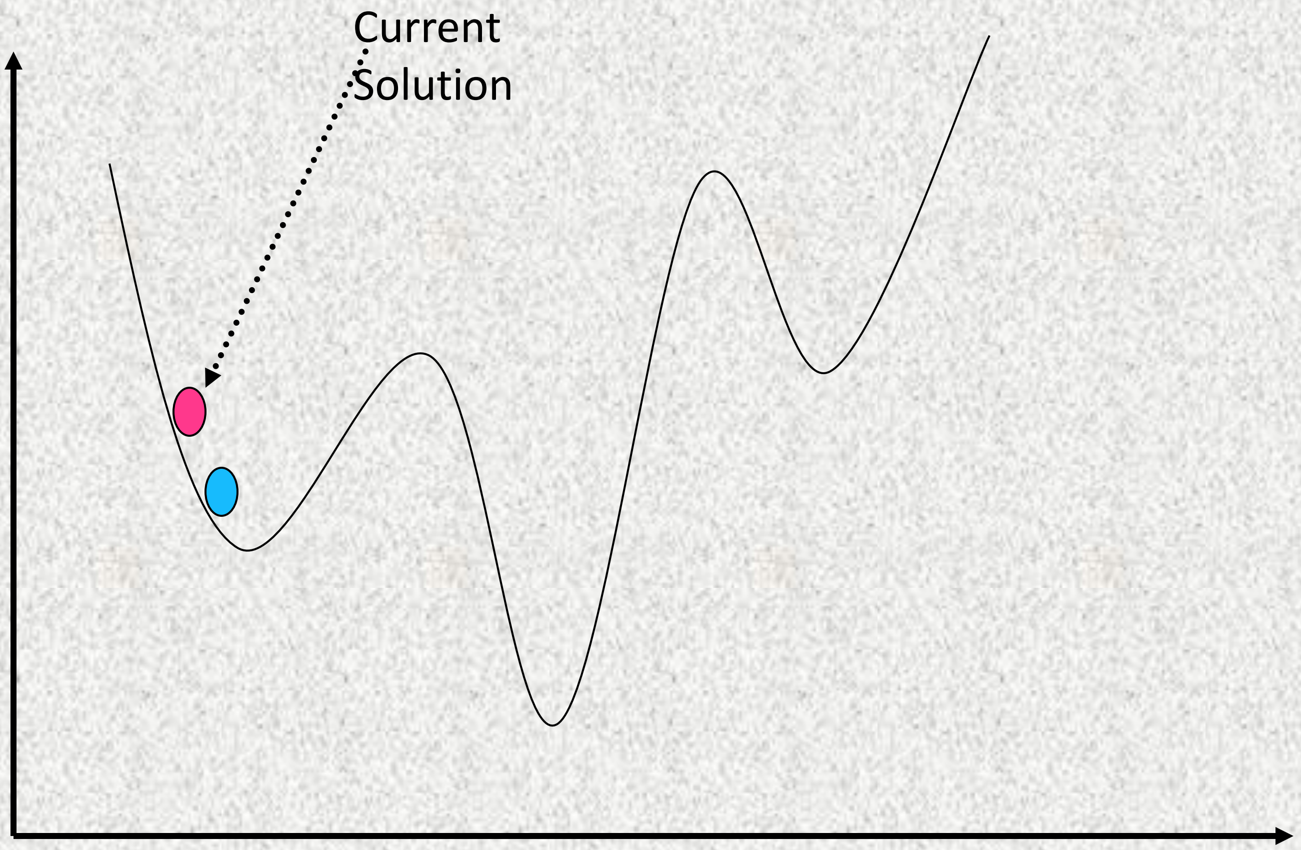
# Hill Climbing



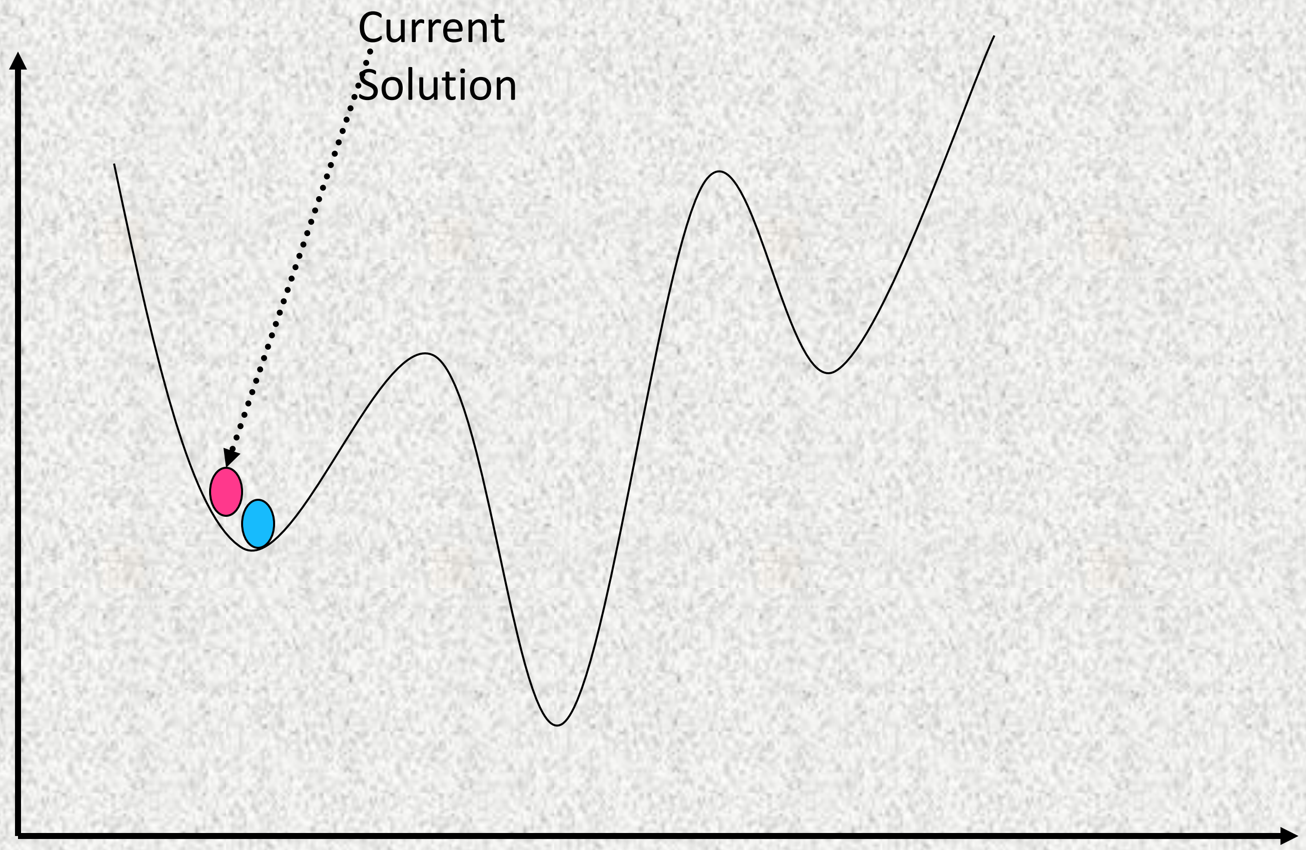
# Hill Climbing



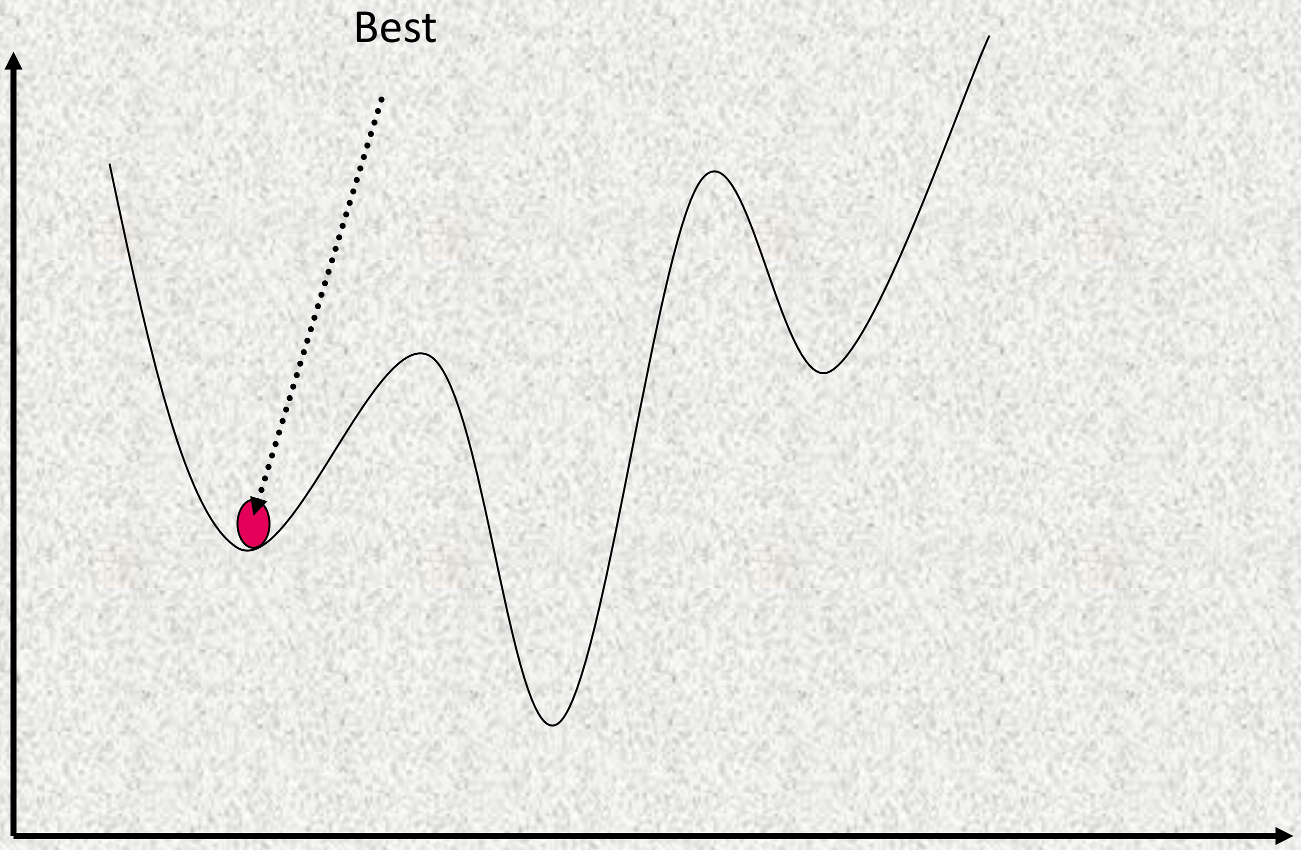
## Hill Climbing



# Hill Climbing



## Hill Climbing



# Simulated annealing

```
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 $e^{\Delta E/T}$
```

The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. Downhill moves are accepted readily early in the annealing schedule and then less often as time goes on. The schedule input determines the value of the temperature T as a function of time.

# Simulated Annealing

- A Hill-climbing algorithm that never makes “downhill" moves towards states with lower value can be incomplete.
- A **random walk**, i.e., moving to a successor chosen at random from the set of successors, is complete  but extremely inefficient.
- How can we combine both?
- This is a classical tradeoff between **exploration** of the search space and **exploitation** of the imperfect solution at hand. How do we resolve this tradeoff?

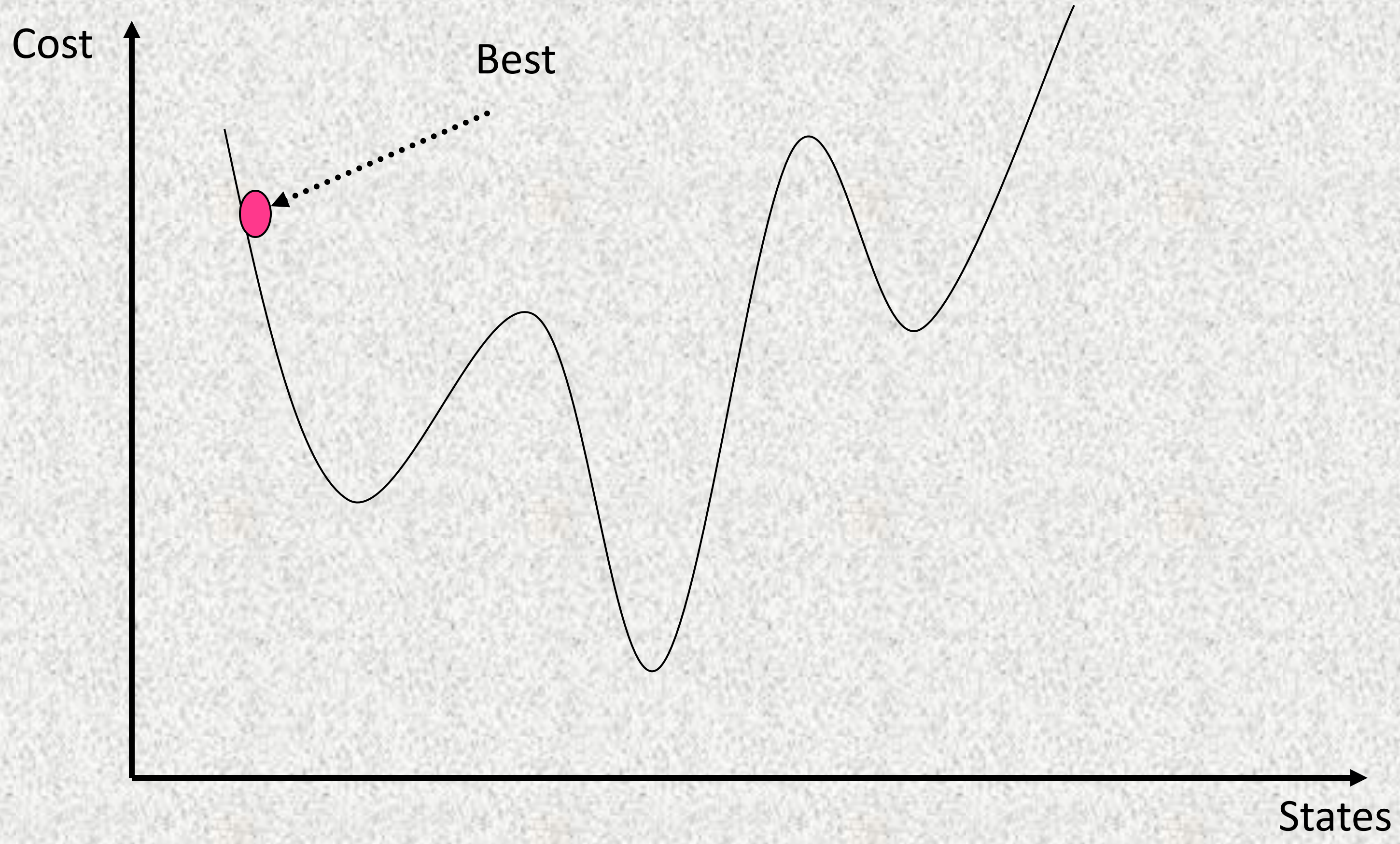
# Simulated Annealing

- Pure hill climbing is not complete, but pure random search is inefficient.
- Allows some apparently *"bad moves"*, in the hope of escaping local maxima
- Simulated annealing offers a compromise.
- Inspired by **annealing** process of gradually cooling a liquid until it changes to a low-energy state.
- Very similar to hill climbing, except include a user-defined **temperature schedule**.
- When temperature is "high", allow some random moves.
- When temperature "cools", reduce probability of random move.
- If T is decreased slowly enough, guaranteed to reach best state.
- Best solution ever found is always remembered

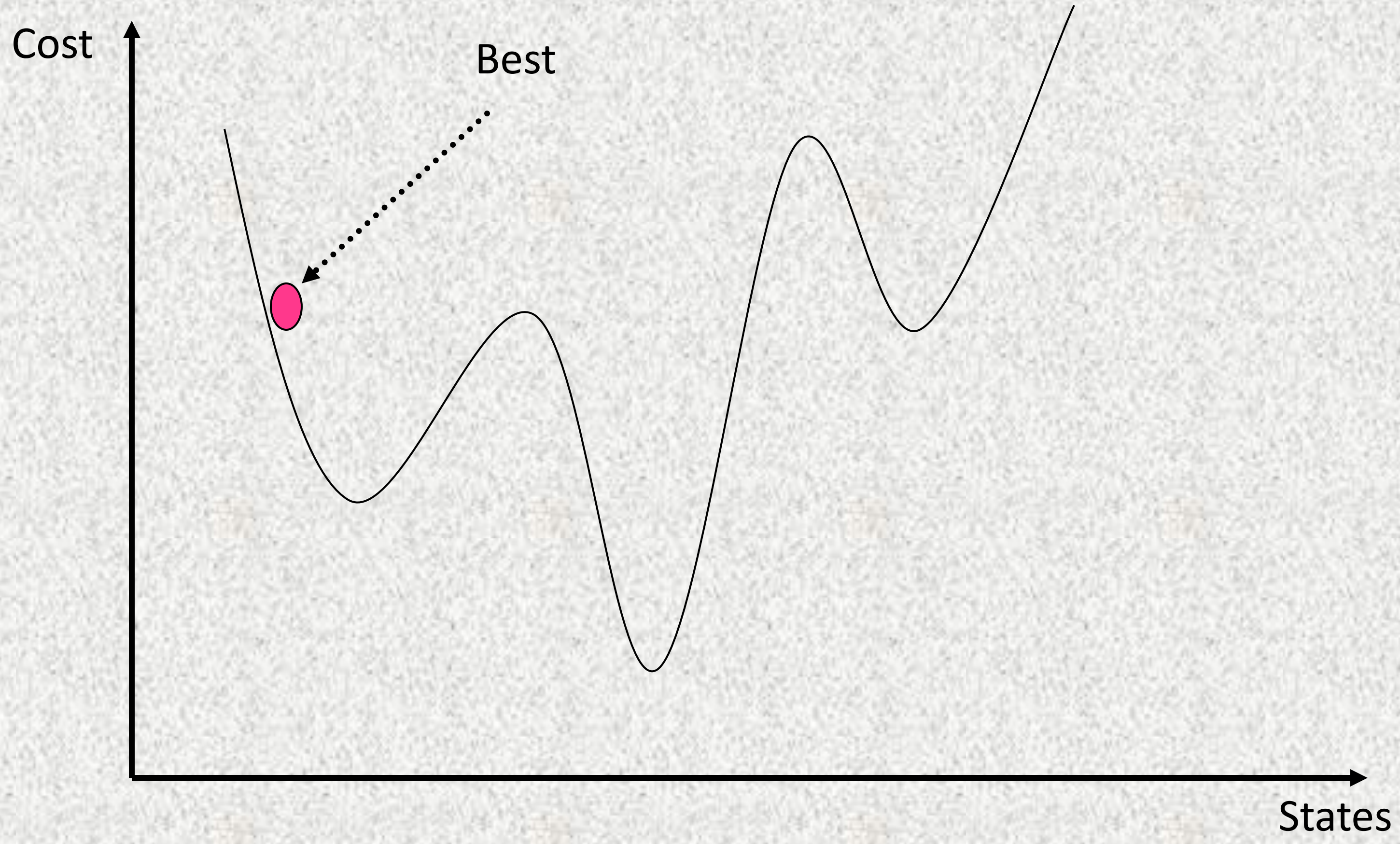
# Simulated Annealing

| **Physical System** | **Optimization Problem** |
|---|---|
| state | feasible solution |
| energy | evaluation function |
| ground state | optimal solution |
| quenching | local search |
| temperature | control parameter *T* |
| careful annealing | simulated annealing |

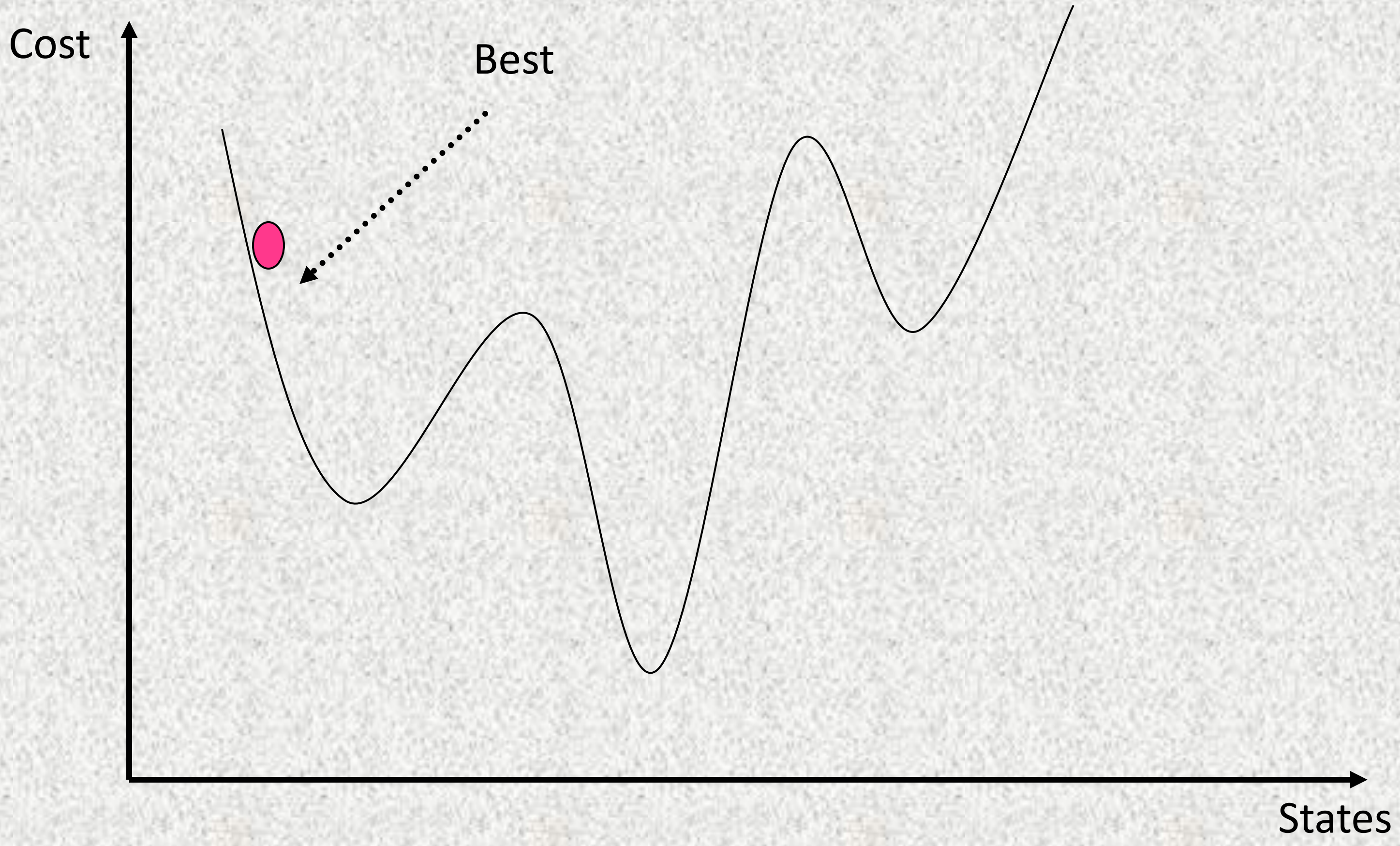
# Simulated Annealing



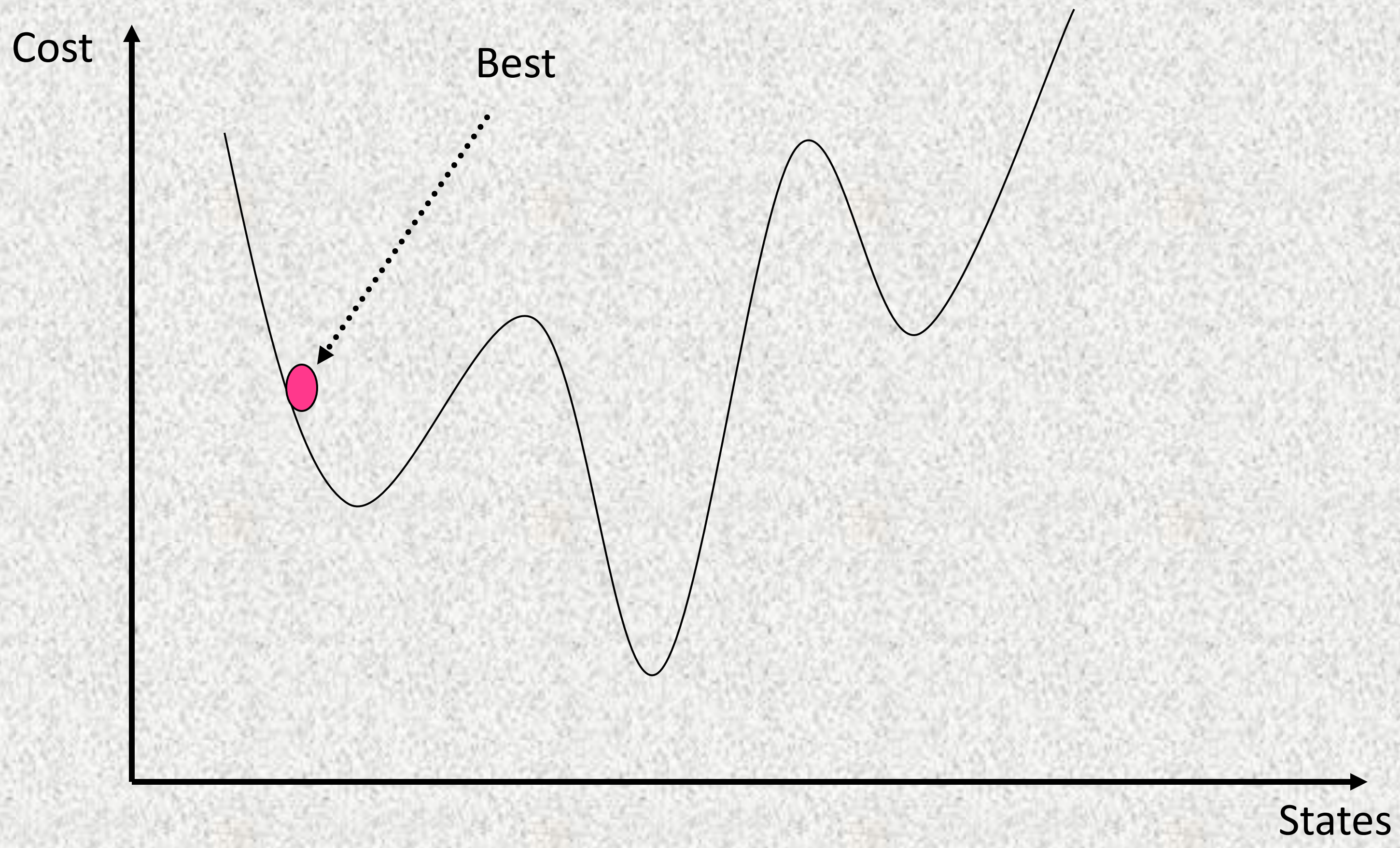
# Simulated Annealing



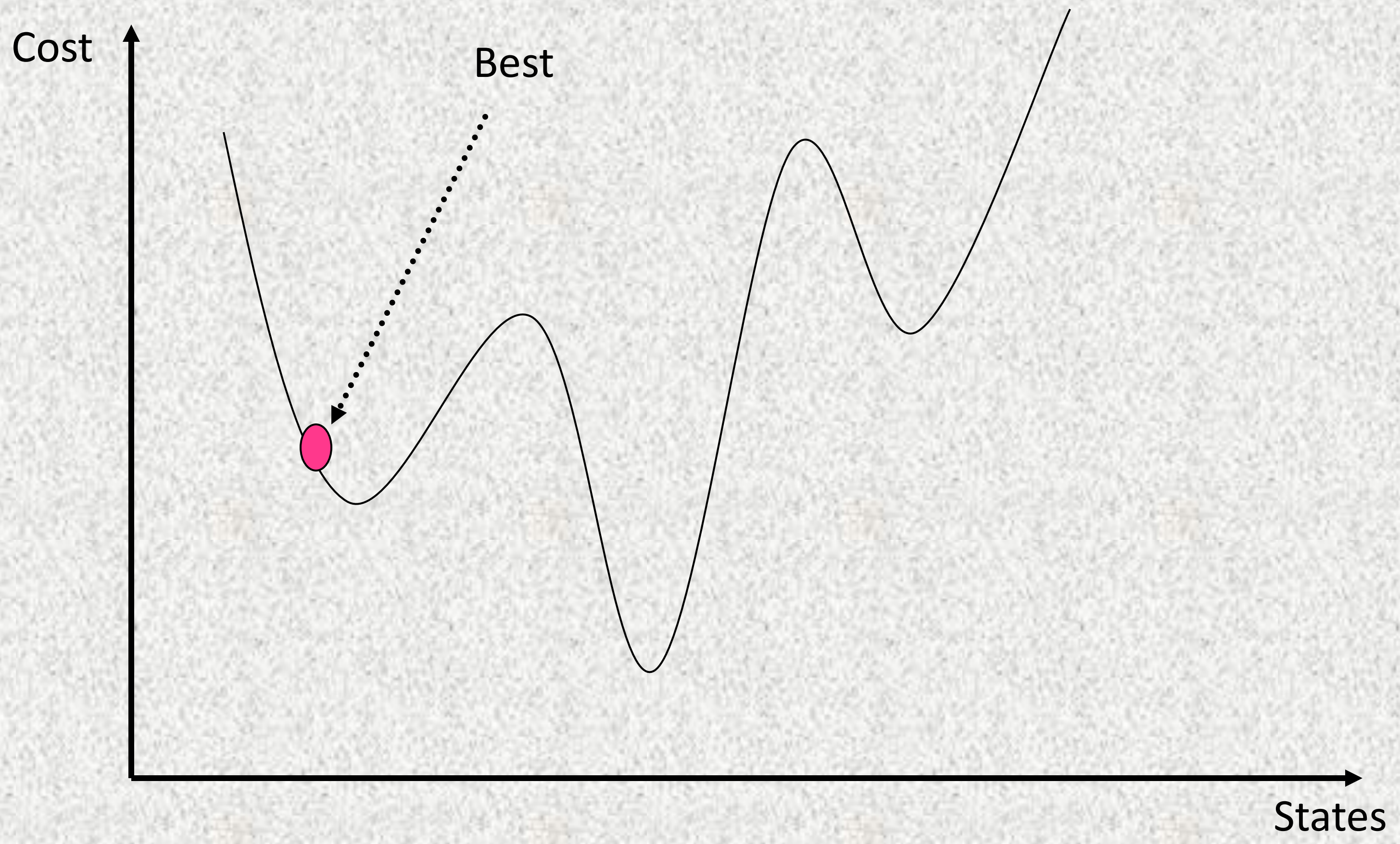
# Simulated Annealing



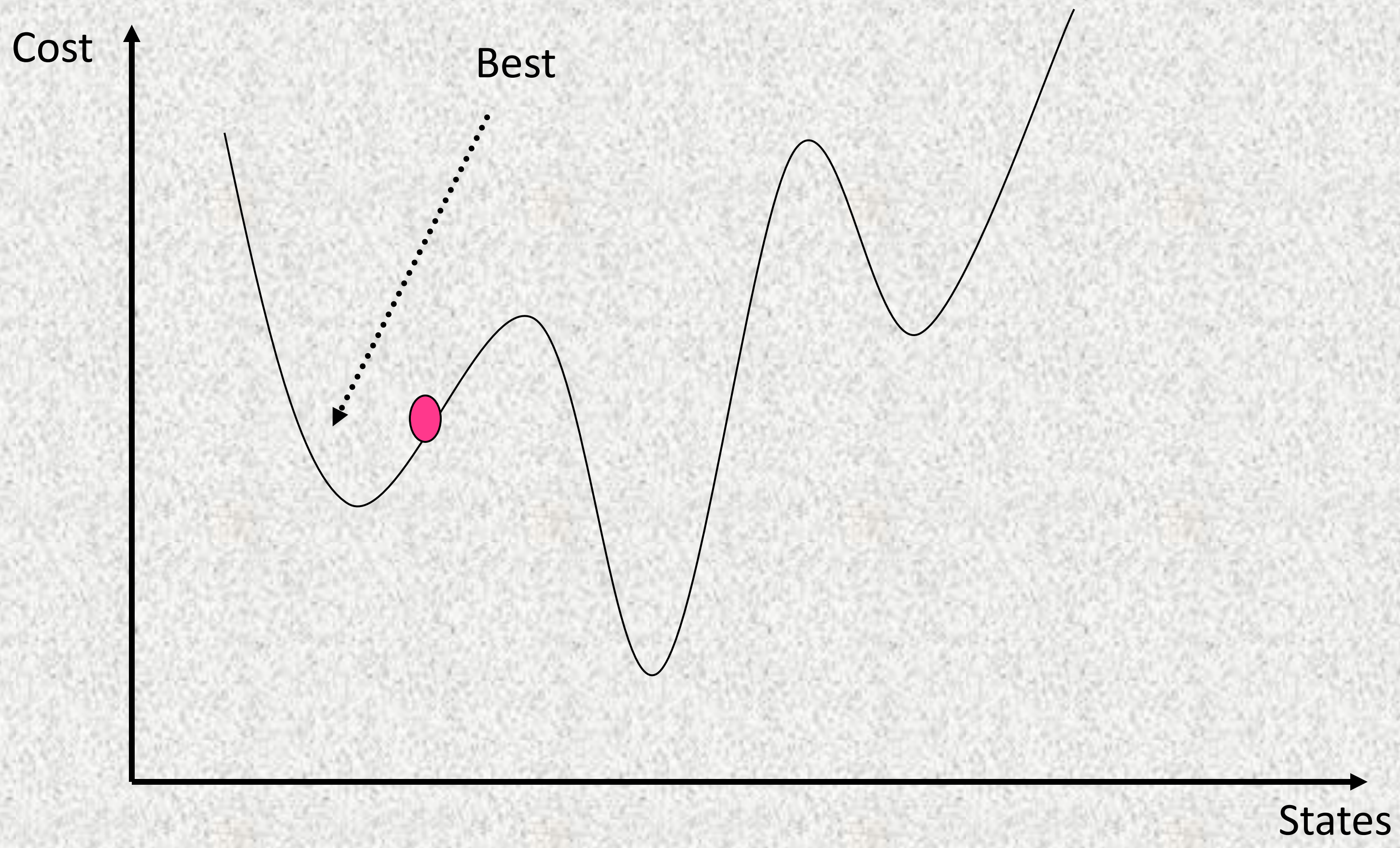
## Simulated Annealing



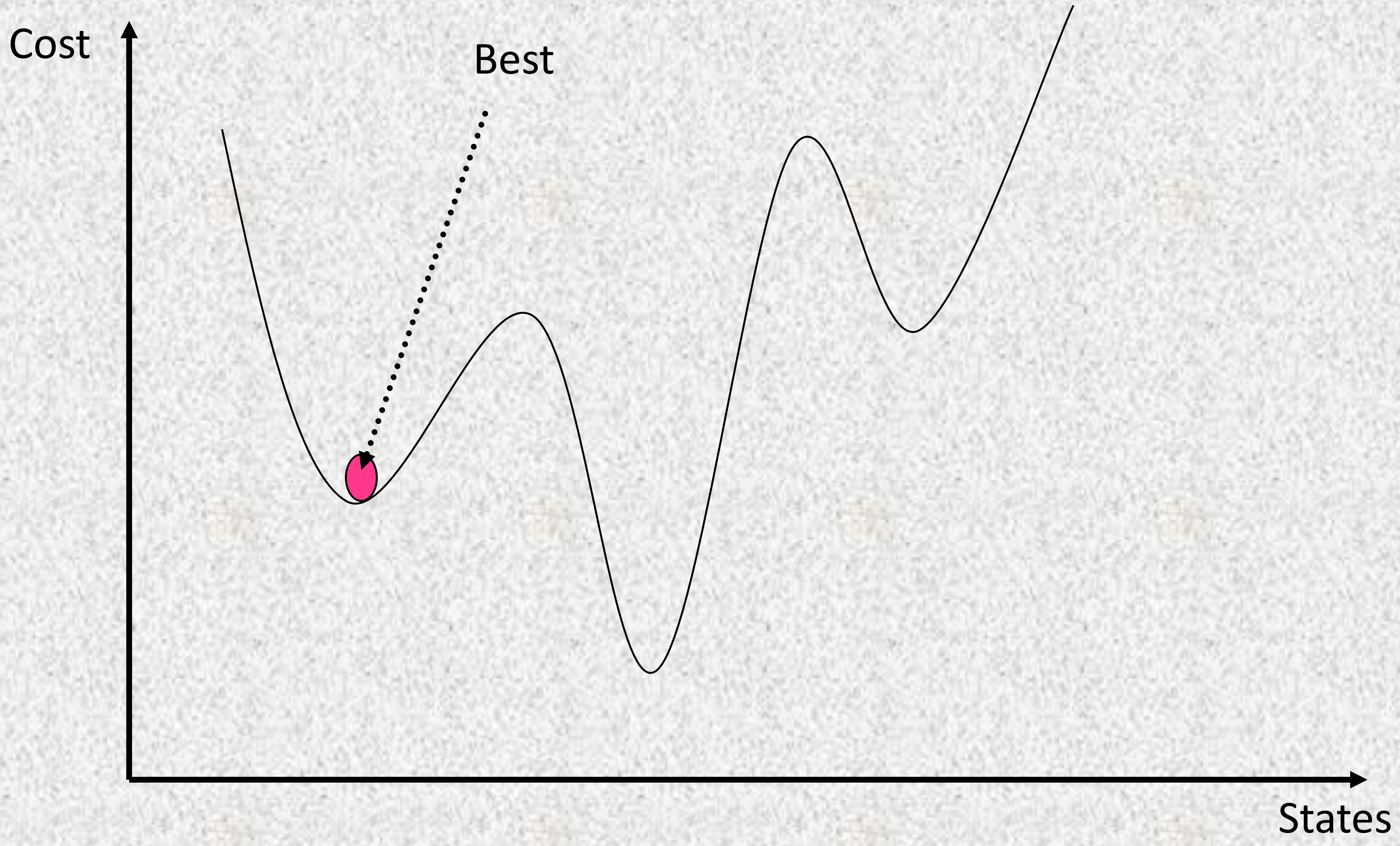
## Simulated Annealing



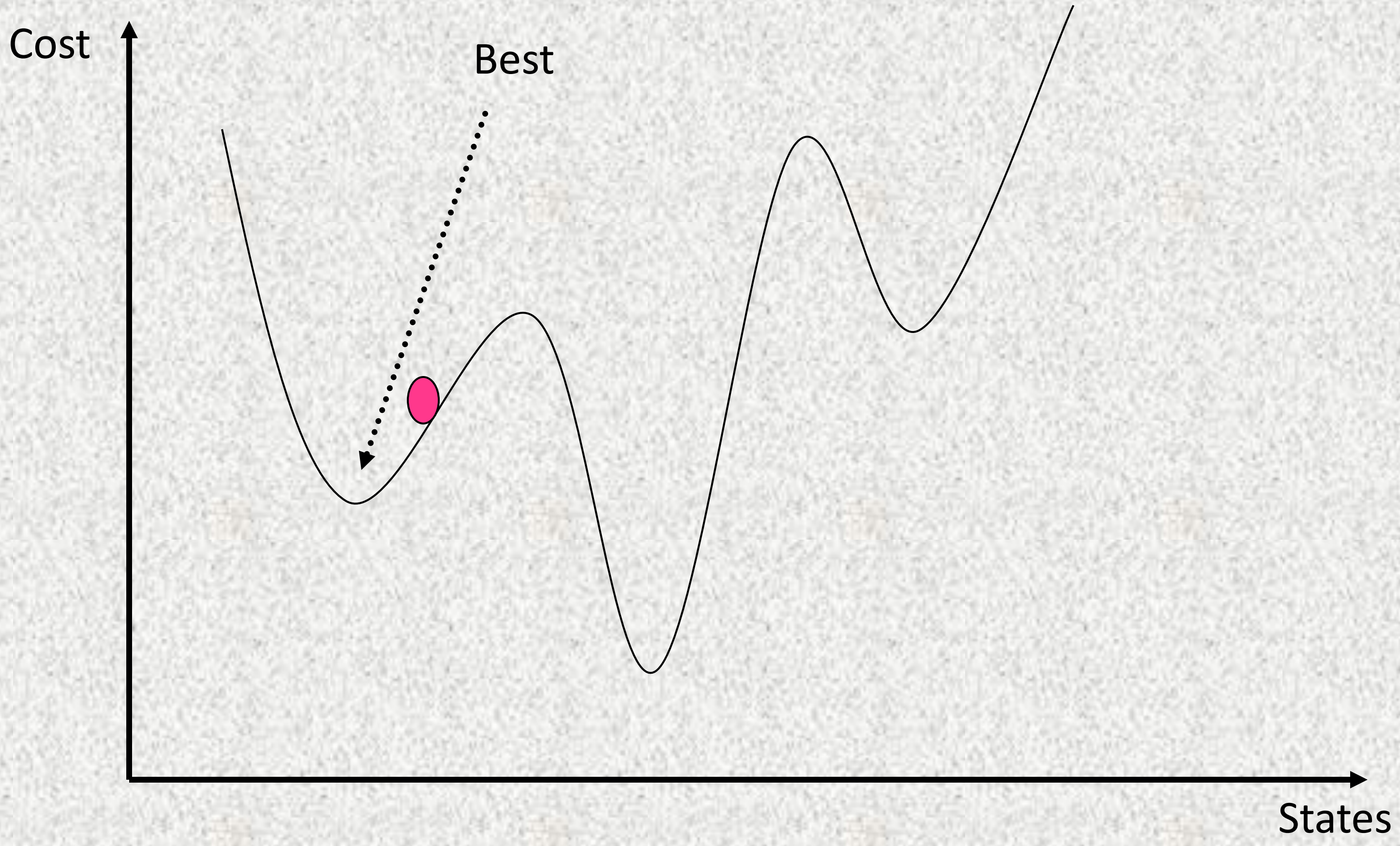
# Simulated Annealing



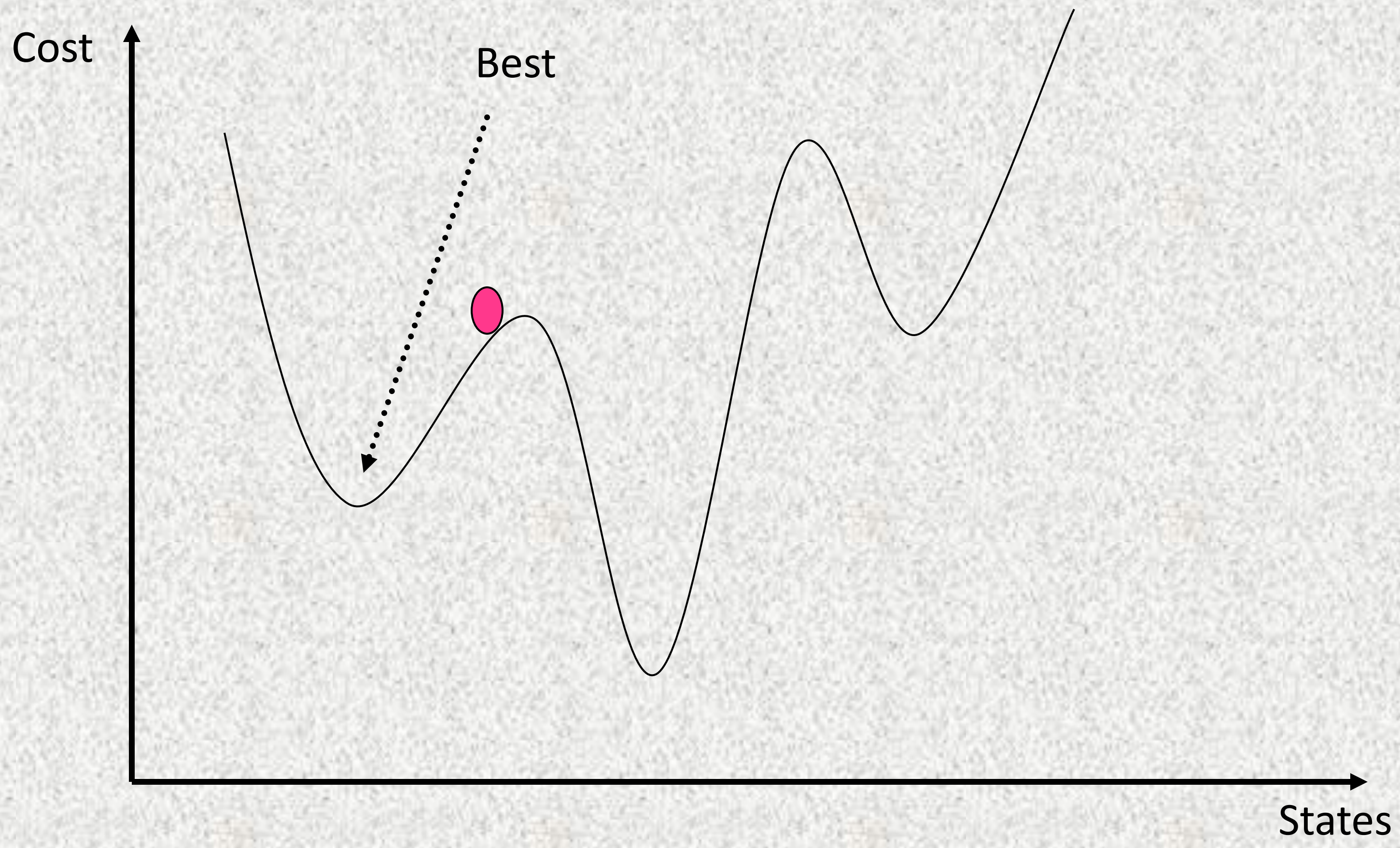
## Simulated Annealing



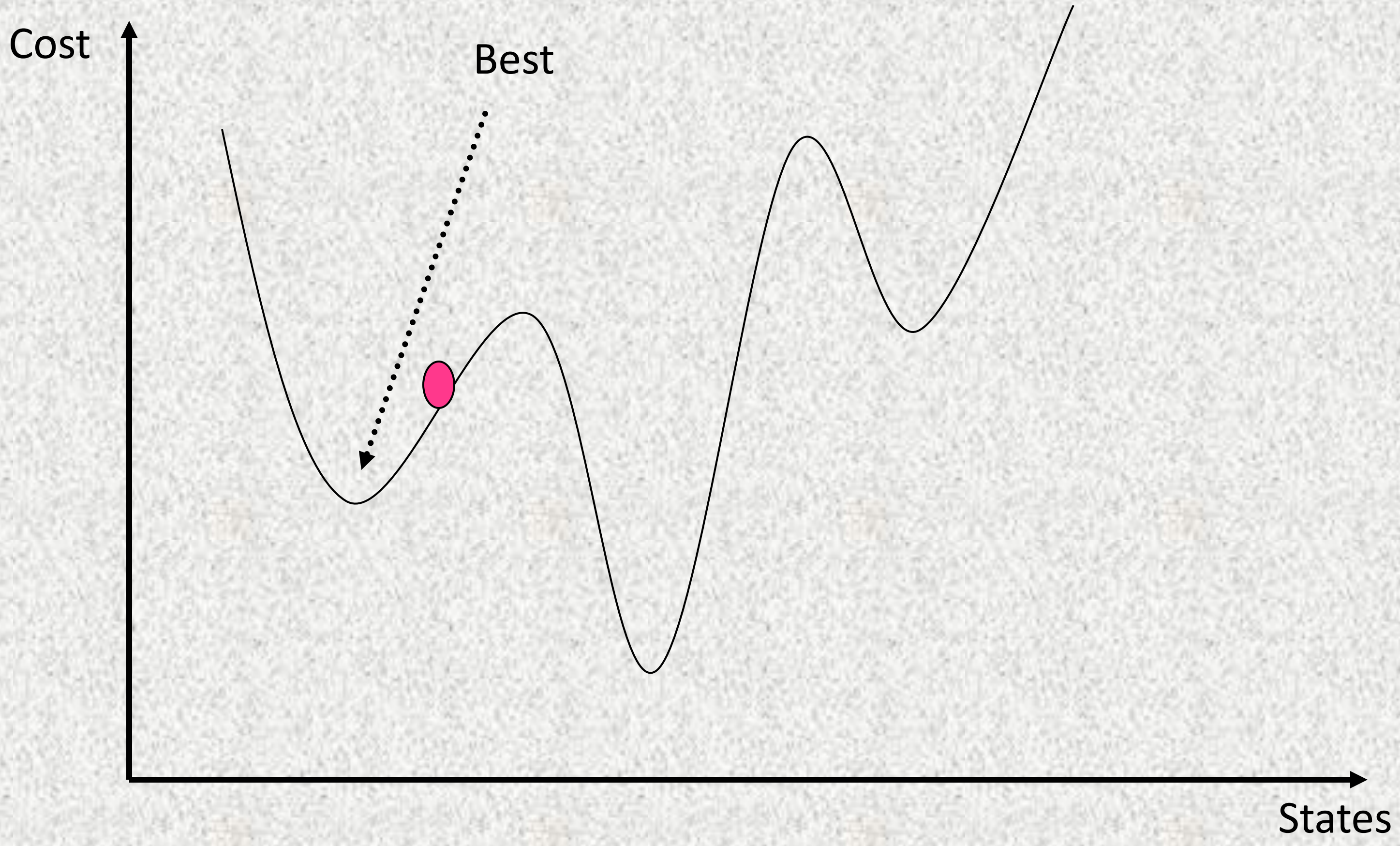
# Simulated Annealing



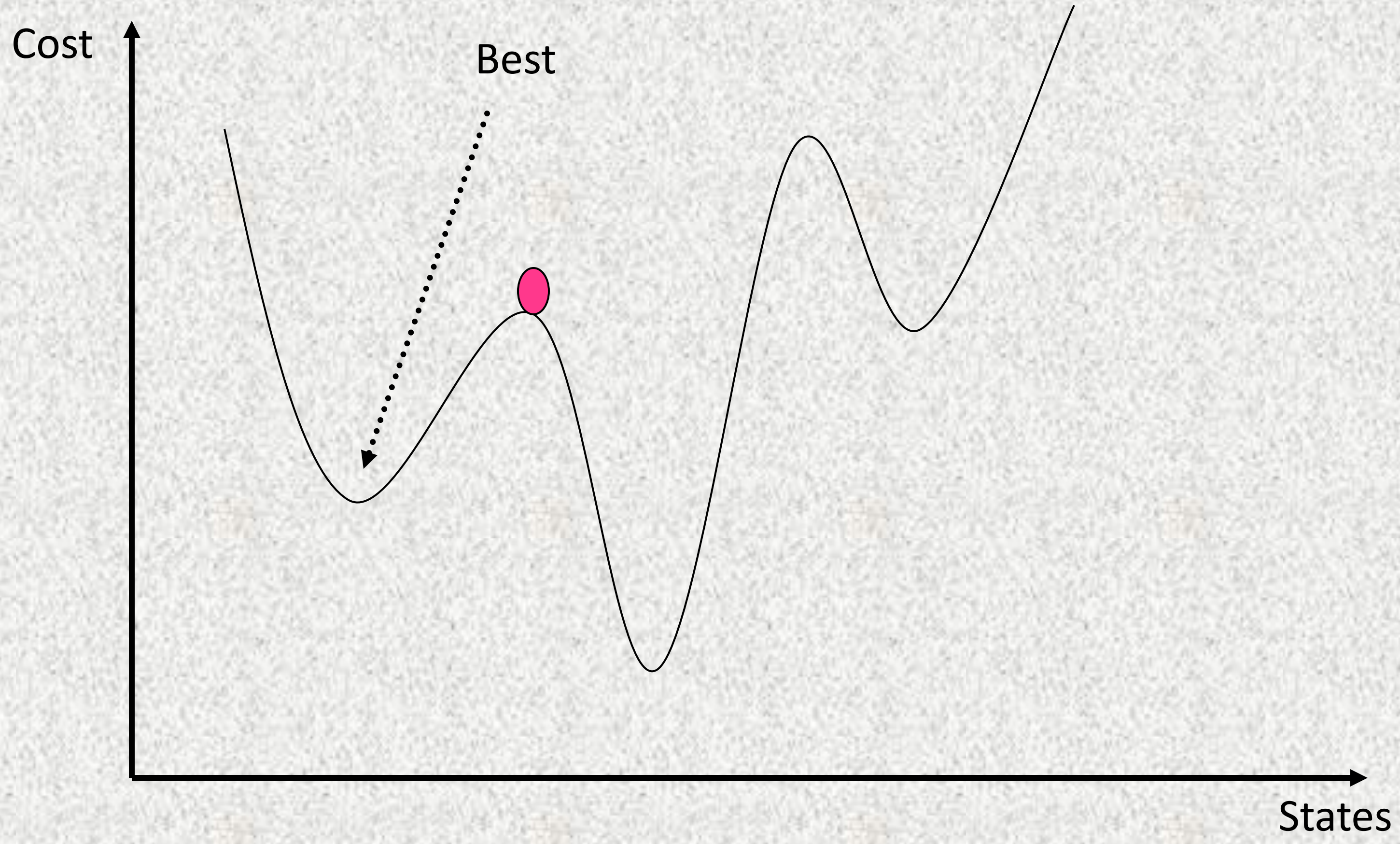
# Simulated Annealing



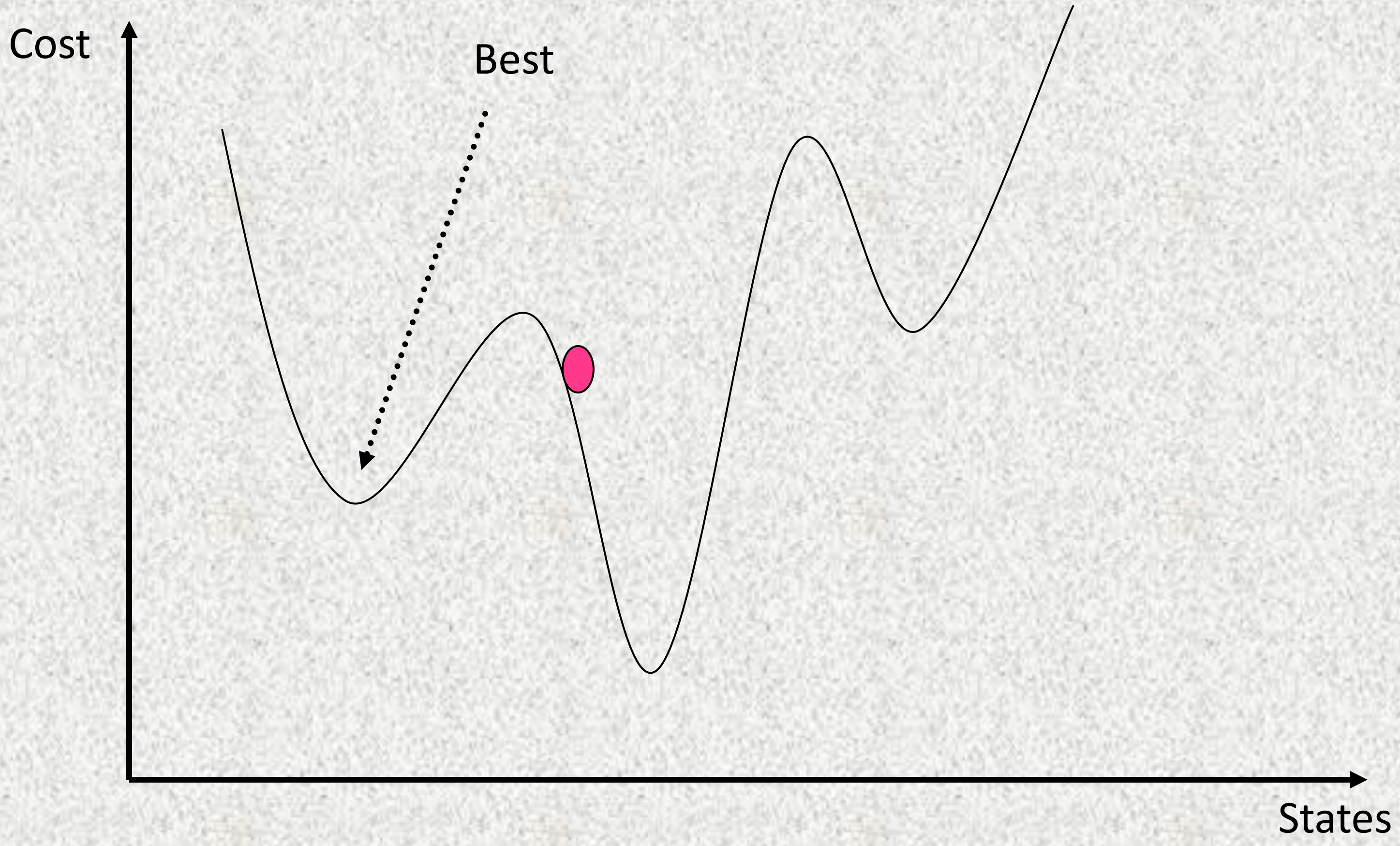
# Simulated Annealing



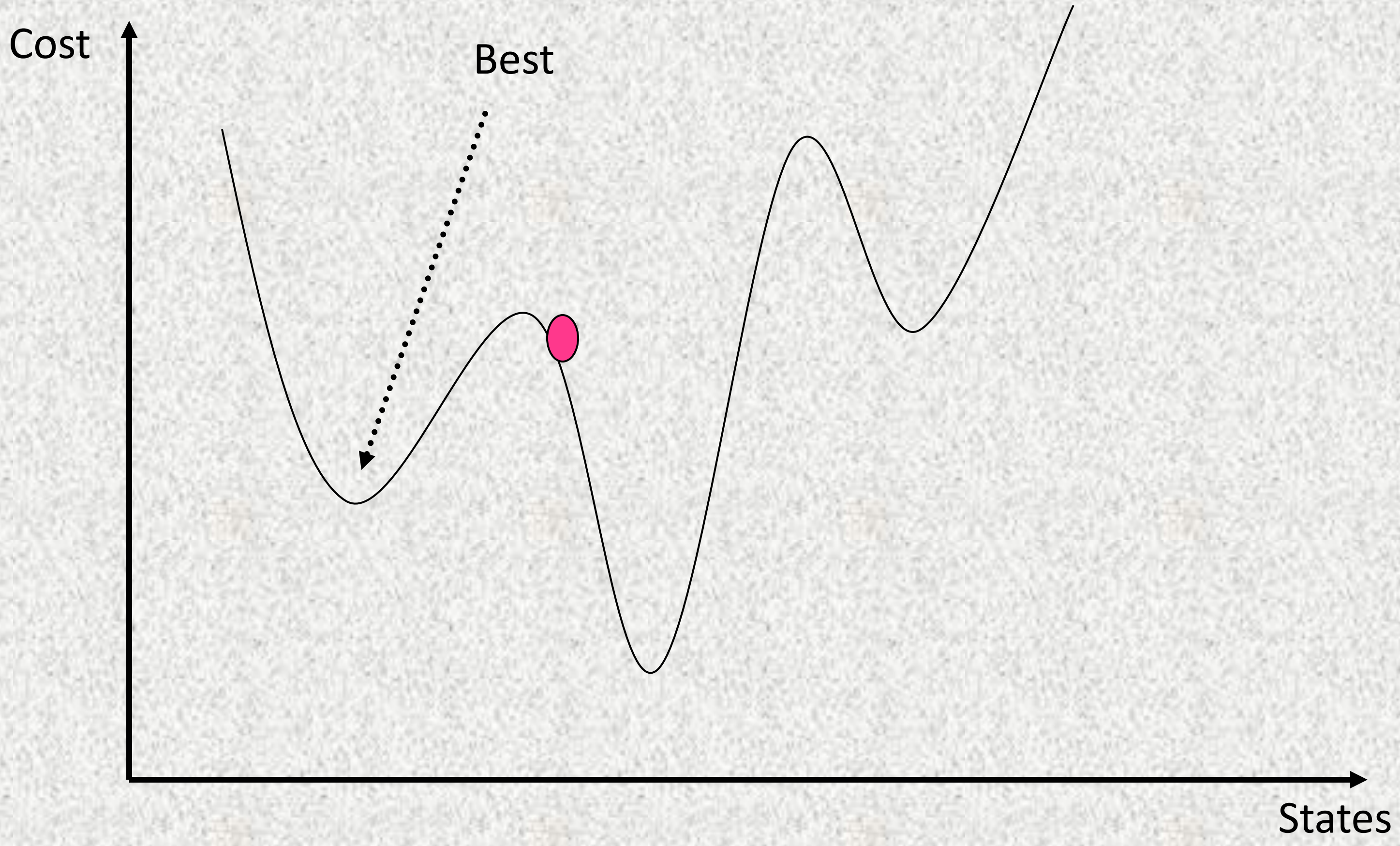
# Simulated Annealing



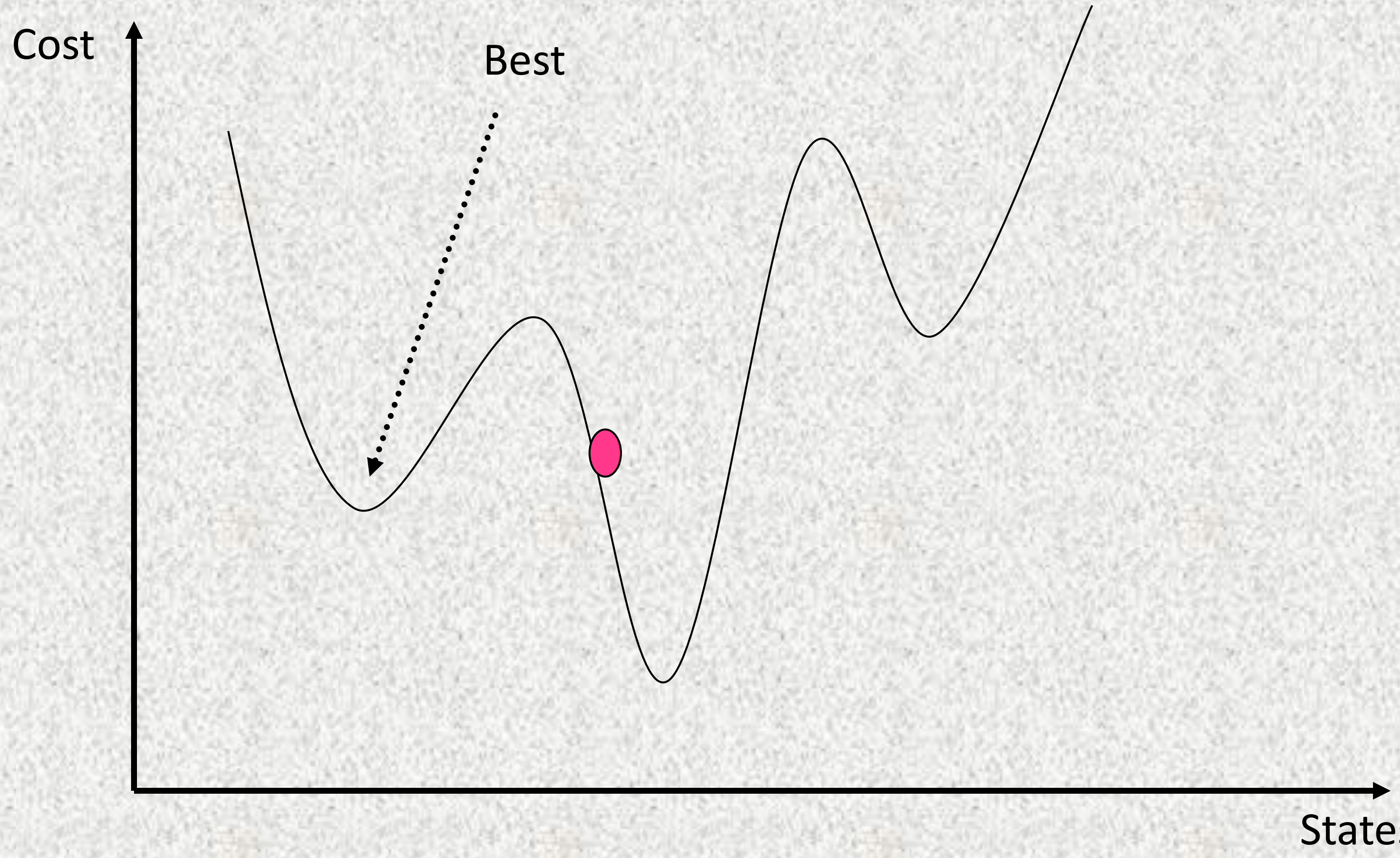
# Simulated Annealing



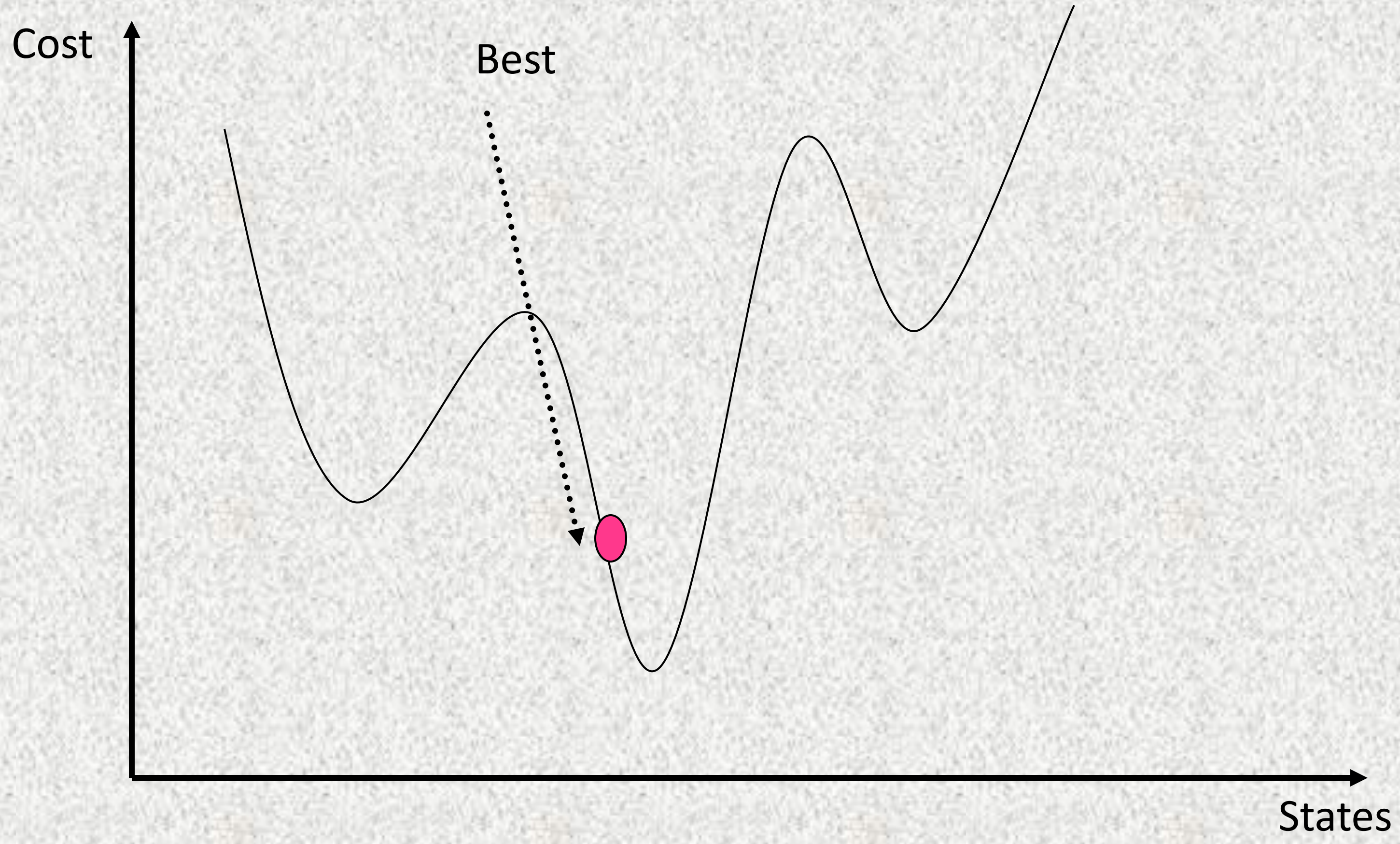
# Simulated Annealing



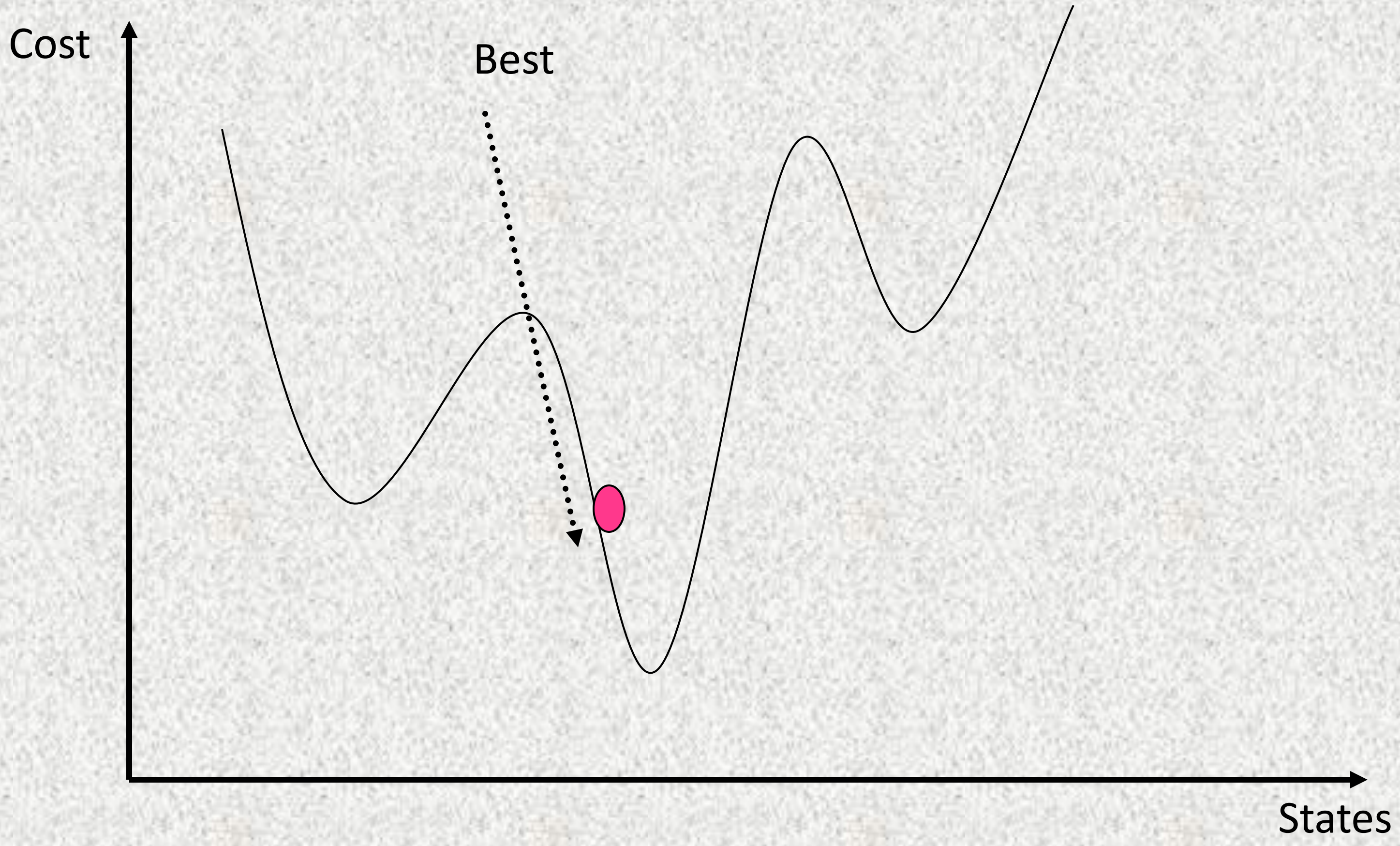
# Simulated Annealing



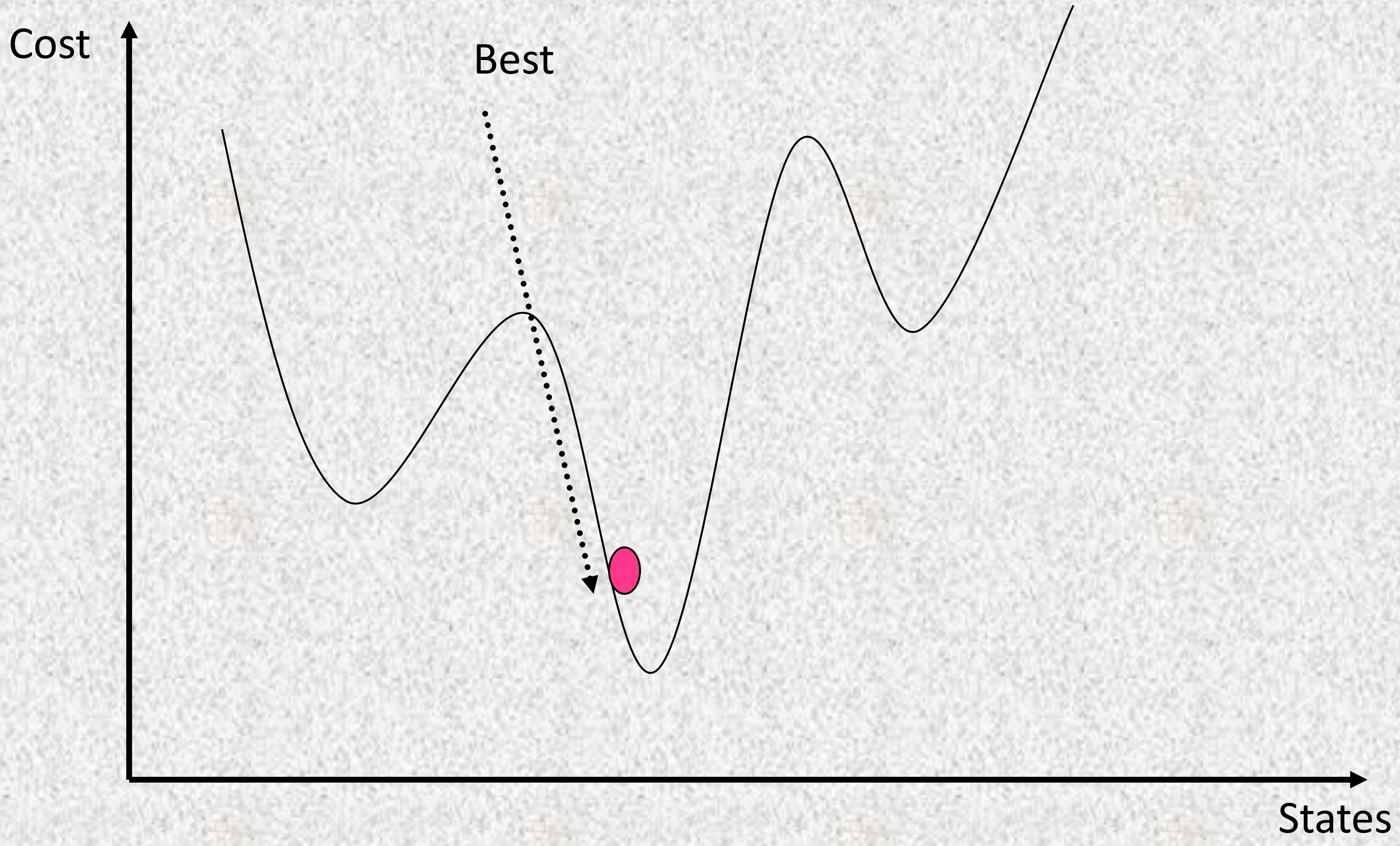
## Simulated Annealing



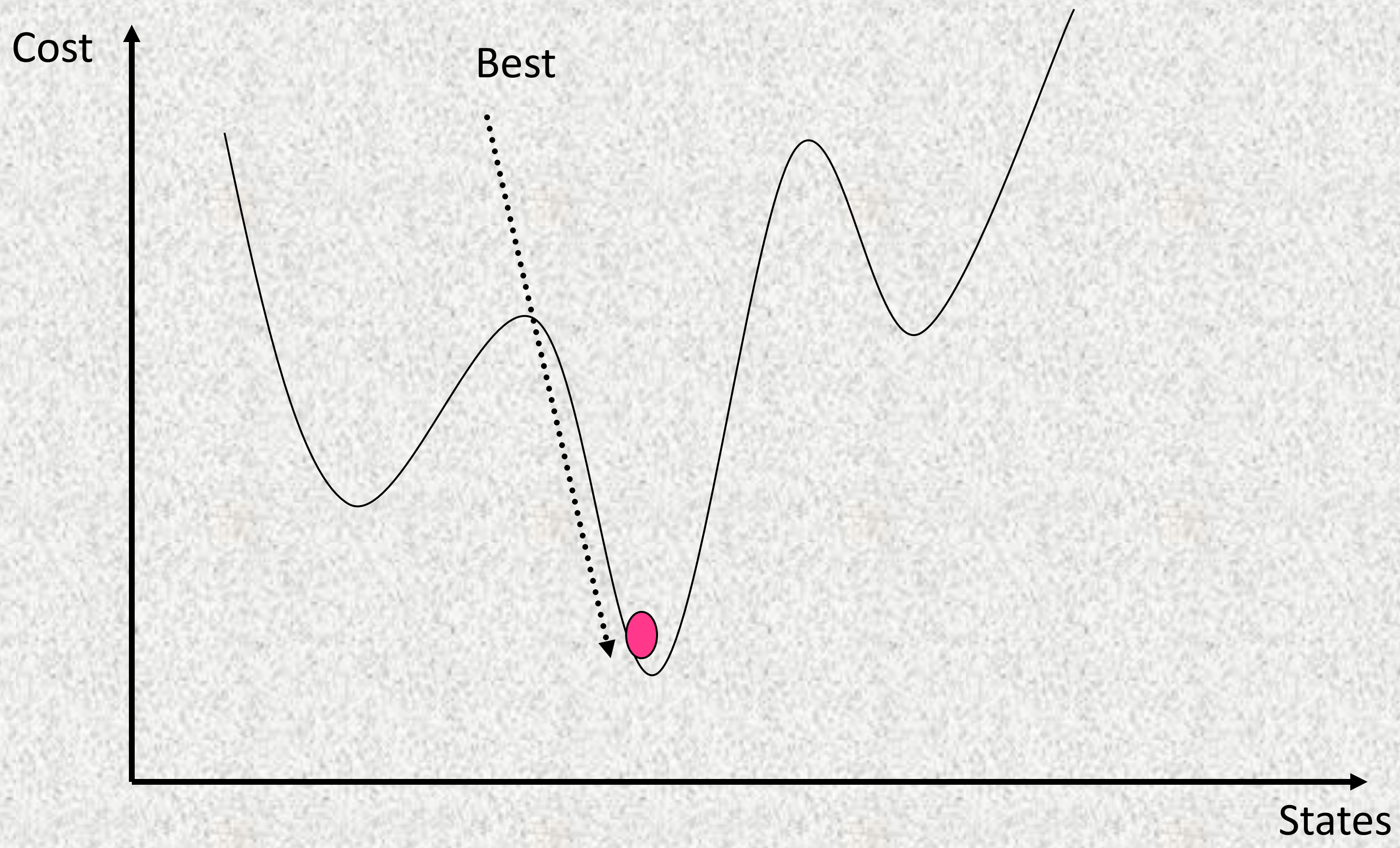
## Simulated Annealing



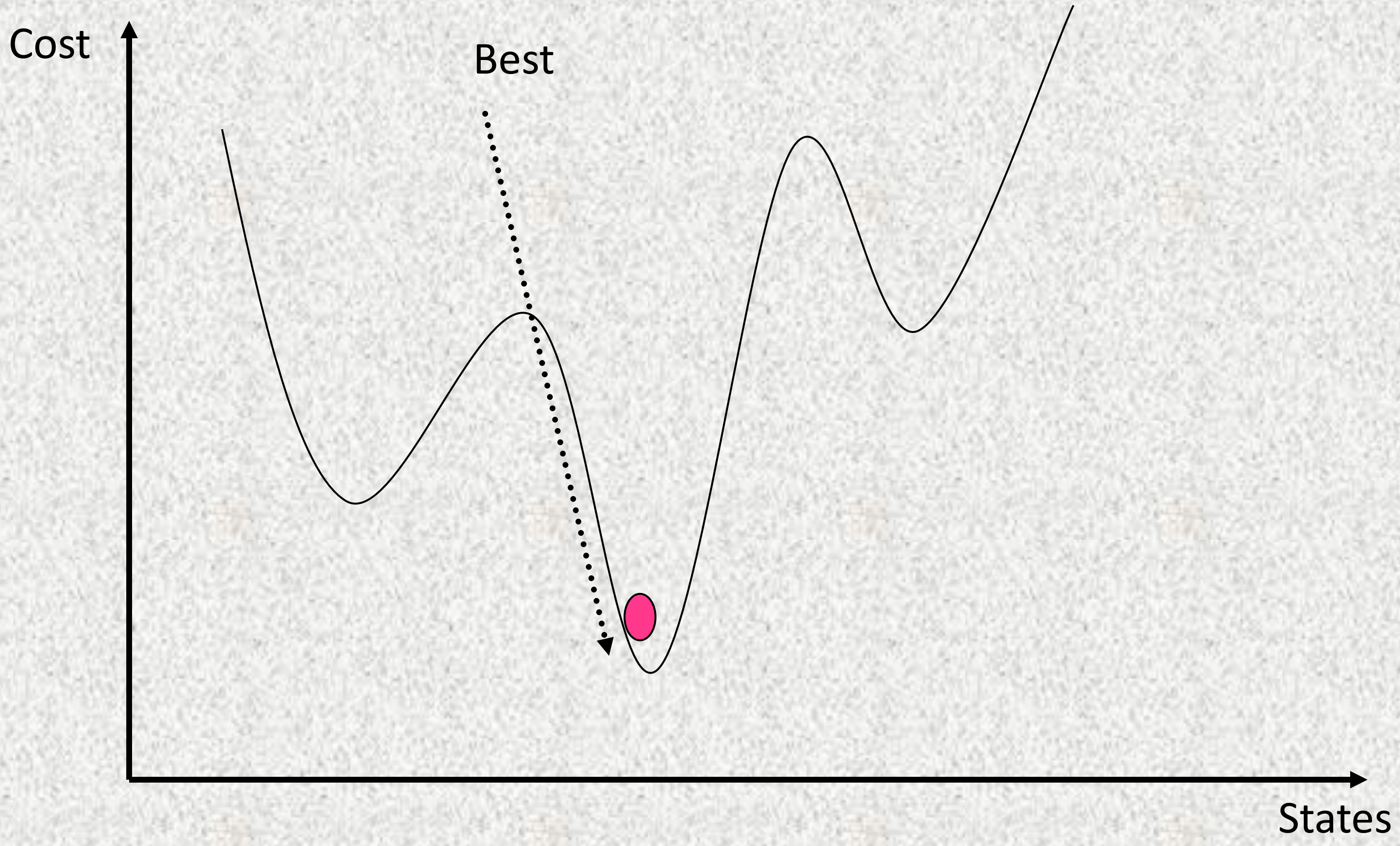
# Simulated Annealing



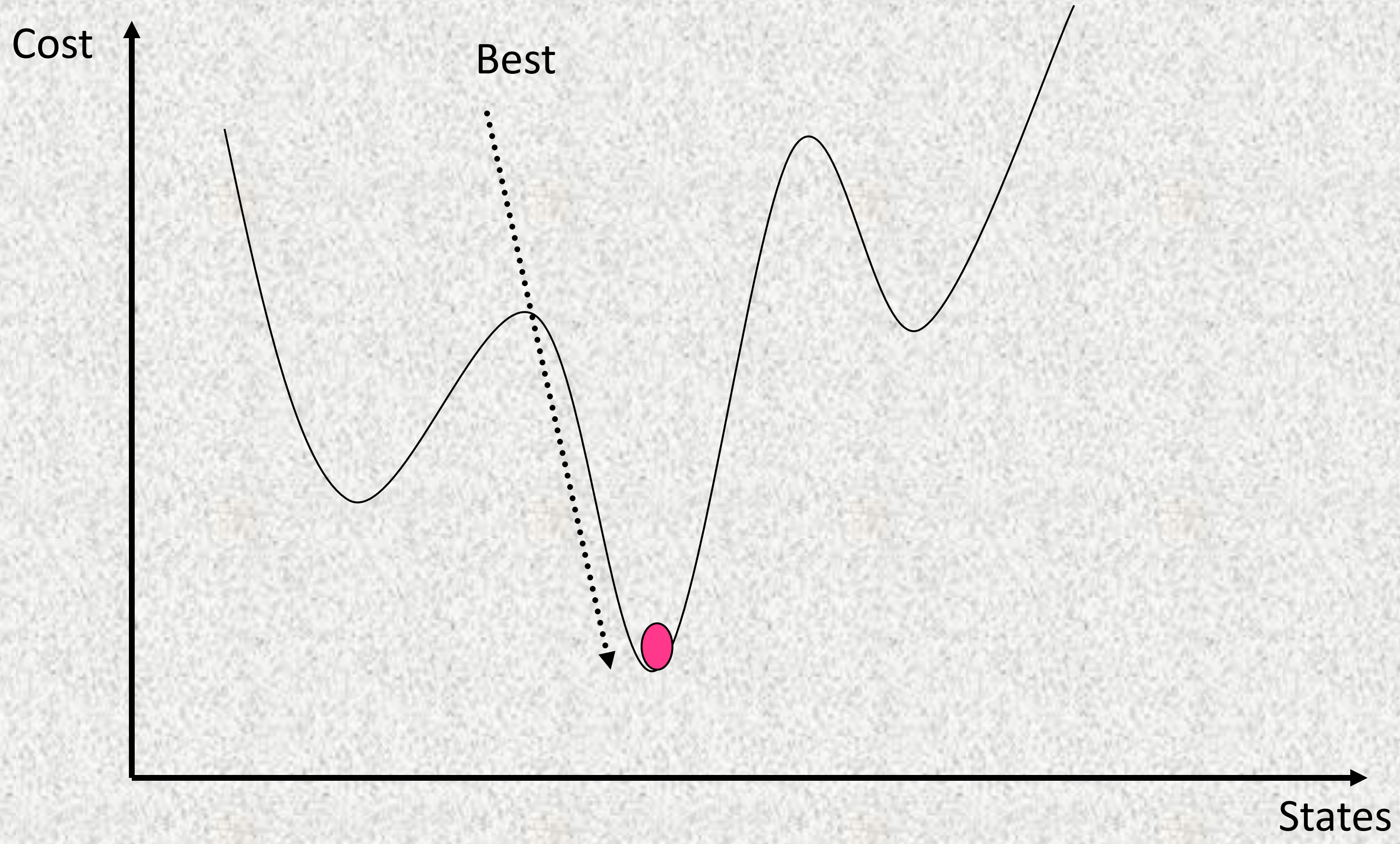
# Simulated Annealing



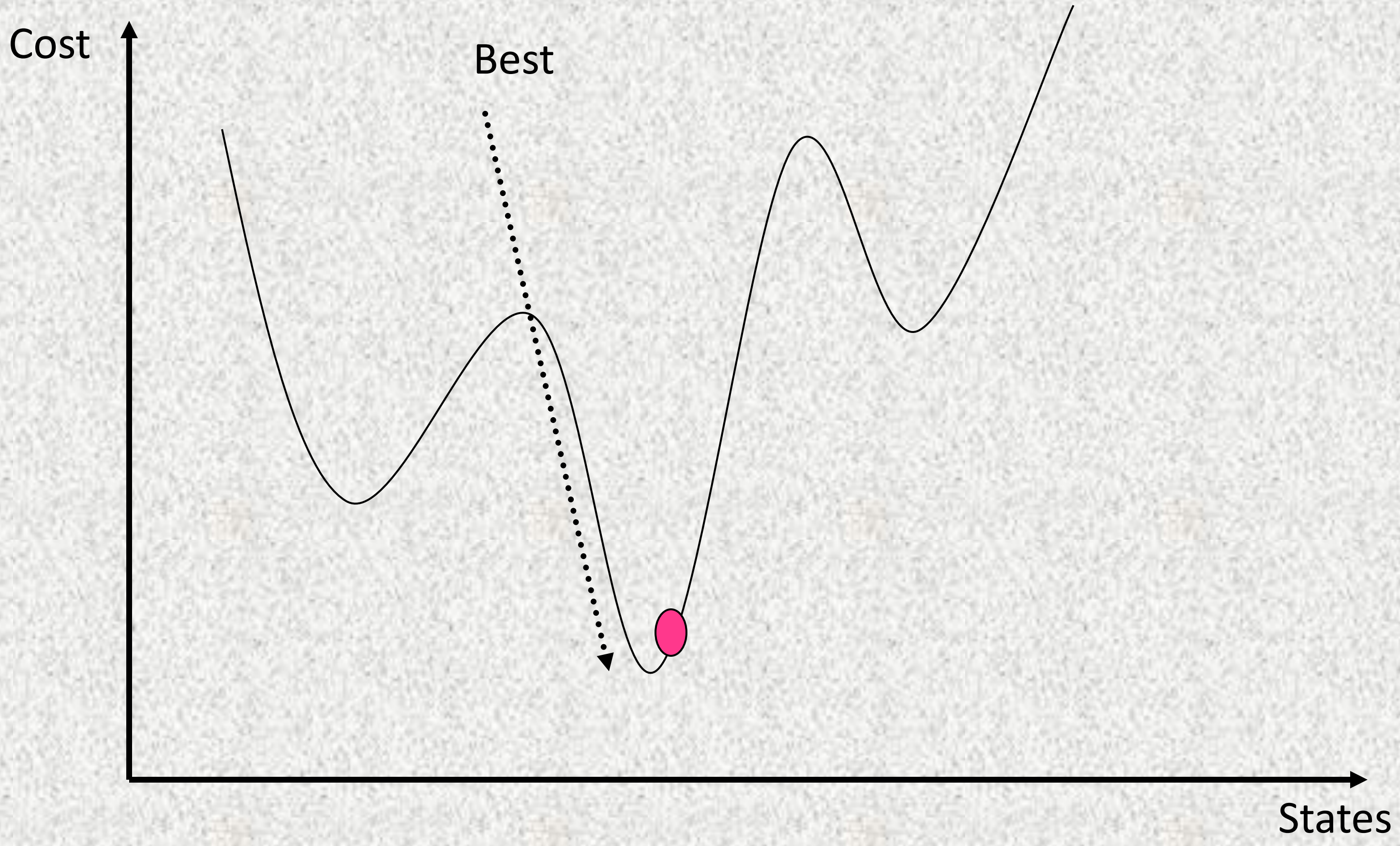
# Simulated Annealing



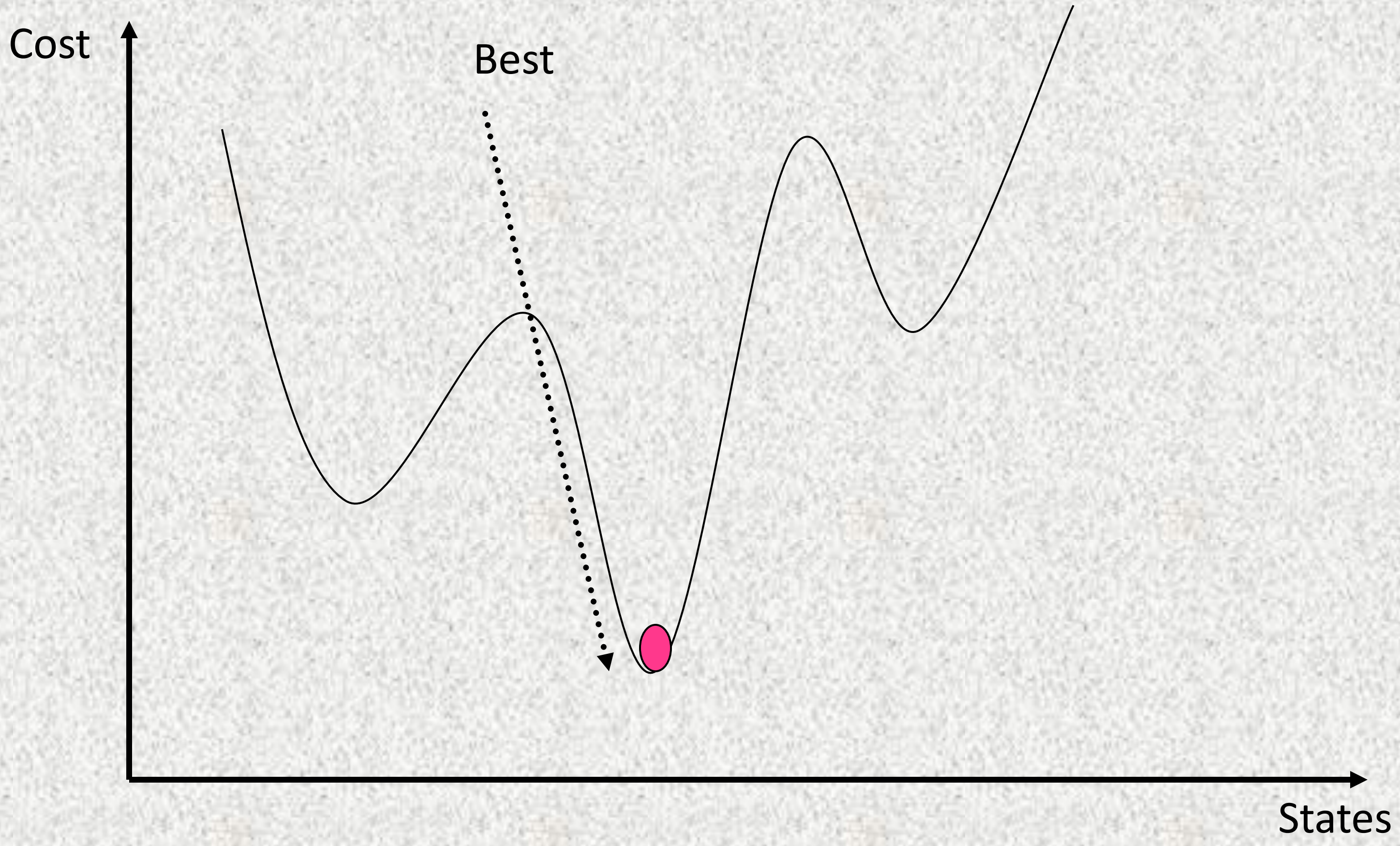
# Simulated Annealing



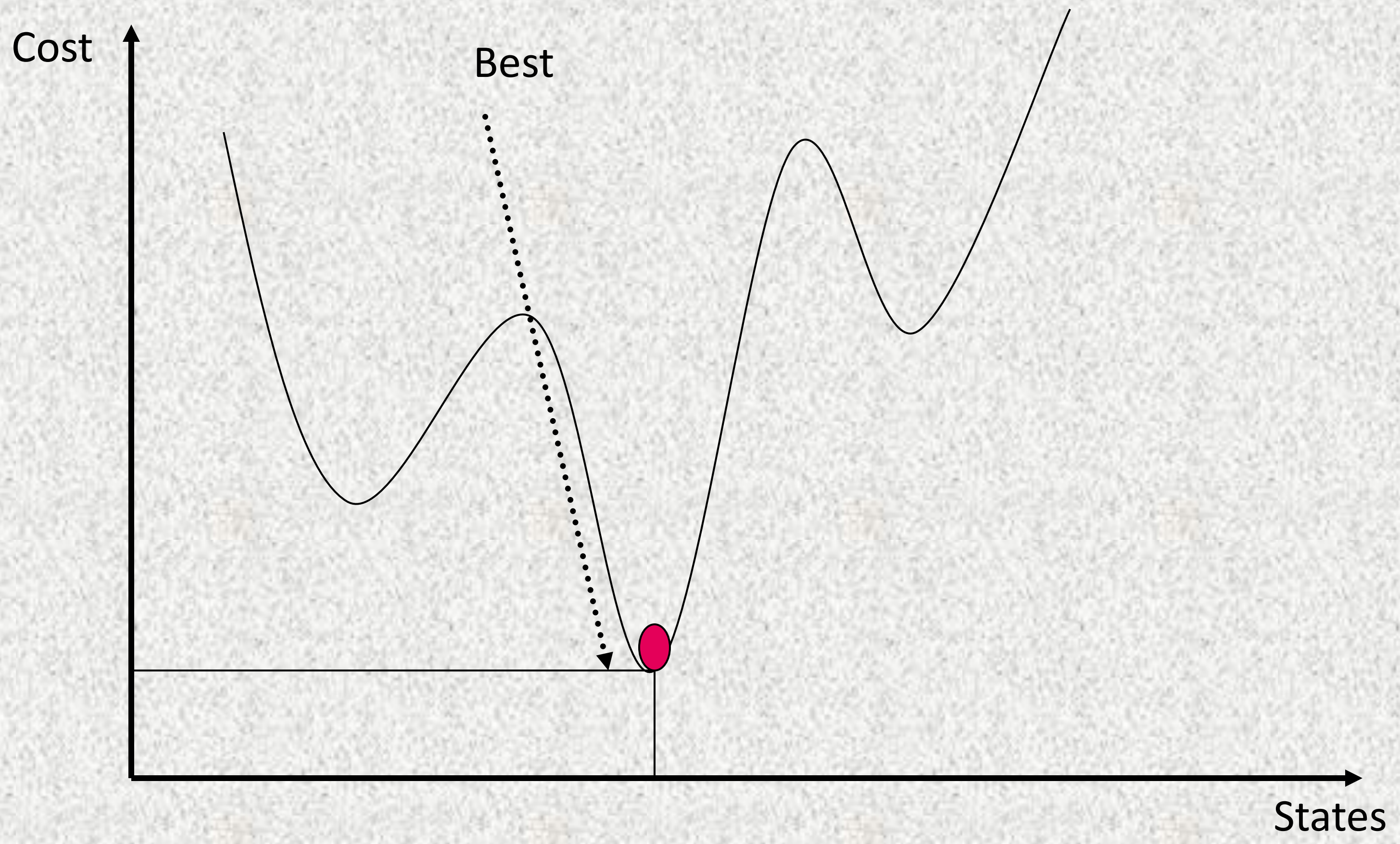
# Simulated Annealing



# Simulated Annealing



## Simulated Annealing



## Local beam search

Idea: keep *k* states instead of 1; choose top *k* of all their successors

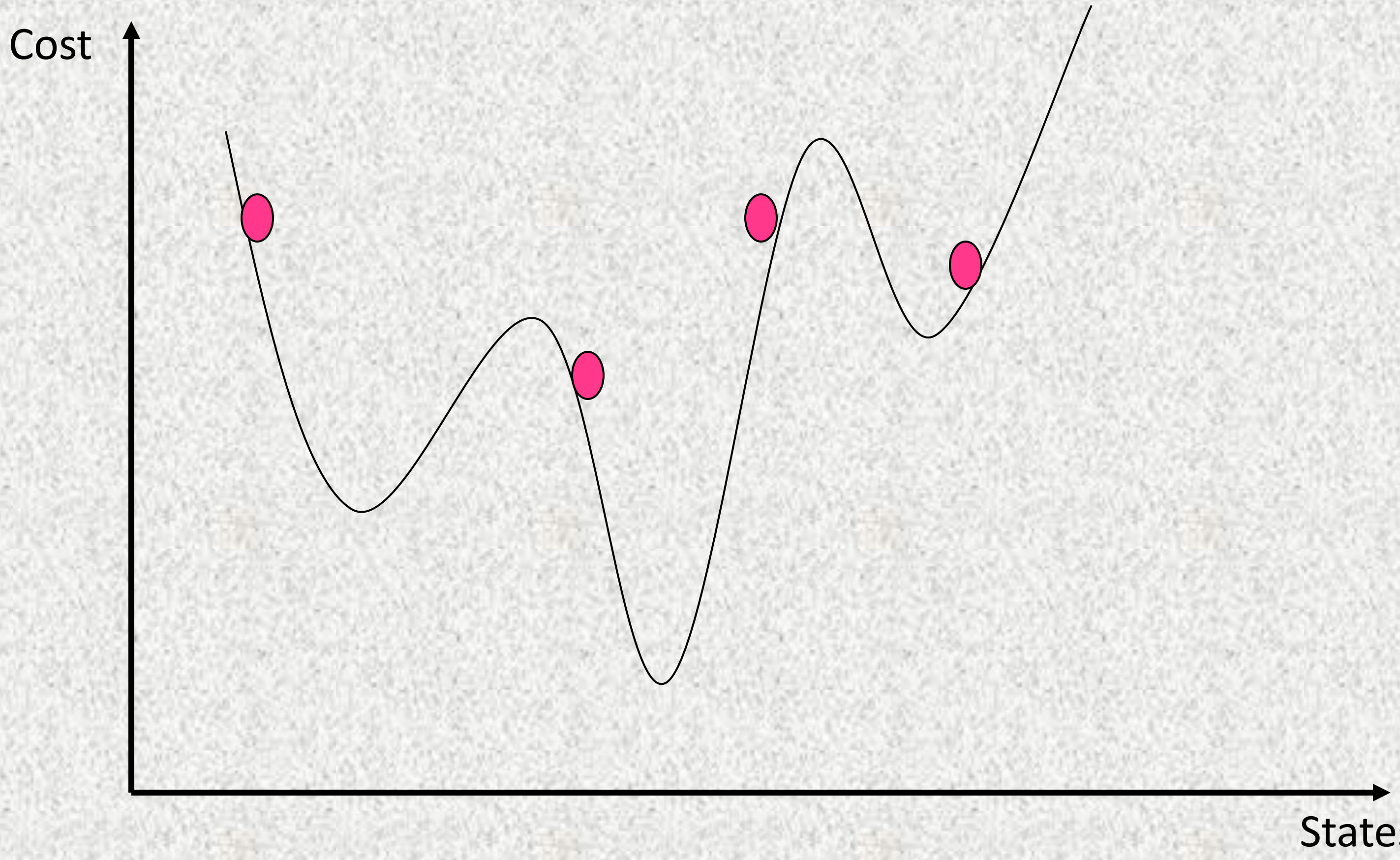
Not the same as *k* searches run in parallel! Searches that find good states recruit other searches to join them.

Problem: quite often, all *k* states end up on same local hill.

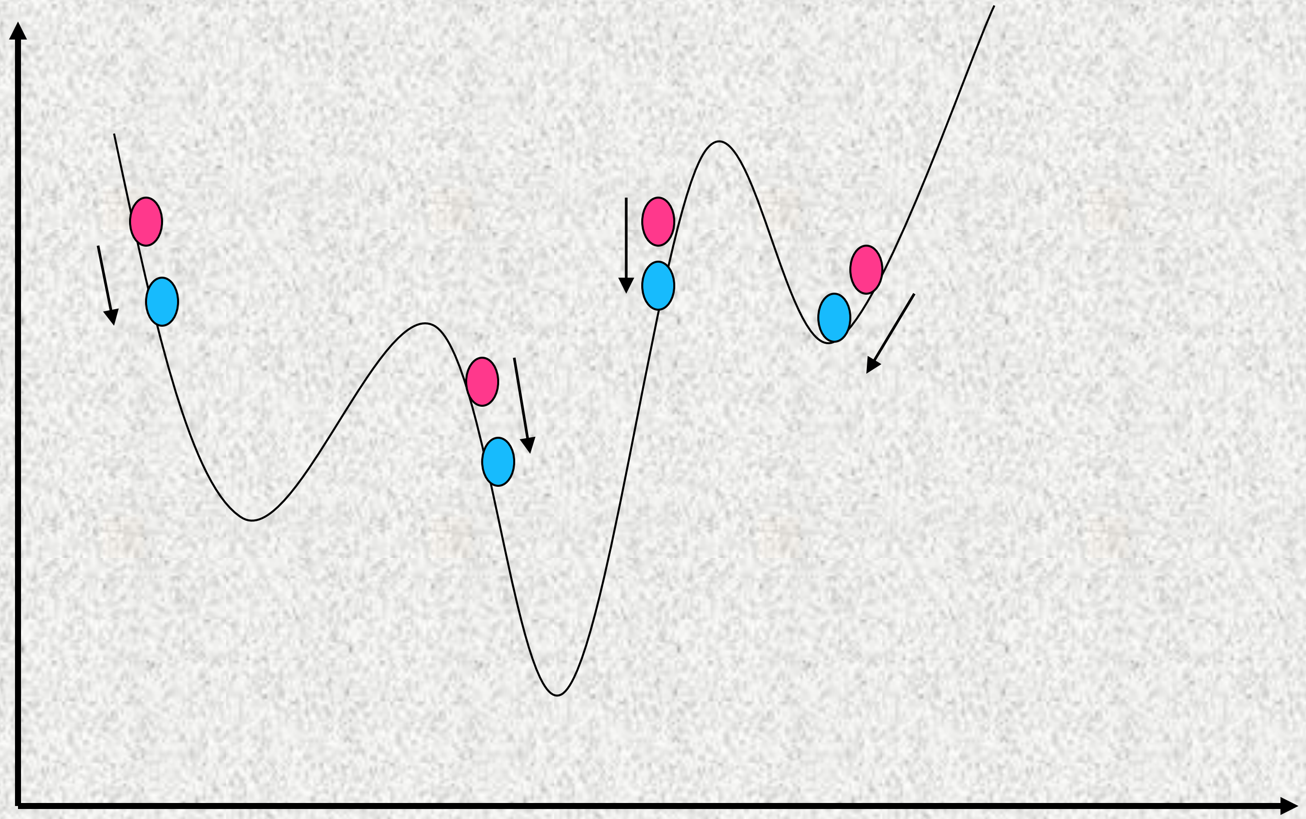
To improve: choose *k* successors randomly, biased towards good ones.

Observe the close analogy to natural selection!

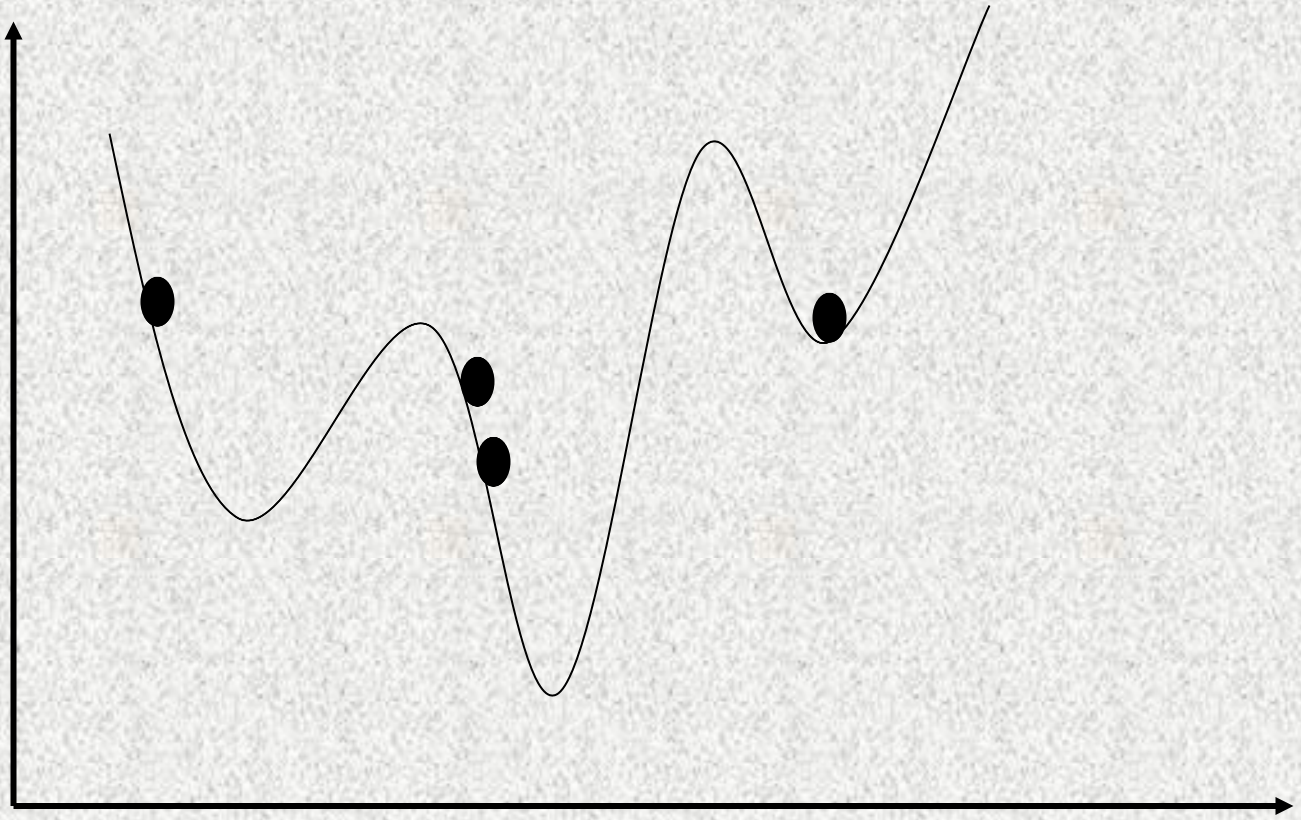
# Local Beam Search



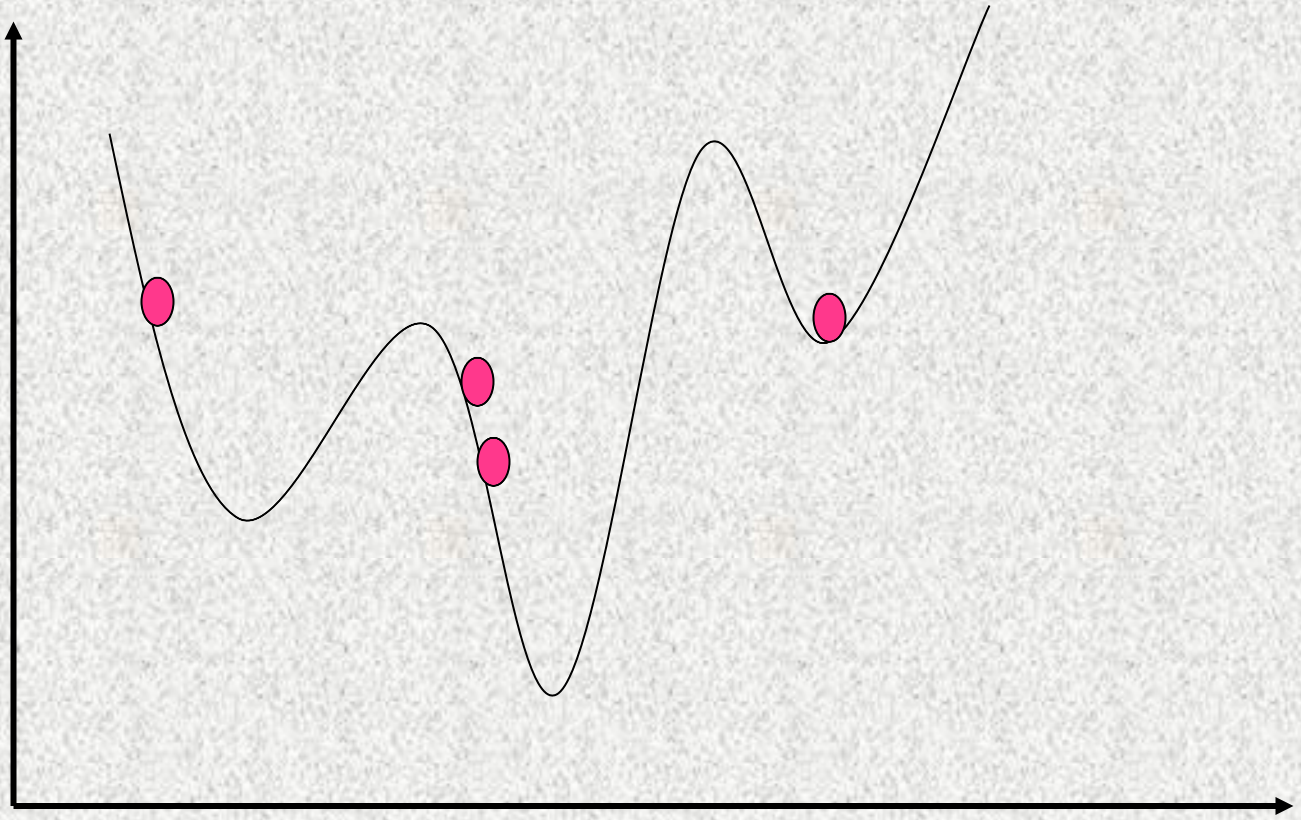
# Local Beam Search



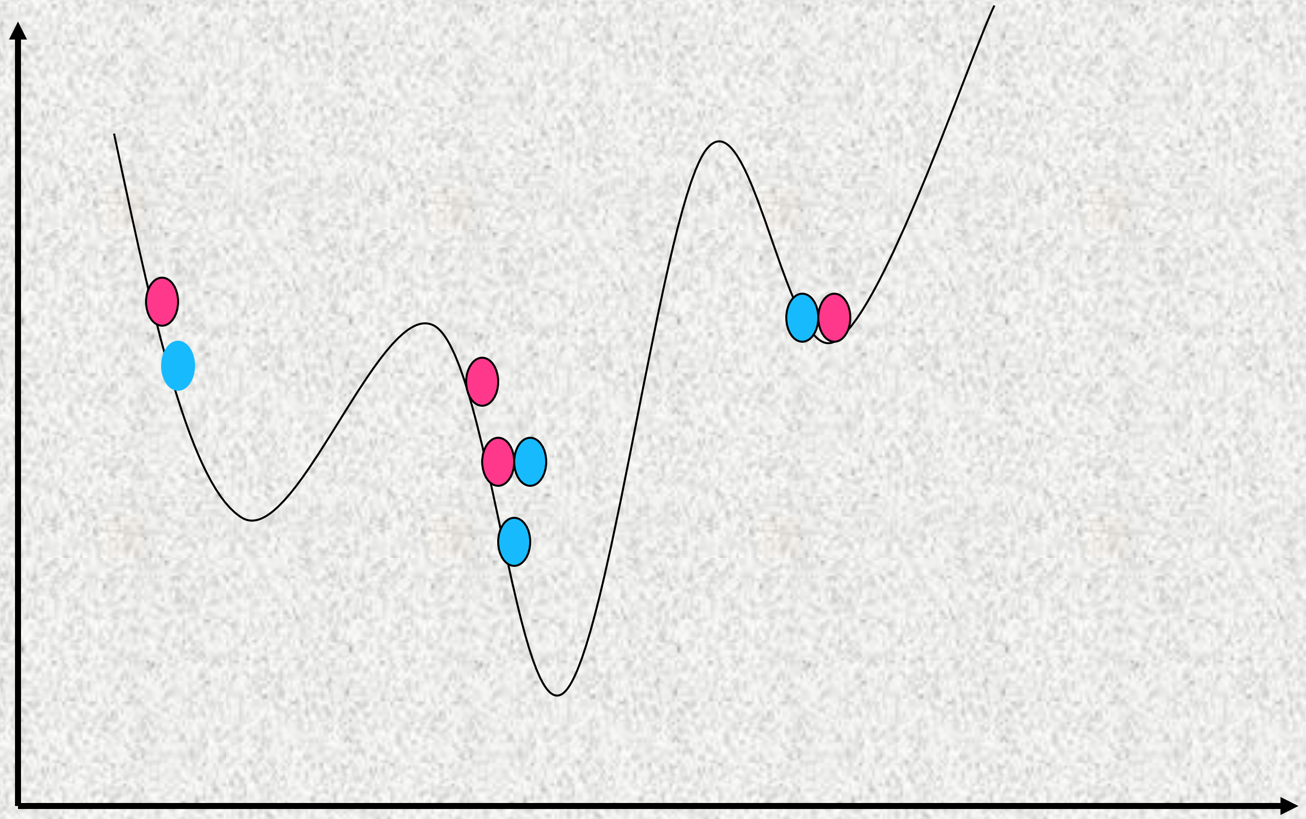
# Local Beam Search



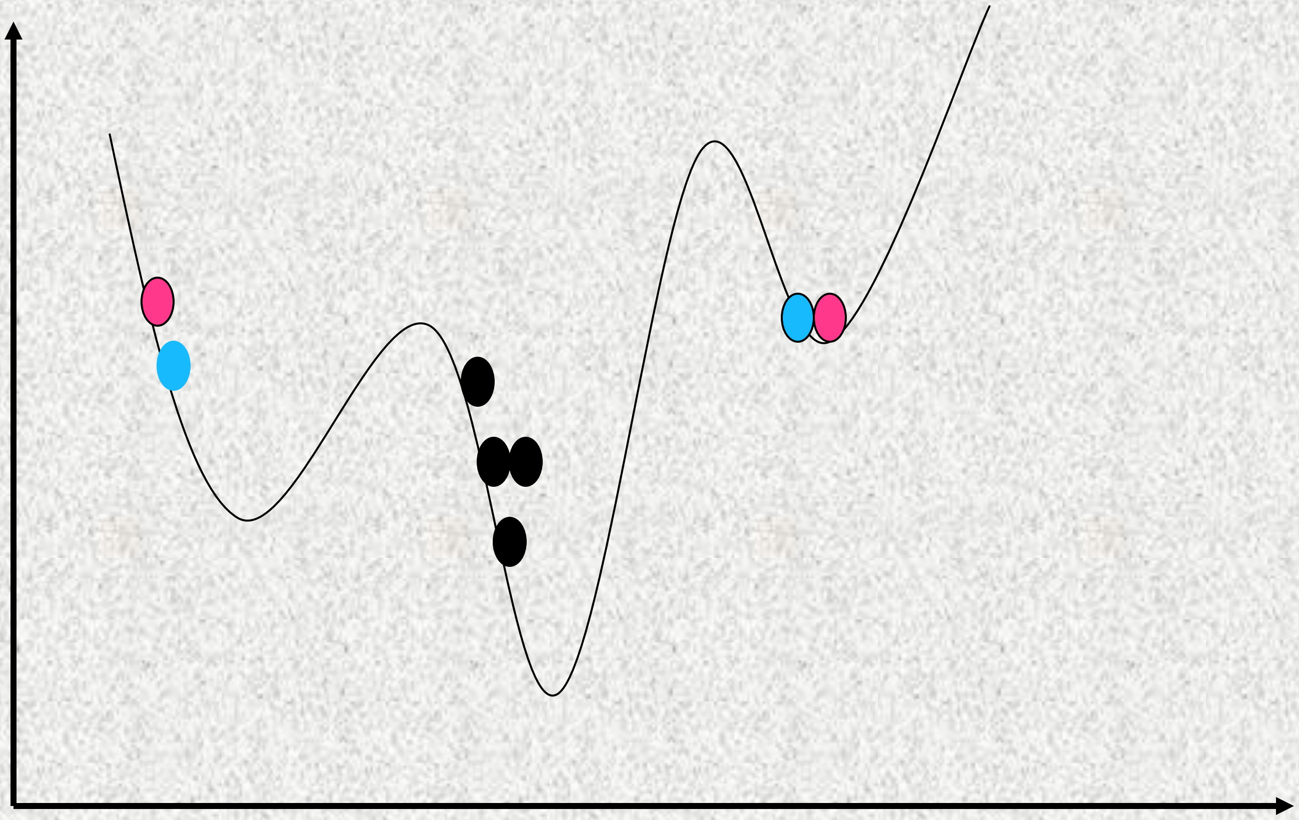
# Local Beam Search



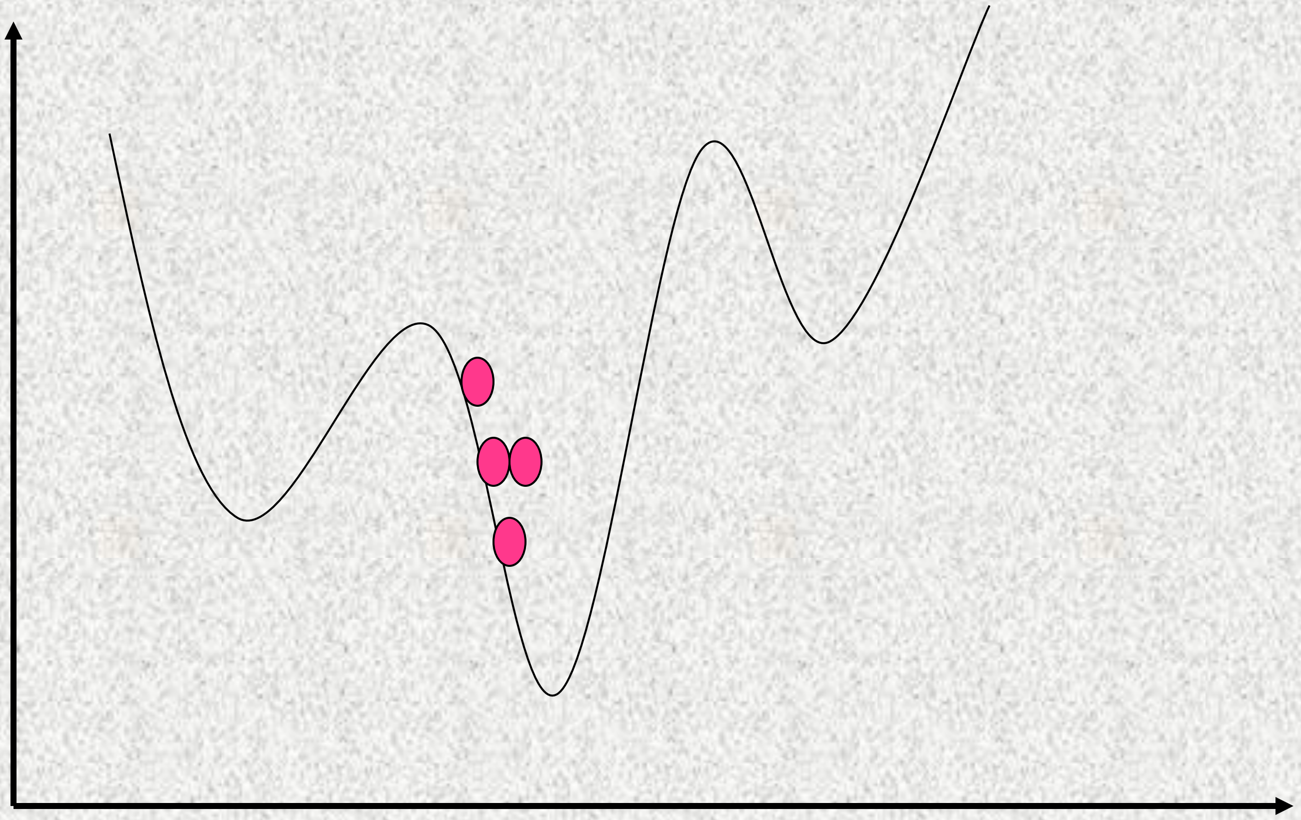
## Local Beam Search



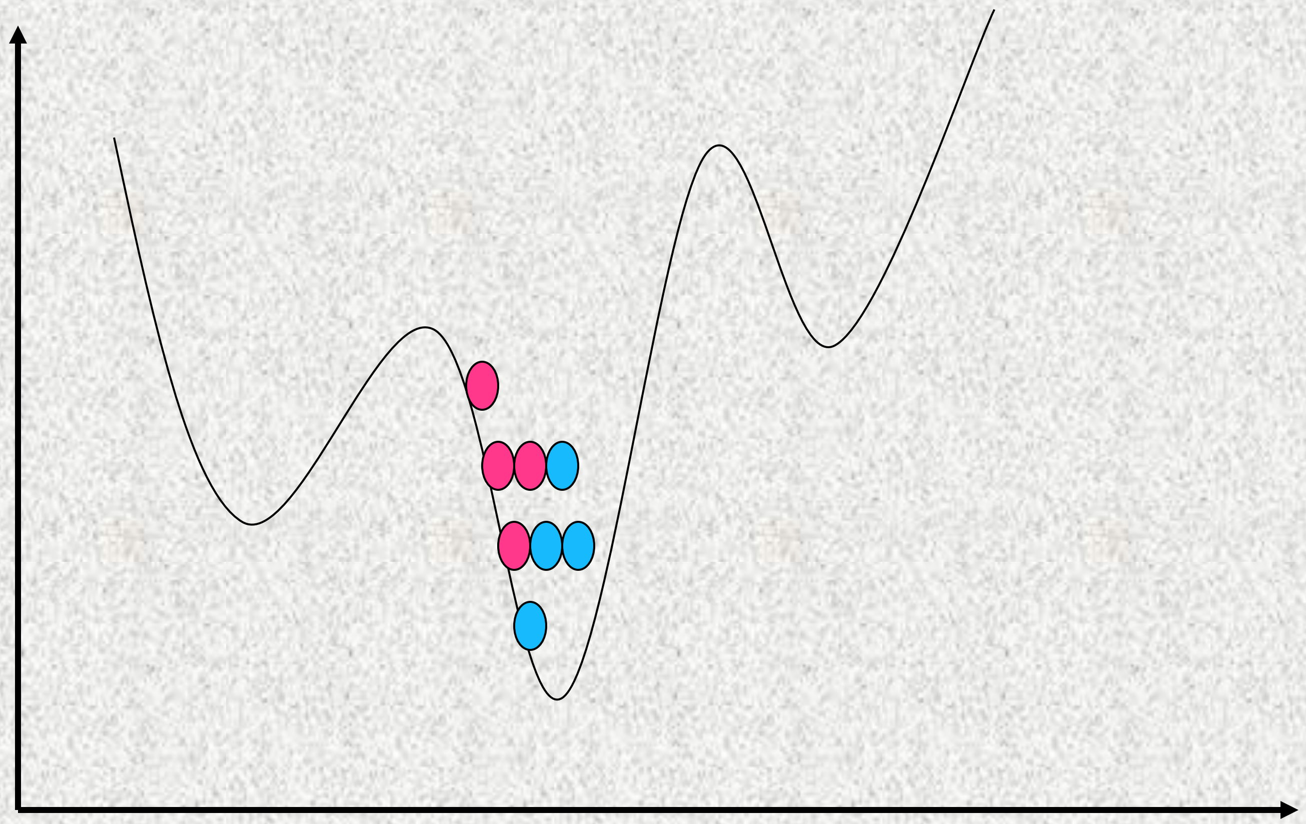
# Local Beam Search



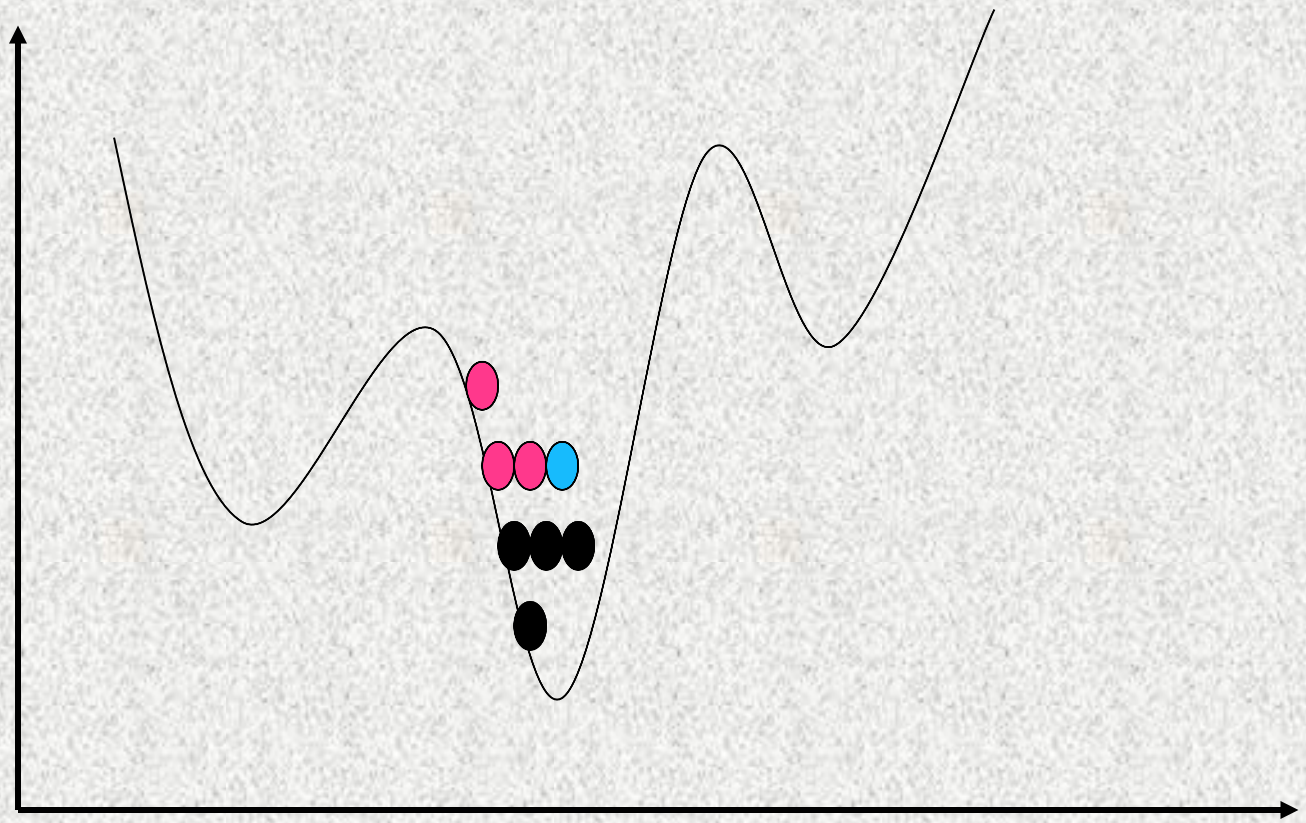
# Local Beam Search



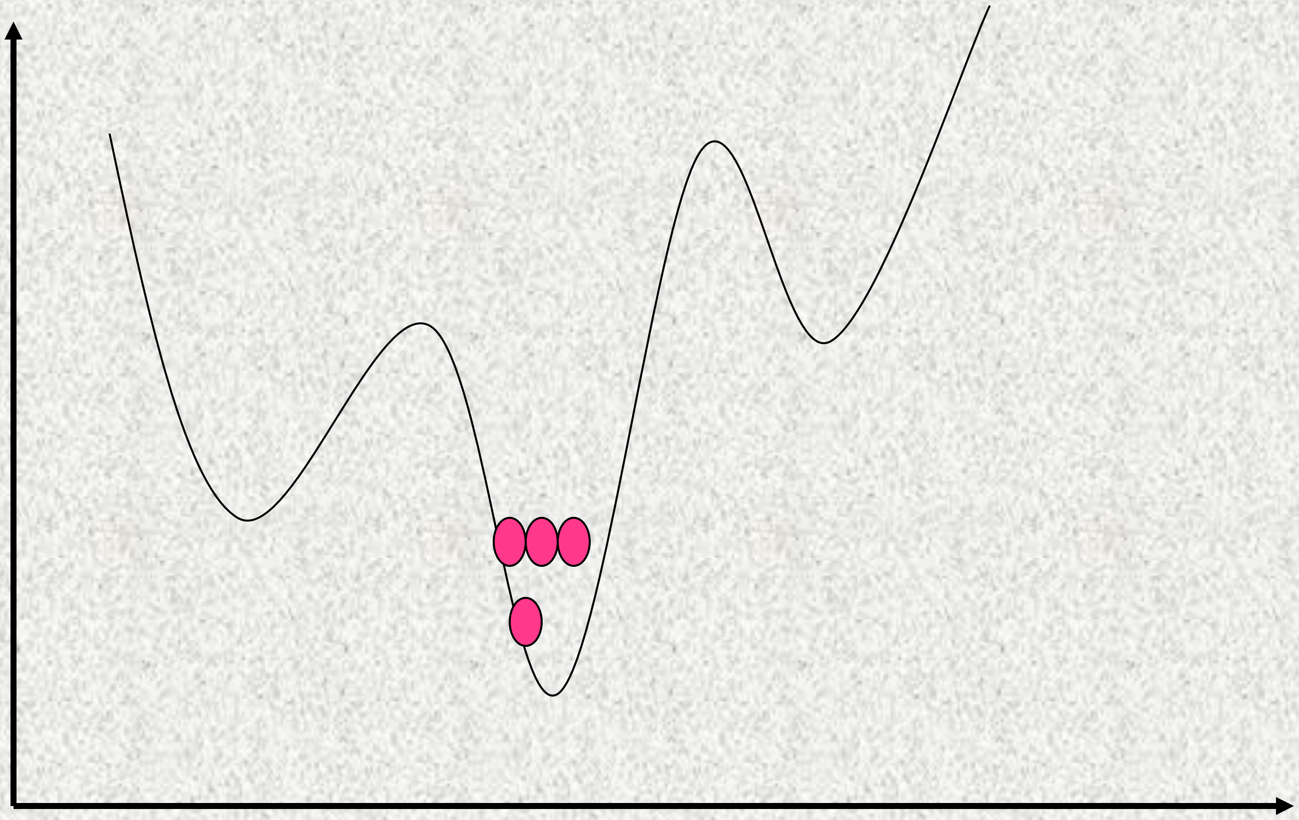
## Local Beam Search



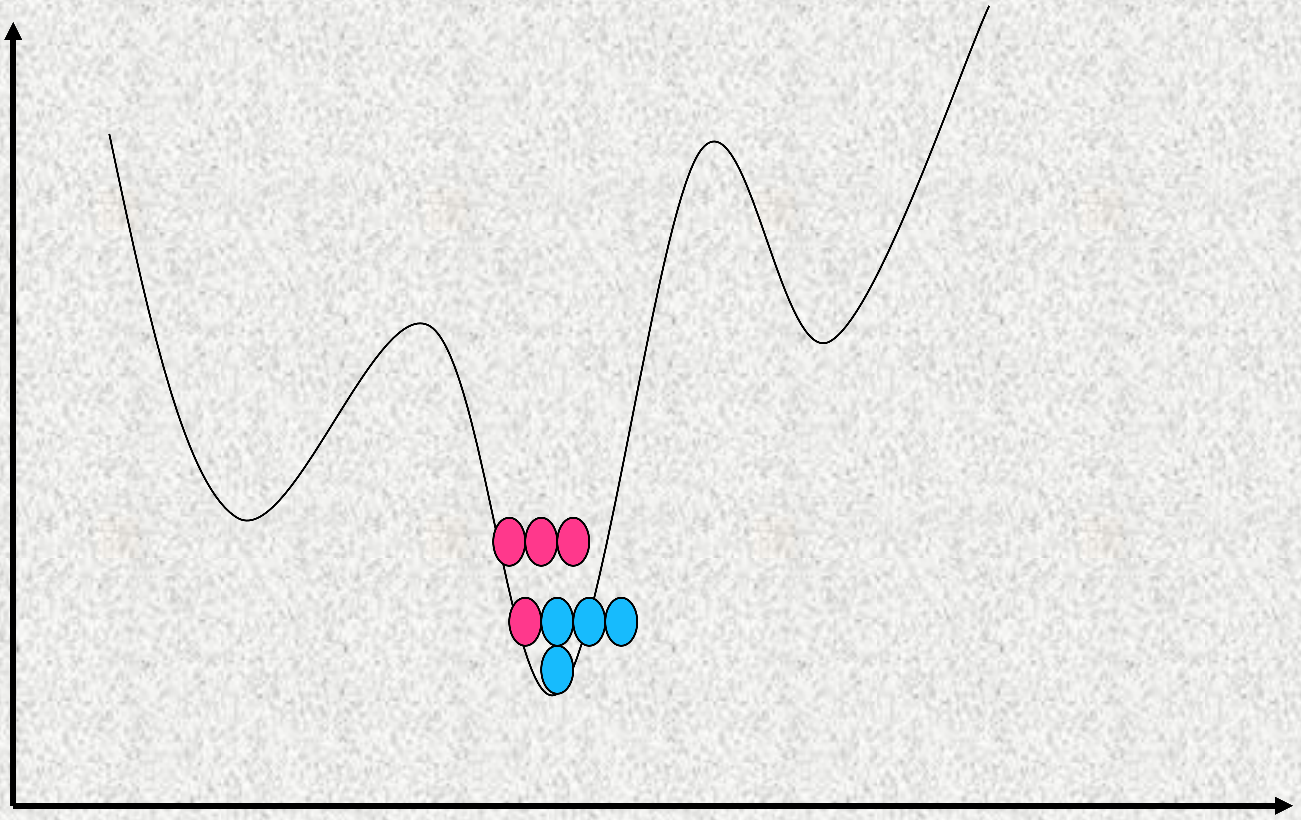
# Local Beam Search



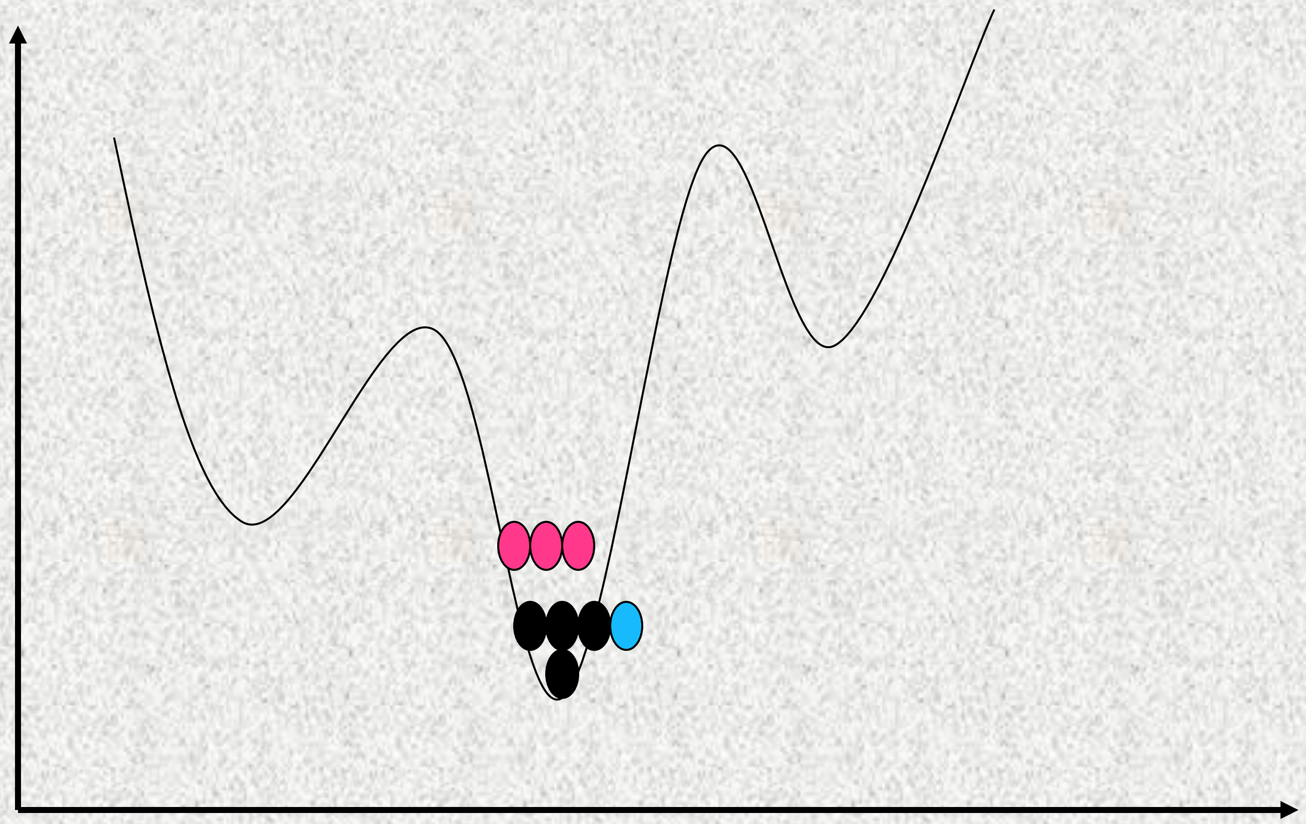
# Local Beam Search



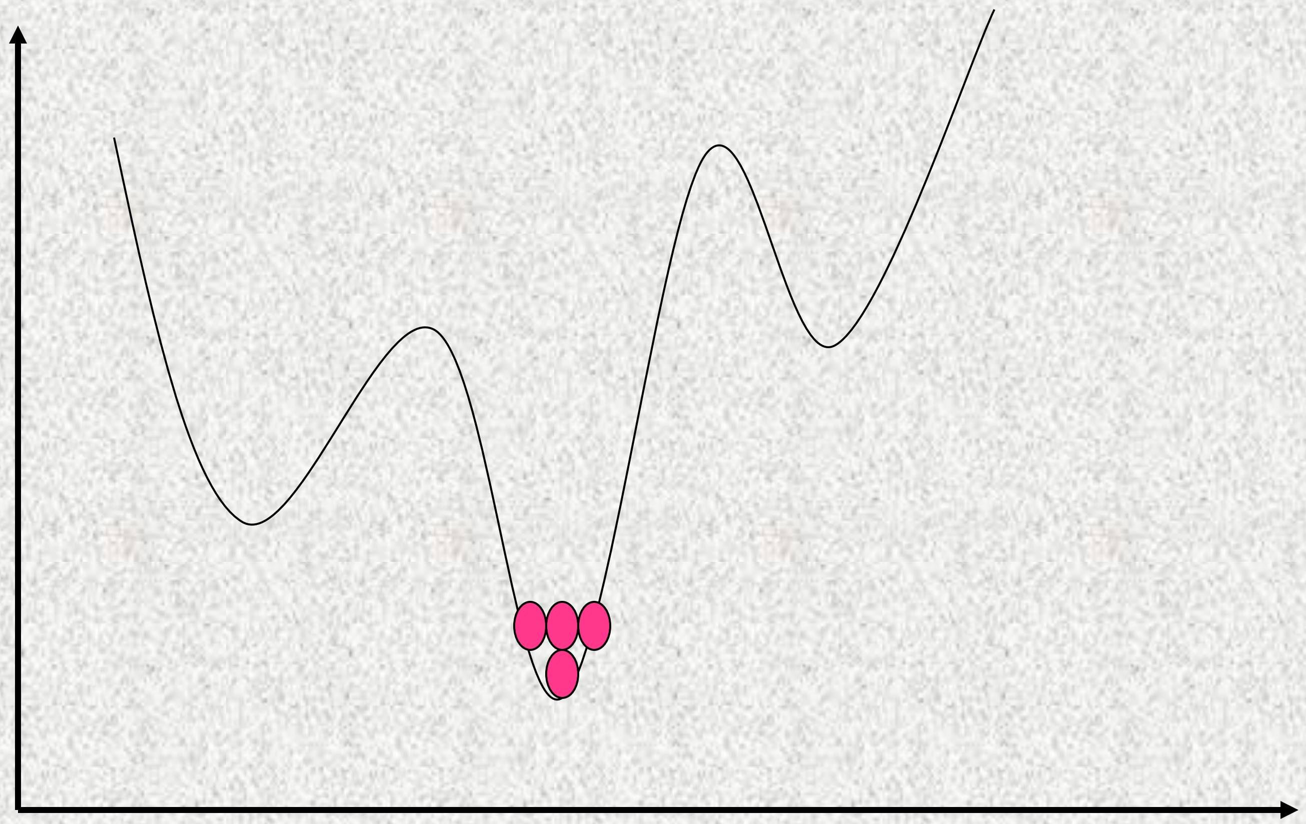
## Local Beam Search



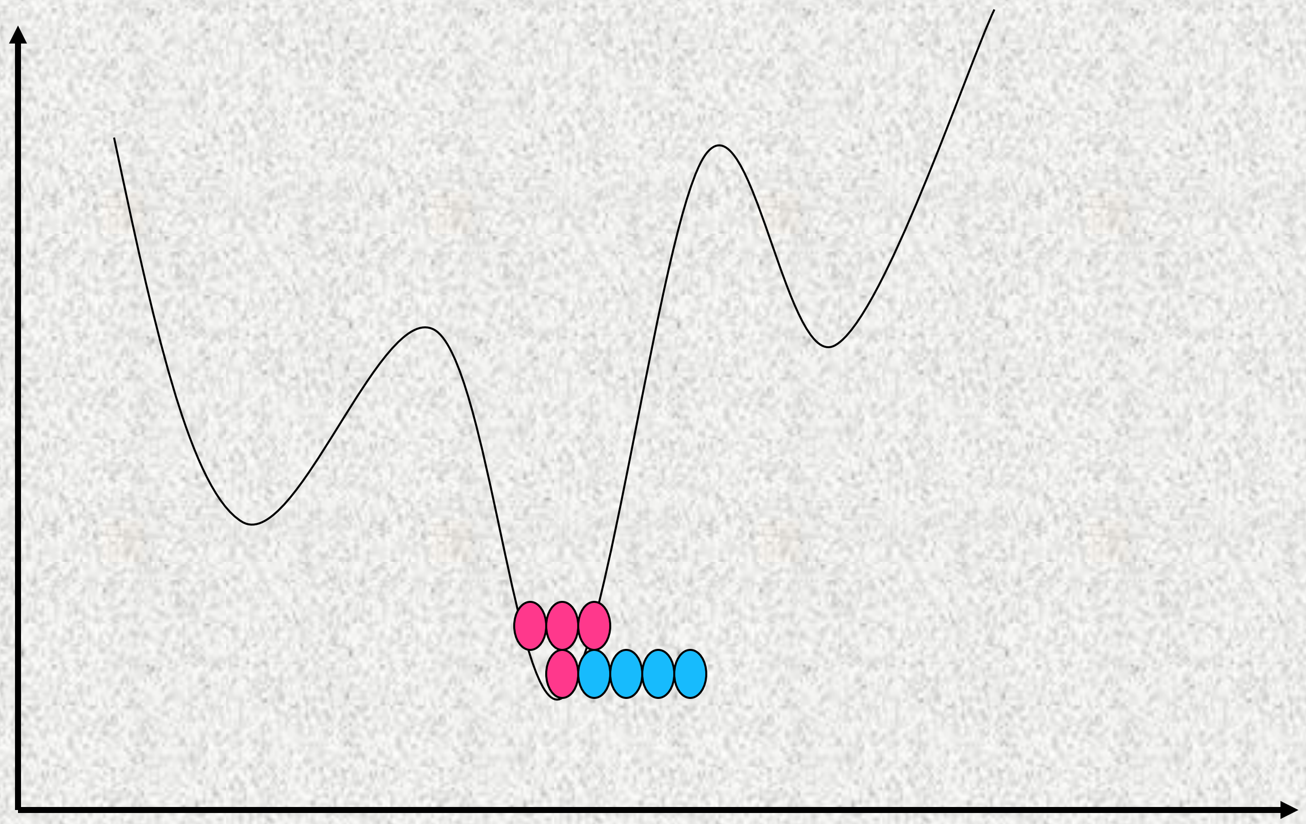
## Local Beam Search



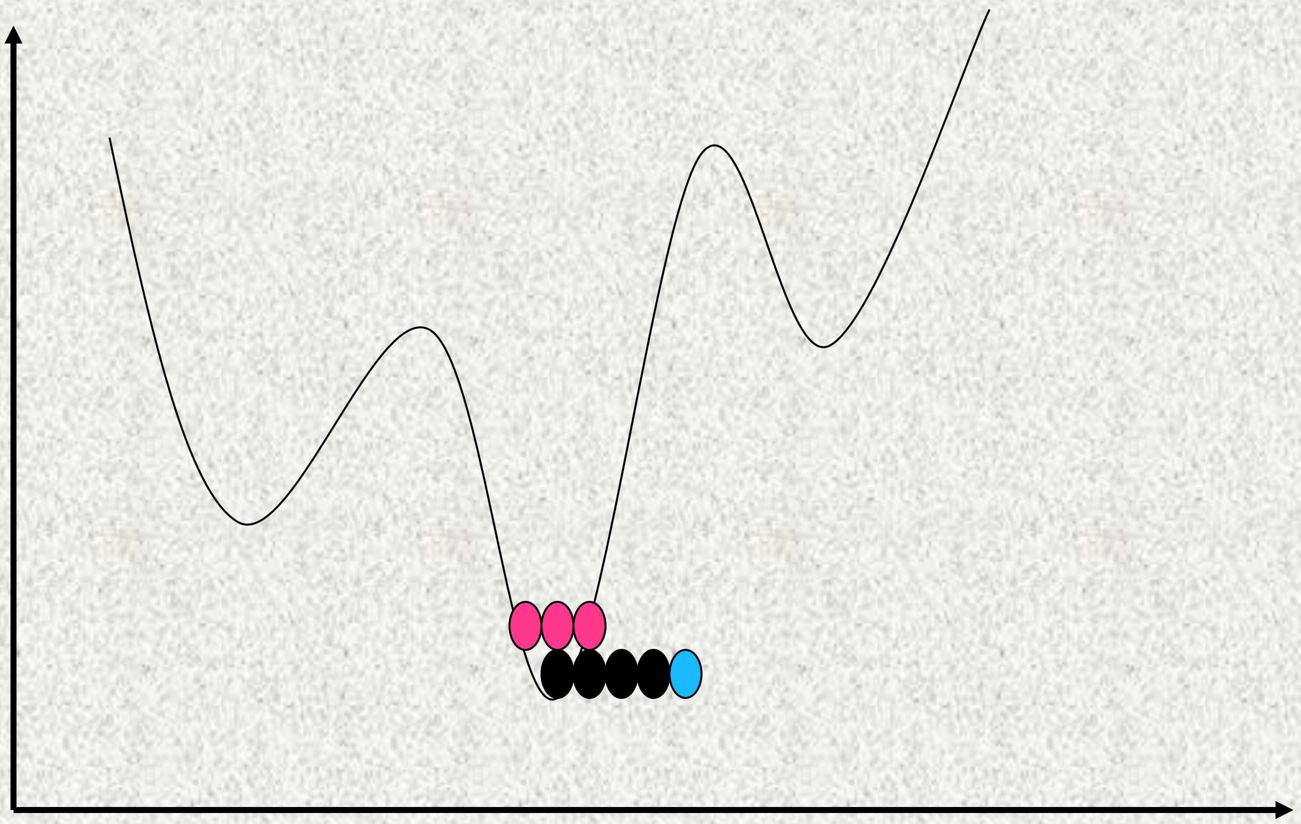
## Local Beam Search



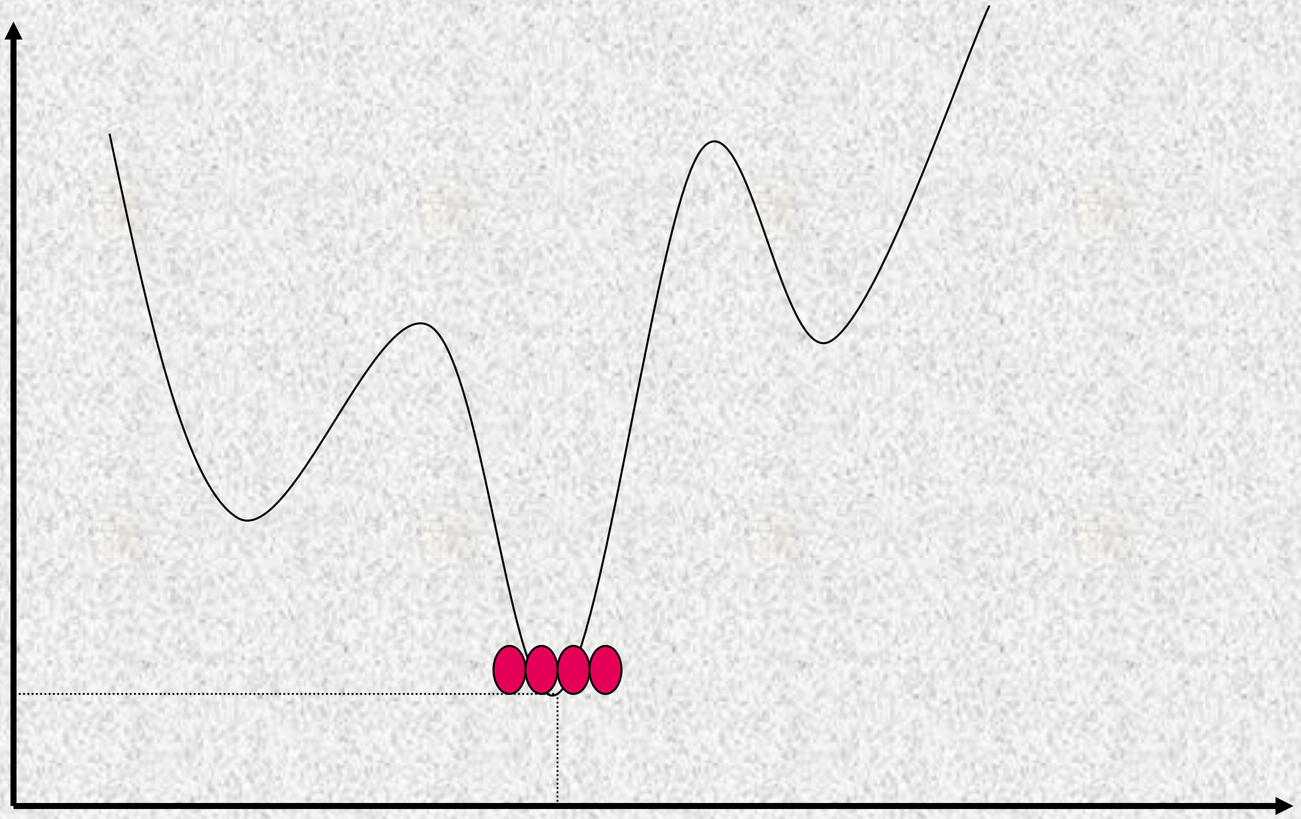
# Local Beam Search



# Local Beam Search



## Local Beam Search



## Genetic Algorithms

Genetic Algorithms are computer algorithms that search for good solutions to a problem from among a large number of possible solutions. They were proposed and developed in the 1960s by John Holland, his students, and his colleagues at the University of Michigan. These computational paradigms were inspired by the mechanics of natural evolution, including survival of the fittest, reproduction, and mutation.

# Genetic Algorithms

**Basic idea behind Gas**

GAs begin with a set of candidate solutions (chromosomes) called population. A new population is created from solutions of an old population in hope of getting a better population. Solutions which are chosen to form new solutions (offspring) are selected according to their fitness. The more suitable the solutions are the bigger chances they have to reproduce. This process is repeated until some condition is satisfied.

## Genetic algorithms (GAs)

A **genetic algorithm** (GA) is a method for solving both constrained and unconstrained optimization problems based on a natural selection process that mimics biological evolution. The **algorithm** repeatedly modifies a population of individual solutions **. GA** is a variant of stochastic beam search in which successor states are generated by **combining two parent states** (sexual reproduction).

**Concepts:**

**Encoding: Individuals** represent states. They are denoted by strings over an alphabet usually {0, 1}.

**Fitness function** is an evaluation function for rating each individual.

**Populations** are sets of individuals.

A **genetic operator** is an **operator** used in **genetic** algorithms to guide the algorithm towards a solution to a given problem. There are three main types of **operators** (mutation, crossover and selection), which must work in conjunction with one another in order for the algorithm to be successful.

# Genetic algorithms (GAs)

Name and describe the main features of Genetic Algorithms (GA).

**Answer:** *Genetic Algorithms (GA) use principles of natural evolution. There are five important features of GA:*

**Encoding** *possible solutions of a problem are considered as individuals in a population. If the solutions can be divided into a series of small steps (building blocks), then these steps are represented by genes and a series of genes (a chromosome) will encode the whole solution. This way different solutions of a problem are represented in GA as chromosomes of individuals.*

**Fitness Function** *represents the main requirements of the desired solution of a problem (i.e. cheapest price, shortest route, most compact arrangement, etc). This function calculates and returns the fitness of an individual solution.*

# Genetic algorithms (GAs)

Name and describe the main features of Genetic Algorithms (GA).
**Selection** *operator defines the way individuals in the current population are selected for reproduction. There are many strategies for that (e.g. roulette–wheel, ranked, tournament selection, etc), but usually the individuals which are more fit are selected.*
**Crossover** operator defines how chromosomes of parents are mixed in order to obtain genetic codes of their offspring (e.g. one–point, two–point, uniform crossover, etc). This operator implements the inheritance property (offspring inherit genes of their parents).
**Mutation** operator creates random changes in genetic codes of the offspring. This operator is needed to bring some random diversity into the genetic code. In some cases GA cannot find the optimal solution without mutation operator (local maximum problem).

## The GA Procedure

1. Initialize a population (of solution guesses)
2. Do (once for each generation)
   a. Evaluate each chromosome (String) in the population using a fitness function
   b. Apply GA operators to population to create a new population
3. Finish when solution is reached or number of generations has reached an allowable maximum.

# The genetic algorithm

```
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))
```

# The genetic algorithm

**Basic elements of GAs**
Most GAs methods are based on the following elements, populations of chromosomes, selection according to fitness, crossover to produce new offspring, and random mutation of new offspring .
***Chromosomes***
The chromosomes in GAs represent the space of candidate solutions. Possible chromosomes encodings are binary, permutation, value, and tree encodings. For the Knapsack problem, we use binary encoding, where every chromosome is a string of bits, 0 or 1.
***Fitness function***
GAs require a fitness function which allocates a score to each chromosome in the current population. Thus, it can calculate how well the solutions are coded and how well they solve the problem.

## The genetic algorithm

***Selection***

The selection process is based on fitness. Chromosomes that are evaluated with higher values (fitter) will most likely be selected to reproduce, whereas, those with low values will be discarded. The fittest chromosomes may be selected several times, however, the number of chromosomes selected to reproduce is equal to the population size, therefore, keeping the size constant for every generation. This phase has an element of randomness just like the survival of organisms in nature. The most used selection methods, are roulette-wheel, rank selection, steady-state selection, and some others.

## The genetic algorithm

***Crossover***

Crossover is the process of combining the bits of one chromosome with those of another. This is to create an offspring for the next generation that inherits traits of both parents.
Crossover randomly chooses a locus and exchanges the subsequences before and after that locus between two chromosomes to create two offspring [2]. For example, consider the following parents and a crossover point at position 3:

Parent 1 1 0 0 | 0 1 1 1
Parent 2 1 1 1 | 1 0 0 0
Offspring 1 1 0 0 1 0 0 0
Offspring 2 1 1 1 0 1 1 1

In this example, Offspring 1 inherits bits in position 1, 2, and 3 from the left side of the crossover point from Parent 1 and the rest from the right side of the crossover point from Parent 2. Similarly, Offspring 2 inherits bits in position 1, 2, and 3 from the left side of Parent 2 and the rest from the right side of Parent 1.

## The genetic algorithm

### ***Mutation***

Mutation is performed after crossover to prevent falling all solutions in the population into a local optimum of solved problem. Mutation changes the new offspring by flipping bits from 1 to 0 or from 0 to 1. Mutation can occur at each bit position in the string with some probability, usually very small (e.g. 0.001). For example, consider the following chromosome with mutation point at position 2:

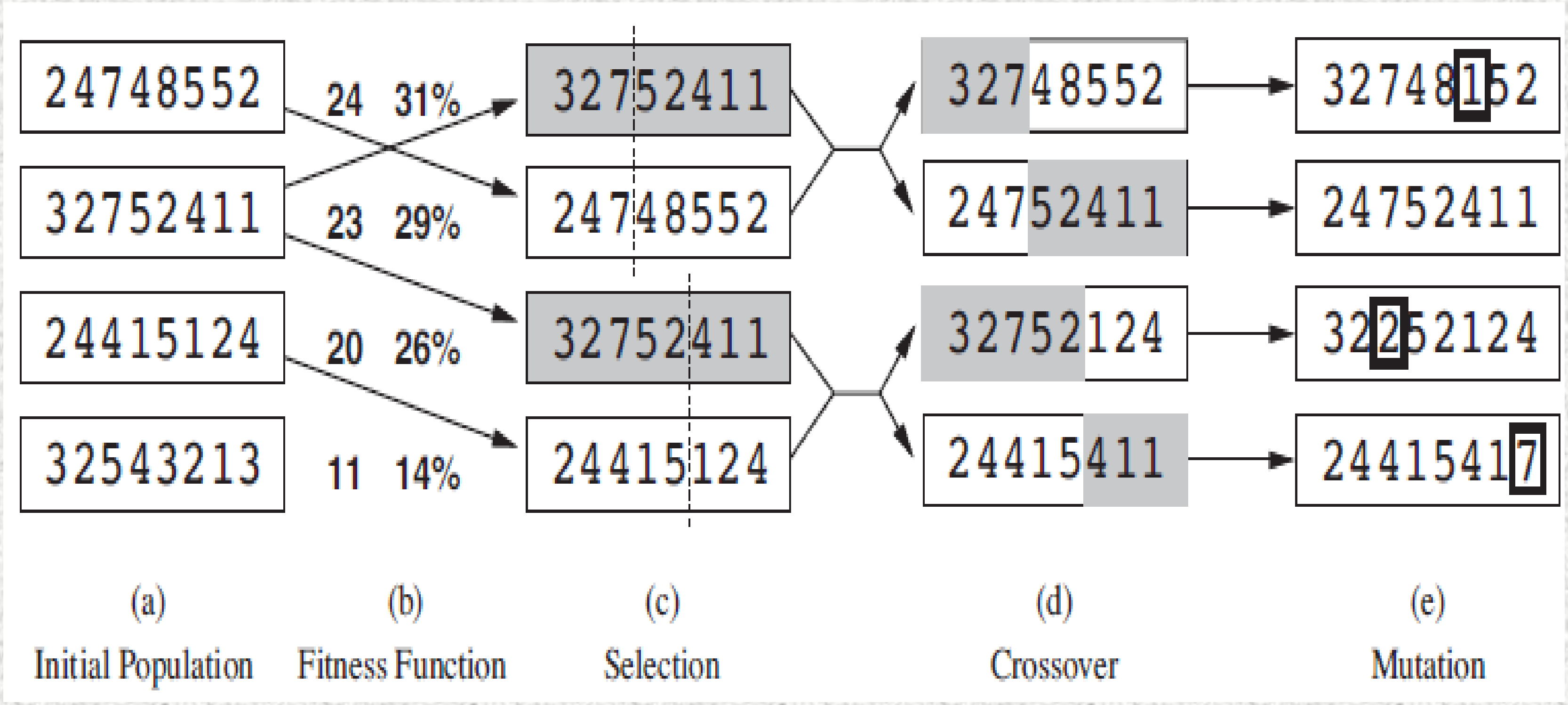
Not mutated chromosome: 1 *0* 0 0 1 1 1

Mutated: 1 *1* 0 0 1 1 1 The 0 at position 2 flips to 1 after mutation.

# Genetic Algorithms

- What two requirements should a problem satisfy in order to be suitable for
  solving it by a GA?
  **Answer:** *GA can only be applied to problems that satisfy the following*
  *requirements:*
  *• The fitness function can be well–defined.*
  *• Solutions should be decomposable into steps (building blocks) which could be then encoded as chromosomes.*

# Genetic algorithm example



The genetic algorithm, illustrated for digit strings representing 8-queens states.
The initial population of four 8-digits string represent 8-queen states

# Genetic algorithm example

In (b): States in (a) are ranked by the fitness function(FF) (FF: no of non attacking queens, for a solution (a). The states with the best values: 24, 23, 20, 11 are chosen. Probability of being chosen is directly proportional to Fitness Score.

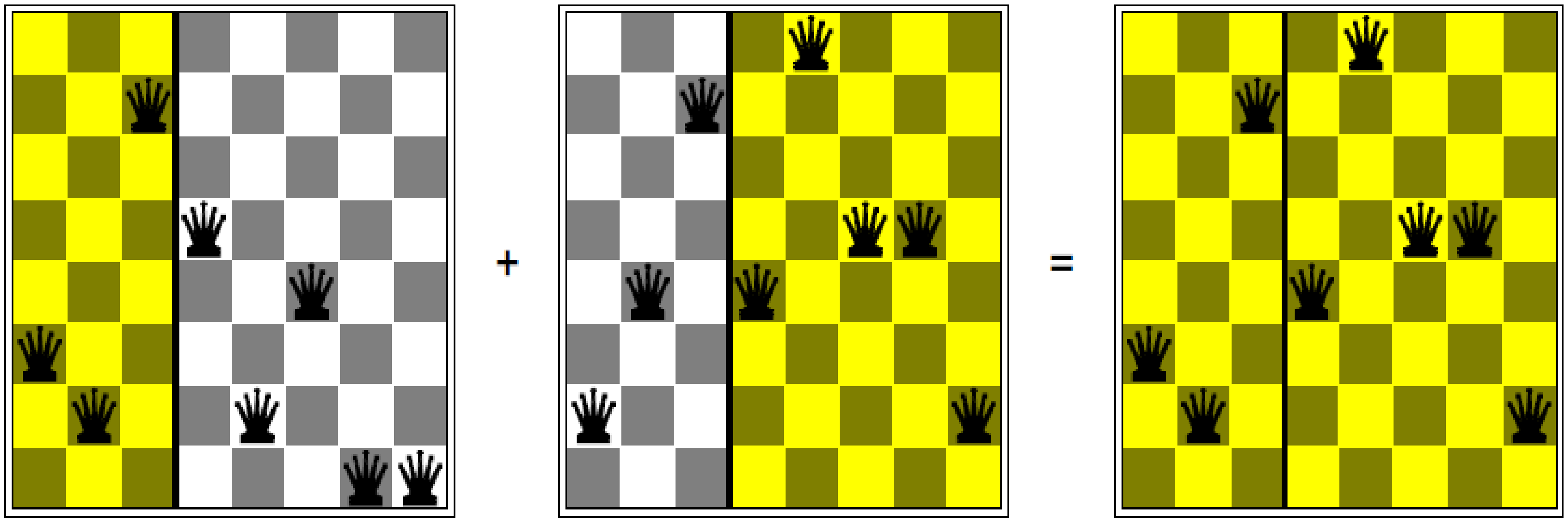
In (c) two pairs are selected at random from (b). One is selected twice and one is not selected at all.

Each pair to be mated, a crossover point is chosen at random from position in the string.

In (d): They produce offspring. This state may be far away from either parent if parent states are quite different. If population is diverse to start with. So crossover process takes large steps (like SA) early and smaller steps when population is quite similar.

In (e) Random Mutation one digit was mutated in 1st, 3rd and 4th offspring.

# The genetic algorithm with the 8-queens problem



The 8-queens states corresponding to the first two parents in previous fig. (c) and the first offspring in Fig. (d). The shaded columns are lost in the crossover step and the un-shaded columns are retained.

## Common Operators

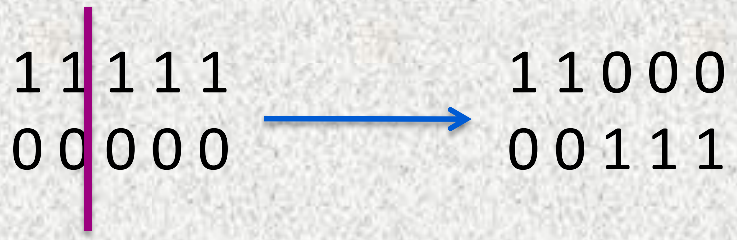
- Reproduction
- Crossover
- Mutation

## Reproduction

- Select individuals x according to their fitness values f(x)
  - Like beam search
- Fittest individuals survive (and possibly mate) for next generation

## Crossover

- Select two parents
- Select cross site
- Cut and splice pieces of one parent to those of the other



# Mutation

- With small probability, randomly alter 1 bit
- Minor operator
- An insurance policy against lost bits
- Pushes out of local minima

Population: Goal: 0 1 1 1 1 1

1 1 0 0 0 0
1 0 1 0 0 0
1 0 0 1 0 0
0 1 0 0 0 0

Mutation needed to find the goal

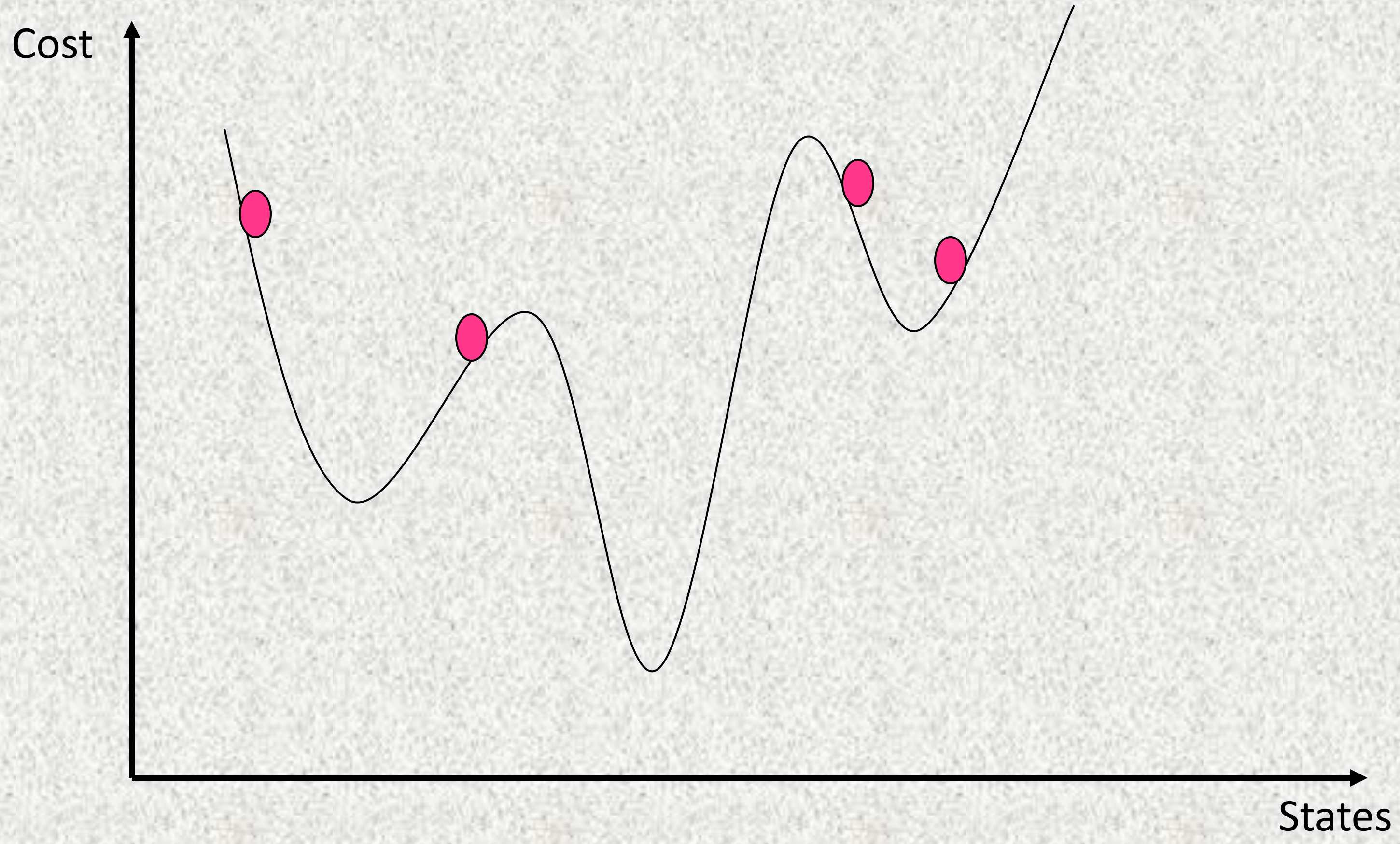
## GA as Search

- Genetic algorithms are local heuristic search algorithms.
- Especially good for problems that have large and poorly understood search spaces.
- Genetic algorithms use a randomized parallel beam search to explore the state space.
- You must be able to define a good fitness function, and of course, a good state representation.

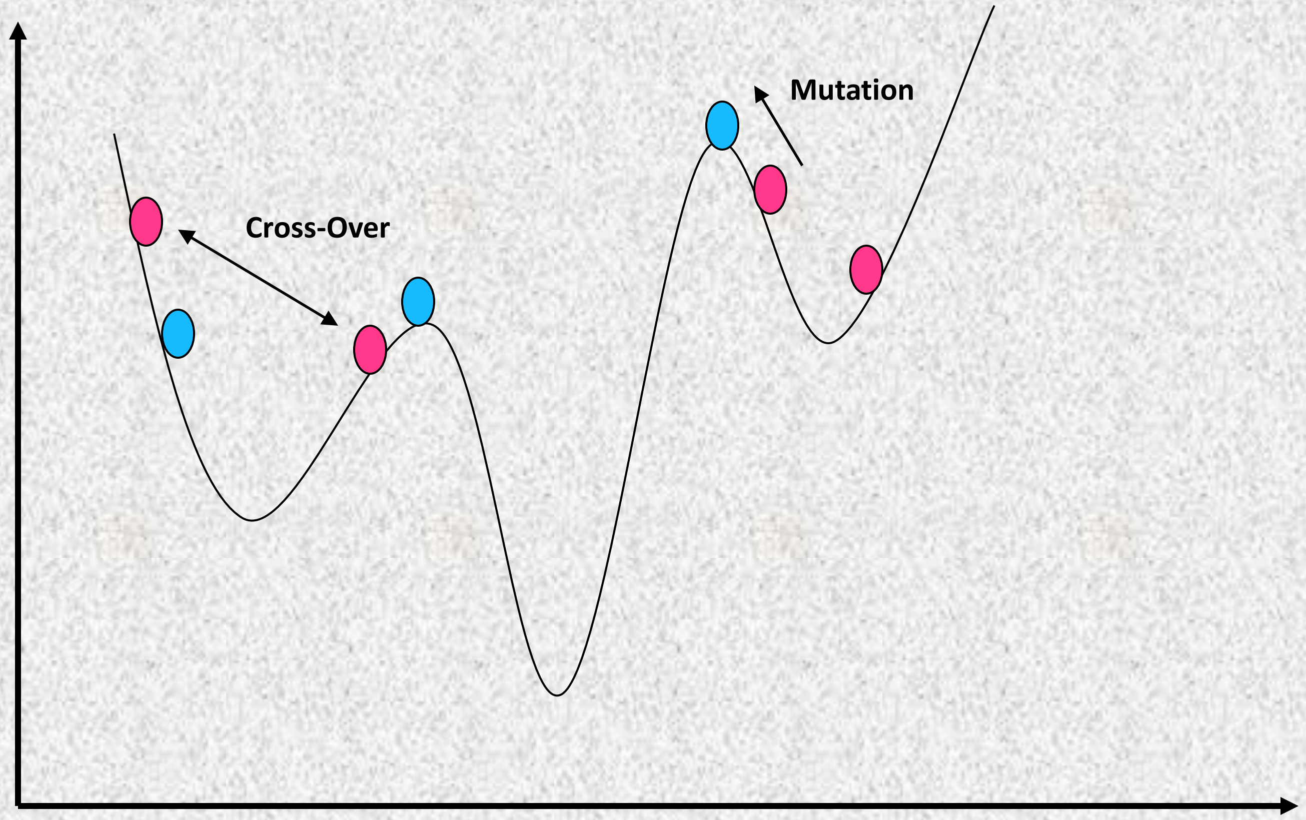
## Issues

- How to select original population?
- How to handle non-binary solution types?
- What should be the size of the population?
- What is the optimal mutation rate?
- How are mates picked for crossover?
- Can any chromosome appear more than once in a population?
- When should the GA halt?
- Local minima?
- Parallel algorithms?

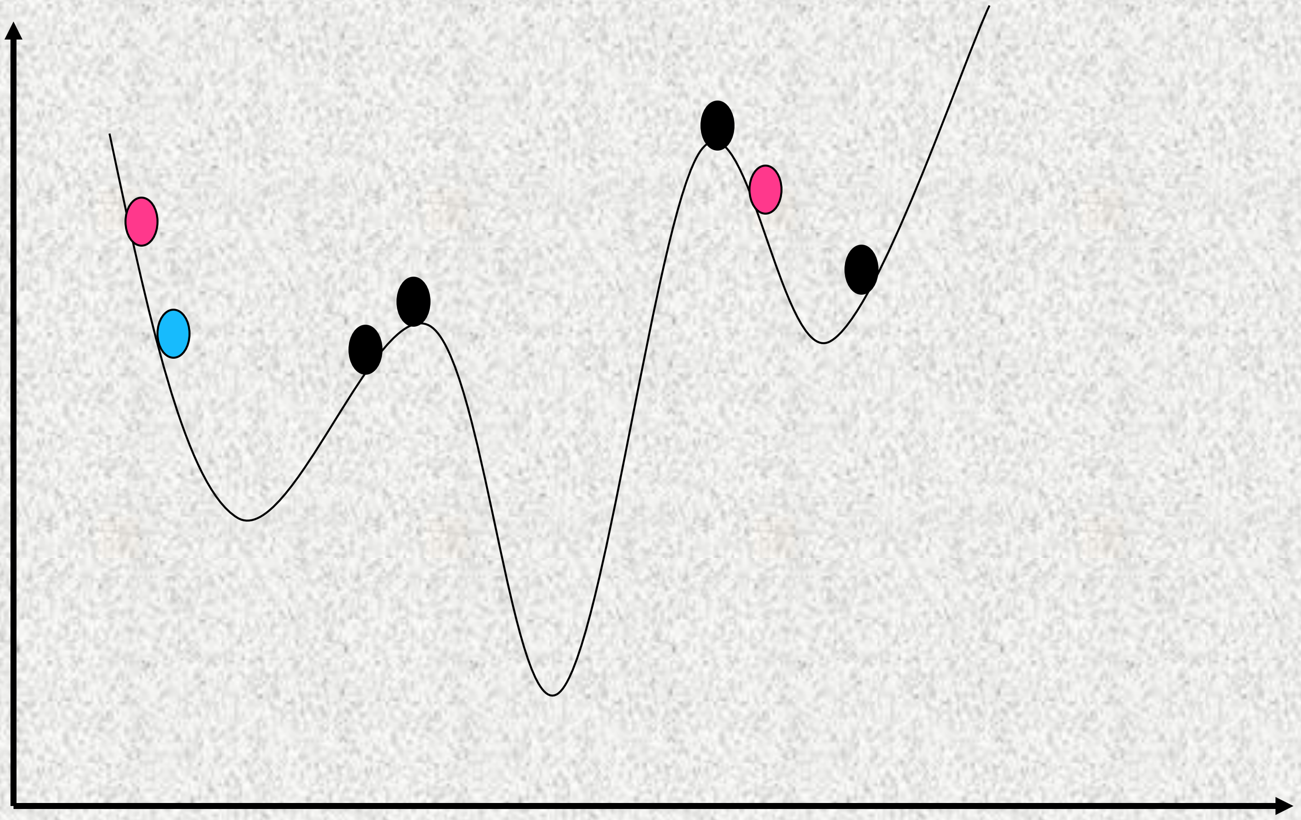
# Genetic Algorithms



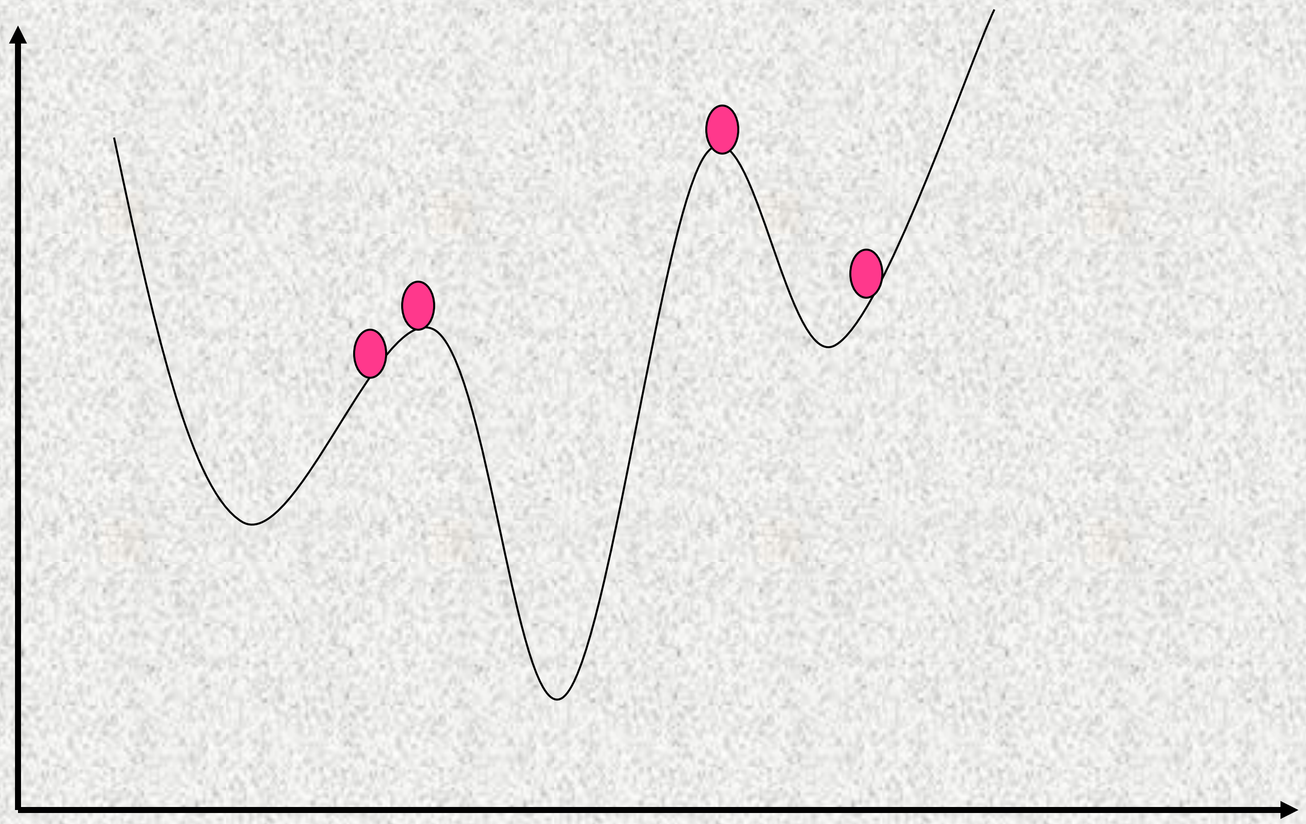
# Genetic Algorithms



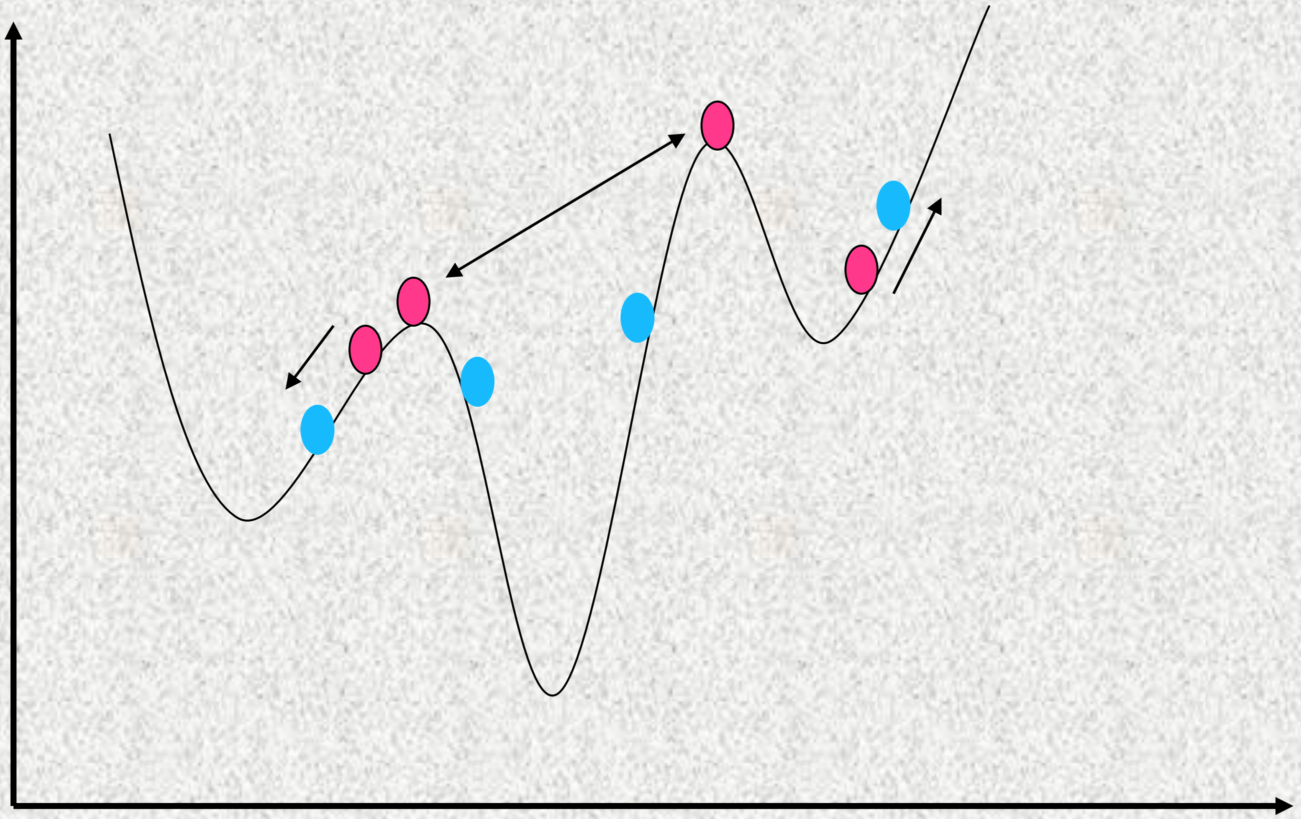
# Genetic Algorithms



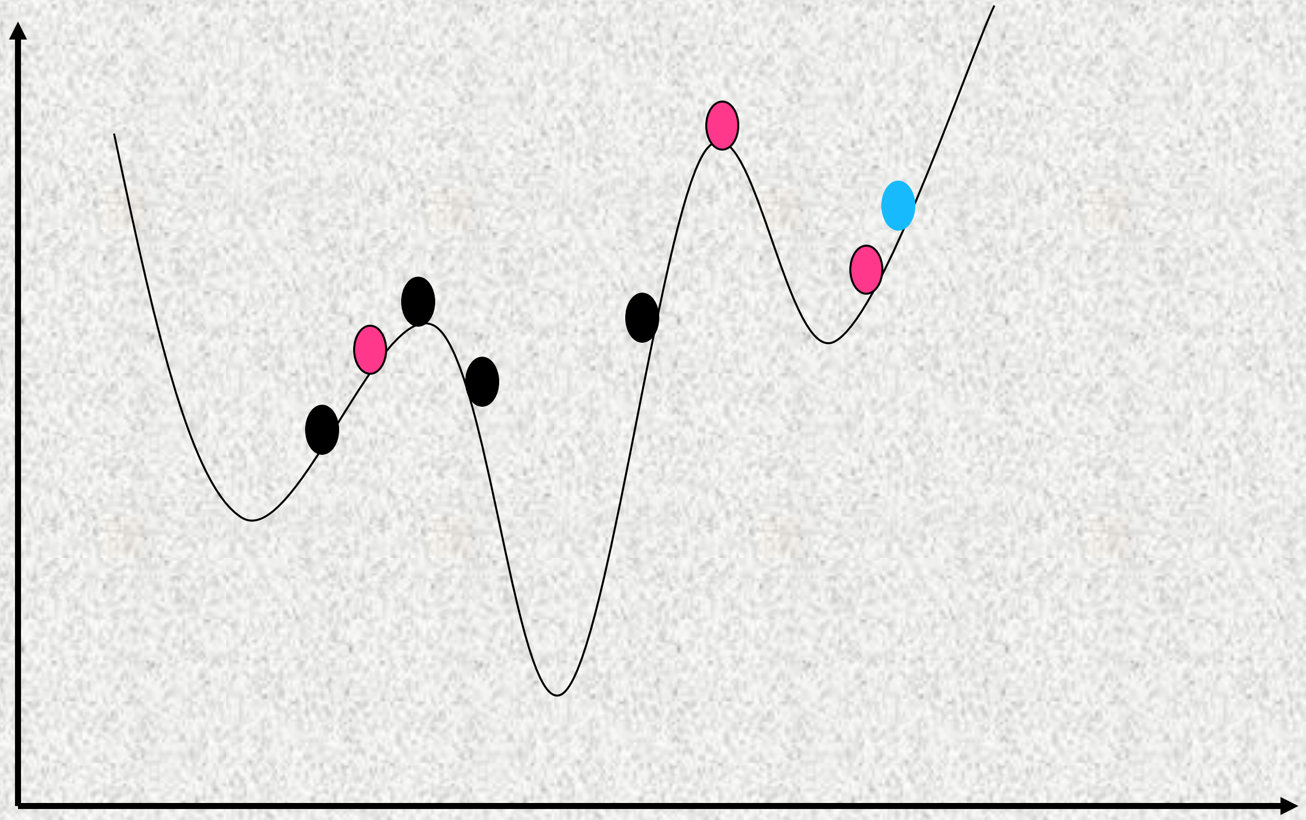
## Genetic Algorithms



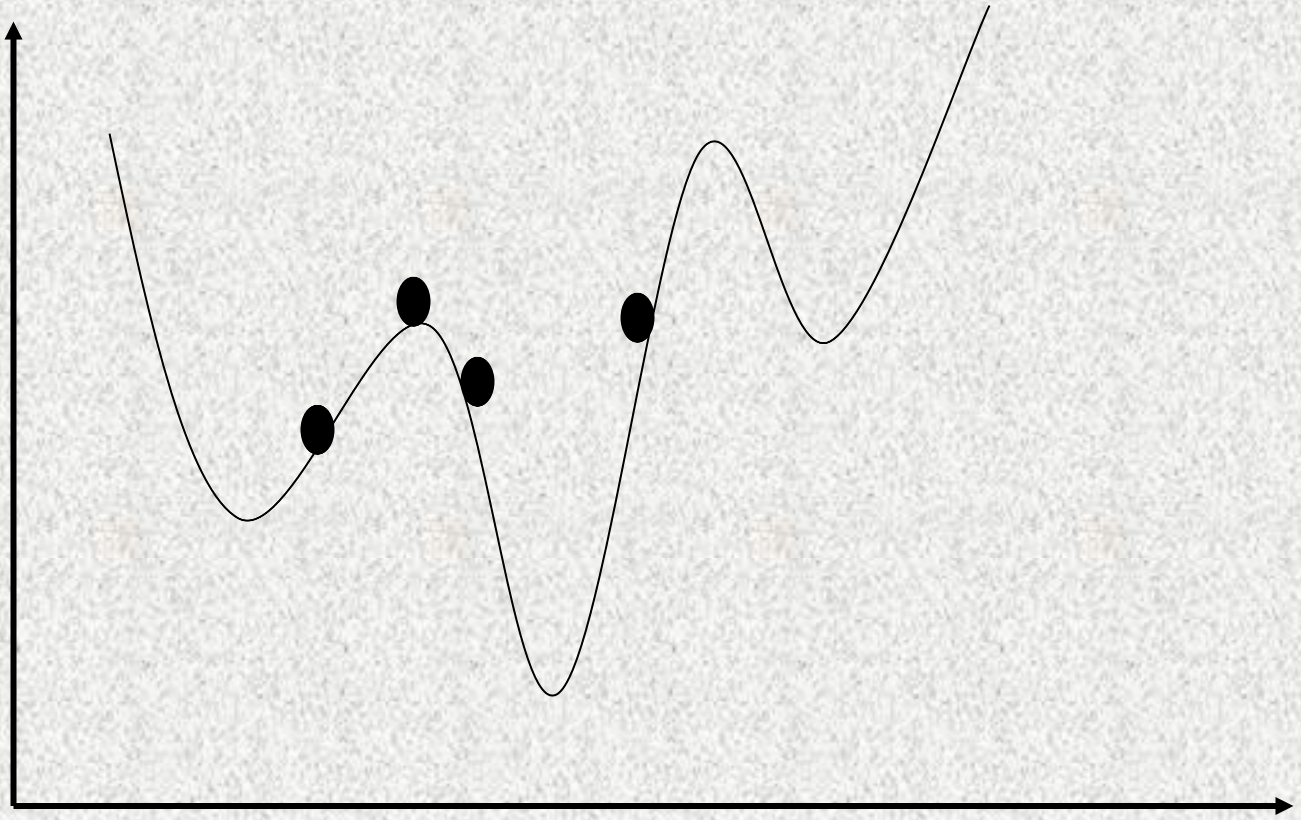
## Genetic Algorithms



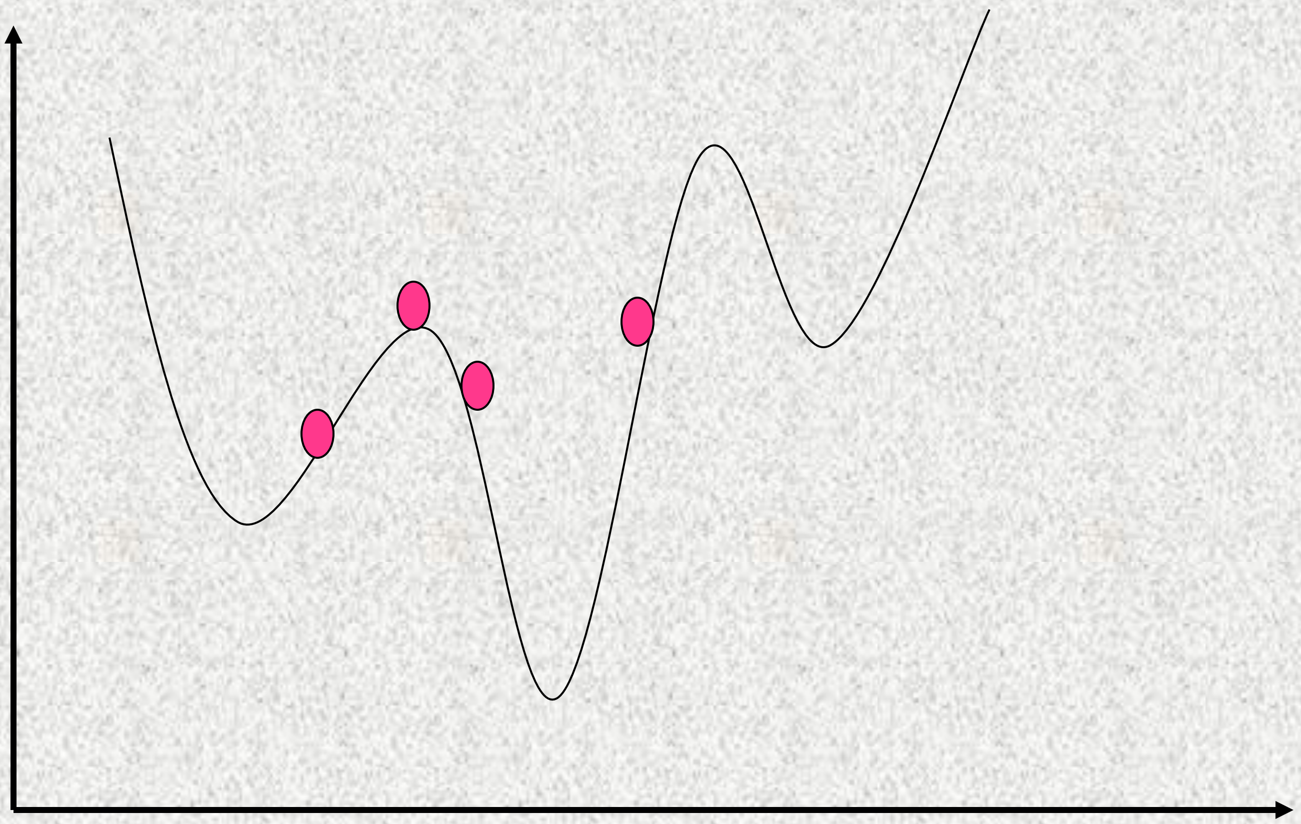
## Genetic Algorithms



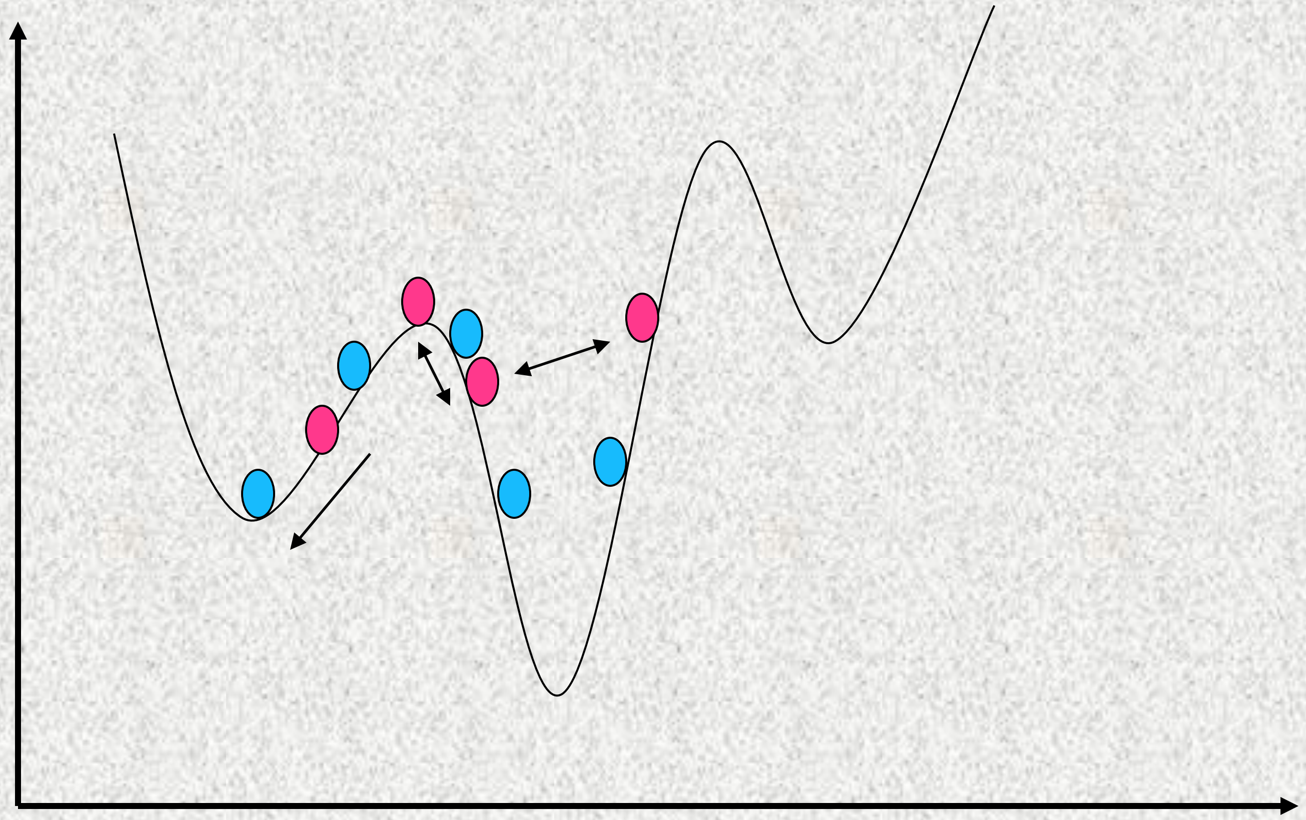
## Genetic Algorithms



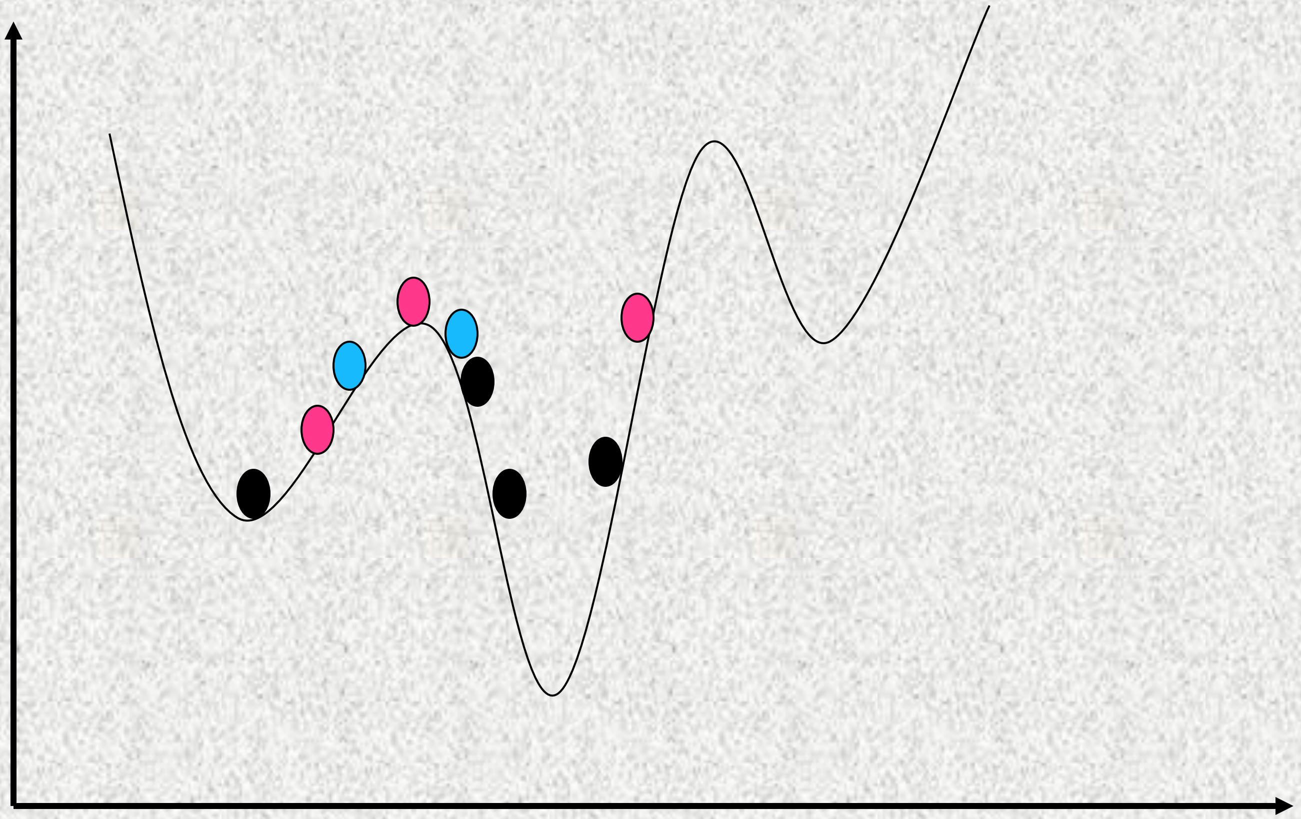
## Genetic Algorithms



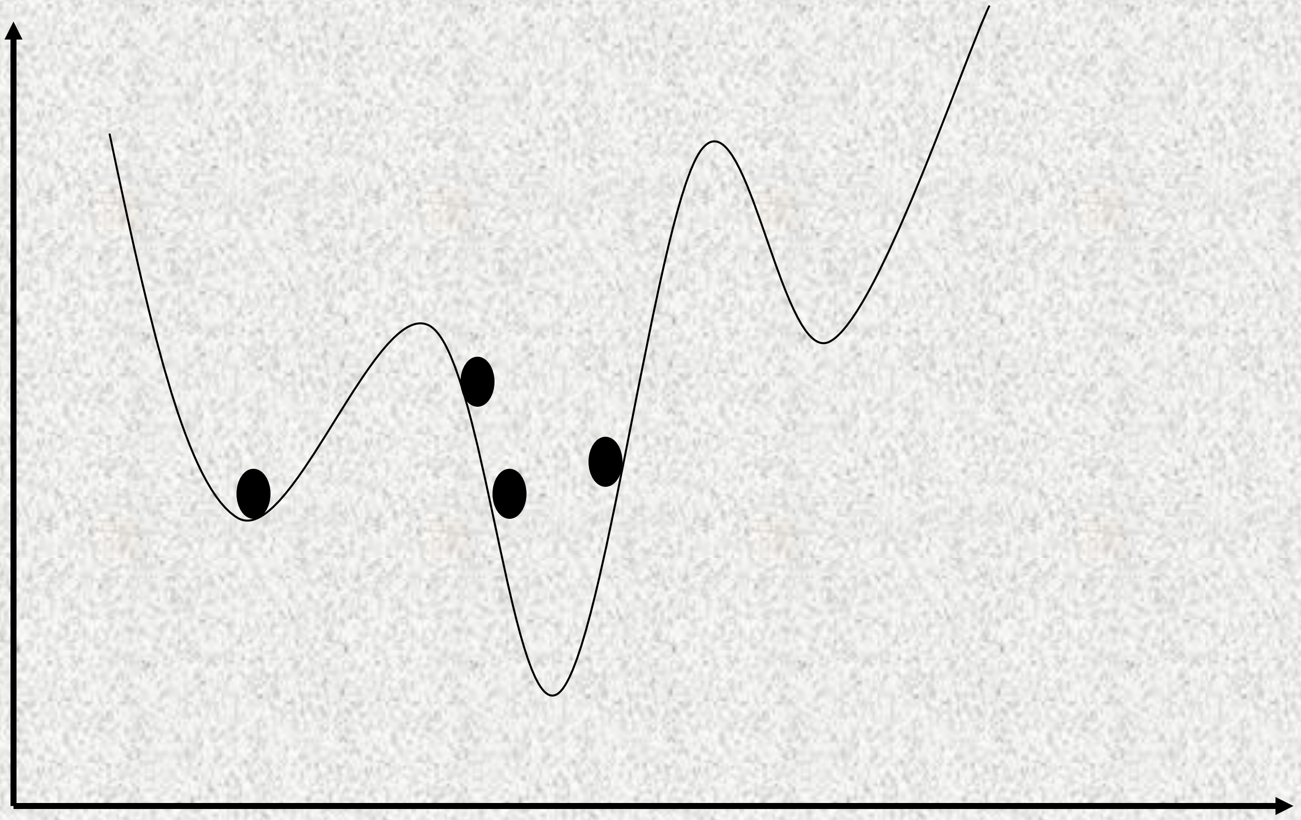
## Genetic Algorithms



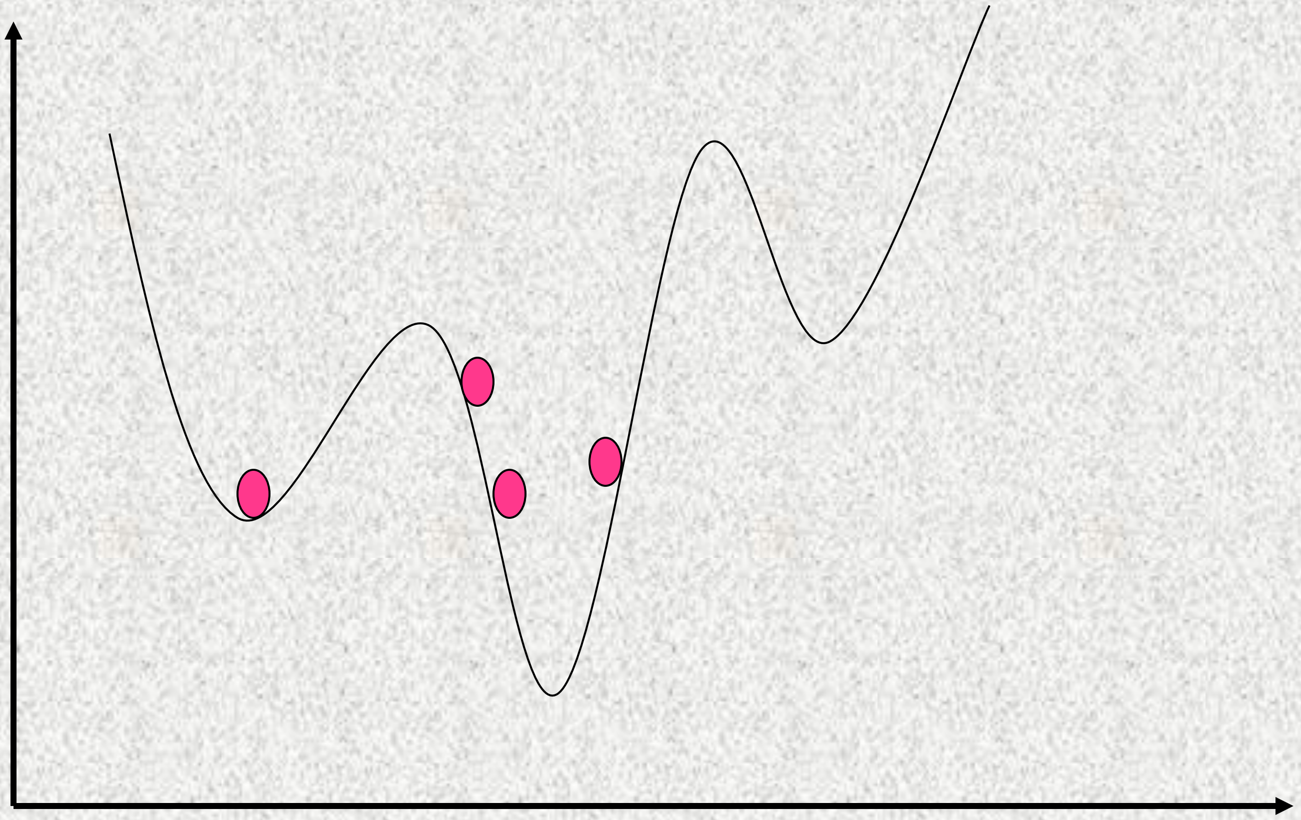
## Genetic Algorithms



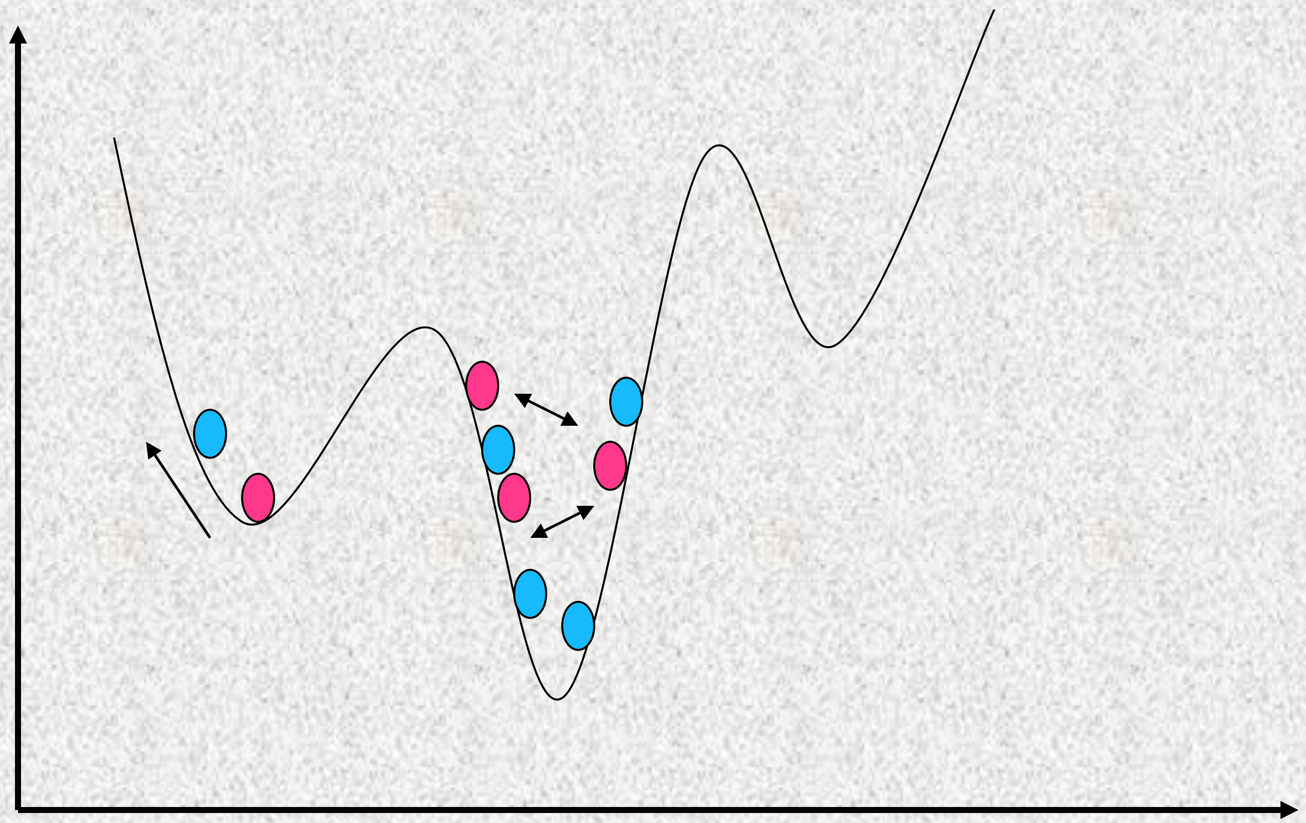
## Genetic Algorithms



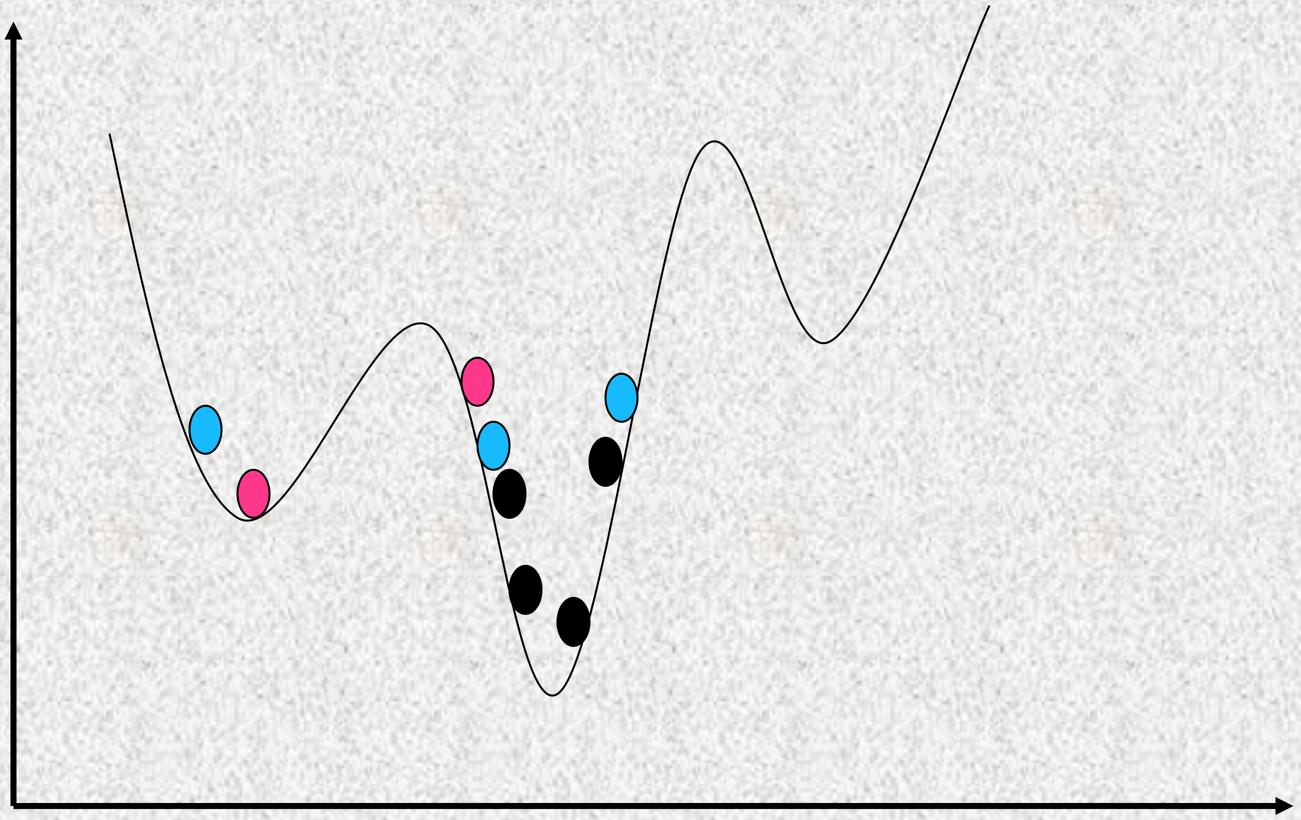
## Genetic Algorithms



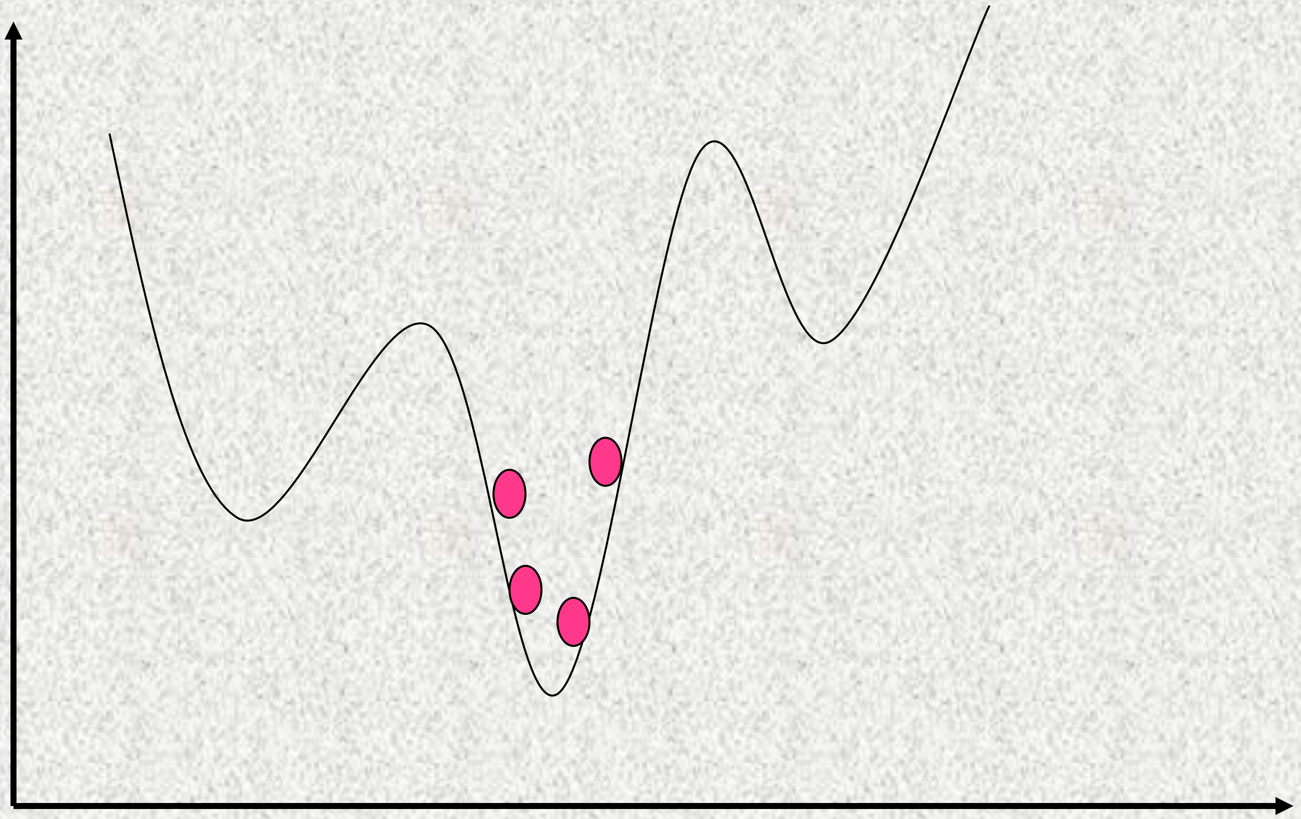
## Genetic Algorithms



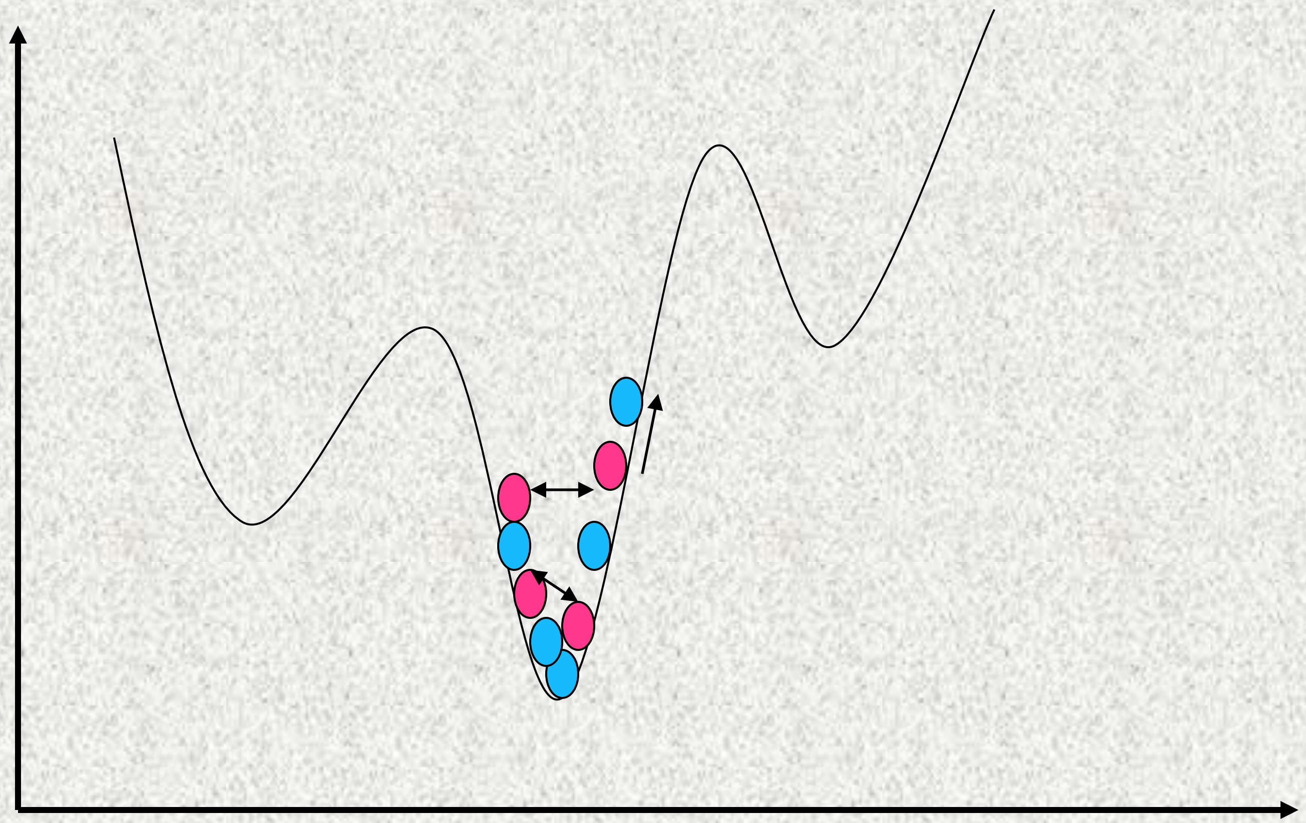
## Genetic Algorithms



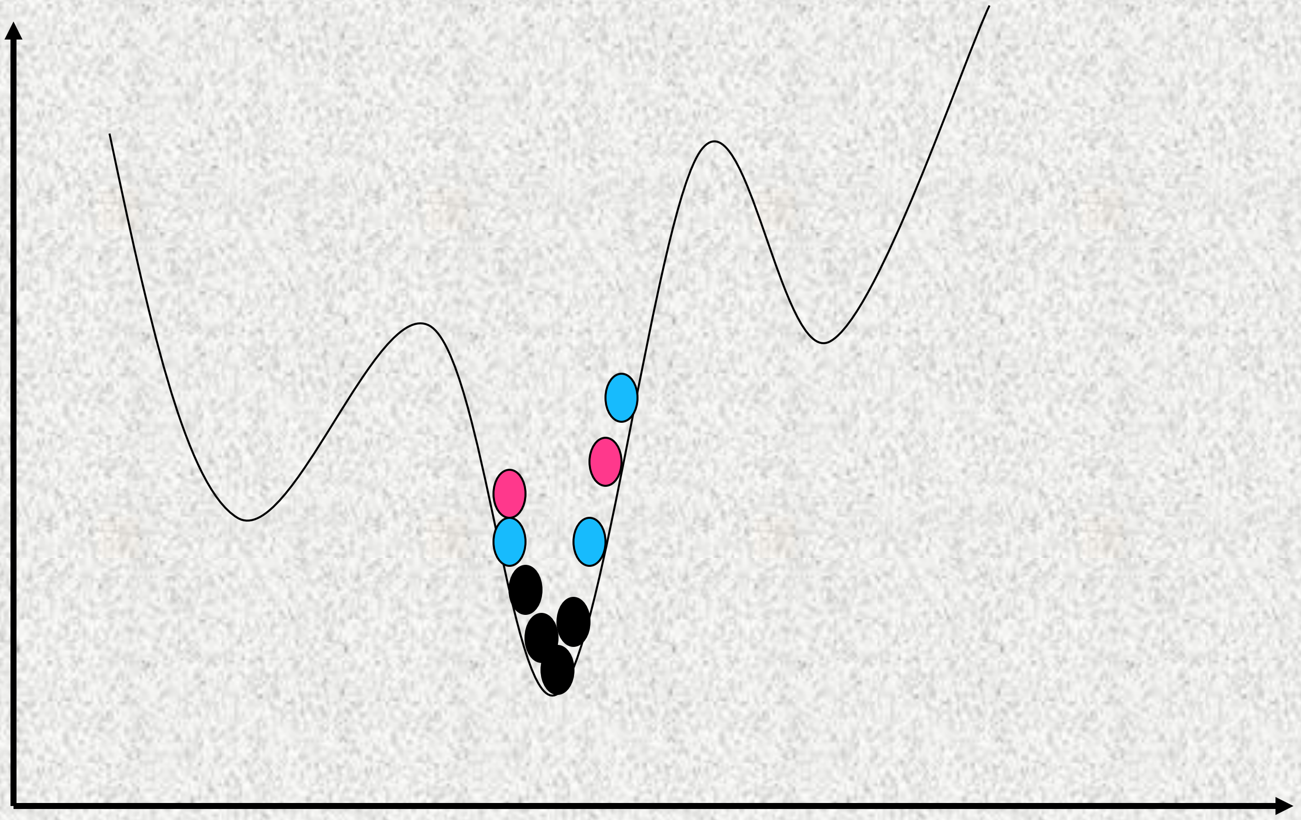
## Genetic Algorithms



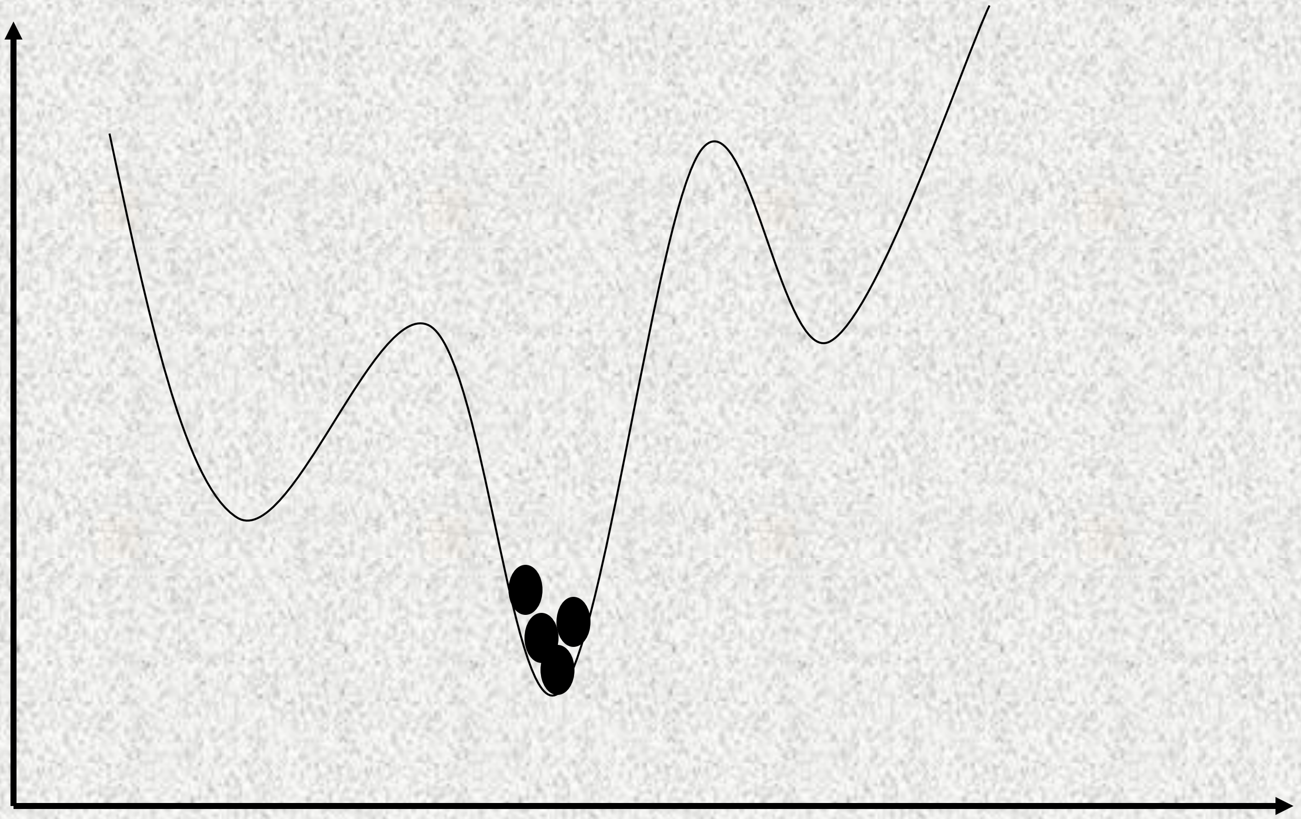
# Genetic Algorithms



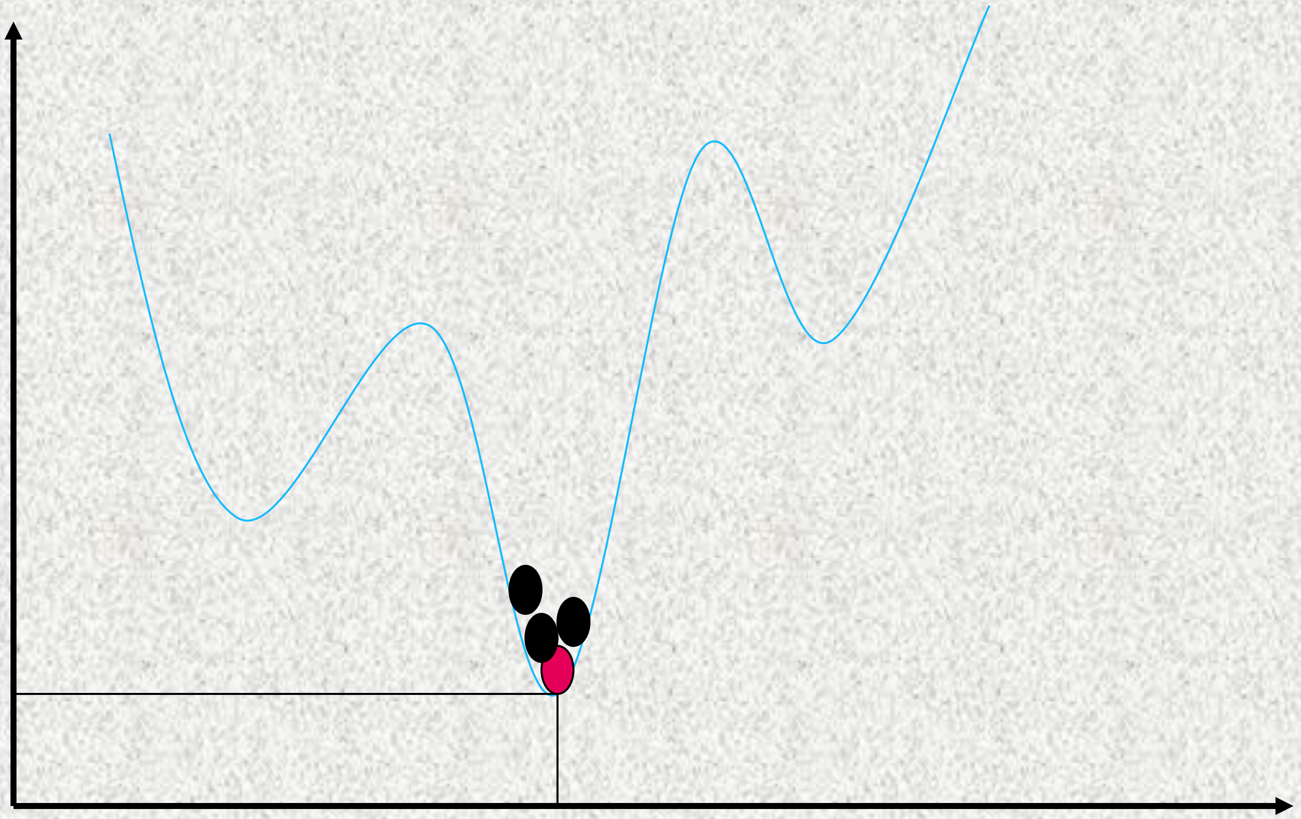
## Genetic Algorithms



# Genetic Algorithms



# Genetic Algorithms



# Summary

Hill climbing is a steady monotonous ascent to better nodes.
Simulated annealing, local beam search, and genetic algorithms are "random" searches with a bias towards better nodes.
All needed very little space which is defined by the population size.
None guarantees to find the globally optimal solution.