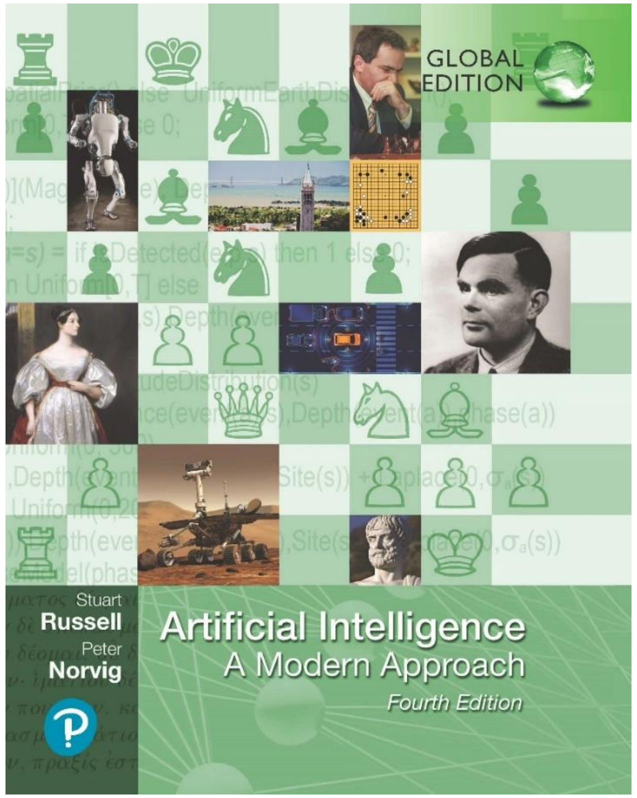
COE 4213564 : Introduction to Artificial Intelligence



Chapter 4

Search in Complex Environments

Fourth Edition, Global Edition

Artificial Intelligence: A Modern Approach

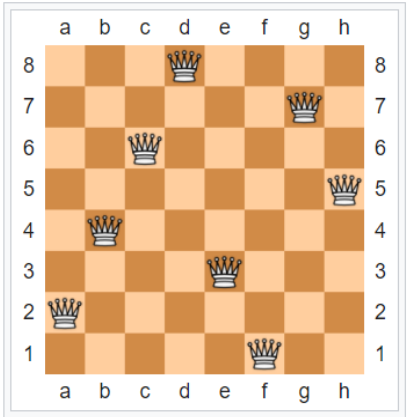
Iterative Improvement Methods

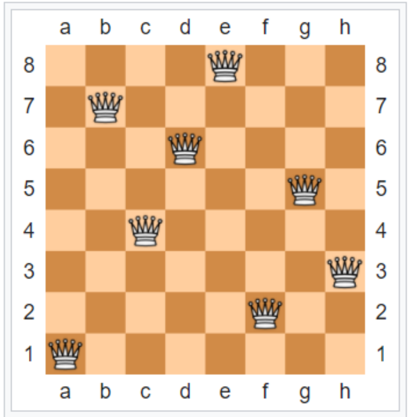
- The search algorithms that we have seen so far are designed to explore search spaces systematically.
  - This systematicity is achieved by keeping one or more paths in memory and by recording which alternatives have been explored at each point along the path.
  - When a goal is found, the path to that goal also constitutes a solution to the problem.
- The best of these methods can currently handle search spaces of up to 10^{100} states / ~1,000 binary variables. (where a set of variables define a state or configuration.)
- What if we have much larger search spaces?
  - Search spaces for some real-world problems may be much larger e.g.,
  - $10^{30,000}$ states as in certain reasoning and planning tasks
  - State space is large/complex and keeping whole frontier in memory is impractical
- Some of these problems can be solved by Iterative Improvement Methods

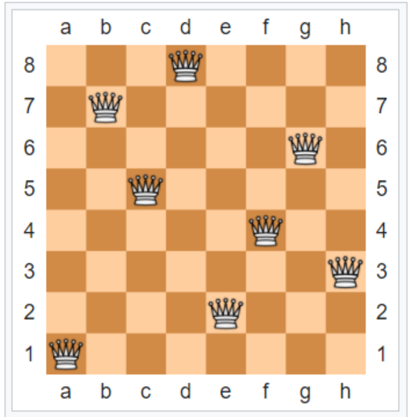
Finding a valid final configuration or state

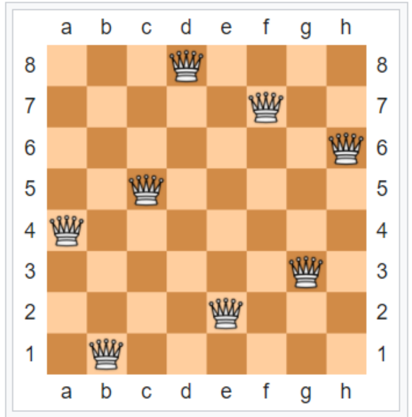
- In many problems, the path to the goal is irrelevant.
- The goal state itself is the solution
- For example, in the 8-queens problem, we care only about finding a valid final configuration (or state) of 8 queens.
- A set of variables define a state or configuration. A valid configuration or state is defined by domains for every variable and constraints among variables.

- The eight queens puzzle is the problem of placing eight chess queens on an 8×8 chessboard so that no two queens threaten each other.
- The eight queens puzzle has 92 distinct solutions.
- If solutions that differ only by the symmetry operations of rotation and reflection of the board are counted as one, the puzzle has 12 solutions.


Solution 1


Solution 2


Solution 3


Solution 4

Iterative Improvement Methods

- In many optimization problems the goal state itself is the solution
- The state space is a set of *complete* configurations
- Search is about
  - finding the optimal configuration (as in TSP) or
  - just a feasible configuration (as in scheduling problems, timetable)
- In such cases, one can use iterative improvement, or local search methods.
- These methods are suitable for problems in which all that matters is the solution state, not the path cost to reach it.
- An evaluation or objective function *h* must be available that measures the quality of each state
- Main Idea: Start with a random initial configuration and make small, local changes to it that improve its quality.
- Many important applications : integrated-circuit design, factory floor layout, job shop scheduling, automatic programming, telecommunications network optimization, crop planning, and portfolio management.

Local Search and Optimization Problems

- Local search algorithms operate by searching from a start state to neighboring states, without keeping track of the paths, nor the set of states that have been reached.
- Two key advantages:
  - they use very little memory; and
  - they can often find reasonable solutions in large or infinite state spaces for which systematic algorithms are unsuitable.
- They are not systematic—they might never explore a portion of the search space where a solution actually resides.
- Local search algorithms can also solve optimization problems, in which the aim is to find the best state according to an objective function.

State-space landscape

- To understand local search, consider the states of a problem laid out in a state-space landscape, as shown in Figure 4.1. Local search algorithms explore this landscape.
- Each point (state) in the landscape has an “elevation” defined by the value of the objective function.
  - If elevation corresponds to an objective function, then the aim is to find the highest peak—a global maximum—and we call the process hill climbing.
  - If elevation corresponds to cost, then the aim is to find the lowest valley—a global minimum—and we call it gradient descent.
- A complete local search algorithm always finds a goal if one exists;
- An optimal algorithm always finds a global minimum/maximum
- Problem: depending on initial state, can get stuck in local maxima/minima

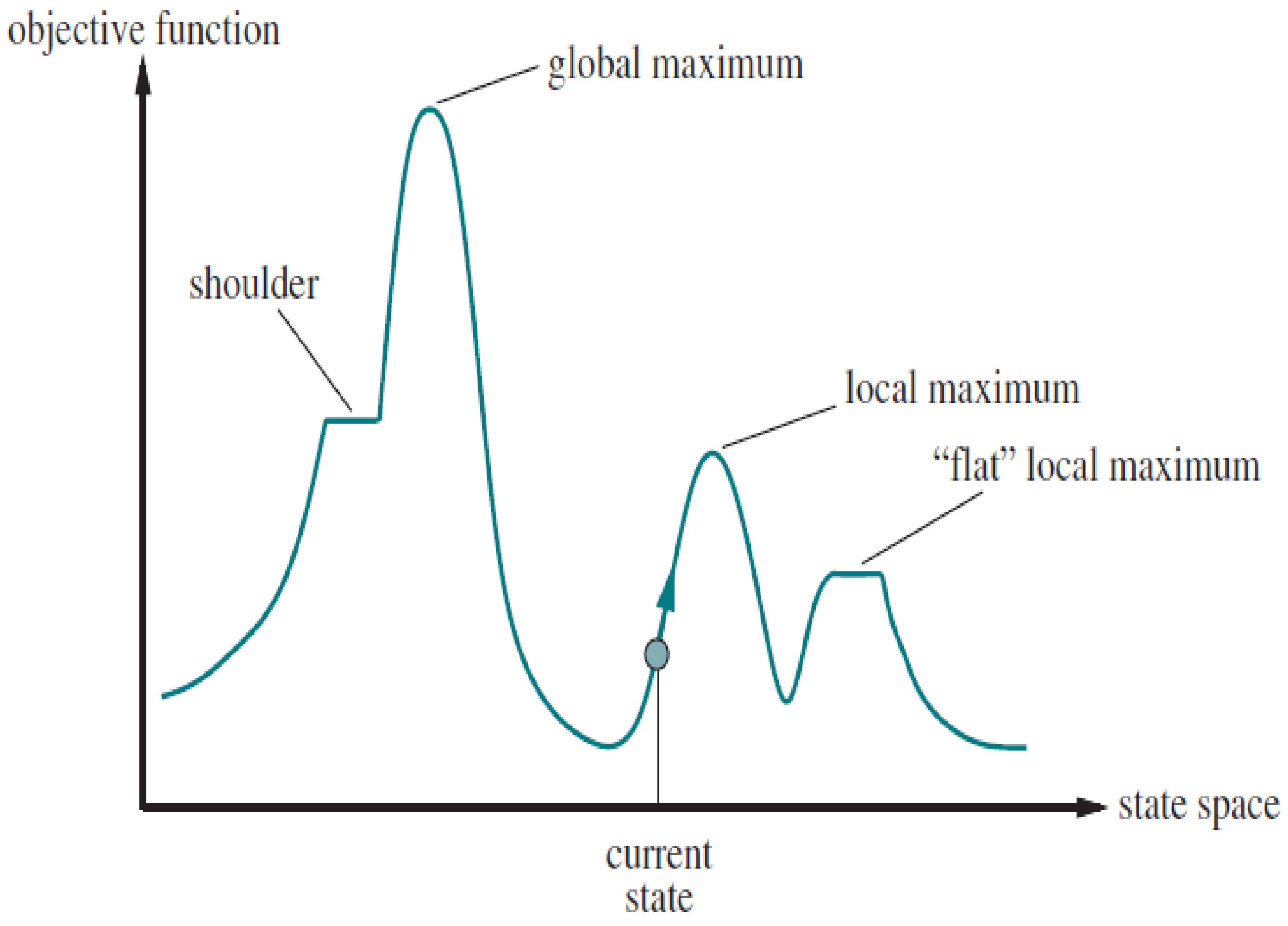


Figure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum.

Exploring the Landscape

- Local Maxima: peaks that aren't the highest point in the space
- Plateaus: the space has a broad flat region that gives the search algorithm no direction (random walk)
- Ridges: flat like a plateau, but with drop-offs to the sides; steps to the North, East, South and West may go down, but a step to the NW may go up.

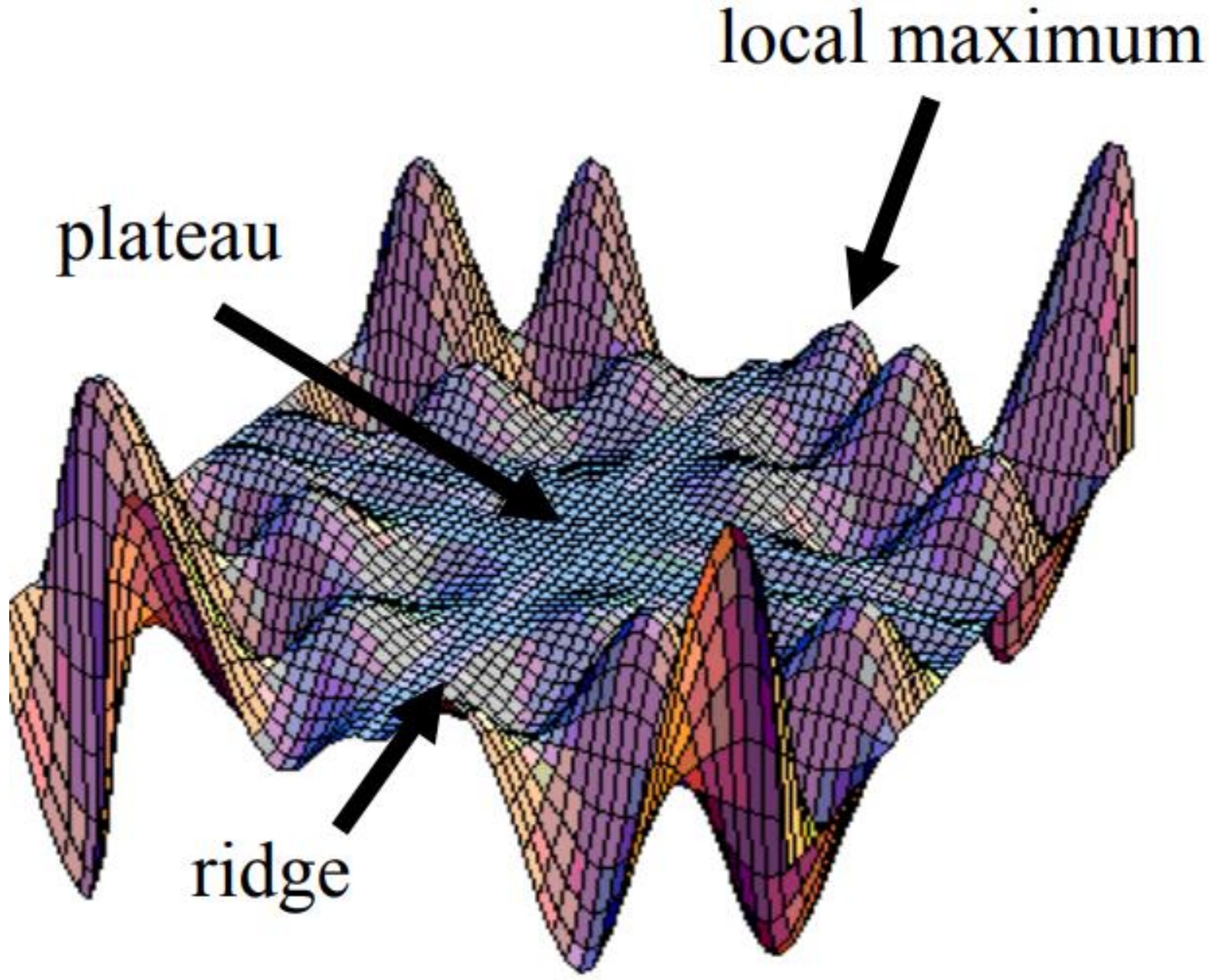
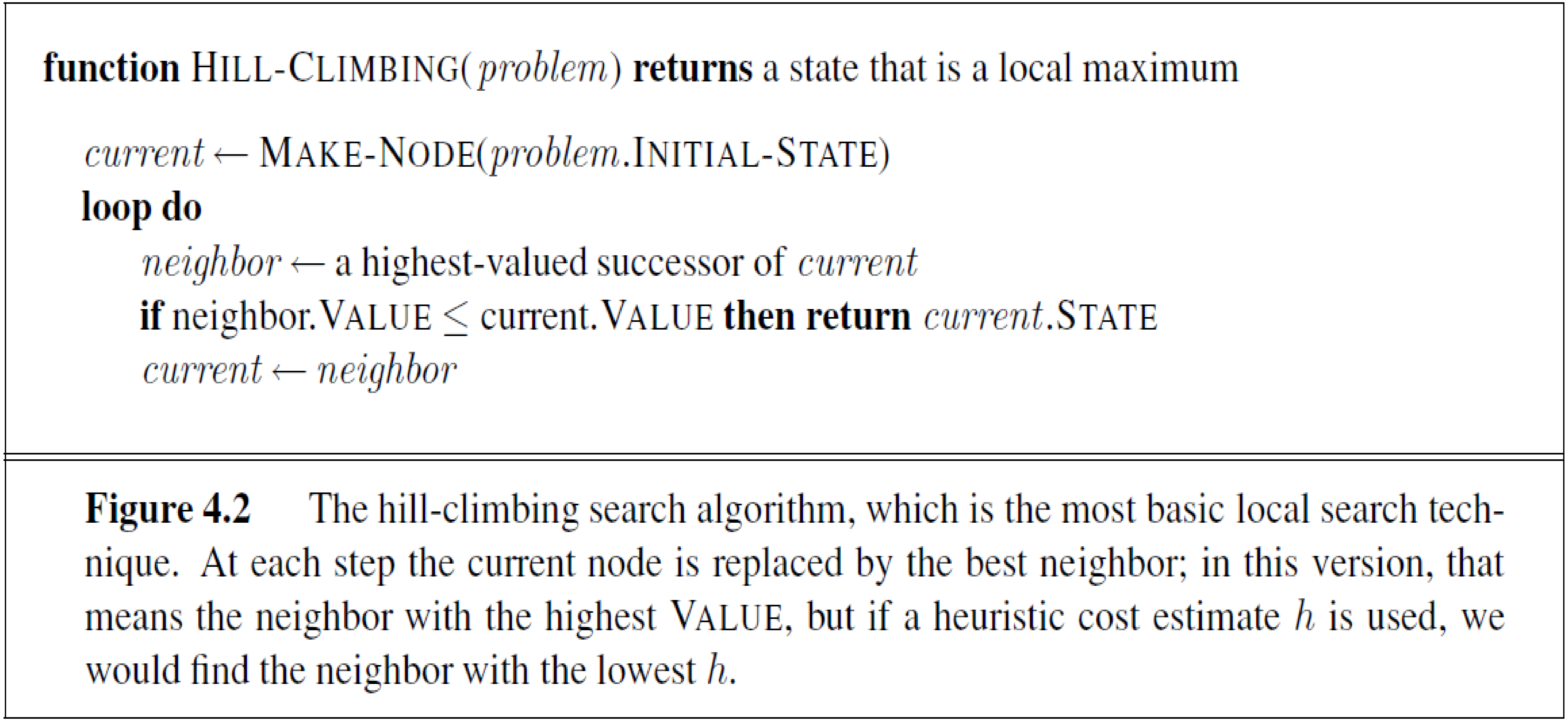


Image from: http://classes.yale.edu/fractals/CA/GA/Fitness/Fitness.html

Hill-climbing search

- Simple, general idea:
  - Start wherever
  - Repeat: move to the best neighboring state
  - If no neighbors better than current, quit

Hill-climbing search



```
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 $\leq$ current.VALUE then return current.STATE
      current ← neighbor
```

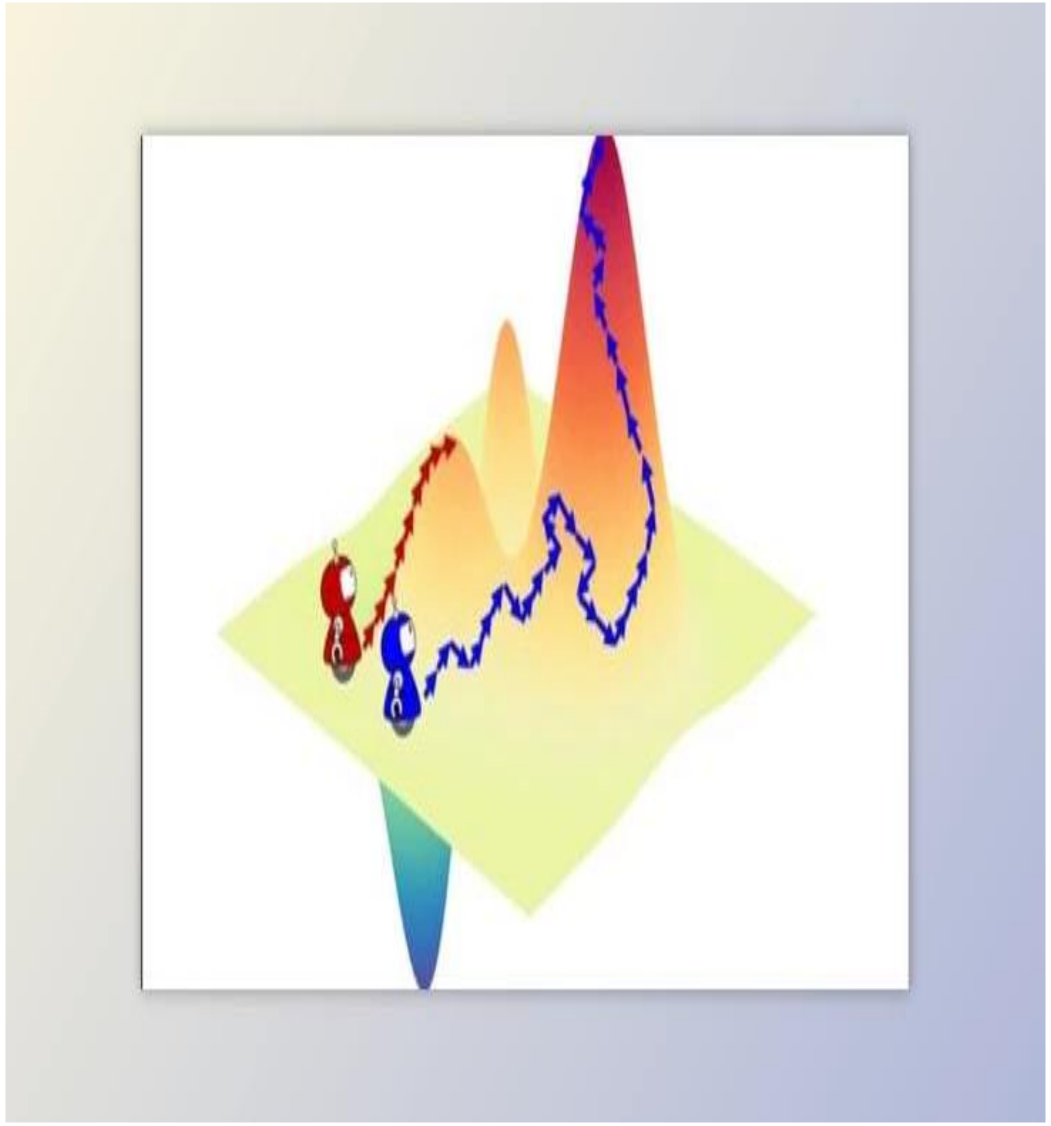
Figure 4.2 The hill-climbing search algorithm, which is the most basic local search technique. At each step the current node is replaced by the best neighbor; in this version, that means the neighbor with the highest VALUE, but if a heuristic cost estimate h is used, we would find the neighbor with the lowest h.

Hill-climbing search

- It is simply a loop that continually moves in the direction of increasing value—that is, uphill. It terminates when it reaches a “peak” where no neighbor has a higher value.
- The algorithm does not maintain a search tree, so the data structure for the current node need only record the state and the value of the objective function.
- Hill climbing does not look ahead beyond the immediate neighbors of the current state. (This resembles trying to find the top of Mount Everest in a thick fog while suffering from amnesia.)
- Hill climbing is sometimes called greedy local search because it grabs a good neighbor state without thinking ahead about where to go next.

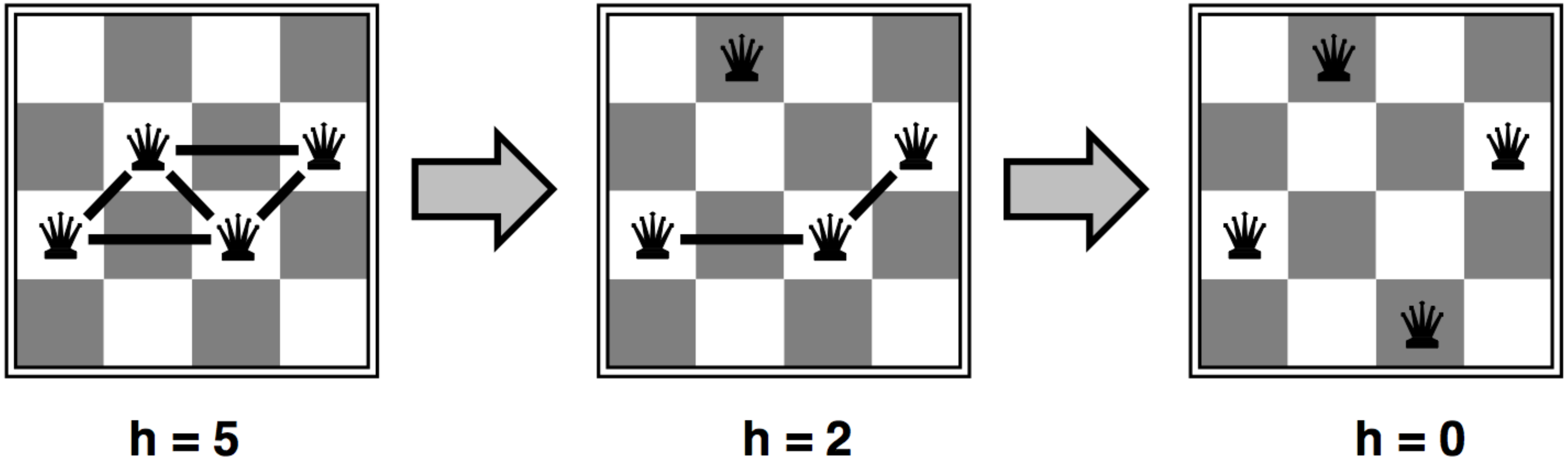
The heuristic objective(cost) function

- An evaluation or objective function *h* must be available that measures the quality of each state
- Main Idea: Start with a random initial configuration and make small, local changes to it that improve its quality.
- Ideally, the evaluation function *h* should be monotonic: the closer a state to an optimal goal state the better its h-value.
- Each state can be seen as a point on a surface.
- The search consists in moving on the surface, looking for its highest peaks: the optimal solutions.
- Similar to Greedy search in that it uses *h*, but does not allow backtracking or jumping to an alternative path since it doesn't "remember" where it has been.

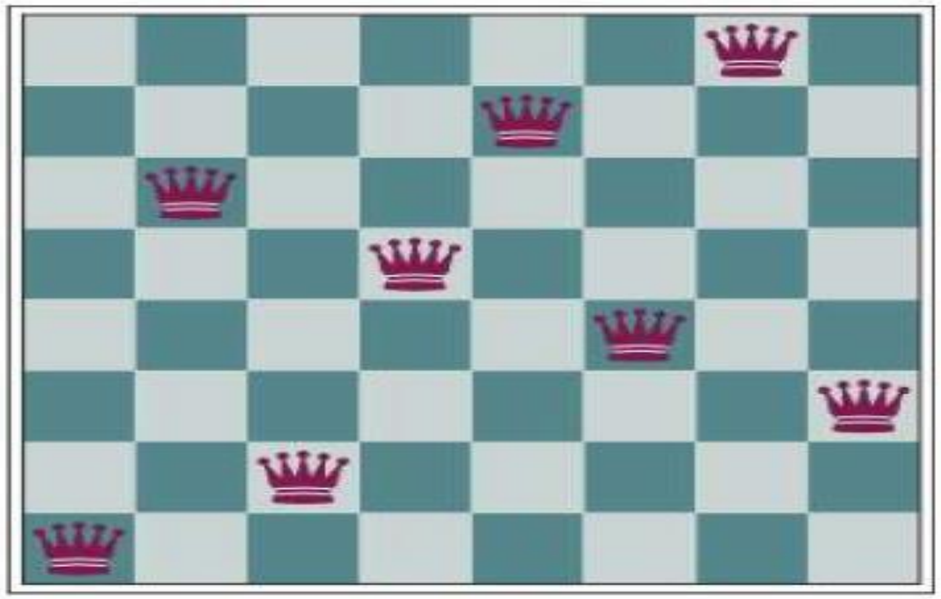


Ex: Heuristic for *n*-queens problem

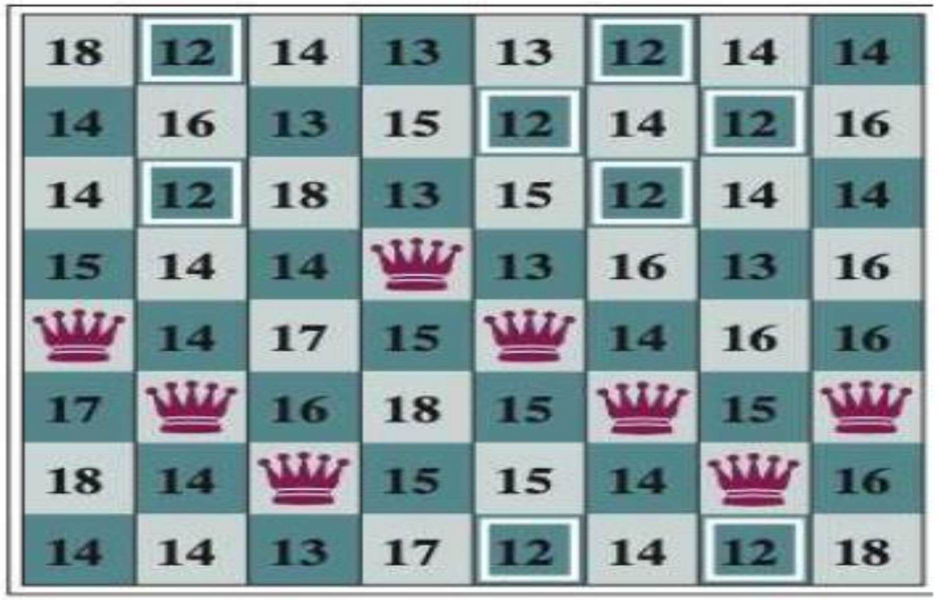
- Goal: n queens on board with no ***conflicts***, i.e., no queen attacking another
- Put n queens on an n × n board with no two queens on the same row, column, or diagonal
- States: n queens on board, one per column
- Actions: move a queen in its column
- Heuristic value function h: number of conflicts (the number of pairs of queens that are attacking each other; this will be zero only for solutions.)
- Strategy: Move a queen to reduce number of conflicts



Ex: Heuristic for *n*-queens problem



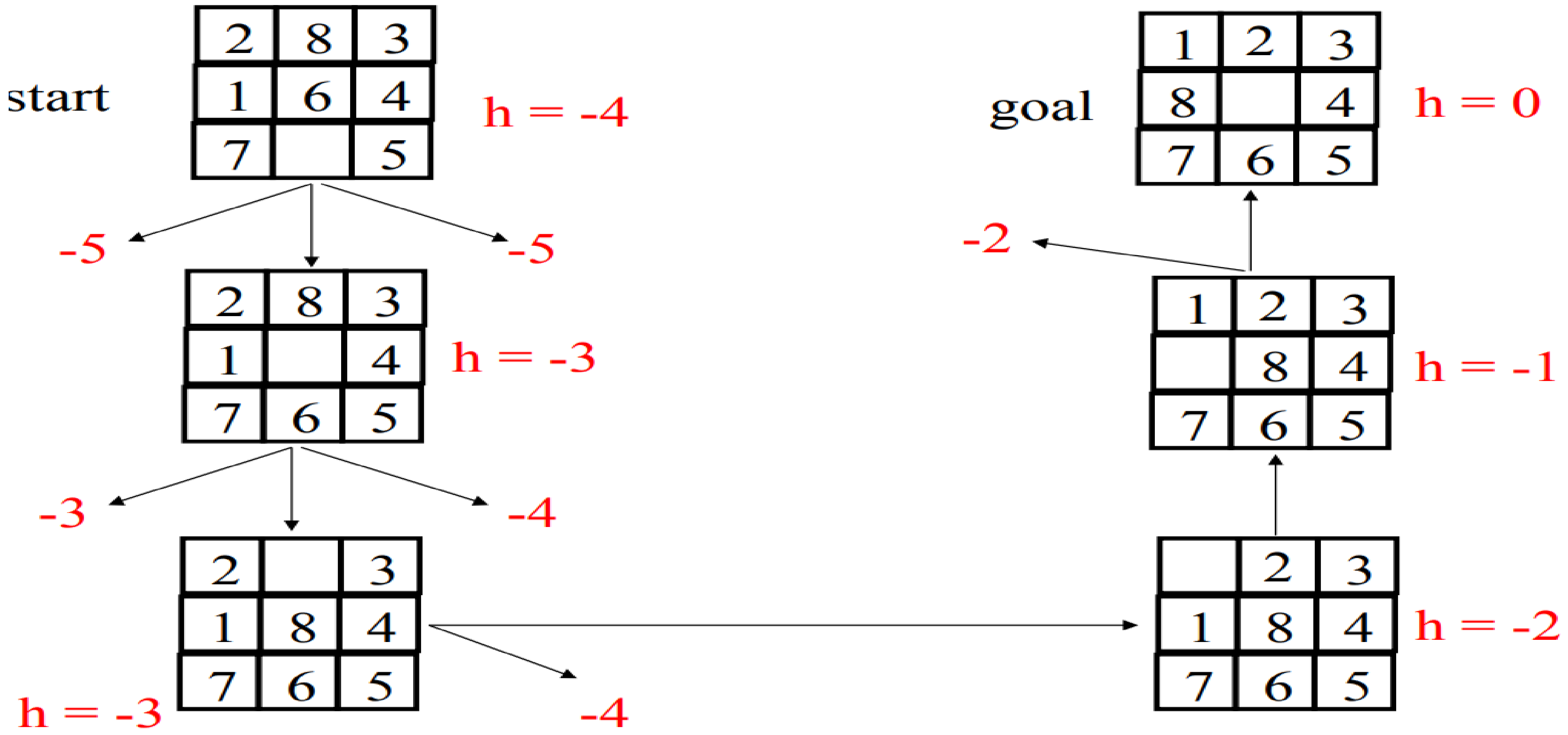
(a)



(b)

Figure 4.3 (a) The 8-queens problem: place 8 queens on a chess board so that no queen attacks another. (A queen attacks any piece in the same row, column, or diagonal.) This position is almost a solution, except for the two queens in the fourth and seventh columns that attack each other along the diagonal. (b) An 8-queens state with heuristic cost estimate $h = 17$. The board shows the value of h for each possible successor obtained by moving a queen within its column. There are 8 moves that are tied for best, with $h = 12$. The hill-climbing algorithm will pick one of these.

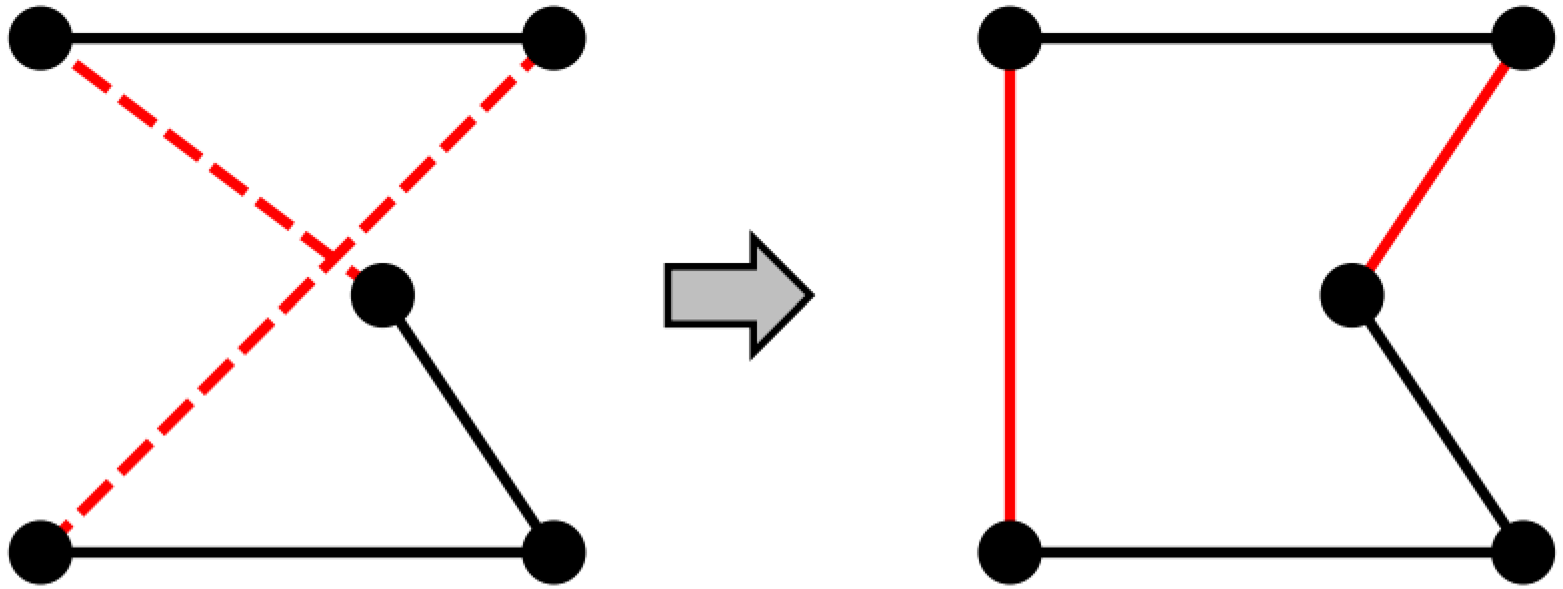
Ex:



$f(n) = -$**(number of tiles out of place)**

Example: TSP

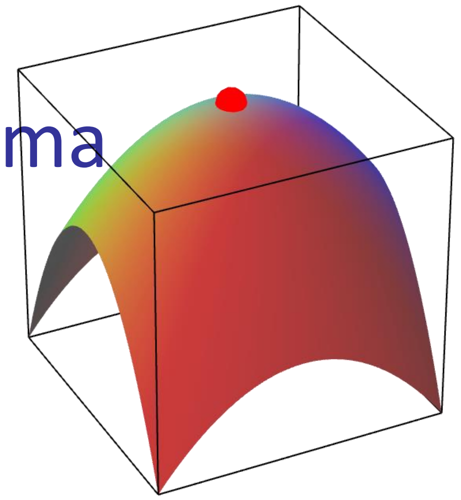
- Travelling Salesperson Problem (TSP) : Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?" It is an NP-hard problem in combinatorial optimization, important in theoretical computer science and operations research.
- Pairwise exchange method: The pairwise exchange or 2-opt technique involves iteratively removing two edges and replacing these with two different edges that reconnect the fragments created by edge removal into a new and shorter tour.
- *h* = length of the tour
- Strategy: Start with any complete tour, perform pairwise exchanges

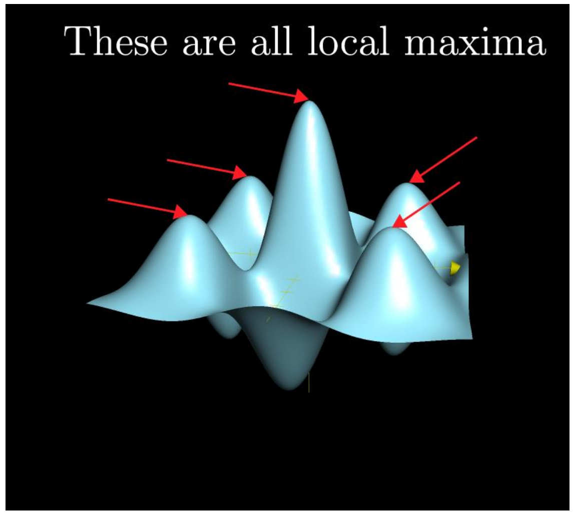


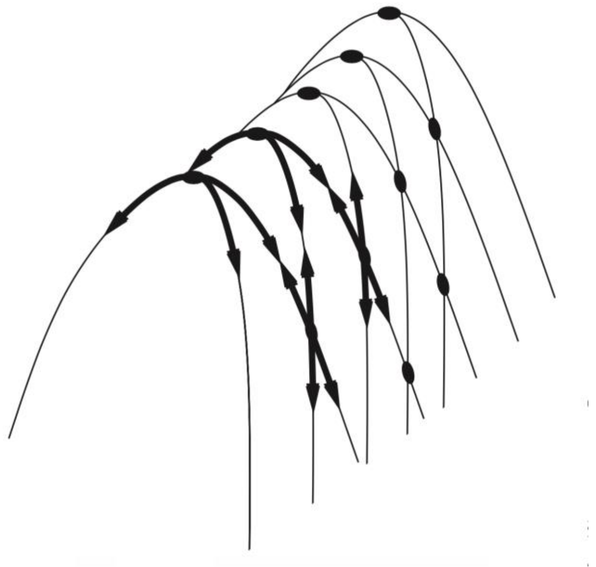
Hill Climbing Drawbacks

It is a greedy algorithm that often perform quite well. Unfortunately, hill climbing can get stuck for any of the following reasons: Local maxima, Ridges (a sequence of local maxima), Plateaus (a flat area of the state-space landscape).

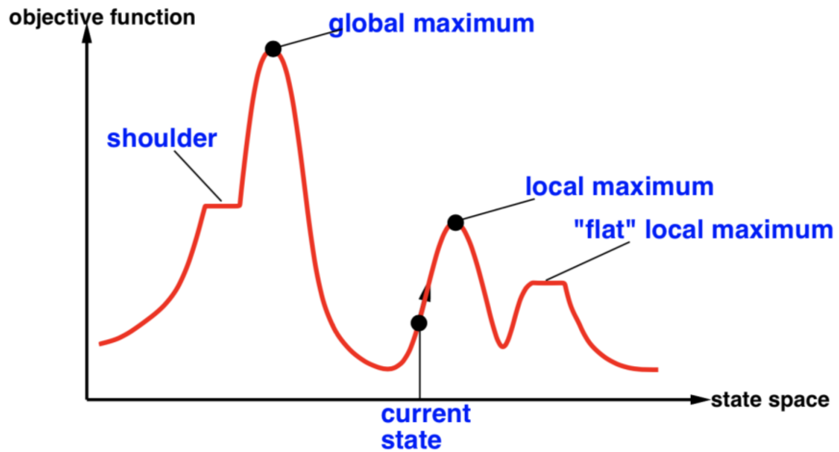
Local maxima







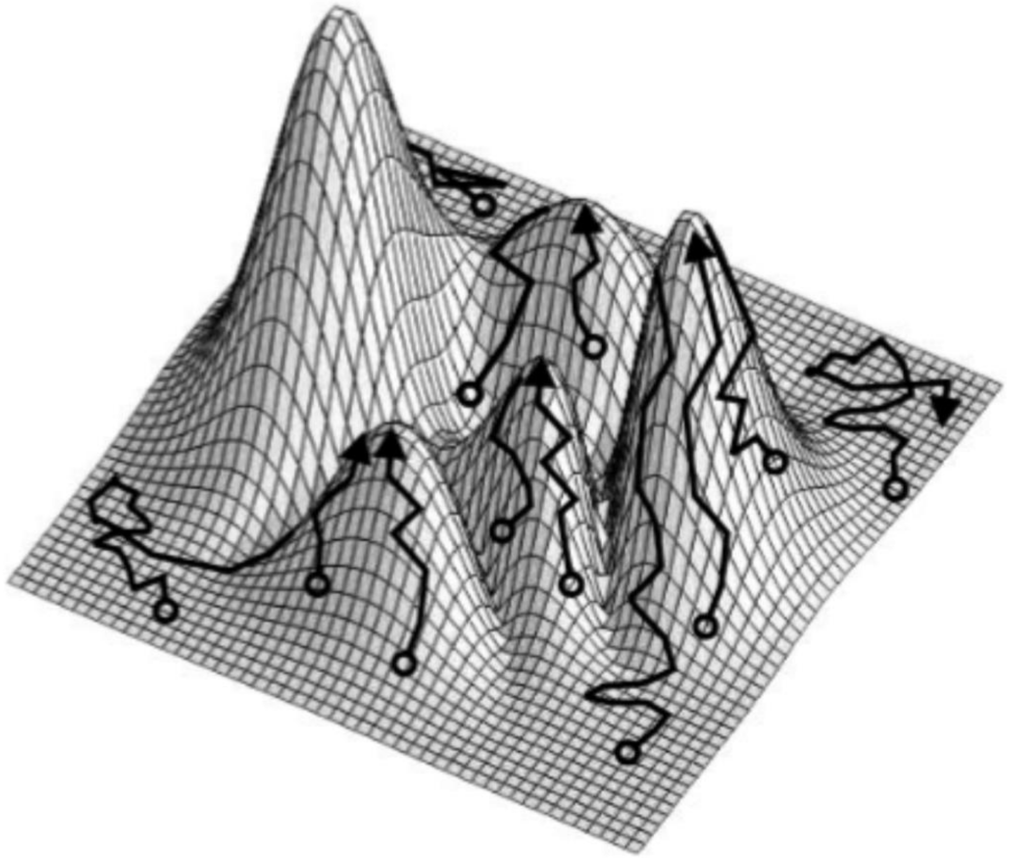
Plateau

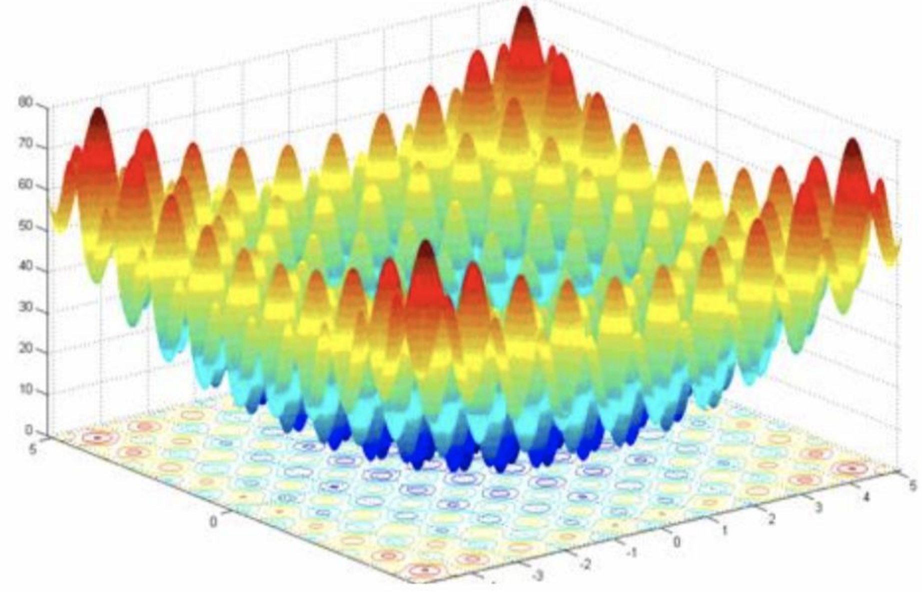


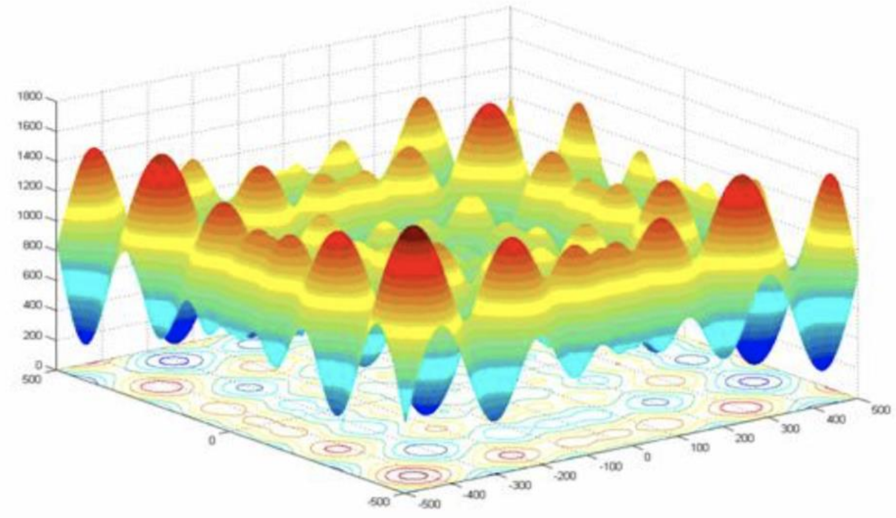
Diagonal ridges:

Figure 4.4 Illustration of why ridges cause difficulties for hill climbing. The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available actions point downhill.

Trajectories, difficulties

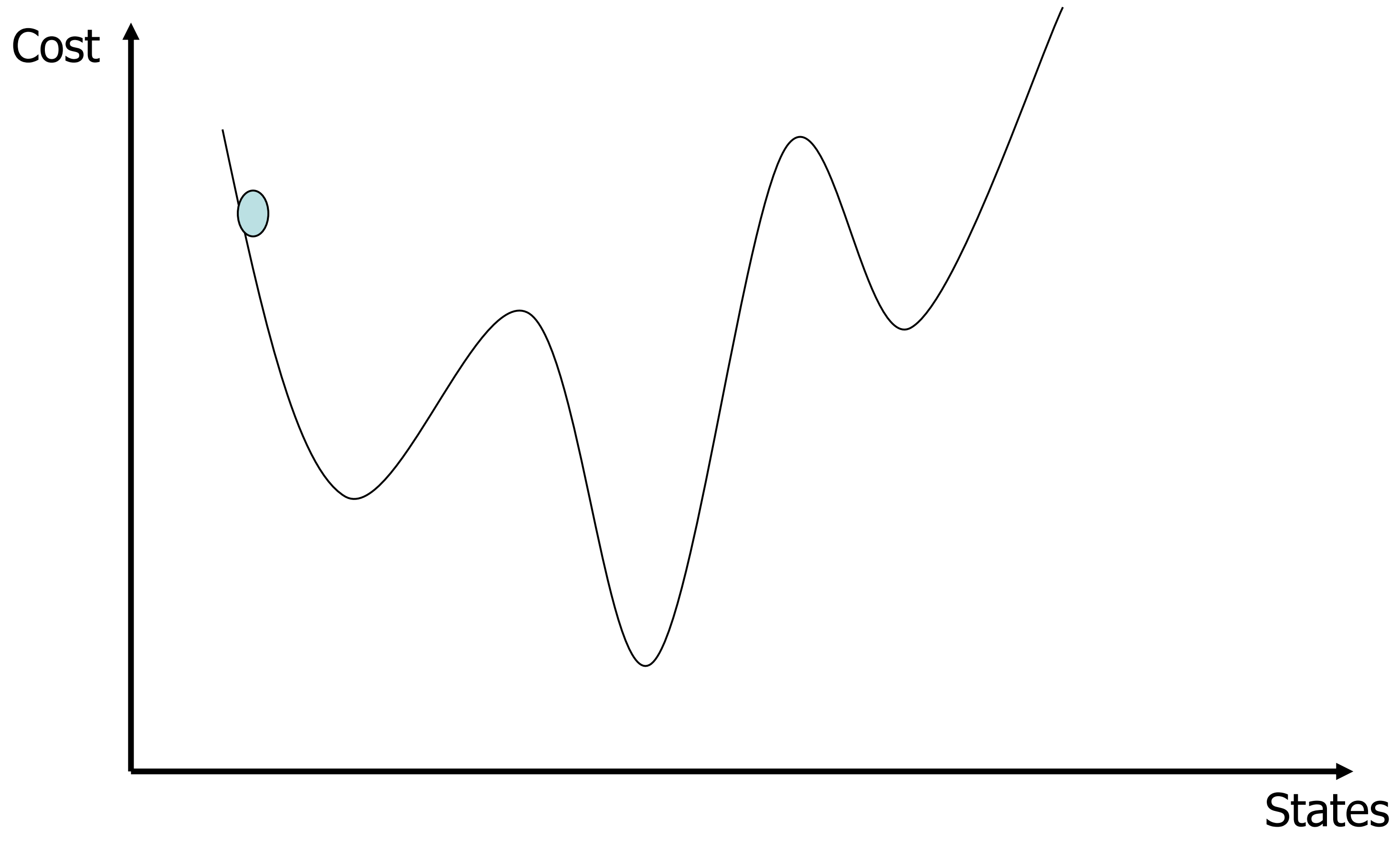




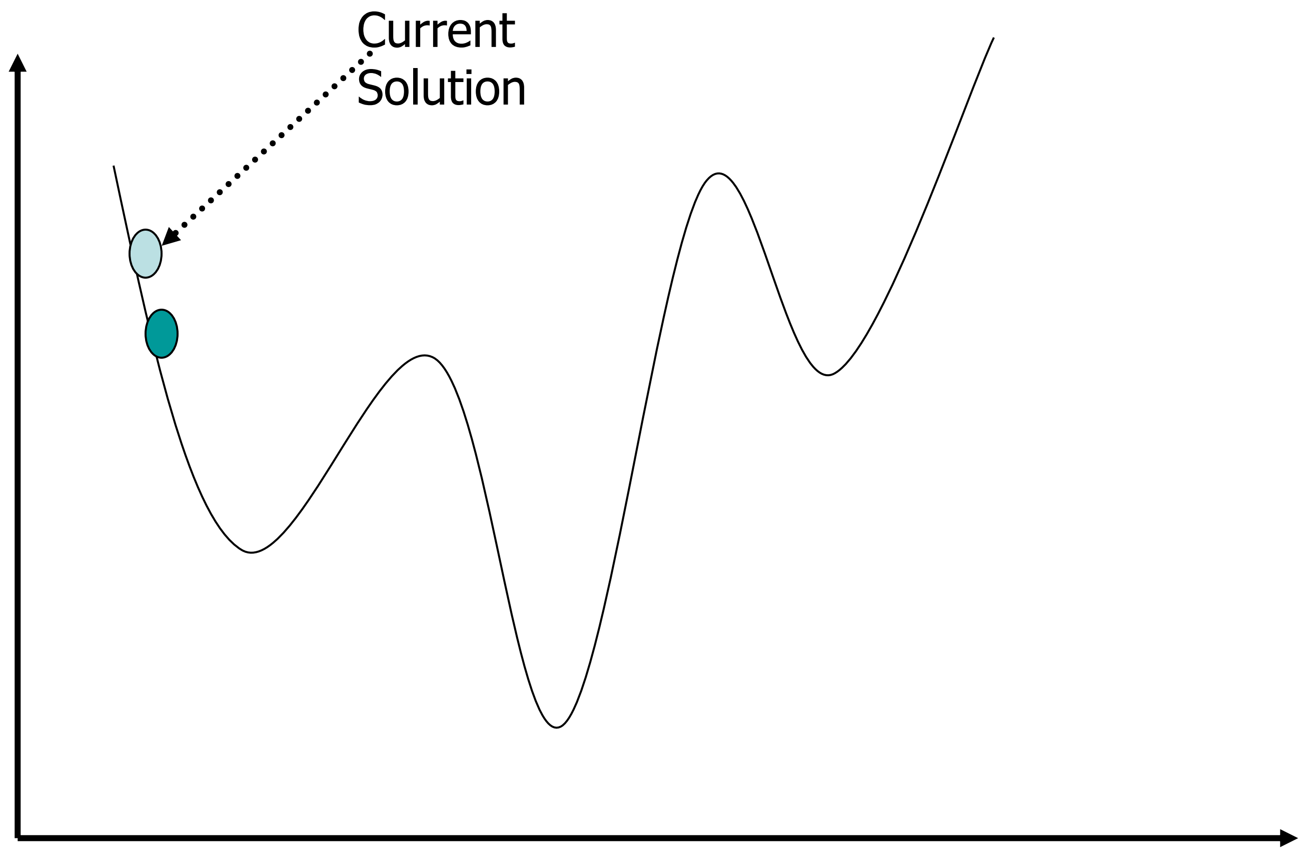


Hill Climbing

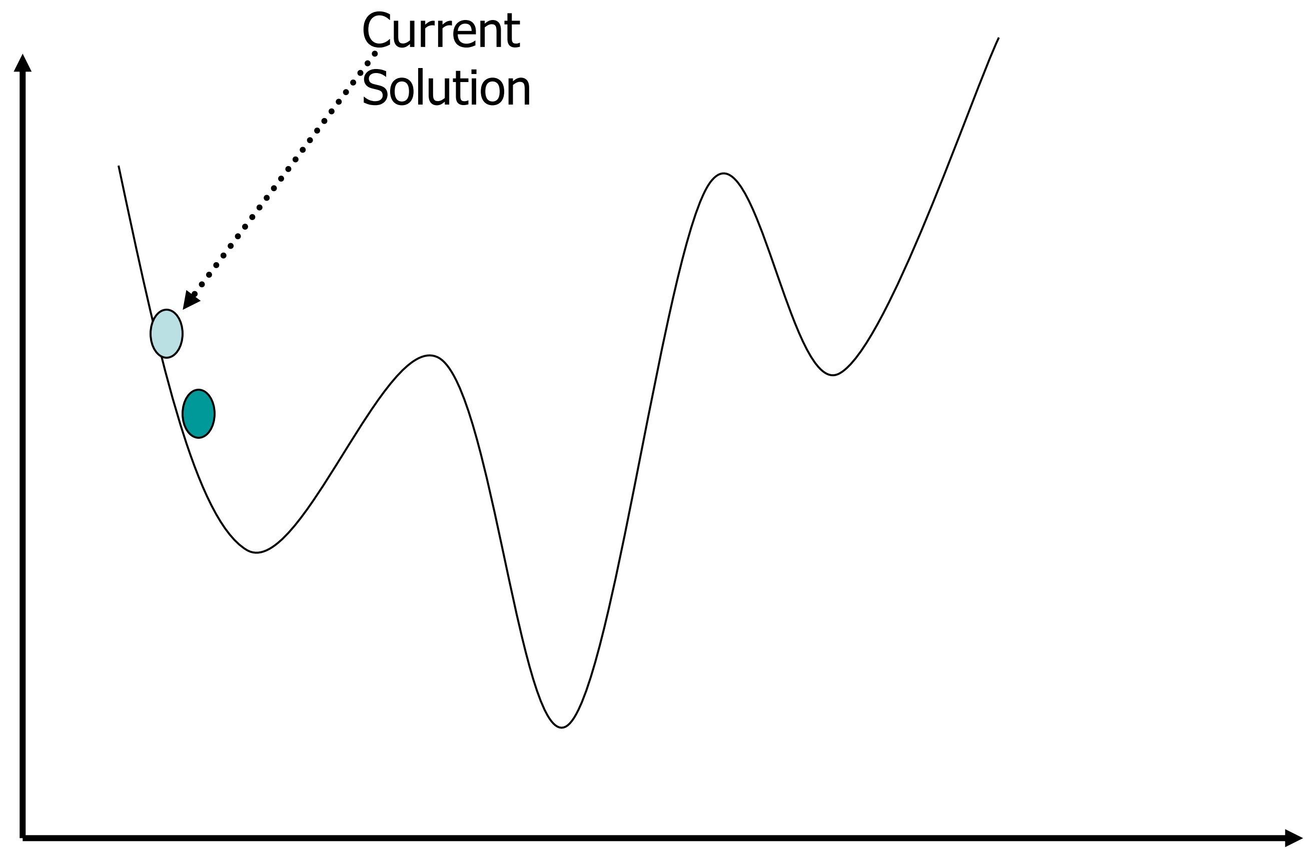
- local search can get stuck on a local maximum/minimum and not find the optimal solution



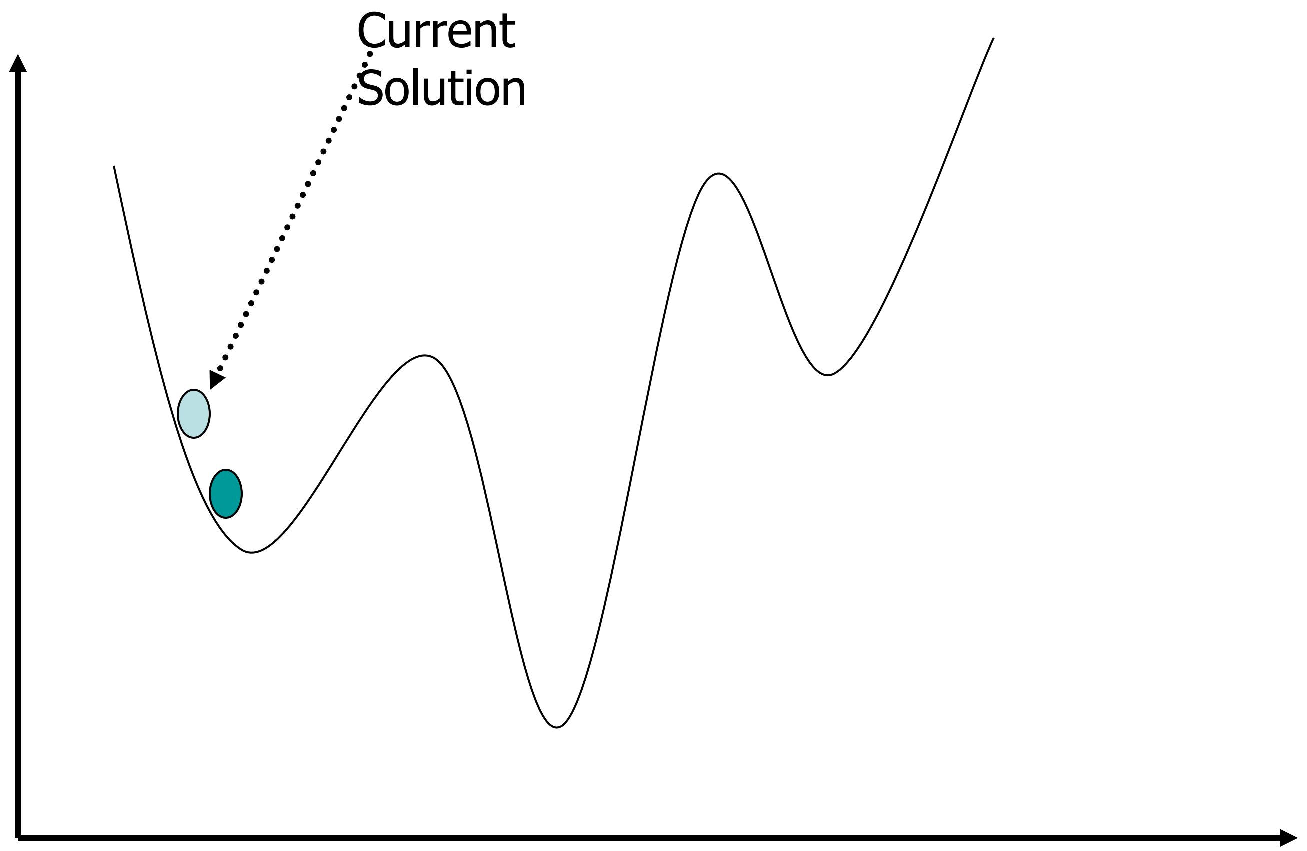
Hill Climbing



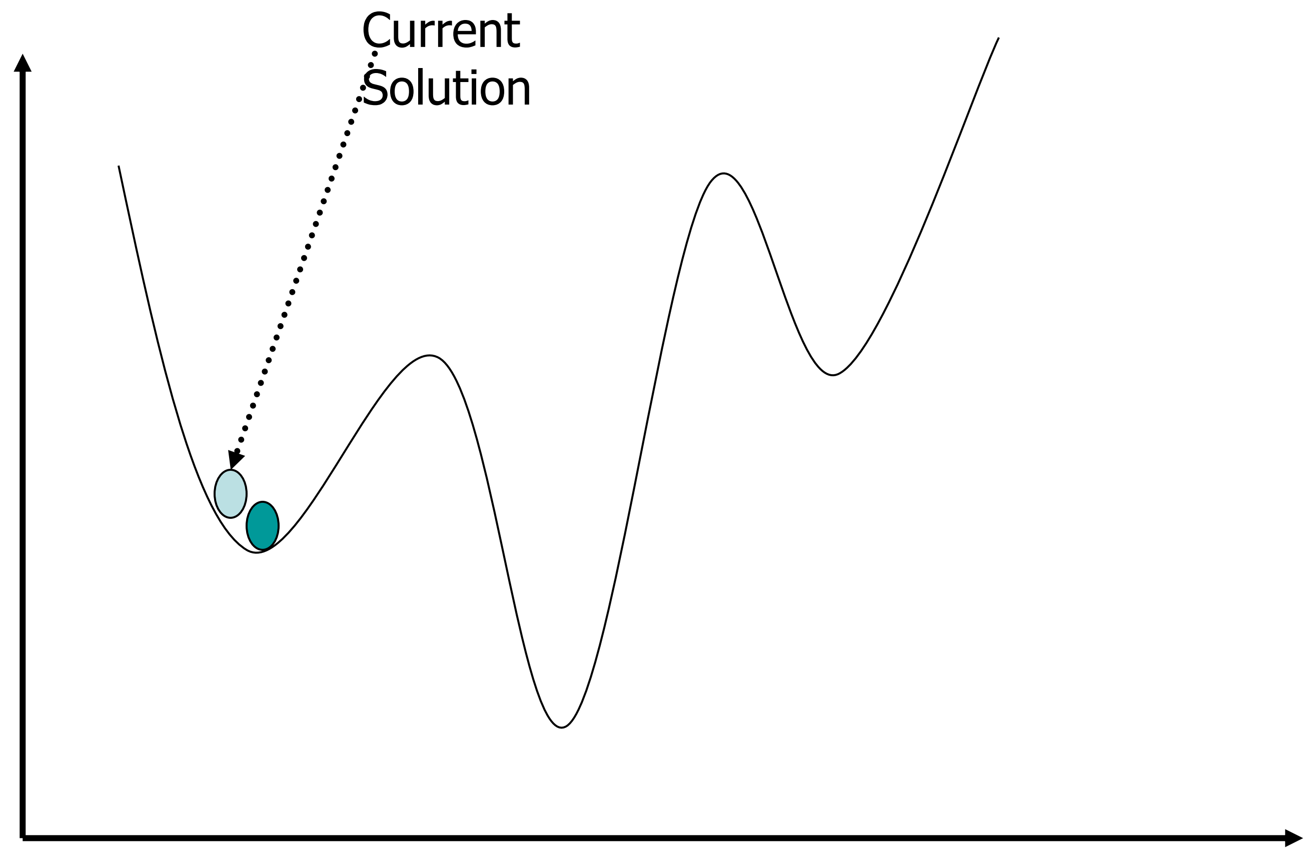
Hill Climbing



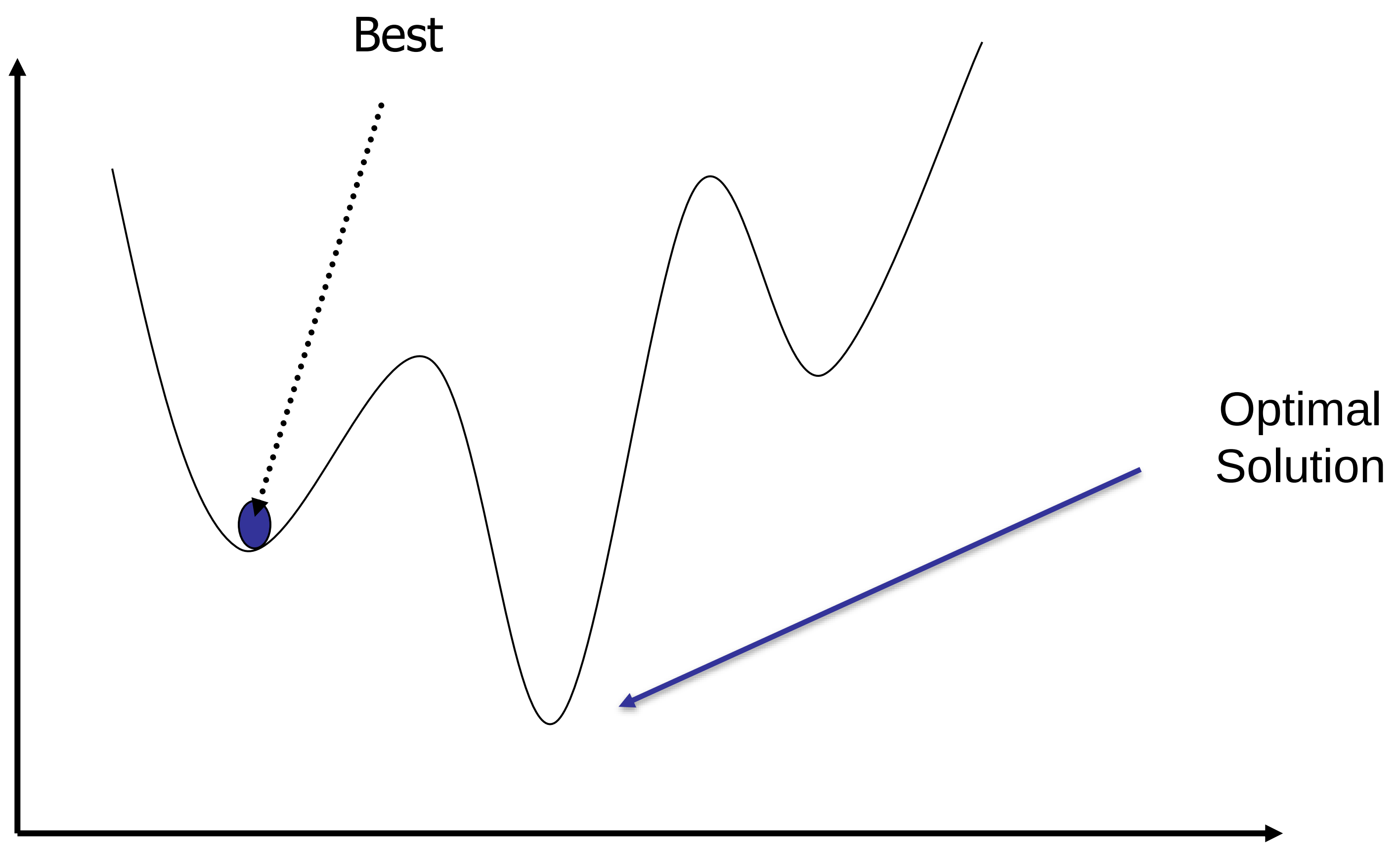
Hill Climbing



Hill Climbing



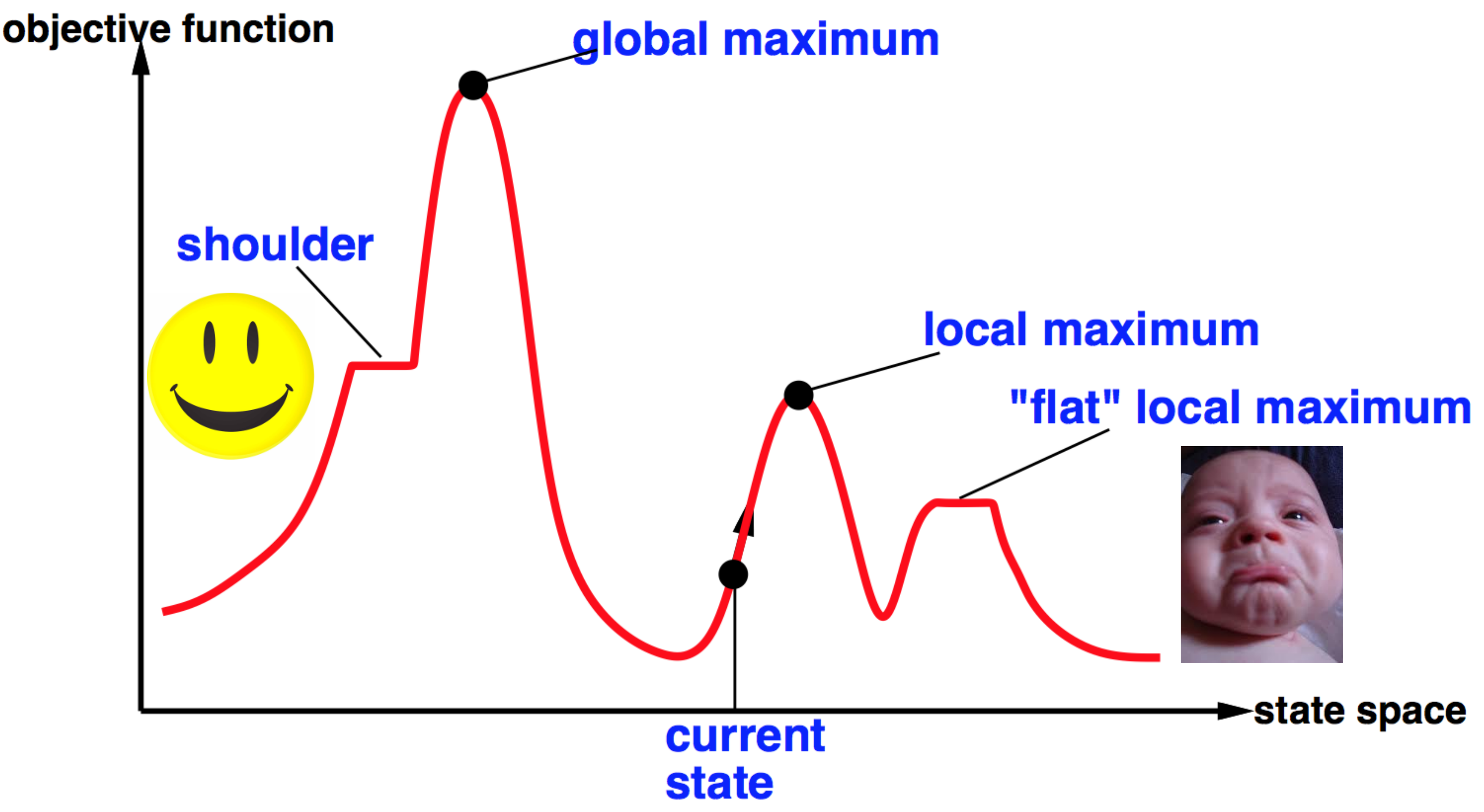
Hill Climbing



Hill-climbing search

- It is a greedy algorithm that often perform quite well. Hill climbing can make rapid progress toward a solution because it is usually quite easy to improve a bad state.
- Unfortunately, hill climbing can get stuck for any of the following reasons: Local maxima, Ridges (a sequence of local maxima), Plateaus (a flat area of the state-space landscape).
- How could we solve more problems?
  - One answer is to keep going when we reach a plateau—to allow a sideways move in the hope that the plateau is really a shoulder, from which progress is possible.
  - Stochastic hill climbing chooses at random from among the uphill moves; the probability of selection can vary with the steepness of the uphill move.
  - First-choice hill climbing implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state.
  - Another variant is random-restart hill climbing, which adopts the adage, “If at first you don’t succeed, try, try again.” It conducts a series of hill-climbing searches from randomly generated initial states, until a goal is found.

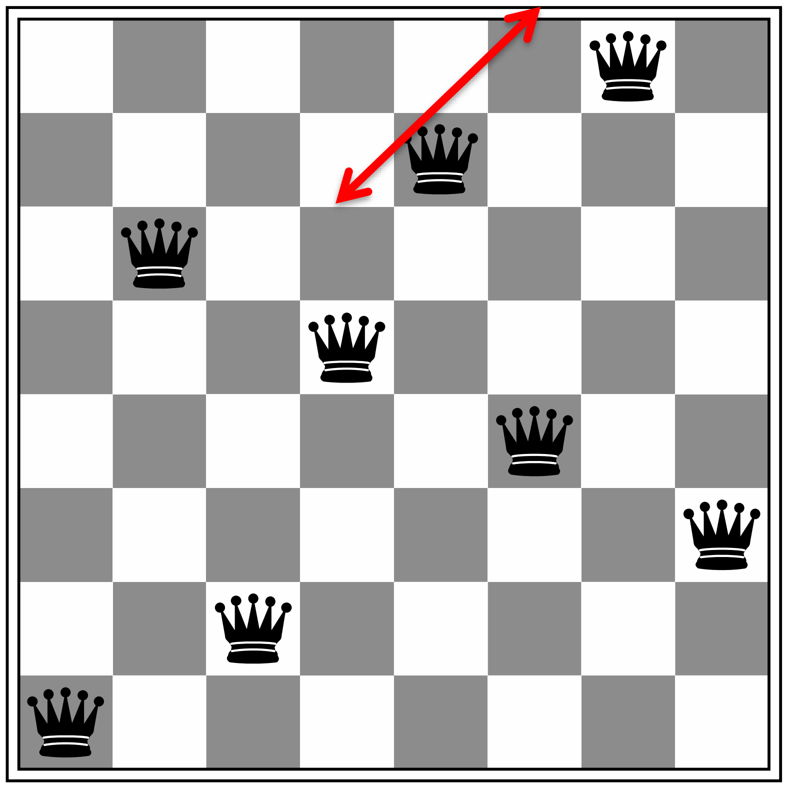
Global and local maxima



- Random-restart hill climbing overcomes local maxima—trivially complete (find global optimum)
- Random sideways moves 😊 escape from shoulders ☹ loop on flat maxima

Sideways move on the 8-queens problem

- Starting from a randomly generated 8-queens state, steepest-ascent hill climbing gets stuck 86% of the time, solving only 14% of problem instances.
- We could allow up to, say, 100 consecutive sideways moves in the 8-queens problem. This raises the percentage of problem instances solved by hill climbing from 14% to 94%.
- Success comes at a cost: the algorithm averages roughly 21 steps for each successful instance and 64 for each failure.



The success of hill climbing

- The success of hill climbing depends very much on the shape of the state-space landscape: if there are few local maxima and plateaux, random-restart hill climbing will find a good solution very quickly.
- On the other hand, many real problems have a landscape that looks more like a widely scattered family of balding porcupines on a flat floor
- NP-hard problems typically have an exponential number of local maxima to get stuck on.
- Despite this, a reasonably good local maximum can often be found after a small number of restarts.

argmax and argmin notation

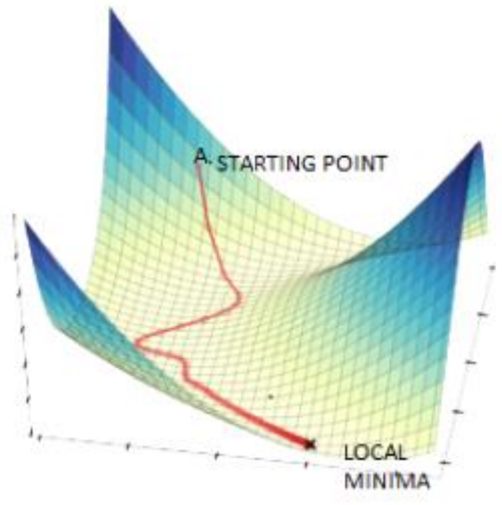
- The argmax and argmin concept is common in many AI and machine learning algorithms
- See argmax on Wikipedia (argmin is similar)

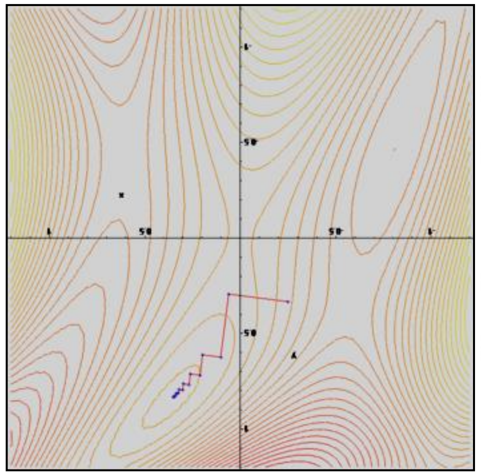
$$\arg\max_x f(x) := \{x \mid \forall y : f(y) \le f(x)\}.$$

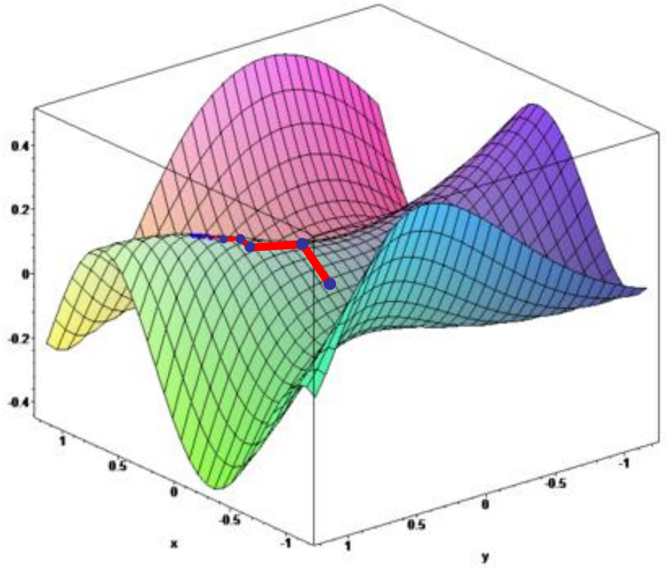
- Argmax$_x$ f(x) finds the value of x for which f(x) is largest

Gradient ascent / descent

- Gradient descent procedure for finding the arg_x *min f(x)*
  - choose initial x_0 randomly
  - repeat
    - $x_{i+1} \leftarrow x_i - \eta f'(x_i)$
  - until the sequence $x_0, x_1, \ldots, x_i, x_{i+1}$ converges
- Step size η (eta) is small (perhaps 0.1 or 0.05)
- Often used in machine learning algorithms

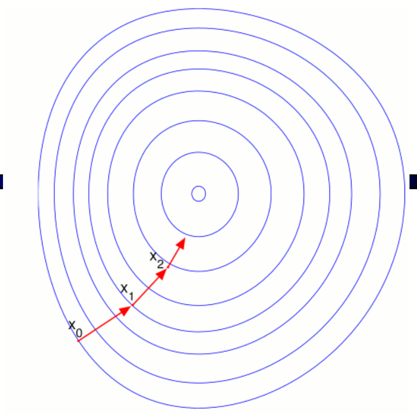






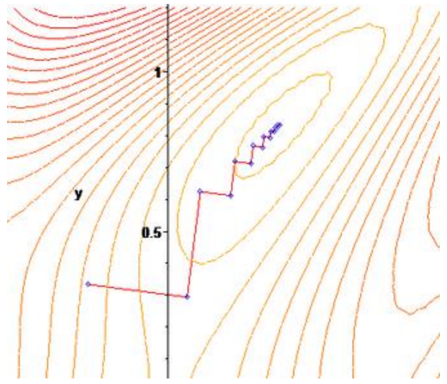
Images from http://en.wikipedia.org/wiki/Gradient_descent

Gradient descent



Hill-climbing in continuous spaces

Gradient = the most direct direction up-hill in the objective (cost) function, so its negative minimizes the cost function.



* Assume we have some cost-function: $J(x_1, x_2, \ldots, x_n)$
and we want minimize over continuous variables $x_1,x_2,..,x_n$

1. Compute the *gradient :* $$\frac{\partial}{\partial x_i} J(x_1, \ldots, x_n) \qquad \forall i$$

2. Take a small step downhill in the direction of the gradient: $$x_i' = x_i - \lambda \frac{\partial}{\partial x_i} J(x_1, \ldots, x_n)$$

3. Check if $$J(x_1', \ldots, x_n') < J(x_1, \ldots, x_n)$$

(or, Armijo rule, etc.)

4. If true then accept move, if not "reject".

(decrease step size, etc.)

5. Repeat.

Gradient descent

Hill-climbing in continuous spaces

- How do I determine the gradient?
  - Derive formula using multivariate calculus.
  - Ask a mathematician or a domain expert.
  - Do a literature search.

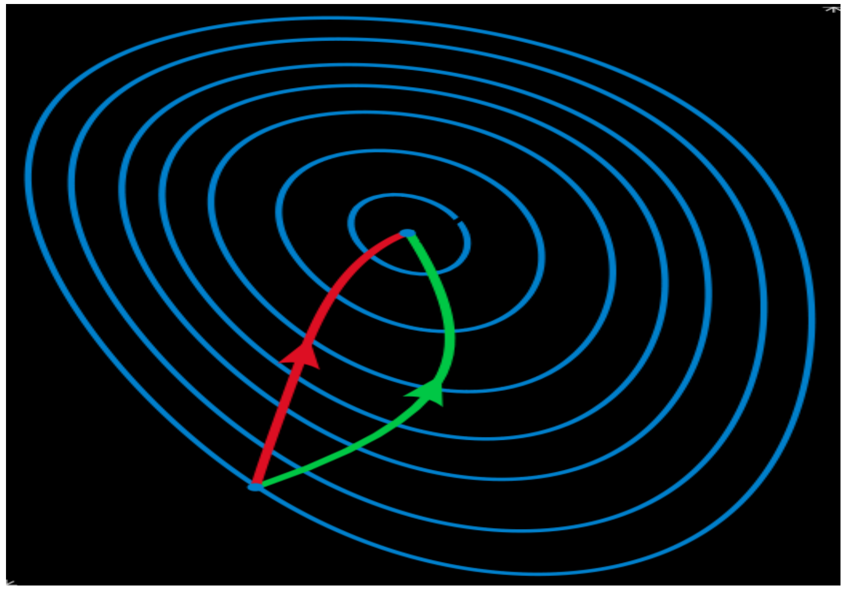
- Variations of gradient descent can improve performance for this or that special case.
  - See Numerical Recipes in C (and in other languages) by Press, Teukolsky, Vetterling, and Flannery.
  - Simulated Annealing, Linear Programming too

- Works well in smooth spaces; poorly in rough.

Gradient methods vs. Newton’s method

- Gradient descent algorithms find local minima by moving along the direction of steepest descent while Newton's method takes into account curvature information and thereby often improves convergence.
- A reminder of Newton’s method from Calculus:
  $x_{i+1} \leftarrow x_i - \eta f'(x_i) / f''(x_i)$
- Newton’s method uses 2nd order information (e.g., 2nd derivative) to take a faster route to a minimum
- Second-order info. is more expensive to compute
- Does not always converge; sometimes unstable
- If converges, usually very fast
- Works well for smooth, non-pathological functions, linearization accurate
- Works poorly for wiggly, ill-behaved functions

- See gradient descent



Contour lines of a function
Gradient descent (green)
Newton’s method (red)
Image from http://en.wikipedia.org/wiki/Newton's_method_in_optimization

Simulated annealing

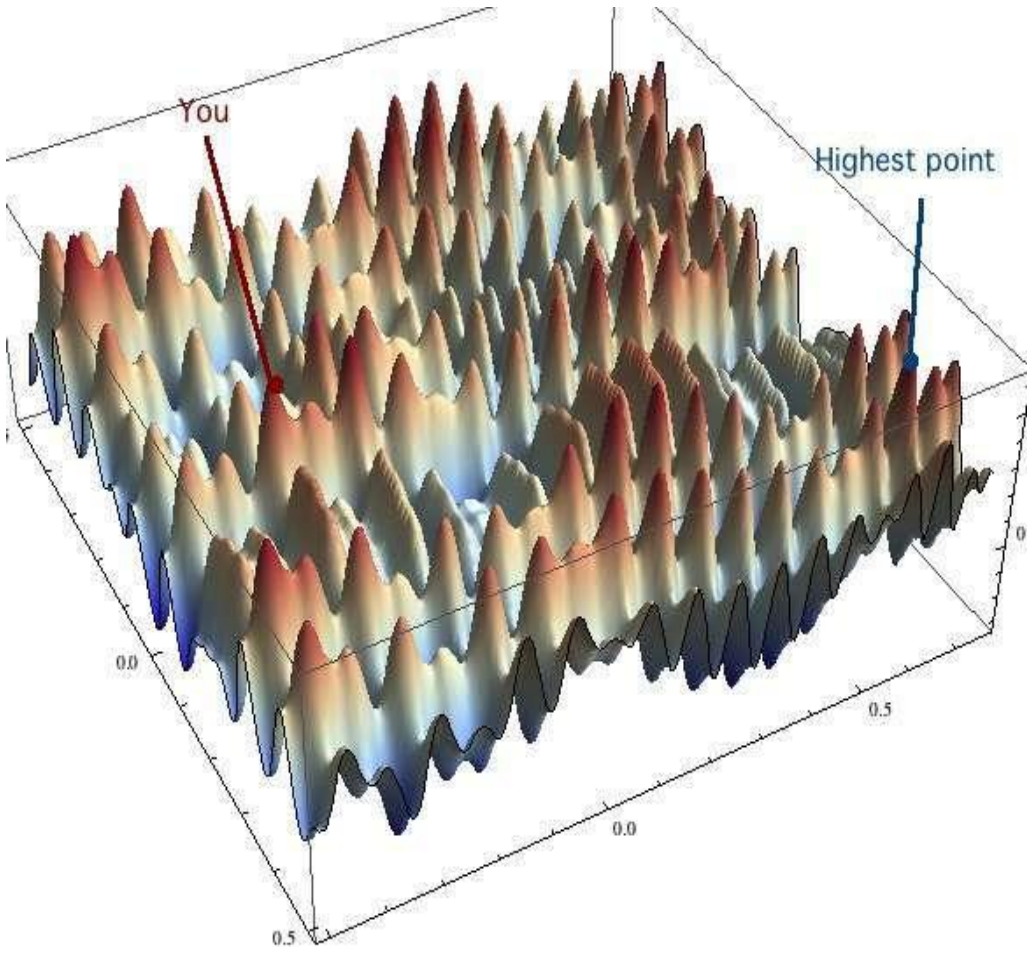
- A hill-climbing algorithm that never makes "downhill" moves toward states with lower value (or higher cost) is always vulnerable to getting stuck in a local maximum. In contrast, a purely random walk that moves to a successor state without concern for the value will eventually stumble upon the global maximum but will be extremely inefficient.
- Therefore, it seems reasonable to try to combine hill climbing with a random walk in a way that yields both efficiency and completeness. **Simulated annealing** is such an algorithm.
- In metallurgy, annealing is a technique involving heating & controlled cooling of a material to increase size of its crystals & reduce defects
- Heat causes atoms to become unstuck from initial positions (local minima of internal energy) and wander randomly through states of higher energy
- Slow cooling gives them more chances of finding configurations with lower internal energy than initial one

Real annealing: Sword

- The worker heats the metal, then slowly cools it as he hammers the blade into shape.
  - If he cools the blade too quickly the metal will form patches of different composition;
  - If the metal is cooled slowly while it is shaped, the constituent metals will form a uniform alloy.

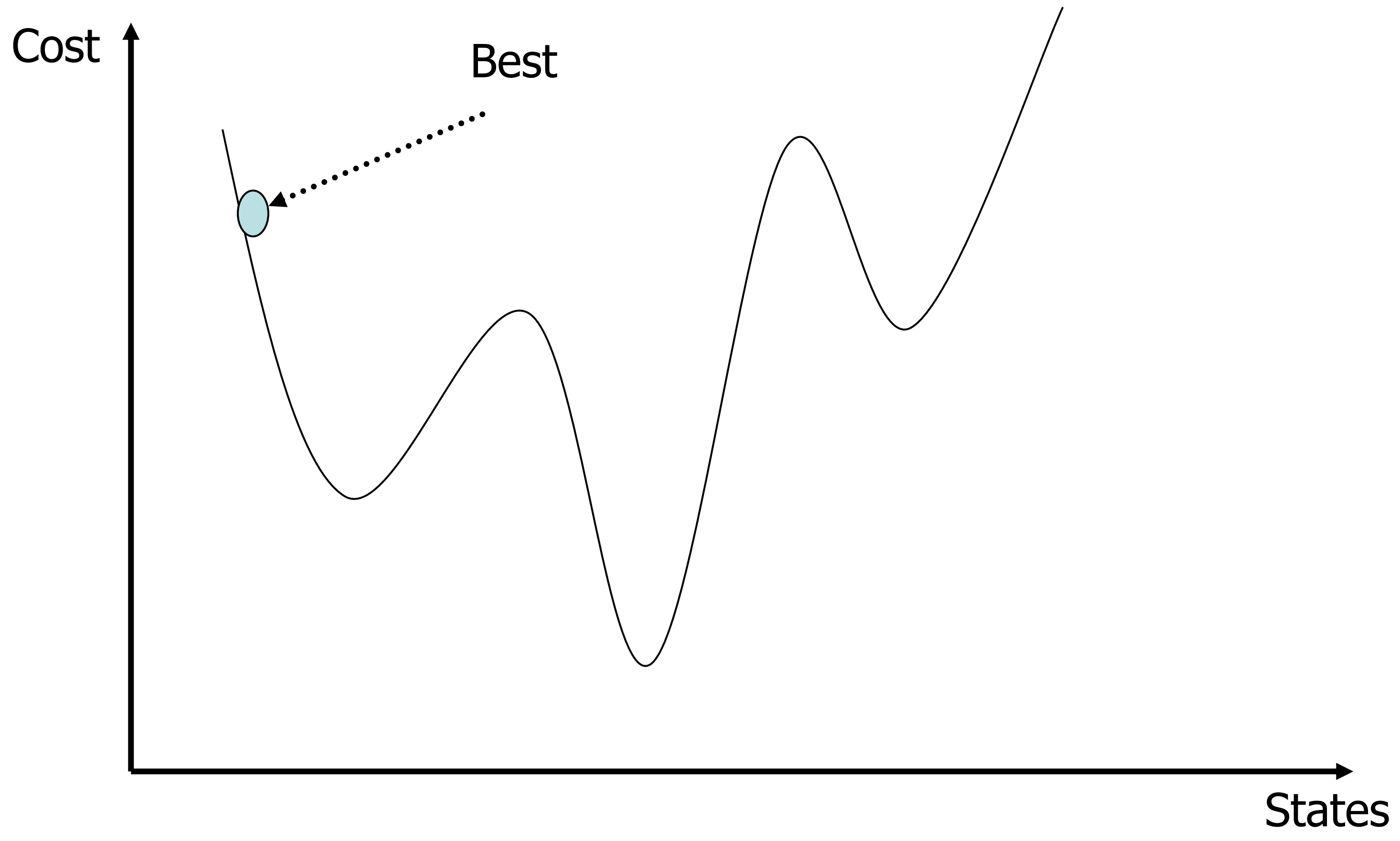
SA intuitions

- Combines **hill climbing** (for efficiency) with **random walk** (for completeness)
- Analogy: getting a ping-pong ball into the deepest depression in a bumpy surface
  - Shake the surface to get the ball out of local minima
  - Don't shake too hard to dislodge it from global minimum
- Simulated annealing:
  - Start shaking hard (high temperature) and gradually reduce shaking intensity (lower temperature)
  - Escape local minima by allowing some "bad" moves
  - But gradually reduce their size and frequency
- The simulated-annealing solution is to start by shaking hard (i.e., at a high temperature) and then gradually reduce the intensity of the shaking (i.e., lower the temperature).

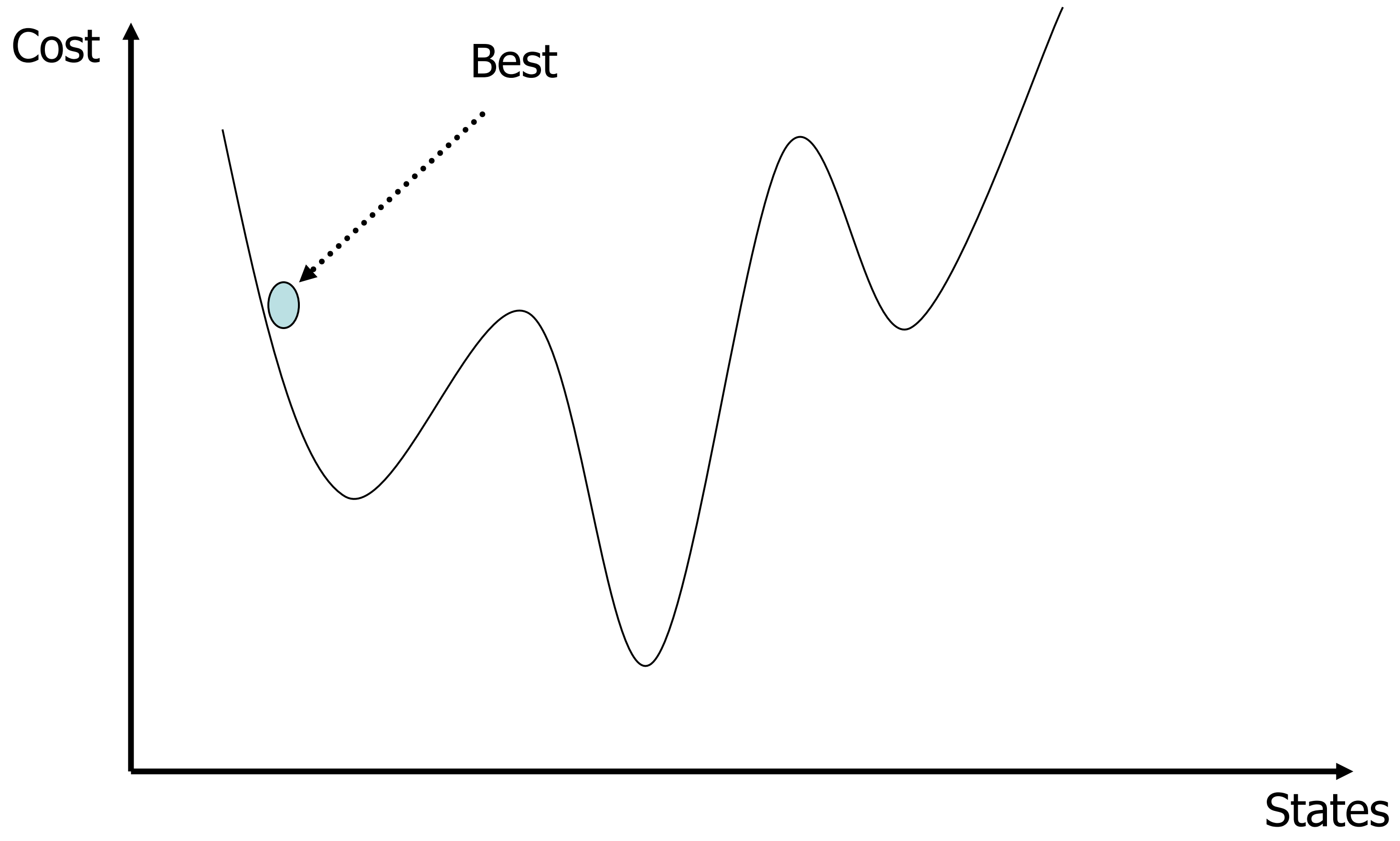


Simulated Annealing

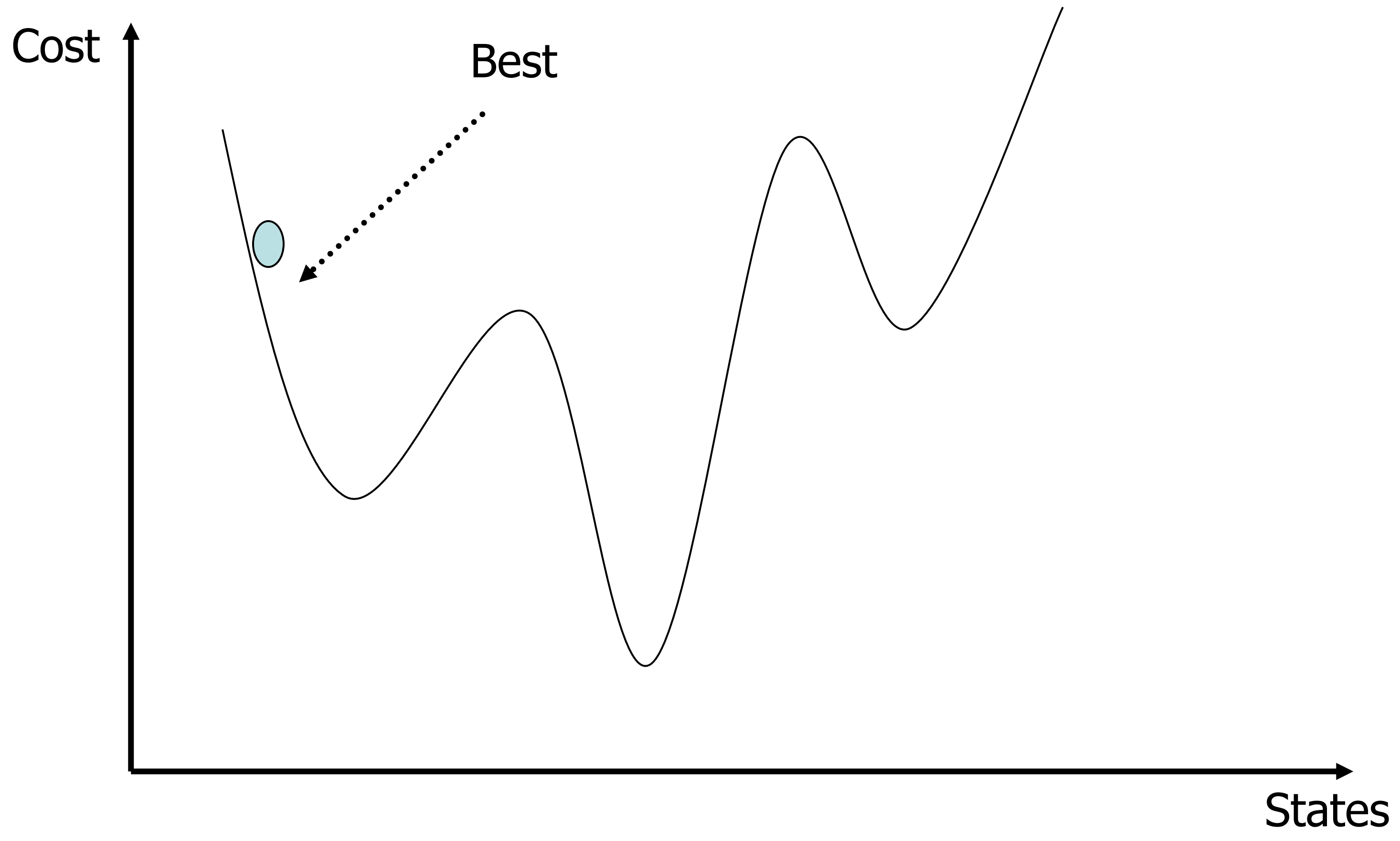
- SA can avoid becoming trapped at local minima
- SA is a stochastic algorithm involving asymptotic convergence and allowing random movements in the searched neighborhood in order to escape local minima



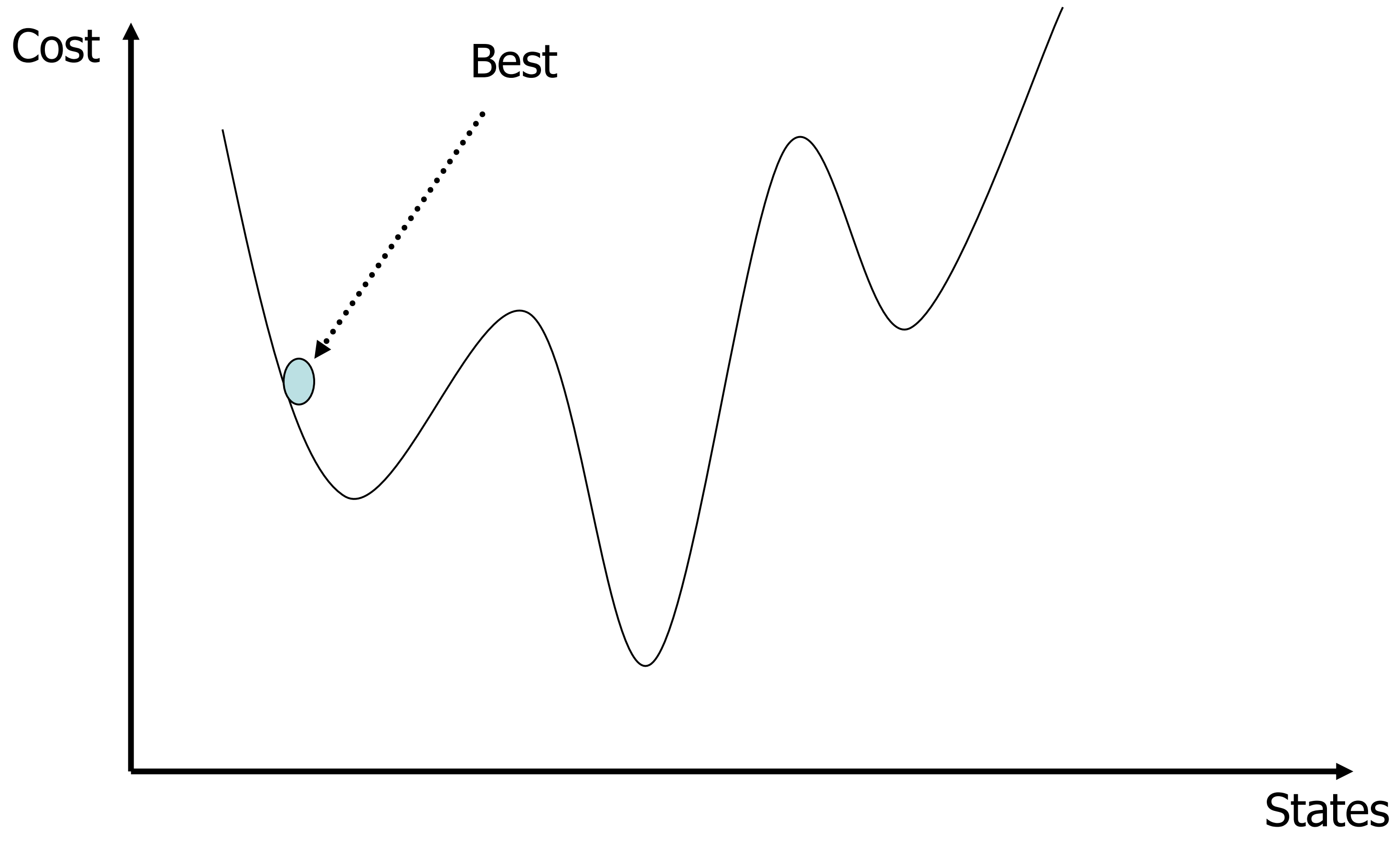
Simulated Annealing



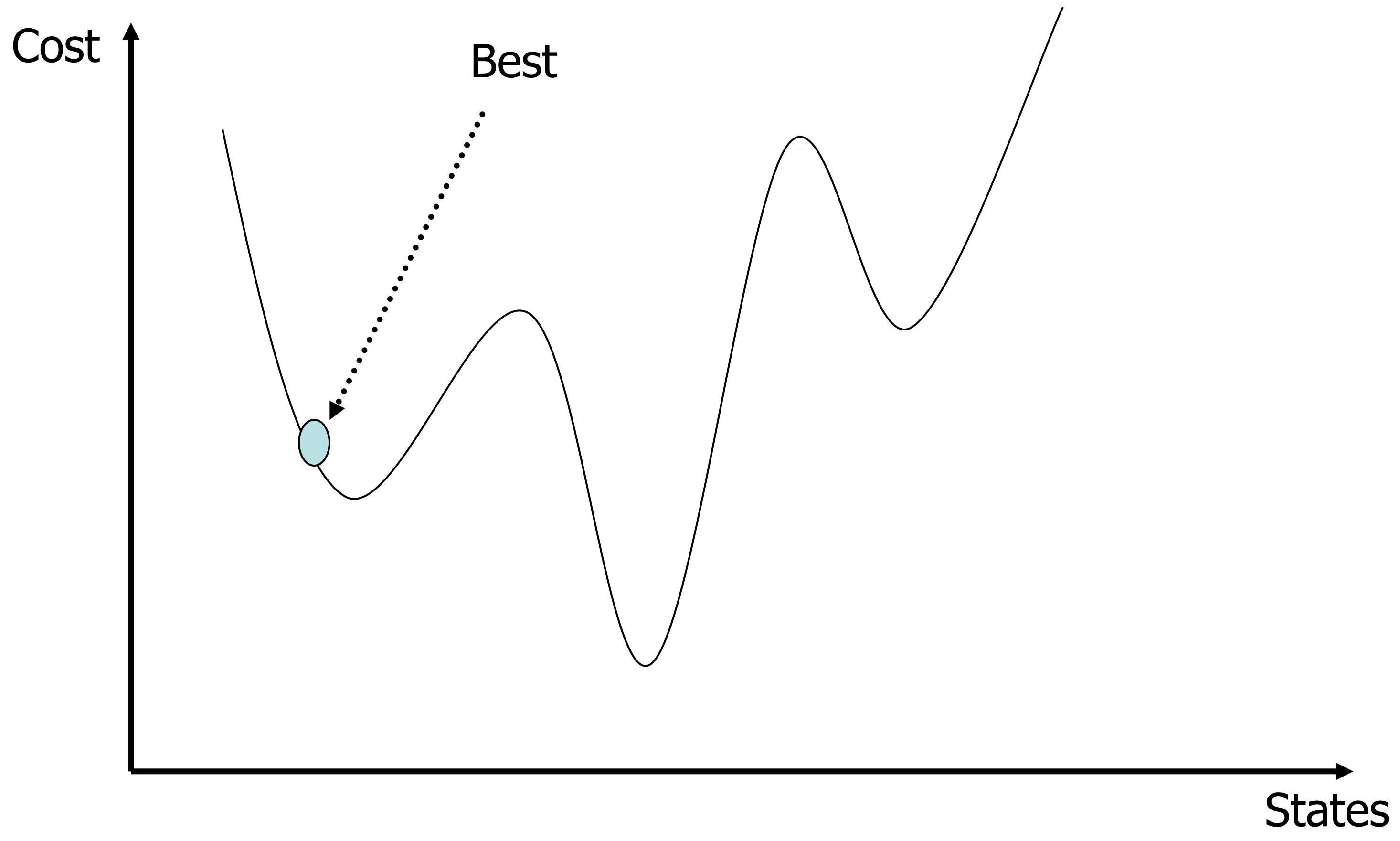
Simulated Annealing



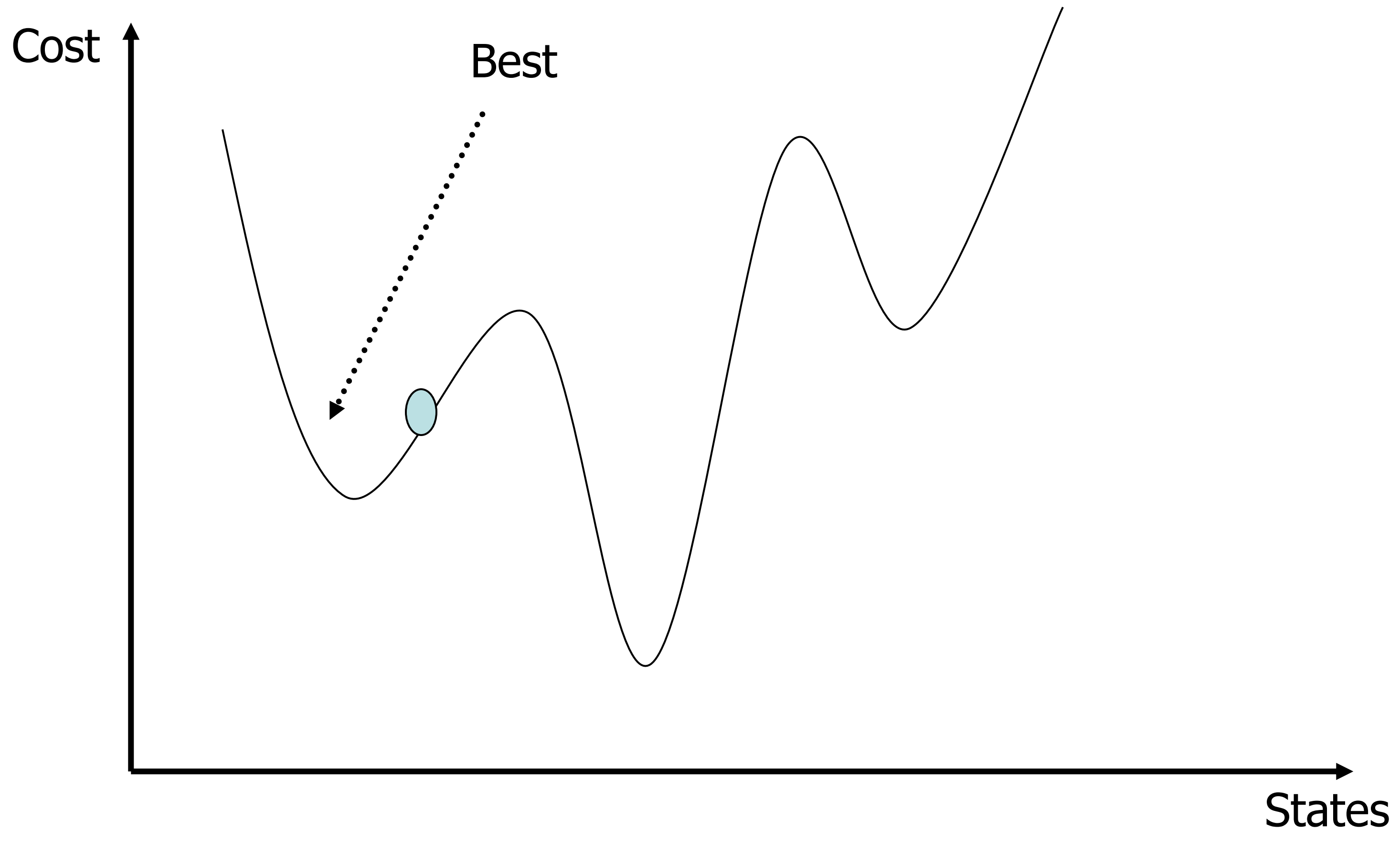
Simulated Annealing



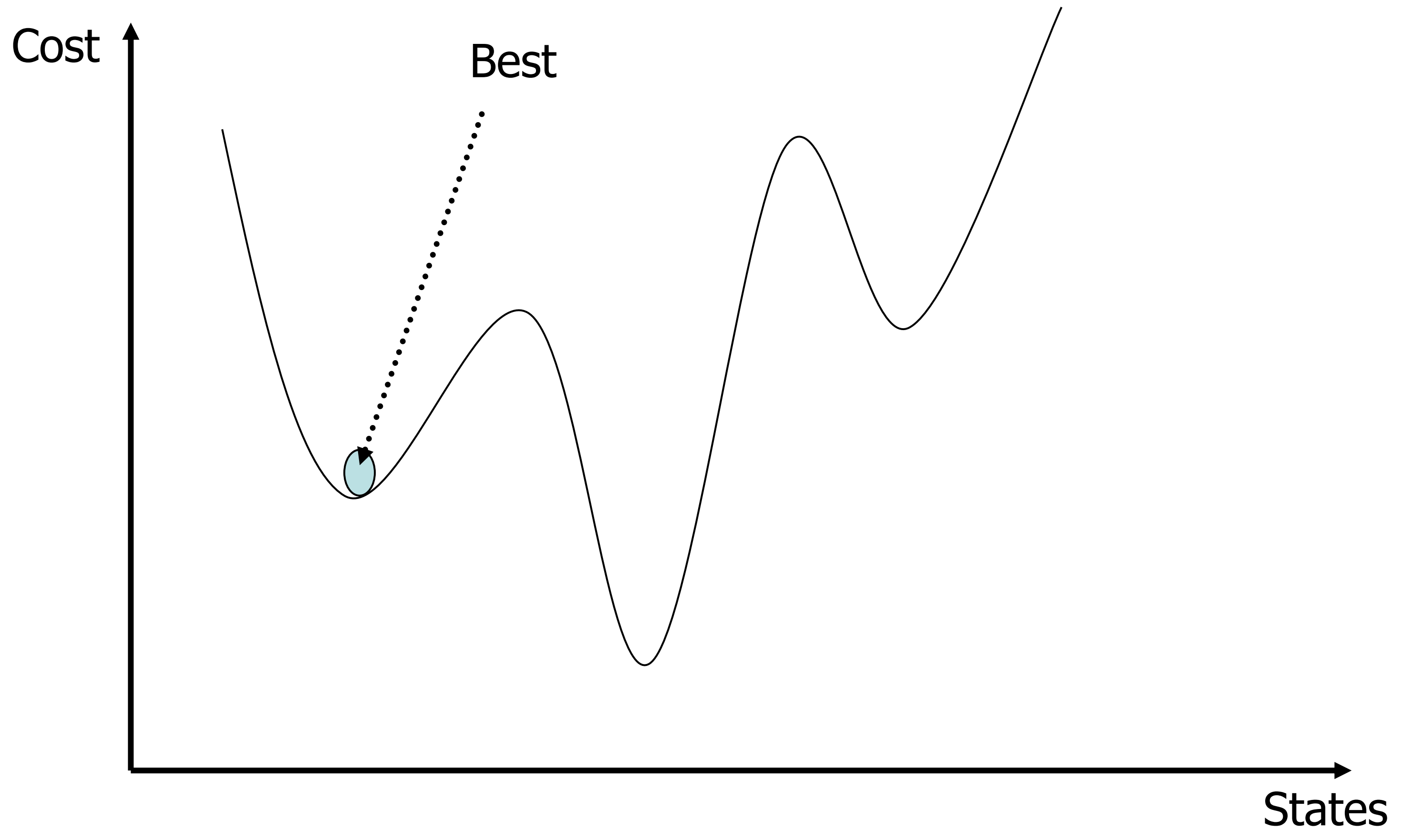
Simulated Annealing



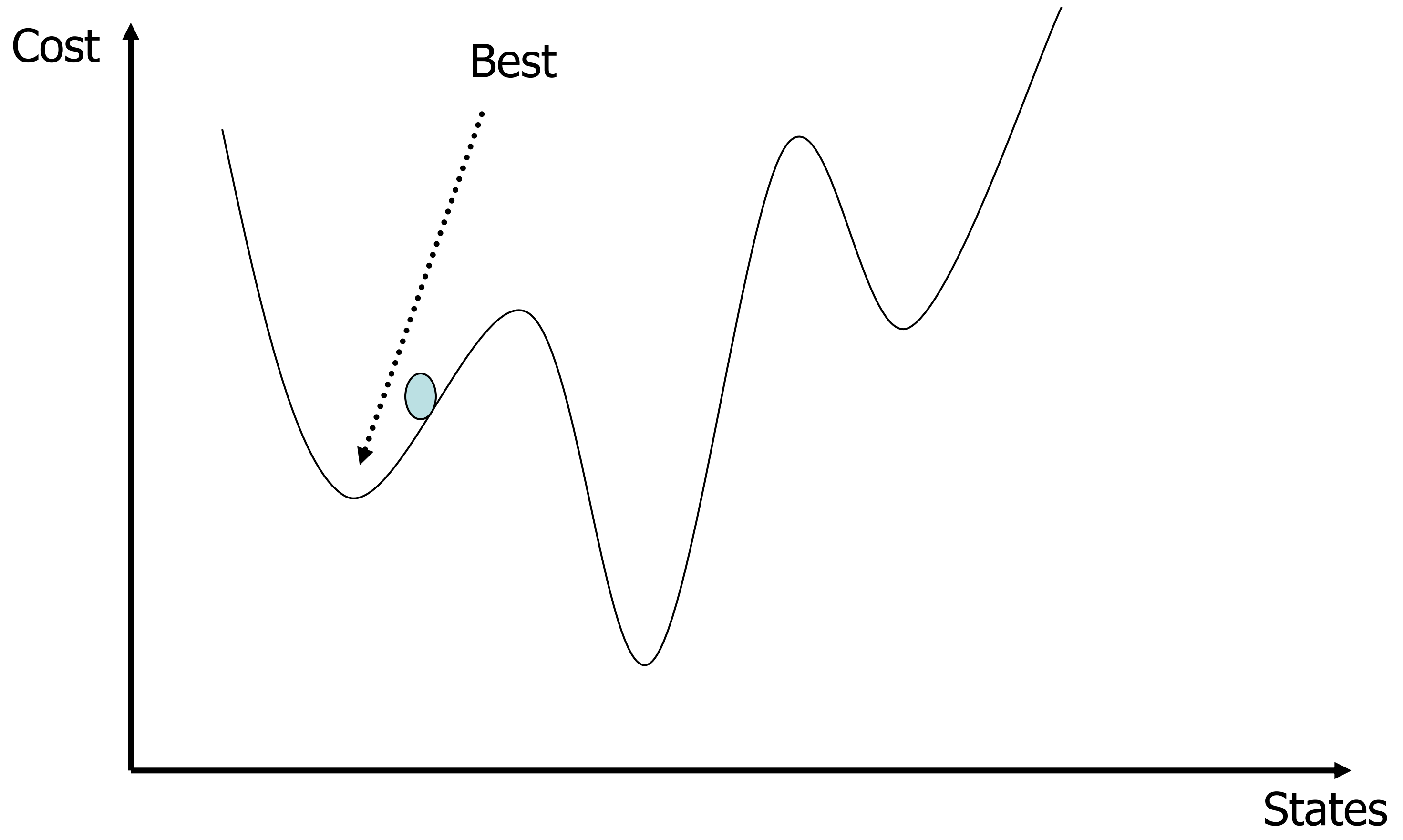
Simulated Annealing



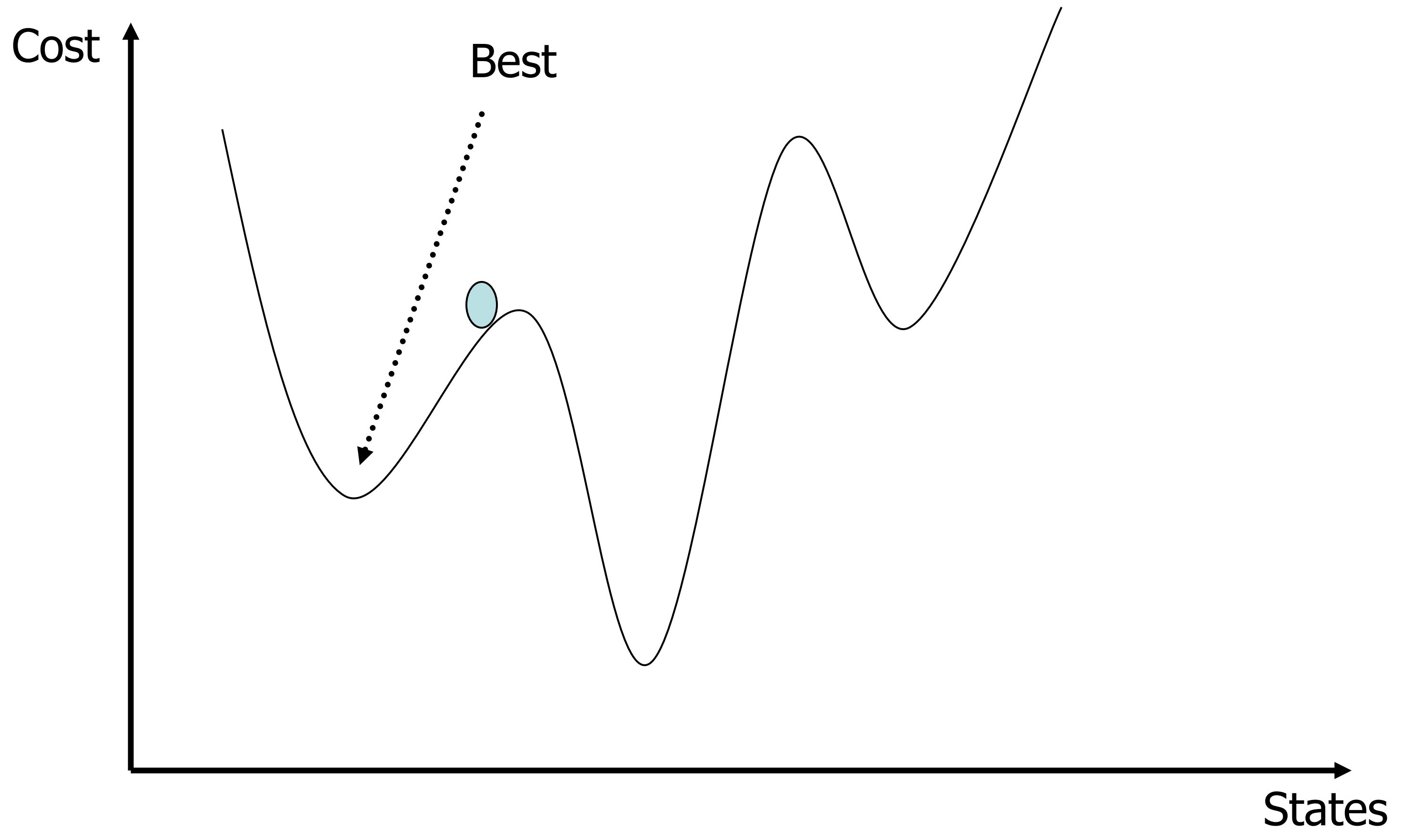
Simulated Annealing



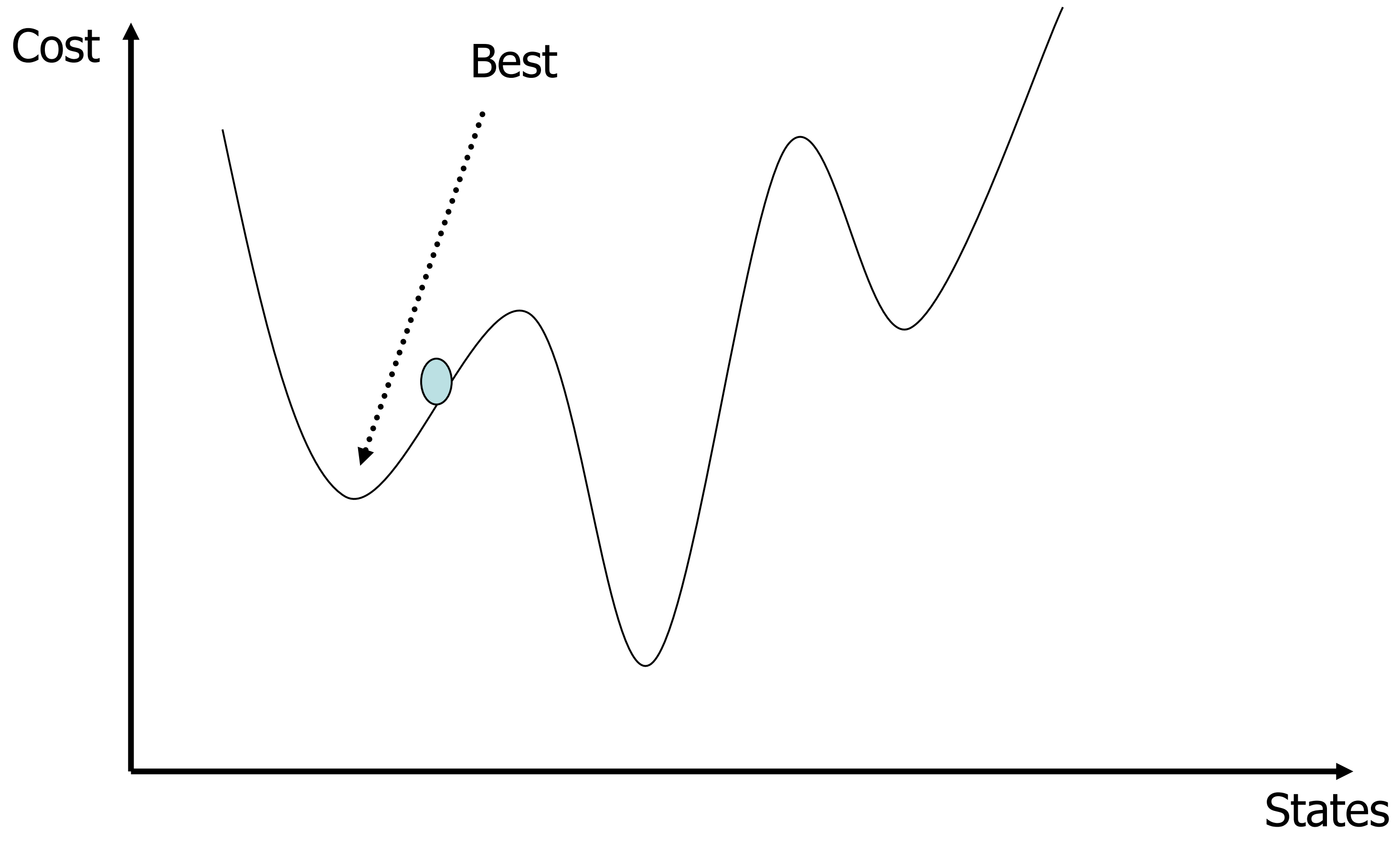
Simulated Annealing



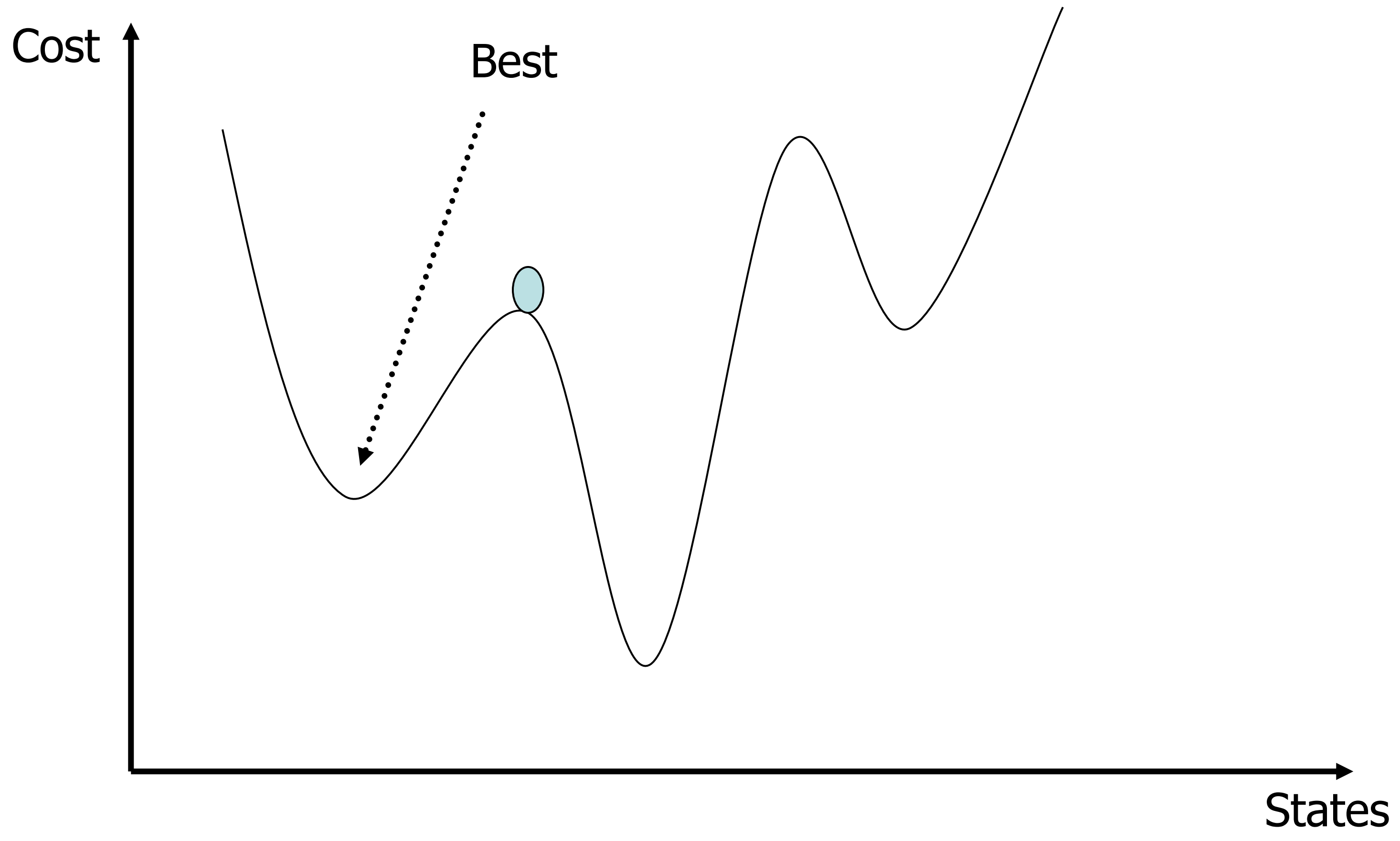
Simulated Annealing



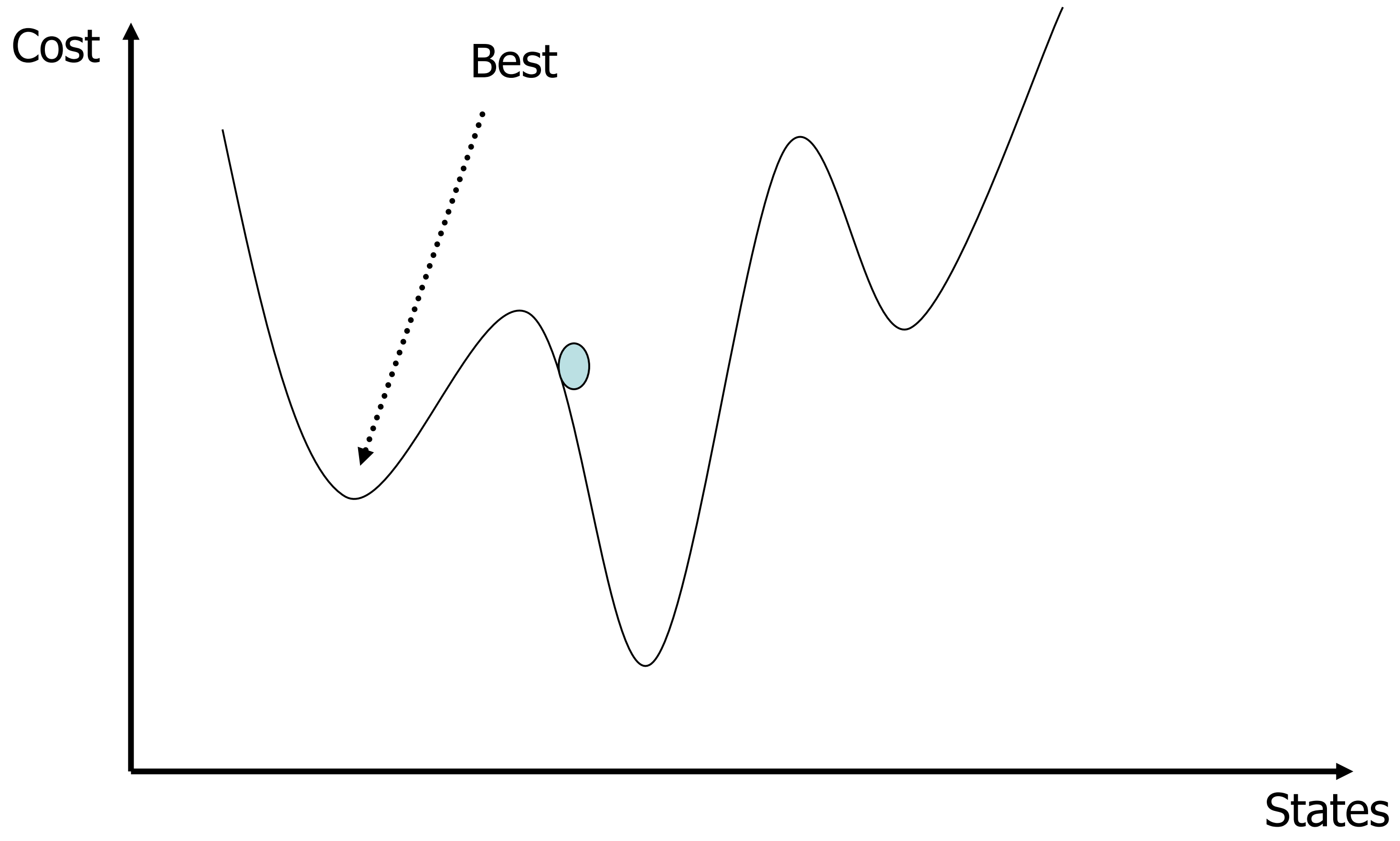
Simulated Annealing



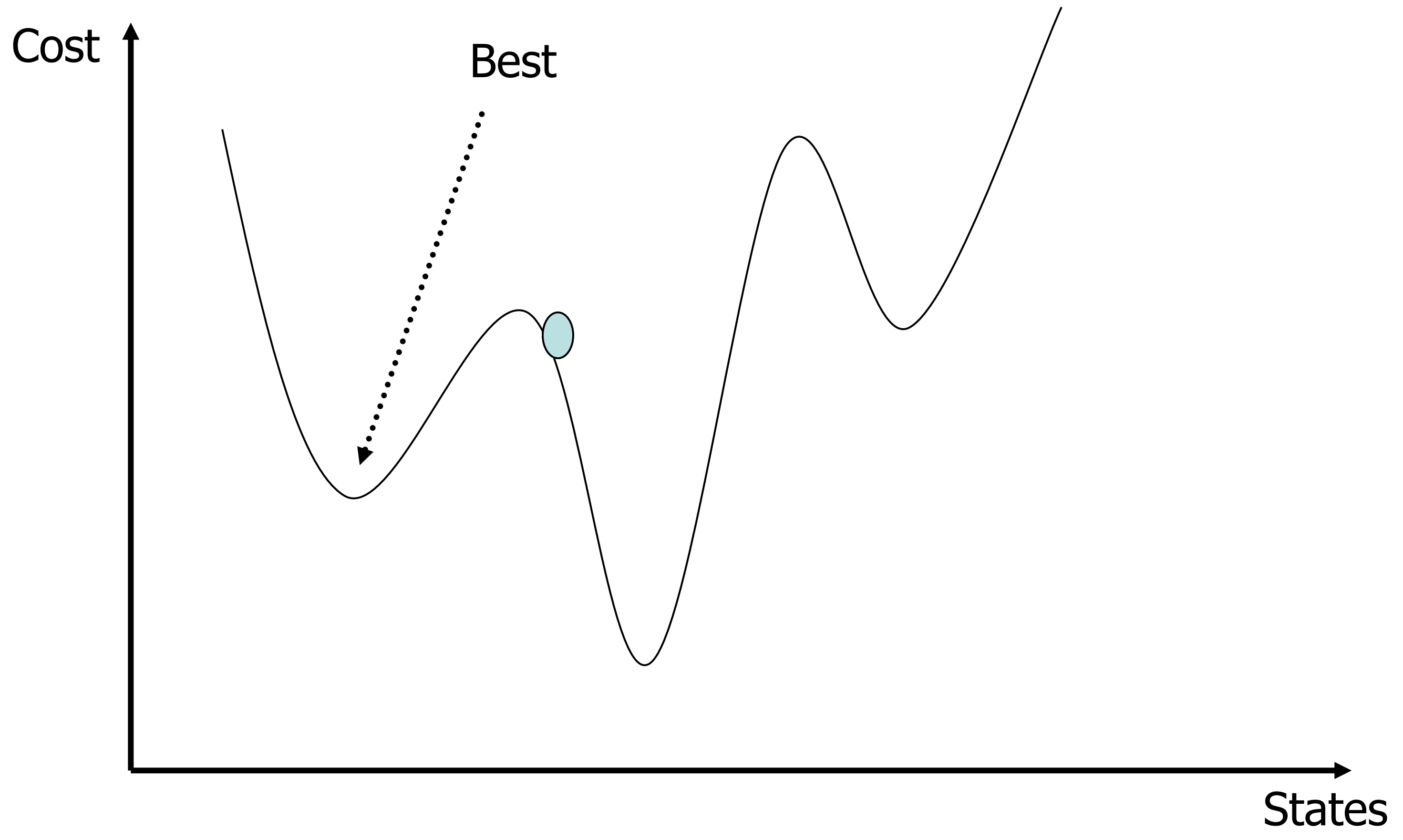
Simulated Annealing



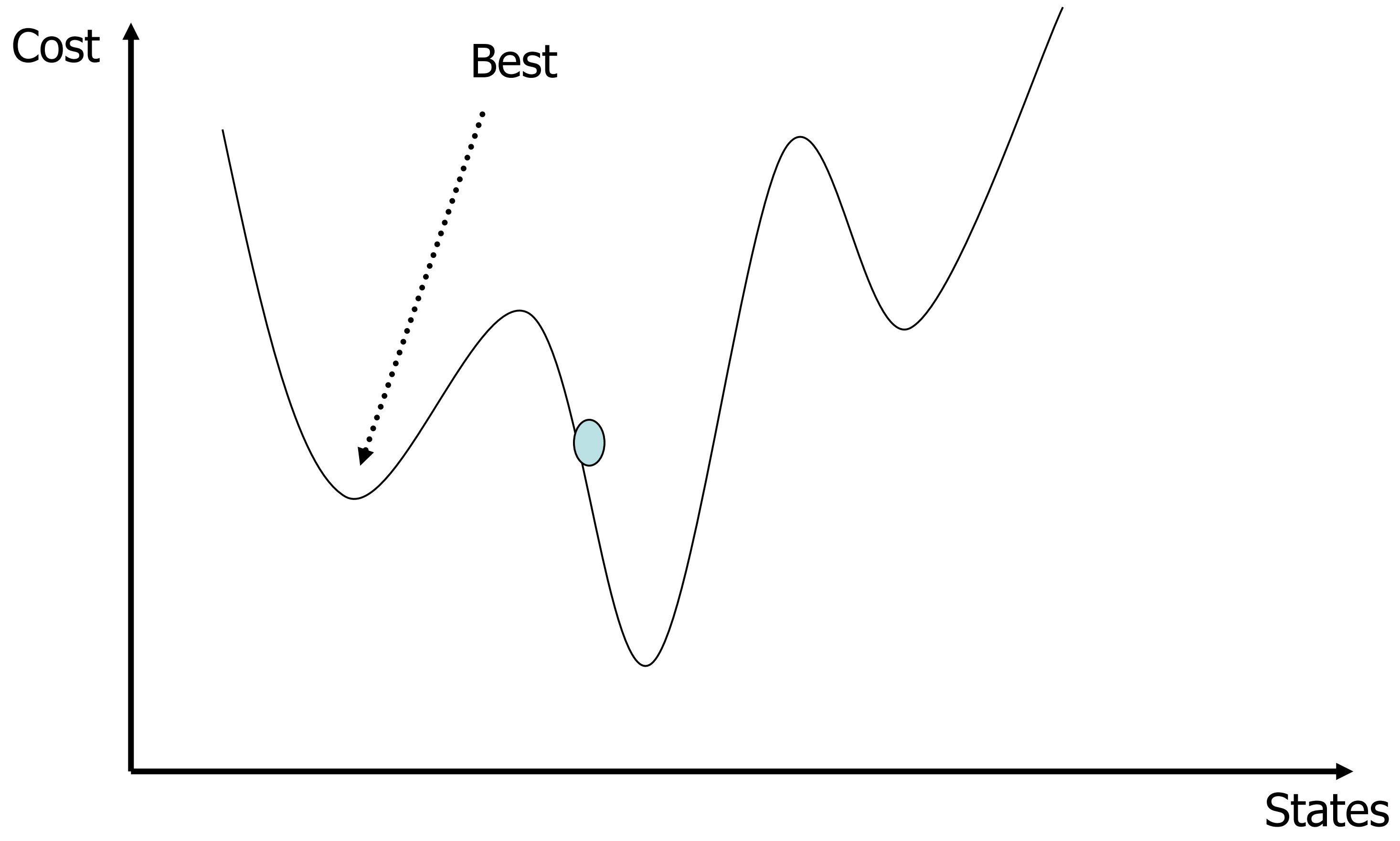
Simulated Annealing



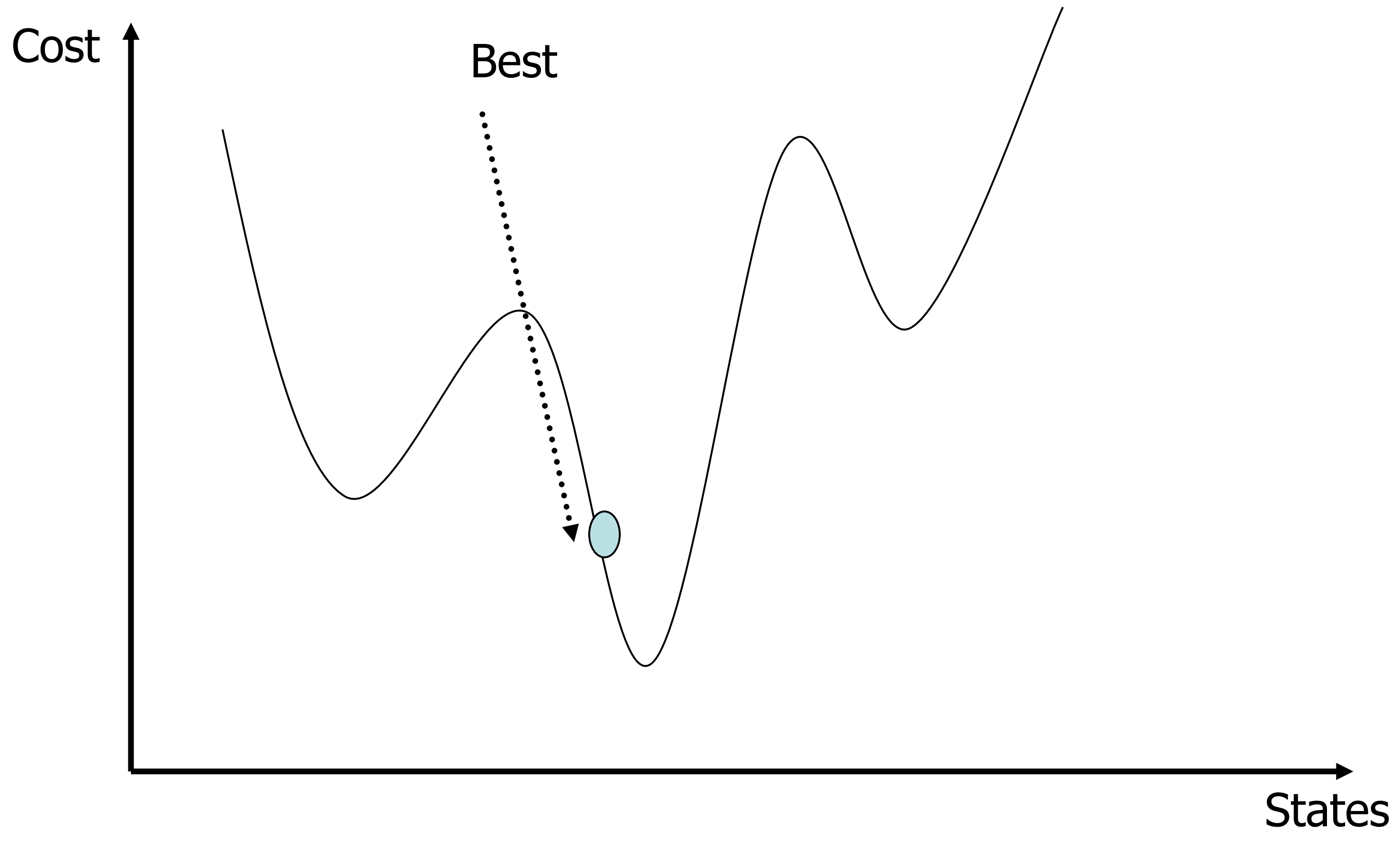
Simulated Annealing



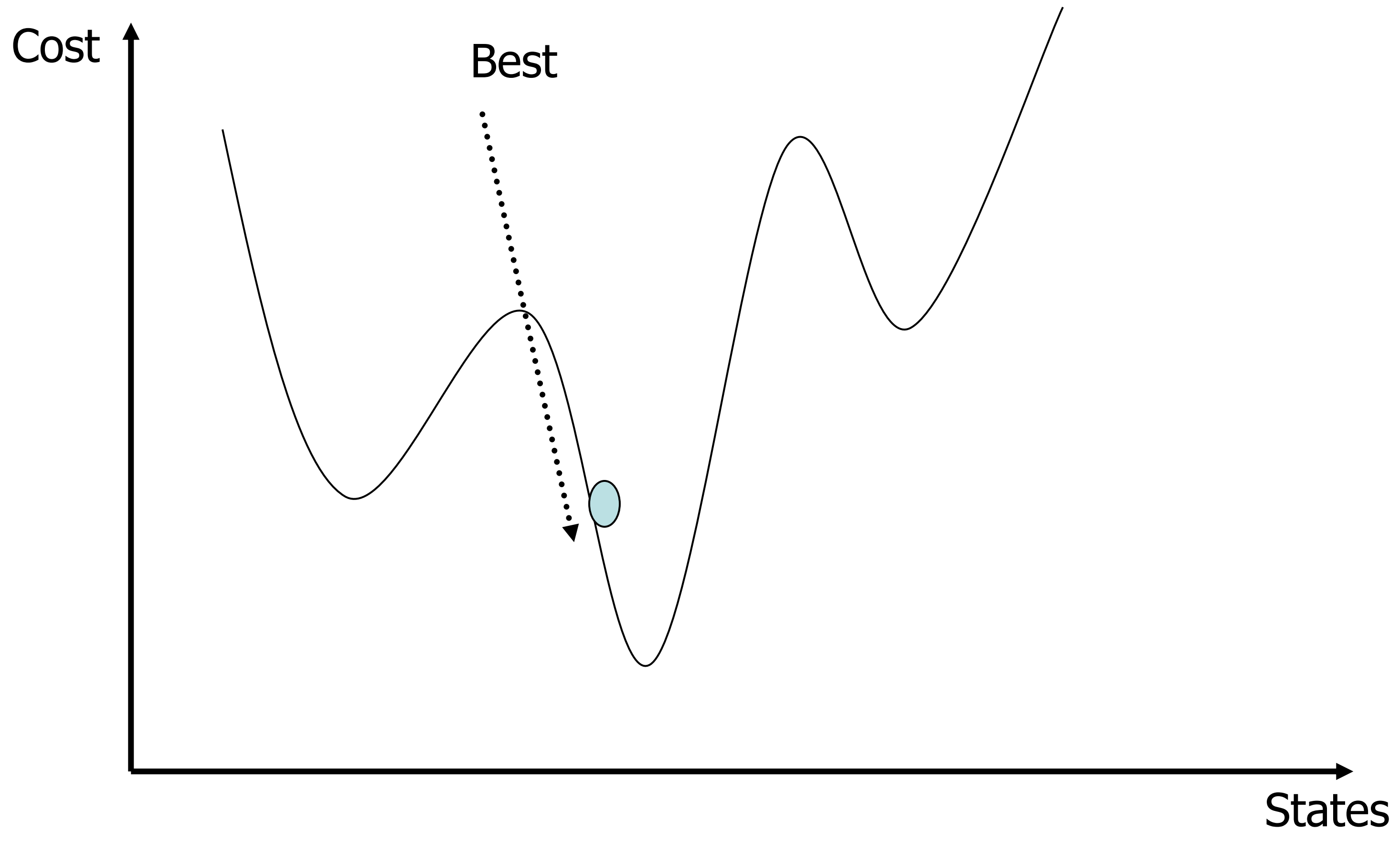
Simulated Annealing



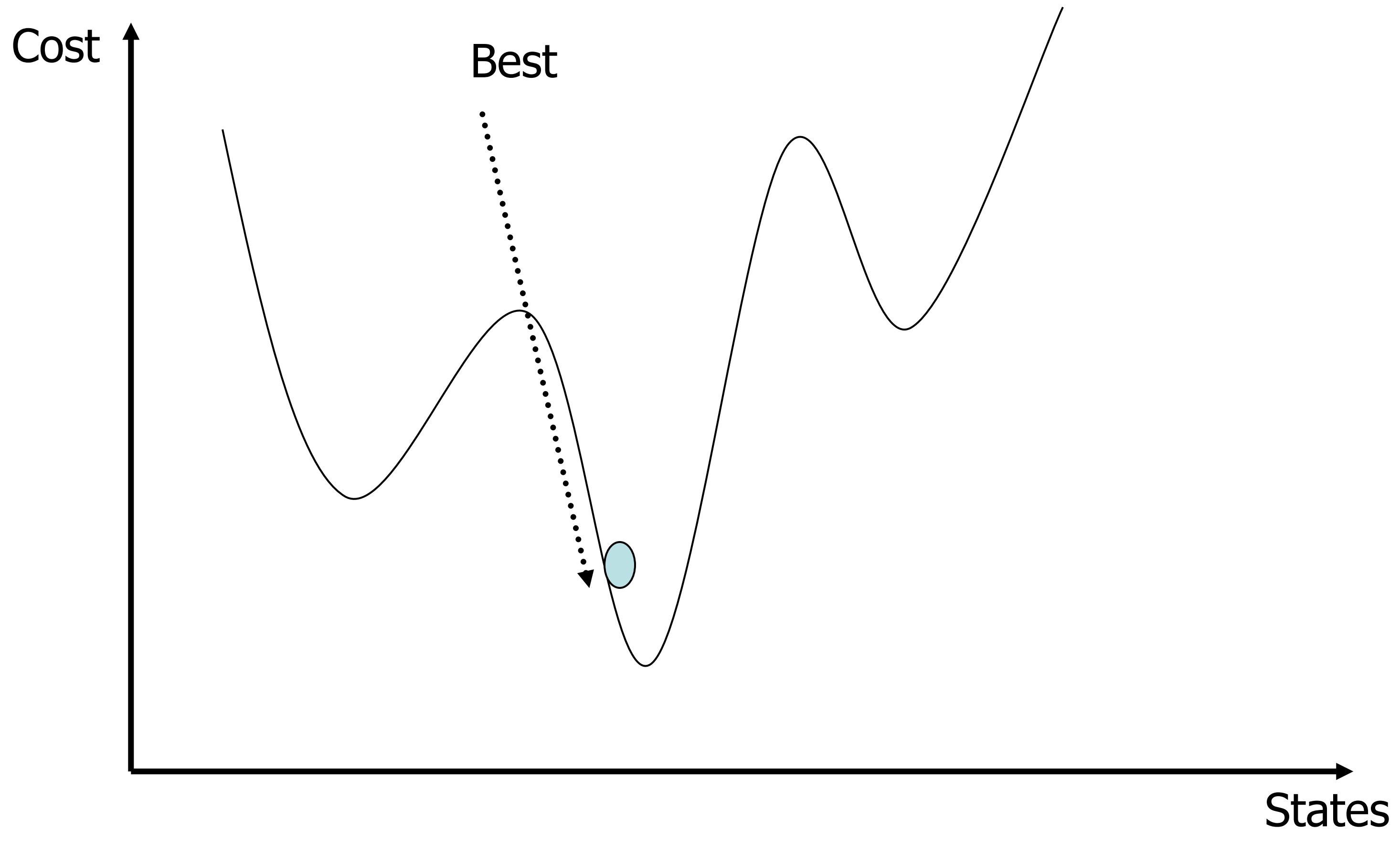
Simulated Annealing



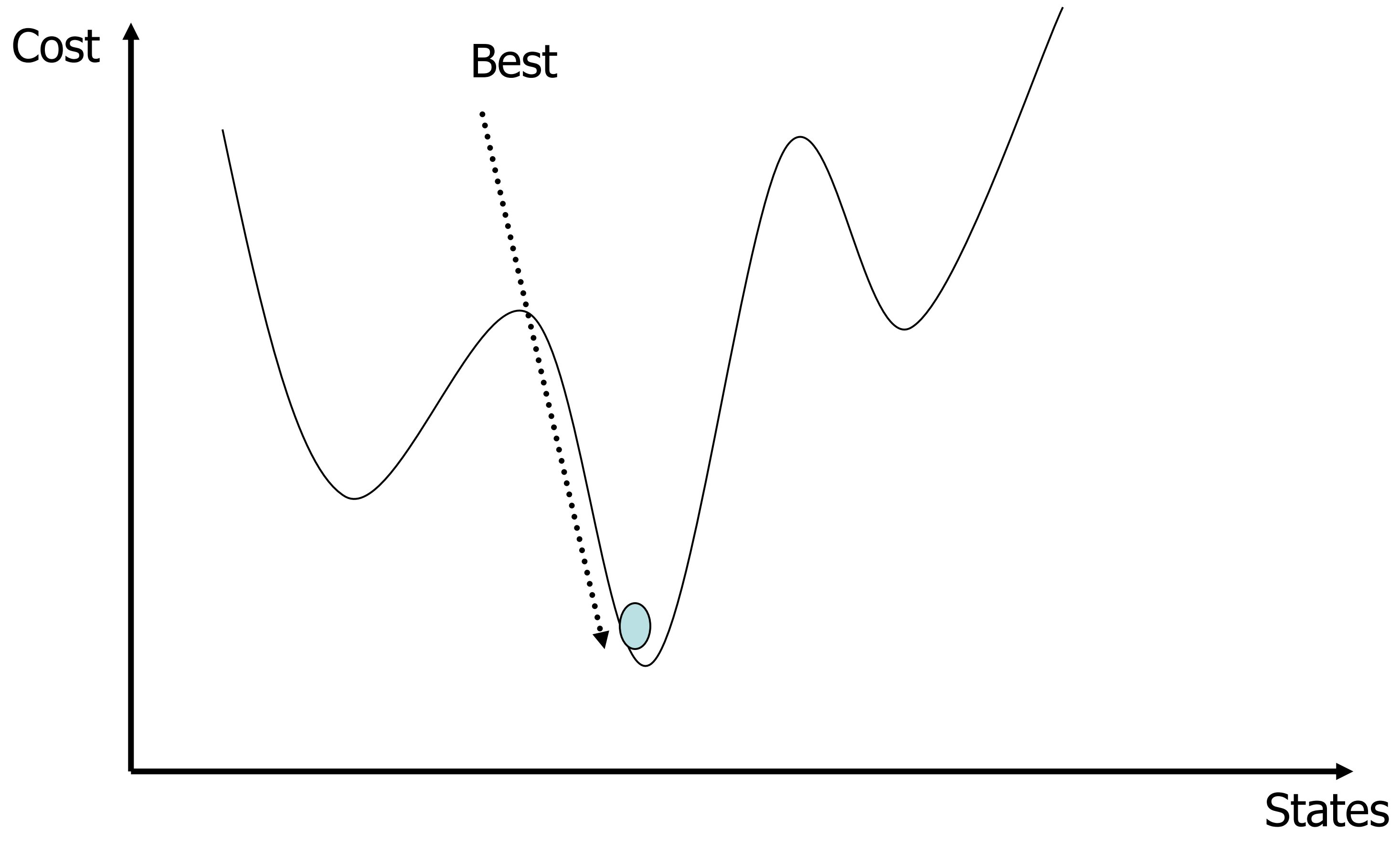
Simulated Annealing



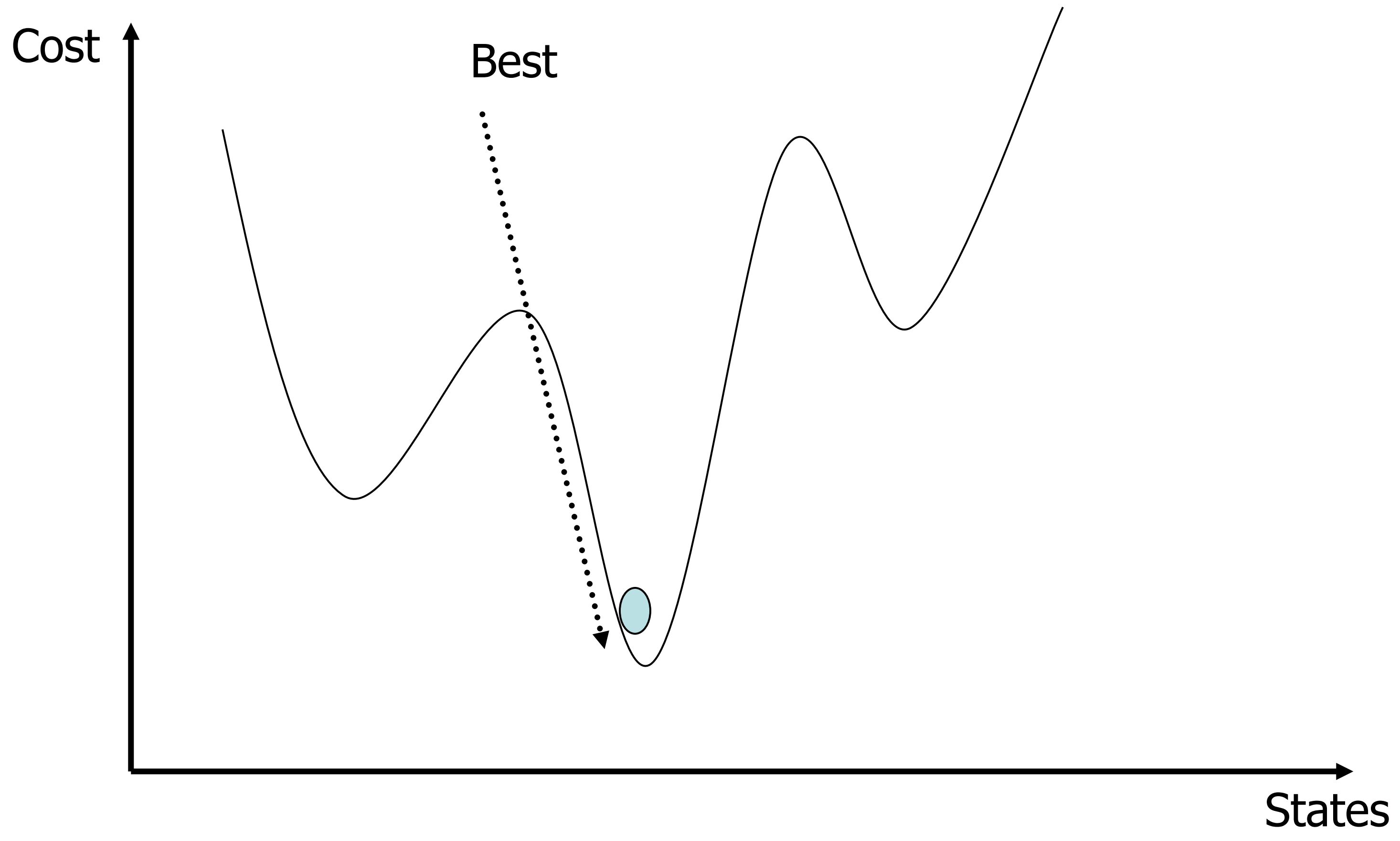
Simulated Annealing



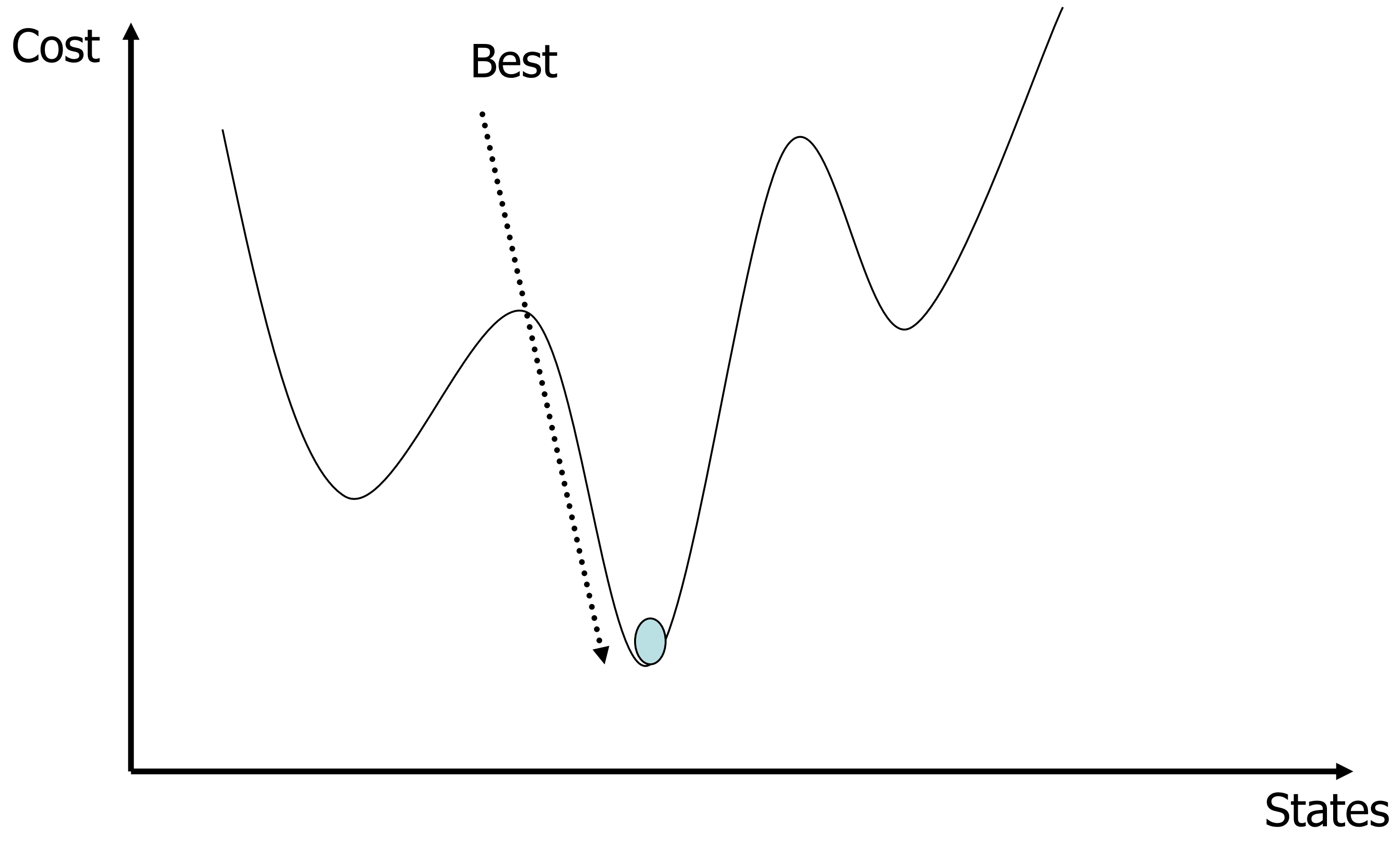
Simulated Annealing



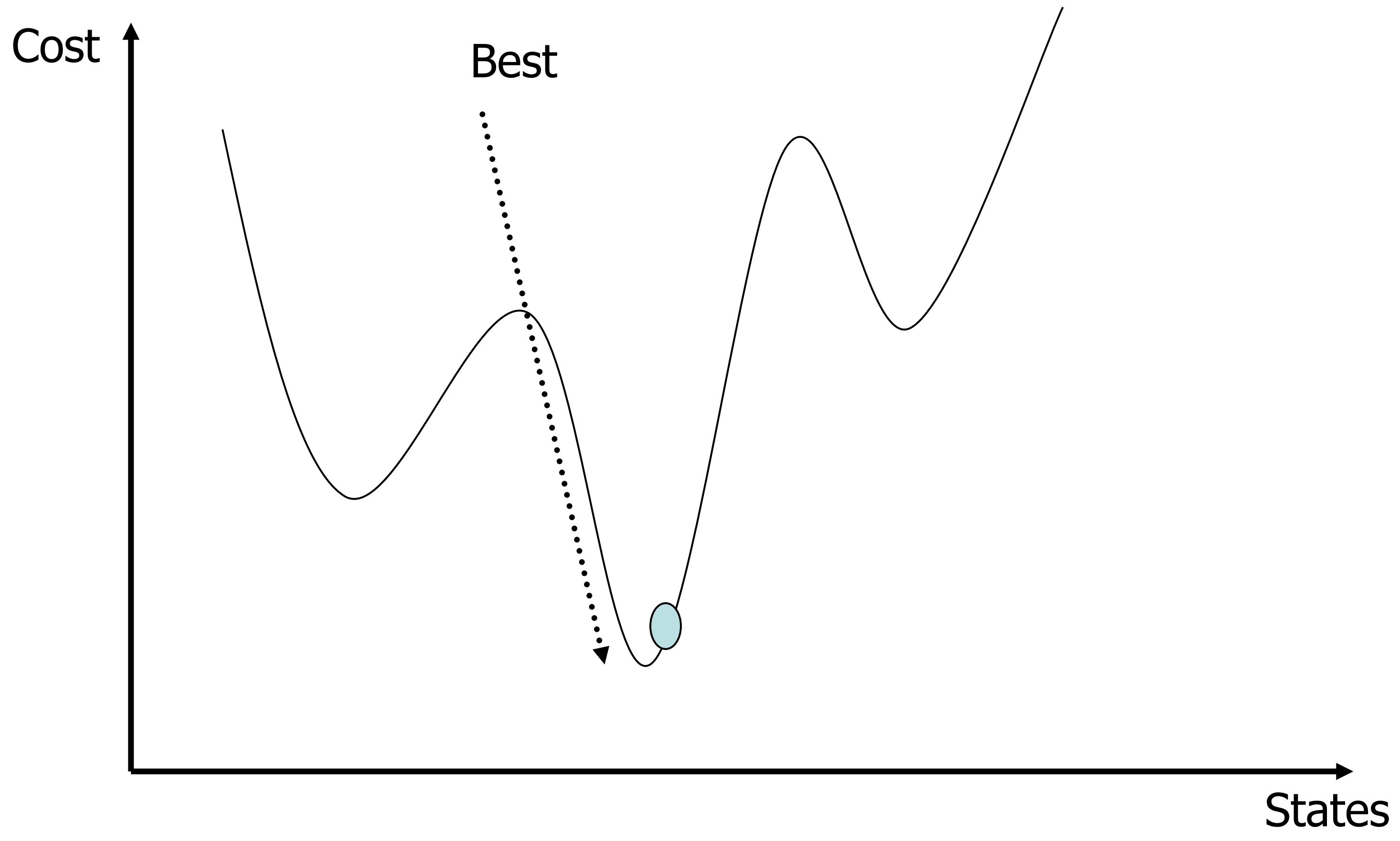
Simulated Annealing



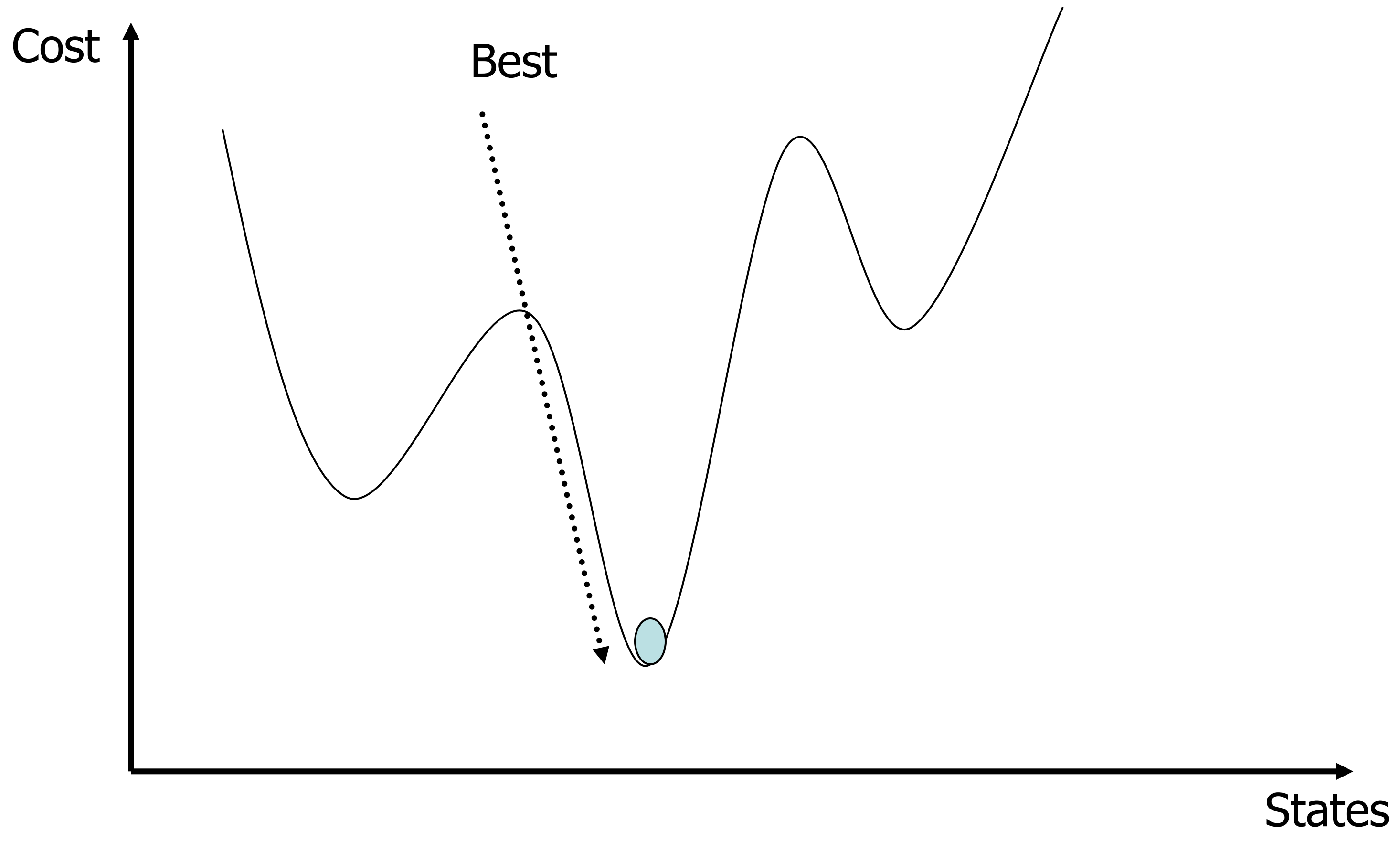
Simulated Annealing



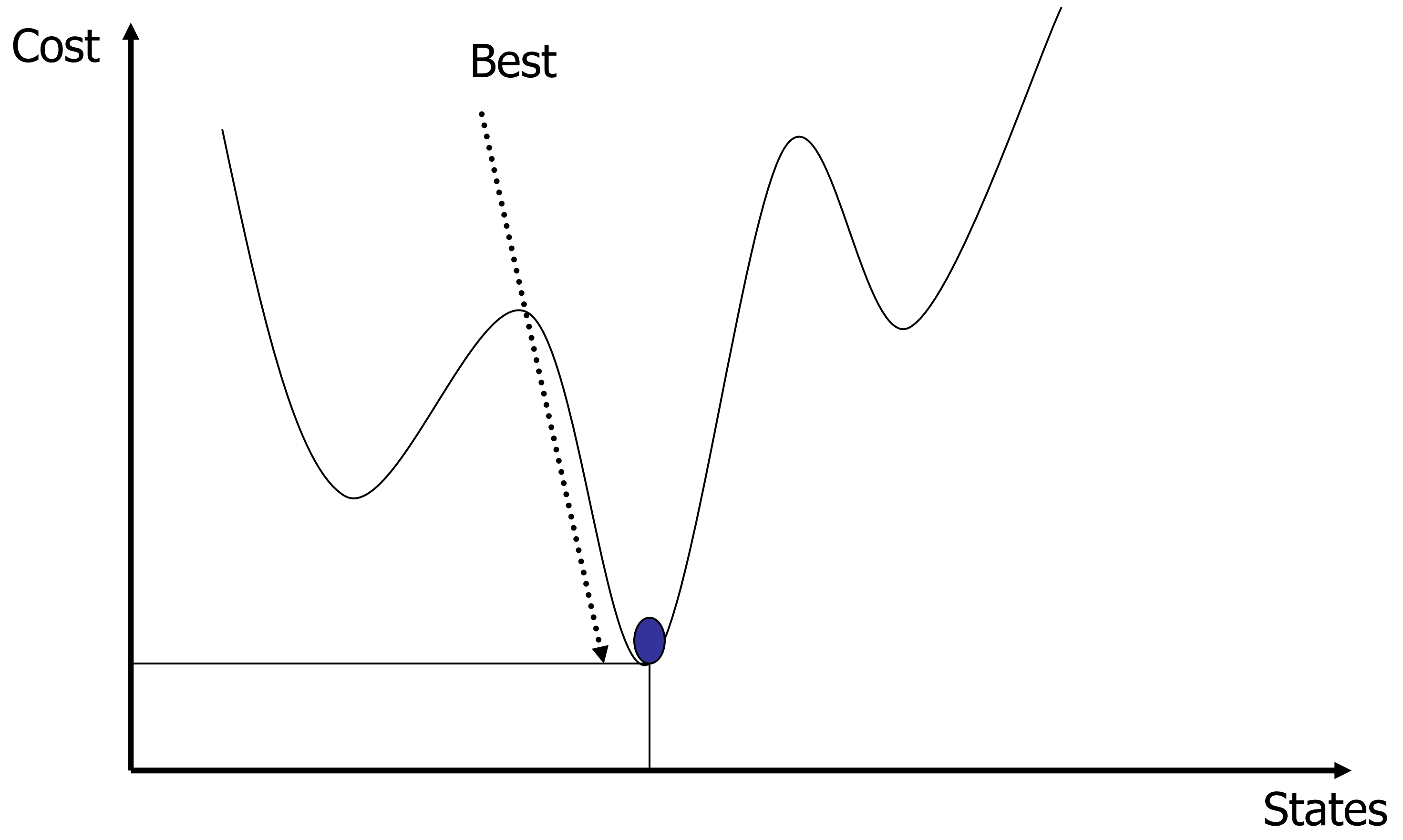
Simulated Annealing



Simulated Annealing

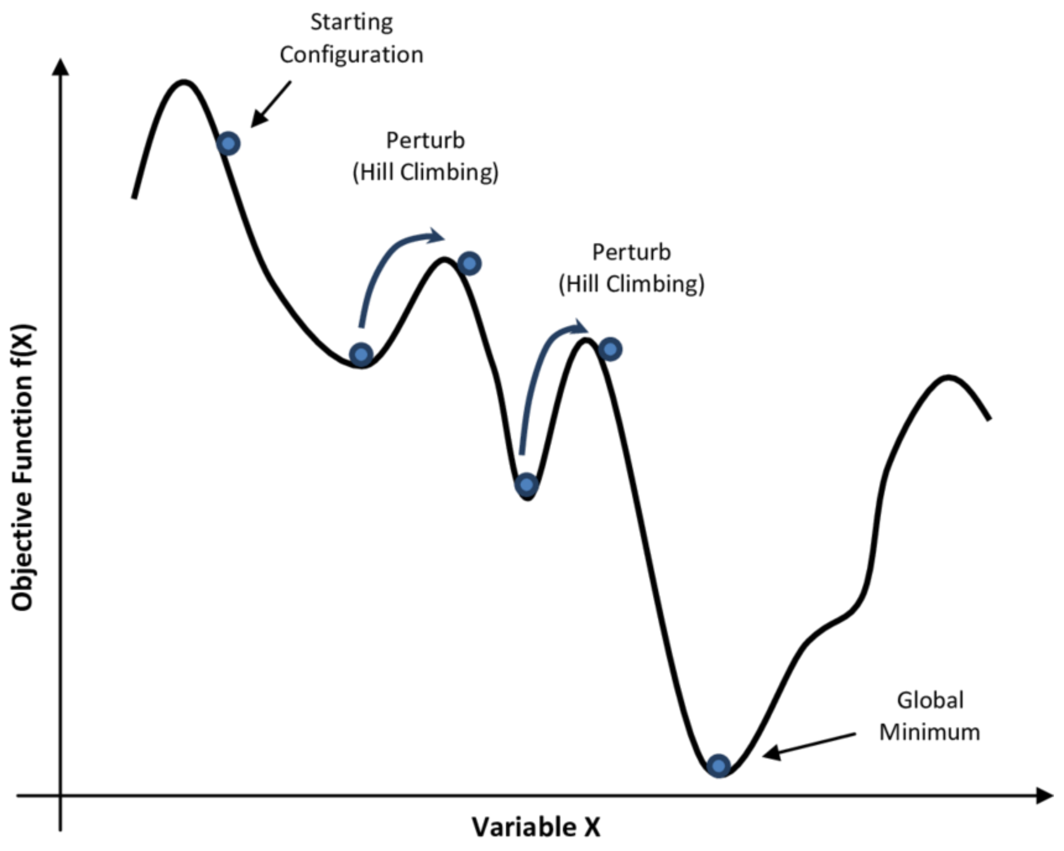


Simulated Annealing



Simulated annealing (SA)

- SA can avoid becoming trapped at local minima
- SA is a stochastic algorithm involving asymptotic convergence and allowing random movements in the searched neighborhood in order to escape local minima
- SA uses a control parameter T, which by analogy with the original application, is known as the system ***temperature***
- The higher the temperature, the higher probability of accepting a worse solution.
- Typically, T decreases as the algorithm runs longer. Theoretically this algorithm always finds the global optimum but it can run very slow for some problems and in practice it would be a problem as to how to decide the rate at which to decrease T.
- T starts out high and gradually decreases toward 0



Simulated annealing

- The overall structure of the simulated-annealing algorithm (Figure 4.5) is similar to hill climbing. Instead of picking the best move, however, it picks a random move.
- If the move improves the situation, it is always accepted. Otherwise, the algorithm accepts the move with some probability less than 1.
- The probability decreases exponentially with the "badness" of the move—the amount ΔE by which the evaluation is worsened.
- A "bad" move from A to B is accepted with a probability

$$e^{-(f(B)-f(A)/T)}$$

- The higher the temperature, the more likely it is that a bad move can be made
- As T tends to zero, probability tends to zero, and SA becomes more like hill climbing
- If T lowered slowly enough, SA is complete and admissible

Simulated annealing algorithm

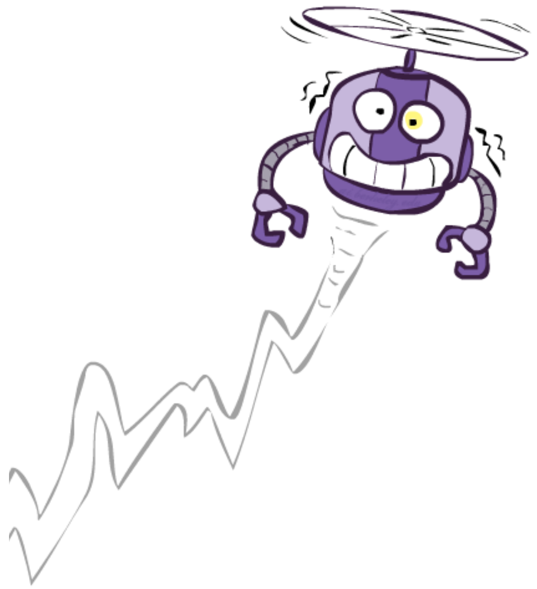
```
**function** SIMULATED-ANNEALING($problem, schedule$) **returns** a solution state
    **inputs**: $problem$, a problem
            $schedule$, a mapping from time to “temperature”
    **local variables**: $T$, a “temperature” controlling the probability of downward steps

    $current \leftarrow$ MAKE-NODE($problem$.INITIAL-STATE)
    **for** $t = 1$ **to** $\infty$ **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 $e^{\Delta E/T}$
```

Figure 4.5 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 T as a function of time.

Simulated Annealing

- Widely used in VLSI layout, airline scheduling, etc.
- Often works very well in practice
  - But usually VERY VERY slow
- One can prove:
  - If T decreases slowly enough, then simulated annealing search will find a global optimum with probability approaching 1
  - Unfortunately this can take a VERY VERY long time
  - Note: in any finite search space, random guessing also will find a global optimum with probability approaching 1
  - So, ultimately this is a very weak claim
- Theoretical guarantee (proof):
  - Stationary distribution (Boltzmann): $P(x) \propto e^{E(x)/T}$
  - If T decreased slowly enough, will converge to optimal state!
- Proof sketch
  - Consider two adjacent states x, y with $E(y) > E(x)$ [high is good]
  - Assume $x \longrightarrow y$ and $y \rightarrow x$ and outdegrees $D(x) = D(y) = D$
  - Let $P(x)$, $P(y)$ be the equilibrium occupancy probabilities at T
  - Let $P(x \rightarrow y)$ be the probability that state x transitions to state y

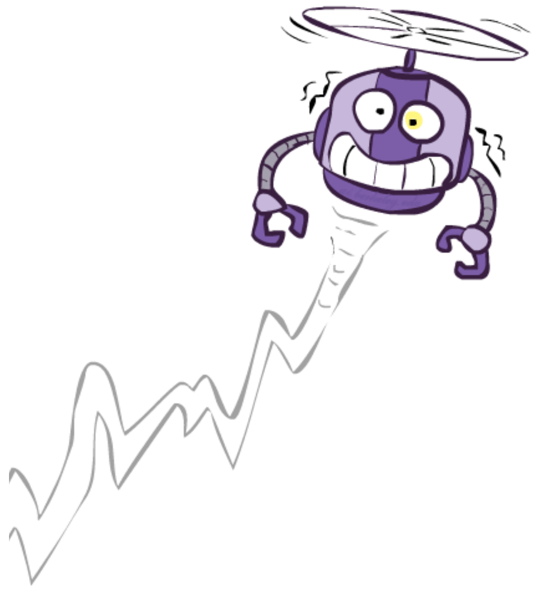


Simulated Annealing

- Is this convergence an interesting guarantee?

- Sounds like magic, but reality is reality:
  - The more downhill steps you need to escape a local optimum, the less likely you are to ever make them all in a row
  - “Slowly enough” may mean exponentially slowly
  - Random restart hillclimbing also converges to optimal state…

- Simulated annealing and its relatives are a key workhorse in VLSI layout and other optimal configuration problems



Local beam search (LBS)

Or, K chosen randomly with a bias towards good ones

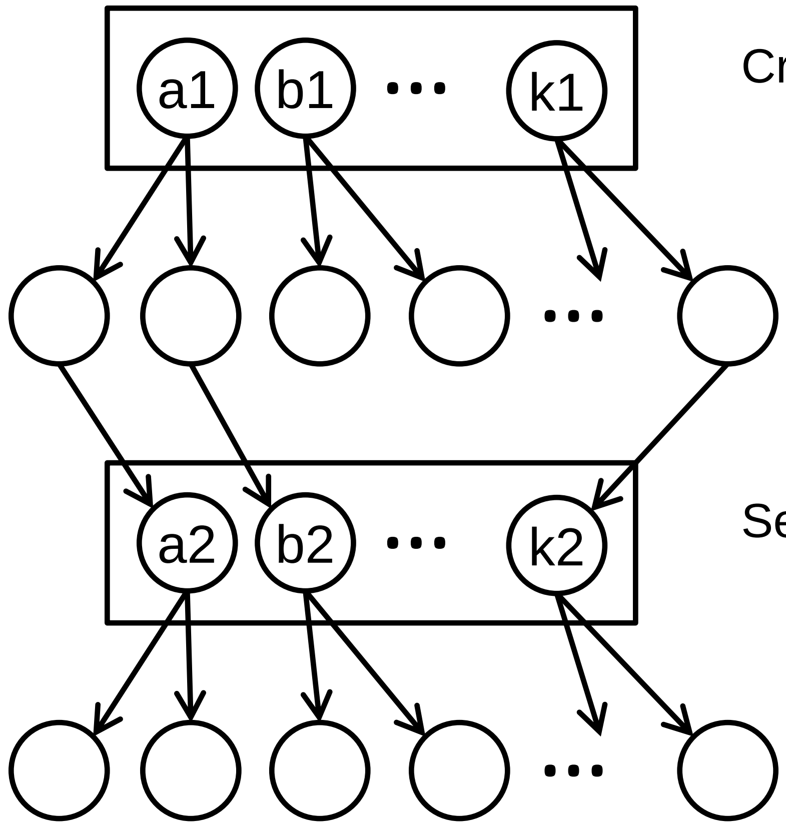
- Basic idea:
  - *K* copies of a local search algorithm, initialized randomly
  - For each iteration
    - Generate ALL successors from *K* current states
    - Choose $\boxed{\text{best } K}$ of these to be the new current states
- This is similar to k searches running in parallel!
- LBS≠ running k random restarts in parallel instead of sequence.

Local beam search (LBS)

- Idea:
  - Unlike Hill Climbing, keep k states instead of just 1

- Algorithm:
  - Local Beam Search keeps track *of k* states rather than just one.
  - It starts with *k* randomly generated states.
  - At each step, all the successors of all the states are generated.
  - If any one is a goal, the algorithm halts, otherwise it selects the *k* best successors from the complete list and repeats.

Local beam search



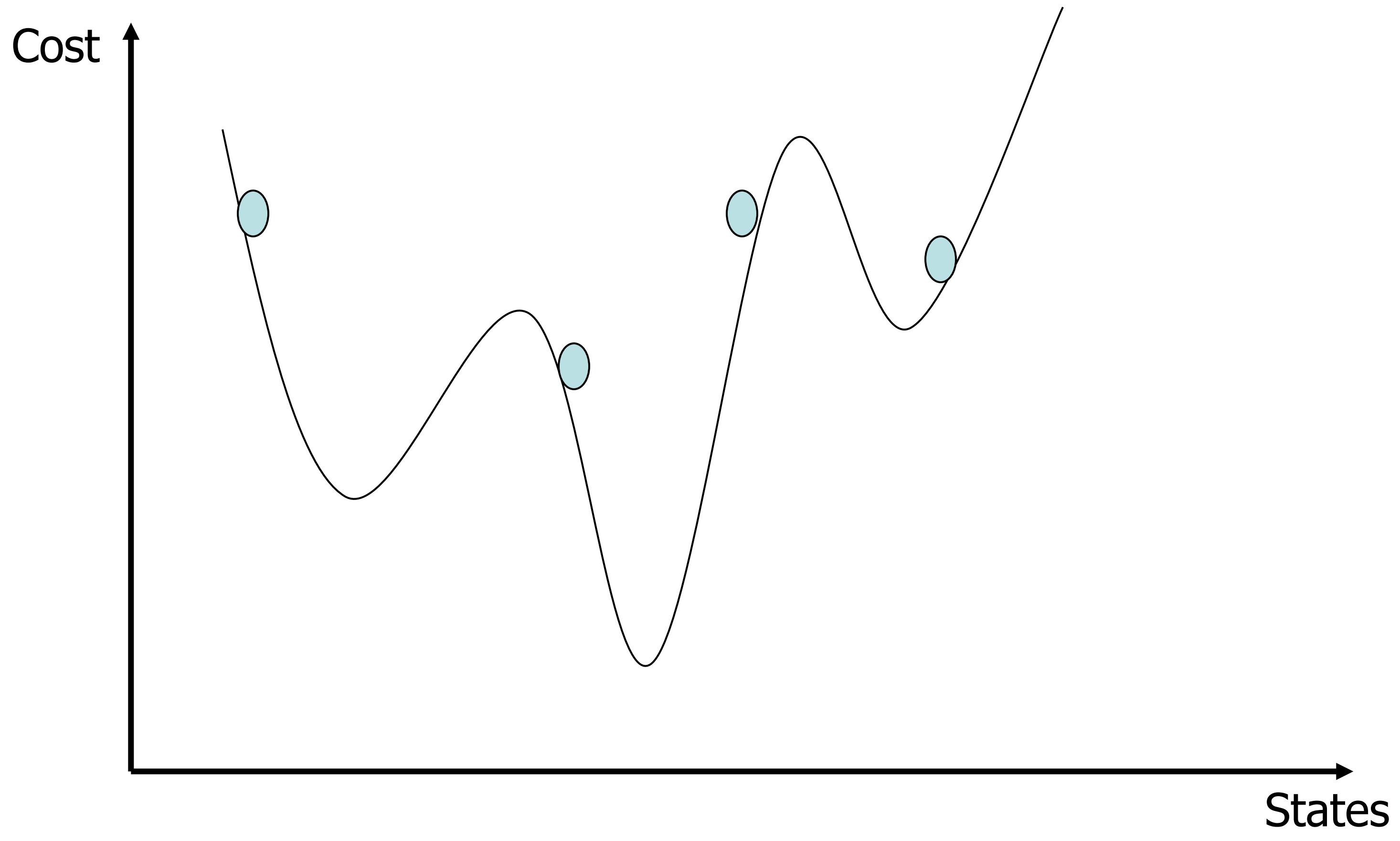
Create *k* random initial states

Generate their children

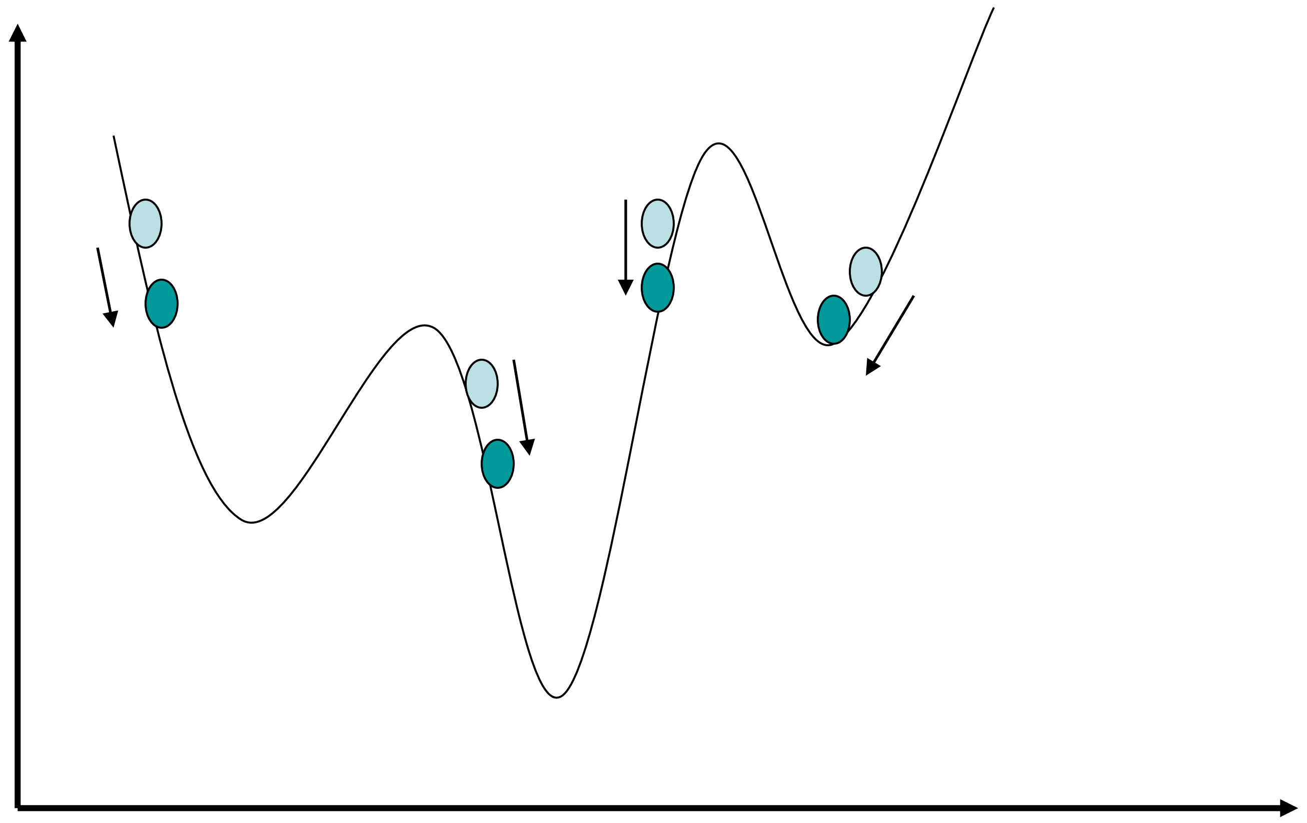
Select the *k* best children

Repeat …

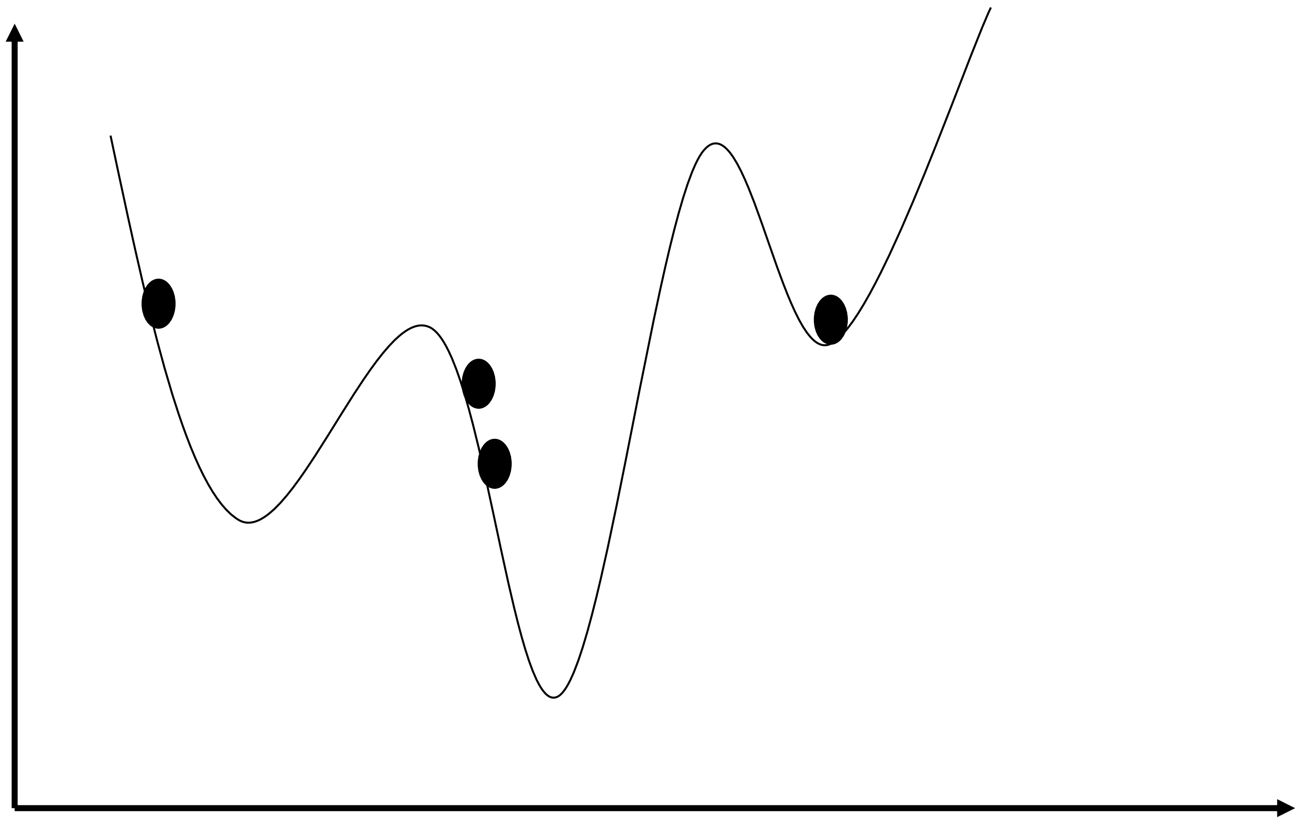
Local Beam Search



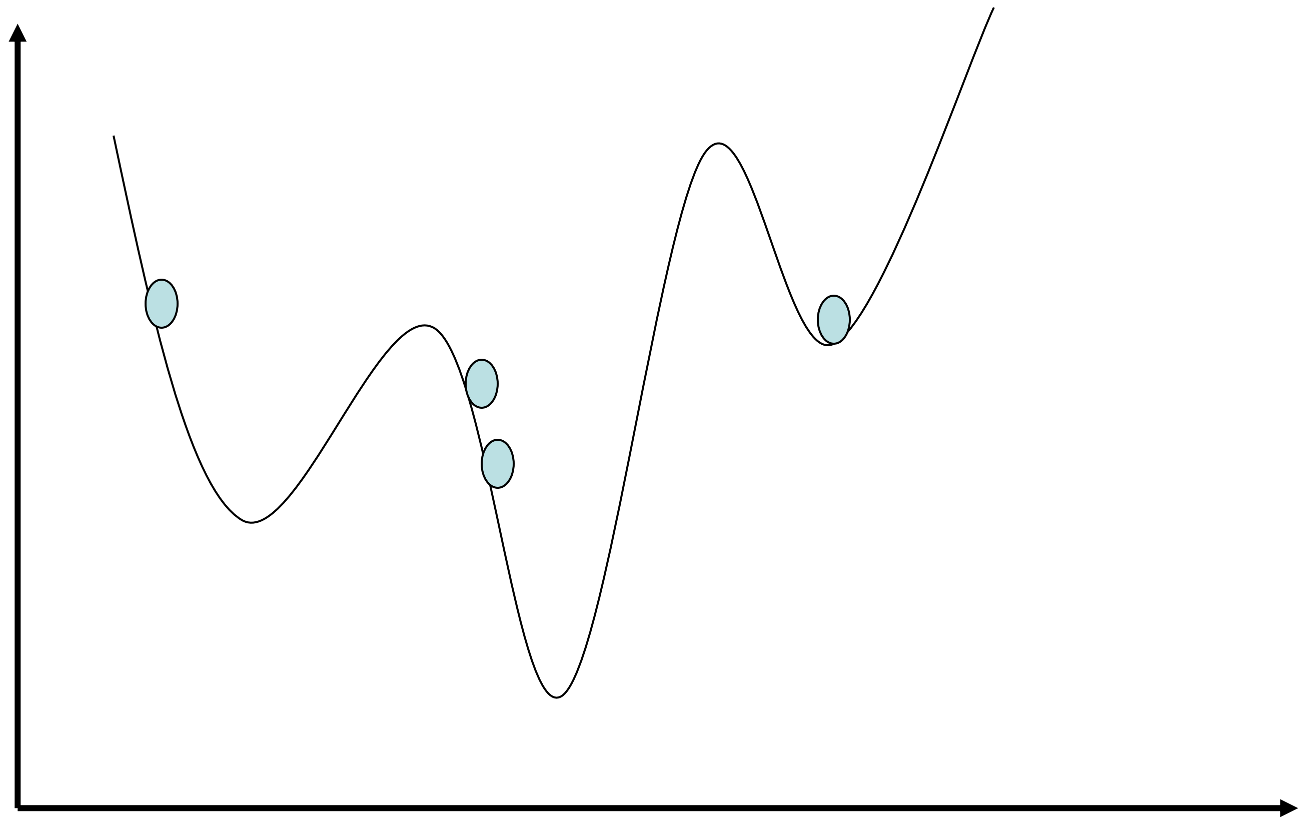
Local Beam Search



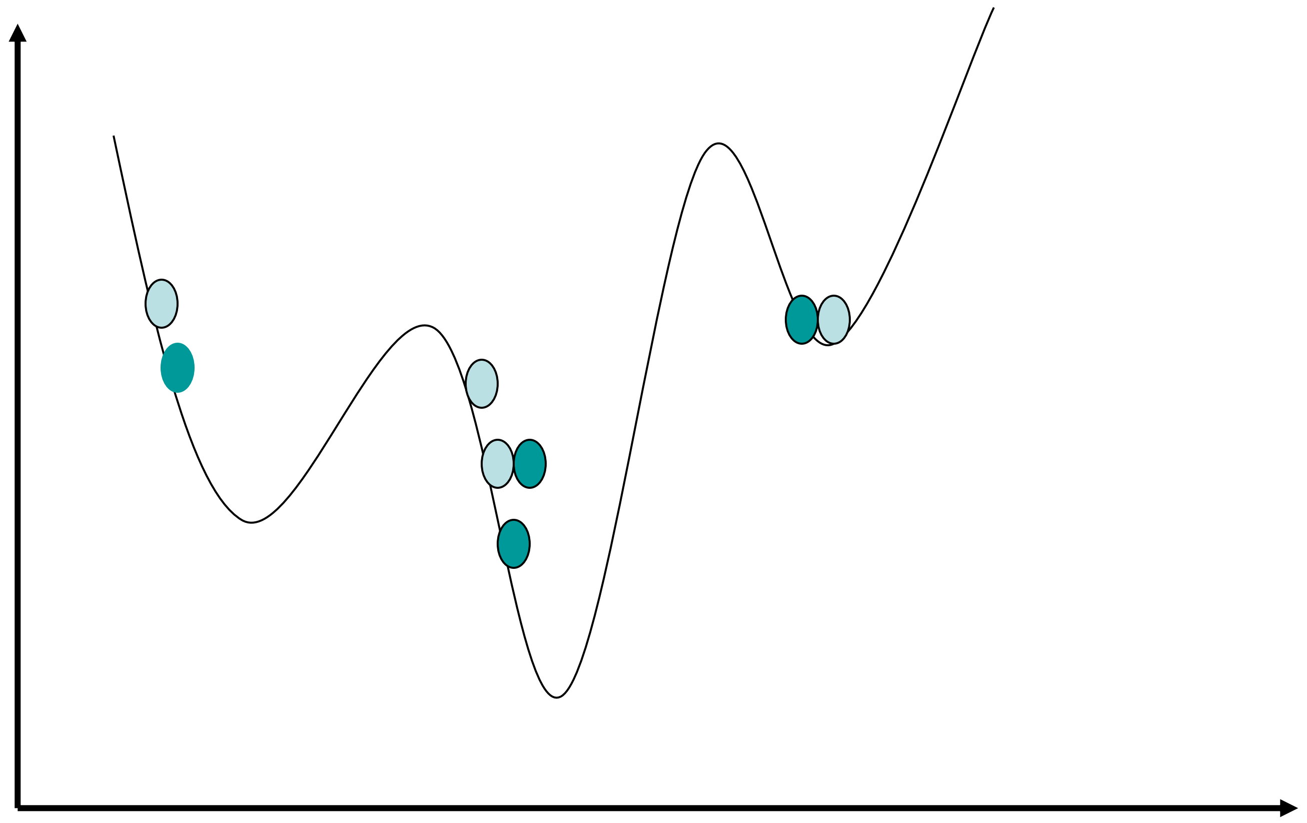
Local Beam Search



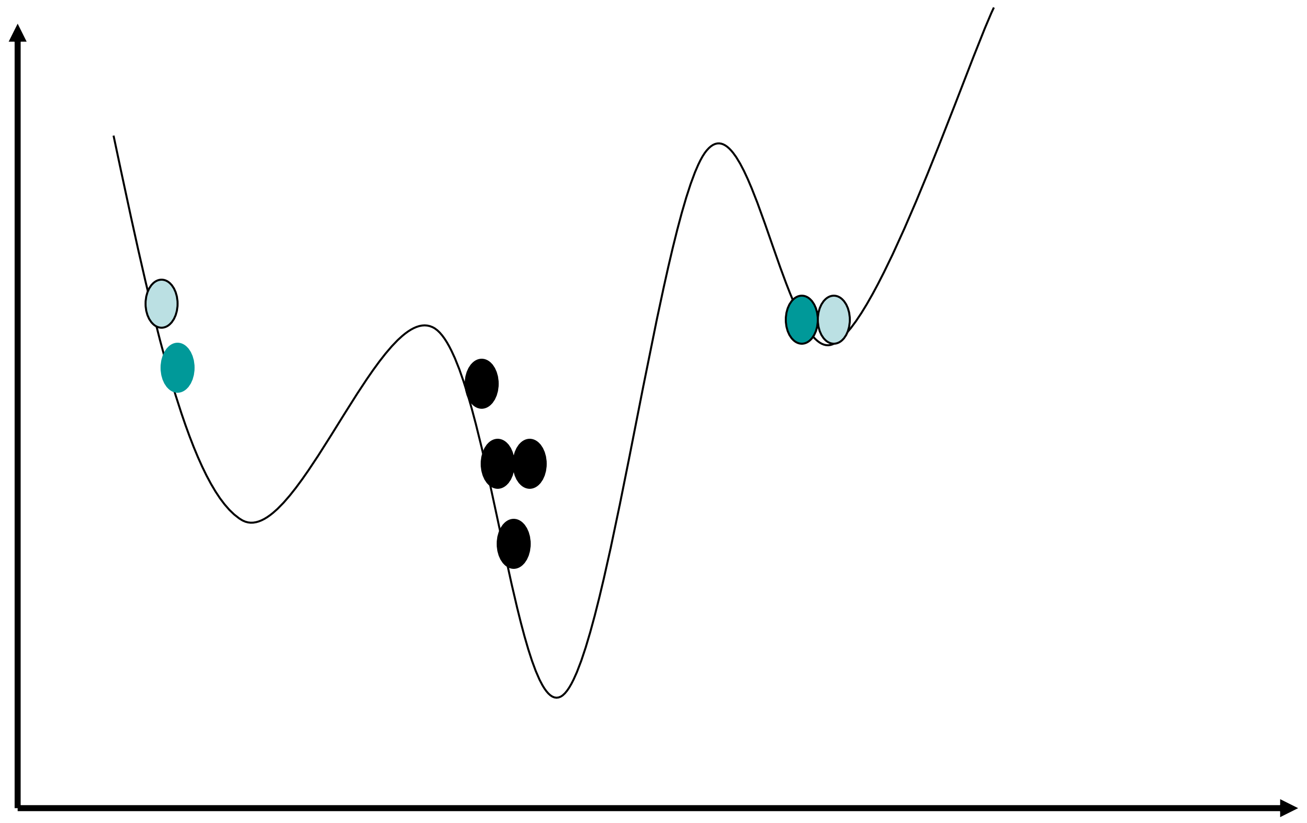
Local Beam Search



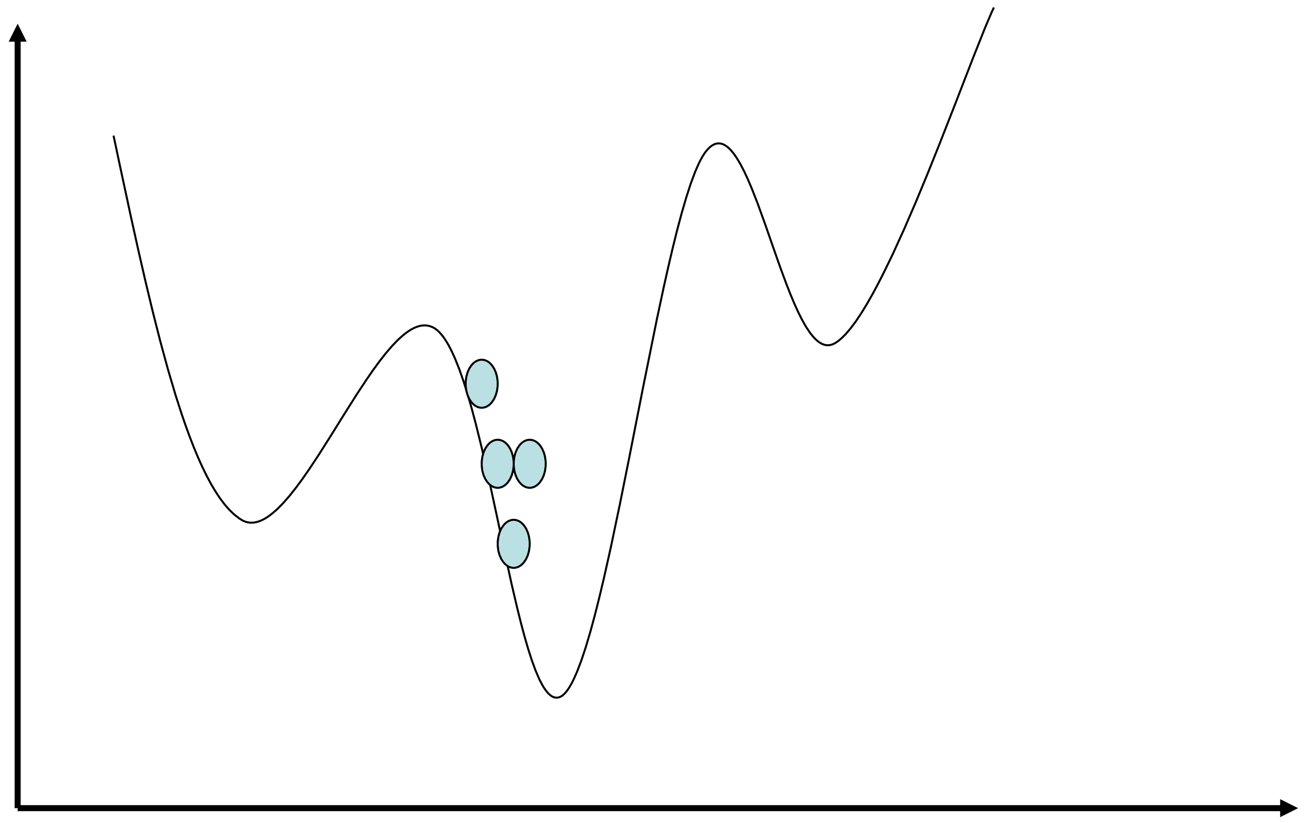
Local Beam Search



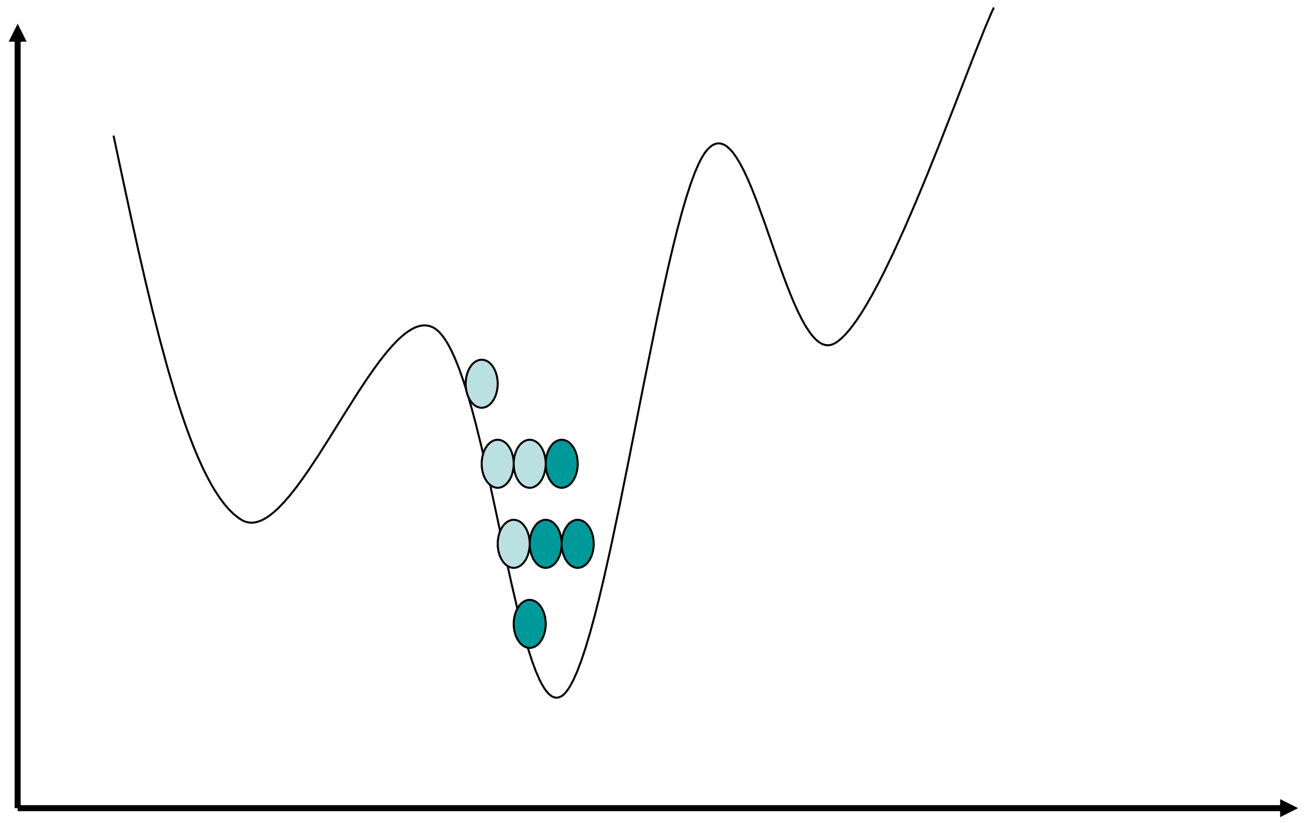
Local Beam Search



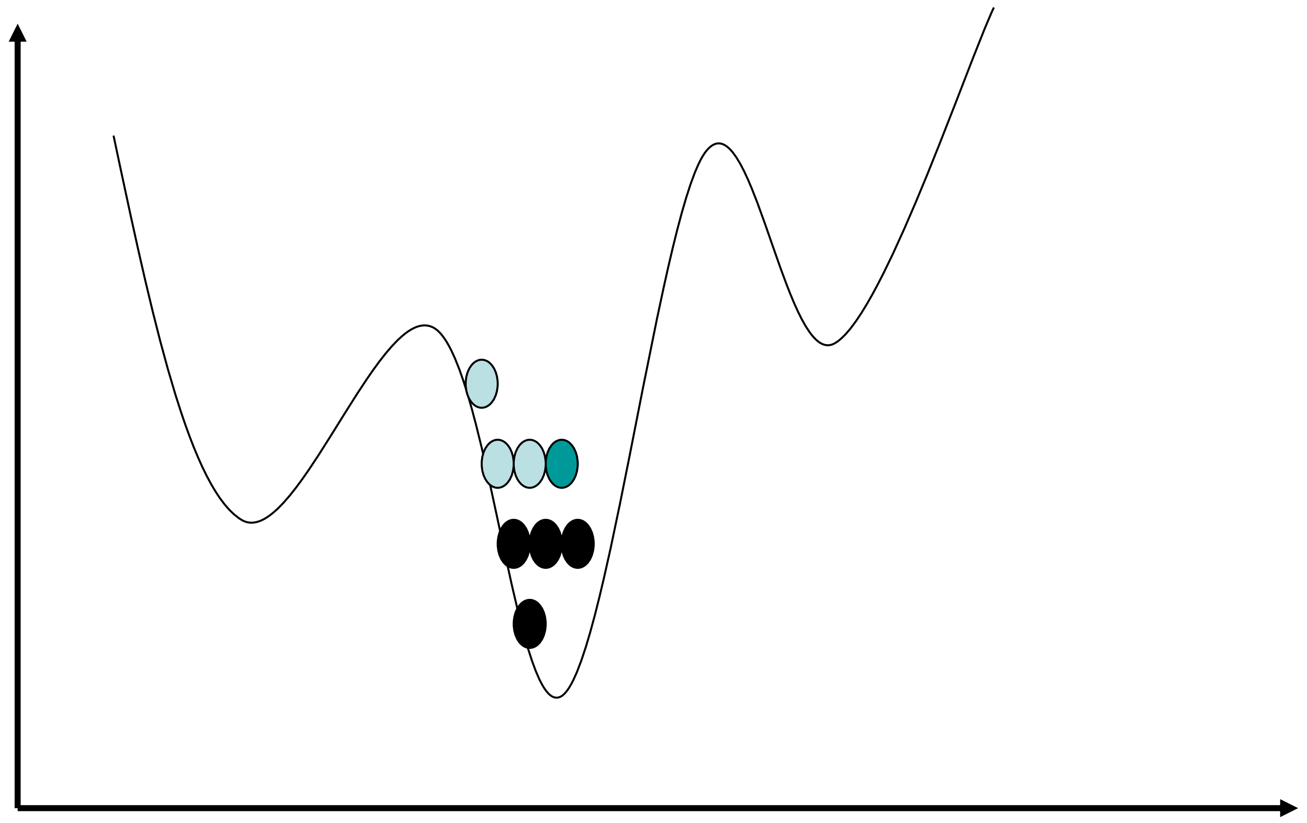
Local Beam Search



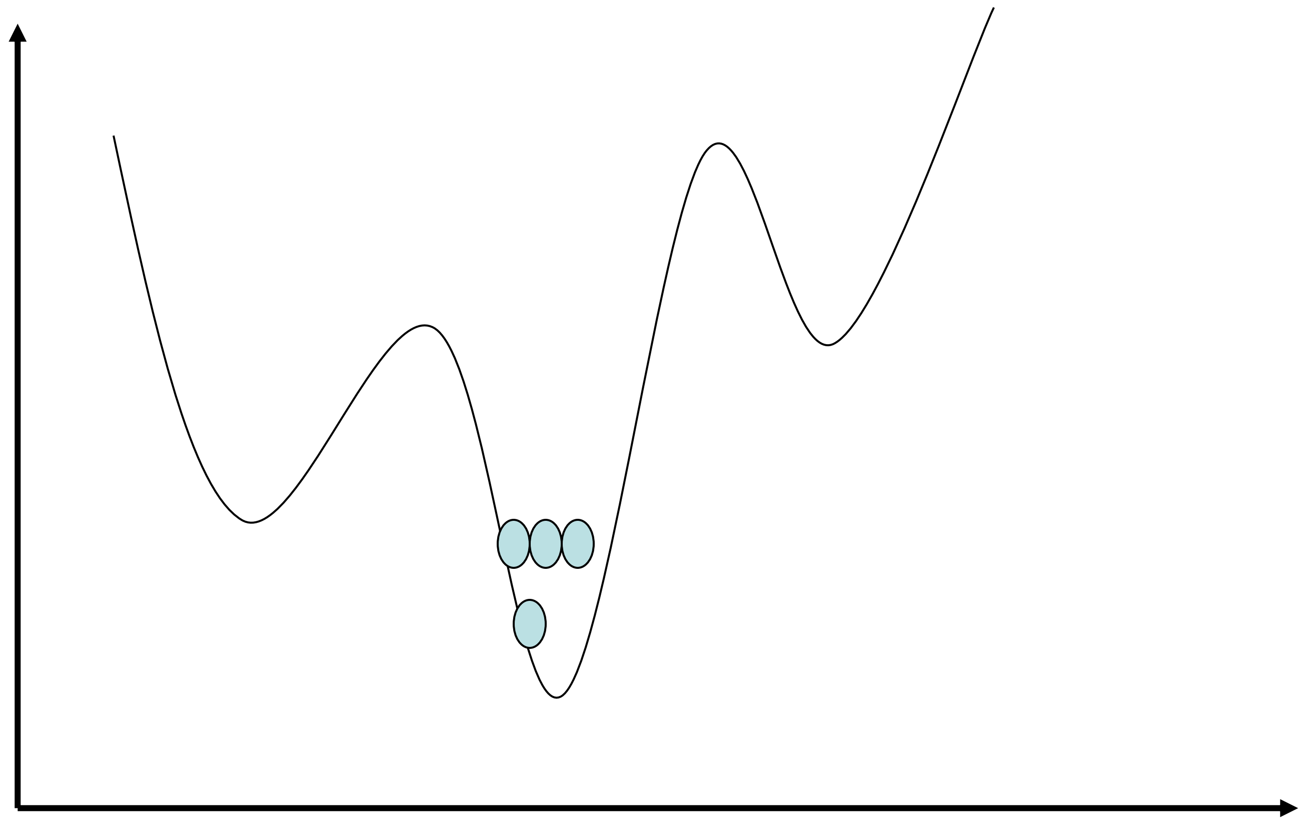
Local Beam Search



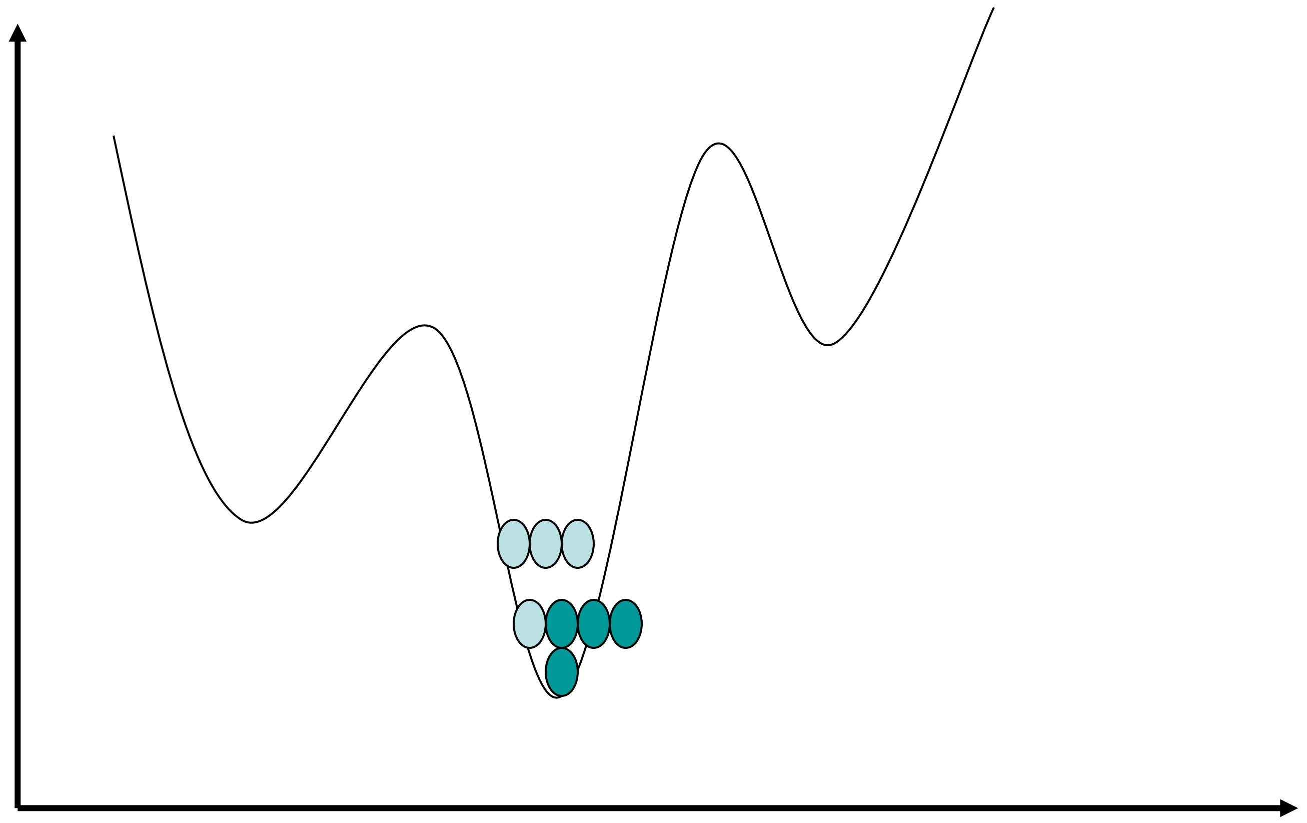
Local Beam Search



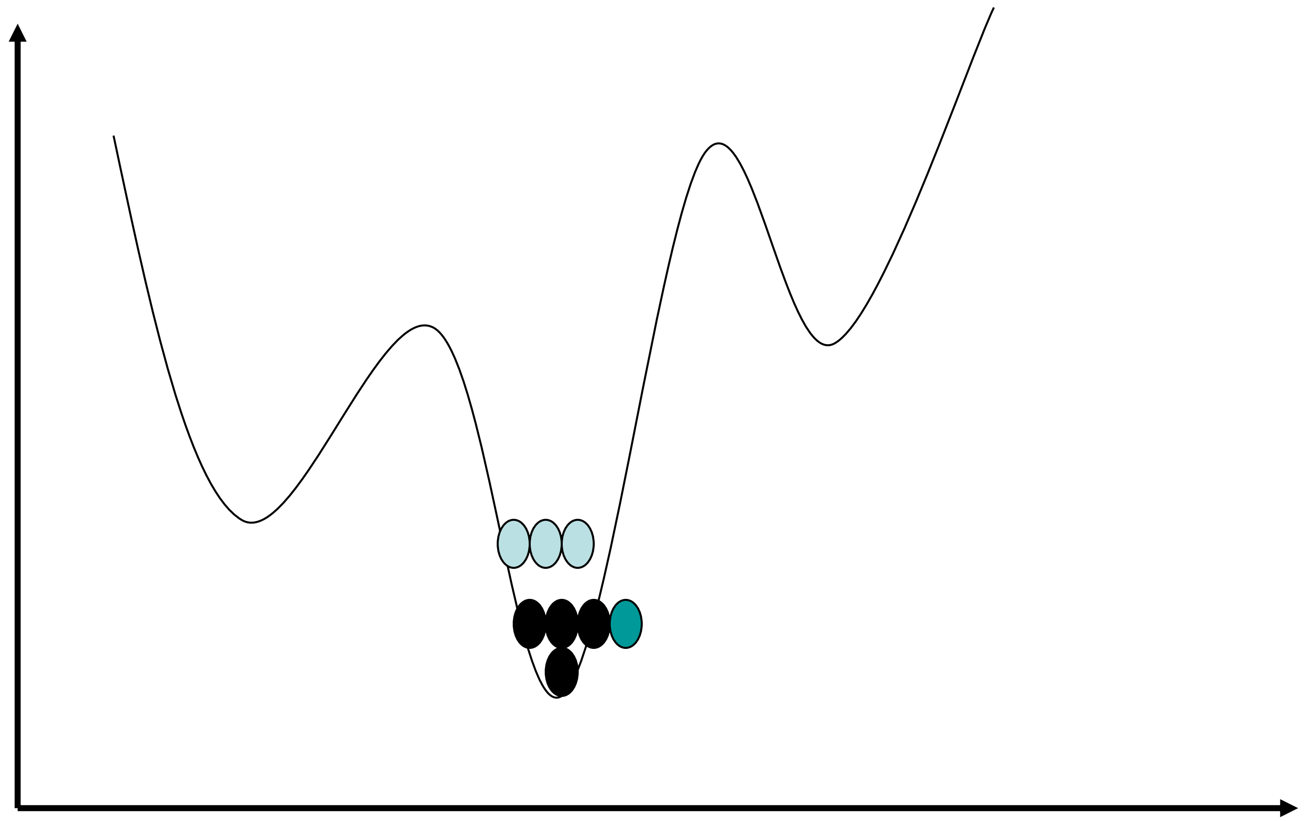
Local Beam Search



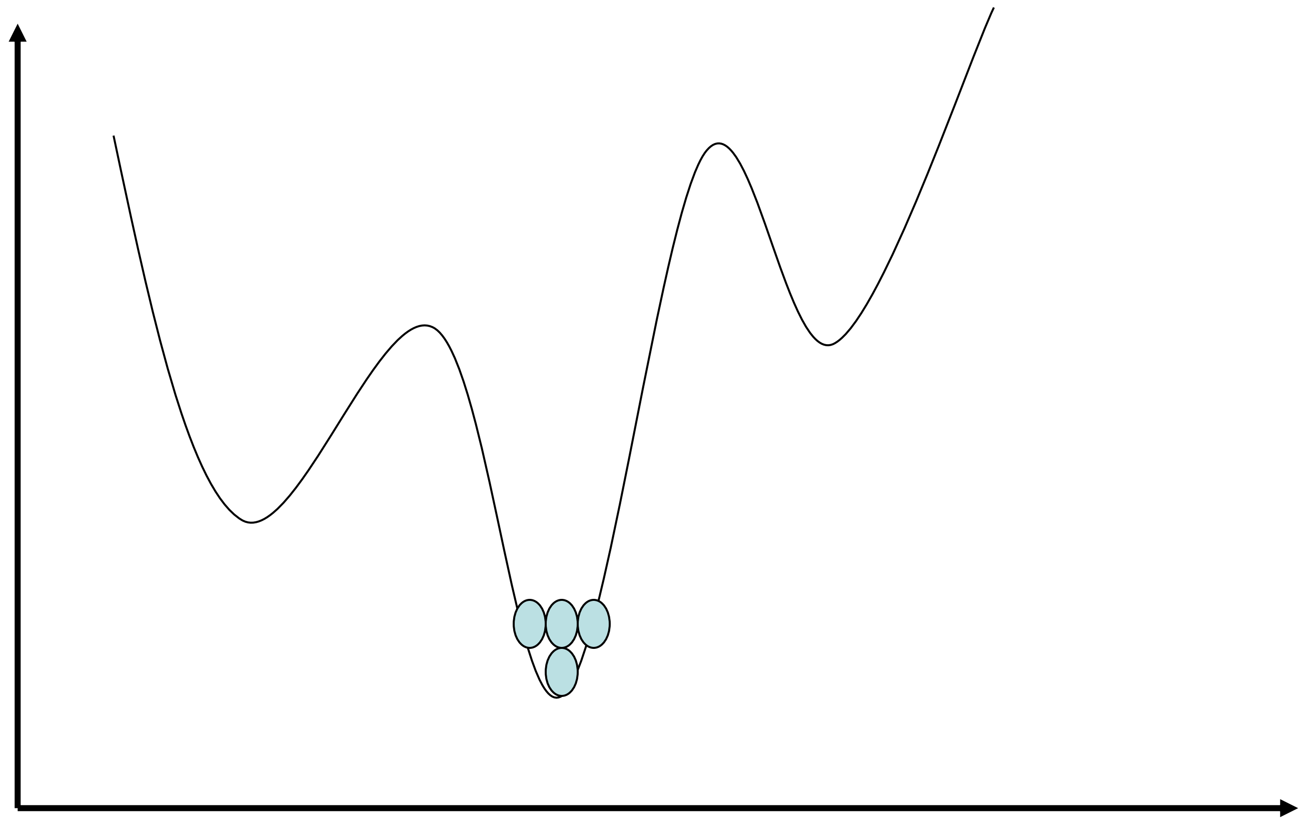
Local Beam Search



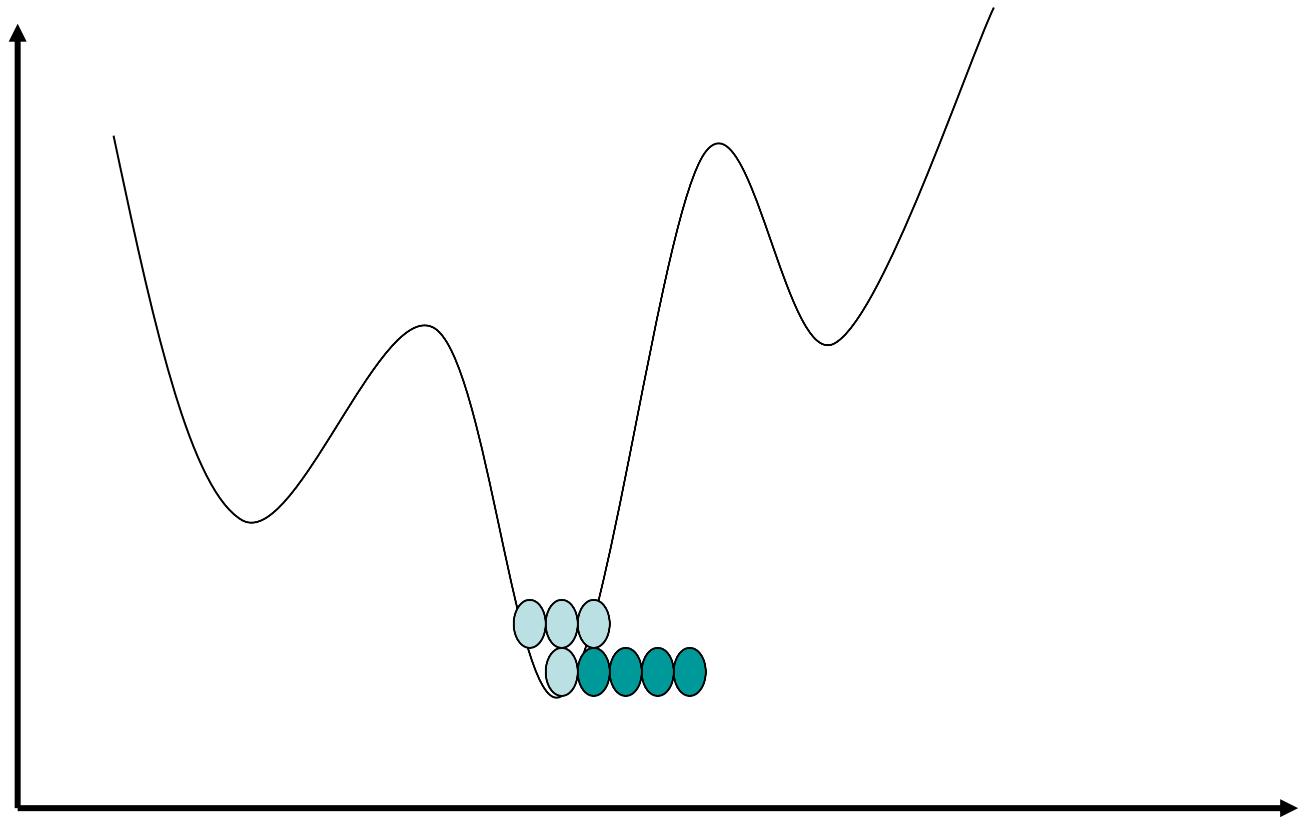
Local Beam Search



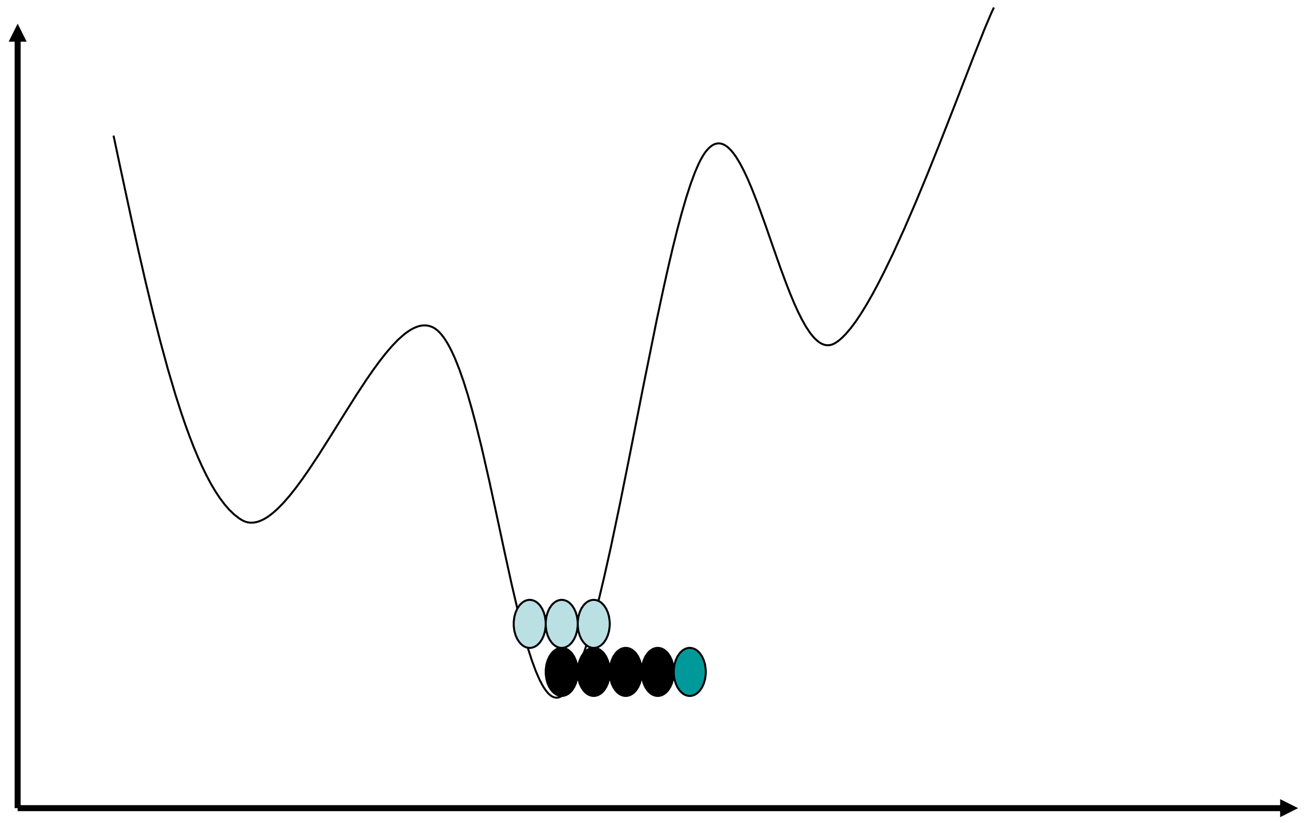
Local Beam Search



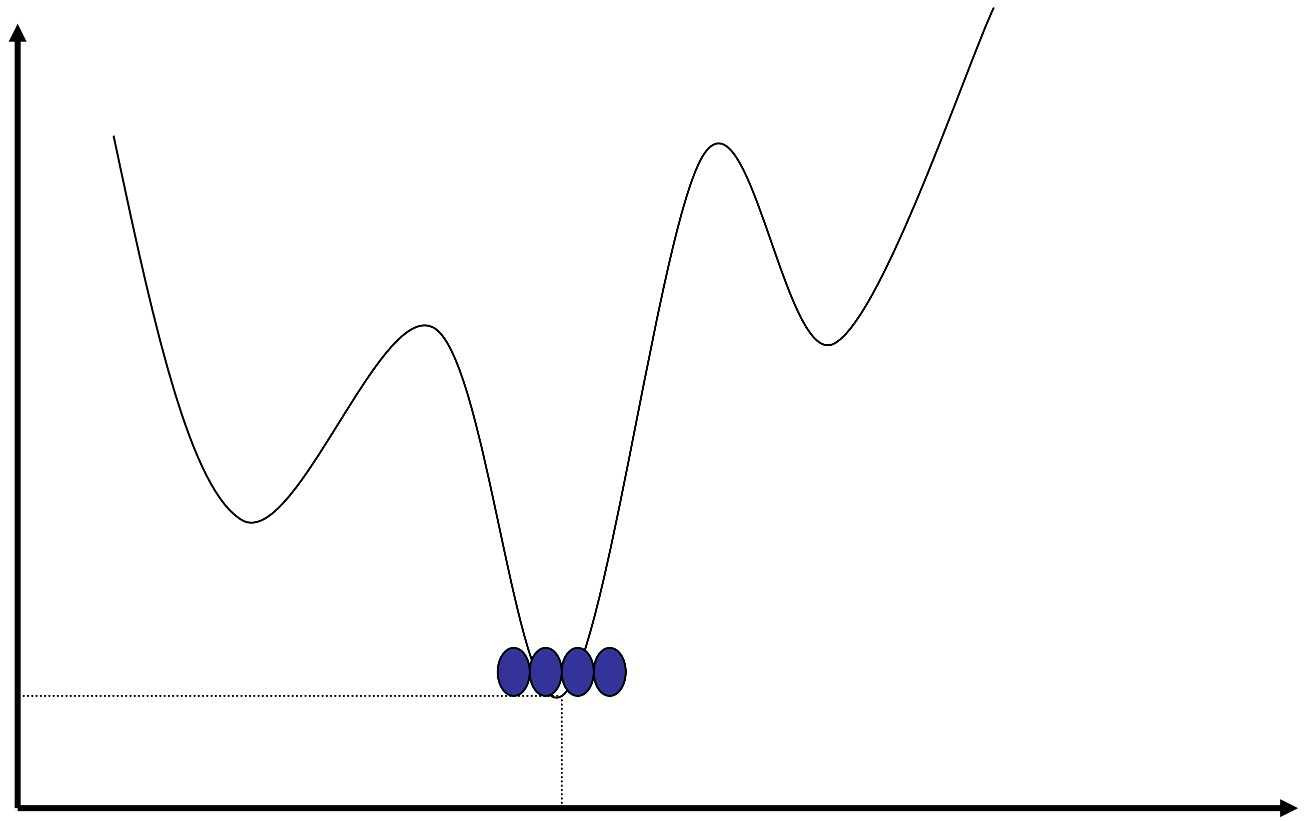
Local Beam Search



Local Beam Search



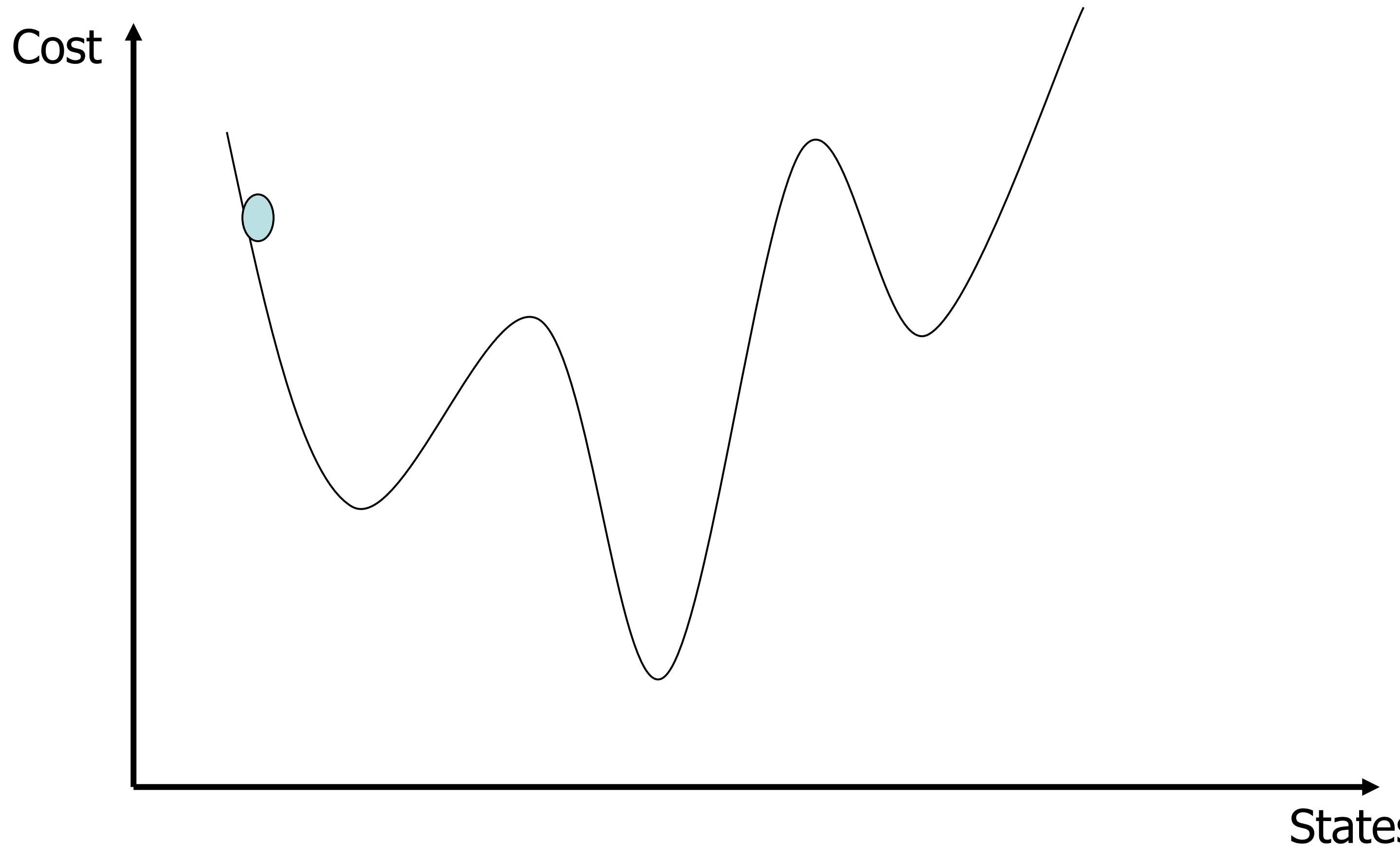
Local Beam Search



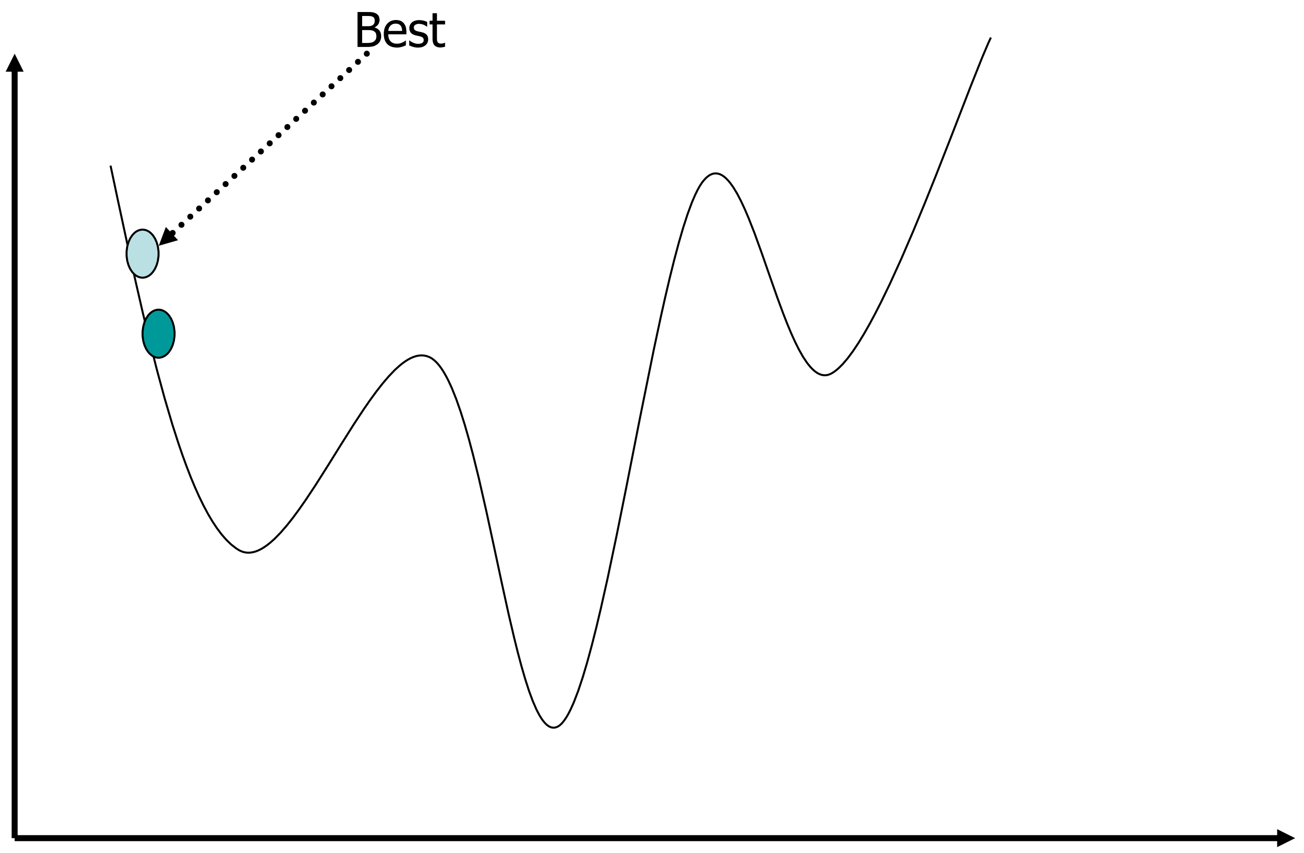
Tabu search

- Tabu search is a meta-heuristic optimization technique, which owes its name to its memory structures, used to store recently evaluated candidate solutions.
  - The candidates stored in these structures are not eligible for generation of further candidates and are thereby considered “Tabu” by the algorithm
  - Key idea
    - maintain the sequence of nodes already visited
      - tabu lists and tabu nodes
    - Typically there are two kinds of tabu lists, a long term memory maintaining the history through all the exploration process as a whole and a short term memory to keep the most recently visited tabu movements.
    - Select the best configurations that is not tabu,
      i.e., has not been visited before
- Tabu search enhances the performance of local search by relaxing its basic rule.
  - First, at each step worsening moves can be accepted if no improving move is available (like when the search is stuck at a strict local minimum).
  - In addition, prohibitions (henceforth the term tabu) are introduced to discourage the search from coming back to previously-visited solutions.

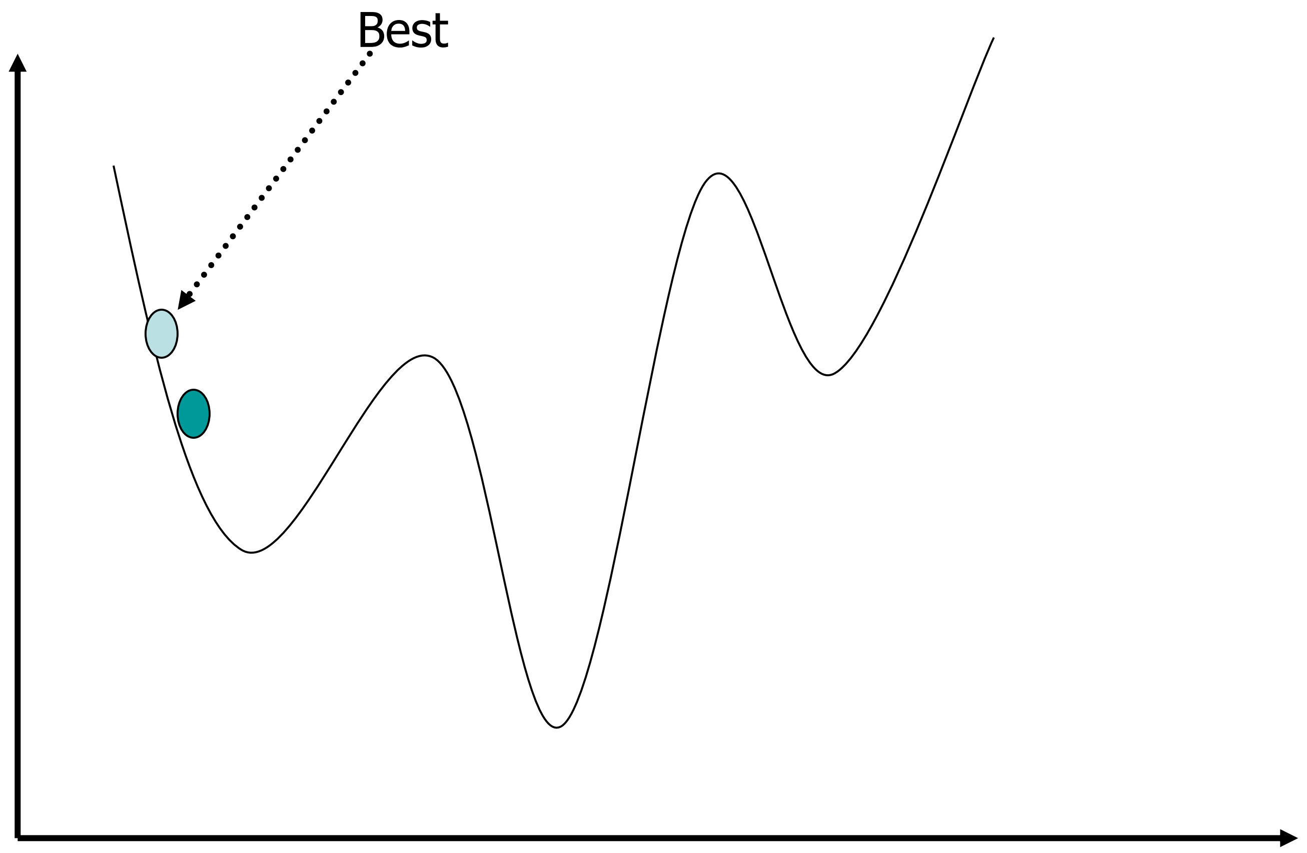
Tabu Search: TS



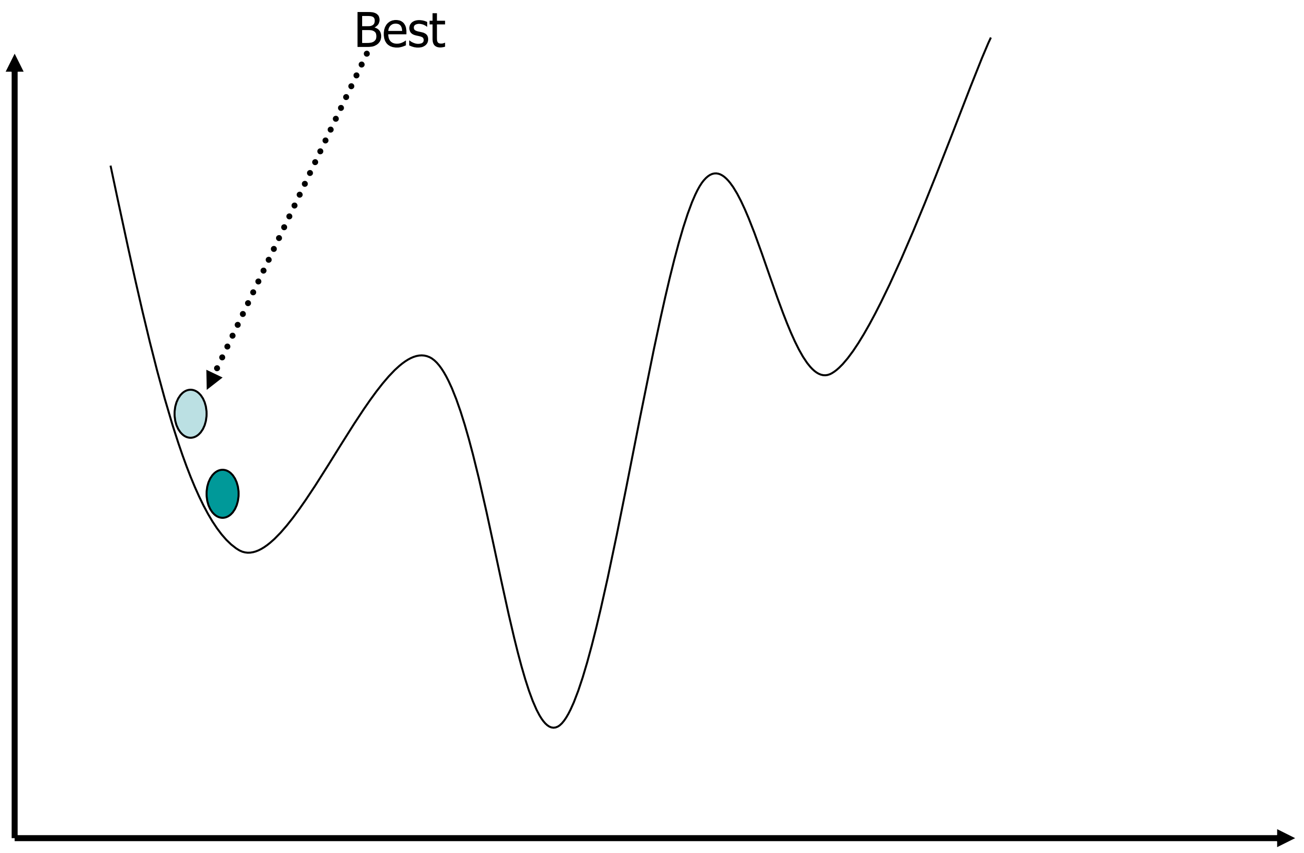
Tabu Search: TS



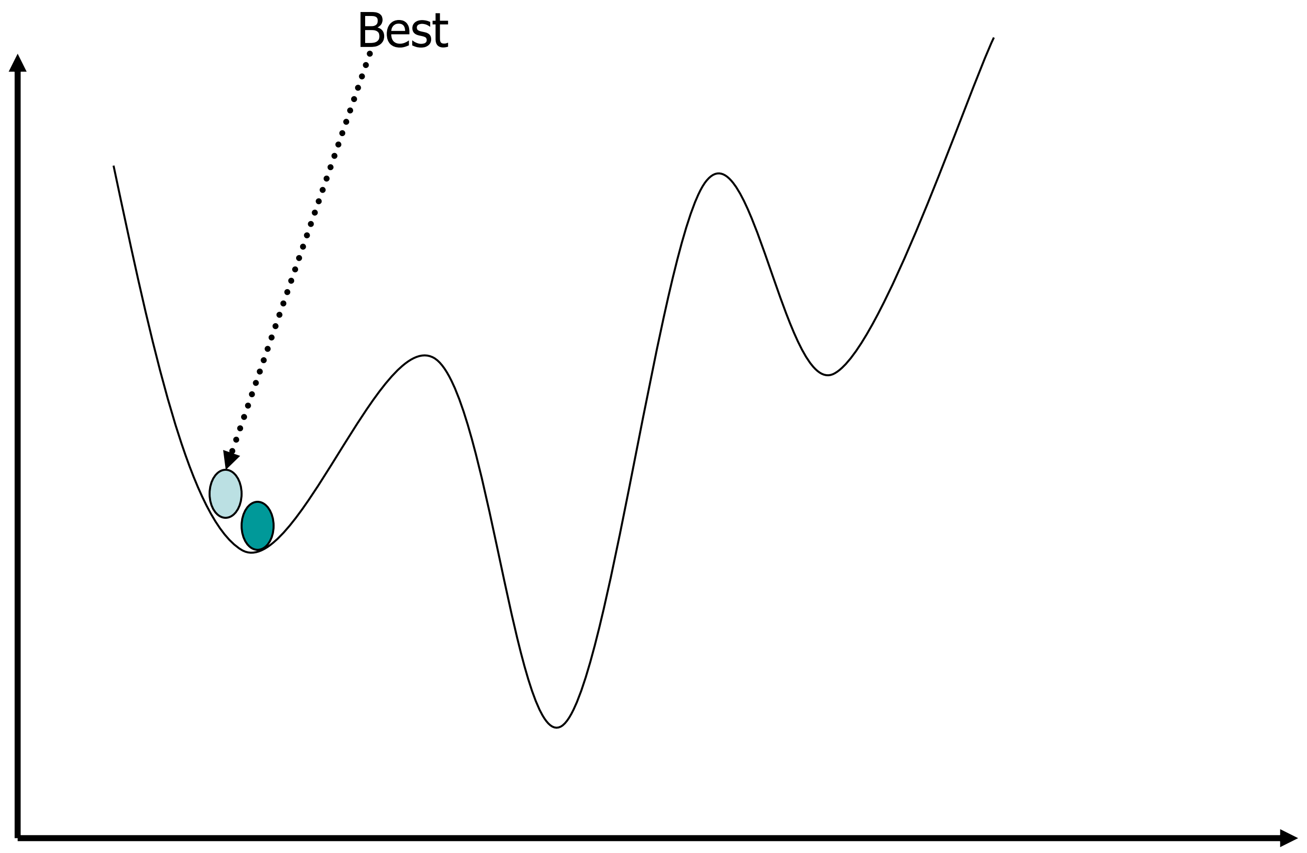
Tabu Search: TS



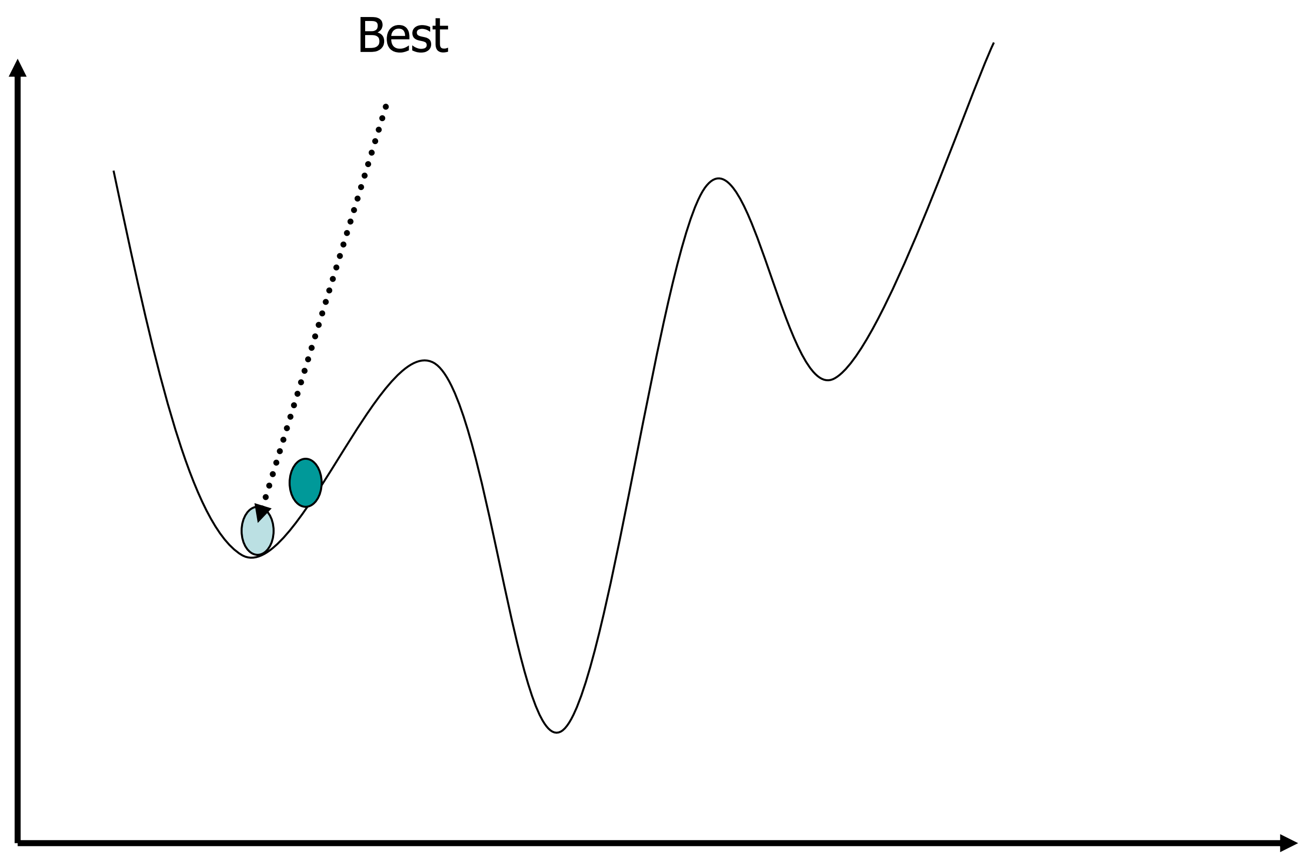
Tabu Search: TS



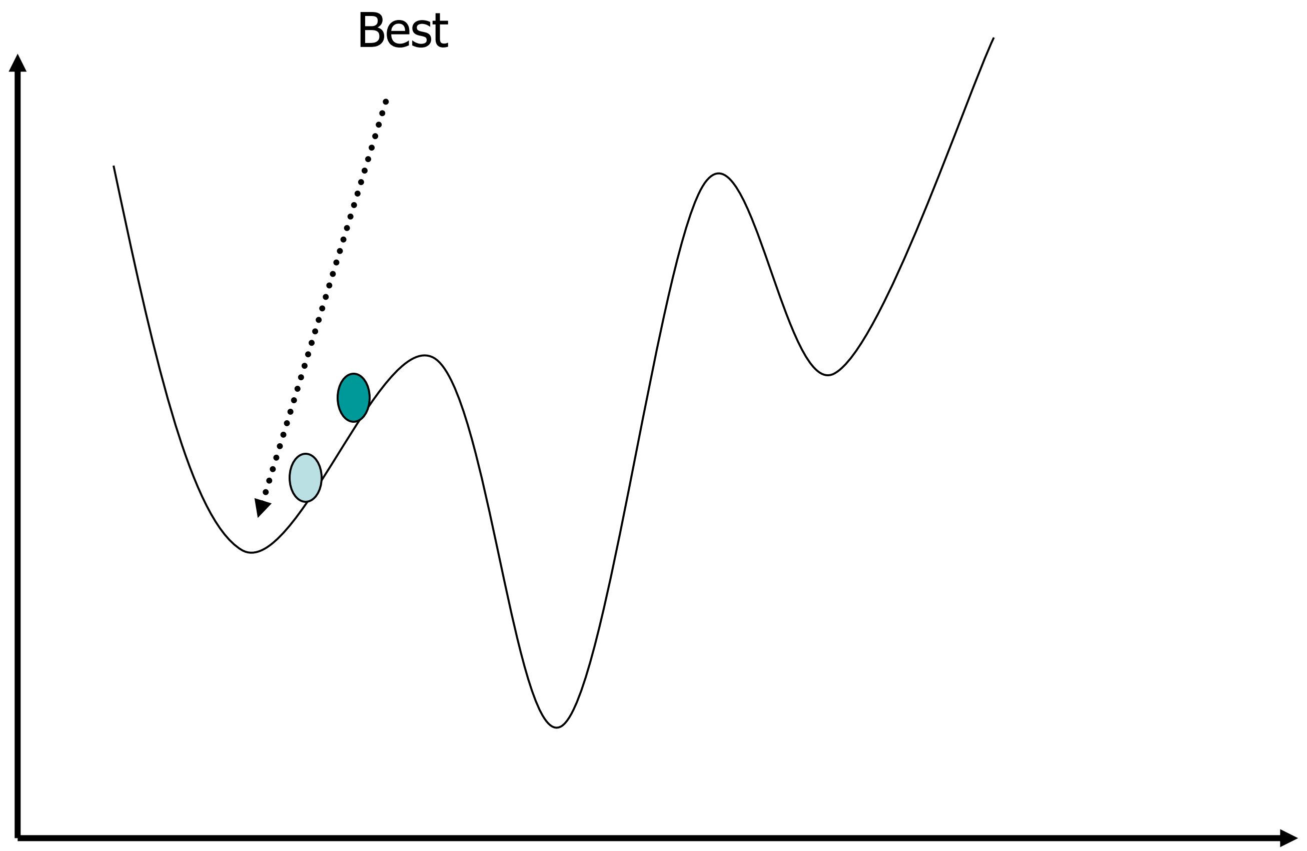
Tabu Search: TS



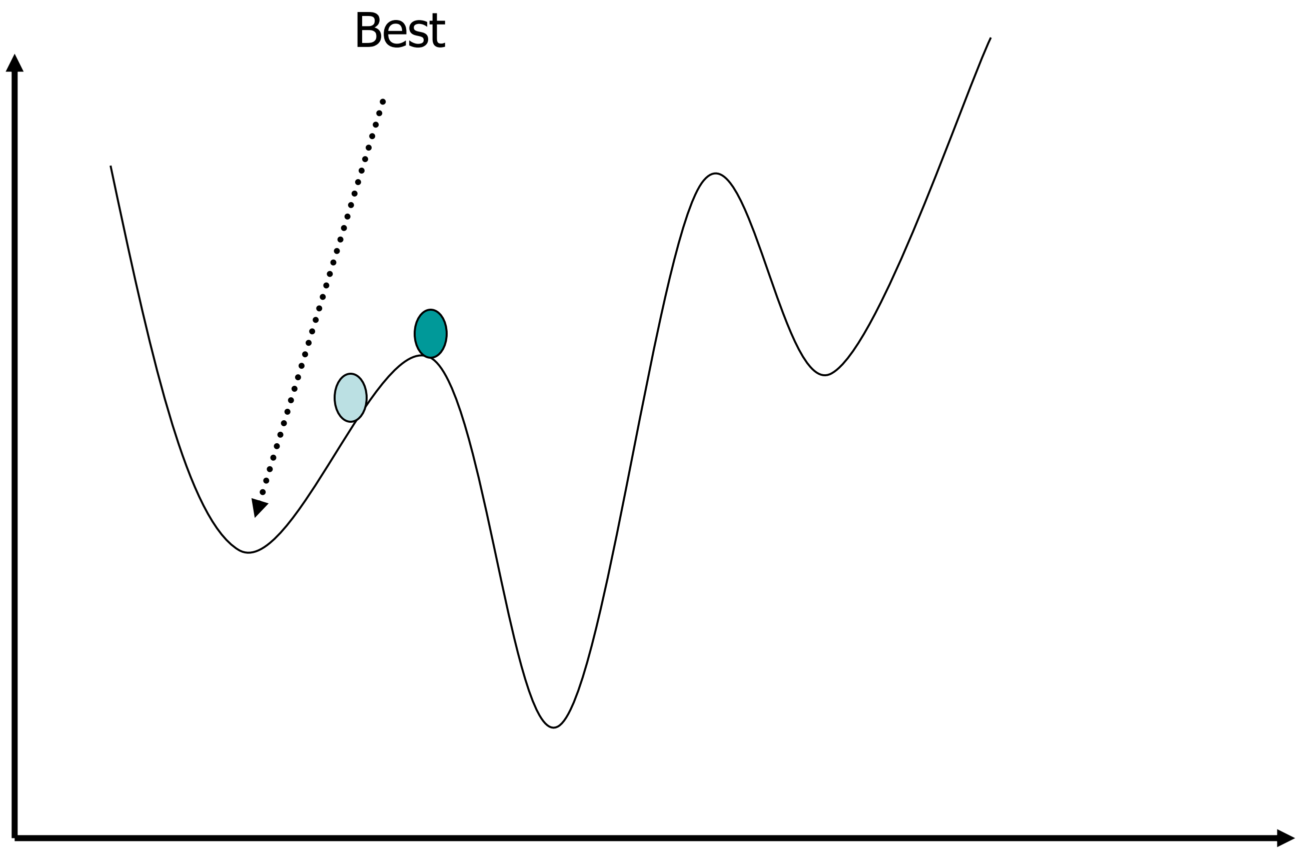
Tabu Search: TS



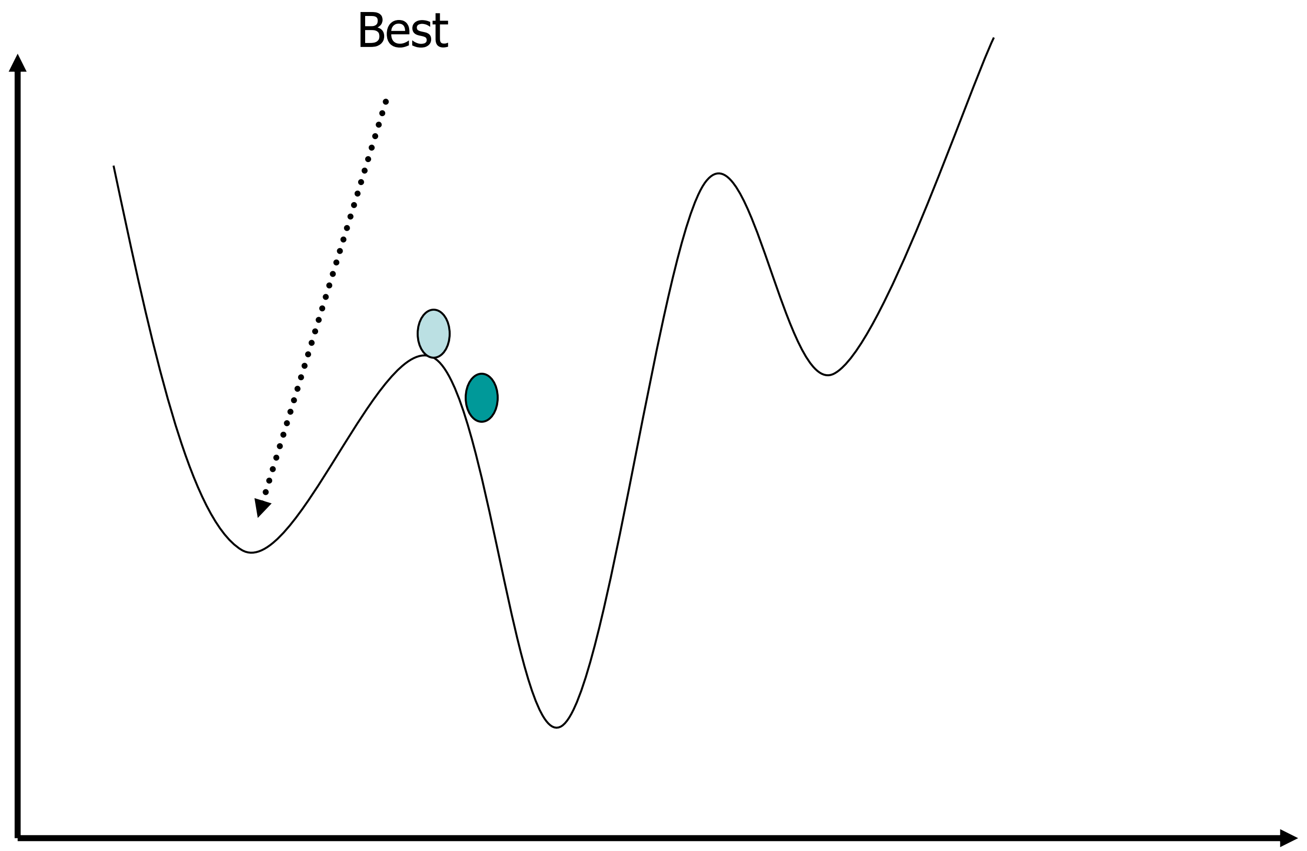
Tabu Search: TS



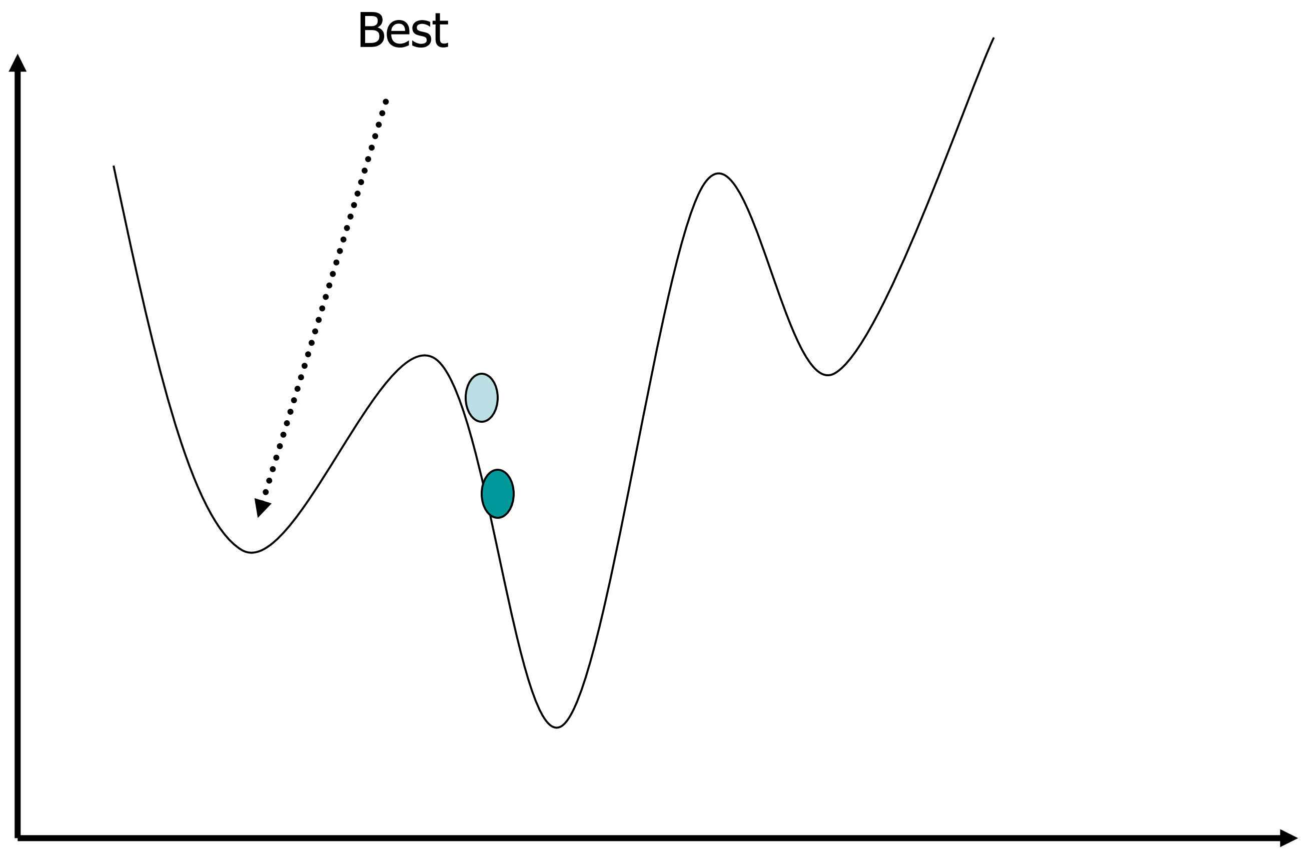
Tabu Search: TS



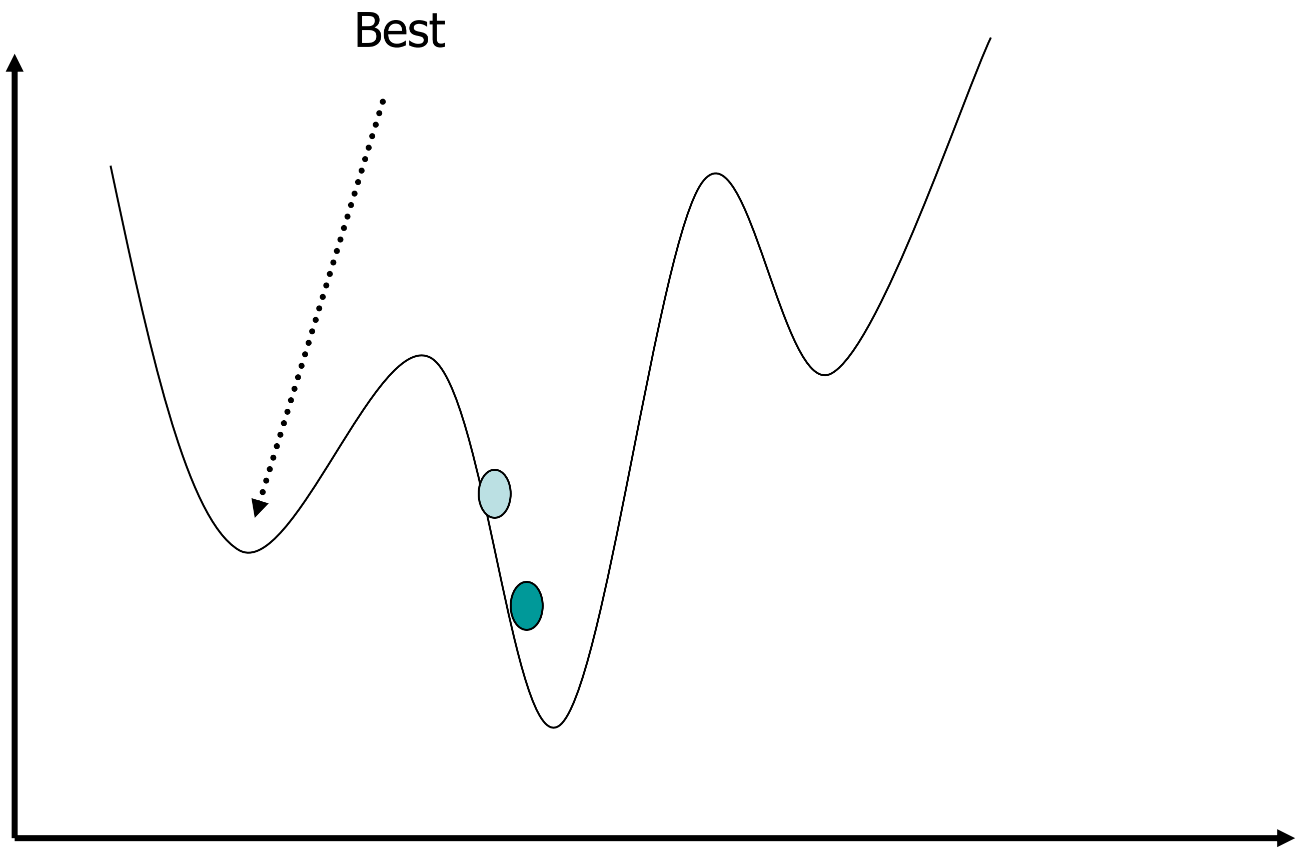
Tabu Search: TS



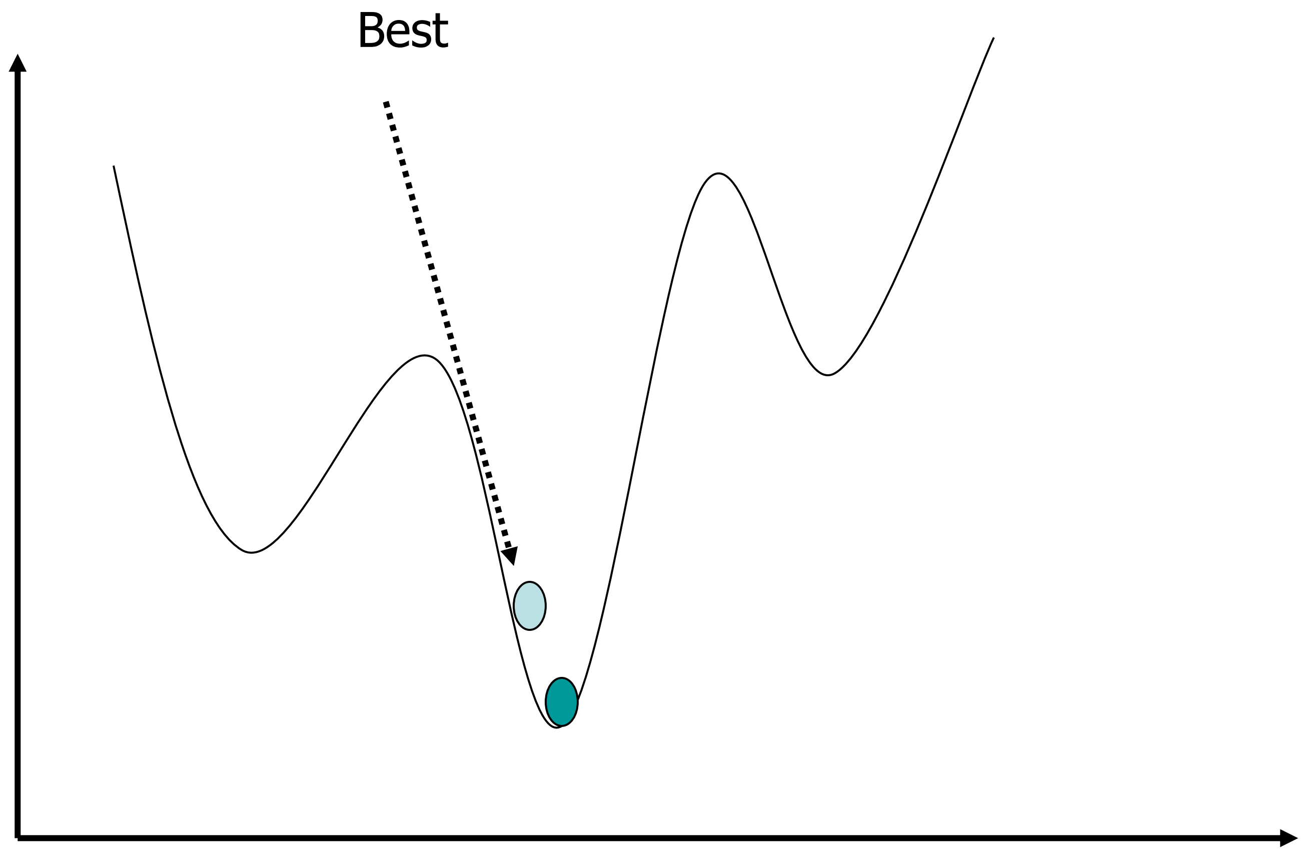
Tabu Search: TS



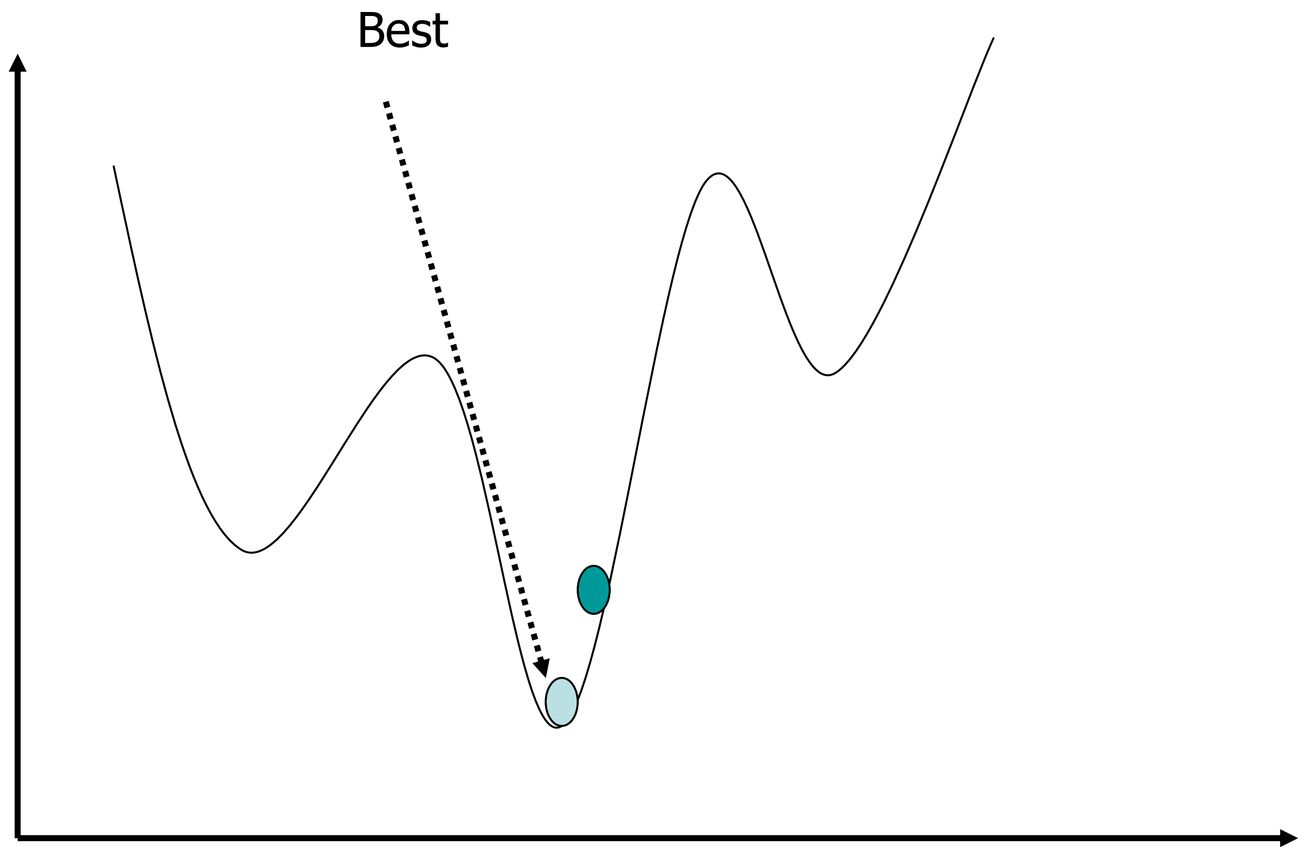
Tabu Search: TS



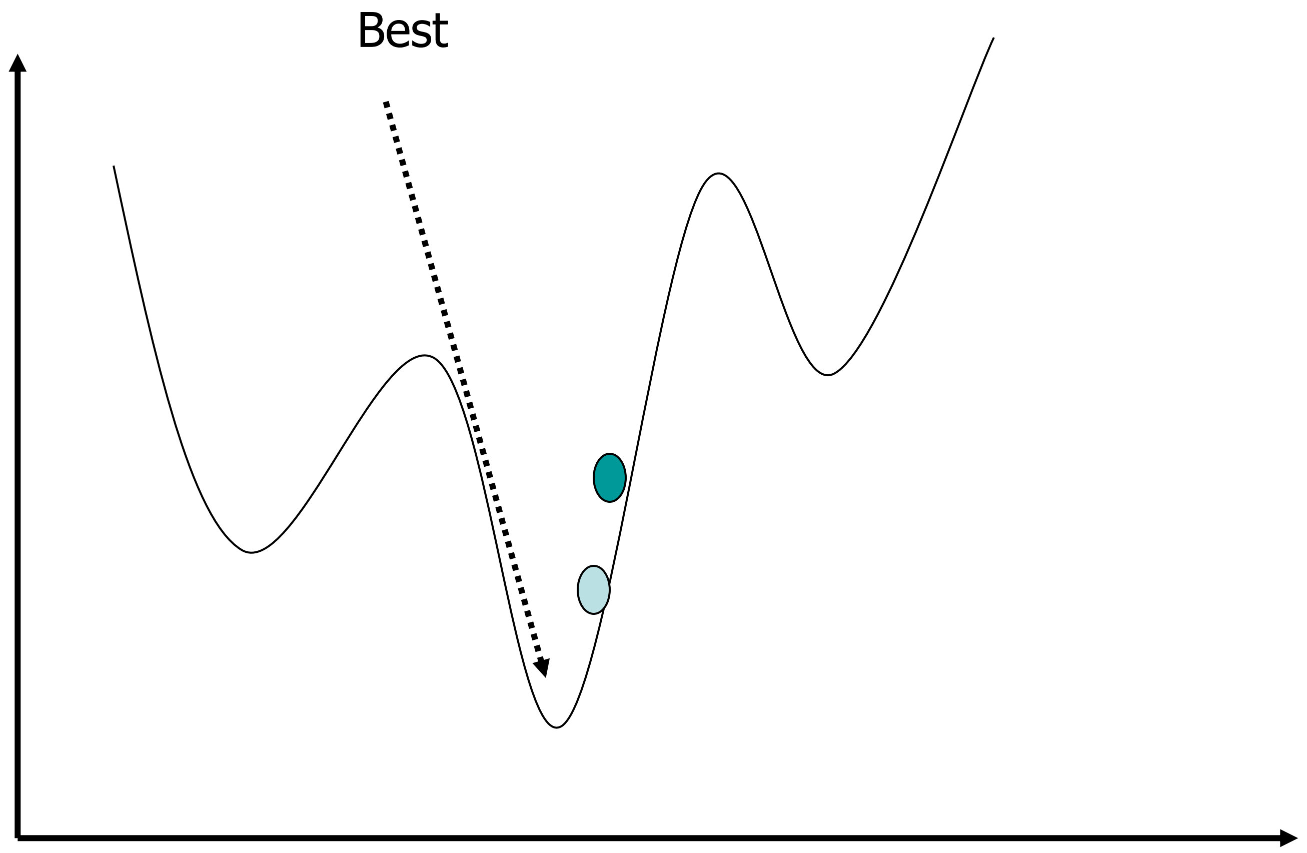
Tabu Search: TS



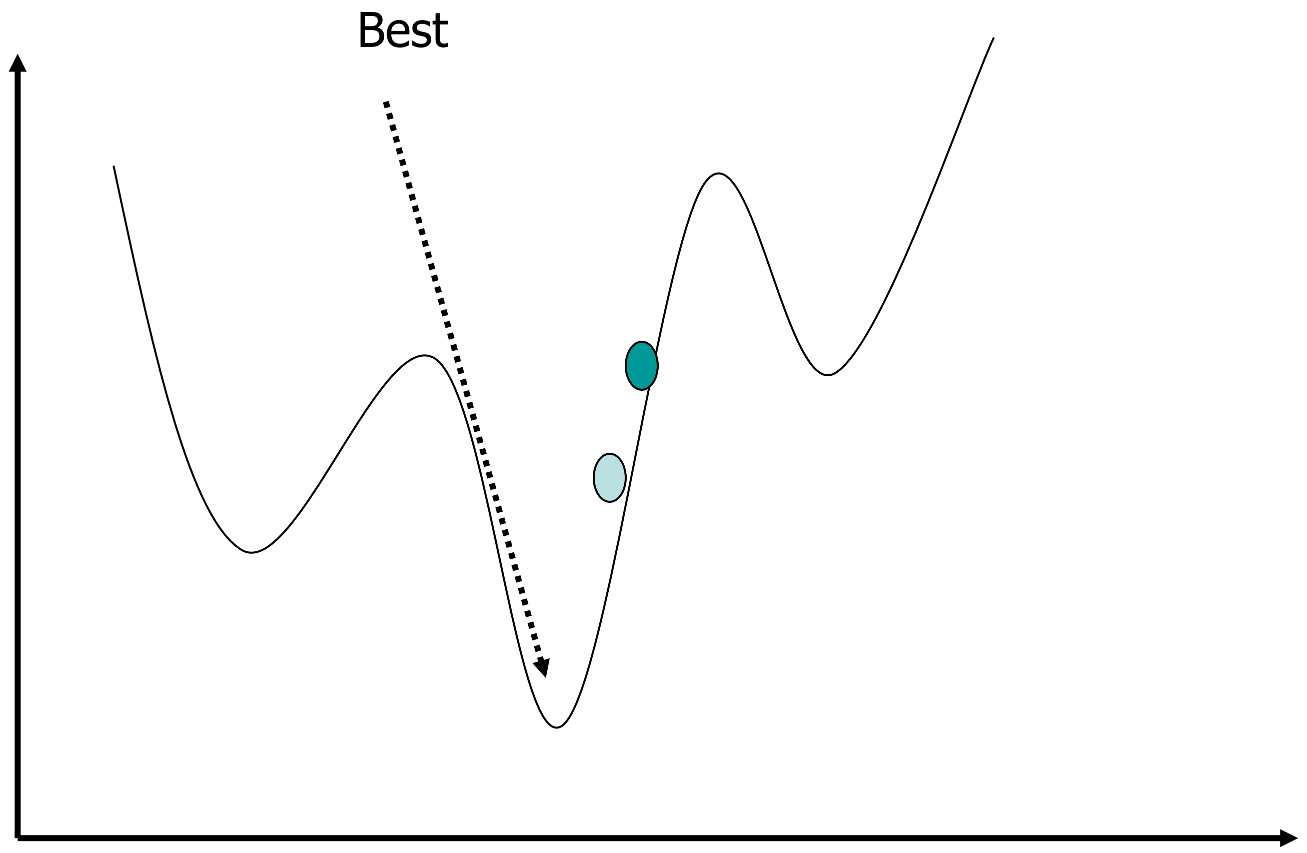
Tabu Search: TS



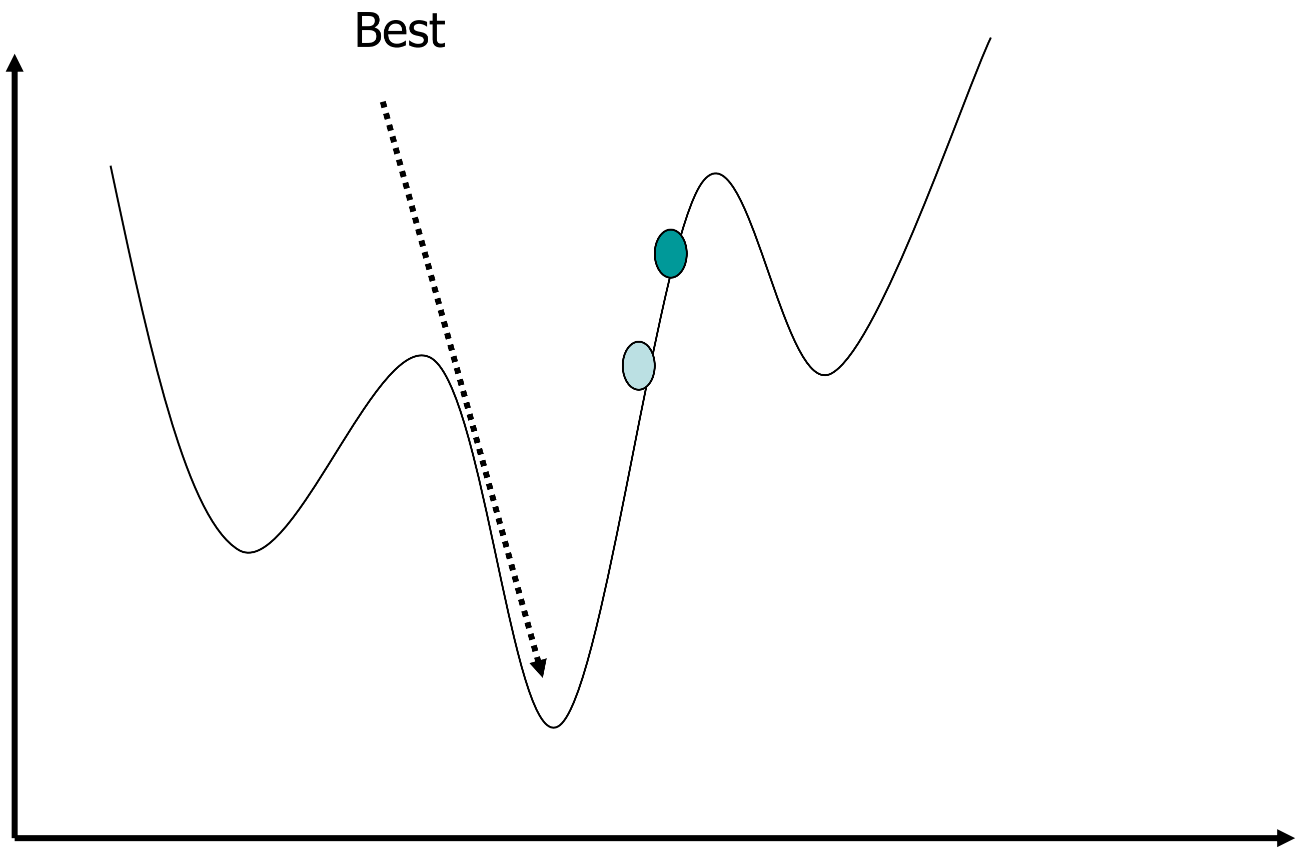
Tabu Search: TS



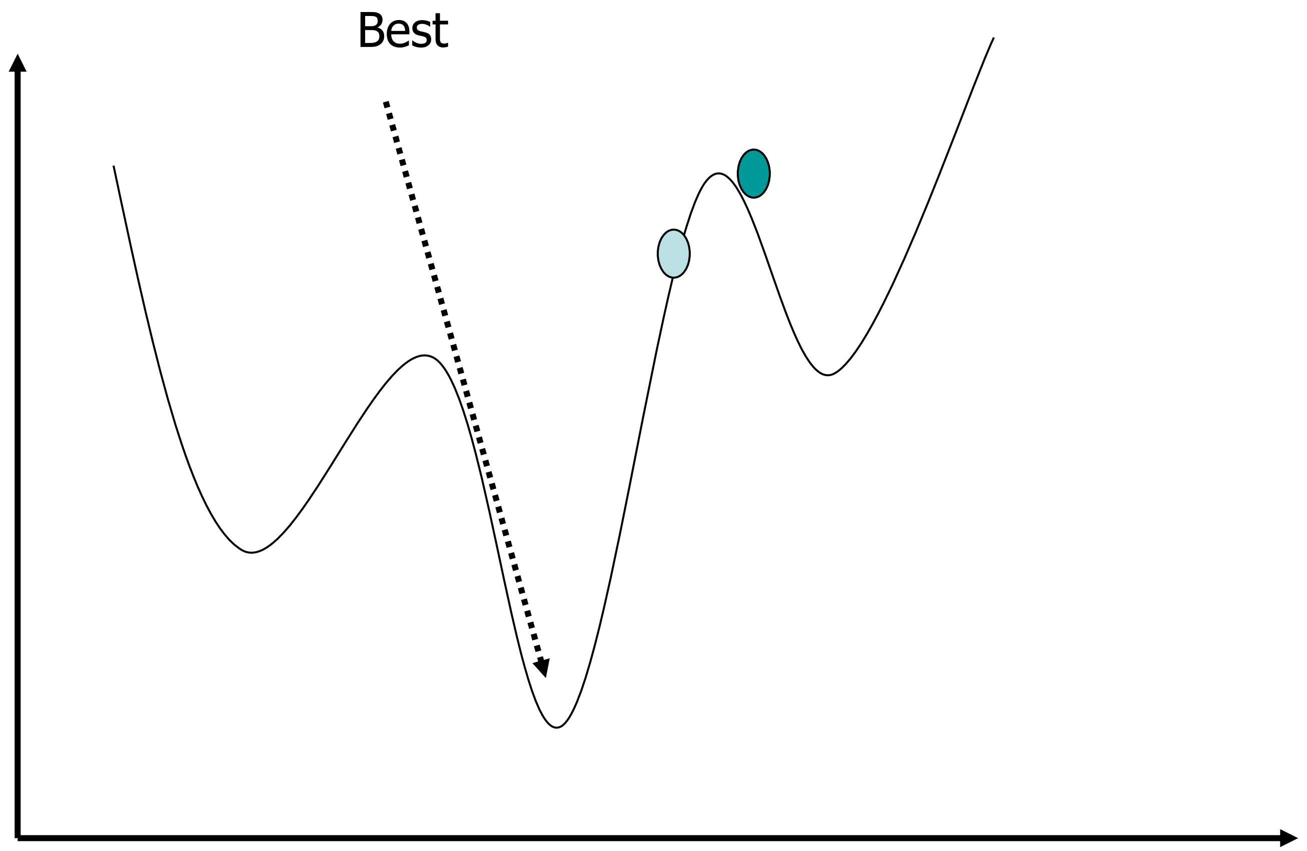
Tabu Search: TS



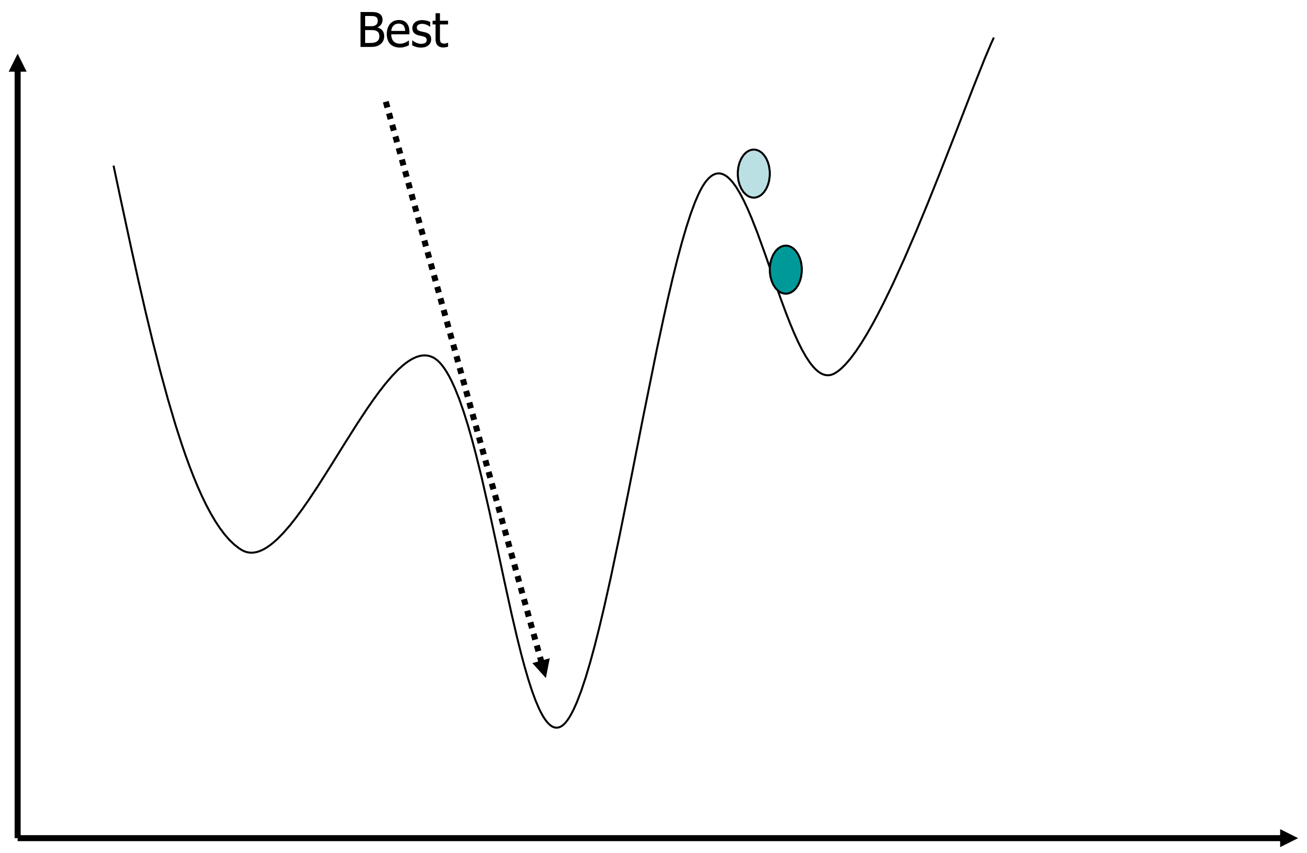
Tabu Search: TS



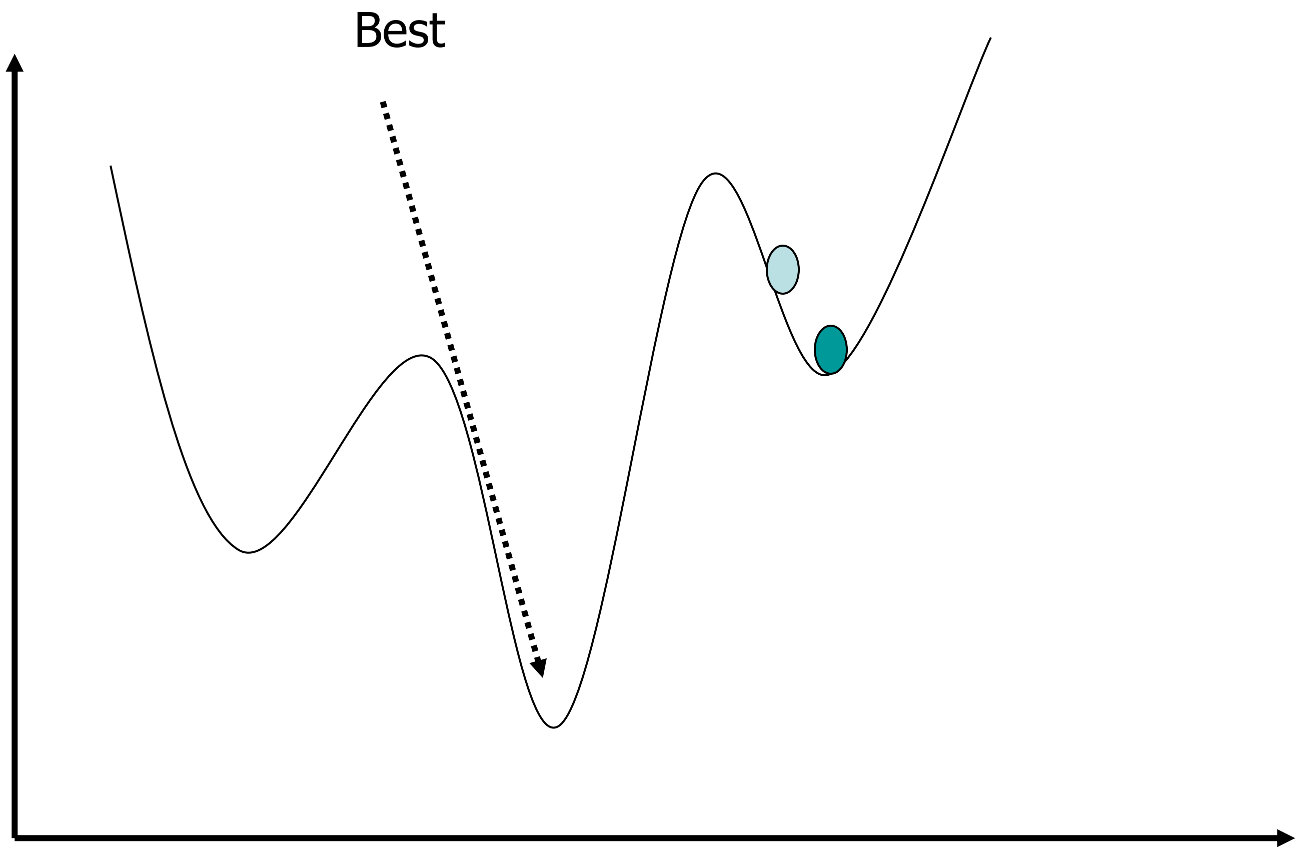
Tabu Search: TS



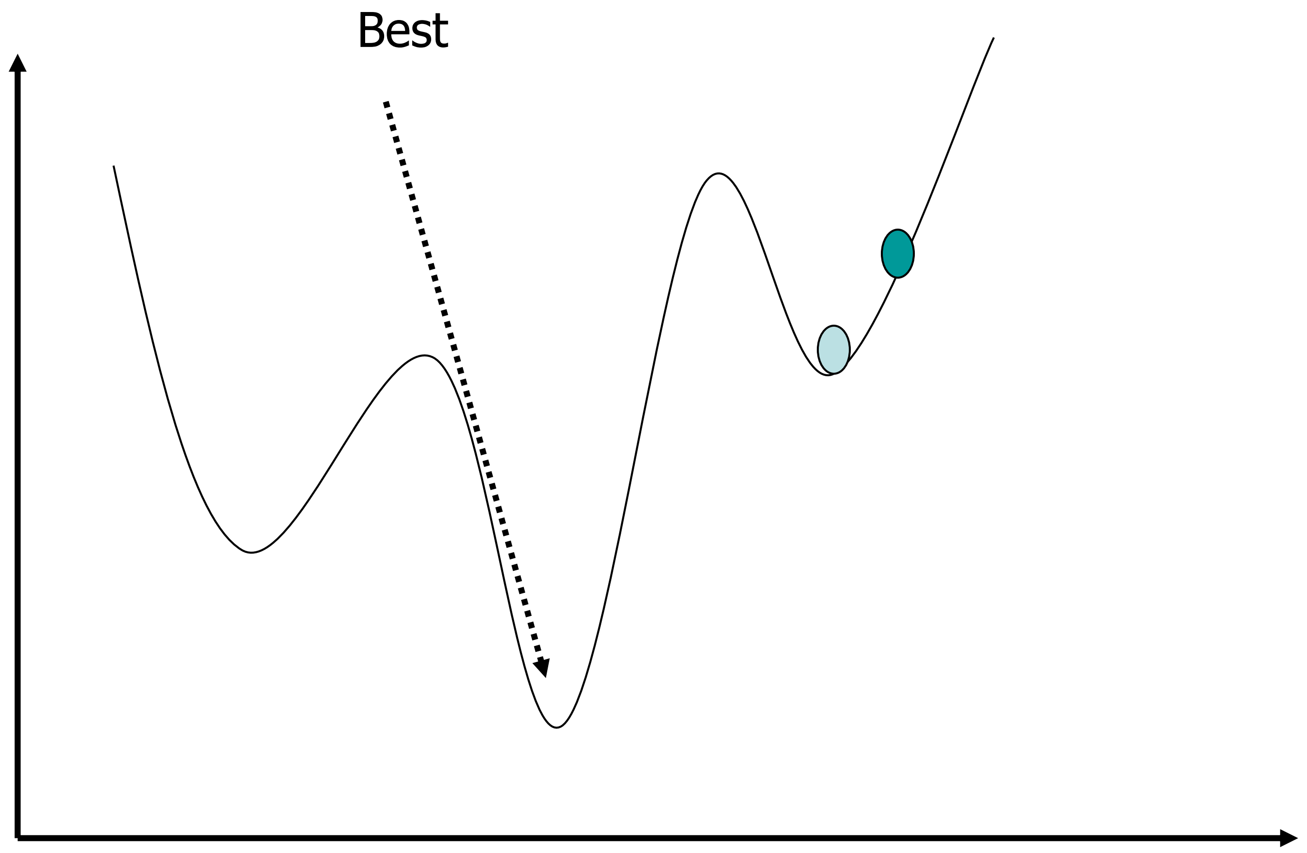
Tabu Search: TS



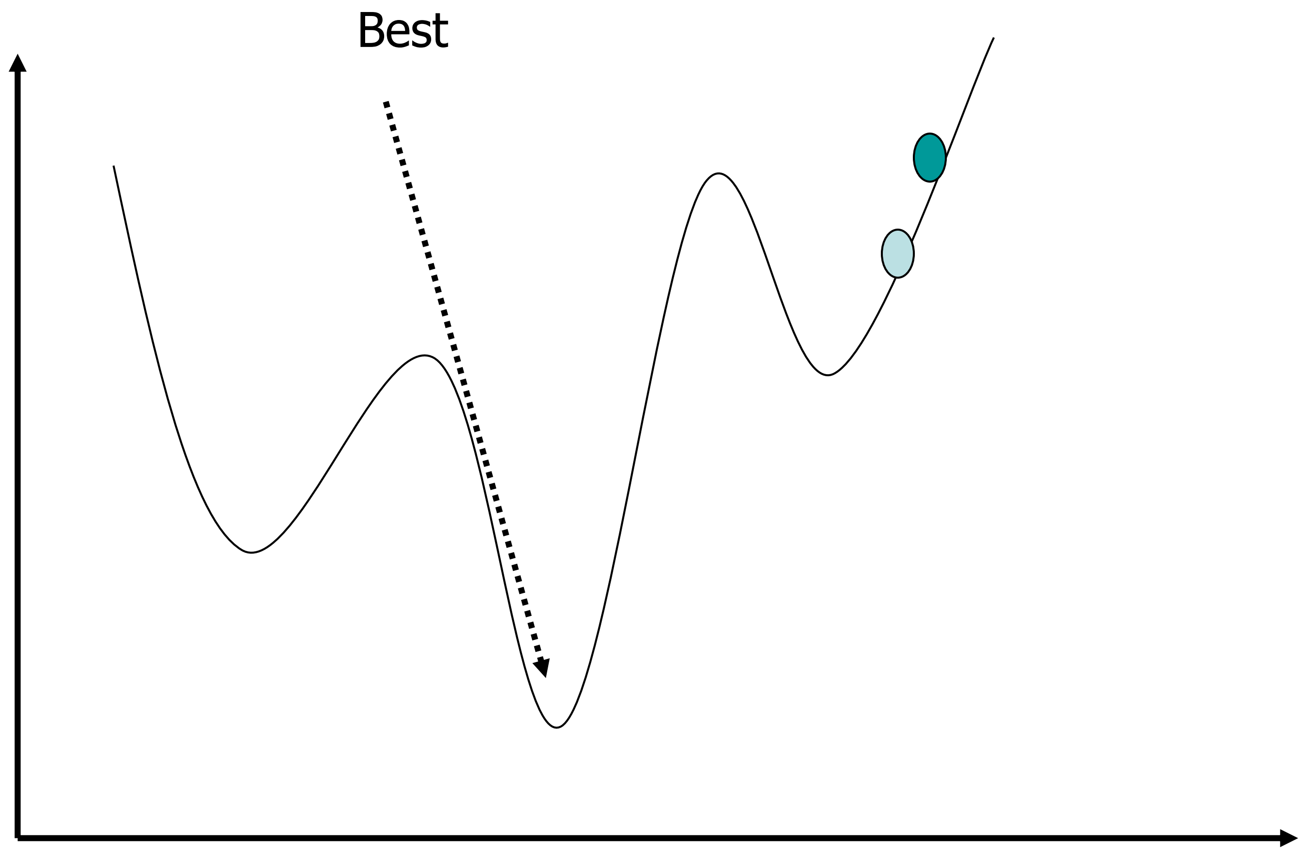
Tabu Search: TS



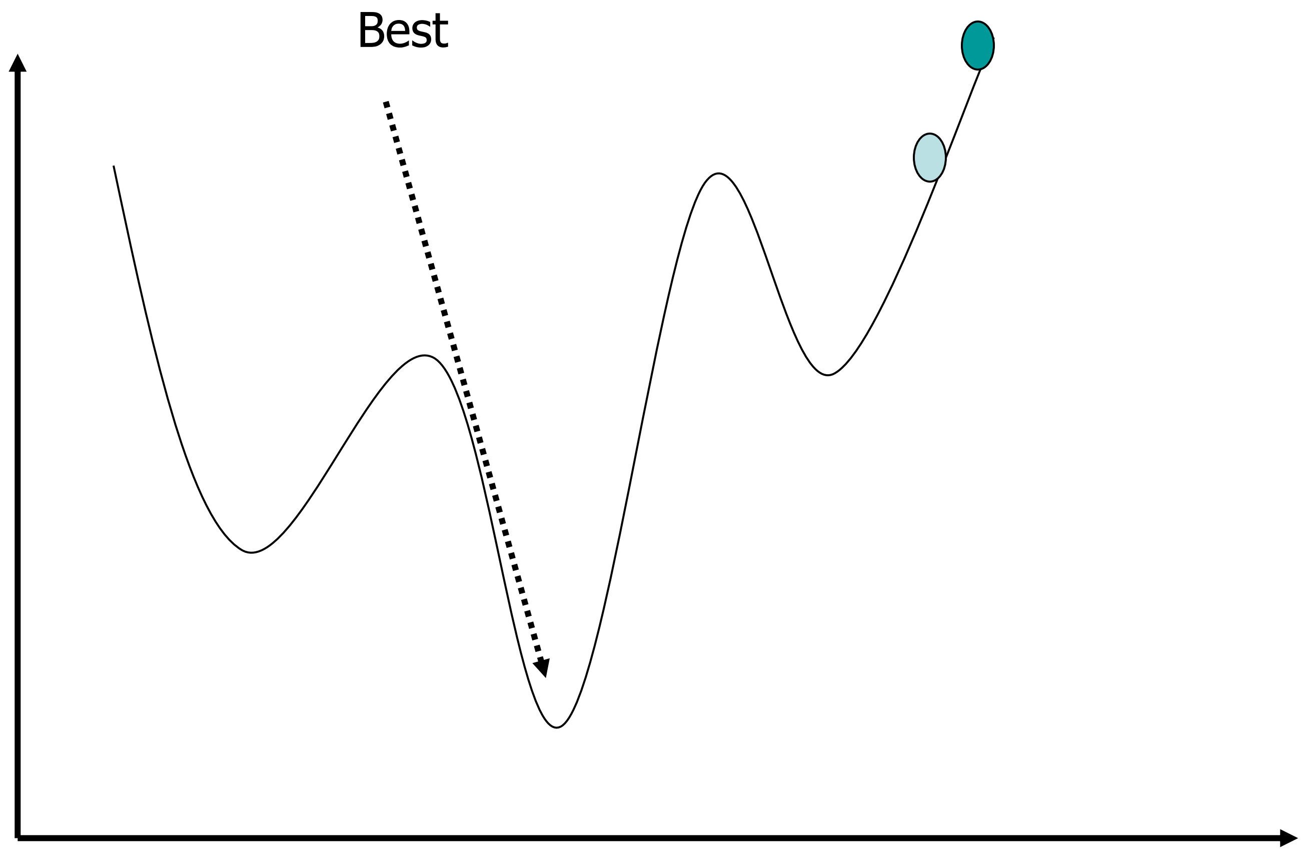
Tabu Search: TS



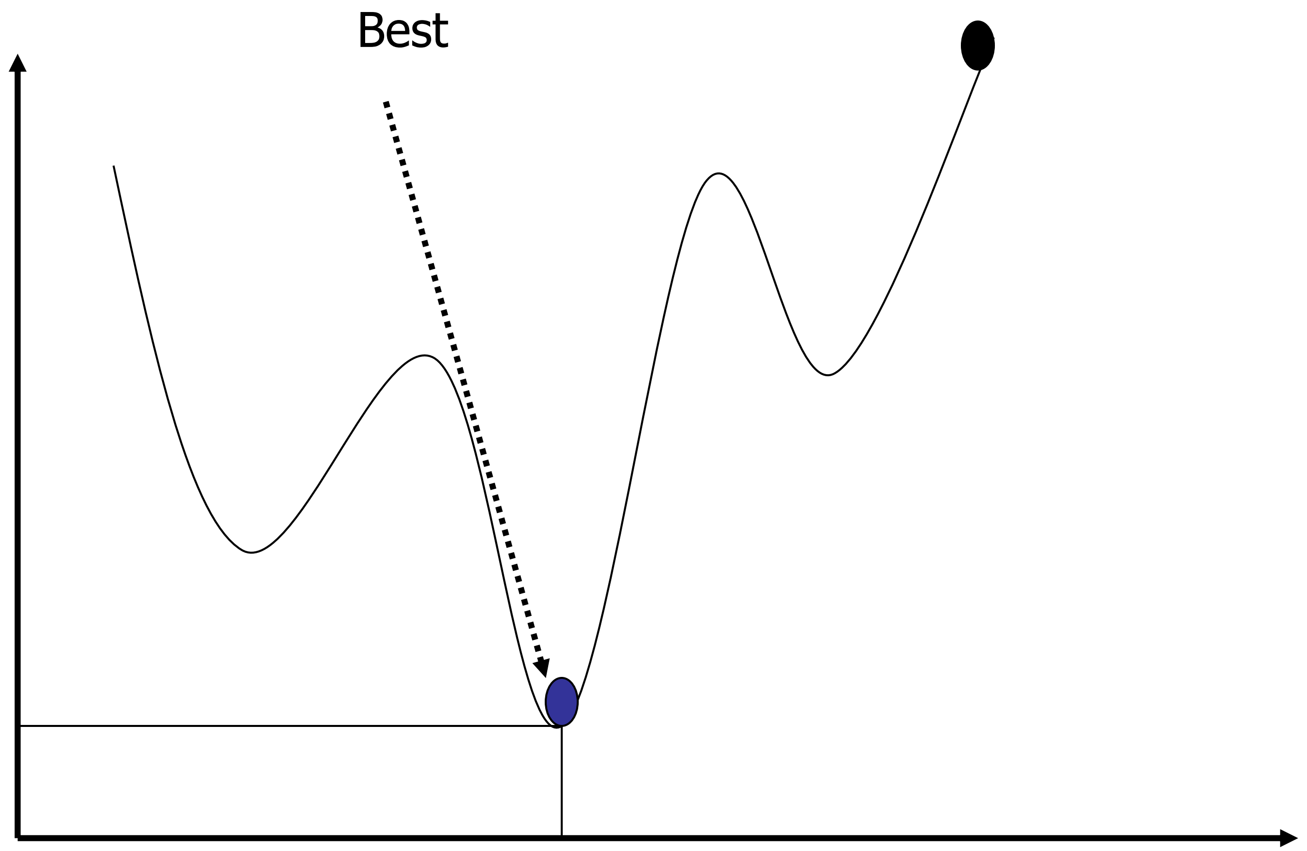
Tabu Search: TS



Tabu Search: TS



Tabu Search: TS



Advantages and Disadvantages of TS

- Advantages and Disadvantages of TS
- Advantages
  - Can escape local optimums by picking non-improving solutions
  - The Tabu List can be used to avoid cycles and reverting to old solutions
  - Can be applied to both discrete and continuous solutions
- Disadvantages
  - Number of iterations can be very high
  - There are a lot of tuneable parameters in this algorithm
  - Key issue with tabu search
    - expensive to maintain all the visited nodes (Short-term memory only keep a small set of recently visited nodes (tabu list)=

Evolutionary computation

- In computer science, **evolutionary computation** is a family of algorithms for global optimization inspired by biological evolution, and the subfield of artificial intelligence and soft computing studying these algorithms.
- Evolutionary computing techniques mostly involve metaheuristic optimization algorithms. Broadly speaking, the field includes:
  - Agent-based modeling
  - Ant colony optimization
  - Artificial immune systems
  - Artificial life (also see digital organism)
  - Cultural algorithms
  - Differential evolution
  - Dual-phase evolution
  - Estimation of distribution algorithms
  - Evolutionary algorithms
  - Evolutionary programming
  - Evolution strategy
  - Gene expression programming
  - Genetic algorithm
  - Genetic programming
  - Grammatical evolution
  - Learnable evolution model
  - Learning classifier systems
  - Memetic algorithms
  - Neuroevolution
  - Particle swarm optimization
  - Self-organization such as self-organizing maps, competitive learning
  - Swarm intelligence

Evolutionary algorithms

- Evolutionary algorithms are based on concepts of biological evolution. A 'population' of possible solutions to the problem is first created with each solution being scored using a 'fitness function' that indicates how good they are. The population evolves over time and (hopefully) identifies better solutions.
- Evolutionary algorithms can be seen as variants of stochastic beam search that are explicitly motivated by the metaphor of natural selection in biology: there is a population of individuals (states), in which the fittest (highest value) individuals produce offspring (successor states) that populate the next generation, a process called recombination. There are endless forms of evolutionary algorithms, varying in the following ways:
  - The size of the population.
  - The representation of each individual.
  - The mixing number, which is the number of parents that come together to form offspring.
  - The selection process for selecting the individuals who will become the parents of the next generation
  - The recombination procedure. One common approach (assuming =2), is to randomly select a crossover point to split each of the parent strings, and recombine the parts to form two children.
  - The mutation rate, which determines how often offspring have random mutations to their representation.
  - The makeup of the next generation. This can be just the newly formed offspring, or it can include a few top-scoring parents from the previous generation (a practice called elitism)

Genetic Algorithms

- Each state is seen as an individual in a population.
- A genetic algorithm applies selection and reproduction operators to an initial population
- The aim is to generate individuals that are most successful, according to a given fitness function.
- Select parents based on fitness, and "reproduce" to get the next generation (using "crossover" and mutations)
- Replace the old generation with the new generation.

```
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
     parents ← SELECTION(population, FITNESS-FN)
     population ← REPRODUCTION(parents)
  until some individual is fit enough
  return the best individual in population, according to FITNESS-FN
```

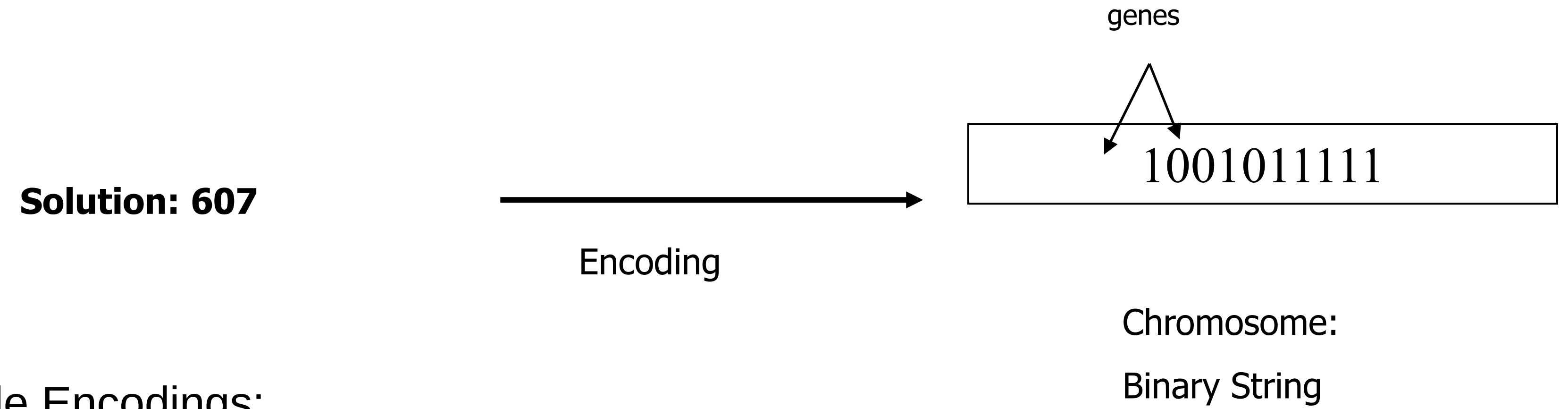
- Variation operators used in Reproduction create the necessary diversity, facilitating novelty
- Selection reduces diversity but pushes quality by increasing fitness

Genetic Algorithms

- Before we can apply Genetic Algorithm to a problem, we need to answer:
  - How to represent an individual?
  - What is the fitness function?
  - How to select individuals?
  - How to reproduce individuals?

Representation of states (solutions)

- Each state or individual is represented as a string over a finite alphabet. It is also called **chromosome** which Contains **genes.** Each character in the string is a gene.



- Possible Encodings:
  - Character strings 0101 · · · 1100
  - Sequences of real numbers (43.2 -33.1 · · · 0.0 89.2)
  - Tuples of elements (E11 E3 E7 · · · E1 E15)
  - Lists of rules (R1 R2 R3 · · · R22 R23)
- Choosing the right encoding of state configurations to strings is crucial.

8-queens puzzle encoding

- 8-digit strings represents 8-queens states
  - Figure 4.6(a) shows a population of four 8-digit strings, each representing a state of the 8-queens puzzle: the c-th digit represents the row number of the queen in column c. In (b), each state is rated by the fitness function. Higher fitness values are better, so for the 8-queens problem we use the number of nonattacking pairs of queens, which has a value of 8*7/2=28.

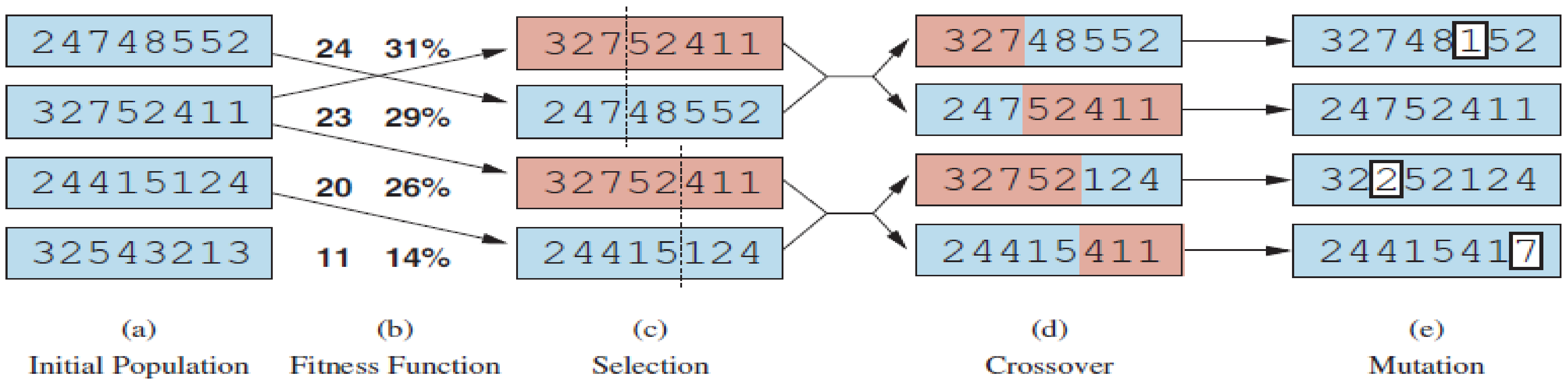


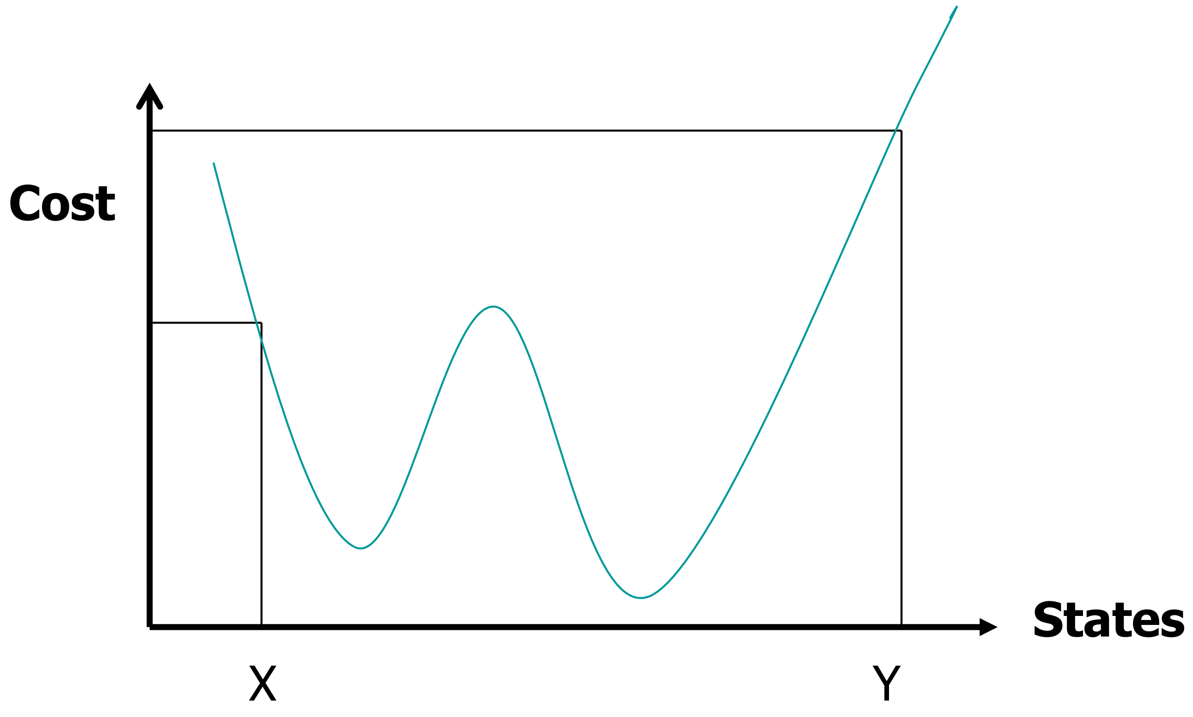
Figure 4.6 A genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by a fitness function in (b) resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

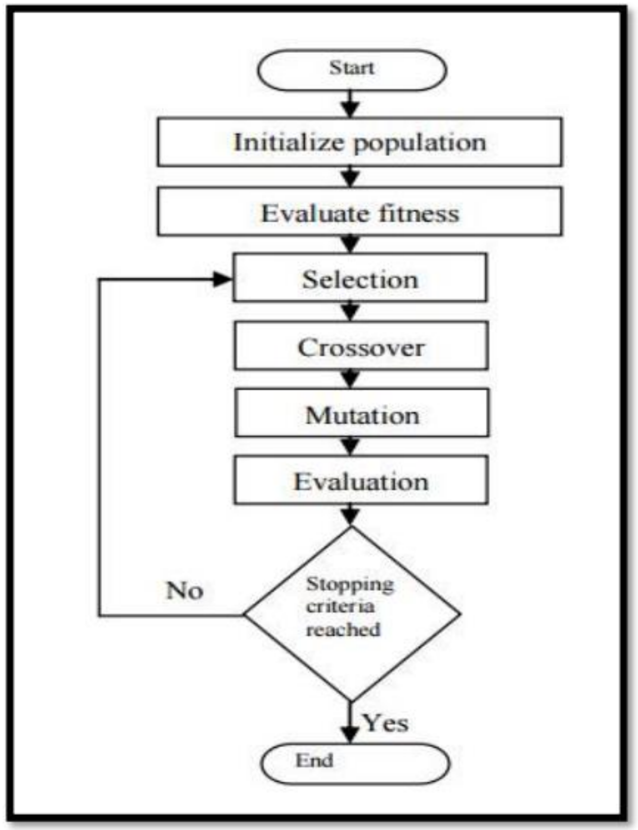
Fitness Function

- Each state is rated by the evaluation function called fitness function. Fitness function should return higher values for better states:

Fitness(X) should be greater than Fitness(Y) !!

[Fitness(x) = 1/Cost(x)]



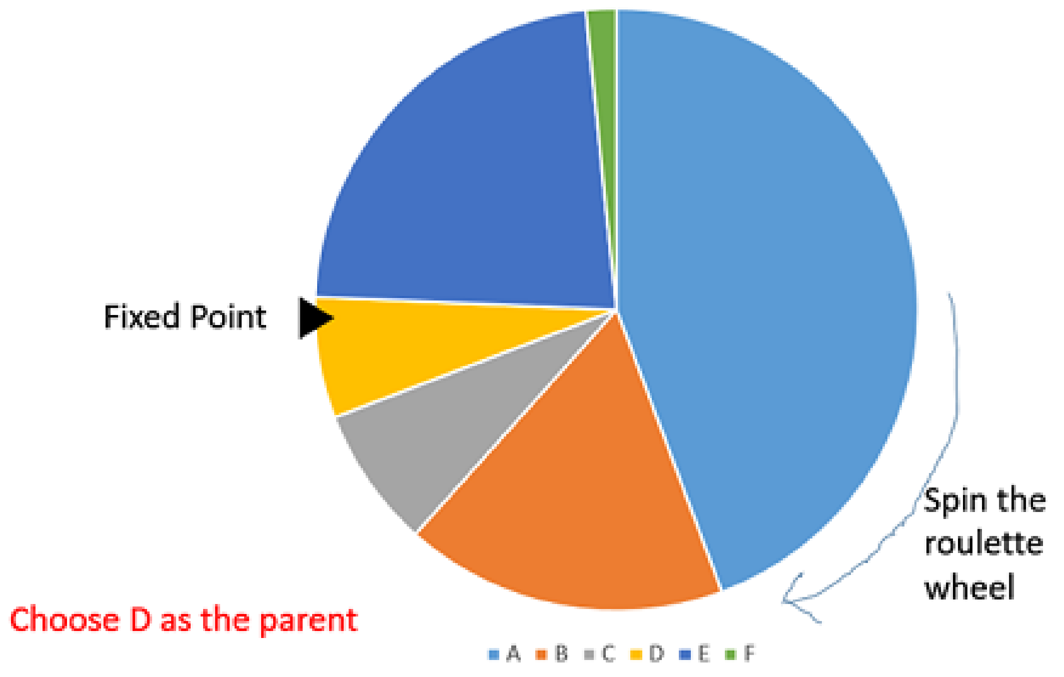


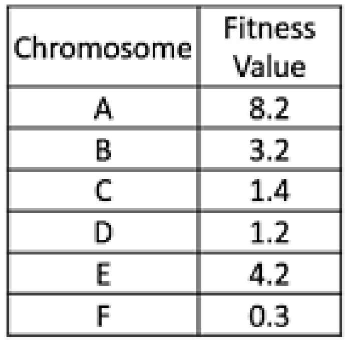
Selection

- **Selection** is the stage of a genetic algorithm in which individual genomes are chosen from a population for later breeding (using the crossover operator).
- Methods of Selection (Genetic Algorithm)
  - 1.1 Roulette Wheel Selection
  - 1.2 Rank Selection
  - 1.3 Steady State Selection
  - 1.4 Tournament Selection
  - 1.5 Elitism Selection
  - 1.6 Boltzmann Selection

Roulette Wheel Selection

- In the roulette wheel selection, the probability of choosing an individual for breeding of the next generation is proportional to its fitness, the better the fitness is, the higher chance for that individual to be chosen. Choosing individuals can be depicted as spinning a roulette that has as many pockets as there are individuals in the current generation, with sizes depending on their probability.



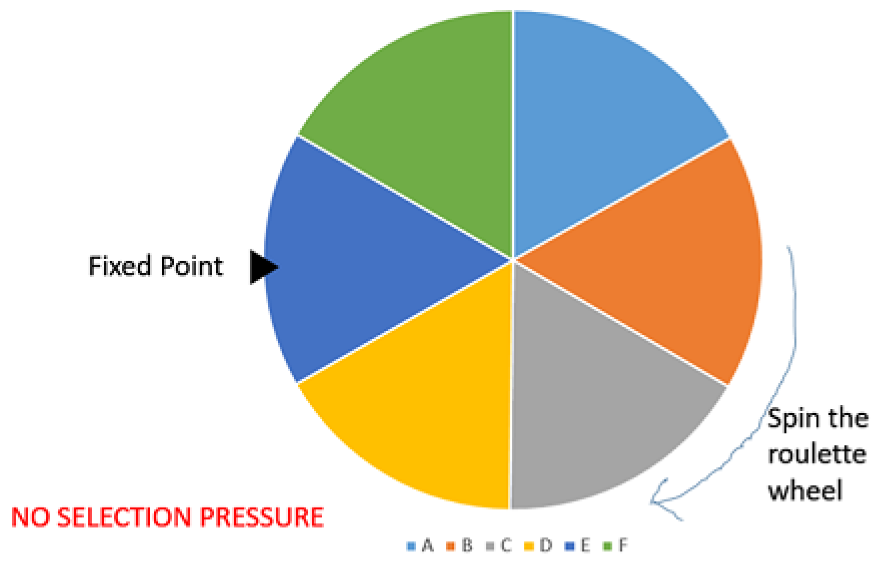


Chromosome	Fitness Value
A	8.2
B	3.2
C	1.4
D	1.2
E	4.2
F	0.3

https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_parent_selection.htm

Rank Selection

- Rank Selection also works with negative fitness values and is mostly used when the individuals in the population have very close fitness values (this happens usually at the end of the run). This leads to each individual having an almost equal share of the pie (like in case of fitness proportionate selection) and hence each individual no matter how fit relative to each other has an approximately same probability of getting selected as a parent. This in turn leads to a loss in the selection pressure towards fitter individuals, causing the GA to make poor parent selections in such situations.



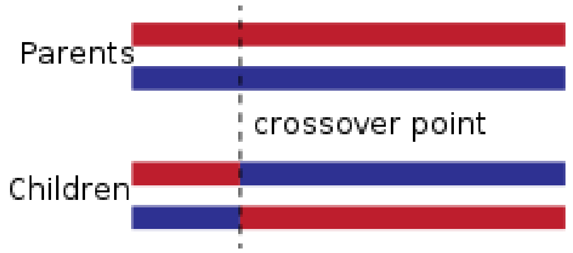
Chromosome	Fitness Value
A	8.1
B	8.0
C	8.05
D	7.95
E	8.02
F	7.99

https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_parent_selection.htm

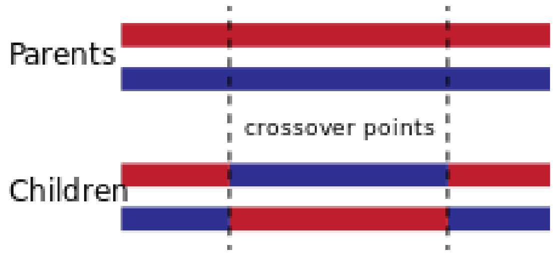
Cross-Over and Mutation

- How to reproduce individuals?
  - Crossover, also called recombination, is a genetic operator used to combine the genetic information of two parents to generate new offspring.
  - The crossover of two parent strings produces offspring/children (new solutions) by swapping parts or genes of the chromosomes.
  - Crossover has a higher probability, typically 0.8-0.95.

One-point crossover:
A point on both parents' chromosomes is picked randomly, and designated a 'crossover point'. Bits to the right of that point are swapped between the two parent chromosomes. This results in two offspring, each carrying some genetic information from both parents.

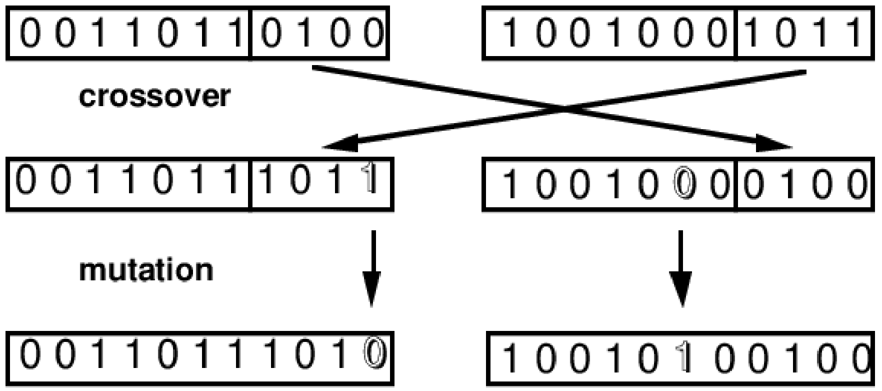


Two-point and k-point crossover:
In two-point crossover, two crossover points are picked randomly from the parent chromosomes. The bits in between the two points are swapped between the parent organisms.



Cross-Over and Mutation

- Mutation is a genetic operator used to maintain genetic diversity from one generation of a population of genetic algorithm chromosomes to the next. It is analogous to biological mutation.
- Mutation is carried out by flipping some digits of a string, which generates new solutions. This mutation probability is typically low, from 0.001 to 0.05.
- New solutions generated in each generation will be evaluated by their fitness that is linked to the objective function of the optimization problem.
- The new solutions are selected according to their fitness—selection of the fittest. Sometimes, in order to make sure that the best solutions remain in the population, the best solutions are passed on to the next generation without much change. This is called elitism.



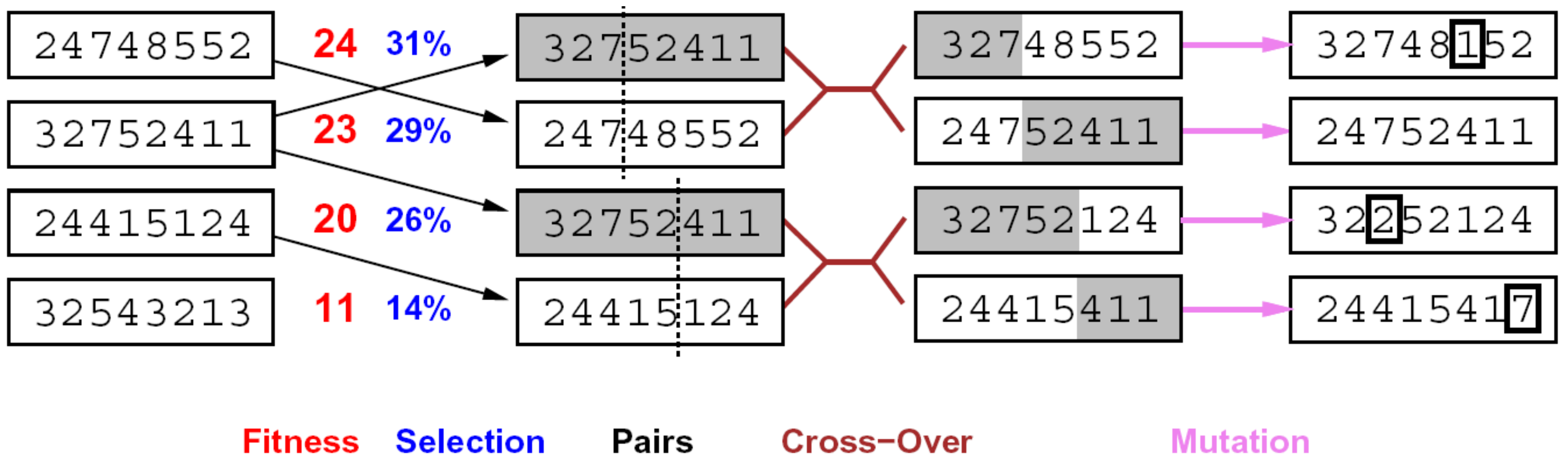
Order changing - two numbers are selected and exchanged : (1 **2** 3 4 5 6 **8** 9 7) => (1 **8** 3 4 5 6 **2** 9 7)

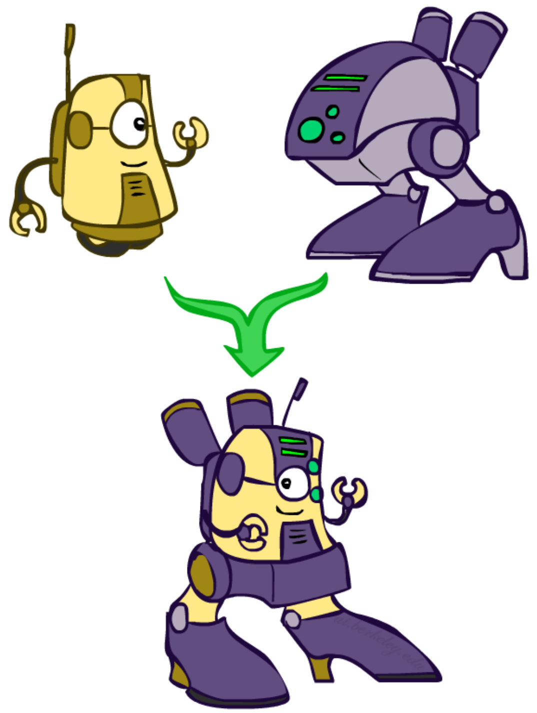
Adding a small number (for real value encoding) - to selected values is added (or subtracted) a small number:
(1.29 5.68 **2.86** **4.11** 5.55) => (1.29 5.68 **2.73** **4.22** 5.55)

Bit inversion - selected bits are inverted : 1**1**001001 => 1**0**001001

https://www.obitko.com/tutorials/genetic-algorithms/crossover-mutation.php

8-queens puzzle : selection





- Genetic algorithms use a natural selection metaphor
  - Resample *K* individuals at each step (selection) weighted by fitness function
  - Combine by pairwise crossover operators, plus mutation to give variety
- Fitness function: number of non-attacking pairs of queens (min = 0, max = 8 × 7/2 = 28)
- The fitness values of the four states in (b) are 24, 23, 20, and 11.
  - 24/(24+23+20+11) = 31%
  - 23/(24+23+20+11) = 29% etc.
- Higher fitness values are better.

8-queens puzzle : crossover and mutation

- The fitness values of the four states in (b) are 24, 23, 20, and 11. The fitness scores are then normalized to probabilities, and the resulting values are shown next to the fitness values in (b). In (c), two pairs of parents are selected, in accordance with the probabilities in (b). Notice that one individual is selected twice and one not at all. For each selected pair, a crossover point (dotted line) is chosen randomly. In (d), we cross over the parent strings at the crossover points, yielding new offspring.
- Finally, in (e), each location in each string is subject to random mutation with a small independent probability. One digit was mutated in the first, third, and fourth offspring. In the 8-queens problem, this corresponds to choosing a queen at random and moving it to a random square in its column.

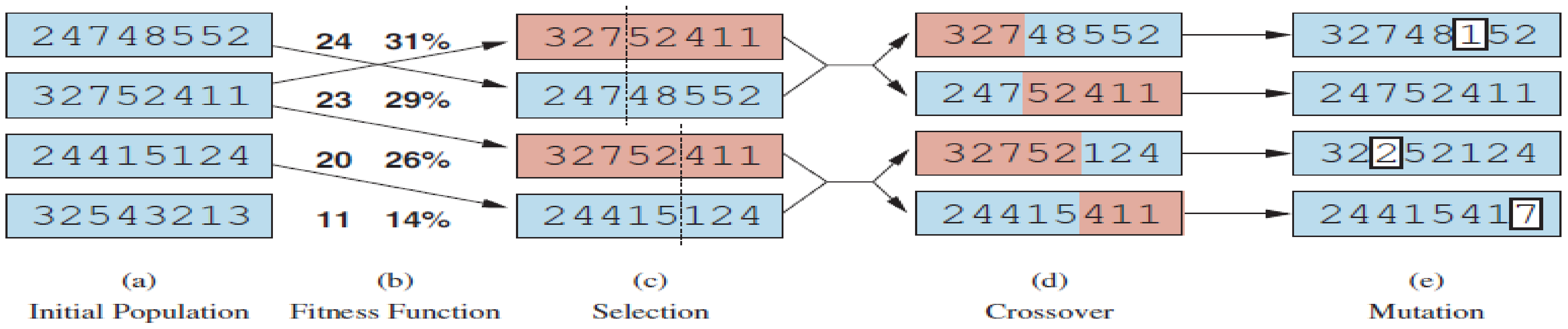
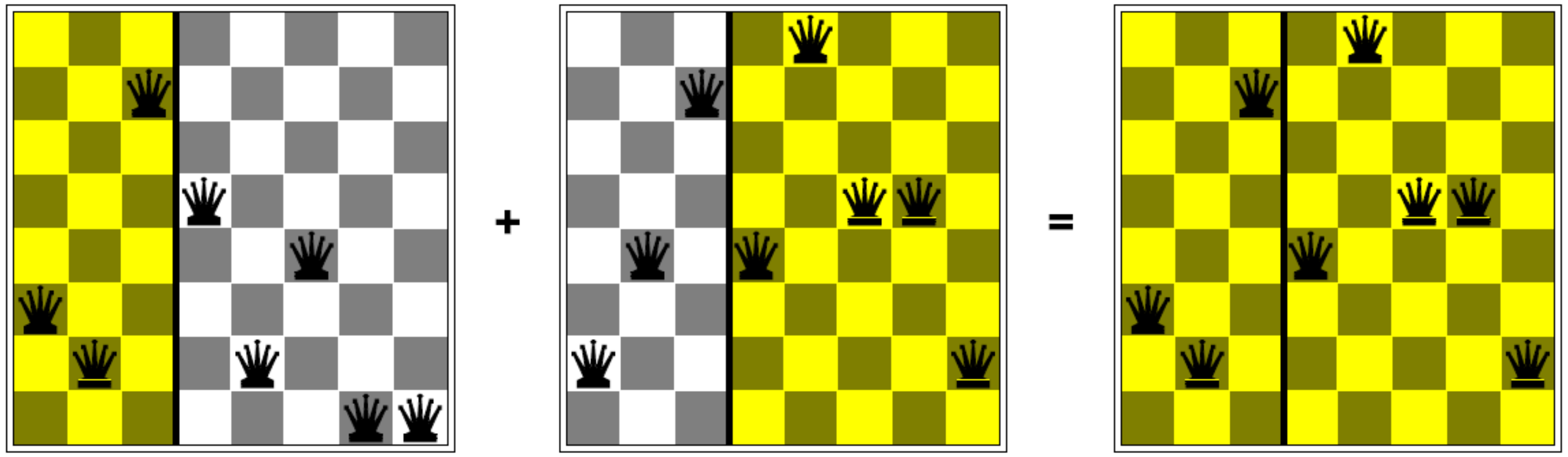


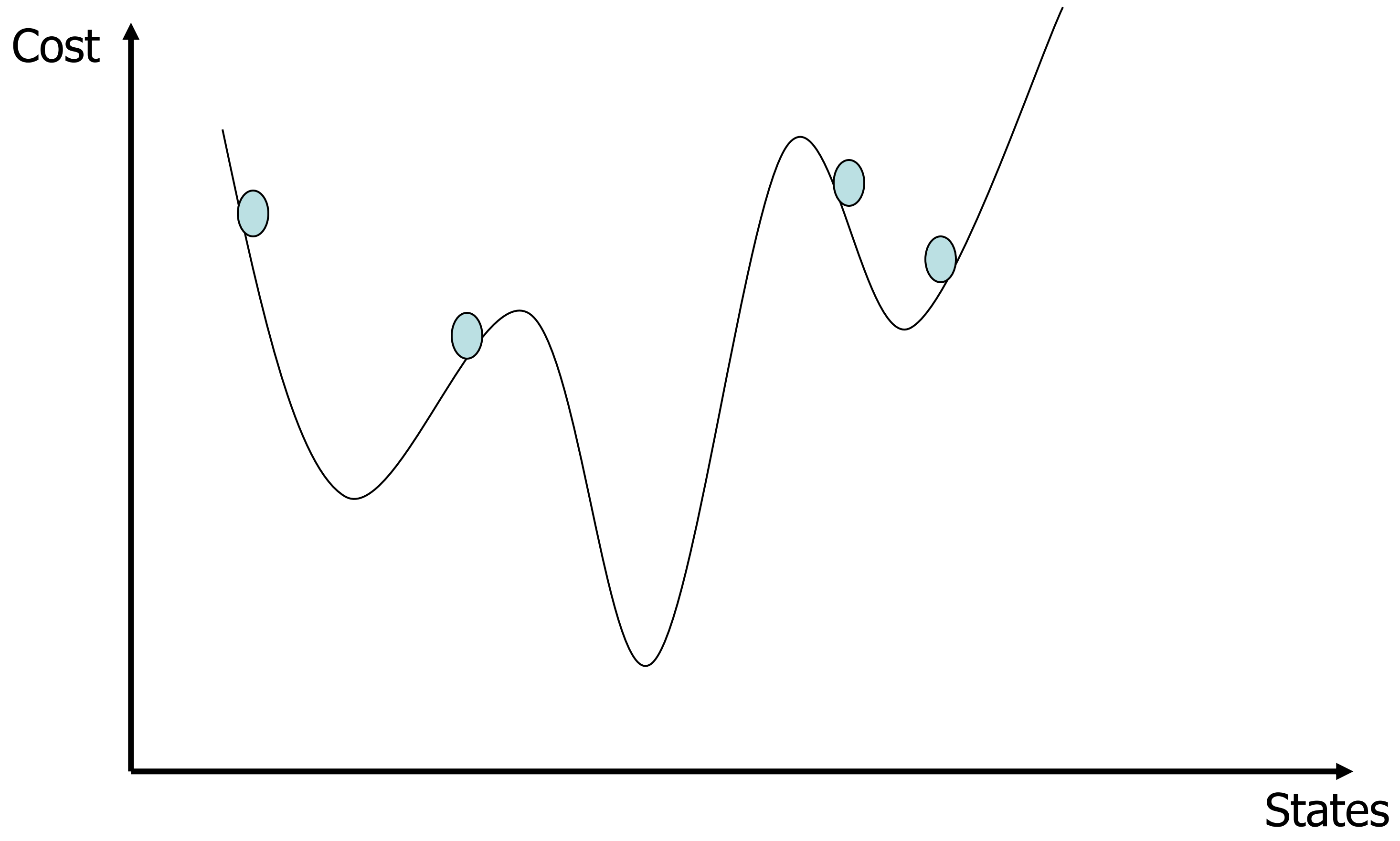
Figure 4.6 A genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by a fitness function in (b) resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

Example: N-Queens

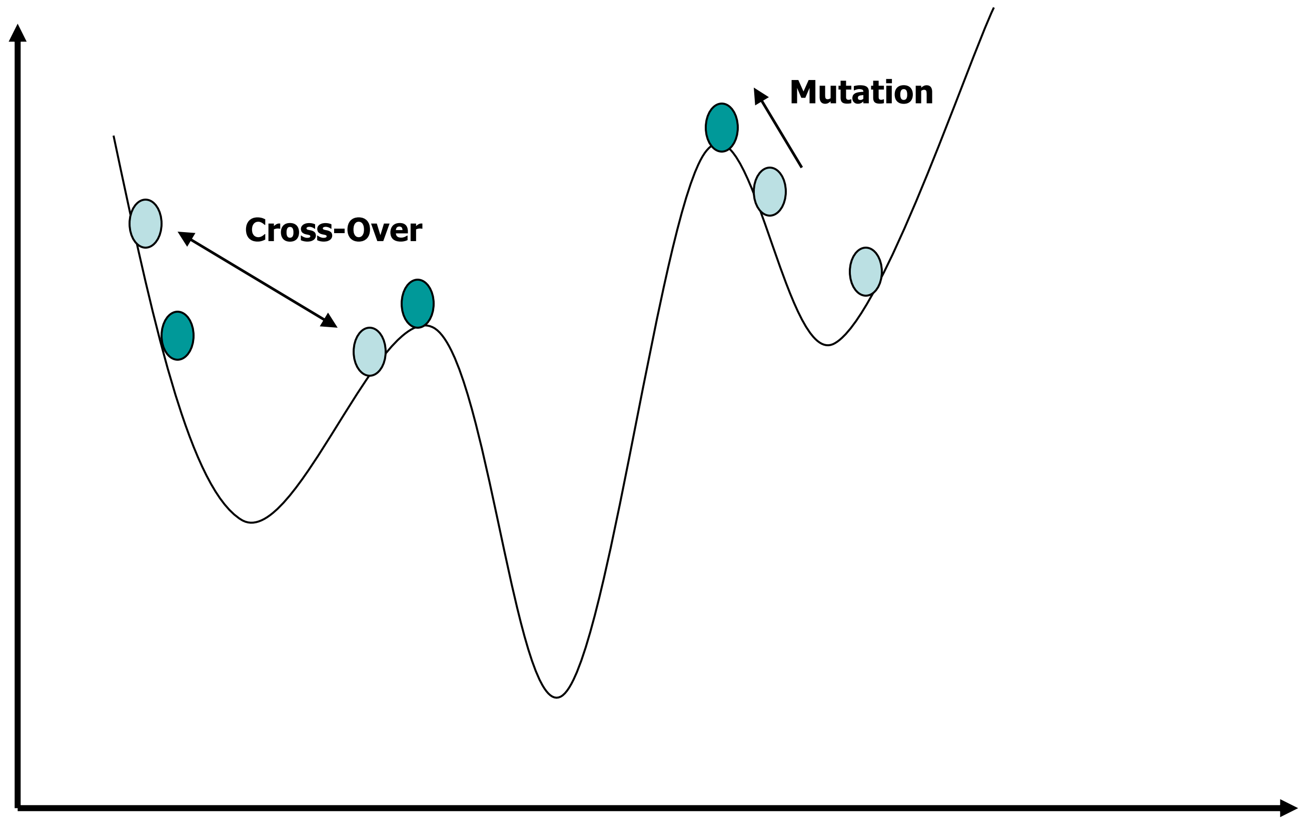


- Crossover helps only if substrings are meaningful components that can be reassembled into a new meaningful configuration.

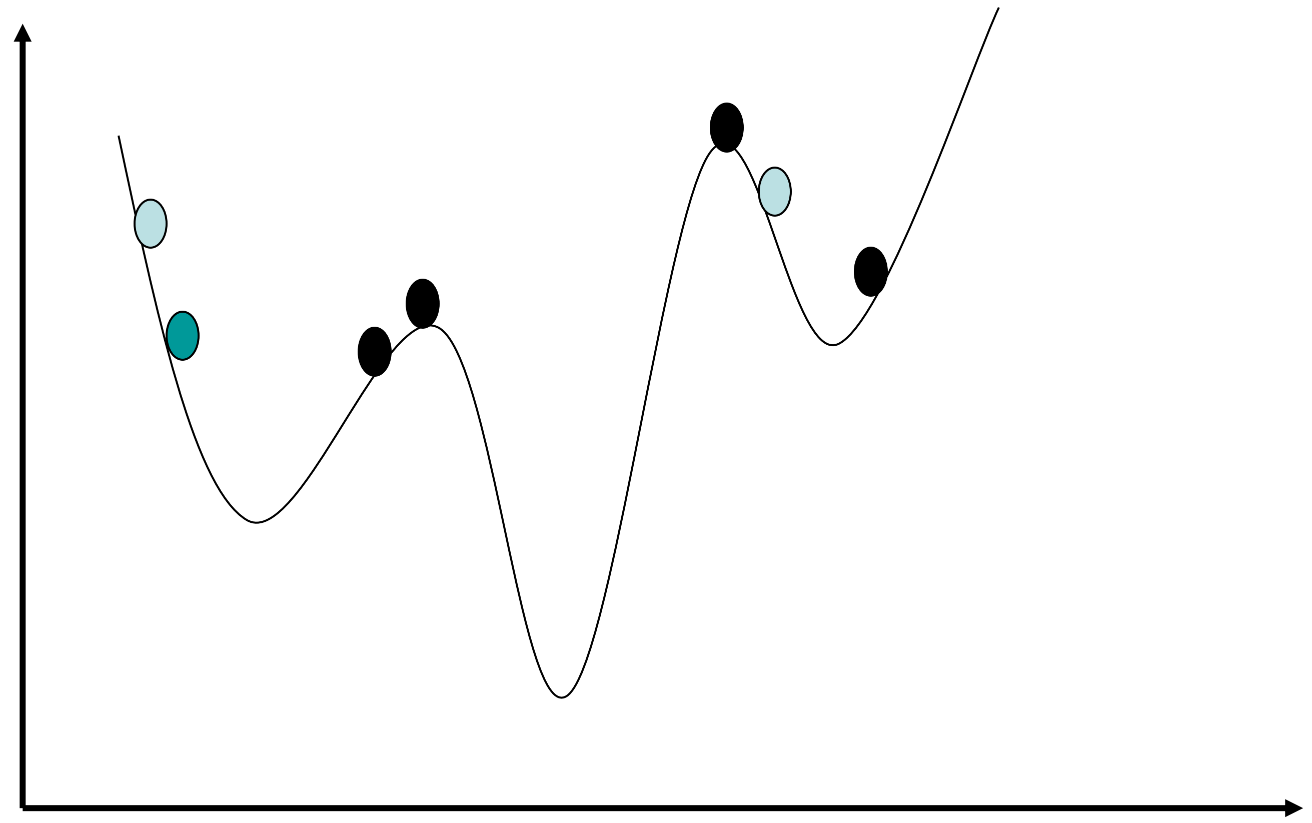
Genetic Algorithms



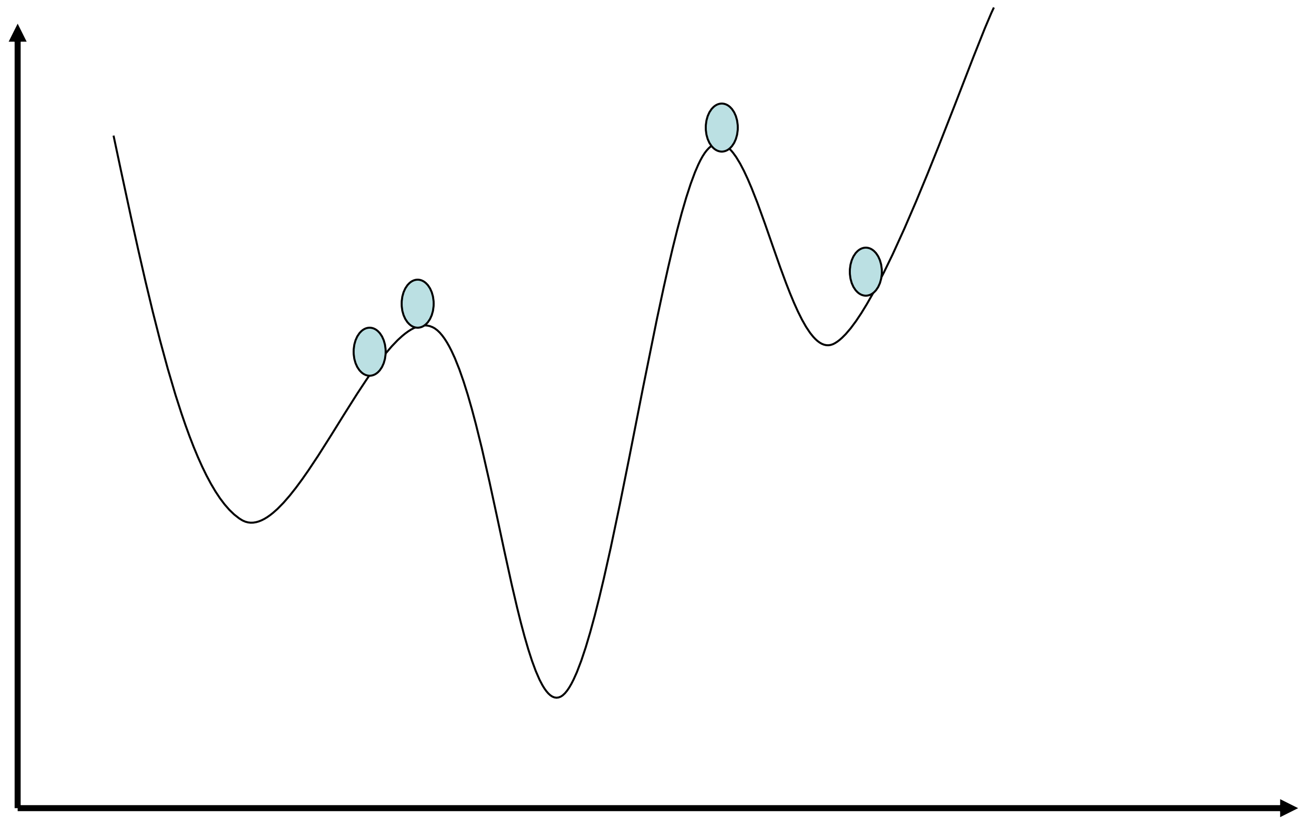
Genetic Algorithms



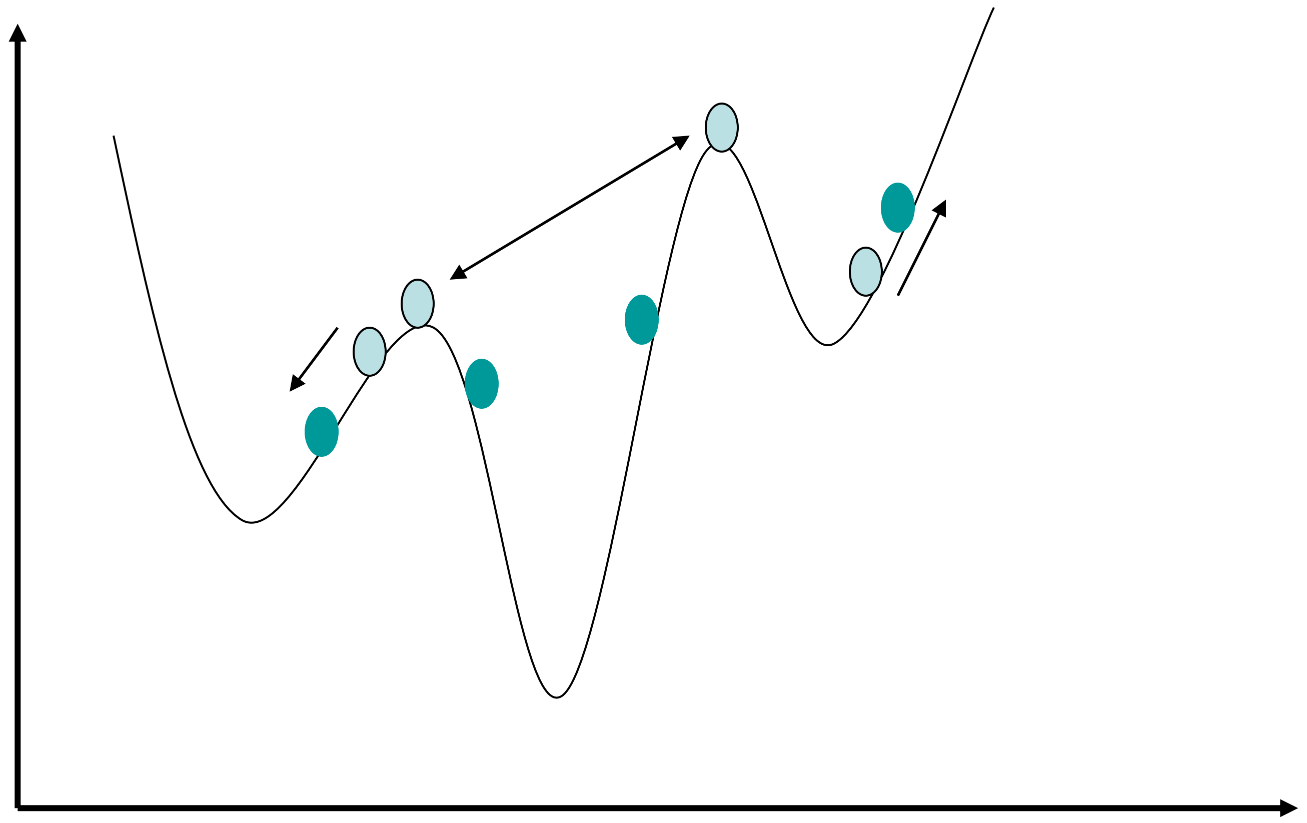
Genetic Algorithms



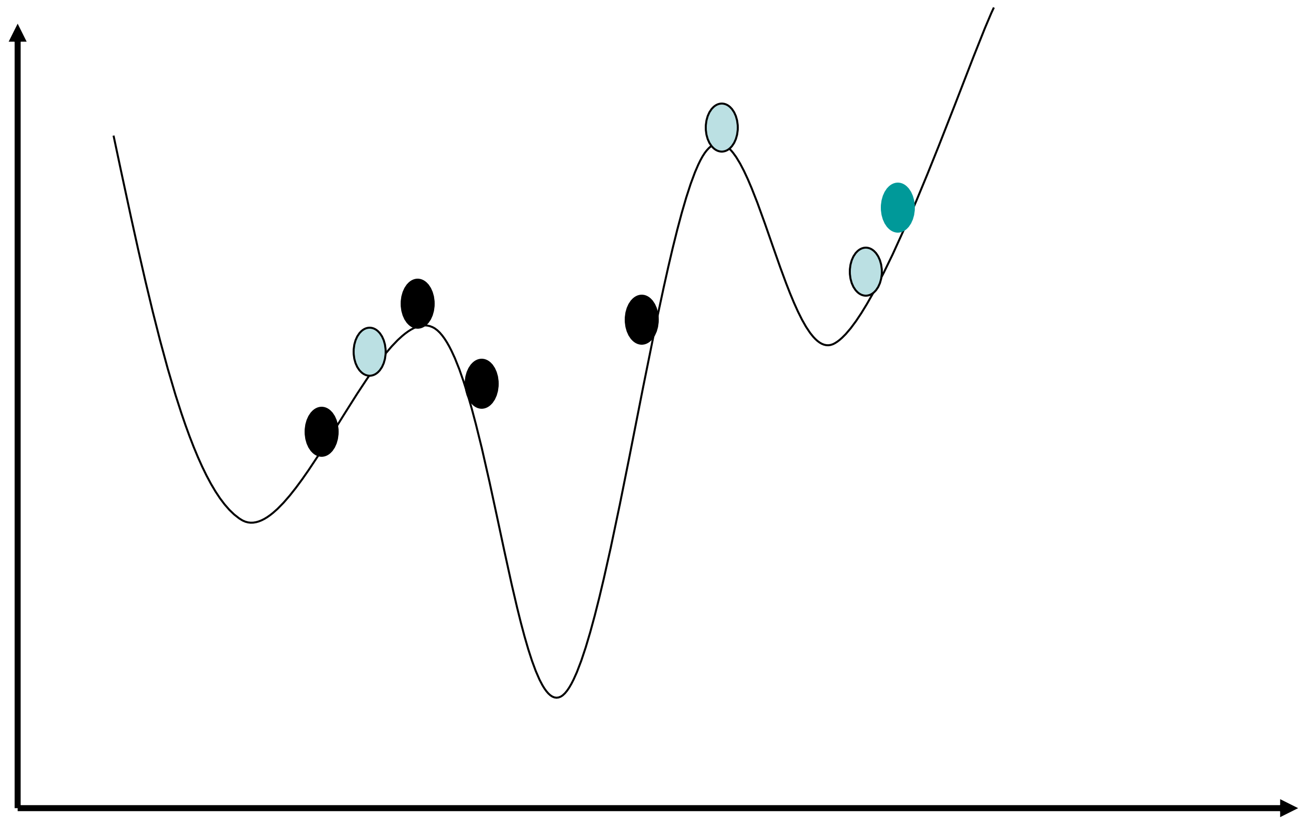
Genetic Algorithms



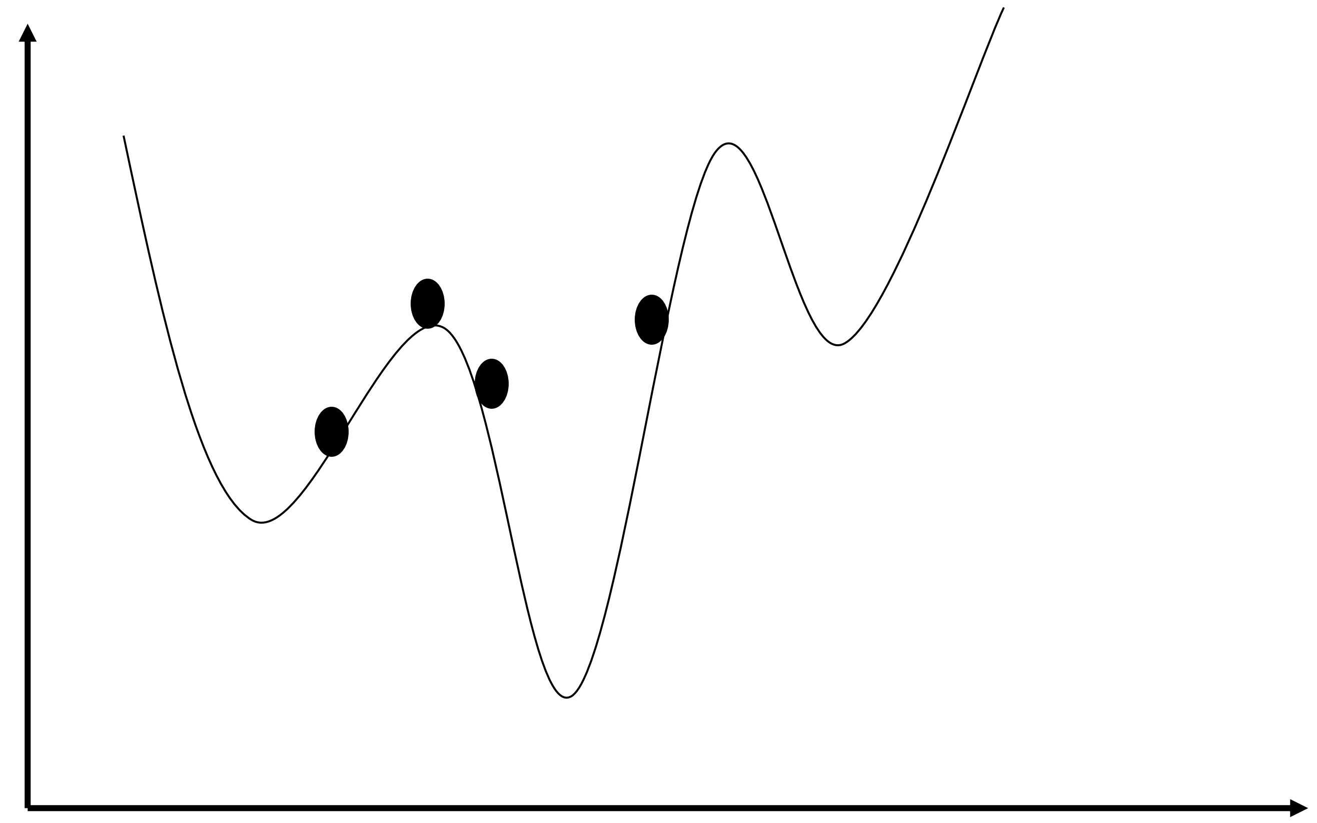
Genetic Algorithms



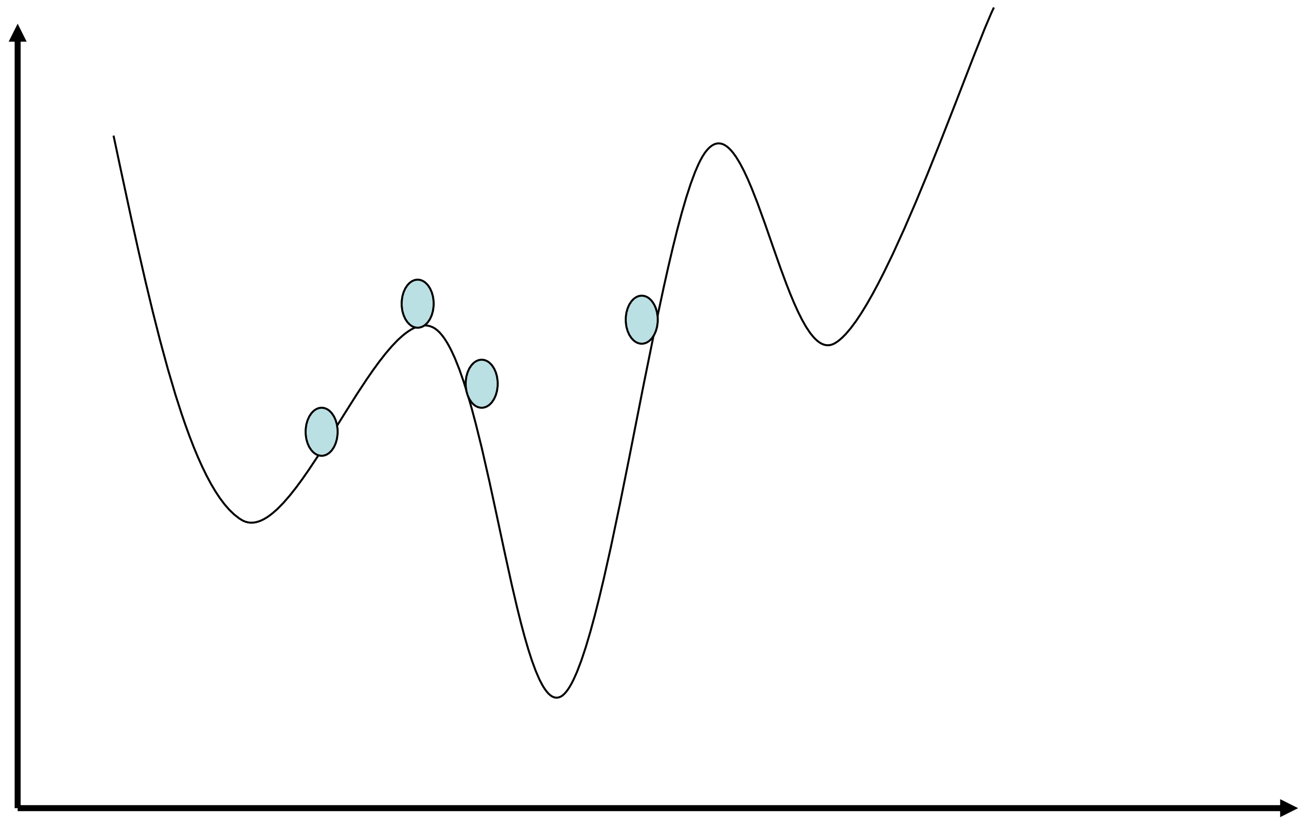
Genetic Algorithms



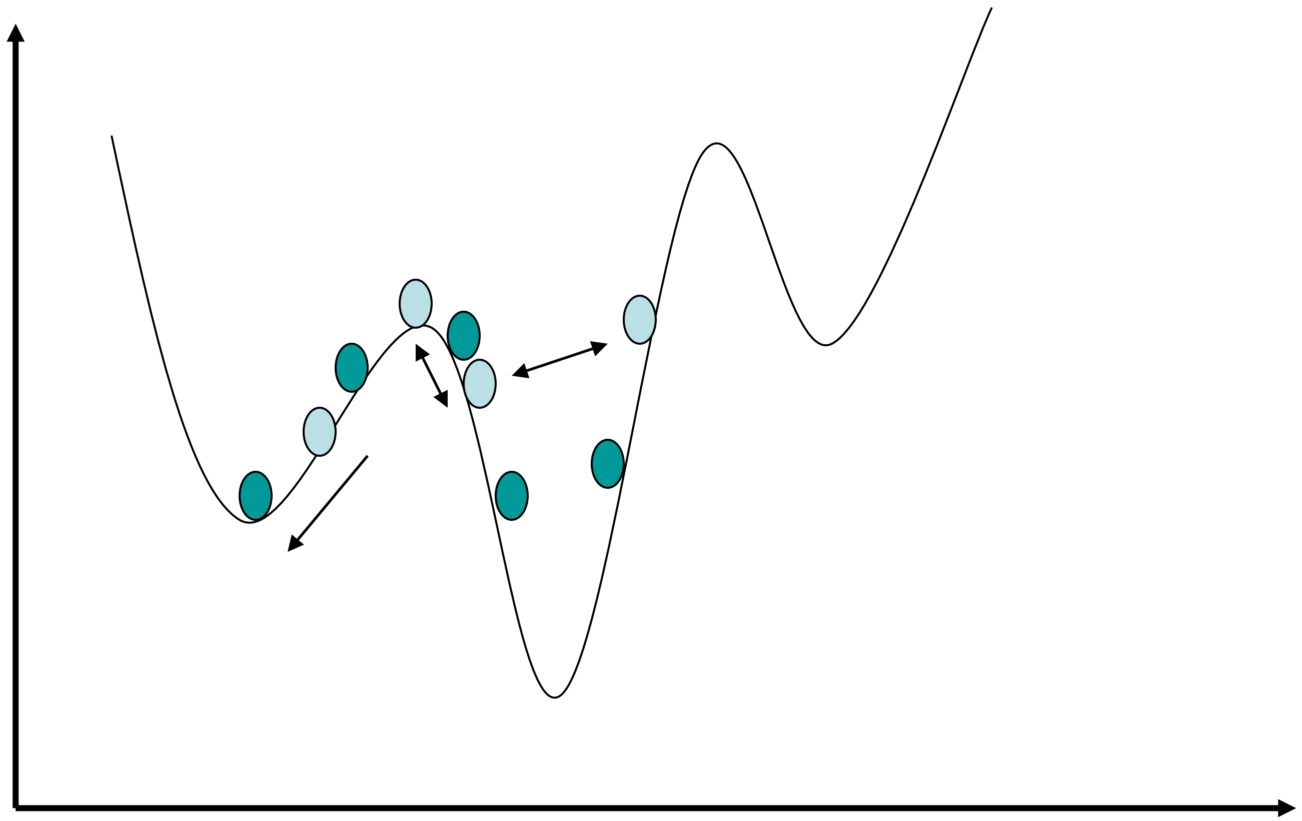
Genetic Algorithms



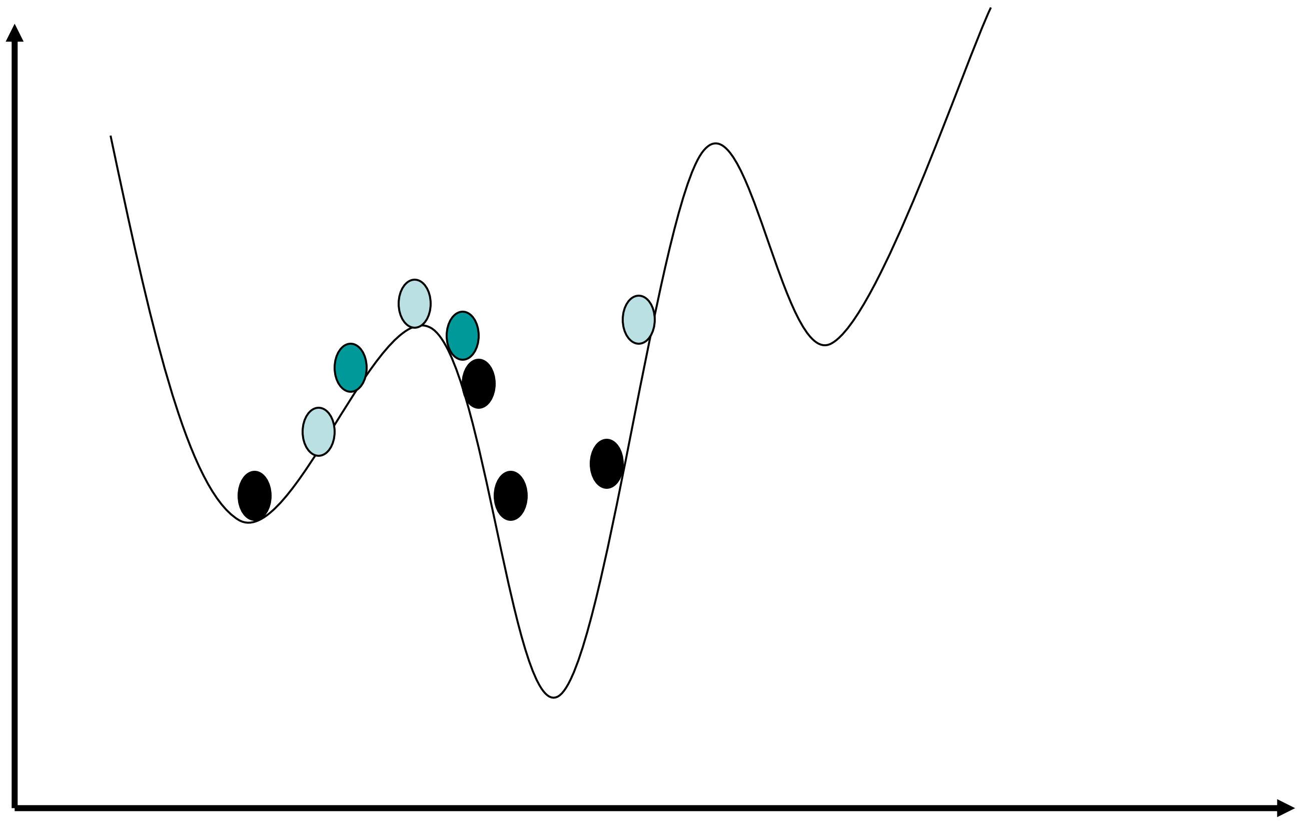
Genetic Algorithms



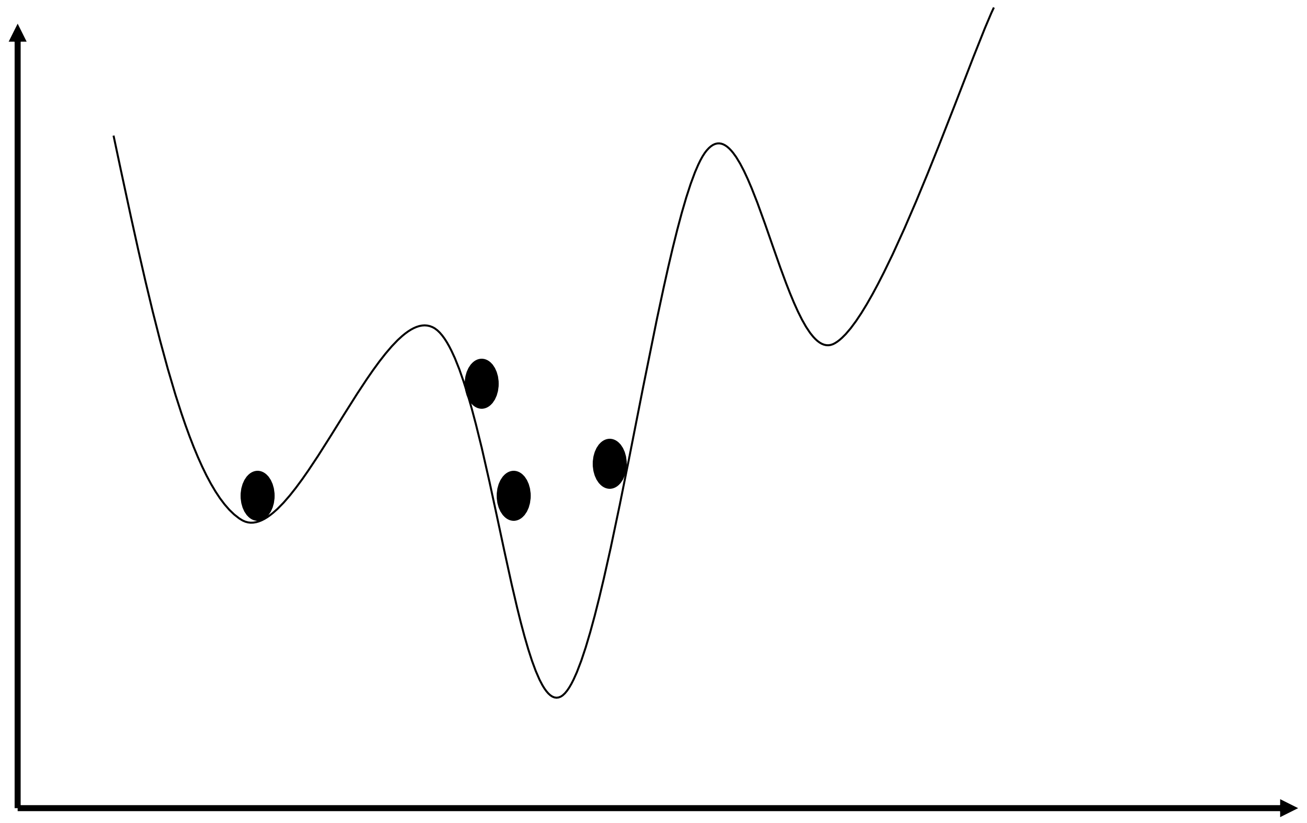
Genetic Algorithms



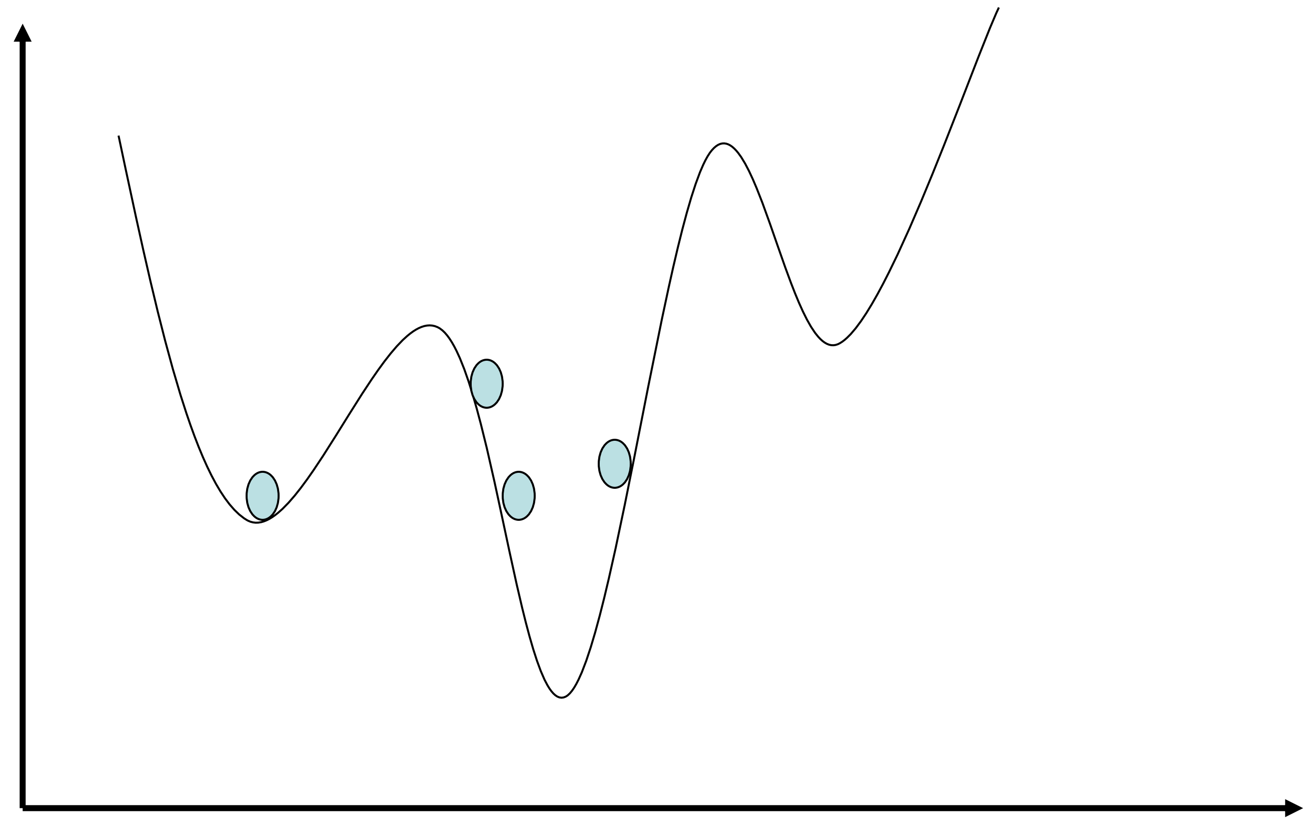
Genetic Algorithms



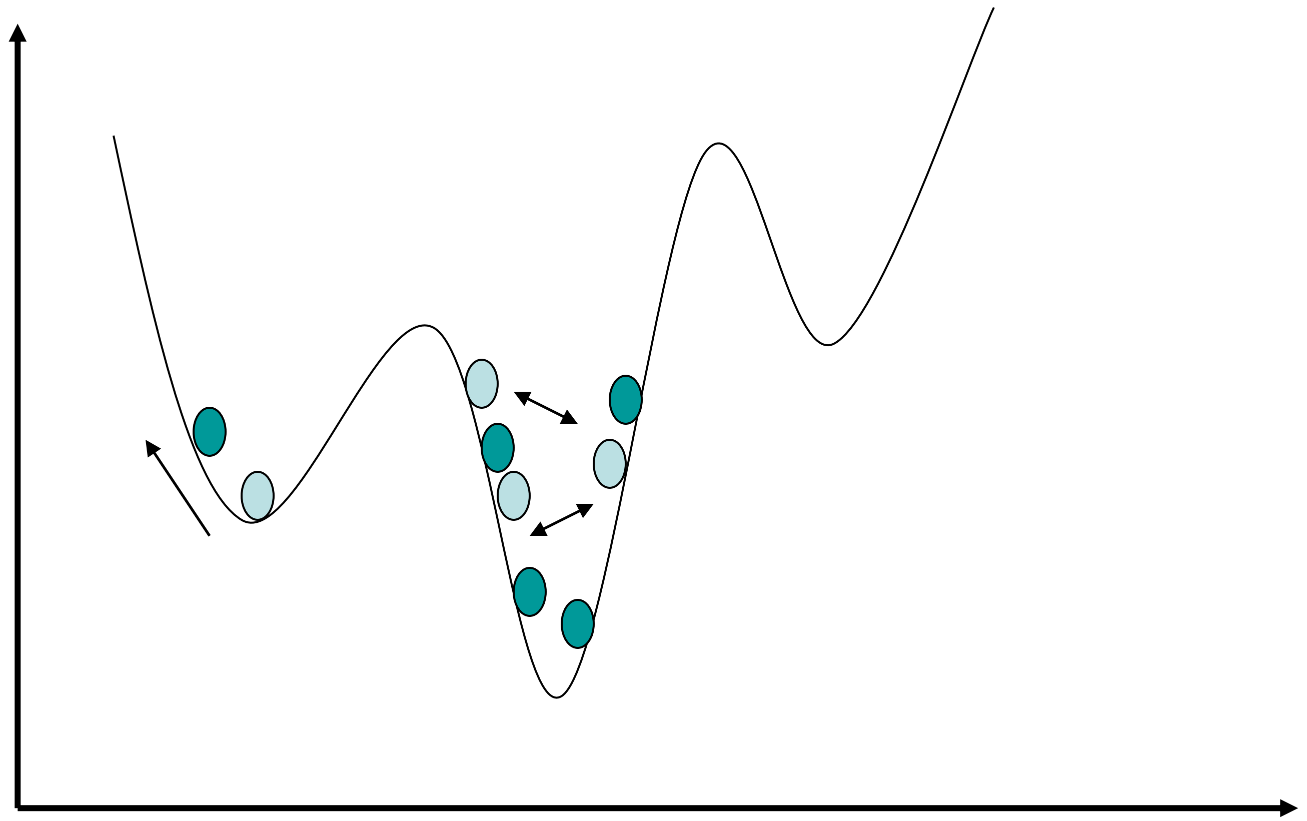
Genetic Algorithms



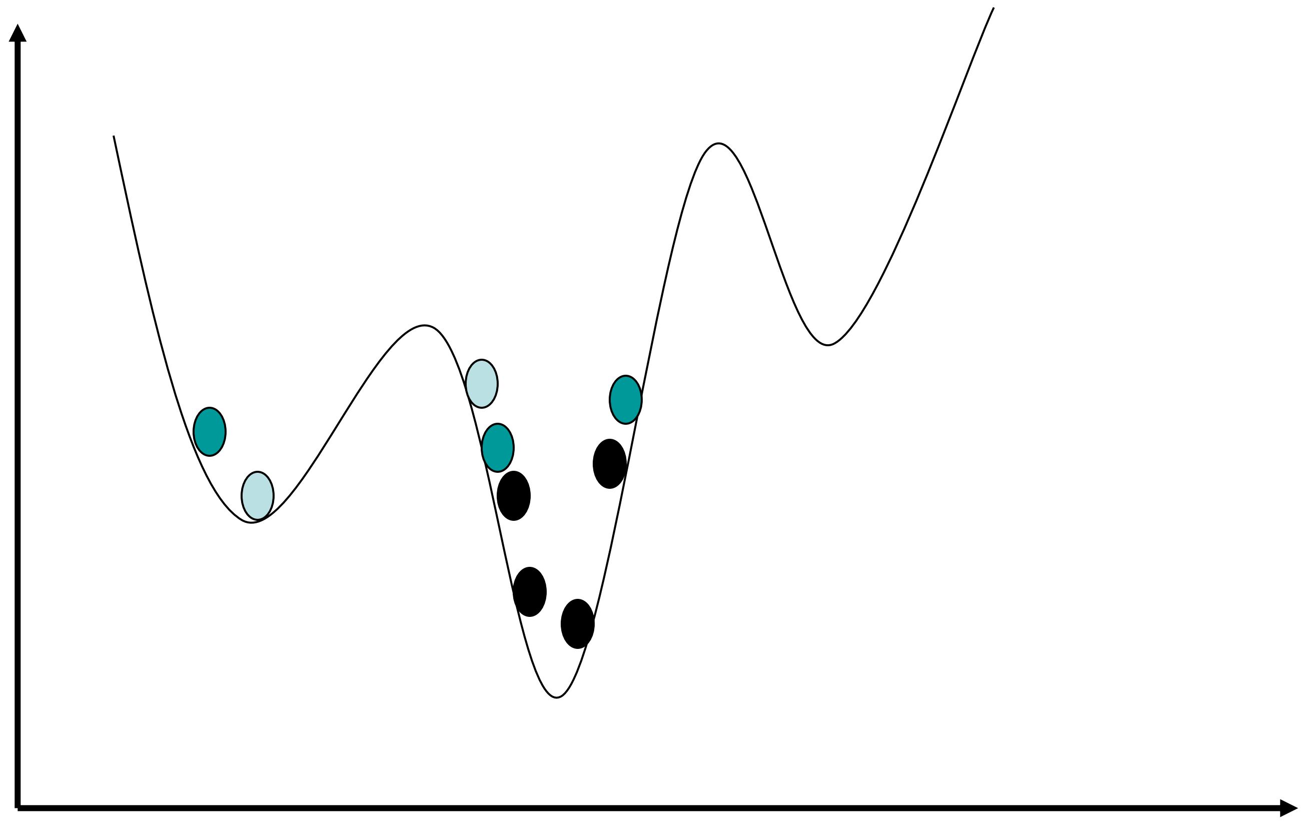
Genetic Algorithms



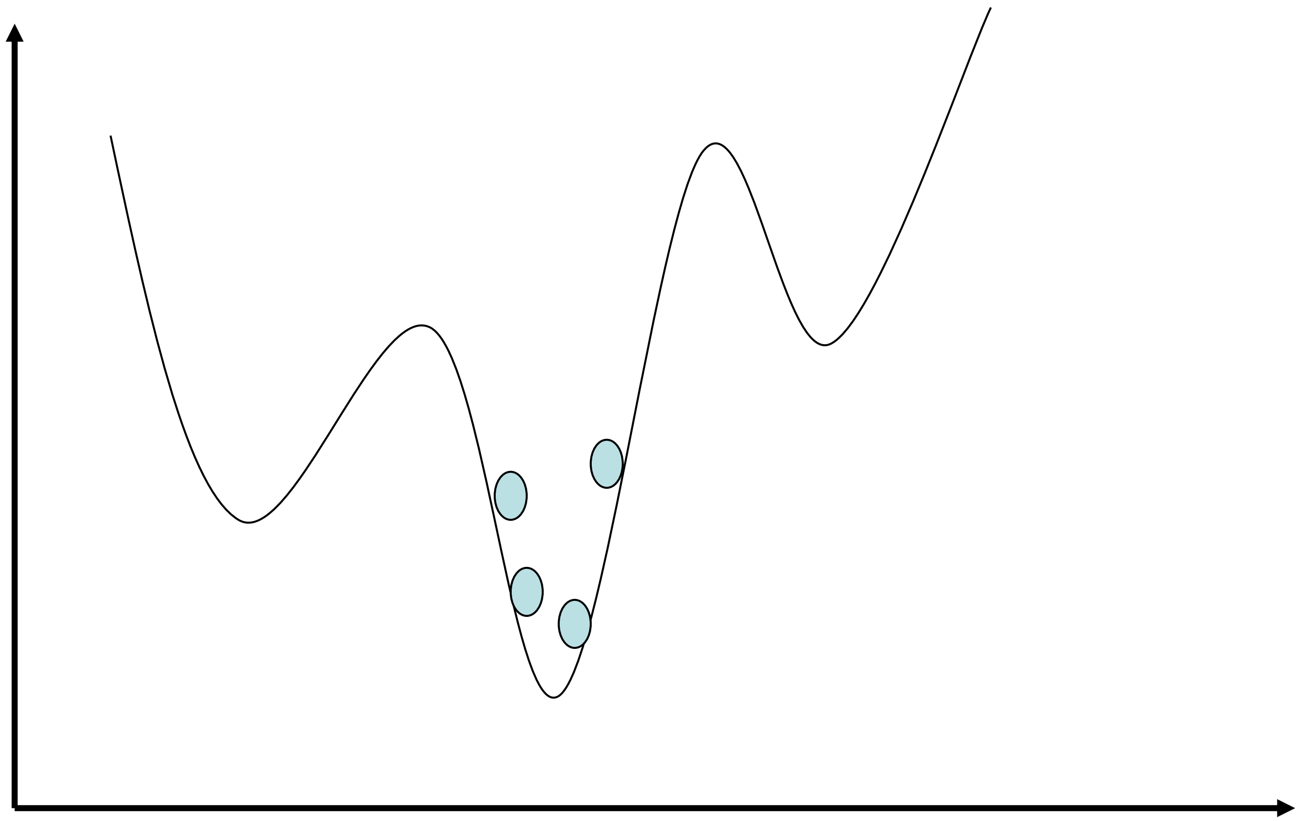
Genetic Algorithms



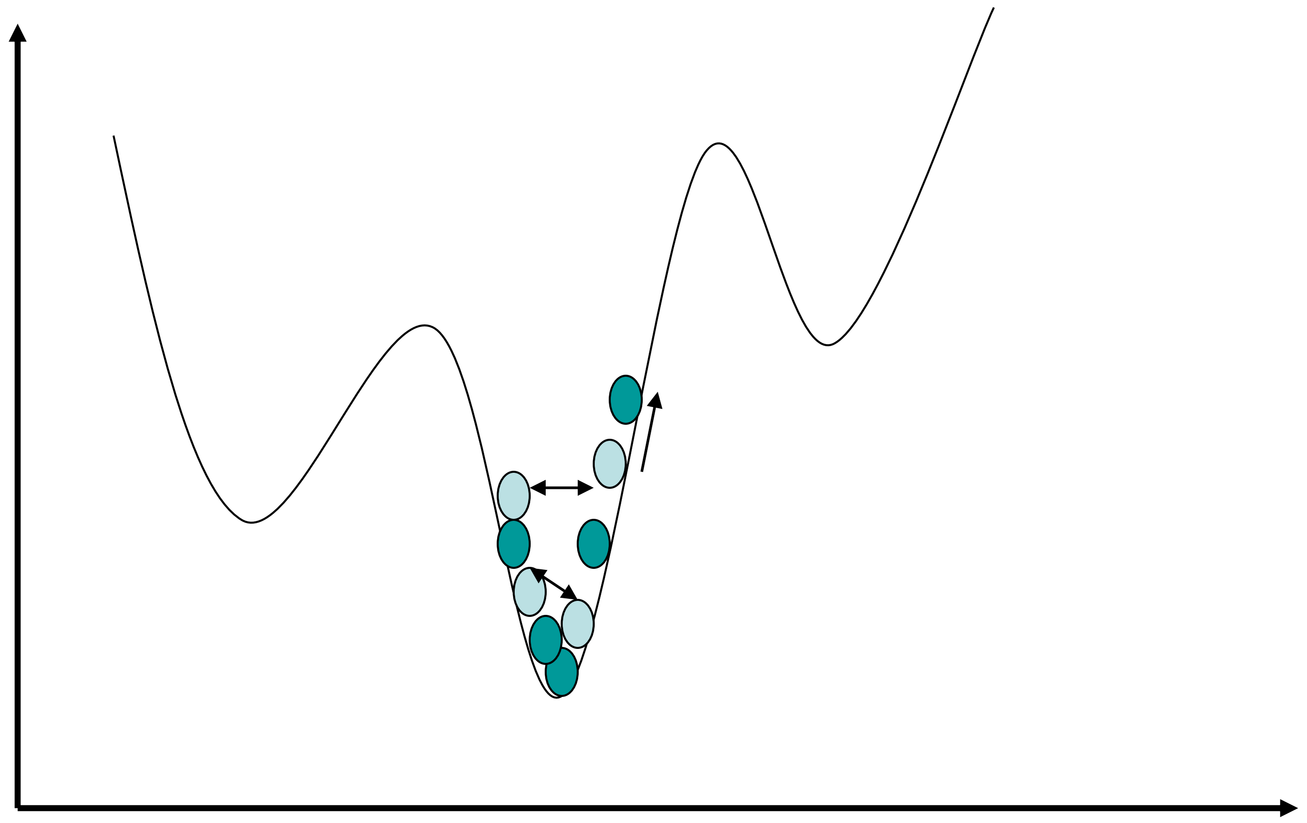
Genetic Algorithms



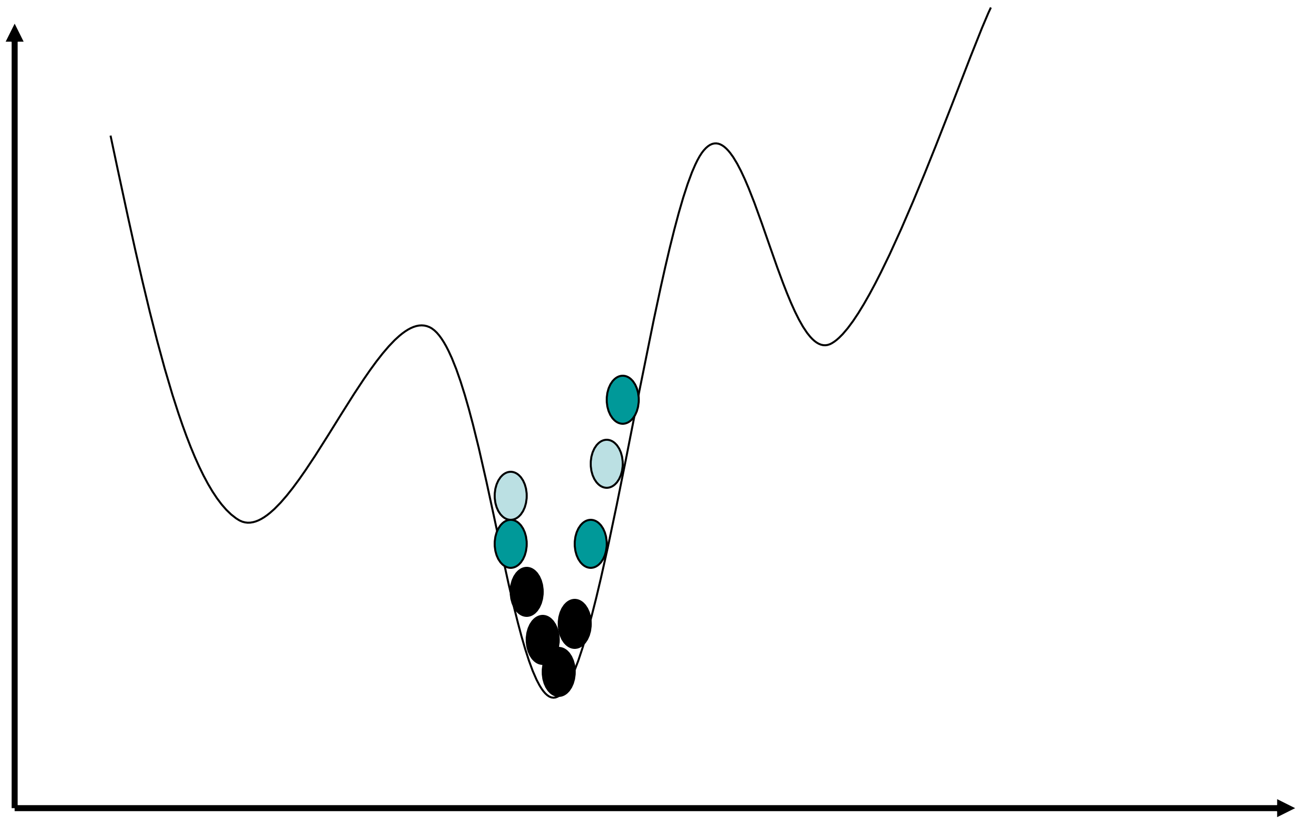
Genetic Algorithms



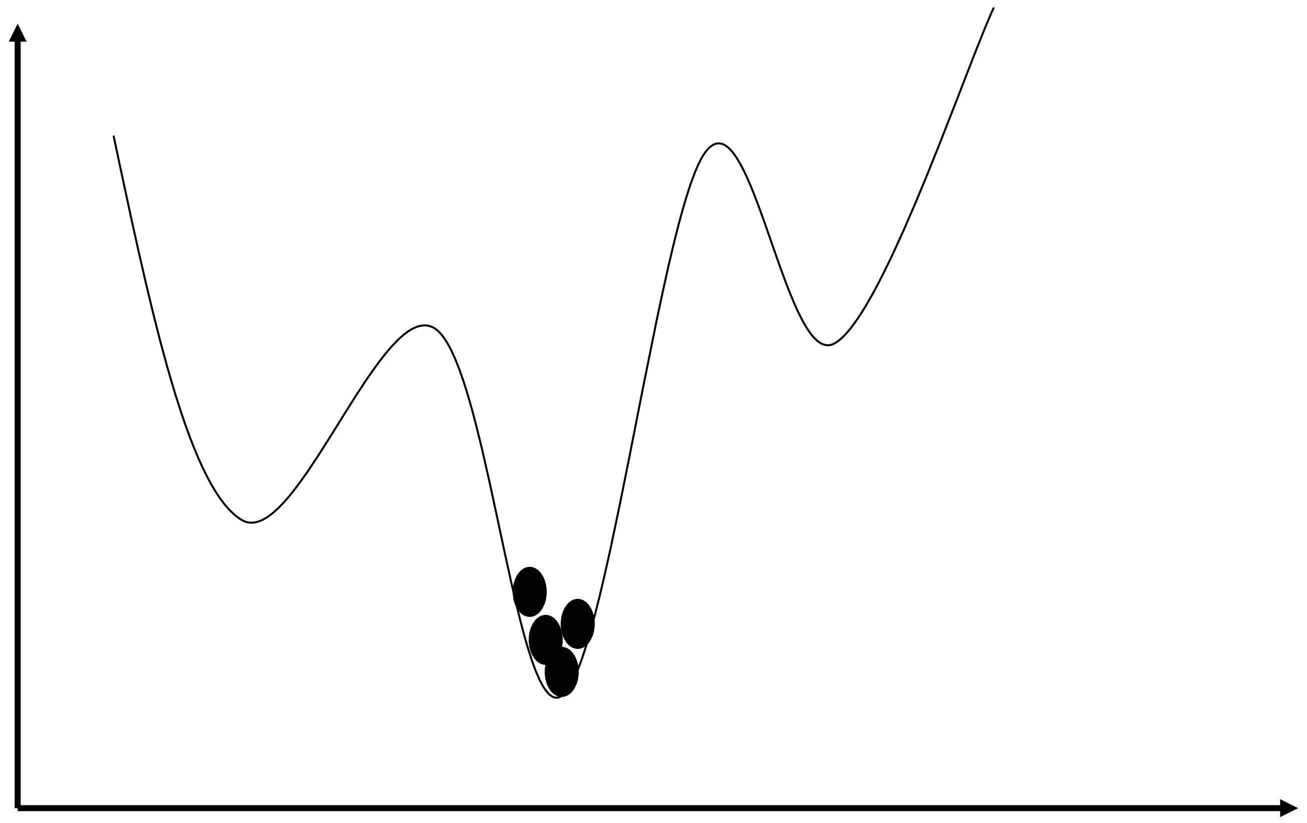
Genetic Algorithms



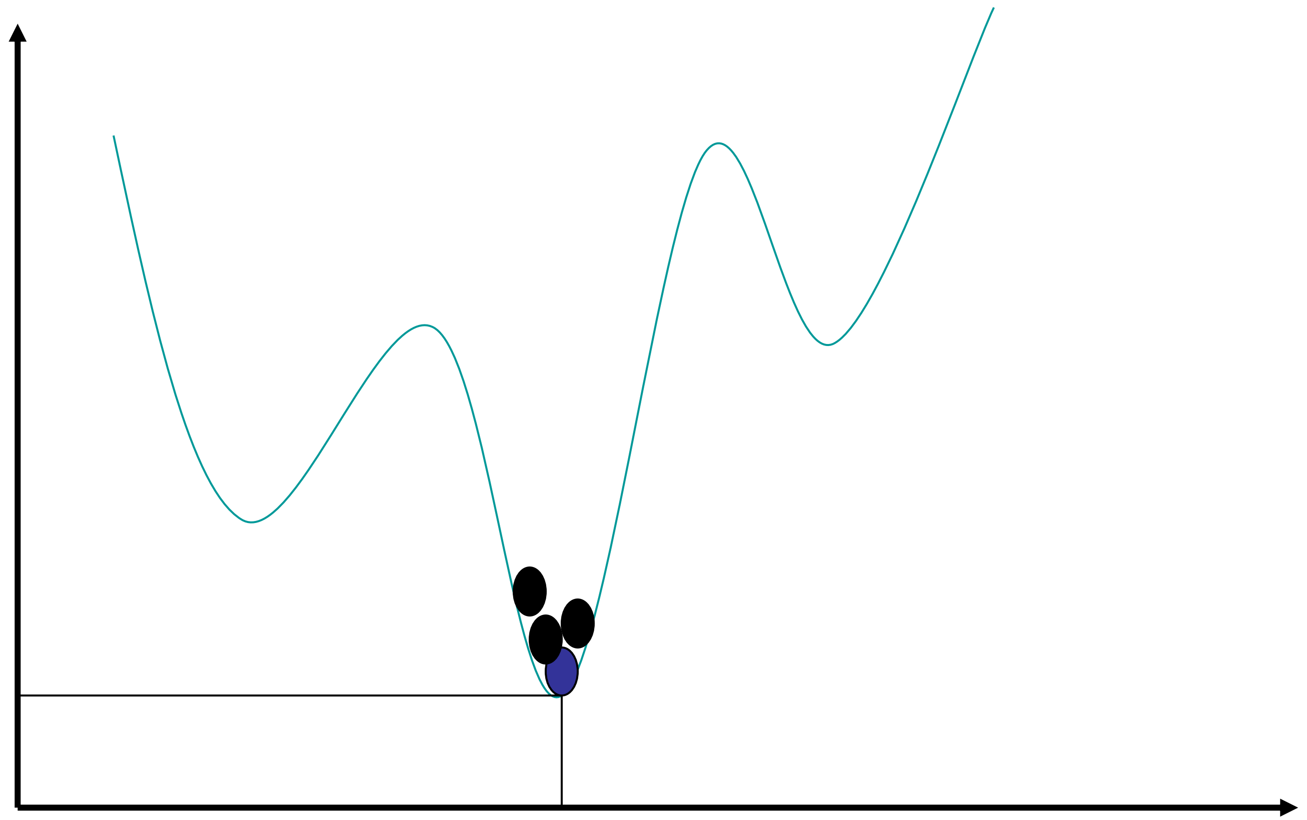
Genetic Algorithms



Genetic Algorithms



Genetic Algorithms



Problem Encoding

Problem: Find location that is to several given cities

Population encoding: Express location l as a 16-bit string

$$l = 1001010101011100$$

with the first 8 bits representing the location's X-coordinate and the second 8 bits representing the Y-coordinate

Fitness function: Median distance of location from each city

Combination: Crossover of X-coordinate from one parent and Y-coordinate from the other

Mutation: one or more bit flips

Problem Encoding

Problem: Finding the max value of some function $f : [0, 1)^n \to \mathbb{R}$

Population encoding: Vectors of size n with elements from $[0, 1)$

Combination: Various options

Mutation: randomly replace a value in the vector with one from $[0, 1)$

Recombination : Similar to crossover

Discrete Recombination

Similar to crossover

Equal probability of receiving each parameter from either parent

Example:

$$(8, 12, 31, \dots, 5) \ (2, 5, 23, \dots, 14)$$

$$\Downarrow$$

$$(2, 12, 31, \dots, 14)$$

Intermediate Recombination

Each child component is the average of the corresponding parent components

Example:

$$(8, 12, 31, \dots, 5) \ (2, 5, 23, \dots, 14)$$

$$\Downarrow$$

$$(5, 8.5, 27, \dots, 9.5)$$

From: CS:4420

GA for the Traveling Salesperson Problem

Problem: Find a tour of a given set of cities so that each city is visited only once and total traveled is minimal

Representation: An ordered list of city numbers (known as order-based GA)

1) London	3) Iowa City	5) Beijing	7) Tokyo
2) Venice	4) Singapore	6) Phoenix	8) Victoria

Ex.,

CityList1 (3 5 7 2 1 6 4 8)
CityList2 (2 5 7 6 8 1 3 4)

From: CS.4420

GA for the Traveling Salesperson Problem

Combination: Order 1 crossover (combines inversion and recombination):

1. Copy a randomly selected portion of Parent 1 to Child
2. Fill the blanks in Child with those numbers in Parent 2 from left to right, avoiding duplicates in Child

Parent 1	(3 5 7 2 1 6 4 8)
Parent 2	(2 5 7 6 8 1 3 4)
Child	(5 8 7 2 1 6 3 4)

GA for the Traveling Salesperson Problem

Mutation: swap two numbers in the list

1. Copy a randomly selected portion of Parent 1 to Child
2. Fill the blanks in Child with those numbers in Parent 2 from left to right, avoiding duplicates in Child

Before: (5 8 7 2 1 6 3 4)

After: (5 8 6 2 1 7 3 4)

Local search in continuous spaces

Local Search in Continuous Spaces

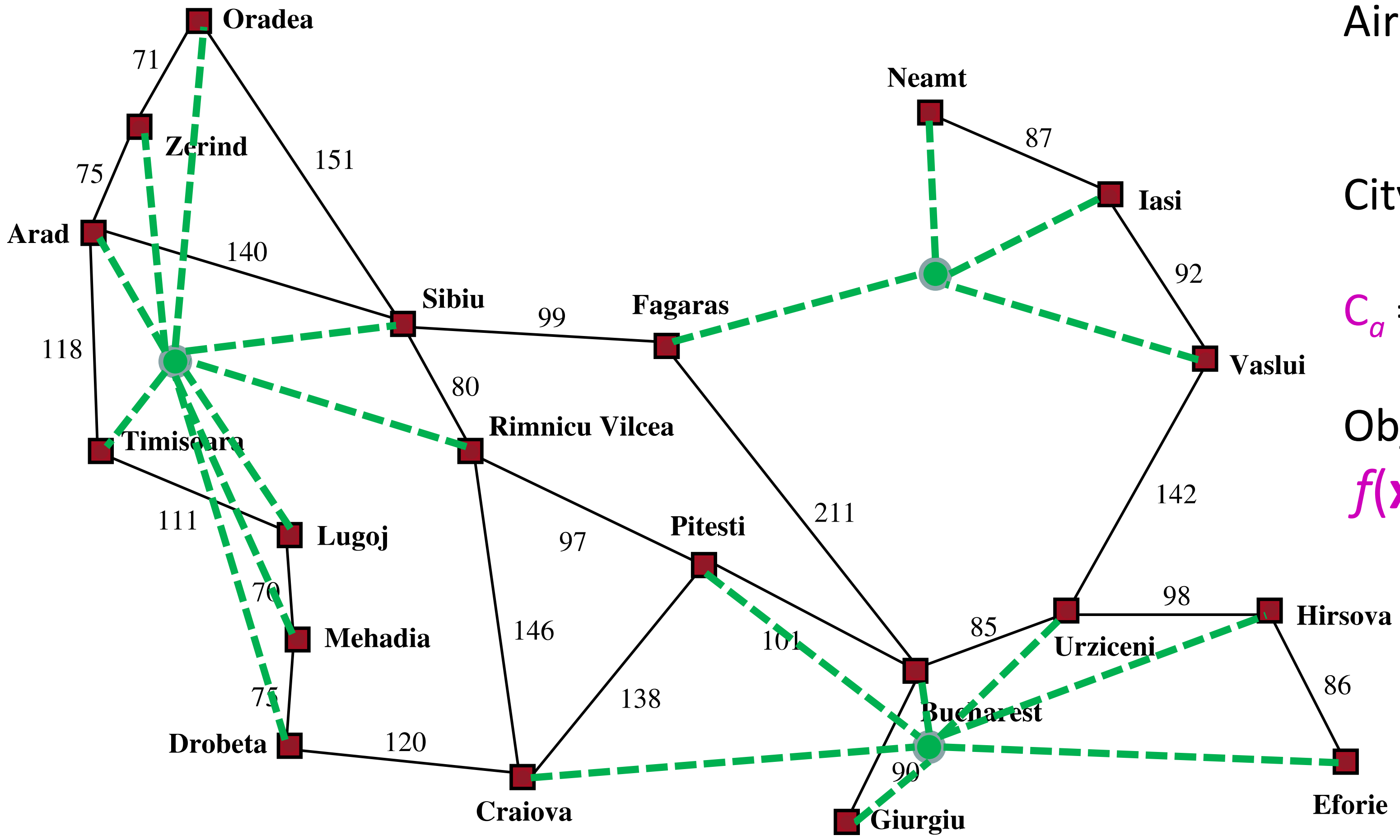
- There is a distinction between discrete and continuous environments. The most real-world environments are continuous. A continuous action space has an infinite branching factor, and thus can't be handled by most of the algorithms we have covered so far (with the exception of first-choice hill climbing and simulated annealing).
- Suppose we want to place three new airports anywhere in Romania, such that the sum of squared straight-line distances from each city on the map to its nearest airport is minimized. (See Figure 3.1 for the map of Romania.) The state space is then defined by the coordinates of the three airports:

  (x1;y1), (x2;y2), and (x3;y3).
- This is a six-dimensional space; we also say that states are defined by six variables.
- The objective function f (x) = f (x_1;y_1;x_2;y_2;x_3;y_3) is relatively easy to compute for any particular state once we compute the closest cities. Let C_i be the set of cities whose closest airport (in the state x) is airport i. Then, we have

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

Example: Siting airports in Romania

Place 3 airports to minimize the sum of squared distances from each city to its nearest airport



Airport locations

$\mathbf{x} = (x_1,y_1), (x_2,y_2), (x_3,y_3)$

City locations (x_c,y_c)

C_a = cities closest to airport a

Objective: minimize

$$f(\mathbf{x}) = \Sigma_a \, \Sigma_{c \in C_a} \, (x_a - x_c)^2 + (y_a - y_c)^2$$

Handling a continuous state/action space

- One way to deal with a continuous state space is to discretize it. For example, instead of allowing the (x_i;y_i) locations to be any point in continuous two-dimensional space, we could limit them to fixed points on a rectangular grid with spacing of size δ (delta).

1. Discretize it!
   - Define a grid with increment δ, use any of the discrete algorithms
2. Choose random perturbations to the state
   a. First-choice hill-climbing: keep trying until something improves the state
   b. Simulated annealing
3. Compute gradient of $f(\mathbf{x})$ analytically

Finding extrema in continuous space

- The gradient of the objective function is a vector $\nabla f(\mathbf{x})$ that gives the magnitude and direction of the steepest slope. For our problem, we have

  Gradient vector: $\nabla f(\mathbf{x}) = (\partial f/\partial x_1, \partial f/\partial y_1, \partial f/\partial x_2, \ldots)^T$

  For the airports: $f(\mathbf{x}) = \Sigma_a \Sigma_{c \in C_a} (x_a - x_c)^2 + (y_a - y_c)^2$
- In some cases, we can find a maximum by solving the equation $\nabla f(\mathbf{x}) = 0$
- For example, with three airports, the expression for the gradient depends on what cities are closest to each airport in the current state. This means we can compute the gradient locally (but not globally); for example,

$$\partial f/\partial x_1 = \Sigma_{c \in C_1} 2(x_1 - x_c)$$

- Given a locally correct expression for the gradient, we can perform steepest-ascent hill climbing by updating the current state according to the gradient descent formula:

$$\mathbf{x} \leftarrow \mathbf{x} - \alpha \nabla f(\mathbf{x})$$

  where α (alpha) is a small constant often called the step size.
- There are huge range of algorithms for finding extrema using gradients.

Summary

- Many configuration and optimization problems can be formulated as local search.
- Local search methods keep small number of nodes in memory. They are suitable for problems where the solution is the goal state itself and not the path.
- General families of algorithms:
  - Hill-climbing, continuous optimization
  - Simulated annealing (and other stochastic methods)
  - Local beam search: multiple interaction searches
  - Genetic algorithms: break and recombine states
- Genetic algorithms are a kind of stochastic hill-climbing search in which a large population of states is maintained. New states are generated by mutation and by crossover which combines pairs of states from the population.

Many machine learning algorithms are local searches