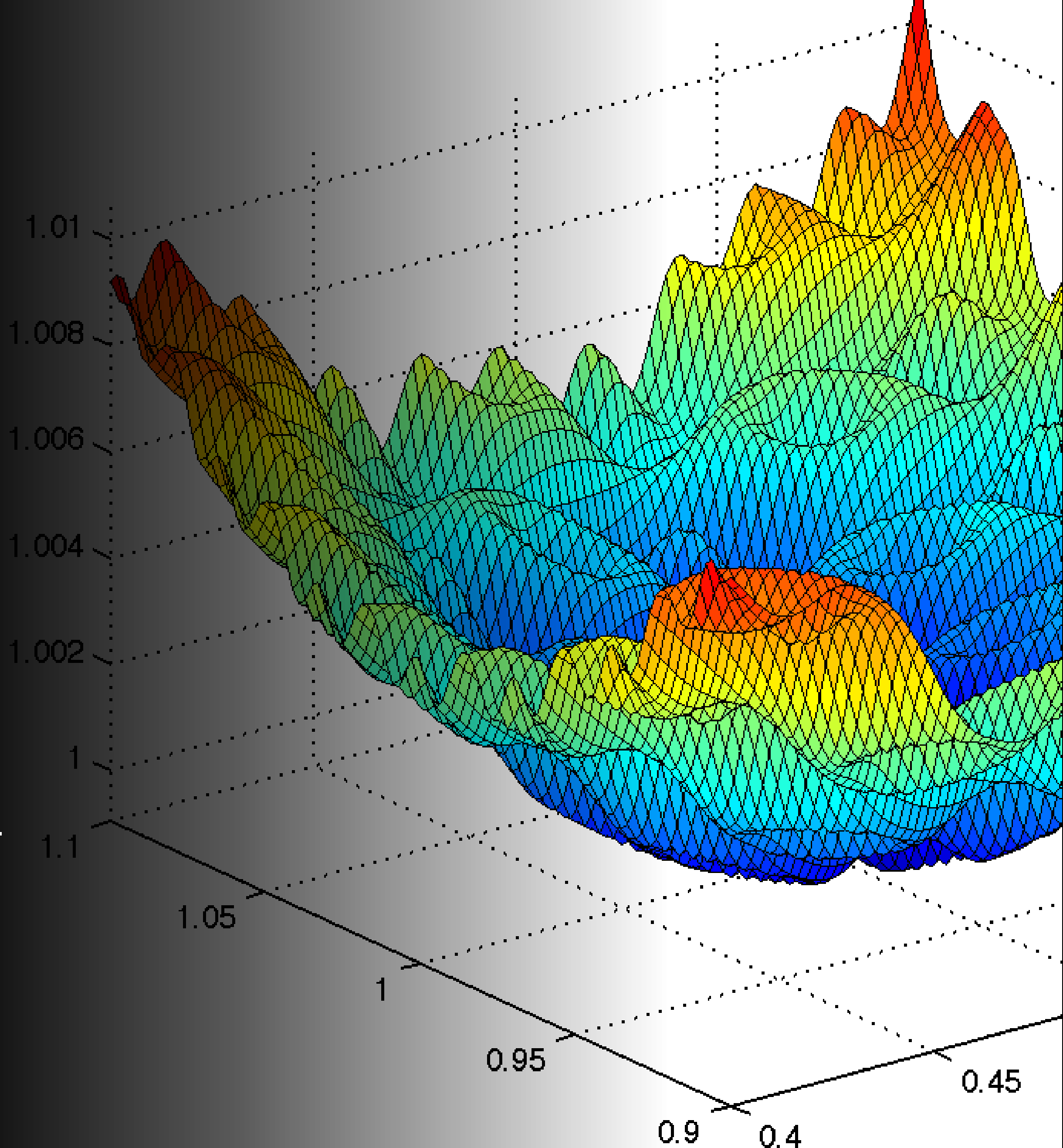
# CS 5/7320
## Artificial Intelligence

# Local Search
## AIMA Chapters 4.1 & 4.2


Slides by Michael Hahsler
based on slides by Svetlana Lazepnik
with figures from the AIMA textbook.

# Recap: Uninformed and Informed Search

Tries to plan the

**best path**

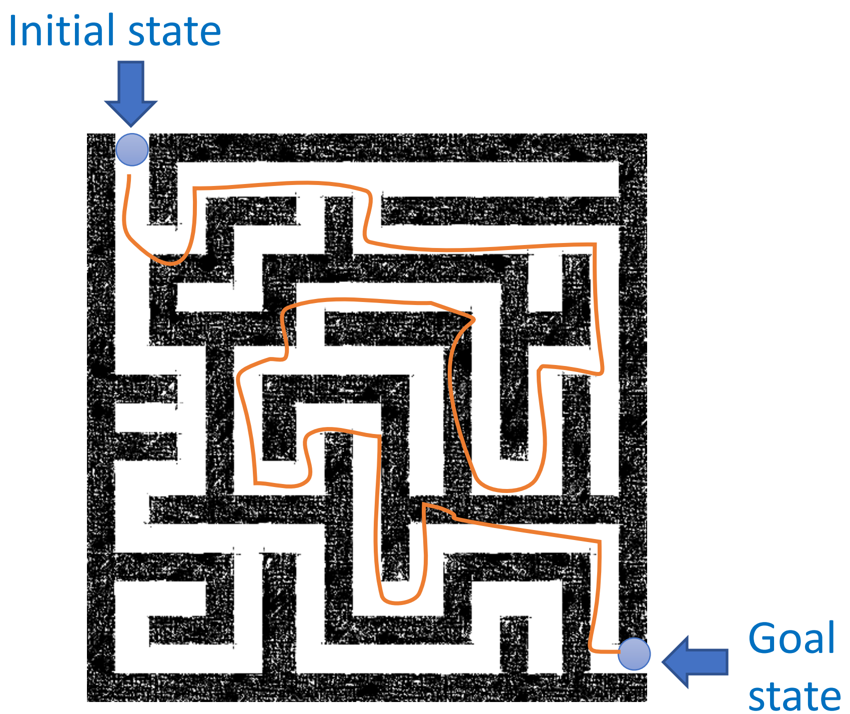
from a

**given initial state**

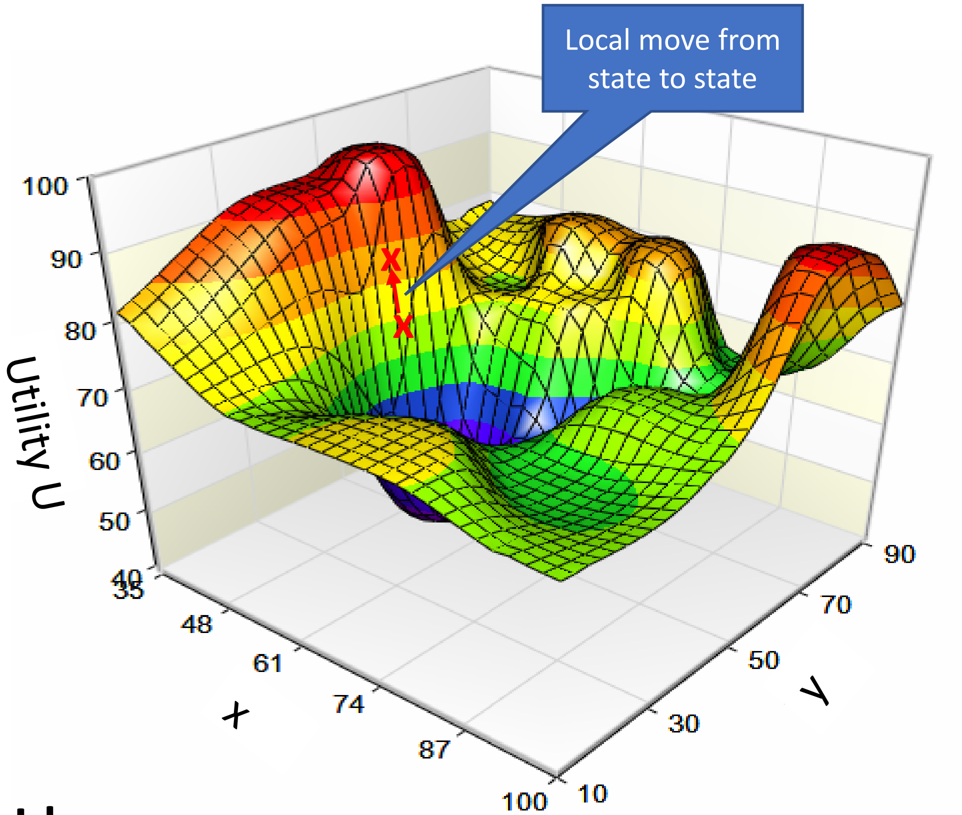
to a

**given goal state.**

- Typically searches a large portion of the search space (needs time and memory).
- Often comes with optimality guarantees (BFS, A* Search, IDS).



# Local Search Algorithms




- What if we do not know the goal state, but the utility of different states is given by a utility function $U = u(s)$?
- We use a factored state description. Here $s = (x, y)$
- We could try to identify the best or at least a "good" state?
- This is the **optimization problem**: $$s^* = \operatorname*{argmax}_{s \in S} u(s)$$
- We need a fast and memory-efficient way to find the best/a good state.

**Idea**:

Start with a current solution (a state) and improve the solution by moving from the current state to a "neighboring" better state (a.k.a. performing a series of **local moves**).

# Local Search Algorithms

**Difference to search from the previous chapter:**

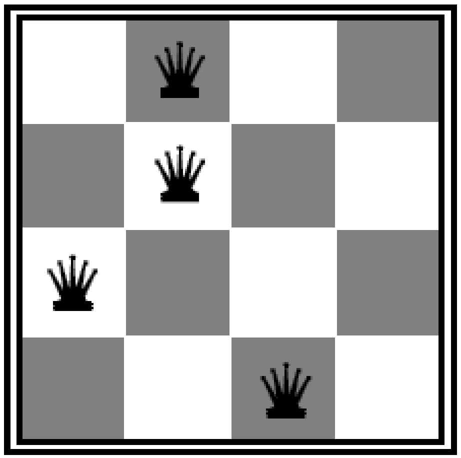
a) **Goal state is unknown,** but we know or can calculate the utility for each state. We want to identify the state with the highest utility.
b) Often no explicit initial state + **path to goal and path cost are not important.**
c) **No search tree**. Just stores the current state and move to a “better” state if possible.

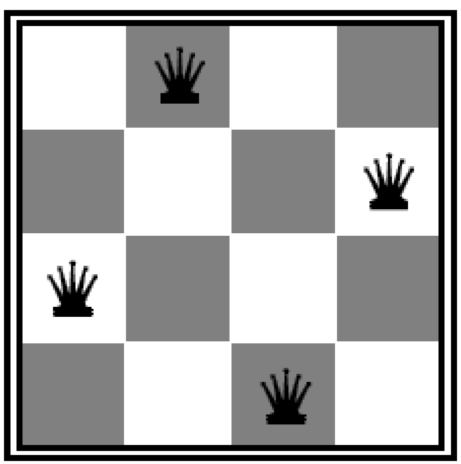
**Use in AI**

- **Goal-based agent**: Identify a good goal state with a good utility before planning a path to that state.
- **Utility-based agent**: Always move to neighboring higher utility states. A simple greedy method used for complicated/large state spaces or online search.
- **General optimization**: $u(s)$ can be replaced by a general objective function. Local search is an effective heuristic to find good solutions in large or continuous search spaces. E.g., gradient descend to train neural networks.

# Example: *n*-Queens Problem

states





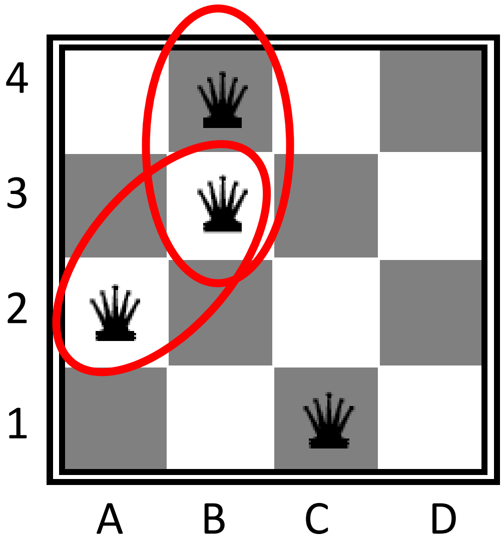
**Goal**: Put *n* queens on an *n* × *n* board with no two queens on the same row, column, or diagonal.

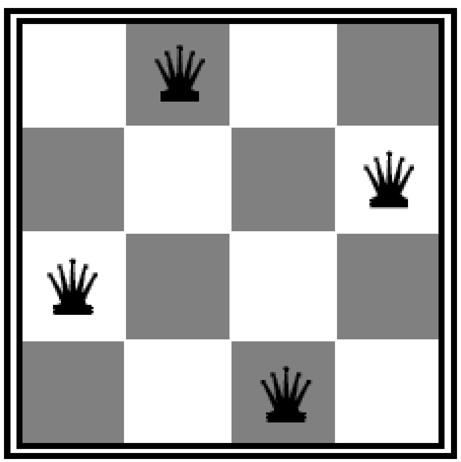
**Defining the search problem:**

- **State space:** All possible *n*-queen configurations. How many are there?
- **State representation:** How do we define a factored representation?
- **Objective function**: What is a possible utility function given the state representation?
- **Local neighborhood:** What states are close to each other?

# Example: *n*-Queens Problem

2 conflicts = utility of -2



0 conflicts = utility of 0

**Defining the search problem:**

- **State space:** All possible *n*-queen configurations. How many are there?
  4-queens problem: $\binom{16}{4} = 1820$
- **State representation:** How do we define a facroted representation?
  E.g. $(A2, B3, B4, C1)$
- **Objective function**: What is a possible utility function given the state representation?
  Maximizing utility means minimize the number of pairwise conflicts based on the state representation.
- **Local neighborhood:** What states are close to each other?
  Move a single queen.


# Example: Traveling salesman problem

- **Goal**: Find the shortest tour connecting a given set of cities
- **State space:** all possible tours (states are not individual cities!)
- **State representation:** Order of cities in the tour.
- **Objective function:** minimize the length of the tour
- **Local neighborhood:** Change the order of visiting a few cities.

Note: We have solved a different problem with uninformed/informed search! Each city was defined as a state and the path was the solution.


# Hill-Climbing Search
# aka Greedy Local Search

**Idea:** keep a single “current” state and try to find better neighboring states.

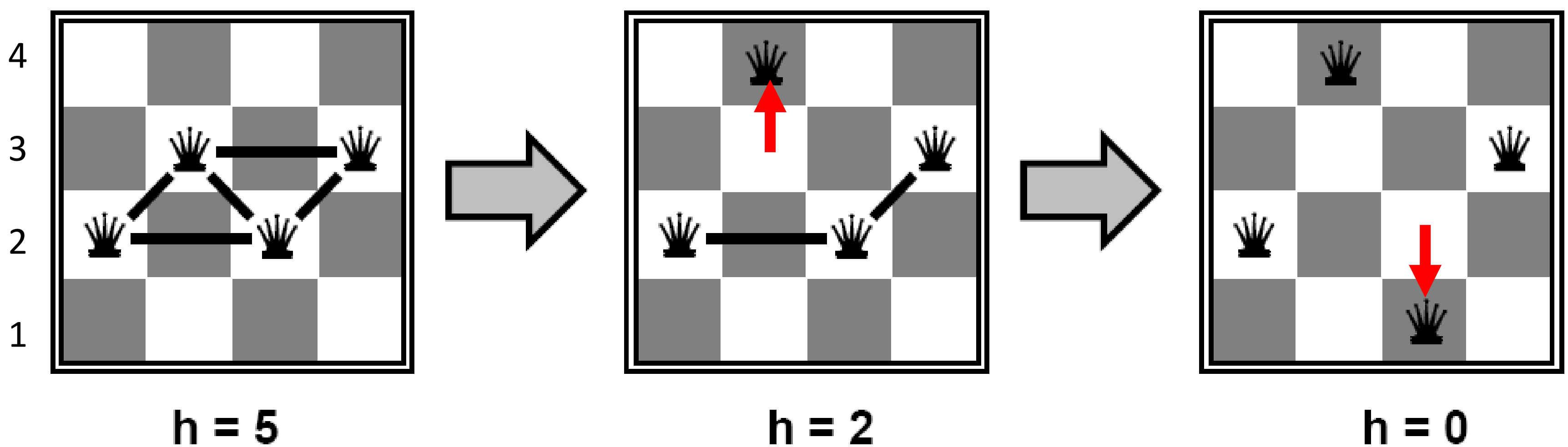
# Example: *n*-Queens Problem

- **Goal:** Put $n$ queens on an $n \times n$ board with no two queens on the same row, column, or diagonal.
- **State space:** all possible $n$-queen configurations. We can restrict the state space: Only one queen per column.
- **State representation:** row position of each queen in its column (e.g., 2, 3, 2, 3)
- **Objective function:** minimize the number of pairwise conflicts.
- **Local neighborhood:** Move one queen anywhere in its column.

State space is reduced from 1820 to $4^4 = 256$

**Improvement strategy**

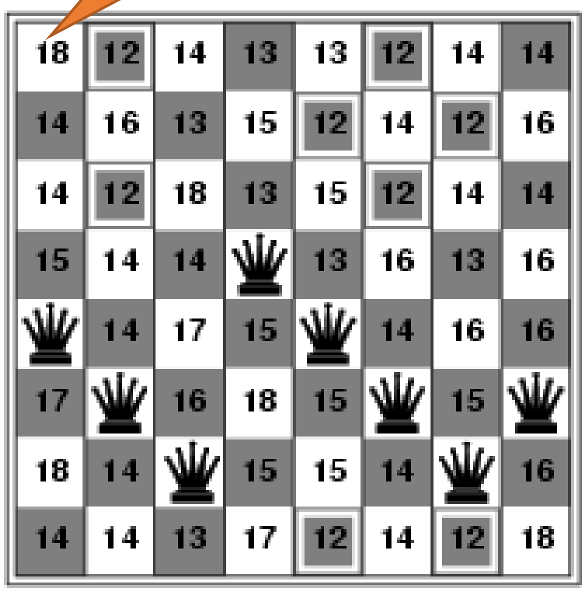
- Find a local neighboring state (move one queen within its column) to reduce conflicts



# Example: *n*-Queens Problem

To find the best local move, we must evaluate all local neighbors (moving a single queen in its column while leaving the others in place) and calculate the objective function.

Objective value after moving the queen to this square




Current objective value: $h = 17$
best local improvement has $h = 12$

Notes:

- There are many options with $h = 12$. We must choose one!
- Calculating all the objective values may be expensive!

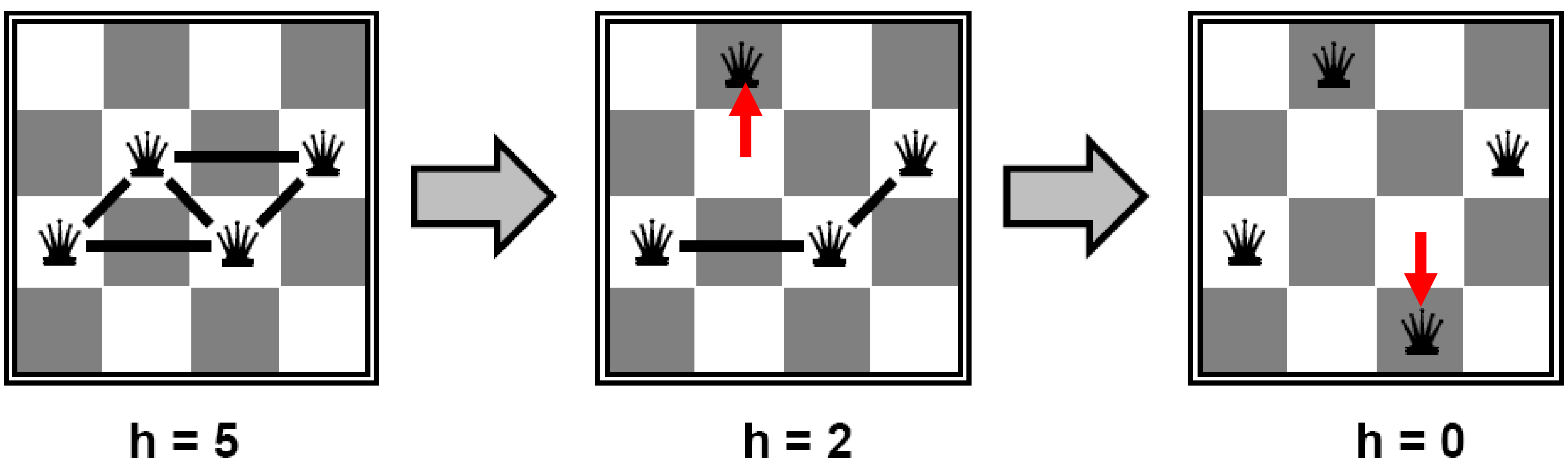
# Example: *n*-Queens Problem

Formulation as an optimization problem:
Find the best state $s^*$ representing an arrangement of queens.

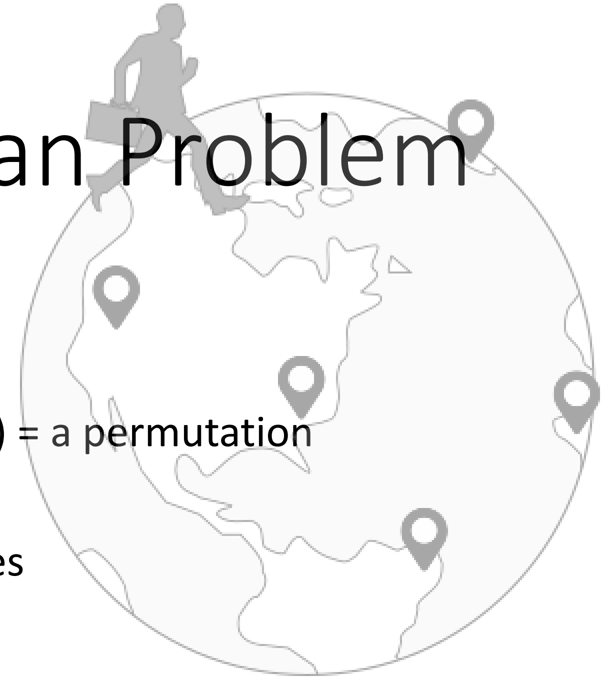
$$s^* = \operatorname{argmin}_{s \in S} \ \text{conflicts}(s)$$

subject to: $s$ has one queen per column

Remember: This makes the problem a lot easier.

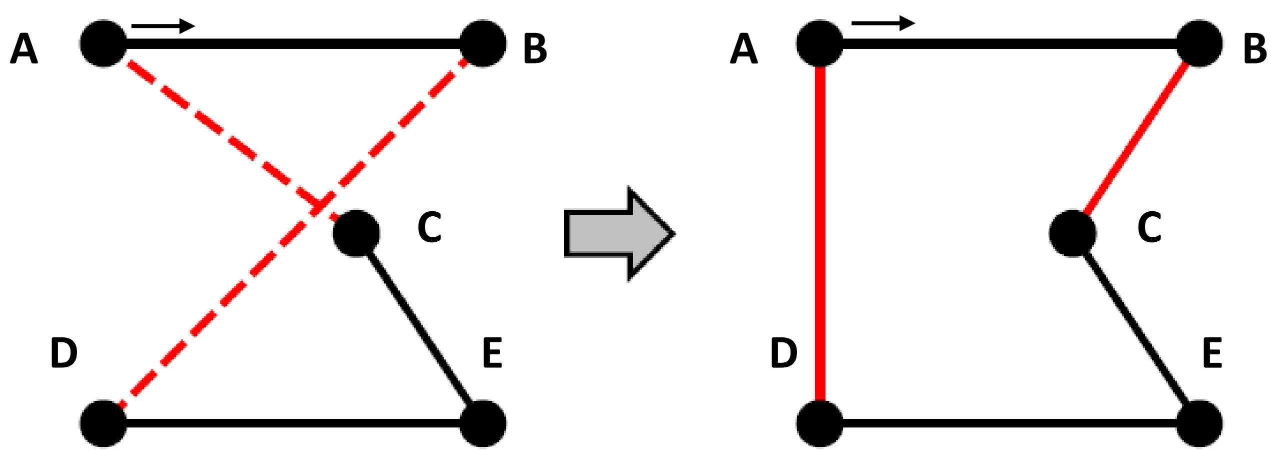


# Example: Traveling Salesman Problem



- **Goal:** Find the shortest tour connecting n cities
- **State space:** all possible tours
- **State representation:** tour (order in which to visit the cities) = a permutation
- **Objective function:** length of tour
- **Local neighborhood:** reverse the order of visiting a few cities

Local move to reverse the order of cities C, E and D:




State representation: **ABDEC** **ABCED**

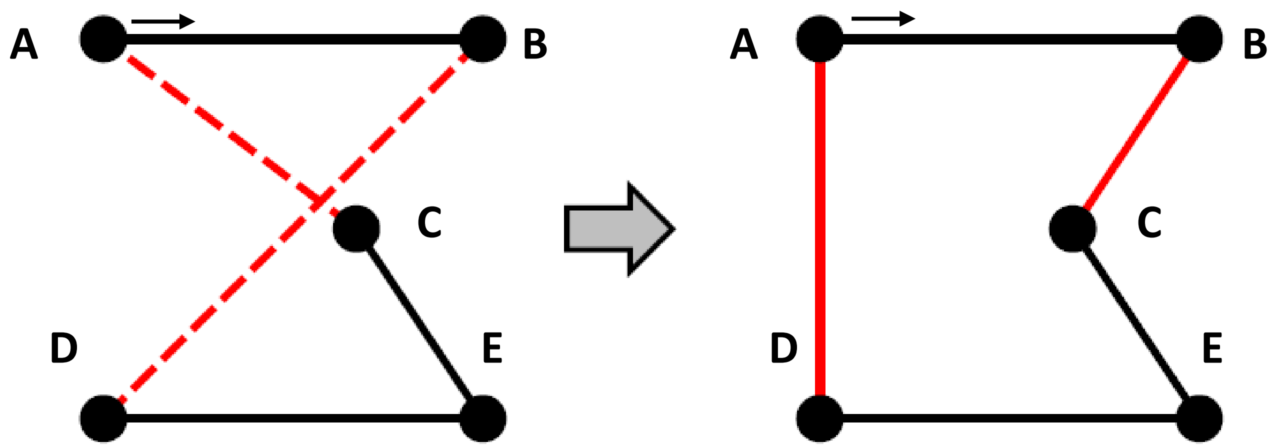
# Example: Traveling Salesman Problem

Formulation as an optimization problem:
Find the best tour $\pi$

$$\pi^* = \operatorname{argmin}_{\pi} \ \text{tourLength}(\pi)$$

s.t. $\pi$ is a valid permutation (i.e., sub-tour elimination)

Local move to reverse the order of cities C, E and D:




State representation: **ABDEC** **ABCED**

# Hill-Climbing Search (= Greedy Local Search)

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

Typically, we start with a random state

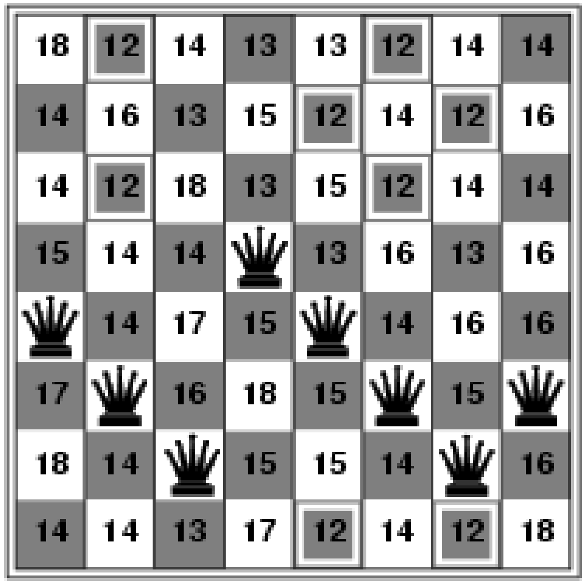
Variants:

**Steepest-ascend hill climbing**

- Check all possible successors and choose the highest-valued successors.

**Stochastic hill climbing**

- choose randomly among all uphill moves, or
- generate randomly one new successor at a time until a better one is found = first-choice hill climbing – **the most popular variant**, this is what people often mean when they say “stochastic hill climbing”

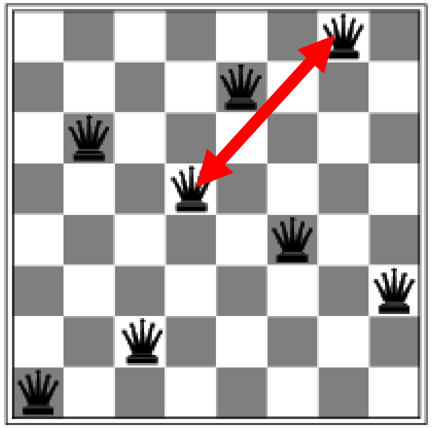


# Local Optima

Hill-climbing search is like greedy best-first search with the objective function as a (maybe not admissible) heuristic and no frontier (just stops in a dead end).

Is it complete/optimal?

- No – can get stuck in local optima



$h = 1$

Example: local optimum for the 8-queens problem. No single queen can be moved within its column to improve the objective function.

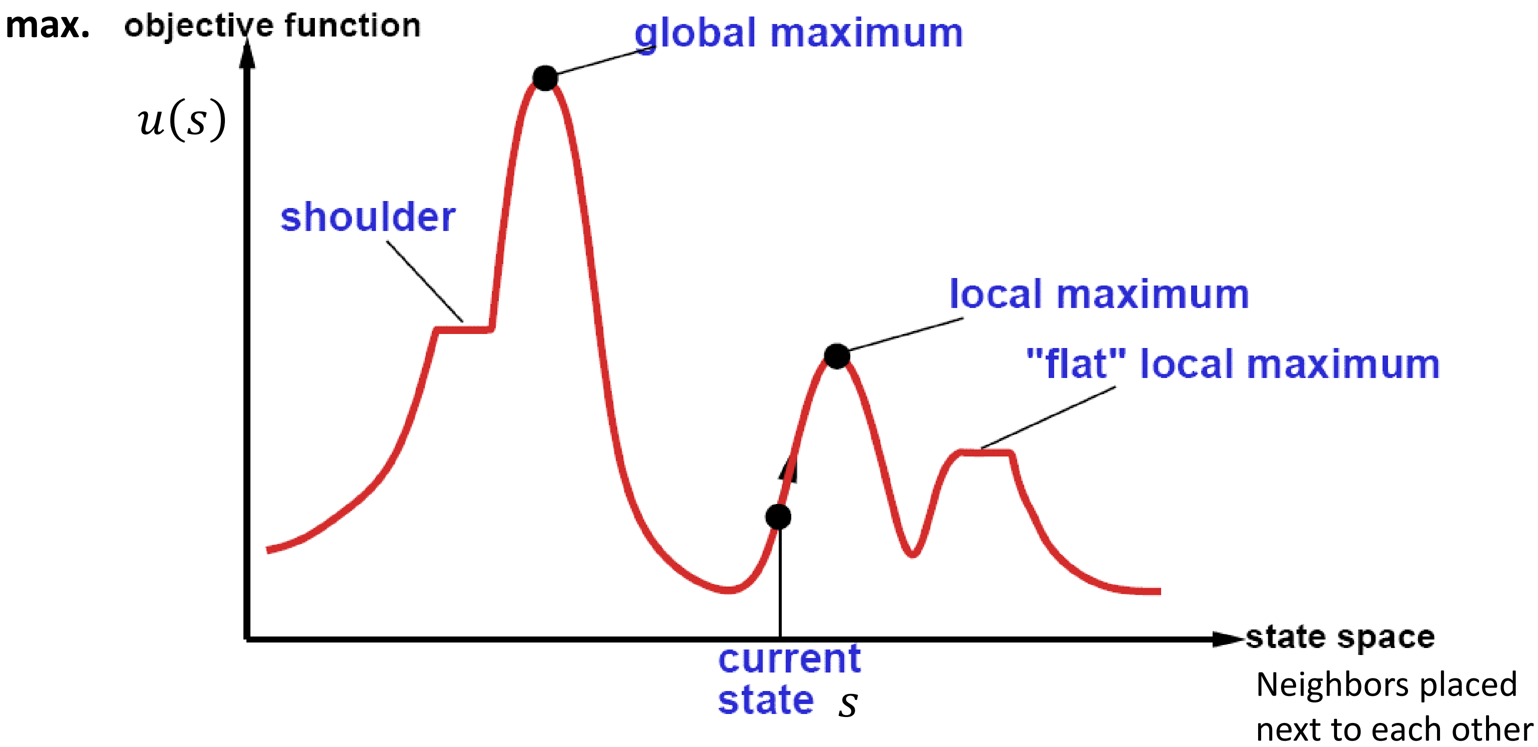
Simple approach that can help with local optima:

**Random-restart hill climbing**

- Restart hill-climbing many times with random initial states and return the best solution.

# The State Space “Landscape”

We can get the utility (objective function value) from the state description using $U = u(s)$.




How to escape local maxima?

→ Random restart hill-climbing can help.

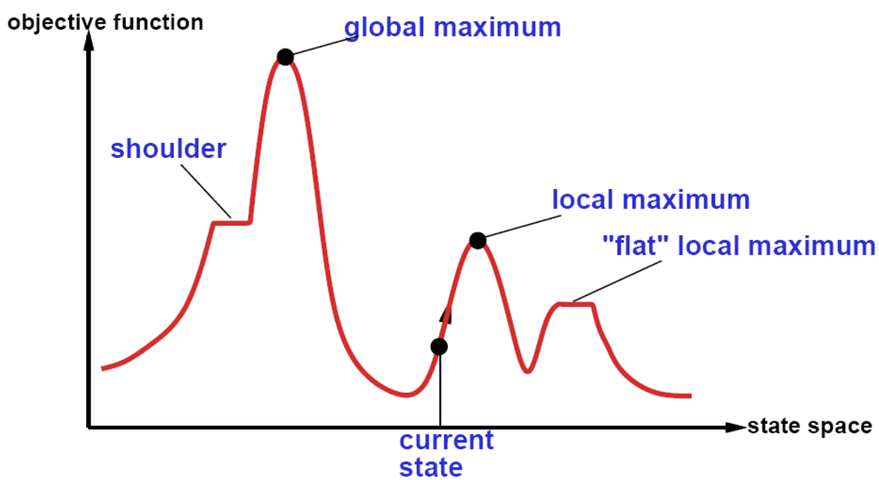
What about “shoulders” (called “ridges” in higher dimensional space)?

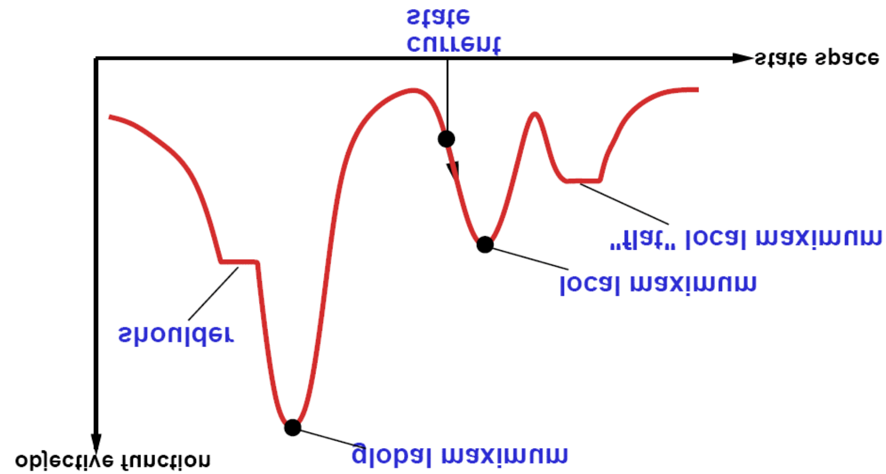
→ Hill-climbing that allows sideways moves and uses momentum.

# Minimization vs. Maximization

- The name hill climbing implies **maximizing a function**.
- Optimizers like to state problems as **minimization problems** and call hill climbing gradient descent instead.
- Both types of problems are equivalent:

$$\max(f(x)) \quad \Leftrightarrow \quad \min(-f(x))$$

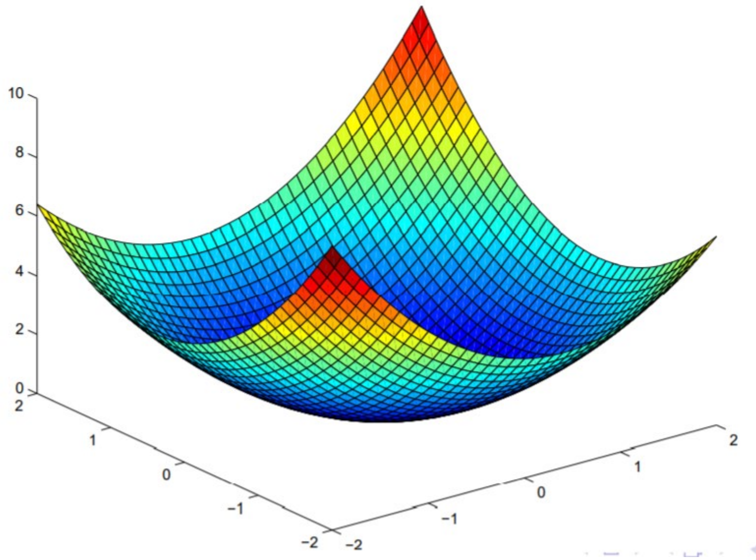


# Convex vs. Non-Convex Optimization Problems

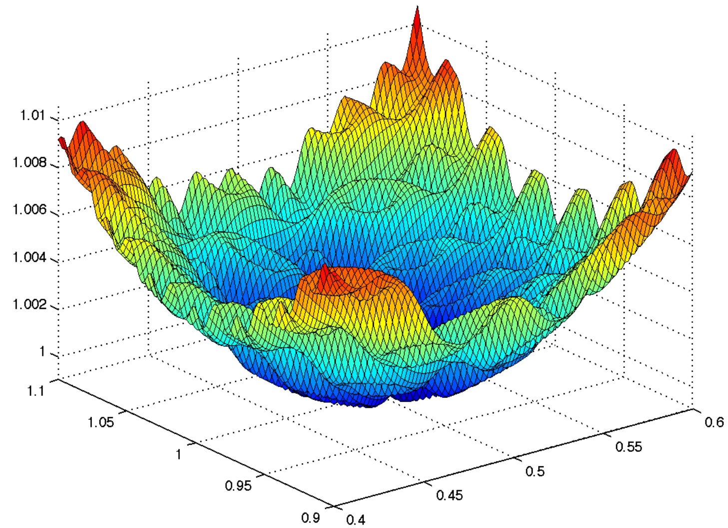
Minimization problems

Convex Problem



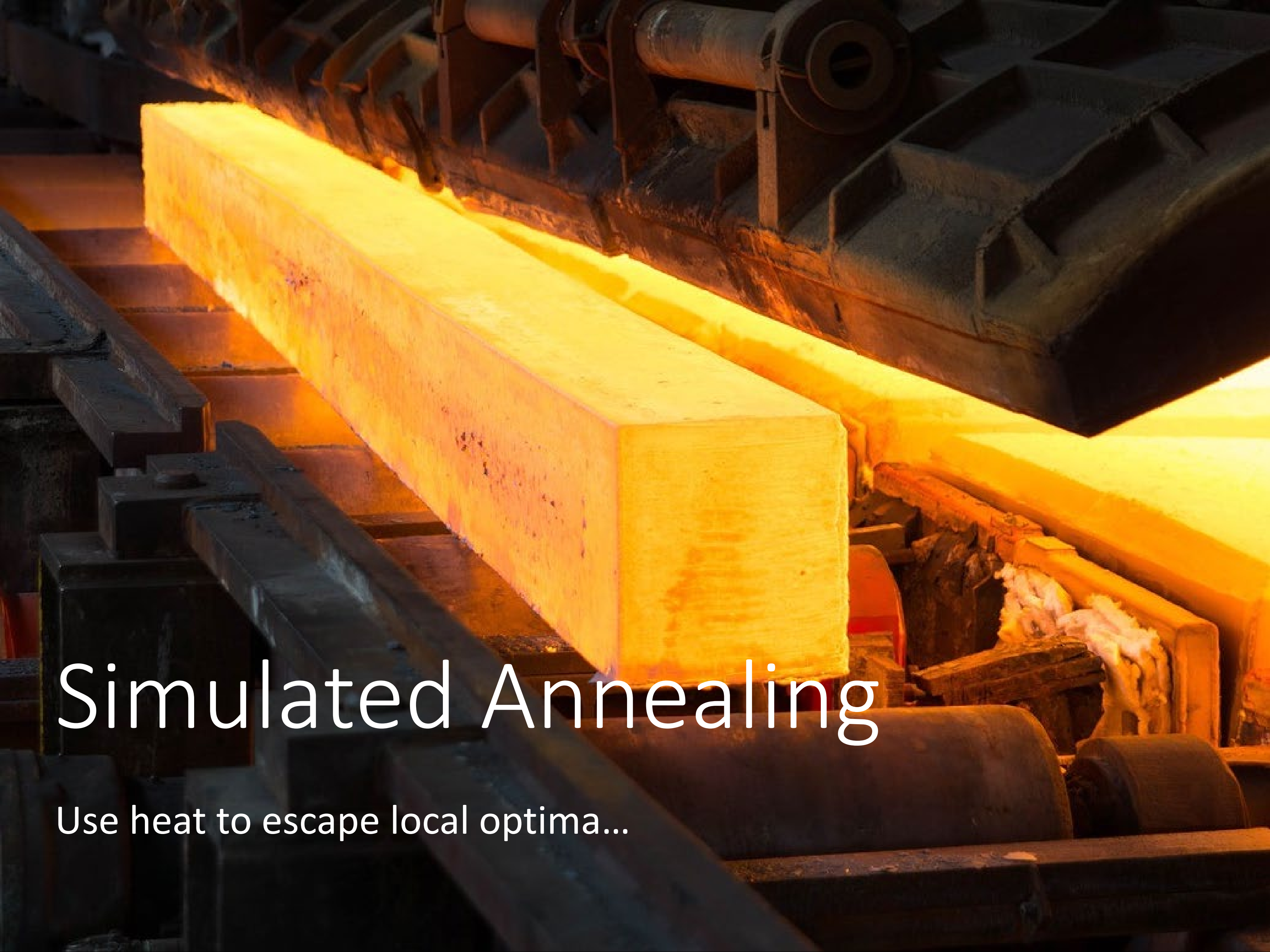

One global optimum + smooth function → calculus makes it easy

Non-convex Problem




Many local optima → hard

Many discrete optimization problems are like this.

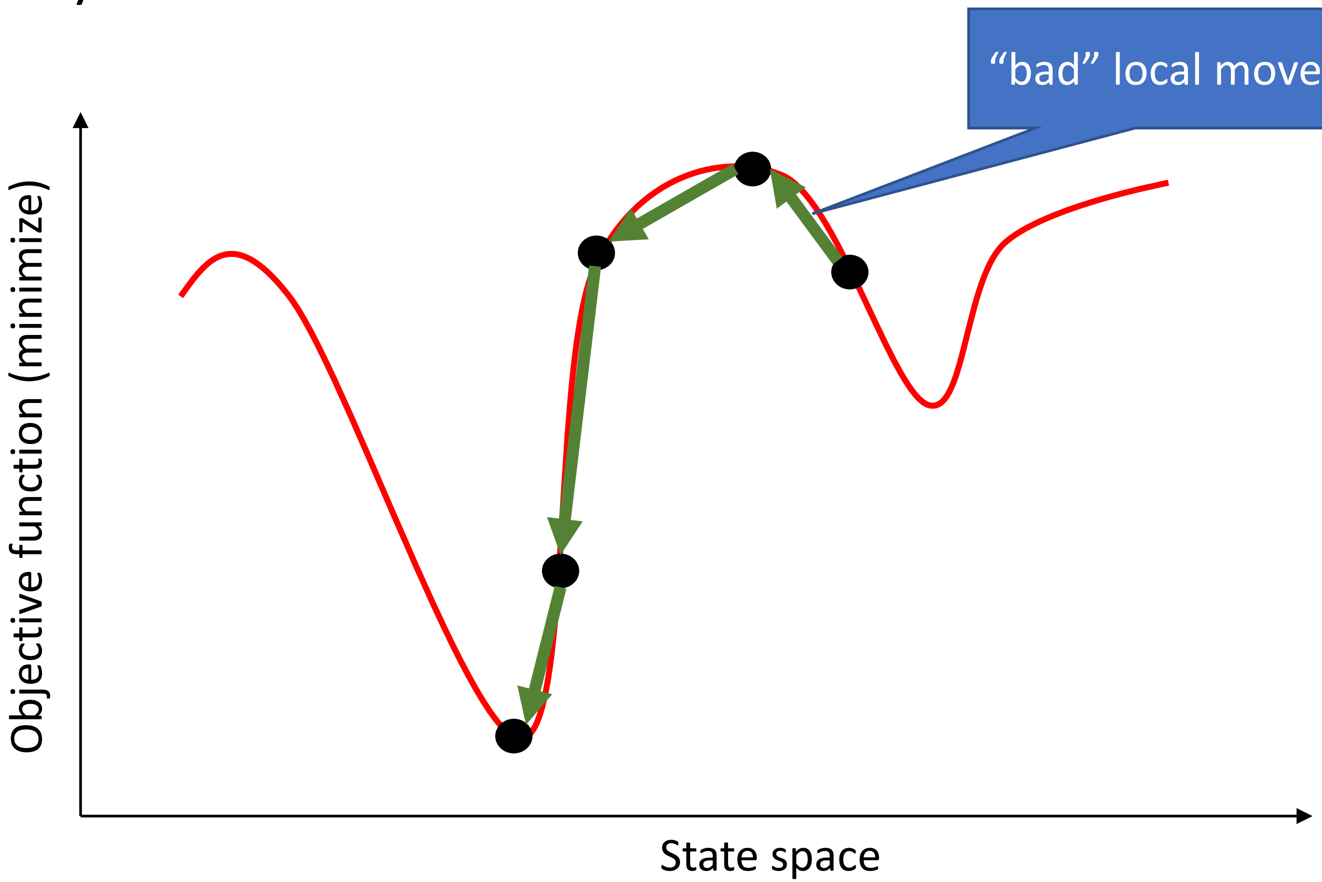


# Simulated Annealing

Use heat to escape local optima…

# Simulated Annealing

- **Idea**: First-choice stochastic hill climbing + escape local minima by **allowing some "bad" moves** but gradually decrease their frequency.
- Inspired by the process of controlled cooling of glass or metals by decreasing the temperature (here chance of accepting bad moves) gradually.



# Simulated Annealing

- **Idea**: First-choice stochastic hill climbing + escape local minima by allowing some "bad" moves but gradually decreasing their frequency as we get closer to the solution.
- The probability of accepting "bad" moves follows **an annealing schedule** that reduces the temperature $T$ over time $t$.

Maximization

**function** SIMULATED-ANNEALING(*problem*, *schedule*) **returns** a solution state
    *current* ← *problem*.INITIAL
    **for** $t$ = 1 **to** $\infty$ **do**
        $T$ ← *schedule*($t$)
        **if** $T$ = 0 **then return** *current*
        *next* ← a randomly selected successor of *current*
        $\Delta E$ ← VALUE(*current*) – VALUE(*next*)
        **if** $\Delta E < 0$ **then** *current* ← *next*
        **else** *current* ← *next* only with probability $e^{-\Delta E/T}$

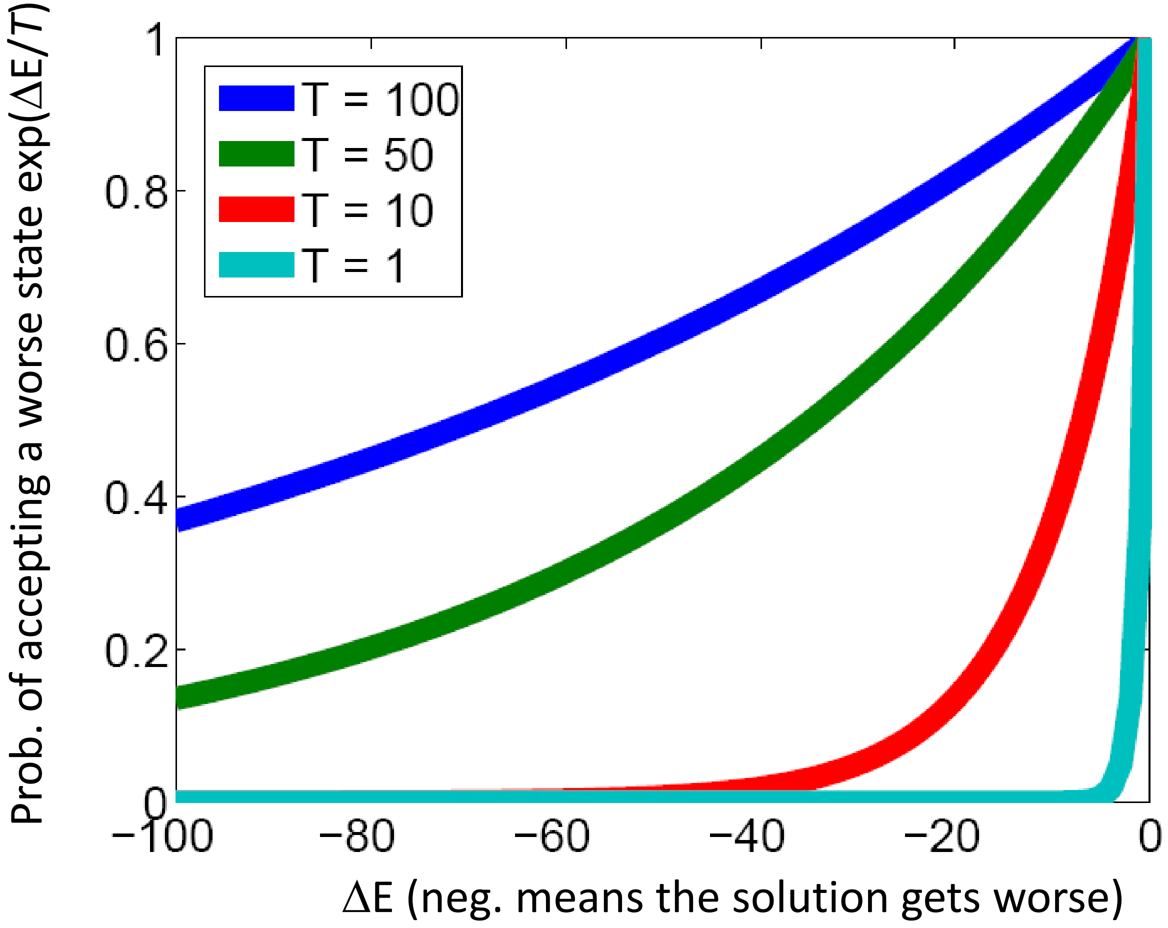
Typically, we start with a random state

Always accept good moves

Uses the Metropolis acceptance criterion to accept "bad" moves

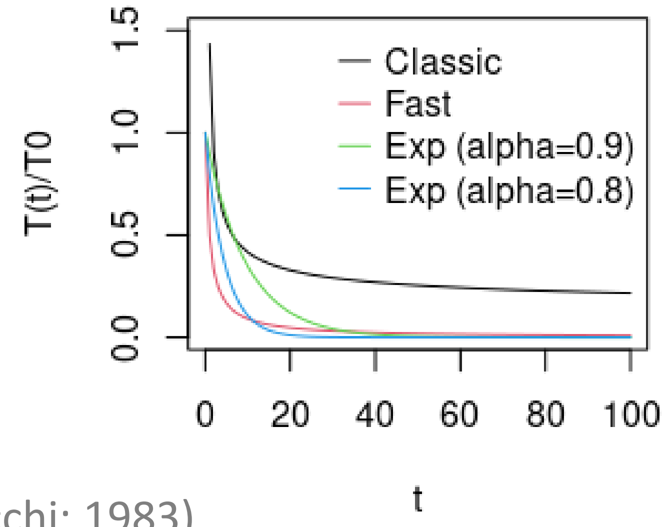
Note: Use $VALUE(next)$ – $VALUE(current)$ for minimization

# The Effect of Temperature




The lower the temperature, the less likely the algorithm will accept a worse state.

# Cooling Schedule




The cooling schedule is very important.
Popular schedules for the temperature at time $t$:

- **Classic simulated annealing:** $T_t = T_0 \frac{1}{\log(1+t)}$
- **Exponential cooling** (Kirkpatrick, Gelatt and Vecchi; 1983)

$$T_t = T_0 \alpha^t \quad \text{for} \quad 0.8 < \alpha < 1$$

- **Fast simulated annealing** (Szy and Hartley; 1987)

$$T_t = T_0 \frac{1}{1+t}$$

Notes:

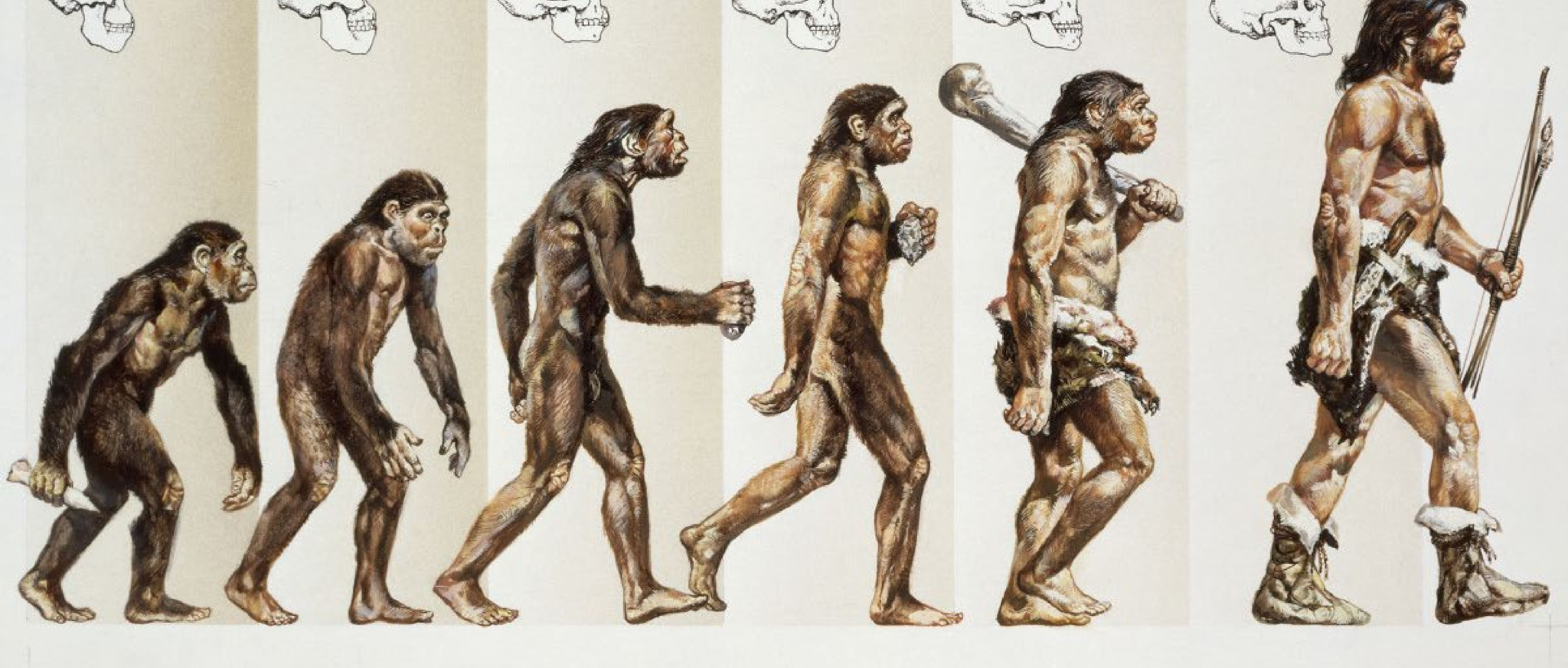
- The best schedule is typically determined by trial-and-error.
- Choose $T_0$ to provide a high probability that any move will be accepted at time $t = 0$.
- $T_t$ will not become 0 but very small. Stop when $T < \epsilon$ ($\epsilon$ is a very small constant).

# Simulated Annealing Search

**Guarantee:** If temperature decreases **slowly enough**, then simulated annealing search will find a global optimum with probability approaching one.

However:

- This usually takes impractically long.



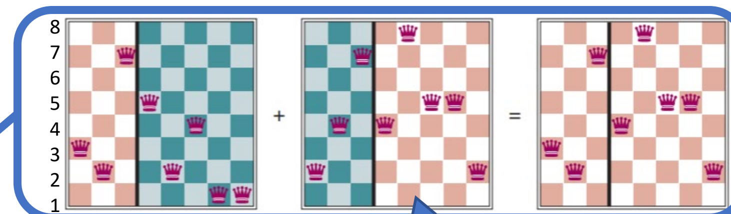
# Evolutionary Algorithms

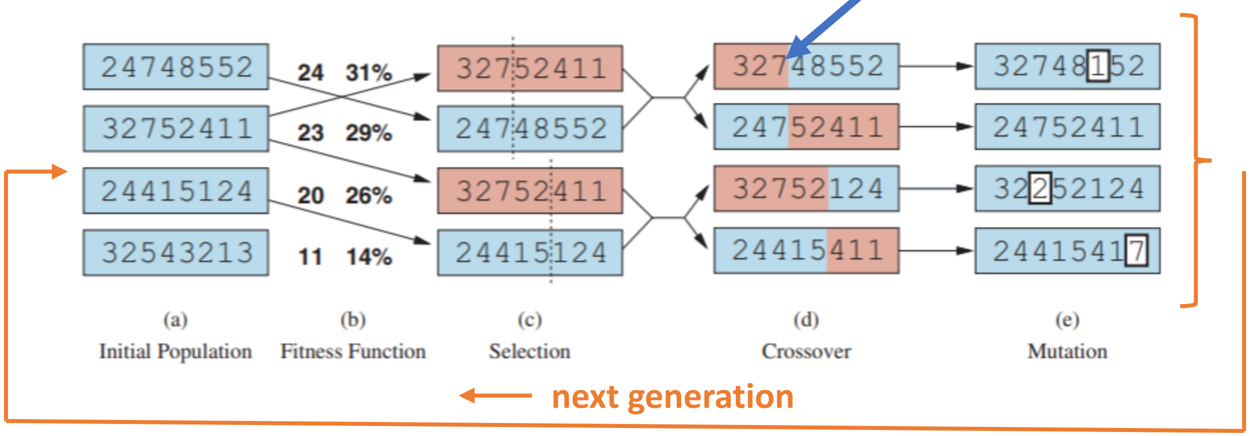
A Population-based Metaheuristics

# Evolutionary Algorithms / Genetic Algorithms

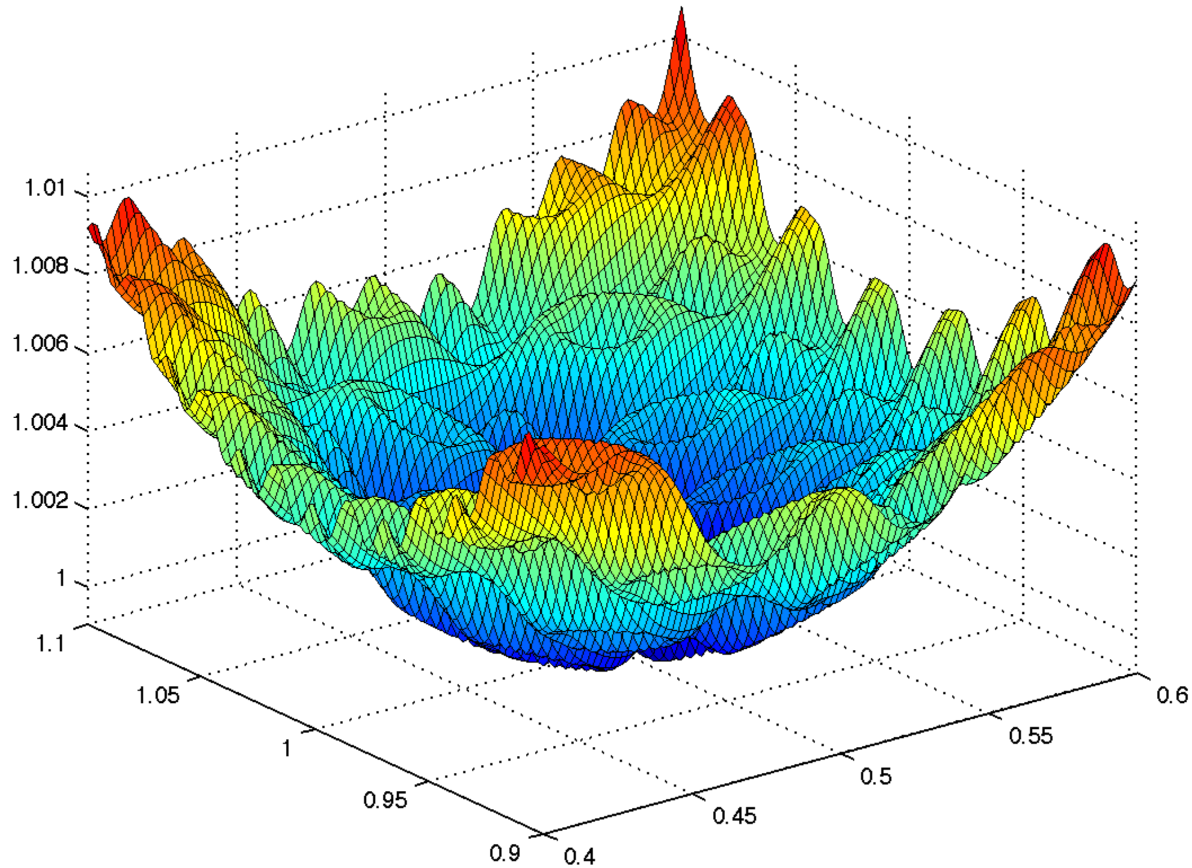
- A metaheuristic for **population-**based optimization.
- Uses mechanisms inspired by biological evolution (genetics):
  - Reproduction: Random selection with probability based on a **fitness** function.
  - Random recombination (crossover)
  - Random mutation
  - Repeated for many generations

- Example: 8-queens problem



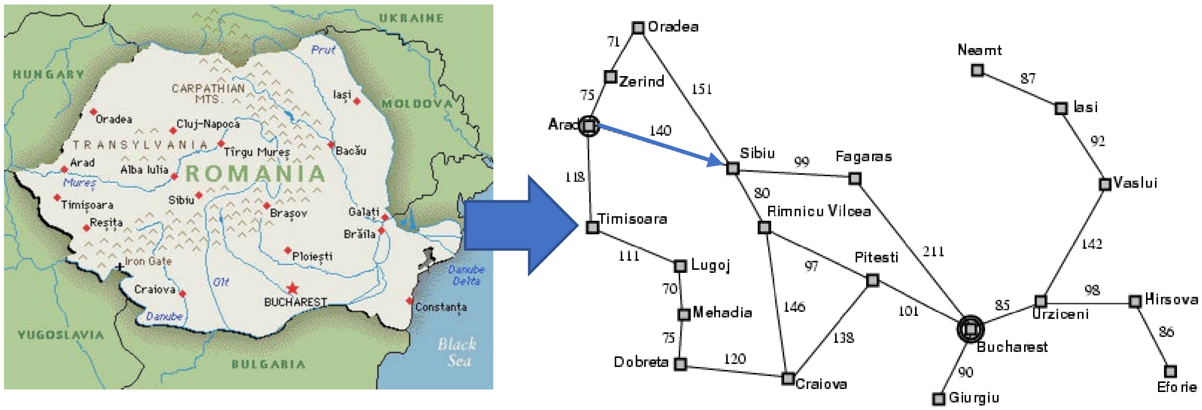

Individual = state **representation** as a chromosome: row of the queen in each column




# Search in Continuous Spaces

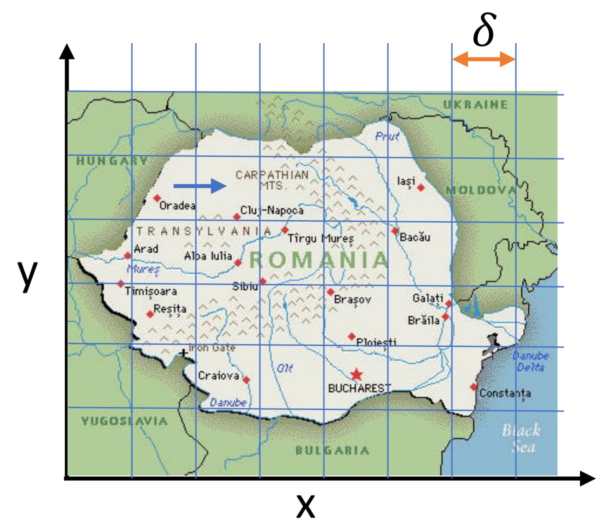
# Discretization of Continuous Space

- Use atomic states and create a graph as the transition function.



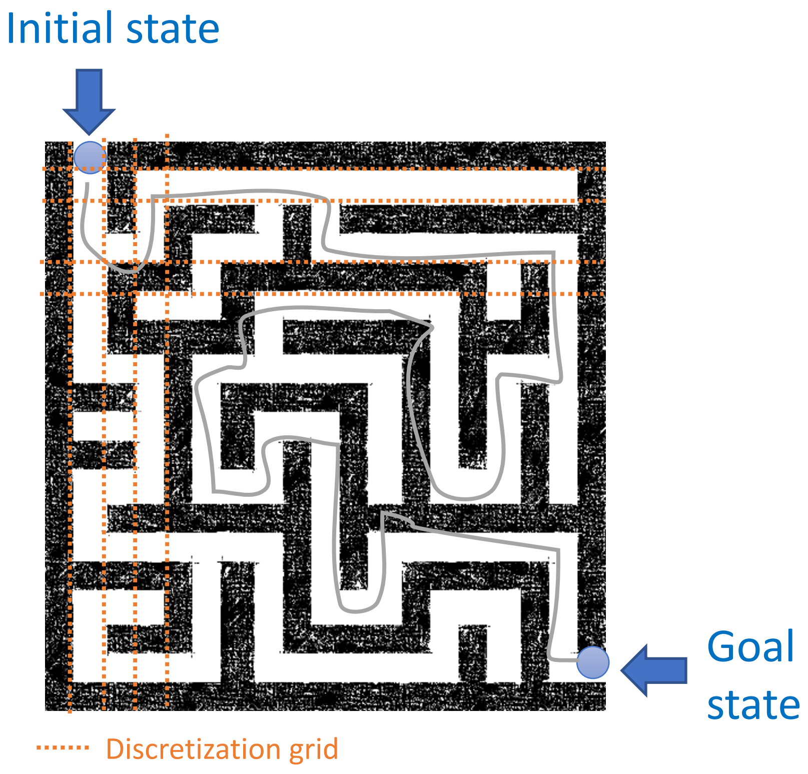

- Use a grid with spacing of size $\delta$

  Note: You probably need a way finer grid!

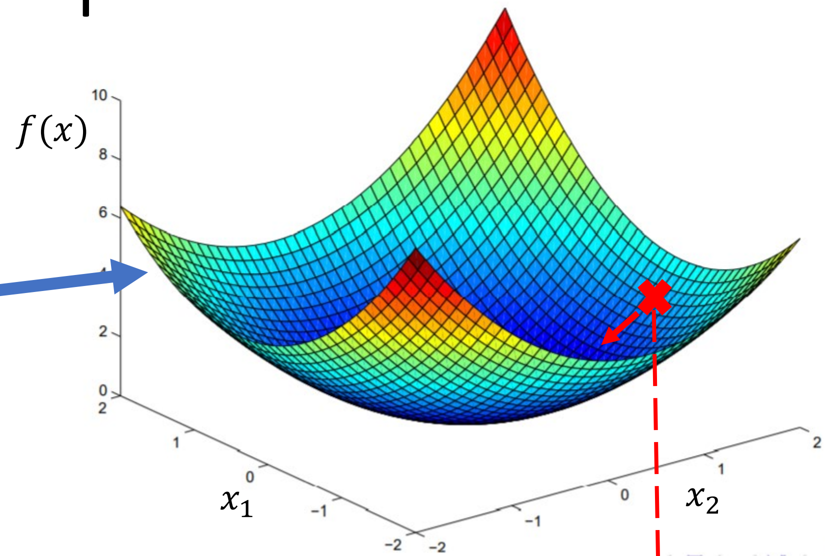


# Discretization of Continuous Space

## How did we discretize this space?



# Search in Continuous Spaces: Gradient Descent




**State space**: infinite

**State representation**: $\boldsymbol{x} = (x_1, x_2, \dots, x_k)$

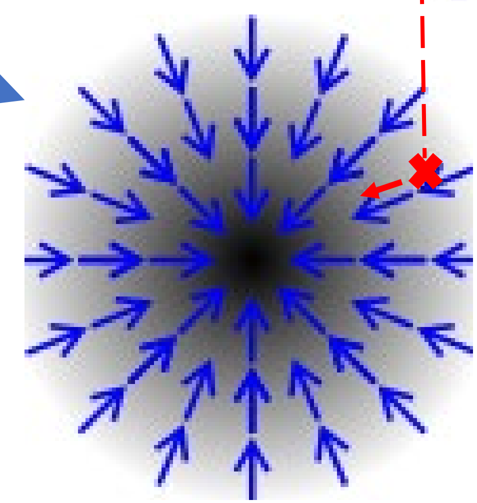
**Objective function**: min $f(\boldsymbol{x}) = f(x_1, x_2, \dots, x_k)$

**Local neighborhood**: small changes in $x_1, x_2, \dots, x_k$

Gradient at point $\boldsymbol{x}$: $\nabla f(\boldsymbol{x}) = \left(\frac{\partial f(\boldsymbol{x})}{\partial x_1}, \frac{\partial f(\boldsymbol{x})}{\partial x_2}, \dots, \frac{\partial f(\boldsymbol{x})}{\partial x_k}\right)$

(=evaluation of the Jacobian matrix at $\boldsymbol{x}$)

Find optimum by solving: $\nabla f(\boldsymbol{x}) = 0$



- **Gradient descent (= Steepest-ascend hill climbing for minimization)** with step size $\alpha$

  Repeat: $\boldsymbol{x} \leftarrow \boldsymbol{x} - \alpha \nabla f(\boldsymbol{x})$

- **Newton-Raphson method** uses the inverse of the Hessian matrix (second-order partial derivative of $f(\boldsymbol{x})$) $H_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}$ as the optimal step size

  Repeat: $\boldsymbol{x} \leftarrow \boldsymbol{x} - \boldsymbol{H}_f^{-1}(\boldsymbol{x}) \nabla f(\boldsymbol{x})$

Note: May get stuck in a local optimum if the search space is non-convex! Use simulated annealing, momentum or other methods to escape local optima.

# Search in Continuous Spaces: Stochastic Gradient Descent

- What if a complete mathematical formulation of the objective function over is not known?
- We may have objective values at fixed points, called the **training data**.
- In this case, we can perform gradient descent on an approximation of the gradient using the data points. This is called **stochastic gradient descent (SGD).**

→ We will talk more about search in continuous spaces with loss functions using gradient descend when we talk about **parameter learning for machine learning.**