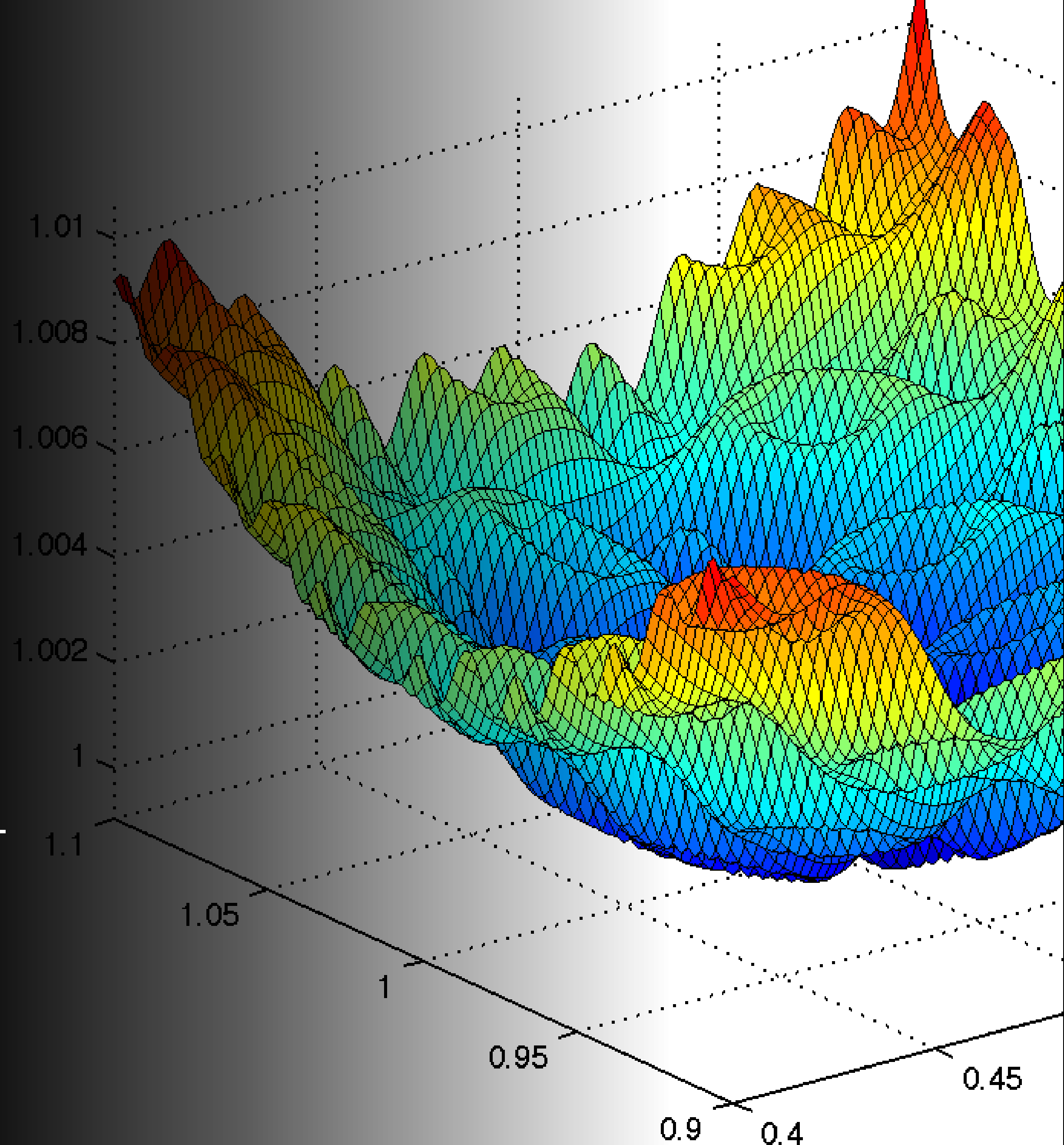
CS 5/7320
Artificial Intelligence

Local Search
AIMA Chapters 4.1 & 4.2



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

Contents

- What is Local Search?
- Hill-Climbing Search
- Simulated Annealing
- Evolutionary Algorithms
- Search in Continuous Spaces

Recap: Uninformed and Informed Search

Tries to **plan** the

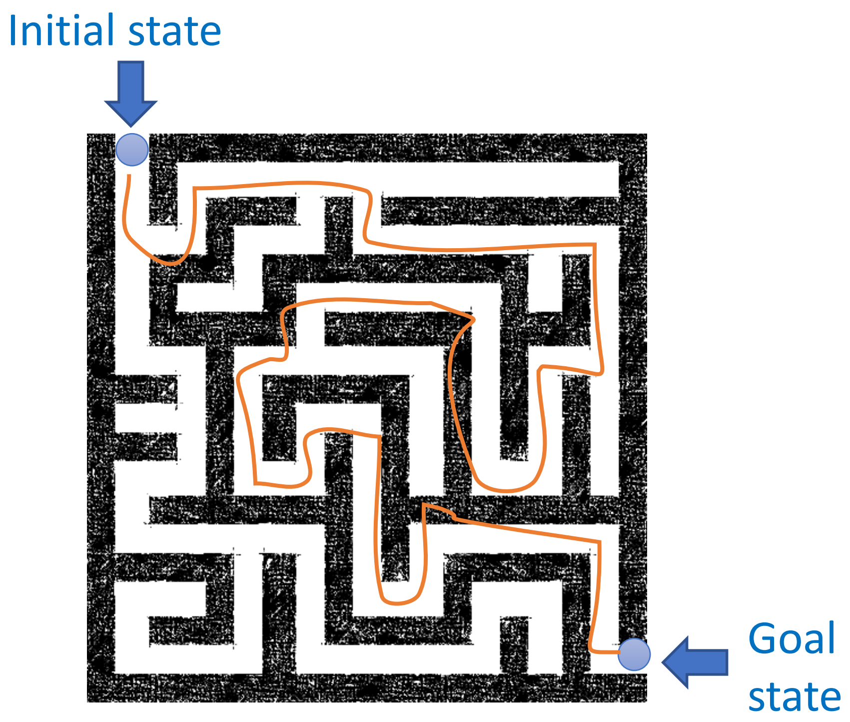
best path

from a

given initial state

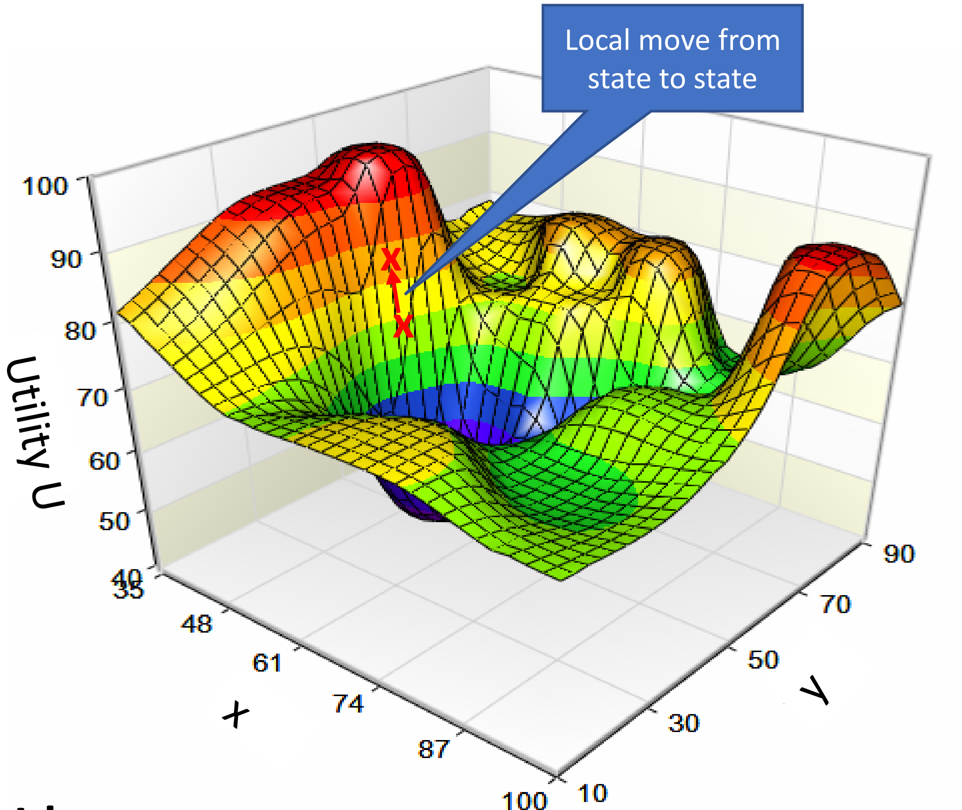
to a

given goal state.



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

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**).

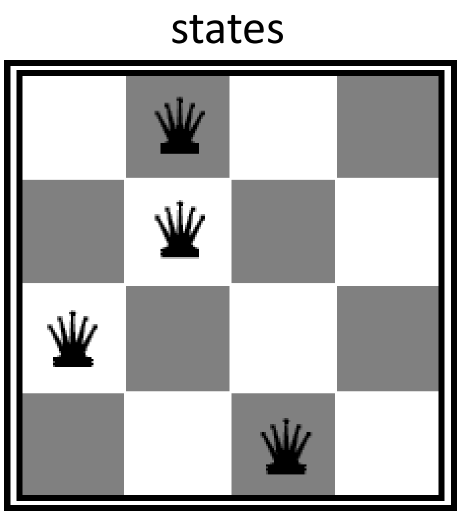
Local Search Algorithms

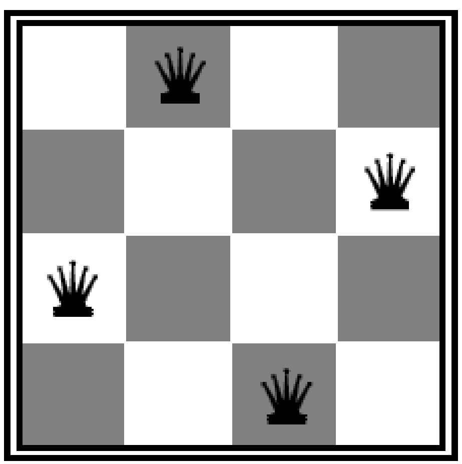
Difference to search from the previous chapter:

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 a neighboring state with higher utility. 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., stochastic gradient descend to train neural networks learns to approximate a function by using the prediction error as the objective function.





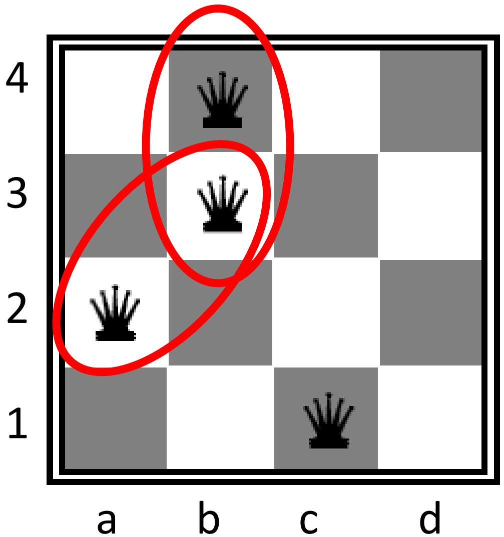
Example: *n*-Queens Problem

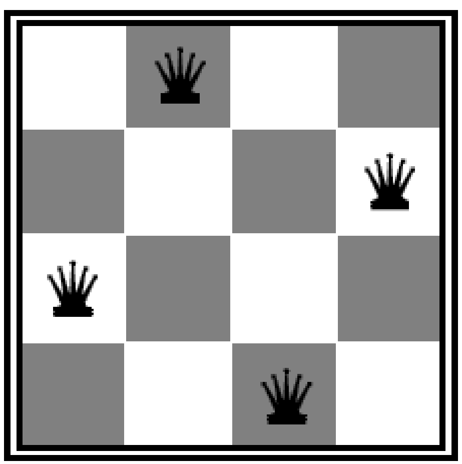
Goal: Put n queens on an $n \times n$ board with no two queens on the same row, column, or diagonal.

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?

2 conflicts = utility of -2





0 conflicts = utility of 0

Example: *n*-Queens Problem

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 factored 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.

Has its optimum at the goal state. Similar to a heuristic in A* search.

- **Local neighborhood:** What states are close to each other?
  Move a single queen.

Defines a transition function.

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.

MotorCycleUSA.com

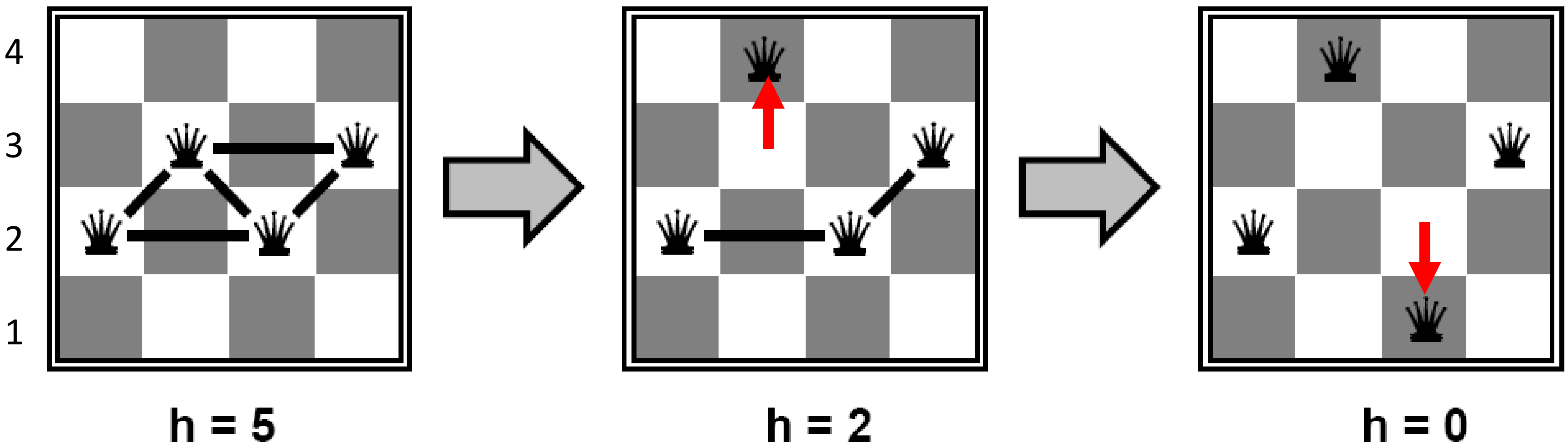
Example: *n*-Queens Problem

- **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

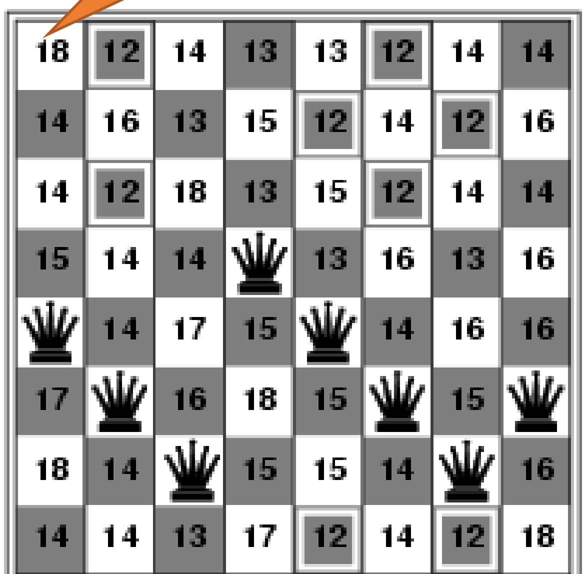
- Find a local neighboring state (move one queen within its column) to reduce conflicts



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 in column 1 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!

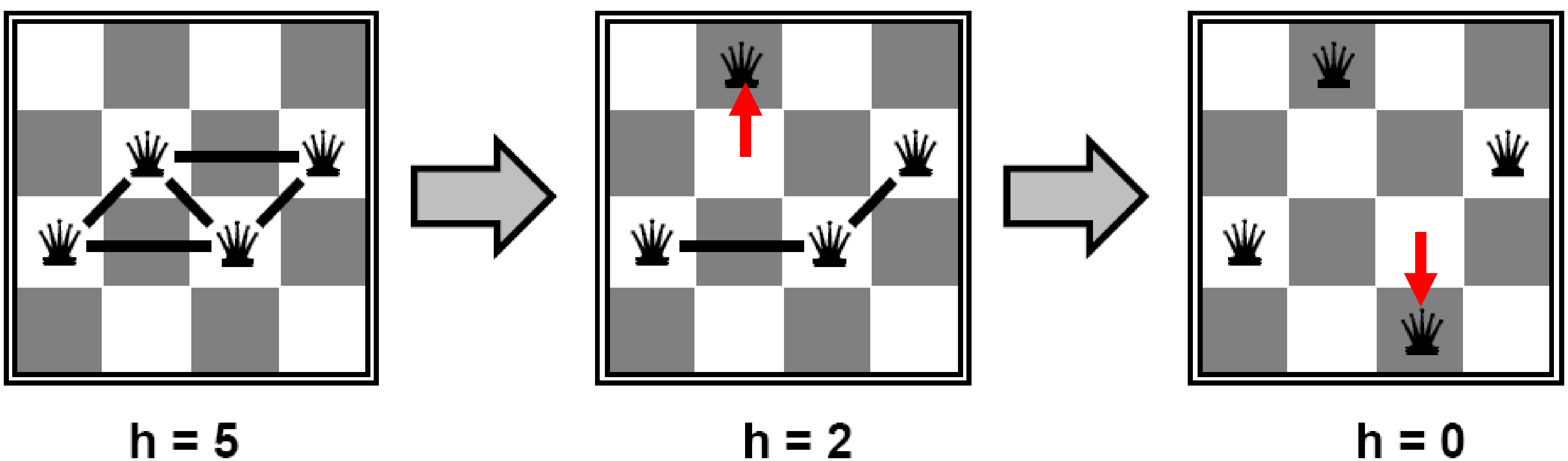
Example: *n*-Queens Problem

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

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

subject to: s has one queen per column

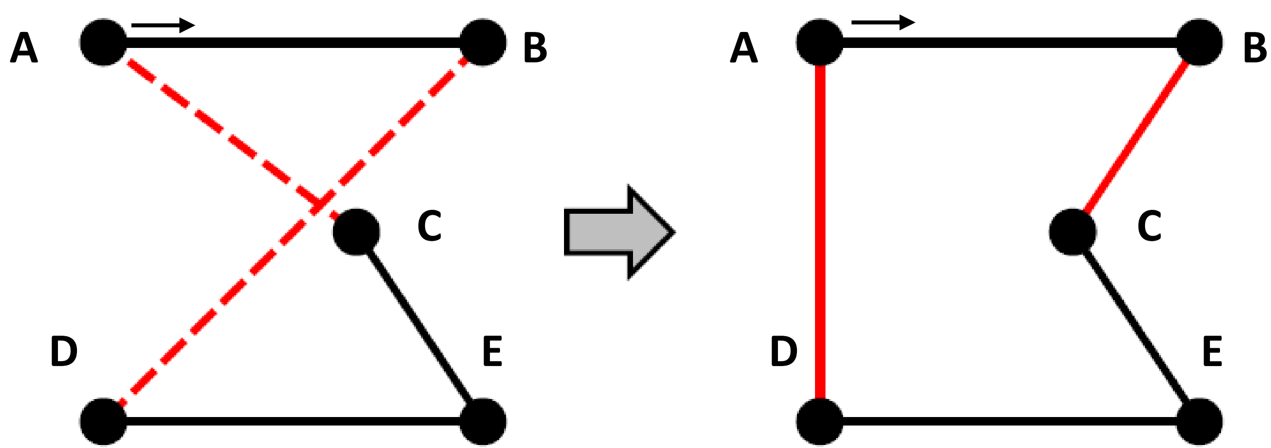
Remember: This makes the problem a lot easier.



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. There are n! Many permutations.
- **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 (permutation): **ABDEC** **ABCED**

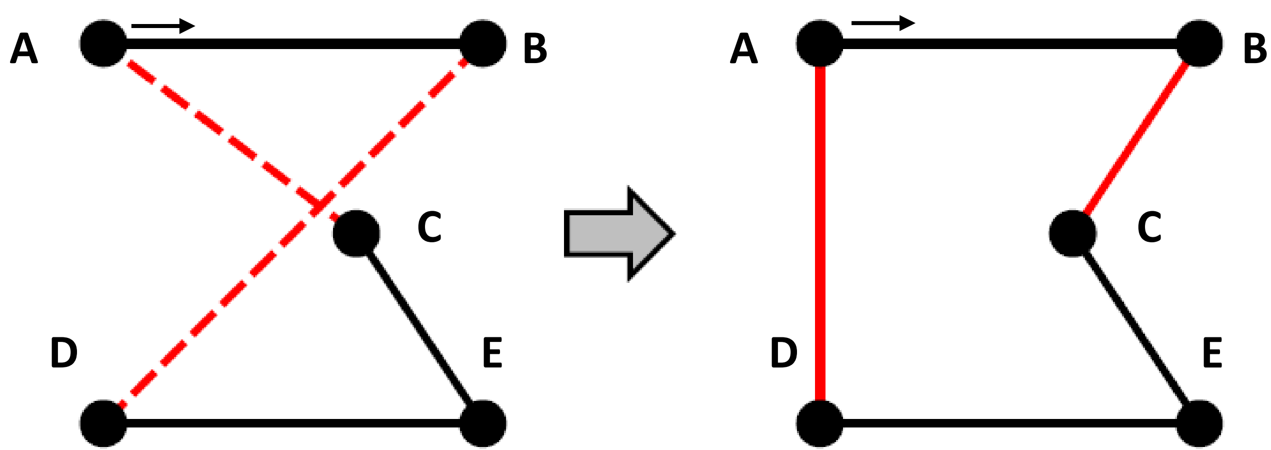
Example: Traveling Salesman Problem

Formulation as an optimization problem:
Find the best tour π

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

s.t. π 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

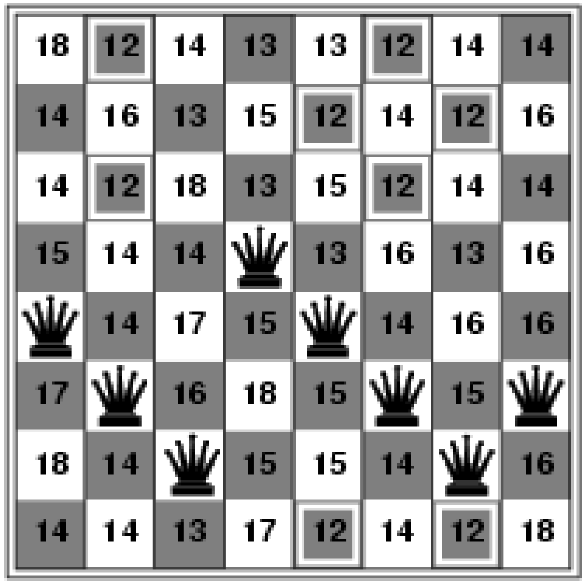
Variants:

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 and only move to better ones = first-choice hill climbing – **the most popular variant**, this is what people often mean when they say "stochastic hill climbing"

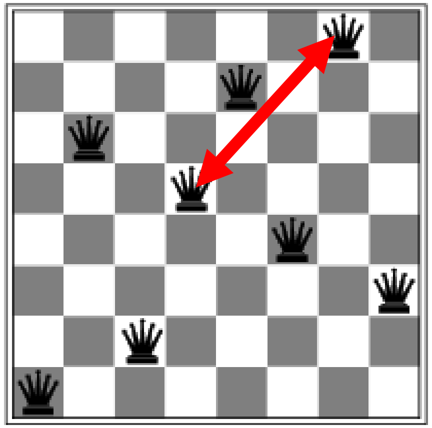


Local Optima

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



$h = 1$

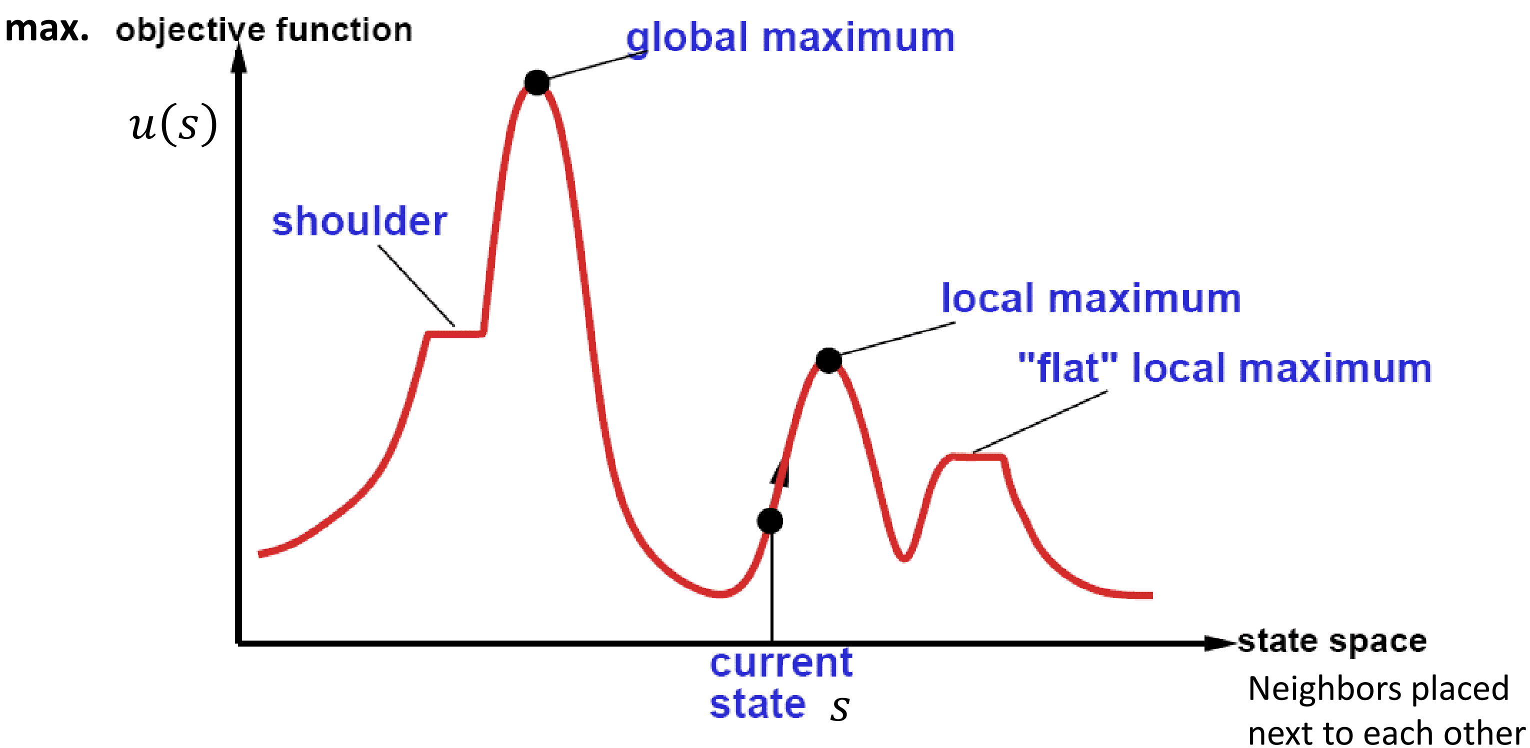
Example: local optimum for the 8-queens problem. No single queen can be moved within its column to improve the objective function.

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 $u(s)$.



How to escape local maxima?

→ Random restart hill-climbing can help.

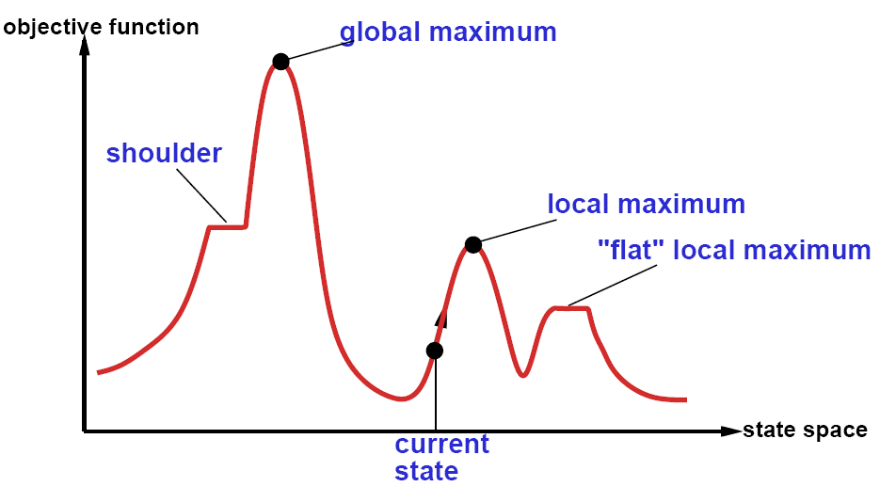
What about "shoulders" (called "ridges" in higher dimensional space)?

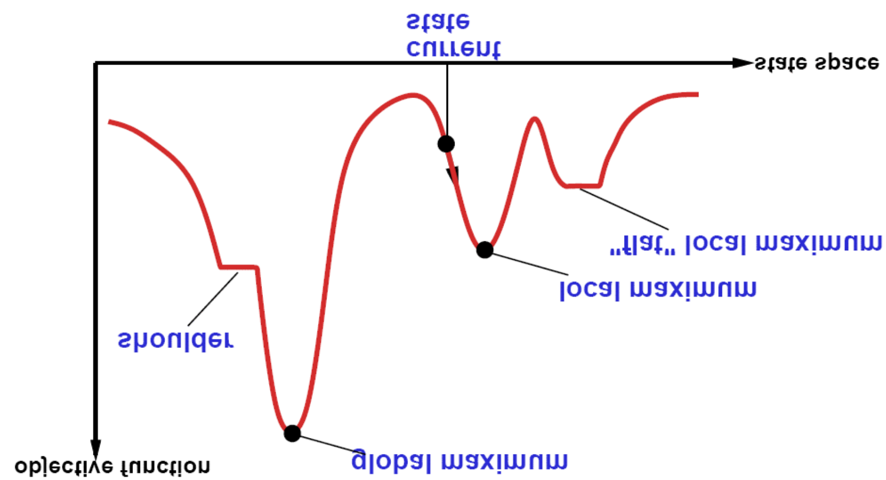
→ Hill-climbing that allows sideways moves and uses momentum.

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:

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

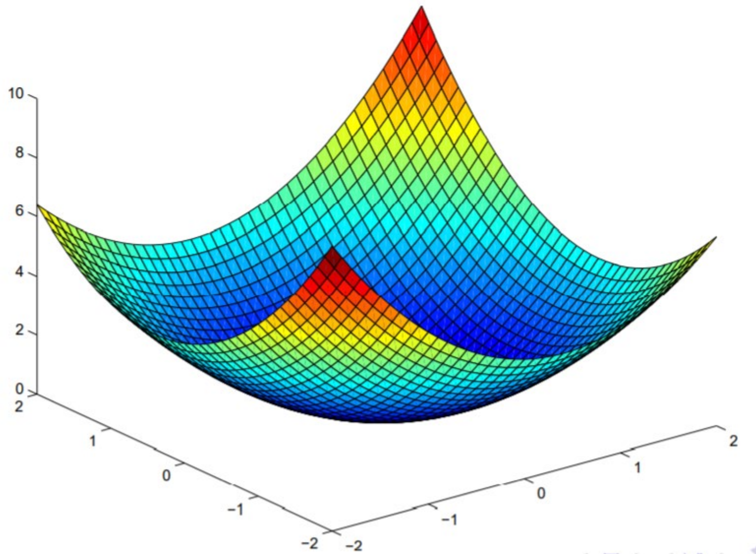




Convex vs. Non-Convex Optimization Problems

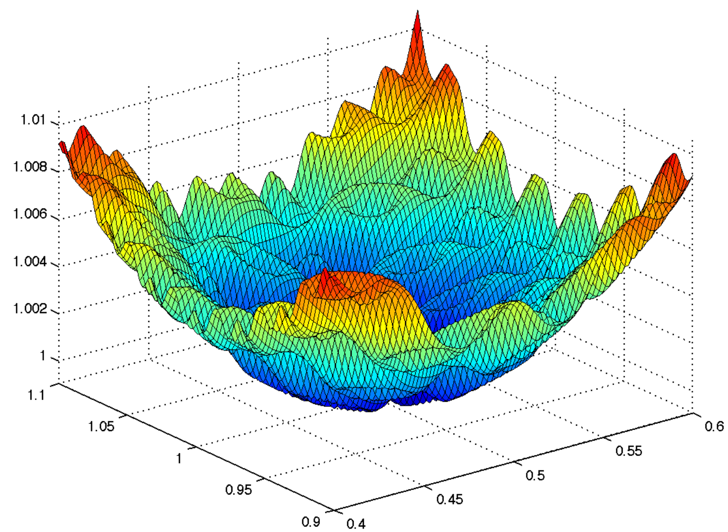
Minimization problems

Convex Problem



One global optimum + smooth function → calculus makes it easy

Non-convex Problem

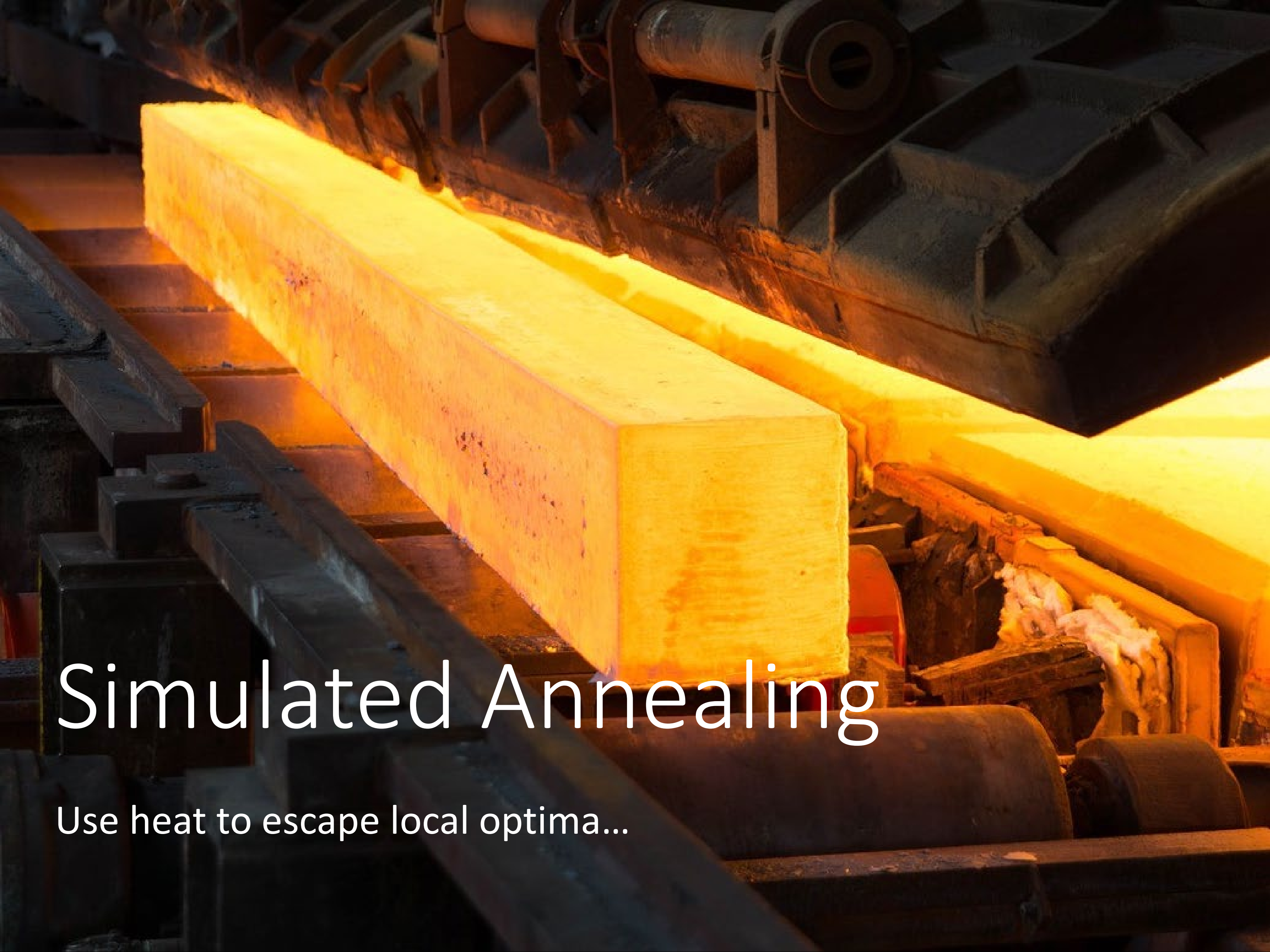


Many local optima → hard

Many discrete optimization problems are like this.

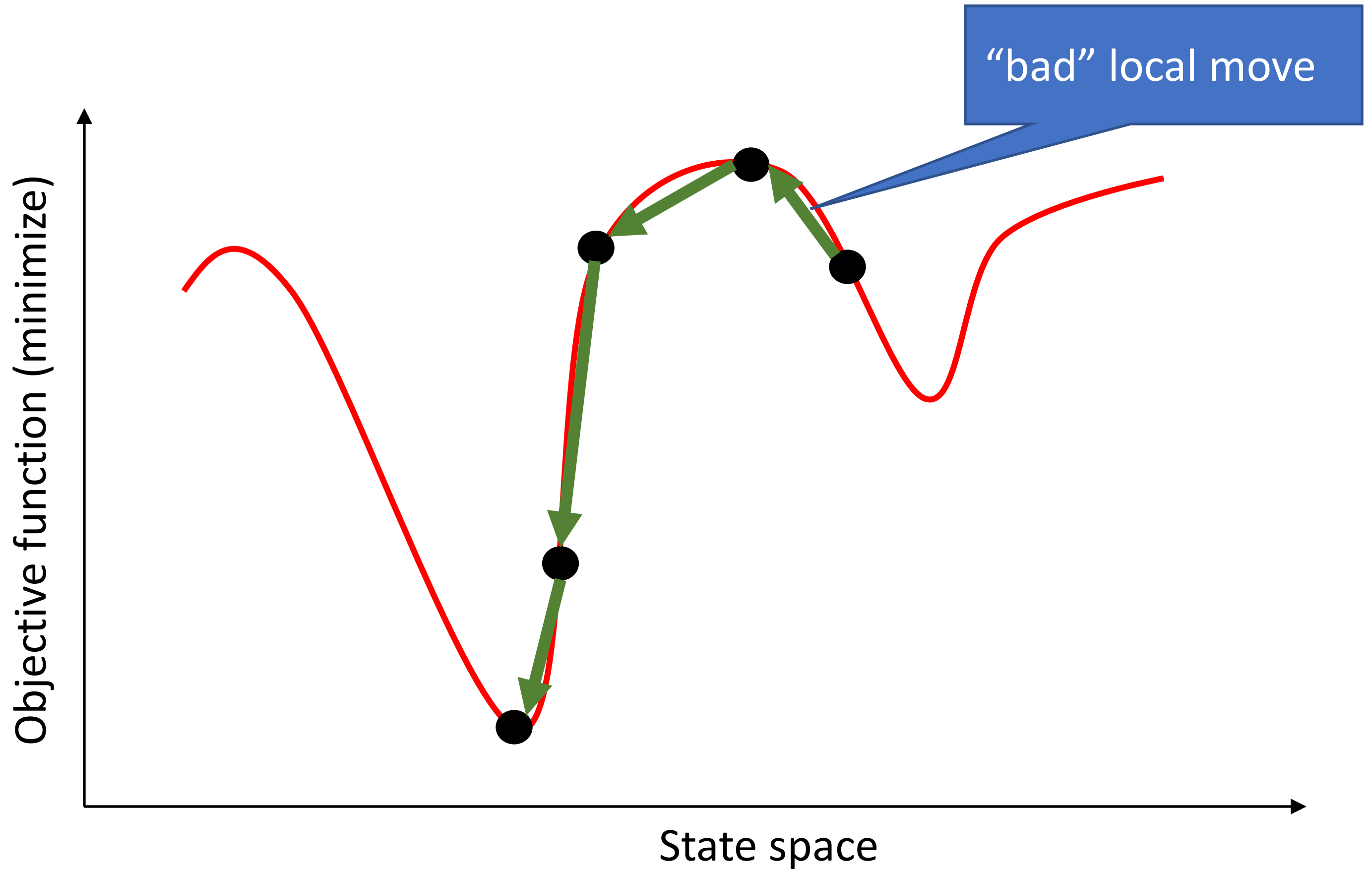
Simulated Annealing

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.
- Annealing tries to reach a low energy state so a negative ΔE means the solution gets better.
- The probability of accepting “bad” moves follows the **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** ∞ **do**
    $T \leftarrow schedule(t)$
    if $T = 0$ **then return** *current*
    next ← a randomly selected successor of *current*
    $\Delta E \leftarrow$ 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

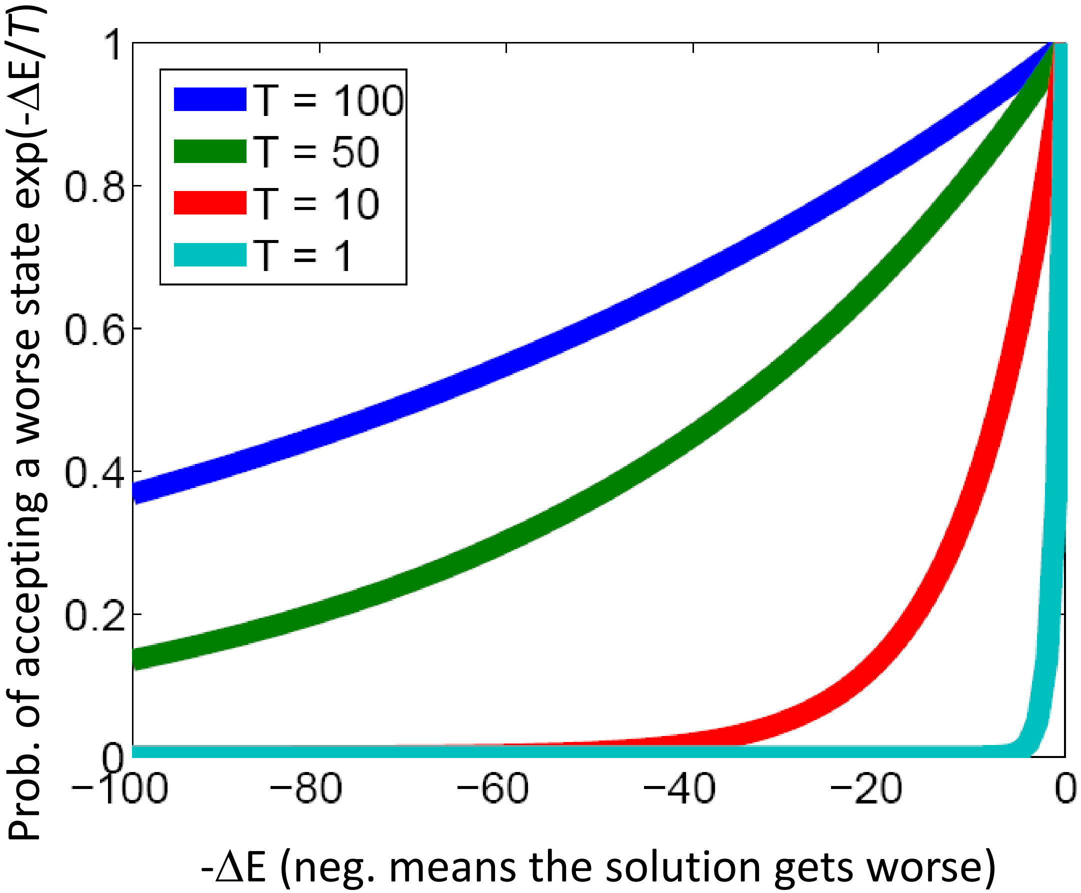
Always accept good moves that reduce the energy.

Accept “bad” moves with a probability inspired by the acceptance criterion in the Metropolis–Hastings MCMC algorithm.

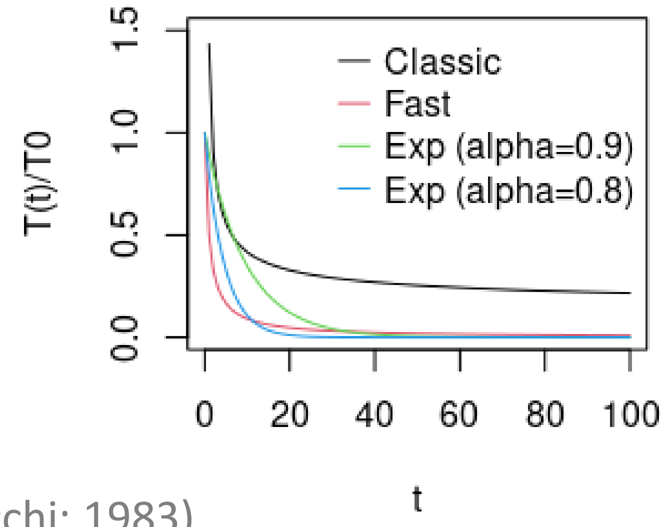
Note: Use $VALUE(next)$ – $VALUE(current)$ for minimization

The Effect of Temperature

Convert the changes due to “bad” moves into an acceptance probability depending on the temperature. The criterion uses the negative part of the exponential function.



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:

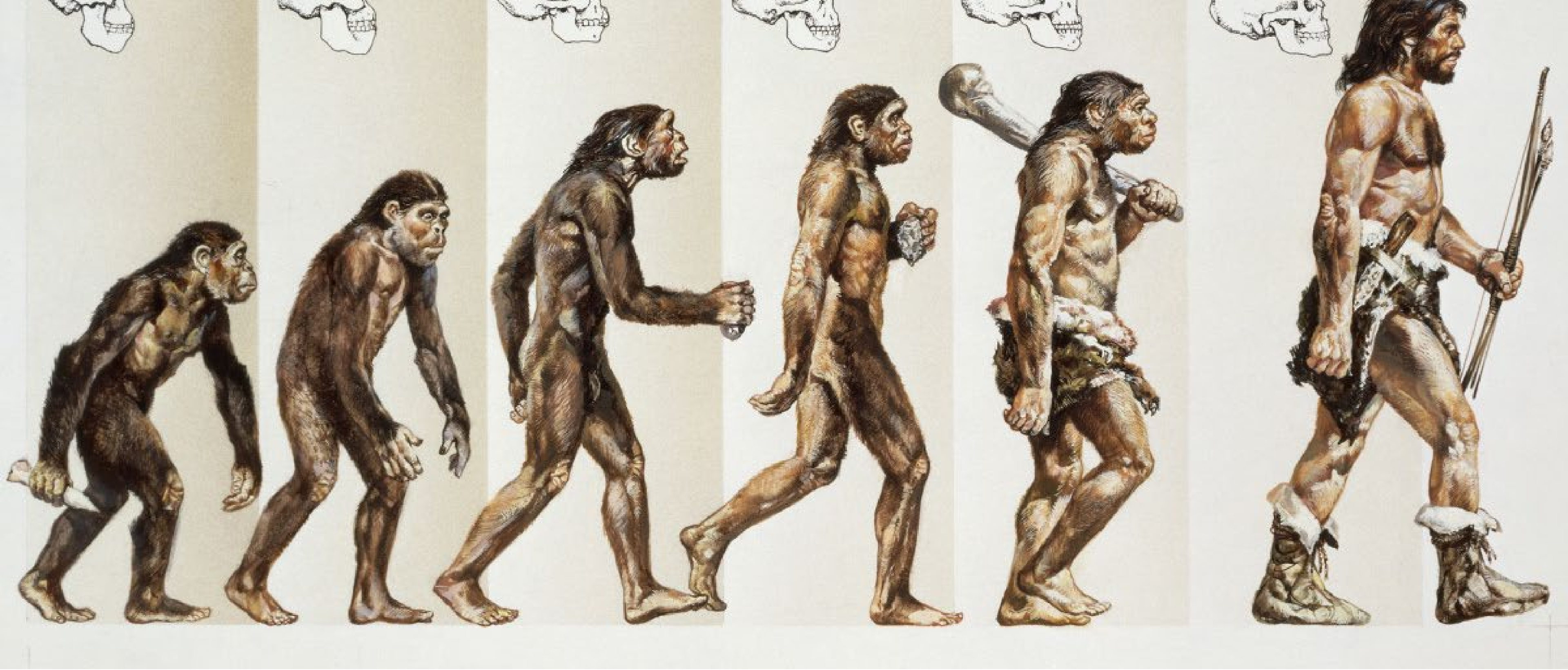
- Choose T_0 to provide a high probability $p_0 = e^{-\frac{\Delta E}{T_0}}$ that any move will be accepted at time $t = 0$. ΔE is determined by the worst possible move.
- T_t will not become 0 but very small. Stop when $T < \epsilon$ (ϵ is a very small constant).
- The best schedule (cooling rate) is typically determined by trial-and-error. The goal is to have a low chance of getting stuck in a local optima.

Simulated Annealing Search

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

However:

- This usually takes impractically long.
- We need to experiment with the cooling schedule to find one that typically avoids local optima.

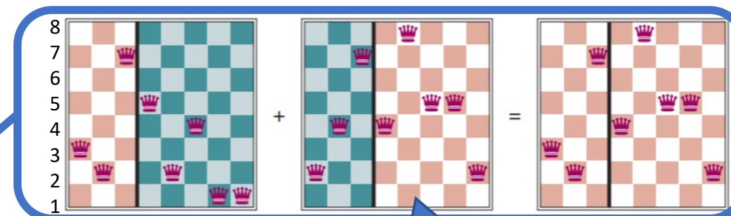


Evolutionary Algorithms

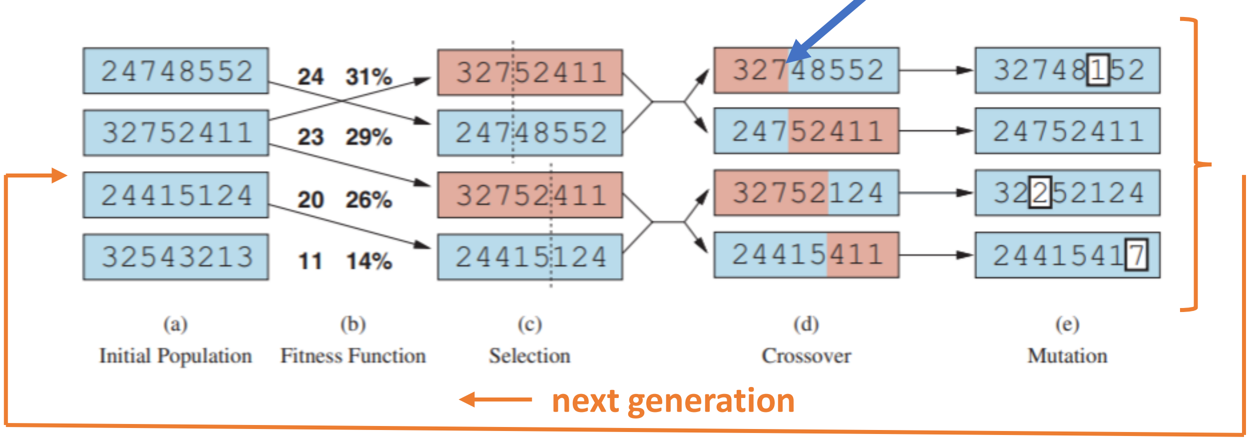
A Population-based Metaheuristics

Evolutionary Algorithms / Genetic Algorithms

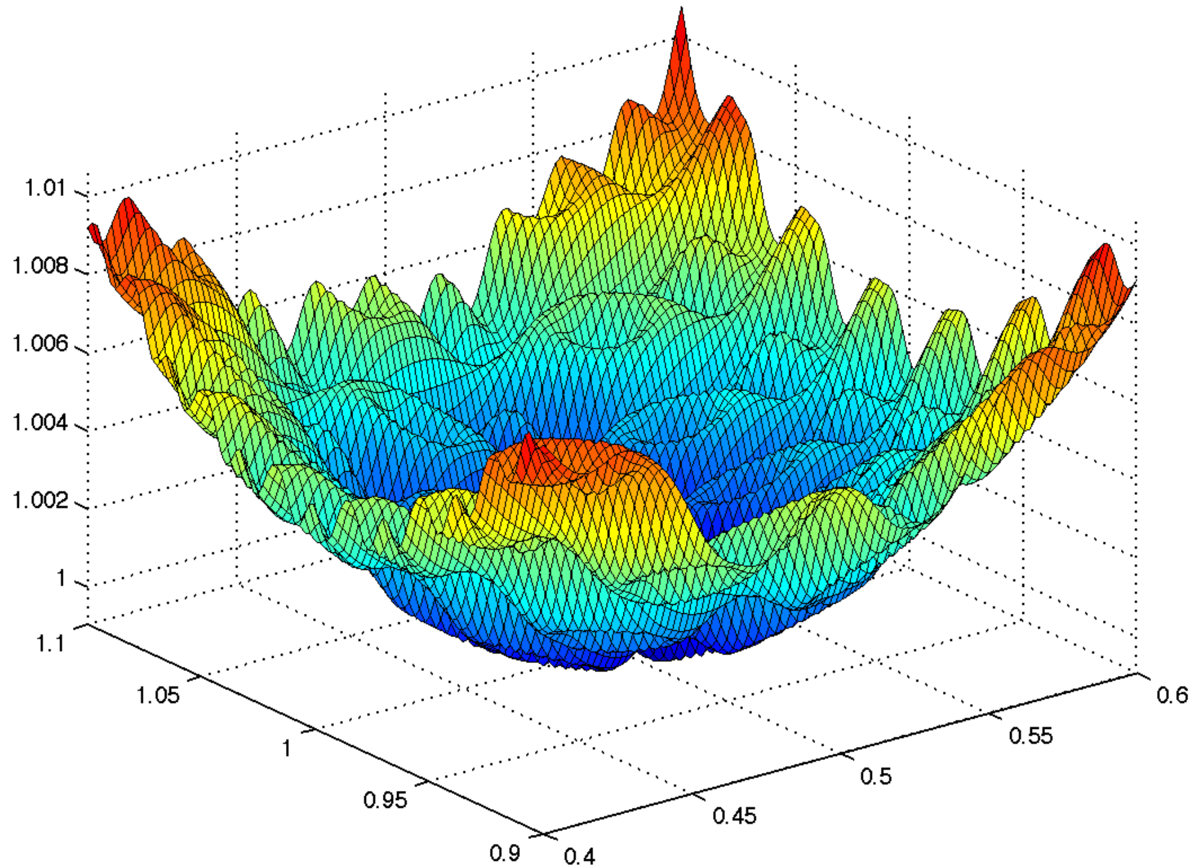
- A metaheuristic for **population-**based optimization.
- Uses mechanisms inspired by biological evolution (genetics):
  - Reproduction: Random selection with probability based on a **fitness** function.
  - Random recombination (crossover)
  - Random mutation
  - Repeated for many generations



- Example: 8-queens problem



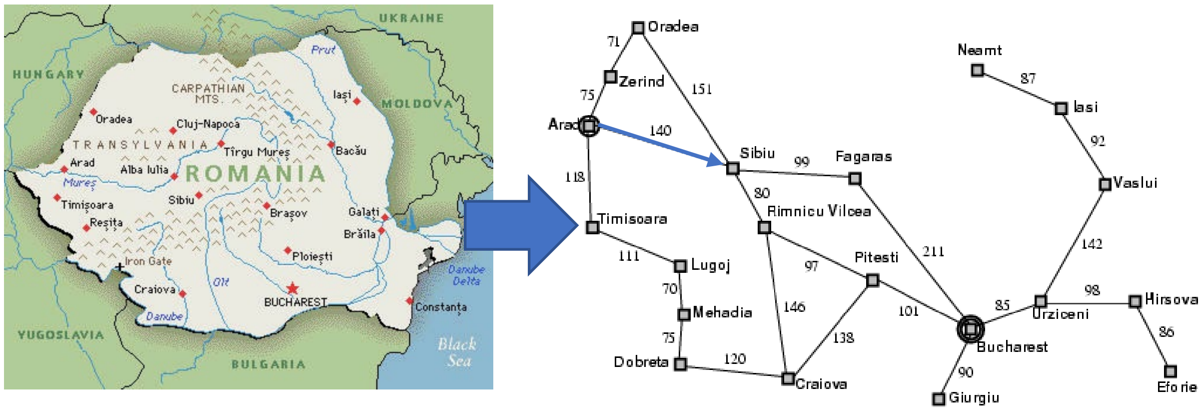
Individual = state **representation** as a chromosome: row of the queen in each column



Search in Continuous Spaces

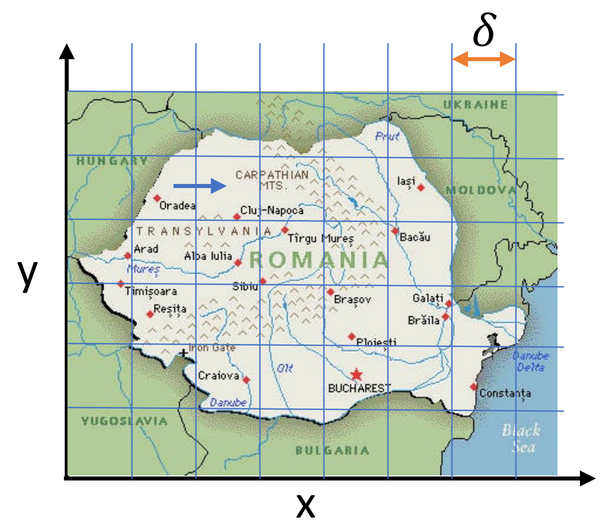
Discretization of Continuous Space

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



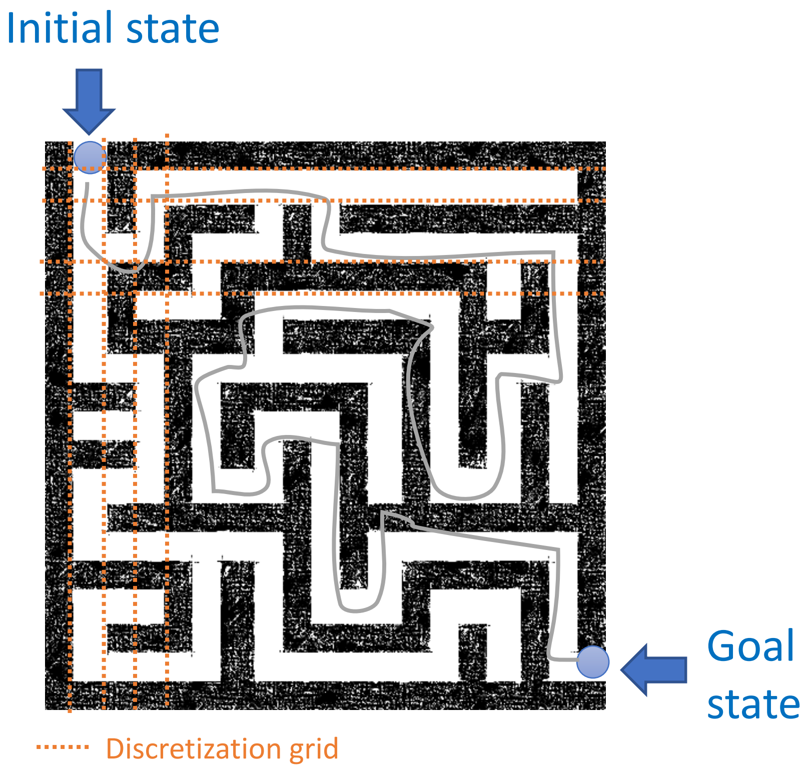
- Use a grid with spacing of size δ

  Note: You probably need a way finer grid!

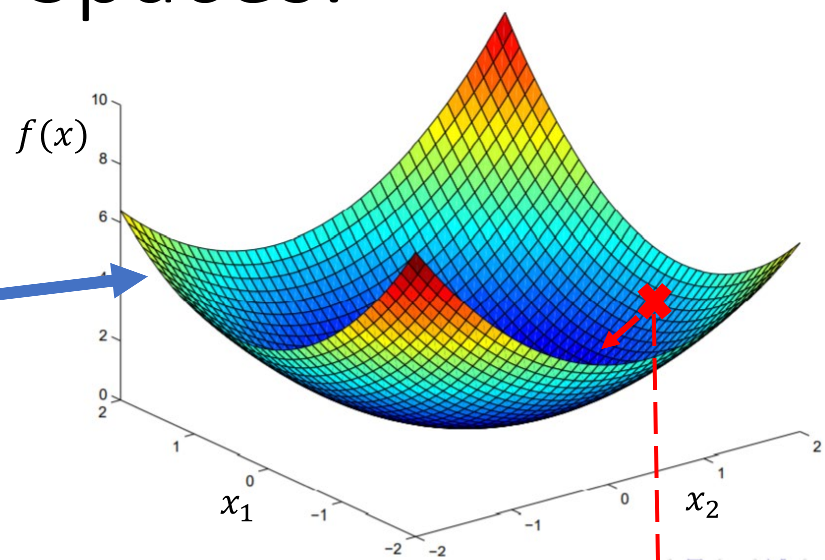


Discretization of Continuous Space

How did we discretize this space?



Search in Continuous Spaces: Gradient Descent



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

State space size: infinite

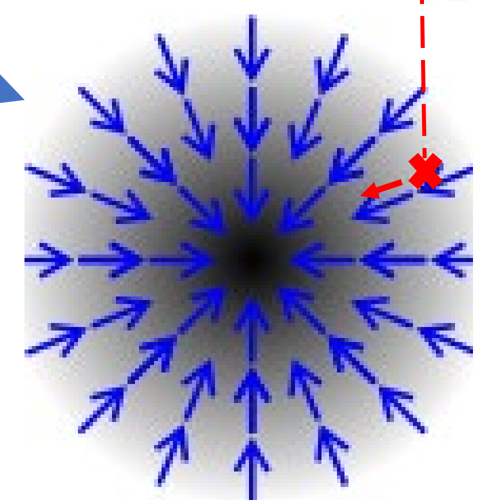
Objective function: min $f(\boldsymbol{x}) = f(x_1, x_2, \dots, x_k)$

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 α (typically reduced over time)

  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 the mathematical formulation of the objective function is not known?
- We may have objective values at fixed points, called the **training data**.
- In this case, we can perform gradient descent with 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 learning from examples (machine learning).**

Conclusion

- Local search provides a fast method to find good solutions to many difficult optimization problems.
- Local optima are a big issue that can be addressed with random restarts and simulated annealing.