

# **Data Science 2**

## **Meta-heuristieken (deel 1)**

---

# Agenda

1. Inleiding – Optimalisatieproblemen
2. Algoritme versus heuristiek
  - ↳ Computationale complexiteit
3. Soorten heuristieken
4. Simulated annealing

---

# **Inleiding - Optimalisatieproblemen**



# Inleiding - Optimalisatieproblemen

---

Problemen –in bedrijfscontext-

- steken dagelijks de kop op
- toenemende complexiteit

Probleem oplossen

= Beslissing nemen

= Keuze maken uit verschillende alternatieven

**Doel: beste resultaat**, rekeninghoudend met **bestaande beperkingen** (geld, tijd, beschikbare mensen, aanwezige kennis, grondstoffen, wetgeving,...)

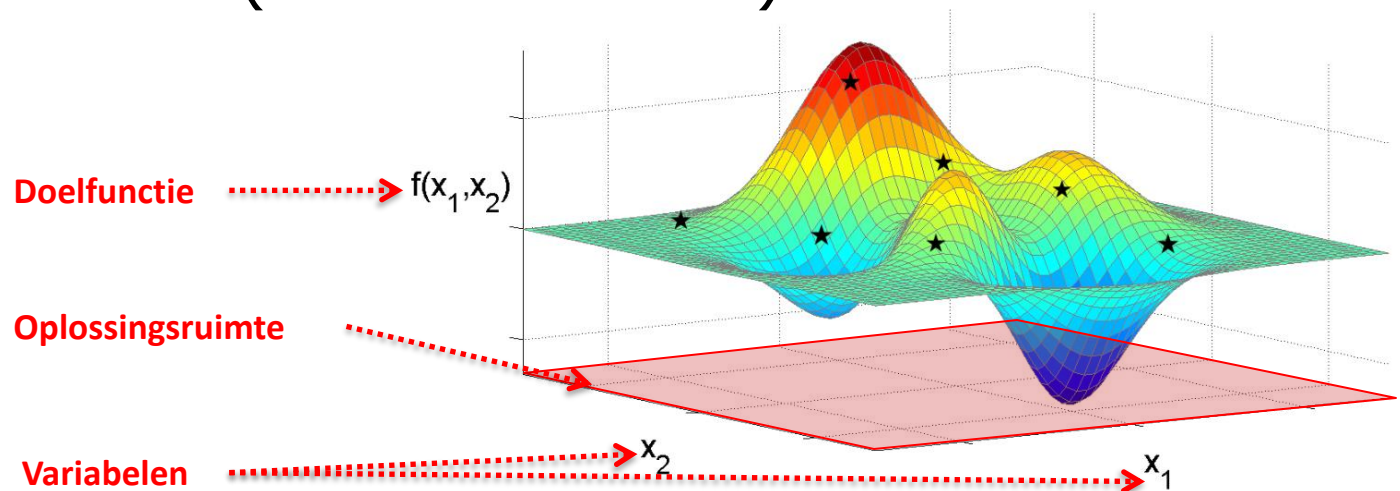


# Inleiding - Optimalisatieproblemen

Optimalisatieprobleem omvat:

- **Variabelen**
- Omschrijving van de verzameling van alle mogelijke **oplossingen**(\*)
- Lijst van beperkingen: **constraints**
- Een te maximaliseren of te minimaliseren **doelfunctie** (of **kostfunctie**)

**oplossings-  
ruimte**

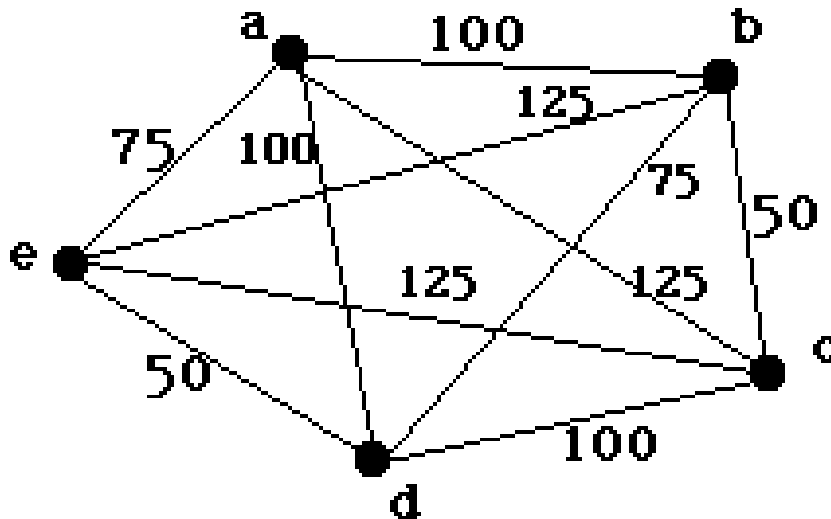


(\*) een oplossing = toekennen van concrete waarden aan de variabelen

# Inleiding - Optimalisatieproblemen

## Voorbeeld: Handelsreizigersprobleem (Traveling Salesman Problem of TSP)

Als er  $n$  steden gegeven zijn die een **handelsreiziger** moet bezoeken, samen met de afstand tussen ieder paar van deze steden, **zoek** dan **de kortste weg** die kan worden gebruikt, waarbij iedere stad eenmaal wordt bezocht.



	a	b	c	d	e
a	0	100	125	100	75
b	100	0	50	75	100
c	125	50	0	100	125
d	100	75	100	0	50
e	75	100	125	50	0

*Hoe ziet een oplossing eruit?*

Opeenvolging van de steden : (a,e,d,b,c)

*Waarde doelfunctie?*

$$75 + 50 + 75 + 50 + 125 = 375$$

*Aantal mogelijke oplossingen?*

$$5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$$

(alle mogelijke permutaties, algemeen:  $n!$ )

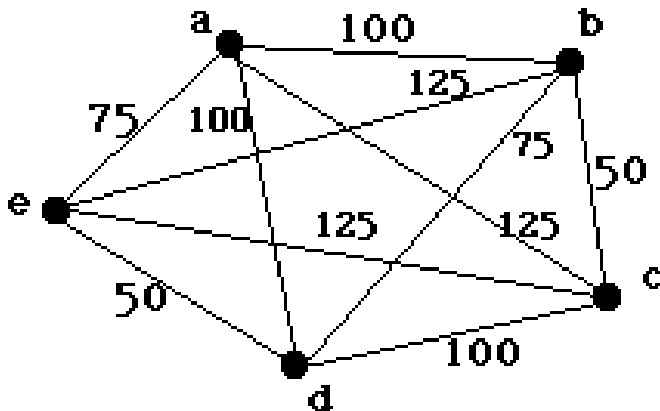
# Inleiding - Optimalisatieproblemen

Hoe de 'beste' oplossing bepalen?

Voor de hand liggende benadering:

- alle oplossingen bepalen
- voor elke oplossing de doelfunctie bepalen
- de oplossing nemen met de laagste doelfunctie

⇒ **Enumeration method**



Oplossingsruimte  
kan onhandel-  
baar groot zijn



n	n!
5	120
10	3.628.800
15	1.307.674.368.000
20	2.432.902.008.176.640.000



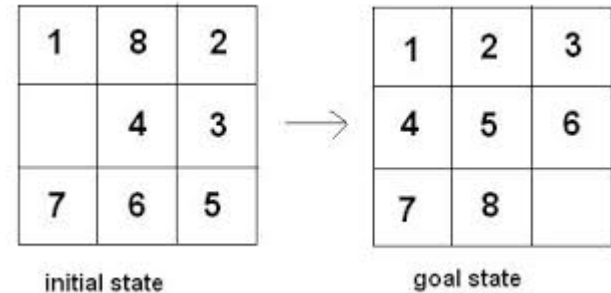
# Inleiding - Optimalisatieproblemen

---

## De 8 – puzzel

Vakjes in juiste volgorde zetten

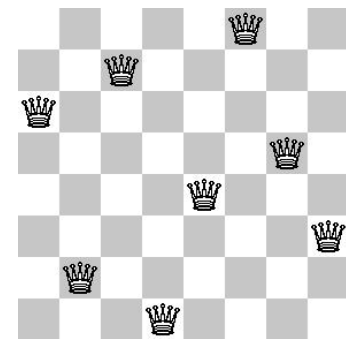
Hoe ziet een oplossing eruit?



## N – Queens

Acht koninginnen op schaakbord plaatsen  
zonder dat ze elkaar kunnen aanvallen

Hoe ziet een oplossing eruit?



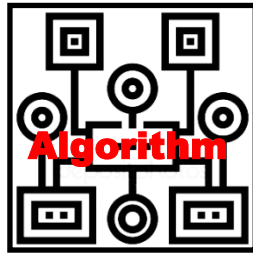
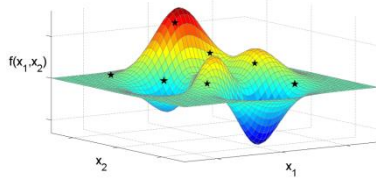


# **Algoritme versus heuristiek**

# Wat is een algoritme en wat is een heuristiek?

**Doel optimalisatieprobleem:** 'Beste' oplossing vinden tussen alle mogelijke oplossingen.

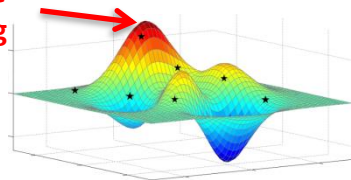
Optimalisatieprobleem



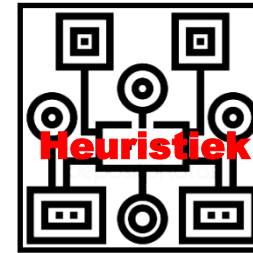
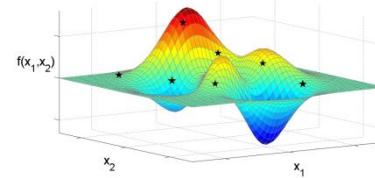
**Algorithm**



**Optimale oplossing**



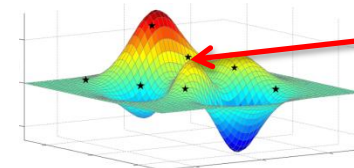
Optimalisatieprobleem



**Heuristiek**



**Goede oplossing**

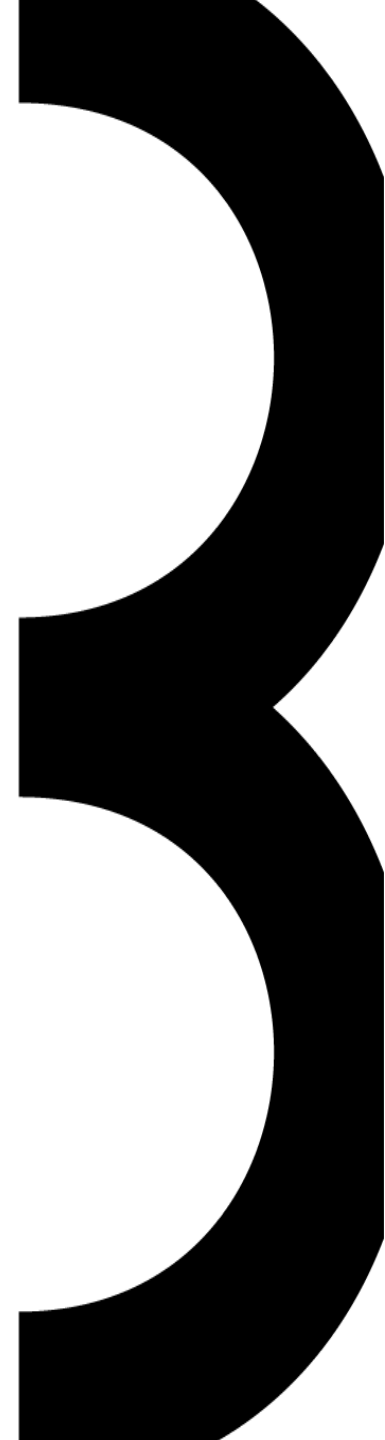


Waarom een heuristiek gebruiken?



---

# **Soorten heuristieken**



## 'Custum made'-heuristieken

- ontwikkeld voor een specifiek optimalisatieprobleem
- niet herbruikbaar voor andere optimalisatieproblemen.
- gebruikt/exploiteert specifieke aspecten en eigenschappen niet noodzakelijk aanwezig in andere optimalisatieproblemen.



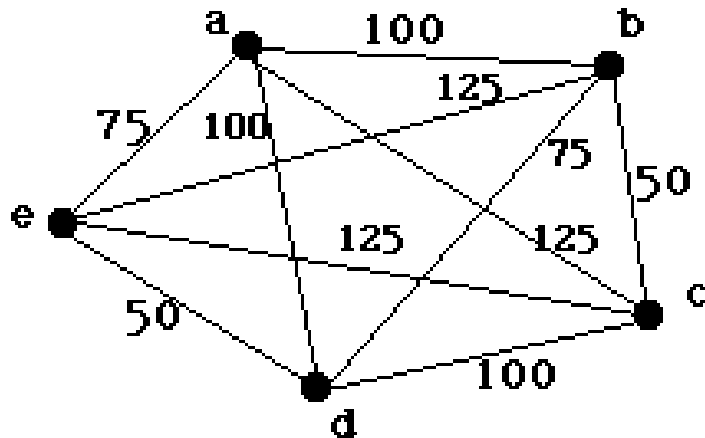
## Soorten heuristieken

---

### '*Custum made*'-heuristieken

D.m.v. een eenvoudig toe te passen criterium snel een goede oplossing bepalen

vb *nearest neighbour* benadering van het handelsreizigersprobleem:



Startpunt : a

a - dichtste buur: e

e - dichtste buur (nog niet bezocht): d

d - dichtste buur (nog niet bezocht): b

b - dichtste buur (nog niet bezocht): c

c – terug naar startpunt a

Oplossing : (a, e, d, b, c) met afstand 375

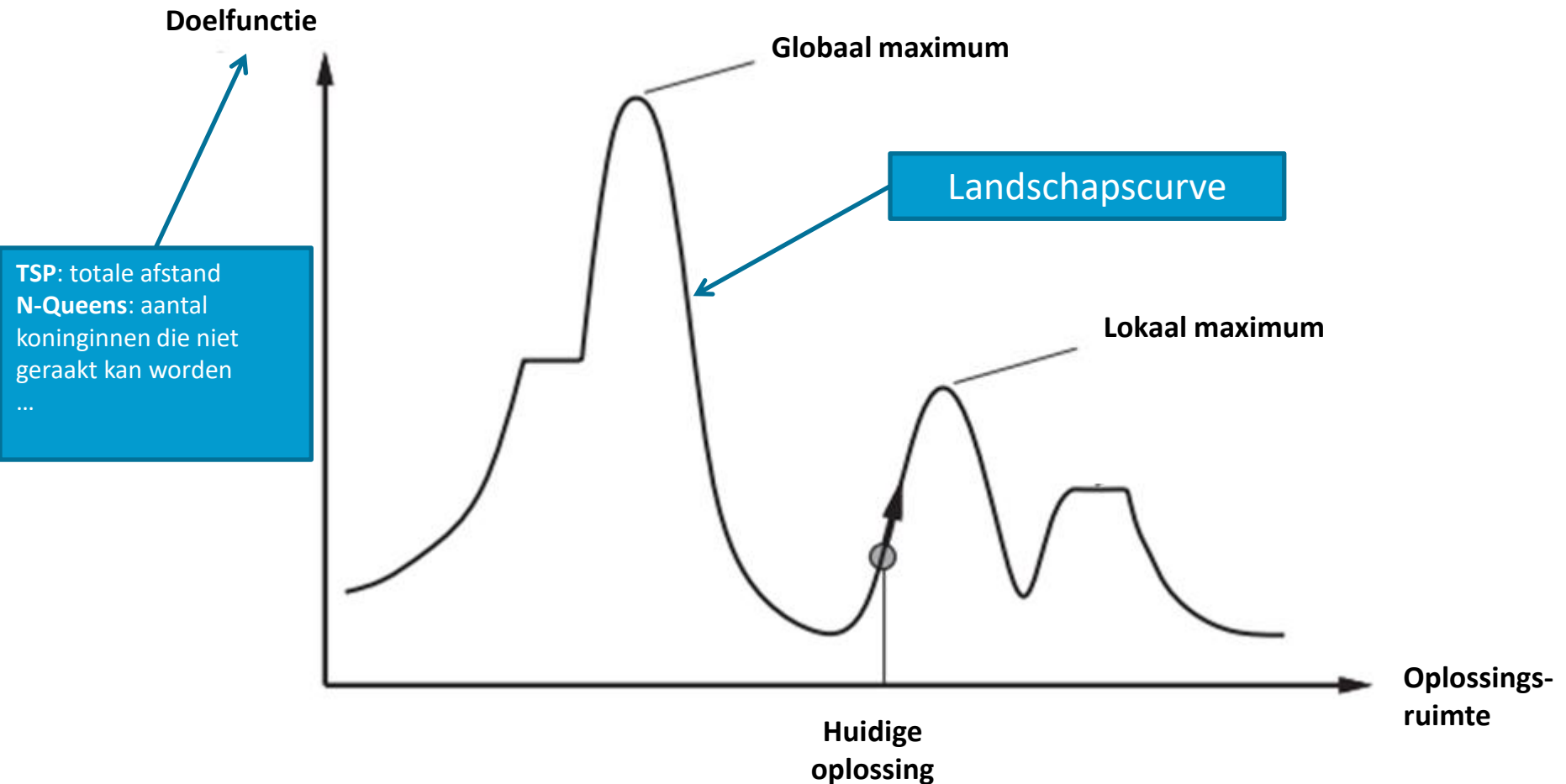
- Oplossing kan 'ver' van optimale oplossing liggen.
- 'Quick & dirty'-oplossing –beter dan random-oplossing-
- Worden vaak als startoplossing voor andere heuristieken (zie verder) gebruikt

### ***'Lokale zoek'-heuristieken***

Zoekmethoden die steeds in de 'buurt' zoeken van de vorige oplossing naar een betere oplossing. Een of meerdere stopcriteria worden gehanteerd.

- Slechts één of enkele oplossingen bijhouden en die "verbeteren"
- Oplossingen in de buurt zijn oplossingen met 'kleine' aanpassingen
- Resultaat is de 'beste' oplossing die tijdens de zoektocht werd gevonden (niet noodzakelijk de laatste oplossing)
- Risico 'vast' te raken in een lokaal minimum

# Soorten heuristieken – Vizualiseren 'lokaal zoek'-heuristiek



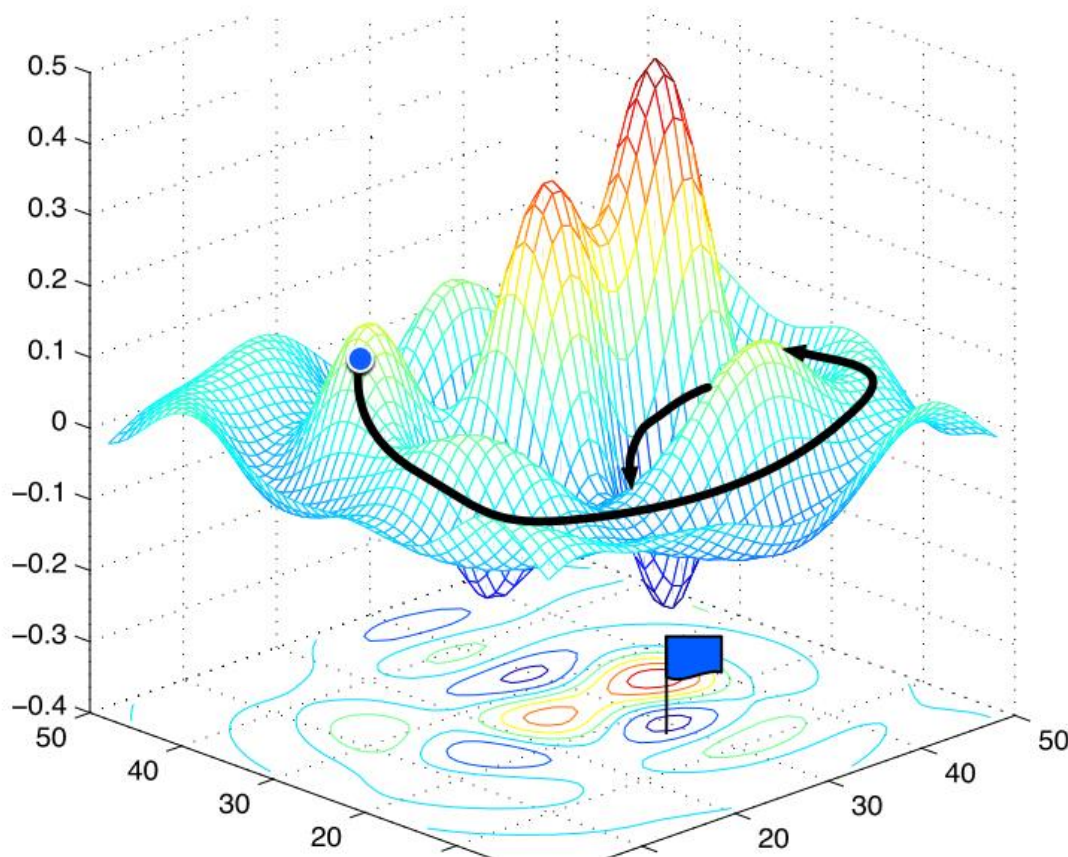
- Hoe voorkomen dat we vast komen te zitten?
  - Meerdere keren na elkaar uitgevoerd met andere initiële oplossingen
  - Zie meta-heuristieken



# Soorten heuristieken – Visualiseren 'lokaal zoek'-heuristiek

## Oplossingruimte als n-dimensionaal "landschap"

- Oplossingen zijn punten
- Hoogte van landschap is kwaliteit van de oplossing



Per stap evolueren de oplossingen richting een (lokaal) optimum

# Soorten heuristieken

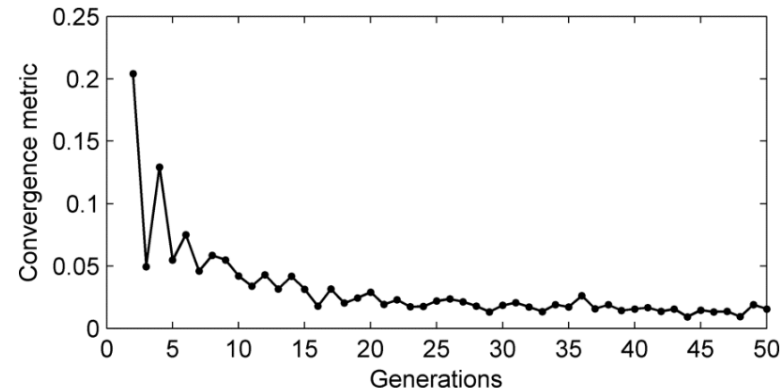
---

## ***Meta-heuristieken***

High level procedure toepasbaar op gelijk welk optimalisatie-probleem.

Het bevat de volgende elementen:

- Initiële oplossing bepalen  
(at random of d.m.v. een eenvoudige heuristiek)
- Toepassen 'lokaal zoek'-principe: huidige oplossing vervangen door 'betere' oplossing in de buurt
- Toelaten om af en toe toch naar een 'slechtere' buur te gaan  
(een manier om aan een lokaal optimum te ontsnappen)
- Gebaseerd op een analogie uit de fysica, de biologie of de ethologie
- Parameters sturen de duur van de heuristiek en de kwaliteit van de gevonden oplossing



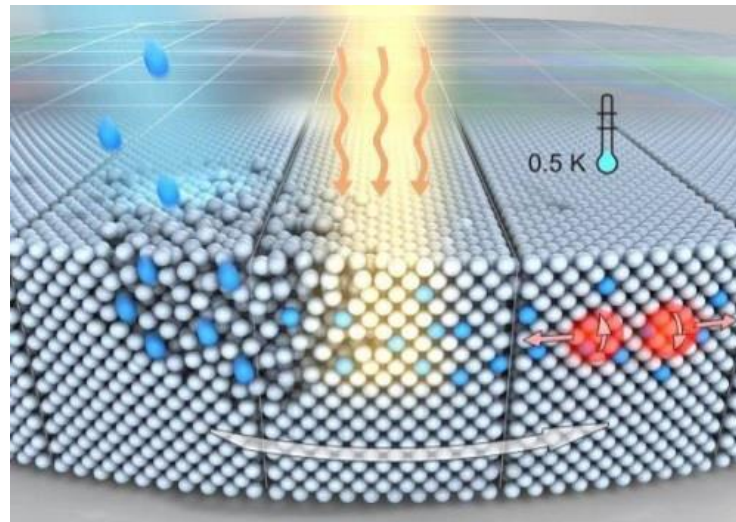
# Soorten heuristieken

---

## Voorbeeld van meta-heuristiek

### Simulated Annealing(Kirkpatrick, e.a. ,1970)

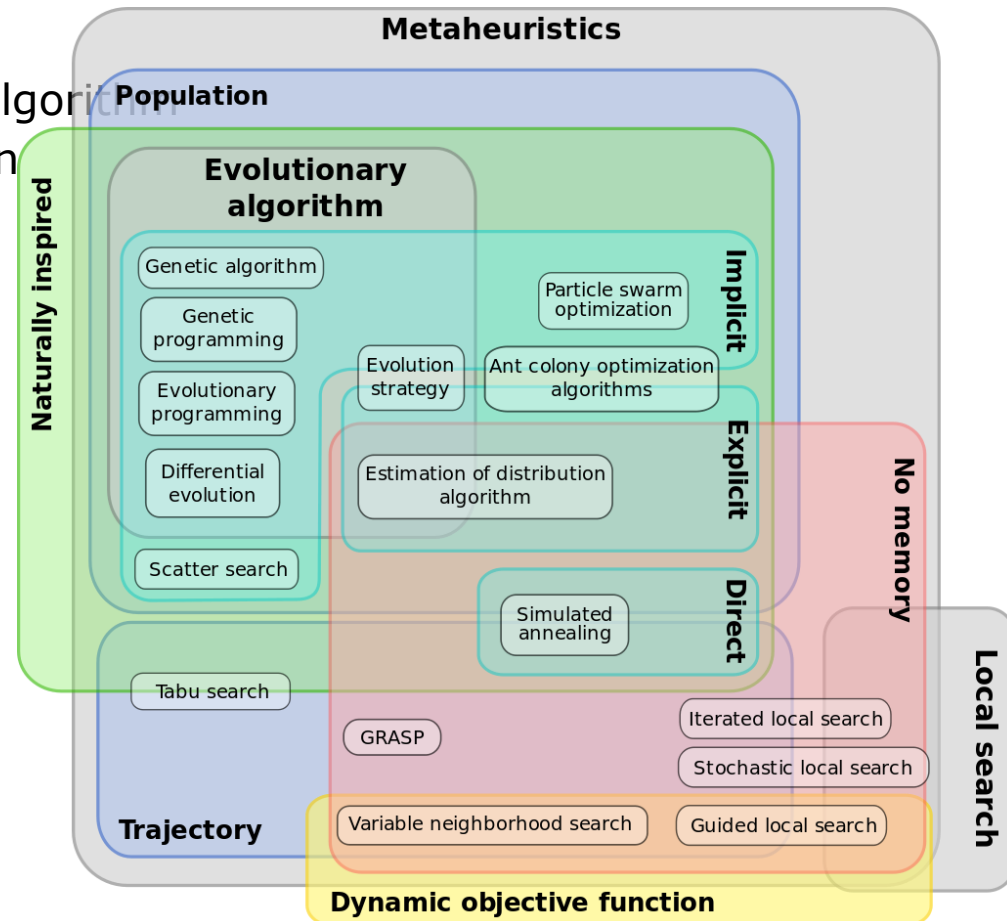
- gebaseerd op het afkoelen van een metaal
- potentiële oplossingen kristalliseren uit



# Soorten heuristieken

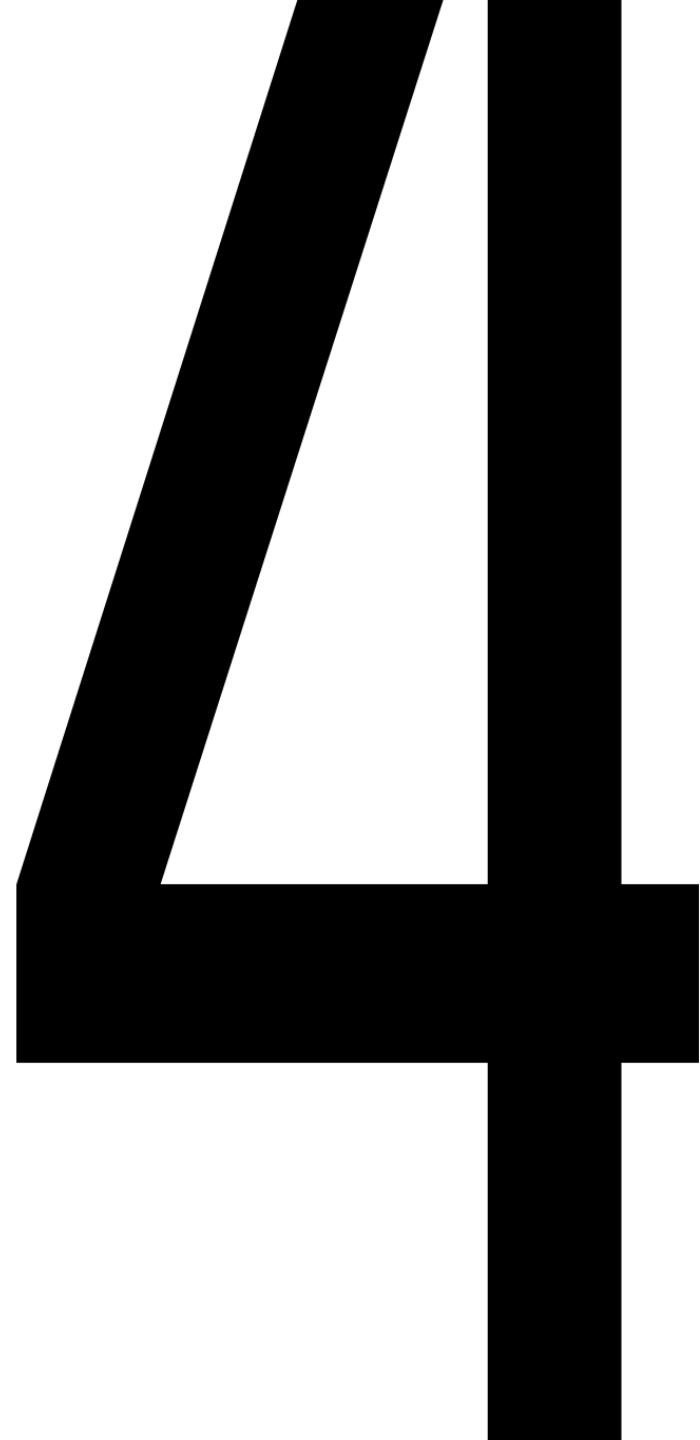
## Andere voorbeelden van meta-heuristieken:

- ant colony optimisation algorithm
- genetic algorithm
- particle swarm optimization
- artificial immune systems
- bacterial foraging optimization algorithm
- biogeography-based optimization
- coevolutionary algorithms
- cultural algorithms
- differential evolution algorithm
- greedy randomized adaptive search procedure
- scatter search
- ...



---

# **Simulated Annealing**

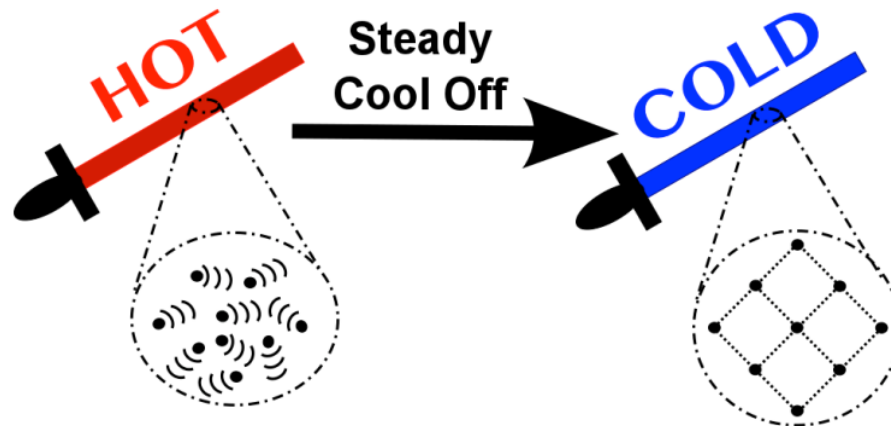


# Simulated Annealing

---

## Gebaseerd op afkoelingsproces in materialen: annealing

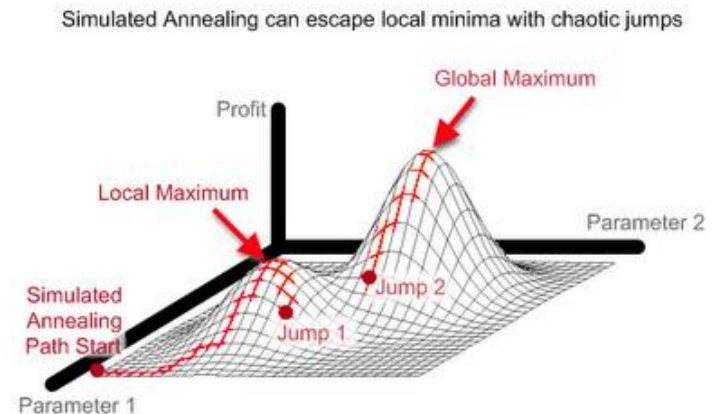
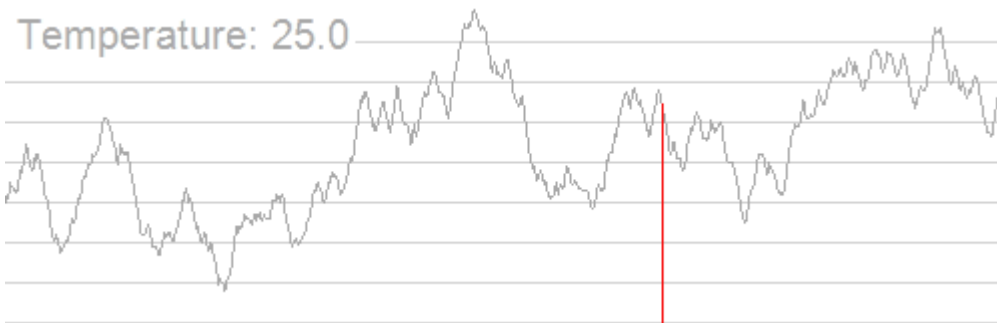
- atomen in materiaal bewegen, maar hoe lager de temperatuur, hoe minder
- Materiaal opwarmen om in juiste vorm te krijgen, nadien terug afkoelen
- afkoeling te snel  $\Rightarrow$  onzuiverheden (amorfe structuren)
- afkoeling geleidelijk  $\Rightarrow$  sterkere kristalstructuren
- Eindtoestand wordt dus geleidelijk aan bereikt.



# Simulated Annealing

## Simulated annealing

- Willekeurige zoektocht naar oplossing d.m.v. “**sprongen**” (random walk)
- In principe enkel “sprong” naar een betere oplossing  
→ soms ook minder goede (*Waarom?*).
- Na elke “sprong” **beste oplossing** onthouden.
- Kans op “sprongen” naar minder goede oplossingen worden kleiner met **dalende temperatuur**  
→ oplossing kristalliseert uit



# Simulated Annealing – Pseudo-code

---

```
InitializeParameters (Temperature  $t$ , TemperatureReduction  $\alpha$ )
initialSolution (Solution  $s$ )
 $s^* = s$     //best found solution
while  $t > TMIN$ 
    temperatureIteration = 0
    While temperatureIteration < maxIterations
         $s' = \text{SelectNeighbour}(s)$ 
         $\Delta = \text{objectiveFunction}(s') - \text{objectiveFunction}(s)$ 
        // objectiveFunction must be minimized
        if ( $\Delta < 0$ )
            then  $s = s'$ 
                if  $\text{objectiveFunction}(s') < \text{objectiveFunction}(s^*)$ 
                    then  $s^* = s'$ 
            else if  $\text{atRandom}[0,1] < \exp(-\Delta/T)$ 
                then  $s = s'$ 
        end while
     $t = \alpha * t$ 
end while
return  $s^*$ 
```

---



# Simulated Annealing - Temperatuurdaling

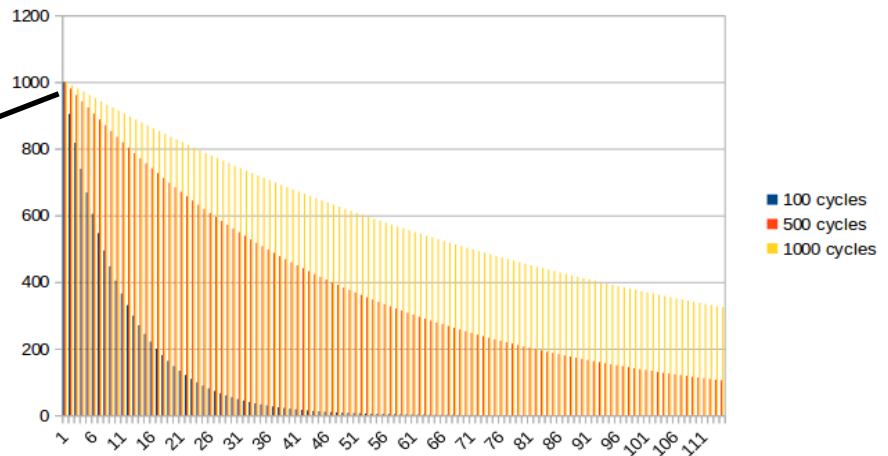
## Twée methoden

**1. Constante temperatuurdaling:** per tijdseenheid temperatuur vaste stapgrootte laten dalen

**2. Exponentiële temperatuurdaling:** kies een begin- ( $T_s$ ) en eindtemperatuur ( $T_e$ ) kiezen en verlaag temperaturen telkens

$e^{\left(\frac{\log \frac{T_e}{T_s}}{c-1}\right)}$  met  $c$  = aantal cycles (iteraties) per temperatuurwaarde

Daling vanaf 1000 °C



# Simulated annealing

---

In Python: Meerdere Python libraries ondersteunen Simulated Annealing. Een daarvan is de *simanneal*.

Zie [GitHub - perrygeo/simanneal: Python module for Simulated Annealing optimization](#) voor meer informatie omtrent deze package

```
>>> pip install simanneal    # alleen de eerste keer
```

*Het doel van de simanneal library is om de probleem-specifieke berekeningen te scheiden van de Metaheuristiek-specifieke berekeningen:*

## Probleem-specifieke berekeningen

- **move**: hoe van een oplossing naar een buur-oplossing te gaan
- **energy**: berekent de waarde van de objectieve functie voor een oplossing

## Metaheuristiek-specifieke berekeningen

- Opgeven van de annealing parameters, zo niet, worden er default waarden gebruikt
- Uitvoeren van de simulated annealing heuristiek

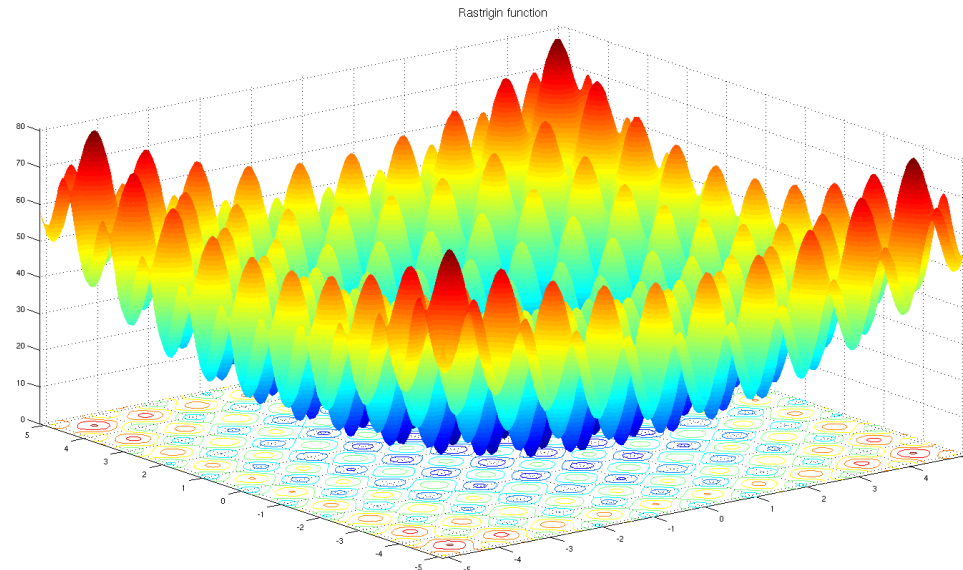
# Simulated annealing

---

Laten we aan de hand van een voorbeeld *simanneal* illustreren:

**Rastrigin functie** - klassieke case om optimalisatie algoritmen en heuristieken te testen

- Oplossingsruimte:  $(x, y)$  met  $x$  en  $y \in [-5.12, 5.12]$
- Doelfunctie:  $f(x) = 20 + x^2 - 10 \cdot \cos(2 \cdot \pi \cdot x) + y^2 - 10 \cdot \cos(2 \cdot \pi \cdot y)$
- Minimaliseer doelfunctie



# Simulated annealing

---

## Voorbeeld: Rastrigin functie

```
>>> from simanneal import Annealer

>>> class RastriginProblem(Annealer):
    def move(self):
        # x: self.state[0] en y: self.state[1]
        i = np.random.randint(0, 2)
        self.state[i] += np.random.normal(0, 0.1)
        self.state[i] = np.clip(self.state[i], -5.12, 5.12)

    def energy(self):
        sum = 20 + self.state[0]**2 - 10*math.cos(2*math.pi*self.state[0])
            + self.state[1]**2 - 10*math.cos(2*math.pi*self.state[1])
        return sum

>>> init_sol = np.random.uniform(-5.12, 5.12, size=2) # initiele [x, y]

>>> rastrigin = RastriginProblem(init_sol) # initiele [x, y] -> self.state

>>> # opgeven van de annealing parameters. Zo niet: default waarden
>>> rastrigin.anneal()
```

# Simulated annealing

---

We hebben in ons voorbeeld verschillende default waarden gebruikt, maar deze kunnen worden gewijzigd - voordat het annealing proces wordt uitgevoerd -:

- `Tmax = 25000.0` # Max (start) temperatuur
- `Tmin = 2.5` # Min (eind temperatuur)
- `temperature steps = 50000` # Aantal iteraties
- `updates = 100` # Aantal updates (per default wordt een update geprint op stdout)

Dit zijn attributen van het object `rasterin`. Vb.:

```
>>> rasterin.Tmin = 25.0
```

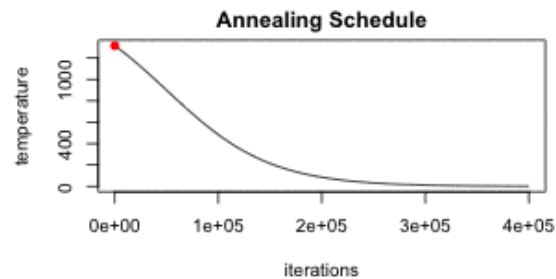
Opmerkingen:

- Per default, wordt de *energy* functie **geminimaliseerd**, dus dient het resultaat van de objectieve functie met -1 te worden vermenigvuldigd indien de objectieve functie gemaximaliseerd dient te worden
- **Boundaries** op de waarden van de oplossing kunnen in de *move* functie worden opgenomen
- **Andere constraints** naast boundaries op de waarden van een oplossing dienen in de *energy* functie (= objectieve functie) te worden opgenomen.

# Simulated annealing - TSP

---

Distance: 43,499 miles  
Temperature: 1,316  
Iterations: 0



# Simulated annealing

---

## Voorbeeld: TSP

```
distance_matrix = np.array([[0, 100, 125, 100, 75],
                             [100, 0, 50, 75, 100],
                             [125, 50, 0, 100, 125],
                             [100, 75, 100, 0, 50],
                             [75, 100, 125, 50, 0]])
```

```
>>> class TSPProblem(Annealer):
    def move(self):
        # Swaps two cities in the route
        a = np.random.randint(0, len(self.state))
        b = np.random.randint(0, len(self.state))
        self.state[a], self.state[b] = self.state[b], self.state[a]

    def energy(self):
        # Calculates the length of the route
        dist = 0
        for i in range(len(self.state)):
            dist += distance_matrix[self.state[i - 1], self.state[i]]
        return dist

>>> initial_state = [0, 4, 1, 3, 2]
>>> tsp = TravellingSalesmanProblem(initial_state)
>>> route, distance = tsp.anneal()
```

# Simulated annealing

---

## Voorbeeld: TSP

```
distance_matrix = np.array([[0, 100, 125, 100, 75],
                             [100, 0, 50, 75, 100],
                             [125, 50, 0, 100, 125],
                             [100, 75, 100, 0, 50],
                             [75, 100, 125, 50, 0]])
```

```
>>> class TSPProblem(Annealer):
    def move(self):
        # Swaps two cities in the route
        a = np.random.randint(0, len(self.state))
        b = np.random.randint(0, len(self.state))
        self.state[a], self.state[b] = self.state[b], self.state[a]

    def energy(self):
        # Length of the route without for loop
        from_city = self.state
        # shift the array one position to the right
        to_city = np.roll(from_city, -1)
        return distance_matrix[from_city, to_city].sum()

>>> initial_state = [0, 4, 1, 3, 2]
>>> tsp = TravellingSalesmanProblem(initial_state)
>>> route, distance = tsp.anneal()
```