# Praktikum 02 – Grundlagen der KI

Name: Umar Farooq

Datum: 26.10.2025

# EA.01: Modellierung von GA

## 8- Queens-Problem

Ziel: Setze 8 Damen auf einem Schachbrett so, dass keine sich gegenseitig bedroht

## Kodierung

Ein Individuum wird als Permutation der Zahlen 1–8 kodiert:

- Index = Spalte
- Wert = Zeile der Dame
  Beispiel: `[1, 5, 8, 6, 3, 7, 2, 4]` bedeutet:
- Spalte 1: Dame in Zeile 1
- Spalte 2: Dame in Zeile 5
  usw.

Diese Kodierung stellt sicher, dass keine zwei Damen in derselben Spalte oder Zeile stehen.

## Fitnessfunktion

Zähle die Anzahl der konfliktfreien Paare.
Es gibt insgesamt 28 mögliche Paare.
Für jedes Paar, das sich nicht auf gleicher Diagonale befindet, +1 Punkt.
Fitnesswert f=28−Anzahl der Konfliktpaare.
Ziel: Maximierung von f=28.

## Operatoren

- **Crossover:** Order-Crossover (OX) oder Partially-Mapped Crossover (PMX).
  -> Bewahrt die Permutationsstruktur.
- **Mutation:** Swap-Mutation (Tausch zweier zufälliger Positionen).
  -> Erhält ebenfalls gültige Permutationen.

## Begründung

Die Kodierung als Permutation garantiert gültige Individuen, und die gewählten Operatoren sind für Permutationsprobleme geeignet (verhindern Duplikate).

## Landkarten-Färbeproblem

Ziel: Färbe Regionen A, B, C, D, E, F so, dass keine benachrbarten Regionen dieselbe Farbe haben.

Domäne: {rot, grün, blau, gelb, violett}

## Kodierung

[A, B, C, D, E, F] ∈ {1,...,5}^6

-> ein Individum ist ein Farbvektor der Länge 6(Zahl steht für Farbe)

## Fitnessfunktion

Ziel: minimale Konflikte und möglichst wenig verwendete Farben.
Formel z. B.:

f=w1 ×(Anzahl konfliktfreier Nachbarschaften)−w2 ×(Anzahl genutzter Farben)

Ziel: Maximierung von f

## Operatoren

- **Crossover:** Uniform-Crossover oder 1-Point-Crossover.
  → mischt die Farbzuweisungen der Eltern.
- **Mutation:** Zufällige Änderung der Farbe einer Region (mit Wahrscheinlichkeit pm ).

## Begründung:

Die Kodierung ist einfach, Operatoren erhalten Struktur (keine ungültigen Lösungen). Mutation sorgt für Exploration.

## Simulated Annealing

Um die genannten Probleme mit **Simulated Annealing (SA)** lösen zu können, sind folgende zusätzliche Komponenten erforderlich:

1. **Eine Nachbarschaftsfunktion**, die definiert, wie eine kleine Veränderung einer Lösung aussieht.
   Beim 8-Queens-Problem kann eine Nachbarschaft durch das Vertauschen zweier Damenpositionen erzeugt werden. Beim Landkarten-Färbeproblem kann sie durch das Ändern der Farbe einer einzelnen Region realisiert werden.
2. **Eine Kosten- oder Bewertungsfunktion**, die die Qualität einer Lösung quantifiziert.
   Für das 8-Queens-Problem ist dies die Anzahl der Konflikte, die minimiert werden soll. Für das Landkarten-Färbeproblem entspricht sie der Anzahl der Farbkollisionen, eventuell mit einem Zusatzterm für die Anzahl der genutzten Farben.
3. **Ein Startzustand**, also eine initiale Lösung, beispielsweise eine zufällige Permutation oder eine zufällige Einfärbung.
4. **Ein Temperaturplan (Cooling Schedule)**, der festlegt, wie stark zufällige, verschlechternde Änderungen zu Beginn akzeptiert werden und wie diese Wahrscheinlichkeit im Verlauf abnimmt.

Mit diesen Elementen lässt sich für beide Probleme ein SA-Algorithmus formulieren, der durch sukzessives Abkühlen (Temperaturreduktion) schrittweise bessere Lösungen findet.

# EA.03: Anwendungen

## Here’s Waldo

**Problemabbildung**
Olson formalisierte das Waldo-Suchen als Traveling-Salesman-Problem über 68 mögliche Waldo-Positionen, die aus den Büchern extrahiert wurden. Ein Individuum ist eine Permutation dieser 68 Punkte, also eine Besuchsreihenfolge. Ziel ist eine kurze Route, die alle Kandidatenpunkte möglichst effizient abdeckt.

**Kodierung**
Permutation der 68 Koordinaten als Genom. Jede Position kommt genau einmal vor.

**Fitnessfunktion**
Negativer Pfadweg bzw. Minimierung der Gesamtdistanz der Route über alle Punkte. Kürzere Route entspricht höherer Fitness.

**Operatoren**
Olson beschreibt den Einsatz eines genetischen Algorithmus; typische TSP-Operatoren sind permutationsverträgliche Crossover und lokale Mutationen (Swap/Insert). Das Blog verweist auf den begleitenden Code und zeigt, dass GA iterativ bessere Routen findet und ein Hill Climber schlechter konvergierte. Kernaussage: GA optimiert die Besuchsreihenfolge, bis keine Verbesserung mehr gefunden wird.

**Ergebnis/Interpretation**
Die resultierende Route legt eine praktische Startreihenfolge nahe (links unten beginnen, dann obere rechte Seite usw.).

# Evolution Simulator

**Was simuliert wird**
Ein evolutionärer Physiksimulator von Cary Huang (carykh). Es werden viele Kreaturen generiert, die über eine Hindernislandschaft laufen bzw. klettern. Das System sortiert, eliminiert und wiederholt über Generationen.

**Kodierung**
Genome beschreiben den Körperbau und die Antriebe der Kreaturen. In der

veröffentlichten Processing-Version sind unter anderem Mindestzahlen an Knoten und “Muskeln” konfigurierbar (CREATURE_MIN_MUSCLES). Damit ist das Individuum eine Struktur aus Knoten und verbindenden “Muskeln” mit Parametern.

**Operatoren**

Mutation dominiert. Es gibt einen globalen Mutationsfaktor (MUTABILITY_FACTOR) und optional wechselnde Umgebungen durch zufällig veränderte Hindernisse. Ein klassischer rekombinierender Crossover ist nicht zentral dokumentiert; die Selektion erfolgt über “sort, kill and repeat”, also Auswahl der Leistungsstarken und Ersetzen der Schwachen.

**Fitnessfunktion**

Leistungsmaß ist die Fortbewegung über das Terrain; im UI werden u. a. „median distance“ und Generationenverlauf geplottet. Praktisch entspricht die Fitness der zurückgelegten Distanz bzw. dem Fortschritt über Hindernisse.

## American fuzzy lop

**Zweck**

AFL ist ein sicherheitsorientierter Gray-Box-Fuzzer, der mit Compiler-Instrumentierung und einem genetischen Algorithmus neue Testfälle generiert, die neue interne Programmpfade erreichen. Ziel ist maximale Codeabdeckung und das Finden von Crashes.

**Kodierung**

Individuen sind rohe Eingabebytes (Testdateien) für das Zielprogramm. Keine spezialisierte Struktur nötig.

**Fitnessfunktion**

Primär Codeabdeckung: Ein Testfall ist “interessant”, wenn er neue Kanten/Zustände im instrumentierten Programm auslöst. Solche Fälle kommen in die Queue. Es gibt

außerdem ein “Power schedule”, das Auswahlgewichte u. a. aus Laufzeit und Dateigröße ableitet.

**Operatoren**

Mutationspipeline in zwei Stufen

1. Deterministische Mutationen: Bitflips, Inkremente/Dekremente in 8/16/32 Bit, Überschreiben mit „interessanten“ Werten, Wörterbuchersetzungen.
2. Havoc-Phase: zufällige Ketten von Mutationen, außerdem Blockoperationen (löschen, duplizieren, überschreiben).
   Crossover-ähnlich: Splicing zweier Testfälle, bevor wieder Havoc angewandt wird. Neue Abdeckung oder Crashes werden gespeichert.

# Weitere Anwendungen für EA/GA

- **Routen- und Tourenplanung** (TSP, VRP) in Logistik und Fertigung; Permutationskodierung, Kosten als Distanz/Zeiten.
- **Neuroevolution** für Controller und Agenten (etwa Evolution von Netzgewichten und Topologien, “NEAT”).
- **Parameter- und Hyperparameteroptimierung** in Machine Learning, wenn Gradientensuche schwer ist.
- **Symbolische Regression und Programmsynthese** mit Genetischer Programmierung.
- **Designoptimierung** in Ingenieurwesen, z. B. Tragwerks- und Flügelprofile mit Multi-Objective GAs (Pareto-fronten).
  Diese Punkte sind Standardanwendungen der Evolutionären Algorithmen; sie passen methodisch zu den drei Beispielen hier.