ALGORITHMS IN AI & DATA SCIENCE 1 (AKIDS 1)

Metaheuristic Search

Prof. Dr. Goran Glavaš

15.1.2024

Content

- Constrained Discrete Optimization & Metaheuristics
- Single Point Search Algorithms
- Population-Based Algorithms
    - Genetic algorithm

Recap: State Space Search

- We will denote the set of all states (state space) with **S**
  - The state space is commonly **so large** that we can’t iteratively list all states
  - All states in the space are not really „known” in advance
  - When in state s, we typically only then compute the set of possible next states

State space search

A state space search problem is defined with a triple (s_0, *succ*, goal) where $s_0 \in S$ is the **initial state**, *succ*: **S** → ℘(**S**) is the **successor function** that for some state s returns a set of states that we can **transition to** from s, and *goal*: **S** → {True, False} is a **predicate** (function that returns a boolean value) that for a given state s determines if s is a **goal state** or not (there can be multiple states that satisfy the goal predicate). A state space search (typically) ends as soon as any goal state is found.

Recap: Heuristic Search

- There are generally two types of search
- **Uninformed (blind) search**
  - No additional information about the problem, that could indicate whether one state is perhaps closer to the goal state than another state
- **Informed (directed, heuristic) search**
  - Additional information helps avoid some states and speed up the search
  - Problem-specific estimate of state’s distance from the goal is available

Heuristics

- If we have a (vague) **idea in which direction to look** for the solution, why not use this information to improve the search?

- **Heuristics =** problem-specific rules („vague ideas”) about the nature of the problem
  - **Purpose**: direct the search towards the goal so it becomes more efficient

Heuristic function

Heuristic function h: $\mathbf{S} \rightarrow \mathbb{R}^+$ assigns to each state $s \in \mathbf{S}$ an **estimate** of the **distance between that state and the goal state**

State Space Search vs. Constrained Optimization

- In **State Space Search**, we're looking to reach the **goal state** with the minimal cost (maximal gain)
  - **Heuristics** (task-specific) help reduce the search space
  - The **solution** is the **path** of transitions from s_0 to goal state
- In **Discrete Constrained Optimization** (aka **combinatorial optimization**), we search across a very large space of states, but each state represents one possible solution
  - There is a **function that assigns a quality value to each state**, indicating how good of a solution it may be
  - There is often also a **set of constraints**: if a state does not satisfy the constraints it is not a valid solution

Discrete Constrained Optimization

Discrete Constrained Optimization Problems

In **discrete constrained optimization**, we search for an **optimal state** in large space of possible states. Each state **X** can be seen as consisting of n variables **X** = $x_1, x_2, \ldots, x_n$, each with a corresponding domain $D_1, D_2, \ldots, D_n \subseteq \mathbb{Z}$ (whole numbers). The optimal state is the one that maximizes/minimizes the **objective function** $f: D_1 \times \cdots \times D_n \rightarrow \mathbb{R}$. Finally, the constraints $C_1, \ldots, C_m$, with $C_i \subseteq D_1, D_2, \ldots, D_n$ define the subsets of the state space that encompass **valid solutions** to the problem

- Optimal state (or the state with the best *f* that was found) is the **solution**
- No path between start and goal state – commonly there isn't even a clear start state
- We're not making moves like in classic **SSS** problems, just **searching for the best possible solution** over a very large space of candidate solutions

State Space Search vs. Constrained Optimization

- **State Space Search**
  - **Goal states** represent a very small portion of the states in the search space
  - Only **paths** that reach one of goal states are (candidate) **solutions**
  - Explicit transitions between the states (*succ* function)
  - **Problem**: how to get to a goal state with **minimal cost/maximal gain**

- **Discrete Constrained Optimization**
  - **Every state** represents one candidate solution to the problem
  - Each state (candidate solution) has a measure of **"quality"** – **the objective function** *f* – *a*ssigned to it
  - **No** explicit „start state" nor „state transitions" – instead **neighbourhood** (or distance) between states (but not in the sense of transition cost)
  - **Problem**: how to find the state with minimal/maximal value of the objective *f*

Heuristics vs. Metaheuristics

- **Heuristics** in State Space Search
  - **Trim** the number of paths to be explored
  - Estimate the **distance** of the current state from the goal state
- **Discrete Constrained Optimization**
  - **No goal state**, every state is a **possible solution**
  - Often no explicit „start state" nor explicit „state transitions"
  - **Q:** Where to start? Where to look next after evaluating some state?
    - When in a state, no idea how **„far"** that state/solution is from the optimal state/solution
  - Concept of **distance** / **neighbourhood** in DCO problems:
    - Measures **how similar** two **states = candidate solutions** are (not a cost of transitioning between states, there are no transitions!)
  - **Metaheuristic** frameworks: **search strategies** for selecting the next state(s)/solution(s) to be evaluated, typically in a **problem-agnostic manner** (applicable to any DCO problem)

Discrete Constrained Optimization: Example

Traveling Salesman Problem

The **travelling salesman** needs to visit n cities and wants to make the **minimal possible path**. Given a list of cities and distances between each pair of cities, find the shortest possible route that visits each city exactly once and returns to the original city.

- **NP-Hard combinatorial problem** (no known polynomial time algorithm for solving it) – **factorial complexity** – **Hamiltonian cycle** of a (fully connnected) graph
- Real-world applications: vehicle routing, chip manufacturing, ...
- TSP as **combinatorial optimization**:
  - **X**: $x_1, x_2, \ldots, x_n, x_{n+1}$
  - $\mathbf{D}_1 = \mathbf{D}_2 = \ldots = \mathbf{D}_n = \mathbf{D}_{n+1} = \{1, 2, \ldots, n\}$
  - $f(\mathbf{X})$: $d(x_1, x_2) + d(x_2, x_3) + \ldots + d(x_{n-1}, x_n) + d(x_n, x_{n+1})$
  - **Constraints**:
    - $x_1 = x_{n+1}$ (some concrete city from $\{1, \ldots, n\}$)
    - $x_1 \neq x_2 \neq x_3 \neq \ldots \neq x_{n-1} \neq x_n$ (no repetition of cities along the path)

Metaheuristic search strategies

- **Metaheuristics strategies guide** the search process
  - **Direct** the search (selection of next states to evaluate) so that the chances of finding a good (or near-optimal) state increase

- They are approximate – no guarantee of finding an **optimal solution**

- Most commonly, they are also **non-deterministic** (and most often **stochastic**) – there is randomness involved

- **Metaheuristics** are **problem-agnostic**, but may use problem-specific heuristics as part of the strategy (but as „black boxes”, without caring what they are)

Classifying metaheuristics

- **Single point search** (one candidate solution examined at a time) vs. **Population-based search** (a set of candidates examined at each step)
- **Nature-inspired** (e.g., *evolutionary algorithms* or *ant colony optimization*) vs. **Others** (not nature inspired)
- **Static** (*f* does not change during search) vs. **Dynamic** objective function (changes during the search)
- **Using memory** vs. **Memory-less**
  - Memory as in experience from previous searches on similar/same problem

Content

- Constrained Discrete Optimization & Metaheuristics
- Single Point Search Algorithms
- Population-Based Algorithms
    - Genetic algorithm

Single-Point Search

- **Single point search algorithms**, also known as **trajectory methods**, examine one state (candidate solution) at a time
  - They then choose the next candidate solution to be examined, typically from a **local neighbourhood** of the current solution
  - The **neighbourhood** of a state *N*(s) needs to be defined for a concrete problem
    - In principle, similar purpose as *succ* in state space search
    - But *succ* in **SSS** typically clearly defined by the problem, *N*(s) in DCO <u>not</u> obvious
- **Local search**
  - Choose (usually randomly) an initial solution (s_0)
  - Given *N*(s), determine the neighbourhood of current solution s
  - Explore the neighbourhood and select one neighbor
  - Proceed with the selected neighbor as the next state/solution

Simple Descent/Ascent

- We will assume we're minimizing the objective *f*
  - Thus, the algorithms will be called „descent"
  - If the objective is to be maximized, we would be „ascending"

- **Simple Descent**
  - When in a state s, selects any neighbour s' for which f(s') < f(s) (in case of maximization, f(s') > f(s))
  - The order of exploration of neighbours in *N*(s) is **underspecified**, but typically random

```
simple_descent(s, N)
 while True
   better = False
   for s' in  N(s)
      if f(s') < f(s)
        s = s'
        better = True
        break
   if not better
     break
 return s
```

Deepest Descent

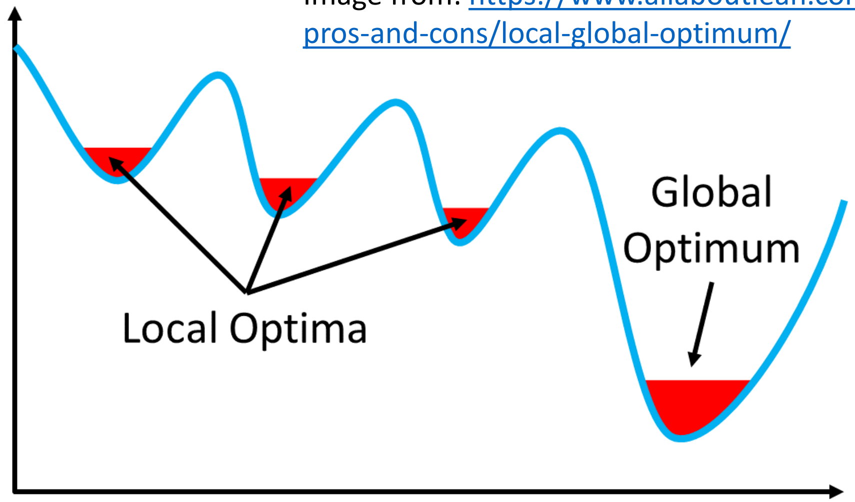
- We will assume we're minimizing the objective *f*
  - Thus, the algorithms will be called „descent"
  - If the objective is to be maximized, we would be „ascending"
- **Deepest Descent**
  - Greedy strategy: in each step we select the neighbour with the smallest *f*(s')
  - The order of exploration of neighbours in *N*(s) is **underspecified**, but doesn't matter because we have to check all neighbours anyways
  - Guaranteed to lead to the closest local optimum (minimum) from the initial state

```
deepest_descent(s,N)
  while True
    best = s
    for s' in  N(s)
      if f(s') < f(best)
        best = s'
    if best == s
      break
    else
       s = best
  return best
```

Local and Global Optima

- With greedy search (deepest descent)
  - We are **guaranteed** to find the closest local optimum from the initial state
- **Q:** is that good or bad?
  - Depends where we start
  - We typically choose the starting state randomly
- **Solution**: run the „deepest descent” multiple times
  - Each time from a different initial state

Image from: https://www.allaboutlean.com/polca-pros-and-cons/local-global-optimum/



```
multistart_deepest_descent(iters)
  best = null # f(null) = +inf
  for i in 1 to iters
    s0 = randomly select initial state s0
    s = deepest_descent(s0)

    if f(s) < f(best)
        best = s
  return best
```

Simulated Annealing

- **Simulated annealing** is a metaheuristic strategy that borrows the idea from from material physics about reaching the minimal energy state
  - For example, for glass or metal
- **Annealing**: heating the material and then slowly cooling it

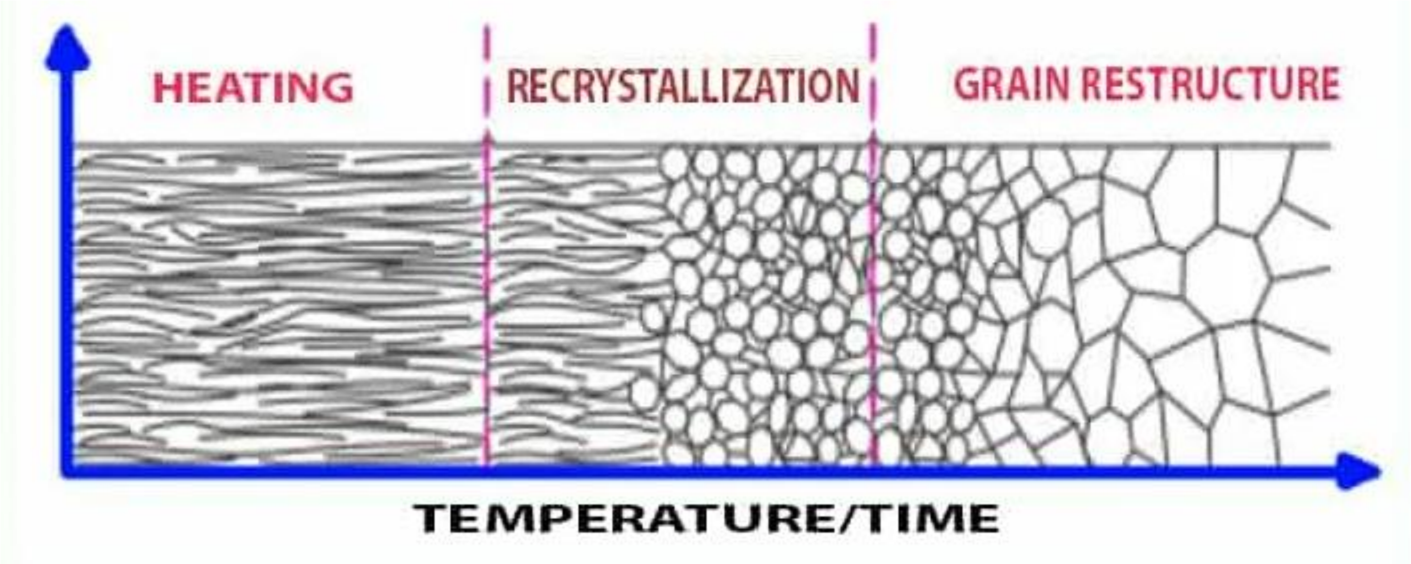


Image from: https://www.mechstudies.com/annealing-process-definition-meaning-types-applications/

Simulated Annealing

- **Annealing**: heating the material and then slowly cooling it
- Compared to strict „descents”, SA allows to select the next state with larger value of *f* („hotter solution” or „worse solution”)
  - With decreasing probability as the search procedes
  - Allows for more of „random search” in the **early stages** and more focused (minimizing) search later on

- Selects (randomly) a state from the neighbourhood and accepts it according to the following probability p(s’)

$$p(T, s', s) = \begin{cases} \mathbf{1} & \textbf{if} \quad f(s') < f(s) \qquad \text{\# always accept a better solution} \\ e^{-(f(s')-f(s))/T} & \textbf{otherwise} \end{cases}$$

Simulated Annealing

$$p(T, s', s) = \begin{cases} \mathbf{1} & \textbf{if} \quad f(s') < f(s) \quad \text{\# always accept a better solution} \\ e^{-(f(s')-f(s))/T} & \textbf{otherwise} \end{cases}$$

- The temperature T plays the crucial role
  - If $T = 0$, $p(T, s', s) = 0$
- It is gradually reduced
  - **Linear annealing**: T -> a*T, where a is a constant (typically between 0.8 and 0.99)
- When does it end?
  - Fixed number of iterations
  - Or when T becomes close enough to 0

```
simulated_ annealing($s_0$, N , T, end)
  iter = 0
  while not end(T, iter)
    iter = iter + 1
    s' = randomly select from N(s)
    if f(s') < f(s)
      s = s'
    else
      p = exp(-(f(s') < f(s))/T)
      p' = random(0, 1)
      if p' < p
        s = s'
  return s
```

Content

- Constrained Discrete Optimization & Metaheuristics
- Single Point Search Algorithms
- Population-Based Algorithms
    - Genetic algorithm

Population-Based Search

- At each step, more than a single solution is evaluated – we keep the **population of solutions**
    - Between the iterations, the **population** is partially or completely replaced

- **Nature-inspired population-based search algorithms**
    - Draw inspiration from processed in nature / biology
    - **Genetic algorithm** (more generally, **evolutionary algorithms**)
    - Ant Colony Optimization
    - Swarm Optimization
    - Artificial Imunological Systems
    - ...

Genetic Algorithm

- Evolution as inspiration – each **solution** is a **“chromosome”**
- The solutions (chromosomes) with better value of the objective function have higher chances of “survival” and for “reproduction”
- New solutions are created from existing ones via **recombination**
  - The exact recombination operation depends on how chromosomes look like
- Finally, the **mutation** (random change of some value) in the chromosome is possible with some probability
  - Allows for bigger jumps in the solution space and **escaping** local optima

Genetic Algorithm

- Objective function value of the solution f(s) is called **fitness** in GA

- Let S be the size of the **population**
- *end* function determines when the algorithm finishes, based on

  (1) fitness of the **best found solution** or

  (2) average **fitness of the population** or

  (3) number of iterations

```
genetic_algorithm(S,end)
  p = create_init_population(S)
  iter = 0
  evaluate(p)
  while not end(p, iter)
    iter = iter + 1
    p’ = recombine(p)
    mutate(p’)
    evaluate(p’)
    p = select(p ∪ p’)

  return p
```

Genetic Algorithm: **Chromosome**

- **Q:** How do we represent one candidate solution as a **chromosome**
  - Depends on the problem
- **Travelling salesman problem**
  - A chromosome is a vector of n-1 values: **X**: $x_2, \ldots, x_n$
    - Because x_1 and x_{n+1} are fixed (the start/end city is given)
  - **Fitness** of the chromosome? $f(\mathbf{X})$: $d(x_1, x_2) + d(x_2, x_3) + \ldots + d(x_{n-1}, x_n) + d(x_n, x_{n+1})$
  - **Population initialization**
    - Randomly generate a sample of S different vectors, each with all n-1 numbers (but in different order), without repeating the numbers?
    - **Q:** How many such vectors are there?
    - **Q:** Write an algorithm for `create_init_population(S)`!

Genetic Algorithm: **Recombination**

- TSP, toy example: 10 cities, start and end in city 1
- Two example **chromosomes**
    - Chromosome #1: [7, 2, 8, 9, 4, 10, 3, 5, 6]
    - Chromosome #2: [3, 6, 5, 10, 6, 7, 4, 2, 8]
- **Recombination** (also called **crossover**) needs to create „children” chromosomes (one or more) from the „parent” chromosomes
    - The children must also be valid solutions for the problem
    - For TSP that means no repetition of cities!

Genetic Algorithm: **Recombination**

- Parents

  [7, 2, 8, 9, 4, 10, 3, 5, 6]

  [3, 6, 5, 10, 6, 7, 4, 2, 8]

- **Common crossover operators**
  - **Single-point crossover**: select (typically randomly) the location at which to cut the chromosomes and „exchange them" → two „child" chromosomes

[7, 2, 8, 10, 6, 7, 4, 2, 8]

[3, 6, 5, 9, 4, 10, 3, 5, 6]

Doesn't work for TSP: repetition of cities!

Genetic Algorithm: **Recombination**

- Parents

  [7, 2, 8, 9, 4, 10, 3, 5, 6]

  [3, 6, 5, 10, 6, 7, 4, 2, 8]

- **Common crossover operators**
  - **2 (or more)-point crossover**: select two or more locations at which to cut the chromosomes and „exchange them” → two „child” chromosomes

  [7, 2, 8, 10, 6, 7, 4, 5, 6]

  [3, 6, 5, 9, 4, 10, 3, 2, 8]

Doesn’t work for TSP: repetition of cities!

Genetic Algorithm: **Recombination**

- Parents

  [7, 2, 8, 9, 4, 10, 3, 5, 6]

  [3, 6, 5, 10, 6, 7, 4, 2, 8]

- **Common crossover operators**
  - **Uniform crossover**: each bit is selected randomly (50% chance, typically, or proportionally based on parents' fitness)

[7, 2, 5, 10, 4, 7, 4, 5, 8]

[3, 6, 8, 9, 6, 10, 3, 2, 6]

Doesn't work for TSP: repetition of cities!

Genetic Algorithm: **Recombination**

- Parents

  [7, 2, 8, 9, 4, 10, 3, 5, 6]
  [3, 6, 5, 10, 6, 7, 4, 2, 8]

- **Partially mapped crossover**: crossover that works for TSP ☺

  (1) Choose 2 random cuts

  (2) Create mappings from the middle portion

  (3) Copy the rest if it doesn't cause repetition and

  (4) Use mappings to resolve repetitions

[7 2 8 | 9 4 10 | 3 5 6]
[3 6 5 | 10 6 7 | 4 2 8]

Mappings: 9 <-> 10, 4 <-> 6, 10 <-> 7

[x x x | 10 6 7 | x x x]
[x x x | 9 4 10 | x x x]

Copy everything that doesn't cause repetition

[x 2 8 | 10 6 7 | 3 5 x]
[3 6 5 | 9 4 10 | x 2 8]

Use mappings to resolve repetitions

7 → already in, **mapping** 10 <-> 7, but 10 also already in, **mapping** 9 <-> 10, **9 not in!**

[9 2 8 | 10 6 7 | 3 5 4]
[3 6 5 | 9 4 10 | 7 2 8]

Genetic Algorithm: **Mutation**

- Selecting parents for recombination based on fitness → over time, the **populations** will consists of more and more similar chromosomes
- This means the GA is heading towards some local optimum
  - Random **mutations** moves (some) chromosomes from that local region
  - Allow the GA to **escape** the local optima

- Common types of mutation
  - **Element change** → randomly change the value of one chromosome element

    [7, 2, 8, 9, 4, 10, **3**, 5, 6] → [7, 2, 8, 9, 4, 10, **8**, 5, 6]

  - Doesn’t work for TSP!

Genetic Algorithm: **Mutation**

- Selecting parents for recombination based on fitness → over time, the **populations** will consists of more and more similar chromosomes
- This means the GA is heading towards some local optimum
  - Random **mutations** moves (some) chromosomes from that local region
  - Allow the GA to **escape** the local optima

- Common types of mutation
  - **Element swap** → randomly choose two elements and exchange their values

[7, 2, 8, 9, 4, 10, 3, 5, 6] → [7, 2, 3, 9, 4, 10, 8, 5, 6]

  - Works for TSP!

Genetic Algorithm: **Selection**

- How do we choose the parents which to **recombine**

- Conflicting objectives
  - We want to give **better chances to better chromosomes**
  - But if we always recombine the same few chromosomes, we will very quickly obtain a very **uniform population**
  - We typically try to balance between the two

- If the population becomes too uniform – **diversify**
  - Need a measure for diversity of the population
  - By **increasing the chance of mutation** or
  - **Relaxing the selection pressure** (based on fitness)

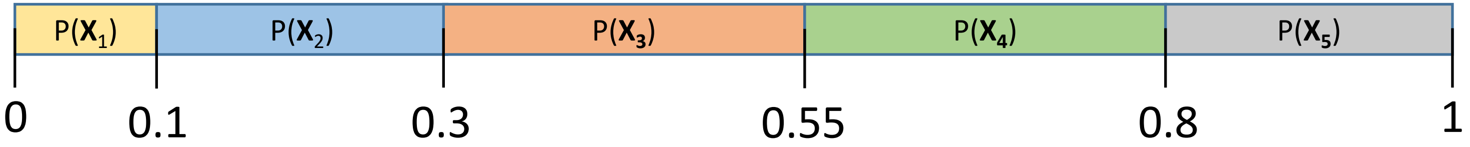
Genetic Algorithm: **Selection**

- Common types of selection:
  - **Roulette wheel**
  - **Tournament**

- **Roulette wheel** (or **proportional**) selection: probability of being selected for reproduction proportional to the fitness of the chromosome

$$P(\mathbf{X}_i) = f(\mathbf{X}_i) / \sum_j^S f(\mathbf{X}_j)$$

- Let us have a population of 5 chromosomes and let
  - $f(\mathbf{X}_1) = 10$, $f(\mathbf{X}_2) = 20$, $f(\mathbf{X}_3) = 25$, $f(\mathbf{X}_4) = 25$, $f(\mathbf{X}_5) = 20$ → convert into probabilities



Genetic Algorithm: **Selection**

- Common types of selection:
  - **Roulette wheel**
  - **Tournament**

- **Tournament selection**
  - Select randomly N chromosomes and find the best among them (with best *f*)
  - To get two parents, we can:
    - Run two tournaments, select **winner** from each
    - Run one tournament, select **two best** chromosomes
  - **Selection pressure**: defined by N: if N is big, more pressure
    - **Q:** what if N = S?

- **Elitism**: placing (keeping) one or more best chromosomes from the previous population into the next population
  - Keeping the best solution found throughout the search

Takeaways

- **Discrete optimization algorithms** (aka **combinatorial optimization**) search over a large space of states
  - Each state if one possible **solution**
  - **Q:** differences w.r.t. state space search problems?
- Metaheuristic strategies define **how to search** through this large space, in order to find the **good/near-optimal solution**
  - **Single-point search**
    - Single solution examined in each iteration
    - **Q:** How does **simulated annealing** avoid local optima?
  - **Population-based search**
    - A population of possible solutions, changed between iterations
    - **Q:** How does **genetic algorithm** avoid local optima?

Questions?

Awọn ibeere?

ਸਵਾਲ?

Küsimusi?

Turite klausimų?

Pitanja?

Sorusu olan?

Dúvidas?

質問は？

有问题吗？

Fragen?

¿Preguntas?

Frågor?

Questions?

Pytania?

Domande?

Питання?

Vragen?

Ερωτήσεις;

Porandukuéra?

คำถาม?

أسئلة؟