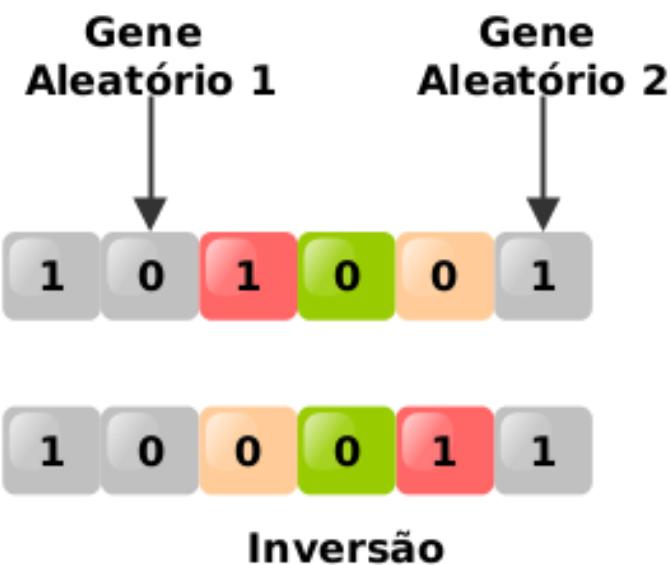
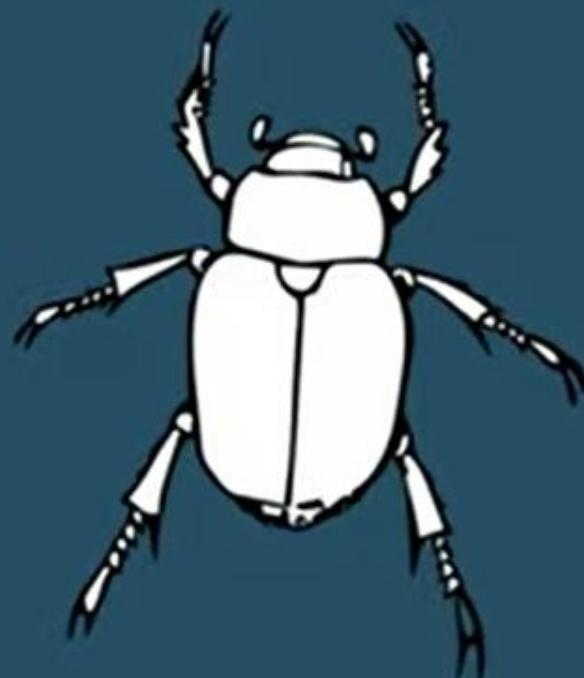


# Algoritmo Genético



# Seleção natural



# Seleção natural



# Seleção natural



# Seleção natural



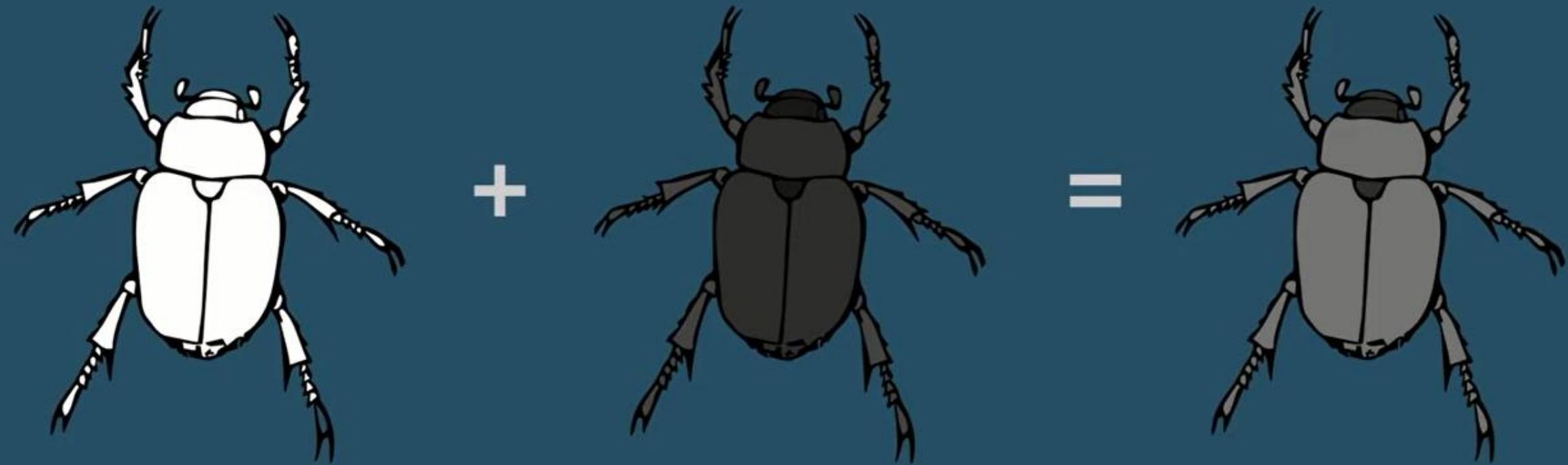
# Seleção natural



# Seleção natural

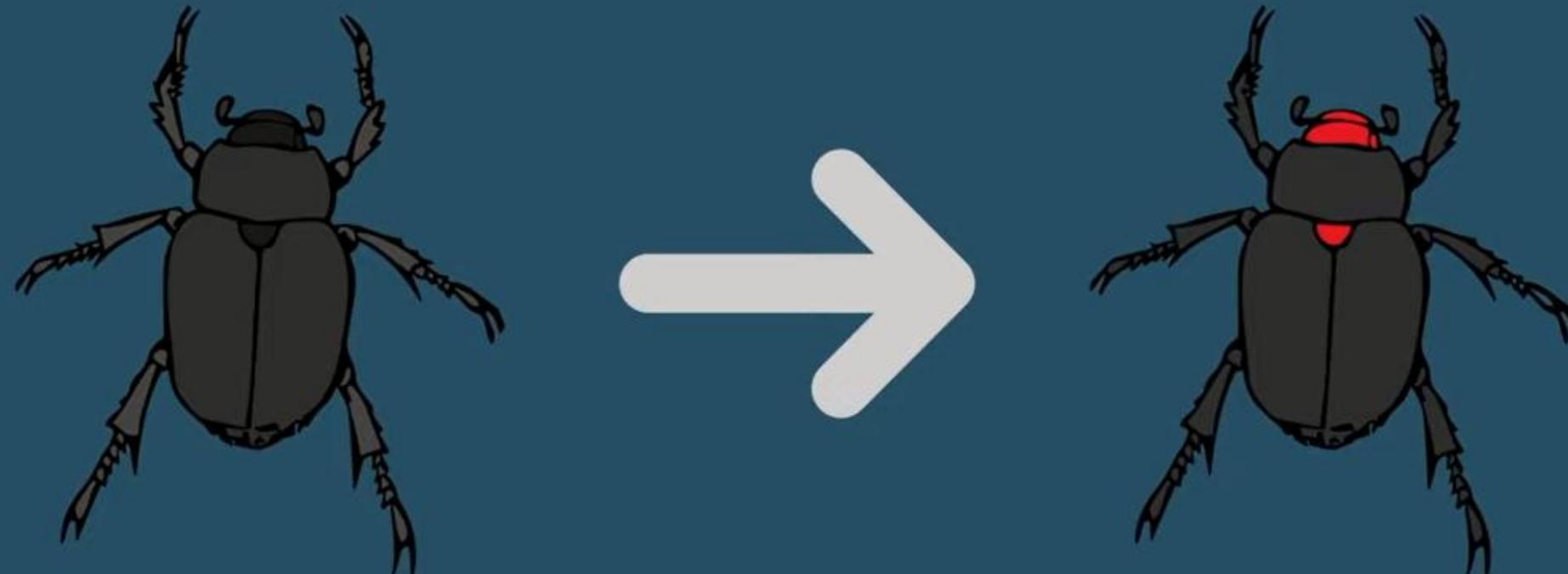


# Cross-over



MISTURA OS GENES DOS PAIS DE MANEIRA ALEATÓRIA

# Mutação



ADICIONAR VARIEDADE A POPULAÇÃO

# Como isso vai me ajudar?

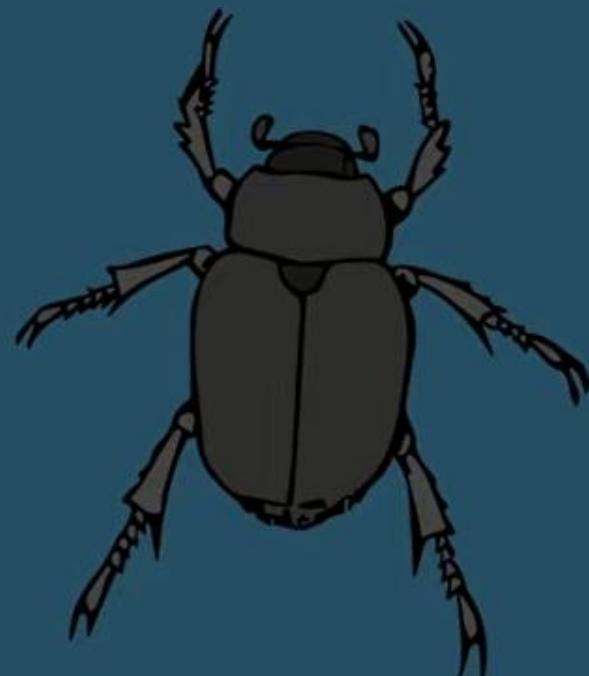
## 1. Otimização de Caminhos e Navegação

- Em robôs autônomos, os AGs podem ser usados para encontrar **rotas ótimas** em ambientes dinâmicos e desconhecidos, minimizando tempo e energia consumida.
- Em **enxames de robôs**, os AGs ajudam a coordenar a movimentação para evitar colisões e otimizar a distribuição espacial.

## 2. Coordenação de Sistemas Multirrobóticos

- Em grupos de robôs, os AGs podem otimizar **táticas colaborativas**, como formação de patrulha ou divisão de tarefas em ambientes de exploração.
- Permitem que os robôs evoluam estratégias de comunicação eficientes sem programação explícita.

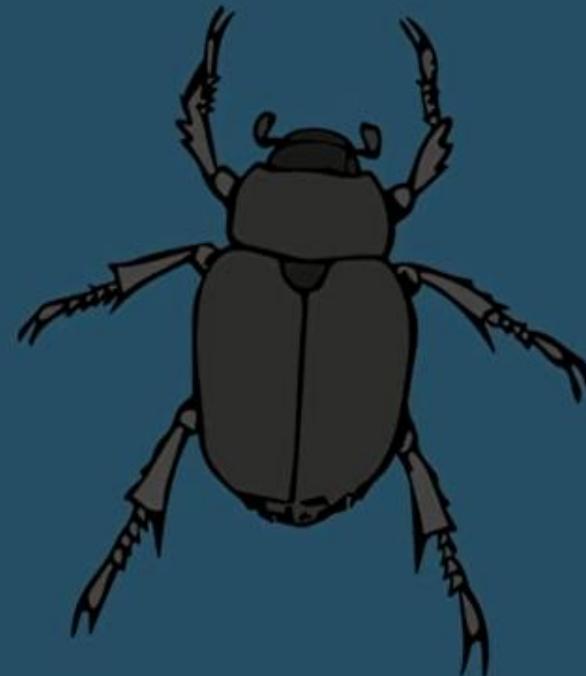
# Modelo



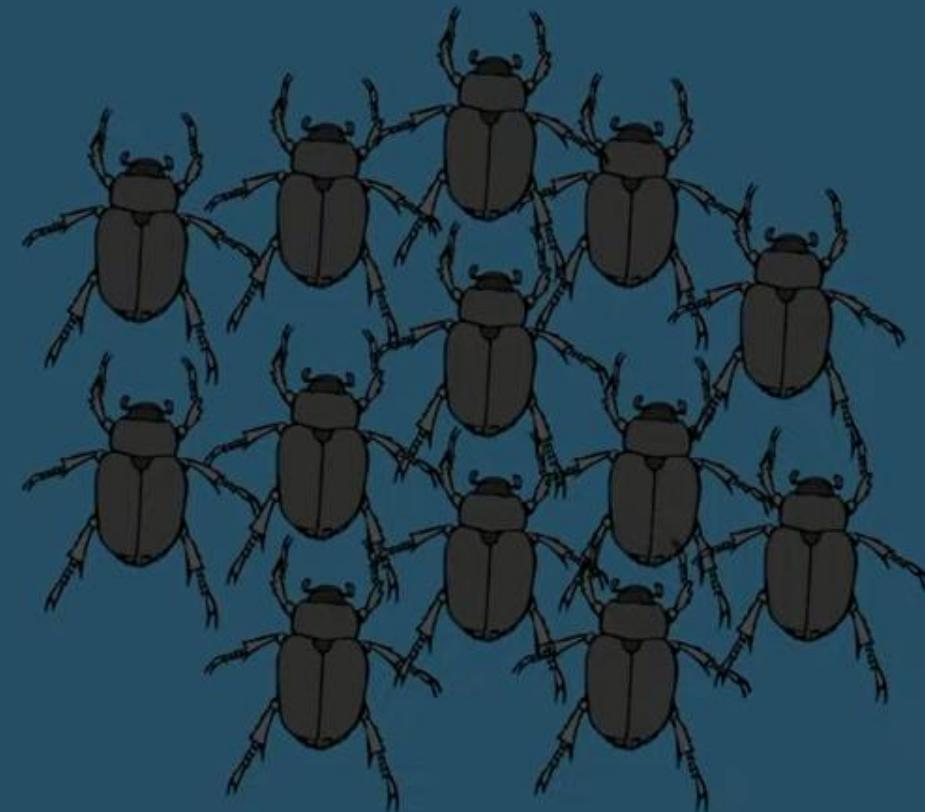
= R G B

0 a 255	0 a 255	0 a 255
---------	---------	---------

# Terminologia

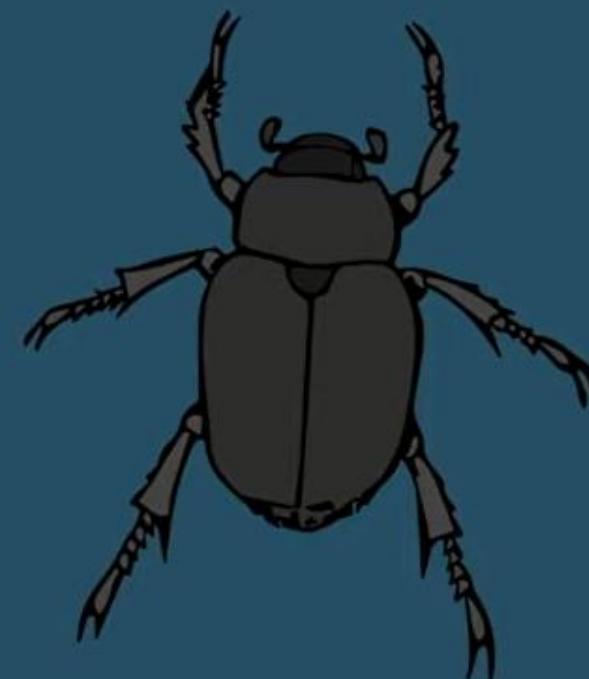


Indivíduo

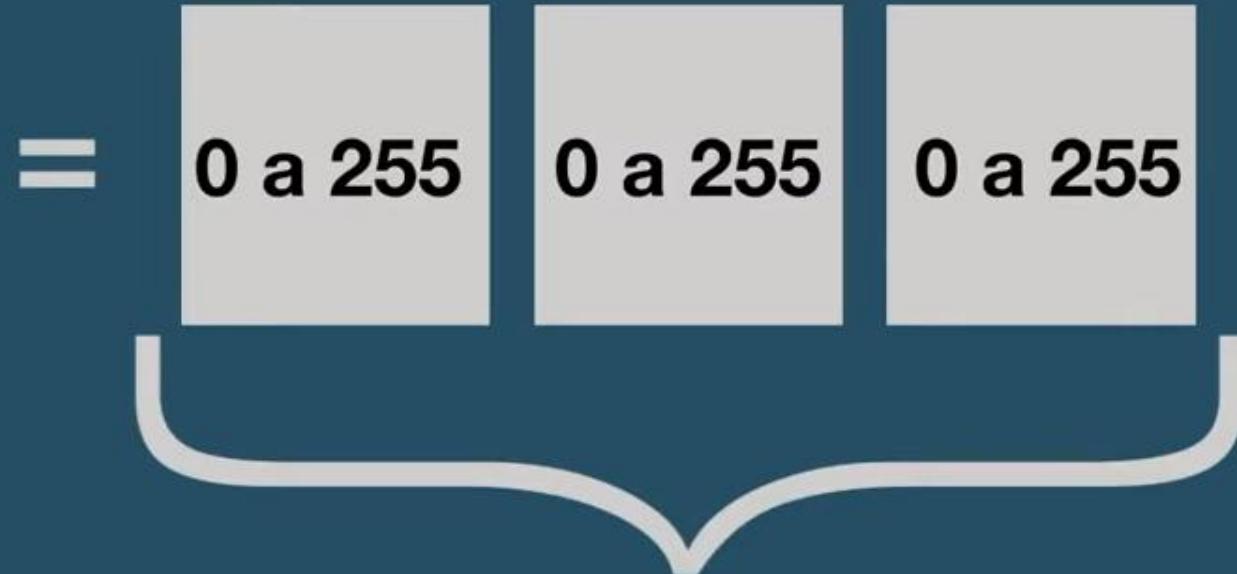


População

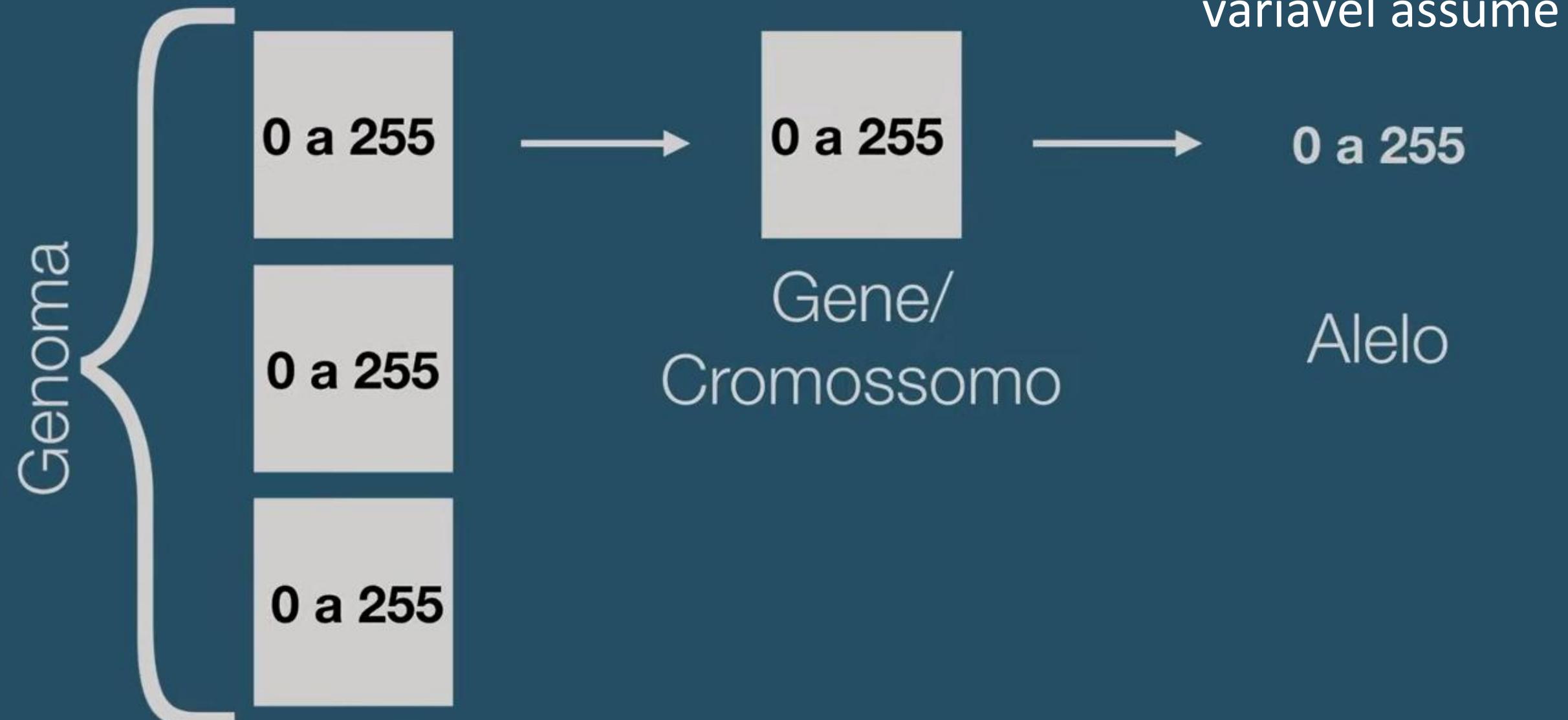
# Terminologia



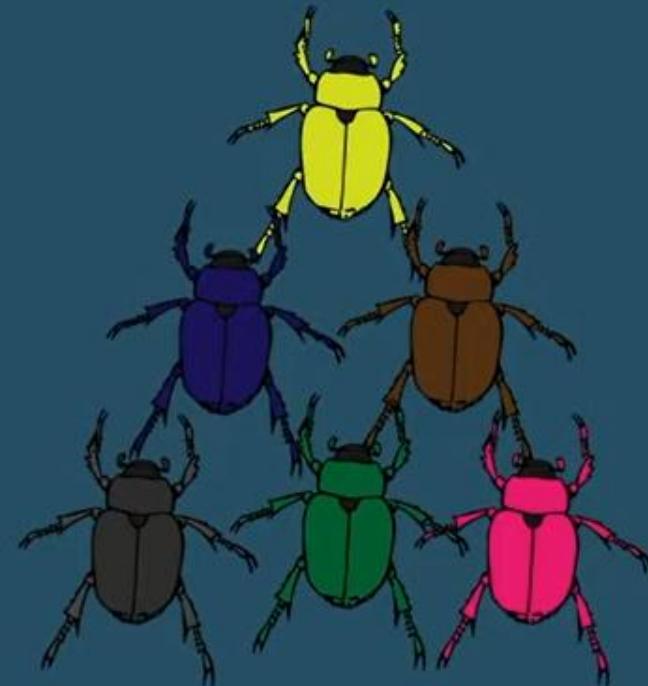
Indivíduo



# Terminologia

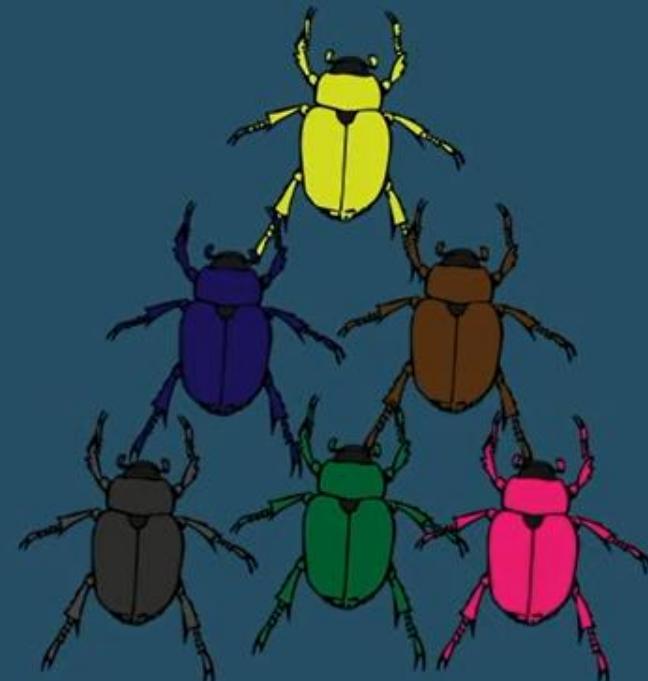


# Início do algoritmo



População

# Início do algoritmo



População

$$f(\text{yellow beetle}) = 0.22$$
$$f(\text{dark brown beetle}) = 0.95$$

Fitness Function

$$f(\text{beetle}) = 0.95$$

$$f(\text{beetle}) = 0.83$$

$$f(\text{beetle}) = 0.78$$

$$f(\text{beetle}) = 0.67$$

$$f(\text{beetle}) = 0.22$$

$$f(\text{beetle}) = 0.14$$

Avaliação

$$f(\text{Beetle}) = 0.95$$

$$f(\text{Beetle}) = 0.83$$

$$f(\text{Beetle}) = 0.78$$

$$f(\text{Beetle}) = 0.67$$

$$f(\text{Beetle}) = 0.22$$

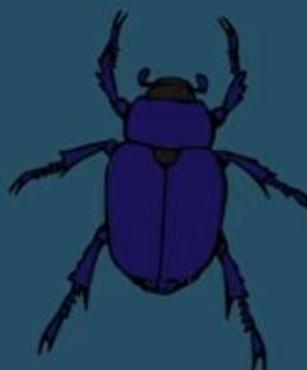
$$f(\text{Beetle}) = 0.14$$

Avaliação

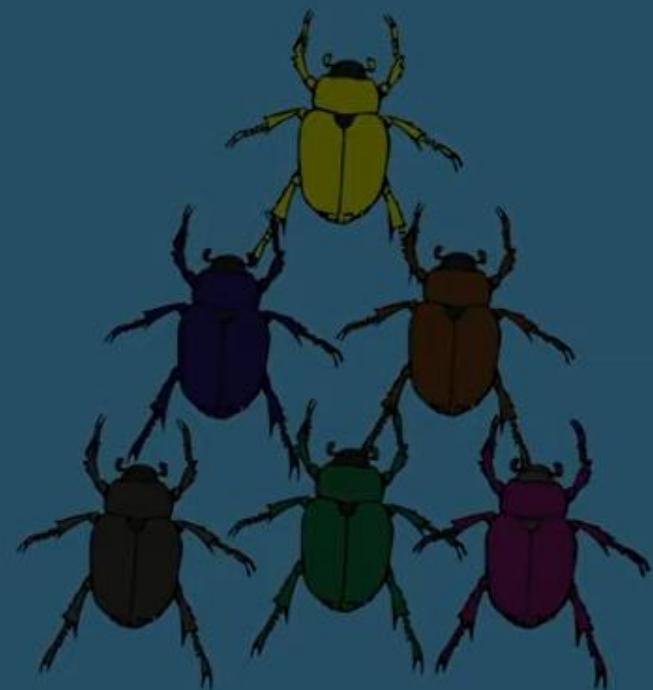
Seleção

Cross-over

Mutação



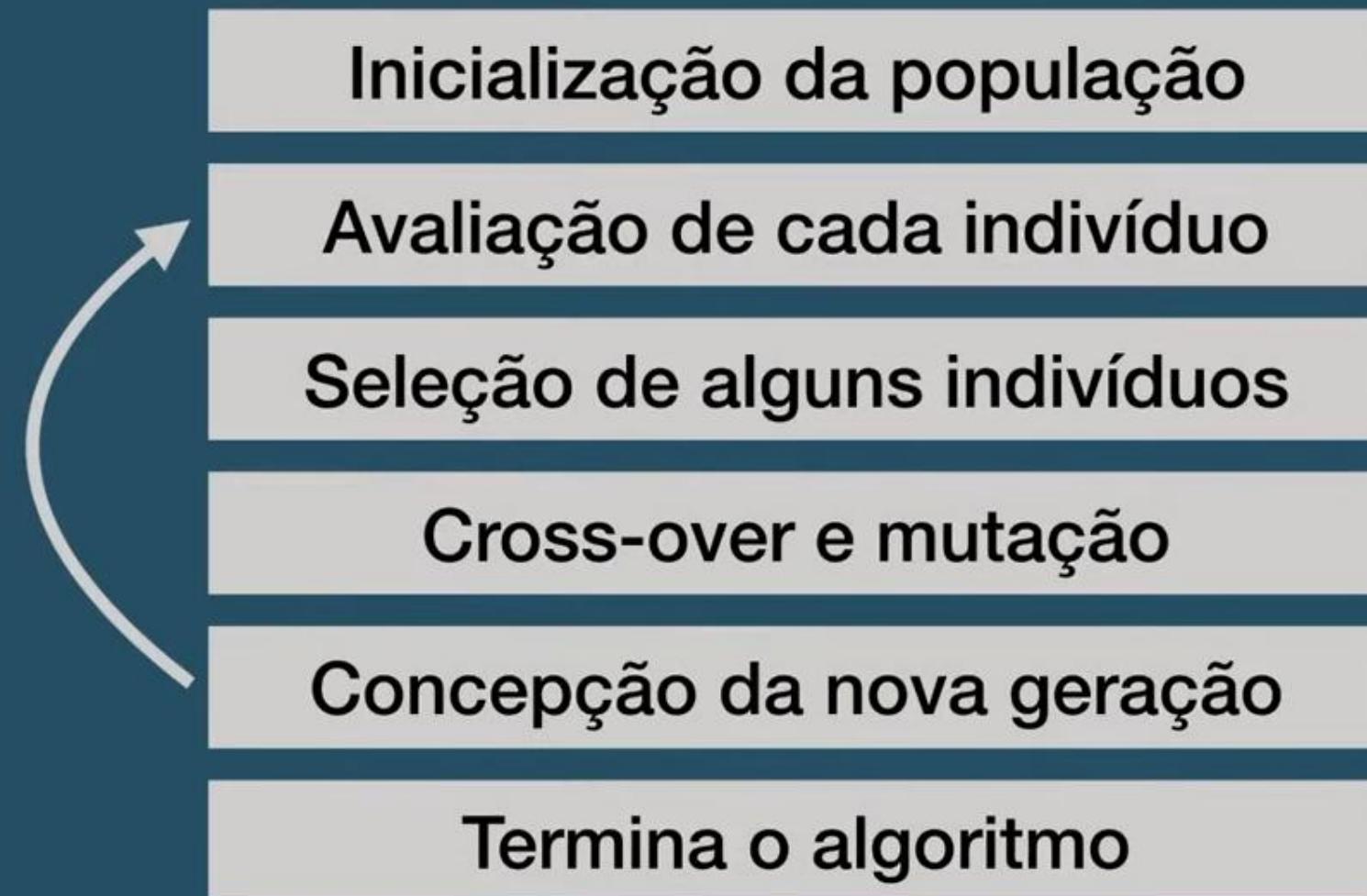
# Fim do algoritmo



Nova geração

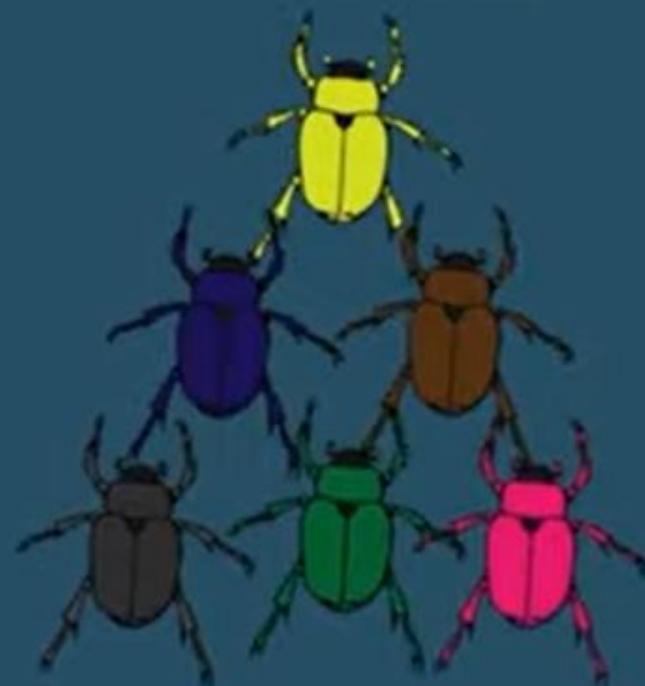
# Estrutura do algoritmo genético

Repita até estar satisfeito com as soluções



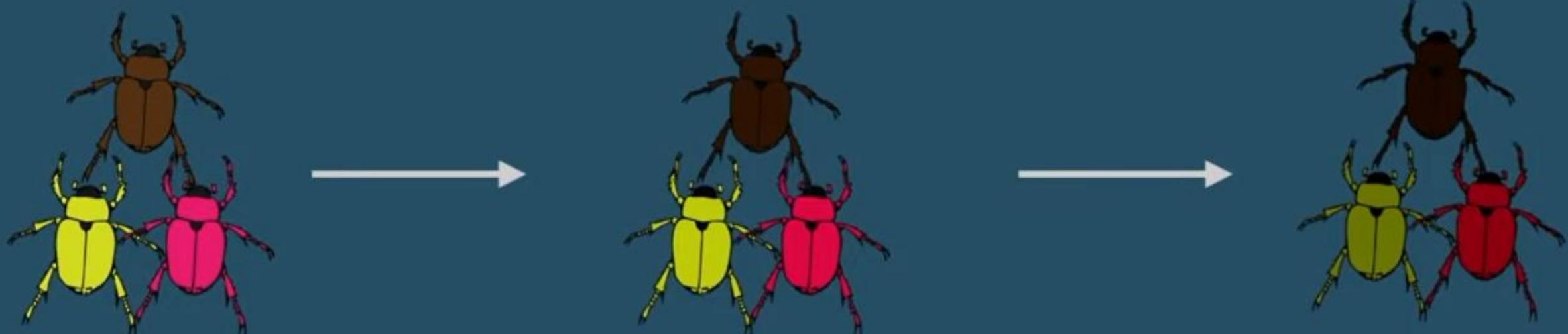
# Estrutura do algoritmo genético

Inicialização da população



# Estrutura do algoritmo genético

## Inicialização da população



Se a população for muito pequena

# Estrutura do algoritmo genético

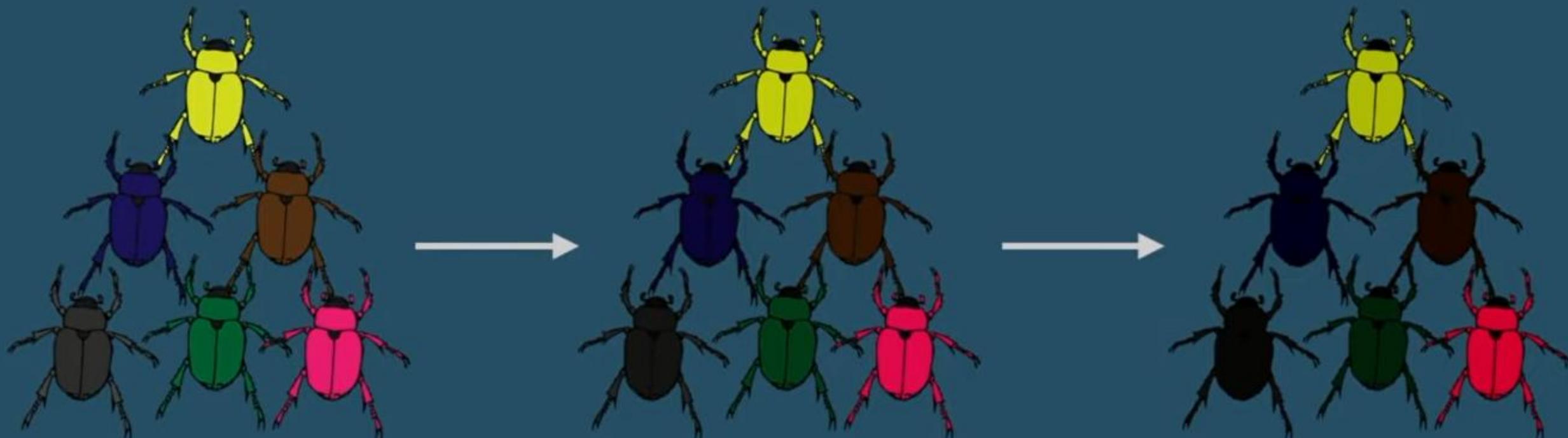
Inicialização da população



Máximo local

# Estrutura do algoritmo genético

## Inicialização da população



# Estrutura do algoritmo genético

Inicialização da população



Máximo global

# Estrutura do algoritmo genético

Inicialização da população



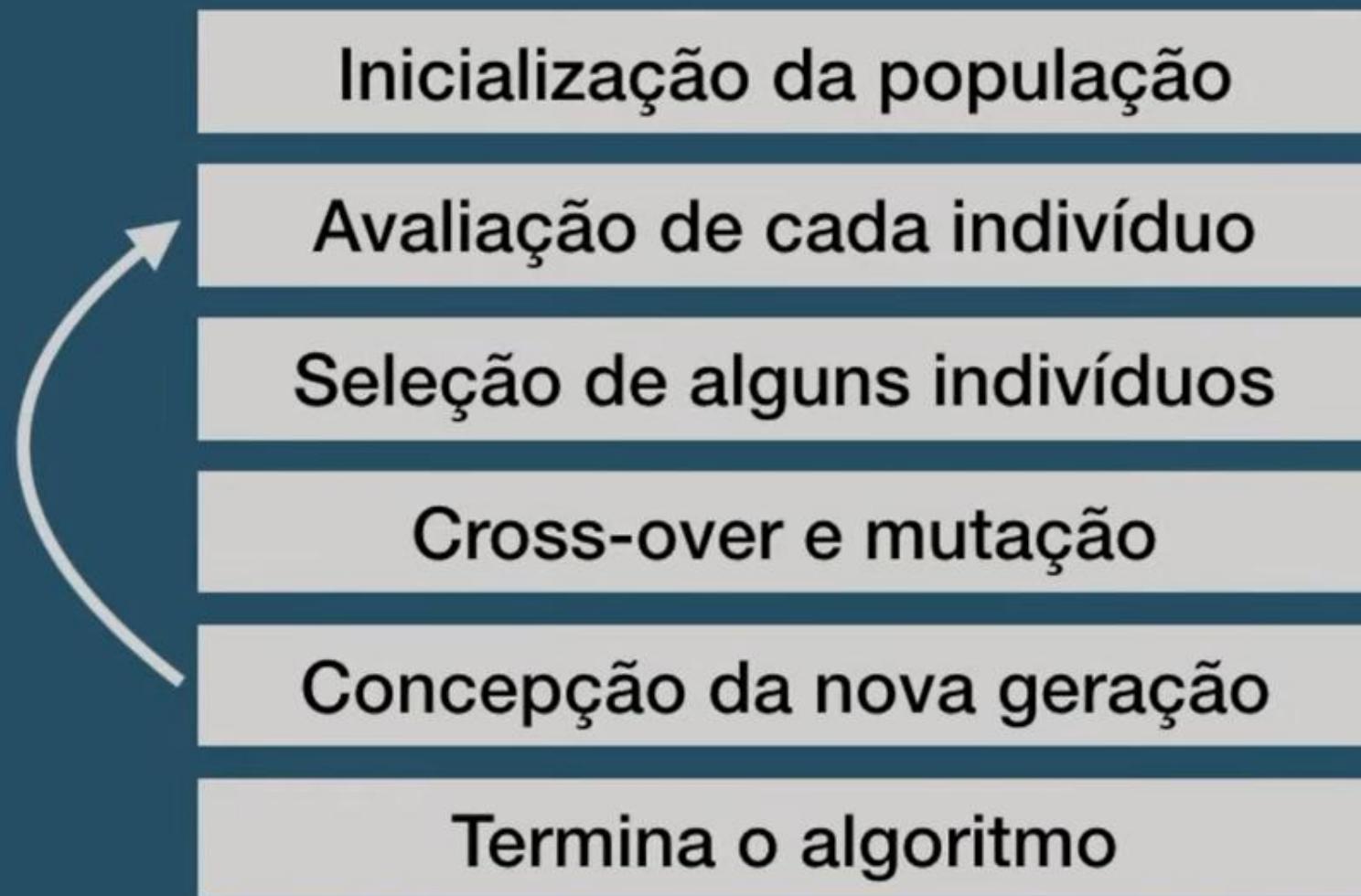
Máximo local



Máximo global

# Estrutura do algoritmo genético

Repita até estar satisfeito com as soluções



# Estrutura do algoritmo genético

Avaliação de cada indivíduo

# Estrutura do algoritmo genético

Avaliação de cada indivíduo

Fitness Function

# Estrutura do algoritmo genético

Avaliação de cada indivíduo

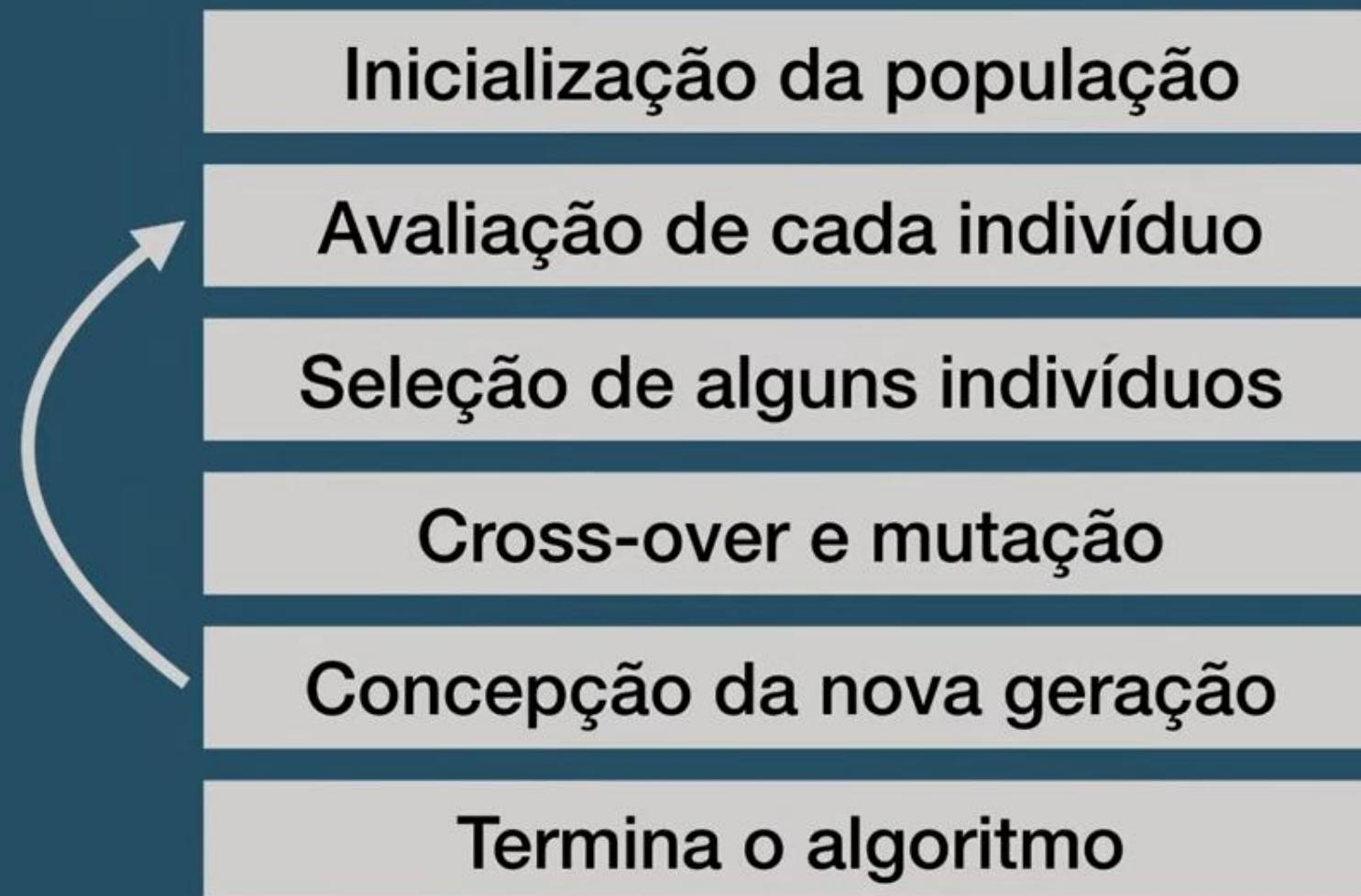
Fitness Function

$$f(\text{beetle}) = 0.95$$

$$f(\text{yellow beetle}) = 0.22$$

# Estrutura do algoritmo genético

Repita até estar satisfeito com as soluções



# Estrutura do algoritmo genético

## Seleção de alguns indivíduos

É importante selecionar não apenas os melhores indivíduos para diversificar as soluções e evitar máximos locais

# Estrutura do algoritmo genético

## Seleção de alguns indivíduos

$$f(\text{bug 1}) = 0.95$$

$$f(\text{bug 2}) = 0.83$$

$$f(\text{bug 3}) = 0.78$$

$$f(\text{bug 4}) = 0.67$$

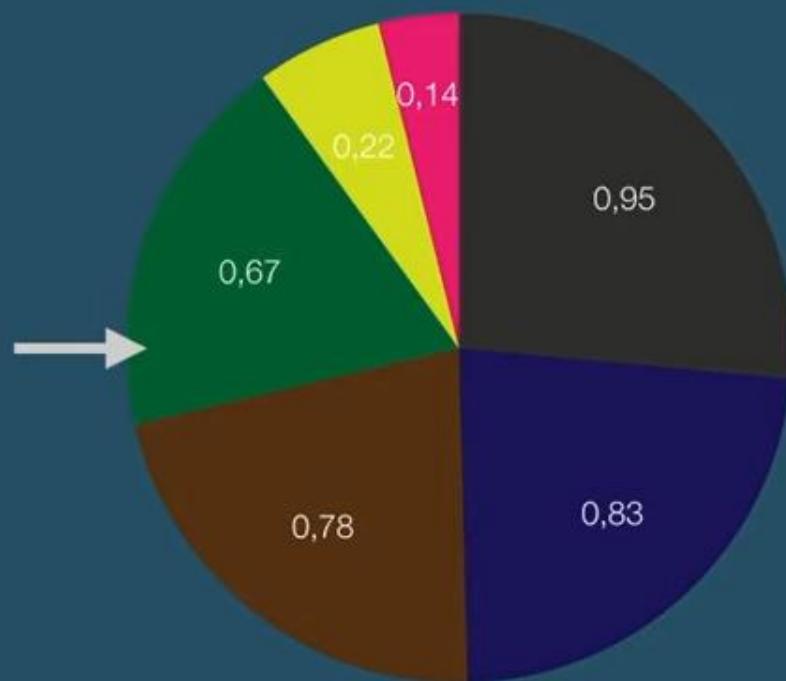
$$f(\text{bug 5}) = 0.22$$

$$f(\text{bug 6}) = 0.14$$

# Estrutura do algoritmo genético

## Seleção de alguns indivíduos

$f(\text{bug 1}) = 0.95$   
 $f(\text{bug 2}) = 0.83$   
 $f(\text{bug 3}) = 0.78$   
 $f(\text{bug 4}) = 0.67$   
 $f(\text{bug 5}) = 0.22$   
 $f(\text{bug 6}) = 0.14$



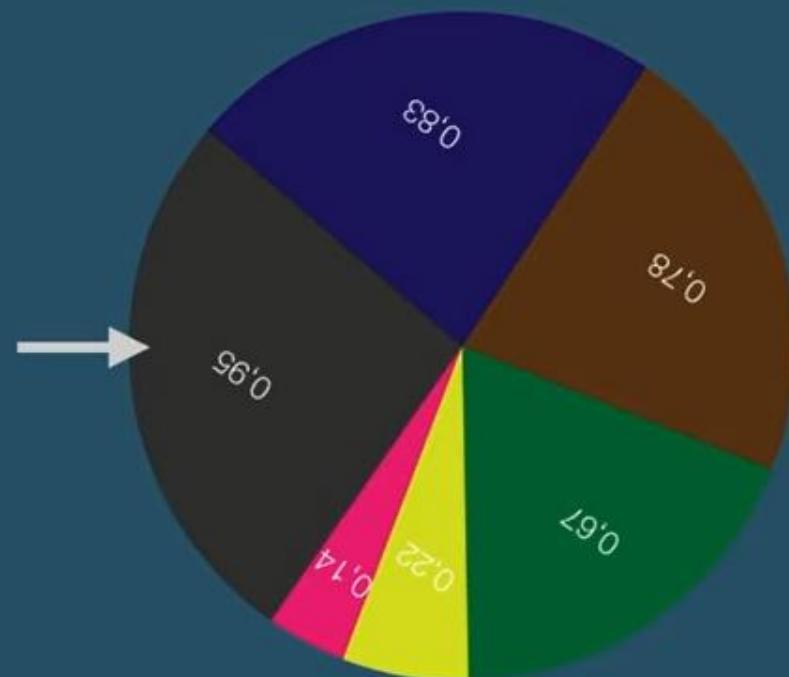
Seleção proporcional  
ao fitness

↓  
os indivíduos tem  
chance de serem  
escolhidos proporcional  
ao seu fitness value

# Estrutura do algoritmo genético

## Seleção de alguns indivíduos

$f(\text{escarabajo}) = 0.95$   
 $f(\text{escarabajo azul}) = 0.83$   
 $f(\text{escarabajo laranja}) = 0.78$   
 $f(\text{escarabajo verde}) = 0.67$   
 $f(\text{escarabajo amarelo}) = 0.22$   
 $f(\text{escarabajo rosa}) = 0.14$



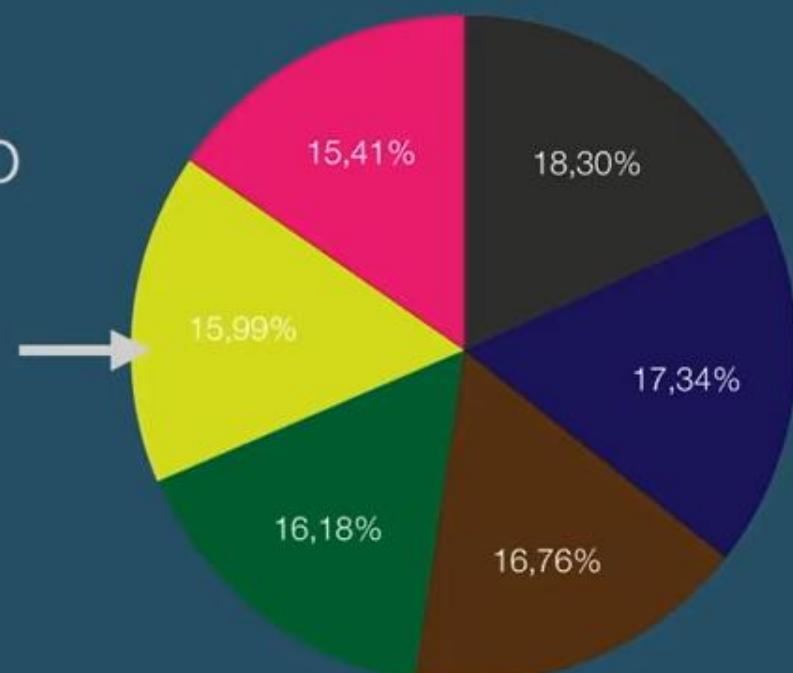
Seleção proporcional  
ao fitness

↓  
os indivíduos tem  
chance de serem  
escolhidos proporcional  
ao seu fitness value

# Estrutura do algoritmo genético

## Seleção de alguns indivíduos

Problemático se os valores de fitness são muito parecidos



Seleção proporcional ao **fitness**

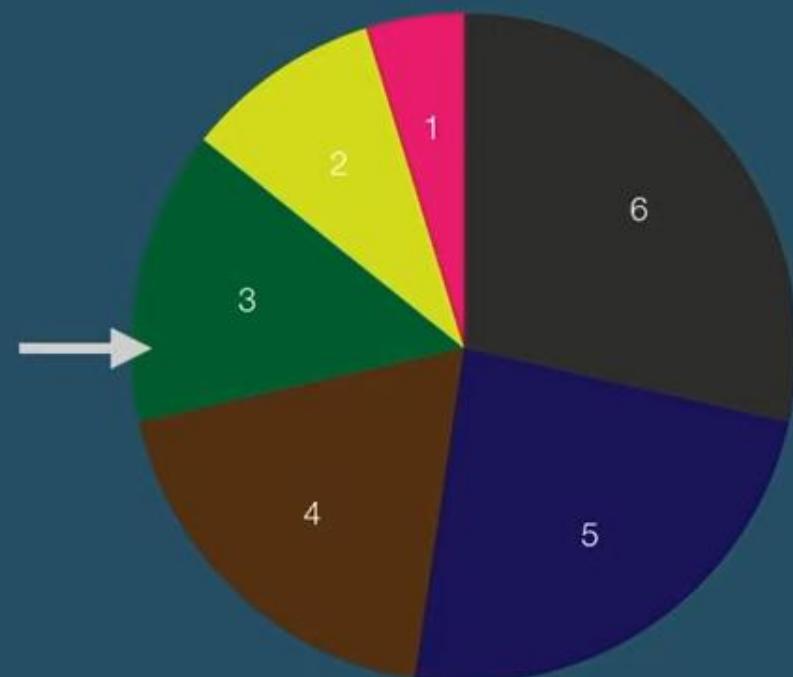
↓

os indivíduos tem chance de serem escolhidos proporcional ao seu fitness value

# Estrutura do algoritmo genético

## Seleção de alguns indivíduos

- 1)  $f(\text{bug 1}) = 0.95$
- 2)  $f(\text{bug 2}) = 0.83$
- 3)  $f(\text{bug 3}) = 0.78$
- 4)  $f(\text{bug 4}) = 0.67$
- 5)  $f(\text{bug 5}) = 0.22$
- 6)  $f(\text{bug 6}) = 0.14$

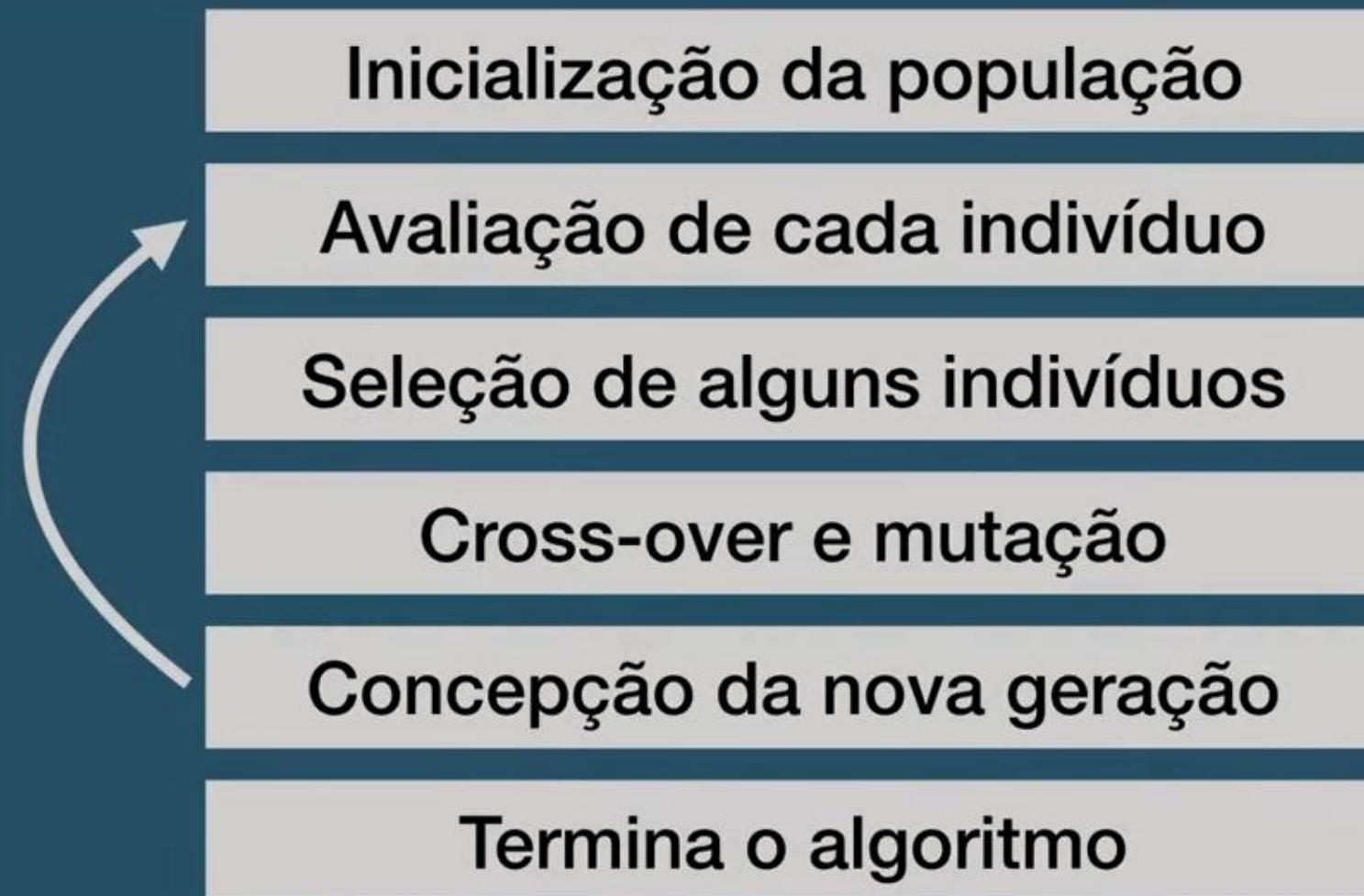


Seleção proporcional  
ao **ranking**

↓  
os indivíduos tem  
chance de serem  
escolhidos proporcional  
ao seu ranking dentre  
os indivíduos

# Estrutura do algoritmo genético

Repita até estar satisfeito com as soluções



# Estrutura do algoritmo genético

Cross-over e mutação

# Estrutura do algoritmo genético

## Cross-over



=

51	51	51
----	----	----

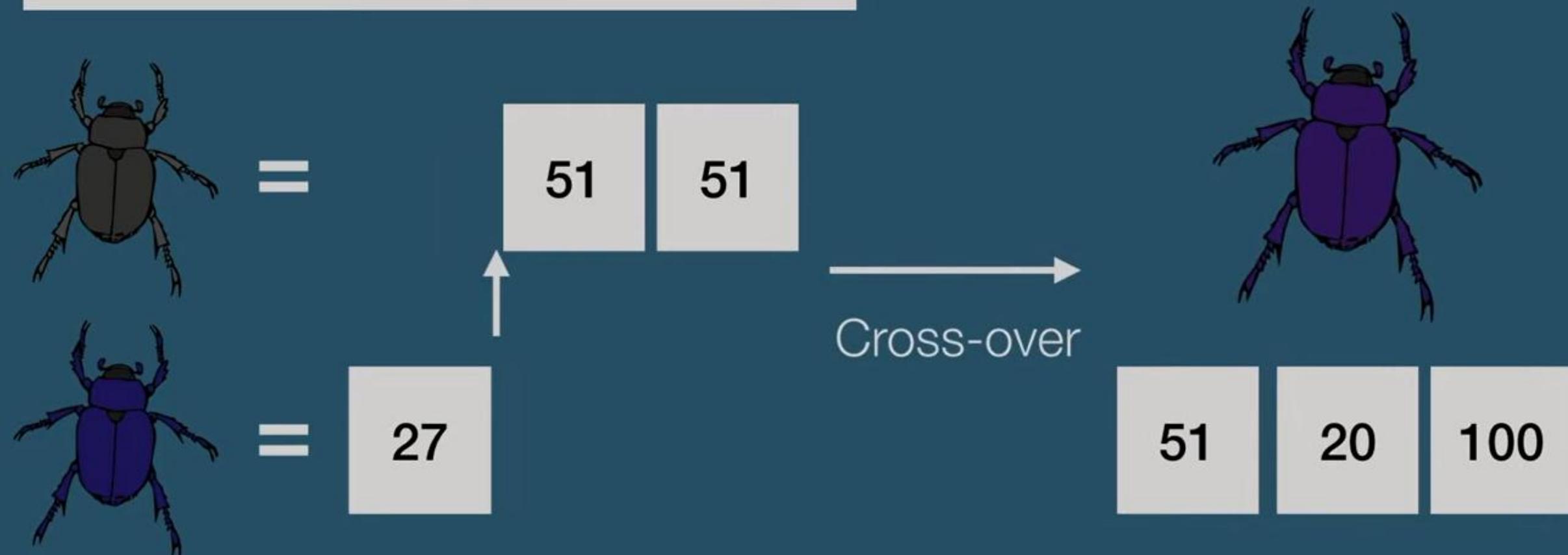


=

27	20	100
----	----	-----

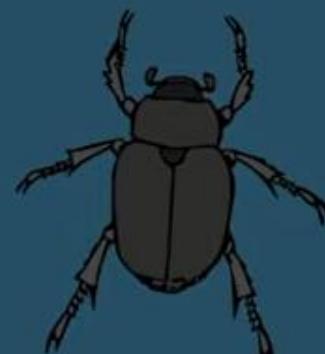
# Estrutura do algoritmo genético

## Cross-over



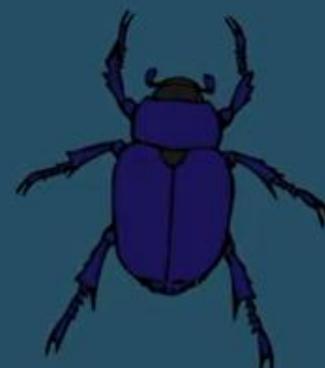
# Estrutura do algoritmo genético

## Cross-over



=

51	51	51
----	----	----



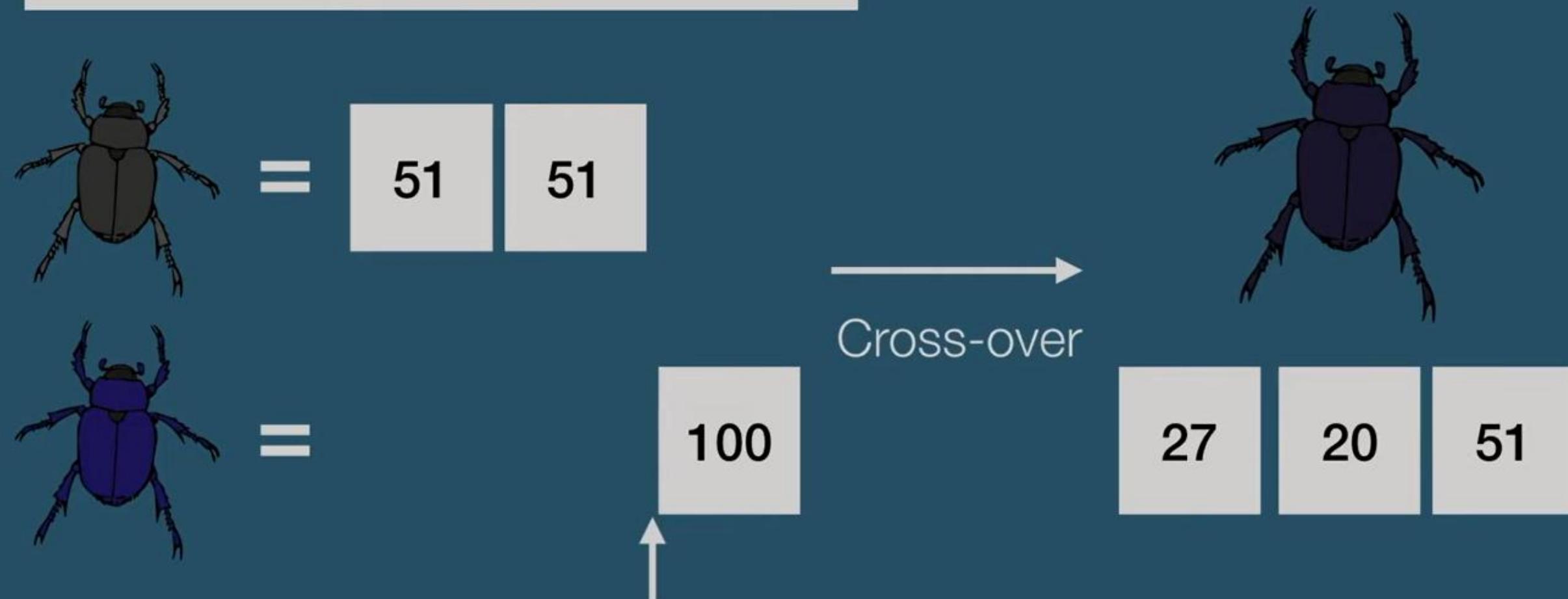
=

27	20	100
----	----	-----



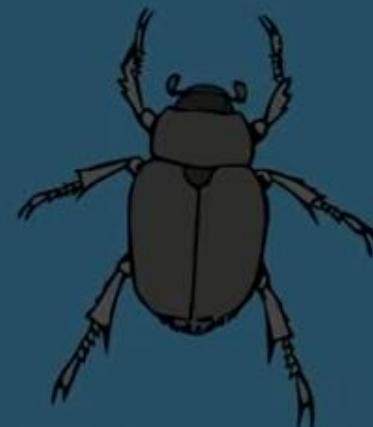
# Estrutura do algoritmo genético

## Cross-over



# Estrutura do algoritmo genético

## Mutação



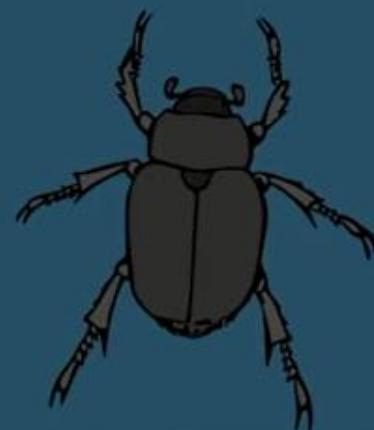
51

51

51

# Estrutura do algoritmo genético

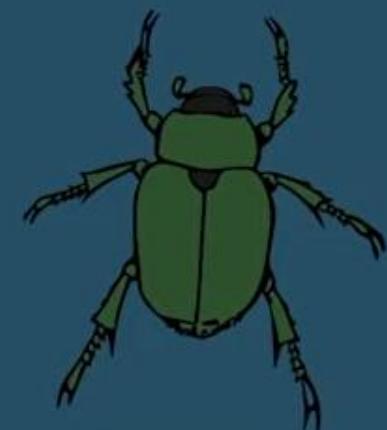
## Mutação



51	51	51
----	----	----



Mutação

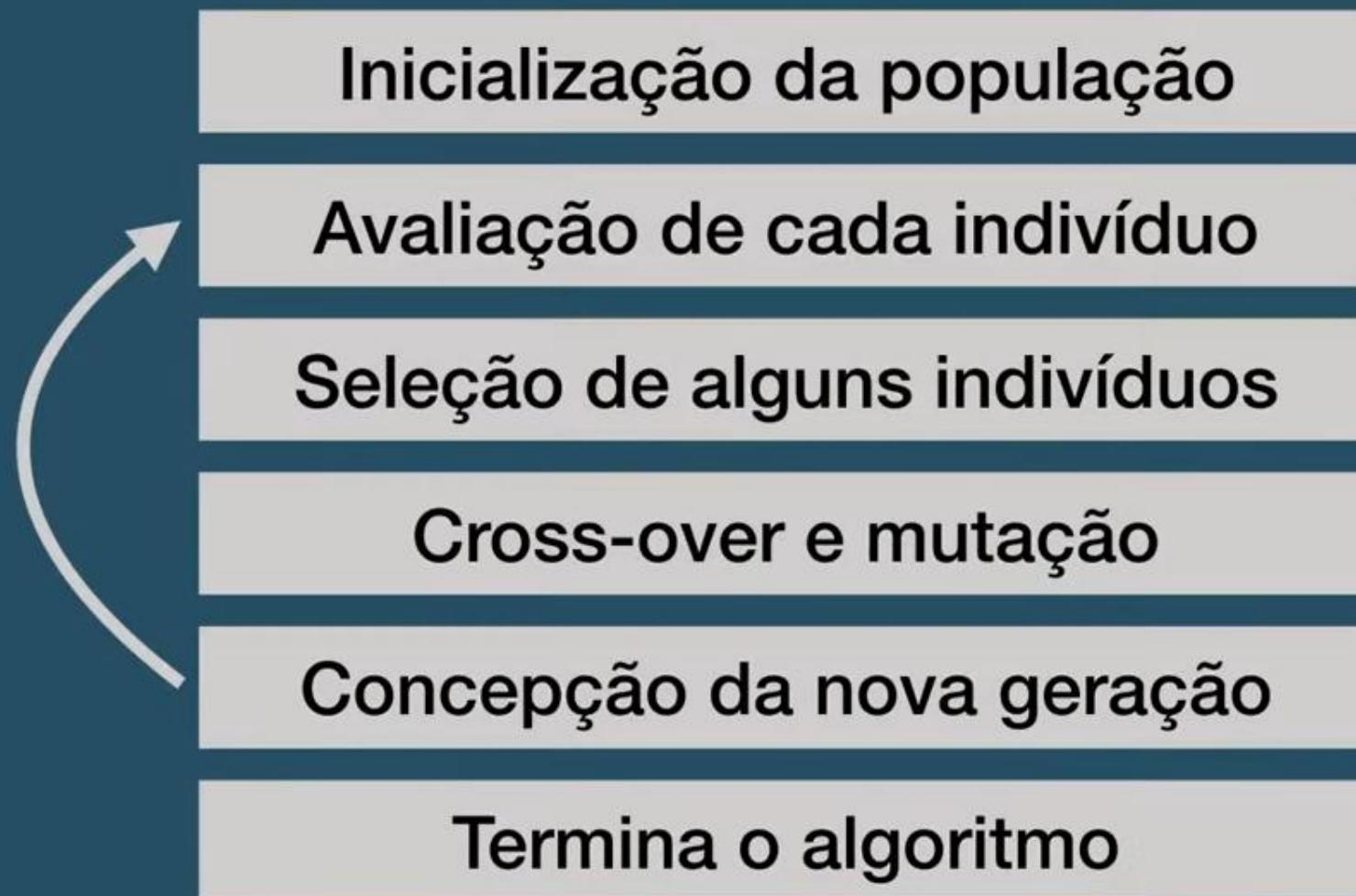


51	91	51
----	----	----

Altera o valor de um  
gene aleatoriamente

# Estrutura do algoritmo genético

Repita até estar satisfeito com as soluções



# Estrutura do algoritmo genético

Concepção da nova geração

$$f(\text{Beetle}) = 0.95$$

$$f(\text{Beetle}) = 0.83$$

$$f(\text{Beetle}) = 0.78$$

$$f(\text{Beetle}) = 0.67$$

$$f(\text{Beetle}) = 0.22$$

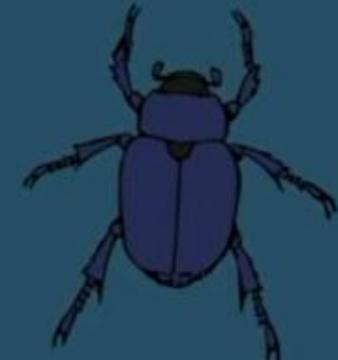
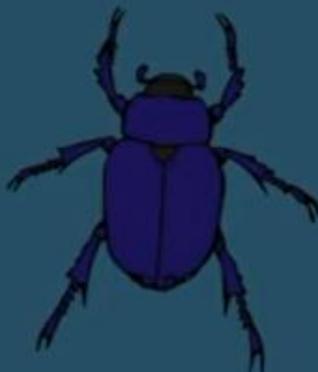
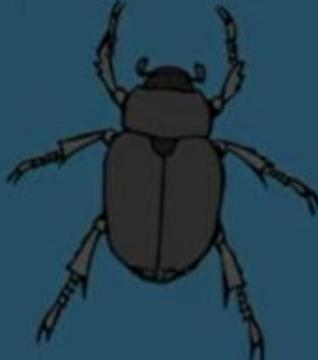
$$f(\text{Beetle}) = 0.14$$

Avaliação

Seleção

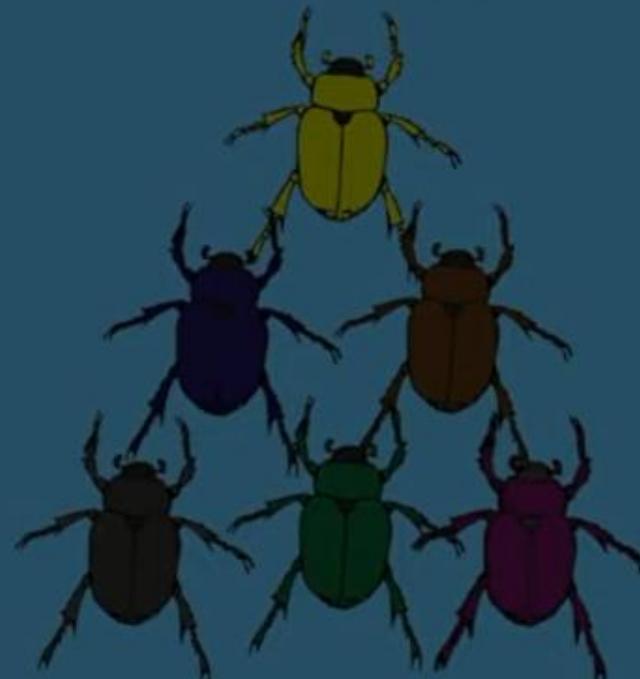
Cross-over

Mutação



# Estrutura do algoritmo genético

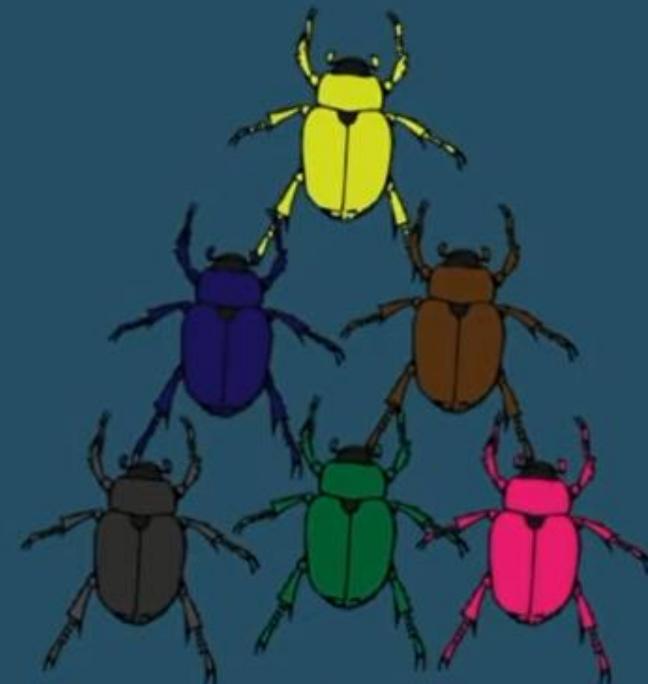
Concepção da nova geração



Nova geração

# Estrutura do algoritmo genético

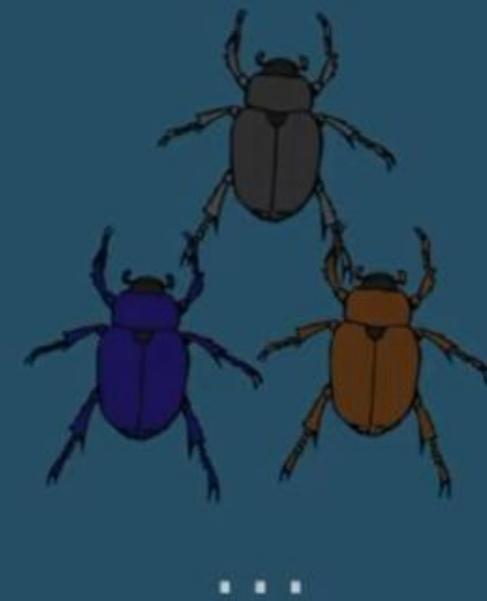
Concepção da nova geração



Geração atual

Manter os melhores

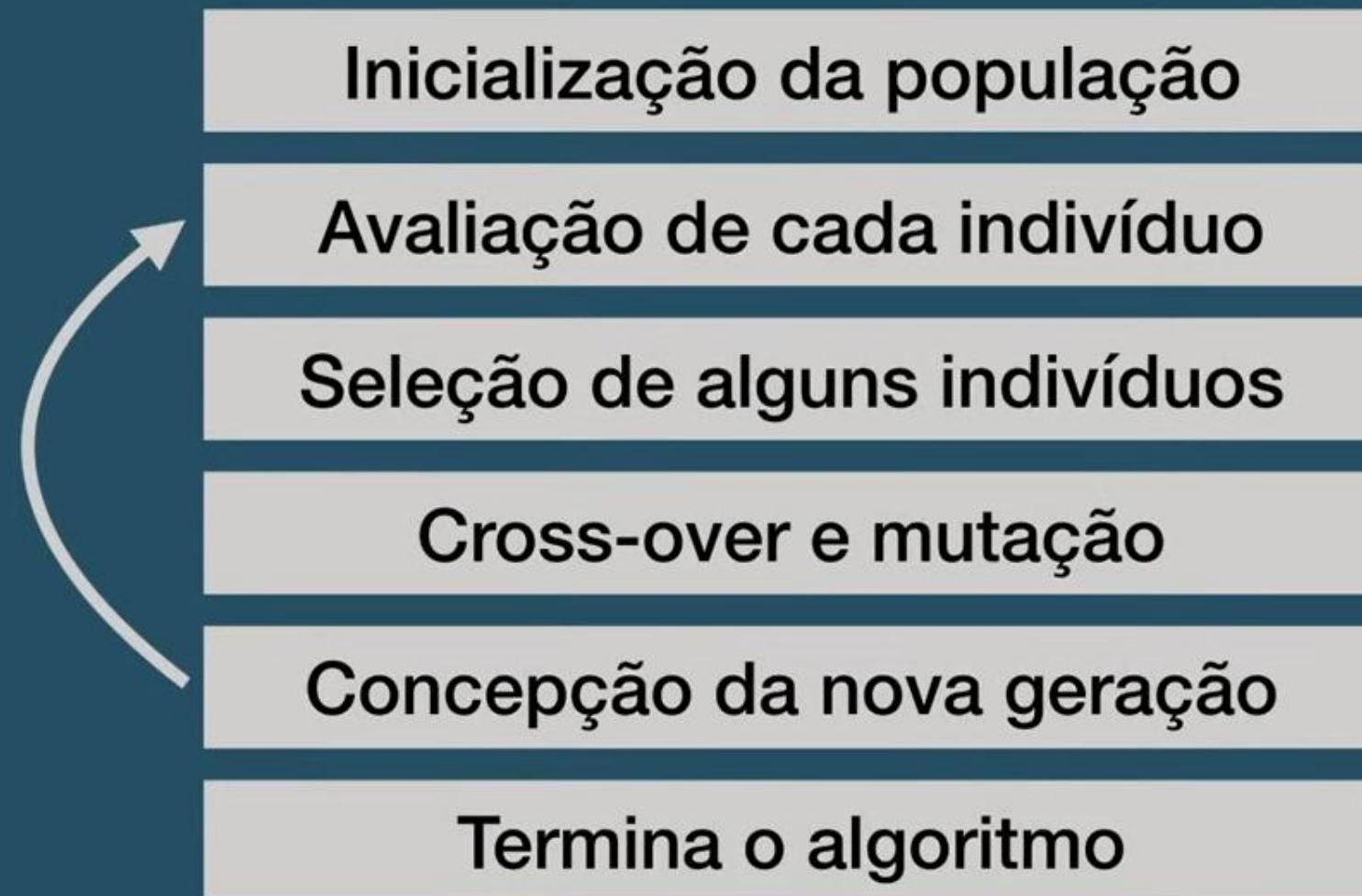
→  
Elitismo



Próxima geração

# Estrutura do algoritmo genético

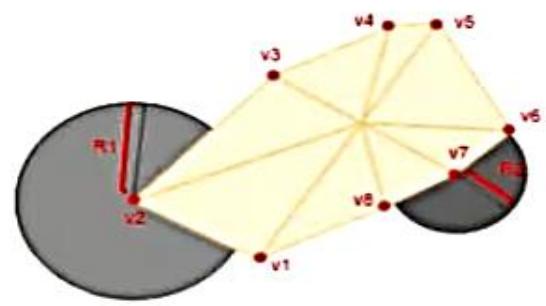
Repita até estar satisfeito com as soluções



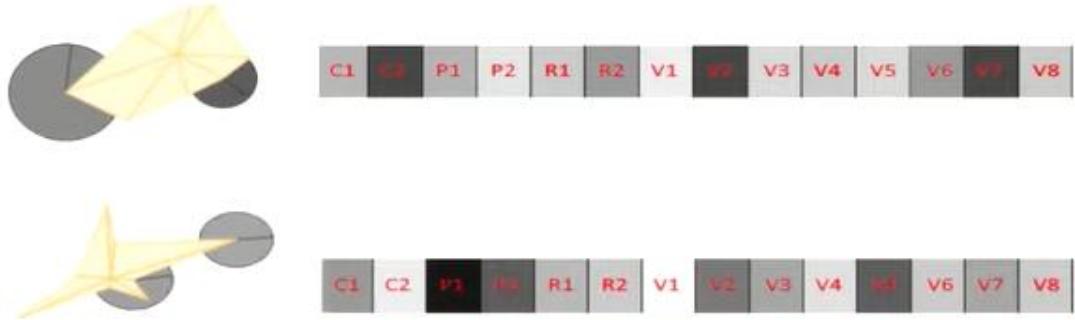
# Estrutura do algoritmo genético

**Termina o algoritmo**

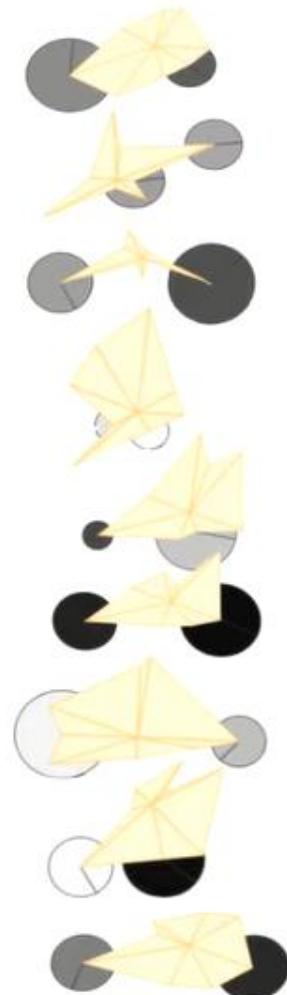
- 1) Terminar o algoritmo quando atingir uma meta
- 2) Terminar o algoritmo quando não houver mais progresso



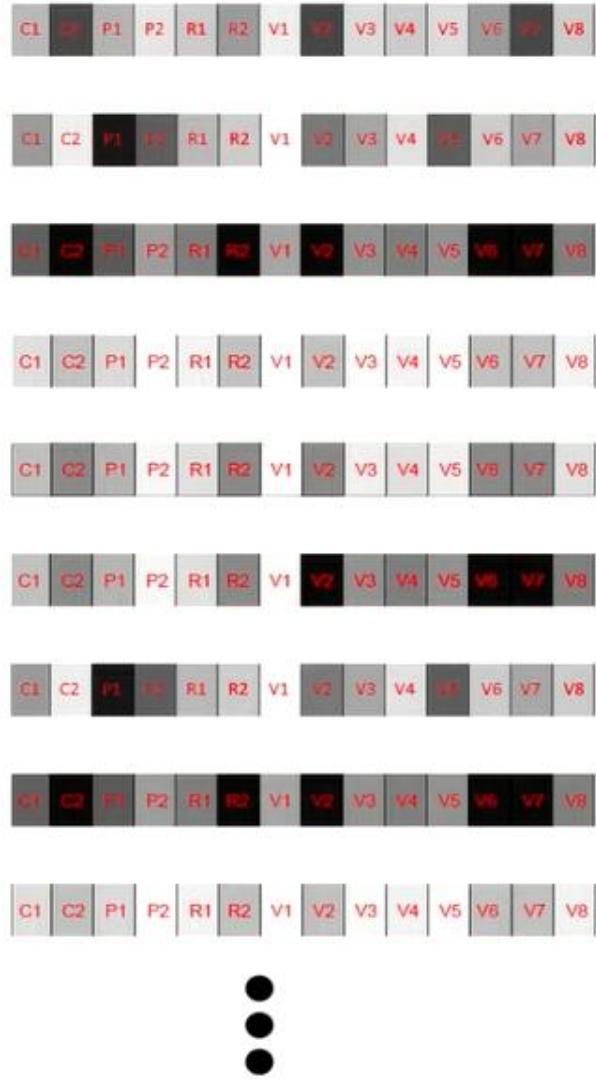
v4 v5  
v3  
R1 v6  
v7 R2  
P1 v2  
C1 v8 C2  
v1 P2

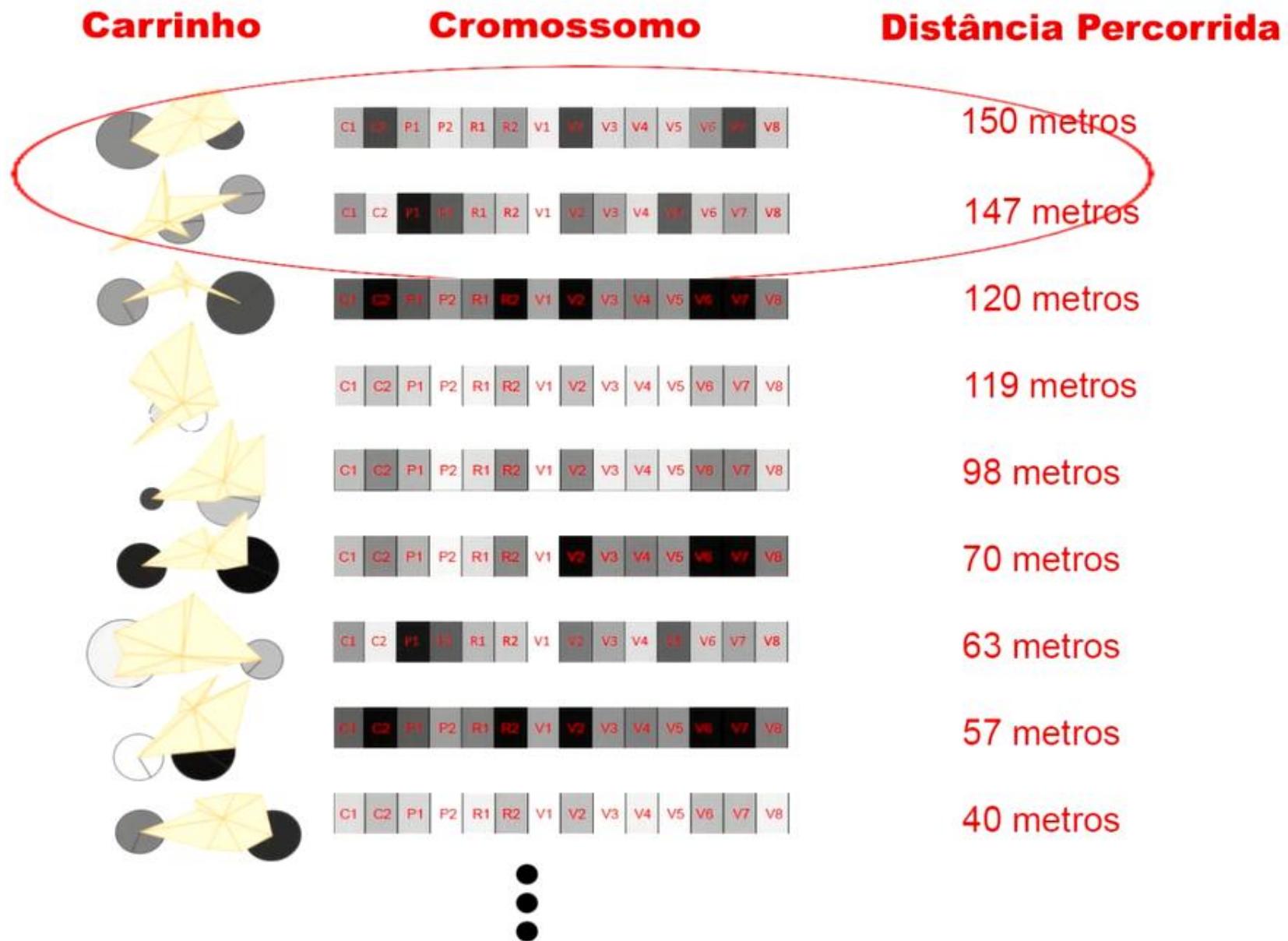


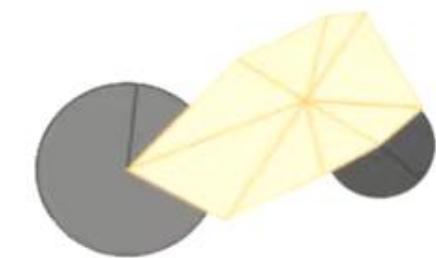
## Carrinho

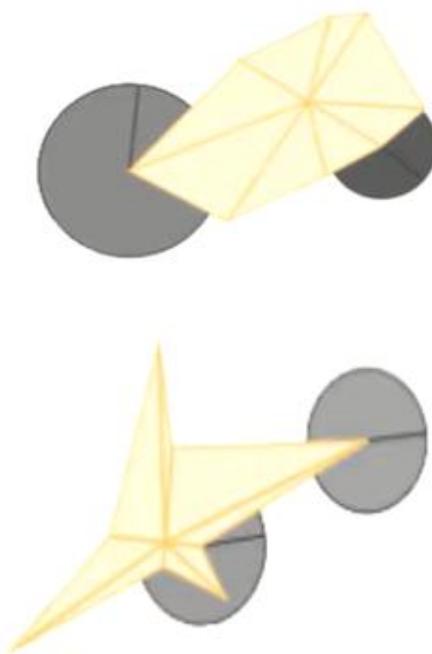


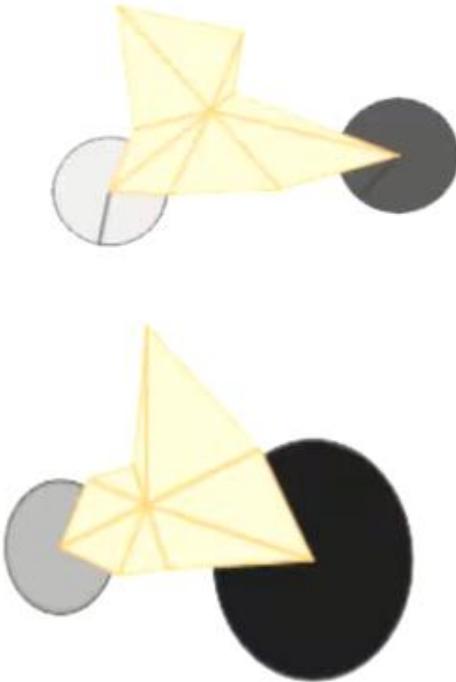
## Cromossomo

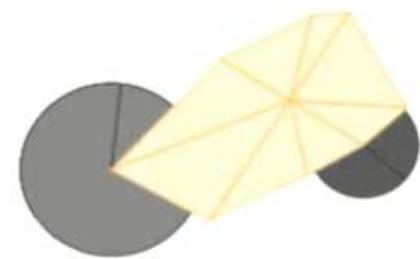
















# O que são Algoritmos Genéticos?

- Os **Algoritmos Genéticos (AGs)** são inspirados na **biologia evolutiva**, ou seja, imitam processos como **seleção natural, hereditariedade, cruzamento e mutação** para resolver problemas complexos. Eles são muito usados para encontrar **soluções aproximadas** para problemas difíceis, especialmente quando não há um algoritmo eficiente que forneça a resposta exata em tempo viável.

# Exemplo prático: Problema do Caixeiro Viajante

- Um problema clássico onde **Algoritmos Genéticos** podem ser usados é o **Problema do Caixeiro Viajante (TSP - Travelling Salesman Problem)**.
- **O que é esse problema?**
  - Imagine que um vendedor precisa visitar várias cidades, **passando por cada uma apenas uma vez e retornando ao ponto inicial**, percorrendo a **menor distância possível**. Como existem muitas combinações possíveis, encontrar a melhor solução de forma exata pode ser inviável para um número grande de cidades.
- **Como representar esse problema?**
  - As **cidades** são representadas como **vértices** de um grafo.
  - As **conexões entre cidades** são as **arestas**, com uma **distância (custo) associada**.
  - O desafio é encontrar o **caminho mais curto possível** que passe por todas as cidades **exatamente uma vez**.

# Implementação no código

No código apresentado, algumas etapas são mostradas:

## 1. Importação de bibliotecas

- A biblioteca **DEAP** é usada para Algoritmos Genéticos.
- numpy** e **random** ajudam na criação da matriz de distâncias entre cidades.

## 2. Geração do grafo (matriz de distâncias)

A função graphTSP(numCities, minDist, maxDist) cria uma **matriz bidimensional**, onde:

- Cada **linha e coluna** representam uma cidade.
- Os valores da matriz são as **distâncias entre as cidades**, escolhidas aleatoriamente entre **10 e 100**.

## 3. Entrada do usuário

O usuário escolhe **quantas cidades** deseja considerar (mínimo 5). O código então cria o grafo e imprime a matriz de distâncias.

# Indivíduos e População

Nos **Algoritmos Genéticos**, cada solução possível é chamada de **indivíduo**, que pode ser representado como uma **sequência de genes** (um possível caminho para o caixeiro).

- **Genótipo:** A sequência de cidades visitadas. Exemplo: [0, 2, 1, 4, 3].
- **Fenótipo:** O percurso real no mapa, considerando as distâncias.
- **População:** O conjunto de indivíduos que serão analisados para encontrar a melhor solução.

# 1 Importação de Bibliotecas

```
import random  
import numpy
```

- **random**: Biblioteca padrão do Python usada para gerar números aleatórios.
- **numpy**: Biblioteca usada para lidar com arrays e operações matemáticas, útil para manipular matrizes e vetores.

# Instalação da Biblioteca DEAP

```
!pip install deap # Instalação de DEAP
```

- O DEAP (Distributed Evolutionary Algorithms in Python) é uma biblioteca para **criar Algoritmos Evolutivos**, incluindo Algoritmos Genéticos (AGs).
- O comando **!pip install deap instala essa biblioteca**. O ! indica que o comando está sendo executado diretamente no terminal (ou em um notebook Jupyter, por exemplo).

# Importação de Módulos da DEAP

```
from deap import algorithms  
from deap import base  
from deap import creator  
from deap import tools
```

A DEAP é dividida em módulos específicos. Vamos ver o que cada um faz:

- **algorithms**: Contém **algoritmos genéticos pré-implementados**, como seleção, mutação e cruzamento.
- **base**: Fornece **estruturas básicas** para criar indivíduos e definir o comportamento do algoritmo.
- **creator**: Permite **criar classes personalizadas**, como indivíduos e funções de avaliação.
- **tools**: Contém **funções auxiliares** como seleção de pais, cruzamento, mutação e avaliação.

O código está preparando o ambiente para trabalhar com **Algoritmos Genéticos** no Python. Ele:

- 1** Importa bibliotecas essenciais (random e numpy).
- 2** Instala a biblioteca **DEAP**, caso ainda não esteja instalada.
- 3** Importa os módulos necessários da **DEAP** para a implementação do Algoritmo Genético.

# O que é o problema do Caixeiro Viajante (TSP)?

- O **TSP** consiste em encontrar o caminho mais curto que passa por **todas as cidades** exatamente **uma vez** e retorna à cidade inicial. Cada cidade está conectada a outras por **arestas com pesos (distâncias)**, formando um **grafo completo**.

O código cria um **grafo de cidades**, onde as distâncias entre elas são geradas aleatoriamente.

📌 Função **graphTSP(numCities, minDist, maxDist)**

```
def graphTSP(numCities, minDist, maxDist):  
    cities = numpy.zeros((numCities, numCities), dtype = int)
```

- Cria uma matriz numCities x numCities inicializada com zeros, representando o grafo.
- A matriz armazenará as distâncias entre as cidades.



# Preenchendo a matriz

```
for i in range(numCities):
    for j in range(numCities):
        if (j > i):
            cities[i, j] = random.randint(minDist, maxDist)
        elif (j < i):
            cities[i, j] = cities[j, i]
```

O **grafo é simétrico**, ou seja, a distância da cidade A → B é a mesma de B → A.

- **Se  $j > i$**  → Sorteia uma distância entre minDist e maxDist.
- **Se  $j < i$**  → Copia o valor da posição espelhada ( $\text{cities}[j, i]$ ).
- **Se  $i == j$**  → A distância de uma cidade para si mesma permanece 0.



## Entrada do número de cidades

```
numCities = 5 # Número inicial de cidades

while(True):
    numCities = int(input('Digite o número de cidades: '))
    if (numCities > 4):
        break
    else:
        print('O número de cidades deve ser maior que 4!')
```

O programa pede ao usuário para inserir um número de cidades **maior que 4**.

- Se o usuário digitar um número menor que 5, ele exibe uma mensagem de erro e pede novamente.



## Gerando e exibindo o grafo

```
cities = graphTSP(numCities, 10, 100)
```

```
print('Grafo:\n', cities)
```

Chama `graphTSP()` para gerar a matriz de distâncias com valores entre 10 e 100.

- ◆ Exibe a matriz no console.

## Exemplo de Saída

Se o usuário digitar 5 cidades, o programa pode gerar um grafo assim:

```
Digite o número de cidades: 5
```

```
Grafo:
```

```
[[ 0 34 72 23 98]
 [34 0 65 49 51]
 [72 65 0 13 99]
 [23 49 13 0 88]
 [98 51 99 88 0]]
```

- Cada número na matriz representa a distância entre duas cidades.

-  O código gera um **grafo completo** onde cada cidade está conectada a todas as outras.
-  A **matriz simétrica** representa as distâncias aleatórias entre as cidades.
-  O usuário escolhe o número de cidades (**mínimo de 5**).

# Conceitos Importantes

- 1 População:** Conjunto de soluções (indivíduos) que será avaliado e evoluído ao longo do algoritmo genético.
- 2 Indivíduo (ou cromossomo):** Cada possível solução do problema.
- 3 Gene:** Um elemento dentro do indivíduo. No caso do TSP, **cada gene representa uma cidade.**
- 4 Genótipo:** A sequência de genes que compõem um indivíduo.
  - Exemplo: [0, 2, 1, 4, 3] representa um caminho possível para um problema com 5 cidades.
- 5 Fenótipo:** Como o genótipo se traduz no mundo real.
  - No TSP, o **fenótipo** é a **rota percorrida** pelo caixeiro viajante.



# Criando a função de avaliação da aptidão

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
```

Isso define a função de fitness para o problema.

- ◆ FitnessMin significa que o objetivo é minimizar a função (menor distância no TSP).
- ◆ base.Fitness:Classe base da DEAP que representa a aptidão (fitness) de um indivíduo.
- ◆ weights=(-1.0,):

Define que o objetivo é minimizar a função de aptidão.-1.0 significa que menores valores são melhores (minimização).

Se fosse 1.0, seria para maximização.



# Criando o tipo de Indivíduo

```
creator.create("Individual", list, fitness=creator.FitnessMin)
```

Isso define que um **indivíduo** será uma **lista de cidades** (list).

- ◆ Cada indivíduo herda a estrutura da classe list.
- ◆ Ele também terá um atributo de fitness associado (fitness=creator.FitnessMin).



# Criando a Caixa de Ferramentas (toolbox)

```
toolbox = base.Toolbox()
```

O toolbox é um conjunto de ferramentas para manipular os indivíduos e a população no algoritmo genético.

# Criando um gerador de genes

```
toolbox.register("attr_int", random.randint, 0, numCities-1)
```

Isso define um **gene** como um número aleatório entre **0 e numCities-1**.

- ◆ Esse número representa **uma cidade** dentro do percurso do caixeiro.



# Criando um indivíduo

```
toolbox.register("individual", tools.initRepeat, creator.Individual, toolbox.attr_int, numCities)
```

Isso cria um indivíduo, que é uma lista de cidades.

- ◆ `tools.initRepeat` gera uma sequência de `numCities` números aleatórios.
- ◆ Ele usa `toolbox.attr_int` para gerar os genes (cidades).
- ◆ O resultado é um indivíduo contendo um percurso aleatório de cidades.



# Criando a população

```
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
```

Isso define uma **população**, que é uma lista de indivíduos.

- ◆ **tools.initRepeat** cria vários indivíduos e os armazena em uma **lista**.
- ◆ Cada indivíduo é gerado com base na função `toolbox.individual`.

# Exemplo de Funcionamento

Se numCities = 5, a população inicial pode conter indivíduos assim:

Indivíduo 1: [3, 1, 0, 4, 2]

Indivíduo 2: [4, 2, 3, 1, 0]

Indivíduo 3: [1, 0, 4, 2, 3]

- Cada indivíduo representa **um caminho diferente** pelo conjunto de cidades.

O código usa **Algoritmos Genéticos** para resolver o **problema do Caixeiro Viajante**.

- ✓ Define a **função de aptidão (fitness)** para **minimizar a distância** percorrida.
- ✓ Cria **indivíduos** (rotas aleatórias).
- ✓ Define uma **população** de possíveis soluções.

- **Explicação da Avaliação da Aptidão no Problema do Caixeiro Viajante**
- A função de aptidão (fitness function) mede a qualidade de cada indivíduo (rota) na população, ou seja, quanto menor a distância total percorrida, melhor a solução.

```
def evalRoute(individual):
    cost = 0
    for i in range(1, len(individual)):
        if (individual.count(individual[i]) > 1):
            cost = cost + 1000000 # penalidade por repetir cidade
        cost = cost + cities[individual[i-1], individual[i]]
    cost = cost + cities[individual[i], individual[0]]
    return (cost,)
```

## Entendendo a Função evalRoute()

A função **calcula o custo total de uma rota (indivíduo)** e retorna esse valor em forma de **tupla** ((cost,)), conforme exigido pelo **DEAP**.

- ◆ **Parâmetro:**

- individual: uma lista representando a sequência de cidades visitadas (por exemplo, [0, 2, 1, 4, 3]).

- ◆ **Saída:**

- Retorna uma **tupla** contendo o custo total da rota (cost,).

# 1 Inicializa o Custo da Rota

**cost = 0**

- 📍 A variável cost armazenará a **distância total da rota**.

## 2 Percorre a Rota

```
for i in range(1, len(individual)):
```

Percorre cada cidade do indivíduo (rota).

- 💡 O for começa de 1 porque o custo é calculado entre **cidades consecutivas**.

### 3

## Penalidade para Rotas Inválidas

```
if (individual.count(individual[i]) > 1):  
    cost = cost + 1000000 # Penalidade por repetir cidade
```

- 📍 Se uma cidade for visitada mais de uma vez, uma **penalidade de 1.000.000** é adicionada ao custo.
- 📍 Isso força o algoritmo a evitar soluções inválidas (o caixeiro só pode visitar cada cidade uma vez).

4

## Soma as Distâncias Entre Cidades

```
cost = cost + cities[individual[i-1], individual[i]]
```

- 📍 Soma a distância entre cidades consecutivas na rota.
- 📍 `cities[individual[i-1], individual[i]]` obtém a distância da cidade anterior para a cidade atual a partir da matriz de distâncias.

## 5 Fecha o Ciclo (Volta à Cidade Inicial)

```
cost = cost + cities[individual[i], individual[0]]
```

- 💡 O problema do Caixeiro Viajante exige que o viajante retorne à cidade de origem.
- 💡 Soma a distância entre a última cidade visitada e a primeira cidade da rota.

# Estrutura do Algoritmo Genético

## 1 Inicialização

- Gera uma **população inicial** aleatória.

## 2 Avaliação de Aptidão (Fitness)

- Cada indivíduo da população é **avaliado** com base em uma função de aptidão.
  - No problema do **Caixeiro Viajante**, a aptidão está ligada à menor **distância percorrida**.

## 3 Seleção

- Os melhores indivíduos são escolhidos para **reprodução**.

- Técnicas comuns:

- **Roleta**: Probabilidade proporcional à aptidão.
- **Torneio**: Seleciona um subconjunto aleatório e escolhe o melhor.

#### 4 Cruzamento (Crossover)

Combina genes de **dois pais** para gerar novos indivíduos.

Exemplo: **Crossover de um ponto**

- Exemplo:

- Pai 1: [0, 1, 2, | 3, 4]

- Pai 2: [4, 3, 2, | 1, 0]

- Filhos: [0, 1, 2, | 1, 0] e [4, 3, 2, | 3, 4]

#### 5 Mutação

Altera **aleatoriamente** um ou mais genes para **manter a diversidade genética**.

Exemplo:

- Antes: [0, 1, 2, 3, 4]

- Depois da mutação: [0, 1, 4, 3, 2]

#### 6 Nova Geração

Substitui a população antiga pelos novos indivíduos gerados.

#### 7 Repetição até Critério de Parada

O processo se repete por várias gerações até atingir:

- Um número máximo de iterações.

- Uma solução ótima.

- Um critério de convergência.

# Etapas do Código

## 1 Definição dos Operadores Genéticos

- O toolbox é um conjunto de funções personalizadas para o algoritmo.
- Ele registra os principais operadores:

- **Avaliação de Aptidão:** toolbox.register("evaluate", evalRoute)
  - Usa a função evalRoute para calcular o custo da rota.
- **Seleção:** tools.selTournament, tourysize=3
  - Método **torneio**: Escolhe três indivíduos aleatoriamente e seleciona o melhor.
- **Cruzamento:** tools.cxOnePoint
  - Método **de um ponto**: Divide os cromossomos dos pais em um ponto e troca partes.
- **Mutação:** tools.mutUniformInt, low=0, up=numCities-1, indpb=0.05
  - **Mutação uniforme**: Altera um gene com probabilidade de 5%.

## 2 Configuração da Execução

• No main(), são definidos os **parâmetros do AG**:

- NGEN = 100 → Número de gerações.
- MU = 50 → Tamanho da população inicial.
- LAMBDA = 100 → Número de indivíduos gerados por geração.
- CXPB = 0.7 → Probabilidade de cruzamento.
- MUTPB = 0.3 → Probabilidade de mutação.

• A população inicial é criada:

```
pop = toolbox.population(n=MU)
```

- Um **\*Hall of Fame (hof)** é criado para armazenar os melhores indivíduos.
- Estatísticas da evolução são registradas com a função stats.

### 3 Execução do Algoritmo

💡 O algoritmo evolui a população usando a função eaMuPlusLambda, que implementa um AG elitista:

```
algorithms.eaMuPlusLambda(pop, toolbox, MU, LAMBDA, CXPB, MUTPB, NGEN, stats, halloffame=hof)
```

- Mantém **MU** indivíduos da geração anterior.
- Gera **LAMBDA** novos indivíduos por cruzamento e mutação.

- **MU** = 50 → Tamanho da população inicial.
- **LAMBDA** = 100 → Número de indivíduos gerados por geração.



## 4 Exibição da Melhor Solução

💡 Após a execução, imprime a melhor rota encontrada e seu custo:

```
print('\nRota:', hof[0], '\nCusto:', evalRoute(hof[0])[0])
```

- `hof[0]`: Melhor indivíduo (rota ótima).
- `evalRoute(hof[0])[0]`: Custo da rota (distância total).

---

**Cria uma população inicial aleatória.**

---

**Avalia os indivíduos** (menor distância).

---

**Seleciona os melhores** (torneio).

---

**Aplica cruzamento** (troca partes das rotas).

---

**Aplica mutação** (modifica cidades aleatoriamente).

---

**Repete o processo** até atingir 100 gerações.

---

**Retorna a melhor solução encontrada.**

# Estrutura do Algoritmo Genético

O código segue os passos básicos de um AG:

## 1.Criação da População Inicial

- Gera um conjunto inicial de soluções aleatórias.

## 2.Avaliação de Aptidão (Fitness Evaluation)

- Mede a qualidade de cada indivíduo baseado na função evalRoute.

## 3.Seleção (Selection)

- Usa **seleção por torneio** (selTournament) para escolher indivíduos melhores.

## 4.Cruzamento (Crossover/Recombination)

- Aplica **crossover de um ponto** (cxOnePoint) para gerar novos indivíduos.

## 5.Mutação (Mutation)

- Usa **mutação uniforme** (mutUniformInt) para introduzir variação.

## 6.Evolução por Várias Gerações

- Repete o processo por **100 gerações** (NGEN = 100), tentando melhorar a solução.

# Detalhamento do Código

## 1 Definição dos Operadores Genéticos

O toolbox da DEAP registra as funções usadas no algoritmo:

```
toolbox.register("evaluate", evalRoute) # Avaliação da rota  
toolbox.register("select", tools.selTournament, tournsize=3) # Seleção por torneio  
toolbox.register("mate", tools.cxOnePoint) # Crossover de um ponto  
toolbox.register("mutate", tools.mutUniformInt, low=0, up=numCities-1, indpb=0.05) # Mutação uniforme
```

- **Seleção por torneio:** Escolhe 3 indivíduos aleatoriamente e seleciona o melhor.
- **Crossover de um ponto:** Divide os cromossomos dos pais em um ponto e troca as partes.
- **Mutação uniforme:** Modifica um gene aleatoriamente com 5% de chance.

## 2 Configuração do Algoritmo

```
NGEN = 100      # Número de gerações  
MU = 50        # Tamanho da população inicial  
LAMBDA = 100    # Número de filhos gerados por geração  
CXPB = 0.7      # Probabilidade de cruzamento  
MUTPB = 0.3     # Probabilidade de mutação
```

- População inicial de 50 indivíduos.
- Cada geração gera 100 novos indivíduos.
- Probabilidade de cruzamento de 70% e de mutação de 30%.

### 3 Criando a População Inicial

```
pop = toolbox.population(n=MU)
hof = tools.ParetoFront() # Melhor indivíduo será armazenado aqui
```

O Hall of Fame (**hof**) guarda os melhores indivíduos.

## 4 Estatísticas do Algoritmo

```
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("avg", numpy.mean, axis=0) # Média
stats.register("std", numpy.std, axis=0) # Desvio padrão
stats.register("min", numpy.min, axis=0) # Melhor indivíduo (menor custo)
stats.register("max", numpy.max, axis=0) # Pior indivíduo (maior custo)
```

Coleta estatísticas sobre os indivíduos ao longo das gerações.

## 5 Execução do Algoritmo

```
algorithms.eaMuPlusLambda(pop, toolbox, MU, LAMBDA, CXPB, MUTPB, NGEN, stats, halloffame=hof)
```

- **EAMu + Lambda** → Mantém **MU** melhores indivíduos da geração anterior.
- Gera **LAMBDA novos indivíduos** a partir deles usando cruzamento e mutação.

## 6

# Exibição da Melhor Solução

```
print('\nRota:', hof[0], '\nCusto:', evalRoute(hof[0])[0])
```

- **Mostra a melhor rota encontrada e seu custo total.**

- Algoritmo Genético otimiza a rota.
- Usa **seleção por torneio, crossover de um ponto e mutação uniforme**.
- Evolui por **100 gerações** até encontrar uma solução de menor custo.

Rota: [0, 2, 4, 3, 9, 5, 1, 8, 6, 7] Custo: 471

Parece que o algoritmo genético convergiu para um valor mínimo de 471 nas últimas gerações. Alguns pontos de análise:

- 1. Convergência precoce:** A população chegou a um valor fixo (596) por muitas gerações antes de começar a diminuir. Isso pode indicar falta de diversidade genética ou um problema na estratégia de mutação/seleção.
- 2. Estagnação e reinicialização:** Em certas gerações, a média ficou constante por muito tempo e depois sofreu variações (exemplo: após a geração 65, houve uma nova queda). Isso pode ser um efeito da introdução de mutações mais fortes ou mudanças na estratégia do algoritmo.
- 3. Valor final:** O menor valor encontrado foi 471, e ele se manteve por algumas gerações, indicando que pode ser o ótimo global ou pelo menos um ótimo local difícil de superar.

- <https://colab.research.google.com/drive/1Oj0jrYf4-LHdxPN8OYD2uUFzHfoL7rLI?usp=sharing>

