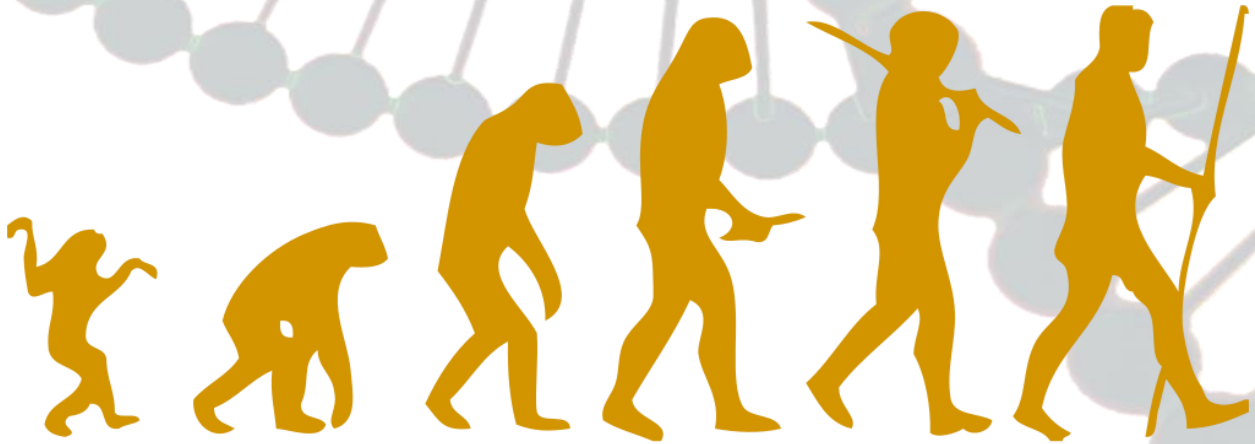
AI VIETNAM
All-in-One Course

GENETIC ALGORITHM



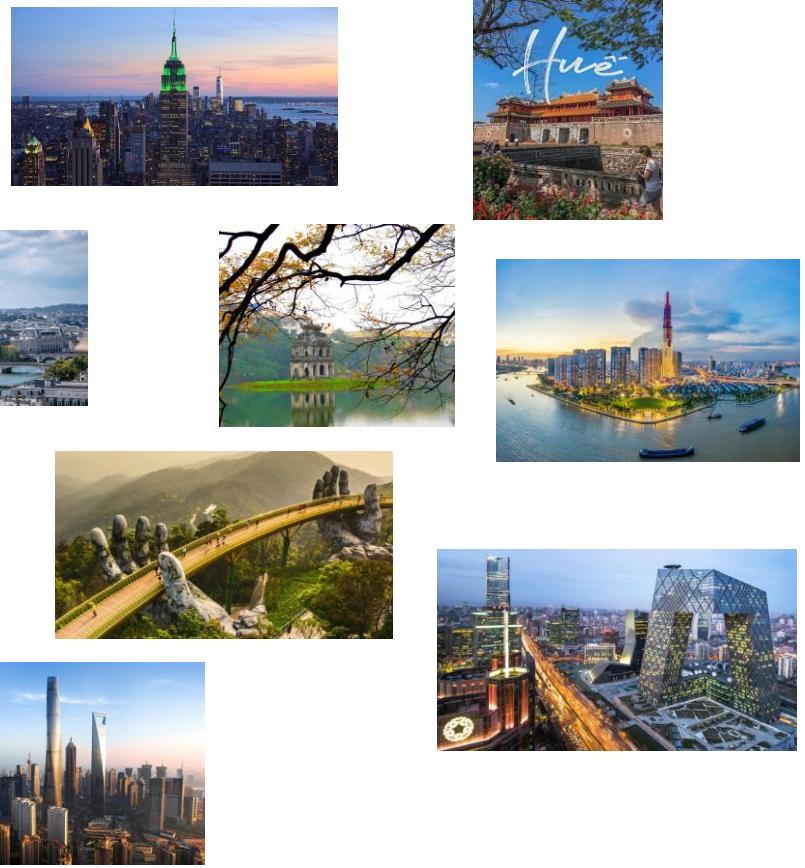
AIO2023

Outline

- Introduction Genetic Algorithm
- Population
- Evaluation
- Selection
- Crossover
- Mutation

Introduction Genetic Algorithm

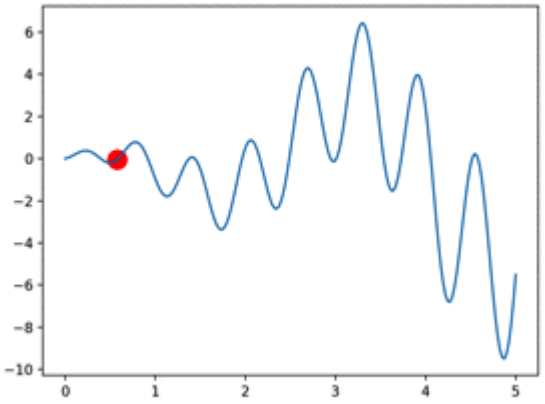
❑ Nguồn gốc thuật toán GA?



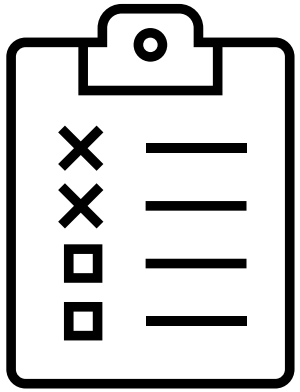
Một du khách muốn thăm những thành phố anh quan tâm; mỗi thành phố thăm qua đúng một lần; rồi trở về điểm khởi hành. Biết trước chi phí di chuyển giữa hai thành phố bất kỳ.

Introduction Genetic Algorithm

- **Nguồn gốc thuật toán GA?**



Optimization(*)



Lập kế hoạch

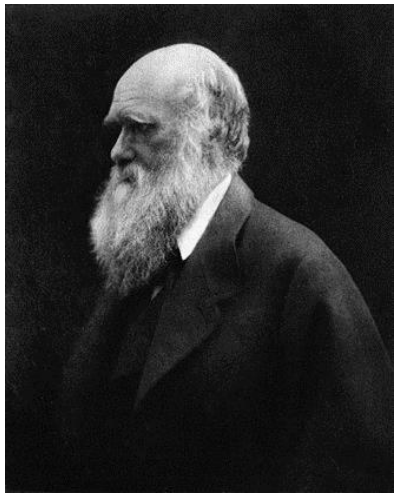
Flappy bird(**)

(*) https://github.com/MorvanZhou/Evolutionary-Algorithm

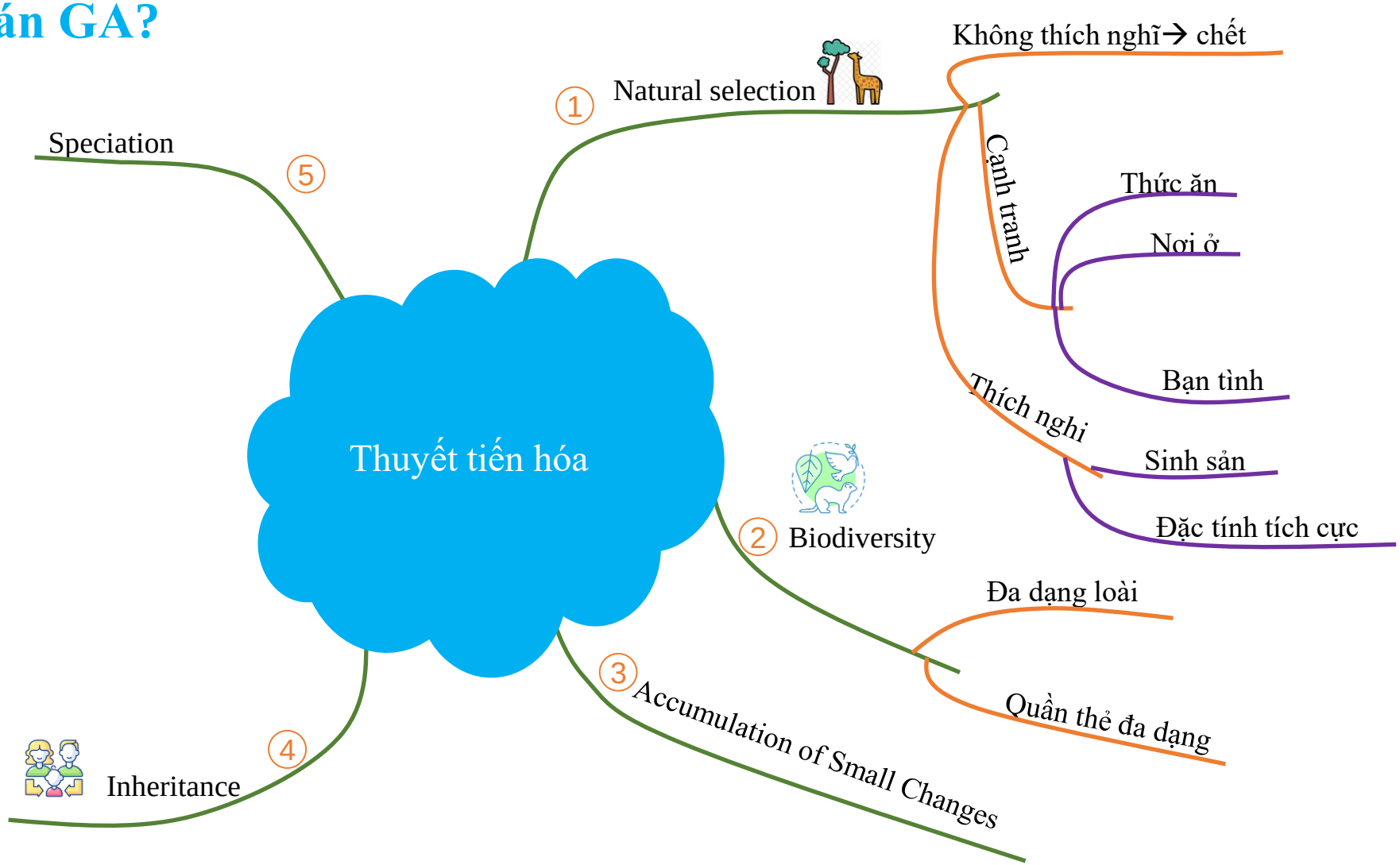
(**) https://threadsiiithyderabad.quora.com/Neuro-Evolution-with-Flappy-Bird-Genetic-Evolution-on-Neural-Networks

Introduction Genetic Algorithm

- **Nguồn gốc thuật toán GA?**



Charles Darwin(1809–1882)

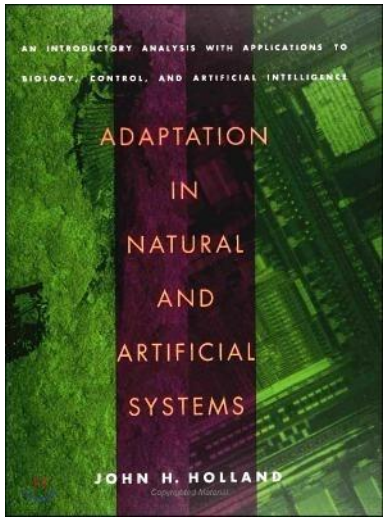


Introduction Genetic Algorithm

❑ Nguồn gốc thuật toán GA?

The *genetic algorithm* (GA), developed by John Holland and his collaborators in the 1960s and 1970s

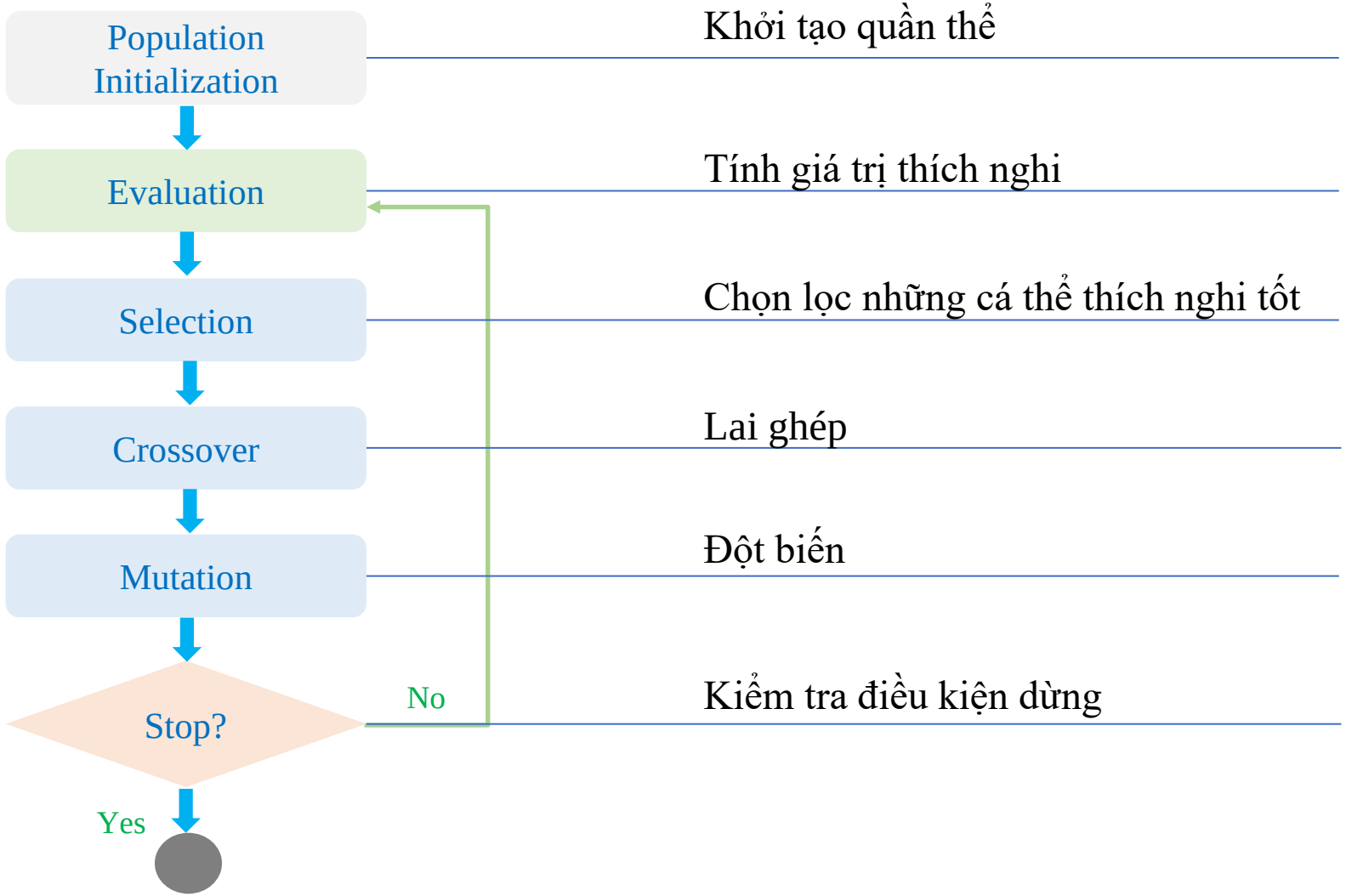
John Henry Holland
(1929-2015)



Adaptation in Natural and Artificial Systems(1975)

Introduction Genetic Algorithm

❑ Thuật toán GA

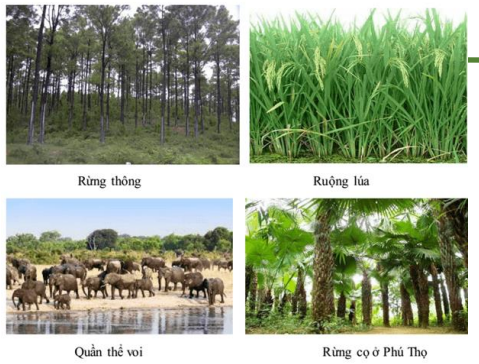


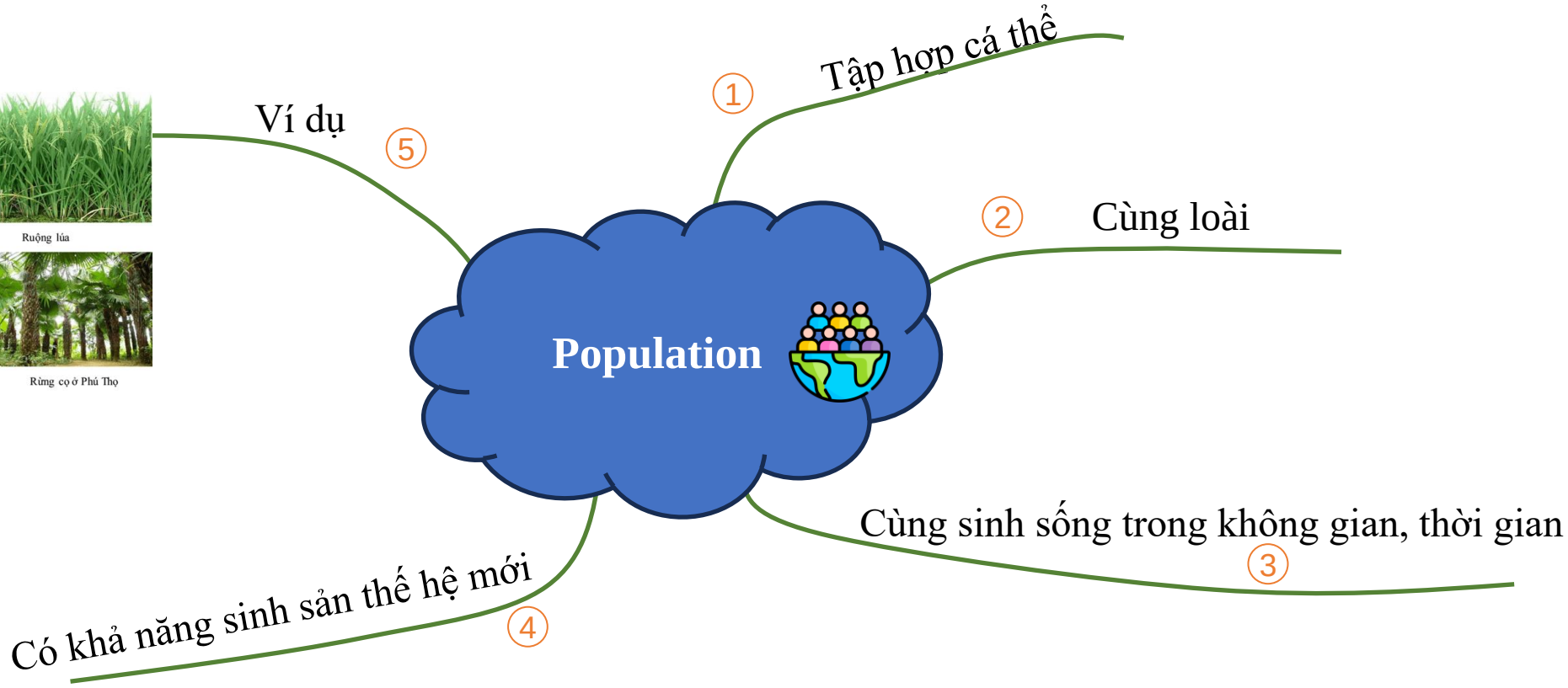
Outline

- Introduction Genetic Algorithm
- Population
- Evaluation
- Selection
- Crossover
- Mutation

Population

❑ Population là gì?

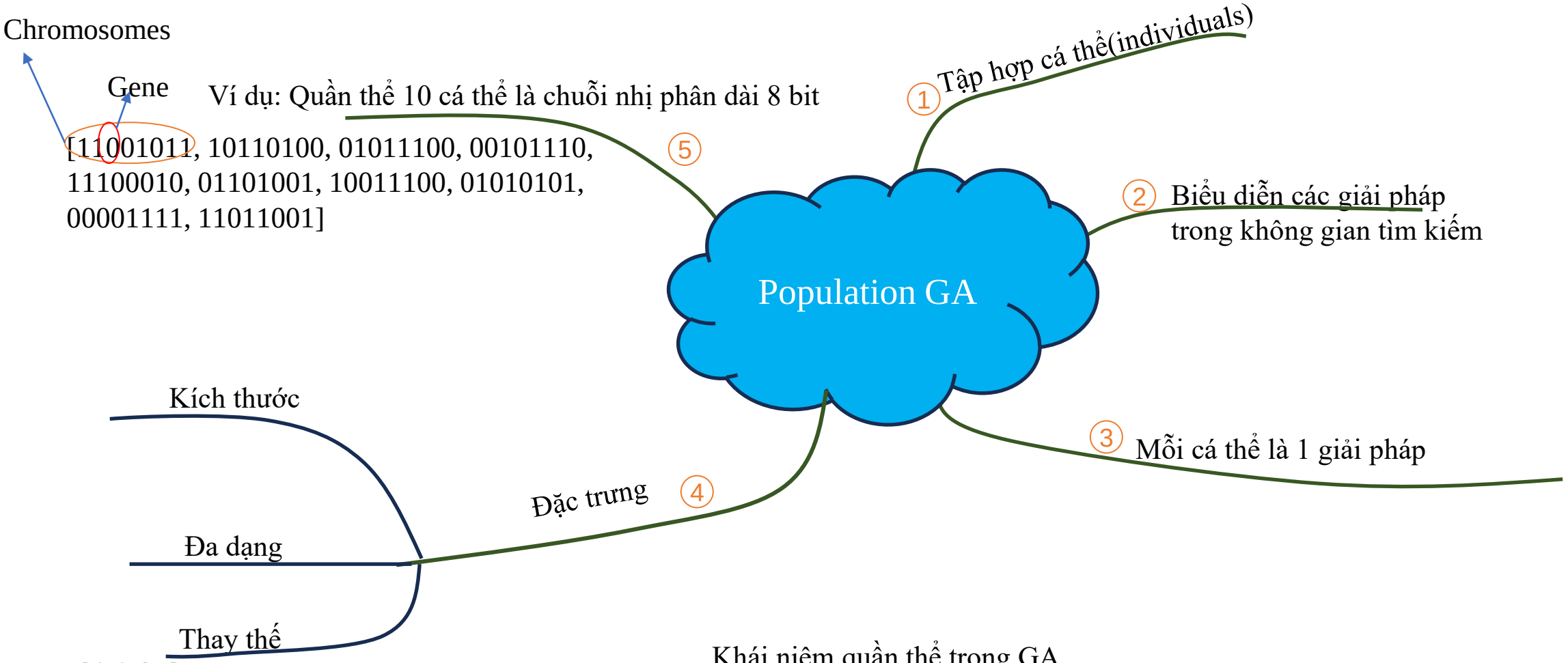




Khái niệm quần thể sinh học

Population

❑ Population là gì?



Khái niệm quần thể trong GA

Population

Bài tập 1: Xác định population cho bài toán tối ưu hóa chuỗi nhị phân

Giả sử bạn muốn giải quyết bài toán tối ưu hóa chuỗi nhị phân dài 10 bit, trong đó ta cần tìm chuỗi nhị phân có tổng giá trị các bit là lớn nhất.

Yêu cầu: Hãy viết một hàm có tên create_binary_population(pop_size, bit_length) nhận vào hai tham số: pop_size (kích thước quần thể) và bit_length (độ dài của mỗi chuỗi nhị phân cá thể). Hàm này sẽ trả về một mảng NumPy biểu diễn quần thể gồm pop_size cá thể, mỗi cá thể là một chuỗi nhị phân có bit_length bit.

Population

❑ Solution

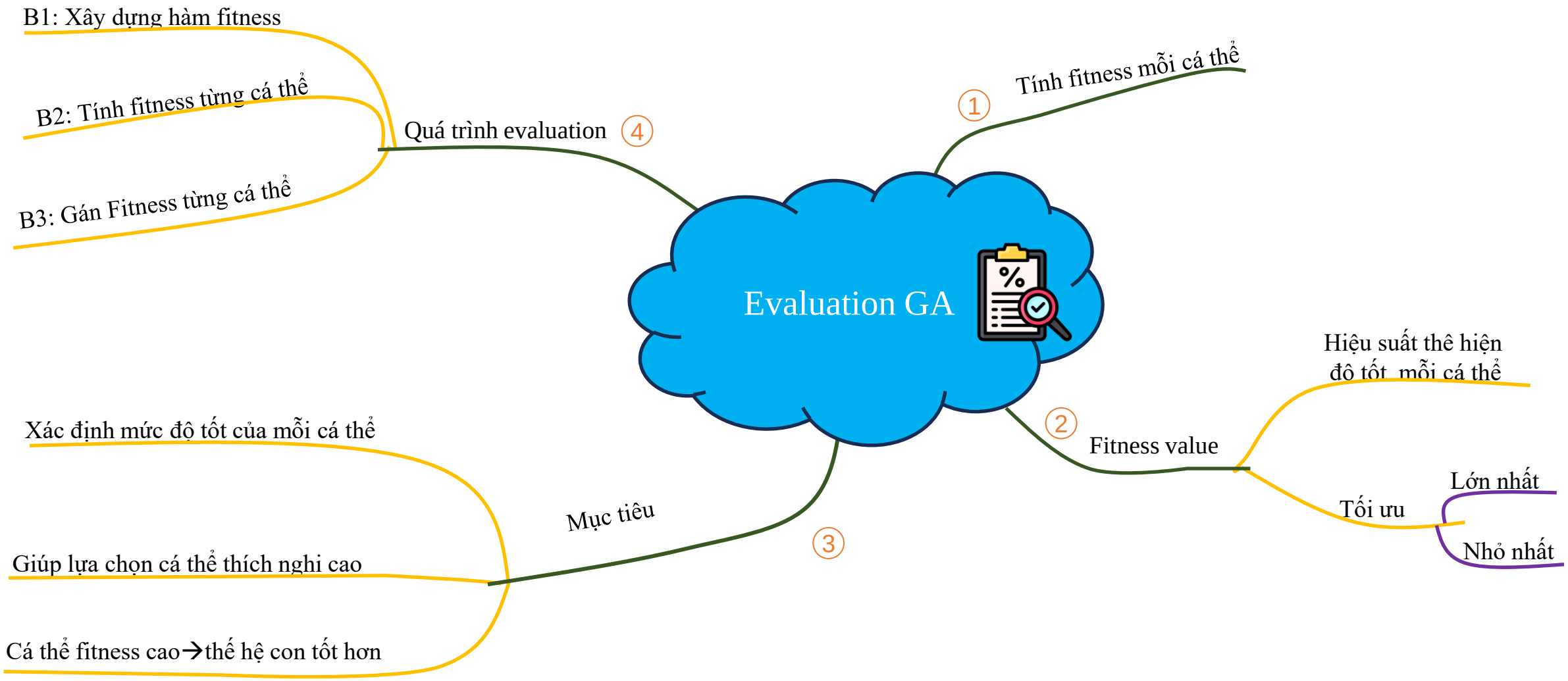
```
import numpy as np

def create_binary_population(pop_size, bit_length):
    population = np.random.randint(2, size=(pop_size, bit_length))
    return population



population_size = 20
bit_length = 10

population = create_binary_population(population_size, bit_length)
print("Population:")
print(population)
```

```
Population:
[[0 1 0 1 0 1 0 1 0 1]
 [0 0 0 0 1 1 0 0 1 0]
 [1 0 1 1 0 0 1 1 0 1]
 [0 0 0 1 0 1 1 0 1 0]
 [0 0 1 1 0 0 1 1 1 0]
 [1 0 0 1 0 1 0 1 1 0]
 [0 1 0 1 1 1 1 0 1 0]
 [1 0 0 1 0 1 1 1 1 1]
 [0 1 0 0 0 0 0 1 0 1]
 [0 0 1 1 0 1 1 0 0 0]
 [0 0 1 0 1 0 1 0 1 1]
 [1 1 0 0 1 1 0 1 0 1]
 [0 1 0 1 0 0 1 0 0 0]
 [1 0 0 1 0 0 0 0 0 1]
 [1 1 0 0 0 1 0 1 0 0]
 [1 1 0 0 1 1 1 1 1 1]
 [1 1 0 1 1 1 1 0 1 1]
 [1 0 1 0 1 0 0 0 1 0]
 [0 0 0 0 0 0 1 1 1 1]
 [1 1 1 0 1 1 0 0 0 1]]
```

Outline

- Introduction Genetic Algorithm
- Population
- Evaluation
- Selection
- Crossover
- Mutation

Evaluation



Evaluation trong GA

Evaluation

Bài tập 2: Tính fitness cho từng cá thể chuỗi nhị phân

Giả sử bạn muốn giải quyết bài toán tối ưu hóa chuỗi nhị phân dài 10 bit, trong đó ta cần tìm chuỗi nhị phân có tổng giá trị các bit là lớn nhất.

Yêu cầu: Hãy viết một hàm có tên fitness_binary_individual(individual) nhận vào một tham số individual, đại diện cho một cá thể chuỗi nhị phân. Hàm này sẽ tính và trả về giá trị fitness của cá thể individual, dựa vào tổng số bit có giá trị 1 trong chuỗi nhị phân.

Evaluation

❑ Solution

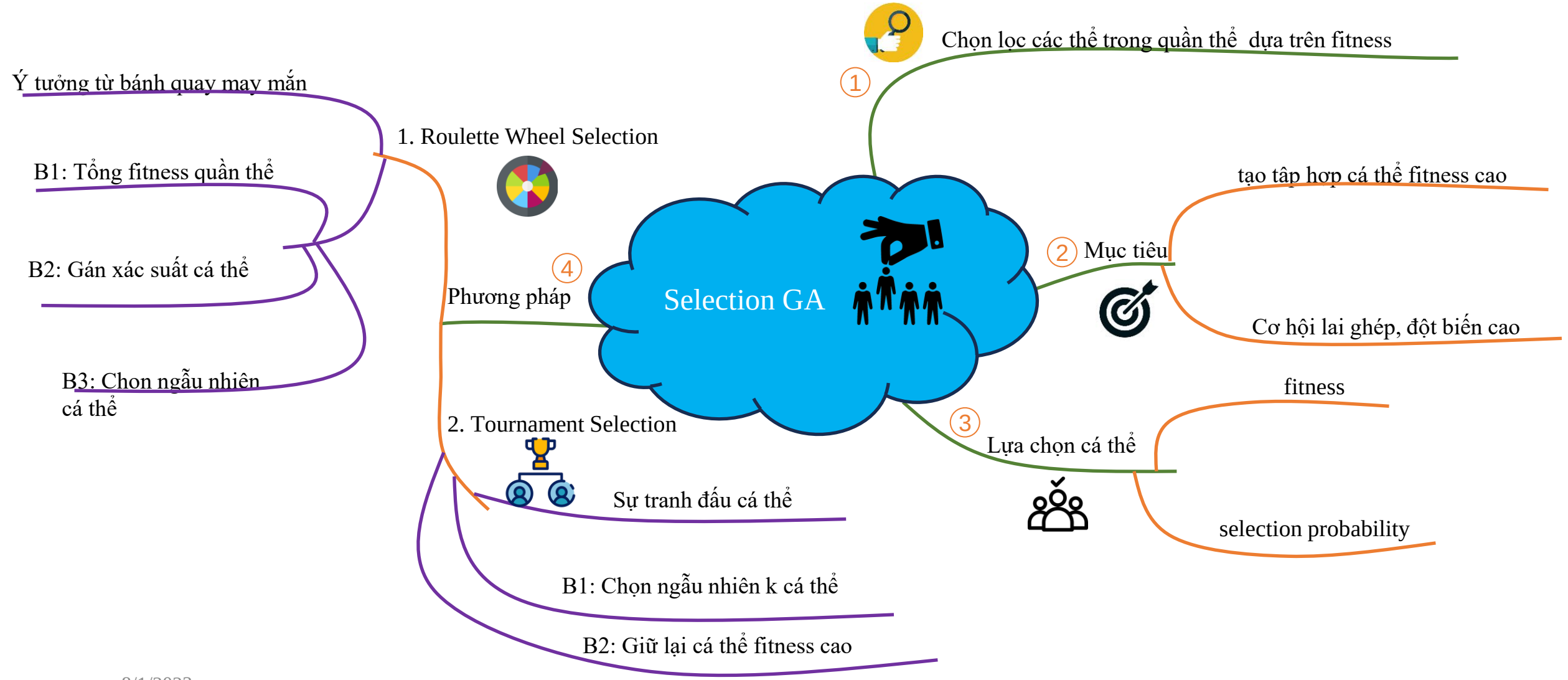
```
def fitness_binary_individual(individual):
    return np.sum(individual)


fitness_values = np.apply_along_axis(fitness_binary_individual, 1, population)

print("Population:")
print(population)
print("Fitness values:")
print(fitness_values)
```

```
Population:
[[0 1 0 1 0 1 0 1 0 1]
 [0 0 0 0 1 1 0 0 1 0]
 [1 0 1 1 0 0 1 1 0 1]
 [0 0 0 1 0 1 1 0 1 0]
 [0 0 1 1 0 0 1 1 1 0]
 [1 0 0 1 0 1 0 1 1 0]
 [0 1 0 1 1 1 1 0 1 0]
 [1 0 0 1 0 1 1 1 1 1]
 [0 1 0 0 0 0 0 1 0 1]
 [0 0 1 1 0 1 1 0 0 0]
 [0 0 1 0 1 0 1 0 1 1]
 [1 1 0 0 1 1 0 1 0 1]
 [0 1 0 1 0 0 1 0 0 0]
 [1 0 0 1 0 0 0 0 0 1]
 [1 1 0 0 0 1 0 1 0 0]
 [1 1 0 0 1 1 1 1 1 1]
 [1 1 0 1 1 1 1 0 1 1]
 [1 0 1 0 1 0 0 0 1 0]
 [0 0 0 0 0 0 1 1 1 1]
 [1 1 1 0 1 1 0 0 0 1]]
Fitness values:
[5 3 6 4 5 5 6 7 3 4 5 6 3 3 4 8 8 4 4 6]
```

Outline

- Introduction Genetic Algorithm
- Population
- Evaluation
- Selection
- Crossover
- Mutation

Selection


Selection GA
1
Chọn lọc các thể trong quần thể dựa trên fitness
2
Mục tiêu
tạo tập hợp cá thể fitness cao
Cơ hội lai ghép, đột biến cao
3
Lựa chọn cá thể
fitness
selection probability
4
Phương pháp
1. Roulette Wheel Selection
Ý tưởng từ bánh quay may mắn
B1: Tổng fitness quần thể
B2: Gán xác suất cá thể
B3: Chọn ngẫu nhiên cá thể
2. Tournament Selection
Sự tranh đấu cá thể
B1: Chọn ngẫu nhiên k cá thể
B2: Giữ lại cá thể fitness cao

Selection

Bài tập 3: Lựa chọn cá thể trong quần thể bằng phương pháp Roulette Wheel Selection

Giả sử bạn muốn giải quyết bài toán tối ưu hóa chuỗi nhị phân dài 10 bit, trong đó ta cần tìm chuỗi nhị phân có tổng giá trị các bit là lớn nhất.

Yêu cầu: Hãy viết một hàm có tên roulette_wheel_selection(population, fitness_values) nhận vào hai tham số: population (quần thể) và fitness_values (giá trị fitness tương ứng của từng cá thể trong quần thể). Hàm này sẽ thực hiện phương pháp lựa chọn cá thể trong quần thể dựa trên phương pháp Roulette Wheel Selection và trả về một mảng NumPy biểu diễn quần thể được lựa chọn.

Selection

❑ Solution

```
def roulette_wheel_selection(population, fitness_values):
    total_fitness = np.sum(fitness_values)
    selection_probs = fitness_values / total_fitness
    selected_indices = np.random.choice(len(population), size=len(population), p=selection_probs)
    selected_population = population[selected_indices]
    return selected_population


selected_population = roulette_wheel_selection(population, fitness_values)

print("Population:")
print(population)
print("Fitness values:")
print(fitness_values)
print("Selected population:")
print(selected_population)
```

```
Selected population:
[[0 0 1 0 1 0 1 0 1 1]
...
 [0 1 0 1 0 1 0 1 0 1]
 [0 0 1 1 0 0 1 1 1 0]
 [1 1 0 1 1 1 1 0 1 1]
 [1 1 0 0 1 1 1 1 1 1]]
```

Outline

- Introduction Genetic Algorithm
- Population
- Evaluation
- Selection
- Crossover
- Mutation

Crossover

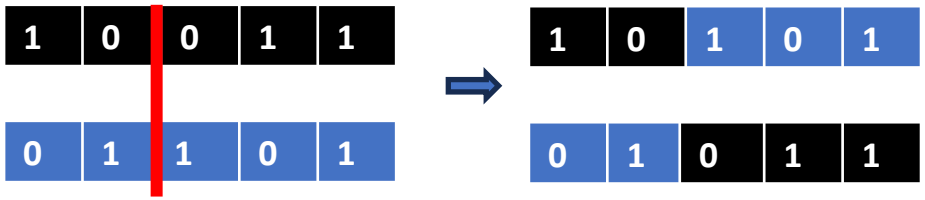
Chọn 1 điểm chia từ 2 cha mẹ

Phân trước điểm chia → con 1

Phân sau điểm chia → con 2

One-Point Crossover

4

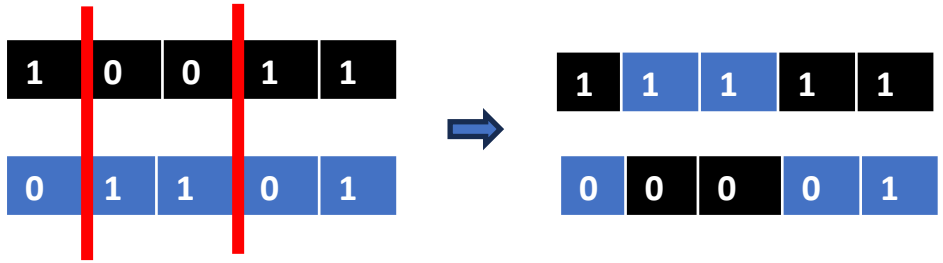


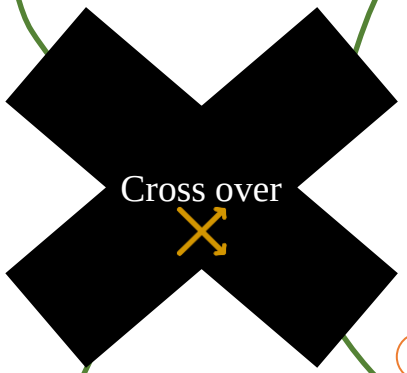
Chọn 2 điểm chia từ cha mẹ

Trao đổi phần giữa 2 điểm chia
→ 2 cá thể con

3

Two-Point Crossover





1

Lai ghép cha mẹ

Mô phỏng di truyền tự nhiên

Cá thể con kế thừa đặc tính

2

Ý nghĩa

Đa dạng trong quần thể

Giải pháp tiềm năng→cá thể fitness cao

Crossover

Bài tập 4: Thực hiện phép lai ghép One-Point Crossover giữa hai cá thể

Giả sử bạn muốn giải quyết bài toán tối ưu hóa chuỗi nhị phân dài 10 bit, trong đó ta cần tìm chuỗi nhị phân có tổng giá trị các bit là lớn nhất.

Yêu cầu: Hãy viết một hàm có tên one_point_crossover(parent1, parent2) nhận vào hai tham số parent1 và parent2, đại diện cho hai cá thể cha mẹ. Hàm này sẽ thực hiện phép lai ghép One-Point Crossover giữa hai cá thể cha mẹ và trả về hai cá thể con mới tạo ra từ phép lai ghép này.

Crossover

❑ Solution

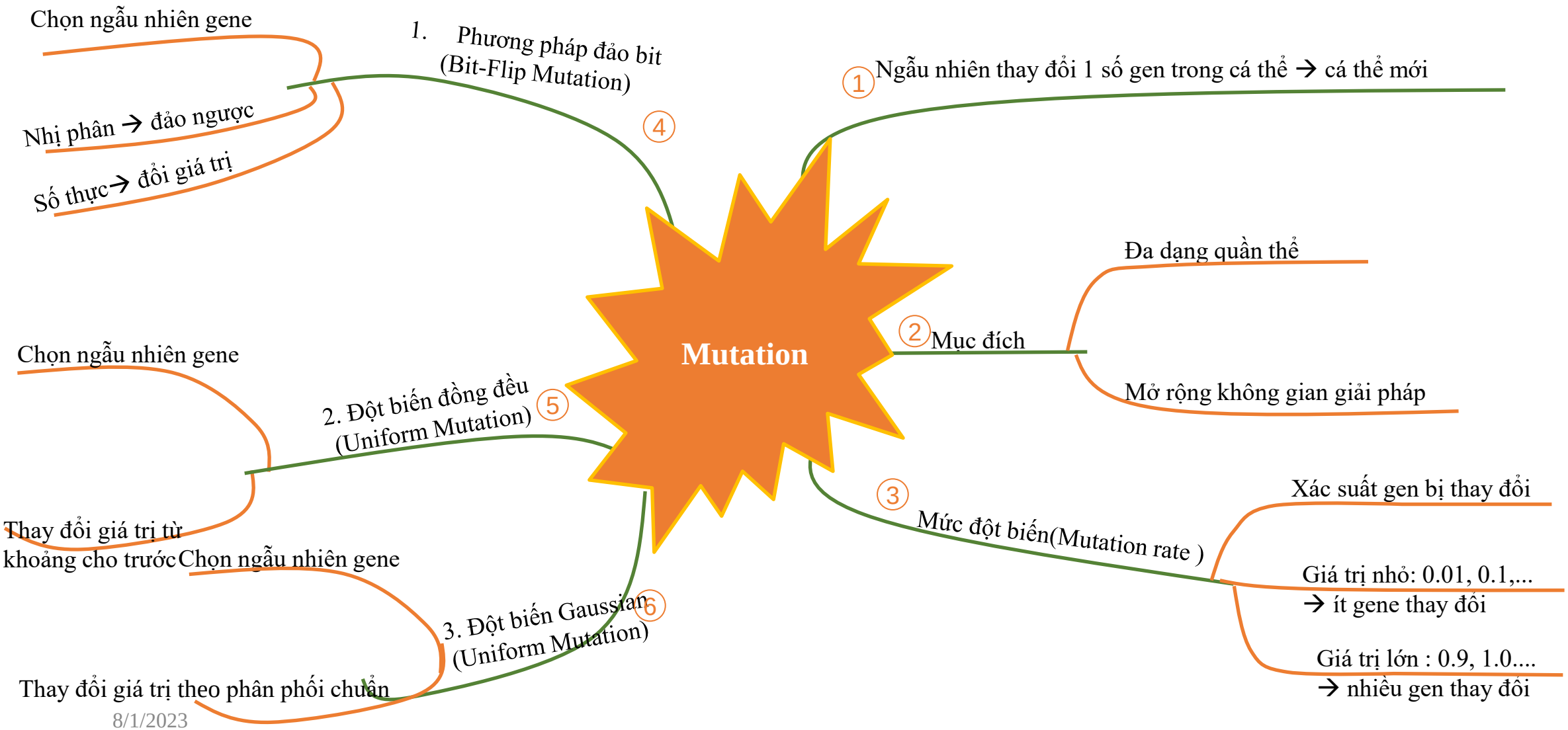
```
def one_point_crossover(parent1, parent2):
    crossover_point = np.random.randint(1, len(parent1))  # Chọn một điểm chia ngẫu nhiên
    child1 = np.concatenate((parent1[:crossover_point], parent2[crossover_point:]))
    child2 = np.concatenate((parent2[:crossover_point], parent1[crossover_point:]))
    return child1, child2

# Chọn ngẫu nhiên hai cá thể cha mẹ từ quần thể
parent1_idx, parent2_idx = np.random.choice(population_size, size=2, replace=False)
parent1 = selected_population[parent1_idx]
parent2 = selected_population[parent2_idx]

child1, child2 = one_point_crossover(parent1, parent2)

print("Parent 1:")
print(parent1)
print("Parent 2:")
print(parent2)
print("Child 1:")
print(child1)
print("Child 2:")
print(child2)
```

```
Parent 1:
[1 0 1 1 0 0 0 1 0 1]
Parent 2:
[1 0 1 0 1 1 1 1 1 0]
Child 1:
[1 0 1 1 1 1 1 1 1 0]
Child 2:
[1 0 1 0 0 0 0 1 0 1]
```

Outline

- Introduction Genetic Algorithm
- Population
- Evaluation
- Selection
- Crossover
- Mutation

Mutation


Mutation
1. Phương pháp đảo bit (Bit-Flip Mutation)
Chọn ngẫu nhiên gene
Nhị phân → đảo ngược
Số thực→ đổi giá trị
2. Đột biến đồng đều (Uniform Mutation)
Chọn ngẫu nhiên gene
Thay đổi giá trị từ khoảng cho trước
3. Đột biến Gaussian (Uniform Mutation)
Chọn ngẫu nhiên gene
Thay đổi giá trị theo phân phối chuẩn
1 Ngẫu nhiên thay đổi 1 số gen trong cá thể → cá thể mới
2 Mục đích
Đa dạng quần thể
Mở rộng không gian giải pháp
3 Mức đột biến(Mutation rate)
Xác suất gen bị thay đổi
Giá trị nhỏ: 0.01, 0.1,... → ít gene thay đổi
Giá trị lớn : 0.9, 1.0.... → nhiều gen thay đổi
4
5
6

Mutation

Bài tập 5: Thực hiện phép đột biến Bit-Flip Mutation cho một cá thể chuỗi nhị phân

Giả sử bạn muốn giải quyết bài toán tối ưu hóa chuỗi nhị phân dài 10 bit, trong đó ta cần tìm chuỗi nhị phân có tổng giá trị các bit là lớn nhất.

Yêu cầu: Hãy viết một hàm có tên bit_flip_mutation(individual, mutation_rate) nhận vào hai tham số individual (cá thể chuỗi nhị phân) và mutation_rate (tỷ lệ đột biến). Hàm này sẽ thực hiện phép đột biến Bit-Flip Mutation cho cá thể individual, với xác suất đột biến là mutation_rate, và trả về cá thể mới sau khi thực hiện đột biến.

Selection

❑ Solution

```
def bit_flip_mutation(individual, mutation_rate):
    mutated_individual = individual.copy()
    for i in range(len(mutated_individual)):
        if np.random.rand() < mutation_rate:
            mutated_individual[i] = 1 - mutated_individual[i]  # Đảo bit 0 thành 1 và 1 thành 0
    return mutated_individual
```

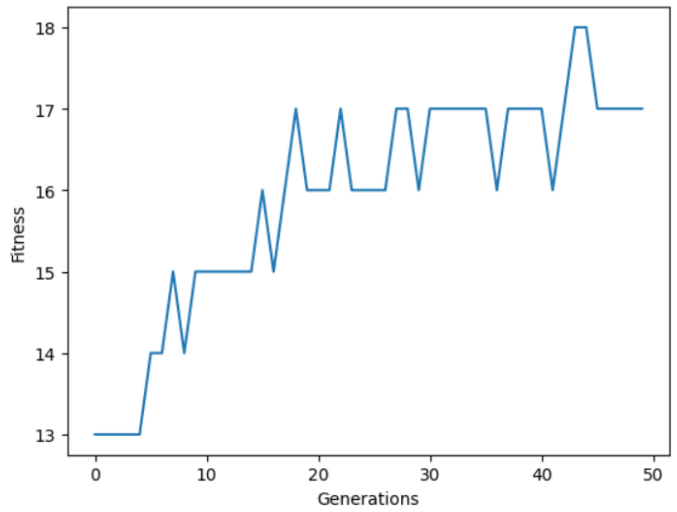
```
mutation_rate = 0.9

# child 1 mutation
child1_mutation = bit_flip_mutation(child1, mutation_rate)
print('child 1', child1)
print('child 1 mutation', child1_mutation)
```

```
child 1 [1 0 1 1 1 1 1 1 1 0]
child 1 mutation [0 1 0 0 0 0 0 0 0 1]
```

Replacement and Termination

Solution

```
population_size = 20
bit_length = 20
n_generations = 50
mutation_rate = 0.01

fitnesses = []
```




```
population = create_binary_population(pop_size=population_size, bit_length=bit_length)

for i in range(n_generations):
    fitness_values = np.apply_along_axis(fitness_binary_individual, 1, population)
    fitnesses.append(np.max(fitness_values))
    print("Best:",np.max(fitness_values))

    selected_population = roulette_wheel_selection(population, fitness_values)
    new_population = []

    while(len(new_population) < population_size):
        # Selection
        # Chọn ngẫu nhiên hai cá thể cha mẹ từ quần thể
        parent1_idx, parent2_idx = np.random.choice(population_size, size=2, replace=False)
        parent1 = selected_population[parent1_idx]
        parent2 = selected_population[parent2_idx]

        # Crossover
        child1, child2 = one_point_crossover(parent1, parent2)

        # Mutation
        child1_mutation = bit_flip_mutation(child1, mutation_rate)
        child2_mutation = bit_flip_mutation(child2, mutation_rate)

        # Add children to new population
        new_population.append(child1_mutation)
        new_population.append(child2_mutation)
    if n_generations % 2 == 0:
        population = np.array(new_population)
    else:
        population = np.array(new_population[:-1])
```