TahaAbbasAli_P20-0119_6A_AILab_12

May 3, 2023

[2]:
```
# 2D array
s = (4,4)
arr2d = np.ones(s, dtype=int)
arr2d
```

[2]:
```
array([[1, 1, 1, 1],
       [1, 1, 1, 1],
       [1, 1, 1, 1],
       [1, 1, 1, 1]])
```

[9]:
```
# Flatten
arr2d = arr2d.flatten()
print(arr2d)
```

```
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
```

[3]:
```
arr2d.shape
```

[3]:
```
(4, 4)
```

[4]:
```
s = (4,4)
arr1d = np.zeros(s, dtype=int)
arr1d
```

[4]:
```
array([[0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]])
```

[5]:
```
arr1d.shape
```

[5]:
```
(4, 4)
```

[31]:
```
import numpy as np
import random



# helper functionss
```

```python
def calculate_fitness(chromosome):
    return np.count_nonzero(chromosome == 1)

def selection(population, fitnesses):
    fitnesses = np.array(fitnesses) + 1e-6
    # selection probabilities based on fitness values
    total_fitness = sum(fitnesses)
    selection_probabilities = [fitness / total_fitness for fitness in fitnesses]
    # normalizing the selection probabilities... Probability rule
    selection_probabilities = [prob / sum(selection_probabilities) for prob in selection_probabilities]
    # two parents randomly selected using the selection probabilities
    parents_indices = np.random.choice(len(population), size=2, replace=False, p=selection_probabilities)
    parents = [population[i] for i in parents_indices]
    return parents

def crossover(parents):
    # single-point crossover operation on the selected parents
    point_of_crossover = random.randint(0, len(parents[0]))
    child1 = np.concatenate((parents[0][:point_of_crossover], parents[1][point_of_crossover:]))
    child2 = np.concatenate((parents[1][:point_of_crossover], parents[0][point_of_crossover:]))
    return [child1, child2]

def mutation(children, mutation_rate):
    # mutation operator with a low probability value
    for i in range(len(children)):
        for j in range(len(children[i])):
            if np.random.rand() < mutation_rate:
                children[i][j] = 1 - children[i][j]
    return children

def genetic_algo(population, max_iter, mutation_rate):
    iteration = 0

    while iteration < max_iter:
        fitnesses = [calculate_fitness(chromosome) for chromosome in population]

        # checking for solution
        if np.max(fitnesses) == len(initial_chromosome):
            return population[np.argmax(fitnesses)]

        # parents selection
        parents = selection(population, fitnesses)
```

```
        # Offsprings
        children = crossover(parents)

        #  mutation to Offsprings
        children = mutation(children, mutation_rate)

        # calculating fitness values of the new children
        children_fitness = [calculate_fitness(child) for child in children]

        # check if any of the children is a solution
        if np.max(children_fitness) == len(oned_array):
            return children[np.argmax(children_fitness)]

        # replace the least fit members of the population with the new children
        worst_fitness_indices = np.argsort(fitnesses)[:len(children)]
        for i, child in enumerate(children):
            population[worst_fitness_indices[i]] = child

        iteration += 1

    # Return best solution found
    return population[np.argmax(fitnesses)]

# the initial and goal states
initial_state = np.zeros((4, 4), dtype=int)
goal_state = np.ones((4, 4), dtype=int)


oned_array = initial_state.flatten()

# for initial population of chromosomes
population_size = 20
chromosome_length = 16

# 1D NumPy array of zeros
population = np.zeros((population_size, chromosome_length), dtype=int)
#print(population)

# randomly initializing the population with either 0 or 1
for i in range(population_size):
    for j in range(chromosome_length):
        population[i,j] = np.random.randint(2)

#print(population)
```

```python
max_iter = 1000
mutation_rate = 0.01

resultant_chromosome = genetic_algo(population, max_iter, mutation_rate)

resultant_chromosome
```

[31]:
```
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
```

[ ]: