Roll No: 101803186 Name: Kulpreet Singh Group: 3COE9

Computer Science and Engineering Department
Artificial Intelligence (UCS-521)
Lab Assignment-5

1. Solve the given 0/1 knapsack problem by considering the following points:

Name	Weight	Value
A	45	3
B	40	5
C	50	8
D	90	10

Chromosome is a 4-bit string. – {x_A x_B x_C x_D}
Population size = 4, Maximum Capacity of the bag (W) = 100.
First two fittest chromosomes selected as it is. 3rd and 4th fittest use for one-point crossover in the middle followed by single bit mutation of first offspring.
Bits chosen for mutation follows this cyclic order (x_D, x_C, x_B, x_A).
Initial population: {1 1 1 1, 1 0 0 0, 1 0 1 0, 1 0 0 1}.
Output the result after 10 iterations.

CODE:

```
import numpy as np

item_number = np.arange(1,5)
weight = [45, 40, 50, 90]
value = [3, 5, 8, 10]
knapsack_threshold = 100    #Maximum weight that the bag of thief can hold
print('The list is as follows:')
print('Item No.   Weight   Value')
for i in range(item_number.shape[0]):
    print('{0}          {1}          {2}'.format(chr(ord('A')+item_number[i]-
1), weight[i], value[i]))

solutions_per_pop = 4
pop_size = (solutions_per_pop, item_number.shape[0])
print('Population size = {}'.format(pop_size))
initial_population = np.array([[1, 1, 1, 1], [1, 0, 0, 0], [1, 0, 1, 0], [1,
0, 0, 1]])
num_generations = 10
print('Initial population: \n{}'.format(initial_population))


def cal_fitness(weight, value, population, threshold):
    fitness = np.empty(population.shape[0])
    for i in range(population.shape[0]):
        S1 = np.sum(population[i] * value)
        S2 = np.sum(population[i] * weight)
        if S2 <= threshold:
            fitness[i] = S1
        else :
            fitness[i] = 0
    return fitness.astype(int)


def selection(fitness, num_parents, population):
    fitness = list(fitness)
    parents = np.empty((num_parents, population.shape[1]))
    fittest = np.empty((num_parents, population.shape[1]))
    for i in range(num_parents):
```

```
        max_fitness_idx = np.where(fitness == np.max(fitness))
        fittest[i, :] = population[max_fitness_idx[0][0], :]
        fitness[max_fitness_idx[0][0]] = -999999
    for i in range(num_parents):
        max_fitness_idx = np.where(fitness == np.max(fitness))
        parents[i,:] = population[max_fitness_idx[0][0], :]
        fitness[max_fitness_idx[0][0]] = -999999
    return fittest, parents


def crossover(parents, num_offsprings):
    offsprings = np.empty((num_offsprings, parents.shape[1]))
    crossover_point = int(parents.shape[1]/2)
    offsprings[0,0:crossover_point] = parents[0,0:crossover_point]
    offsprings[0,crossover_point:] = parents[1,crossover_point:]
    offsprings[1,0:crossover_point] = parents[1,0:crossover_point]
    offsprings[1,crossover_point:] = parents[0,crossover_point:]
    return offsprings


def mutation(offsprings, mutation_index):
    mutants = np.empty((offsprings.shape))
    for i in range(mutants.shape[0]):
        mutants[i,:] = offsprings[i,:]
    if mutants[0, mutation_index] == 0:
        mutants[0, mutation_index] = 1
    else:
        mutants[0, mutation_index] = 0
    return mutants


def optimize(weight, value, population, pop_size, num_generations, threshold):
    parameters, fitness_history = [], []
    num_parents = int(pop_size[0] / 2)
    num_offsprings = pop_size[0] - num_parents
    mutation_order = [3, 2, 1, 0]
    j = 0
    for i in range(num_generations):
        fitness = cal_fitness(weight, value, population, threshold)
        fitness_history.append(fitness)
        fittest, parents = selection(fitness, num_parents, population)
        offsprings = crossover(parents, num_offsprings)
        mutation_index = mutation_order[j]
        j = (j+1) % pop_size[0]
        mutants = mutation(offsprings, mutation_index)
        population[0:parents.shape[0], :] = fittest
        population[parents.shape[0]:, :] = mutants

    print('Last generation: \n{}\n'.format(population))
    fitness_last_gen = cal_fitness(weight, value, population, threshold)
    print('Fitness of the last generation: \n{}\n'.format(fitness_last_gen))
    max_fitness = np.where(fitness_last_gen == np.max(fitness_last_gen))
    parameters.append(population[max_fitness[0][0], :])
    return parameters, fitness_history


parameters, fitness_history = optimize(weight, value, initial_population,
pop_size, num_generations, knapsack_threshold)
print('The optimized parameters for the given inputs are:
\n{}'.format(parameters))
selected_items = item_number * parameters
```

```python
print('\nSelected items that will maximize the knapsack without breaking it:')
for i in range(selected_items.shape[1]):
  if selected_items[0][i] != 0:
     print('{}'.format(chr(ord('A')+selected_items[0][i]-1)))
```

OUTPUT:

```
assign5_q1
C:\Users\kulpr\PycharmProjects\OpenCVpython\venv\Scripts\python.exe C:/Users/ku
The list is as follows:
Item No.   Weight   Value
A          45        3
B          40        5
C          50        8
D          90        10
Population size = (4, 4)
Initial population:
[[1 1 1 1]
 [1 0 0 0]
 [1 0 1 0]
 [1 0 0 1]]
Last generation:
[[1 0 1 0]
 [1 0 1 0]
 [0 0 1 0]
 [0 1 0 1]]

Fitness of the last generation:
[11 11  8  0]

The optimized parameters for the given inputs are:
[array([1, 0, 1, 0])]

Selected items that will maximize the knapsack without breaking it:
A
C

Process finished with exit code 0
```

2. A thief enters a house for robbing it. He can carry a maximal weight of 9 kg into his bag. There are 4 items in the house with the following weights and values. The thief has to plan the items he should take to maximize the total value if he either takes the item completely or leaves it completely?

Item	Item Name	Weight (in Kg)	Value (in $)
A	Mirror	2	3
B	Silver Nugget	3	5
C	Painting	4	7
D	Vase	5	9

The problem is solved using Genetic Algorithm with population size 4 and each individual encoded as $\{X_A, X_B, X_C, X_D\}$ where $X_i = \{0,1\}$ and i=A, B, C, D.
Consider initial population as 1111, 1000, 1010, and 1001.
Generate the population for next iteration as follows: Select the 1st and 2nd fittest individual as it is in the next iteration. Apply 1-point crossover in the middle between 3rd and 4th fittest chromosome followed by single bit mutation of first offspring (produced through crossover). Bit chosen for mutation follows this cyclic order $\{X_C, X_A, X_D, X_B\}$
Output the result after four iterations.

CODE:

```
import numpy as np

item_number = np.arange(1, 5)
weight = [2, 3, 4, 5]
value = [3, 5, 7, 9]
knapsack_threshold = 9  # Maximum weight that the bag of thief can hold
print('The list is as follows:')
print('Item No.   Weight   Value')
for i in range(item_number.shape[0]):
    print('{0}          {1}          {2}'.format(chr(ord('A') + item_number[i]
- 1), weight[i], value[i]))

solutions_per_pop = 4
pop_size = (solutions_per_pop, item_number.shape[0])
print('Population size = {}'.format(pop_size))
initial_population = np.array([[1, 1, 1, 1], [1, 0, 0, 0], [1, 0, 1, 0], [1,
0, 0, 1]])
num_generations = 4
print('Initial population: \n{}'.format(initial_population))


def cal_fitness(weight, value, population, threshold):
    fitness = np.empty(population.shape[0])
    for i in range(population.shape[0]):
        S1 = np.sum(population[i] * value)
        S2 = np.sum(population[i] * weight)
        if S2 <= threshold:
            fitness[i] = S1
        else:
            fitness[i] = 0
    return fitness.astype(int)


def selection(fitness, num_parents, population):
    fitness = list(fitness)
    parents = np.empty((num_parents, population.shape[1]))
    fittest = np.empty((num_parents, population.shape[1]))
    for i in range(num_parents):
        max_fitness_idx = np.where(fitness == np.max(fitness))
        fittest[i, :] = population[max_fitness_idx[0][0], :]
        fitness[max_fitness_idx[0][0]] = -999999

    for i in range(num_parents):
        max_fitness_idx = np.where(fitness == np.max(fitness))
        parents[i, :] = population[max_fitness_idx[0][0], :]
        fitness[max_fitness_idx[0][0]] = -999999
    return fittest, parents
```

```
def crossover(parents, num_offsprings):
    offsprings = np.empty((num_offsprings, parents.shape[1]))
    crossover_point = int(parents.shape[1] / 2)
    offsprings[0, 0:crossover_point] = parents[0, 0:crossover_point]
    offsprings[0, crossover_point:] = parents[1, crossover_point:]
    offsprings[1, 0:crossover_point] = parents[1, 0:crossover_point]
    offsprings[1, crossover_point:] = parents[0, crossover_point:]
    return offsprings


def mutation(offsprings, mutation_index):
    mutants = np.empty((offsprings.shape))
    for i in range(mutants.shape[0]):
        mutants[i, :] = offsprings[i, :]
    if mutants[0, mutation_index] == 0:
        mutants[0, mutation_index] = 1
    else:
        mutants[0, mutation_index] = 0
    return mutants


def optimize(weight, value, population, pop_size, num_generations, threshold):
    parameters, fitness_history = [], []
    num_parents = int(pop_size[0] / 2)
    num_offsprings = pop_size[0] - num_parents
    mutation_order = [2, 0, 3, 1]
    j = 0

    for i in range(num_generations):
        fitness = cal_fitness(weight, value, population, threshold)
        fitness_history.append(fitness)
        fittest, parents = selection(fitness, num_parents, population)
        offsprings = crossover(parents, num_offsprings)
        mutation_index = mutation_order[j]
        j = (j + 1) % pop_size[0]
        mutants = mutation(offsprings, mutation_index)
        population[0:parents.shape[0], :] = fittest
        population[parents.shape[0]:, :] = mutants

    print('Last generation: \n{}\n'.format(population))
    fitness_last_gen = cal_fitness(weight, value, population, threshold)
    print('Fitness of the last generation: \n{}\n'.format(fitness_last_gen))
    max_fitness = np.where(fitness_last_gen == np.max(fitness_last_gen))
    parameters.append(population[max_fitness[0][0], :])
    return parameters, fitness_history


parameters, fitness_history = optimize(weight, value, initial_population,
pop_size, num_generations, knapsack_threshold)

print('The optimized parameters for the given inputs are:
\n{}'.format(parameters))

selected_items = item_number * parameters
print('\nSelected items that will maximize the knapsack without breaking it:')

for i in range(selected_items.shape[1]):
    if selected_items[0][i] != 0:
        print('{}'.format(chr(ord('A') + selected_items[0][i] - 1)))
```

OUTPUT:

```
assign5_q2
C:\Users\kulpr\PycharmProjects\OpenCVpython\venv\Scripts\python.exe C:/Us
The list is as follows:
Item No.   Weight   Value
A          2         3
B          3         5
C          4         7
D          5         9
Population size = (4, 4)
Initial population:
[[1 1 1 1]
 [1 0 0 0]
 [1 0 1 0]
 [1 0 0 1]]
Last generation:
[[1 1 1 0]
 [1 0 0 1]
 [1 1 0 1]
 [0 0 0 1]]

Fitness of the last generation:
[15 12  0  9]

The optimized parameters for the given inputs are:
[array([1, 1, 1, 0])]

Selected items that will maximize the knapsack without breaking it:
A
B
C

Process finished with exit code 0
```

3. Consider the following 2-SAT problem with 4 Boolean variables a, b, c, d:
F=(¬a∨d)∧(c∨b) ∧ (¬c∨¬d) ∧ (¬d∨¬b) ∧ (¬a∨¬d)
The MOVEGEN function to generate new solution be arbitrary changing value of any one variable
Let the candidate solution be of the order (abcd) and the initial candidate solution be (1111).
Let heuristic to evaluate each solution be number of clauses satisfied in the formula.
Apply Simulated Annealing (Consider T= 500 and cooling function = T-50)
(Assume the following 3 random numbers:0.655,0.254.0.432)
Accept every good move and accept a bad move if probability is greater than 50%.

CODE:

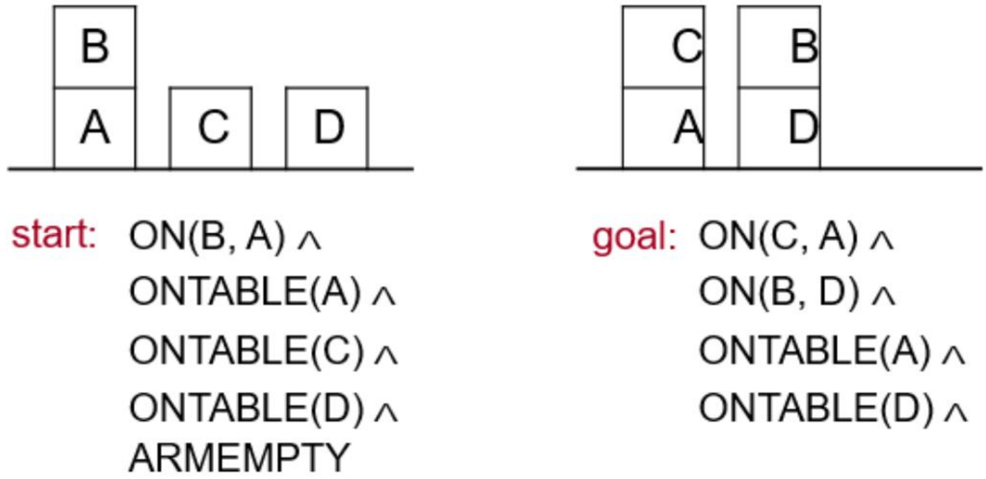
```
# F=(¬aVd)∧(cVb) ∧ ( ¬cV¬d) ∧ ( ¬dV¬b) ∧ (¬aV¬d)

import random
import numpy as np
import copy


def entropy(parent):
    a = parent[0]
    b = parent[1]
    c = parent[2]
    d = parent[3]
    term1 = 1 if ((1 - a) + d) > 0 else 0
    term2 = 1 if (c + b) > 0 else 0
    term3 = 1 if ((1 - c) + (1 - d)) > 0 else 0
    term4 = 1 if ((1 - d) + (1 - b)) > 0 else 0
    term5 = 1 if ((1 - a) + (1 - d)) > 0 else 0
    return term1 + term2 + term3 + term4 + term5


parent = [1, 1, 1, 1]
iterations = 10
result = []
T = 500


for i in range(iterations):
    print("Parent", parent)
    parent_heuristic = entropy(parent)
    if parent_heuristic == 5 and parent not in result:
        result.append(parent)
    child = copy.deepcopy(parent)
    mut = random.randint(0, 3)
    if parent[mut] == 0:
        child[mut] = 1
    else:
        child[mut] = 0
    print("Child", child)

    child_heuristic = entropy(child)
    E = child_heuristic - parent_heuristic
    cooling_func = 1 / (1 + np.exp(-E / T))
    print("Sigmoid Value", cooling_func)
    if E > 0:
        print('Good Move')
        parent = copy.deepcopy(child)
    elif cooling_func > 0.5:
        print('Bad Move')
        parent = copy.deepcopy(child)
    else:
        print('No Move')

    T = T - 50
    print()

print("Values of [a,b,c,d] which will make function F true after", iterations,
"iterations are", result)
```

OUTPUT:

assign5_q3

```
C:\Users\kulpr\PycharmProjects\OpenCVpython\venv\Scripts\python.exe C:/Users/kulpr/PycharmProjec
Parent [1, 1, 1, 1]
Child [1, 1, 0, 1]
Sigmoid Value 0.5004999998333334
Good Move

Parent [1, 1, 0, 1]
Child [1, 0, 0, 1]
Sigmoid Value 0.5
No Move

Parent [1, 1, 0, 1]
Child [1, 0, 0, 1]
Sigmoid Value 0.5
No Move

Parent [1, 1, 0, 1]
Child [1, 1, 1, 1]
Sigmoid Value 0.49928571477162254
No Move

Parent [1, 1, 0, 1]
Child [1, 1, 0, 0]
Sigmoid Value 0.5008333325617292
Good Move

Parent [1, 1, 0, 0]
Child [1, 0, 0, 0]
Sigmoid Value 0.49900000133333117
No Move

Parent [1, 1, 0, 0]
Child [1, 0, 0, 0]
Sigmoid Value 0.49875000260416025
No Move

Parent [1, 1, 0, 0]
Child [0, 1, 0, 0]
Sigmoid Value 0.5016666604938546
Good Move
Parent [0, 1, 0, 0]
Child [0, 1, 1, 0]
Sigmoid Value 0.5
No Move

Parent [0, 1, 0, 0]
Child [0, 1, 1, 0]
Sigmoid Value 0.5
No Move

Values of [a,b,c,d] which will make function F true after 10 iterations are [[0, 1, 0, 0]]

Process finished with exit code 0
```

4. For the given problem generate a plan:



Store the generated plan in a text file.

PLAN:

Start:
ON(B,A)^
ONTABLE(A)^
ONTABLE(C)^
ONTABLE(D)^
ARMEMPTY

Step 1: **UNSTACK(B,A)**
ONTABLE(A)
ONTABLE(C)
ONTABLE(D)
ARM(B)

Step 2: **PUTDOWN(B)**
ONTABLE(A)
ONTABLE(B)
ONTABLE(C)
ONTABLE(D)
ARMEMPTY

Step 3: **PICKUP(C)**
ONTABLE(A)
ONTABLE(B)
ONTABLE(D)
ARM(C)

Step 4: **STACK(C,A)**
ON(C,A)
ONTABLE(A)
ONTABLE(B)
ONTABLE(D)
ARMEMPTY

	Step 5: **PICKUP(B)** ON(C,A) ONTABLE(A) ONTABLE(D) ARM(B) Step 6: **STACK(B,D)** ON(C,A) ON(B,D) ONTABLE(A) ONTABLE(D) ARMEMPTY Goal State Reached!!