LAB 9 | Artificial Intelligence

Aim : Knapsack problem by using Genetic Algorithm.

Code :

```
import random
import sys
import operator
import random as rd
from random import randint,randrange

class Knapsack(object):

    #initialize variables and lists
    def __init__(self):

        self.C = 0
        self.weights = []
        self.profits = []
        self.parents = []
        self.newparents = []
        self.bests = []
        self.best_p = []
        self.iterated = 1
        self.population = 0
        self.best_all = []

        # increase max recursion for long stack
        iMaxStackSize = 15000
        sys.setrecursionlimit(iMaxStackSize)

    # create the initial population
    def initialize(self,size):

        for i in range(self.population):
            parent = []
            for k in range(0, size):
                k = random.randint(0, 1)
                parent.append(k)
            self.parents.append(parent)

    # set the details of this problem
    def properties(self, weights, profits, C, population,size):

        self.weights = weights
```

```
    self.profits = profits
    self.C = C
    self.population = population
    self.initialize(size)

# calculate the fitness function of each list (sack)
def fitness(self, item):

    sum_w = 0
    sum_p = 0

    # get weights and profits
    for index, i in enumerate(item):
        if i == 0:
            continue
        else:
            sum_w += self.weights[index]
            sum_p += self.profits[index]

    # if greater than the optimal return -1 or the number otherwise
    if sum_w > self.C:
        return -1
    else:
        return sum_p

# run generations of GA
def evaluation(self):

    # loop through parents and calculate fitness
    best_pop = self.population // 2
    for i in range(len(self.parents)):
        parent = self.parents[i]
        ft = self.fitness(parent)
        self.bests.append((ft, parent))

    # sort the fitness list by fitness
    self.bests.sort(key=operator.itemgetter(0), reverse=True)
    self.best_p = self.bests[:best_pop]
    self.best_p = [x[1] for x in self.best_p]

# mutate children after certain condition
def mutation(self, ch):

    for i in range(len(ch)):
        k = random.uniform(0, 1)
        if k > 0.5:
            #if random float number greater than 0.5 flip 0 with 1 and vice versa
```

```
            if ch[i] == 1:
                ch[i] = 0
            else:
                ch[i] = 1
        return ch

    # crossover two parents to produce two children by mixing them under random ration each
time
    def crossover(self, ch1, ch2):

        threshold = random.randint(1, len(ch1)-1)
        tmp1 = ch1[threshold:]
        tmp2 = ch2[threshold:]
        ch1 = ch1[:threshold]
        ch2 = ch2[:threshold]
        ch1.extend(tmp2)
        ch2.extend(tmp1)

        return ch1, ch2

    # run the GA algorithm
    def run(self,num_gen):
        for gen in range(num_gen):
            # run the evaluation once
            self.evaluation()
            self.best_all.append((self.iterated,self.bests[0][0],self.bests[0][1]))
            newparents = []
            pop = len(self.best_p)-1

            # create a list with unique random integers
            sample = random.sample(range(pop), pop)
            for i in sample:
                # select the random index of best children to randomize the process
                if i < pop-1:
                    r1 = self.best_p[i]
                    r2 = self.best_p[i+1]
                    nchild1, nchild2 = self.crossover(r1, r2)
                    newparents.append(nchild1)
                    newparents.append(nchild2)
                else:
                    r1 = self.best_p[i]
                    r2 = self.best_p[0]
                    nchild1, nchild2 = self.crossover(r1, r2)
                    newparents.append(nchild1)
                    newparents.append(nchild2)

            # mutate the new children and potential parents to ensure global optima found
```

```
            for i in range(len(newparents)):
                newparents[i] = self.mutation(newparents[i])

            self.iterated += 1
            self.parents = newparents
            self.bests = []
            self.best_p = []

#define number of items
num_items = 4

#Capacity
C = 12

population = 16
number_generations = 15
weights = [5,3,7,2]
values = [12,5,10,7]
print('The list is as follows:')
print('Item No.   Weight   Value')
for i in range(num_items):
    print('{0}         {1}         {2}\n'.format(i+1, weights[i], values[i]))


k = Knapsack()
k.properties(weights, values, C, population, num_items)
k.run(number_generations)
fitness_history_max = [fitness[1] for fitness in k.best_all]
k.best_all.sort(key=operator.itemgetter(1), reverse=True)
print("Best Profit is {} at generation {} by selecting the tuples in the following order :
{}".format(k.best_all[0][1],k.best_all[0][0],k.best_all[0][2]))
```

Output :

```
This list is as follows:
Item No.        Weight          Value
1               5               12
2               3                5
3               7               10
4               2                7

Best Profit is 24 at generation 3 by selecting the tuples in the following order : [1, 1, 0, 1]
```