Filip Uherek
Michał Wilkosz

Genetic Algorithms – Laboratory 4/05/20

A. Problem description

The objective of the laboratory task is to solve discrete knapsack problem using genetic algorithm.

The discrete knapsack problem is a problem in combinatorial optimization. With given weights and values of a certain number of N items. The optimization task is to achieve the highest value of items in fixed-size knapsack.

In the presented solution we assume that we can put it in the knapsack:

- N = 10 items whose weights and values have been randomly generated
- N = 15 items whose weights and values have been randomly generated

The following elements were used to implement the given problem using a genetic algorithm:

- Roulette wheel selection
- One point crossover
- One bit mutation

In the implementation of the problem, two following knapsack problems were considered

according to the equations:

$$\max \leftarrow v = \sum_{i=1}^{N} x_i \; v_i$$

$$w = \sum_{i=1}^{N} x_i \; w_i \leq w_{max}$$

1. First problem

$w_{max} = 20$

i	w_i	v_i
1	1	5
2	7	9
3	9	4
4	2	6
5	2	1
6	2	5
7	4	9
8	7	6
9	2	4
10	4	9

2. Second problem

$w_{max} = 40$

i	w_i	v_i
1	1	5
2	7	9
3	9	4
4	2	6
5	2	1
6	2	5
7	4	9
8	7	6
9	2	4
10	4	9
11	5	5
12	1	2
13	8	5
14	6	8
15	3	7

B. Program code and executable file

1. Program code:

```
import numpy as np
from numpy.random import choice

def create_population(chromosome_length,pop_size,threshold):
    index = np.arange(1,chromosome_length+1)
    #Create knapsack problem indexes.
    weights = np.random.randint(1,10,size=chromosome_length)
    values = np.random.randint(1,10,size=chromosome_length)
    #Assigns random weights and values.
    # chromosome = np.column_stack((index,weights,values))

    chromosome = np.array([(1,1,5), #Knapsack problem to chromosome of length 15.
                        (2,7,9),
                        (3,9,4),
                        (4,2,6),
                        (5,2,1),
                        (6,2,5),
                        (7,4,9),
                        (8,7,6),
                        (9,2,4),
                        (10,4,9),
                        (11,5,5),
                        (12,1,2),
                        (13,8,5),
                        (14,6,8),
                        (15,3,7)])

    # chromosome = np.array([(1,1,5), #Knapsack problem to chromosome of length 10.
    #                     (2,7,9),
```

```
    #                   (3,9,4),
    #                   (4,2,6),
    #                   (5,2,1),
    #                   (6,2,5),
    #                   (7,4,9),
    #                   (8,7,6),
    #                   (9,2,4),
    #                   (10,4,9)])

    #Creates obtained knapsack problem array.
    print("KNAPSACK PROBLEM")
    print("Knapsack capacity V={}".format(threshold))
    print("index;volume;benefit")

    print(chromosome)
    population = np.zeros((pop_size,len(chromosome)))
    weight_value = np.zeros((pop_size,2))
    insert = 0
    while insert != pop_size:
        row = np.random.randint(2, size=(len(chromosome)))
        weight = 0
        value = 0
        for x in range(len(row)):
            if row[x] == 1:
                weight += chromosome[x,1]
                value += chromosome[x,2]
            else:
                pass
        if weight <= threshold:
            population[insert] = row
            weight_value[insert]= weight,value
            insert += 1
        else:
            pass
    #Creates a population of chromosomes that meet the backpack's capacity threshold.
    return population, weight_value, chromosome
    #Returns entire population, its weights and values to particular chromosomes.

def selection(population, weight_value, num_parents):
    pop = population
    wval = weight_value
    #Copy population of chromosomes and its weights and values.
    values = weight_value[:,1].copy()
    chromosome_len = len(population[0])
    selected_chromosomes = np.empty((num_parents,chromosome_len))
    #Empty array for chromosomes which will be chosen in roulette wheel selection.
    selected_values = []
    #Empty array for chromosomes weigths and values which will be chosen in roulette wheel
selection.
    chosen_indx = []
    for row in range(num_parents):
        #For the number of parents who will create new offspring by crossover.
        probabilities = []
        #Population probability list.
        sum_values = sum(values)
        for i in range(len(population)):
            probabilities.append(values[i]/sum_values)
```

```
            #Calculating the probability of a given population.
        index = choice(population.shape[0], 1, p=probabilities)
        #Chosing index of chromosome via roulette wheel selection.
        selected_chromosomes[row] = population[index]
        #Assigns chosen chromosomes.
        values[int(index)] = 0
        #Deletes the value of the selected chromosome to change the population probability.
    return selected_chromosomes, pop, wval

def crossover(selected_chromosomes, offspring_size, num_parents):
    idx = np.random.randint(num_parents, size=offspring_size)
    #Choses random indexes of elements which were chosed in roulette wheel selection.
    offspring = selected_chromosomes[idx]
    #Chooses offspring
    crossover_result = np.empty((offspring_size,len(offspring[0])))
    split_point = len(offspring[0])//2
    #Chose point in which chromosomes will be splitted.
    pairs = [((i), (i + 1) % len(offspring))
            for i in range(len(offspring))]
    #Pairs of chromosomes that are subject to crossing.
    for pair in pairs:
        parent_1 = offspring[pair[0]]
        parent_2 = offspring[pair[1]]
        #Chose two parents to crossover.
        if(len(offspring[0])%2==0):
        #If modulo of chromosome length is equal to 0, then split in the middle of chromosome.
            parent_1_corss = np.concatenate((parent_1[:split_point],parent_2[-split_point:]))
            parent_2_corss = np.concatenate((parent_2[:split_point],parent_1[-split_point:]))
        if(len(offspring[0])%2==1):
        #If modulo of chromosome length is equal to 1, then split in the right proportion.
            parent_1_corss = np.concatenate((parent_1[:split_point],parent_2[(-split_point-
1):]))
            parent_2_corss = np.concatenate((parent_2[:split_point],parent_1[(-split_point-
1):]))
        crossover_result[pair[0]] = parent_1_corss
        crossover_result[pair[1]] = parent_2_corss
    return crossover_result

def evaluate_population(population, chromosome, threshold):
    threshold_population = np.zeros((len(population),len(chromosome)))
    #Creates empty array for elements of population, which meet the capacity of the knapsack.
    threshold_weight_value = np.zeros((len(population),2))
    #Creates empty array for weights and values of elements population.
    insert = 0
    column = len(chromosome)
    while insert != len(population):
        #Retrieve the entire population of items
        weight = 0
        value = 0
        for i in range(column):
            if population[insert,i] == 1:
                weight += chromosome[i,1]
                value += chromosome[i,2]
            else:
                pass
        if weight <= threshold:
        #Checks if a given element weight, meets the capacity of the knapsack.
```

```
            threshold_population[insert] = population[insert]
            threshold_weight_value[insert] = weight,value
            insert += 1
        else:
        #If the weight of the element does not match the capacity of the backpack, it fills the
chromosome with zeros.
            threshold_population[insert] = np.zeros((1,len(chromosome)))
            threshold_weight_value[insert] = np.zeros((1,2))
            insert +=1
    sort_order=np.argsort(threshold_weight_value[:,1])
    #Sorts the entire population and the corresponding weight and values.
    threshold_weight_value = threshold_weight_value[sort_order]
    threshold_population = threshold_population[sort_order]
    return threshold_population, threshold_weight_value

def mutate_offspring(population):
    #Choses random element of chromosome and flips one gene for population chromosomes.
    for i in population:
        gene = np.random.randint(0,len(population[0]))
        i[gene] = 0 if i[gene] == 1 else 1
    return(population)

def next_gen(crossover_result, crossover_values, parents_left, parents_left_value):
    #Combines the chromosomes of the population of parents and chromosomes obtained as a result
of roulette wheel           selection, crossover and mutation.
    new_pop = crossover_result[~np.all(crossover_result == 0, axis=1)]
    new_values = crossover_values[~np.all(crossover_values == 0, axis=1)]
    new_population = np.concatenate((new_pop, parents_left), axis=0)
    new_population_val = np.concatenate((new_values, parents_left_value), axis=0)
    chosen_index = new_population_val[:,1].argsort()[(len(new_values)):][::1]
    chosen_population = new_population[chosen_index]
    chosen_values = new_population_val[chosen_index]
    return chosen_population, chosen_values

def
knapsack_ga(num_generations,chromosome,population,weight_value,knapsack_threshold,pairs_crossed,
crossover_size):
    #Main knapsack genetic algorithm funciton.
    best_chromosome = 0
    #List for best chromosomes in each iteration.
    best_value = np.zeros((2,))
    #List for best values in each iteration.
    iteration = 0
    for i in range(num_generations):
        #Main loop.
        selected_chromosomes, parents_left, parents_left_values = selection(population,
weight_value, crossover_size)
        #Selection.
        crossover_result = crossover(selected_chromosomes, pairs_crossed,crossover_size)
        #Crossover.
        mutate_offspring(crossover_result)
        #Mutation.
        ev_population, ev_population_values = evaluate_population(crossover_result,
chromosome,knapsack_threshold)
        #Evaluation of elements after crossover and mutations
        population, weight_value = next_gen(ev_population, ev_population_values, parents_left,
parents_left_values)
```

```
        #Evaluation of elements from crossover and parents population.
        #Creating new population.
        if weight_value[-1,1] > best_value[1]:
            best_value = weight_value[-1]
            best_chromosome = population[-1]
        iteration+=1
    print("OBTAINED VOLUME;BENEFIT{}".format(best_value,best_chromosome))
    print("CHROMOSOME:{}".format(best_chromosome))

chromosome_length = 15 #Chromosome size(N)
population_size = 1000 #Population size
knapsack_capacity = 40 #Knapsack capcity
num_generation = 10 #Population of size
pairs_crossed = 800 #Pairs crossed
crossover_size = 800 #Parents from gneration-1 chosen to create pairs

population, weight_value,chromosome =
create_population(chromosome_length,population_size,knapsack_capacity)

if __name__ == "__main__":


knapsack_ga(num_generation,chromosome,population,weight_value,knapsack_capacity,pairs_crossed,cr
ossover_size)

    pass
```

2. Executable file

Presented solution was developed in Python language with external library - Numpy(because of easier matrix operations). In connection with the necessity to contain used libraries. exe file exceed the allowable capacity which is possible to upload on PZE. A popular python interpreter like Anaconda allows to run the program without any problem. Therefore, here is a link which allows to download the whole program in .exe.

Link: https://drive.google.com/file/d/1gu9S42NFLV51QCaVtkdYNJPB5Lswz6VQ/view?usp=sharing

C. Solution of the problem obtained for various parameters

There is an additional parameter in the implementation of the problem – crossover size, parameter refers to number of parents selected to create crossover. Where it is equal to numbers of pairs being crossed, they form one parameter just numbers of pairs being crossed. However, it is possible to create a much larger number of pairs from certain number of parents and in evaluation the chromosomes after crossover with the best value will be selected.

1. First problem

Generations	Population size	Numbers of pairs being crossed	Crossover size	Chromosome size N	Volume	Benefit
10	10	8	8	10	20	41
10	10	80	8	10	20	42
10	10	800	8	10	20	43
10	100	8	8	10	20	43
10	100	80	80	10	20	43
10	100	800	80	10	20	43
10	1000	800	800	10	20	43

Best benefit chromosome : [1. 1. 0. 1. 0. 1. 1. 0. 0. 1.]

2. Second problem

Generations	Population size	Numbers of pairs being crossed	Crossover size	Chromosome size N	Volume	Benefit
10	10	8	8	15	40	63
10	10	80	8	15	39	67
10	10	800	8	15	39	70
10	100	8	8	15	37	66
10	100	80	80	15	40	69
10	100	800	80	15	39	70
10	1000	800	800	15	39	70

Best benefit chromosome : [1. 1. 0. 1. 1. 1. 1. 0. 1. 1. 1. 1. 0. 1. 1.]

Conclusions:

- In case of knapsack problem for problem with chromosome length = 10. The algorithm quickly found the optimal solution even for a small initial population. This is due to the simplicity of the problem.
- The number of parents from previous population selected to produce offspring has the highest impact on finding the optimal solution.
- Large number of pairs formed from the previous population allows for optimal solution even when the number of parents was low.