Lab3 – AI

Zinat Abo Hgool 206721714

Elie Hadad 207931536

How we translated the input:

In order to implement our algorithms according to the input samples we have got, we added a new file 'GetInput' that help us with taking the relevant information from the samples file and assign the propitiate values to the arguments.

Here is a little bit of our code: (you can check the source file for the full code)

```
def GetInput(name):
        file = open(r"C:\Users\WIN10\Desktop\CVRP\CVRP\problem1.txt")
        file.readline()
        file.readline()
        file.readline()
        line=file.readline()
        words = line.split()
        dimension=int(words[2])
        file.readline()
        line = file.readline()
        words = line.split()
        capacity=int(words[2])
        file.readline()
        citylst=[]
        i=0
        for l in range(dimension):
            line = file.readline()
            words = line.split(' ')
            city=City(int(words[0]),int(words[1]),int(words[2]))
            citylst.append(city)
        file.readline()
        for l in range(dimension):
            line = file.readline()
            words = line.split(' ')
            citylst.__getitem__(i).setDemand(int(words[1]))
            i += 1
```

In order to translate the problem that was given in the assignment "CVRP", we added new file 'CVRP' which included the following functions:

1)function to calculate the distance between two cities in the map and to place this distance in the propriate indices in the distance matrix.

```
def Distance_mat(self):
    Cities = self.Cities
    numCities = len(Cities)
    rows, cols = numCities, numCities
    arr_row = []
    for i in range(rows):
        arr_col = []
        for j in range(cols):
            dx = (Cities[i].Xcor - Cities[j].Xcor) * (Cities[i].Xcor - Cities[j].Xcor)
            dy = (Cities[i].Ycor - Cities[j].Ycor) * (Cities[i].Ycor - Cities[j].Ycor)
            distance = sqrt(dx + dy)
            arr_col.append(distance)
        arr_row.append(arr_col)
    return arr_row
```

2) function to print the output in the specific order(matrix) as the instruction given in the assignment

```
def print_result(self):
    print(self.best_cost)
    all = self.Vehicles_tour(self.best_tour)
    for tour in all:
        print(*tour, sep=' ')
```

3) function that returns the path for every vehicle given the whole path permutation.

Here is a little bit of the code:

```
def Vehicles_tour(self, tour): #this function returns the path for every vehicle given the whole path permutation
    first = 0 #index for the first of the vehicle's path
    last = 0  #index for the last of the vehicle's path
    veh_capacity = self.Capacity
    veh_arr = []
    veh_capacity -= self.Cities[tour[last] - 1].demand
    tour_length = len(tour)
    veh_num = 1
    while last < tour_length - 1:
        curr = tour[last + 1]
        if self.Cities[curr - 1].demand <= veh_capacity:
            veh_capacity -= self.Cities[curr - 1].demand #this vehicle can supply the demand of this city
        else:
            veh_arr.append(tour[first: last+1])
            first = last + 1
            veh_num += 1
            veh_capacity = self.Capacity - self.Cities[curr - 1].demand
        last += 1
    veh_arr.append(tour[first:tour_length])
```

4) function that calculates the cost of a given a tour

```
def tour_cost_veh(self, tour):# this function calculates the cost of the veh
    matrix = self.matrix
    count = 0
    count += matrix[tour[0]][0]
    last = len(tour)
    veh_capacity = self.Capacity - self.Cities[tour[0] - 1].demand
    for i in range(last - 2):
        curr = tour[i]
        target = tour[i+1]
        if self.Cities[target-1].demand <= veh_capacity:
            veh_capacity -= self.Cities[target-1].demand
            count += matrix[curr][target]
        else:
            count += matrix[curr][0] + matrix[target][0]
            veh_capacity = self.Capacity - self.Cities[target - 1].demand

    count += matrix[tour[last - 1]][0]
    return count
```

Heuristics we used:

We added new file named 'heuristic' with in it we declared the nearest neighbor heuristic that we learned, using this heuristic we select at first random member of the permutation and apply inverse mutation that we used in previous lab, this heuristic is helpful, because using it we can reach the optimal solution (the shortest path that can supply as mush as possible the cities' demands) here is a little bit of the code:

```
def get_best_neighbor(problem, size):
    arr = []
    city = randint(1, size)
    arr.append(city)
    is_available = {city: 1}
    index = size - 1
    while index > 0:
        matrix = problem.Distance_mat()
        sub_arr = matrix[city]
        min_dis = float('inf')
        for i in range(1, len(sub_arr)):
            if sub_arr[i] < min_dis and is_available.get(i) != 1:
                min_dis = sub_arr[i]
                min_city = i
        arr.append(min_city)
        is_available[min_city] = 1
        index -= 1
        city = min_city
    return arr
```

```
def get_all_neighborhood(best, size):
    neighborhood = []
    for i in range(size):
        neighborhood.append(simple_inverse_mutate(best))
    return neighborhood

def simple_inverse_mutate(member):#inverse the string between specific range
    tmp = member[:]
    size = len(member)
    posi = randint(0, size-2)
    posj = randint(posi+1, size-1)
    if posi > posj:
        posi, posj = posj, posi
    while posi < posj:
        tmp[posi], tmp[posj] = tmp[posj], tmp[posi]
        posi += 1
        posj -= 1
    return tmp
```

Some illustrations:

***In every algorithm we added the CPU and elapsed time, you can see it in the output pictures that we took.**

***Most of the output result you will see were produced by giving the algorithms these values:**

maxiter,lastN,popsize,eliteRate, mutationRate

(50, 1000, 20, 0.1, 0.5)

***we downloaded the input folder that has some input values for our problem on our pc, we had run multiply examples on our different algorithms, but most of the result you will see in the picture were token when we run file 1 of the input files.**

*** We changed the city's id that was in the input file, according to the warehouse, we started the warehouse's id = 0 and so on until the last city's id = 21 instead of 22. So it can be easy for us to print the output as required. (the output matrix)**

Simulated Annealing:

Here is a little bit of our code implementing the simulated annealing algorithm:

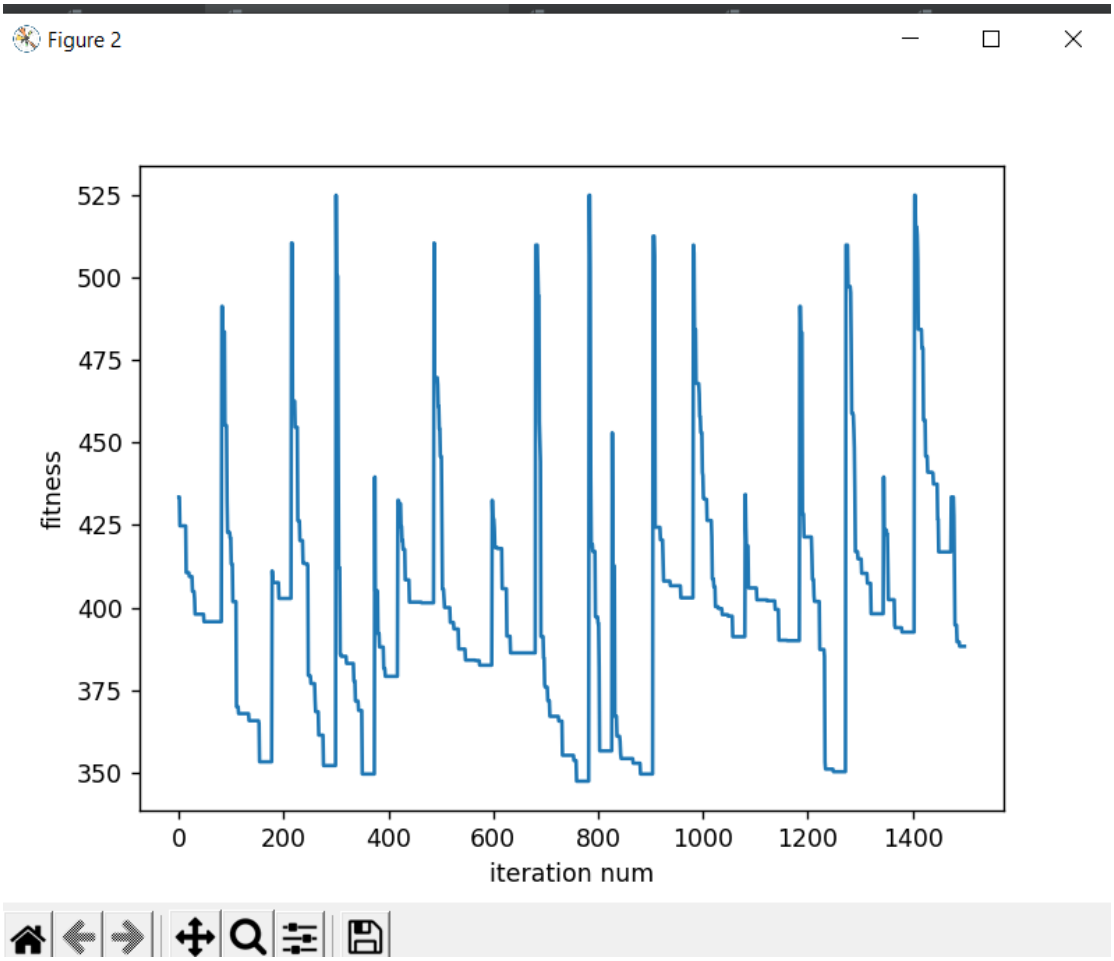
```
def Simulated_Annealing(problem, temp, alpha, neighbor_size, maxIter, stop):
    start = time.time()
    size = len(problem.Cities)
    best_neighbor = get_best_neighbor(problem, size)
    best_tour = problem.tour_cost_veh(best_neighbor, problem.Cities)
    curr_best_neigh = best_neighbor
    curr_best_tour = best_tour
    final_best_neigh = best_neighbor
    final_best_tour = best_tour
    k = 30
    optimum = 0
    temp1 = float(temp)
    y = []
    for i in range(maxIter):
        iterTime = time.time()
        neighborhood = get_all_neighborhood(best_neighbor, neighbor_size)
        y.append(best_tour)
        for j in range(k):
            index = randint(0, len(neighborhood) - 1)
            randNeigh = neighborhood[index]
            randTour = problem.tour_cost_veh(randNeigh, problem.Cities)
            d = randTour - best_tour
            result = float(exp(float(-1 * d) / temp1))
```

And when we run our code on the 'sample 1' file (E-n22-k4) we have got the following output, with the following elapsed and CPU time:

```
result:  [9, 7, 5, 2, 1, 3, 12, 15, 18, 20, 17, 14, 19, 21, 16, 10, 6, 8, 11, 4, 13]
total:  382.5126809995335
clock ticks: 0.007652997970581055
result:  [9, 7, 5, 2, 1, 3, 12, 15, 18, 20, 17, 14, 21, 19, 16, 10, 6, 8, 11, 4, 13]
total:  380.6688703563424
clock ticks: 0.008371114730834961
Time elapsed:  12.862154960632324
338.938745100024
0 10 8 6 3 4 11 13 0
0 16 19 21 0
0 12 15 18 20 17 0
0 9 7 5 2 1 14 0

Process finished with exit code 0
```

Here is a graph that shows the relation between the fitness(cost) and the num of the iteration:



COOPERATIVE PSO:

In order to implement cooperative PSO algorithm that we learned in the lecture we updated the velocity of each particle and the position to be equal to:

```
def move(self, gbest_pos, c1, c2, W):
      gloabl_best = self.all_dist()
      i = c1 * random() * (gloabl_best - self.all_dist(self.position))
      j = c2 * random() * (gloabl_best - self.all_dist(self.position))
      self.velocity = int(self.velocity * random() * W + i + j)
      if self.velocity < 0:
          self.velocity *= -1
      self.velocity = self.velocity % len(self.position)

      for i in range(len(self.position)/2):
          newhold = self.position[i]
          self.position[i] = self.position[(i+self.velocity) % len(self.position)]
          self.position[(i + self.velocity) % len(self.position)] = newhold
```

And we updated the PSO parameters as the equation we saw in the lecture:

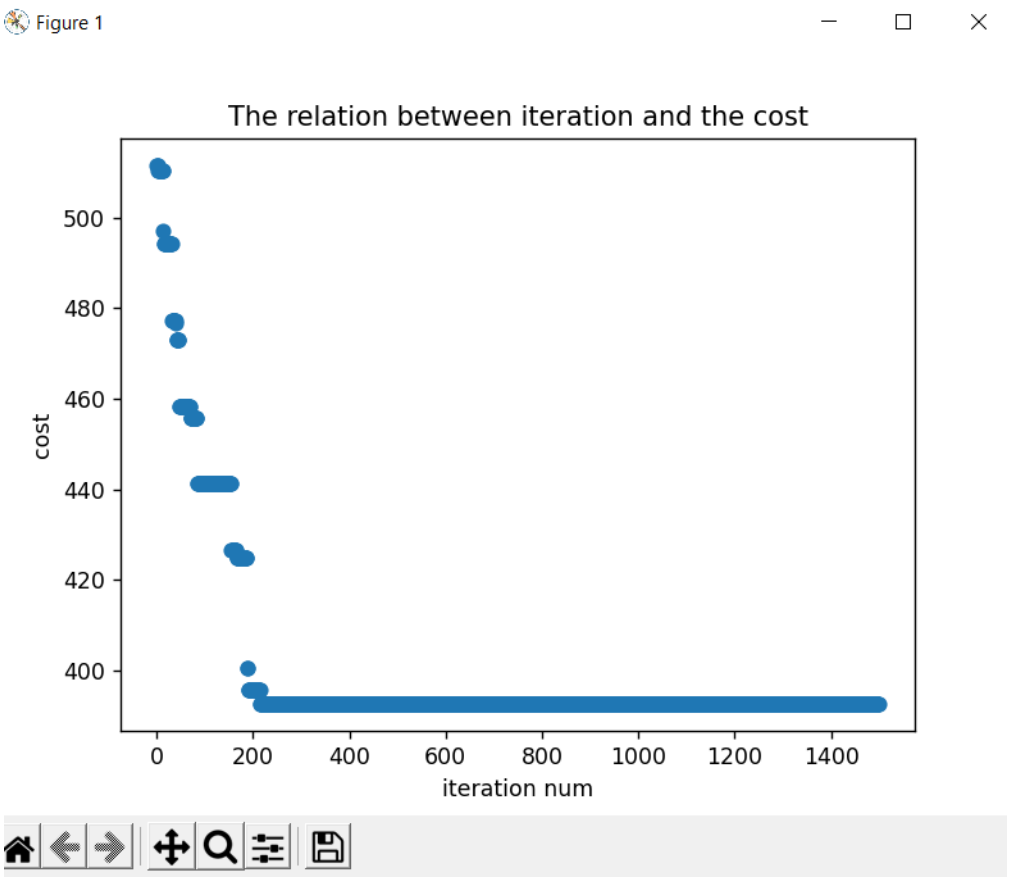
```
def update_para(self, index, size):#function to update pso parameters
      self.c1 = -3 * (index / size) + 3.5
      self.c2 = 3 * (index / size) + 0.5
      self.W = 0.4 * ((index - size)/ size ** 2) + 0.4
```

Here is the output of PSO on file (E-n22-k4) :

```
result:  [9, 7, 5, 6, 8, 10, 20, 18, 15, 12, 14, 13, 11, 4, 3, 1, 2, 17, 21, 19, 16]
total:  397.2280914716784
clock ticks: 0.020984649658203125
Time elapsed:  31.93283987045288
397.2280914716784
0 9 7 5 6 8 10 0
0 20 18 15 12 0
0 14 13 11 4 3 1 2 0
0 17 21 19 16 0

Process finished with exit code 0
```

Here is the graph:



TabuSearch:

here is a little bit of our code implementing Tabu search algorithm:

```
def start(self):
    lst=[]
    best=get_best_neighbor(self.cvrp, len(self.cvrp.Cities))
    random.shuffle(best)
    fitness = self.cvrp.tour_cost_veh(best) #best_fitness->fitness
    lst.append(fitness)
    temp = best #bestcandidate->temp
    recent = {str(best): True}#check_at_the_end
    y = []

    recentcities = [best]
    for j in range(self.args.maxiter):
        iterTime = time.time()
        neighborhood = get_all_neighborhood(temp, self.cvrp.Dimension)
        y.append(fitness)
        #neighborhood = self.findneighbor(temp)  # get neighborhood of current solution
        temp = neighborhood[0]
        min= self.cvrp.tour_cost_veh(temp)

        for n in neighborhood:
            cost= self.cvrp.tour_cost_veh(n)
            if cost < min and not recent.get(str(n), False):
                min = cost
                temp = n
        if min < fitness:  # update best (take a step towards the better neighbor)
            fitness = min
            best = temp
```

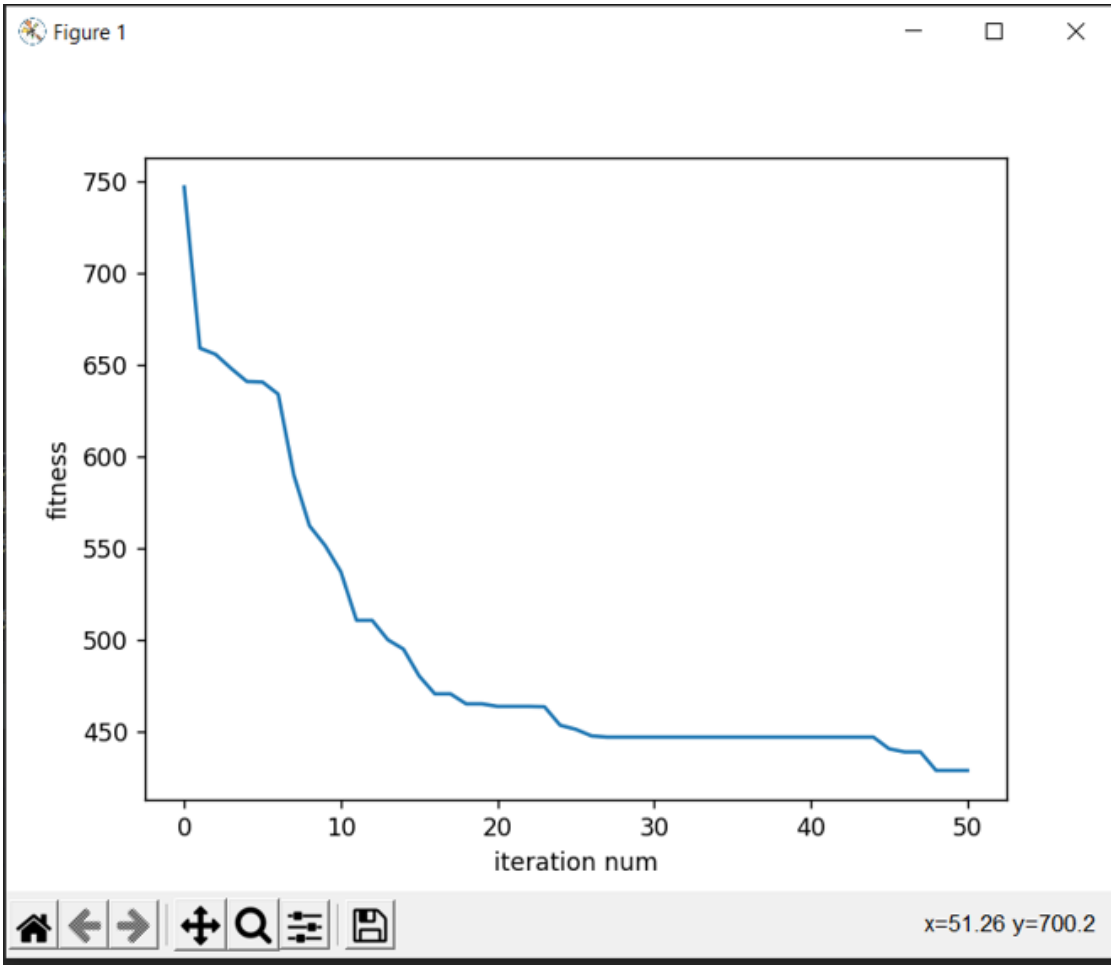
TabuSearch › SwapMove()

Here is the output:

```
cost is:  446.68599869023416
best solution is:  [10, 6, 8, 1, 3, 4, 11, 5, 2, 9, 7, 12, 14, 15, 17, 21, 20, 18, 13, 16, 19]
cost is:  440.30135500681644
best solution is:  [10, 6, 8, 1, 3, 4, 11, 5, 2, 9, 7, 12, 19, 16, 13, 18, 20, 21, 17, 15, 14]
cost is:  438.5882661626099
best solution is:  [10, 6, 8, 1, 3, 4, 11, 5, 2, 9, 7, 12, 19, 16, 13, 18, 20, 21, 17, 15, 14]
cost is:  438.5882661626099
best solution is:  [11, 4, 3, 1, 8, 6, 10, 5, 2, 9, 7, 12, 16, 19, 13, 18, 20, 21, 17, 15, 14]
cost is:  428.48624381760254
best solution is:  [11, 4, 3, 1, 8, 6, 10, 7, 9, 2, 5, 12, 16, 19, 13, 18, 20, 21, 17, 15, 14]
cost is:  428.46658000271617
best solution is:  [11, 4, 3, 1, 8, 6, 10, 7, 9, 2, 5, 12, 16, 19, 13, 18, 20, 21, 17, 15, 14]
cost is:  428.46658000271617
428.46658000271617
0 11 4 3 1 8 6 10 0
0 7 9 2 5 12 0
0 16 19 13 0
0 18 20 21 17 15 14 0

Process finished with exit code 0
```

Here is the graph of the fitness and the iteration number



GA- Island model:

Here is a little bit of our code implementing GA in island model, we take the code of GA that we build in the last lab assignment and we change it a little bit so it can be fitted to our problem using the heuristics.

Here is a little bit of the code:

```
def gastart(self):
    start = time.time()
    fitlst = []
    final_best = float('inf')
    self.init_population()
    for i in range(self.args.maxiter):
        iteration_Time = time.time()
        self.calcfitness()
        self.sort_by_fitness(self.population)
        self.print_best(self.population)
        fitlst.append(self.population[0].fitness)
        if self.population[0].fitness == 0:
            print('Clock tiks: ', time.time() - iteration_Time)
            print()
            print("Best string: " + str(self.population[0].str))
            print(f'found in {i} iterations out of {self.args.maxiter
            break
        best = self.population[0].getString()[:]
        best_fitness = self.population[0].fitness
        self.mate(self.population, self.buffer)
        self.population, self.buffer = self.buffer, self.population
        print('Clock tiks: ', time.time() - iteration_Time)
    self.cvrp.best_tour = best
    self.cvrp.best_cost = best_fitness
    print('Time elapsed: ', time.time() - start)
    return fitlst
```

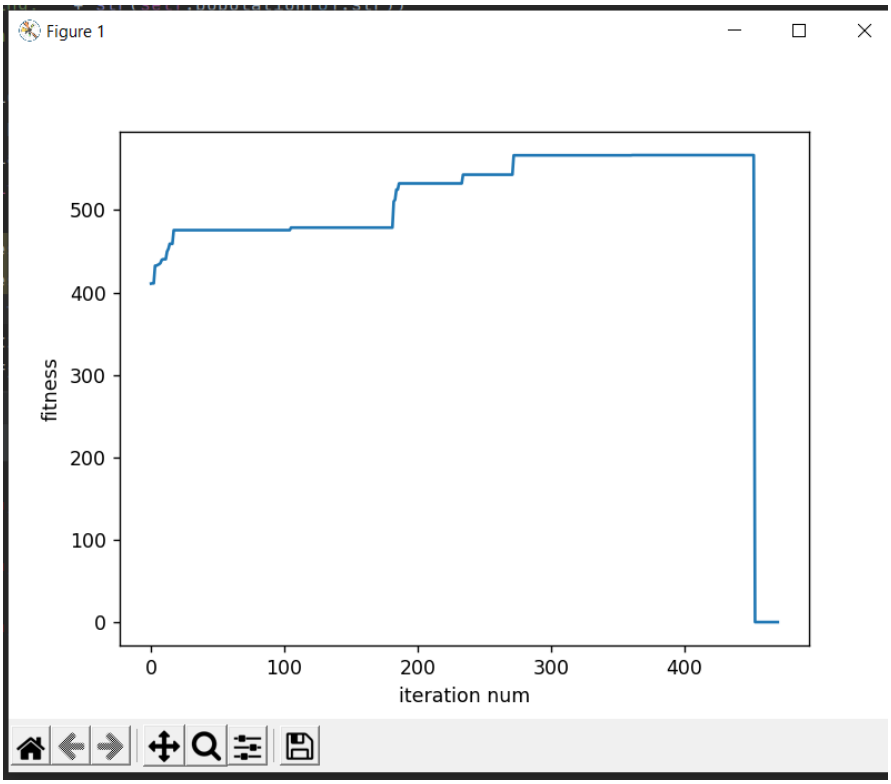
And here is the calcfitness() function:

```
def calcfitness(self):
    fitness = 0
    for gene in self.population:
        fitness = self.cvrp.tour_cost_veh(gene.str)
        gene.fitness = fitness
```

Here is the output:

```
Clock tiks:  0.005937337875366211
Best:  [1, 2, 5, 7, 9, 10, 8, 6, 3, 4, 11, 13, 16, 14, 12, 15, 18, 20, 17, 21, 19] ( 410.7688735263276 )
Clock tiks:  0.0049860477447509766
Time elapsed:  0.11830615997314453
410.7688735263276
0 1 2 5 7 9 10 8 0
0 6 3 4 11 13 0
0 16 14 12 15 18 0
0 20 17 21 19 0
```

Here is the graph, that show the fitness of every gene in the population as iteration number increase:



ACO:

in this algorithm we used many help functions, you can find them in 'cvrp.py' file:

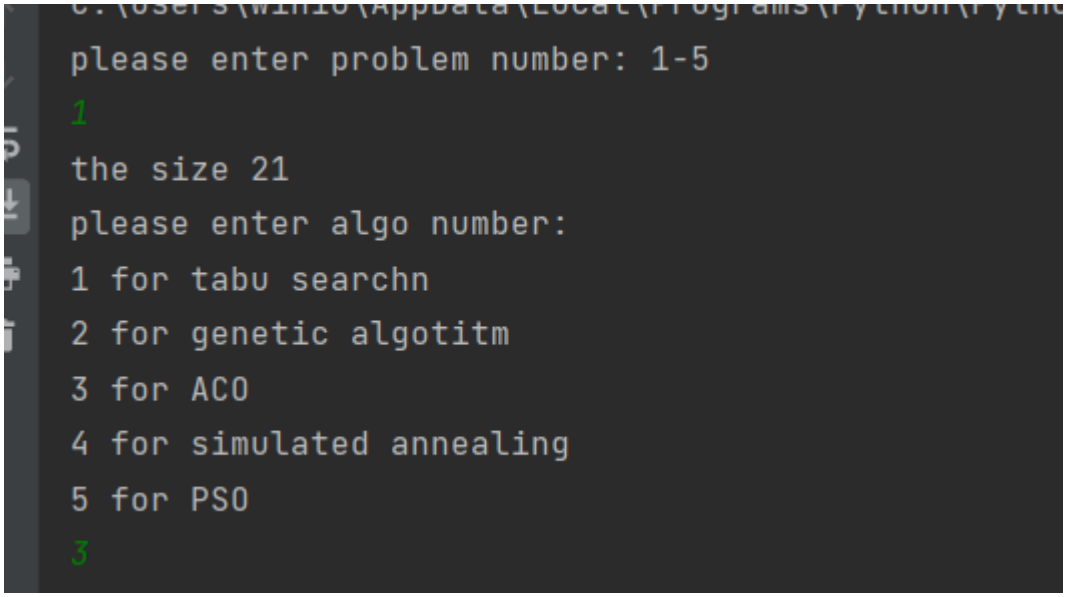
Here is a little bit of our code implementing ACO

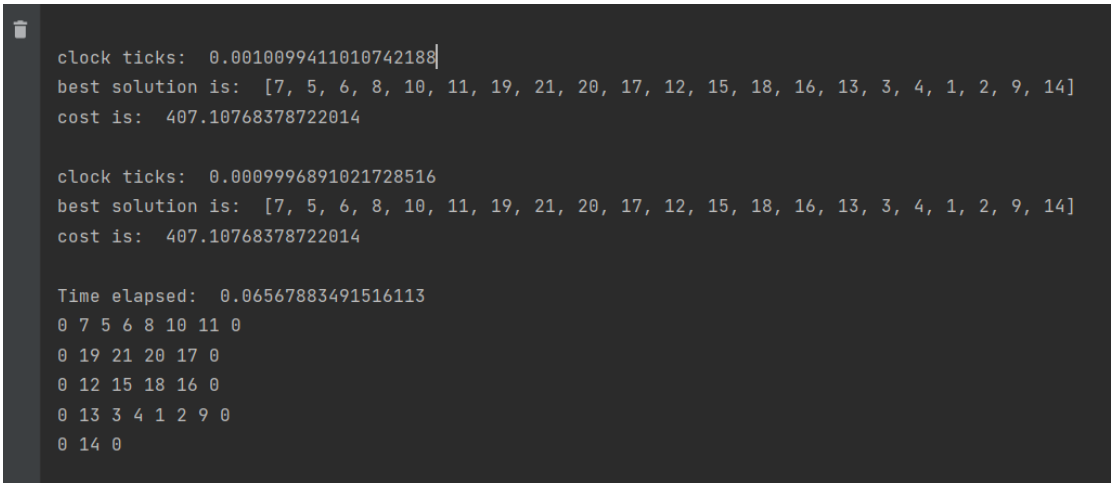
```python
def ACO(cv:CVRP, args):
    startTime = time.time()
    # start time
    pmatrix = [[float(1000) for _ in range(len(cv.Cities))] for _ in range(len(cv.Cities))]
    # initialize the p matrix
    bestPath = []
    fitlst=[]
    bestFitness = float('inf') #start with infinite fitness
    currentPath = []
    currentFitness = float('inf')
    globalBest = []
    globalFitness = float('inf')
    counter = 0
    for k in range(args.maxiter):
        iterTime = time.time()
        temp = getPath(cv, pmatrix)
        tempFitness = cv.tour_cost_veh(temp)
        if tempFitness < currentFitness:
            currentFitness = tempFitness
            currentPath = temp
        if currentFitness < bestFitness:  # update best (take a step towards the better neigh
            bestFitness = currentFitness
            bestPath = currentPath
```

```python
def acophmatrix(acophmatrix, path, cost):
    current = [[float(0) for _ in range(len(acophmatrix[0]))] for _ in range(len(acophmatrix[0]))]
    for i in range(len(path) - 1):
        current[path[i] - 1][path[i + 1] - 1] = float(float(2) / float(cost))
        current[path[i + 1] - 1][path[i] - 1] = float(float(2) / float(cost))

    for i in range(len(acophmatrix[0])):
        for j in range(len(acophmatrix[0])):
            num1 = float(acophmatrix[i][j] * (1 - 2))
            num2 = float(current[i][j] * 2)
            acophmatrix[i][j] = float(num1 + num2)
            if acophmatrix[i][j] < 0.0001:
                acophmatrix[i][j] = 0.0001
```

Here is the output:





Here is the fitness graph: