

## Assignment 1 - Evolutionary Computation

- Bruno Jerkovic s1061740
- Maksim Popov s1058129
- Christos Mavrikis s1059094

The link to the notebook on the Github is here:

<https://github.com/brunogreen25/EvolutionaryComputing> .

In [1]:

```

1 import numpy as np
2 import random, math
3 import matplotlib.pyplot as plt
4 %matplotlib inline

```

### Exercise 1 - (Schemata)

We have two schemata A1 and A2 consisting of 9 bits. In order to get further insight, we should calculate the order of each schema. The order of a schema is the number of each fixed bits.

- $o(A1) = 4$
- $o(A2) = 6$

For a schema to survive all fixed bits must remain untouched. The effect of a mutation is given by the equation  $(1-pm)$  . For our example, it would be  $(1-pm)=1-0.01 = 99.99$  .

- $Sm(A1) = (1-pm)^{o(A1)}$
- $Sm(A2) = (1-pm)^{o(A2)}$

From the lectures, we were taught that a schema with a lower order has a higher chance of survival. Thus, A1 has a higher chance to survive, as it has a lower order compared to A2.

### Exercise 2 - (Building Block Hypothesis)

BBH does not hold when there is no information available which could guide GA to global optimum through composition of partial sub-optimal solutions.

The problem where the Building Block Hypothesis does not hold is the Kronecker delta function:

$$f(x) = Kronecker\_delta(x),$$

meaning  $f(x)=1$  when  $x=0$  and  $f(x)=0$  when  $x \neq 0$ . The global optimum is achieved at  $x=0$  and for every other  $x$  the value is 0. There is no way to guide the search towards the global optimum since none of the  $x$ 's (where  $x \neq 0$ ) provide information on where the global optima is.

### Exercise 3 - (Selection Pressure)

```
In [2]: ► 1 import matplotlib.pyplot as plt
2 def fitness_fx(x):
3     return x*x
4
5 def fitness_f1x(x):
6     return 20+fitness_fx(x)
7
```

```
In [3]: ► 1 #Calculate fitness for x=2, x=3, x=4 and sum them up
2 x1 = fitness_fx(2)
3 print('For x1 = 2 fitness function fx(x1) = ',x1)
4 x2 = fitness_fx(3)
5 print('For x2 = 2 fitness function fx(x2) = ',x2)
6 x3 = fitness_fx(4)
7 print('For x3 = 3 fitness function fx(x3) = ',x3)
8 sum1 = x1 + x2 + x3
9 print('Total sum of all individuals = ',sum1)
```

For x1 = 2 fitness function fx(x1) = 4  
For x2 = 2 fitness function fx(x2) = 9  
For x3 = 3 fitness function fx(x3) = 16  
Total sum of all individuals = 29

In [4]:

```

1 #Calculate probability of selection for fx(x)
2 print('Probability of selection for x1 -> x1/sum = ',round(x1/sum1,2))
3 print('Probability of selection for x2 -> x2/sum = ',round(x2/sum1,2))
4 print('Probability of selection for x3 -> x3/sum = ',round(x3/sum1,2))
5
6 labels = 'x1', 'x2', 'x3'
7 sizes = [round(x1/sum1,2), round(x2/sum1,2), round(x3/sum1,2)]
8
9
10 fig1, ax1 = plt.subplots()
11 ax1.pie(sizes, labels=labels, autopct='%1.1f%%', shadow=True, startangle=90)
12 ax1.axis('equal')
13 txt="Fig.1: Fitness fitneess function"
14 plt.figtext(0.5, 0.01, txt, wrap=True, horizontalalignment='center', fontweight='bold')
15 plt.show()

```

Probability of selection for x1 -> x1/sum = 0.14  
 Probability of selection for x2 -> x2/sum = 0.31  
 Probability of selection for x3 -> x3/sum = 0.55

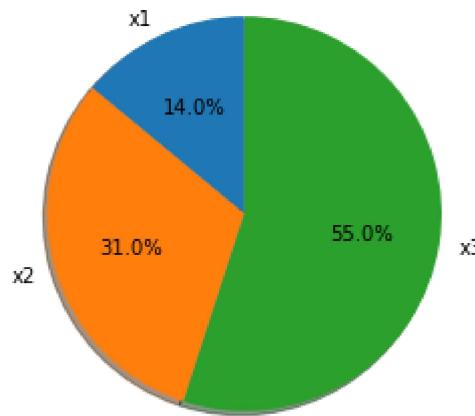


Fig.1: Fitness fitneess function

In [5]:

```

1 #Calculate fitness for x=2, x=3, x=4 for new fitness function f1(x) and sum
2 x1 = fitness_f1x(2)
3 print('For x1 = 2 fitness function fx(x1) = ',x1)
4 x2 = fitness_f1x(3)
5 print('For x2 = 2 fitness function fx(x2) = ',x2)
6 x3 = fitness_f1x(4)
7 print('For x3 = 3 fitness function fx(x3) = ',x3)
8 sum1 = x1 + x2 + x3
9 print('Total sum of all individuals = ',sum1)

```

For x1 = 2 fitness function fx(x1) = 24  
 For x2 = 2 fitness function fx(x2) = 29  
 For x3 = 3 fitness function fx(x3) = 36  
 Total sum of all individuals = 89

In [6]:

```

1 #Calculate probability of selection for f1x(x)
2 print('Probability of selection for x1 -> x1/sum = ',round(x1/sum1,2))
3 print('Probability of selection for x2 -> x2/sum = ',round(x2/sum1,2))
4 print('Probability of selection for x3 -> x3/sum = ',round(x3/sum1,2))
5
6 labels = 'x1', 'x2', 'x3'
7 sizes = [round(x1/sum1,2), round(x2/sum1,2), round(x3/sum1,2)]
8
9
10 fig1, ax1 = plt.subplots()
11 ax1.pie(sizes, labels=labels, autopct='%1.1f%%', shadow=True, startangle=90)
12 ax1.axis('equal')
13 txt="Fig.2: Fitness for new fitness function"
14 plt.figtext(0.5, 0.01, txt, wrap=True, horizontalalignment='center', fontweight='bold')
15 plt.show()

```

Probability of selection for x1 -> x1/sum = 0.27  
 Probability of selection for x2 -> x2/sum = 0.33  
 Probability of selection for x3 -> x3/sum = 0.4

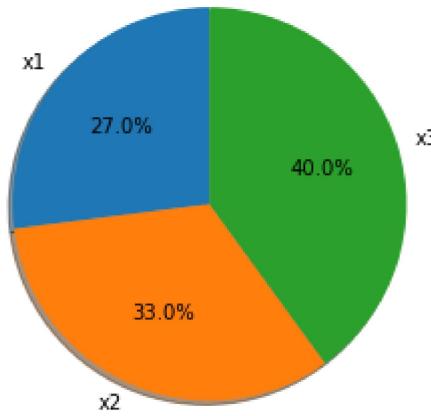


Fig.2: Fitness for new fitness function

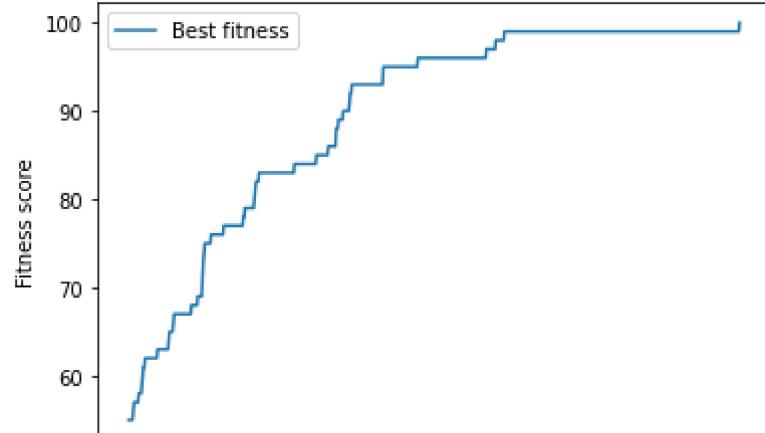
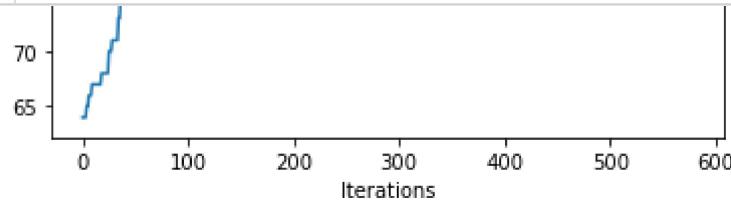
## Conclusion

The first fitness function  $f_1(x)$  has a higher selection pressure. In the first figure plotted for the function  $f_1(x)$ , we can see a dominant probability of 55% and then two lower probabilities of 31% and 14%. The two lower probabilities of the individuals suggest a smaller probability of being chosen for survival or as parents and thus a higher survival for individual  $x=3$ . When the new fitness function  $f_2$  is introduced, the new values of the individuals are more fairly distributed. Thus, selection pressure decreases. Compared to the previous figure now, the two lower individuals have seen an increase in their survival probabilities. This suggests a more equal chance for the individuals. Overall, fitter individuals are more likely to be chosen for survival or as parents. </p>

## Exercise 4 - (Role of selection in GA's)

```
In [7]: 1 def counting_ones(num_iter, num_elems=100, swap=False, plot_ind=1):
2     p = 1 / num_elems
3     x = [random.randint(0, 1) for x in range(num_elems)]
4     fitness = []
5     for i in range(num_iter):
6         x_m = [1-el if random.uniform(0,1) < p else el for el in x]
7         if sum(x_m) > sum(x) and not swap:
8             x = x_m
9         if swap:
10            x = x_m
11         fitness.append(sum(x))
12         if sum(x) == len(x):
13             break
14
15     plt.figure(plot_ind)
16     plt.xlabel("Iterations")
17     plt.ylabel("Fitness score")
18     plt.plot(fitness)
19     plt.legend(['Best fitness'])
```

```
In [8]: 1 counting_ones(1500, 100)
2 for i in range(10):
3     counting_ones(1500, 100, plot_ind=i+2)
```

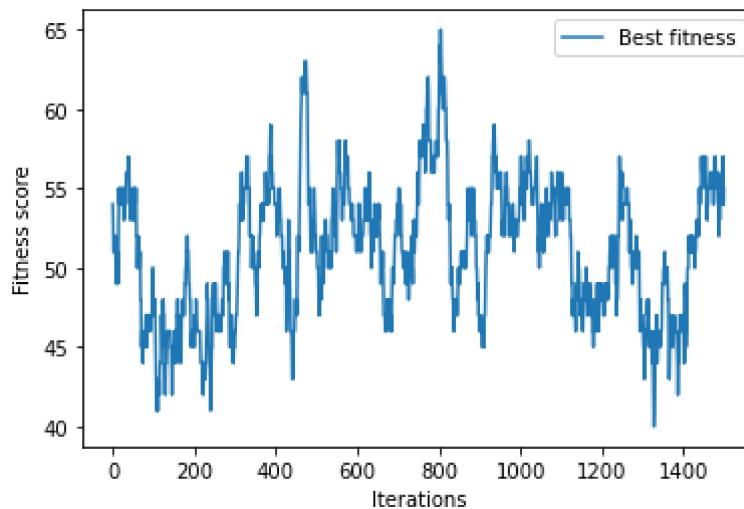


In [9]:

```

1 counting_ones(1500, 100, swap=True, plot_ind=1)
2 for i in range(10):
3     counting_ones(1500, 100, swap=True, plot_ind=i+2)

```



For the plots outputted from the function which checks (for the better fit) before it swaps, we can see that after 1 500 iterations it reaches (at some point) the wanted score. With 10 runs, it converges to the goal 10 times. By selecting only the most fit one, the fit can only get better and there is no problem of getting stuck in local optimum. Hence, there is a good balance between exploration and exploitation. For the plots outputted from the function which always swaps, compared to the previous plots, we can see a difference. By removing the condition to select the string with most ones (the string with the better fit), the algorithm is not converging towards no optimum. Exploration overdominates exploitation and the algorithm is, basically, performing a random walk. By running it 10 times, the stochasticity of the algorithm is even more visible.

### Exercise 5

(1+5)ES differs from (1+1)ES as it generates only 1 offspring after each generation. Since both of them look at  $\rho = 1$  best, the population size of both of them is 1. However, (1+5)ES searches a more space of the solutions, therefore it is less likely to get stuck in the local optimum as well as more likely to find the solution faster.

(1+ $\lambda$ )ES resembles greedy algorithms less as the  $\lambda$  gets larger. With the  $\lambda = 1$ , (1+ $\lambda$ )ES becomes a greedy algorithm.

### Exercise 6 - (Memetic algorithms vs simple EAs)

```
In [10]: # LOAD DATA
1 data = []
2 with open("file-tsp.txt") as fp:
3     for line in fp.readlines():
4         x, y = line.split()
5         data.append([float(x), float(y)])
6
7
8 data = np.array(data)
```

```
In [11]: # LOAD DATA for second dataset
1 data2 = []
2 with open("bavaria.tsp") as fp:
3     reached_data = False
4     for line in fp.readlines():
5         if 'DISPLAY_DATA_SECTION' in line:
6             reached_data = True
7             continue
8         if not reached_data or 'EOF' in line:
9             continue
10        x, y = line.split()[1:]
11        data2.append([float(x), float(y)])
12
13
14 data2 = np.array(data2)
```

In [12]:

```

1  def order_crossover(parent1, parent2, p_c=1):
2      # There is a probability that the crossover will not happen (it is 1-p_c)
3      if random.uniform(0, 1) > p_c:
4          return parent1, parent2
5
6      p1 = parent1[0]
7      p2 = parent2[0]
8
9      # Initialize offsprings
10     off1 = []
11     off2 = []
12
13     # Create 2 cut points and store them in a list
14     while True:
15         indexes = []
16         indexes.append(random.randint(0, len(p1)-1))
17         indexes.append(random.randint(0, len(p2)-1))
18         if indexes[0] != indexes[1]:
19             break
20     indexes.sort()
21
22     # Copy between parents' cutpoints to children
23     off1 = p1[indexes[0] : indexes[1]]
24     off2 = p2[indexes[0] : indexes[1]]
25
26     # Rotate parents starting from 2nd cut point
27     p1 = p1[-(len(p1)-indexes[1]):] + p1[:indexes[1]]
28     p2 = p2[-(len(p2)-indexes[1]):] + p2[:indexes[1]]
29
30     # Fill missing cities in 1st offspring
31     i = 0
32     help_list = []
33     for el in p2:
34         if el not in off1:
35             i += 1
36             if i <= len(p2) - indexes[1]:
37                 off1.append(el)
38             else:
39                 help_list.append(el)
40     off1 = help_list + off1
41
42     # Fill missing cities in 2nd offspring
43     i = 0
44     help_list = []
45     for el in p1:
46         if el not in off2:
47             i += 1
48             if i <= len(p1) - indexes[1]:
49                 off2.append(el)
50             else:
51                 help_list.append(el)
52     off2 = help_list + off2
53
54     return off1, off2
55
56 order_crossover(['3', '5', '7', '2', '1', '6', '4', '8'], 222, 2), ([ '2',
```

Out[12]: ([['2', '5', '7', '8', '1', '6', '3', '4'], ['3', '5', '7', '2', '6', '1', '4', '8']])

In [13]:

```

1 def mutate(offspring, p_m=0.001):
2     # There is a probability that the mutation will not happen (it is 1-p_m)
3     if random.uniform(0, 1) > p_m:
4         return offspring
5
6     # Generate 2 indexes for mutation
7     while True:
8         ind1 = random.randint(0, len(offspring)-1)
9         ind2 = random.randint(0, len(offspring)-1)
10        if ind1 != ind2:
11            break
12
13
14    # Swap indexes of offspring
15    h = offspring[ind1]
16    offspring[ind1] = offspring[ind2]
17    offspring[ind2] = h
18
19    return offspring
20
21 mutate(['5', '8', '7', '2', '1', '6', '3', '4'])

```

Out[13]: ['5', '8', '7', '2', '1', '6', '3', '4']

In [14]:

```

1 def init_population(pop_size, chromosome_len):
2     pop = []
3     # Initialize population as multiple random permutations of ints between 0 and chromosome_len
4     for i in range(pop_size):
5         chromosome = list(range(chromosome_len))
6         random.shuffle(chromosome)
7         pop.append(chromosome)
8     return pop

```

In [15]:

```

1 def fitness(data, order=list(range(len(data)))):
2     # Calculate the overall distance as the Euclidian distance between each point in order
3     distance = 0
4     for i in range(1, len(order)):
5         ind = order[i]
6         ind_prev = order[i-1]
7         distance += ((data[ind][0]-data[ind_prev][0])**2 + (data[ind][1]-data[ind_prev][1])**2)**0.5
8     return distance
9
10 def evaluate(pop, data):
11     # Evaluate the population
12     if type(pop[0]) == list:
13         new_pop = []
14         for chromosome in pop:
15             new_pop.append((chromosome, fitness(data, chromosome)))
16     return new_pop
17 else:
18     return (pop, fitness(data, pop))

```

```
In [16]: ► 1 def binary_tournament_selection(pop, data):
2     # Set K to 2 (because it is BINARY tournament selection)
3     K = 2
4
5     # Tournament selection
6     while True:
7         ind1 = random.randint(0, len(pop)-1)
8         ind2 = random.randint(0, len(pop)-1)
9         if ind1 != ind2:
10             break
11
12     # Store those 2 parents
13     p1 = pop[ind1]
14     p2 = pop[ind2]
15
16     # Return parents
17     return p1, p2
18
19
20     off1, off2 = order_crossover(pop[ind1], pop[ind2])
21     if pop[ind1] > pop[ind2]:
22         return pop[ind1]
```

```
In [17]: ► 1 def next_gen(pop, parents, offsprings):
2     # Unpack values
3     p1, p2 = parents
4     off1, off2 = offsprings
5
6     # Calculate which offspring has better fitness function and which parent
7     best_offspring = off1 if off1[1] > off2[1] else off2
8     worst_parent = p2 if p1[1] > p2[1] else p1
9
10    ind = pop.index(worst_parent)
11    pop[ind] = best_offspring
12
13    return pop
```

In [18]:

```

1 def plot_TSP(data, tour):
2     # Initialize figure with random ID
3     plt.figure(random.randint(1,100))
4
5     # Draw cities
6     for x,y in data:
7         plt.scatter(x, y, color='black')
8
9     # Draw tour
10    for i in range(1, len(tour)):
11        city_now = tour[i]
12        city_prev = tour[i-1]
13
14        x2, y2 = data[city_now]
15        x1, y1 = data[city_prev]
16
17        # Plot the data
18        plt.plot((x1, x2), (y1, y2), color='black')
19        plt.title("Best solution found after 1000 iterations")
20        plt.xlabel("x-axis")
21        plt.ylabel("y-axis")
22    plt.show()

```

In [19]:

```

1 def local_search(pop, data, n=1):
2     new_pop = []
3     for el in pop:
4         # For every element in population, find n neighbors
5         neighbors = []
6         for _ in range(n):
7             neighbor = mutate(el[0])
8             neighbors.append((neighbor, fitness(data, neighbor)))
9
10        # Find neighbor with the highest fitness
11        neighbors.sort(key = lambda x: x[1])
12        best_neighbor = neighbors[-1]
13
14        # Save neighbor with the highest fitness if its score is higher
15        if best_neighbor[1] > el[1]:
16            new_pop.append(best_neighbor)
17        else:
18            new_pop.append(el)
19
20    return new_pop

```

```
In [20]: ┌ 1 def genetic_algorithm(data, pop_size = 5, iter_num=10, loc_search=False)
  2     pop = init_population(pop_size, len(data))
  3     pop = evaluate(pop, data)
  4     average_fits = []
  5
  6     # Perform Local search
  7     if loc_search:
  8         pop = local_search(pop, data, n=10)
  9
 10    for i in range(iter_num):
 11        # Select parents for reproduction
 12        p1, p2 = binary_tournament_selection(pop, data)
 13
 14        # Perform crossover of parents
 15        off1, off2 = order_crossover(p1, p2)
 16
 17        # Perform mutation on offsprings
 18        off1 = mutate(off1)
 19        off2 = mutate(off2)
 20
 21        # Evaluate offsprings
 22        off1 = evaluate(off1, data)
 23        off2 = evaluate(off2, data)
 24
 25        # Perform Local search
 26        if loc_search:
 27            off1, off2 = local_search([off1, off2], data, n=10)
 28
 29        # Generate next population
 30        pop = next_gen(pop, (p1, p2), (off1, off2))
 31
 32        # Save average fit
 33        average_fits.append(sum([el[1] for el in pop])/len(pop))
 34
 35        print(pop)
 36        pop.sort(key = lambda x: x[1])
 37        print("Average fit: ", sum([el[1] for el in pop])/len(pop))
 38        print("Printed instance: ", pop[-1][1])
 39        plot_TSP(data, pop[-1][0])
 40
 41    return average_fits
```

In [21]:

```

1 iter_num = 1000
2 average_fits = genetic_algorithm(data, iter_num = iter_num)

```

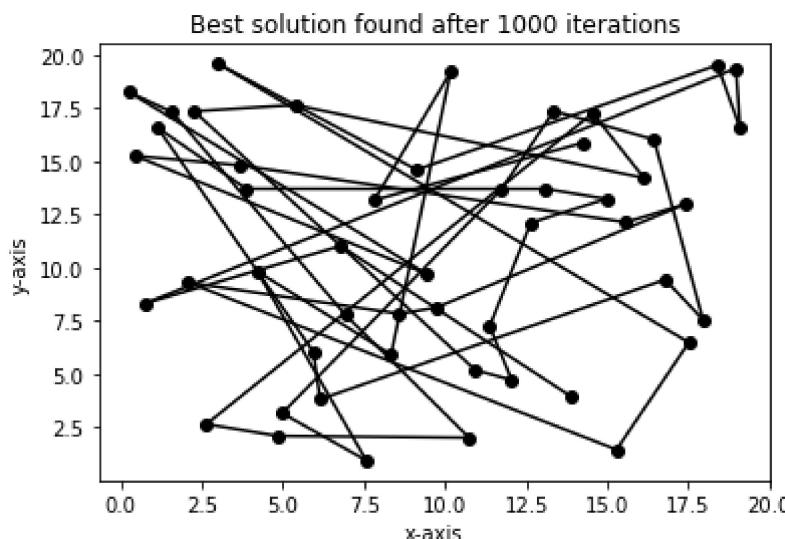
```

[([36, 20, 26, 21, 11, 19, 13, 30, 34, 42, 46, 43, 16, 15, 3, 10, 33, 38, 3
2, 29, 31, 28, 6, 14, 41, 37, 7, 12, 27, 18, 4, 0, 24, 1, 9, 40, 44, 25, 2
2, 5, 39, 45, 8, 23, 47, 49, 48, 2, 17, 35], -367.7623485405646), ([36, 20,
26, 21, 11, 19, 13, 30, 34, 42, 46, 43, 16, 15, 3, 10, 33, 38, 32, 29, 31,
28, 6, 14, 41, 37, 7, 12, 27, 18, 4, 0, 24, 1, 9, 40, 44, 25, 22, 5, 39, 4
5, 8, 23, 47, 49, 48, 2, 17, 35], -367.7623485405646), ([36, 20, 26, 21, 1
1, 19, 13, 30, 34, 42, 46, 43, 16, 15, 3, 10, 33, 38, 32, 29, 31, 28, 6, 1
4, 41, 37, 7, 12, 27, 18, 4, 0, 24, 1, 9, 40, 44, 25, 22, 5, 39, 45, 8, 23,
47, 49, 48, 2, 17, 35], -367.7623485405646), ([36, 20, 26, 21, 11, 19, 13,
30, 34, 42, 46, 43, 16, 15, 3, 10, 33, 38, 32, 29, 31, 28, 6, 14, 41, 37,
7, 12, 27, 18, 4, 0, 24, 1, 9, 40, 44, 25, 22, 5, 39, 45, 8, 23, 47, 49, 4
8, 2, 17, 35], -367.7623485405646), ([36, 20, 26, 21, 11, 19, 13, 30, 34, 4
2, 46, 43, 16, 15, 3, 10, 33, 38, 32, 29, 31, 28, 6, 14, 41, 37, 7, 12, 27,
18, 4, 0, 24, 1, 9, 40, 44, 25, 22, 5, 39, 45, 8, 23, 47, 49, 48, 2, 17, 3
5], -367.7623485405646)]

```

Average fit: -367.7623485405646

Printed instance: -367.7623485405646

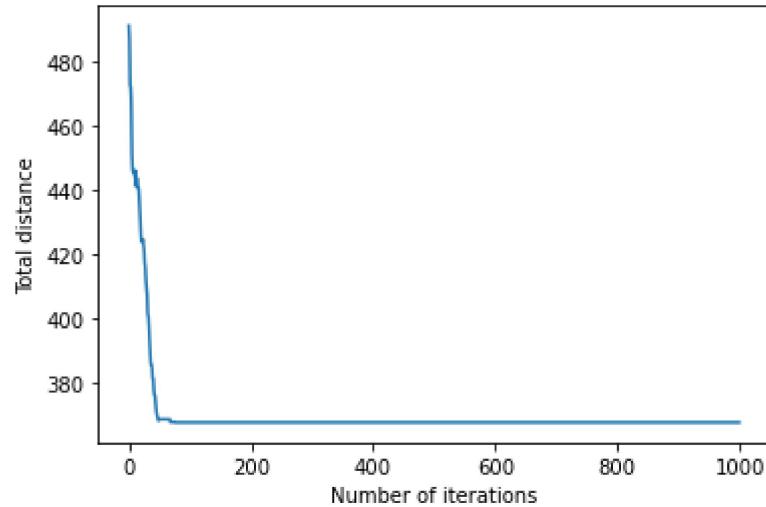


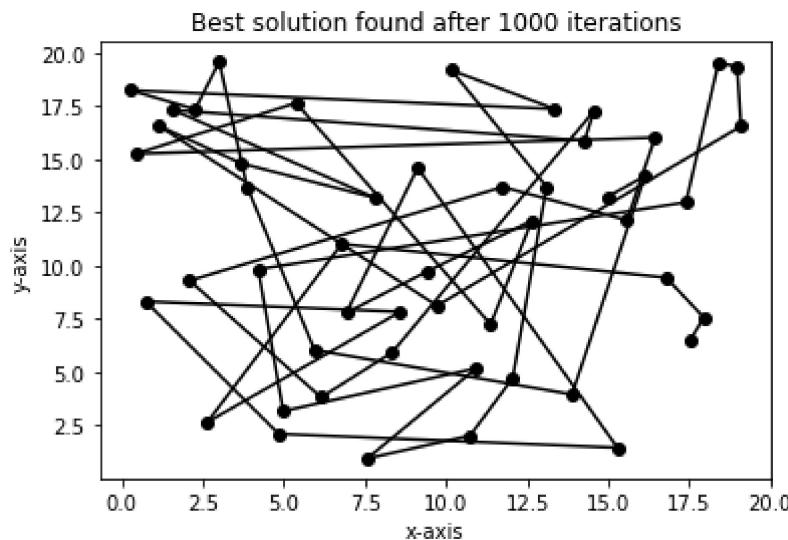
In [22]:

```
1 # Print average fits
2 plt.figure()
3 plt.plot(list(range(iter_num)), np.full(len(average_fits), -1)*average_fi
4 plt.title("Total distance over the number of iterations for TSP with no ")
5 plt.xlabel("Number of iterations")
6 plt.ylabel("Total distance")
```

Out[22]: Text(0, 0.5, 'Total distance')

Total distance over the number of iterations for TSP with no local search on the first dataset



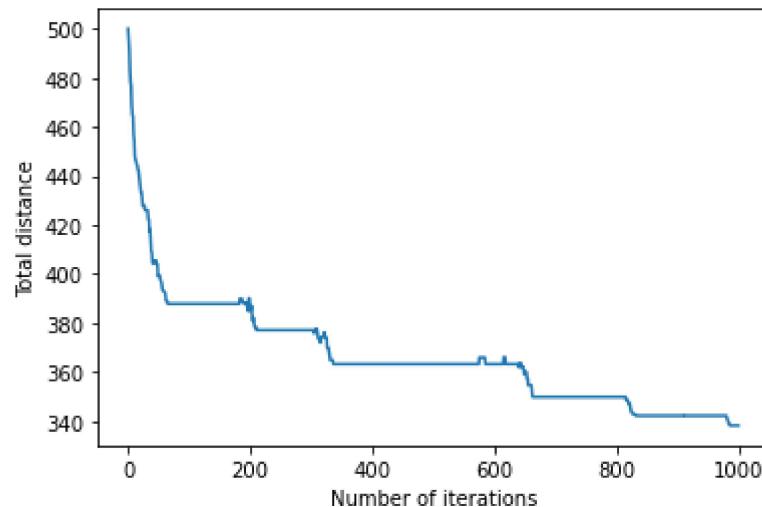


In [24]:

```
1 # Print average fits
2 plt.figure()
3 plt.plot(list(range(iter_num)), np.full(len(average_fits), -1)*average_fi
4 plt.title("Total distance over the number of iterations for TSP with loca
5 plt.xlabel("Number of iterations")
6 plt.ylabel("Total distance")
```

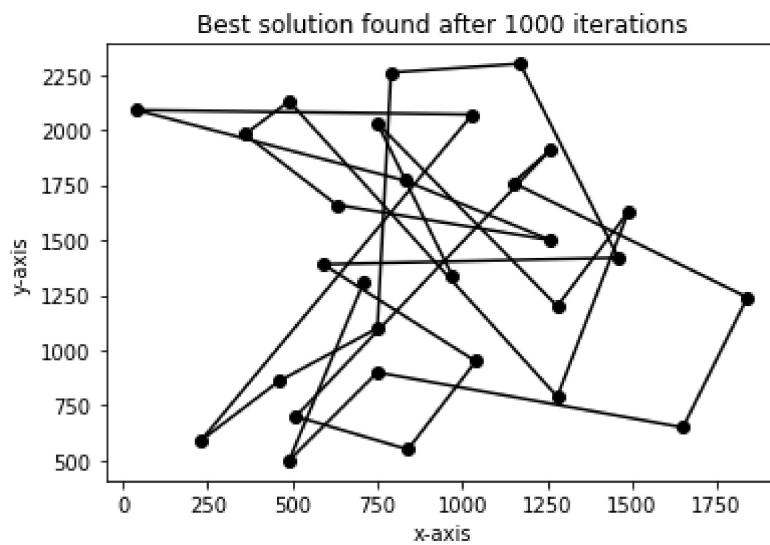
Out[24]: Text(0, 0.5, 'Total distance')

Total distance over the number of iterations for TSP with local search on the first dataset



```
In [25]: 1 iter_num = 1000
          2 average_fits = genetic_algorithm(data2, iter_num = iter_num)
```

```
[[([9, 21, 14, 6, 22, 0, 27, 13, 10, 18, 19, 26, 11, 8, 3, 17, 16, 5, 2, 20, 23, 1, 28, 25, 24, 7, 15, 4, 12], -20762.769200214894), ([9, 21, 14, 6, 22, 0, 27, 13, 10, 18, 19, 26, 11, 8, 3, 17, 16, 5, 2, 20, 23, 1, 28, 25, 24, 7, 15, 4, 12], -20762.769200214894), ([9, 21, 14, 6, 22, 0, 27, 13, 10, 18, 19, 26, 11, 8, 3, 17, 16, 5, 2, 20, 23, 1, 28, 25, 24, 7, 16, 5, 2, 20, 23, 1, 28, 25, 24, 7, 15, 4, 12], -20762.769200214894), ([9, 21, 14, 6, 22, 0, 27, 13, 10, 18, 19, 26, 11, 8, 3, 17, 16, 5, 2, 20, 23, 1, 28, 25, 24, 7, 15, 4, 12], -20762.769200214894), ([9, 21, 14, 6, 22, 0, 27, 13, 10, 18, 19, 26, 11, 8, 3, 17, 16, 5, 2, 20, 23, 1, 28, 25, 24, 7, 15, 4, 12], -20762.769200214894)]]
Average fit: -20762.769200214894
Printed instance: -20762.769200214894
```

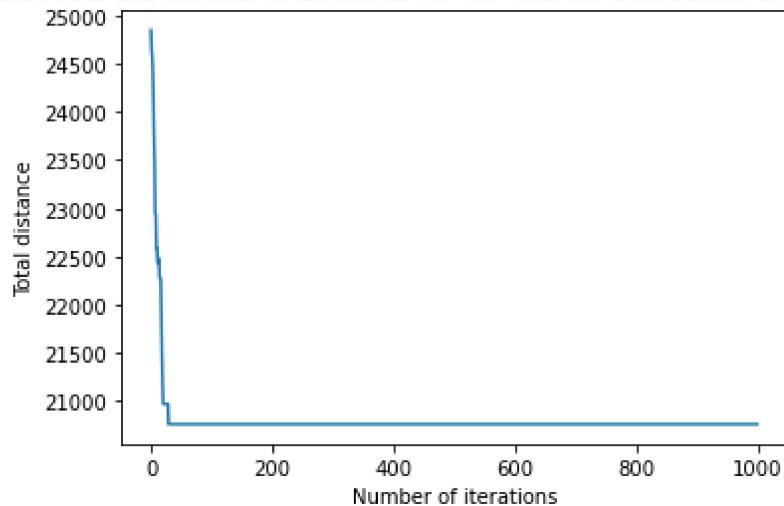


In [26]:

```
1 # Print average fits
2 plt.figure()
3 plt.plot(list(range(iter_num)), np.full(len(average_fits), -1)*average_fi
4 plt.title("Total distance over the number of iterations for TSP with no ")
5 plt.xlabel("Number of iterations")
6 plt.ylabel("Total distance")
```

Out[26]: Text(0, 0.5, 'Total distance')

Total distance over the number of iterations for TSP with no local search on the second dataset



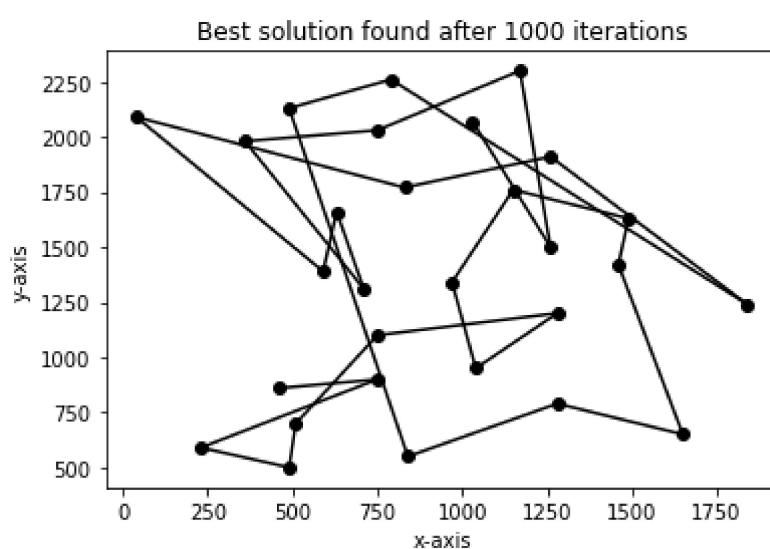
In [27]: 1 iter\_num = 1000

```
2 average fits = genetic algorithm(data2, iter num = iter num, loc search=
```

```
[([17, 14, 16, 21, 13, 3, 15, 18, 12, 0, 7, 26, 6, 24, 10, 25, 8, 22, 27, 2
0, 2, 19, 1, 9, 28, 4, 11, 23, 5], -16036.025318400882), ([17, 14, 16, 21,
13, 3, 15, 18, 12, 0, 7, 26, 6, 24, 10, 25, 8, 22, 27, 20, 2, 19, 1, 9, 28,
4, 11, 23, 5], -16036.025318400882), ([17, 14, 16, 21, 13, 3, 15, 18, 12,
0, 7, 26, 6, 24, 10, 25, 8, 22, 27, 20, 2, 19, 1, 9, 28, 4, 11, 23, 5], -16
036.025318400882), ([17, 14, 16, 21, 13, 3, 15, 18, 12, 0, 7, 26, 6, 24, 1
0, 25, 8, 22, 27, 20, 2, 19, 1, 9, 28, 4, 11, 23, 5], -16036.025318400882),
([17, 14, 16, 21, 13, 3, 15, 18, 12, 0, 7, 26, 6, 24, 10, 25, 8, 22, 27, 2
0, 2, 19, 1, 9, 28, 4, 11, 23, 5], -16036.025318400882)]
```

Average fit: -16036.025318100882

Average fit: -16036.025318400882



In [28]:

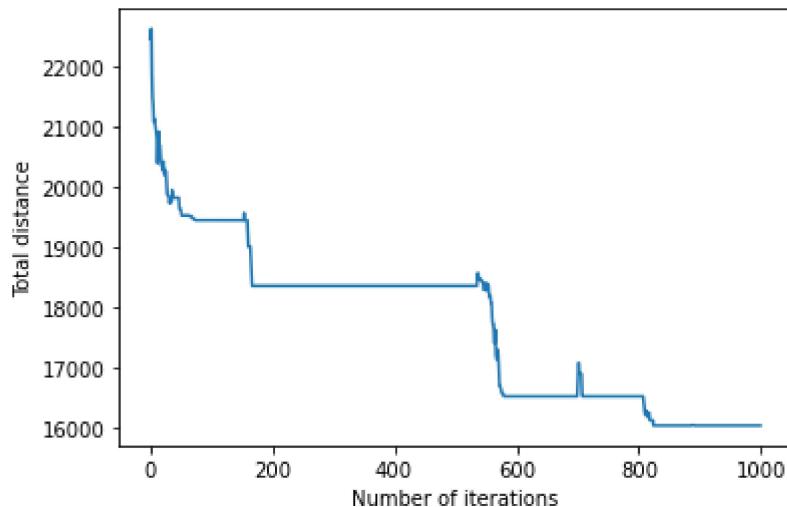
```

1 # Print average fits
2 plt.figure()
3 plt.plot(list(range(iter_num)), np.full(len(average_fits), -1)*average_fi
4 plt.title("Total distance over the number of iterations for TSP with loca
5 plt.xlabel("Number of iterations")
6 plt.ylabel("Total distance")

```

Out[28]: Text(0, 0.5, 'Total distance')

Total distance over the number of iterations for TSP with local search on the second dataset



## CONCLUSION

The use of the memetic algorithms is more effective. That is possible to see from both of these experiments conducted. The fit (distance) gets lower when using local search.

As (Merz, Peter, and Bernd Freisleben. "Memetic algorithms for the traveling salesman problem." complex Systems 13.4 (2001): 297-346.) have shown that as well.

## Exercise 7 - (Genetic Programming representation)

a) The function set and terminal set are listed below:

- function set:  $\wedge$ ,  $\rightarrow$ ,  $\vee$ ,  $\leftrightarrow$
- terminal set:  $y$ , true,  $x$ ,  $z$
- s-expression set:  $\rightarrow(\wedge y \text{ true})(\vee(\vee x y)(\leftrightarrow z(\wedge x y)))$

b) The function set and terminal set are listed below:

- function set:  $*$ ,  $+$ ,  $-$
- terminal set: 0.234,  $x$ ,  $z$ , 0.789
- s-expression set:  $+(*0.234 z)(-x 0.789)$

## Exercise 8 - (Genetic Programming behaviour)

In [29]:

```
1 random.seed(2021)
2 #initialize the sets
3 binary_set = ['+', '-', '*', 'div']
4 unary_set = ['log', 'exp', 'sin', 'cos']
5 function_set = binary_set + unary_set
6
7 terminal_set = ['x']
8
9 full_set = function_set + terminal_set
```

In [30]:

```

1  class Tree(object):
2      def __init__(self, name):
3          assert name in full_set
4          self.name = name
5          self.children = []
6          if name in terminal_set: self.type = 'term'
7          elif name in unary_set: self.type = 'unary'
8          else: self.type = 'binary'
9
10     def __repr__(self):
11         return '<tree>'
12
13     def __str__(self, l = 0):
14         ret = "    " * l + str(l) + ": " + self.name + "\n"
15         for child in self.children:
16             ret += child.__str__(l + 1)
17         return ret
18
19     def add_children(self, node):
20         assert isinstance(node, Tree)
21         self.children.append(node)
22
23     #get the value of the equation made by this tree for x
24     def get_value(self, x):
25         if self.name in terminal_set: return x
26         elif self.name in unary_set:
27             child_value = self.children[0].get_value(x)
28             if (child_value == 'error'): return 'error'
29             if self.name == 'log':
30                 if child_value > 0: return math.log(child_value)
31                 else: return 'error'
32             elif self.name == 'exp':
33                 if child_value < 700: return math.exp(child_value)
34                 else: return 'error'
35             elif self.name == 'sin': return math.sin(child_value)
36             else: return math.cos(child_value)
37         else:
38             left_child_value = self.children[0].get_value(x)
39             right_child_value = self.children[1].get_value(x)
40             if (left_child_value == 'error' or right_child_value == 'error'):
41                 else:
42                     if self.name == '+': return left_child_value + right_child_value
43                     elif self.name == '-': return left_child_value - right_child_value
44                     elif self.name == '*': return left_child_value * right_child_value
45                     else:
46                         if right_child_value != 0: return left_child_value / right_child_value
47                         else: return 'error'
48
49     def get_fitness(self, X_Y):
50         abs_errors = ['error' if self.get_value(x) == 'error' else abs(self.get_value(x) - X_Y) for x in X_Y]
51         return 'error' if 'error' in abs_errors else sum(abs_errors)

```

```
In [31]: 1  def add_random_children(tree, current_depth, max_depth, grow_method = True):
2      if tree.type == 'unary':
3          if current_depth + 1 == max_depth:
4              child = Tree(random.choice(terminal_set))
5              tree.add_children(child)
6      else:
7          child = Tree(random.choice(full_set)) if grow_method else Tree()
8          add_random_children(child, current_depth + 1, max_depth)
9          tree.add_children(child)
10
11     if tree.type == 'binary':
12         if current_depth + 1 == max_depth:
13             left_child = Tree(random.choice(terminal_set))
14             right_child = Tree(random.choice(terminal_set))
15             tree.add_children(left_child)
16             tree.add_children(right_child)
17         else:
18             left_child = Tree(random.choice(full_set)) if grow_method else Tree()
19             add_random_children(left_child, current_depth + 1, max_depth)
20             right_child = Tree(random.choice(full_set)) if grow_method else Tree()
21             add_random_children(right_child, current_depth + 1, max_depth)
22             tree.add_children(left_child)
23             tree.add_children(right_child)
24
25 def create_random_tree(max_depth = 3, grow_method = True):
26     tree = Tree(random.choice(full_set))
27     add_random_children(tree, 1, max_depth, grow_method)
28     return tree
```

In [32]:

```

1 def size_of(tree):
2     if tree.type == 'term': return 1
3     if tree.type == 'unary': return size_of(tree.children[0]) + 1
4     if tree.type == 'binary': return size_of(tree.children[0]) + 1 + size_of(tree.children[1])
5
6 def get_random_node(tree):
7     if tree.type == 'term': return tree
8
9     left_part_size = size_of(tree.children[0])
10
11    rand_num = random.randint(1, size_of(tree))
12
13    if rand_num <= left_part_size: return get_random_node(tree.children[0])
14    elif rand_num == left_part_size + 1: return tree
15    else: return get_random_node(tree.children[1])
16
17 def swap_random_nodes(first_tree, second_tree):
18     first_node = get_random_node(first_tree)
19     second_node = get_random_node(second_tree)
20
21     old_names = (first_node.name, second_node.name)
22     old_children = (first_node.children, second_node.children)
23     old_types = (first_node.type, second_node.type)
24
25     first_node.name = old_names[1]
26     first_node.children = old_children[1]
27     first_node.type = old_types[1]
28
29     second_node.name = old_names[0]
30     second_node.children = old_children[0]
31     second_node.type = old_types[0]

```

In [33]:

```

1 def create_offsprings(parent1, parent2, crossover_probability = 0.7):
2     if random.uniform(0, 1) > crossover_probability:
3         return [parent1, parent2]
4     else:
5         swap_random_nodes(parent1, parent2)
6         return [parent1, parent2]

```

In [34]:

```

1 def takeSecond(elem):
2     return elem[1]

```

In [35]:

```

1 X_Y = [[-1.0, 0.0000], [-0.9, -0.1629], [-0.8, -0.2624], [-0.7, -0.3129]]

```

In [36]:

```

1 population_size = 1000
2 generations = range(0, 50)
3 max_depth = 10
4
5 #ramped half-and-half initialisation
6 new_population = [create_random_tree(max_depth, True) for x in range(0, population_size/2)]
7 new_population += [create_random_tree(max_depth, False) for x in range(0, population_size/2)]

```

```
In [37]: 1 import copy
2 best_from_populations = []
3
4 for generation in generations:
5     population_and_errors = [(inst, inst.get_fitness(X_Y)) for inst in new
6     population_and_errors.sort(key=takeSecond)
7     population, errors = zip(*population_and_errors)
8
9     best_from_populations.append((size_of(population[0]), errors[0]))
10
11    sum_of_errors = sum(errors)
12    weights = [error/sum_of_errors for error in errors]
13
14    new_population = []
15    #keep top 10
16    new_population += population[:10]
17
18    #this cycle may take some time and memory due to copying objects
19    for j in range(0, int(population_size / 2) - 5):
20        parent_1 = copy.deepcopy(random.choices(population=population, wei
21        parent_2 = copy.deepcopy(random.choices(population=population, wei
22        new_population += create_offsprings(parent_1, parent_2)
```

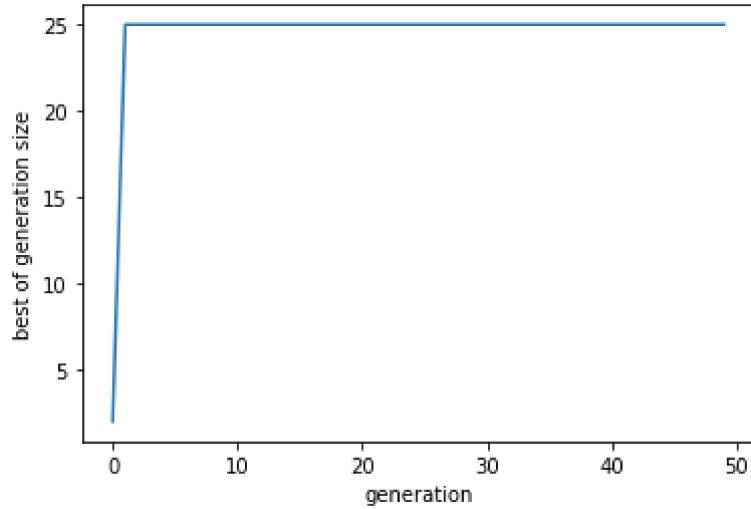
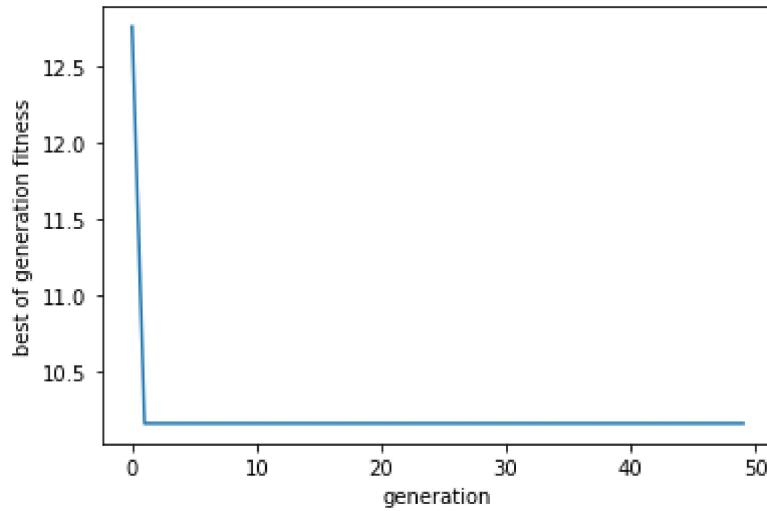
```
In [38]: 1 sizes, errors = zip(*best_from_populations)
```

In [39]:

```

1 import matplotlib.pyplot as plt
2
3 plt.plot(generations, errors)
4 plt.xlabel('generation')
5 plt.ylabel('best of generation fitness')
6 plt.show()
7
8 plt.plot(generations, sizes)
9 plt.xlabel('generation')
10 plt.ylabel('best of generation size')
11 plt.show()

```



## Conclusion

As can be seen from the graphs, in this case, there is a great selection pressure, due to which our implementation of the algorithm finds a more or less optimal value in the second generation (by increasing the tree size), and due to the lack of diversity, this value remains dominant. This problem can be solved by replacing the parent selection mechanism: from fitness proportional to a tournament one. It is also possible to improve the algorithm by adjusting the survivor selection and other hyperparameters (e.g. population size, number of generations or additional terminals).

