CMPSC 441
November 9, 2020

Homework 4

Due: November 22, 2020

In this assignment, we will implement a genetic algorithm discussed in the lecture and apply them to solve three classic problems. Two start-up files are provided on Canvas:

- `hw4_utils.py`: containing the class `GeneticProblem` that abstracts the problem-specific operations for the genetic algorithm. It also contains other utilities such as the class `Graph` and test data.
- `hw4.py`: containing the prototype of the genetic algorithm and the empty definitions of the classes for the three classic problems.

You must:

- Download the two start-up files and place them in the same directory.
- Rename `hw4.py` by replacing `hw4` with your PSU access ID. For example, if your PSU email address is suk1234@psu.edu, then you must rename it to `suk1234.py`.
- Not change the file name of `hw4_utils.py` nor modify the contents.
- Upload your complete `suk1234.py` to the correct assignment area on Canvas by 11:59pm on the due date. If you upload your file to the wrong assignment area or if you fail to submit it by 11:59pm on the due date, it will not be graded and you will receive an automatic Zero on the assignment.

Note that you are not allowed to change anything in `hw4_utils.py`. Further, you are not to submit `hw4_utils.py` when you submit your complete homework. Even if you submit it, we will replace it with the original version initially distributed. This means that, if your program depends on the changes you make in `hw4_utils.py`, it will very likely fail when it is graded.

In the description below, many examples of use cases are provided for each function or method. These examples are simply the typical use cases to clarify the specification and is not meant to be a comprehensive test cases. You are strongly encouraged to test your code with these examples and to test further with your own test cases before you submit. Note that your results might not be the same as shown in this handout, due to the use of random numbers. However, they should still be the correct answers to the problems.

1 Genetic Algorithm.

Implement `genetic_algorithm`. The pseudocode for the algorithm is given below:

```
def genetic_algorithm(problem, f_thres, ngen =1000):
    population ← problem.init_population() ;
    /* see if we generated the solution by accident                         */
    best ← problem.fittest (population, f_thres) ;
    if best exists: return -1, best;
    for i in range(ngen):
        population ← problem.next_generation(population) ;
        best ← problem.fittest (population, f_thres) ;
        if best exists: return i, best;
    /* the chromosome with fitness value better than f_thres is not found   */
    best ← problem.fittest (population) ;
    return ngen, best
```

You should implement the algorithm using the methods of the class `GeneticProblem` as shown in the pseudocode. For the details of the class `GeneticProblem`, see `hw4_utils.py`. If you have completed implementing the three classic problems as described in the later sections of this handout, you can test your implementation of the algorithm as follows.

```
>>> p = NQueensProblem(10, (0,1,2,3), 4, 0.2)
>>> i, sol = genetic_algorithm(p, f_thres=6, ngen=1000)
>>> i, sol
(2, (1, 3, 0, 2))
>>> p.fitness_fn(sol)
6
>>> p = NQueensProblem(100, (0,1,2,3,4,5,6,7), 8, 0.2)
>>> i, sol = genetic_algorithm(p, f_thres=25, ngen=1000)
>>> i, sol
(1, (4, 1, 1, 6, 2, 3, 7, 3))
>>> p.fitness_fn(sol)
25
>>> p = NQueensProblem(100, (0,1,2,3,4,5,6,7), 8, 0.2)
>>> i, sol = genetic_algorithm(p, f_thres=28, ngen=1000)
>>> i, sol
(218, (2, 5, 7, 1, 3, 0, 6, 4))
>>> p.fitness_fn(sol)
28

>>> p = FunctionProblem(12, (10,10), 2, 0.2)
>>> i, sol = genetic_algorithm(p, f_thres=-18, ngen=1000)
>>> i, sol
(25, (9.004375331562994, 8.533970262737833))
>>> p.fitness_fn(sol)
-18.128675496887553
>>> p = FunctionProblem(20, (10,10), 2, 0.2)
>>> i, sol = genetic_algorithm(p, f_thres=-18.5519, ngen=1000)
>>> i, sol
(386, (9.039685760225542, 8.670632807448825))
>>> p.fitness_fn(sol)
-18.554571709857512

>>> p = HamiltonProblem(100, univ_bases, 20, 0.1, univ_map)
>>> i, sol = genetic_algorithm(p, f_thres=15000, ngen=1000)
>>> i, sol
(19, ('Yale', 'Pittsburgh', 'Michigan', 'Louisville', 'Oklahoma', 'Louisiana', 'Notre_Dame',
      'Wisconsin', 'Oregon', 'Brigham_Young', 'Arizona_State', 'Stanford', 'New_Mexico',
      'Colorado', 'North_Dakota', 'Texas_AM', 'Florida_State', 'Duke', 'Ohio', 'Brown'))
>>> p.fitness_fn(sol)
14318
>>> p = HamiltonProblem(200, univ_bases, 20, 0.1, univ_map)
>>> i, sol = genetic_algorithm(p, f_thres=12000, ngen=1000)
>>> i, sol
(48, ('Florida_State', 'Duke', 'Wisconsin', 'Notre_Dame', 'Louisville', 'Michigan',
      'Pittsburgh', 'Brown', 'Yale', 'Ohio', 'Oklahoma', 'Texas_AM', 'Colorado', 'New_Mexico',
      'Arizona_State', 'Stanford', 'Oregon', 'Brigham_Young', 'North_Dakota', 'Louisiana'))
>>> p.fitness_fn(sol)
11966
```

2 N-Queens Problem

Implement the class `NQueensProblem` for `genetic_algorithm`. A chromosome of `NQueensProblem` will be represented by an N-tuple $(Q_0, Q_1, \cdots, Q_{N-1})$, where Q_i represents the row number of the queen on the column i. Thus, the length (`g_len`) of a chromosome is N and the domain (`g_bases`) of each Q_i is $0 \leq Q_i < N$.

1. `__init__(self, n, g_bases, g_len, m_prob)` should initialize the parent portion of the instance by calling the parent's `__init__` method with the proper arguments. The arguments are

   - `n`: the population size, i.e., the number of chromosomes in the population
   - `g_bases`: the domain of each gene in a chromosome
   - `g_len`: the length of a chromosome
   - `m_prob`: the mutation probability

```
>>> p = NQueensProblem(10, (0,1,2,3,4,5,6,7), 8, 0.1)
>>> p.n, p.g_len, p.m_prob
(10, 8, 0.1)
>>> p.g_bases
(0, 1, 2, 3, 4, 5, 6, 7)

>>> p = NQueensProblem(10, (0,1,2,3), 4, 0.2)
>>> p.n, p.g_len, p.m_prob
(10, 4, 0.2)
>>> p.g_bases
(0, 1, 2, 3)
>>>
```

2. `init_population(self)` returns a list of `n` chromosomes that are generated randomly. A chromosome is a tuple of `g_len` items randomly selected from `g_bases`.

```
>>> p = NQueensProblem(5, range(4), 4, 0.2)
>>> p.init_population()
[(2, 1, 2, 0), (0, 2, 2, 1), (0, 3, 1, 3), (2, 2, 1, 0), (2, 0, 1, 3)]

>>> p = NQueensProblem(5, range(8), 8, 0.2)
>>> p.init_population()
[(0, 7, 3, 5, 3, 6, 1, 2), (0, 7, 3, 2, 2, 6, 0, 1), (4, 5, 5, 6, 2, 7, 2, 3),
 (5, 1, 2, 1, 6, 2, 1, 7), (3, 0, 0, 6, 0, 5, 3, 7)]
```

3. `next_generation(self, population)` returns the next generation of population. The next generation is a list of `n` chromosomes obtained by applying `crossover` and `mutate` to the given `population`. Note that we will not keep any chromosomes from `population`. We will generate `n` new chromosomes to form a new generation.

```
>>> p = NQueensProblem(5, range(8), 8, 0.2)
>>> population = [(2, 4, 7, 7, 5, 2, 4, 3), (3, 6, 6, 2, 3, 0, 0, 0),
                  (0, 2, 6, 0, 4, 1, 2, 7), (2, 6, 2, 0, 6, 0, 5, 5),
                  (0, 5, 0, 0, 2, 0, 1, 5)]
>>> p.next_generation(population)
[(2, 4, 7, 7, 6, 0, 5, 5), (3, 6, 6, 1, 3, 0, 0, 5),
    (3, 6, 6, 2, 3, 0, 5, 5), (2, 4, 7, 7, 5, 2, 4, 5),
    (2, 2, 6, 0, 4, 1, 2, 7)]
```

4. `mutate(self, chrom)` returns a chromosome obtained by mutating the given `chrom` at a random position with the probability of `m_prob`. To do this, first generate a random floating point number `p`, $0 \leq$ `p` ≤ 1. If `p` is greater than `m_prob`, the function simply returns `chrom` without mutating it. If `p` is less than or equal to `m_prob`, we will generate a random index `i`, $0 \leq$ `i` $<$ `g_len`, and replace `chrom[i]` with a new item randomly selected from `g_bases`. The resulting mutated chromosome will then be returned.

```
>>> p = NQueensProblem(10, range(8), 8, 0.5)
>>> p.mutate((4, 4, 4, 2, 6, 6, 4, 3))
(4, 4, 4, 2, 1, 6, 4, 3)
>>> p.mutate((4, 4, 4, 2, 6, 6, 4, 3))
(4, 4, 4, 2, 6, 6, 4, 3)
```

5. `crossover(self, chrom1, chrom2)` returns an offspring obtained by crossing over the given chromosomes, `chrom1` and `chrom2`. If the crossover occurs at a random index `i`, then the offspring is created by concatenating `chrom1[:i]` and `chrom2[i:]`.

```
>>> p = NQueensProblem(10, range(8), 8, 0.2)
>>> p.crossover((1, 4, 5, 6, 7, 1, 0, 2), (1, 3, 5, 1, 4, 7, 4, 4))
(1, 4, 5, 6, 4, 7, 4, 4)

>>> p = NQueensProblem(10, range(15), 15, 0.2)
>>> p.crossover((7, 13, 3, 6, 10, 8, 3, 13, 0, 12, 12, 2, 14, 4, 12),
                (1, 1, 8, 13, 3, 8, 5, 7, 0, 11, 12, 11, 11, 1, 9))
(7, 13, 3, 6, 10, 8, 5, 7, 0, 11, 12, 11, 11, 1, 9)
```

6. `fitness_fn(self, chrom)` returns the fitness value of `chrom`. The fitness value of a chromosome is calculated by counting the number of queen pairs that are not attacking each other. Note that the maximum fitness value of a chromosome in N-Queens problem will be $\binom{N}{2}$. For example, for a chromosome in 4-Queens problem, (Q_0, Q_1, Q_2, Q_3), there is a total of six queen pairs: $\{Q_0, Q_1\}$, $\{Q_0, Q_2\}$, $\{Q_0, Q_3\}$, $\{Q_1, Q_2\}$, $\{Q_1, Q_3\}$, $\{Q_2, Q_3\}$. Thus, a chromosome that is a correct solution to the 4-Queens problem will have the maximum fitness value of 6, since none of the 6 pairs will have the two queens attacking each other.

```
>>> p = NQueensProblem(10, range(4), 4, 0.2)
>>> p.fitness_fn((2,3,0,1))
2
>>> p.fitness_fn((2,0,3,1))
6

>>> p = NQueensProblem(10, range(8), 8, 0.2)
>>> p.fitness_fn((4, 6, 6, 2, 7, 6, 6, 4))
17
>>> p.fitness_fn((5, 3, 0, 4, 7, 1, 6, 2))
28
```

7. `select(self, m, population)` returns a list of `m` chromosomes randomly selected from `population` using the fitness proportionate selection. The probability of k^{th} chromosome to be selected from `population` of `n` chromosomes is calculated as follows:

$$\text{Prob}(c_k) = \frac{\texttt{fitness\_fn}(c_k)}{\sum_{i=0}^{\texttt{n}-1} \texttt{fitness\_fn}(c_i)}$$

Note that the following must hold:

$$\sum_{k=0}^{\texttt{n}-1} \text{Prob}(c_k) = 1$$

This probability distribution is used to randomly select m chromosomes from `population` according to the fitness proportionate selection process. For example, consider `population` containing 5 chromosomes, $[\ c_0,\ c_1,\ c_2,\ c_3,\ c_4\ ]$. Suppose that the corresponding probability distribution calculated by the above equation is [0.03, 0.15, 0.41, 0.12, 0.29]. From the probability distribution, we calculate the cumulative distribution, [0.03, 0.18, 0.59, 0.71, 1.00]. Now we are ready to select a chromosome. First, generate a random floating point number `p`, $0 \le \texttt{p} \le 1$. If $\texttt{p} \le 0.03$, select c_0. If $0.03 < \texttt{p} \le 0.18$, select c_1. If $0.18 < \texttt{p} \le 0.59$, select c_2. If $0.59 < \texttt{p} \le 0.71$, select c_3. Finally, if $0.71 < \texttt{p} \le 1.00$, select c_4. Repeat this selection process `m` times to randomly select `m` chromosomes.

Note that the test cases below show that the higher the fitness value of a chromosome, the better chance of the chromosome being selected.

```
>>> p = NQueensProblem(5, range(8), 8, 0.2)
>>> population = [(4, 2, 6, 4, 7, 4, 3, 4), (3, 5, 5, 1, 5, 0, 7, 7),
                  (0, 4, 0, 3, 4, 5, 6, 6), (5, 7, 3, 1, 7, 4, 5, 7),
                  (6, 7, 5, 7, 4, 7, 5, 7)]
>>> list(map(p.fitness_fn, population))
[17, 22, 15, 21, 18]
>>> p.select(0, population)
[]
>>> p.select(2, population)
[(6, 7, 5, 7, 4, 7, 5, 7), (3, 5, 5, 1, 5, 0, 7, 7)]
>>> p.select(5, population)
[(0, 4, 0, 3, 4, 5, 6, 6), (6, 7, 5, 7, 4, 7, 5, 7), (5, 7, 3, 1, 7, 4, 5, 7),
 (3, 5, 5, 1, 5, 0, 7, 7), (5, 7, 3, 1, 7, 4, 5, 7)]
>>> p.select(10, population)
[(0, 4, 0, 3, 4, 5, 6, 6), (3, 5, 5, 1, 5, 0, 7, 7), (5, 7, 3, 1, 7, 4, 5, 7),
 (4, 2, 6, 4, 7, 4, 3, 4), (4, 2, 6, 4, 7, 4, 3, 4), (6, 7, 5, 7, 4, 7, 5, 7),
 (5, 7, 3, 1, 7, 4, 5, 7), (3, 5, 5, 1, 5, 0, 7, 7), (5, 7, 3, 1, 7, 4, 5, 7),
 (6, 7, 5, 7, 4, 7, 5, 7)]
```

8. `fittest(self, population, f_thres=None)` returns the best chromosome in `population` if `f_thres` is `None`. If `f_thres` is not `None`, it returns the best chromosome only if its fitness value is **greater than or equal to** `f_thres`. Otherwise, it returns `None`. Note that the last test case below prints nothing since it returns `None`.

```
>>> p = NQueensProblem(5, range(8), 8, 0.2)
>>> population = [(5, 4, 0, 2, 1, 1, 4, 3), (1, 4, 5, 2, 0, 1, 5, 7),
                  (0, 2, 7, 4, 6, 0, 4, 5), (6, 3, 5, 5, 2, 3, 1, 0),
                  (6, 5, 1, 7, 7, 2, 2, 3)]
```

```
>>> list(map(p.fitness_fn, population))
[17, 22, 23, 19, 22]
>>> p.fittest(population)
(0, 2, 7, 4, 6, 0, 4, 5)
>>> p.fittest(population, 23)
(0, 2, 7, 4, 6, 0, 4, 5)
>>> p.fittest(population, 25)
>>>
```

3 Function Optimization Problem

Implement the class FunctionProblem for genetic_algorithm. In this problem, we will find (x, y) that **minimizes** the following function:

$$f(x, y) = x \cdot \sin(4x) + 1.1 \cdot y \cdot \sin(2y)$$

Instead of using any encryption scheme, a chromosome of FunctionProblem will be represented simply by a tuple of 2 floating point numbers (x, y), where $0 \le \mathtt{x} \le \mathtt{x}_{\max}$ and $0 \le \mathtt{y} \le \mathtt{y}_{\max}$. Thus, g_len for FunctionProblem is 2 and g_bases will be represented by a tuple $(\mathtt{x}_{\max}, \mathtt{y}_{\max})$.

1. __init__(self, n, g_bases, g_len, m_prob) should initialize the parent portion of the instance by calling the parent's __init__ method with the proper arguments.

```
>>> p = FunctionProblem(10, (5,5), 2, 0.1)
>>> p.n, p.g_len, p.m_prob
(10, 2, 0.1)
>>> p.g_bases
(5, 5)

>>> p = FunctionProblem(20, (10,20), 2, 0.2)
>>> p.n, p.g_len, p.m_prob
(20, 2, 0.2)
>>> p.g_bases
(10, 20)
```

2. init_population(self) returns a list of n chromosomes that are generated randomly. A chromosome is a tuple of 2 random floating point numbers (x, y) in the range of $0 \le \mathtt{x} \le \mathtt{x}_{\max}$ and $0 \le \mathtt{y} \le \mathtt{y}_{\max}$.

```
>>> p = FunctionProblem(3, (5,5), 2, 0.2)
>>> p.init_population()
[(1.069291442240965, 3.4258608103087766), (0.5814586733970367, 0.1062245125561545),
 (3.2155287126033154, 0.26426882116695194)]
>>> p = FunctionProblem(5, (10,20), 2, 0.1)
>>> p.init_population()
[(8.181250719670372, 14.869963460562444), (2.903310334690822, 18.61952673728084),
 (3.498760496405838, 16.46420547601544), (1.2659877772201877, 14.089176499032348),
 (9.823642185170762, 8.441220137200729)]
```

3. next_generation(self, population) returns the next generation of population. Instead of replacing the given population entirely as we did in NQueensProblem, we will keep the best half of population

and generate the other half by applying `crossover` and `mutate` to the best half we decided to keep. Note that the lower the fitness value of a chromosome, the better the quality of a chromosome, since we are trying to find the solution that minimizes the given function.

```
>>> p = FunctionProblem(6, (5,5), 2, 0.2)
>>> population =
     [(2.066938780637087, 1.650998608284147), (0.7928608069345605, 1.678831697303177),
      (3.685189771001436, 1.4280879354988107), (3.860362372295962, 1.0789325520768145),
      (4.673003350118171, 4.780722998076655), (2.5701433643372247, 4.9078727479819335)]
>>> list(map(p.fitness_fn, population))
[1.6024472559453082, -0.41958730465195776, 3.4763217693695787,
 2.0048163673123365, -1.4496290991082836, -3.9980346848519694]
>>> p.next_generation(population)
[(4.673003350118171, 4.780722998076655), (4.673003350118171, 4.780722998076655),
 (2.5701433643372247, 4.9078727479819335), (2.5701433643372247, 4.9078727479819335),
 (4.673003350118171, 4.780722998076655), (0.7928608069345605, 1.678831697303177)]
```

4. `mutate(self, chrom)` returns a chromosome obtained by mutating the given `chrom` at a random position with the probability of `m_prob`. To do this, first generate a random floating point number `p`, $0 \leq$ `p` ≤ 1. If `p` is greater than `m_prob`, the function simply returns `chrom` without mutating it. If `p` is less than or equal to `m_prob`, we will generate a random index `i`, $0 \leq$ `i` $<$ `len(chrom)`, and replace `chrom[i]` with a new random floating point number `f`, $0 \leq$ `f` $\leq$ `g_bases[i]`. The resulting mutated chromosome will then be returned.

```
>>> p = FunctionProblem(10, (10,20), 2, 0.5)
>>> c = (4.525394646650255, 13.005368584538973)
>>> p.mutate(c)
(4.525394646650255, 9.505694180053396)

>>> p.mutate(c)
(4.525394646650255, 13.005368584538973)

>>> p.mutate(c)
(0.3146989344654505, 13.005368584538973)
```

5. `crossover(self, chrom1, chrom2)` returns an offspring obtained by crossing over the given chromosomes, `chrom1` and `chrom2`. Since a chromosome is simply a pair of two floating point numbers without any encryption scheme, we will use the linear interpolation as a crossover operation. For example, suppose two parent chromosomes are (x_1, y_1) and (x_2, y_2). We first randomly choose which component of the chromosomes to interpolate. We also need to generate a random floating point number α such that $0 \leq \alpha \leq 1$. If x component was chosen to be interpolated, the offspring will be (x_{new}, y_1), where

$$x_{\text{new}} = (1 - \alpha) \cdot x_1 + \alpha \cdot x_2$$

Similarly, if y component was chosen, the offspring will be (x_1, y_{new}), where

$$y_{\text{new}} = (1 - \alpha) \cdot y_1 + \alpha \cdot y_2$$

The function returns this new offspring as its result.

```
>>> p = FunctionProblem(10, (10,10), 2, 0.2)
>>> c1 = (7.377910209443304, 3.6708167924621793)
>>> c2 = (2.5195159164374434, 7.248941413091508)
>>> p.crossover(c1, c2)
(7.377910209443304, 4.912246497034061)

>>> p.crossover(c1, c2)
(4.562265309739303, 3.6708167924621793)
```

6. `fitness_fn(self, chrom)` returns the fitness value of the given `chrom`. The fitness value of a chromosome is calculated using the function to be minimized:

$$f(x, y) = x \cdot \sin(4x) + 1.1 \cdot y \cdot \sin(2y)$$

Note that the fitness value of a chromosome can be negative.

```
>>> p = FunctionProblem(10, (10,20), 2, 0.2)
>>> p.fitness_fn((1,1))
0.24342467420032166
>>> p.fitness_fn((4.562265309739303, 3.6708167924621793))
0.9415043737665614
>>> p.fitness_fn((2.5409025501787963, 16.828483737264428))
12.79566330899491
>>> p.fitness_fn((9.0449, 8.6643))
-18.55190308788758
```

7. `select(self, m, population)` returns a list of `m` chromosomes randomly selected from `population` using the fitness proportionate selection. Since the fitness value of a chromosome can be negative, we cannot determine the probability of a chromosome to be selected simply as the ratio of its fitness value to the sum of the fitness values of all chromosomes. Instead, we will use the rank-weighted probability. As an example, consider the following six chromosomes sorted in the order of their fitness values:

Rank	Chromosome	Fitness Value
0	(2.5701433643372247, 4.9078727479819335)	-3.9980346848519694
1	(4.673003350118171, 4.780722998076655)	-1.4496290991082836
2	(0.7928608069345605, 1.678831697303177)	-0.41958730465195776
3	(2.066938780637087, 1.650998608284147)	1.6024472559453082
4	(3.860362372295962, 1.0789325520768145)	2.0048163673123365
5	(3.685189771001436, 1.4280879354988107)	3.4763217693695787

The probability of a chromosome of rank k to be selected from `population` of `n` chromosomes is calculated as follows:

$$\text{Prob}(c_k) = \frac{\mathtt{n} - k}{\sum_{i=1}^{\mathtt{n}} i}$$

Note that the following must hold:

$$\sum_{k=0}^{\mathtt{n}-1} \text{Prob}(c_k) = 1$$

The probability distribution of the six chromosomes calculate as above is

```
[0.2857142857142857, 0.23809523809523808, 0.19047619047619047,
 0.14285714285714285, 0.09523809523809523, 0.047619047619047616]
```

and the cumulative distribution is

```
[0.2857142857142857, 0.5238095238095237, 0.7142857142857142,
 0.857142857142857, 0.9523809523809522, 0.9999999999999999]
```

Now we can select `m` chromosomes at random according to the fitness proportionate selection as we did in `NQueensProblem`.

```
>>> p = FunctionProblem(6, (5,5), 2, 0.2)
>>> population = [(3.2785989393078507, 1.5854499726282851),
                  (2.797299697316915, 1.6070456140483396),
                  (1.7691382088986807, 2.893332226588736),
                  (4.718397971920581, 0.5107148205878392),
                  (1.6729300262695035, 1.1621142165573013),
                  (4.83467584947063, 4.824603641199292)]
>>> list(map(p.fitness_fn, population))
[1.657054107957212, -2.87307443334604, -0.2552252197910254, 0.5925228026882771,
 1.5969373196316696, 1.090598835733723]
>>> p.select(0, population)
[]
>>> p.select(2, population)
[(4.83467584947063, 4.824603641199292), (1.7691382088986807, 2.893332226588736)]
>>> p.select(6, population)
[(1.7691382088986807, 2.893332226588736), (3.2785989393078507, 1.5854499726282851),
 (2.797299697316915, 1.6070456140483396), (4.718397971920581, 0.5107148205878392),
 (1.7691382088986807, 2.893332226588736), (1.7691382088986807, 2.893332226588736)]
>>> p.select(10, population)
[(1.7691382088986807, 2.893332226588736), (2.797299697316915, 1.6070456140483396),
 (2.797299697316915, 1.6070456140483396), (1.6729300262695035, 1.1621142165573013),
 (1.7691382088986807, 2.893332226588736), (1.7691382088986807, 2.893332226588736),
 (1.7691382088986807, 2.893332226588736), (2.797299697316915, 1.6070456140483396),
 (4.83467584947063, 4.824603641199292), (1.6729300262695035, 1.1621142165573013)]
```

8. `fittest(self, population, f_thres=None)` returns the best chromosome in `population` if `f_thres` is `None`. If `f_thres` is not `None`, it returns the best chromosome only if its fitness value is **less than or equal to** `f_thres`. Otherwise, it returns `None`. Note that the last test case below prints nothing since it returns `None`.

```
>>> p = FunctionProblem(6, (5,5), 2, 0.2)
>>> population = [(2.066938780637087, 1.650998608284147),
                  (3.860362372295962, 1.0789325520768145),
                  (2.5701433643372247, 4.9078727479819335),
                  (4.673003350118171, 4.780722998076655),
                  (3.685189771001436, 1.4280879354988107),
                  (0.7928608069345605, 1.678831697303177)]
>>> list(map(p.fitness_fn, population))
```

```
[1.6024472559453082, 2.0048163673123365, -3.9980346848519694,
 -1.4496290991082836, 3.4763217693695787, -0.41958730465195776]
>>> p.fittest(population)
(2.5701433643372247, 4.9078727479819335)

>>> p.fittest(population, f_thres=-3.9)
(2.5701433643372247, 4.9078727479819335)

>>> p.fittest(population, f_thres=-5.0)
>>>
```

4 Hamilton Circuit Problem

Implement the class `HamiltonProblem` for `genetic_algorithm`. In this problem, we will find a Hamilton circuit with a minimum length in a graph of N cities. A chromosome of `HamiltonProblem` will be represented by an N-tuple $(\texttt{City}_0, \texttt{City}_1, \cdots, \texttt{City}_{N-1})$, representing the cycle

$$\texttt{City}_0 \rightarrow \texttt{City}_1 \rightarrow \cdots \rightarrow \texttt{City}_{N-1} \rightarrow \texttt{City}_0$$

Thus, `g_len` and `g_bases` are the number of cities and the list containing the names of cities in the graph, respectively.

1. `__init__(self, n, g_bases, g_len, m_prob, graph=None)` should initialize the parent portion of the instance by calling the parent's `__init__` method with the proper arguments and initialize its own instance variables as needed.

```
>>> p = HamiltonProblem(5, test_bases, 5, 0.2, test_map)
>>> p.n, p.g_len, p.m_prob
(5, 5, 0.2)
>>> p.g_bases
['A', 'B', 'C', 'D', 'E']
```

2. `init_population(self)` returns a list of `n` chromosomes that are generated randomly. Since each city in the map needs to be present once and only once in a chromosome, a random chromosome is simply a random shuffle of `g_bases`. For random shuffling of a tuple, see `random.shuffle()` function.

```
>>> p = HamiltonProblem(5, test_bases, 5, 0.2, test_map)
>>> p.init_population()
[('D', 'E', 'C', 'B', 'A'), ('A', 'E', 'C', 'B', 'D'), ('A', 'C', 'B', 'E', 'D'),
 ('C', 'D', 'A', 'E', 'B'), ('D', 'C', 'A', 'B', 'E')]
```

3. `next_generation(self, population)` returns the next generation of population. The next generation is a list of `n` chromosomes by selecting `n` best chromosomes from the list of $(2 \times \texttt{n})$ chromosomes containing the given `population` of `n` chromosomes and the new population of `n` chromosomes obtained by applying `crossover` and `mutate` to `population`.

```
>>> p = HamiltonProblem(5, test_bases, 5, 0.2, test_map)
>>> population = [('D', 'E', 'C', 'B', 'A'), ('A', 'E', 'C', 'B', 'D'),
                  ('A', 'C', 'B', 'E', 'D'), ('C', 'D', 'A', 'E', 'B'),
                    (’D’, ’C’, ’A’, ’B’, ’E’)]
>>> p.next_generation(population)
[(’D’, ’E’, ’C’, ’B’, ’A’), (’D’, ’E’, ’C’, ’B’, ’A’), (’D’, ’E’, ’C’, ’B’, ’A’),
 (’A’, ’E’, ’C’, ’B’, ’D’), (’C’, ’E’, ’A’, ’B’, ’D’)]
```

4. `mutate(self, chrom)` returns a chromosome obtained by mutating the given `chrom` at a random position with the probability of `m_prob`. To do this, first generate a random floating point number `p`, $0 \leq$ `p` ≤ 1. If `p` is greater than `m_prob`, the function simply returns `chrom` without mutating it. If `p` is less than or equal to `m_prob`, then the mutation occurs. Since each city in the map needs to be present once and only once in a chromosome, we cannot mutate the chromosome in a usual way. Instead, we will generate two random indices and swap the values at the indices in `chrom`. The resulting mutated chromosome will then be returned.

```
>>> p = HamiltonProblem(5, test_bases, 5, 0.5, test_map)
>>> p.mutate((’D’, ’E’, ’C’, ’B’, ’A’))
(’D’, ’E’, ’C’, ’B’, ’A’)

>>> p.mutate((’D’, ’E’, ’C’, ’B’, ’A’))
(’D’, ’C’, ’E’, ’B’, ’A’)
```

5. `crossover(self, chrom1, chrom2)` returns an offspring obtained by crossing over the given chromosomes, `chrom1` and `chrom2`. Since each city in the map needs to be present once and only once in a chromosome, the usual way of crossover will not work for `HamiltonProblem`. Instead, we will use a technique called **cycle crossover**. First generate a random index, `i`, to begin the cycle crossover process. The two parent chromosomes exchange the genes at this location to create a new offspring. Unless the exchanged genes are the same value, each offspring has a duplicate value in its chromosome. We swap the duplicate gene in the first offspring with whatever value is in the same location in the second offspring. We repeat this process until there are no duplicates in the first offspring. The function finally returns the first offspring. This process is illustrated with two parent chromosomes of length 6 and the random initial index of 3 below:

Parents	Step 1	Step 2	Step 3	Step 4
$C_4C_1C_5\underline{C_3}C_2C_6$	$C_4C_1C_5C_2\underline{C_2}C_6$	$C_4\underline{C_1}C_5C_2C_1C_6$	$\underline{C_4}C_4C_5C_2C_1C_6$	$C_3C_4C_5C_2C_1C_6$
$C_3C_4C_6\enclose{circle}{C_2}C_1C_5$	$C_3C_4C_6C_3\enclose{circle}{C_1}C_5$	$C_3\enclose{circle}{C_4}C_6C_3C_2C_5$	$\enclose{circle}{C_3}C_1C_6C_3C_2C_5$	$C_4C_1C_6C_3C_2C_5$

```
>>> p = HamiltonProblem(10, [’C1’, ’C2’, ’C3’, ’C4’, ’C5’, ’C6’], 6, 0.2)
>>> p.crossover((’C4’,’C1’,’C5’,’C3’,’C2’,’C6’), (’C3’,’C4’,’C6’,’C2’,’C1’,’C5’))
(’C3’, ’C4’, ’C5’, ’C2’, ’C1’, ’C6’)

>>> p = HamiltonProblem(5, test_bases, 5, 0.5, test_map)
>>> p.crossover((’D’, ’E’, ’C’, ’B’, ’A’), (’A’, ’E’, ’C’, ’B’, ’D’))
(’A’, ’E’, ’C’, ’B’, ’D’)
>>> p.crossover((’A’, ’E’, ’C’, ’B’, ’D’), (’A’, ’C’, ’B’, ’E’, ’D’))
(’A’, ’C’, ’B’, ’E’, ’D’)
(’C3’, ’C4’, ’C5’, ’C2’, ’C1’, ’C6’)
```

6. `fitness_fn(self, chrom)` returns the fitness value of `chrom`. The fitness value of a chromosome is simply the cost of the circuit represented by the chromosome. For example, given the chromosome $(\mathtt{c}_0, \mathtt{c}_1, \cdots, \mathtt{c}_{\mathtt{n}-1})$, the circuit cost is

$$\left[\sum_{i=0}^{\mathtt{n}-2} d(\mathtt{c}_i, \mathtt{c}_{i+1})\right] + d(\mathtt{c}_{\mathtt{n}-1}, \mathtt{c}_0)$$

where, $d(\mathtt{c}_i, \mathtt{c}_j)$ is the distance between the two cities $\mathtt{c}_i$ and $\mathtt{c}_j$.

```
>>> p = HamiltonProblem(5, test_bases, 5, 0.5, test_map)
>>> p.fitness_fn(('D', 'E', 'C', 'B', 'A'))
37
>>> p.fitness_fn(('A', 'E', 'C', 'B', 'D'))
39
>>> p.fitness_fn(('C', 'D', 'A', 'E', 'B'))
53
```

7. `select(self, m, population)` returns a list of `m` chromosomes randomly selected from `population` using the fitness proportionate selection. Consider the given `population` containing `n` chromosomes: $(\mathtt{c}_0, \mathtt{c}_1, \cdots, \mathtt{c}_{\mathtt{n}-1})$. Suppose the corresponding list of their fitness values is $(f_0, f_1, \cdots, f_{\mathtt{n}-1})$. Thus, the sum of the fitness values of all chromosomes in `population` is $T = \sum_{i=0}^{\mathtt{n}-1} f_i$. We then calculate the probability of k^{th} chromosome to be selected from `population` as follows:

$$\text{Prob}(c_k) = \frac{T - f_k}{\sum_{j=0}^{\mathtt{n}-1}(T - f_j)}$$

Note that the following must hold:

$$\sum_{k=0}^{\mathtt{n}-1} \text{Prob}(\mathtt{c}_k) = 1$$

Finally, we can calculate the cumulative distribution from this probability distribution and select `m` chromosomes at random using the fitness proportionate selection as we did in `NQueensProblem`.

```
>>> p = HamiltonProblem(5, test_bases, 5, 0.5, test_map)
>>> population = [('D', 'E', 'A', 'B', 'C'), ('B', 'C', 'D', 'E', 'A'),
                  ('A', 'D', 'B', 'C', 'E'), ('B', 'E', 'C', 'D', 'A'),
                  ('B', 'E', 'A', 'D', 'C')]
>>> list(map(p.fitness_fn, population))
[52, 52, 39, 43, 53]
>>> p.select(0, population)
[]
>>> p.select(2, population)
[('B', 'E', 'A', 'D', 'C'), ('D', 'E', 'A', 'B', 'C')]
>>> p.select(5, population)
[('B', 'E', 'C', 'D', 'A'), ('B', 'E', 'C', 'D', 'A'), ('B', 'C', 'D', 'E', 'A'),
 ('A', 'D', 'B', 'C', 'E'), ('B', 'C', 'D', 'E', 'A')]
>>> p.select(7, population)
[('B', 'C', 'D', 'E', 'A'), ('B', 'C', 'D', 'E', 'A'), ('B', 'E', 'A', 'D', 'C'),
 ('A', 'D', 'B', 'C', 'E'), ('B', 'E', 'C', 'D', 'A'), ('A', 'D', 'B', 'C', 'E'),
 ('B', 'C', 'D', 'E', 'A')]
```

8. `fittest(self, population, f_thres=None)` returns the best chromosome in `population` if `f_thres` is `None`. If `f_thres` is not `None`, it returns the best chromosome only if its fitness value is **less than or equal to** `f_thres`. Otherwise, it returns `None`. Note that the last test case below prints nothing since it returns `None`.

```
>>> p = HamiltonProblem(5, test_bases, 5, 0.5, test_map)
>>> population = [('D', 'E', 'A', 'B', 'C'), ('B', 'C', 'D', 'E', 'A'),
                  ('A', 'D', 'B', 'C', 'E'), ('B', 'E', 'C', 'D', 'A'),
                  ('B', 'E', 'A', 'D', 'C')]
>>> list(map(p.fitness_fn, population))
[52, 52, 39, 43, 53]
>>> p.fittest(population)
('A', 'D', 'B', 'C', 'E')
>>> p.fittest(population, f_thres=40)
('A', 'D', 'B', 'C', 'E')
>>> p.fittest(population, f_thres=30)
>>>
```