Code

```
"""
Minimum Spanning Tree
Using *Genetic Algorithm*
"""
import math
from math import inf, isinf
from random import random, randint, seed
from typing import List

true, false, null = True, False, None


class CityRepository:
    number_of_cities = 10
    costs = [
        [inf, 10,  5,   inf, inf, inf, inf, inf, 6,   inf],
        [10,  inf, 4,   30,  28,  19,  12,  4,   inf, inf],
        [5,   4,   inf, inf, 25,  inf, inf, inf, 13,  inf],
        [inf, 30,  inf, inf, 7,   inf, 5,   40,  inf, inf],
        [inf, 28,  25,  7,   inf, 60,  inf, inf, inf, 11],
        [inf, 19,  inf, inf, 60,  inf, inf, 17,  6,   1, ],
        [inf, 12,  inf, 5,   inf, inf, inf, 8,   inf, inf],
        [inf, 4,   inf, 40,  inf, 17,  8,   inf, inf, 14],
        [6,   inf, 13,  inf, inf, 6,   inf, inf, inf, 4, ],
        [inf, inf, inf, inf, 11,  1,   inf, 14,  4,   inf]
    ]

    @staticmethod
    def cities_from(src):
        """
        Returns a list of cities accessible from src
        :rtype: List[(int, int)]
        """
        # return [(to, dist) for frm, to, dist in CityRepository.edges if frm == src]
        return list(enumerate(CityRepository.costs[src]))

    @staticmethod
    def distance(src, dest):
        """
        Distance of road from src to dest
        :rtype: float|int
        """
```

```
        return CityRepository.costs[src][dest]

    @staticmethod
    def cost_of_road(index):
        """
        Returns the cost of a road by index
        :rtype: Union[int, float]
        """
        return CityRepository.at(index)[2]

    @staticmethod
    def at(index):
        """
        Get the road at the specified index
        :return: Tuple[int, int, float]
        """
        count = 0
        for frm in range(CityRepository.number_of_cities):
            for to in range(CityRepository.number_of_cities):
                if not isinf(CityRepository.costs[frm][to]):
                    if count == index:
                        return frm, to, CityRepository.costs[frm][to]
                    count += 1
        raise IndexError("No road with this index")

    @staticmethod
    def number_of_roads():
        """
        Get the number of roads
        :rtype: int
        """
        return 44


class Chromosome:
    """
    A chromosome is essentially a list of roads.
    If n is the number of cities, n-1 roads are needed to connect the cities.
    """
    genes: List[int]

    def __init__(self, genes=null, initialize=false):
        """
        Initializes the Chromosome
```

```
        gene can be list of Genes or null.
        If gene is not null, initialize will not be used.
        If initialize is true, a random list will be generated.
        Otherwise, a null list will be generated.
        :param genes: List[Gene] or null
        :param initialize: boolean
        """

        if genes is null:
            chrome_size = (CityRepository.number_of_cities - 1)
            if initialize:
                max_road_index = CityRepository.number_of_roads() - 1
                self.genes = []
                for g in range(chrome_size):
                    gene = randint(0, max_road_index)
                    self.genes.append(gene)
            else:
                # Create a list of null value, same sized as number of cities
                self.genes = [null] * chrome_size
        else:
            # Use provided list of genes
            self.genes = genes

        # Cache for total distance
        self.cost_cache = null

    @property
    def cost(self):
        if self.cost_cache is not null:
            return self.cost_cache

        disconnected_sets = []
        cities = set()
        total_cost = 0.
        for gene in self.genes:
            frm, to, cost = CityRepository.at(gene)
            total_cost += cost
            cities.add(frm)
            cities.add(to)

            set_of_from = -1
            set_of_to = -1
            for i, disconnected_set in enumerate(disconnected_sets):
                if frm in disconnected_set:
```

```
                set_of_from = i
            if to in disconnected_set:
                set_of_to = i
            if set_of_from != -1 and set_of_to != -1:
                break
        if set_of_from == -1:
            if set_of_to == -1:
                disconnected_sets.append([frm, to])
            else:
                disconnected_sets[set_of_to].append(frm)
        else:
            if set_of_to == -1:
                disconnected_sets[set_of_from].append(to)
            elif set_of_from != set_of_to:
                disconnected_sets[set_of_from] += disconnected_sets[set_of_to]
                del disconnected_sets[set_of_to]

    # If all cities ain't present, its invalid
    if len(cities) < CityRepository.number_of_cities:
        total_cost = inf
    if len(cities) > CityRepository.number_of_cities:
        raise ValueError("Gene contains cities more than actually exists")

    # Cost is (sum of road costs) * (number of sets)
    self.cost_cache = total_cost * len(disconnected_sets)
    return self.cost_cache

@property
def fitness(self):
    fit = 1 / self.cost
    if math.isnan(fit):
        raise RuntimeError("Culprit found!")
    return fit

def crossover(self, parent2):
    parent1 = self
    child1, child2 = Chromosome(), Chromosome()
    assert len(parent1) == len(parent2)
    length = len(parent1) - 1
    break_point = randint(0, length)

    for i in range(break_point):
        child1.set(i, parent1.get(i))
        child2.set(i, parent2.get(i))
```

```
        for i in range(break_point, length + 1):
            child1.set(i, parent2.get(i))
            child2.set(i, parent1.get(i))

        return child1, child2

    def mutate(self, mutation_rate):
        if random() < mutation_rate:
            index = randint(0, len(self) - 1)
            value = randint(0, CityRepository.number_of_roads() - 1)
            self.set(index, value)
        return self

    def set(self, index, gene):
        self.cost_cache = null
        self.genes[index] = gene

    def get(self, index):
        return self.genes[index]

    def contains(self, gene):
        return gene in self.genes

    def index(self, gene):
        return self.genes.index(gene)

    def __len__(self):
        return len(self.genes)

    def __iter__(self):
        return iter(self.genes)

    def __repr__(self):
        return ', '.join([str(g) for g in self.genes])


class Population:
    chromosomes: List[Chromosome]

    def __init__(self, chrome=null, initialize=false):
        """
        Initializes a population with either a list of chromosomes
        or a number of chromosomes or null by default.
        if chrome is int and initialize is true, then a list of
```

```
    random chromosomes will be produced.
    :type chrome: Union[list, int, null]
    :type initialize: bool
    """
    if chrome is null:
        self.chromosomes = []
    elif isinstance(chrome, int):
        self.chromosomes = [Chromosome(initialize=initialize) for i in range(chrome)]
    elif isinstance(chrome, list):
        self.chromosomes = chrome
    else:
        raise TypeError()

    # Cache for superlative chromosomes
    self.best_cache = null
    self.worst_cache = null

def best(self, return_index=false):
    if not self.best_cache:
        # best_cache => Tuple(Chromosome, index)
        self.best_cache = (self.chromosomes[0], 0)

        for i in range(1, len(self)):
            if self.best_cache[0].fitness < self.chromosomes[i].fitness:
                self.best_cache = (self.chromosomes[i], i)
    if return_index:
        return self.best_cache
    return self.best_cache[0]

def worst(self, return_index=false):
    if not self.worst_cache:
        # worst_cache => Tuple(Chromosome, index)
        self.worst_cache = (self.chromosomes[0], 0)

        for i in range(1, len(self)):
            if self.worst_cache[0].fitness > self.chromosomes[i].fitness:
                self.worst_cache = (self.chromosomes[i], i)
    if return_index:
        return self.worst_cache
    return self.worst_cache[0]

def add(self, chromosome):
    """
    Add a chromosome or a population to population
```

```
        :param chromosome: Chromosome or Population
        :return: None
        """
        if isinstance(chromosome, Chromosome):
            self.chromosomes.append(chromosome)
        elif isinstance(chromosome, Population):
            self.chromosomes += chromosome.chromosomes
        elif isinstance(chromosome, list):
            self.chromosomes += chromosome
        else:
            raise TypeError(
                "Only chromosome or population can be added to population. " + type(chromosome) + "
given."
            )

    def at(self, index):
        return self.chromosomes[index]

    def at_range(self, frm=0, to=null):
        if to is null:
            to = len(self)
        return self.chromosomes[frm: to]

    def remove(self, index):
        del self.chromosomes[index]

    def sort(self):
        self.chromosomes = sorted(self.chromosomes, key=lambda ch: ch.cost)

    def __len__(self):
        return len(self.chromosomes)

    def __iter__(self):
        return iter(self.chromosomes)


class Environment:

    def __init__(self, population=null, mutation_rate=.02, strategy='whole_new'):
        if population is not null:
            self.population = population
        else:
            self.population = Population(Environment.default_population_size, initialize=true)
```

```
    if strategy not in Environment.strategies:
        raise RuntimeError("Unsupported update strategy")
    self.strategy = strategy
    self.mutation_rate = mutation_rate

def evolve(self, times=100, log=false):
    for time in range(times):
        new_pop = Population()
        for i in range(int(len(self) / 2)):
            parent1 = self.select_for_crossover()
            parent2 = self.select_for_crossover()
            offspring1, offspring2 = parent1.crossover(parent2)
            if random() < self.mutation_rate:
                offspring1 = offspring1.mutate(self.mutation_rate)
            if random() < self.mutation_rate:
                offspring2 = offspring2.mutate(self.mutation_rate)
            new_pop.add(offspring1)
            new_pop.add(offspring2)

        if self.strategy == Environment.strategies[0]:  # whole_new
            self.population = new_pop
        elif self.strategy == Environment.strategies[1]:  # best_only
            _, worst_index = new_pop.worst(return_index=true)
            new_pop.remove(worst_index)
            best_parent = self.population.best()
            new_pop.add(best_parent)
            self.population = new_pop
        elif self.strategy == Environment.strategies[2]:  # keep_parents
            new_pop.add(self.population)
            new_pop.sort()
            best_half = new_pop.at_range(to=len(self.population))
            self.population = Population(best_half)

        if log:
            print("At iteration {}, best cost: {}".format(time, self.population.best().cost))

    return self.population.best()

def __len__(self):
    return len(self.population)

def select_for_crossover(self):
    """
    Using roulette method
```

```python
        :rtype: Chromosome
        """
        total_fitness = 0.
        for chromosome in self.population:
            total_fitness = total_fitness + chromosome.fitness

        roulette = random()
        revolution = 0
        for chromosome in self.population:
            revolution += chromosome.fitness
            if revolution / total_fitness >= roulette:
                return chromosome

        raise RuntimeError("This can only be raised by precision error.")


Environment.strategies = ['whole_new', 'best_only', 'keep_parents']
Environment.default_population_size = 500


def main():
    seed(2)
    env = Environment(mutation_rate=.5, strategy='whole_new')
    env.evolve(times=30, log=true)
    best = env.population.best()
    print("Roads:", best, "with cost:", best.cost)
    print("Full Path:")
    for gene in best:
        print(CityRepository.at(gene))


main()
```

Output

```
At iteration 0, best cost: 64.0

At iteration 1, best cost: 64.0

At iteration 2, best cost: 60.0

At iteration 3, best cost: 51.0

At iteration 4, best cost: 51.0

At iteration 5, best cost: 51.0

At iteration 6, best cost: 51.0
```

```
At iteration 7, best cost: 51.0
At iteration 8, best cost: 47.0
At iteration 9, best cost: 49.0
At iteration 10, best cost: 48.0
At iteration 11, best cost: 47.0
At iteration 12, best cost: 44.0
At iteration 13, best cost: 44.0
At iteration 14, best cost: 47.0
At iteration 15, best cost: 44.0
At iteration 16, best cost: 47.0
At iteration 17, best cost: 47.0
At iteration 18, best cost: 46.0
At iteration 19, best cost: 46.0
At iteration 20, best cost: 46.0
At iteration 21, best cost: 44.0
At iteration 22, best cost: 44.0
At iteration 23, best cost: 44.0
At iteration 24, best cost: 44.0
At iteration 25, best cost: 44.0
At iteration 26, best cost: 44.0
At iteration 27, best cost: 44.0
At iteration 28, best cost: 44.0
At iteration 29, best cost: 44.0
Roads: 15, 36, 30, 31, 41, 29, 10, 4, 43 with cost: 44.0
Full Path:
(3, 4, 7)
(8, 0, 6)
(6, 7, 8)
(7, 1, 4)
(9, 5, 1)
(6, 3, 5)
```

```
(2, 0, 5)
(1, 2, 4)
(9, 8, 4)
```