# 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  
 :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)