Assigment1

1.1

For BFS algorithm, the operation of list should follow the principle of first in first out. According to the requirements of the topic, the optimal path is searched according to the width-first search algorithm. The starting point is S and the ending point is G. Firstly, the child nodes of S are searched and S' s children A, B and C are sent to the queue frontier in order. Then S is marked as the visited node and S is set as the parent node of its child node. Next, take out the child node of the first element in the queue, mark it as accessing node, and judge whether there is a target node in the child node. If there is one, it indicates that the best path is found according to the width-first algorithm. If not, it will queue the node that is not accessed in the child node and not in the frontier. Then repeat the above steps until the target is found. The elements in specific lists change as follows:

Visit_number	Visit-Order	Frontier	Visited	Farther
0	S	ABC	S	{S:None,A:S,B:S,C:S}
1	A	BCD	SA	{S:None,A:S,B:S,C:S, D:A}
2	B	CDF	SAB	{S:None,A:S,B:S,C:S, D:A,F:B}
3	C	DFE	SABC	{S:None,A:S,B:S,C:S, D:A,F:B,E:C}
4	D	FEG	SABCD	{S:None,A:S,B:S,C:S, D:A,F:B,E:C,G:D}
5	F	EG	SABCDF	{S:None,A:S,B:S,C:S, D:A,F:B,E:C,G:D}
6	E	G	ABCDFE	{S:None,A:S,B:S,C:S, D:A,F:B,E:C,G:D}
7	G	Empty	ABCDFE	{S:None,A:S,B:S,C:S, D:A,F:B,E:C,G:D}
Solution:S->A->D->G Cost:48				

Using python, the BFS algorithm is designed for the problem. After the program runs, the output is as follows:

```
['s', 'a', 'b', 'c', 'd', 'f', 'e', 'g']
{'s': None, 'a': 's', 'b': 's', 'c': 's', 'd': 'a', 'f': 'b', 'e': 'c', 'g': 'd'}
```

The first tuple is the visit order of each point under the breadth-first search algorithm, and the second dictionary records the parent node of each node. According to the parent node information, we can find the last node, and then we can extract the result path: the parent node of G is d, the parent node of D is a, and the parent node of a is s, that is, the path is s - > A - > D - > G. Cost is calculated to be 48.

Code for Question 1.1

1.2

For DFS algorithm, we should adopt the principle of last-in-first-out of stack. The elements in specific lists change as follows:

Visit_number	Visit-Order	Frontier	Visited	Farther
0	S	ABC	S	{S:None, A:S, B:S, C:S}
1	C	ABE	SC	{S:None, A:S, B:S, C:S, E:C}
2	E	ABFG	SCE	{S:None, A:S, B:S, C:S, E:C, F:E, G:E}
3	G	ABF	SCE	{S:None, A:S, B:S, C:S, E:C, F:E, G:E}
4	F	ABDG	SCEF	{S:None, A:S, B:S, C:S, E:C, F:E, G:F, D:F}
5	G	ABD	SCEF	{S:None, A:S, B:S, C:S, E:C, F:E, G:F, D:F}
6	D	ABG	SCEFD	{S:None, A:S, B:S, C:S, E:C, F:E, G:D, D:F}
7	G	AB	SCEFD	{S:None, A:S, B:S, C:S, E:C, F:E, G:D, D:F}
8	B	A	SCEFDB	{S:None, A:S, B:S, C:S, E:C, F:E, G:D, D:F}
9	A	Empty	SCEFDB	{S:None, A:S, B:S, C:S, E:C, F:E, G:D, D:F}
Solution1:S->C->E->G Cost:52 Solution2:S->C->E->F->G Cost:38 Solution3:S->C->E->F->D->G Cost:81				

Through the analysis of DFS algorithm, the solution of three paths is obtained, and the most economical solution is S->C->E->F->G, Cost=38.

Similarly, the DFS algorithm program is designed to solve the problem. After running, the output of the DFS algorithm program is as follows:

```
['s', 'c', 'e', 'g', 'f', 'g', 'd', 'g', 'b', 'a']
{'s': None, 'a': 's', 'b': 's', 'c': 's', 'e': 'c', 'f': 'e', 'g': 'd', 'd': 'f'}
```

The first tuple is the visit order of each point under the depth-first search algorithm, and the second dictionary records the parent node of each node. According to the parent node information, we can find the last node, and then we can extract the result path: the parent node of G is e, the parent node of E is c, and the parent node of C is s, that is, the path is s - > C - > e - > G. Cost is calculated to be 52.

Code for Question 1.2

1.3

Task: from S to G.
[1]. {[S,0]}
[2]. {[S→A,8]},{[S→B,5]},{[S→C,10]}
[3]. {[S→A,8]},{[S→B→F,22]},{[S→C,10]}
[4]. {[S→A→D,17]},{[S→B→F,22]},{[S→C,10]}
[5]. {[S→A→D,17]},{[S→B→F,22]},{[S→C→E,23]}
[6]. {[S→A→D→G,48]},{[S→B→F,22]},{[S→C→E,23]}
[7]. {[S→B→F→G,26]},{[S→C→E,23]}, {[S→B→F→E,33]}, {[S→B→F→D,38]}
[8]. {[S→C→E→F,34]}, {[S→B→F→E,33]}, {[S→B→F→D,38]}, {[S→C→E→G,52]}
[9]. {[S→C→E→F,34]}, {[S→B→F→E→G,62]}, {[S→B→F→D,38]}
[10]. {[S→C→E→F→B,51]}, {[S→C→E→F→D,50]}, {[S→C→E→F→G,38]}, {[S→B→F→D,38]}
[11]. {[S→C→E→F→B,51]}, {[S→C→E→F→D,50]}, {[S→B→F→D→G,69]}
[12]. {[S→C→E→F→B,51]}, {[S→C→E→F→D→G,81]}
[13]. {[S→C→E→F→B→A→D→G,115]}

The shortest path is S→B→F→G, the cost is 26.

Because Uniform-Cost Search is a simplification of Dijktra algorithm, Dijktra algorithm is directly designed to solve the problem under sufficient memory conditions.

After the program runs, the output is:

```
['s', 'b', 'a', 'c', 'd', 'f', 'e', 'g', 'g']
{'s': 0, 'a': 8, 'b': 5, 'c': 10, 'd': 17, 'e': 23, 'f': 22, 'g': 26}
{'s': None, 'a': 's', 'b': 's', 'c': 's', 'f': 'b', 'd': 'a', 'e': 'c', 'g': 'f'}
```

The first tuple is the visit order of each point under the depth-first search algorithm. The second dictionary records the shortest distance from the starting point to each node. The third dictionary records the parent node of each node. According to the information of the parent node, we can find the last node, and then we can extract the result path: the parent node of G is f, the parent node of F is b, and the parent node of B is s, that is, the path is s - > b - > F - > G. Cost was calculated to be 26.

Code for Question 1.3

2.1

Number the city, 0, A, B, C, D, E correspond to 0, 1, 2, 3, 4, 5, respectively.

A complete and legal path is assumed to be a chromosome. For example, [0,1,2,3,4,5] or [0,2,4,3,1,5], the initialization population can be composed of 50 such array sequences, i.e. chromosomes. Set the crossover rate to 0.7 or the mutation rate to 0.02, and check the results 100 times. Initialize the genetic operator and run the main function as follows:

```
    def __init__(self, aLifeCount=50, ):
        self.initCitys()
        self.lifeCount = aLifeCount
        self.ga = GA(aCrossRate=0.7,
                     aMutationRage=0.02,
                     aLifeCount=self.lifeCount,
                     aGeneLenght=len(self.citys),
                     aMatchFun=self.matchFun())
    def run(self, n):
        distance_list = []
        generate = [index for index in range(1, n + 1)]
        while n > 0:
            self.ga.next()
            distance = self.distance(self.ga.best.gene)
            distance_list.append(distance)
            n -= 1
        print(("Generation %d: Minimum Distance is %f") % (self.ga.generation,
distance))
        print('Current Optimal Route:')
        string = ''
        for index in self.ga.best.gene:
            string += str(index) + '->'
        print(string[0:len(string)-2])
```

2.2

In TSP, the shorter the distance, the better. The fitness function can be set to the reciprocal of the total distance, 1/distance. The distance between two points that are not directly connected is $d = +\infty$, For example $d_{AB} = +\infty$.

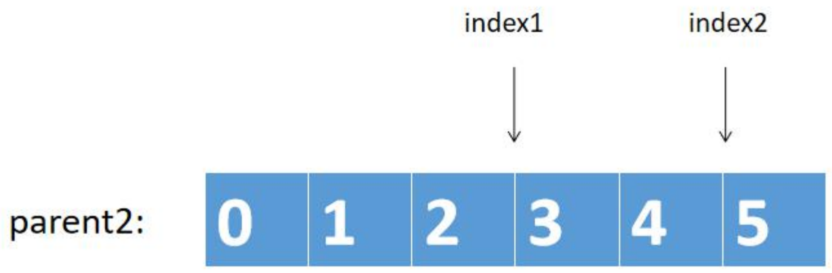
The initialization distance list and fitness calculation function are as follows:

```
    def initCitys(self):
        self.citys = [0,1,2,3,4,5]
        self.dist = [
            [0, 5, 3, math.inf, math.inf, math.inf],
            [5, 0, math.inf, 8, 14, 17],
            [3, math.inf, 0, 16, 11, math.inf],
            [math.inf, 8, 16, 0, math.inf, 9],
            [math.inf, 14, 11, math.inf, 0, 10],
            [math.inf, 17, math.inf, 9, 10, 0]
         ]
    def matchFun(self):
        return lambda life: 1.0 / self.distance(life.gene)
```

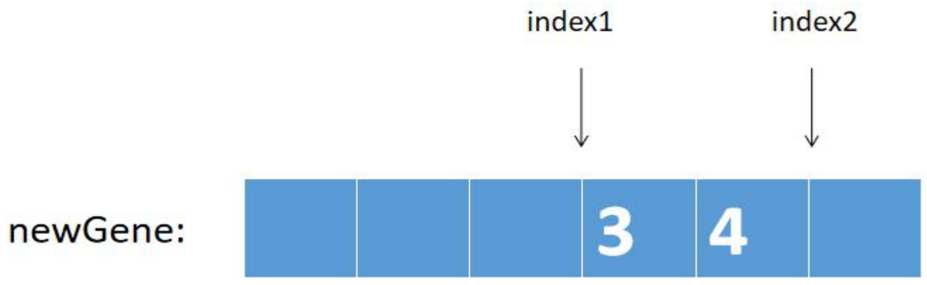
2.3

I use Order Crossover operator to solve this problem. For example, consider the sequence intersection of parent2 [0, 1, 2, 3, 4, 5] and parent1 [5, 2, 1, 3, 4, 0].

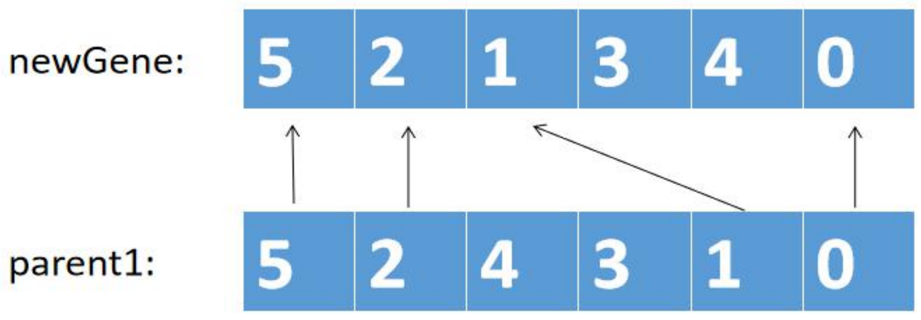
Firstly, the starting and ending positions of several genes in parent2 of a chromosome were randomly selected.



Then a progeny is generated and the location of the selected gene in the progeny is ensured to be the same as that of the parent.



Finally, another parent parent 1 is traversed, and the different genes from tempGene (Selected genes) are sequenced into the new offspring:



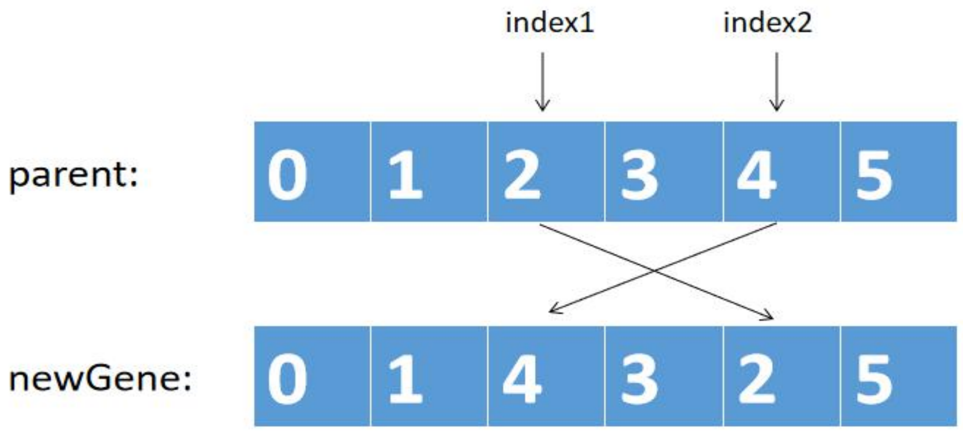
In this way, a new generation is formed —— child[5, 2, 1, 3, 4, 0]. The Python language algorithm is designed as follows:

```
def cross(self, parent1, parent2):
index1 = random.randint(0, self.geneLenght - 1)
index2 = random.randint(index1, self.geneLenght - 1)
tempGene = parent2.gene[index1:index2]
newGene = []
p1len = 0
for g in parent1.gene:
    if p1len == index1:
        newGene.extend(tempGene)
        p1len += 1
    if g not in tempGene:
        newGene.append(g)
        p1len += 1
self.crossCount += 1
return newGene
```

It has been proved that the most economical path can be found within 10 generations (genetic algorithm is uncertain), i.e. A - > C - > E - > D - > B - > 0, and the total route is 46.

2.4

I choose random commutation for mutation operator. For example, given a parent [0, 1, 2, 3, 4, 5], two genes are randomly exchanged to obtain offspring:



In this way, a new generation of children [0, 1, 4, 3, 2, 5] is formed. The Python language algorithm is designed as follows:

```
def mutation(self, gene):
    index1 = random.randint(0, self.geneLenght - 1)
    index2 = random.randint(0, self.geneLenght - 1)
    newGene = gene[:]
    newGene[index1], newGene[index2] = newGene[index2], newGene[index1]
    self.mutationCount += 1
    return newGene
```

It has been proved that the most economical path can be found within 10 generations, i.e. A - > C - > E - > D - > B - > 0, and the total route is 46.

Complete Code for Question 2