TSP 최적화를 위한 유전 알고리즘

22/11/26 / 2017156034 전상민

1 분석 개요

- TSP 문제: NP-hard 문제. 따라서 Heuristic 하게 최적화 문제로 접근.
- 잘 알려진 알고리즘 중 하나가 Genetic Algorithm.

2 제조데이터 소개

2.1 데이터 수집 방법

- (선택) TSP 문제 데이터를 직접 생성: rand 함수 사용. 노드 30 개.
- (선택) TSP 문제로 해결할 수 있는 실제 문제의 데이터 수집: ATT48 - 미국 수도 48 개 좌표.
  http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

2.2 데이터 유형/구조: x 좌표값, y 좌표값

2.3 데이터 품질 전처리

- 생성 데이터는 균등분포된 세자리 수치값이어 충분히 정제되었다고 볼 수 있음.
- 수집한 ATT48 데이터는 정해진 단순한 수치이기 때문에 아날로그 수치보다 전처리 할 여유공간이 적음.
  - ATT48 의 좌표값들이 수천대여서 크기를 줄이려고 생각해 보았으나, 결과값의 큰 차이가 없었음. – Fitness 를 백분율로 계산하기 때문에 평균과 분산의 차이는 없음.
  -

3 알고리즘 설계

3.1 개요

3.1.1 최적화 문제 수식화

3.1.1.1 목적함수

c_{ij}: i부터 j까지 거리.

y_{ij}: 결정변수

$$y_{ij} = \begin{cases} 1, & \text{if city } j \text{ is visited immediately after city } i \\ 0, & otherwise \end{cases}$$

$$\min \sum_i \sum_j c_{ij} y_{ij}$$

(노드 i와 j가 연결될 때만 1 을 곱하고, 아니면 0 을 곱하여 값을 처리하지 않음.

→ 연결될 수 있는 모든 가짓수에서 처음부터 순서대로 쭉 연결된 노드만 보겠다.)

3.1.1.2 제약함수

- 한 도시로만 가야 하므로

$$\sum_j y_{ij} = 1, i = 0, 1, \dots, n-1$$

- 한 도시에서만 왔어야 하므로

$$\sum_i y_{ij} = 1, i = 0, 1, \dots, n-1$$

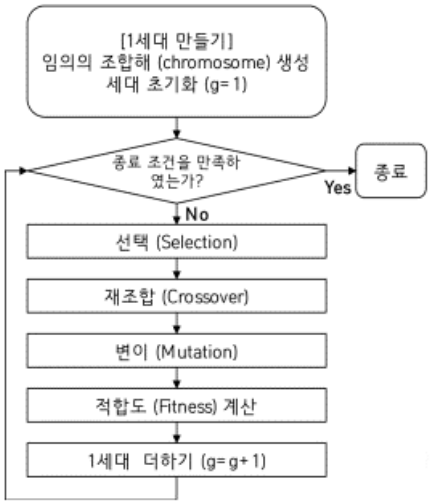
- 도시 전체가 한 번씩 연결되야 하므로(subtour 가 없어야 하므로)

$$\sum_i \sum_j y_{ij} \leq |S| - 1 \quad S \subset V, 2 \leq |S| \leq n-2$$

where S is the set of all tours of $G = (V, E)$

(연결된 선(E)의 개수가 노드(V)의 개수보다 작으면서, 모든 노드를 한 번만 연결한 수 보다 커야 한다. → 모든 V 를 한 번씩 겹치지 않게 선을 연결해야 한다. $(Path)$)

3.1.2 순서도



3.1.3 알고리즘 결정

(후에 설명)

- Selection: Roulette Wheel + Elitism
- Crossover: Ordered Crossover
- Mutation: Swap Mutation

3.2 GA 구축

3.2.1 데이터 형식: [(x 좌표, y 좌표), 거리]

```
class City:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, city):
        xDis = abs(self.x - city.x)
        yDis = abs(self.y - city.y)
        distance = np.sqrt((xDis ** 2) + (yDis ** 2))
        return distance

    def __repr__(self):
        return "(" + str(self.x) + "," + str(self.y) + ")"
```

3.2.2 Set Fitness: 1/거리 – 목적이 거리를 좁히는 것이므로 1/x 로 결정.

```
    def routeDistance(self):
        if self.distance ==0:
            pathDistance = 0
            for i in range(0, len(self.route)):
                fromCity = self.route[i]
                toCity = None
                if i + 1 < len(self.route):
                    toCity = self.route[i + 1]
                else:
                    toCity = self.route[0]
                pathDistance += fromCity.distance(toCity)
            self.distance = pathDistance
        return self.distance

    def routeFitness(self):
        if self.fitness == 0:
            self.fitness = 1 / float(self.routeDistance())
        return self.fitness
```

3.2.3 Set Population

3.2.3.1 경로 알고리즘: 노드를 랜덤으로 선택.

```
def createRoute(cityList):
    route = random.sample(cityList, len(cityList))
    return route
```

3.2.3.2 Population: 경로 여러개 생성.

```
def initialPopulation(popSize, cityList):
    population = []

    for i in range(0, popSize):
        population.append(createRoute(cityList))
    return population
```

3.2.4 Fitness 적용: 등수 가리기.

```
def rankRoutes(population):
    fitnessResults = {}
    for i in range(0,len(population)):
        fitnessResults[i] = Fitness(population[i]).routeFitness()
    return sorted(fitnessResults.items(), key = operator.itemgetter(1), reverse = True)
```

3.2.5 Set Selection Algo: Roulette Wheel + Elitism

- 기본적 Roulette Wheel 방법 사용. 그러나 이만으로는 결과가 좋지 않게 나왔다.
- ATT48 의 경우, 미국 동북부 지역은 작은 주들이 밀집되어 있다. 이후에 단순히 crossover 하고 mutation 을 한다면, 명백히 최소길이를 가진 가까운 노드들끼리 연결되어 있다가, 매우 먼 노드와 연결이 되어 fitness 값의 변동이 심하다.
- 따라서 Elitism 방법도 추가한다.

```
def selection(popRanked, eliteSize):
    selectionResults = []
    df = pd.DataFrame(np.array(popRanked), columns=["Index","Fitness"])
    df['cum_sum'] = df.Fitness.cumsum()
    df['cum_perc'] = 100*df.cum_sum/df.Fitness.sum()

    for i in range(0, eliteSize):
        selectionResults.append(popRanked[i][0])
    for i in range(0, len(popRanked) - eliteSize):
        pick = 100*random.random()
        for i in range(0, len(popRanked)):
            if pick <= df.iat[i,3]:
                selectionResults.append(popRanked[i][0])
                break
    return selectionResults
```

3.2.6 Set Crossover algo: Ordered crossover.

- TSP 는 모든 노드를 한 번씩 도므로 중복이 되면 안 된다. 따라서 Ordered crossover 를 이용.
    - Ordered crossover: A 구간을 B 에 복사하고 남은 걸 그 순서대로 채움

```
def breed(parent1, parent2):
    child = []
    childP1 = []
    childP2 = []

    geneA = int(random.random() * len(parent1))
    geneB = int(random.random() * len(parent1))

    startGene = min(geneA, geneB)
    endGene = max(geneA, geneB)

    for i in range(startGene, endGene):
        childP1.append(parent1[i])

    childP2 = [item for item in parent2 if item not in childP1]

    #child = childP1 + childP2
    child = childP2[:startGene] + childP1 + childP2[startGene:]
    return child
```

- Elitism 으로 순위가 높은 노드들은 놔두고, 나머지를 crossover.

```
def breedPopulation(matingpool, eliteSize):
    children = []
    length = len(matingpool) - eliteSize
    pool = random.sample(matingpool, len(matingpool))

    for i in range(0,eliteSize):
        children.append(matingpool[i])

    for i in range(0, length):
        child = breed(pool[i], pool[len(matingpool)-i-1])
        children.append(child)
    return children
```

3.2.7 Set Mutation algo: Swap mutation

- 모든 노드를 사용해야 해서 (노드가 하나라도 없으면 안 돼서) Swap mutation 이용.
    - Swap mutation: A 와 B 의 한 노드를 교환

```
def mutate(individual, mutationRate):
    for swapped in range(len(individual)):
        if(random.random() < mutationRate):
            swapWith = int(random.random() * len(individual))

            city1 = individual[swapped]
            city2 = individual[swapWith]

            individual[swapped] = city2
            individual[swapWith] = city1
    return individual
```

3.2.8 Generation loop: (ranking →) Selection → Crossover → Mutation

```
def nextGeneration(currentGen, eliteSize, mutationRate):
    popRanked = rankRoutes(currentGen)
    selectionResults = selection(popRanked, eliteSize)
    matingpool = matingPool(currentGen, selectionResults)
    children = breedPopulation(matingpool, eliteSize)
    nextGeneration = mutatePopulation(children, mutationRate)
    return nextGeneration
```

3.2.9 종료 조건: 세대 수

- ATT48 데이터셋은 계산된 해답이 있어서, 그 값의 근사할 때 종료하게 만들 수도 있었으나, 개인용 컴퓨터에 부하를 가늠하지 못하여 세대 수를 종료 조건으로 결정.

```
def geneticAlgorithm(population, popSize, eliteSize, mutationRate, generations):
    pop = initialPopulation(popSize, population)
    print("Initial distance: " + str(1 / rankRoutes(pop)[0][1]))

    for i in range(0, generations):
        pop = nextGeneration(pop, eliteSize, mutationRate)

    print("Final distance: " + str(1 / rankRoutes(pop)[0][1]))
    bestRouteIndex = rankRoutes(pop)[0][0]
    bestRoute = pop[bestRouteIndex]
    return bestRoute
```

3.3 데이터 생성

3.3.1 직접 생성: (0~200, 0~200)

```
cityList = []
for i in range(0,30):
    cityList.append(City(x=int(random.random() * 200), y=int(random.random() * 200)))
```

```
[(187,132), (93,38), (40,192), (46,20), (136,119), (52,86), (186,51), (151,115), (106,96),
(5,1), (193,95), (191,130), (158,100), (32,44), (106,72), (198,69), (174,112), (22,131),
(90,192), (141,144), (13,2), (37,103), (64,189), (182,173), (132,5), (123,81), (138,188),
(17,188), (92,70), (98,101)]
```

3.3.2 48 개 미국 수도 좌표[ATT48]

```
cityList = []
with open('att48_xy.txt') as f:
    for line in f:
        cityList.append(City(x=int(line.split()[0]), y=int(line.split()[1])))
```

```
[(6734,1453), (2233,10), (5530,1424), (401,841), (3082,1644), (7608,4458), (7573,3716),
(7265,1268), (6898,1885), (1112,2049), (5468,2606), (5989,2873), (4706,2674), (4612,2035),
(6347,2683), (6107,669), (7611,5184), (7462,3590), (7732,4723), (5900,3561), (4483,3369),
(6101,1110), (5199,2182), (1633,2809), (4307,2322), (675,1006), (7555,4819), (7541,3981),
(3177,756), (7352,4506), (7545,2801), (3245,3305), (6426,3173), (4608,1198), (23,2216),
(7248,3779), (7762,4595), (7392,2244), (3484,2829), (6271,2135), (4985,140), (1916,1569),
(7280,4899), (7509,3239), (10,2676), (6807,2993), (5185,3258), (3023,1942)]
```

3.4 결과 시각화

3.4.1 Fitness 그래프

```
def geneticAlgorithmPlot(population, popSize, eliteSize, mutationRate, generations):
    pop = initialPopulation(popSize, population)
    progress = []
    progress.append(1 / rankRoutes(pop)[0][1])

    for i in range(0, generations):
        pop = nextGeneration(pop, eliteSize, mutationRate)
        progress.append(1 / rankRoutes(pop)[0][1])

    plt.plot(progress)
    plt.ylabel('Distance')
    plt.xlabel('Generation')
    plt.show()

    ###
    showMap(pop[-1])
```

3.4.2 지도

```
def showMap(cityList):
    print(cityList)
    prev=City(0,0)
    for i in cityList:
        plt.plot(i.x, i.y,'ro')
        plt.plot(prev.x,prev.y, 'k-')
        if(prev.x == 0 and prev.y == 0):
            prev=i
            continue;
        else:
            plt.plot([prev.x,i.x],[prev.y, i.y],'k-')
            #print('prev x:', prev.x, ' prev y:', prev.y,' i x:', i.x, ' i y:',
            prev=i
    plt.show()
```

3.4.3 루트 순서

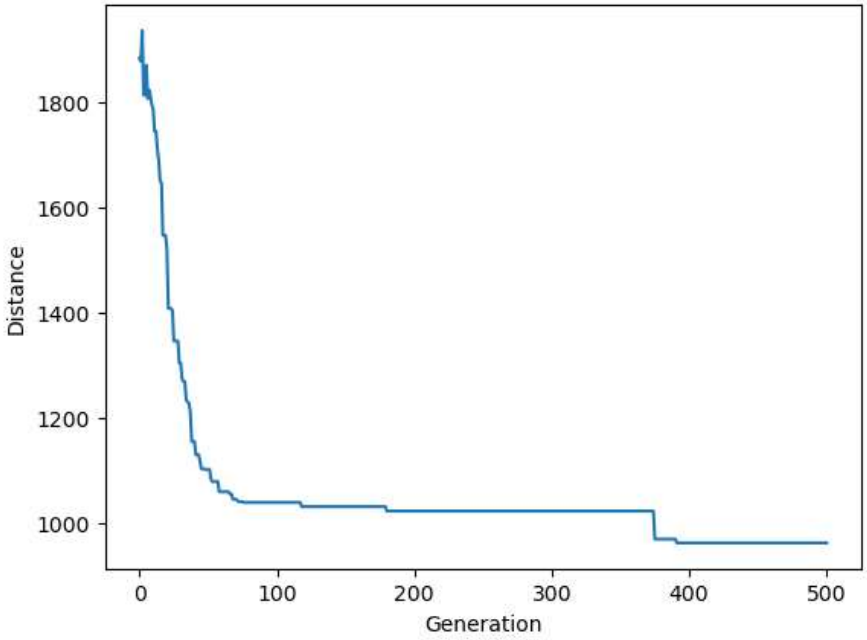
```
def findIndex(cityResult):
    cityResultIndex=[]
    for ix in cityResult:
        cityResultIndex.append(cityList.index(ix))
    print(cityResultIndex)
```

4 알고리즘 실행/hyperparameter 튜닝

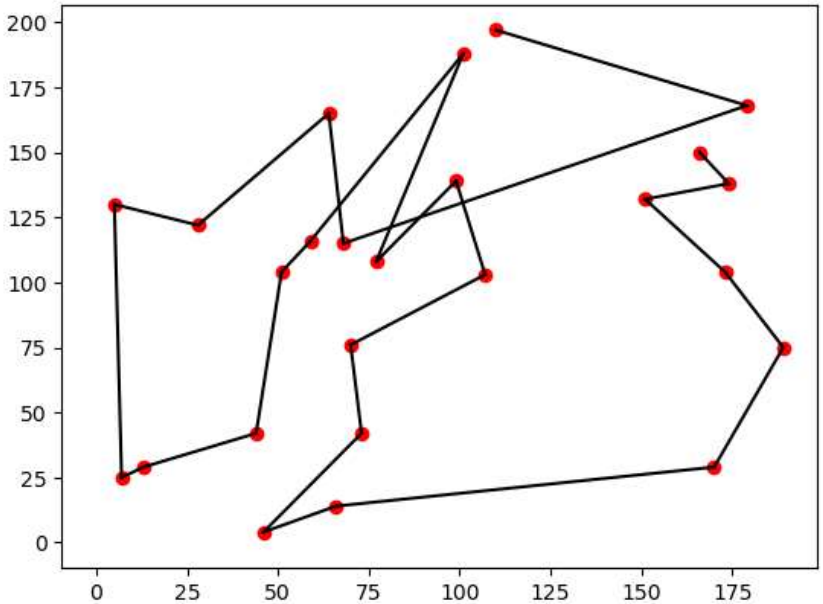
4.1 1 차 시도

- 랜덤 데이터

```
geneticAlgorithmPlot(population=cityList, popSize=100, eliteSize=20, mutationRate=0.01, generations=500)
```

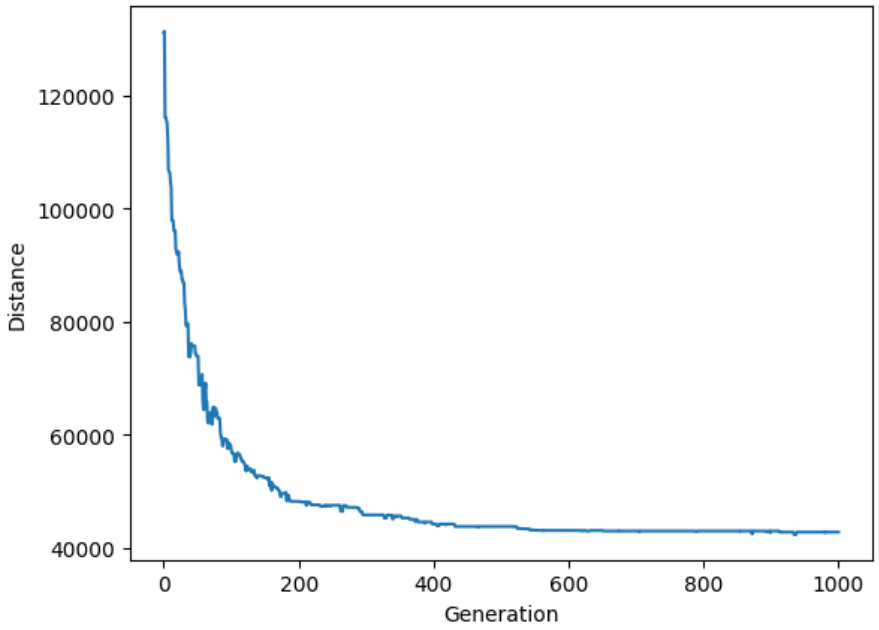



```
[(110,197), (179,168), (68,115), (64,165), (28,122), (5,130), (7,25), (13,29), (44,42), (51,104), (59,116
8), (99,139), (107,103), (70,76), (73,42), (46,4), (66,14), (170,29), (189,75), (173,104), (151,132), (17
```

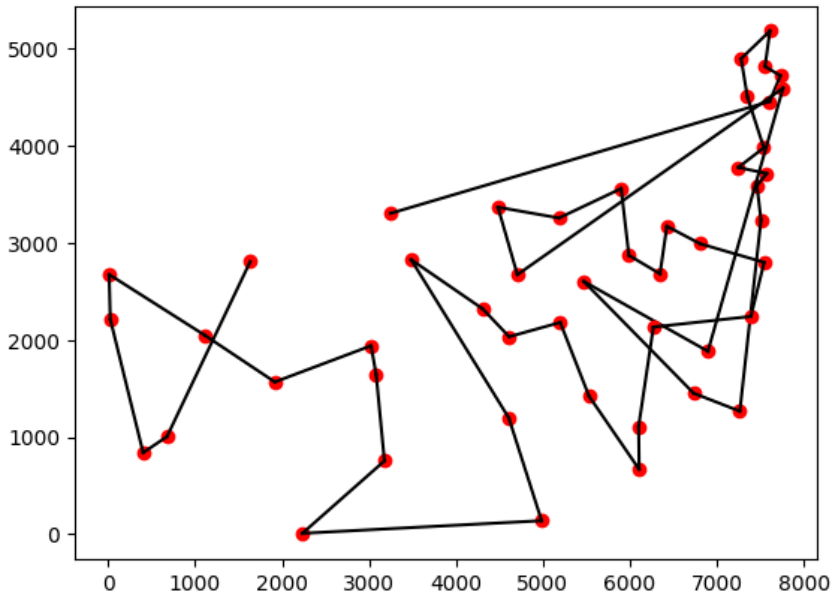


- 미국 44 개 수도 [ATT48]

```
geneticAlgorithmPlot(population=cityList, popSize=100, eliteSize=20, mutationRate=0.01, generations=1000)
```




```
[(3245,3305), (7608,4458), (7732,4723), (7555,4819), (7611,5184), (7280,4899), (7352,4506), (7541,3981), (7
6), (7462,3590), (7509,3239), (7265,1268), (6734,1453), (5468,2606), (6898,1885), (7762,4595), (4706,2674),
3258), (5900,3561), (5989,2873), (6347,2683), (6426,3173), (6807,2993), (7545,2801), (7392,2244), (6271,213
07,669), (5530,1424), (5199,2182), (4612,2035), (4307,2322), (3484,2829), (4608,1198), (4985,140), (2233,10
2,1644), (3023,1942), (1916,1569), (1112,2049), (10,2676), (23,2216), (401,841), (675,1006), (1633,2809)]
```



4.2 Hyperparameter 튜닝 [ATT48]

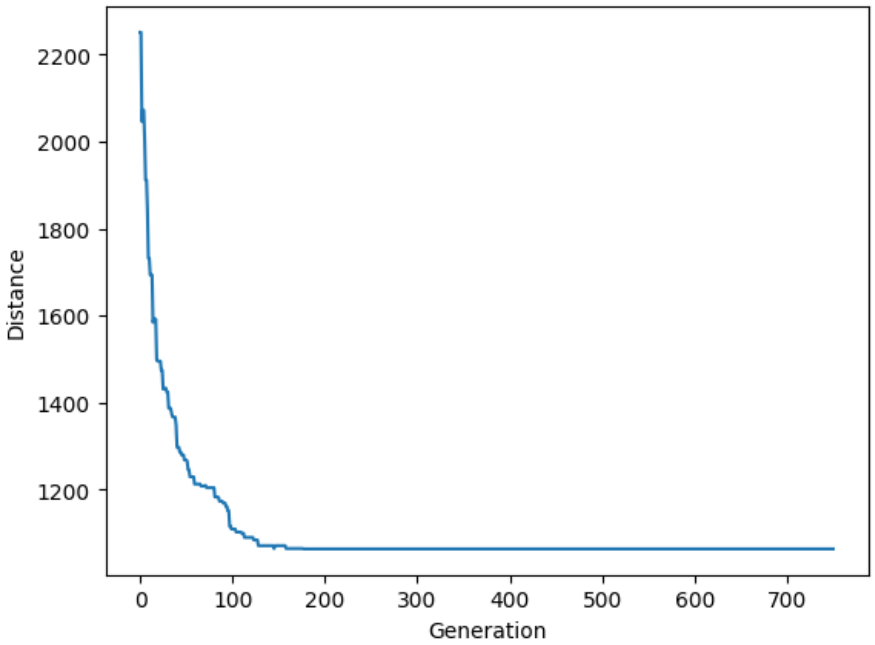
- 처음에는 감을 잡기위해 단체로 실행하지 않고 개별적으로 하나씩 조정해갔다. 그런데도 눈에 띄게 나아지지 않았다. 따라서 무작정 시도하는 것을 멈추고 수집 데이터와 결정한 알고리즘이 어떻게 작동하는지를 생각해 보았다. 그것과 test run 에서 발견한 여러 현상들과 연관하여 hyperparameter 를 조정하였다.
- 미국 동북부에는 작은 주들이 많이 밀집되어 있다. 이 노드들이 mutate 된다면 fitness 에 큰 손실이 생길 것이기에, 이들을 mutation 대상에서 제외하고 싶다.
    - ➔ Elitism 사용. Elite 노드 개수(eliteSize)를 그 지역의 수인 20~25 로 조정.
    - ➔ MutationRate 도 이에 따라 0.004~0.0045 로 조정.
        - 이가 유동적으로 조정이 되었으면 더 좋았을 듯싶다.
- 계산 수 및 속도를 줄이기 위하여 1000 대 값인 데이터 값을 10 으로 나눠 100 대로 줄이는 것을 시도하였다. 그러나 큰 차이가 없었다. 알고리즘 내 데이터는 인덱스로 다루고, 계산은 절대적인 크기가 줄었더라도 평균과 분산의 분포가 동일하기 때문이어, 큰 변화가 없던 것으로 생각한다.
- populationSize 는 컴퓨터 부하로 인해 100 으로 고정하였다.
- generation 은 주로 1000 대에 변화속도가 줄어들어 거기서 멈추기로 하였다.

5 결과

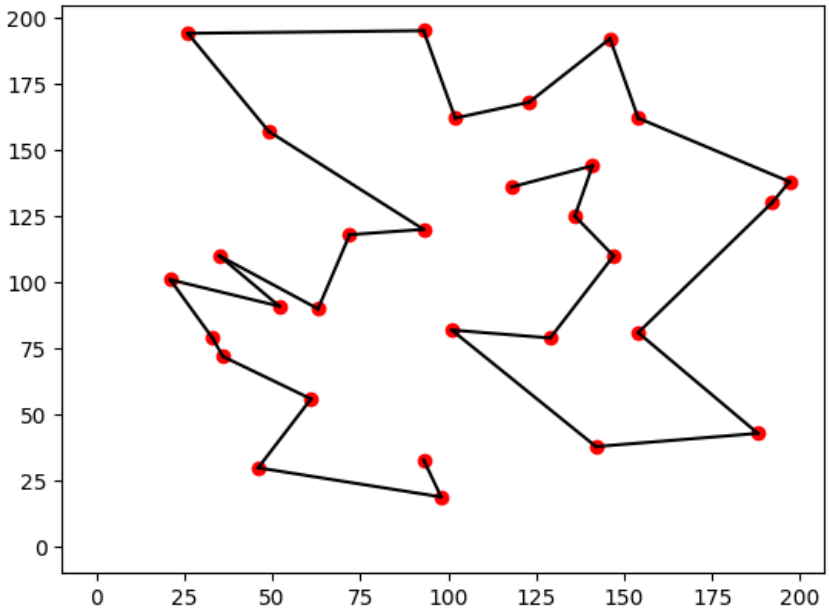
5.1 튜닝 후 결과 [랜덤 30 개]

```
runGA(population=cityList, popSize=100, eliteSize=10, mutationRate=0.005, generations=750)
```

```
Initial distance: 2251.265153718805
```




```
[(93,33), (98,19), (46,30), (61,56), (36,72), (33,79), (21,101), (52,91), (35,110), (63,90), (7
194), (93,195), (102,162), (123,168), (146,192), (154,162), (197,138), (192,130), (154,81), (18
79), (147,110), (136,125), (141,144), (118,136)]
```

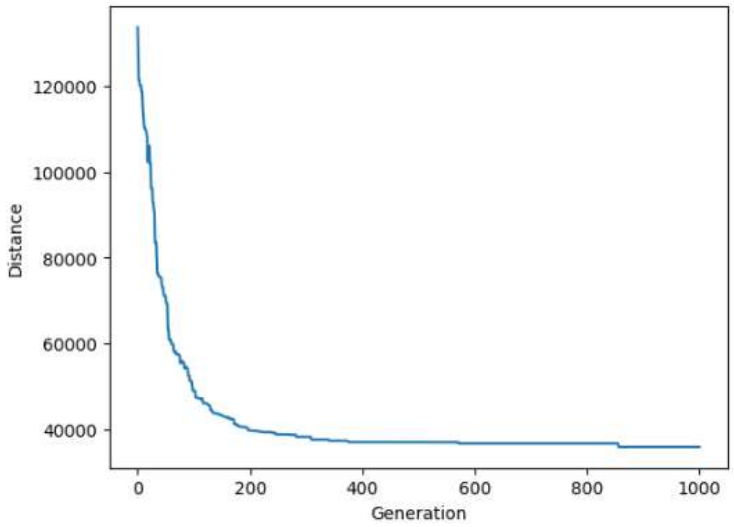



```
Final distance: 1063.1648394011152
```

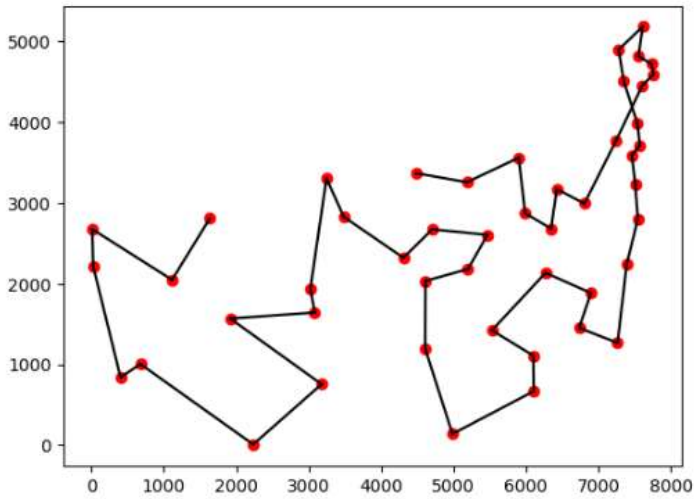
5.2 튜닝 후 결과 [ATT48] (minimal tour length = 33523)

```
cityResult=runGA(population=cityList, popSize=100, eliteSize=15, mutationRate=0.005, generations=1000)
findIndex(cityResult)
```

```
Initial distance: 133669.59384865308
```



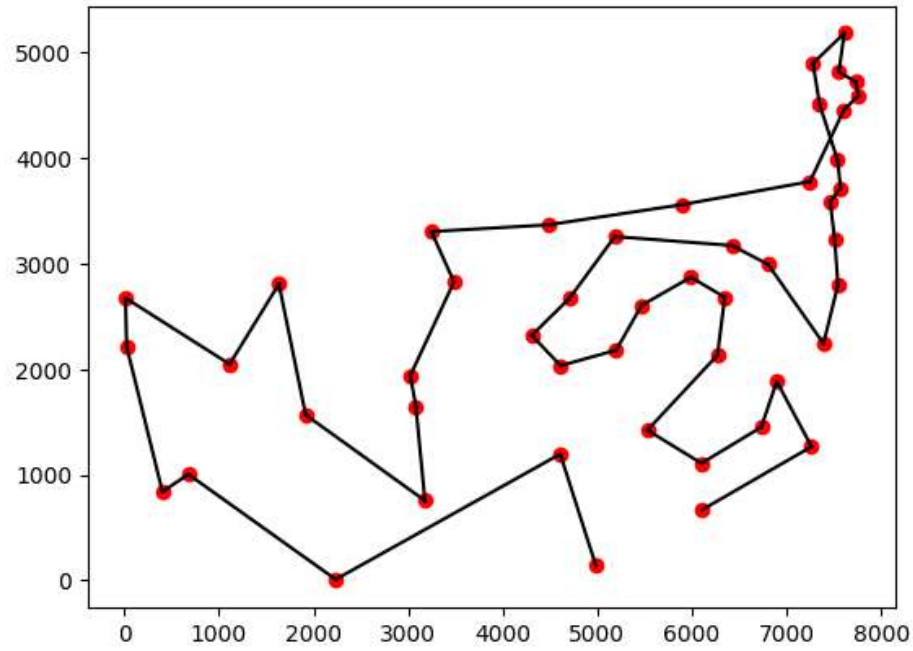

```
[(1633,2809), (1112,2049), (10,2676), (23,2216), (401,841), (675,1006), (2233,10), (3177,756), (1916,1569), (3082,1644), (3023,
1942), (3245,3305), (3484,2829), (4307,2322), (4706,2674), (5468,2606), (5199,2182), (4612,2035), (4608,1198), (4985,140), (610
7,669), (6101,1110), (5530,1424), (6271,2135), (6898,1885), (6734,1453), (7265,1268), (7392,2244), (7545,2801), (7509,3239), (7
462,3590), (7573,3716), (7541,3981), (7352,4506), (7280,4899), (7611,5184), (7555,4819), (7732,4723), (7762,4595), (7608,4458),
(7248,3779), (6807,2993), (6426,3173), (6347,2683), (5989,2873), (5900,3561), (5185,3258), (4483,3369)]
```



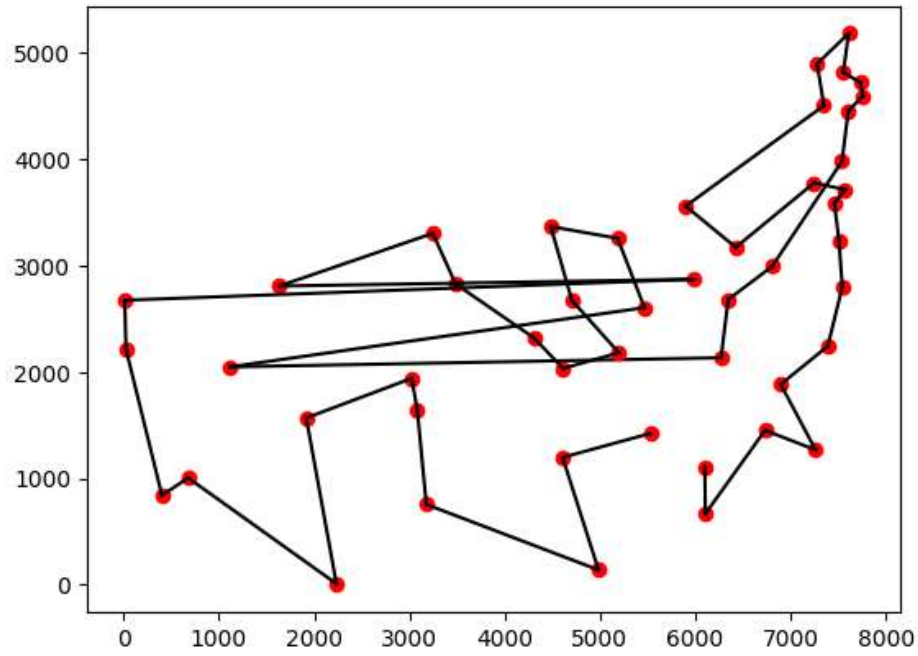

```
Final distance: 35895.62884448029
[23, 9, 44, 34, 3, 25, 1, 28, 41, 4, 47, 31, 38, 24, 12, 10, 22, 13, 33, 40, 15, 21, 2, 39, 8, 0, 7, 37, 30, 43, 17, 6, 27, 29,
42, 16, 26, 18, 36, 5, 35, 45, 32, 14, 11, 19, 46, 20]
```

- 최종계산거리/최초생성거리 = 26.85%
- 최종거리/해답(최소거리) = ×1.0708

- Runner-ups



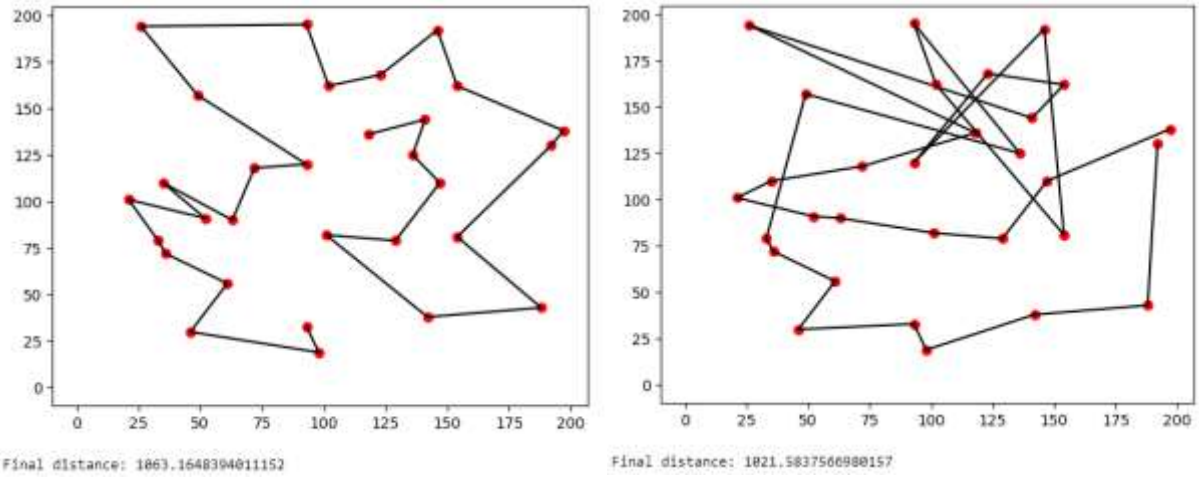

```
Final distance: 37532.68424377219
```




```
Final distance: 36321.98613496958
```

6 느낀점

- Hyperparameter 들을 고정값으로 주었다. 여기서 업그레이드를 하여, GA 내에서 피드백을 받으며 유동적으로 설계를 하면 더욱 좋은 결과가 나올 것으로 기대한다.
- 지도에서 그럴듯하게 보인다고 해도 그보다 난해한데 결과값은 좋은 경우들이 자자하다. (좌측: 1063/2251=47.2%, 우측: 1022/2285=44.7%)
  초반에는 결과를 수치를 보지 않고 insight 로 지도를 보면서 outlier 가 없는지를 확인했는데, 그건 믿을 것이 안된다. 역시 논리 > 통찰력.



7 References

https://optimization.mccormick.northwestern.edu/index.php/Traveling_salesman_problems

https://towardsdatascience.com/evolution-of-a-salesman-a-complete-genetic-algorithm-tutorial-for-python-6fe5d2b3ca35

http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

https://people.sc.fsu.edu/~jburkardt/datasets/tsp/tsp.html