IE260 Homework #4 Report

20150359 Chiyoon Park

1. Code Description

- Encoding Part

```
def listIntoDic(self, list):
    list = list
    dic = {}
    itr = 0
    current = 0
    while itr < len(list):
        dic[current] = list[current]
        current = list[current]
        itr = itr + 1
    return dic
```

```
def dicIntoPath(self, dic):
    dic = dic
    keys = dic.keys()
    lst = list(keys)
    lst.append(0)
    return lst
```

```
def pathIntoDic(self, list):
    lst = list
    dic = {}
    for i in range(len(lst)):
        if i == len(lst) - 1:
            break
        dic[lst[i]] = lst[i + 1]
    return dic
```

기본적으로 제공된 코드의 Encoding 방식을 그대로 사용하되, Crossover와 Mutataion 작업에서 생기는 오류를 컨트롤하기 위해서 위의 세 메소드를 디자인했습니다. 먼저 listIntoDic( ) 메소드의 경우 리스트의 형태로 얻어진 경로를 딕셔너리의 형태로 변환해주는 함수입니다. 이 함수의 경우 createInitialPopulation( ) 에서 처음 생성된 리스트 형태의 경로를 변환하기 위해서 설계했습니다.

두번째로 dicIntoPat( ) 메소드는 딕셔너리 형태의 경로를 path의 형태로 바꿔주는 것입니다. 여기서 path는 리스트의 형태로 처음 방문하는 도시부터 가장 마지막에 방문하는 도시를 그대로 적은 것입니다. 이 path가 crossover를 효율적으로 할 수 있게 하므로 이 메소드를 구상했습니다.

마지막으로 pathIntoDic( ) 메소드는 여러 연산에서 사용된 path를 다른 함수 (fitness( ), calculateTotalDistanc( ))에서 사용할 수 있도록 다시 Dictionary의 형태로 변환해주는 함수입니다.

예를 들어, createInitailPopulation( )을 통해서 [14, 2, 15, 6, 8, 4, 11, 17, 7, 3, 18, 0, 16, 1, 13, 19, 10, 12, 9, 5]라는 Genotype이 생성된 경우, 이를 listIntoDic( ) 함수를 사용하면 {0: 14, 14: 13, 13: 1, 1: 2, 2: 15, 15: 19, 19: 5, 5: 4, 4: 8, 8: 7, 7: 17, 17: 12, 12: 16, 16: 10, 10: 18, 18: 9, 9: 3, 3: 6, 6: 11, 11: 0}로 변환됩니다. 그리고 이를 다시 dicIntoList( )로 변환할 경우 [0, 14, 13, 1, 2, 15, 19, 5, 4, 8, 7, 17, 12, 16, 10, 18, 9, 3, 6, 11, 0]로 변환됩니다.

- Crossover part

```
def crossoverParents1(self, instance1, instance2):
    genotype1 = instance1.getGenotype()
    genotype2 = instance2.getGenotype()
    if type(genotype1) == list:
        genotype1 = self.listIntoDic(genotype1)
    if type(genotype2) == list:
        genotype2 = self.listIntoDic(genotype2)
    pathGeno1 = self.dicIntoPath(genotype1)
    pathGeno2 = self.dicIntoPath(genotype2)
    lenpath1 = len(pathGeno1)
    lenpath2 = len(pathGeno2)
    temp1 = [0] * lenpath1
    temp2 = [0] * lenpath2
    pathGeno1 = pathGeno1[1:len(pathGeno1) - 1]
    pathGeno2 = pathGeno2[1:len(pathGeno2) - 1]
    newInstance = GeneticAlgorithmInstance()
    crossInd = random.randint(0, len(pathGeno1) - 1)
    leftList1 = pathGeno1[0:crossInd]
    rightList1 = pathGeno1[crossInd:]
    leftList2 = pathGeno2[0:crossInd]
    rightList2 = pathGeno2[crossInd:]
    for i in range(len(rightList2)):
        if rightList2[i] in pathGeno1:
            common = rightList2[i]
            pathGeno1.remove(common)
    for i in range(len(rightList1)):
        if rightList1[i] in pathGeno2:
            common = rightList1[i]
            pathGeno2.remove(common)
    temp1[crossInd+1:len(temp1)-1] = rightList2
    temp1[1:crossInd+1] = pathGeno1
    temp2[crossInd+1:len(temp2)-1] = rightList1
    temp2[1:crossInd+1] = pathGeno2
    rand = random.randint(0,1)
    if rand == 0:
        dic = self.pathIntoDic(temp1)
    else:
        dic = self.pathIntoDic(temp2)
    newInstance.setGenotype(dic)
    return newInstance
```

crossoverParents1( ) 메소드는 일점 교차를 구현한 메소드입니다. 이 메소드에서 오류가 없고 편하게 구현하기 위해서 위해서 고안한 path를 사용합니다. 먼저 parameter로 두 instance를 받아서 그 genotype을 추출한 후 path의 형태로 변환합니다. 이후 어느 부위에서 교차가 일어날지 random.randint 메소드를 통해서 얻어냅니다. 그리고 이 위치를 기점으로 앞부분과 뒷부분으로 두 genotype을 교환합니다.

예를 들어서 [0,1,6,5,3,2,8,4,9,7,0]와 [0,3,7,6,1,9,4,8,2,5,0]가 두 path로 변환된 genotype인 경우, 먼저 앞에과 뒤에서 0을 제거합니다. 이를 통해서 [1,6,5,3,2,8,4,9,7]와 [3,7,6,1,9,4,8,2,5]로 변환됩니다. 이후 random.randint로 정수를 선택하여 교차할 부위를 결정합니다. 만약 3이 선택된 경우 [1,6,5] 와 [3,2,8,4,9,7], [3,7,6] 와 [1,9,4,8,2,5]로 두 genotype이 나누어집니다. 이후 뒷부분은 그대로 상대 genotype에 전달됩니다. 즉 [3,2,8,4,9,7]과 [1,9,4,8,2,5]는 그대로 반대로 전달됩니다. 이때 [1,9,4,8,2,5]가 전달되고 [1,6,5]와 그대로 붙을 경우 1과 5가 중복되게 되며, 부족한 도시도 생기면서 불완전한 genotype이 됩니다. 이를 해결하기 위해서 원래 genotype [1,6,5,3,2,8,4,9,7]에서 [1,9,4,8,2,5]와 중복되는 도시를 제거합니다. 그러면 [6,3,7]이 남게 됩니다. 따라서 이를 [1,9,4,8,2,5]와 결합하여 [6, 3, 7, 1, 9, 4, 8, 2, 5]로 교차가 됩니다. 상대 유전자도 같은 과정으로 [6, 1, 5, 3, 2, 8, 4, 9, 7]로 교차가 완료됩니다. 이후 다시 정상적인 path의 형태로 변환하기 위해서 앞뒤에 0을 붙여 [0, 6, 3, 7, 1, 9, 4, 8, 2, 5, 0]과 [0, 6, 1, 5, 3, 2, 8, 4, 9, 7, 0]으로 변환하고, 이를 pathIntoDic( ) 메소드를 통해서 dictionary의 형태로 변환하여 새로운 instance에 저장하여 반환합니다.

이 알고리즘을 통해서 교차를 통해 서로 유전자의 일부를 교환하면서, 원래 가지고 있었던 유전자의 배열을 최대한 살릴 수 있습니다.

```
def crossoverParents2(self, instance1, instance2):
    genotype1 = instance1.getGenotype()
    genotype2 = instance2.getGenotype()
    if type(genotype1) == list:
        genotype1 = self.listIntoDic(genotype1)
    if type(genotype2) == list:
        genotype2 = self.listIntoDic(genotype2)
    pathGeno1 = self.dicIntoPath(genotype1)
    pathGeno2 = self.dicIntoPath(genotype2)
    lenpath1 = len(pathGeno1)
    lenpath2 = len(pathGeno2)
    temp1 = [0] * lenpath1
    temp2 = [0] * lenpath2
    pathGeno1 = pathGeno1[1:len(pathGeno1) - 1]
    pathGeno2 = pathGeno2[1:len(pathGeno2) - 1]
    crossInd1 = -1
    crossInd2 = -1
    while crossInd1 == crossInd2:
        crossInd1 = random.randint(0, len(pathGeno1) - 1)
        crossInd2 = random.randint(0, len(pathGeno1) - 1)
    temp = [crossInd1, crossInd2]
    temp.sort()
    crossInd1 = temp[0]
    crossInd2 = temp[1]
    swap1 = pathGeno1[crossInd1:crossInd2 + 1]
    swap2 = pathGeno2[crossInd1:crossInd2 + 1]
    for i in range(len(swap2)):
        if swap2[i] in pathGeno1:
            common = swap2[i]
            pathGeno1.remove(common)
    for i in range(len(swap1)):
        if swap1[i] in pathGeno2:
            common = swap1[i]
            pathGeno2.remove(common)
    temp1[1:crossInd1 + 1] = pathGeno1[0:crossInd1]
    temp1[crossInd1 + 1:crossInd2 + 2] = swap2
    temp1[crossInd2 + 2:len(temp1)-1] = pathGeno1[crossInd1:]
    temp2[1:crossInd1 + 1] = pathGeno2[0:crossInd1]
    temp2[crossInd1 + 1:crossInd2 + 2] = swap1
    temp2[crossInd2 + 2:len(temp2)-1] = pathGeno2[crossInd1:]
    rand = random.randint(0, 1)
    if rand == 0:
        dic = self.pathIntoDic(temp1)
    else:
        dic = self.pathIntoDic(temp2)
    newInstance = GeneticAlgorithmInstance()
    newInstance.setGenotype(dic)
    return newInstance
```

crossoverParents1( ) 메소드는 이점 교차를 구현한 메소드입니다. 이 메소드에서 오류가 없고 편하게 구현하기 위해서 위해서 고안한 path를 사용합니다. 먼저 parameter로 두 instance를 받아서 그 genotype을 추출한 후 path의 형태로 변환합니다. 이후 어느 부위에서 교차가 일어날지 random.randint 메소드를 두 번 사용함을 통해서 얻어냅니다. 그리고 두 위치를 기준으로 서로의 유전자를 교환합니다.

예를 들어서 [0,1,6,5,3,2,8,4,9,7,0]와 [0,3,7,6,1,9,4,8,2,5,0]가 두 path 로 변환된 genotype 인 경우, 먼저 앞에과 뒤에서 0 을 제거합니다. 이를 통해서 [1,6,5,3,2,8,4,9,7]와 [3,7,6,1,9,4,8,2,5]로 변환됩니다. 이후 random.randint 로 두 정수를 선택하여 교차될 위치를 결정합니다. 여기서 while 문을 사용하여 두 정수가 다르게 하였으며, 두 정수를 크기 순으로 정렬하여 두 위치를 오류 없이 결정할 수 있도록 하였습니다. 이 과정에서 2 와 5 가 결정된 경우, [5,3,2,8]과 [6,1,9,4]가 상대 유전자에게 전달되는 교차 부위가 됩니다. 이때 위의 일점 교차와 마찬가지로 유전자가 전달되는 과정에서 중복되는 도시나 결손되는 도시가 생기게 되면서 오류가 생기게 됩니다. 예시로 [6,1,9,4]가 그대로 전달되는 경우, [1,6]과 [4,9,7]에서 중복과 결손을 일으키게 됩니다. 따라서 이를 해결하기 위해서 일점 교차와 동일하게 원래 유전자와 전달되는 유전자 사이에서 중복을 제거하여 줍니다. [1,6,5,3,2,8,4,9,7] 에서 [6,1,9,4]와 중복되는 유전자를 제거해줍니다. 그러면 [5,3,2,8,7]이 남게됩니다. 그러면 교차 위치 2 앞쪽에 두 도시가 들어가야 하므로 [5,3]이 교차부위 앞쪽에 들어가고, [2,8,7]이 교차부위 뒤에 들어가게 됩니다. 이를 통해서 최종적으로 [5,3,6,1,9,4,2,8,7]이 최종적인 genotype 이 됩니다. 같은 방법으로 상대 유전자의 교차를 진행하면, [5,3,2,8]가 교차되며, [3,7,6,1,9,4,8,2,5]에서 중복되는 유전자를 제거하여 [7,6,1,9,4]가 남게 되고 이를 slice 하여 교차가 일어나 최종적으로 [7,6,5,3,2,8,1,9,4]가 됩니다.

앞의 일점 교차의 경우 꽤 많은 부위가 그대로 전달되면서 자식 유전자가 부모 유전자와 유사하다는 특징이 있습니다. 이점 교차의 경우 일점 교차에 비해서 상대적으로 변이가 큽니다.

- Mutation part

```
def mutation1(self, instance, factor):
    genotype = instance.getGenotype()
    listgenotype = self.dicIntoPath(genotype)
    mutation = random.random()
    if mutation <= factor:
        mutationindex1 = -1
        mutationindex2 = -1
        while mutationindex1 == mutationindex2:
            mutationindex1 = random.randint(1, len(listgenotype) - 2)
            mutationindex2 = random.randint(1, len(listgenotype) - 2)
        mutationgene1 = listgenotype[mutationindex1]
        mutationgene2 = listgenotype[mutationindex2]
        listgenotype[mutationindex1] = mutationgene2
        listgenotype[mutationindex2] = mutationgene1
    temp = self.pathIntoDic(listgenotype)
    instance.setGenotype(temp)
```

이 mutation의 경우 정해진 확률을 통해서 일어날지 아닐지를 결정한 후 genotype의 두 도시를 교환하는 메소드입니다. 여기서 오류를 제거하고 편하게 메소드를 구현하기 위해서 path를 사용했습니다. Mutaiton의 경우 list의 형태의 경로가 아닌 항상 dictionary 형태의 경로만 입력받으므로, dicIntoPath( ) 메소드를 사용하여 path로 변환합니다. 이후 random.random( ) 메소드를 통해서 0과 1 사이의 숫자를 하나 생성한 후, mutation Factor와의 비교를 통해서 mutation의 진행 유무를 결정합니다. 이후 random.randint를 통해서 서로 바꿀 두 index를 결정합니다. While문을 통해서 두 인덱스가 동일하지 않게 합니다. 이후 서로의 유전자를 교환한 후 instance에 저장하여 return합니다.

- Subsitute part

```
def substitutePopulationNum(self, population, children, num):
    for itr1 in range(len(population)):
        for itr2 in range(itr1+1,len(population)):
            if self.fitness(population[itr1]) < self.fitness(population[itr2]):
                population[itr1], population[itr2] = population[itr2], population[itr1]

    childIndex = random.randint(0,len(children) - num)
    for i in range(num):
        population[-num+i] = children[childIndex + i]

    return population
```

Subsitution의 경우 기본적으로 제공된 메소드에서 큰 변화를 주진 않았습니다. 대신 num이라는 parameter를 통해서 population에 추가할 자손의 개수를 컨트롤 할 수 있게 했습니다. 이는 이후 performEvolution에서 사용되게 됩니다.

- Show Utillity Method

```
def showUtility(self):
    utilList = []
    utildomain = []
    for i in range(len(self.bestList)):
        utilList.append(self.fitness(self.bestList[i][0]))
        utildomain.append(self.bestList[i][1])
    t = range(0,len(utilList))
    x = utildomain
    y = [utilList[i] for i in t]
    plt.plot(x, y)
    plt.show()
```

ShowUtility 메소드는 performEvolution을 시행한 후 변화하는 경로의 utillity를 그래프의 형태로 보여주는 메소드입니다. 이를 통해서 변수들을 세팅하는데 사용하였습니다.

- Perform Evolution Method

```
mutation = mutationFactor + float((time.time()-startTime) / 90) * mutationFactor
self.mutation1(offsprings[itr], mutation)
popNum = int(float(( -num / 180) * (time.time()-startTime)) + num)
# print(popNum)
if popNum <= 1:
    popNum = 1
population = self.substitutePopulationNum(population, offsprings, popNum)
```

Perform Evolution Method 메소드에선 총 두 부분을 변경하였습니다. 먼저 mutation factor를 시간에 따라 변하게 하였습니다. 이후 서술하지만 crossoverParents1 메소드를 사용하는 후반의 경우 변이의 크기가 크지 않아 solution set이 좁혀지게 됩니다. 이를 해결하기 위해서 오히려 시간이 지날수록 mutation factor의 크기를 키우게 했습니다.

또 subsitution이 일어나는 자손의 수를 점점 줄이는 방법을 사용했습니다. 너무 많은 자손이 질 좋은 population을 대체하는 것을 막기 위해서 후반에 갈수록 population을 대체하는 자손이 적어지도록 설계했습니다.

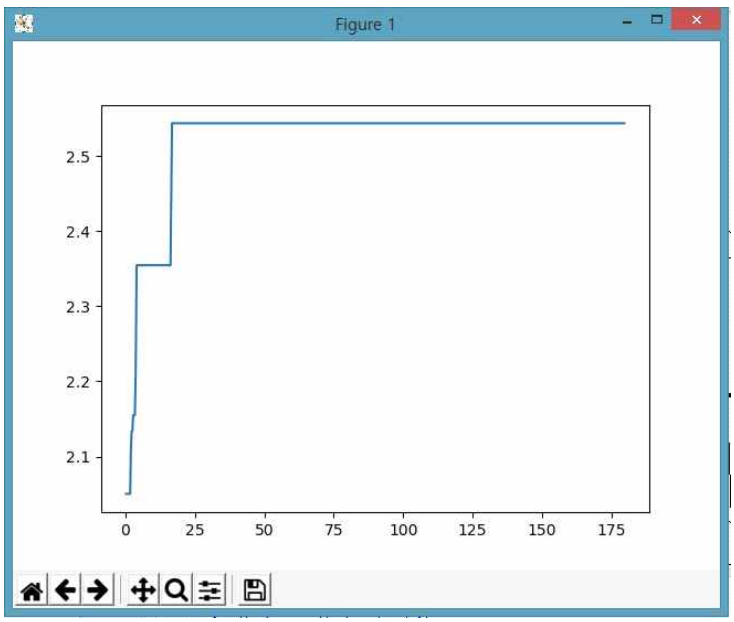
2. Parameter Experiment

먼저 crossover 단계에서 총 7가지 plan을 설계하였고, 같은 환경에서 이를 테스트하여 어느 plan이 효율적일지 실험해봤습니다.

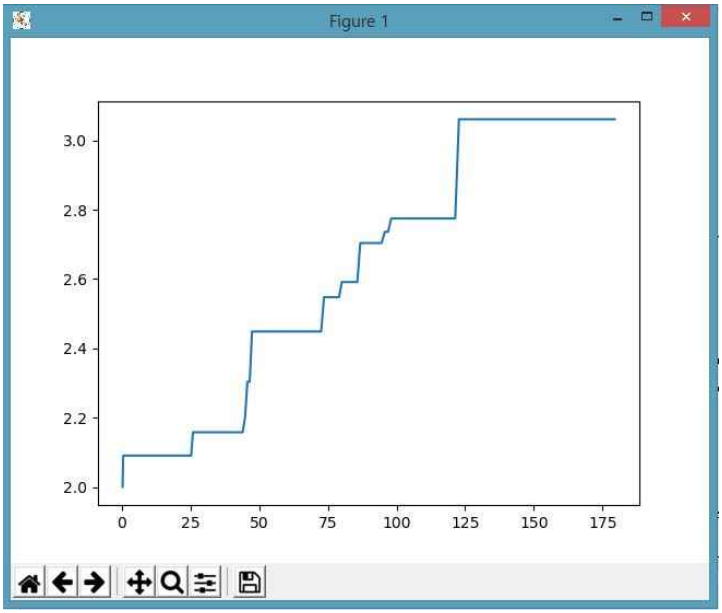
```
# Plan 1
if n == 1:
    offsprings[itr] = self.crossoverParents(p1, p2)

# Plan 2
elif n == 2:
    offsprings[itr] = self.crossoverParents1(p1, p2)

# Plan 3
elif n ==3 :
    if time.time() - startTime <= 60:
        offsprings[itr] = self.crossoverParents(p1, p2)
    else:
        offsprings[itr] = self.crossoverParents1(p1, p2)

# Plan 4
elif n == 4:
    offsprings[itr] = self.crossoverParents2(p1, p2)
```

```
# Plan 5
elif n == 5:
    if time.time() - startTime <= 60:
        offsprings[itr] = self.crossoverParents(p1, p2)
    elif time.time() - startTime > 60 and time.time() - startTime <= 120:
        offsprings[itr] = self.crossoverParents2(p1, p2)
    else:
        offsprings[itr] = self.crossoverParents1(p1, p2)

# Plan 6
elif n == 6:
    if time.time() - startTime <= 20:
        offsprings[itr] = self.crossoverParents(p1, p2)
    elif time.time() - startTime > 20 and time.time() - startTime <= 60:
        offsprings[itr] = self.crossoverParents2(p1, p2)
    else:
        offsprings[itr] = self.crossoverParents1(p1, p2)

# Plan 7
elif n == 7:
    if time.time() - startTime <= 30:
        offsprings[itr] = self.crossoverParents2(p1, p2)
    else:
        offsprings[itr] = self.crossoverParents1(p1, p2)
```

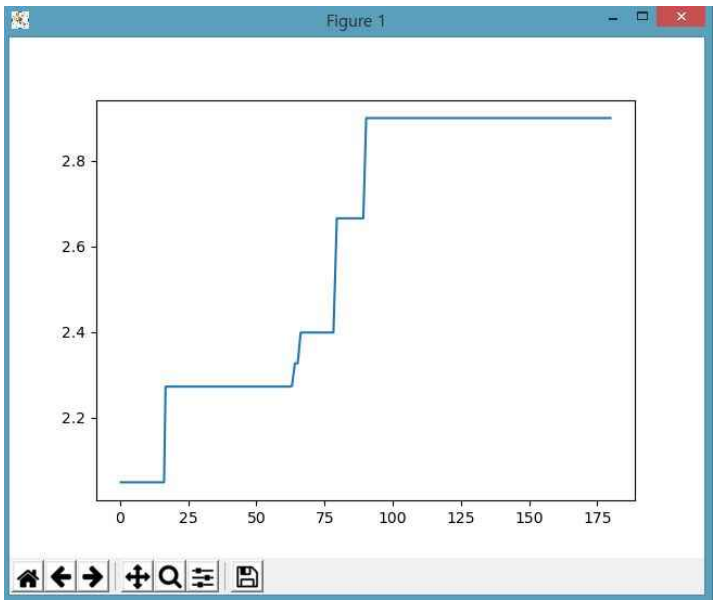
이에 앞서서 먼저 세 종류의 crossover 메소드가 각각 어떤 특징이 있는지 확인해봤습니다. 이 경우 모두 초기에 주어진 parameter를 반영하여 자손의 수는 49, population의 크기는 50으로 진행했습니다.



이 그래프는 showUtillity 메소드를 통해서 crossoverParents 메소드만 사용하여 crossover를 한 결과입니다. 보면 어느 정도의 fitness까지는 변이가 잘 일어나나, 일정 수준에 도달하면 crossover의 랜덤성으로 인하여 자손들의 질 향상이 잘 일어나지 않음을 알 수 있었습니다.



이 그래프는 showUtillity 메소드를 통해서 crossoverParents1 메소드만 사용하여 crossover를 한 결과입니다. 보면 초기의 변이 속도가 crossoverParents 메소드에 비해서 느리나, 일정 시간이 지나면서 자손의 질이 점점 향상될수록 새로운 해를 잘 찾아내며, 변이의 정도가 크지 않기에 작은 수준의 변이가 꾸준히 일어나는, linear한 형태의 결과를 얻을 수 있었습니다.



이 그래프는 showUtillity 메소드를 통해서 crossoverParents2 메소드만 사용하여 crossover를 한 결과입니다. 이 경우 초기 변이 속도가 crossoverParents1에 비해서 꽤 빠른 것을 알 수 있습니다. 그러나 crossover Parents1에 비해서 변이가 큰 편이므로 어느 정도 변이가 축적될 경우 변이로 인해 생기는 좋지 않은 유전자로 인해서 새로운 해가 잘 생성되지 않아 정체되는 것을 볼 수 있습니다.

이 실험을 통해서 각 crossvover의 특성을 확인하였고, 이를 적절히 사용하여 plan들을 생성해봤습니다.

먼저 plan1의 경우 시작부터 끝까지 초기에 제공된 crossoverParents 메소드를 통해서 crossover를 진행하는 것입니다. 그리고 plan2의 경우 위에서 설계한 일점 교차 crossoverParents1 메소드를 사용하는 것입니다. Plan4의 경우 위에서 설계한 이점 교차 crossoverParents2 메소드를 사용하는 것입니다. 이 셋은 일종의 대조군으로 사용됩니다.

이 둘은 일종의 대조군으로 사용됩니다.

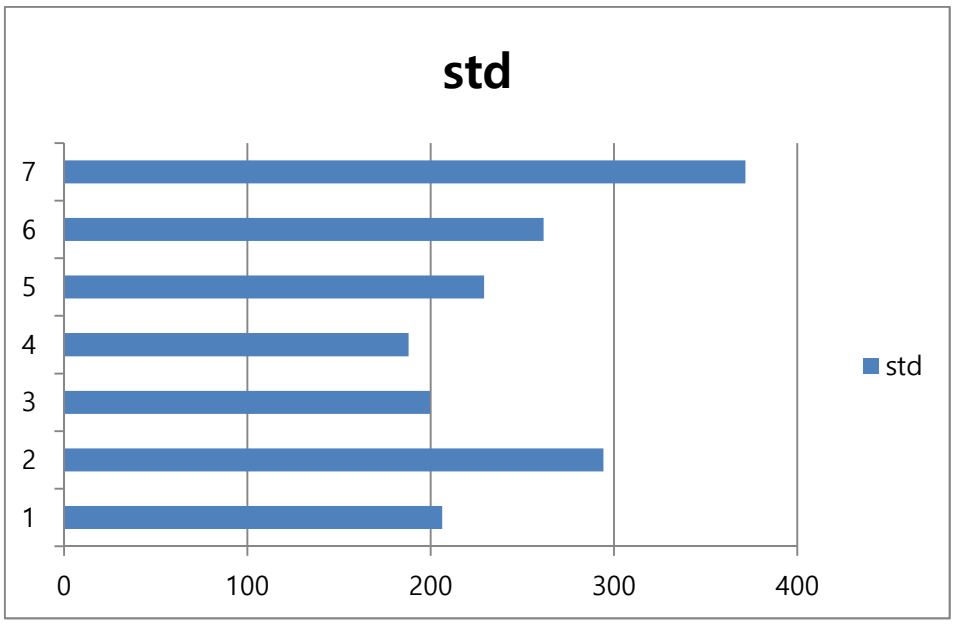
Plan3는 기본 제공된 crossoverParents 메소드와 crossoverParents1 메소드를 둘다 사용하는 것입니다. crossoverParents 메소드의 경우 초기 빠른 변이를 통해 좋은 해를 생성하는 것을 이용하여 초창기에는 이를 사용하고, 이후 crossoverParents1 메소드를 통해서 좋은 자손을 활용하여 새로운 solution을 구하게 설계해봤습니다.

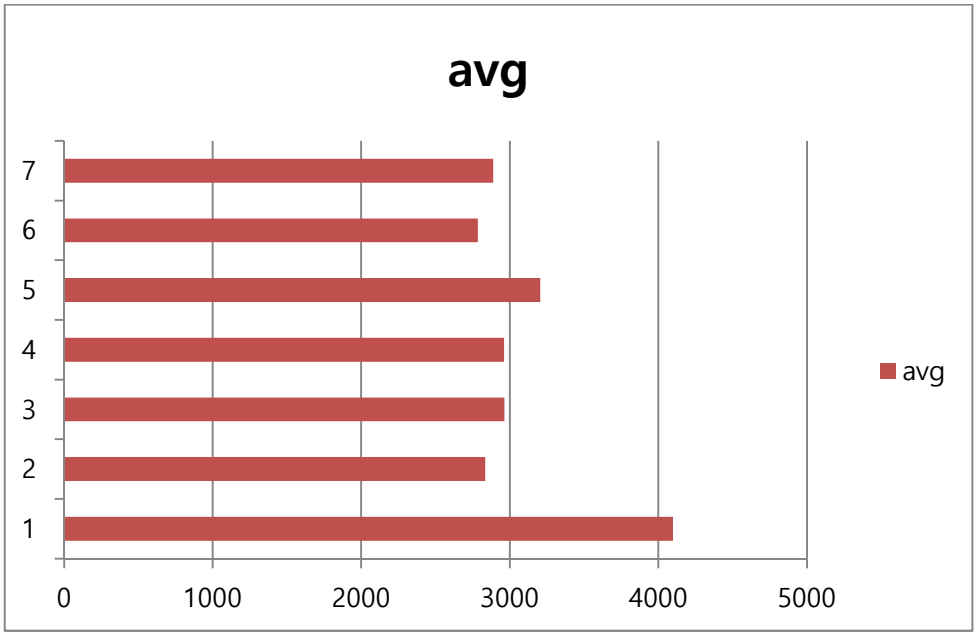
Plan5는 기본 제공된 crossoverParents 메소드와 crossoverParents1 메소드, crossoverParents2 메소드 를 셋 다 사용하는 것입니다. crossoverParents 메소드가 1/3만큼 작동하고, 중간 단계에선 crossoverParents2 메소드를 사용하며, 마지막으로 crossoverParents1 메소드를 통해서 자손의 질을 올리는 plan입니다.

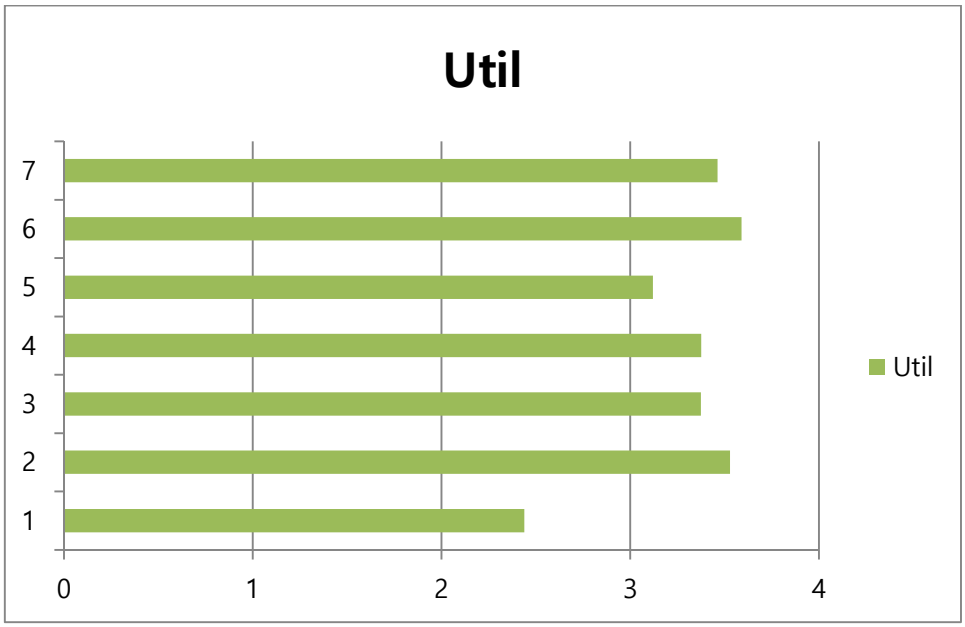
Plan6는 plan5의 특성을 약간 변화하여 각 메소드가 작용하는 시간을 변화시킨 plan입니다.

Plan7는 crossoverParents1 메소드, crossoverParents2 메소드 둘을 사용합니다. crossoverParents2 메소드도 충분한 변이의 다양성을 보여주므로 앞 1/6은 이 메소드를 사용하고, 이후 crossoverParents1 메소드로 자손의 질을 향상시키는 plan입니다.

각 플랜은 자손의 수 70, polulation의 크기 100으로 모두 3분씩 진행되었으며, mutaion과 subsitution은 모두 같은 조건을 사용했습니다. 이 과정을 총 30번 시뮬레이션 하였습니다.

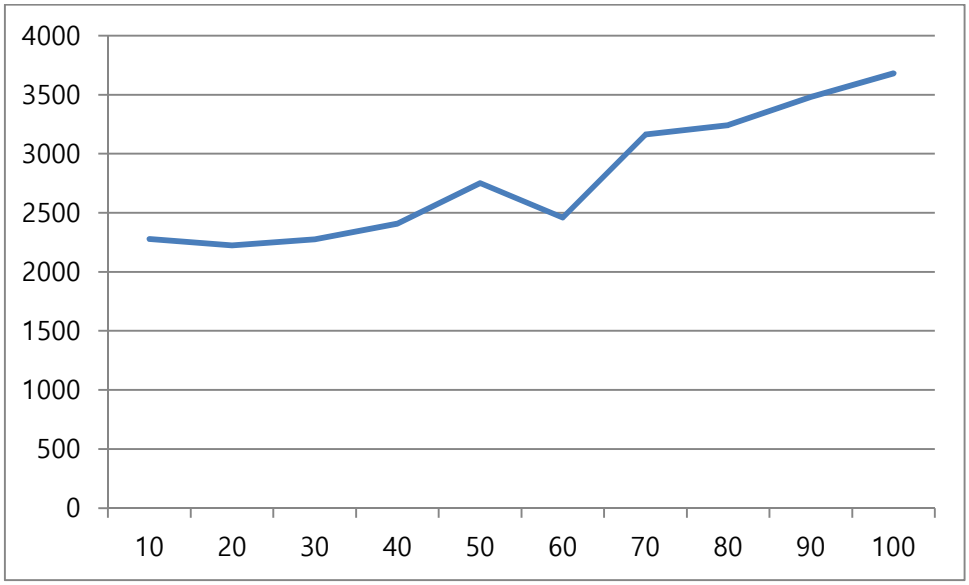






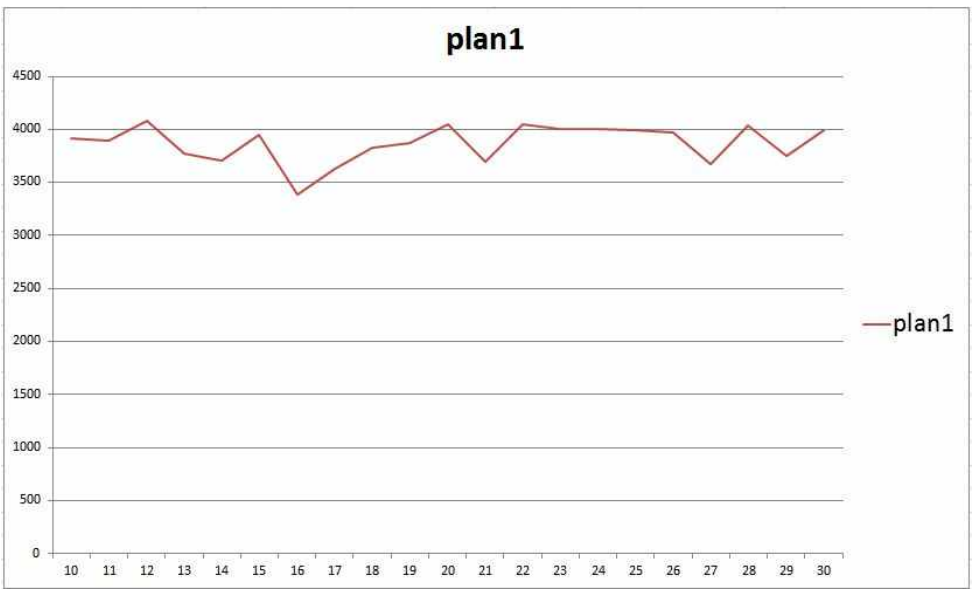
30번을 실행한 결과 2번 6번 7번 플랜이 효과적인 결과를 보여줬습니다. 여기서 표준편차의 경우 얼마나 초기 랜덤성에 잘 대처하냐는 지표로 볼 수 있습니다. Std가 낮을 경우 초기에 랜덤으로 생성된 풀에서도 좋은 결과를 잘 찾아낸다는 것이며, Std가 높을 경우 초기 결과에 많은 영향을 받는다는 것입니다. 여기서 여러 면을 고려했을 때 6번 플랜이 좋다는 것을 발견했습니다.

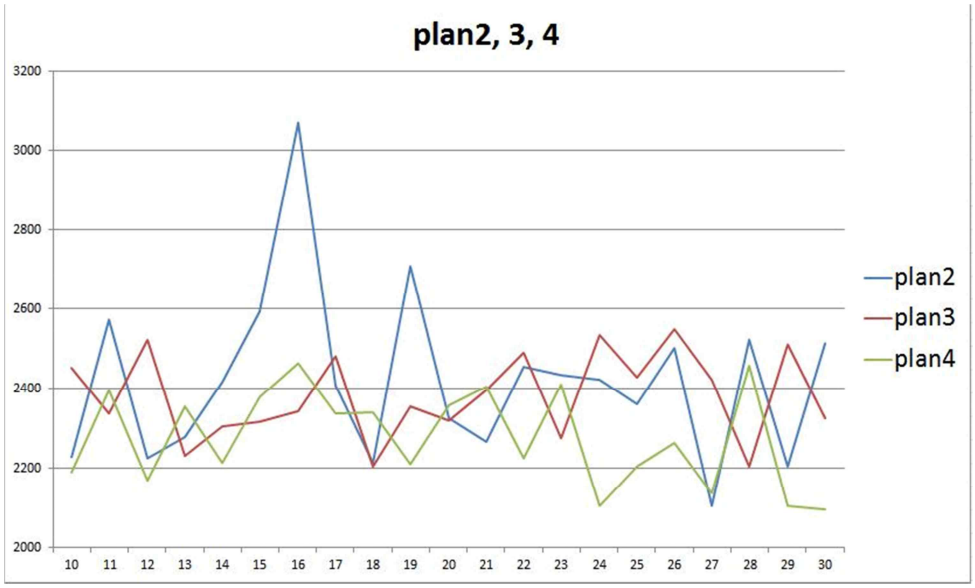
이후 numOffsprings와 numPopulation을 결정하기 위해서 둘을 변형시키면서 6번 플랜에서 테스트한 결과,



다음과 같은 결과를 얻을 수 있었습니다. x축의 경우 numOffsprings를 나타냅니다. 이렇게 점점 offspring의 숫자가 증가할수록 만들어야 할 자손의 수가 많아지면서 오히려 효율이 감소함을 알 수 있었습니다.

이를 통해서 numOffsprings를 10, numPopulation을 20으로 시작하여 각각 1씩 증가시키면서 30, 40까지 키우면서 각 플랜에서 어떤 결과를 보여주는지를 테스트를 했습니다





각 플랜을 검토해본 결과 plan 6이 가장 효율적인 플랜이라는 생각이 들며, numoffsprings가 15 정도에서 가장 효율적인 결과를 가져다 줌을 알 수 있었습니다. 사실 각 반복 횟수가 많지 않아서 신뢰할 수 없으나, numoffsprings를 10에서 30까지 변화해본 결과 큰 편차가 있지 않기에 그 중 15를 선택하였고, numpopulation은 25를 선택하였습니다.

3. Reference

'인공지능 개론', 마이클 네그네빗스키 저, 김용혁 역, 한빛아카데미

이외에 recitation 4 ppt를 참고하였습니다.