AIP

HW 09 – REPORT

소속 : 정보컴퓨터공학부

학번 : 202255535

이름 : 김진우

1. 서론

이번 과제에서는 GA알고리즘을 추가해야 한다. GA알고리즘은 생물체가 진화해가는 모습을 모티브로 하여 최적의 방법을 찾아내는 알고리즘으로 하나의 모집단에서 individual에 crossover과 mutation을 통해 최적의 solution을 찾아내는 알고리즘이다. 또한 이제까지 설계한 모든 알고리즘을 numeric/tsp 모든 문제에 적용시켜보고 파라미터들을 바꿔가며 값을 표로 정리한 후 분석까지 진행해야 한다.

2. 본론

2.1 setup.py

Resolution 값을 setup 클래스에 추가하고 이를 setVariables에도 값 할당을 해준다.

```
class Setup:
    def __init__(self):
        self._pType = 0
        self._aType = 0
        self._delta = 0
        self._alpha = 0
        self._dx = 0
        self._resolution = 0 # 이번과제에서 추가된 변수

    def setVariables(self, parameters):
        self._pType = parameters['pType']
        self._aType = parameters['aType']
        self._delta = parameters['delta']
        self._alpha = parameters['alpha']
        self._dx = parameters['dx']
        self._resolution = parameters['resolution'] # 추가
```

2.2. main.py

GA알고리즘을 추가하였기 때문에 이에 맞게 main도 수정을 해야한다.

2.2.1. readPlan()

GA에 사용되어 새롭게 추가된 파라미터들을 읽어와야 하기 때문에 파라미터를 추가해준다.

```
def readPlan(): # problem과 algorithm 객체를 설정해줄 parameter 받아오기
    fileName = input("Enter the file name of experimental setting: ")
    infile = open(fileName, 'r')

    #조교님이 제공해준 코드보고 parameters부분 채우기***
    parameters = { 'pType':0, 'pFileName':'', 'aType':0, 'delta':0,
                   'limitStuck':0, 'alpha':0, 'dx':0, 'numRestart':0,
                   'limitEval':0, 'popSize':0, 'resolution':0, 'uXp':0,
                   'mrF':0,'XR':0,'mR':0,'numExp':0 }
```

2.2.2createOptimizer()

GA알고리즘도 선택할 수 있게 추가한다

```
def createOptimizer(parameters): ###
    # Create an optimizer instance (a class object) 'alg' of the type
    # as specified by 'aType', set the class variables, and return 'alg'.
    optimizers = {1: 'SteepestAscent()',
                  2: 'FirstChoice()',
                  3: 'Stochastic()',
                  4: 'GradientDescent()',
                  5: 'SimulatedAnnealing()',
                  6: 'GA()'
                  }
```

2.3. problem.py

2.3.1 numeric class

GA에서 사용하기 위한 initializePop, randBinStr, evalInd, decode, binaryToDecimal, crossover, uXover, mutation, indToSol 메소드를 새롭게 정의한다.

- initializePop(size) : Population size 만큼 individual을 만든다. 이때 randBinStr메소드를 사용하여 chromosome을 만든다.

```
# initializePop과 randBinStr는 GA를 위한 메소드
def initializePop(self, size):
    pop = []
    for i in range(size):
        chromosome = self.randBinStr()
        pop.append([0, chromosome])
    return pop
```

- randBinStr() : "resolution*변수의 수" 길이만큼의 0/1로 구성된 리스트를 만들고 이를 리턴한다.

```
def randBinStr(self):
    # Numeric 문제의 변수 N (self._domain[0]) 개에 대해서,
    # 각 변수 별 self._resolution 크기의 random binary 생성
    # N=5, self._resolution=10 이라면,
    # 50길이의 [1, 0, 1, 0, 0, 1, 0, 1, …] 배열 생성하여 반환
    #k = len(self._domain[0]) * self._resolution
    k = len(self._domain[0]) * self._resolution
    chromosome = []
    for i in range(k):
        allele = random.randint(0, 1)
        chromosome.append(allele)
    return chromosome
```

- evalInd(ind) : Individual의 값을 evaluate 메소드를 사용해 계산한다. 이때 [1,0,1,0,0,1...]로 표현한 값을 evaluate할 수 있도록 실수로 변환하는 decode메소드가 필요하다.

```
def evalInd(self, ind):
    ind[0] = self.evaluate(self.decode(ind[1])) # [1,0,1,0,0,1...]로 표현한 값을 evaluate할 수 있도록 실수로 변환하는 decode함수 필요
```

- decode(chromosome) : chromosome은 0과 1로 구성된 리스트인데, 이를 evaluate하기 위해서는 실수로 변환해야한다. 이 과정을 수행하는 것이 바로 decode 메소드이다.

```
def decode(self, chromosome):
    r = self._resolution
    low = self._domain[1]
    up = self._domain[2]
    genotype = chromosome[:]
    phenotype = []
    start = 0
    end = r

    #print(start, "  ",end)
    for var in range(len(self._domain[0])):
        value = self.binaryToDecimal(genotype[start:end], low[var], up[var])
        phenotype.append(value)

        start += r
        end += r
    return phenotype
```

- binaryToDecimal(binCode, l, u) : binCode를 최댓값과 최솟값 사이의 실수로 변환하여 주는 메소드이다.

```
def binaryToDecimal(self, binCode, l, u):
    # binCode [0, 1, 0, 1, …]을
    # low, upper bound내의 값으로 표현해서 return
    # l=0, u=10, binCode=[x, x, x, x] 일 때,
    # [0,0,0,0]=0, [0,0,0,1]=0.625, [0,0,1,0]=1.25
    # [0,0,1,1]=1.875, …, [1,1,1,1]=9.375
    # u가 포함 안되는데, binCode가 충분히 길다는 가정하에 포함 여부는 구현에 큰 문제없음
    r = len(binCode)
    decimalValue = 0
    for i in range(r):
        decimalValue += binCode[i] * (2 ** (r - i - 1))
    return l + (u - l) * decimalValue / (2 ** r - 1)
```

- crossover(ind1, ind2, uXp) : Crossover를 통해 새로운 2개의 individual을 반환한다. 이때 uXover이라는 메소드를 사용한다,

```
def crossover(self, ind1, ind2, uXp):
    chr1, chr2 = self.uXover(ind1[1], ind2[1], uXp)
    return [0, chr1], [0, chr2]
```

- uXover(chrInd1, chrInd2, uXp) : uniform crossover를 구현한 메소드로 uXp을 기준으로 crossover 진행 여부를 결정한다.

```
def uXover(self, chrInd1, chrInd2, uXp): # uniform crossover
    # chrInd1, chrInd2의 각 원소를 확률적(uXp)으로 crossover
    chr1 = chrInd1[:] # Make copies
    chr2 = chrInd2[:]
    # implement
    for i in range(len(chr1)):
        if random.uniform(0, 1) < uXp:
            chr1[i], chr2[i] = chr2[i], chr1[i]
    return chr1, chr2
```

- mutation(ind, mrF) : mrF * (1/ lnegth of individual) 확률로 mutation 유무를 결정한다.

```
def mutation(self, ind, mrF): # bit-flip mutation
    # mrF * (1/ lnegth of individual) 확률로 ind의 개별 원소 bit-flip
    child = ind[:] # Make copy
    n = len(child[1])
    # implement
    for i in range(n):
        if random.uniform(0, 1) < mrF * (1/n):
            child[1][i] = 1 - child[1][i]
    return child
```

- indToSol(ind) : individual을 solution 형태로 변환한다.

```
def indToSol(self, ind): # ind를 solution으로 변환
    return self.decode(ind[1])
```

2.3.2 tsp class

Numeric class에서와 마찬가지로 GA에서 사용하기 위한 initializePop, evalInd, crossover, oXover, indToSol, mutation 메소드를 정의한다.

- initializePop(size) : 랜덤한 방문 순서를 pop 크기만큼 만들고, randomInit을 통해 chromosome을 만든 다음 이를 통해 population을 만들어 return을 한다.

```
def initializePop(self, size):
    #n = self._numCities
    pop = []
    for i in range(size):
    # tsp.randomInit 메서드 이용하여 chromosome 생성하고 pop에 추가
    # chromosome = [eval_value, [tour order]]
    # = [0, [5, 12, 17, 11, 7, 22, …]]
        chromosome = self.randomInit()
        pop.append([0, chromosome])
    return pop
```

- evalInd(ind)

```
def evalInd(self, ind):
    ind[0] = self.evaluate(ind[1])
```

- crossover(ind1, ind2, XR) : Crossover를 통해 새로운 2개의 individual을 반환한다. 이때 oXover이라는 메소드를 사용한다.

```
def crossover(self, ind1, ind2, XR):
    if random.uniform(0,1) <= XR:
        chr1, chr2 = self.oXover(ind1[1], ind2[1])
    else:
        chr1, chr2 = ind1[1][:], ind2[1][:]
    return [0, chr1], [0, chr2]
```

- mutation(ind,mR) : mR값에 따라 mutation을 진행한다.

```
def mutation(self, ind, mR): # 두 지점 사이의 값을 거꾸로 뒤집는 방식으로 변이가 일어난다
    mInd = ind[:]
    if random.uniform(0,1) < mR:
        while True:
            a = random.randint(0, len(mInd[1]))
            b = random.randint(0, len(mInd[1]))
            if a < b:
                break
        mInd[1][a:b] = mInd[1][b-1:a-1 if a != 0 else None:-1]
    return mInd
```

- indToSol(ind) : individual을 solution형태로 저장한다.

```
def indToSol(self, ind):
    return ind[1]
```

- oXover(chrInd1, chrInd2) : crossover를 진행하는 부분으로 ppt에서 제공된 과정에 맞게 코드를 구현하였다. 자세한 설명은 주석을 참고하길 바란다. 핵심 내용은 중복된 숫자를 제외하기 위한 과정을 거쳐야 하고, 지정된 구간만큼이 서로 교차되어 새로운 chronosome이 만들어지는 것이다.

```python
def oXover(self, chrInd1, chrInd2):
    max = self._numCities
    while True:
    # 범위에 맞게 난수 2개 생성
        num1 = random.randint(1, max)
        num2 = random.randint(1, max)
        # 생성된 난수를 비교하여 작은 값을 a에, 큰 값을 b에 할당
        if num1 < num2:
            a = num1
            b = num2
        else:
            a = num2
            b = num1
        if a != b:
            break
    chr1 = chrInd1[:]
    chr2 = chrInd2[:]
    # 중복 도시 제거를 위해 교차 구간에 해당하는 도시를 -1로 표시
    for i in range(len(chr1)):
        if chr1[i] in chrInd2[a:b]:
            chr1[i] = -1
        if chr2[i] in chrInd1[a:b]:
            chr2[i] = -1
    # a:b 사이를 비우는 작업 (left shift)
    for i in range(b):
        chr1.append(chr1[0])
        chr2.append(chr2[0])
        del chr1[0]
        del chr2[0]
    # -1 값을 제거
    chr1 = [x for x in chr1 if x != -1]
    chr2 = [x for x in chr2 if x != -1]
    # 나머지 부분 추가 (left shift 후 남은 부분 복원)
    for i in range(len(chrInd1) - b):
        chr1.append(chr1[i])
        chr2.append(chr2[i])
    del chr1[:len(chrInd1) - b]
    del chr2[:len(chrInd1) - b]
    # a:b 구간 교차
    for i in range(a, b):
        chr1.insert(i, chrInd2[i])  # chrInd2의 a:b 구간을 chr1에 삽입
        chr2.insert(i, chrInd1[i])  # chrInd1의 a:b 구간을 chr2에 삽입
    return chr1, chr2
```

2.4. optimizer.py

GA 알고리즘에 필요한 method 들과 생성자, setVariables, run, displaySetting method 를 정의한다.

이때 제공된 skeleton code 와 실습때 진행한 코드 대부분을 그대로 사용하였다. 그렇기 때문에 코드만 첨부를 하고 설명은 하지 않았다.

1) __init__

```python
class GA(Metaheuristics):
    def __init__(self):
        Metaheuristics.__init__(self)
        self._popSize = 0     # Population size
        self._uXp = 0   # Probability of swappping a locus for Xover
        self._mrF = 0   # Multiplication factor to 1/n for bit-flip mutation
        self._XR = 0    # Crossover rate for permutation code
        self._mR = 0    # Mutation rate for permutation code
        self._pC = 0    # Probability parameter for Xover
        self._pM = 0    # Probability parameter for mutation

    def setVariables(self, parameters):
        Metaheuristics.setVariables(self, parameters)
        self._popSize = parameters['popSize']
        self._uXp = parameters['uXp']
        self._mrF = parameters['mrF']
        self._XR = parameters['XR']
        self._mR = parameters['mR']
        if self._pType == 1:
            self._pC = self._uXp
            self._pM = self._mrF
        if self._pType == 2:
            self._pC = self._XR
            self._pM = self._mR
```

```python
def run(self, p):
    # Population 생성
    pop = p.initializePop(self._popSize)
    # Population 중 최적해 찾기
    best = self.evalAndFindBest(pop, p)
    numEval = p.getNumEval()
    whenBestFound = numEval
    # limitEval 까지 [다음세대 생성-평가] 반복
    while numEval < self._limitEval:
        newPop = []
        I = 0
    # 다음 세대 생성; start
        while I < self._popSize:
            par1, par2 = self.selectParents(pop)
            ch1, ch2 = p.crossover(par1, par2, self._pC)
            newPop.extend([ch1, ch2])
            I += 2
        newPop = [p.mutation(ind, self._pM) for ind in newPop]
        pop = newPop
        # 다음 세대 생성; end
        # 다음 세대 값 평가 및 best 업데이트
        newBest = self.evalAndFindBest(pop, p)
        numEval = p.getNumEval()
        if newBest[0] < best[0]:
            best = newBest
            whenBestFound = numEval
    self._whenBestFound = whenBestFound
    bestSolution = p.indToSol(best)
    p.storeResult(bestSolution, best[0])
```

```
def evalAndFindBest(self, pop, p):
    best = pop[0]
    p.evalInd(best)
    bestValue = best[0]
    for i in range(1, len(pop)):
        p.evalInd(pop[i])
        newValue = pop[i][0]
        if newValue < bestValue:
            best = pop[i]
            bestValue = newValue
    return best

def selectParents(self, pop):
    ind1, ind2 = self.selectTwo(pop)
    par1 = self.binaryTournament(ind1, ind2)
    ind1, ind2 = self.selectTwo(pop)
    par2 = self.binaryTournament(ind1, ind2)
    return par1, par2

def selectTwo(self, pop):
# pop에서 random하게 2개의 individuals 선택해서 반환
    popCopy = pop[:]
    random.shuffle(popCopy)
    return popCopy[0], popCopy[1]
```

```
def binaryTournament(self, ind1, ind2):
# 2개의 individuals 중 더 좋은 ind 선택해서 반환
    if ind1[0] < ind2[0]:
        return ind1
    else:
        return ind2

def displaySetting(self):
    print()
    print("Search Algorithm: Genetic Algorithm")
    print()
    Metaheuristics.displaySetting(self)
    print()
    print("Population size:", self._popSize)
    if self._pType == 1:   # Numerical optimization
        print("Number of bits for binary encoding:", self._resolution)
        print("Swap probability for uniform crossover:", self._uXp)
        print("Multiplication factor to 1/L for bit-flip mutation:",
              self._mrF)
    elif self._pType == 2: # TSP
        print("Crossover rate:", self._XR)
        print("Mutation rate:", self._mR)
```

3. 결론

numRestart = 10, numExp = 10

<Numeric>

Avg object value/ avg num of evaluations/best objective value/avg iteration of find best solution

		Convex	Griewank	Ackley
Steepest Ascent	Delta = 0.01	0.0 794,804 0.0	0.227 70.855 0.069	17.948 12.261 14.258
First Choice	Delta = 0.01 Limistuck = 1000	0.0 281.880 0.0	0.182 37,746 0.084	16.936 14,335 13.401
Stochastic	Delta = 0.01 Limistuck = 1000	0.0 1,963,752 0.0	0.160 377,200 0.054	18.175 143,267 15.346
Gradient Descent	alpha = 0.01 dx = 10^-4	0.0 71,366 0.0	0.135 455,454 0.049	17.585 5,616 16.245
Simulated Annealing	limitEval = 50000	0.002 275,110 0.000 48,346	0.409 275,110 0.197 16,259	18.952 275,110 17.086 4,867
GA	resolution = 10 limitEval = 50000 popSize = 100 uXp = 0.2 mrF = 1	744.964 50,450 4.247 49,490	0.530 50,450 0.082 46,620	9.435 50,450 0.163 49,230

1. Steepest Ascent : 계산양이 적고 빠르게 수렴하나, 복잡한 문제에서는 최적의 값을 잘 찾아내지 못한다.

2. First Choice : 계산 속도는 Steepest Ascent와 유사하지만 steepest ascent에 비해 Griewank와 Ackley에서 약간의 성능 향상이 있다.

3. Stochastic : 실행해본 결과 가장 시간이 오래 걸리고 연산양이 많으며, Convex에서만 최적해를 보장하는 것을 확인할 수 있다

4. Gradient Descent : hill-climbing 알고리즘들 중에서는 계산 효율이 가장 좋고 Convex 및

Griewank에서 우수한 성능을 보인다.

5. Simulated Annealing : 높은 계산량에도 불구하고 상대적으로 다른 알고리즘들에 비해 최적의 solution을 잘 찾아낸 것을 확인할 수 있다.

6. Genetic Algorithm (GA) : 높은 계산량을 필요로하지만 Ackley에서 다른 알고리즘에 비해 압도적 성능을 가짐을 확인할 수 있다.

즉, Numeric 문제를 모든 알고리즘에 대해 수행했을 때, 간단한 문제(Convex)는 Gradient Descent가 가장 효율적이고 복잡한 문제 (Griewank, Ackley): Genetic Algorithm이 최적의 품질을 제공하며, Simulated Annealing도 높은 성능을 발휘한다고 분석할 수 있다. 그러나 시간이 너무 오래 걸리기 때문에 계산 효율이 더 중요하면 Steepest Ascent나 Gradient Descent를, 최적해 찾는 것이 더 중요하다면 Simulated Annealing이나 GA를 선택하면 된다고 결론지을 수 있다.

<TSP>

Avg object value/ avg num of evaluations/best objective value/avg iteration of find best solution

		Tsp30	Tsp50	Tsp100
Steepest Ascent		571 6,650 545	807 19,890 757	1236 87,001 1,104
First Choice	Limistuck = 1000	461 27,674 454	614 54,413 596	914 127,779 878
Stochastic	Limistuck = 1000	459 690,082 454	608 1,972,874 597	914 127,779 878
Gradient Descent	alpha = 0.01 dx = 10	불가능	불가능	불가능
Simulated Annealing	limitEval = 50000	457 275,110 454 33,012	610 275,110 598 43,881	880 275,110 851 48,216
GA	limitEval = 50000 popSize = 100 XR = 0.1 mR = 0.1	1195 50,450 759 46,700	2119 50,450 1,318 47,200	4420 50,450 2,668 50,270

1. Steepest Ascent : numeric에서와 마찬가지로 계산량은 적지만, 복잡한 문제에서는 성능이 크게 저하되는 것을 확인할 수 있다.

2. First Choice : 계산량이 적지만, tsp5과 tsp100에서 steepest ascent에 비해 성능이 향상됨을 확인할 수 있다.

3. Stochastic : 계산량이 매우 크고 실제로도 실행시간이 가장 오래 걸리는 비효율적인 알고리즘이다.

4. Gradient Descent : TSP 문제에서는 미분이 불가능하기 때문에 사용 할 수 없다.

5. Simulated Annealing : 계산량이 적고 품질도 뛰어나다.

6. Genetic Algorithm (GA) : 계산량이 크지만, 성능은 다른 알고리즘들에 비해 좋지 못하다.

즉, tsp문제를 모든 알고리즘에 대해 실행시켰을 때, 대부분의 문제가 optimal solution이거나 이와 비슷한 값을 찾아주는 것을 확인할 수 있었다. 그래도 좀 더 분석하자면 소규모 문제 (TSP30)는 First Choice 또는 Simulated Annealing 방법이 효율적이고 중규모 문제 (TSP50)는 Simulated Annealing이 성능과 계산 효율성에서 균형을 이루고 대규모 문제 (TSP100)는 Simulated Annealing이 적합하다는 결론을 내릴 수 있다.