AIP

HW 06 – REPORT

소속 : 정보컴퓨터공학부

학번 : 202255535

이름 : 김진우

1. 서론

지난 HW05에서 hill climbing algorithm을 이용해 tsp와 numeric 문제를 해결하였다.

을 해결하였다. HW06에서는 기존에 함수로 작성된 코드들을 class화 하고 numeric 문제에 대해서 gradient descent 방법을 추가해야 한다.

Class화 하기 위해서는 먼저 problem class와 이를 superclass로 갖는 numeric class와 tsp class가 있게 구현하면 된다.

Gradient Descent 는 w <- w – af'(w)라는 식(이때 a는 step size를 의미함)을 통해 a만큼 움직이면서 critical point를 찾는 방법을 뜻한다. 이 방법의 경우 미분계산이 가능한 numeric에만 적용이 되고 미분 계산이 불가능한 tsp에서는 적용되지 않는 다는 점을 유의해야 한다.

2. 본론

본론에 들어가기 앞서 간단히 본론에 설명할 내용을 이야기하자면, OOP 스타일로 바꾸기 위해 code migration 과정을 거쳤고 그러기 위해서 전체적인 코드를 function 호출하는 부분을 method 호출로 바꿨다. 이전 과제에서 사용한 것이 아니라 새롭게 생성한 메소드를 위주로 설명을 하되 지난 과제에서 사용했던 함수에 대해서는 간략하게 설명만 하였다.

2.1. problem.py

2.1.1 problem class

Problem class는 numeric과 tsp class의 super class로 solution, value, numEval 3개의 field를 가진다.

```
class Problem:
    def __init__(self):
        self._solution = []
        self._value = 0
        self._numEval = 0
```

나머지 메소드들에 대해서는 problem class를 super class로 갖는 numeric과 tsp class에서 메소드를 덮어쓰기 때문에 메소드를 구현할 필요가없고 report 메소드만 구현하면 된다.

```
def report(self):
    print()
    print("Total number of evaluations: {0:,}".format(self._numEval))
```

2.1.2. Numeric class

Numeric class에서는 __init__, setVariables, getDelta, randomInit, evaluate, mutate, mutants,

randomMutant, describe, report, coordinate method를 구현해야한다. 또한 gradient descent 문제를 해결하기 위한 takeStep, getAlpha, getDx, gradient, isLegal method가 필요하다. 이때 gradient와 isLegal 메소드는 takeStep에서 사용되는 메소드이다.

__init__(self) 는 주어졌기 때문에 그 다음 메소드들부터 설명하겠다.

1) setVariables

이 메소드는 파일로부터 문제를 읽어와 각 변수에 값을 set해주는 메소드이다.

```
def setVariables(self):
    filename = input("Enter the file name of a function: ")
    infile = open(filename, "r")
    self._expression = infile.readline()
    varNames, low, up = [], [], [] # 각각 변수 이름, 최소값, 최대값을 저장할 리스트
    line = infile.readline()

    while line != "":
        data = line.split(",")
        varNames.append(data[0])
        low.append(eval(data[1]))
        up.append(eval(data[2]))
        line = infile.readline()
    infile.close()
    self._domain = [varNames, low, up]
```

2) getDelta

이 메소드는 numeric class의 delta값을 반환해주는 메소드이다

```
def getDelta(self): # new method
    return self._delta
```

3) randomInit

이 메소드는 초기값을 random하게 지정해주는 메소드로 random.uniform을 사용해 최소값과 최댓값 사이의 랜덤한 실수값을 지정하고 이렇게 가져온 값들을 list에 저장하여 이 list를 return한다.

```
def randomInit(self):
    domain = self._domain
    low, up = domain[1], domain[2]
    init = []

    for i in range(len(low)):
        r = random.uniform(low[i], up[i])
        init.append(r)
    return init
```

4) evalutate(current)

이 메소드는 한번 실행될 때 마다 problem class의 _numEval 값을 1씩 증가시키고, current에 저장되어있는 값들을 읽어 변수명 = '값' 형태의 string을 return한다.

```
def evaluate(self, current):
    self._numEval += 1 # 총 eval 횟수 하나식 증가
    expr = self._expression
    varNames = self._domain[0]
    for i in range(len(varNames)):
        assignment = varNames[i] + '=' + str(current[i]) # 변수명 = '값' 형태로 출력되게 설정
        exec(assignment) # 설정된 값을 실행
    return eval(expr)
```

5) mutants(current)

이 메소드는 steepest-ascent를 위해 필요한 메소드로 current의 모든 값들에 대해 mutate한 이웃들을 return한다

```
# steepest-ascent
def mutants(self, current): # steepest ascent (n)에서 Numeric problem 관련 함수를 numeric class로 이동
    neighbors = []
    for i in range(len(current)): # For each variable
        mutant = self.mutate(current, i, self._delta)
        neighbors.append(mutant)
        mutant = self.mutate(current, i, -self._delta)
        neighbors.append(mutant)
    return neighbors
```

6) mutate(current, i, d)

이 메소드는 current의 값들 중 i번째 value에 대해 d만큼 mutate하고 이 값이 범위안에 속하면 mutate된 값을 범위 밖에 속하면 current를 리턴한다.

```
def mutate(self, current, i, d):
    mutant = current[:]
    domain = self._domain
    l= domain[1][i]
    u = domain[2][i]
    if l <= (mutant[i] + d) <= u:
        mutant[i] += d
    return mutant
```

7) randomMutant(current)

이 메소드는 first-choice를 위해 필요한 메소드로 current에 있는 랜덤한 i번째 값을 증가 또는 감소 시킨 리스트를 mutate 메소드를 통해 구한 뒤 그 값을 리턴한다. 이때 증가 또는 감소의 비율을 0.5이다.

```
# first-choice
def randomMutant(self, current): ###
    i = random.randint(0, len(current) - 1)  # 몇 번째 var를 변경할 것인지를 랜덤하게 지정
    if random.uniform(0,1) < 0.5: # 이때 D는 0.5의 확률로 증가할지 감소할지 결정되어야함
        d = self._delta
    else:
        d = - self._delta

    return self.mutate(current, i, d)
```

8) describe

문제를 설명해주는 메소드이다.

```
def describe(self): # describeProblem
    print()
    print("Objective function:")
    print(self._expression)
    print("Search space:")
    varNames = self._domain[0]
    low = self._domain[1]
    up = self._domain[2]
    for i in range(len(low)):
        print(" " + varNames[i] + ":", (low[i], up[i]))
```

9) report

Solution이 무엇인지 print해주는 메소드이다. 이때 problem class의 report 메소드가 사용된다.

```
def report(self):
    print()
    print("Solution found: ")
    print(self.coordinate())
    print("Minimum value: {0:,.3f}".format(self._value))
    Problem.report(self)
```

10) coordinate

Solution을 반올림해서 tuple의 형태로 return하게 해주는 함수이다.

```
def coordinate(self):
    c = [round(value,3) for value in self._solution]
    return tuple(c)
```

여기서부터는 gradient descent를 위해 필요한 메소드들이다.

11) getAlpha

Numeric class의 _alpha값을 리턴해준다.

```
# gradient descent
def getAlpha(self): # self._alpha Accessor
    return self._alpha
```

12) getDx

Numeric class의 _dx값을 리턴해준다.

```
def getDx(self): # self._dx Accessor
    return self._dx
```

13) takeStep(current, valueC)

Gradient한 방법을 통해 next step을 계산하는 메소드이다. 이때 w′ = w – alpha*f′(w) 라는 식을 사용한다. f′(w)를 구하기 위해서 gradient 메소드가 필요하고 새롭게 만들어진 값이 범위내에 들어가는지를 확인하기 위해 isLegal이라는 메소드의 추가적인 구현이 필요하다. 이때 새로 만들어진 값이 범위내에 속한다면 새로운 값을 리턴하고 그렇지 않다면 원래의 값을 반환한다.

```
# gradient를 통해 next step을 계산
# next step이 domain 범위 이내 일 때만 next step을 반환
def takeStep(self, currentP, valueC):
    next = currentP[:] # 리스트 복사
    gradient = self.gradient(currentP,valueC) # gradient 값 계산
    for i in range(len(next)): # 각각의 변수에 대해
        next[i] -= self._alpha * gradient[i] # gradient 값에 따라 다음 step을 계산 -> w' = w - alpha*f'(w)
    if self.isLegal(next): # 새로 만들어진 값이 범위 내에 있다면
        return next # 새로운 값을 반환
    else:
        return currentP # 범위 밖이면 원래 값을 반환
```

14) isLegal(x)

주어진 변수의 값인 x가 범위 내에 속하는지 아닌지를 확인하고 이에 맞게 T/F를 반환하는 메소드이다.

```
# 주어진 변수 값들(x)이 도메인 범위 이내인지 확인하고 T/F를 반환
def isLegal(self, x):
    low = self._domain[1]
    up = self._domain[2]
    for i in range(len(x)): # 각 변수에 대해 실행
        if(low[i] > x[i] or up[i] < x[i]): # 범위 밖이면
            return False # False 반환
    return True # 범위 안이면 True 반환
```

15) gradient(x, v)

f′(x)의 값을 구할 때 필요한 메소드이다. 이때 f'(x) = (f(x+dx) - f(x)) / dx 식을 사용한다.

```
# '각 변수'의 gradient를 list형으로 반환
def gradient(self, x, v): # x는 현재 변수 값, v는 현재 값 list
    gradient = [] # gradient 값을 저장할 리스트
    for i in range(len(x)): # 각 변수에 대해
        changedX = x[:] # 리스트 복사
        changedX[i] += self._dx # i번째 변수에 f(x+dx) 해주기
        gradient.append((self.evaluate(changedX) - v) / self._dx) # f'(x) = (f(x+dx) - f(x)) / dx 계산
    return gradient # gradient 값 반환
```

2.1.3 Tsp class

Tsp class에서는 __init__, setVariables, calcDistanceTable, randomInit, evaluate, mutants, inversion, randomMutant, describe, report, tenPerRow 메소드를 구현해야한다. __init__은 제외하고 그 다음 메소드부터 설명하도록 하겠다.

1) setVariables

이 메소드는 파일로부터 문제를 읽어와 각 변수에 값을 set해주는 메소드이다.

이때 값 초기화를 제대로 해두지 않으면 나중에 코드를 실행시켰을 때 제대로 작동하지 않으니 주의해야 한다.

```
def setVariables(self):
    ## Read in a TSP (# of cities, locatioins) from a file.
    ## Then, create a problem instance and return it.
    fileName = input("Enter the file name of a TSP: ")
    infile = open(fileName, 'r')
    # First line is number of cities
    self._numCities = int(infile.readline())
    locations = []
    line = infile.readline()  # The rest of the lines are locations
    while line != '':
        locations.append(eval(line)) # Make a tuple and append
        line = infile.readline()
    infile.close()
    self._locations = locations
    self._distanceTable = self.calcDistanceTable()
```

2) calcDistanceTable

이 메소드는 한 도시에서 다른 도시까지의 거리를 계산하고, 이 거리를 2차원 matrix로 만드는 메소드이다. 이때 두 도시간의 거리는 거리를 구하는 공식을 사용하고, 제곱근을 위해 math.sqrt()를 사용하였다.

```
def calcDistanceTable(self):
        n = self._numCities
        locations = self._locations
        table = [[0] * n for _ in range(n)] # 2차원 리스트 초기화

        for i in range(n):
            for j in range(n):
                if i != j:  # 같은 도시 간 거리는 계산하지 않음
                    table[i][j] = math.sqrt((locations[j][0] - locations[i][0]) ** 2 +
                                            (locations[j][1] - locations[i][1]) ** 2) # 두 점 사이의 거리를 구하는 공식 사용해서 table에 저장
        return table  # A symmetric matrix of pairwise distances
```

3) randomInit

이 메소드는 랜덤하게 방문 순서를 return해주는 메소드이다. 이를 위해 radom.shuffle()을 사용하였다.

```
def randomInit(self):   # Return a random initial tour
    n = self._numCities
    init = list(range(n))
    random.shuffle(init)
    return init
```

4) evaluate(current)

이 메소드는 한번 실행될 때 마다 super class인 problem class의 _numEval의 값을 1씩 증가시킨다. 또한 도시를 방문할 때 드는 cost를 다 더한 값을 return 해주는 메소드이다.

```
def evaluate(self, current): ###
    ## Calculate the tour cost of 'current'
    ## 'p' is a Problem instance
    ## 'current' is a list of city ids
    self._numEval += 1 # 총 eval횟수 늘이기
    n = self._numCities
    table = self._distanceTable
    cost = 0 # 비용 초기화

    # 첫 도시부터 맨 마지막 도시까지 투어하면서 distance table을 참고하여 비용 계산
    for i in range(n-1):
        locFrom = current[i]
        locTo = current[i+1]
        cost += table[locFrom][locTo]
    # 맨 마지막 도시에서 처음 도시로 돌아오는 비용을 고려해서 cost에 더해줌
    cost += table[current[-1]][current[0]]

    return cost
```

5) mutants(current)

Steepest-ascent를 위해 필요한 메소드이다.

```
#steepest-ascent
def mutants(self, current): # Apply inversion
    n = self._numCities
    neighbors = []
    count = 0
    triedPairs = []
    while count <= n:  # Pick two random loci for inversion
        i, j = sorted([random.randrange(n) for _ in range(2)])
        if i < j and [i, j] not in triedPairs:
            triedPairs.append([i, j])
            curCopy = self.inversion(current, i, j)
            count += 1
            neighbors.append(curCopy)
    return neighbors
```

6) inversion(current, i, j)

I와 j를 통해 current 값에 변화를 주고 이 변화한 값들을 return한다

```
def inversion(self, current, i, j):  # Perform inversion
    curCopy = current[:] # 리스트 복사
    while i < j:
        curCopy[i], curCopy[j] = curCopy[j], curCopy[i]
        i += 1
        j -= 1
    return curCopy
```

7) randomMutant(current)

First-choice를 위해 필요한 메소드이다.

```
#first-choice
def randomMutant(self, current): # Apply inversion
    while True:
        i, j = sorted([random.randrange(self._numCities)
                       for _ in range(2)])
        if i < j:
            curCopy = self.inversion(current, i, j)
            break
    return curCopy
```

8) describe

문제를 설명해주는 메소드이다.

```
def describe(self):
    print()
    n = self._numCities
    print("Number of cities:", n)
    print("City locations:")
    locations = self._locations
    for i in range(n):
        print("{0:>12}".format(str(locations[i])), end = '')
        if i % 5 == 4:
            print()
```

9) report

Solution이 무엇인지 출력해주는 메소드이다. 이때 problem class의 report 메소드가 사용된다

```
def report(self):
    print()
    print("Best order of visits:")
    self.tenPerRow()       # Print 10 cities per row
    print("Minimum tour cost: {0:,}".format(round(self._value)))
    Problem.report(self)
```

10) tenPerRow

Solution의 값을 한 줄에 10개씩 출력되도록 하는 메소드이다.

```
def tenPerRow(self):
    solution = self._solution
    for i in range(len(solution)):
        print("{0:>5}".format(solution[i]), end='')
        if i % 10 == 9: # 10개씩 출력 -> 줄바꿈
            print()
```

[problem.py] 전체 코드

```
import random
import math

# class에서는 함수가 아니라 메소드를 사용하는것임!! 혼동하지 말것

class Problem:
    def __init__(self):
        self._solution = []
        self._value = 0
        self._numEval = 0

    def setVariables(self):
        pass

    def randomInit(self):
        pass

    def evaluate(self):
        pass

    def mutants(self):
        pass

    def randomMutant(self, current):
        pass

    def describe(self):
        pass

    def storeResult(self, solution, value):
        self._solution = solution
        self._value = value

    def report(self):
        print()
        print("Total number of evaluations: {0:,}".format(self._numEval))
```

```python
class Numeric(Problem):
    def __init__(self):
        Problem.__init__(self)
        self._expression = ''
        self._domain = []     # domain as a list
        self._delta = 0.01    # Step size for axis-parallel mutation

        self._alpha = 0.01    # Update rate for gradient descent
        self._dx = 10 ** (-4) # Increment for calculating derivative

    def setVariables(self):
        filename = input("Enter the file name of a function: ")
        infile = open(filename, "r")
        self._expression = infile.readline()
        varNames, low, up = [], [], [] # 각각 변수 이름, 최소값, 최대값을 저장할 리스트
        line = infile.readline()

        while line != "":
            data = line.split(",")
            varNames.append(data[0])
            low.append(eval(data[1]))
            up.append(eval(data[2]))
            line = infile.readline()
        infile.close()
        self._domain = [varNames, low, up]

    def getDelta(self): # new method
        return self._delta

    def describe(self): # describeProblem
        print()
        print("Objective function:")
        print(self._expression)
        print("Search space:")
        varNames = self._domain[0]
        low = self._domain[1]
        up = self._domain[2]
        for i in range(len(low)):
            print(" " + varNames[i] + ":", (low[i], up[i]))
```

```
def report(self):
    print()
    print("Solution found: ")
    print(self.coordinate())
    print("Minimum value: {0:,.3f}".format(self._value))
    Problem.report(self)

def coordinate(self):
    c = [round(value,3) for value in self._solution]
    return tuple(c)

def randomInit(self):
    domain = self._domain
    low, up = domain[1], domain[2]
    init = []

    for i in range(len(low)):
        r = random.uniform(low[i], up[i])
        init.append(r)
    return init

def evaluate(self, current):
    self._numEval += 1 # 총 eval 횟수 하나식 증가
    expr = self._expression
    varNames = self._domain[0]
    for i in range(len(varNames)):
        assignment = varNames[i] + '=' + str(current[i]) # 변수명 = '값' 형태로 출력되게 설정
        exec(assignment) # 설정된 값을 실행
    return eval(expr)

def mutate(self, current, i, d):
    mutant = current[:]
    domain = self._domain
    l= domain[1][i]
    u = domain[2][i]
    if l <= (mutant[i] + d) <= u:
        mutant[i] += d
    return mutant
```

```
# steepest-ascent
def mutants(self, current): # steepest ascent (n)에서 Numeric problem 관련 함수를 numeric class로 이동
    neighbors = []
    for i in range(len(current)): # For each variable
        mutant = self.mutate(current, i, self._delta)
        neighbors.append(mutant)
        mutant = self.mutate(current, i, -self._delta)
        neighbors.append(mutant)
    return neighbors

# first-choice
def randomMutant(self, current): ###
    i = random.randint(0, len(current) - 1)  # 몇 번째 var를 변경할 것인지를 랜덤하게 지정
    if random.uniform(0,1) < 0.5: # 이때 D는 0.5의 확률로 증가할지 감소할지 결정되어야함
        d = self._delta
    else:
        d = - self._delta

    return self.mutate(current, i, d)

# gradient descent
def getAlpha(self): # self._alpha Accessor
    return self._alpha

def getDx(self): # self._dx Accessor
    return self._dx

# gradient를 통해 next step을 계산
# next step이 domain 범위 이내 일 때만 next step을 반환
def takeStep(self, currentP, valueC):
    next = currentP[:] # 리스트 복사
    gradient = self.gradient(currentP,valueC) # gradient 값 계산
    for i in range(len(next)): # 각각의 변수에 대해
        next[i] -= self._alpha * gradient[i] # gradient 값에 따라 다음 step을 계산 -> w' = w - alpha*f'(w)
    if self.isLegal(next): # 새로 만들어진 값이 범위 내에 있다면
        return next # 새로운 값을 반환
    else:
        return currentP # 범위 밖이면 원래 값을 반환
```

```
# 주어진 변수 값들(x)이 도메인 범위 이내인지 확인하고 T/F를 반환
def isLegal(self, x):
    low = self._domain[1]
    up = self._domain[2]
    for i in range(len(x)): # 각 변수에 대해 실행
        if(low[i] > x[i] or up[i] < x[i]): # 범위 밖이면
            return False # False 반환
    return True # 범위 안이면 True 반환

# '각 변수'의 gradient를 list형으로 반환
def gradient(self, x, v): # x는 현재 변수 값, v는 현재 값 list
    gradient = [] # gradient 값을 저장할 리스트
    for i in range(len(x)): # 각 변수에 대해
        changedX = x[:] # 리스트 복사
        changedX[i] += self._dx # i번째 변수에 f(x+dx) 해주기
        gradient.append((self.evaluate(changedX) - v) / self._dx) # f'(x) = (f(x+dx) - f(x)) / dx 계산
    return gradient # gradient 값 반환
```

```python
class Tsp(Problem):
    def __init__(self):
        Problem.__init__(self)
        self._numCities = 0
        self._locations = []       # A list of tuples
        self._distanceTable = []

    def setVariables(self):
        ## Read in a TSP (# of cities, locatioins) from a file.
        ## Then, create a problem instance and return it.
        fileName = input("Enter the file name of a TSP: ")
        infile = open(fileName, 'r')
        # First line is number of cities
        self._numCities = int(infile.readline())
        locations = []
        line = infile.readline()  # The rest of the lines are locations
        while line != '':
            locations.append(eval(line)) # Make a tuple and append
            line = infile.readline()
        infile.close()
        self._locations = locations
        self._distanceTable = self.calcDistanceTable()

    def calcDistanceTable(self):
            n = self._numCities
            locations = self._locations
            table = [[0] * n for _ in range(n)] # 2차원 리스트 초기화

            for i in range(n):
                for j in range(n):
                    if i != j:  # 같은 도시 간 거리는 계산하지 않음
                        table[i][j] = math.sqrt((locations[j][0] - locations[i][0]) ** 2 +
                                                (locations[j][1] - locations[i][1]) ** 2) # 두 점 사이의 거리를 구하는 공식 사용해서 table에 저장
            return table  # A symmetric matrix of pairwise distances

    def randomInit(self):   # Return a random initial tour
        n = self._numCities
        init = list(range(n))
        random.shuffle(init)
        return init
```

```python
def evaluate(self, current): ###
    ## Calculate the tour cost of 'current'
    ## 'p' is a Problem instance
    ## 'current' is a list of city ids
    self._numEval += 1 # 총 eval횟수 늘이기
    n = self._numCities
    table = self._distanceTable
    cost = 0 # 비용 초기화

    # 첫 도시부터 맨 마지막 도시까지 투어하면서 distance table을 참고하여 비용 계산
    for i in range(n-1):
        locFrom = current[i]
        locTo = current[i+1]
        cost += table[locFrom][locTo]
    # 맨 마지막 도시에서 처음 도시로 돌아오는 비용을 고려해서 cost에 더해줌
    cost += table[current[-1]][current[0]]

    return cost

#steepest-ascent
def mutants(self, current): # Apply inversion
    n = self._numCities
    neighbors = []
    count = 0
    triedPairs = []
    while count <= n:  # Pick two random loci for inversion
        i, j = sorted([random.randrange(n) for _ in range(2)])
        if i < j and [i, j] not in triedPairs:
            triedPairs.append([i, j])
            curCopy = self.inversion(current, i, j)
            count += 1
            neighbors.append(curCopy)
    return neighbors

def inversion(self, current, i, j):  # Perform inversion
    curCopy = current[:] # 리스트 복사
    while i < j:
        curCopy[i], curCopy[j] = curCopy[j], curCopy[i]
        i += 1
        j -= 1
    return curCopy
```

```
#first-choice
def randomMutant(self, current): # Apply inversion
    while True:
        i, j = sorted([random.randrange(self._numCities)
                       for _ in range(2)])
        if i < j:
            curCopy = self.inversion(current, i, j)
            break
    return curCopy

def describe(self):
    print()
    n = self._numCities
    print("Number of cities:", n)
    print("City locations:")
    locations = self._locations
    for i in range(n):
        print("{0:>12}".format(str(locations[i])), end = '')
        if i % 5 == 4:
            print()

def report(self):
    print()
    print("Best order of visits:")
    self.tenPerRow()       # Print 10 cities per row
    print("Minimum tour cost: {0:,}".format(round(self._value)))
    Problem.report(self)

def tenPerRow(self):
    solution = self._solution
    for i in range(len(solution)):
        print("{0:>5}".format(solution[i]), end='')
        if i % 10 == 9: # 10개씩 출력 -> 줄바꿈
            print()
```

2.2. gradient descent.py

코드가 주어졌기 때문에 전체 코드 캡쳐만 하였다.

```
from problem import Numeric


def main():
    # Create a Problme object for numerical optimization
    p = Numeric()    # Create a problem object
    p.setVariables() # Set its class variables (expression, domain)
    # Call the search algorithm
    gradientDescent(p)
    # Show the problem and algorithm settings
    p.describe()
    displaySetting(p)
    # Report results
    p.report()

def gradientDescent(p):
    currentP = p.randomInit()  # Current point
    valueC = p.evaluate(currentP)
    while True:
        nextP = p.takeStep(currentP, valueC)
        valueN = p.evaluate(nextP)
        if valueN >= valueC:
            break
        else:
            currentP = nextP
            valueC = valueN
    p.storeResult(currentP, valueC)

def displaySetting(p):
    print()
    print("Search algorithm: Gradient Descent")
    print()
    print("Udate rate:", p.getAlpha())
    print("Increment for calculating derivative:", p.getDx())

main()
```

2.3부터 2.6까지의 코드는 지난번 과제인 HW05와 동일하되 main에서 problem.py에서 구현된 메소드를 사용하고 method화 하지 못한 코드는 그대로 함수로 남아있다는 점만 다르기 때문에 설명은 생략하고 코드만 첨부하였다

2.3 steepest ascent (n).py

```python
from problem import Numeric

def main():
    p = Numeric()
    p.setVariables()
    steepestAscent(p) # 이건 아직 class가 아니니까 함수
    p.describe()
    displaySetting(p) # 이것도 아직 class가 아니니까 함수
    p.report()

def steepestAscent(p):
    current = p.randomInit() # 'current' is a list of values
    valueC = p.evaluate(current)
    while True:
        neighbors = p.mutants(current)
        successor, valueS = bestOf(neighbors, p)
        if valueS >= valueC:
            break
        else:
            current = successor
            valueC = valueS
    p.storeResult(current, valueC)


def bestOf(neighbors, p): ###
    best = neighbors[0]
    bestValue = p.evaluate(best)
    for i in range(1, len(neighbors)):
        newValue = p.evaluate(neighbors[i])
        if newValue < bestValue:
            best = neighbors[i]
            bestValue = newValue

    return best, bestValue

def displaySetting(p):
    print()
    print("Search algorithm: Steepest-Ascent Hill Climbing")
    print()
    print("Mutation step size:", p.getDelta())

main()
```

2.4 first-choice (n).py

```python
from problem import Numeric

LIMIT_STUCK = 100 # Max number of evaluations enduring no improvement


def main():
    p = Numeric()
    p.setVariables()
    firstChoice(p) # 이건 아직 class가 아니니까 함수
    p.describe()
    displaySetting(p) # 이것도 아직 class가 아니니까 함수
    p.report()

def firstChoice(p):
    current = p.randomInit()   # 'current' is a list of values
    valueC = p.evaluate(current)
    i = 0
    while i < LIMIT_STUCK:
        successor = p.randomMutant(current)
        valueS = p.evaluate(successor)
        if valueS < valueC:
            current = successor
            valueC = valueS
            i = 0              # Reset stuck counter
        else:
            i += 1
    p.storeResult(current, valueC)

def displaySetting(p):
    print()
    print("Search algorithm: First-Choice Hill Climbing")
    print()
    print("Mutation step size:", p.getDelta())

main()
```

2.5 steepest ascent (tsp).py

```python
from problem import Tsp

def main():
    p = Tsp()
    p.setVariables()
    steepestAscent(p) # 이건 아직 class가 아니니까 함수
    p.describe()
    displaySetting() # 이것도 아직 class가 아니니까 함수
    p.report()

def steepestAscent(p):
    current = p.randomInit()   # 'current' is a list of city ids
    valueC = p.evaluate(current)
    while True:
        neighbors = p.mutants(current)
        (successor, valueS) = bestOf(neighbors, p)
        if valueS >= valueC: # 더 좋은 이웃이 없으면
            break # 종료
        else: # 더 좋아지면
            current = successor # 현재값을 업데이트
            valueC = valueS
    p.storeResult(current, valueC)

def bestOf(neighbors,p): ###
    best = neighbors[0]
    bestValue = p.evaluate(neighbors[0]) # 첫번째 값으로 초기화
    for i in range(1, len(neighbors)): # 두번째 값부터 비교
        tmpResult = p.evaluate(neighbors[i]) # 후보들의 값을 계산
        if tmpResult < bestValue: # 더 좋은 값이 나오면 바꿈 (값이 작을 수록 좋은 것임)
            best = neighbors[i]
            bestValue = tmpResult
    return best, bestValue

def displaySetting():
    print()
    print("Search algorithm: Steepest-Ascent Hill Climbing")

main()
```

2.6 first-choice (tsp).py

```python
from problem import Tsp

LIMIT_STUCK = 100 # Max number of evaluations enduring no improvement


def main():
    p = Tsp()
    p.setVariables()
    firstChoice(p)
    p.describe()
    displaySetting()
    p.report()

def firstChoice(p):
    current = p.randomInit()   # 'current' is a list of city ids
    valueC = p.evaluate(current)
    i = 0
    while i < LIMIT_STUCK:
        successor = p.randomMutant(current) # 값을 random하게 바꿔줌
        valueS = p.evaluate(successor)
        if valueS < valueC: # 더 좋은 이웃이 있으면
            current = successor
            valueC = valueS # 현재값을 업데이트
            i = 0              # Reset stuck counter
        else:
            i += 1
    p.storeResult(current, valueC)

def displaySetting():
    print()
    print("Search algorithm: First-Choice Hill Climbing")

main()
```

2.7. 실행 결과

1) problem/Convex.txt

Steepest ascent

```
Enter the file name of a function: problem/Convex.txt

Objective function:
(x1 - 2) ** 2 +5 * (x2 - 5) ** 2 + 8 * (x3 + 8) ** 2 + 3 * (x4 + 1) ** 2 + 6 * (x5 - 7) ** 2

Search space:
 x1: (-30, 30)
 x2: (-30, 30)
 x3: (-30, 30)
 x4: (-30, 30)
 x5: (-30, 30)

Search algorithm: Steepest-Ascent Hill Climbing

Mutation step size: 0.01

Solution found:
(2.004, 4.997, -8.0, -1.0, 7.0)
Minimum value: 0.000

Total number of evaluations: 64,791
```

First-choice

```
Enter the file name of a function: problem/Convex.txt

Objective function:
(x1 - 2) ** 2 +5 * (x2 - 5) ** 2 + 8 * (x3 + 8) ** 2 + 3 * (x4 + 1) ** 2 + 6 * (x5 - 7) ** 2

Search space:
 x1: (-30, 30)
 x2: (-30, 30)
 x3: (-30, 30)
 x4: (-30, 30)
 x5: (-30, 30)

Search algorithm: First-Choice Hill Climbing

Mutation step size: 0.01

Solution found:
(2.001, 4.997, -7.997, -1.004, 7.003)
Minimum value: 0.000

Total number of evaluations: 33,807
```

Gradient descent

```
Enter the file name of a function: problem/Convex.txt

Objective function:
(x1 - 2) ** 2 +5 * (x2 - 5) ** 2 + 8 * (x3 + 8) ** 2 + 3 * (x4 + 1) ** 2 + 6 * (x5 - 7) ** 2

Search space:
 x1: (-30, 30)
 x2: (-30, 30)
 x3: (-30, 30)
 x4: (-30, 30)
 x5: (-30, 30)

Search algorithm: Gradient Descent

Udate rate: 0.01
Increment for calculating derivative: 0.0001

Solution found:
(2.0, 5.0, -8.0, -1.0, 7.0)
Minimum value: 0.000

Total number of evaluations: 10,531
```

2) problem/tsp50.txt

Steepest ascent

```
Enter the file name of a TSP: problem/tsp50.txt

Number of cities: 50
City locations:
     (1, 7)    (14, 92)    (45, 97)    (17, 60)    (22, 44)
    (4, 38)    (13, 73)    (79, 68)    (76, 95)    (62, 14)
   (25, 75)     (26, 9)    (88, 81)    (56, 65)    (64, 71)
   (92, 20)     (7, 20)     (8, 20)    (61, 39)    (17, 11)
   (10, 40)    (18, 72)    (89, 72)    (58, 25)    (57, 57)
   (66, 70)    (36, 72)    (89, 91)    (18, 90)    (72, 49)
   (82, 38)    (22, 26)    (36, 56)    (23, 44)    (45, 45)
    (7, 27)     (84, 6)    (32, 78)     (0, 29)    (64, 63)
   (45, 24)    (21, 81)    (37, 16)    (86, 57)    (65, 99)
   (25, 53)    (98, 24)    (83, 81)     (50, 5)    (58, 80)

Search algorithm: Steepest-Ascent Hill Climbing

Best order of visits:
    9   42   48   19   17   16    0   38   35   31
   34   32   44   49    2    1   28   26   10   37
   41    6   21    3   45   33    4    5   20   11
   40   30   29   22   47   12   27    8   14   25
    7   43   39   13   24   18   15   36   46   23
Minimum tour cost: 873

Total number of evaluations: 2,245
```

First-choice

```
Enter the file name of a TSP: problem/tsp50.txt

Number of cities: 50
City locations:
     (1, 7)    (14, 92)    (45, 97)    (17, 60)    (22, 44)
    (4, 38)    (13, 73)    (79, 68)    (76, 95)    (62, 14)
   (25, 75)     (26, 9)    (88, 81)    (56, 65)    (64, 71)
   (92, 20)     (7, 20)     (8, 20)    (61, 39)    (17, 11)
   (10, 40)    (18, 72)    (89, 72)    (58, 25)    (57, 57)
   (66, 70)    (36, 72)    (89, 91)    (18, 90)    (72, 49)
   (82, 38)    (22, 26)    (36, 56)    (23, 44)    (45, 45)
    (7, 27)     (84, 6)    (32, 78)     (0, 29)    (64, 63)
   (45, 24)    (21, 81)    (37, 16)    (86, 57)    (65, 99)
   (25, 53)    (98, 24)    (83, 81)     (50, 5)    (58, 80)

Search algorithm: First-Choice Hill Climbing

Best order of visits:
   19    0   16   17   38   35   48   36   15   46
   30   43   22   12   27    8   49   13   24   34
   18   29   39    7   14   25   47   44    2    1
   28   41    3   21    6   10   37   26   33   20
    5   32   45    4   31   40   23    9   42   11
Minimum tour cost: 789

Total number of evaluations: 1,369
```

3. 결론

캡쳐된 결과를 보면 numeric의 경우 평가 횟수는 First choice < Steepest ascent < Gradient descent 순서임을 알 수 있고, 이때 minimum value는 모두 0으로 동일하다는 것을 확인할 수 있다. tsp의 경우 평가 횟수는 First choice < Steepest ascent 순서인데 저번 과제에서 설명했듯 first choice가 연산 횟수가 더 적기 때문이다.

이번 과제에서는 class를 이용하여 코드를 작성해보았는데 class를 사용해보니 지난 과제를 할 때 생각했던 중복되는 코드의 문제를 해결할 수 있었다. 지난번 과제를 할 때 겹치는 코드가 꽤 있어 비효율적이라고 생각했는데 이번 과제를 통해 이 문제를 해결할 수 있었고, class를 사용하면 코드의 유지보수에도 유리할 것 같다고 생각했다.

또한 gradient descent를 처음 사용해보았는데, 보고서에는 캡쳐하지는 않았지만, a값이 변화함에 따라 evaluate하는 횟수가 크게 변화하는 것을 확인할 수 있었다. 즉, 이 결과를 통해 gradient descent 방법을 사용할 때는 a값이 결과에 큰 영향을 주기 때문에 적절한 a값을 찾는 것이 중요하다고 생각했다.