AIP

HW 05 – REPORT

소속 :정보컴퓨터공학부

학번 : 202255535

이름 : 김진우

날짜 : 24.11.03

1. 서론

Conventional Algorithm vs AI Algorithm

Conventional Algorithm은 정확한 답을 찾을 수 있다는 장점이 있지만, TSP같은 복잡한 문제는 시간이 너무 오래 걸려 해결할 수 없는 경우가 생긴다는 단점이 있다. 이 문제를 해결할 수 있는 것이 AI Algorithm인데 이는 optimal solution을 보장하지는 못하지만 그에 근접한 답을 훨씬 빠른 시간 안에 찾아낼 수 있다는 장점을 가진다.

AI 알고리즘 중 가장 대표적인 것이 Hill Climbing인데 이번 과제에서 이와 관련해 numeric과 tsp 문제를 풀어보며 이에 대해 학습할 수 있었다.

Hill Climbing 알고리즘 (Steepest / First Choice)

Hill-climbing 알고리즘은 목표 상태에 도달하기 위해 현재 상태에서 점진적으로 더 나은 상태로 이동하는 탐색 방법이다. 이 알고리즘에는 대표적으로 Steepest Ascent와 First Choice 방법이 있다.

- Steepest Ascent 방법: random하게 solution을 하나 만들고 현재 solution에서부터 현재 상태 주변의 모든 가능한 상태를 평가하여, 그중 가장 좋은 상태로 이동하는 방식이다. 이 방법은 모든 이웃 상태를 고려해 최선의 방향으로 진행한다는 특징이 있다.

- First Choice 방법: random하게 solution을 하나 만들고 현재 상태보다 더 나은 상태가 발견되면 즉시 그 상태로 이동하는 방식이다. 이는 steepest-Ascent와는 다르게 모든 이웃 상태를 살펴보지 않고 더 나은 상태를 찾을 때마다 이동하기 때문에 계산양이 줄어드는 장점이 있다.

Numeric / TSP

Numeric은 주어진 expression의 값을 가장 작게 하는 변수를 구하는 문제이고 TSP는 여러 도시를 모두 방문할 때 최소한의 cost가 들게 하는 순서를 구하는 문제이다.

2. 본론

2.1 tsp.py

tsp.py에서는 steepest ascent(tsp)와 first-choice(tsp)를 풀 때 공통으로 사용되는 함수와 global 변수 와 같은 것들이 선언되어 있다. NumEval은 evaluation의 수를 의미하는데 여기서는 evaluate 함수 호출 횟수를 의미한다.

주요 함수들을 하나하나 살펴보면 아래와 같다.

- **createProblem()** : txt파일을 읽어와서 문제를 만드는 함수이다.

- **calcDistanceTable(numCities, locations)** : 도시 간의 거리를 계산해서 이를 2차원 matrix으로 만들어준다. 이때 도시 사이의 거리는 거리공식을 사용해 구한다.

```
def calcDistanceTable(numCities, locations): ###
    table = [[0 for _ in range(numCities)] for _ in range(numCities)]
    for i in range(numCities):   #첫번째(index 0)도시부터 이중 for문 돌려가며 다른 도시와의 거리를 두 점 사이 거리공식 사용해서 구하기
        for j in range(numCities):
            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 사용
    return table # A symmetric matrix of pairwise distances
```

- **randomInit(p)** : random하게 초기 solution을 하나 만들어준다.

- **evaluate(current, p)** : solution의 cost를 계산하는 함수이다. 이 함수가 실행되면 current의 순서대로 cost가 계산되게 된다. 이때 이전에 만들어둔 2차원 matrix인 table을 사용하게 되는데 table[i][i+1]을 계속해서 더하다 보면 전체 cost를 구할 수 있게 된다.

```
def evaluate(current, p): ###
    ## Calculate the tour cost of 'current'
    ## 'p' is a Problem instance
    ## 'current' is a list of city ids
    global NumEval # 총 eval횟수 체크
    NumEval += 1

    n = p[0]
    table = p[2] # p[2] = distance table임
    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
```

- **inversion(current, i, j)** : 도시의 방문위치를 바꿔준다.

[전체 코드]

```
import random
import math

NumEval = 0    # Total number of evaluations


def createProblem():
    ## 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
    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()
    table = calcDistanceTable(numCities, locations)
    return numCities, locations, table

def calcDistanceTable(numCities, locations): ###
    table = [[0 for _ in range(numCities)] for _ in range(numCities)]
    for i in range(numCities):   #첫번째(index 0)도시부터 이중 for문 돌려가며 다른 도시와의 거리를 두 점 사이 거리공식 사용해서 구하기
        for j in range(numCities):
            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 사용
    return table # A symmetric matrix of pairwise distances

def randomInit(p):   # Return a random initial tour
    n = p[0]
    init = list(range(n))
    random.shuffle(init)
    return init
```

```python
def evaluate(current, p): ###
    ## Calculate the tour cost of 'current'
    ## 'p' is a Problem instance
    ## 'current' is a list of city ids
    global NumEval # 총 eval횟수 체크
    NumEval += 1

    n = p[0]
    table = p[2] # p[2] = distance table임
    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


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

def describeProblem(p):
    print()
    n = p[0]
    print("Number of cities:", n)
    print("City locations:")
    locations = p[1]
    for i in range(n):
        print("{0:>12}".format(str(locations[i])), end = '')
        if i % 5 == 4:
            print()
```

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

def describeProblem(p):
    print()
    n = p[0]
    print("Number of cities:", n)
    print("City locations:")
    locations = p[1]
    for i in range(n):
        print("{0:>12}".format(str(locations[i])), end = '')
        if i % 5 == 4:
            print()

def displayResult(solution, minimum):
    print()
    print("Best order of visits:")
    tenPerRow(solution)       # Print 10 cities per row
    print("Minimum tour cost: {0:,}".format(round(minimum)))
    print()
    print("Total number of evaluations: {0:,}".format(NumEval))

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

2.1.1. first-choice (tsp)

First-choice(tsp).py는 tsp.py를 import한다.

여기서 중요한 함수는 first-choice와 randomMutant함수이다.

[전체 코드]

```
from tsp import *

LIMIT_STUCK = 100 # Max number of evaluations enduring no improvement


def main():
    # Create an instance of TSP
    p = createProblem()    # 'p': (numCities, locations, distanceTable)
    # Call the search algorithm
    solution, minimum = firstChoice(p)
    # Show the problem and algorithm settings
    describeProblem(p)
    displaySetting()
    # Report results
    displayResult(solution, minimum)

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

def randomMutant(current, p): # Apply inversion
    while True:
        i, j = sorted([random.randrange(p[0])
                       for _ in range(2)])
        if i < j:
            curCopy = inversion(current, i, j)
            break
    return curCopy

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

main()
```

2.1.2. steepest ascent (tsp)

Steepest ascent (tsp).py 역시 tsp.py를 import한다.

여기서 구현해야 하는 주요 함수는 bestOf(neighbors,p)이다.

- **bestOf(neighbors, p)** : 이 함수는 가장 좋은 값을 가지는 neighbor를 리턴하는 함수인데 값이 좋다는 것은 value가 더 작은 상황을 의미한다. 이때 여기서는 tsp.py의 evaluate 함수를 사용한다.

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

[전체 코드]

```
from tsp import *

def main():
    # Create an instance of TSP
    p = createProblem()    # 'p': (numCities, locations, table)
    # Call the search algorithm
    solution, minimum = steepestAscent(p)
    # Show the problem and algorithm settings
    describeProblem(p)
    displaySetting()
    # Report results
    displayResult(solution, minimum)

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

def mutants(current, p): # Apply inversion
    n = p[0]
    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 = inversion(current, i, j)
            count += 1
            neighbors.append(curCopy)
    return neighbors
```

```
def bestOf(neighbors, p): ###
    best = neighbors[0]
    bestValue = evaluate(neighbors[0], p) # 첫번째 값으로 초기화
    for i in range(1, len(neighbors)): # 두번째 값부터 비교
        tmpResult = evaluate(neighbors[i], p) # 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()
```

2.2 Numeric.py

Numeric.py에는 steepest ascent(n)과 first-choice(n)을 풀 때 공통으로 사용되는 함수와 global 변수 와 같은 것들이 선언되어 있다. DELTA값은 step-size를 의미하고, NumEval은 evaluation의 수를 의미하는데 여기서는 evaluate함수 호출 횟수를 의미한다.

함수 하나하나를 따져보면 아래와 같다.

- **createProblem()** : txt파일을 읽어와서 이를 return 해준다. 이때 return 값인 expression은 함수식이고, domain은 varNames, low, up으로 구성된 list로 각각은 변수이름, 상한값, 하한값을 의미한다.

```
def createProblem(): ###
    ## Read in an expression and its domain from a file.
    ## Then, return a problem 'p'.
    ## 'p' is a tuple of 'expression' and 'domain'.
    ## 'expression' is a string.
    ## 'domain' is a list of 'varNames', 'low', and 'up'.
    ## 'varNames' is a list of variable names.
    ## 'low' is a list of lower bounds of the varaibles.
    ## 'up' is a list of upper bounds of the varaibles.
    filename = input("Enter the file name of a function: ")
    infile = open(filename, "r")
    expression = infile.readline().rstrip()
    varNames, low, up = [], [], [] # 각각 변수 이름, 최소값, 최대값을 저장할 리스트

    while(True):
        varinfo = infile.readline()
        if varinfo == "":
            break
        varinfo = varinfo.rstrip().split(",") # (변수이름, low, up) 형식으로 저장된 정보를 가져온다
        varNames.append(varinfo[0]) #
        low.append(eval(varinfo[1]))
        up.append(eval(varinfo[2]))
    domain = [varNames, low, up]
    return expression, domain
```

- **randomInit(p)** : 변수들의 값을 초기화해주는 함수로 random.uniform을 통해 범위 안의 랜덤한 x 값을 가져온다.

```
def randomInit(p): ###
    init = []
    for i in range(len(p[1][0])): # 변수의 수만큼 무작위로 생성하기
        low = p[1][1][i]
        up = p[1][2][i]
        init.append(random.uniform(low, up)) #각 변수의 max min값을 알아낸뒤 random.uniform을 사용해 임의의 실수 저장
    return init    # Return a random initial point
                   # as a list of values
```

- **evaluate(current, p)** : 변수의 값을 통해 이를 evaluate하고 NumEval을 계산

- **mutate(current, i, d, p)** : 현재 값으로부터 값을 변형

- **describeProblem(p)** : 읽어온 파일을 출력

- **displayResult(solution, minimum) :** solution을 터미널에 출력

[전체 코드]

```
import random

DELTA = 0.01   # Mutation step size
NumEval = 0    # Total number of evaluations


def createProblem(): ###
    ## Read in an expression and its domain from a file.
    ## Then, return a problem 'p'.
    ## 'p' is a tuple of 'expression' and 'domain'.
    ## 'expression' is a string.
    ## 'domain' is a list of 'varNames', 'low', and 'up'.
    ## 'varNames' is a list of variable names.
    ## 'low' is a list of lower bounds of the varaibles.
    ## 'up' is a list of upper bounds of the varaibles.
    filename = input("Enter the file name of a function: ")
    infile = open(filename, "r")
    expression = infile.readline().rstrip()
    varNames, low, up = [], [], [] # 각각 변수 이름, 최소값, 최대값을 저장할 리스트

    while(True):
        varinfo = infile.readline()
        if varinfo == "":
            break
        varinfo = varinfo.rstrip().split(",") # (변수이름, low, up) 형식으로 저장된 정보를 가져온다
        varNames.append(varinfo[0]) #
        low.append(eval(varinfo[1]))
        up.append(eval(varinfo[2]))
    domain = [varNames, low, up]
    return expression, domain

def randomInit(p): ###
    init = []
    for i in range(len(p[1][0])): # 변수의 수만큼 무작위로 생성하기
        low = p[1][1][i]
        up = p[1][2][i]
        init.append(random.uniform(low, up)) #각 변수의 max min값을 알아낸뒤 random.uniform을 사용해 임의의 실수 저장
    return init    # Return a random initial point
                   # as a list of values
```

```python
def evaluate(current, p):
    ## Evaluate the expression of 'p' after assigning
    ## the values of 'current' to the variables
    global NumEval

    NumEval += 1
    expr = p[0]         # p[0] is function expression
    varNames = p[1][0]  # p[1] is domain: [varNames, low, up]
    for i in range(len(varNames)):
        assignment = varNames[i] + '=' + str(current[i])
        exec(assignment)
    return eval(expr)

def mutate(current, i, d, p): ## Mutate i-th of 'current' if legal
    curCopy = current[:]
    domain = p[1]        # [VarNames, low, up]
    l = domain[1][i]     # Lower bound of i-th
    u = domain[2][i]     # Upper bound of i-th
    if l <= (curCopy[i] + d) <= u:
        curCopy[i] += d
    return curCopy

def describeProblem(p):
    print()
    print("Objective function:")
    print(p[0])   # Expression
    print("Search space:")
    varNames = p[1][0] # p[1] is domain: [VarNames, low, up]
    low = p[1][1]
    up = p[1][2]
    for i in range(len(low)):
        print(" " + varNames[i] + ":", (low[i], up[i]))

def displayResult(solution, minimum):
    print()
    print("Solution found:")
    print(coordinate(solution))  # Convert list to tuple
    print("Minimum value: {0:,.3f}".format(minimum))
    print()
    print("Total number of evaluations: {0:,}".format(NumEval))

def coordinate(solution):
    c = [round(value, 3) for value in solution]
    return tuple(c)  # Convert the list to a tuple
```

2.2.1. first-choice (n)

First-choice(n).py는 numeric.py를 import한다. 여기서 LIMIT_STUCK이라는 것이 사용되는데 이는 Max number of evaluations enduring no improvement를 뜻한다. 즉 LIMIT_STUCK의 수만큼 randomMutant함수를 실행하였음에도 불구하고 값이 improve되지 않았다면 종료되게 되는 것이다. 여기서 구현해야 하는 주요 함수는 randomMutant함수이다.

- **randomMutant(current, p)** : 이 함수를 통해 랜덤한 i번째 변수에 d만큼의 변형을 준다. 이때도 역시 numeric.py의 mutate함수를 통해 값을 변경하고 d는 문제에서 지정해준 대로 +DELTA/-DELTA중 하나로 선택된다.

```python
def randomMutant(current, p): ###
    i = random.randint(0, len(current) - 1)  # 몇 번째 var를 변경할 것인지를 랜덤하게 지정
    d = random.choice([DELTA, -DELTA]) # 어떻게 변경할 것인지도 랜덤하게 선택

    return mutate(current, i, d, p) # numeric.py의 mutate 함수를 사용해 값을 변경
```

[전체 코드]

```python
from numeric import *

LIMIT_STUCK = 100 # Max number of evaluations enduring no improvement


def main():
    # Create an instance of numerical optimization problem
    p = createProblem()   # 'p': (expr, domain)
    # Call the search algorithm
    solution, minimum = firstChoice(p)
    # Show the problem and algorithm settings
    describeProblem(p)
    displaySetting()
    # Report results
    displayResult(solution, minimum)

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

def randomMutant(current, p): ###
    i = random.randint(0, len(current) - 1)  # 몇 번째 var를 변경할 것인지를 랜덤하게 지정
    d = random.choice([DELTA, -DELTA]) # 어떻게 변경할 것인지도 랜덤하게 선택

    return mutate(current, i, d, p) # numeric.py의 mutate 함수를 사용해 값을 변경

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

main()
```

2.2.2. steepest ascent (n)

steepest ascent (n).py 역시 구현해둔 numeric.py를 import한다.

여기서 구현해야 하는 주요 함수는 mutants와 bestOf함수이다.

- **mutants(current, p)** : 이 함수는 변수의 현재 값에 DELTA만큼 더하거나 뺀 값으로 neighbors를 만든다. 이때 numeric.py에 선언해둔 mutate함수가 사용되는데 current는 현재 값, i는 몇 번째 변

수, d는 변형되는 정도, p는 주어진 변수들을 의미한다. mutate함수는 low, up 범위를 넘어서면 원래의 값을 그대로 리턴하기 때문에 범위를 벗어나지 않는 neighbors들만 저장되게 된다.

```
def mutants(current, p): ###
    neighbors = []
    for i in range(len(current)): # 모든 변수에 대해서 수행
        neighbors.append(mutate(current, i, DELTA, p)) # 변수의 현재 값에 DELTA만큼 더한 값을 저장
        neighbors.append(mutate(current, i, -DELTA, p)) # 변수의 현재 값에 DELTA만큼 뺀 값을 저장
    return neighbors     # Return a set of successors
```

- **bestOf(neighbors, p)** : 이 함수는 mutants를 통해 얻은 이웃들 중 가장 좋은 이웃을 찾는 함수이다. 이때 numeric.py의 evaluate함수를 사용한다.

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

    return best, bestValue
```

[전체 코드]

```
from numeric import *

def main():
    # Create an instance of numerical optimization problem
    p = createProblem()   # 'p': (expr, domain)
    # Call the search algorithm
    solution, minimum = steepestAscent(p)
    # Show the problem and algorithm settings
    describeProblem(p)
    displaySetting()
    # Report results
    displayResult(solution, minimum)

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

def mutants(current, p): ###
    neighbors = []
    for i in range(len(current)): # 모든 변수에 대해서 수행
        neighbors.append(mutate(current, i, DELTA, p)) # 변수의 현재 값에 DELTA만큼 더한 값을 저장
        neighbors.append(mutate(current, i, -DELTA, p)) # 변수의 현재 값에 DELTA만큼 뺀 값을 저장
    return neighbors     # Return a set of successors

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

    return best, bestValue
```

```
def displaySetting():
    print()
    print("Search algorithm: Steepest-Ascent Hill Climbing")
    print()
    print("Mutation step size:", DELTA)

main()
```

2.3 결과 화면

2.3.1. TSP문제 - First-choice/ Steepest Ascent

1) problem/tsp30.txt

[first-choice]

```
Search algorithm: First-Choice Hill Climbing

Best order of visits:
   18   22   27   17    4   26   16   24   12   10
    0    8   15   14   21    5   25    1    6    9
    2   23    7   28   11   29   20    3   13   19
Minimum tour cost: 566

Total number of evaluations: 852
```

[Steepest Ascent]

```
Search algorithm: Steepest-Ascent Hill Climbing

Best order of visits:
    0   10    8   15   25    1    6   26    4   11
   28   29    7   23    9    5   21   14    2   12
   16    3   20   13   19   18   22   17   24   27
Minimum tour cost: 642

Total number of evaluations: 745
```

2) problem/tsp50.txt

[first-choice]

```
Search algorithm: First-Choice Hill Climbing

Best order of visits:
    4   33    6   21   10   41   28    1   44   25
   43   22   12   27   47    8    7   29   18   23
   40   45   34   32   13    2   49   26   24   39
   14   37    3   20    5   38   35   16    0   19
   11   48   15   46   36   30    9   42   31   17
Minimum tour cost: 932

Total number of evaluations: 1,002
```

[Steepest Ascent]

```
Search algorithm: Steepest-Ascent Hill Climbing

Best order of visits:
    1   41   21    6    3   32   48   31   19   11
   42   40    4   20   17    0   16   35   38    5
   33   45   34   29   43   18   23    9   15   36
   46   30    7   25   39   14   49    2   44    8
   27   22   12   47   13   24   26   10   37   28
Minimum tour cost: 833

Total number of evaluations: 2,398
```

3) problem/tsp100.txt

[first-choice]

```
Search algorithm: First-Choice Hill Climbing

Best order of visits:
   60   59   16    9   93    4   68   90   65   83
   38   51   81   89    8   73   39   99   92   35
   31   53   44    0   42   25    2   13   79   82
   18   27    1   96   97   15   23    3   70   11
   37   24   98   22   34   46   20   47    7   45
   56   41   67   66    6   29   85   52   88   62
   40   84   94   28   48   69   10   30   36    5
   19   80   78   63   76   71   33   57   14   61
   87   74   12   32   21   86   26   95   17   77
   50   72   54   58   75   64   91   49   43   55
Minimum tour cost: 1,738

Total number of evaluations: 1,680
```

[Steepest Ascent]

```
Search algorithm: Steepest-Ascent Hill Climbing

Best order of visits:
   75   58   72    9   16   68   52   62    6   48
   99   73    8   15   18    1   27   67   96   23
   39   64   80   63   78   44   46   47   24   98
   34   22    7   36   35   92   97    5   79   11
   56   13    2   82   41   45   94   10   30   28
   69   66   84   40   88   85    4   29   89   90
   51   17   95   38   83   93   77   50   70    3
   55   43   42   86   57   32   74   21   87   65
   59   71   60   54   91   49   76   53   31   26
   33   61   14   12    0   20   25   19   37   81
Minimum tour cost: 1,424

Total number of evaluations: 8,687
```

2.3.2. Numeric 문제 - First-choice/ Steepest Ascent

1) problem/Ackely.txt

[First-choice]

```
Search algorithm: First-Choice Hill Climbing

Mutation step size: 0.01

Solution found:
(5.002, 12.003, -1.004, 24.0, -25.0)
Minimum value: 19.271

Total number of evaluations: 368
```

[Steepest ascent]

```
Search algorithm: Steepest-Ascent Hill Climbing

Mutation step size: 0.01

Solution found:
(27.004, 17.001, -21.999, 17.004, -29.99)
Minimum value: 19.808

Total number of evaluations: 1,001
```

2) problem/Convex.txt

[First-choice]

```
Search algorithm: First-Choice Hill Climbing

Mutation step size: 0.01

Solution found:
(1.999, 4.997, -8.001, -0.999, 6.999)
Minimum value: 0.000

Total number of evaluations: 32,705
```

[Steepest ascent]

```
Search algorithm: Steepest-Ascent Hill Climbing

Mutation step size: 0.01

Solution found:
(2.002, 5.003, -7.996, -1.0, 7.001)
Minimum value: 0.000

Total number of evaluations: 77,341
```

3) problem/Griewank.txt

[First-choice]

```
Search algorithm: First-Choice Hill Climbing

Mutation step size: 0.01

Solution found:
(3.143, -26.632, 27.171, -25.086, -0.001)
Minimum value: 0.522

Total number of evaluations: 2,924
```

[Steepest ascent]

```
Search algorithm: Steepest-Ascent Hill Climbing

Mutation step size: 0.01

Solution found:
(-12.564, -22.188, 5.438, 12.542, -14.019)
Minimum value: 0.259

Total number of evaluations: 6,541
```

3. 결론

<2.3>의 결과를 보면 tsp문제와 numeric문제 모두 first-choice와 steepest ascent 방법이 비슷한 solution을 찾아내지만, evaluation의 횟수가 first-choice방법이 steepest ascent 방법보다 훨씬 적다는 것을 확인할 수 있다. 이는 서론에서 말했듯 first-choice 방법은 steepest-Ascent와는 다르게 모든 이웃 상태를 살펴보지 않고 더 나은 상태를 찾을 때마다 이동하기 때문이라고 생각할 수 있다.

또한 코드를 돌릴 때마다 결과가 계속 다르게 나오고, 가끔씩 다른 답들과 많이 다른 결과가 나오기도 했다. 이는 아마도 알고리즘의 특성상 가장 가까운 상태들만 확인하기 때문에 local minimum 에 빠지는 등의 경우로 인해 발생한 결과라고 볼 수 있었다. 그럼에도 불구하고 많은 경우에 서로 유사한 결과가 나옴을 확인할 수 있었고, Conventional Algorithm 으로 시간 내에 해결할 수 없는 문제를 AI Algorithm 으로 해결할 수 있는 새로운 방법을 알게 되었다.