CS 330: Assignment 6

Due 7PM, Thursday, April 28

April 15, 2016

Collaborators Kristel Tan (ktan@bu.edu) & Joseph Cho (chojoe@bu.edu)

Question 4

Python Code

The following is our Python source code for this question. Before trying each heuristic, we initialize an array of 50 arrays with 100 random integers in each to run our tests on. Then, we perform each heuristic defined in separate functions on this set of arrays. The code has been commented for clarification on what data structures the individual functions are utilizing and manipulating.

```
'''

Kristel Tan (ktan@bu.edu), Joseph Cho (chojoe@bu.edu)
CAS CS330 Spring 2016 - Professor Byers
hw7 - hw7-question4.py

'''

import heapq
import math
import random
import pandas
import matplotlib.pyplot as plt

arrays = [] # Holds all the arrays
array = []  # Array that we're currently inserting

# Generate 50 random instances
for i in range(50):
    array = []              # Reinitialize to empty
    for j in range(100):    # Add sets of 100 integers chosen uniformly from [1, 10^12]
        array.append(random.randint(1, 10**12))
    arrays.append(array)

'''

Karmarkar-Karp Implementation

'''
def karmarkarKarp(A):
    a = A[:]
    heapq.heapify(a)
    # Sort the array of integers in descending order
    a.sort(reverse=True)
    while(len(a) > 1):
        a.sort(reverse=True)
        n1 = heapq.heappop(a)
        a.sort(reverse=True)
        n2 = heapq.heappop(a)
        heapq.heappush(a, abs(n1 - n2))

    return heapq.heappop(a)

kkArray = []

# Perform Karmarkar-Karp for each array in arrays
for a in arrays:
    kkArray.append(karmarkarKarp(a))

print('Karmarkar-Karp Solutions:', kkArray)


'''

Repeated Random Implementation

'''
def repeatedRandom(A, K):
    solutions = []
    current = []

    for k in range(K):
        s = []
        for i in range(len(A)):
            # Flip a coin to generate a sign
            coin = random.randint(1,10)
            if coin <= 5:
                s.append(1)
            elif coin > 5:
                s.append(-1)

        for j in range(len(s)):
            # Multiply the randomly generated sign array with the current
            # array being looked at
            current.append((A[j] * s[j]))
            # Append the current array to the solutions array
            solutions.append(abs(sum(current)))

    # Return the solution with the minimal residue
    return min(solutions)

rrArray = []

# Perform Repeated Random for each array in arrays
for a in arrays:
    rrArray.append(repeatedRandom(a, 25000))

print()
print('Repeated Random Solutions: ', rrArray)


'''
```

```
Gradient Descent Implementation

'''
def gradientDescent(A, K):
    s = []
    sPrime = []

    # Generate an initial random solution S
    for i in range(len(A)):
        coin = random.randint(1,10)
        if coin <= 5:
            s.append(1)
        elif coin > 5:
            s.append(-1)

    for k in range(K):
        sPrime = s[:]
        # Choose two random indices to define the swap
        i, j = random.sample(range(1, 100), 2)

        sPrime[i] *= -1            # Set s[i] to its negation with probability of 1
        coin = random.random()
        if coin < 0.5:
            sPrime[j] *= -1       # Set s[j] to its negation with probability of 0.5

        newS = [a*b for a,b in zip(A,s)]
        newSPrime = [a*b for a,b in zip(A,sPrime)]

        sRes = abs(sum(newS))
        sPrimeRes = abs(sum(newSPrime))
        if sPrimeRes < sRes:
            s = sPrime

    solution = [a*b for a,b in zip(A,s)]
    return abs(sum(solution))

gdArray = []

# Perform Gradient Descent for each array in arrays
for a in arrays:
    gdArray.append(gradientDescent(a, 25000))

print()
print('Gradient Descent Solutions: ', gdArray)


'''

Simulated Annealing Implementation

'''
def simulatedAnnealing(A, K):
    s = []
    sPrime = []

    # Generate an initial random solution S
    for i in range(len(A)):
        coin = random.randint(1,10)
        if coin <= 5:
            s.append(1)
        elif coin > 5:
            s.append(-1)

    for k in range(K):
        sPrime = s[:]
        # Choose two random indices to define the swap
        i, j = random.sample(range(1, 100), 2)

        sPrime[i] *= -1          # Set s[i] to its negation with probability of 1
        coin = random.random()
        if coin < 0.5:
            sPrime[j] *= -1      # Set s[j] to its negation with probability of 0.5

        newS = [a*b for a,b in zip(A,s)]
        newSPrime = [a*b for a,b in zip(A,sPrime)]

        sRes = abs(sum(newS))
        sPrimeRes = abs(sum(newSPrime))
        if sPrimeRes < sRes:
            s = sPrime
        else:
            temp = (10**10)*(0.8**(k/300))
            prob = math.exp(sPrime - sRes) / temp
            coin = random.random()
            if coin < prob:
                s = sPrime

    solution = [a*b for a,b in zip(A,s)]
    return abs(sum(solution))

saArray = []

# Perform Gradient Descent for each array in arrays
for a in arrays:
    saArray.append(gradientDescent(a, 25000))

print()
print('Simulated Annealing Solutions: ', saArray)

kkAvg = [sum(list(kkArray)[:x])/x for x in range(1, 51)]
rrAvg = [sum(list(rrArray)[:x])/x for x in range(1, 51)]
gdAvg = [sum(list(gdArray)[:x])/x for x in range(1, 51)]
saAvg = [sum(list(saArray)[:x])/x for x in range(1, 51)]

residues = {'Karmarkar-Karp': kkAvg, 'Repeated Random': rrAvg,
            'Gradient Descent': gdAvg, 'Simulated Annealing': saAvg}

pandas.DataFrame(residues).plot()
plt.show()
pandas.DataFrame(residues).plot(figsize=(12,10))
plt.savefig(‘hw7-question4-plot')
```

Sample Output

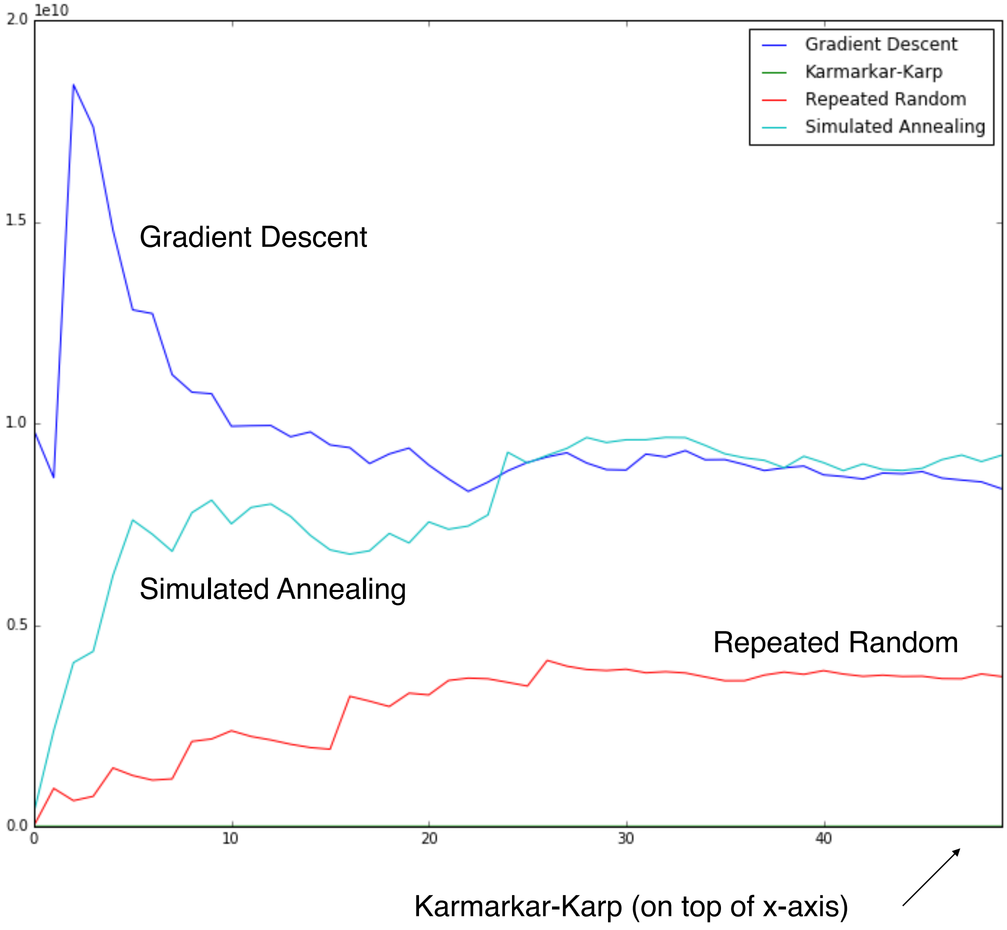
As seen in the code, after testing each heuristic, the program prints out the best residue for each array in the large set of arrays to the console. The following is a sample output from running the tests:

```
Karmarkar-Karp Solutions: [59264, 23041, 495946, 105827, 295326, 131059, 1758245,
78434, 83970, 249715, 912798, 75867, 577, 741486, 2827956, 73815, 93450, 48161,
1381057, 51838, 325665, 349343, 189624, 292904, 165782, 185628, 133947, 886157,
107646, 780989, 313259, 15310, 75289, 273902, 391941, 58067, 9532, 38876, 56232,
586817, 117163, 71131, 482393, 744304, 43809, 1280840, 1031294, 331301, 979759, 12404]

Repeated Random Solutions:  [156945666, 1751687705, 6363149328, 3114253555,
32504476935, 1792966401, 37280743122, 8763007820, 77305959283, 26653993308,
2270486627, 5105089075, 8276847745, 3064443837, 2617729230, 1822053908, 402564745,
66742819487, 1002084318, 5939509508, 27668611620, 35813822996, 4477894410,
33383976957, 9177431735, 7870135432, 142945133, 16739546330, 38969315, 43635076013,
109491842, 10221639416, 2262901534, 5323104408, 2779510251, 5803109156, 24777124598,
5918764331, 519210087, 4743091303, 680150724, 8091696343, 30705867552, 7057899505,
28175765707, 123602792013, 10536846823, 6665140824, 26624064329, 21604426176]

Gradient Descent Solutions:  [65111504100, 1417152091, 355978514294, 21762367495,
21080444418, 89702524959, 10611339655, 176616360596, 261840298946, 48014764177,
46487192392, 253836640323, 169591241917, 7293396066, 60368003340, 20618086429,
84462174142, 181696215135, 90602889363, 19548293786, 62896421529, 37668003317,
92272764188, 12798170160, 90968432834, 149835973080, 177483783461, 110578526873,
103688576404, 327256832789, 4376722035, 13300117950, 93786605137, 2627421314,
26151591391, 67796465889, 19253798914, 14962134316, 246174093392, 65490495955,
163329338519, 333013805761, 303599690733, 171839429706, 84780531293, 117570239312,
34607878606, 158539637345, 498054174365, 185872199434]

Simulated Annealing Solutions:  [42499136484, 74146178453, 134696720792, 20694262591,
98527362508, 262256243191, 297748105095, 2593062819806, 78272608734, 91878460987,
41916405632, 181814891391, 389589858939, 314315761296, 14439204158, 34382105201,
89563232954, 46411514621, 104966856819, 99677838476, 157874931377, 904838815873,
178622272434, 211024406610, 231548952480, 33212021544, 39857115381, 130681086711,
90204238106, 50233750759, 47900579589, 438286690868, 79748821911, 17062177192,
101325867119, 30739815575, 139782439474, 101006762096, 18416311960, 15955413883,
47044693761, 40248164721, 484594149629, 64965895910, 79527142473, 64694424606,
117794871094, 135192164343, 41317309379, 151573098760]
```

Graph of Residue Averages

A comparison graph of the average of these results is also generated using the Pandas and Matplotlib Python framework. Because the graph is printed in black and white, each individual line has been labeled for clarification of which line indicates the performance of which heuristic.

[See next page]



Explanation

- *Karmarkar-Karp* — As seen above, the performance of the Karmarkar-Karp differencing algorithm turned out to the best of all the heuristics. This was expected because it is the only method of the four that uses a consistent and optimal decision-making iteration. It continuously selects the two largest numbers in a given set of integers and replaces them with their difference until only the residue is left. This process also makes KK perform faster than other heuristics because it does not need to exhaustively search through the entire array to identify the two maximum elements. Instead, one can implement a priority queue or heap queue to simplify this search.

- *Repeated Random* — The average residues resulting from the Repeated Random heuristic was the second best performer of the four heuristics. We suspect that this occurred because the probability that the algorithm generates an optimal sign array increases as you increase the amount of k random solutions. It will then always select the sign array that results in the smallest residue. Although this process allows Repeated Random to have a wide variety of solutions to investigate, it does sacrifice running time as well. The more k random solutions we generate, the more comparisons we have to make for each set of integers.

- *Gradient Descent* — Overall, Gradient Descent did not do as well as Karmarkar-Karp and Repeated Random. By its very nature, Gradient Descent does not look at as many nor as diverse possible solution arrays as Repeated Random. Its decision-making technique is also not as optimal as Karmarkar-Karp because it only looks at two random indices within one solution array to see if negating those indices would minimize the residue and repeats this for k iterations. In doing this, there is a chance that Repeated Random does not pick two indices that make a great impact on minimizing the residue. Therefore, its less optimal performance is expected.

- *Simulated Annealing* — Similar to Repeated Random, Simulated Annealing did not perform as well as its counterparts. The reason for this is because it uses almost the same logic as Repeated Random. However, what causes it to distinctively perform worse is that it allows for the possibility to pick a solution array that does not minimize the residue, accepting worse moves with a probability according to the function provided. If by chance Simulated Annealing picks the worse move over and over again, its performance will also inevitably get worse.