CS168 Miniproject 7

Henry Lin, Kaylee Bement

1. A.

All of the graphs are irreducible because they are all connected so starting from any state, we can reach any other state through random walks through the graph.

(1), the circle graph with n = 10, is not aperiodic. To see this, consider how many steps it could take to walk from node 1 back to node 1 (though this could be applied equally to any node in the circle). This could just take two steps, 1 → 2 → 1 (or 1 → n → 1, but we will focus on moving toward increasing numbers for this example). When considering nodes 1, 2, and 3, it could take 4 steps, 1 → 2 → 3 → 2 → 1, or any multiple of two past 4 (1 → 2 → 3 → 2 → X \* (3 → 2 →) 1 for any X). Let y = x – 1, where x is the furthest node in any path from node 1 and back; this is equivalent to the number of steps it takes to get to node x from 1 when there is no jumping back and forth between nodes. Let z = the number of times we bounce back and forth between nodes. Thus, it will take 2 \* (y + z) steps to get from node 1 to node x and back, since any movement away from node 1 needs to eventually be countered with a move toward node 1. Finally, if we go all the way around the circle to end up back at one, this will take 10 steps + 2 \* z. This is true for any starting node in the circle, and in either direction around the circle. Thus, the gcd of the number of steps is 2 rather than 1, making (1) periodic.

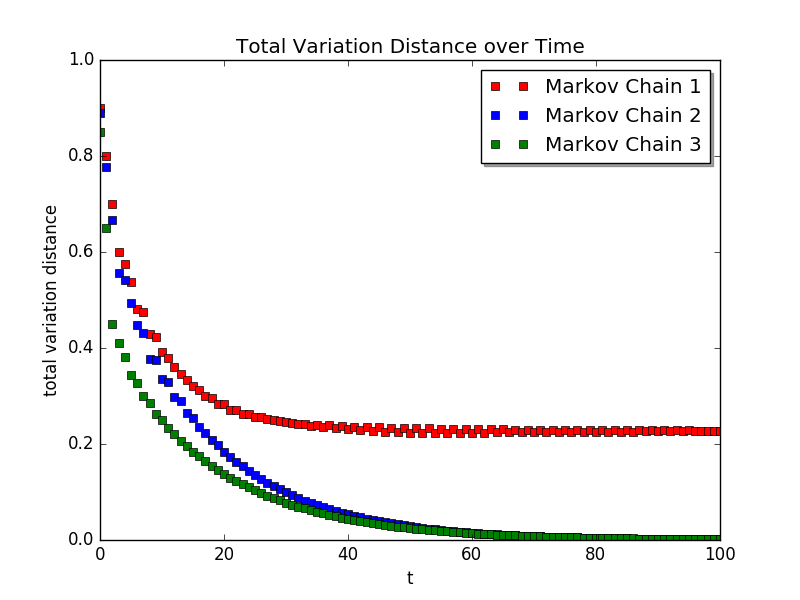
(2), however, is aperiodic. Everything we said about (1) is true for (2), except that going all the way around the circle will take 9 steps + 2 \*z rather than 10, which means that the number of steps needed to go all the way around the circle to get back to a given node is not divisible by 2. Therefore, the gcd of the number of steps is 1, making (2) periodic.

Its stationary distribution is [0.11, 0.11, 0.11, 0.11, 0.11, 0.11, 0.11, 0.11, 0.11].

(3) is also aperiodic. Everything we said about (2) is true for (3), except additionally, using the shortcut between nodes 1 and 5 will take either 5 steps + 2 \* z or 6 steps + 2 \* z. The gcd of 9, 5 + 2z, 6 + 2z, and 2(y+z) is 1, making (3) periodic.

Its stationary distribution is [0.15, 0.1, 0.1, 0.1, 0.15, 0.1, 0.1, 0.1, 0.1].

B.



import random

import pandas as pd

import numpy as np

import matplotlib

matplotlib.use('Agg')

import matplotlib.pyplot as plt

import math

# Question 1

# PART A

part\_1\_1 = np.matrix([[0, 0.5, 0, 0, 0, 0, 0, 0, 0, 0.5], [0.5, 0, 0.5, 0, 0, 0, 0, 0, 0, 0], [0, 0.5, 0, 0.5, 0, 0, 0, 0, 0, 0], [0, 0, 0.5, 0, 0.5, 0, 0, 0, 0, 0], [0, 0, 0, 0.5, 0, 0.5, 0, 0, 0, 0], [0, 0, 0, 0, 0.5, 0, 0.5, 0, 0, 0], [0, 0, 0, 0, 0, 0.5, 0, 0.5, 0, 0], [0, 0, 0, 0, 0, 0, 0.5, 0, 0.5, 0], [0, 0, 0, 0, 0, 0, 0, 0.5, 0, 0.5], [0.5, 0, 0, 0, 0, 0, 0, 0, 0, 0.5]])

print("CHAIN 1: ", part\_1\_1)

chain\_1\_eigs = np.linalg.eig(part\_1\_1.T)

print("Eigs: ", chain\_1\_eigs)

stationary\_dist\_1 = np.array([0.1 for i in range(10)])

part\_1\_2 = np.matrix([[0, 0.5, 0, 0, 0, 0, 0, 0, 0.5], [0.5, 0, 0.5, 0, 0, 0, 0, 0, 0], [0, 0.5, 0, 0.5, 0, 0, 0, 0, 0], [0, 0, 0.5, 0, 0.5, 0, 0, 0, 0], [0, 0, 0, 0.5, 0, 0.5, 0, 0, 0], [0, 0, 0, 0, 0.5, 0, 0.5, 0, 0], [0, 0, 0, 0, 0, 0.5, 0, 0.5, 0], [0, 0, 0, 0, 0, 0, 0.5, 0, 0.5], [0.5, 0, 0, 0, 0, 0, 0, 0.5, 0]])

print("CHAIN 2: ", part\_1\_2)

chain\_2\_eigs = np.linalg.eig(part\_1\_2.T)

print("Eigs: ", chain\_2\_eigs)

dist\_vec = chain\_2\_eigs[1][:,3]

stationary\_dist\_2\_nonorm = np.squeeze(np.asarray(dist\_vec))

stationary\_dist\_2 = stationary\_dist\_2\_nonorm / sum(stationary\_dist\_2\_nonorm)

print("Std dist: ", stationary\_dist\_2)

part\_1\_3 = np.matrix([[0, 1./3, 0, 0, 1./3, 0, 0, 0, 1./3], [0.5, 0, 0.5, 0, 0, 0, 0, 0, 0], [0, 0.5, 0, 0.5, 0, 0, 0, 0, 0], [0, 0, 0.5, 0, 0.5, 0, 0, 0, 0], [1./3, 0, 0, 1./3, 0, 1./3, 0, 0, 0], [0, 0, 0, 0, 0.5, 0, 0.5, 0, 0], [0, 0, 0, 0, 0, 0.5, 0, 0.5, 0], [0, 0, 0, 0, 0, 0, 0.5, 0, 0.5], [0.5, 0, 0, 0, 0, 0, 0, 0.5, 0]])

print("CHAIN 3: ", part\_1\_3)

chain\_3\_eigs = np.linalg.eig(part\_1\_3.T)

print("Eigs: ", chain\_3\_eigs)

dist\_vec\_2 = chain\_3\_eigs[1][:,4]

stationary\_dist\_3\_nonorm = np.squeeze(np.asarray(dist\_vec\_2))

stationary\_dist\_3 = stationary\_dist\_3\_nonorm / sum(stationary\_dist\_3\_nonorm)

print("Std dist: ", stationary\_dist\_3)

# PART B

state\_s\_1 = np.array([1, 0, 0, 0, 0, 0, 0, 0, 0, 0])

state\_s\_2 = np.array([1, 0, 0, 0, 0, 0, 0, 0, 0])

state\_s\_3 = np.array([1, 0, 0, 0, 0, 0, 0, 0, 0])

times = [i for i in range(101)]

markov\_dists\_1 = []

markov\_dists\_2 = []

markov\_dists\_3 = []

for time in times:

dist\_1 = 0

dist\_2 = 0

dist\_3 = 0

for i in range(len(state\_s\_1)):

dist\_1 += abs(state\_s\_1[i] - stationary\_dist\_1[i])

for i in range(len(state\_s\_3)):

dist\_2 += abs(state\_s\_2[i] - stationary\_dist\_2[i])

dist\_3 += abs(state\_s\_3[i] - stationary\_dist\_3[i])

dist\_1 /= 2

dist\_2 /= 2

dist\_3 /= 2

markov\_dists\_1.append(dist\_1)

markov\_dists\_2.append(dist\_2)

markov\_dists\_3.append(dist\_3)

state\_s\_1 = np.squeeze(np.asarray(np.dot(state\_s\_1, part\_1\_1)))

state\_s\_2 = np.squeeze(np.asarray(np.dot(state\_s\_2, part\_1\_2)))

state\_s\_3 = np.squeeze(np.asarray(np.dot(state\_s\_3, part\_1\_3)))

plt.plot(times, markov\_dists\_1, 'rs', label="Markov Chain 1")

plt.plot(times, markov\_dists\_2, 'bs', label="Markov Chain 2")

plt.plot(times, markov\_dists\_3, 'gs', label="Markov Chain 3")

plt.title("Total Variation Distance over Time")

plt.xlabel("t")

plt.ylabel("total variation distance")

plt.legend(shadow=True, loc=0)

plt.savefig("1\_b.png", format = 'png')

plt.close()

C.

Chain 1: 0.84125353

Chain 2: 0.76604444

Chain 3: 0.76759188

D.

In the power iteration algorithm, the ratio of the second largest eigenvalue of a matrix to its largest determines the rate at which we expect our state vector to converge to the eigenvector with the largest eigenvalue. In the case of chains 2 and 3, the largest eigenvector is also the stationary distribution, thus this ratio determines the mixing time for these chains. As one can see in the graph, chain 3 reaches a total variation distance of 0 between the state vector and the stationary distribution slightly quicker than chain 2 does, which makes sense since its second highest eigenvalue is slightly higher. Although chain 1 has the highest rate, since it is not aperiodic, its highest valued eigenvector is not the same as the uniform distribution we used for its π; thus, chain 1 does plateau at a faster rate than chain 2 and 3, but never reaches 0.

1. A. The state space of this Markov Chain is equal to the number of unique ways we can list the parks, which is 30!. When T = 0, the route will only update when a shorter route is found by randomly switching parks. That means that any route with distance longer than the randomly initialized one will not be seen. With T > 0 however, there is always a probability of it switching routes even if the newly found route is longer. This means that as MAXITER tends towards infinity, all routes will be seen.

Question 2

def read\_csv(csv\_name):

df = pd.read\_csv(csv\_name, header = None)

return df.as\_matrix()

parks\_info = read\_csv('parks.csv')

#print("parks info: ", parks\_info)

parks\_info = np.delete(parks\_info, (0), axis=0)

#converting parks info from np array to dict for easier use key: park name, value: (longitude, latitude)

parks\_dict = {}

for row in range(parks\_info.shape[0]):

park\_name = parks\_info[row][0]

longitude = float(parks\_info[row][1])

latitude = float(parks\_info[row][2])

parks\_dict[park\_name] = (longitude, latitude)

#print(parks\_dict)

def calculate\_distance\_two\_parks(park\_1, park\_2):

longitude\_1, latitude\_1 = parks\_dict[park\_1]

longitude\_2, latitude\_2 = parks\_dict[park\_2]

return math.sqrt((longitude\_1 - longitude\_2) \*\* 2 + (latitude\_1 - latitude\_2) \*\* 2)

def calculate\_route\_total\_distance(parks):

total\_distance = 0

for i in range(len(parks) - 1):

current\_park = parks[i]

next\_park = parks[i + 1]

total\_distance += calculate\_distance\_two\_parks(current\_park, next\_park)

#forgot about going from last park to home, doing it outside for loop

last\_park = parks[-1]

first\_park = parks[0]

total\_distance += calculate\_distance\_two\_parks(last\_park, first\_park)

#print("hello")

#print(total\_distance)

return total\_distance

all\_park\_names = list(parks\_dict.keys())

temp = all\_park\_names.copy()

temp.sort()

#print("Distance between Acadia and Arches: ", calculate\_distance\_two\_parks("Acadia", "Arches"))

#print("Distance Alphabetical ", calculate\_route\_total\_distance(temp))

def MCMC\_algorithm(max\_iterations, park\_list, T, c = False):

#print(park\_list)

random.shuffle(park\_list) #creates random route

#print("starting random route: ", park\_list)

#print("starting distance: ", calculate\_route\_total\_distance(park\_list))

best\_route = park\_list

#best\_route\_distance = calculate\_route\_total\_distance(best\_route)

#route\_distance\_history = [calculate\_route\_total\_distance(park\_list)]

route\_distance\_history = []

for i in range(max\_iterations):

if c == False:

random\_park\_index = random.randint(0, len(park\_list) - 2)

consec\_park\_index = random\_park\_index + 1

else:

random\_park\_index, consec\_park\_index = random.sample(range(len(park\_list)), 2)

random\_park = park\_list[random\_park\_index]

consec\_park = park\_list[consec\_park\_index]

curr\_route\_copy = park\_list.copy()

curr\_route\_copy[random\_park\_index] = consec\_park

curr\_route\_copy[consec\_park\_index] = random\_park

change\_distance\_traveled = calculate\_route\_total\_distance(curr\_route\_copy) - calculate\_route\_total\_distance(park\_list)

route\_distance\_history.append(calculate\_route\_total\_distance(curr\_route\_copy))

if change\_distance\_traveled < 0 or (T > 0 and random.uniform(0, 1) < math.exp( - change\_distance\_traveled / T)):

park\_list = curr\_route\_copy

if calculate\_route\_total\_distance(park\_list) < calculate\_route\_total\_distance(best\_route):

best\_route = park\_list

#print("Best Distance: ", calculate\_route\_total\_distance(best\_route))

#print("Best Route: ", best\_route)

return best\_route, route\_distance\_history #best route is best one we found, distance history is the distance of each route we tried

#found\_route, route\_distance\_history = MCMC\_algorithm(1000, all\_park\_names, 0.1)

def plot\_2b\_c(X\_axis, sets\_route\_histories, title, file\_name):

for history\_index in range(len(sets\_route\_histories)):

history = sets\_route\_histories[history\_index]

plt.scatter(X\_axis, history, label = "Iteration " + str(history\_index + 1))

plt.title(title)

plt.xlabel("Iterations")

plt.ylabel("Route Distance")

plt.legend(shadow=True, loc = 0)

plt.savefig(file\_name + ".png", format = "png")

plt.close()

def part\_b\_c(c = False):

list\_of\_T = [0, 1, 10, 100]

number\_iterations = 10000

num\_trials = 10

iterations\_list = [x for x in range(10000)]

for T in list\_of\_T:

history\_distances\_list = [] #list of route\_distance\_history lists

for trial in range(num\_trials):

best\_route, route\_distance\_history = MCMC\_algorithm(number\_iterations, all\_park\_names, T, c)

history\_distances\_list.append(route\_distance\_history)

if c == False:

plot\_2b\_c(iterations\_list, history\_distances\_list, "T = " + str(T), "2b\_T=" + str(T))

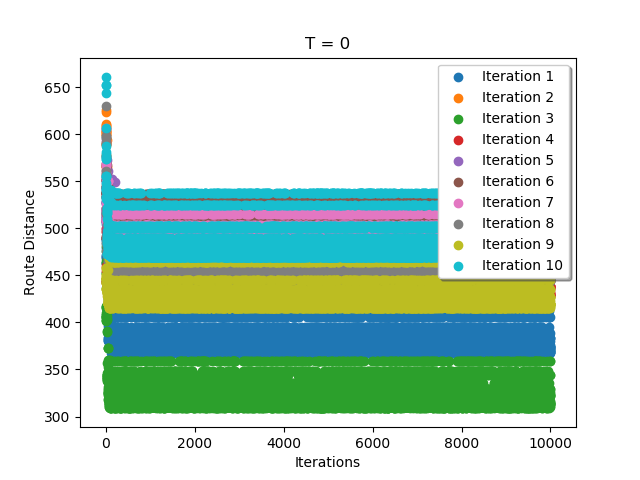
else:

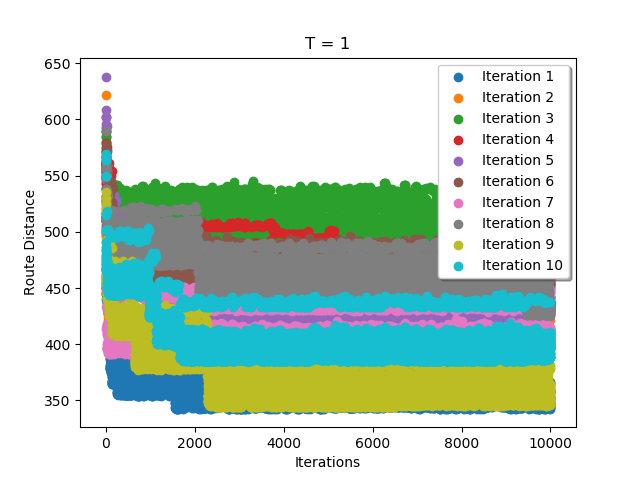
plot\_2b\_c(iterations\_list, history\_distances\_list, "T = " + str(T), "2c\_T=" + str(T))

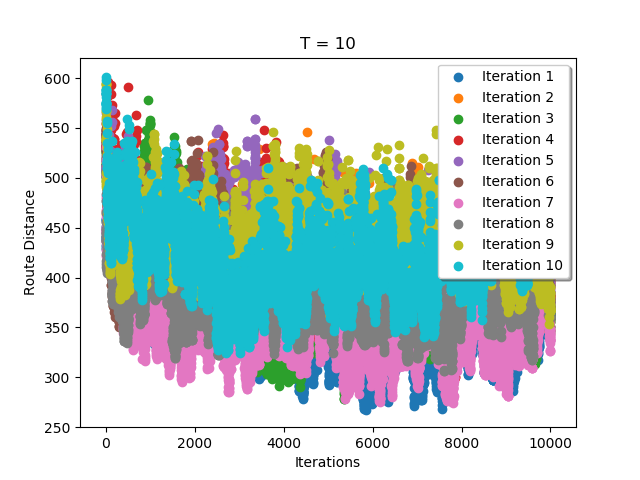
#part\_b\_c()

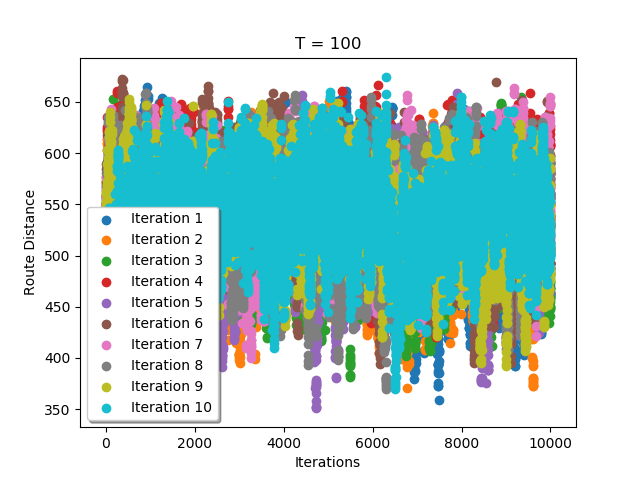
#part\_b\_c(c = True)

B. It seems that T = 10 is the optimal setting for this algorithm given the current constraints.

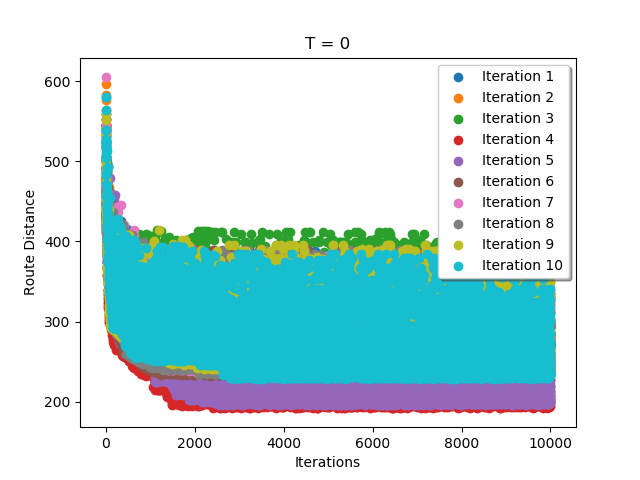


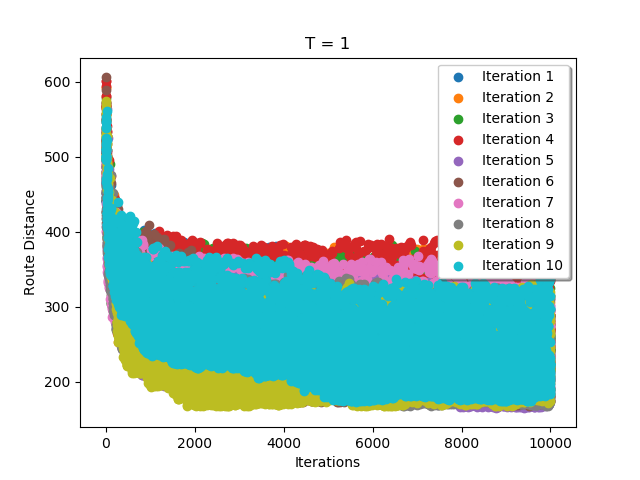


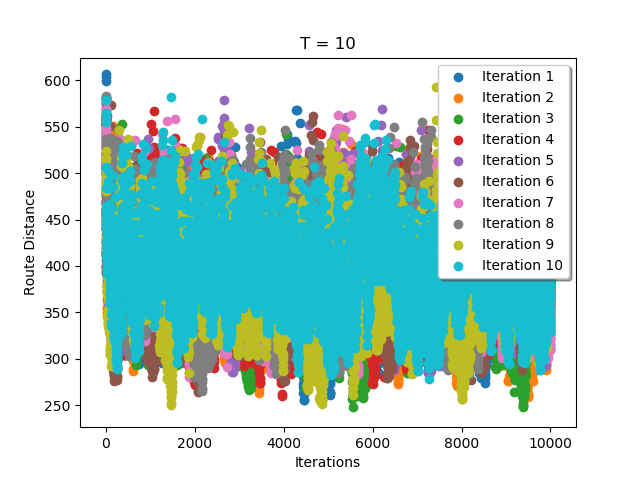


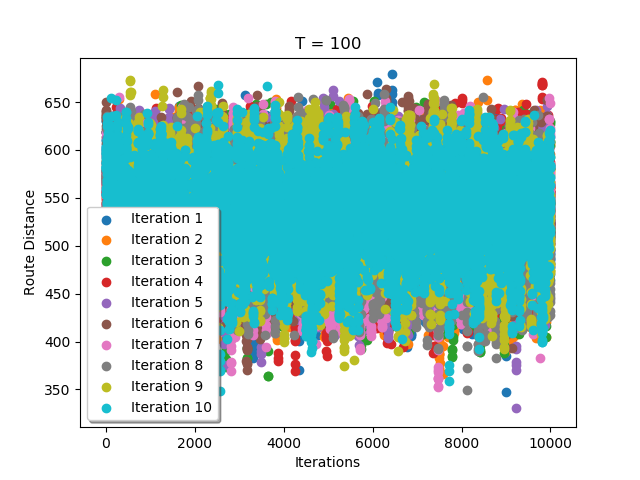


C. With the modification, it seems that T = 1 is now the best setting for the algorithm.









D. Between part B and C, it seems that for C, a lower value of T is better for the algorithm. T = 10 for B and T = 1 for C. A lower T value for part C is better because the modified algorithm allows for the switching of any two parks in the sequence. This means there is a lot more freedom and randomization in possible new routes during each iteration as opposed to the original algorithm in B. This means there is less need for a random switching of routes, which is the factor controlled by T. Larger values of T means a higher probability of switching to a newer, longer route while smaller values means lesser probability. In part C, it is very likely to find a shorter route and then not disrupt it by accidentally switching back to something longer.

1. A.

B. See code and video.

Video link: <https://www.youtube.com/watch?v=s_wuhGQz1Kk&feature=youtu.be&ab_channel=HenryLin>

If above link does not work, I have also included it in the zip file submitted.

MCMC = True

if MCMC:

max\_iterations = 3000

#MCMC algorithm

best\_plan = plan

best\_plan = plan

furthest\_distance = sim(plan)

T = 10

replace\_number = 4

print\_every = 500

number\_trials = 50

best\_distance = float('-infinity')

best\_plan\_overall\_all\_trials = []

for g in range(number\_trials):

print("Trial: ", g)

for i in range(max\_iterations):

#furthest\_distance = float('-infinity')

if i % print\_every == 0 :

print("Iteration: ", i)

#random\_index = random.randint(0, len(plan) - 1)

#random\_index\_two = random.randint(0, len(plan) - 1)

possible\_new\_plan = best\_plan.copy()

#possible\_new\_plan[random\_index] = random.uniform(-1 , 1)

#possible\_new\_plan[random\_index\_two] = random.uniform(-1 , 1)

for i in range(replace\_number):

random\_index = random.randint(0, len(plan) - 1)

possible\_new\_plan[random\_index] = random.uniform(-1 , 1)

change\_distance\_traveled = sim(possible\_new\_plan) - sim(plan)

if change\_distance\_traveled < 0 or (T > 10 and random.uniform(0, 1) < math.exp( - change\_distance\_traveled / T)):

plan = possible\_new\_plan

if sim(possible\_new\_plan) > furthest\_distance:

best\_plan = possible\_new\_plan

furthest\_distance = sim(possible\_new\_plan)

print(best\_plan)

print("This trial's furthest distance: ", furthest\_distance)

if furthest\_distance > best\_distance:

best\_distance = furthest\_distance

best\_plan\_overall\_all\_trials = best\_plan

print("Furthest Distance reached over all trials: ", best\_distance)

data = []

sim(best\_plan\_overall\_all\_trials)

print("best plan post MCMC: ", best\_plan\_overall\_all\_trials)

C. For our approach we tried two different methods to try and maximize the distance traveled in QWOP. The first approach was a simple gradient descent algorithm that given a random initialization, would try to take steps toward the local minimum using numerical gradients. We found out that while this did lead to an improvement over random numbers, the improvement was not significant enough so this approach was abandoned. Next we attempted an MCMC algorithm, which is what we used to achieve our best result. The MCMC algorithm originally reached around ~6 distance with randomly replacing 1 number at a time and 5000 iterations, but we decided to tune the hyperparameters a bit to try and improve results. We found the best results came when we randomly changed 4 numbers at a time. Additionally, we ran it over 50 different random initializations that each performed 3000 iterations of MCMC. We then outputted the best result from all of this. Ultimately, we were able to get a best distance of 9.377 using this method.