Hallyburton_HW5

March 2, 2018

1 Spencer Hallyburton

1.1 Homework #4

1.2 Collaborator: Salvador Barragan

Harvard University Spring 2018 Instructors: Rahul Dave Due Date: Friday, March 2nd, 2018 at 11:00am

```
In [1]: import matplotlib.pyplot as plt
        import numpy as np
        import csv
        import time
        from tqdm import tnrange, tqdm_notebook
        import time
        import math
        from functools import partial
        import random
        from tqdm import tnrange, tqdm_notebook

        start_time = time.time()
        %matplotlib inline
```

1.3 Problem 1: Optimization (contd)

Suppose you are building a pricing model for laying down telecom cables over a geographical region. Your model takes as input a pair of coordinates, (x, y), and contains two parameters, λ_1, λ_2. Given a coordinate, (x, y), and model parameters, the loss in revenue corresponding to the price model at location (x, y) is described by

$$L(x, y, \lambda_1, \lambda_2) = 0.000045\lambda_2^2 y - 0.000098\lambda_1^2 x + 0.003926\lambda_1 x \exp\left\{\left(y^2 - x^2\right)\left(\lambda_1^2 + \lambda_2^2\right)\right\}$$

Read the data contained in `HW3_data.csv`. This is a set of coordinates configured on the curve $y^2 - x^2 = -0.1$. Given the data, find parameters λ_1, λ_2 that minimize the net loss over the entire dataset.

1.3.1 Simulated Annealing

Implement Simulated Annealing initalized at $(\lambda_1, \lambda_2) = (-5, 0)$ to minimize our loss function L. Compare your results to what you obtained for gradient descent and stochastic gradient descent initialized at $(\lambda_1, \lambda_2) = (-5, 0)$.

For your Simulated Annealing implementation, we suggest *starting* with following settings for parameters (you should further experiment with and tweak these or feel free to set your own):

- Proposal distribution: bivariate normal with covariance $[[1, 0], [0, 1]]$
- Min Length: 500
- Max Temperature: 10

You should also set your own cooling schedule.

For each iteration, plot the parameters accepted or the cost function with respect to the iteration number. What is happening to the these parameters or costs over iterations? Connect the trends you observe in the visualization to the lecture on Markov Chains.

```
In [2]: my_data = np.genfromtxt('HW3_data.csv', delimiter=',')
        print('Data Shape:', my_data.shape)
```

```
Data Shape: (2, 16000)
```

```
In [3]: # Set up functions
        from mpl_toolkits.mplot3d import Axes3D

        def loss(X, Y, LAMBDA):
            T1 =  .000045*LAMBDA[1]**2 * Y
            T2 = -.000098*LAMBDA[0]**2 * X
            T3 =  .003926*LAMBDA[0] * X * np.exp( (Y**2 - X**2) * (LAMBDA[0]**2 + LAMBDA[1]**2
            return np.sum(T1 + T2 + T3)

        def make_3d_plot(xfinal, yfinal, zfinal, history, loss, X, Y):
            L1s = np.linspace(xfinal - 10 , xfinal + 10, 40)
            L2s = np.linspace(yfinal - 10 , yfinal + 10, 40)
            L1, L2 = np.meshgrid(L1s, L1s)
            zs = np.array([loss(X, Y, LAMBDA)
                           for LAMBDA in zip(np.ravel(L1), np.ravel(L2))])
            Z = zs.reshape(L1.shape)
            fig = plt.figure(figsize=(10, 6))
            ax = fig.add_subplot(111, projection='3d')

            off = -10
            ax.plot_surface(L1, L2, Z, rstride=1, cstride=1, color='b', alpha=0.1)
            ax.contour(L1, L2, Z, 20, alpha=0.5, offset=off, stride=30)
            ax.set_xlabel('Lambda 1')
            ax.set_ylabel('Lambda 2')
            ax.set_zlabel('Loss Function')
            ax.view_init(elev=30., azim=30)
```

```
            ax.plot([xfinal], [yfinal], [zfinal] , markerfacecolor='r', markeredgecolor='r', ma
            ax.plot([t[0] for t in history], [t[1] for t in history], loss , markerfacecolor='
            ax.plot([t[0] for t in history], [t[1] for t in history], off , alpha=0.5, markerfa
            plt.show()

        def sa_plot(X, Y, LAMBDA, loss, history):
            if not isinstance(loss, list):
                loss = [loss]
            make_3d_plot(LAMBDA[0], LAMBDA[1], loss[-1], history, loss, X, Y)
```

```
In [4]: # Proposal Function:
        covNorm = np.array([[1,0],[0,1]])
        pfxs = lambda mean: np.random.multivariate_normal(mean, covNorm)
        print('Example:', pfxs([0,2]))
```

```
Example: [ 0.11511539  1.24709003]
```

```
In [5]: # Temperature and iterations
        tf = lambda t: 0.8*t #temperature function
        itf = lambda length: math.ceil(1.2*length) #iteration function
```

```
In [6]: def sa(energyfunc, initials, epochs, tempfunc, iterfunc, proposalfunc, opt_cond='Dist'
            accumulator=[]
            best_solution = old_solution = initials['solution']
            T=initials['T']
            length=initials['length']
            best_energy = old_energy = energyfunc(old_solution)
            accepted=0
            total=0
            for index in range(epochs):
                print("Epoch", index)
                if index > 0:
                    T = tempfunc(T)
                    length=iterfunc(length)
                print("Temperature", T, "Length", length)
                for it in range(length):
                    total+=1
                    new_solution = proposalfunc(old_solution)
                    new_energy = energyfunc(new_solution)
                    # Use a min here as you could get a "probability" > 1
                    alpha = min(1, np.exp((old_energy - new_energy)/T))
                    if ((new_energy < old_energy) or (np.random.uniform() < alpha)):
                        # Accept proposed solution
                        accepted+=1
                        accumulator.append((T, new_solution, new_energy))
                        if new_energy < best_energy:
                            # Replace previous best with this one
```

```
                        best_energy = new_energy
                        best_solution = new_solution
                        best_index=total
                        best_temp=T
                    old_energy = new_energy
                    old_solution = new_solution
                else:
                    # Keep the old stuff
                    accumulator.append((T, old_solution, old_energy))

        best_meta=dict(index=best_index, temp=best_temp)
        print("frac accepted", accepted/total, "total iterations", total, 'bmeta', best_me
        return best_meta, best_solution, best_energy, accumulator
```

```
In [7]: loss_lam = partial(loss,my_data[0,:], my_data[1,:])
        inits=dict(solution=[-5,0], length=100, T=100)
        # Run the SA algorithm
        bmeta, bs, be, out = sa(loss_lam, inits, 30, tf, itf, pfxs)
```

```
Epoch 0
Temperature 100 Length 100
Epoch 1
Temperature 80.0 Length 120
Epoch 2
Temperature 64.0 Length 144
Epoch 3
Temperature 51.2 Length 173
Epoch 4
Temperature 40.96000000000001 Length 208
Epoch 5
Temperature 32.76800000000001 Length 250
Epoch 6
Temperature 26.21440000000001 Length 300
Epoch 7
Temperature 20.97152000000001 Length 360
Epoch 8
Temperature 16.777216000000006 Length 432
Epoch 9
Temperature 13.421772800000006 Length 519
Epoch 10
Temperature 10.737418240000006 Length 623
Epoch 11
Temperature 8.589934592000004 Length 748
Epoch 12
Temperature 6.871947673600004 Length 898
Epoch 13
Temperature 5.497558138880003 Length 1078
Epoch 14
```

```
Temperature 4.398046511104003 Length 1294
Epoch 15
Temperature 3.5184372088832023 Length 1553
Epoch 16
Temperature 2.814749767106562 Length 1864
Epoch 17
Temperature 2.25179981368525 Length 2237
Epoch 18
Temperature 1.8014398509482001 Length 2685
Epoch 19
Temperature 1.4411518807585602 Length 3222
Epoch 20
Temperature 1.1529215046068482 Length 3867
Epoch 21
Temperature 0.9223372036854786 Length 4641
Epoch 22
Temperature 0.7378697629483829 Length 5570
Epoch 23
Temperature 0.5902958103587064 Length 6684
Epoch 24
Temperature 0.4722366482869651 Length 8021
Epoch 25
Temperature 0.3777893186295721 Length 9626
Epoch 26
Temperature 0.3022314549036577 Length 11552
Epoch 27
Temperature 0.24178516392292618 Length 13863
Epoch 28
Temperature 0.19342813113834095 Length 16636
Epoch 29
Temperature 0.15474250491067276 Length 19964
frac accepted 0.227145397208803 total iterations 119232 bmeta {'index': 35873, 'temp': 0.590295
```

```
In [8]: # Print out results:
        print('Global Minimum Suggestion:',bs,'\nWith Loss Function:', be)
```

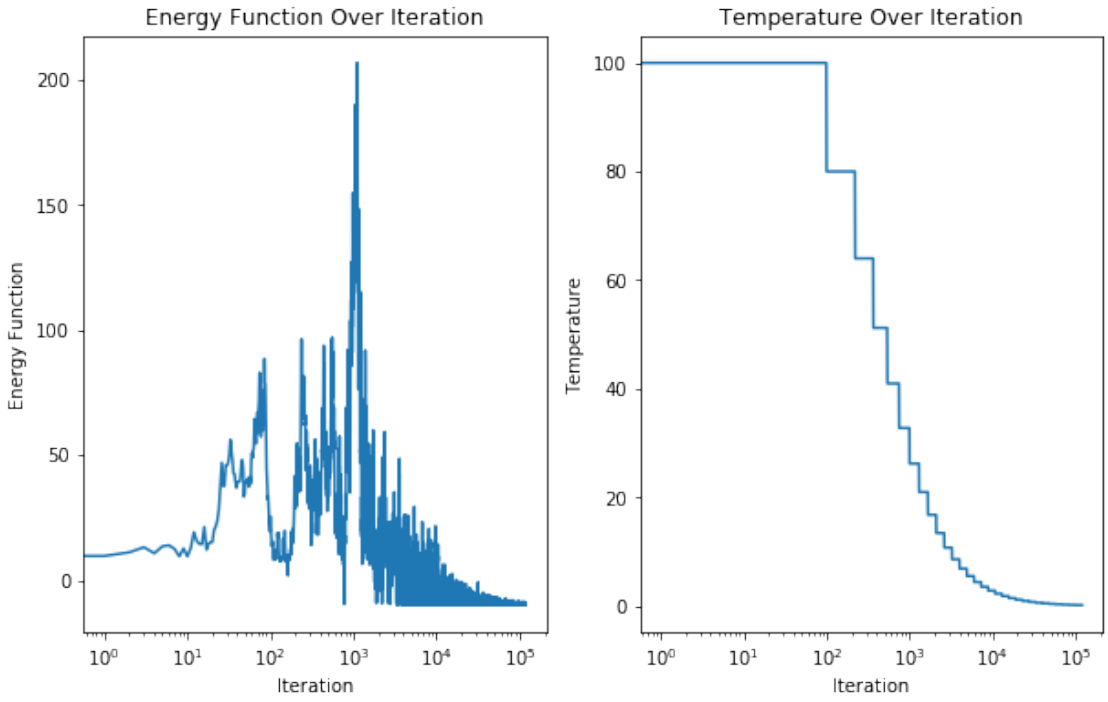
```
Global Minimum Suggestion: [  2.05281771e+00  -1.70592608e-03]
With Loss Function: -9.93409762445
```

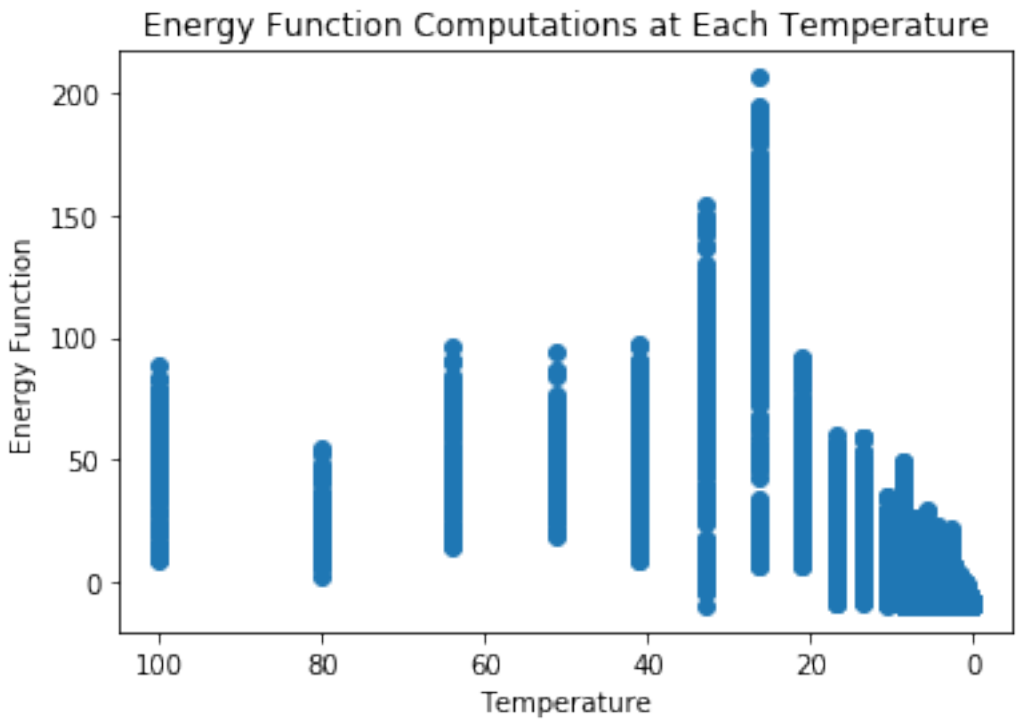
1.3.2 Now, plot the process

```
In [9]: # Create plots of the energy (cost) and the temperature over iterations
        accum_list = list(zip(*out))
        T = accum_list[0]
        LAM = accum_list[1]
        E = accum_list[2]
```

```
fig, ax = plt.subplots(1,2,figsize=(10,6))

# Plotting energy function
ax[0].plot(E)
ax[0].set_xlabel('Iteration')
ax[0].set_ylabel('Energy Function')
ax[0].set_xscale('log')
ax[0].set_title('Energy Function Over Iteration')
# Plotting temperature
ax[1].plot(T)
ax[1].set_xlabel('Iteration')
ax[1].set_ylabel('Temperature')
ax[1].set_title('Temperature Over Iteration')
ax[1].set_xscale('log')
plt.show()
```

```
# Plotting temperature
plt.scatter(T,E)
plt.xlabel('Temperature')
plt.ylabel('Energy Function')
plt.title('Energy Function Computations at Each Temperature')
plt.gca().invert_xaxis()
plt.show()
```





1.3.3 Compare to Gradient Descent and SGD:

When we compare this implementation to GD and SGD, we notice a few things. First, we see that the suggested global minimum is indeed the global minimum. This beats the attempts made by GD and SGD to compute minima, as they both got stuck in local minima. Second, one of the requirements of an annealing algorithm is that we have to be able to access all of the locations on our parameter space. This is similar to trio of aperiodicity, irreducibility, and recurrence that we demanded in Markov Chain construction. In order to ensure that we have reached the global minimum, we must meet the condition that we have to be able to access all of these locations. At each temperature, the system acts like a Markov Chain where there is a set probability to transition from one state to any other. This defines our Markov transition probabilities.

1.4 Problem 2: A Tired Salesman

In the famous traveling salesman problem, the quality of the solution can be measured in different ways, beyond finding the shortest path. For example, the total time of travel may also be important, and may depend on the means of transportation that connect pairs of cities. Consider a random distribution of N points on a plane representing the cities that must be visited by the traveling salesman. Each point is an (x,y) coordinate where both x and y are integers in the range [1, 50). Assign a value s_i where $i \in [1, \ldots, N]$ to each city that represents its size measured by population. Let $\forall s_i$, $s_i \in [1, 10)$. If two cities are farther away from each other than a **distance threshold of 10** and both have populations greater than a **population threshold of 5** assume there is a flight connection between them. In all other cases assume that our poor salesman would have to drive between cities. Flying is faster than driving by a factor of 10.

1. Use Simulated Annealing to find solutions to the traveling salesman problem for $N = 100$, optimizing the travel path for the total distance travelled (but keeping track of the time of travel).

2. Now redo the problem by optimizing the the path for the total time of travel (but keeping track of the distance traveled). Are the two solutions similar or different?

3. How do your results change if the population and distance thresholds for the exisitence of a flight between two cities are altered?

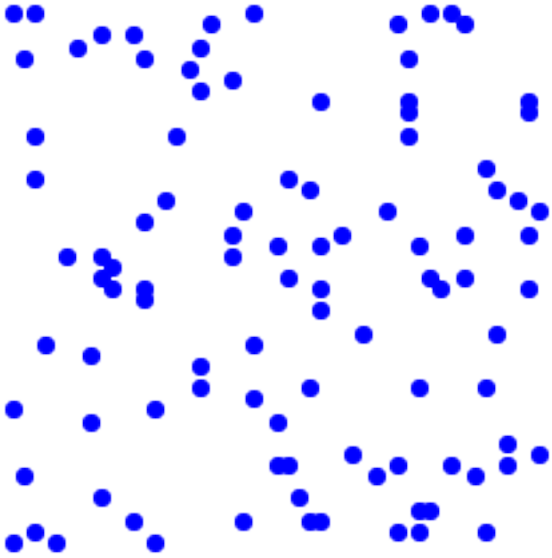
In [10]:
```
# Functions to get the permutations of the tours
def alltours(cities):
    "Return a list of tours, each a permutation of cities, but each one starting with
    start = first(cities)
    return [[start] + Tour(rest)
            for rest in itertools.permutations(cities - {start})]

# Gets the first city of a collection
def first(collection):
    "Start iterating over collection, and return the first element."
    return next(iter(collection))

Tour = list  # Tours are implemented as lists of cities

# Function to get the length of the tour
def tour_length(threshD, threshP, tour):
    "The total of distances between each pair of consecutive cities in the tour."
    tour_dist=0
    tour_time=0
    for i in range(len(tour)):
        tour_dist_new, tour_time_new = distance(tour[i], tour[i-1], threshD, threshP)
        tour_dist+=tour_dist_new
        tour_time+=tour_time_new

    return {'Dist':tour_dist, 'Time':tour_time}
```

In [11]:
```
# Cities are represented as Points, which are a subclass of complex numbers
from collections import namedtuple

# City is a named tuple with x and y position and a population
City = namedtuple('City', ['x', 'y', 'pop'], verbose=False)
```

In [12]:
```
# Get the distance between two cities
def distance(A, B, threshD, threshP): # Note: we have modified this function to refle
    "The distance between two cites."
    euc_dist = math.sqrt((A.x - B.x)*(A.x - B.x) + (A.y - B.y)*(A.y - B.y))
    # Convert distance to a time
    if (euc_dist > threshD) & (A.pop > threshP) & (B.pop > threshP):
        time = euc_dist / 10
```

```
        else:
            time = euc_dist
        return(euc_dist, time)

    def Cities(n, width=900, height=600, popmax=10, seed=42):
        "Make a set of n cities, each with random coordinates within a (width x height) re
        random.seed(seed * n)
        return frozenset(City(random.randrange(1,width), random.randrange(1,height), popma
                         for c in range(n))

    def plot_tour(tour):
        "Plot the cities as circles and the tour as lines between them. Start city is red
        start = tour[0]
        plot_lines(list(tour) + [start])
        plot_lines([start], 'rs') # Mark the start city with a red square

    def plot_lines(points, style='bo-'):
        "Plot lines to connect a series of points."
        plt.plot([p.x for p in points], [p.y for p in points], style)
        plt.axis('scaled'); plt.axis('off')

    def plot_tsp(algorithm, cities, threshD, threshP):
        "Apply a TSP algorithm to cities, plot the resulting tour, and print information."
        tour_length_partial = partial(tour_length, threshD, threshP)
        # Find the solution and time how long it takes
        t0 = time.clock()
        tour = algorithm(cities)
        t1 = time.clock()
        assert valid_tour(tour, cities)
        plot_tour(tour); plt.show()
        print("{} city tour with length {:.1f} and time {:.3f}s"
              .format(len(tour), tour_length_partial(tour)['Dist'],tour_length_partial(tou

    def valid_tour(tour, cities):
        "Is tour a valid tour for these cities?"
        return set(tour) == set(cities) and len(tour) == len(cities)
```

In [13]:
```
# Create list of cities
mycities = Cities(100, seed=332, width=50, height=50, popmax=10)
plot_lines(mycities, 'bo')
```



In [14]:
```
def change_tour(input_tour):
    "Change a tour for tsp iteration"

    #possible_indices = range(1, len(input_tour)) why not?
    possible_indices = range(len(input_tour))
    # take two random indices to swap
    c1 = np.random.choice(possible_indices)
    c2 = np.random.choice(possible_indices)

    new_tour = change_path(input_tour, c1, c2)

    return new_tour

def swap_cities(input_tour, i, j):
    "Swap two cities at index i and j in a tour"

    # save city1
    city1 = input_tour[i]

    # save city2
    city2 = input_tour[j]

    new_tour = input_tour.copy()

    # swap
    new_tour[j] = city1
```

```
            new_tour[i] = city2

            return new_tour

        change_path = swap_cities
```

In [15]:
```
def plot_optimization(out):
    # Plot the results of the optimization routine
    fig = plt.figure()
    ax = fig.add_subplot(111)

    lns1 = ax.plot(range(len(out)),[e[2]['Dist'] for e in out], alpha=0.6, lw=2, colo
    ax2 = ax.twinx()
    lns2 = ax2.plot(range(len(out)),[e[2]['Time'] for e in out], alpha=0.6, lw=2, col

    # added these two lines
    lns = lns1+lns2
    labs = [l.get_label() for l in lns]
    ax.legend(lns, labs, loc=0)

    # Plot result
    ax.grid()
    ax.set_xlabel('Iteration')
    ax.set_ylabel('Trip Distance')
    ax2.set_ylabel('Trip Time')
    plt.xscale('log')
    plt.show()
```

In [16]:
```
def sa_tsp(energyfunc, initials, epochs, tempfunc, iterfunc, proposalfunc, OC='Dist')
    accumulator=[]
    best_solution = old_solution = initials['solution']
    T=initials['T']
    length=initials['length']

    # Reannealing
    ANN_INT = 100
    beta = 1.5

    best_energy = old_energy = energyfunc(old_solution)
    accepted=0
    total=0
    i = 0
    for index in tnrange(epochs):
        i+=1
        if i % ANN_INT == 0: # Reannealing
            T = initials['T'] * (1 - index/epochs)
            length = initials['length']
            ANN_INT = np.ceil(beta*ANN_INT)
```

```
                    i = 0
                    print('Epoch=%d, Reannealing...' % index)

                if index > 0:
                    T = tempfunc(T)
                    length=iterfunc(length)
                for it in range(length):
                    total+=1
                    new_solution = proposalfunc(old_solution)
                    new_energy = energyfunc(new_solution)
                    # Use a min here as you could get a "probability" > 1
                    alpha = min(1, np.exp((old_energy[OC] - new_energy[OC])/T))
                    if ((new_energy[OC] < old_energy[OC]) or (np.random.uniform() < alpha)):
                        # Accept proposed solution
                        accepted+=1
                        accumulator.append((T, new_solution, new_energy))
                        if new_energy[OC] < best_energy[OC]:
                            # Replace previous best with this one
                            best_energy = new_energy
                            best_solution = new_solution
                            best_index=total
                            best_temp=T
                        old_energy = new_energy
                        old_solution = new_solution
                    else:
                        # Keep the old stuff
                        accumulator.append((T, old_solution, old_energy))

            best_meta=dict(index=best_index, temp=best_temp)
            print("RESULTS:\nFrac accepted: %.4f\nTotal Iterations: %d\nBest Meta - Index=%d,
                  (accepted/total, total, best_meta['index'], best_meta['temp']))
            return best_meta, best_solution, best_energy, accumulator
```

In [17]:
```
# Initialize functions
initial_tour = list(mycities)
length_func1 = lambda temperature: np.max((np.floor(np.sqrt(temperature)).astype(int)
length_func2 = lambda length: min(int(math.ceil(1.2*length)), 100)
length_func = length_func2
temp_func = lambda t: 0.94*t

# Initialize state
init_length = length_func(10)
inits=dict(solution=initial_tour, length=init_length, T=50.0)
```

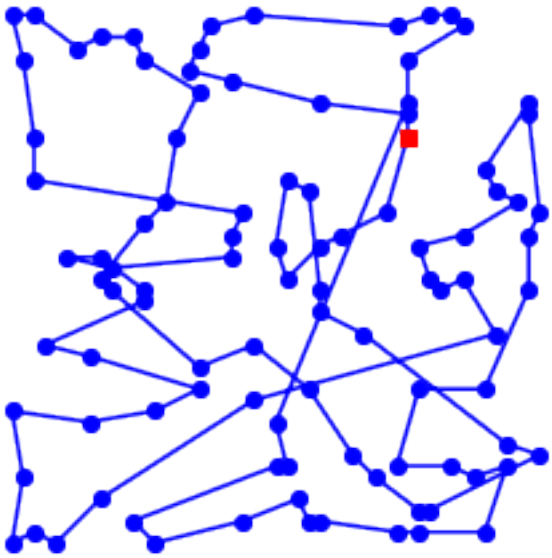
1.4.1 Optimizing for Tour Distance:

In [18]:
```
# Run annealing algorithm
threshD = 10
```

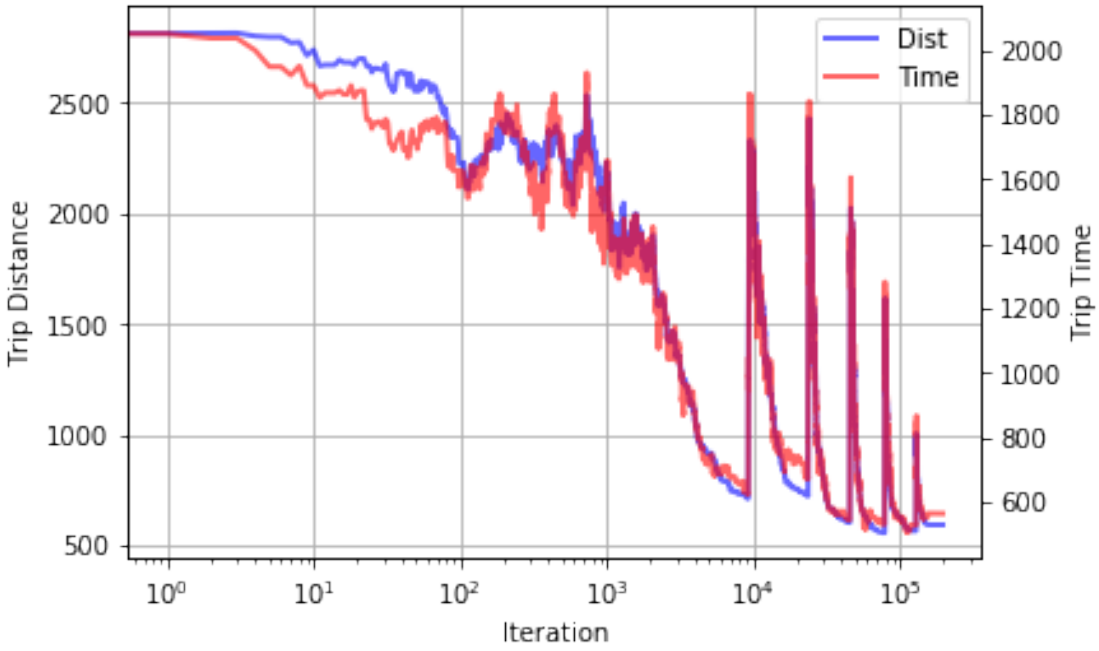
```
        threshP = 5
        tour_length_partial = partial(tour_length, threshD, threshP)
        opt_cond = 'Dist'
        t0 = time.clock()
        bmeta, bs, be, out = sa_tsp(tour_length_partial, inits, 2000, temp_func, length_func,
        t1 = time.clock()
        print('Algorithm Time:', t1-t0)
```

```
HBox(children=(IntProgress(value=0, max=2000), HTML(value='')))


Epoch=99, Reannealing...


/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:32: RuntimeWarning: overflow encou


Epoch=249, Reannealing...
Epoch=474, Reannealing...
Epoch=812, Reannealing...
Epoch=1319, Reannealing...

RESULTS:
Frac accepted: 0.0389
Total Iterations: 196402
Best Meta - Index=76860, Temp=1.0216e-07
Algorithm Time: 42.65616899999999
```

```
In [19]: plot_tsp(lambda x: bs, mycities, threshD, threshP)
         plot_optimization(out)
```



```
100 city tour with length 559.7 and time 532.406s
```



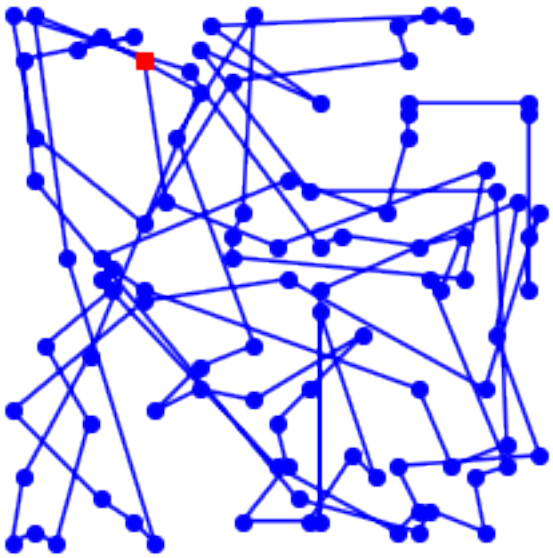
1.4.2 Optimizing for Tour Time:

```
In [20]: # Run annealing algorithm
         threshD = 10
         threshP = 5
         tour_length_partial = partial(tour_length, threshD, threshP)
         opt_cond = 'Time'
         t0 = time.clock()
         bmeta, bs, be, out = sa_tsp(tour_length_partial, inits, 2000, temp_func, length_func,
         t1 = time.clock()
         print('Algorithm Time:', t1-t0)
```

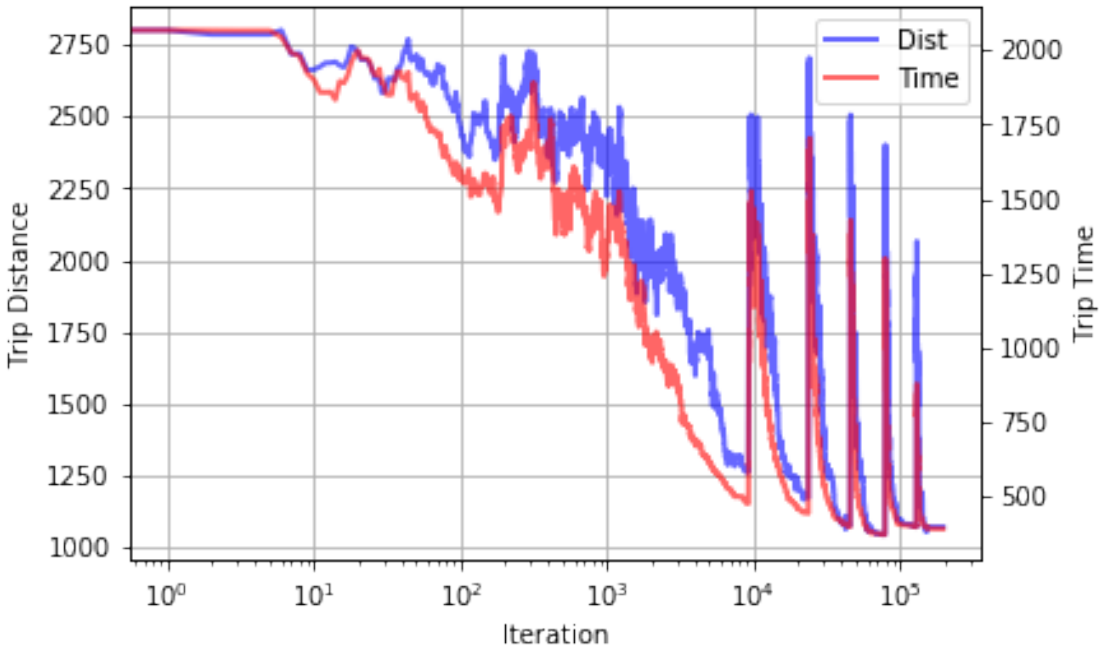
```
HBox(children=(IntProgress(value=0, max=2000), HTML(value='')))


Epoch=99, Reannealing...
Epoch=249, Reannealing...


/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:32: RuntimeWarning: overflow encou
```

```
Epoch=474, Reannealing...
Epoch=812, Reannealing...
Epoch=1319, Reannealing...

RESULTS:
Frac accepted: 0.0481
Total Iterations: 196402
Best Meta - Index=76219, Temp=1.4809e-07
Algorithm Time: 42.062062
```

```
In [21]: plot_tsp(lambda x: bs, mycities, threshD, threshP)
         plot_optimization(out)
```



```
100 city tour with length 1044.8 and time 369.985s
```



After performing the annealing algorithm to optimize tour time and tour distance, we see that the results are quite different. In this particular iteration, we see that performing the optimization for distance led to a lower distance and a higher time when compared to the optimization for time. This is to be expected given our conditions for maximization.

1.4.3 Trying different threshold conditions

```
In [29]: threshD = [5, 10, 15]
         threshP = [3, 5, 7]
         Dmesh, Pmesh = np.meshgrid(threshD, threshP)
         be_d_dist = np.zeros(np.shape(Dmesh))
         be_d_time = np.zeros(np.shape(Dmesh))
         be_t_dist = np.zeros(np.shape(Dmesh))
         be_t_time = np.zeros(np.shape(Dmesh))
```

```
In [30]: # Run annealing algorithm over a grid of parameters optimizing for distance
         for i, threshDval in enumerate(threshD):
             for j, threshPval in enumerate(threshP):
                 tour_length_partial = partial(tour_length, threshDval, threshPval)
                 # Optimize for distance
                 opt_cond = 'Dist'
                 bmeta_d, bs_d, be_d, out_d = sa_tsp(tour_length_partial, inits, 2000,
                                                     temp_func, length_func, change_tour, OC =
                 be_d_dist[i,j] = be_d['Dist']
                 be_d_time[i,j] = be_d['Time']
```

```
                # Optimize for time
                opt_cond = 'Time'
                bmeta_t, bs_t, be_t, out_t = sa_tsp(tour_length_partial, inits, 2000,
                                                    temp_func, length_func, change_tour, OC =
                be_t_dist[i,j] = be_t['Dist']
                be_t_time[i,j] = be_t['Time']
```

```
HBox(children=(IntProgress(value=0, max=2000), HTML(value='')))


Epoch=99, Reannealing...


/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:32: RuntimeWarning: overflow encou


Epoch=249, Reannealing...
Epoch=474, Reannealing...
Epoch=812, Reannealing...
Epoch=1319, Reannealing...

RESULTS:
Frac accepted: 0.0384
Total Iterations: 196402
Best Meta - Index=160473, Temp=3.7867e-08


HBox(children=(IntProgress(value=0, max=2000), HTML(value='')))


Epoch=99, Reannealing...
Epoch=249, Reannealing...
Epoch=474, Reannealing...
Epoch=812, Reannealing...
Epoch=1319, Reannealing...

RESULTS:
Frac accepted: 0.0684
Total Iterations: 196402
Best Meta - Index=45203, Temp=4.7382e-05


HBox(children=(IntProgress(value=0, max=2000), HTML(value='')))


Epoch=99, Reannealing...
Epoch=249, Reannealing...
Epoch=474, Reannealing...
Epoch=812, Reannealing...
```

```
Epoch=1319, Reannealing...

RESULTS:
Frac accepted: 0.0388
Total Iterations: 196402
Best Meta - Index=160812, Temp=2.9564e-08


HBox(children=(IntProgress(value=0, max=2000), HTML(value='')))


Epoch=99, Reannealing...
Epoch=249, Reannealing...
Epoch=474, Reannealing...
Epoch=812, Reannealing...
Epoch=1319, Reannealing...

RESULTS:
Frac accepted: 0.0488
Total Iterations: 196402
Best Meta - Index=162269, Temp=1.2432e-08


HBox(children=(IntProgress(value=0, max=2000), HTML(value='')))


Epoch=99, Reannealing...
Epoch=249, Reannealing...
Epoch=474, Reannealing...
Epoch=812, Reannealing...
Epoch=1319, Reannealing...

RESULTS:
Frac accepted: 0.0387
Total Iterations: 196402
Best Meta - Index=169677, Temp=1.2765e-10


HBox(children=(IntProgress(value=0, max=2000), HTML(value='')))


Epoch=99, Reannealing...
Epoch=249, Reannealing...
Epoch=474, Reannealing...
Epoch=812, Reannealing...
Epoch=1319, Reannealing...

RESULTS:
Frac accepted: 0.0378
```

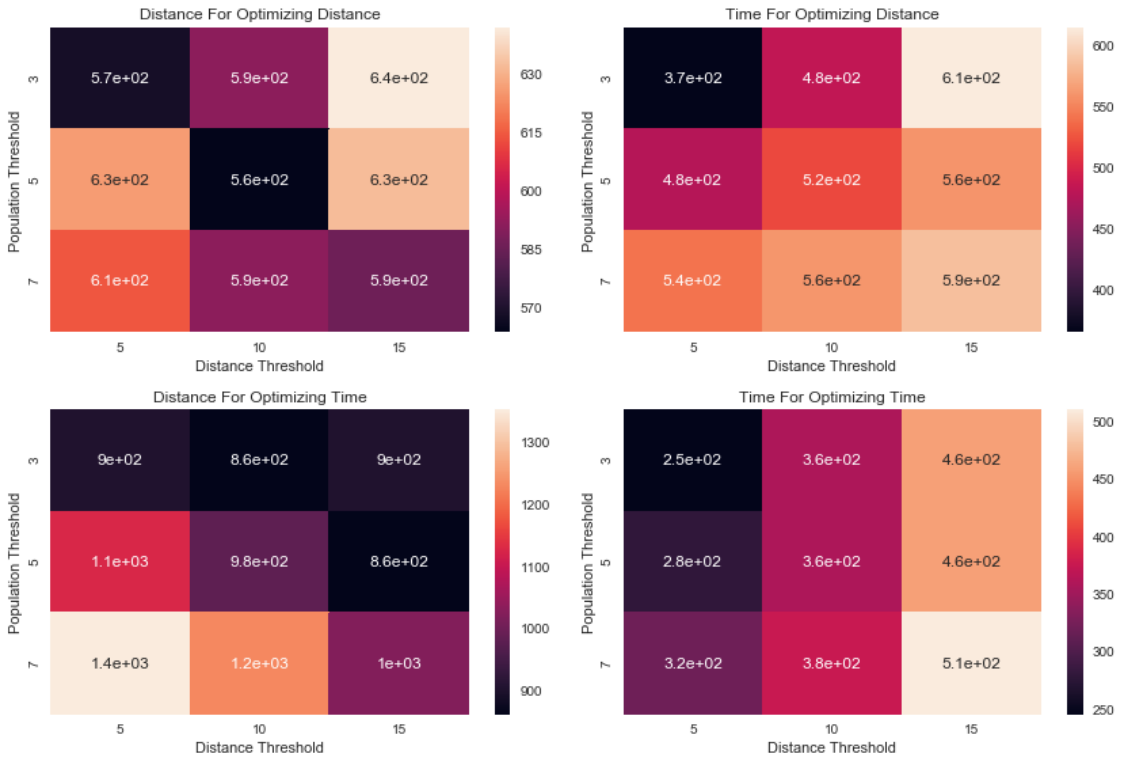
```
Total Iterations: 196402
Best Meta - Index=160106, Temp=4.5590e-08


HBox(children=(IntProgress(value=0, max=2000), HTML(value='')))


Epoch=99, Reannealing...
Epoch=249, Reannealing...
Epoch=474, Reannealing...
Epoch=812, Reannealing...
Epoch=1319, Reannealing...

RESULTS:
Frac accepted: 0.0394
Total Iterations: 196402
Best Meta - Index=44635, Temp=6.8682e-05


HBox(children=(IntProgress(value=0, max=2000), HTML(value='')))


Epoch=99, Reannealing...
Epoch=249, Reannealing...
Epoch=474, Reannealing...
Epoch=812, Reannealing...
Epoch=1319, Reannealing...

RESULTS:
Frac accepted: 0.0650
Total Iterations: 196402
Best Meta - Index=78026, Temp=4.8620e-08


HBox(children=(IntProgress(value=0, max=2000), HTML(value='')))


Epoch=99, Reannealing...
Epoch=249, Reannealing...
Epoch=474, Reannealing...
Epoch=812, Reannealing...
Epoch=1319, Reannealing...

RESULTS:
Frac accepted: 0.0382
Total Iterations: 196402
Best Meta - Index=153614, Temp=2.5443e-06
```

```
HBox(children=(IntProgress(value=0, max=2000), HTML(value='')))


Epoch=99, Reannealing...
Epoch=249, Reannealing...
Epoch=474, Reannealing...
Epoch=812, Reannealing...
Epoch=1319, Reannealing...

RESULTS:
Frac accepted: 0.0470
Total Iterations: 196402
Best Meta - Index=156223, Temp=5.0920e-07


HBox(children=(IntProgress(value=0, max=2000), HTML(value='')))


Epoch=99, Reannealing...
Epoch=249, Reannealing...
Epoch=474, Reannealing...
Epoch=812, Reannealing...
Epoch=1319, Reannealing...

RESULTS:
Frac accepted: 0.0384
Total Iterations: 196402
Best Meta - Index=181008, Temp=1.1030e-13


HBox(children=(IntProgress(value=0, max=2000), HTML(value='')))


Epoch=99, Reannealing...
Epoch=249, Reannealing...
Epoch=474, Reannealing...
Epoch=812, Reannealing...
Epoch=1319, Reannealing...

RESULTS:
Frac accepted: 0.0387
Total Iterations: 196402
Best Meta - Index=158278, Temp=1.4772e-07


HBox(children=(IntProgress(value=0, max=2000), HTML(value='')))


Epoch=99, Reannealing...
Epoch=249, Reannealing...
```

```
Epoch=474, Reannealing...
Epoch=812, Reannealing...
Epoch=1319, Reannealing...

RESULTS:
Frac accepted: 0.0379
Total Iterations: 196402
Best Meta - Index=160948, Temp=2.7790e-08


HBox(children=(IntProgress(value=0, max=2000), HTML(value='')))


Epoch=99, Reannealing...
Epoch=249, Reannealing...
Epoch=474, Reannealing...
Epoch=812, Reannealing...
Epoch=1319, Reannealing...

RESULTS:
Frac accepted: 0.0588
Total Iterations: 196402
Best Meta - Index=73467, Temp=8.3740e-07


HBox(children=(IntProgress(value=0, max=2000), HTML(value='')))


Epoch=99, Reannealing...
Epoch=249, Reannealing...
Epoch=474, Reannealing...
Epoch=812, Reannealing...
Epoch=1319, Reannealing...

RESULTS:
Frac accepted: 0.0390
Total Iterations: 196402
Best Meta - Index=175875, Temp=2.7539e-12


HBox(children=(IntProgress(value=0, max=2000), HTML(value='')))


Epoch=99, Reannealing...
Epoch=249, Reannealing...
Epoch=474, Reannealing...
Epoch=812, Reannealing...
Epoch=1319, Reannealing...
```

```
RESULTS:
Frac accepted: 0.0447
Total Iterations: 196402
Best Meta - Index=44788, Temp=6.0687e-05


HBox(children=(IntProgress(value=0, max=2000), HTML(value='')))


Epoch=99, Reannealing...
Epoch=249, Reannealing...
Epoch=474, Reannealing...
Epoch=812, Reannealing...
Epoch=1319, Reannealing...

RESULTS:
Frac accepted: 0.0388
Total Iterations: 196402
Best Meta - Index=181992, Temp=6.3204e-14


HBox(children=(IntProgress(value=0, max=2000), HTML(value='')))


Epoch=99, Reannealing...
Epoch=249, Reannealing...
Epoch=474, Reannealing...
Epoch=812, Reannealing...
Epoch=1319, Reannealing...

RESULTS:
Frac accepted: 0.0377
Total Iterations: 196402
Best Meta - Index=78759, Temp=3.1529e-08


In [32]: print(be_t_time)
         print(be_d_time)

[[ 245.43067721  358.77818589  458.37294333]
 [ 278.94696319  358.41195025  458.80409411]
 [ 320.67367795  376.57929458  510.03173955]]
[[ 365.33370241  483.22866319  614.56501256]
 [ 476.59117446  519.26436637  559.78574084]
 [ 540.81888888  560.48251898  585.70566413]]


In [33]: import seaborn as sns; sns.set()
```

```
fig, ax = plt.subplots(2,2, figsize=(12,8))

# Plot distance for optimizing distance
sns.heatmap(be_d_dist,annot=True, xticklabels=threshD, yticklabels=threshP, ax=ax[0,0]
ax[0,0].set_xlabel('Distance Threshold')
ax[0,0].set_ylabel('Population Threshold')
ax[0,0].set_title('Distance For Optimizing Distance')

# Plot time for optimizing distance
sns.heatmap(be_d_time,annot=True, xticklabels=threshD, yticklabels=threshP, ax=ax[0,1]
ax[0,1].set_xlabel('Distance Threshold')
ax[0,1].set_ylabel('Population Threshold')
ax[0,1].set_title('Time For Optimizing Distance')

# Plot distance for optimizing time
sns.heatmap(be_t_dist,annot=True, xticklabels=threshD, yticklabels=threshP, ax=ax[1,0]
ax[1,0].set_xlabel('Distance Threshold')
ax[1,0].set_ylabel('Population Threshold')
ax[1,0].set_title('Distance For Optimizing Time')

# Plot time for optimizing time
sns.heatmap(be_t_time,annot=True, xticklabels=threshD, yticklabels=threshP, ax=ax[1,1]
ax[1,1].set_xlabel('Distance Threshold')
ax[1,1].set_ylabel('Population Threshold')
ax[1,1].set_title('Time For Optimizing Time')
plt.tight_layout()
plt.show()
```



Above we show a heatmap of the results of the four different optimization combinations, showing distance and time for each optimization condition. We would expect that a change in population threshold should not impact the result of the distance output in the distance optimization, so long as the algorithm had enough time to converge to a suitable minimum. This is because our optimization was over distance and paid no attention to the time. The reverse is sometime true in the case of time output in optimizing over time, but the distance is correlated with the time, so we should not expect the exact same behavior. These are validated in the above figures.