APMTH 207: Advanced Scientific Computing:

Stochastic Methods for Data Analysis, Inference and Optimization

Homework #5

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

Instructions:

- Upload your final answers in a Jupyter notebook containing all work to Canvas.
- Structure your notebook and your work to maximize readability.

In [1]:
```
import numpy as np
import pandas as pd
from copy import deepcopy
import time
import itertools
import functools

import matplotlib
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import axes3d, Axes3D

import seaborn as sns
sns.set_style("whitegrid", {'axes.grid' : False})
sns.set_context('talk')
%matplotlib inline
```

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.

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 temperature, 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.

Answer to Problem 1

In [2]:

```
# Load data

data = np.genfromtxt('HW3_data.csv', delimiter=',')
x = data[0, ]
y = data[1, ]

print('Number of data points: {}.'.format(len(x)))

lam_best = np.array([2.05384, 0])
```

```
Number of data points: 16000.
```

In [3]:

```
# In GD, we use the gradient of total loss at each iteration
# In SGD, we multiply the gradient by total sample size at each iteration

def L(x, y, lam):

    # Average loss

    return np.mean(0.000045 * lam[1]**2 * y - 0.000098 * lam[0]**2 * x \
                   + 0.003926 * lam[0] * x * np.exp((y**2 - x**2) * (lam[0]**2

def L_total(x, y, lam):

    # Average loss

    return np.sum(0.000045 * lam[1]**2 * y - 0.000098 * lam[0]**2 * x \
                   + 0.003926 * lam[0] * x * np.exp((y**2 - x**2) * (lam[0]**2

def dL(x, y, lam):

    # Gradient of total loss

    z = y*y - x*x
    z1 = x*np.exp((lam[0]**2+lam[1]**2)*z)
    a = np.sum(-0.000196*lam[0]*x + (0.003926+0.007852*lam[0]**2*z)*z1)
    b = np.sum(0.00009*lam[1]*y + 0.007852*lam[0]*lam[1]*z*z1)
    return np.array([a, b])

class GD:
    def __init__(self, x, y, lam_init, step=0.001, max_iter=10000, tol=0.001)
        self.name = 'Gradient Descent'
        self.x = deepcopy(x)
        self.y = deepcopy(y)
        self.m = x.size
        self.lam_init = lam_init
        self.step = step
        self.max_iter = max_iter
        self.tol = tol
        self.costs = []
        self.time_ = []
        self.total_time = 0
        self.history = []
        self.iter_ = 0

    def run_gd(self):

        # Run max_iter iterations

        total_start = time.time()
        self.history.append(self.lam_init)
        self.costs.append(L(self.x, self.y, self.lam_init))
        for _ in range(self.max_iter):
            start = time.time()
            self.iter_ += 1
            self.history.append(self.history[-1] - self.step * dL(self.x, sel
            self.costs.append(L(self.x, self.y, self.history[-1]))
            self.time_.append(time.time() - start)
```

```
        self.total_time = time.time() - total_start
        return self

    def run_gd_test(self, actual=np.array([2.05384, 0])):

        # Run until approaching actual within tol or reaching max_iter

        total_start = time.time()
        self.history.append(self.lam_init)
        self.costs.append(L(self.x, self.y, self.lam_init))
        for _ in range(self.max_iter):
            start = time.time()
            self.iter_ += 1
            self.history.append(self.history[-1] - self.step * dL(self.x, sel
            self.costs.append(L(self.x, self.y, self.history[-1]))
            if np.isnan(self.costs[-1]):
                self.time_.append(time.time() - start)
                break
            if np.linalg.norm(self.history[-1] - actual) <= self.tol:
                self.time_.append(time.time() - start)
                break
            self.time_.append(time.time() - start)
        self.total_time = time.time() - total_start
        return self

class SGD:
    def __init__(self, x, y, lam_init, step=0.001, max_epoch=5, tol=0.001):
        self.name = 'Stochastic Gradient Descent'
        self.x = deepcopy(x)
        self.y = deepcopy(y)
        self.m = x.size
        self.lam_init = lam_init
        self.step = step
        self.max_epoch = max_epoch
        self.tol = tol
        self.costs = []
        self.total_cost = 0
        self.time_ = []
        self.total_time = 0
        self.history = []
        self.iter_ = 0

    def run_sgd(self):

        # Run until reaching max_epoch

        total_start = time.time()
        self.costs.append(L(self.x[0], self.y[0], self.lam_init))
        self.history.append(self.lam_init)
        for _ in range(self.max_epoch):
            for i in range(self.m):
                start = time.time()
                self.iter_ += 1
                self.history.append(self.history[-1]\
                                    - self.step * self.m* dL(self.x[i], self
                self.total_cost += L(self.x[i], self.y[i], self.history[-1])
                self.costs.append(self.total_cost / self.iter_)
```

```
                self.time_.append(time.time() - start)
            neworder = np.random.permutation(self.m)
            self.x = self.x[neworder]
            self.y = self.y[neworder]
        self.total_time = time.time() - total_start
        return self

    def run_sgd_test(self, actual=np.array([2.05384, 0])):

        # Run until approaching actual within tol or reaching max_epoch

        total_start = time.time()
        self.costs.append(L(self.x[0], self.y[0], self.lam_init))
        self.history.append(self.lam_init)
        done = False
        for _ in range(self.max_epoch):
            for i in range(self.m):
                start = time.time()
                self.iter_ += 1
                self.history.append(self.history[-1]\
                                    - self.step * self.m * dL(self.x[i], sel
                self.total_cost += L(self.x[i], self.y[i], self.history[-1])
                self.costs.append(self.total_cost / self.iter_)
                if np.isnan(self.costs[-1]):
                    done = True
                    self.time_.append(time.time() - start)
                    break
                if np.linalg.norm(self.history[-1] - actual) <= self.tol:
                    done = True
                    self.time_.append(time.time() - start)
                    break
                self.time_.append(time.time() - start)
            if done:
                break
            neworder = np.random.permutation(self.m)
            self.x = self.x[neworder]
            self.y = self.y[neworder]
        self.total_time = time.time() - total_start
        return self
```

In [4]:

```
lam_init = np.array([-5, 0])
gd = GD(x, y, lam_init).run_gd_test()
sgd = SGD(x, y, lam_init).run_sgd_test()
```

In [5]:

```
class SA:

    # Reference: https://am207.github.io/2018spring/wiki/simanneal.html

    def __init__(self):
        pass

    def run_sa(self, energyfunc, initials, epochs, tempfunc, iterfunc, propos
        start = time.time()
        accumulator = []
        self.initials = initials
        best_solution = old_solution = initials['solution']
        T = initials['T']
        length = initials['length']
        T_index = [(0, length)]
        best_energy = old_energy = energyfunc(old_solution)
        accepted = 0
        total = 0
        for ind in range(epochs):
            if verbose:
                print('Epoch', ind + 1)
            if ind > 0:
                T = tempfunc(T)
                length = iterfunc(length)
                T_index.append((T_index[-1][1], T_index[-1][1] + length))
            if verbose:
                print('Temperature', T, 'Length', length)
            for i in range(length):
                total += 1
                new_solution = proposalfunc(old_solution)
                new_energy = energyfunc(new_solution)
                alpha = min(1, np.exp((old_energy - new_energy) / T))
                if ((new_energy < old_energy) or (np.random.uniform() < alpha
                    accepted += 1
                    accumulator.append((T, new_solution, new_energy))
                    if new_energy < best_energy:
                        best_energy = new_energy
                        best_solution = new_solution
                        best_index = total
                        best_temp = T
                    old_energy = new_energy
                    old_solution = new_solution
                else:
                    accumulator.append((T, old_solution, old_energy))
            if verbose:
                print('Best T', best_temp, 'Best solution', best_solution, \
                      'Best energy', best_energy)
        self.accumulator = accumulator
        self.T_index = T_index
        self.best_meta = dict(index=best_index, temp=best_temp)
        self.best_solution = best_solution
        self.best_energy = best_energy
        self.accepted = accepted
        self.total = total
        print('Frac accepted', accepted / total, 'Total iterations', total, '
        self.total_time = time.time() - start
```

```
        return self
```

After several trials, we decide to set proposal distribution as bivariate normal with covariance [[1.2, 0], [0, 1.2]], min length as 500, and max temperature as 10.

We decrease the temperature by 20% and increase the length by 20% at each epoch.

In [6]:
```
ef = functools.partial(L_total, x, y)

def tf(T):
    return 0.8 * T

def itf(length):
    return int(np.ceil(1.2 * length))

def pf(lam):
    return np.random.multivariate_normal(lam, cov=[[1.5, 0], [0, 1.5]])

inits = dict(solution=np.array([-5, 0]), length=500, T=10)
epochs = 20
```

In [7]:
```
sa = SA().run_sa(ef, inits, epochs, tf, itf, pf, verbose=True)
```

```
Epoch 1
Temperature 10 Length 500
Best T 10 Best solution [ 2.17002534  0.23011703] Best energy -9.83260666308
Epoch 2
Temperature 8.0 Length 600
Best T 8.0 Best solution [ 2.11755422 -0.07839078] Best energy -9.91605198318
Epoch 3
Temperature 6.4 Length 720
Best T 6.4 Best solution [ 2.07636303 -0.05547057] Best energy -9.92889232722
Epoch 4
Temperature 5.120000000000001 Length 864
Best T 6.4 Best solution [ 2.07636303 -0.05547057] Best energy -9.92889232722
Epoch 5
Temperature 4.096000000000001 Length 1037
Best T 4.096000000000001 Best solution [ 2.07420924 -0.00942663] Best energy
-9.93294630497
Epoch 6
Temperature 3.276800000000001 Length 1245
Best T 4.096000000000001 Best solution [ 2.07420924 -0.00942663] Best energy
-9.93294630497
Epoch 7
Temperature 2.621440000000001 Length 1494
Best T 4.096000000000001 Best solution [ 2.07420924 -0.00942663] Best energy
-9.93294630497
Epoch 8
Temperature 2.097152000000001 Length 1793
Best T 4.096000000000001 Best solution [ 2.07420924 -0.00942663] Best energy
-9.93294630497
Epoch 9
Temperature 1.6777216000000008 Length 2152
Best T 4.096000000000001 Best solution [ 2.07420924 -0.00942663] Best energy
-9.93294630497
Epoch 10
Temperature 1.3421772800000007 Length 2583
Best T 1.3421772800000007 Best solution [ 2.03376002 -0.00602936] Best energy
-9.93303736532
Epoch 11
Temperature 1.0737418240000005 Length 3100
Best T 1.3421772800000007 Best solution [ 2.03376002 -0.00602936] Best energy
-9.93303736532
Epoch 12
Temperature 0.8589934592000005 Length 3720
Best T 1.3421772800000007 Best solution [ 2.03376002 -0.00602936] Best energy
-9.93303736532
Epoch 13
Temperature 0.6871947673600004 Length 4464
Best T 0.6871947673600004 Best solution [ 2.04546617  0.0055603 ] Best energy
-9.93388708687
Epoch 14
Temperature 0.5497558138880003 Length 5357
Best T 0.6871947673600004 Best solution [ 2.04546617  0.0055603 ] Best energy
-9.93388708687
Epoch 15
Temperature 0.4398046511104003 Length 6429
Best T 0.4398046511104003 Best solution [  2.06099476e+00  -1.68357696e-03] B
```

```
est energy -9.93397163529
Epoch 16
Temperature 0.3518437208883203 Length 7715
Best T 0.4398046511104003 Best solution [  2.06099476e+00  -1.68357696e-03] B
est energy -9.93397163529
Epoch 17
Temperature 0.28147497671065624 Length 9258
Best T 0.28147497671065624 Best solution [ 2.05302754  0.00520398] Best energ
y -9.93406773115
Epoch 18
Temperature 0.22517998136852502 Length 11110
Best T 0.28147497671065624 Best solution [ 2.05302754  0.00520398] Best energ
y -9.93406773115
Epoch 19
Temperature 0.18014398509482002 Length 13332
Best T 0.28147497671065624 Best solution [ 2.05302754  0.00520398] Best energ
y -9.93406773115
Epoch 20
Temperature 0.14411518807585602 Length 15999
Best T 0.28147497671065624 Best solution [ 2.05302754  0.00520398] Best energ
y -9.93406773115
Frac accepted 0.1611605614515577 Total iterations 93472 Best meta {'index': 5
0652, 'temp': 0.28147497671065624}
```

We can compare the result of simulated annealing with that of GD and SGD from the same initial point.

In [8]:
```
print('Simulated annealing')
print('Final lambda: {}'.format(sa.best_solution))
print('L2 distance to the actual optimum: {}'.format(np.linalg.norm(sa.best_s
print('Final total loss on the entire dataset: {}'.format(sa.best_energy))
print()

print('Gradient descent')
print('Final lambda: {}'.format(gd.history[-1]))
print('L2 distance to the actual optimum: {}'.format(np.linalg.norm(gd.histor
print('Final total loss on the entire dataset: {}'.format(L_total(x, y, gd.hi
print()

print('Gradient descent')
print('Final lambda: {}'.format(sgd.history[-1]))
print('L2 distance to the actual optimum: {}'.format(np.linalg.norm(sgd.histo
print('Final total loss on the entire dataset: {}'.format(L_total(x, y, sgd.h
print()
```

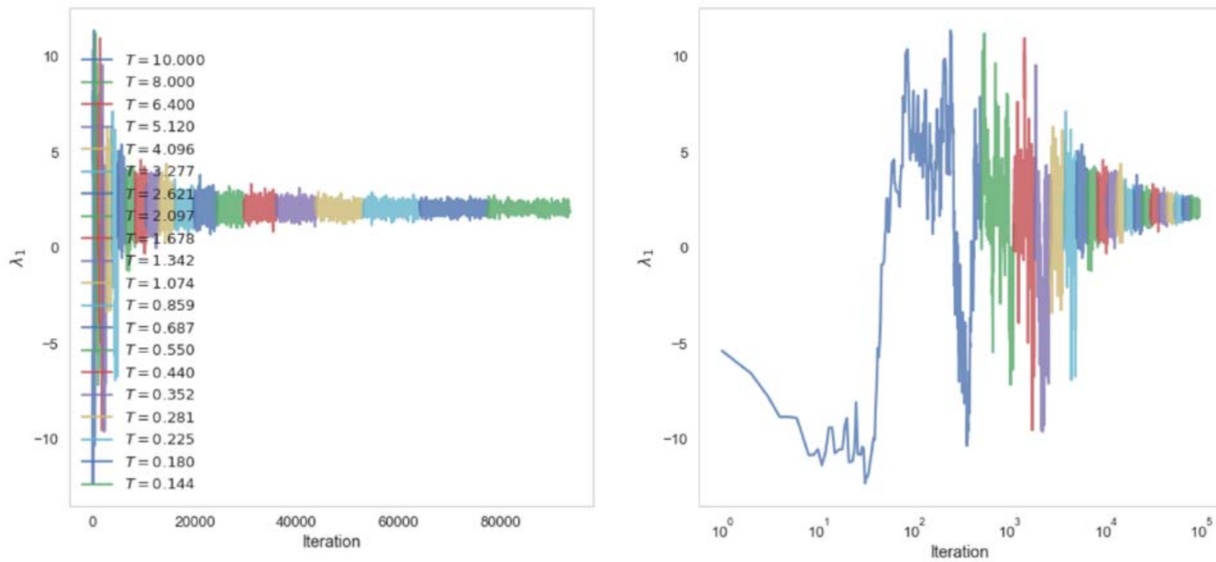
```
Simulated annealing
Final lambda: [ 2.05302754  0.00520398]
L2 distance to the actual optimum: 0.005267016991731391
Final total loss on the entire dataset: -9.934067731150435

Gradient descent
Final lambda: [-5.36324925  0.        ]
L2 distance to the actual optimum: 7.417089249485938
Final total loss on the entire dataset: 8.161528699583187

Gradient descent
Final lambda: [ 2.05364425  0.        ]
L2 distance to the actual optimum: 0.0001957475651179763
Final total loss on the entire dataset: -9.934103919875724
```

As we can see, simulated annealing managed to find a solution very close to the actual global optimum, while gradient descent was "trapped" to another local optimum and stochastic gradient descent managed to "bump into" the global optimum (since we were cheating here by setting the actual global optimum).

For the visualization of the parameters and the cost function with respect to the iteration number for each temperature, we plot iteration number in linear (left) and log (right) scales as shown below.

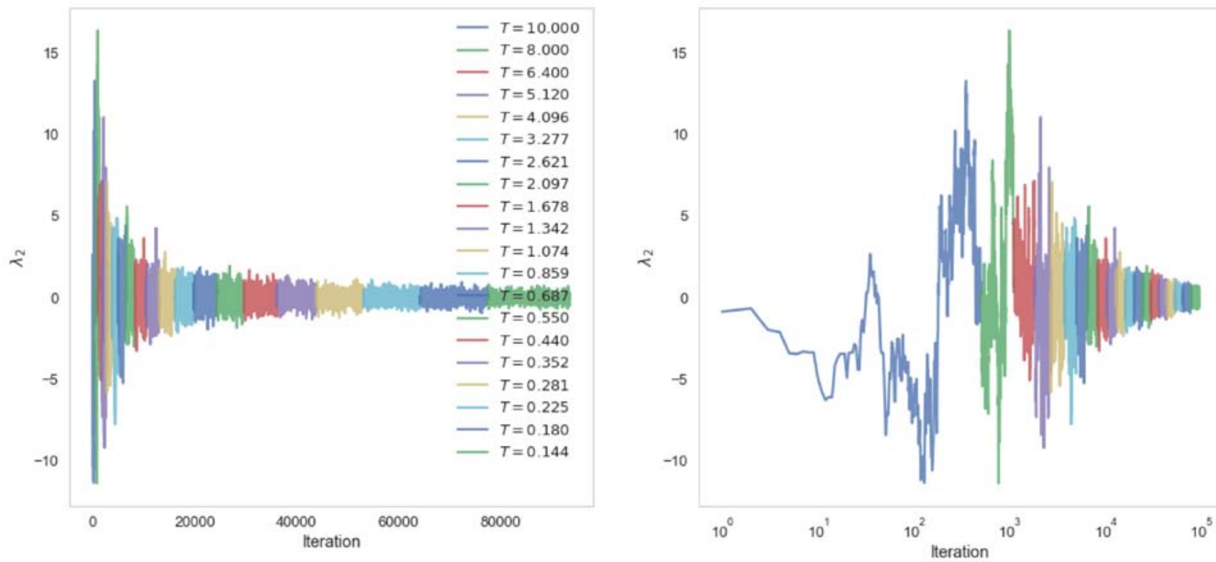
In [9]:

```
plt.figure(figsize=(18, 8))
plt.subplot(1, 2, 1)
for i_start, i_end in sa.T_index:
    plt.plot(range(i_start + 1, i_end + 1), [v[1][0] for v in sa.accumulator[
             label='$T = {:.3f}$'.format(sa.accumulator[i_start][0]))
#plt.xscale('log');
plt.xlabel('Iteration');
plt.ylabel('$\lambda_1$');
plt.legend();

plt.subplot(1, 2, 2)
for i_start, i_end in sa.T_index:
    plt.plot(range(i_start + 1, i_end + 1), [v[1][0] for v in sa.accumulator[
             label='$T = {:.3f}$'.format(sa.accumulator[i_start][0]))
plt.xscale('log');
plt.xlabel('Iteration');
plt.ylabel('$\lambda_1$');
```

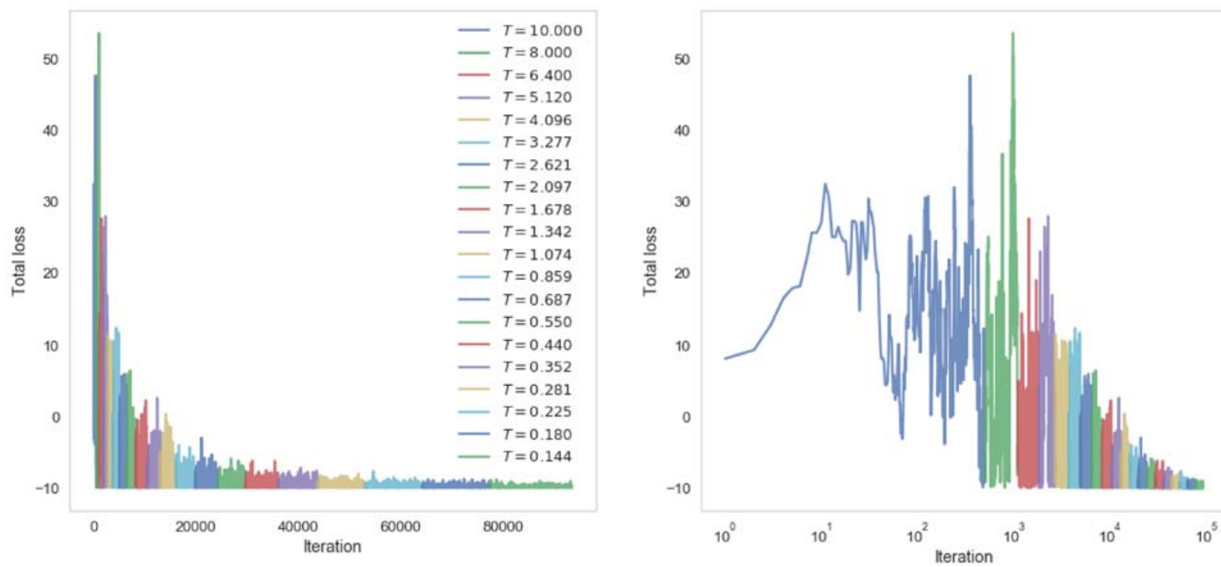


In [10]:

```
plt.figure(figsize=(18, 8))
plt.subplot(1, 2, 1)
for i_start, i_end in sa.T_index:
    plt.plot(range(i_start + 1, i_end + 1), [v[1][1] for v in sa.accumulator[
            label='$T = {:.3f}$'.format(sa.accumulator[i_start][0]))
#plt.xscale('log');
plt.xlabel('Iteration');
plt.ylabel('$\lambda_2$');
plt.legend();

plt.subplot(1, 2, 2)
for i_start, i_end in sa.T_index:
    plt.plot(range(i_start + 1, i_end + 1), [v[1][1] for v in sa.accumulator[
            label='$T = {:.3f}$'.format(sa.accumulator[i_start][0]))
plt.xscale('log');
plt.xlabel('Iteration');
plt.ylabel('$\lambda_2$');
```



In [11]:
```
plt.figure(figsize=(18, 8))
plt.subplot(1, 2, 1)
for i_start, i_end in sa.T_index:
    plt.plot(range(i_start + 1, i_end + 1), [v[2] for v in sa.accumulator[i_s
             label='$T = {:.3f}$'.format(sa.accumulator[i_start][0]))
#plt.xscale('log');
plt.xlabel('Iteration');
plt.ylabel('Total loss');
plt.legend();

plt.subplot(1, 2, 2)
for i_start, i_end in sa.T_index:
    plt.plot(range(i_start + 1, i_end + 1), [v[2] for v in sa.accumulator[i_s
             label='$T = {:.3f}$'.format(sa.accumulator[i_start][0]))
plt.xscale('log');
plt.xlabel('Iteration');
plt.ylabel('Total loss');
#plt.legend();
```



The fluctuation of parameters and cost function decrease as the temperature decreases. Simulated annealing can be viewed as a single inhomogeneous markov chain or a set of homogeneous markove chains, one at each temperature. Parameters can be viewed as states; iterations accepting proposed parameters can be viewed as transitions to new states; iterations rejecting proposed parameters can be viewed as transitions through edges connecting to themselves.

The detailed balance condition satisfied by our proposal ensures that the sequence generated by simulated annealing is a stationary markov chain with the boltzmann distribution as the stationary distribution of the chain as $t \to \infty$. We observed a tigher and tigher stationary distribution for the parameters about the optimum as the temperature decreases, indicating the parameters are able to escape the "trap" of other local optima at high temperature in the first few epochs, and would converge to the global optimum as $t \to \infty$.

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?

Answer to Problem 2

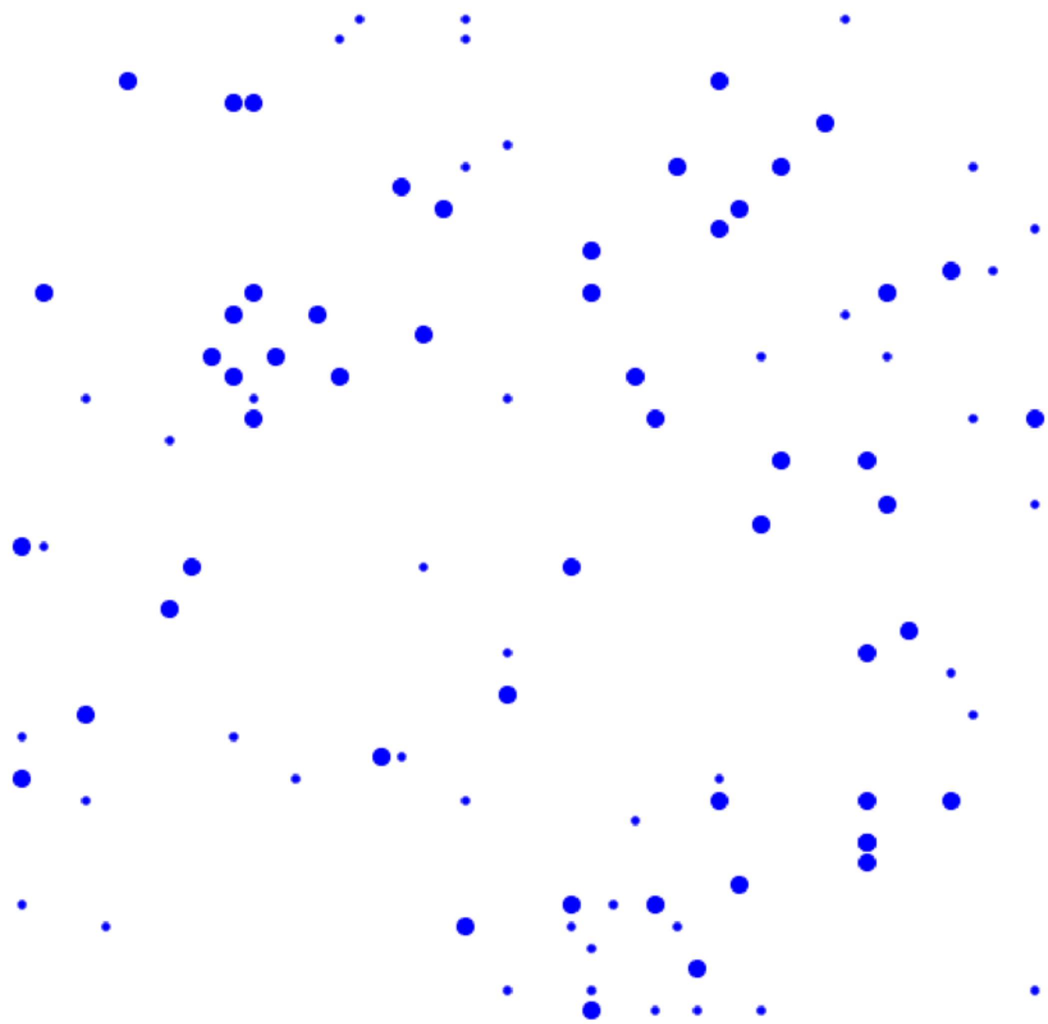
In [2]:

```
class Map:
    def __init__(self, N=100, seed=99, dist_thres=10, pop_thres=5, factor=10)
        self.N = N
        self.seed = seed
        self.dist = - np.ones((N, N))
        self.time_ = - np.ones((N, N))
        self.dist_thres = dist_thres
        self.pop_thres = pop_thres
        self.factor = factor

    def create_cities(self, c_range=[1, 50], s_range=[1, 10], seed=None):
        if seed is None:
            seed = self.seed
        np.random.seed(seed)
        self.coords = np.random.randint(low=c_range[0], high=c_range[1], size
        self.sizes = np.random.rand(self.N) * (s_range[1] - s_range[0]) + s_r
        return self

    def get_costs(self, i, j):
        if self.dist[i, j] < 0:
            self.dist[i, j] = self.dist[j, i] = np.sqrt(np.sum(np.square(self
            if (self.sizes[i] > self.pop_thres) and (self.sizes[j] > self.pop
            and (self.dist[i, j] > self.dist_thres):
                self.time_[i, j] = self.time_[j, i] = self.dist[i, j]
            else:
                self.time_[i, j] = self.time_[j, i] = self.factor * self.dist
        return self.dist[i, j], self.time_[i, j]

    def plot_cities(self):
        # Large cities (size > population threshold) are represented by large
        # Small cities (size <= population threshold) are represented by smal
        plt.plot([p[0] for p in self.coords[self.sizes > self.pop_thres]], \
                 [p[1] for p in self.coords[self.sizes > self.pop_thres]], 'b
        plt.plot([p[0] for p in self.coords[self.sizes <= self.pop_thres]], \
                 [p[1] for p in self.coords[self.sizes <= self.pop_thres]], '
        plt.axis('off')
        plt.axis('equal')

    def plot_tour(self, tour):
        # Flights are represented by green lines
        # Drivings are represented by red lines
        init_tour(self, tour)
        for i in range(len(tour)):
            if self.time_[tour[i], tour[i-1]] > self.dist[tour[i], tour[i-1]]
                plt.plot([self.coords[tour[i]][0], self.coords[tour[i-1]][0]]
                         [self.coords[tour[i]][1], self.coords[tour[i-1]][1]]
            else:
                plt.plot([self.coords[tour[i]][0], self.coords[tour[i-1]][0]]
                         [self.coords[tour[i]][1], self.coords[tour[i-1]][1]]
        plt.plot([self.coords[tour[0]][0]], [self.coords[tour[0]][1]], 'rs',
        plt.axis('off')
        plt.axis('equal')

def init_tour(map_, tour):
    if len(tour) != map_.N or set(tour) != set(range(map_.N)):
        raise ValueError('Invalid tour.')
```

```
    tour = tour.copy()
    costs = np.array([map_.get_costs(tour[i], tour[i-1]) for i in range(len(t
    return np.sum(costs, axis=0)

def change_tour(map_, old_tour, old_costs):
    #np.random.seed()
    c1 = np.random.randint(low=0, high=len(old_tour))
    c2 = np.random.randint(low=0, high=len(old_tour))
    new_tour = old_tour.copy()
    new_tour[c1], new_tour[c2] = new_tour[c2], new_tour[c1]

    new_costs = old_costs.copy()
    new_costs -= np.array(map_.get_costs(old_tour[c1], old_tour[c1 - 1]))
    new_costs -= np.array(map_.get_costs(old_tour[c1], old_tour[(c1 + 1) % le
    new_costs -= np.array(map_.get_costs(old_tour[c2], old_tour[c2 - 1]))
    new_costs -= np.array(map_.get_costs(old_tour[c2], old_tour[(c2 + 1) % le

    new_costs += np.array(map_.get_costs(new_tour[c1], new_tour[c1 - 1]))
    new_costs += np.array(map_.get_costs(new_tour[c1], new_tour[(c1 + 1) % le
    new_costs += np.array(map_.get_costs(new_tour[c2], new_tour[c2 - 1]))
    new_costs += np.array(map_.get_costs(new_tour[c2], new_tour[(c2 + 1) % le

    return new_tour, new_costs
```

We can create 100 cities as follows.

In [3]:
```
m = Map().create_cities()
plt.figure(figsize=(10, 10))
m.plot_cities()
```



Large cities are represented by large circles; small cities are represented by small dots.

In [4]:

```
class SA_TSP:
    def __init__(self, map_, energy_ind=0):

        # energy_ind
        # 0: total distance traveled
        # 1: total time of trave
        self.map_ = map_
        self.energy_ind = energy_ind

    def run_sa(self, init_energy, initials, epochs, tempfunc, iterfunc, propo
        energy_ind = self.energy_ind
        start = time.time()
        accumulator = []
        self.initials = initials
        best_solution = old_solution = initials['solution']
        T = initials['T']
        length = initials['length']
        T_index = [(0, length)]
        Ts = []
        best_energy = old_energy = init_energy(old_solution)
        accepted = 0
        total = 0
        for ind in range(epochs):
            if verbose:
                print('Epoch', ind + 1)
            if ind > 0:
                T = tempfunc(T)
                length = iterfunc(length)
                T_index.append((T_index[-1][1], T_index[-1][1] + length))
            if verbose:
                print('Temperature', T, 'Length', length)
            for i in range(length):
                total += 1
                new_solution, new_energy = proposalfunc(old_solution, old_ene
                alpha = min(1, np.exp((old_energy[energy_ind] - new_energy[en
                if ((new_energy[energy_ind] < old_energy[energy_ind]) or (np.
                    accepted += 1
                    Ts.append(T)
                    accumulator.append(new_energy)
                    if new_energy[energy_ind] < best_energy[energy_ind]:
                        best_energy = new_energy
                        best_solution = new_solution
                        best_index = total
                        best_temp = T
                    old_energy = new_energy
                    old_solution = new_solution
                else:
                    Ts.append(T)
                    accumulator.append(old_energy)
            if verbose:
                print('Best T', best_temp, \
                      'Best energy', best_energy)
        self.Ts = np.array(Ts)
        self.accumulator = np.array(accumulator)
        self.T_index = T_index
        self.best_meta = dict(index=best_index, temp=best_temp)
```

```
        self.best_solution = best_solution
        self.best_energy = best_energy
        self.accepted = accepted
        self.total = total
        print('Frac accepted', accepted / total, 'Total iterations', total, '
        self.total_time = time.time() - start
        return self
```

Problem 2.1

After several trials, we decide to set min length as 100, max temperature as 80. We decrease the temperature by 10% and increase the length by 20% at each epoch.

We are able to get a reasonably good result after running 50 epochs.

In [5]:

```
%%time

N = 100
m = Map(N).create_cities()

ef = functools.partial(init_tour, m)

def tf(T):
    return 0.9 * T

def itf(length):
    return int(np.ceil(1.2 * length))

pf = functools.partial(change_tour, m)

inits = dict(solution=list(range(N)), length=100, T=80)
epochs = 50

#np.random.seed()
tsp1 = SA_TSP(m, 0).run_sa(ef, inits, epochs, tf, itf, pf, verbose=True)
```

```
Epoch 1
Temperature 80 Length 100
Best T 80 Best energy [  2302.4179186  18090.5476194]
Epoch 2
Temperature 72.0 Length 120
Best T 72.0 Best energy [  2238.20135423  17936.22480036]
Epoch 3
Temperature 64.8 Length 144
Best T 64.8 Best energy [  2205.99876484  16629.83879415]
Epoch 4
Temperature 58.32 Length 173
Best T 58.32 Best energy [  2131.69481168  17209.48421083]
Epoch 5
Temperature 52.488 Length 208
Best T 58.32 Best energy [  2131.69481168  17209.48421083]
Epoch 6
Temperature 47.239200000000004 Length 250
Best T 47.239200000000004 Best energy [  2065.37117713  16989.54231986]
Epoch 7
Temperature 42.515280000000004 Length 300
Best T 42.515280000000004 Best energy [  2029.60039322  15113.55836796]
Epoch 8
Temperature 38.263752000000004 Length 360
Best T 38.263752000000004 Best energy [  1993.46662424  16670.24060283]
Epoch 9
Temperature 34.4373768 Length 432
Best T 34.4373768 Best energy [  1989.29637241  14999.87173344]
Epoch 10
Temperature 30.993639120000005 Length 519
Best T 30.993639120000005 Best energy [  1959.21742901  15559.62366192]
Epoch 11
Temperature 27.894275208000003 Length 623
Best T 27.894275208000003 Best energy [  1946.7877688   15832.86718689]
Epoch 12
Temperature 25.104847687200003 Length 748
Best T 25.104847687200003 Best energy [  1882.01306422  15001.66263971]
```

```
Epoch 13
Temperature 22.59436291848 Length 898
Best T 22.59436291848 Best energy [  1813.10610935  14536.03294194]
Epoch 14
Temperature 20.334926626632 Length 1078
Best T 20.334926626632 Best energy [  1701.51369059  13135.46202736]
Epoch 15
Temperature 18.3014339639688 Length 1294
Best T 20.334926626632 Best energy [  1701.51369059  13135.46202736]
Epoch 16
Temperature 16.47129056757192 Length 1553
Best T 16.47129056757192 Best energy [  1673.0810357   12589.54858505]
Epoch 17
Temperature 14.824161510814728 Length 1864
Best T 14.824161510814728 Best energy [  1588.57369348  12192.39688844]
Epoch 18
Temperature 13.341745359733254 Length 2237
Best T 13.341745359733254 Best energy [  1426.20693834  12459.03221699]
Epoch 19
Temperature 12.007570823759929 Length 2685
Best T 12.007570823759929 Best energy [  1411.70083098  11777.55397823]
Epoch 20
Temperature 10.806813741383936 Length 3222
Best T 10.806813741383936 Best energy [ 1243.77376809  9216.84013109]
Epoch 21
Temperature 9.726132367245542 Length 3867
Best T 9.726132367245542 Best energy [ 1195.05370572  9446.0720105 ]
Epoch 22
Temperature 8.753519130520989 Length 4641
Best T 8.753519130520989 Best energy [ 1025.69126057  8582.42453429]
Epoch 23
Temperature 7.8781672174688895 Length 5570
Best T 8.753519130520989 Best energy [ 1025.69126057  8582.42453429]
Epoch 24
Temperature 7.090350495722 Length 6684
Best T 7.090350495722 Best energy [  999.35797508  8971.22236485]
Epoch 25
Temperature 6.3813154461498005 Length 8021
Best T 6.3813154461498005 Best energy [  907.89847731  7563.0053899 ]
Epoch 26
Temperature 5.74318390153482 Length 9626
Best T 5.74318390153482 Best energy [  859.15786842  7167.82254156]
Epoch 27
Temperature 5.168865511381338 Length 11552
Best T 5.74318390153482 Best energy [  859.15786842  7167.82254156]
Epoch 28
Temperature 4.651978960243205 Length 13863
Best T 4.651978960243205 Best energy [  790.08399093  6802.67085061]
Epoch 29
Temperature 4.186781064218884 Length 16636
Best T 4.186781064218884 Best energy [  703.08361124  5808.0792133 ]
Epoch 30
Temperature 3.7681029577969958 Length 19964
Best T 3.7681029577969958 Best energy [  702.49697692  5966.80397649]
Epoch 31
Temperature 3.391292662017296 Length 23957
Best T 3.391292662017296 Best energy [  694.36497175  6487.54321573]
```

```
Epoch 32
Temperature 3.0521633958155667 Length 28749
Best T 3.0521633958155667 Best energy [  656.28847609  5903.53480947]
Epoch 33
Temperature 2.74694705623401 Length 34499
Best T 2.74694705623401 Best energy [  601.9885375   5244.17699295]
Epoch 34
Temperature 2.472252350610609 Length 41399
Best T 2.472252350610609 Best energy [  546.78078237  5370.87485714]
Epoch 35
Temperature 2.2250271155495485 Length 49679
Best T 2.2250271155495485 Best energy [  519.28798529  4993.33109814]
Epoch 36
Temperature 2.0025244039945935 Length 59615
Best T 2.0025244039945935 Best energy [  494.0379493   4742.82346748]
Epoch 37
Temperature 1.8022719635951343 Length 71538
Best T 1.8022719635951343 Best energy [  474.19716714  4444.88200242]
Epoch 38
Temperature 1.6220447672356209 Length 85846
Best T 1.6220447672356209 Best energy [  463.34539435  4411.16548248]
Epoch 39
Temperature 1.4598402905120589 Length 103016
Best T 1.6220447672356209 Best energy [  463.34539435  4411.16548248]
Epoch 40
Temperature 1.313856261460853 Length 123620
Best T 1.313856261460853 Best energy [  462.47577757  4527.82480915]
Epoch 41
Temperature 1.1824706353147678 Length 148344
Best T 1.1824706353147678 Best energy [  456.57372974  4360.4299797 ]
Epoch 42
Temperature 1.064223571783291 Length 178013
Best T 1.064223571783291 Best energy [  445.40599484  4248.75263063]
Epoch 43
Temperature 0.957801214604962 Length 213616
Best T 0.957801214604962 Best energy [  438.72540793  4181.94676159]
Epoch 44
Temperature 0.8620210931444658 Length 256340
Best T 0.8620210931444658 Best energy [  435.92034964  4262.2705299 ]
Epoch 45
Temperature 0.7758189838300192 Length 307608
Best T 0.7758189838300192 Best energy [  430.55117827  4100.20446499]
Epoch 46
Temperature 0.6982370854470173 Length 369130
Best T 0.6982370854470173 Best energy [  427.36888804  3958.52256211]
Epoch 47
Temperature 0.6284133769023156 Length 442956
Best T 0.6284133769023156 Best energy [  426.69431922  4060.15122512]
Epoch 48
Temperature 0.565572039212084 Length 531548
Best T 0.565572039212084 Best energy [  426.45135549  4167.58058833]
Epoch 49
Temperature 0.5090148352908757 Length 637858
Best T 0.5090148352908757 Best energy [  426.38463395  4166.91337299]
Epoch 50
Temperature 0.4581133517617881 Length 765430
Best T 0.4581133517617881 Best energy [  425.44614427  4049.15412496]
```

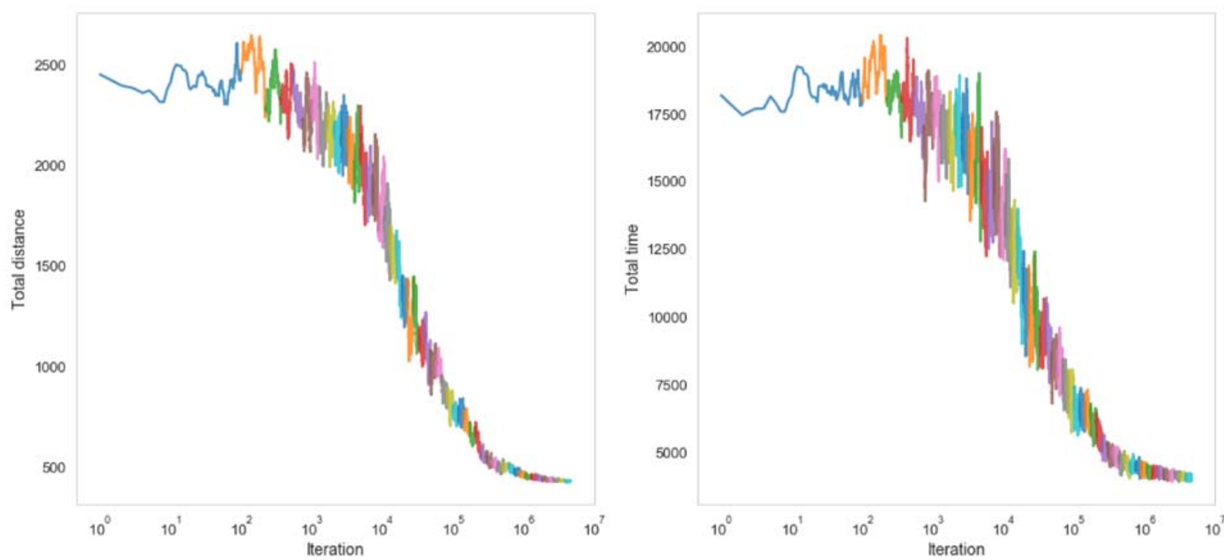
```
Frac accepted 0.019436005237812862 Total iterations 4591993 Best meta {'inde
x': 4238818, 'temp': 0.4581133517617881}
Wall time: 4min 47s
```

We can plot the total distance traveled, as well as total time of travel, over iterations as follows.

In [6]:

```
%%time

plt.figure(figsize=(18, 8))
plt.subplot(1, 2, 1)

for i_start, i_end in tsp1.T_index:
    plt.plot(range(i_start + 1, i_end + 1), tsp1.accumulator[i_start:i_end, 0
plt.xscale('log');
plt.xlabel('Iteration');
plt.ylabel('Total distance');

plt.subplot(1, 2, 2)
for i_start, i_end in tsp1.T_index:
    plt.plot(range(i_start + 1, i_end + 1), tsp1.accumulator[i_start:i_end, 1
plt.xscale('log');
plt.xlabel('Iteration');
plt.ylabel('Total time');
```

```
Wall time: 2.34 s
```

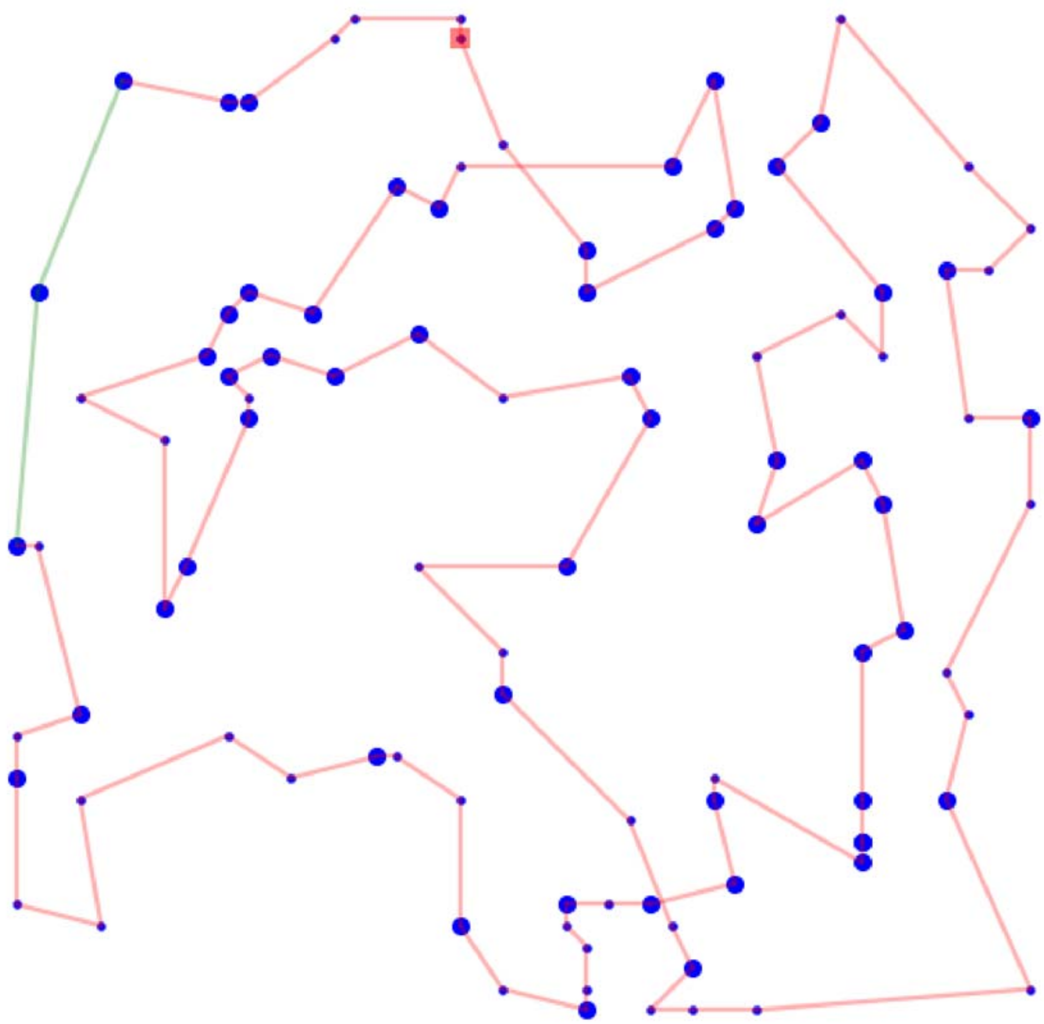


Observations:

1. Both total distance and total time decrease over iterations, and seem to converge in the end.
2. The fluctuations decrease as the temperature decreases.
3. The percentage of decrease in total distance is higher than that in total time.

We can visualize the solution on the city map as follows. Large cities are represented by large circles, and small cities are represented by small dots. Starting point is indicated by a red square. Drivings are represented by red lines and flights are represented by green lines.

In [7]:
```
plt.figure(figsize=(10, 10))
tsp1.map_.plot_cities()
tsp1.map_.plot_tour(tsp1.best_solution)
```



Observations:

1. The solution is dominated by drivings instead of flights.
2. The solution is dominated by short paths and there are few intersections.

The solution looks reasonable since we are optimizing for the total distance tranveled.

Problem 2.2

In this case, we are supposed to increase the max temperature since the value of the total time is generally an order of magnitude higher than that of the total distance.

After several trials, we max temperature as 500 and min length as 100. We decrease the temperature by 10% and increase the length by 20% at each epoch.

We are able to get a reasonably good result after running 50 epochs.

Basically, we use the same cooling schedule and initialization as the previous part, except the max temperature.

In [8]:

```
%%time

N = 100
m = Map(N).create_cities()

ef = functools.partial(init_tour, m)

def tf(T):
    return 0.9 * T

def itf(length):
    return int(np.ceil(1.2 * length))

pf = functools.partial(change_tour, m)

inits = dict(solution=list(range(N)), length=100, T=500)
epochs = 50

#np.random.seed()
tsp2 = SA_TSP(m, 1).run_sa(ef, inits, epochs, tf, itf, pf, verbose=True)
```

```
Epoch 1
Temperature 500 Length 100
Best T 500 Best energy [  2341.74750431  16439.16422592]
Epoch 2
Temperature 450.0 Length 120
Best T 450.0 Best energy [  2272.62061649  15844.90612818]
Epoch 3
Temperature 405.0 Length 144
Best T 405.0 Best energy [  2297.50844466  14541.91137833]
Epoch 4
Temperature 364.5 Length 173
Best T 405.0 Best energy [  2297.50844466  14541.91137833]
Epoch 5
Temperature 328.05 Length 208
Best T 405.0 Best energy [  2297.50844466  14541.91137833]
Epoch 6
Temperature 295.245 Length 250
Best T 295.245 Best energy [  2086.09475493  14269.47972204]
Epoch 7
Temperature 265.7205 Length 300
Best T 265.7205 Best energy [  2073.45559606  12855.18660389]
Epoch 8
Temperature 239.14845000000003 Length 360
Best T 239.14845000000003 Best energy [  2066.55282596  12736.91677083]
Epoch 9
Temperature 215.23360500000004 Length 432
Best T 239.14845000000003 Best energy [  2066.55282596  12736.91677083]
Epoch 10
Temperature 193.71024450000004 Length 519
Best T 193.71024450000004 Best energy [  1860.48419661  11455.30816441]
Epoch 11
Temperature 174.33922005000005 Length 623
Best T 193.71024450000004 Best energy [  1860.48419661  11455.30816441]
Epoch 12
Temperature 156.90529804500005 Length 748
Best T 156.90529804500005 Best energy [  1813.78434752  10622.09214448]
```

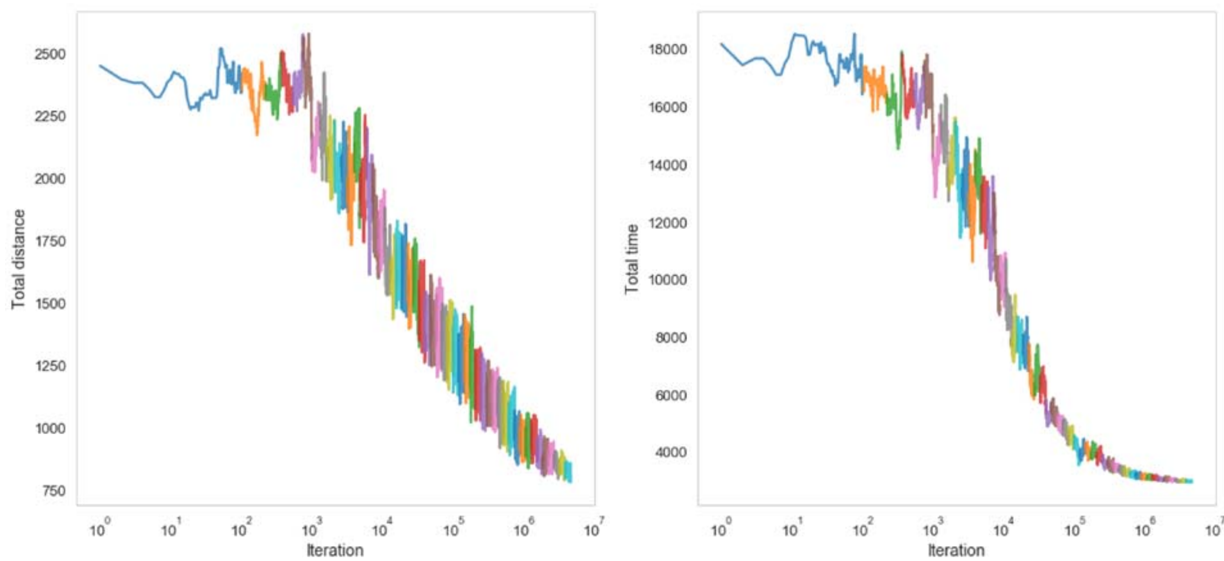
```
Epoch 13
Temperature 141.21476824050006 Length 898
Best T 156.90529804500005 Best energy [  1813.78434752  10622.09214448]
Epoch 14
Temperature 127.09329141645006 Length 1078
Best T 156.90529804500005 Best energy [  1813.78434752  10622.09214448]
Epoch 15
Temperature 114.38396227480506 Length 1294
Best T 114.38396227480506 Best energy [ 1644.21923814  9977.48978826]
Epoch 16
Temperature 102.94556604732455 Length 1553
Best T 102.94556604732455 Best energy [ 1691.94898618  8769.81831885]
Epoch 17
Temperature 92.6510094425921 Length 1864
Best T 102.94556604732455 Best energy [ 1691.94898618  8769.81831885]
Epoch 18
Temperature 83.3859084983329 Length 2237
Best T 83.3859084983329 Best energy [ 1695.37713896  7668.8018691 ]
Epoch 19
Temperature 75.04731764849961 Length 2685
Best T 75.04731764849961 Best energy [ 1556.99314716  7138.63782997]
Epoch 20
Temperature 67.54258588364965 Length 3222
Best T 67.54258588364965 Best energy [ 1652.33511126  6877.01086346]
Epoch 21
Temperature 60.78832729528469 Length 3867
Best T 60.78832729528469 Best energy [ 1558.48740473  6820.33052073]
Epoch 22
Temperature 54.70949456575622 Length 4641
Best T 54.70949456575622 Best energy [ 1502.28942322  5840.54167077]
Epoch 23
Temperature 49.2385451091806 Length 5570
Best T 54.70949456575622 Best energy [ 1502.28942322  5840.54167077]
Epoch 24
Temperature 44.31469059826254 Length 6684
Best T 44.31469059826254 Best energy [ 1300.80758411  5348.39570061]
Epoch 25
Temperature 39.88322153843628 Length 8021
Best T 39.88322153843628 Best energy [ 1424.83886797  4907.65627076]
Epoch 26
Temperature 35.894899384592655 Length 9626
Best T 35.894899384592655 Best energy [ 1285.05712024  4899.64536005]
Epoch 27
Temperature 32.30540944613339 Length 11552
Best T 32.30540944613339 Best energy [ 1321.30106989  4650.41395439]
Epoch 28
Temperature 29.07486850152005 Length 13863
Best T 29.07486850152005 Best energy [ 1214.26643248  4274.04562546]
Epoch 29
Temperature 26.167381651368046 Length 16636
Best T 26.167381651368046 Best energy [ 1349.25602757  4093.90028712]
Epoch 30
Temperature 23.55064348623124 Length 19964
Best T 23.55064348623124 Best energy [ 1146.38217173  3566.92310196]
Epoch 31
Temperature 21.195579137608117 Length 23957
Best T 23.55064348623124 Best energy [ 1146.38217173  3566.92310196]
```

```
Epoch 32
Temperature 19.076021223847306 Length 28749
Best T 23.55064348623124 Best energy [ 1146.38217173  3566.92310196]
Epoch 33
Temperature 17.168419101462575 Length 34499
Best T 23.55064348623124 Best energy [ 1146.38217173  3566.92310196]
Epoch 34
Temperature 15.451577191316318 Length 41399
Best T 23.55064348623124 Best energy [ 1146.38217173  3566.92310196]
Epoch 35
Temperature 13.906419472184686 Length 49679
Best T 13.906419472184686 Best energy [ 1110.91763364  3392.17072761]
Epoch 36
Temperature 12.515777524966218 Length 59615
Best T 12.515777524966218 Best energy [ 1029.85788486  3295.82391098]
Epoch 37
Temperature 11.264199772469597 Length 71538
Best T 12.515777524966218 Best energy [ 1029.85788486  3295.82391098]
Epoch 38
Temperature 10.137779795222638 Length 85846
Best T 10.137779795222638 Best energy [  995.18029584  3189.84072082]
Epoch 39
Temperature 9.124001815700375 Length 103016
Best T 9.124001815700375 Best energy [  945.17518142  3149.49818588]
Epoch 40
Temperature 8.211601634130338 Length 123620
Best T 8.211601634130338 Best energy [  940.97330194  3077.46805609]
Epoch 41
Temperature 7.390441470717304 Length 148344
Best T 7.390441470717304 Best energy [  906.09675942  3058.42483291]
Epoch 42
Temperature 6.651397323645574 Length 178013
Best T 6.651397323645574 Best energy [  879.0771514   3053.64068087]
Epoch 43
Temperature 5.986257591281016 Length 213616
Best T 5.986257591281016 Best energy [  888.79064128  3038.25564386]
Epoch 44
Temperature 5.387631832152914 Length 256340
Best T 5.387631832152914 Best energy [  871.20323183  3014.44526123]
Epoch 45
Temperature 4.848868648937623 Length 307608
Best T 4.848868648937623 Best energy [  842.92251259  2994.14788083]
Epoch 46
Temperature 4.363981784043861 Length 369130
Best T 4.363981784043861 Best energy [  831.79221181  2984.6673769 ]
Epoch 47
Temperature 3.927583605639475 Length 442956
Best T 3.927583605639475 Best energy [  834.34963944  2974.54023751]
Epoch 48
Temperature 3.5348252450755275 Length 531548
Best T 3.5348252450755275 Best energy [  817.58820827  2955.48419604]
Epoch 49
Temperature 3.181342720567975 Length 637858
Best T 3.5348252450755275 Best energy [  817.58820827  2955.48419604]
Epoch 50
Temperature 2.8632084485111777 Length 765430
Best T 3.5348252450755275 Best energy [  817.58820827  2955.48419604]
```

```
Frac accepted 0.022511576128273713 Total iterations 4591993 Best meta {'inde
x': 3042743, 'temp': 3.5348252450755275}
Wall time: 4min 51s
```

In [9]:

```python
plt.figure(figsize=(18, 8))
plt.subplot(1, 2, 1)

for i_start, i_end in tsp2.T_index:
    plt.plot(range(i_start + 1, i_end + 1), tsp2.accumulator[i_start:i_end, 0
plt.xscale('log');
plt.xlabel('Iteration');
plt.ylabel('Total distance');

plt.subplot(1, 2, 2)
for i_start, i_end in tsp2.T_index:
    plt.plot(range(i_start + 1, i_end + 1), tsp2.accumulator[i_start:i_end, 1
plt.xscale('log');
plt.xlabel('Iteration');
plt.ylabel('Total time');
```

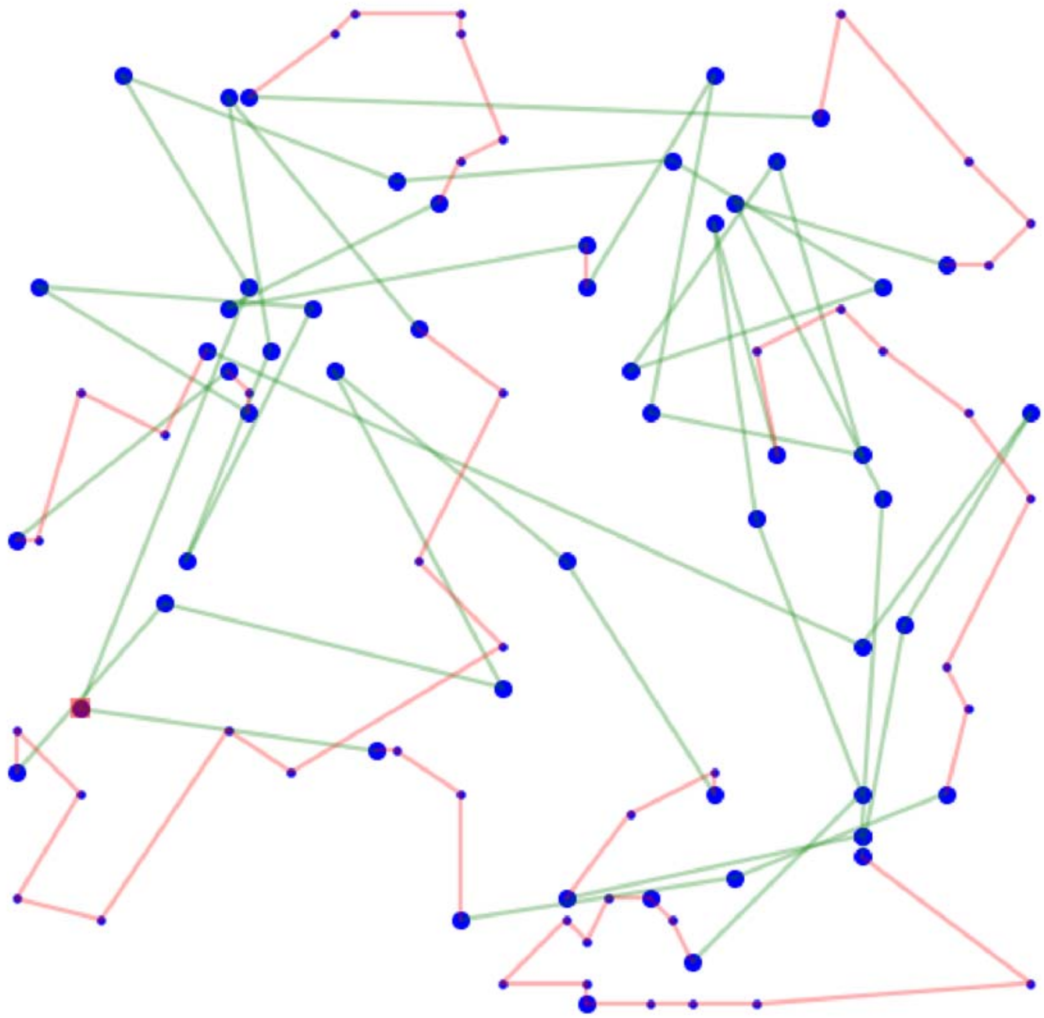


Observations:

1. Both total distance and total time decrease over iterations. While total time seems to converge in the end, total distance seems to converge much slower.
2. The fluctuations decrease as the temperature decreases.
3. The percentage of decrease in total distance is lower than that in total time, which is opposite to the observation in the previous part.
4. The final total distance is much higher than that in the previous part, while the final total time is much lower.

We can visualize the solution on the city map. Large cities are represented by large circles, and small cities are represented by small dots. Starting point is indicated by a red square. Drivings are represented by red lines and flights are represented by green lines.

In [10]:
```
plt.figure(figsize=(10, 10))
tsp2.map_.plot_cities()
tsp2.map_.plot_tour(tsp2.best_solution)
```



Observations:

1. The solution is dominated by flights in this case.
2. There are many long paths among large cities, where flights are available.

The solution looks good and reasonable.

Problem 2.3

We keep the same initialization and cooling schedule; we change 2 thresholds in turn. We use the same random seed in each experiment.

2.3.1 Optimization for the total distance

In [11]:

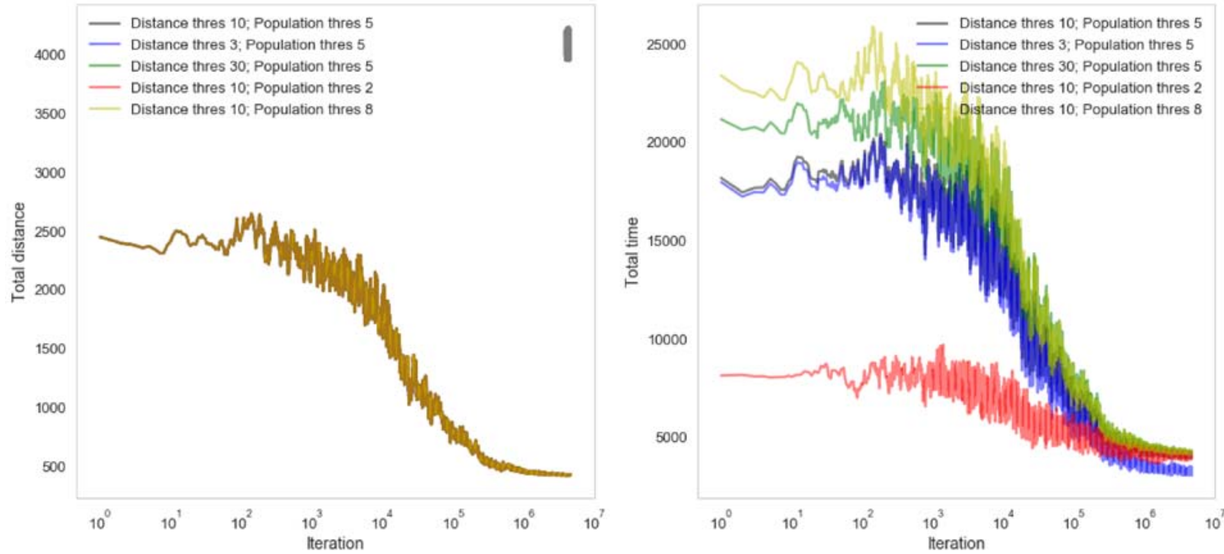
```
%%time

thres = [(3, 5), (30, 5), (10, 2), (10, 8)]
N = 100
tsp31 = []

def tf(T):
    return 0.9 * T

def itf(length):
    return int(np.ceil(1.2 * length))

print('Optimize for the total distance')
for t in thres:
    m = Map(N, dist_thres=t[0], pop_thres=t[1]).create_cities()

    ef = functools.partial(init_tour, m)

    pf = functools.partial(change_tour, m)

    inits = dict(solution=list(range(N)), length=100, T=80)
    epochs = 50

    print('Distance threshold {}, Population threshold {}'.format(t[0], t[1])
    tsp31.append(SA_TSP(m, 0).run_sa(ef, inits, epochs, tf, itf, pf, verbose=
```

```
Optimize for the total distance
Distance threshold 3, Population threshold 5
Frac accepted 0.019436005237812862 Total iterations 4591993 Best meta {'index':
4238818, 'temp': 0.4581133517617881}
Distance threshold 30, Population threshold 5
Frac accepted 0.019436005237812862 Total iterations 4591993 Best meta {'index':
4238818, 'temp': 0.4581133517617881}
Distance threshold 10, Population threshold 2
Frac accepted 0.019436005237812862 Total iterations 4591993 Best meta {'index':
4238818, 'temp': 0.4581133517617881}
Distance threshold 10, Population threshold 8
Frac accepted 0.019436005237812862 Total iterations 4591993 Best meta {'index':
4238818, 'temp': 0.4581133517617881}
Wall time: 19min 18s
```

In [12]:

```
%%time

plt.figure(figsize=(18, 8))
plt.subplot(1, 2, 1)

colors = ['b', 'g', 'r', 'y']

for i_start, i_end in tsp1.T_index:
    if i_end != tsp1.T_index[-1][-1]:
        plt.plot(range(i_start + 1, i_end + 1), tsp1.accumulator[i_start:i_en
                 alpha=0.5, color='k')
    else:
        plt.plot(range(i_start + 1, i_end + 1), tsp1.accumulator[i_start:i_en
                 alpha=0.5, color='k', label='Distance thres {}; Population t
plt.xscale('log');
plt.xlabel('Iteration');
plt.ylabel('Total distance');

for i, tsp in enumerate(tsp31):
    for i_start, i_end in tsp.T_index:
        if i_end != tsp.T_index[-1][-1]:
            plt.plot(range(i_start + 1, i_end + 1), tsp.accumulator[i_start:i
                     alpha=0.5, color=colors[i])
        else:
            plt.plot(range(i_start + 1, i_end + 1), tsp.accumulator[i_start:i
                     alpha=0.5, color=colors[i], \
                     label='Distance thres {}; Population thres {}'.format(th

plt.legend();

plt.subplot(1, 2, 2)
for i_start, i_end in tsp1.T_index:
    if i_end != tsp1.T_index[-1][-1]:
        plt.plot(range(i_start + 1, i_end + 1), tsp1.accumulator[i_start:i_en
                 alpha=0.5, color='k')
    else:
        plt.plot(range(i_start + 1, i_end + 1), tsp1.accumulator[i_start:i_en
                 alpha=0.5, color='k', label='Distance thres {}; Population t
plt.xscale('log');
plt.xlabel('Iteration');
plt.ylabel('Total time');

for i, tsp in enumerate(tsp31):
    for i_start, i_end in tsp.T_index:
        if i_end != tsp.T_index[-1][-1]:
            plt.plot(range(i_start + 1, i_end + 1), tsp.accumulator[i_start:i
                     alpha=0.5, color=colors[i])
        else:
            plt.plot(range(i_start + 1, i_end + 1), tsp.accumulator[i_start:i
                     alpha=0.5, color=colors[i], \
                     label='Distance thres {}; Population thres {}'.format(th

plt.legend();
```

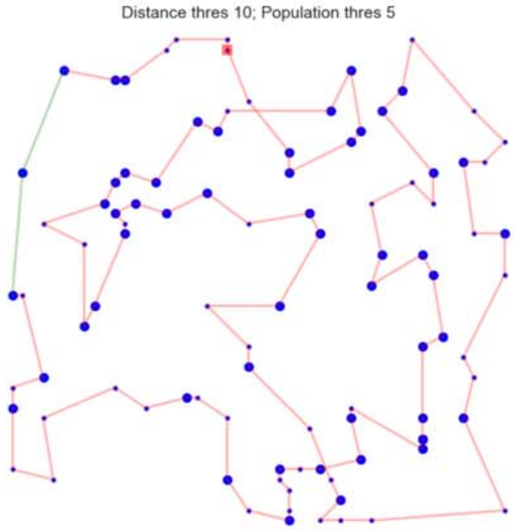
```
Wall time: 10.4 s
```

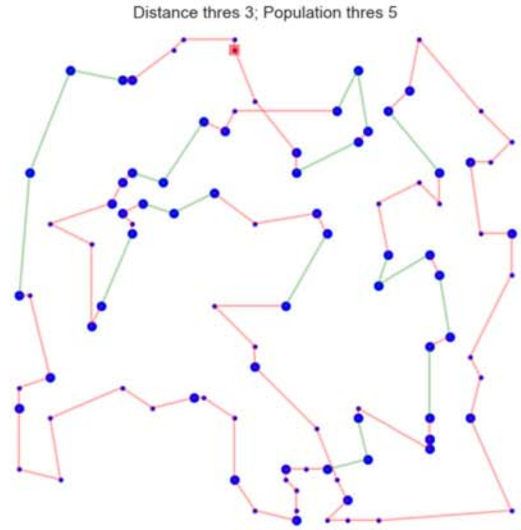


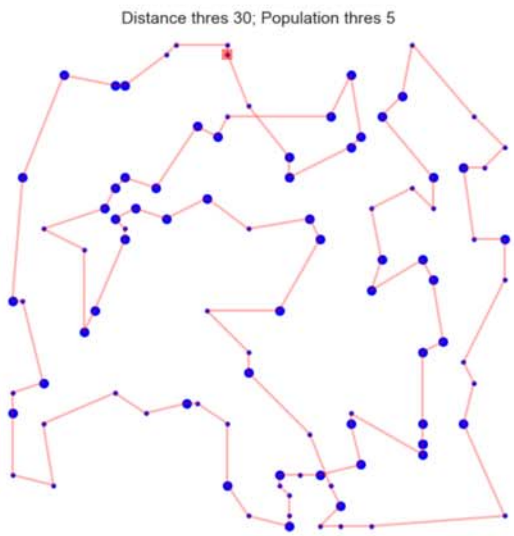
The curves of total distance over iterations overlap, since we use the same random seed for each experiment. The curves of total time over iterations differ, since the available transportation for the same path might change.

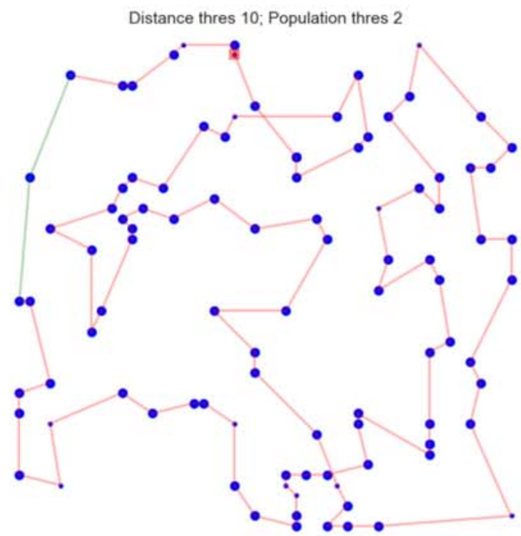
In [13]:
```
%%time

plt.figure(figsize=(20, 30))
plt.subplot(3, 2, 1)
tsp1.map_.plot_cities()
tsp1.map_.plot_tour(tsp1.best_solution)
plt.title('Distance thres {}; Population thres {}'.format(10, 5))

for i, tsp in enumerate(tsp31):
    plt.subplot(3, 2, i + 2)
    tsp.map_.plot_cities()
    tsp.map_.plot_tour(tsp.best_solution)
    plt.title('Distance thres {}; Population thres {}'.format(thres[i][0], th
```

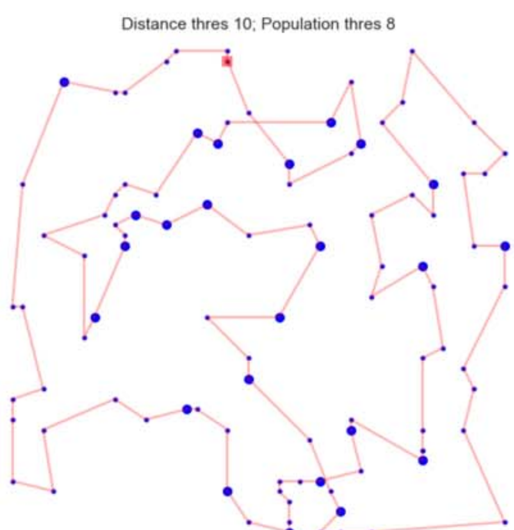
```
Wall time: 790 ms
```











We get the same path in these cases, since the distance between 2 cities wouldn't change when we alter the distance and the population distance.

2.3.2 Optimization for the total time

In [14]:

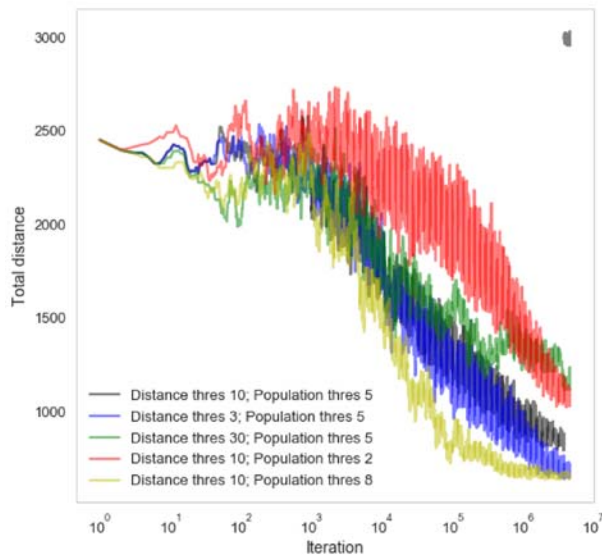
```
%%time

thres = [(3, 5), (30, 5), (10, 2), (10, 8)]
N = 100
tsp32 = []

def tf(T):
    return 0.9 * T

def itf(length):
    return int(np.ceil(1.2 * length))

print('Optimize for the total time')
for t in thres:
    m = Map(N, dist_thres=t[0], pop_thres=t[1]).create_cities()

    ef = functools.partial(init_tour, m)

    pf = functools.partial(change_tour, m)

    inits = dict(solution=list(range(N)), length=100, T=500)
    epochs = 50

    print('Distance threshold {}, Population threshold {}'.format(t[0], t[1])
    tsp32.append(SA_TSP(m, 1).run_sa(ef, inits, epochs, tf, itf, pf, verbose=
```

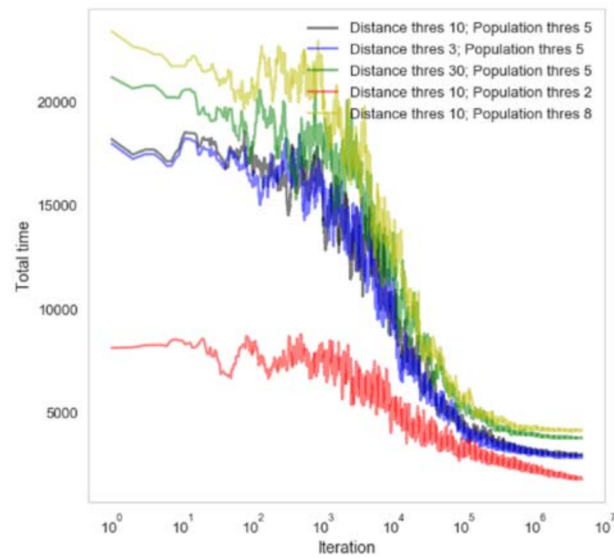
```
Optimize for the total time
Distance threshold 3, Population threshold 5
Frac accepted 0.030566901996584055 Total iterations 4591993 Best meta {'index':
3873599, 'temp': 2.8632084485111777}
Distance threshold 30, Population threshold 5
Frac accepted 0.015264178320829321 Total iterations 4591993 Best meta {'index':
3992796, 'temp': 2.8632084485111777}
Distance threshold 10, Population threshold 2
Frac accepted 0.05196284053568897 Total iterations 4591993 Best meta {'index':
4039425, 'temp': 2.8632084485111777}
Distance threshold 10, Population threshold 8
Frac accepted 0.014901155119356671 Total iterations 4591993 Best meta {'index':
3490414, 'temp': 3.181342720567975}
Wall time: 19min 18s
```

In [15]:

```
%%time

plt.figure(figsize=(18, 8))
plt.subplot(1, 2, 1)

colors = ['b', 'g', 'r', 'y']

for i_start, i_end in tsp2.T_index:
    if i_end != tsp2.T_index[-1][-1]:
        plt.plot(range(i_start + 1, i_end + 1), tsp2.accumulator[i_start:i_en
                 alpha=0.5, color='k')
    else:
        plt.plot(range(i_start + 1, i_end + 1), tsp2.accumulator[i_start:i_en
                 alpha=0.5, color='k', label='Distance thres {}; Population t
plt.xscale('log');
plt.xlabel('Iteration');
plt.ylabel('Total distance');

for i, tsp in enumerate(tsp32):
    for i_start, i_end in tsp.T_index:
        if i_end != tsp.T_index[-1][-1]:
            plt.plot(range(i_start + 1, i_end + 1), tsp.accumulator[i_start:i
                     alpha=0.5, color=colors[i])
        else:
            plt.plot(range(i_start + 1, i_end + 1), tsp.accumulator[i_start:i
                     alpha=0.5, color=colors[i], \
                     label='Distance thres {}; Population thres {}'.format(th

plt.legend();

plt.subplot(1, 2, 2)
for i_start, i_end in tsp2.T_index:
    if i_end != tsp2.T_index[-1][-1]:
        plt.plot(range(i_start + 1, i_end + 1), tsp2.accumulator[i_start:i_en
                 alpha=0.5, color='k')
    else:
        plt.plot(range(i_start + 1, i_end + 1), tsp2.accumulator[i_start:i_en
                 alpha=0.5, color='k', label='Distance thres {}; Population t
plt.xscale('log');
plt.xlabel('Iteration');
plt.ylabel('Total time');

for i, tsp in enumerate(tsp32):
    for i_start, i_end in tsp.T_index:
        if i_end != tsp.T_index[-1][-1]:
            plt.plot(range(i_start + 1, i_end + 1), tsp.accumulator[i_start:i
                     alpha=0.5, color=colors[i])
        else:
            plt.plot(range(i_start + 1, i_end + 1), tsp.accumulator[i_start:i
                     alpha=0.5, color=colors[i], \
                     label='Distance thres {}; Population thres {}'.format(th

plt.legend();
```

```
Wall time: 10.3 s
```





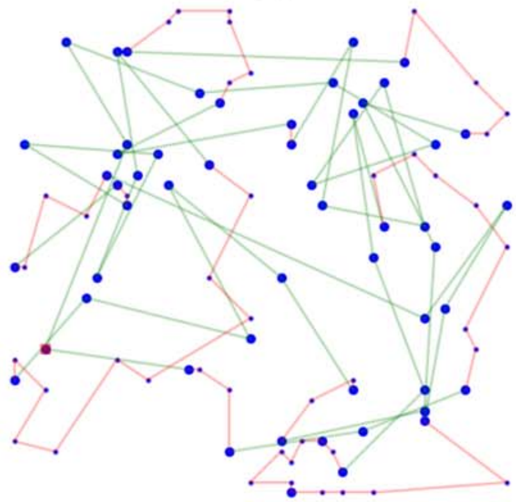
Observations:

1. When we increase the distance threshold (and keep the same population threshold), both the total time and the total distance of the result increase.
2. When we decrease the distance threshold (and keep the same population threshold), the total time of the result doesn't seem to change much but the total distance decreases.
3. When we increase the population threshold (and keep the same distance threshold), the total time of the result increases and the total distance decreases. It is reasonable since there would be less flights and the salesman would have to drive more frequently; so it is better to pick up shorter paths.
4. When we decrease the population threshold (and keep the same distance threshold), the total time of the result decreases and the totla distance increases. The salesman would take more flights.

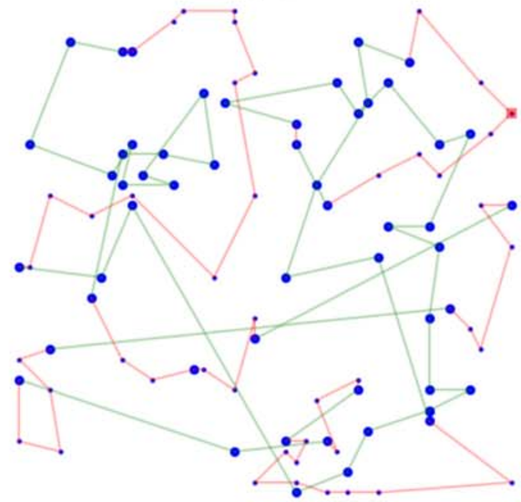
In [16]:
```
%%time

plt.figure(figsize=(20, 30))
plt.subplot(3, 2, 1)
tsp2.map_.plot_cities()
tsp2.map_.plot_tour(tsp2.best_solution)
plt.title('Distance thres {}; Population thres {}'.format(10, 5))

for i, tsp in enumerate(tsp32):
    plt.subplot(3, 2, i + 2)
    tsp.map_.plot_cities()
    tsp.map_.plot_tour(tsp.best_solution)
    plt.title('Distance thres {}; Population thres {}'.format(thres[i][0], th
```

```
Wall time: 1.2 s
```

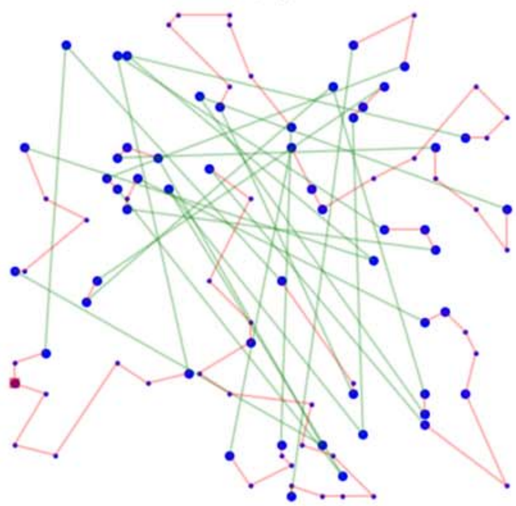
Distance thres 10; Population thres 5



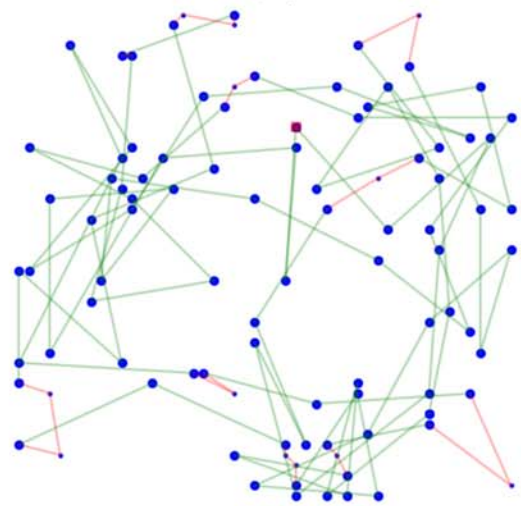
Distance thres 3; Population thres 5



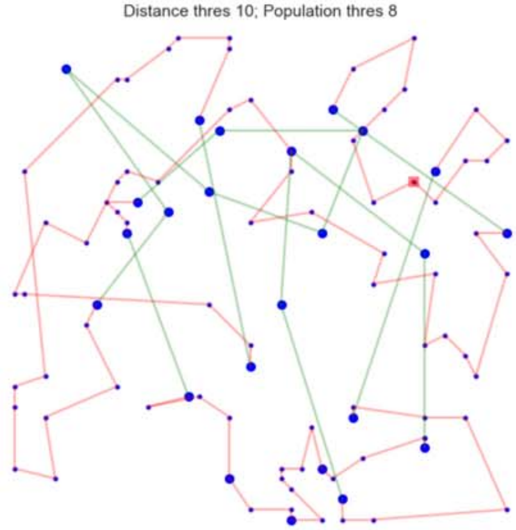
Distance thres 30; Population thres 5



Distance thres 10; Population thres 2



Distance thres 10; Population thres 8



Observations:

1. When we increase the distance threshold (and keep the same population threshold), the result seems to be more tangled (more intersections) and there seem to be more long flights.
2. When we decrease the distance threshold (and keep the same population threshold), the result seems to be less tangled (less intersections).
3. When we increase the population threshold (and keep the same distance threshold), the result includes less long paths and there are less flights, presumably due to the decrease in the number of large cities.
4. When we decrease the population threshold (and keep the same distance threshold), there are more flights and the result seems to be more tangled (more intersections).