Model I. Locations of N drones' platform and warehouse

Introduction:

- Target: Define the most appropriate locations of drones' platform and warehouses
- Input: The number(N) of warehouse, order sample with address(sample size 200)
- Output: N suitable locations of drones' platform and warehouses

Model special function:

- Automaticly convert address into earth distance by using google map geocoding api
- Automaticly calculate distance between order and warehouses by calling geopy function, getcircle
- Apply optimization by using IBM Cplex to figure out N most appropriate locations

Wrote by NI(Mani) MAN, Oct 29, 2017

```
    - Basic library - sys, docplex
```

In [1]:
```
import sys
try:
    import docplex.mp
except:
    if hasattr(sys, 'real_prefix'):
        !pip install docplex
    else:
        !pip install --user docplex
```

```
    - Setup docplex engine
    In this case, we use cplex cloud to solve optimization problem. Please be noted that this ca
se could also be solved by Cplex community with preinstalled IBM Cplex software.
```

In [2]:
```
url = "https://api-oaas.docloud.ibmcloud.com/job_manager/rest/v1/"
key = "api_3d568d67-19d8-4b62-894b-d31981859867"
```

- Step 1 - Import dataset and manipulate the dataset
    - We will collect the list of address of each order. And then, we will convert those addresses into ponit (longitude,latitute), which could be used to calculate the distance.

```
- Define NamedPoint to collect information with same factors.
```

In [3]:

```
class XPoint(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __str__(self):
        return "P(%g_%g)" % (self.x, self.y)

class NamedPoint(XPoint):
    def __init__(self, name, x, y):
        XPoint.__init__(self, x, y)
        self.name = name
    def __str__(self):
        return self.name
```

```
- Import library - geopy.distance
- Define Function 1: computing the earth distance between 2 points on map
```

In [4]:

```
try:
    import geopy.distance
except:
    if hasattr(sys, 'real_prefix'):
        #we are in a virtual env.
        !pip install geopy
    else:
        !pip install --user geopy

import geopy.distance
# Simple distance computation between 2 locations.
from geopy.distance import great_circle

def get_distance(p1, p2):
    return great_circle((p1.y, p1.x), (p2.y, p2.x)).miles
```

- Import library - googlemaps, json
- Define Function 2: Exchange the address into latitude and longitude, collect lat and lng

```
In [5]: import json
        import googlemaps

        def getlat(address):
            gmap = googlemaps.Client(key='AIzaSyD2fUATZAJtzmVCTQi5Fe6xpboAgR5-7J4')
            origin_earth_location = gmap.geocode(address)
            lat=origin_earth_location[0]['geometry']['location']['lat']
            return lat
        def getlng(address):
            gmap = googlemaps.Client(key='AIzaSyD2fUATZAJtzmVCTQi5Fe6xpboAgR5-7J4')
            origin_earth_location = gmap.geocode(address)
            lng = origin_earth_location[0]['geometry']['location']['lng']
            return lng
```

- Import library - xlrd
- Load required dataset
- "order" list represent a collection of all destination addresses of historical orders

```
In [6]: import xlrd
        mydataset=xlrd.open_workbook('Order_target_address.xlsx')
        mytable = mydataset.sheets()[0]

        order = []

        pos = 1
        order.extend(mytable.col_values(pos)[1:])
        #a = len(order)
        #print(order, a)
```

- Create empty list of lat, lng and warehouse_lat_lng list.
- Apply function 2 to each address of warehouse list

```
    - And then use warehouse_lat_lng to collect information of address_with_point(address, longi
tude, latitude)
```

In [7]:
```python
#empty list to collect latitude, longitude and address with point info.
lat = []
lng = []
order_lat_lng = []

for i in range(len(order)):
    latitude = getlat(order[i])
    longitude = getlng(order[i])
    name = order[i]
    cp = NamedPoint(name, longitude, latitude)
    order_lat_lng.append(cp)

print("There are %d orders in NYC" % (len(order_lat_lng)))
```

```
There are 200 orders in NYC
```

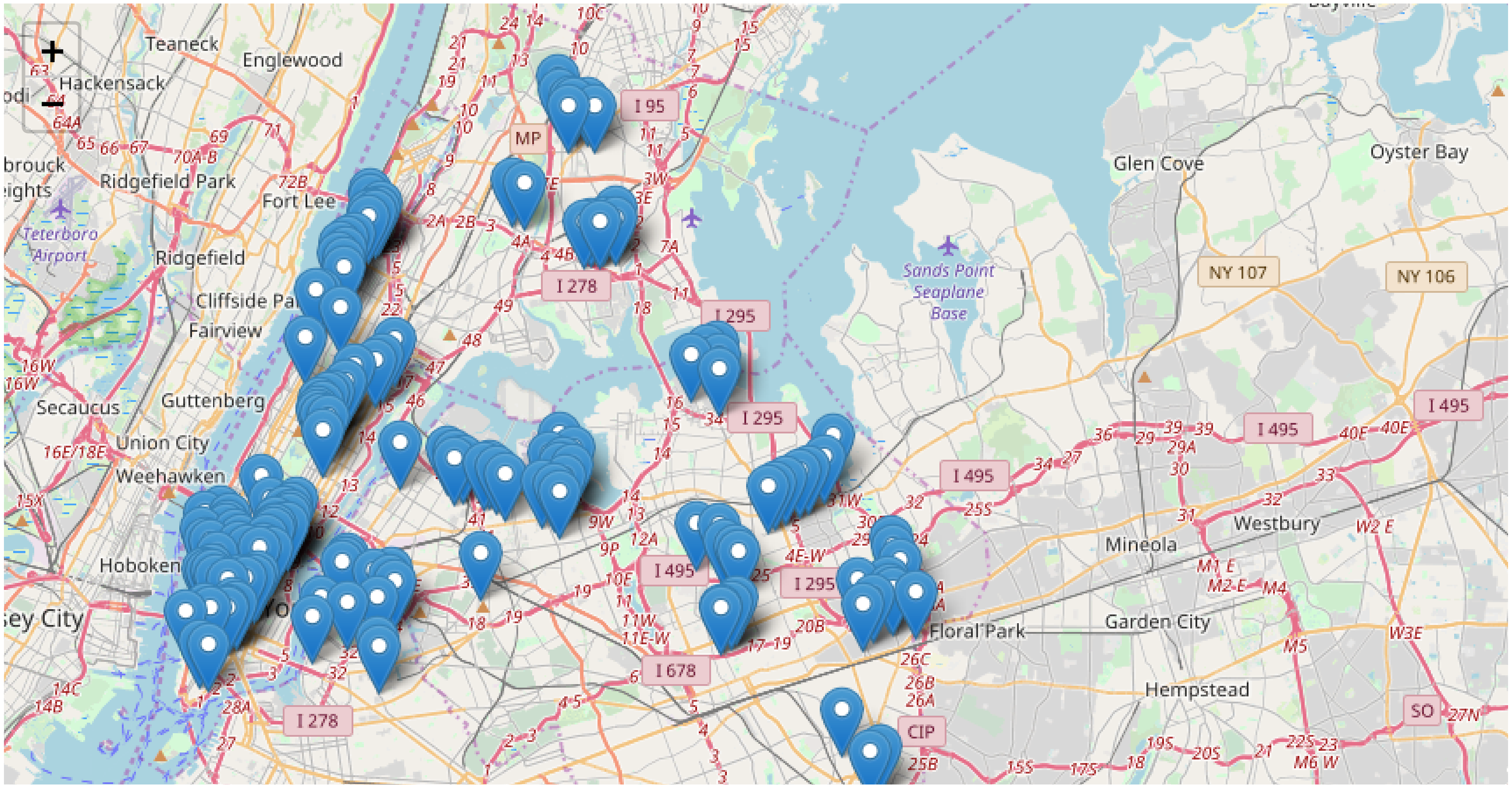
- Step 2 - Visualizaiton of data

```
    - import library folium
    - design folium visualization
    - print map
```

In [8]:
```
try:
    import folium
except:
    if hasattr(sys, 'real_prefix'):
        #we are in a virtual env.
        !pip install folium
    else:
        !pip install --user folium

import folium
map_osm = folium.Map(location=[40.712765, -73.950882], zoom_start=11)
for order in order_lat_lng:
    lt = order.y
    lg = order.x
    folium.Marker([lt, lg]).add_to(map_osm)
map_osm
```

Out[8]:



*Please be noted that this map just represent the result of order list with 200 sample size. Since we only generated the historical order list, it may not represent the real case.

*More data and real data should be input to get a more accurative result.

- Step 3: Optimization of location of warehouse
  Set up DOcplex to write and solve an optimization model that will help us determine where to locate the warehouse in an optimal way.

```
  - Define number of warehouse required to be built
  The number, 7, of warehouses is come from our estimated analysis of NYC(excluding Stanley Is
land)
```

In [9]:
```
nb_warehouse = 7
print("We would like to open %d warehouses." % nb_warehouse)
```

```
We would like to open 7 warehouses.
```

```
 - Set up the prescriptive model
 - create docplex model
```

In [10]:
```
from docplex.mp.environment import Environment
env = Environment()
env.print_information()
from docplex.mp.model import Model
mdl = Model("Warehouse location")
```

```
* system is: Darwin 64bit
* Python is present, version is 3.5.4
* docplex is present, version is (2, 3, 44)
* CPLEX wrapper is present, version is 12.7.0.0, located at: /Applications/anaconda3/envs/cplexenv/li
b/python3.5/site-packages
```

- Define the Decision Variables

In [11]:
```
# Ensure unique points
BIGNUM = 999999999

order_lat_lng = set(order_lat_lng)
warehouse_locations = order_lat_lng

# Decision vars
# Binary vars indicating which warehouse locations will be actually selected
warehouse_vars = mdl.binary_var_dict(warehouse_locations, name="is_warehouse")

# Binary vars representing the "assigned" order for each warehouse
link_vars = mdl.binary_var_matrix(warehouse_locations, order_lat_lng, "link")
```

- Define the constraints

#constraint 1: if the distance is suspect, it needs to be excluded from the problem.

In [12]:
```
for c_loc in warehouse_locations:
    for b in order_lat_lng:
        if get_distance(c_loc, b) >= BIGNUM:
            mdl.add_constraint(link_vars[c_loc, b] == 0, "ct_forbid_{0!s}_{1!s}".format(c_loc, b))
```

#constraint 2: each order must be linked to a warehouse that is open.

```
In [13]: mdl.add_constraints(link_vars[c_loc, b] <= warehouse_vars[c_loc]
                             for b in order_lat_lng
                             for c_loc in warehouse_locations)
         mdl.print_information()
```

```
Model: Warehouse location
 - number of variables: 40200
   - binary=40200, integer=0, continuous=0
 - number of constraints: 40000
   - linear=40000
 - parameters: defaults
```

#constraint 3: each order is linked to exactly one warehouse.

```
In [14]: mdl.add_constraints(mdl.sum(link_vars[c_loc, b] for c_loc in warehouse_locations) == 1
                             for b in order_lat_lng)
         mdl.print_information()
```

```
Model: Warehouse location
 - number of variables: 40200
   - binary=40200, integer=0, continuous=0
 - number of constraints: 40200
   - linear=40200
 - parameters: defaults
```

#constraint 4: there is a fixed number of warehouse to open.

In [15]:
```python
# Total nb of open warehouse
mdl.add_constraint(mdl.sum(warehouse_vars[c_loc] for c_loc in warehouse_locations) == nb_warehouse)

# Print model information
mdl.print_information()
```

```
Model: Warehouse location
 - number of variables: 40200
   - binary=40200, integer=0, continuous=0
 - number of constraints: 40201
   - linear=40201
 - parameters: defaults
```

- Step 4 - Express the objective
  The objective is to minimize the total distance from order to warehouse so that customer always receive packages with less waiting time

In [16]:
```python
# Minimize total distance from points to hubs
total_distance = mdl.sum(link_vars[c_loc, b] * get_distance(c_loc, b) for c_loc in warehouse_locations f
mdl.minimize(total_distance)
```

```
        - Solve with the Decision Optimization solve service
```

In [17]:
```python
print("# Order = %d" % len(warehouse_locations))
print("# warehouse = %d" % nb_warehouse)

assert mdl.solve(url=url, key=key), "!!! Solve of the model fails"
```

```
# Order = 200
# warehouse = 7
```

```
        - create list of open_warehouse, not_warehouse, edges for visualization of data
```

```
In [18]: total_distance = mdl.objective_value
         open_warehouse = [c_loc for c_loc in warehouse_locations if warehouse_vars[c_loc].solution_value == 1]
         not_warehouse = [c_loc for c_loc in warehouse_locations if c_loc not in open_warehouse]
         edges = [(c_loc, b) for b in order_lat_lng for c_loc in warehouse_locations if int(link_vars[c_loc, b])

         print("Total distance = %g" % total_distance)
         print("# warehouse  = {0}".format(len(open_warehouse)))
         for c in open_warehouse:
             print("new warehouse: {0!s}".format(c))
```

```
Total distance = 288.635
# warehouse  = 7
new warehouse: 546 W 146th St, New York, NY 10031
new warehouse: 1370 E 18th St Brooklyn, NY 11230
new warehouse: 1035 Park Ave New York, NY  10028
new warehouse: 2552 Holland Ave Bronx, NY 10467
new warehouse: 88-35 212th Pl Queens Village, NY 11427
new warehouse: 207 E 15th St, New York, NY 10003
new warehouse: 32-23 100th St Flushing, NY 11369
```

- visualization of final results

In [19]:

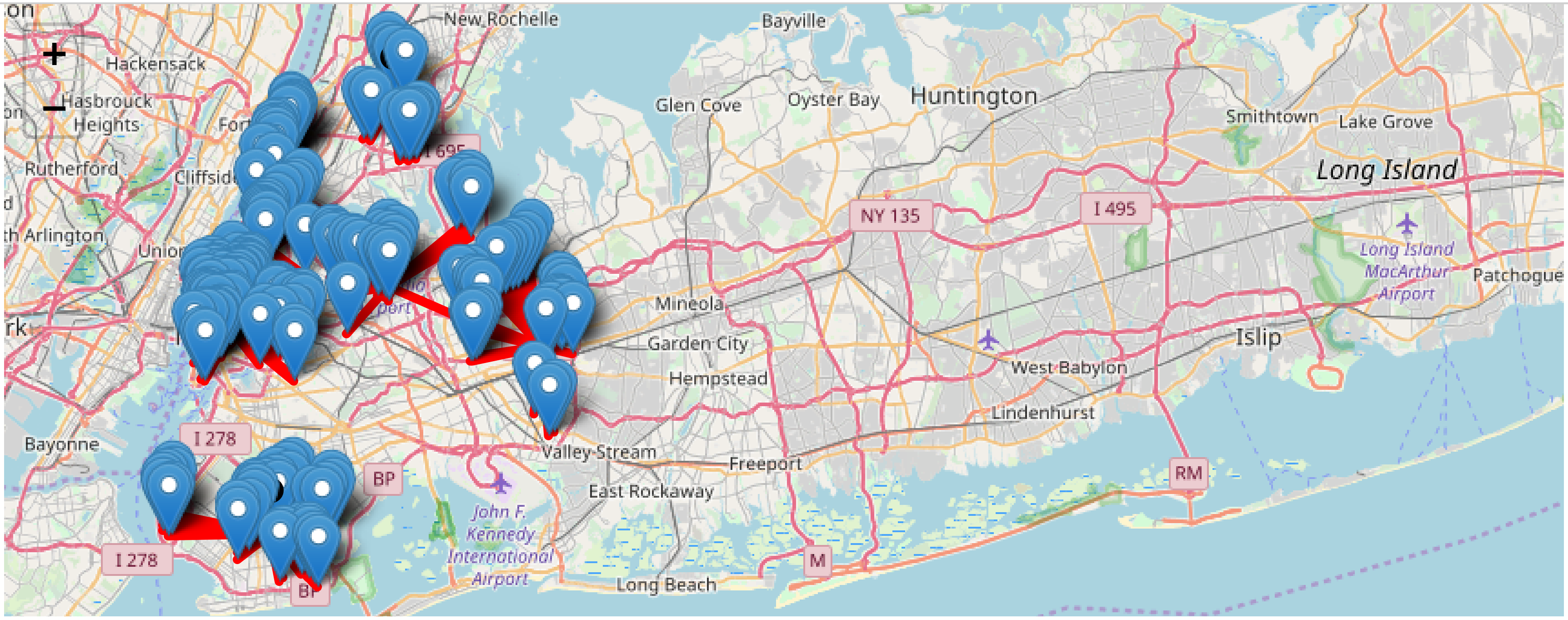
```
import folium
map_osm = folium.Map(location=[40.712765, -73.950882], zoom_start=11)
for warehouse in open_warehouse:
    lt = warehouse.y
    lg = warehouse.x
    folium.Marker([lt, lg], icon=folium.Icon(color='red',icon='info-sign')).add_to(map_osm)

for b in order_lat_lng:
    if b not in open_warehouse:
        lt = b.y
        lg = b.x
        folium.Marker([lt, lg]).add_to(map_osm)


for (c, b) in edges:
    coordinates = [[c.y, c.x], [b.y, b.x]]
    map_osm.add_child(folium.PolyLine(coordinates, color='#FF0000', weight=5))

map_osm
```

Out[19]:



End

```
In [ ]: #End
```