

Граф дорог города России и применение на нем алгоритмов

Реализация практического задания по
дисциплине “Теория конечных графов и
ее приложение”

Выполнила:
Алиева Сабина

Использованные технологии

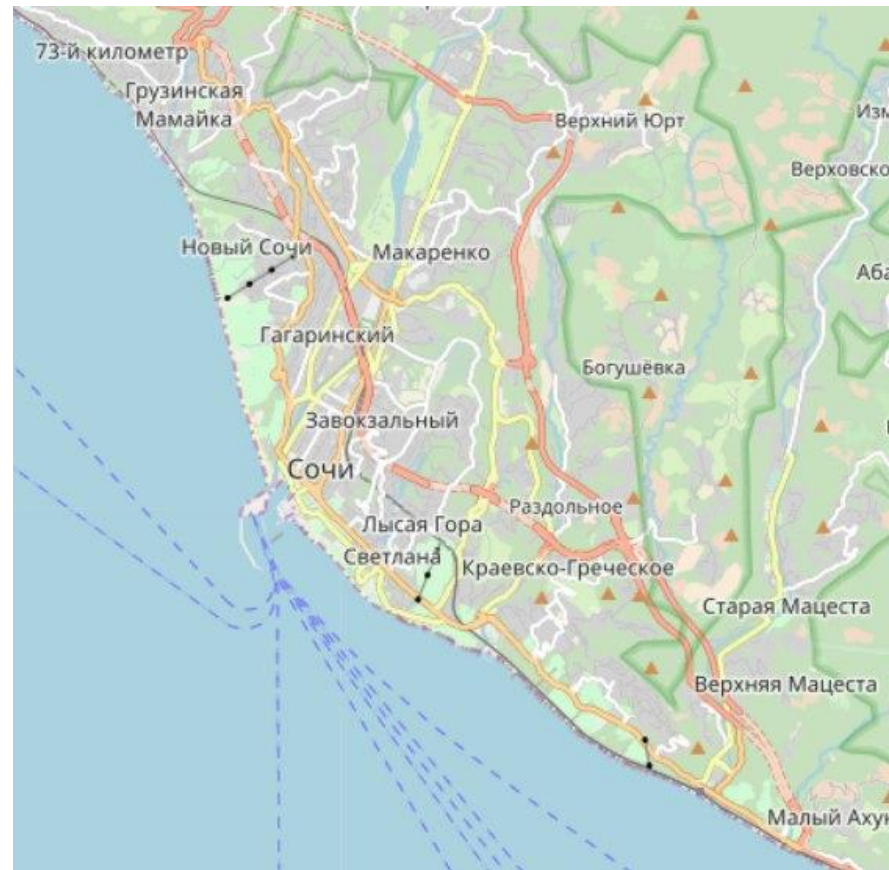
Язык программирование python 2.7

Среда разработки Microsoft Azure Notebooks

Использованные библиотеки:

- lxml - для обработки xml файла
- networkx - готовый интерфейс для работы с графами, реализованные алгоритмы и удобная контролируемая отрисовка
- matplotlib - отрисовка полученных изображений
- numpy - удобная работа с массивами
- csv - удобная работа с текстовыми данными
- heapq - очередь с приоритетом

Изначально был
выбран город
Сочи



Обработка файла xml osm

```
from lxml import etree, objectify
import matplotlib.pyplot as plt
import numpy as np
import csv
```

```
FILENAME = "taganrog.xml"
```

```
with open(FILENAME) as f:
    xml = f.read()
```

```
root = objectify.fromstring(xml)
```

```
road = []           #текущий массив
roads = []          #массив с дорогами, в каждой дороге указаны узлы
```

```
way = root.way
```

```
for row in way:      #проход по каждой дороге
    tag = row.find("tag")
    if not (tag is None) and tag.attrib['k'] == 'highway':
        nd = row.nd      #список узлов
        for ref in nd:    #пробег по каждому узлу
            road.append(int(ref.attrib['ref']))
        roads.append(road)
    road = []
```

Мы удаляем
вершину, если она
не встретилась ни
в одной дороге
или была
промежуточной в
дороге

```
node = root.node                #список всех узлов

#словарь, по номеру узла выдает координаты
nodes = {int(row.attrib['id']): [float(row.attrib['lat']), float(row.attrib['lon'])] for row in node}

#словарь, в котором номеру узла соответствует количество его вхождений в дороги
count_node_in_roads = {int(row.attrib['id']): 0 for row in node}

for road in roads:
    for nd in road:
        count_node_in_roads[int(nd)] = count_node_in_roads[int(nd)] + 1

#словарь, в котором номеру узла соответствует количество ребер ему инцидентных
count_edge_in_roads = {int(row.attrib['id']): 0 for row in node}
for road in roads:
    for i in range(len(road) - 1):
        count_edge_in_roads[road[i]] = count_edge_in_roads[road[i]] + 1
        count_edge_in_roads[road[i + 1]] = count_edge_in_roads[road[i + 1]] + 1

#массив необходимых к удалению узлов
nodes_to_delete = []
for nd in nodes:
    if count_node_in_roads[nd] == 0 or (count_node_in_roads[int(nd)] == 1 and count_edge_in_roads[int(
nd)] == 2):
        nodes_to_delete.append(nd)

#удаление лишних узлов
for nd in nodes_to_delete:
    nodes.pop(nd)
```

Запись списка и словаря смежности

```
#словарь списка смежности
list_adj = {nd:[] for nd in nodes}

row = []

for road in roads:
    for nd in road:
        if nd in nodes:      #если узел в этой дороге не лишний, записываем в текущую строку
            row.append(nd)
    for i in range(len(row) - 1): #две соседние точки в трое образуют ребро, запишем это в список
        list_adj[row[i]].append(int(row[i+1]))
        list_adj[row[i+1]].append(int(row[i]))
    row = []

#убираем повторения в списке смежности
for key in list_adj:
    list_adj[key] = list(set(list_adj[key]))
```

```
#записываем список смежности
line_csv = []
with open("list_taganrog.csv", "w") as csv_file:
    writer = csv.writer(csv_file, delimiter=',')
    for key in list_adj:
        line_csv.append(key)
        for r in list_adj[key]:
            line_csv.append(r)
        writer.writerow(line_csv)
    line_csv = []
```

Код для записи матрицы смежности. Матрица получалась слишком большой и компьютеру не хватило вычислительной мощности, поэтому файла в отчете не прилагается.

```
#записываем матрицу смежности
import csv

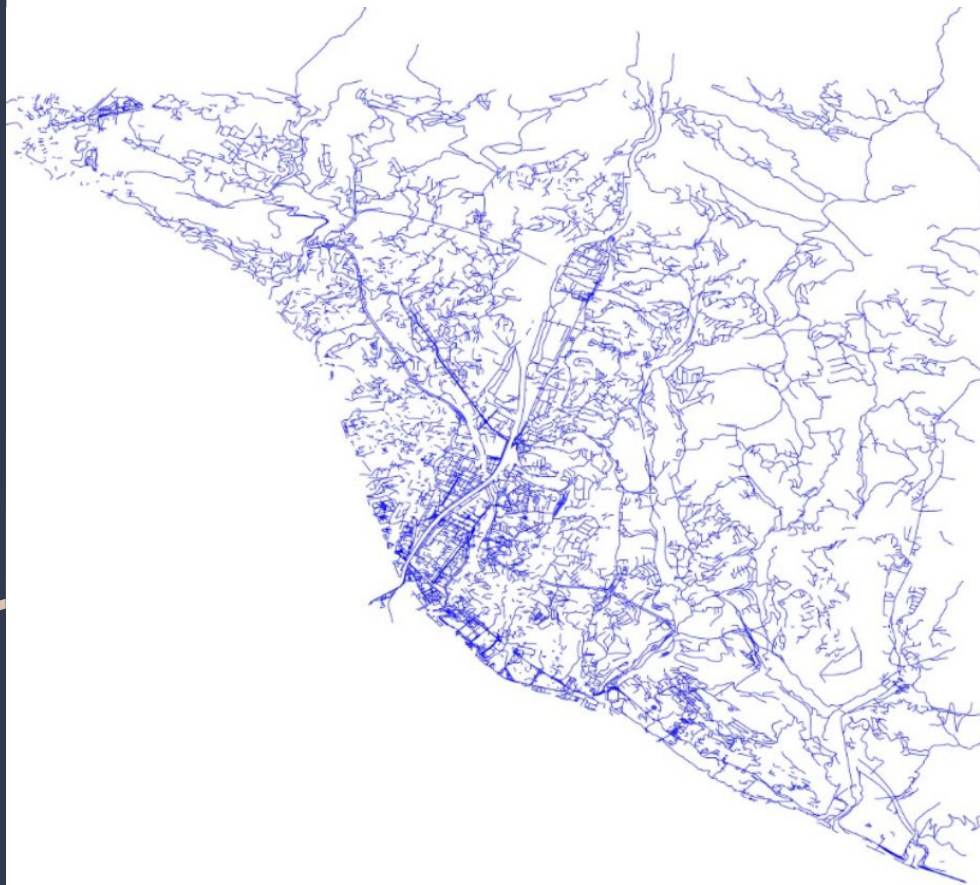
line = []
lline = np.zeros(len(list_adj) + 1)

with open("matr_sochi.csv", "w") as csv_file:
    writer = csv.writer(csv_file, delimiter=',')

    line.append(0)
    for key in list_adj:
        line.append(key)
        writer.writerow(line)

    for key in list_adj:
        for nd in list_adj[key]:
            lline[0] = key
            lline[line.index(nd)] = 1
        writer.writerow(lline)
        lline = np.zeros(len(list_adj) + 1)
```

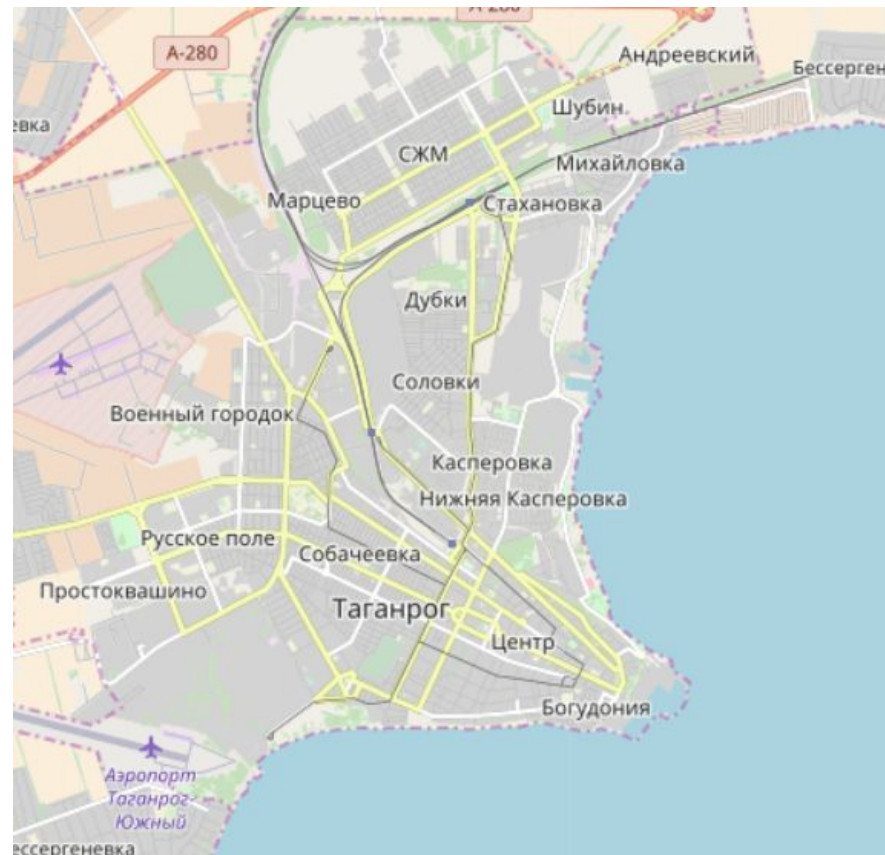

Возникли проблемы с компонентами связности





При более детальном рассмотрении видно, что очень много несвязных компонент

Поэтому был
выбран город
Таганрог



Код для оригинальной отрисовки города

```
fig = plt.gcf()
fig.set_size_inches(20, 24)    #установка большого размера полотна
Lon_Lat = []

for road in roads:              #проход по всем дорогам
    for index_node_in_road in road:    #проход по всем индексам узлов в дорогах
        Lon_Lat.append([nodes.get(index_node_in_road)[1], nodes.get(index_node_in_road)[0]])
    Lon_Lat = np.array(Lon_Lat)
    plt.plot(Lon_Lat[:, 0], Lon_Lat[:, 1], 'blue')
    Lon_Lat = []

fig.set_size_inches(20, 24, forward=True)
fig.savefig('taganrog_original.png', dpi=100)
plt.show()
```


Оригинальная отрисовка



Код для отрисовки после удаления лишних вершин

```
fig = plt.gcf()
fig.set_size_inches(56, 41)    #установка большого размера полотна
Lon_Lat = []

for key in list_adj:
    for nd in list_adj[key]:
        plt.plot([nodes[key][1], nodes[nd][1]], [nodes[key][0], nodes[nd][0]], 'blue')

fig.set_size_inches(56, 41, forward=True)
fig.savefig('taganrog.png', dpi=100)
plt.show()
```

Отрисовка “очищенного” графа



Далее было необходимо найти все больницы города, и вычислять из заданной точки кратчайшие расстояния до всех больниц. В Таганроге их всего оказалось 8. Так как больницы находятся не на дорогах, было необходимо пройти по всем узлам дорог, чтобы найти ближайшие к больницам точки.

```
#выбираем больницы  
hospitals = {}
```

```
#проход по каждому узлу  
for nd in node:  
    tag = nd.find("tag")  
    if not (tag is None) and tag.attrib['v'] == 'hospital':  
        hospitals[int(nd.attrib['id'])] = [float(nd.attrib['lat']), float(nd.attrib['lon'])]  
  
print len(hospitals)
```

8

```
#ищем ближайшие к больницам узлы на дорогах  
import math
```

```
target = []  
for h in hospitals:  
    min_w = 10000  
    nd_min = 0  
    for nd in nodes:  
        w = math.sqrt((nodes.get(nd)[0] - hospitals.get(h)[0])**2 + (nodes.get(nd)[1] - hospitals.get(h)[1])**2)  
        if w < min_w:  
            min_w = w  
            nd_mai = nd  
    target.append(nd_mai)  
  
print target
```

[1130256950, 1499967459, 3975857985, 211530735, 3898358242, 5422015747, 479475678, 3806624937]

Интерфейс для ввода координат стартовой точки. В качестве такой точки здесь и далее будет выступать адрес моего дома. Аналогично находим ближайшую к нему точку на дороге.

#интерфейс для ввода координат

```
def Input_lon():  
    print ('введите долготу между 38.84 и 38.96')  
    global lon  
    lon = input()  
    if lon < 38.84 or lon > 38.96:  
        print ('введите долготу заново')  
        Input_lon()
```

```
def Input_lat():  
    print ('введите широту между 47.20 и 47.28')  
    global lat  
    lat = input()  
    if lat < 47.20 or lat > 47.28:  
        print ('введите широту заново')  
        Input_lat()
```

```
Input_lon()  
Input_lat()  
# 47.261885  
# 38.908703
```

#ищем ближайшую к введенной координате точку на дороге

```
min_w = 1000000  
for nd in nodes:  
    w = math.sqrt((nodes.get(nd)[0] - lat)**2 + (nodes.get(nd)[1] - lon)**2)  
    if w < min_w:  
        min_w = w  
        start = nd
```

Нахождение кратчайших путей



Реализация алгоритма Дейкстры



```
#алгоритм Дейкстры  
from heapq import heappush, heappop
```

```
def Dejcstra(a, b):
```

```
    len_sh_way = {nd: 1000000 for nd in nodes} # вершине соответствует длина пути до нее  
    len_sh_way.update({a: 0})
```

```
    shortest_way = {nd: [] for nd in nodes} #кратчайший путь от вершины a  
    shortest_way.update({a: [a]})
```

```
    current_node = a
```

```
    nd_to_visit = [] #ближайшие смежные  
    deleted = [] #удаленные  
    nd_to_visit.append(a)
```

```
    sort_len = [] #очередь с приоритетом: вес вершины - номер вершины
```

```
    while(1):
```

```
        if current_node != b: # если не конечная вершина
```

```
            sort_len = [len_sh_way.get(nd) for nd in nd_to_visit] # делаем массив из весов оставшихся вершин  
            sort_len.sort() # сортируем его, чтобы найти с наименьшим весом
```

```
            for key in nd_to_visit: # ищем вершину с минимальным весом
```

```
                if len_sh_way.get(key) == sort_len[0]:  
                    current_node = key # запоминаем ее  
                    deleted.append(key) # добавляем в список удаленных  
                    break
```

```
            nd_to_vis = list(tuple(list_adj.get(current_node))) # получаем список вершин, с которыми смежна текущая
```

```

nd_to_vis = list(tuple(list_adj.get(current_node))) # получаем список вершин, с которыми смежна текущая

for nd in nd_to_vis:          #добавляем их в список вершин для посещения
    nd_to_visit.append(nd)
nd_to_visit = list(set(nd_to_visit)) # убираем повторения

for nd in deleted:           # новые вершины могут быть смежны с вершинами, которые уже удалили
    if nd_to_visit.count(nd) > 0:
        nd_to_visit.remove(nd)

for u in nd_to_visit:         #для всех остальных вершин
    weight = math.sqrt((nodes.get(u)[1] + nodes.get(current_node)[1])**2 + (nodes.get(u)[0] + nodes.get(current_node)[0])**2)

    if len_sh_way.get(u) > len_sh_way.get(current_node) + weight: # если от нынешней вершины до нее ближе

        w = len_sh_way.get(current_node) + weight
        len_sh_way.update({u: w}) # перезаписываем длину

        mas = list(tuple(shortest_way.get(current_node)))
        mas.append(u)
        m = list(tuple(mas))
        shortest_way.update({u: m}) # перезаписываем путь

    if len(nd_to_visit) == 0:
        return shortest_way.get(b)
        break

else:
    break

return (shortest_way.get(b))

```

Реализация алгоритма Левита



#Алгоритм Левита

```
from collections import deque
```

```
def Levita(a, b):
```

```
    len_sh_way = {nd: 1000000 for nd in nodes} # вершине соответствует длина пути до нее  
    len_sh_way.update({a: 0})
```

```
    shortest_way = {nd: [] for nd in nodes} #кратчайший путь от вершины a  
    shortest_way.update({a: [a]})
```

```
    nodes_already_calc = [] # M0  
    common_queue = deque((a,)) # M1'  
    rush_queue = deque() # M1''
```

```
    nodes_not_calc = [nd for nd in nodes] # M2  
    nodes_not_calc.remove(a)
```

```
    cur_node = a
```

```
    while(len(common_queue) > 0 or len(rush_queue) > 0):
```

```
        if len(rush_queue) > 0:  
            current_node = rush_queue.popleft()
```

```
        else:  
            cur_node = common_queue.popleft()
```

```
        adj = list_adj.get(cur_node)
```

```
        for nd in adj:
```

```
            w = math.sqrt((nodes.get(nd)[0] - nodes.get(cur_node)[0])**2 + (nodes.get(nd)[1] - nodes.get(cur_node)[1])**2)
```

```
            if nodes_not_calc.count(nd) > 0:  
                common_queue.append(nd)  
                nodes_not_calc.remove(nd)
```



```

for nd in adj:
    w = math.sqrt((nodes.get(nd)[0] - nodes.get(cur_node)[0])**2 + (nodes.get(nd)[1] - nodes.get(cur_node)[1])**2)

    if nodes_not_calc.count(nd) > 0:
        common_queue.append(nd)
        nodes_not_calc.remove(nd)

        if len_sh_way.get(nd) > len_sh_way.get(cur_node) + w:

            len_sh_way.update({nd: len_sh_way.get(cur_node) + w})
            way = shortest_way.get(cur_node)[: ]
            way.append(nd)
            shortest_way.update({nd: way})

    elif common_queue.count(nd) > 0 or rush_queue.count(nd) > 0:
        if len_sh_way.get(nd) > len_sh_way.get(cur_node) + w:

            len_sh_way.update({nd: len_sh_way.get(cur_node) + w})
            way = shortest_way.get(cur_node)[: ]
            way.append(nd)
            shortest_way.update({nd: way})

    elif nodes_already_calc.count(nd) > 0 and len_sh_way.get(nd) > len_sh_way.get(cur_node) + w:

        rush_queue.append(nd)
        nodes_already_calc.remove(nd)

        len_sh_way.update({nd: len_sh_way.get(cur_node) + w})
        way = shortest_way.get(cur_node)[: ]
        way.append(nd)
        shortest_way.update({nd: way})

nodes_already_calc.append(cur_node)

return(shortest_way.get(b))

```

Отрисовка маршрутов, полученных моими алгоритмами

```
#отрисовка в matplotlib для моих алгоритмов

def PlotResult(a, b, way):
    fig = plt.gcf()
    fig.set_size_inches(8, 12)    #установка большого размера полотна
    Lon_Lat = []

    #общая карта
    #     for key in list_adj:
    #         for nd in list_adj[key]:
    #             plt.plot([nodes[key][1], nodes[nd][1]], [nodes[key][0], nodes[nd][0]], 'blue')

    #граничные узлы
    plt.plot(nodes.get(a)[1], nodes.get(a)[0], 'ro')
    plt.plot(nodes.get(b)[1], nodes.get(b)[0], 'ro')

    #путь
    for nd in way:    #проход по узлам найденного пути
        Lon_Lat.append([nodes.get(nd)[1], nodes.get(nd)[0]])
    Lon_Lat = np.array(Lon_Lat)
    plt.plot(Lon_Lat[:, 0], Lon_Lat[:, 1], 'red')
    Lon_Lat = []

    plt.show()
```

Для более
красивой и
быстрой
отрисовки
воспользуемся
библиотекой
networkx

```
#граф в networkx
import networkx as nx

GG = nx.Graph()

# добавляем узлы в граф
for key in nodes:
    GG.add_node(str(key), pos=tuple(nodes.get(key)))

# добавляем ребра в граф
for nd in list_adj:
    for nod in list_adj.get(nd):
        w = math.sqrt((nodes.get(nd)[1] + nodes.get(nod)[1])**2 + (nodes.get(nd)[0] + nodes.get(nod)[0])**2)
        GG.add_edge(str(nd), str(nod), weight=w)

#словарь узлов для networkx
nodesi = {str(key): [nodes.get(key)[1], nodes.get(key)[0]] for key in nodes}

#отрисовка графа в networkx
def DrawGraph():
    nx.draw_networkx(GG, pos=nodesi, node_size = 0.1, width = 0.2, with_labels = False)
    plt.axis('on')
    plt.show()
```

#отрисовка в networkx кратчайшего пути

```
def DrawSaveShortestWay(GG, way, start, end):
    nx.draw_networkx(GG, pos=nodesi, node_size = 0, width = 0.2, with_labels = False, node_color='black', edge_color='black')
    h = GG.subgraph(way)
    nx.draw_networkx_nodes(h,pos=nodesi, node_color='red', node_size = 2)
    nx.draw_networkx_edges(h,pos=nodesi, edge_color='red', width = 2)
    fig = plt.gcf()
    fig.set_size_inches(8, 12)
    #plt.show()
    filename = "ShortestWay_" + str(start) + "_" + str(end) + ".png"
    plt.savefig(filename, dpi = 100)
    fig.clear()
```

#эвристические функции для A стар

import math

```
def ManhattanDist(start, finish):
    return abs(nodes.get(int(start))[0]-nodes.get(int(finish))[0])/2 + abs(nodes.get(int(start))[1]-nodes.get(int(finish))[1])

def ChebDist(start, finish):
    return max(abs(nodes.get(int(start))[0]-nodes.get(int(finish))[0]), abs(nodes.get(int(start))[1]-nodes.get(int(finish))[1])

def EuklidDist(start, finish):
    return math.sqrt((nodes.get(int(start))[0]-nodes.get(int(finish))[0])**2 + (nodes.get(int(start))[1]-nodes.get(int(finish))
```

Пример реализации

```
def timeAstarEukl(start, target):
    start_time = time.time()
    Astar_Eukl = nx.astar_path(GG, str(start), str(target), EuklidDist)
    return time.time() - start_time
    #print("--- %s seconds ---" % (time.time() - start_time))
    #drawShortestWay(GG, Astar_Eukl)
```

#запись получившихся маршрутов в файл

```
def WriteCsvFile(start, target, way):
    filename = "ShortestWay__" + str(start) + "__" + str(target) + ".csv"
    with open(filename, "a") as file:
        writer = csv.writer(file)
        writer.writerow(way)
```

#маршруты до больницы с записью в файл и сохранением картинок

```
for t in target:
    Dejcistra = nx.dijkstra_path(GG, str(start), str(t))
    DrawSaveShortestWay(GG, Dejcistra, str(start), str(t))
    WriteCsvFile(str(start), str(t), Dejcistra)
```


Полученные результаты





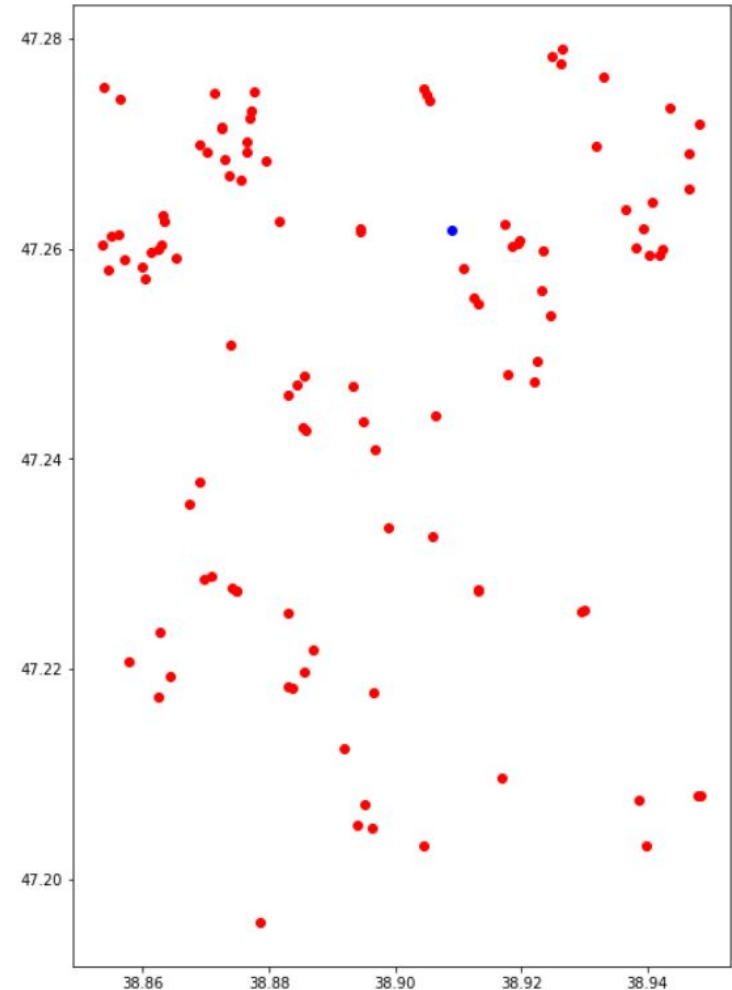


Время работы алгоритма для 100
случайных точек.

A dark blue diagonal gradient bar that starts from the bottom left corner and extends towards the top right corner, covering the lower half of the slide.

Необходимо замерить время работы всех алгоритмов на 100 случайно выбранных точках и записать в файл. Алгоритмы Дейкстры и А стар будут использованы из библиотеки. На рисунке изображены выбранные точки. Синей отмечена стартовая.

```
#выбираем 100 точек  
random = nodes.items()[1:1501:15]
```



```

#запись статистики для 100 точек
filename = "Statistics_100_points" + ".csv"

#шапочка файла
fieldnames = ['idNode', 'Dejicstra', 'Levit', 'AstarEuklid', 'AstarManh', 'AstarCheb']

#запись шапочки
with open(filename, "w") as file:
    writer = csv.DictWriter(file, delimiter=',', fieldnames=fieldnames)
    writer.writeheader()

def csv_writer(path, fieldnames, data):
    with open(path, "a") as out_file:
        writer = csv.DictWriter(out_file, delimiter=',', fieldnames=fieldnames)
        for row in data:
            writer.writerow(row)

for nd in random:
    cur_node = nd[0]
    line = []
    line_dict = []
    dejicstr = timeDejicstra(start, cur_node)
    aStEukl = timeAstarEukl(start, cur_node)
    aStManh = timeAstarManh(start, cur_node)
    aStCheb = timeAstarCheb(start, cur_node)

    start_time = time.time()
    way = Levita(start, cur_node)
    levit = time.time() - start_time

    #чтобы записалось с шапочкой нормально
    line = [cur_node, dejicstr, levit, aStEukl, aStManh, aStCheb]
    inner_dict = dict(zip(fieldnames, line))
    line_dict.append(inner_dict)
    csv_writer(filename, fieldnames, line_dict)

```

#записываем в файл среднее время проезда до 100 вершин

```
import csv
import math
```

#считываем с файла узлы

```
filename = "Statistics_100_points" + ".csv"
```

```
node100 = []
```

```
with open(filename, "r") as file:
    reader = csv.reader(file)
    for row in reader:
        node100.append(row[0])
```

```
filename = "TimeWay_100_points" + ".csv"
```

#шапочка файла

```
fieldnames = ['idNode', 'time']
```

#запись шапочки

```
with open(filename, "w") as file:
    writer = csv.DictWriter(file, delimiter=',', fieldnames=fieldnames)
    writer.writeheader()
```

```
def csv_writer(path, fieldnames, data):
```

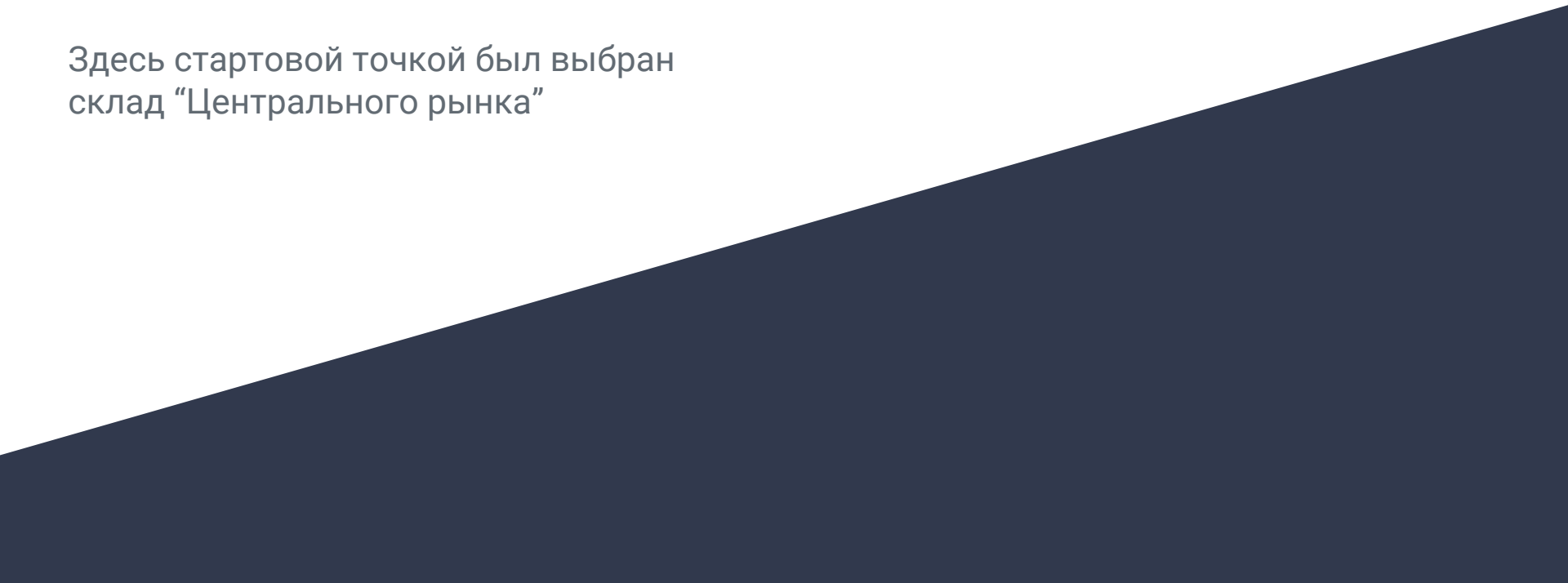
```
    with open(path, "a") as out_file:
        writer = csv.DictWriter(out_file, delimiter=',', fieldnames=fieldnames)
        for row in data:
            writer.writerow(row)
```


Вычисление времени подъезда к точкам

```
# для всех этих узлов получаем кратчайший путь, рассчитываем его длину и умножаем на километраж градусов по широте и долготе
for nd in node100[1:]:
    way = Levita(start, int(nd));
    len_way_km = 0
    for i in range(len(way)-1):
        nd_1 = nodes.get(way[i])
        nd_2 = nodes.get(way[i + 1])
        len_way_km = len_way_km + math.sqrt(((nd_1[0] - nd_2[0])**2)*76057 + ((nd_1[1] - nd_2[1])**2)*111000)
    time = len_way_km/40
    line = []
    line_dict = []
    line = [nd, time]
    inner_dict = dict(zip(fieldnames, line))
    line_dict.append(inner_dict)
    csv_writer(filename, fieldnames, line_dict)
print time
```

Решение задачи коммивояжера

Здесь стартовой точкой был выбран
склад “Центрального рынка”

A dark blue diagonal gradient bar that starts from the bottom left and extends towards the top right, covering the lower half of the slide.

Алгоритм поиска ближайшего соседа



```
#гамильтонов цикл минимального веса поиском ближайшего соседа
```

```
from heapq import heappush, heappop
```

```
hamilton = [start] #цикл
```

```
way_between = {nd:[] for nd in target}
```

```
way_to = [] #дороги до текущего элемента cur_way
```

```
to = [nd for nd in target]
```

```
cur_way = 0
```

```
len_cur_way = 0
```

```
for useless in range(len(hospitals)):
```

```
    for nd in to:
```

```
        cur_way = nx.dijkstra_path(GG, str(start), str(nd))
```

```
        len_cur_way = 0
```

```
        for i in range(len(cur_way)-1):
```

```
            nd_1 = nodes.get(int(cur_way[i]))
```

```
            nd_2 = nodes.get(int(cur_way[i + 1]))
```

```
            len_cur_way = len_cur_way + math.sqrt((nd_1[0] - nd_2[0])**2 + (nd_1[1] - nd_2[1])**2)
```

```
            heappush(way_to, (len_cur_way, nd, cur_way)) #добавляем длину кратчайшего пути для вершины nd от текущей start
```

```
next_node = heappop(way_to)[1] #забираем вершину с минимальным весом
```

```
to.remove(next_node) #удаляем ее из списка еще не посещенных
```

```
hamilton.append(next_node) #добавляем в цикл
```

```
start = next_node #теперь считать кратчайшие пути будем от нее
```

```
way_to = []
```

```
print hamilton
```

```

fig = plt.gcf()
fig.set_size_inches(8, 12)    #установка большого размера полотна
Lon_Lat = []

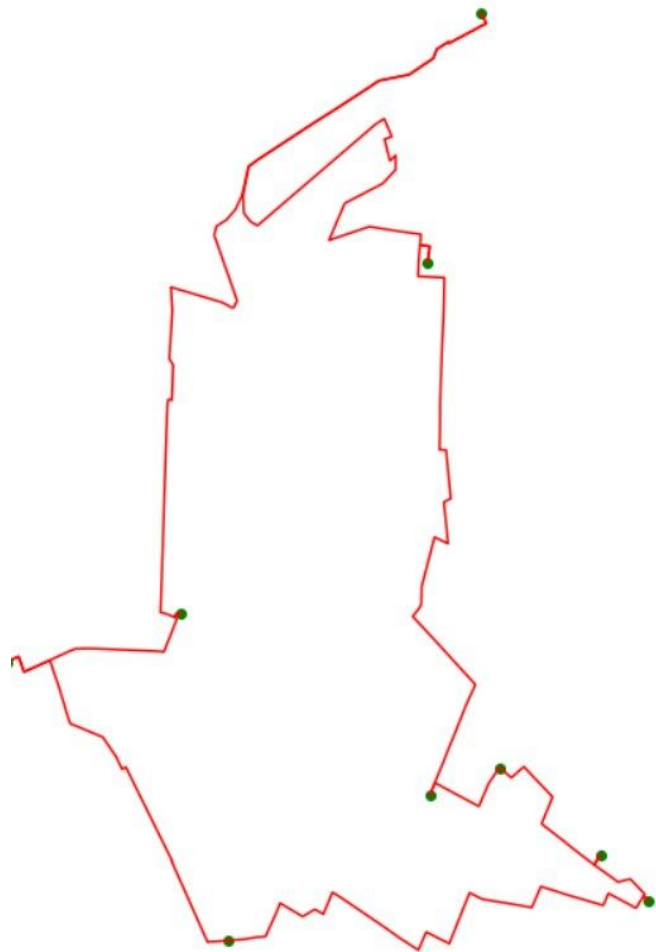
all_way = []
#нужно
for i in range(len(hamilton) - 1):
    all_way.append(nx.dijkstra_path(GG, str(hamilton[i]), str(hamilton[i+1])))
all_way.append(nx.dijkstra_path(GG, str(hamilton[0]), str(hamilton[-1])))

for nd in hamilton:
    plt.plot(nodes.get(nd)[1], nodes.get(nd)[0], 'go')

for way in all_way:
    for nd in way:
        #проход по узлам найденного пути
        Lon_Lat.append([nodes.get(int(nd))[1], nodes.get(int(nd))[0]])
    Lon_Lat = np.array(Lon_Lat)
    plt.plot(Lon_Lat[:, 0], Lon_Lat[:, 1], 'red')
    Lon_Lat = []

plt.show()
plt.savefig("hamilton_my.png", dpi = 100)

```



Алгоритм добавления вершины в
цикл с наименьшим
увеличением периметра

#решение задачи коммивояжера методом добавления вершины в цикл

#записываем все вершины для цикла

```
node_to = [nd for nd in target]
```

```
node_to.append(start)
```

```
hamilton = list(tuple(node_to))
```

```
size = len(node_to) #количество вершин в графе
```

```
print node_to
```

```
#
```

#вес ребра

```
def lenght_edge(i, j):
```

```
    l = math.sqrt((nodes.get(int(i))[0] - nodes.get(int(j))[0])**2 + (nodes.get(int(i))[1] - nodes.get(int(j))[1])**2)
```

```
    return l
```

#длина маршрута между двумя точками в исходном графе

```
def weight_way(way):
```

```
    w = 0
```

```
    for i in range(len(way) - 1):
```

```
        w = w + lenght_edge(way[i], way[i+1])
```

```
    return w
```

#словарь, по ребру выдает его вес

```
edges = {}
```

```
for i in node_to:
```

```
    for j in node_to:
```

```
        t = tuple([i,j])
```

```
        way = nx.dijkstra_path(GG, str(i), str(j)) #путь между этими вершинами в исходном графе
```

```
        edges[t] = weight_way(way) #его длина
```

```
min_weight = 1000000;
```

```
min_edge = () #ребро с минимальным весом, ищем для старта
```

```
for key in edges:
```

```
    if edges.get(key) < min_weight and edges.get(key) != 0 :
```

```
        min_weight = edges.get(key)
```

```
        min_edge = key
```

#чтобы было потом удобно искать по ребрам, устанавливаем порядок очередности вершин, от меньшей к большей

```
if min_edge[0] > min_edge[1]:  
    min_edge = ((min_edge[1],min_edge[0]))
```

#удаляем из списка необходимых к посещению вершин те, которые образуют кратчайшее ребро

```
node_to.remove(min_edge[0])  
node_to.remove(min_edge[1])
```

```
cicle_edge = []    #цикл, записанный ребрами
```

```
cicle_node = []    #вершины в цикле
```

```
cicle_edge.append(min_edge)
```

```
cicle_node.append(min_edge[0])
```

```
cicle_node.append(min_edge[1])
```

#вес цикла, на вход поступают имеющийся цикл, вершина, которую хотим добавить, ребро, которое хотим удалить

```
def lenght_way(cicle_ed, nd, ed):
```

```
    w = 0
```

```
    w = w + edges.get((ed[0], nd)) #добавляем ребро с новой вершиной из одного конца ребра
```

```
    w = w + edges.get((ed[1], nd)) #из другого
```

```
    for nod in cicle_ed:
```

```
        w = w + edges.get(nod)    #прибавляем просто веса всех ребер в этом цикле
```

```
    w = w - edges.get(ed)    #вычитаем вес того, вместо которого добавляли новые
```

```
    return w;
```

#из-за наличия стартового ребра, проходим циклом n-2 раза по точкам графа

```
for useless in range(size - 2):
```

```
    min_w_cicle = 1000000
```

```
    ed_to_remove = ()
```

```
    nd_to_add = 0
```

#из-за наличия стартового ребра, проходим циклом n-2 раза по точкам графа

```
for useless in range(size - 2):
    min_w_cicle = 1000000
    ed_to_remove = ()
    nd_to_add = 0
    #для вершин, которые еще не в цикле
    for nd in node_to:
        for ed in cicle_edge: #для ребер, которые уже в цикле сравниваем
            w = lenght_way(tuple(cicle_edge), nd, ed) #какое ребро будет выгоднее всего заменить двумя ребрами с новой вершиной
            if w < min_w_cicle:
                min_w_cicle = w
                ed_to_remove = ed
                nd_to_add = nd

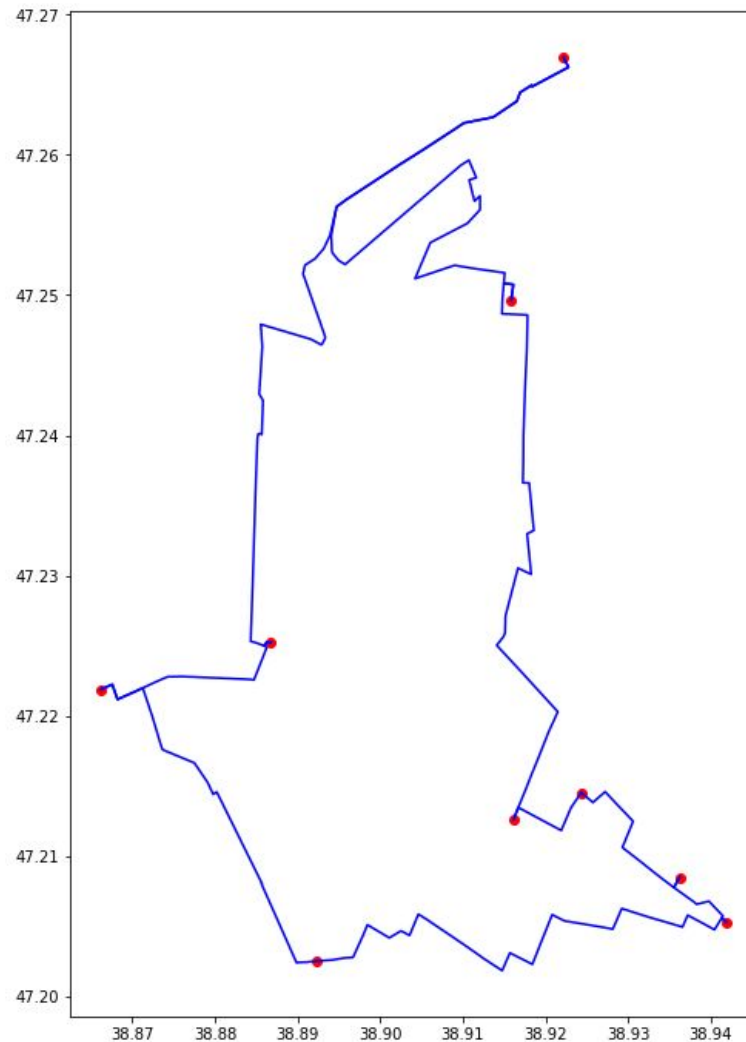
    #снова располагаем в порядке возрастания координаты ребра для удобства удаления
    if ed_to_remove[0] < nd_to_add:
        cicle_edge.append((ed_to_remove[0], nd_to_add))
    else:
        cicle_edge.append((nd_to_add, ed_to_remove[0]))

    if ed_to_remove[1] < nd_to_add:
        cicle_edge.append((ed_to_remove[1], nd_to_add))
    else:
        cicle_edge.append((nd_to_add, ed_to_remove[1]))

    #удаляем старое ребро
    if useless != 0:
        cicle_edge.remove(ed_to_remove)
    #добавляем в список посещенных вершин новую
    cicle_node.append(nd_to_add)
    #удаляем из списка непосещенных новую
    node_to.remove(nd_to_add)

print cicle_edge
```

```
[(1130256950, 3898358242), (211530735, 1130256950), (211530735, 1127283057), (3898358242, 3975857985), (1499967459, 380662493
7), (1499967459, 3975857985), (1127283057, 5422015747), (479475678, 3806624937), (479475678, 5422015747)]
```



Спасибо за
внимание!