Entrée [1]:

```
import osmnx as ox
import sys
sys.path.append("./../scripts")
import simplify_vertices as sv
import eulerian_path as ep
```

Parcours du drone

Simulation sur une petite zone d'un arrondissement

Entrée [2]:

```
G = [(2187161104, 301495983, 42.542), (2187161104, 2187161105, 37.878), (2187161104, 218716
l = sv.extract_vertices(G)
G2 = sv.replace_vertices(G, l)
print("Graphe initial:", G2)
```

```
Graphe initial: [(0, 1, 42.542), (0, 2, 37.878), (0, 3, 67.978), (0, 4, 101.
438), (1, 5, 66.387), (1, 6, 73.866), (1, 7, 40.899), (2, 8, 34.982), (2, 9,
121.617), (3, 5, 40.396), (3, 10, 69.239), (4, 9, 44.747), (4, 10, 73.995),
(4, 11, 38.42), (8, 12, 140.759), (8, 6, 41.433), (9, 12, 37.106), (12, 13,
37.605), (13, 11, 84.301), (13, 14, 122.592), (6, 15, 40.279), (5, 16, 74.54
3), (5, 17, 40.654), (10, 18, 38.812), (11, 18, 77.42), (11, 19, 15.593), (1
6, 20, 71.909), (16, 21, 72.924), (16, 22, 41.737), (20, 23, 32.518), (20, 2
4, 41.736), (21, 23, 85.13), (21, 25, 39.419), (22, 26, 41.084), (22, 27, 24
7.958), (22, 28, 36.514), (29, 30, 35.57), (29, 31, 34.301), (29, 32, 77.43
6), (30, 33, 38.236), (30, 34, 38.503), (30, 35, 119.103), (31, 36, 113.26
6), (31, 37, 37.578), (32, 38, 34.145), (32, 39, 37.001), (32, 40, 71.644),
(38, 37, 40.629), (38, 41, 5.584), (39, 42, 4.948), (39, 34, 38.693), (40, 2
7, 42.729), (40, 43, 39.307), (40, 44, 70.628), (27, 45, 37.378), (27, 46, 3
8.86), (43, 47, 248.819), (43, 48, 36.955), (43, 49, 36.575), (44, 50, 67.25
5), (44, 51, 38.899), (44, 52, 43.081), (50, 53, 39.867), (50, 54, 41.996),
(53, 7, 247.85699999999997), (53, 55, 35.006), (54, 56, 34.856), (51, 56, 3
2.723), (51, 57, 249.076), (51, 49, 33.25), (52, 46, 31.656), (52, 55, 32.53
7), (52, 17, 248.53), (17, 26, 32.431), (17, 58, 33.797), (15, 7, 74.343),
(59, 36, 180.975), (59, 60, 84.696), (61, 62, 228.54), (61, 19, 10.936), (6
2, 63, 7.414), (62, 64, 15.746), (18, 25, 79.669), (36, 65, 39.572), (60, 6
6, 70.446), (60, 67, 40.821), (66, 68, 64.125), (66, 47, 40.295), (67, 69, 3
5.021), (67, 65, 181.114), (67, 70, 136.215), (68, 57, 40.334), (47, 69, 35.
578), (47, 71, 34.047), (57, 71, 30.881), (57, 72, 35.082), (23, 73, 4.733),
(23, 64, 41.681000000000004), (25, 64, 83.134), (74, 24, 127.62), (74, 42, 1
21.265), (74, 75, 37.449), (24, 28, 35.579), (24, 73, 76.72800000000001), (4
2, 45, 34.32), (75, 34, 126.342), (75, 76, 184.89), (75, 77, 38.902), (33, 7
8, 118.978), (73, 79, 41.67200000000001), (73, 76, 33.845), (64, 79, 7.531),
(78, 35, 38.422), (63, 79, 15.667), (79, 80, 75.58300000000001), (80, 81, 4
1.367000000000004), (7, 58, 33.131), (81, 76, 43.505), (81, 77, 166.316), (7
7, 35, 7.358), (14, 82, 75.47500000000001), (26, 46, 248.257), (58, 55, 248.
182), (28, 45, 248.475), (69, 48, 248.402), (48, 41, 34.534), (65, 37, 115.2
59), (65, 70, 41.31), (56, 72, 249.049), (41, 70, 111.751), (71, 49, 248.95
5)]
```

Entrée [3]:

```
path = ep.convert_and_find_eulerian_path(len(l), G2)
print("Chemin avec noeuds simplifiés:", path, '\n')
path = sv.replace_vertices_back_in_path(path, l)
print("Chemin final:", path)
```

```
Chemin avec noeuds simplifiés: [82, 14, 13, 18, 11, 13, 12, 9, 4, 10, 3, 10,
18, 25, 21, 25, 64, 79, 63, 62, 81, 76, 73, 24, 28, 26, 46, 55, 58, 6, 15,
7, 58, 17, 26, 22, 28, 45, 42, 45, 27, 40, 43, 48, 41, 70, 65, 67, 70, 36, 6
5, 37, 38, 41, 48, 69, 71, 49, 56, 72, 57, 71, 47, 69, 67, 60, 66, 47, 43, 4
9, 51, 56, 54, 50, 53, 55, 52, 17, 5, 3, 0, 4, 11, 19, 61, 62, 64, 23, 73, 7
9, 80, 81, 77, 35, 77, 75, 34, 39, 42, 74, 75, 76, 20, 23, 21, 16, 22, 27, 4
6, 52, 44, 51, 57, 68, 66, 60, 59, 36, 31, 29, 31, 37, 38, 32, 39, 34, 30, 3
5, 78, 33, 30, 29, 32, 40, 44, 50, 53, 7, 1, 6, 8, 2, 9, 12, 8, 2, 0, 1, 5,
16, 20, 24, 74]

Chemin final: [615001851, 615001836, 615001834, 615001826, 615001828, 615001
834, 2187161109, 2187161106, 2187161118, 371474024, 2187161117, 371474024, 6
15001826, 615001807, 2188531350, 615001807, 615001817, 9256960177, 925696017
6, 615001811, 1764326134, 2188531392, 9306783005, 2187162887, 2187162918, 21
87162880, 2187162898, 2187162912, 2187162885, 615017546, 2187162922, 2187162
903, 2187162885, 2187162910, 2187162880, 2187162915, 2187162918, 2187162890,
2187162920, 2187162890, 2187162900, 615017537, 2189049760, 2189049777, 21890
49740, 2189049768, 2189049764, 2189049771, 2189049768, 2189049736, 218904976
4, 2189049737, 2189049755, 2189049740, 2189049777, 2189049732, 2189049742, 2
189049765, 2189049754, 2189049761, 2189049763, 2189049742, 2189049759, 21890
49732, 2189049771, 615017585, 615017586, 2189049759, 2189049760, 2189049765,
2189049757, 2189049754, 2189049739, 615017538, 2187162893, 2187162912, 21871
62905, 2187162910, 615017544, 2187161117, 2187161104, 2187161118, 615001828,
615001830, 2140361825, 615001811, 615001817, 2188531370, 9306783005, 9256960
177, 615001803, 1764326134, 2188531394, 2188531396, 2188531394, 2188531384,
2188531381, 2188531375, 2187162920, 2188531362, 2188531384, 2188531392, 2485
11702, 2188531370, 2188531350, 615017532, 2187162915, 2187162900, 218716289
8, 2187162905, 615017543, 2189049757, 2189049763, 615017587, 615017586, 6150
17585, 615017584, 2189049736, 2189049774, 615017534, 2189049774, 2189049737,
2189049755, 615017535, 2188531375, 2188531381, 2188531388, 2188531396, 21885
31386, 2188531373, 2188531388, 615017534, 615017535, 615017537, 615017543, 6
15017538, 2187162893, 2187162903, 301495983, 615017546, 2187161113, 21871611
05, 2187161106, 2187161109, 2187161113, 2187161105, 2187161104, 301495983, 6
15017544, 615017532, 248511702, 2187162887, 2188531362]
```

On peut voir que sur un graphe plus grand, représentant déjà une partie de la ville, nos algorithmes fonctionnent toujours.

On va donc pouvoir passer à des graphes encore plus grands cette fois.

Application sur les 19 arrondissements

Entrée [4]:

```
G_1 = ox.load_graphml('../data/graph.graphml')
G_2 = ox.load_graphml('../data/graph2.graphml')
G_3 = ox.load_graphml('../data/graph3.graphml')
G_4 = ox.load_graphml('../data/graph4.graphml')
G_5 = ox.load_graphml('../data/graph5.graphml')
G_6 = ox.load_graphml('../data/graph6.graphml')
G_7 = ox.load_graphml('../data/graph7.graphml')
G_8 = ox.load_graphml('../data/graph8.graphml')
G_9 = ox.load_graphml('../data/graph9.graphml')
G_10 = ox.load_graphml('../data/graph10.graphml')
G_11 = ox.load_graphml('../data/graph11.graphml')
G_12 = ox.load_graphml('../data/graph12.graphml')
G_13 = ox.load_graphml('../data/graph13.graphml')
G_14 = ox.load_graphml('../data/graph14.graphml')
G_15 = ox.load_graphml('../data/graph15.graphml')
G_16 = ox.load_graphml('../data/graph16.graphml')
G_17 = ox.load_graphml('../data/graph17.graphml')
G_18 = ox.load_graphml('../data/graph18.graphml')
G_19 = ox.load_graphml('../data/graph19.graphml')

L = [G_1, G_2, G_3, G_4, G_5, G_6, G_7, G_8, G_9, G_10, G_11, G_12, G_13, G_14, G_15, G_16,
```

Entrée [5]:

```
# convert given graph to our format
def decomplexify_tograph(graph):
    node_list = graph.edges(data=True)
    to_graph = []
    for node in node_list:
        dist = node[2]["length"]
        n1 = node[0]
        n2 = node[1]
        to_graph.append((n1, n2, dist))
    return to_graph
```

Entrée [6]:

```
# convert every graph in L
for i in range(len(L)):
    L[i] = decomplexify_tograph(L[i])

# get the list of the total length of each district
initial_dist = []
for g in L:
    d = 0
    for _,_,w in g:
        d += w
    initial_dist.append(d)
```

Entrée [7]:

```
paths = []

# process our algo on every graph and store the result in paths
for i in range(len(L)):
    print("Processing on borough n°%d of Montréal" % (i + 1))
    vertices = sv.extract_vertices(L[i])
    n = len(vertices)
    graph = sv.replace_vertices(L[i], vertices)
    graph = ep.to_eulerian(n, graph)
    L[i] = sv.replace_vertices_back(graph, vertices)
    path = ep.find_eulerian_path(n, graph)
    final_path = sv.replace_vertices_back_in_path(path, vertices)
    paths.append(final_path)
    print("Path found!", '\n')
```

```
Processing on borough n°1 of Montréal
Path found!

Processing on borough n°2 of Montréal
Path found!

Processing on borough n°3 of Montréal
Path found!

Processing on borough n°4 of Montréal
Path found!

Processing on borough n°5 of Montréal
Path found!

Processing on borough n°6 of Montréal
Path found!

Processing on borough n°7 of Montréal
Path found!

Processing on borough n°8 of Montréal
Path found!

Processing on borough n°9 of Montréal
Path found!

Processing on borough n°10 of Montréal
Path found!

Processing on borough n°11 of Montréal
Path found!

Processing on borough n°12 of Montréal
Path found!

Processing on borough n°13 of Montréal
Path found!

Processing on borough n°14 of Montréal
Path found!
```

```
Processing on borough n°15 of Montréal
Path found!

Processing on borough n°16 of Montréal
Path found!

Processing on borough n°17 of Montréal
Path found!

Processing on borough n°18 of Montréal
Path found!

Processing on borough n°19 of Montréal
Path found!
```

On va maintenant afficher par exemple le chemin du 1er arrondissement pour vérifier que celui-ci recouvre bien toutes les rues.

On peut voir sur l'image ci-dessous que le 1er chemin recouvre bien l'entièreté du 1er arrondissement de la ville et on a donc la confirmation que notre algorithme fonctionne toujours sur des graphes plus complexes.

Entrée [8]:

```
# get the list of the total length of each district
final_dist = []
for g in L:
    d = 0
    for _,_,w in g:
        d += w
    final_dist.append(d)

montreal_dist = 0
total_dist = 0
for i in range(len(final_dist)):
    montreal_dist += initial_dist[i]
    total_dist += final_dist[i]
    print("Difference of distance in borough n°%d:" % (i + 1), final_dist[i] - initial_dist
print("Initial total distance of montreal streets:", montreal_dist)
print("Total distance traveled by the drone inside all boroughs:", total_dist)
```

```
Difference of distance in borough n°1: 0.0
Difference of distance in borough n°2: 0.0
Difference of distance in borough n°3: 0.0
Difference of distance in borough n°4: 0.0
Difference of distance in borough n°5: 0.0
Difference of distance in borough n°6: 0.0
Difference of distance in borough n°7: 0.0
Difference of distance in borough n°8: 0.0
Difference of distance in borough n°9: 0.0
Difference of distance in borough n°10: 0.0
Difference of distance in borough n°11: 0.0
Difference of distance in borough n°12: 9640.023000000074
Difference of distance in borough n°13: 0.0
Difference of distance in borough n°14: 0.0
Difference of distance in borough n°15: 0.0
Difference of distance in borough n°16: 0.0
Difference of distance in borough n°17: 0.0
Difference of distance in borough n°18: 0.0
Difference of distance in borough n°19: 0.0
Initial total distance of montreal streets: 12960320.593000034
Total distance traveled by the drone inside all boroughs: 12969960.616000034
```

Trouver le chemin le plus court pour passer d'un arrondissment à l'autre

Le but va être maintenant d'optimiser le chemin du drone entre les arrondissements.

Pour cela il va d'abord falloir afficher tous les noeuds d'entrée et de sortie des différents chemins.

Entrée [9]:

```
# return the list of tuple representing the starting and ending node of each path
def get_start_and_end(paths):
    res = []
    for p in paths:
        res.append((p[0], p[len(p)-1]))
    return res
```

Entrée [10]:

```
start_end = get_start_and_end(paths)
print(start_end)
```

```
[(31703078, 31703078), (32662002, 32662002), (224886238, 224886238), (224806
465, 224806465), (109828183, 109828183), (31278805, 31278805), (109828183, 1
09828183), (29239079, 29239079), (26232893, 26232893), (116415771, 11641577
1), (96049028, 96049028), (1700518214, 1700534185), (109828183, 109828183),
(224911407, 224911407), (215622916, 215622916), (29217072, 29217072), (29287
991, 29287991), (26232481, 26232481), (31630405, 31630405)]
```

On peut noter que dans les sous graphes des arrondissements on obtient à chaque fois des cycles eulerien (en utilisant toujours le même algo) et donc le noeud d'entrée est toujours le noeud de sortie.

Entrée [11]:

```
start = [s for s,_ in start_end]
```

Entrée [12]:

```
print(start)
```

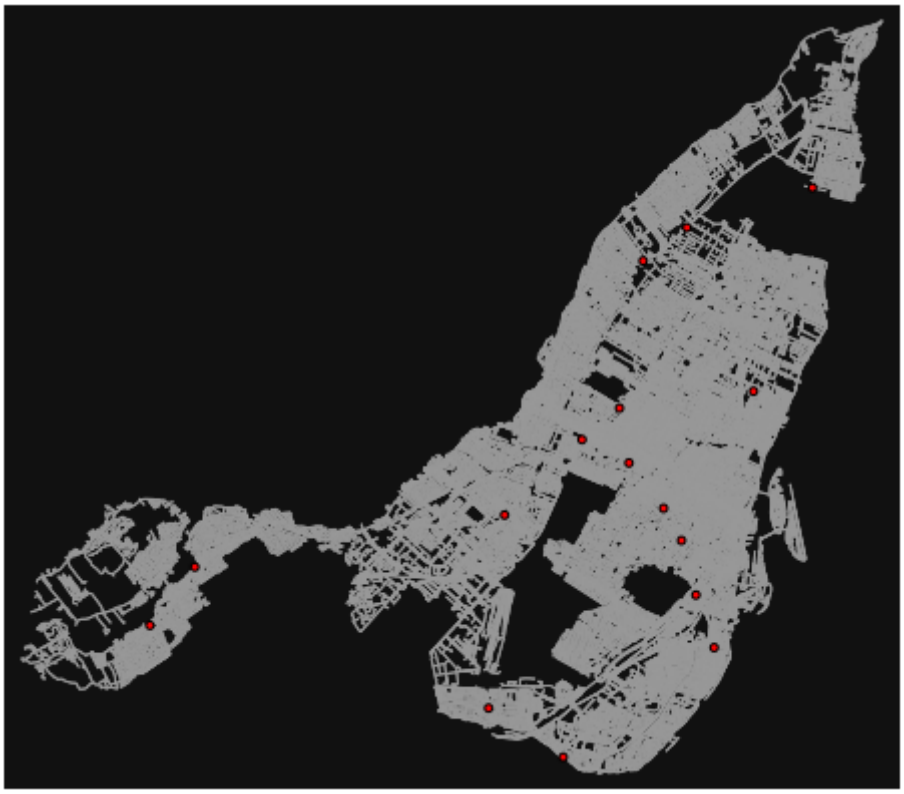
```
[31703078, 32662002, 224886238, 224806465, 109828183, 31278805, 109828183, 2
9239079, 26232893, 116415771, 96049028, 1700518214, 109828183, 224911407, 21
5622916, 29217072, 29287991, 26232481, 31630405]
```

Entrée [13]:

```
# get the graph of Montreal
Montreal = ox.graph_from_place("Montreal, Canada", network_type="walk")
```

Entrée [14]:

```
# show each vertex on the map
ef = ['r' if u in start else 'w' for u in Montreal.nodes()]
size =  [15 if u in start else 0 for u in Montreal.nodes()]
fig, ax = ox.plot_graph(Montreal, node_color=ef, node_edgecolor='k', node_size=size)
```



Entrée [17]:

```
# list containing latitude and longitude of each point from the map above
pos = [(45.485313, -73.865457), (45.495893, -73.852471), (45.510230, -73.688313), (45.44207

def get_graph_from_pos(pos):
    g = []
    for i in range(len(pos)):
        for j in range(len(pos)):
            if i != j:
                g.append((i, j, ox.distance.great_circle_vec(pos[i][0], pos[i][1], pos[j][0
    return g

# create a graph where all nodes are connected to each other with weight representing the d
ga = get_graph_from_pos(pos)

def find_edge(edges, v):
    for (a,b,c) in edges:
        if a == v or b == v:
            return (a,b,c)
    return None

def getNexts(G, v1):
    res = []
    for i in range (len(G)):
        if (G[i][0] == v1 or G[i][1] == v1):
            res.append(G[i])
    return res

def pathSub(G, v1, v2):
    min = []
    nexts = getNexts(G, v1)
    k = 0
    while (True):
        for i in range (len(nexts)):
            if (nexts[i][0] == v2 or nexts[i][1] == v2):
                min.append(v2)
                return min
        for i in range (len(nexts)):
            tmp = pathSub(G, nexts[i][0], v2)
            for i in range(len(tmp)):
                min.append(tmp[i])
        k += 1
        if (k > 10000):
            break
    return min

# function to find the shortest path between the first and the last node of a graph visitin
def get_shortest_path(n, G, v1, v2):
    copy = []
    for i in range(len(G)):
        copy.append(G[i])
    stack = [G[0][0]]
    path = [v1]
    while stack:
        v = stack[-1]
        edge = find_edge(G, v)
        if edge:
            u = edge[0]
            if u == v:
                u = edge[1]
```

```
            stack.append(u)
            G.remove(edge)
        else:
            path.append(stack.pop())
    res = []
    sub = pathSub(copy, v1, path[0])
    end = pathSub(copy, path[len(path) - 1], v2)
    for i in range (len(sub)):
        res.append(sub[i])
    for i in range (len(path)):
        res.append(path[i])
    for i in range(len(end)):
        res.append(end[i])
    if (res[0] == res[1]):
        res.pop(0)
    return res

shortest = get_shortest_path(len(pos), ga, 0, len(pos)-1)
```

On peut donc voir sur l'image ci-dessus les différents points par lesquels le drone devra passer pour se déplacer d'un arrondissement à l'autre.

Le chemin le plus optimisé pour le drone entre chaque arrondissment est donc le suivant: