DS and Graph Compression

May 10, 2024

1 Path Visualizer

1.1 What is this?

This notebook accomplishes the following: 1) **Downloading** and **formatting** data in a form usable by the **Parallang Simulator**. 2) **Visualization** of the true shortest route between **any two points** on the graph. 3) **Reading** and **visualization** of the output of the **Parallang Simulator** for comparison.

1.2 How do I start?

Begin by **installing and importing the below modules**. Then, read the instructions in each section carefully to find what needs to be changed.

[ ]:
```
!pip3 install matplotlib
!pip3 install osmnx
```

[ ]:
```
import matplotlib.pyplot as plt
import networkx as nx
import osmnx as ox
import os
import pprint
import random

from collections import defaultdict
```

1.3 Section 1: Downloading of Data

Set the below variables and run the rest of the blocks in this section before continuing.

[ ]:
```
# CONSTANTS: Useful if you need something specific.
SOUTHERN_ENGLAND = ['Bristol, UK', 'Cornwall, UK', 'Devon, UK', 'Dorset, UK',␣
 ↪'Gloucestershire, UK', 'Somerset, UK', 'Wiltshire, UK', 'Berkshire, UK',␣
 ↪'Buckinghamshire, UK', 'West Sussex, UK', 'East Sussex, UK', 'Kent, UK',␣
 ↪'Oxfordshire, UK', 'Hampshire, UK', 'Isle of Wight, UK', 'Surrey, UK', 'City␣
 ↪of London, UK', 'Greater London, UK', 'Bedfordshire, UK', 'Cambridgeshire,␣
 ↪UK', 'Hertfordshire, UK', 'Essex, UK', 'Norfolk, UK', 'Suffolk, UK']
```

```
# CHANGE ME: Location to get data from.

place_name = "Stroud, UK"

# Optional (not recommended to change): Change me if you want a different type␣
↪of graph. Defaultly just paths you can drive.
mode = None

# Optional: It's a whitelist filter that filters out unwanted edges. For large␣
↪queries, consider implementing a motorway filter
# All others: custom_filter =␣
↪'["highway"~"primary|primary_link|secondary|secondary_link|tertiary|tertiary_link"]'
# Southern England: '["highway"~"motorway"]'
# Sizes (post-compression):
#    * Southern England (1351) <- 29795 (originally)
#    * Gloucestershire (991)   <- 17941
#    * Bristol (762)           <-  8479
#    * Southampton (455)       <-  7188
#    * Cheltenham (267)        <-  3941
#    * Stroud (110)            <-  3570
#    * Bishop's Cleeve (50)    <-   489
custom_filter =␣
↪'["highway"~"primary|primary_link|secondary|secondary_link|tertiary|tertiary_link"]'

# Optional: Change me if you want a directed graph
directed = False

# Optional: Whether to apply graph compression
apply_graph_compression = True & (not directed)

# Optional: Whether to truncate
truncate_to = 400000

# Optional: Number of datasets to generate
num_datasets = 1

# CHANGE ME: Location to store edge list in.
filename = f"analyzed_datasets/{place_name}"

# Optional: Whether to output verbosely.
ox.settings.log_console = True
```

[ ]:
```
# must set simplify to false, otherwise the simplification ratios of this␣
↪technique will be terrible!
graph = ox.graph_from_place(place_name, network_type=mode, simplify=not␣
↪apply_graph_compression, custom_filter=custom_filter)
```

```
if not directed:
    graph = ox.utils_graph.get_undirected(graph)

nodes, edges = ox.graph_to_gdfs(graph)

print("Graph downloaded.")
```

[ ]:
```
translations = {}
nextid = 0

shortest_direct_distances = {}

# length is in meters
lengths = edges[["osmid", "length"]]

print("Formatting graph...")
for u, u_keyed_df in lengths.groupby(level=0):
    for v, uv_keyed_df in u_keyed_df.groupby(level=1):
        if u not in translations:
            translations[u] = nextid
            nextid += 1
        if v not in translations:
            translations[v] = nextid
            nextid += 1

        for (i, length) in enumerate(uv_keyed_df["length"]):
            shortest_direct_distances[(translations[u], translations[v])] =␣
↪min(int(length*1000), shortest_direct_distances.get((u, v), float("inf")))
            if not directed:
                shortest_direct_distances[(translations[v], translations[u])] =␣
↪shortest_direct_distances[(translations[u], translations[v])]
inverted_translations = {v: k for (k, v) in translations.items()}

two_valency_nodes = set()
if apply_graph_compression:
    print("Beginning compression...")
    edge_list_dict = defaultdict(set)
    for ((e1, e2), d) in shortest_direct_distances.items():
        if e1 == e2: continue
        edge_list_dict[e1].add(e2)
        edge_list_dict[e2].add(e1)
    edge_list_dict = {k: list(v) for (k, v) in edge_list_dict.items()}

    # first, find the two valency nodes
    for n in edge_list_dict:
        if len(edge_list_dict[n]) == 2:
            two_valency_nodes.add(n)
```

```python
    print(f"Found {len(two_valency_nodes)} two valency nodes, finding paths...")
    collapsed_paths = []
    i = 0
    visited = set()
    for n in two_valency_nodes:
        if n in visited:
            continue

        left_search = []
        right_search = []

        left_search = [n, edge_list_dict[n][0]]
        right_search = [edge_list_dict[n][1]]

        visited.add(n)

        for search in [left_search, right_search]:
            two_in_search = any(2 in s for s in [left_search, right_search])
            if not search:
                continue
            while search[-1] in two_valency_nodes and any(k not in visited for k in edge_list_dict[search[-1]]):
                visited.add(search[-1])
                k = edge_list_dict[search[-1]][0]
                if k in visited:
                    k = edge_list_dict[search[-1]][1]

                search.append(k)
        collapsed_paths.append((list(reversed(right_search)) + left_search)[1:-1])

    print(f"Collapsed to {len(collapsed_paths)} paths, compressing them...")
    better_egress = {}
    path_length = {}
    node_to_path_id = {}
    collapsed_paths_dict = {}
    egresses_to_path_id = defaultdict(list)
    path_id_to_egresses = {}
    path_id = 0
    for p in collapsed_paths:
        # # don't resolve stand-alone cycles
        # if edge_list_dict[p[-1]] in edge_list_dict[p[0]]:
        #     continue

        collapsed_paths_dict[path_id] = p
        k = edge_list_dict[p[0]][0]
```

```
        if k in visited:
            k = edge_list_dict[p[0]][1]
        j = edge_list_dict[p[-1]][0]
        # some paths will be one node.
        # in such a scenario, you need to
        # ensure differing endpoints are used
        # as p[-1] and p[0] will be the same
        # and hence the len(p) == 1 check
        if j in visited or len(p) == 1:
            j = edge_list_dict[p[-1]][1]

        egresses_to_path_id[(k, j)].append(path_id)
        path_id_to_egresses[path_id] = (k, j)

        for n in p:
            node_to_path_id[n] = path_id

        length = sum(shortest_direct_distances[(a, b)] for (a, b) in zip(p[:-1], p[1:]))
        right_escape = length + shortest_direct_distances[(p[-1], j)]
        left_escape = shortest_direct_distances[(k, p[0])]
        for (a, b) in zip(p[:-1], p[1:]):
            if left_escape < right_escape:
                better_egress[a] = (k, left_escape)
            else:
                better_egress[a] = (j, right_escape)
            left_escape += shortest_direct_distances[(a, b)]
            right_escape -= shortest_direct_distances[(a, b)]
        if left_escape < right_escape:
            better_egress[p[-1]] = (k, left_escape)
        else:
            better_egress[p[-1]] = (j, right_escape)
        shortest_direct_distances[(k, j)] = shortest_direct_distances[(j, k)] = min(shortest_direct_distances.get((k, j), float("inf")), right_escape + left_escape)
        path_length[path_id] = right_escape + left_escape
        path_id += 1
    print(f"Shuffling down vertex numbers to account for removed nodes...")
    compressed_shortest_direct_distances = {}
    compressed_translations = {}
    inverted_compressed_translations = {}
    node_id = 0
    for ((e1, e2), d) in shortest_direct_distances.items():
        if e1 in visited or e2 in visited:
            continue
        for e in (e1, e2):
            if e not in inverted_compressed_translations:
```

```
                compressed_translations[node_id] = e
                inverted_compressed_translations[e] = node_id
                node_id += 1
        e1_new = inverted_compressed_translations[e1]
        e2_new = inverted_compressed_translations[e2]
        compressed_shortest_direct_distances[(e1_new, e2_new)] = d
        compressed_shortest_direct_distances[(e2_new, e1_new)] = d

    # de-default-ify
    egresses_to_path_id = {k : v for (k, v) in egresses_to_path_id.items()}

dist_to_use = shortest_direct_distances if not apply_graph_compression else compressed_shortest_direct_distances

max_node_num = max(max(shortest_direct_distances, key=lambda t: max(t)))
max_node_num_compressed = max_node_num
if apply_graph_compression:
    max_node_num_compressed = max(max(compressed_shortest_direct_distances, key=lambda t: max(t)))

for i in range(num_datasets):
    old_dist_to_use = dist_to_use
    if max_node_num_compressed > truncate_to:
        print(f"Going to truncate graph to {truncate_to}, currently {max_node_num_compressed+1} nodes")
        all_edges = (list(dist_to_use.items()) + list(((y, x), d) for ((x, y), d) in dist_to_use.items()))
        random.shuffle(all_edges)
        edge = random.choice(all_edges)
        nodes = set((edge[0][0], edge[0][1]))
        while len(nodes) < truncate_to:
            for ((v1, v2), dist) in all_edges:
                if v1 in nodes:
                    nodes.add(v2)
                if len(nodes) >= truncate_to:
                    break
        mappings = {v: i for (i, v) in enumerate(nodes)}
        dist_to_use = {(mappings[v1], mappings[v2]) : d for ((v1, v2), d) in dist_to_use.items() if v1 in nodes and v2 in nodes}
        print("Dropped", len(all_edges) - len(dist_to_use), "edges")

    filename = f"analyzed_datasets/{place_name}Trunc{truncate_to}_{i}.points"
    with open(filename, "w+") as F:
        drops = 0
        for i, (k, v) in enumerate(dist_to_use.items()):
            F.write(f"{i} {k[0]} {k[1]} {v}\n")
    dist_to_use = old_dist_to_use
```

```python
print(f"The graph has been saved to {os.path.join(os.getcwd(), filename)}")
if apply_graph_compression:
    # max_node_num_compressed = max(max(compressed_shortest_direct_distances, key=lambda t: max(t)))
    # max_node_num = max(max(shortest_direct_distances, key=lambda t: max(t)))
    node_compression_factor = round((max_node_num_compressed+1)/(max_node_num+1) * 100, 2)
    edge_compression_factor = round(len(compressed_shortest_direct_distances)/len(shortest_direct_distances)*100, 2)
    print(f"Graph edge compression factor: {edge_compression_factor}%")
    print(f"Graph node compression factor: {node_compression_factor}%")
    print(f"Final graph size: {min(max_node_num_compressed, truncate_to)}. Originally: {max_node_num}")
```

[ ]:

```python
fig, ax = ox.plot_graph(graph)
print(f"Visualization of {place_name}")
```

1.4 Section 2: Loading Simulator Output

Before progressing to this section, **run the simulator** on the output `.points` file. You can use the `EdgesReader` utility to load it into the system, and the `MatrixDumper` utility to dump the final results.

[ ]:

```python
# CHANGE ME: Path to successor/predecessor matrix.
path_to_distance_matrix = "distances.txt"

# CHANGE ME: Choose either the successor method or predecessor method depending on the underlying algorithm
SUCCESSOR = "successor"
PREDECESSOR = "predecessor"
method = PREDECESSOR

path_to_matrix = f"{method}.txt"
```

[ ]:

```python
def parseMatrix(filepath: str, local_id_to_global_id=None, do_optimization=True):
    with open(filepath, "r") as F:
        lines = F.readlines()
    lines = map(lambda a: map(int, a), map(lambda s: s.split(" "), lines))
    output = []
    for i, l in enumerate(lines):
        if do_optimization and i not in local_id_to_global_id:
            continue
        translated_line = []
        for e in l:
            if do_optimization and e not in local_id_to_global_id:
                continue
```

```
            translated_line.append(e)
        output.append(translated_line)

    return output

def safe_localize(node_l):
    """
    safely resolves a global ID to a local one with graph compression
    """
    if apply_graph_compression:
        node_l = inverted_compressed_translations[node_l]
    return node_l

def path_from_successor(successor, start_l, fin_l):
    start_l = safe_localize(start_l)
    fin_l = safe_localize(fin_l)

    output = [start_l]
    prev_l = fin_l
    while successor[output[-1]][fin_l] != fin_l:
        if successor[output[-1]][fin_l] == output[-1]:
            return []
        output.append(successor[output[-1]][fin_l])
    output.append(fin_l)

    return list(map(lambda e: compressed_translations[e],  output))
    # return list(map(lambda e: l2g_ids[e], output))

def path_from_predecessor(predecessor, start_l, fin_l):
    start_l = safe_localize(start_l)
    fin_l = safe_localize(fin_l)

    output = [fin_l]
    prev_l = fin_l
    while predecessor[start_l][prev_l] != start_l:
        if prev_l == predecessor[start_l][prev_l]:
            return []
        output.append(predecessor[start_l][prev_l])
        prev_l = predecessor[start_l][prev_l]

    output.append(start_l)
    output.reverse()
    return list(map(lambda e: compressed_translations[e],  output))
    # return list(map(lambda e: l2g_ids[e], output))

def calculatePath(matrix, distance_matrix, start_g, fin_g, g2l_ids, l2g_ids):
    #print("Entry.")
```

```
    if start_g == fin_g:
        return [start_g]
    strategy = path_from_successor if method == SUCCESSOR else␣
↪path_from_predecessor
    start_l = g2l_ids[start_g]
    fin_l = g2l_ids[fin_g]
    start_c = None
    fin_c = None
    if start_l in inverted_compressed_translations:
        start_c = inverted_compressed_translations[start_l]
    if fin_l in inverted_compressed_translations:
        fin_c = inverted_compressed_translations[fin_l]

    if not apply_graph_compression:
        output = strategy(matrix, start_l, fin_l)
        return list(map(lambda e: l2g_ids[e], output))

    def ddash_within_path(a, b, pid):
        (a_egg, a_dist) = better_egress[a]
        (b_egg, b_dist) = better_egress[b]
        if a_egg == b_egg:
            return abs(a_dist - b_dist)
        return abs((path_length[pid] - a_dist) - b_dist)

    def ddash(a, egg, pid):
        if a not in better_egress:
            a, egg = egg, a
        (a_egg, a_dist) = better_egress[a]
        if a_egg == egg:
            return a_dist
        return abs((path_length[pid] - a_dist))


    def dc(a, b):
        return distance_matrix[a][b]

    # case 1: neither vertex removed
    if start_l in inverted_compressed_translations and fin_l in␣
↪inverted_compressed_translations:
        #print("Case 1.")
        path = strategy(matrix, start_l, fin_l)
        output = []
        for (v1, v2) in zip(path[:-1], path[1:]):
            output.append(v1)
            if (v1, v2) in egresses_to_path_id or (v2, v1) in␣
↪egresses_to_path_id:
                f = lambda l: l
```

```
                pathlist = []
                if (v2, v1) in egresses_to_path_id and (v1, v2) not in␣
↪egresses_to_path_id:
                    f = lambda l: reversed(l)
                    pathlist = egresses_to_path_id[(v2, v1)]
                else:
                    pathlist = egresses_to_path_id[(v1, v2)]
                best_path = None
                best_dist = float("inf")
                for p in pathlist:
                    if path_length[p] < best_dist:
                        best_path = collapsed_paths_dict[p]
                        best_dist = path_length[p]
                output += list(f(best_path))
        output.append(v2)
        return list(map(lambda e: l2g_ids[e], output))

    # case 2: both removed, on same path
    if start_l in node_to_path_id and fin_l in node_to_path_id and␣
↪node_to_path_id[start_l] == node_to_path_id[fin_l]:
        #print("Case 2.")
        pid = node_to_path_id[fin_l]
        (s1, a_dist) = better_egress[start_l]
        (s2, b_dist) = better_egress[fin_l]
        s1c = inverted_compressed_translations[s1]
        s2c = inverted_compressed_translations[s2]
        candidate1 = ddash_within_path(start_l, fin_l, pid)
        candidate2 = ddash(start_l, s1, pid) + dc(s1c, s2c) + ddash(s2, fin_l,␣
↪pid)
        path = collapsed_paths_dict[pid]
        if candidate1 > candidate2:
            if path.index(start_l) < path.index(fin_l):
                path = list(reversed(path))
            first_portion = path[path.index(start_l):]
            last_portion = path[:path.index(fin_l)+1]
            first_portion = list(map(lambda e: l2g_ids[e], first_portion))
            last_portion = list(map(lambda e: l2g_ids[e], last_portion))
            return first_portion + calculatePath(matrix, distance_matrix,␣
↪inverted_translations[s1], inverted_translations[s2], g2l_ids, l2g_ids) +␣
↪last_portion
        else:
            if path.index(start_l) > path.index(fin_l):
                path = list(reversed(path))
            output = path[path.index(start_l):path.index(fin_l)+1]
            return list(map(lambda e: l2g_ids[e], output))

    # case 3a: start removed, but end is not
```

```
    if start_l in node_to_path_id and fin_l not in node_to_path_id:
        print("Case 3a.")
        pid = node_to_path_id[start_l]
        (s1, s2) = path_id_to_egresses[pid]
        s1c = inverted_compressed_translations[s1]
        s2c = inverted_compressed_translations[s2]
        candidate1 = ddash(start_l, s1, pid) + dc(s1c, fin_c)
        candidate2 = ddash(start_l, s2, pid) + dc(s2c, fin_c)

        if candidate1 < candidate2:
            egress_begin = s1
        else:
            egress_begin = s2

        path = [s1] + collapsed_paths_dict[pid] + [s2]
        if path.index(start_l) > path.index(egress_begin):
            path = list(reversed(path))
        first_portion = path[path.index(start_l):-1]
        first_portion = list(map(lambda e: l2g_ids[e], first_portion))
        return first_portion + calculatePath(matrix, distance_matrix, inverted_translations[egress_begin], fin_g, g2l_ids, l2g_ids)

    # case 3b: start not removed, but end is
    if start_l not in node_to_path_id and fin_l in node_to_path_id:
        #print("Case 3b.")
        pid = node_to_path_id[fin_l]
        (s1, s2) = path_id_to_egresses[pid]
        s1c = inverted_compressed_translations[s1]
        s2c = inverted_compressed_translations[s2]
        candidate1 = dc(start_c, s1c) + ddash(s1, fin_l, pid)
        candidate2 = dc(start_c, s2c) + ddash(s2, fin_l, pid)

        if candidate1 < candidate2:
            egress_end = s1
        else:
            egress_end = s2

        path = [s1] + collapsed_paths_dict[pid] + [s2]
        if path.index(fin_l) < path.index(egress_end):
            path = list(reversed(path))
        last_portion = path[1:path.index(fin_l)+1]
        last_portion = list(map(lambda e: l2g_ids[e], last_portion))
        return calculatePath(matrix, distance_matrix, start_g, inverted_translations[egress_end], g2l_ids, l2g_ids) + last_portion

    # case 4: both removed but on different chains
    #print("Case 4.")
```

```
    pid_start = node_to_path_id[start_l]
    pid_fin = node_to_path_id[fin_l]

    (s1_start, s2_start) = path_id_to_egresses[pid_start]
    s1c_start = inverted_compressed_translations[s1_start]
    s2c_start = inverted_compressed_translations[s2_start]

    (s1_fin, s2_fin) = path_id_to_egresses[pid_fin]
    s1c_fin = inverted_compressed_translations[s1_fin]
    s2c_fin = inverted_compressed_translations[s2_fin]

    candidate1 = ddash(start_l, s1_start, pid_start) + dc(s1c_start, s1c_fin) + ddash(s1_fin, fin_l, pid_fin)
    candidate2 = ddash(start_l, s2_start, pid_start) + dc(s2c_start, s1c_fin) + ddash(s1_fin, fin_l, pid_fin)
    candidate3 = ddash(start_l, s1_start, pid_start) + dc(s1c_start, s2c_fin) + ddash(s2_fin, fin_l, pid_fin)
    candidate4 = ddash(start_l, s2_start, pid_start) + dc(s2c_start, s2c_fin) + ddash(s2_fin, fin_l, pid_fin)

    best = min([candidate1, candidate2, candidate3, candidate4])
    #print(best)
    if best == candidate1:
        s_start = s1_start
        s_fin = s1_fin
    elif best == candidate2:
        s_start = s2_start
        s_fin = s1_fin
    elif best == candidate3:
        s_start = s1_start
        s_fin = s2_fin
    else: # best == candidate4
        s_start = s2_start
        s_fin = s2_fin

    start_path = [s1_start] + collapsed_paths_dict[pid_start] + [s2_start]
    if start_path.index(start_l) > start_path.index(s_start):
        start_path = list(reversed(start_path))
    first_portion = start_path[start_path.index(start_l):-1]
    first_portion = list(map(lambda e: l2g_ids[e], first_portion))

    fin_path = [s1_fin] + collapsed_paths_dict[pid_fin] + [s2_fin]
    if fin_path.index(fin_l) < fin_path.index(s_fin):
        fin_path = list(reversed(fin_path))
    last_portion = fin_path[1:fin_path.index(fin_l)+1]
    last_portion = list(map(lambda e: l2g_ids[e], last_portion))
```

```
    return first_portion + \
        calculatePath(matrix, distance_matrix, inverted_translations[s_start],␣
 ↪inverted_translations[s_fin], g2l_ids, l2g_ids) + \
        last_portion
```

[ ]:
```
matrix = parseMatrix(path_to_matrix, inverted_translations)
distance_matrix = parseMatrix(path_to_distance_matrix, do_optimization=False)
print("Simulator data loaded!")
```

1.5 Section 3: Comparison of Outputs

To visualize paths, choose two nodes below and see the results.

[ ]:
```
import random

two_valency_nodes_l = list(two_valency_nodes)
two_valency_nodes_g = list(map(lambda e: inverted_translations[e],␣
 ↪two_valency_nodes_l))

# CHANGE ME
orig1 = list(graph.nodes())[0]
dest1 = list(graph.nodes())[-1]
orig2 = list(graph.nodes())[50]
dest2 = list(graph.nodes())[-50]
orig3 = random.choice(two_valency_nodes_g) # 21272693
dest3 = random.choice(two_valency_nodes_g) # 21272694
orig4 = inverted_translations[20] #random.choice(list(graph.nodes()))
dest4 = inverted_translations[27] #random.choice(list(graph.nodes()))
```

[ ]:
```
# calculate shortest paths for the 2 routes
route1 = nx.shortest_path(graph, orig1, dest1, weight='length')
route2 = nx.shortest_path(graph, orig2, dest2, weight='length')
route3 = nx.shortest_path(graph, orig3, dest3, weight='length')
route4 = nx.shortest_path(graph, orig4, dest4, weight='length')

routes = [route1, route2, route3, route4]
rc = ['b', 'y', 'g', 'r']
fig, ax = ox.plot_graph_routes(graph, routes, route_colors=rc,␣
 ↪route_linewidth=6, node_size=0)
print("True shortest paths")

pt = ox.graph_to_gdfs(graph, edges=False).unary_union.centroid
bbox = ox.utils_geo.bbox_from_point((pt.y, pt.x), dist=5000)
#fig, ax = ox.plot_graph_routes(graph, [route4, route4], ['r', 'r'], bbox=bbox)
```

```
[ ]: calcr1 = calculatePath(matrix, distance_matrix, orig1, dest1, translations,␣
      ↪inverted_translations)
     calcr2 = calculatePath(matrix, distance_matrix, orig2, dest2, translations,␣
      ↪inverted_translations)
     calcr3 = calculatePath(matrix, distance_matrix, orig3, dest3, translations,␣
      ↪inverted_translations)
     calcr4 = calculatePath(matrix, distance_matrix, orig4, dest4, translations,␣
      ↪inverted_translations)

     fig, ax = ox.plot_graph_routes(graph, [calcr1, calcr2, calcr3, calcr4],␣
      ↪route_colors=rc, route_linewidth=6, node_size=0)
     print("Simulator shortest paths")
```

```
[ ]: print("Do they agree on path 1? " + "Yes!" if calcr1 == route1 else f"No:␣
      ↪\n{calcr1} \n{route1}")
     print("Do they agree on path 2? " + "Yes!" if calcr2 == route2 else f"No:␣
      ↪\n{calcr2} \n{route2}")
     print("Do they agree on path 3? " + "Yes!" if calcr3 == route3 else f"No:␣
      ↪\n{calcr3} \n{route3}")
     print("Do they agree on path 4? " + "Yes!" if calcr4 == route4 else f"No:␣
      ↪\n{calcr4} \n{route4}")

     # nx.shortest_path_length(graph, orig2, dest2, weight='length')
```

```
[ ]: nodes = list(graph.nodes())
     for start in nodes:
         for end in nodes:
             route = nx.shortest_path(graph, start, end, weight='length')
             try:
                 calcr = calculatePath(matrix, distance_matrix, start, end,␣
      ↪translations, inverted_translations)
             except BaseException as e:
                 print(start)
                 print(end)

                 raise e
             try:
                 assert(route == calcr)
             except AssertionError as e:
                 # IMPORTANT NOTE: Any failed cases require manual investigation.␣
      ↪The model solution will sometimes differ if there are multiple shortest␣
      ↪paths.
                 #                 This is rare, but does occurs thanks to the many␣
      ↪roundabouts in Britain.
                 print("Consider checking", start, end)
     print("Test successful!")
```