BURSA TEKNİK ÜNİVERSİTESİ
LİSANSÜSTÜ EĞİTİM ENSTİTÜSÜ

Homework Series: “Progressive TSP Research”
Assignment 2 – Real Map TSP

Ödev Raporu

Şevval Uzar

AMAÇ

Bu Assignment'ın amacı Python osmnx kütüphanesi ile gerçek harita verileri üzerinden Assignment 1'de kullanmış olduğum bir Travelling Salesman Problem'i olan Nearest Neighbour Algorithm ile çözümü gerçekleştirmektir.

Bu işlemi Ankara'nın Çankaya ilçesinde bulunan Bahçelievler, Yukarı Bahçelievler ve Emek mahalleleri üzerinde ele aldım.

SCRIPT VE KULLANILAN YÖNTEM AÇIKLAMALARI

```
import osmnx as ox
import networkx as nx
import folium
import random
import matplotlib.pyplot as plt
from google.colab import files
```

- Osmnx kütüphanesi import edilerek belirlenilen harita verisi projeye dahil edildi.
- Networkx kütüphanesi ile node'lar ve kenarlar arasında en kısa yol ve connected component gibi işlemler yapıldı.
- Folium ile html formatında harita üzerinde oluşturulmuş yol ve köşe görselleştirmeleri yapıldı.

```
location = ["Bahçelievler, Çankaya, Ankara, Turkey", "Yukarı Bahçelievler, Çankaya, Ankara, Turkey", "Emek, Çankaya, Ankara, Turkey"]
NETWORK_TYPE = "drive"
MAX_TSP_NODES = 600
RANDOM_SEED = 42
SAVE_HTML= "tsp_ankara.html"
```

- Osmnx kütüphanesi ile lokasyon adları yazılarak ilgili bölgelerin yol ağı indirildi ve NETWORK_TYPE olarak sadece araç ile gidilen yollar dahil edildi.

```
G = ox.graph_from_place(location, network_type=NETWORK_TYPE, simplify=True)
```

- graph_from_place() fonksiyonu OpenStreetMap verilerini çekerek her kavşağı düğüm, yolları ise kenar olarak graph yapısına dönüştürür.

```
all_nodes = list(G.nodes())
valid_nodes = [n for n in all_nodes if 'x' in G.nodes[n] and 'y' in
G.nodes[n]]
if len(valid_nodes) > MAX_TSP_NODES:
    tsp_nodes = random.sample(valid_nodes, MAX_TSP_NODES)
else:
    tsp_nodes = valid_nodes
NUM_TSP_NODES = len(tsp_nodes)
print("Selected TSP nodes:", NUM_TSP_NODES)
```

- Bu kod bloğu, TSP algoritmasında kullanılacak noktaları seçmek için yazılmıştır. Önce haritadaki tüm düğümler listelenir ve sadece koordinat bilgisi olanlar alınır. Eğer geçerli düğüm sayısı belirlenen maksimumdan fazlaysa, rastgele bir kısmı seçilir.

```
def nearest_neighbor(nodes, pair_dist):
    unvisited = set(nodes)
    current = nodes[0]
    tour = [current]
    unvisited.remove(current)
    while unvisited:
        next_node = min(unvisited, key=lambda x:
pair_dist.get((current,x), float('inf')))
        tour.append(next_node)
        unvisited.remove(next_node)
        current = next_node
    return tour
```

- Bu fonksiyon Nearest Neighbor algoritmasını kullanarak verilen düğümler için bir başlangıç TSP turu oluşturur. Turu başlatmak için ilk düğümü seçer ve her adımda henüz ziyaret edilmemiş düğümler arasından mevcut düğüme en yakın olanı ekler. Bu işlem tüm düğümler ziyaret edilene kadar devam eder.

PYTHON SCRIPT'İNİN TAMAMI

```
import osmnx as ox
import networkx as nx
import folium
import random
import matplotlib.pyplot as plt
from google.colab import files

location = ["Bahçelievler, Çankaya, Ankara, Turkey", "Yukarı
Bahçelievler, Çankaya, Ankara, Turkey", "Emek, Çankaya, Ankara,
Turkey"]
NETWORK_TYPE = "drive"
MAX_TSP_NODES = 600
RANDOM_SEED = 42
SAVE_HTML = "tsp_ankara.html"
random.seed(RANDOM_SEED)

G = ox.graph_from_place(location, network_type=NETWORK_TYPE,
simplify=True)
Gcc_nodes = max(nx.strongly_connected_components(G), key=len)
G = G.subgraph(Gcc_nodes).copy()
print("Nodes in largest connected component:", len(G.nodes))

all_nodes = list(G.nodes())
valid_nodes = [n for n in all_nodes if 'x' in G.nodes[n] and 'y' in
G.nodes[n]]
if len(valid_nodes) > MAX_TSP_NODES:
    tsp_nodes = random.sample(valid_nodes, MAX_TSP_NODES)
else:
    tsp_nodes = valid_nodes
NUM_TSP_NODES = len(tsp_nodes)
print("Selected TSP nodes:", NUM_TSP_NODES)
def node_latlon(G, node): #enlem ve boylam (latitude, longitude)
    return (G.nodes[node]['y'], G.nodes[node]['x'])

pair_dist = {}
pair_path = {}
for i, a in enumerate(tsp_nodes):
    lengths, paths = nx.single_source_dijkstra(G, a, weight='length')
    for j, b in enumerate(tsp_nodes):
        if a == b: continue
        if b in lengths:
            pair_dist[(a,b)] = lengths[b]
            pair_path[(a,b)] = paths[b]
        else:
            pair_dist[(a,b)] = float('inf')
            pair_path[(a,b)] = None
```

```
def nearest_neighbor(nodes, pair_dist):
    unvisited = set(nodes)
    current = nodes[0]
    tour = [current]
    unvisited.remove(current)
    while unvisited:
        next_node = min(unvisited, key=lambda x:
pair_dist.get((current,x), float('inf')))
        tour.append(next_node)
        unvisited.remove(next_node)
        current = next_node
    return tour

def tour_length(tour, pair_dist):
    return sum(pair_dist.get((tour[i], tour[(i+1)%len(tour)]),
float('inf')) for i in range(len(tour)))

def two_opt(tour, pair_dist):
    improved = True
    best = tour[:]
    best_len = tour_length(best, pair_dist)
    N = len(tour)
    while improved:
        improved = False
        for i in range(1, N-1):
            for j in range(i+1, N):
                if j - i == 1: continue
                new_tour = best[:i] + best[i:j+1][::-1] + best[j+1:]
                new_len = tour_length(new_tour, pair_dist)
                if new_len < best_len:
                    best = new_tour
                    best_len = new_len
                    improved = True
    return best

nn_tour = nearest_neighbor(tsp_nodes, pair_dist)
print("Initial NN tour length (m):", tour_length(nn_tour, pair_dist))
opt_tour = two_opt(nn_tour, pair_dist)
print("After 2-opt tour length (m):", tour_length(opt_tour, pair_dist))

route_nodes_full = []
for i in range(len(opt_tour)):
    a = opt_tour[i]
    b = opt_tour[(i+1) % len(opt_tour)]
    path = pair_path.get((a,b))
    if path:
        if not route_nodes_full:
            route_nodes_full += path
```

```
        else:
            if route_nodes_full[-1] == path[0]:
                route_nodes_full += path[1:]
            else:
                route_nodes_full += path

lats = [node_latlon(G, n)[0] for n in tsp_nodes]
lons = [node_latlon(G, n)[1] for n in tsp_nodes]
center = (sum(lats)/len(lats), sum(lons)/len(lons))
m = folium.Map(location=center, zoom_start=15)

route_coords = []
for u,v in zip(route_nodes_full[:-1], route_nodes_full[1:]):
    data = G.get_edge_data(u, v)
    if data:
        e = data[list(data.keys())[0]]
        if 'geometry' in e and e['geometry'] is not None:
            route_coords.extend([(lat, lon) for lon, lat in
e['geometry'].coords])
        else:
            route_coords.append((G.nodes[u]['y'], G.nodes[u]['x']))
            route_coords.append((G.nodes[v]['y'], G.nodes[v]['x']))
    else:
        route_coords.append((G.nodes[u]['y'], G.nodes[u]['x']))

if route_coords:
    folium.PolyLine(route_coords, weight=5, opacity=0.8,
color='red').add_to(m)

for idx, n in enumerate(opt_tour):
    lat, lon = node_latlon(G, n)
    folium.CircleMarker(location=(lat, lon),
                        radius=5,
                        popup=f"{idx+1}",
                        tooltip=f"Node {idx+1}",
                        fill=True).add_to(m)

m.save(SAVE_HTML)
files.download(SAVE_HTML)
```