Bilgisayar Oyunlarında Yapay Zekâ Ödev-5 Raporu

Metehan SÖZENLİ 25435004037

1. Giriş

Ödev kapsamında, önceki ödevde ele alınan klasik Gezgin Satıcı Problemi (TSP) genişletilerek TSP with Neighborhoods (TSPN) problemine dönüştürülmüştür. Klasik TSP'de her şehir tek bir nokta olarak temsil edilirken, TSPN probleminde her şehir bir bölge (neighborhood) olarak tanımlanmaktadır. Geliştirilen çalışmada her konum, gerçek yol ağı üzerinde konumlandırılmış bir merkez ve bu merkezin etrafında tanımlanan dairesel bir bölge ile modellenmiştir. Bir turun bir bölgeyi ziyaret etmiş sayılabilmesi için, turun ilgili dairenin sınırına temas etmesi yeterli kabul edilmiştir. Böylece tur, bölge merkezlerine gitmek yerine bölge sınırları üzerinden ilerlemektedir.

2. Yöntem

2.1. Yol Ağının Elde Edilmesi (OSMnx)

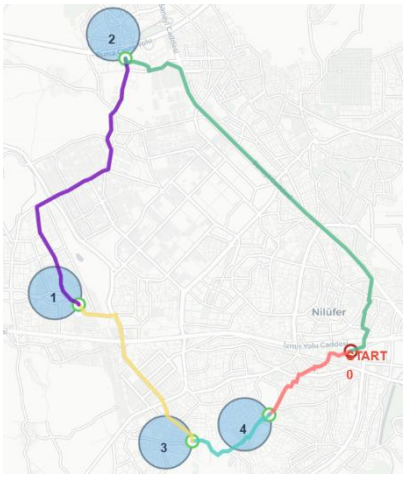
Yol ağı verileri, OpenStreetMap altyapısını kullanan OSMnx kütüphanesi aracılığıyla elde edilmiştir. OSMnx, kullanıcı tarafından tanımlanan bir coğrafi alan için yol ağını indirerek, düğümlerin kavşakları ve kenarların yol segmentlerini temsil ettiği grafik yapılar oluşturmaktadır. Elde edilen yol ağı, kenar ağırlıkları yol uzunluklarını ifade edecek biçimde tanımlanmış ağırlıklı bir grafik olarak modellenmiş ve rota hesaplamalarında kullanılmak üzere uygun hâle getirilmiştir. Şekil 1'de ilgili kod parçası verilmiştir.

```python
def extract_route_geometry(G: nx.MultiDiGraph, path_nodes: List[int]) -> List[Tuple[float, float]]:
    """Extract detailed route geometry (lat, lon) from path nodes using edge geometries"""
    if len(path_nodes) < 2:
        # Single node, return its coordinates
        if len(path_nodes) == 1:
            node = path_nodes[0]
            return [(float(G.nodes[node]["y"]), float(G.nodes[node]["x"]))]
        return []

    coords = []

    for i in range(len(path_nodes) - 1):
        u = path_nodes[i]
        v = path_nodes[i + 1]

        # Add starting node coordinates
        if i == 0:
            coords.append((float(G.nodes[u]["y"]), float(G.nodes[u]["x"])))

        # Check if edge has geometry attribute (from OSMnx simplification)
        edge_data = None
        if G.has_edge(u, v):
            # Get the first edge (key=0) if it exists
            edges = G[u][v]
            if edges:
                edge_data = edges[list(edges.keys())[0]]

        if edge_data and 'geometry' in edge_data:
            # Use the detailed geometry
            geom = edge_data['geometry']
            for point in list(geom.coords)[1:]:
                coords.append((point[1], point[0]))
        else:
            # No geometry, just add the end node
            coords.append((float(G.nodes[v]["y"]), float(G.nodes[v]["x"])))

    return coords
```

Şekil 1: Rastgele bölgeleri oluşturan kod parçası.

2.2. Alan (Neighborhood) Tanımı

Gezgin Satıcı Problemi, TSP with Neighborhoods (TSPN) formülasyonu üzerinden ele alınmıştır. Bu kapsamda her şehir, tekil bir nokta yerine belirli bir alan olarak modellenmiştir. Şekil 2'de gösterildiği gibi alanlar, bir merkez noktası ve bu merkezin etrafında tanımlanan sabit yarıçaplı dairesel bölgeler ile temsil edilmiştir.



Şekil 2: Dairesel alanlar (neighborhoods) ile modellenmiş konumlar.

Bölge merkezleri, gerçek yol ağı üzerinde yer alan düğümler arasından seçilmiş ve her alan için aynı yarıçap değeri kullanılmıştır. Böylece problem, belirli noktalara ulaşma zorunluluğu yerine, tanımlı bir alanın herhangi bir noktasına erişme şartı üzerinden ifade edilmiştir. Başlangıç noktası olarak tanımlanan ilk bölge için alan gösterimi yapılmamış, diğer bölgeler dairesel alanlar olarak modellenmiştir. Tüm bu işlemleri gerçekleştiren kod parçası Şekil 3'teki gibidir.

```python
def generate_random_regions(place: str, n: int, radius_m: float, seed: int, G: nx.MultiDiGraph) -> List[RegionCircle]:
    random.seed(seed)
    np.random.seed(seed)

    gdf = ox.geocode_to_gdf(place)
    geom = gdf.geometry.iloc[0]

    # Collect candidate nodes inside the polygon
    candidates = []
    for node, data in G.nodes(data=True):
        lon = float(data.get("x"))
        lat = float(data.get("y"))
        pt = ox.utils_geo.Point(lon, lat)
        if geom.contains(pt):
            candidates.append(int(node))

    if len(candidates) < n:
        raise RuntimeError(f"Not enough road nodes inside '{place}' to sample {n} regions (found {len(candidates)}).")

    sampled = random.sample(candidates, n)
    regions = []
    for i, node in enumerate(sampled):
        lat = float(G.nodes[node]["y"])
        lon = float(G.nodes[node]["x"])
        is_start = (i == 0)  # First region is starting point
        regions.append(RegionCircle(region_id=i, lat=lat, lon=lon, radius_m=radius_m, is_start=is_start))
    return regions
```

Şekil 3: Rastgele bölgeleri oluşturan kod parçası.

2.3. Bölgeye Giriş Kriteri

TSPN probleminde bir bölgenin ziyaret edilmiş sayılabilmesi için, turun ilgili alanın merkezine ulaşması gerekmez. Bu nedenle ziyaret koşulu, turun dairesel alanın sınırına temas ettiği ilk nokta olarak tanımlanmıştır. Böylece ziyaret, alanın tamamını kapsayan bir koşul hâline getirilmiştir. Bölgeye giriş noktası, iki bölge arasındaki yol üzerinde alan sınırının ilk kez kesildiği nokta olarak ele alınmıştır. Bu tanım sayesinde tur uzunluğu, merkez noktaya olan mesafeye bağlı olmaktan çıkarılarak alan temelli bir ölçüt üzerinden değerlendirilmiştir. Bu işlemi gerçekleştiren ilgili kod Şekil 4'te gösterilmiştir.

```python
def find_boundary_intersection(G: nx.MultiDiGraph, path_nodes: List[int], target_region: RegionCircle) -> Tuple[int, Tuple[float, float]]:
    if target_region.is_start:
        node = path_nodes[-1]
        node_lat = float(G.nodes[node]["y"])
        node_lon = float(G.nodes[node]["x"])
        return node, (node_lat, node_lon)

    # Walk through path and detect crossing
    for i in range(1, len(path_nodes)):
        prev_node = path_nodes[i - 1]
        curr_node = path_nodes[i]
        prev_lat = float(G.nodes[prev_node]["y"])
        prev_lon = float(G.nodes[prev_node]["x"])
        curr_lat = float(G.nodes[curr_node]["y"])
        curr_lon = float(G.nodes[curr_node]["x"])
        prev_dist = geodesic_distance(prev_lat, prev_lon, target_region.lat, target_region.lon)
        curr_dist = geodesic_distance(curr_lat, curr_lon, target_region.lat, target_region.lon)

        # Crossing detected: prev outside, curr inside
        if prev_dist > target_region.radius_m and curr_dist <= target_region.radius_m:
            entry_point = line_circle_intersection(
                prev_lat, prev_lon,
                curr_lat, curr_lon,
                target_region.lat, target_region.lon,
                target_region.radius_m
            )
            return prev_node, entry_point

    # No crossing - find closest point to boundary
    best_node = path_nodes[-1]
    best_point = (float(G.nodes[best_node]["y"]), float(G.nodes[best_node]["x"]))
    min_dist_to_boundary = float('inf')

    for node in path_nodes:
        node_lat = float(G.nodes[node]["y"])
        node_lon = float(G.nodes[node]["x"])
        dist_to_center = geodesic_distance(node_lat, node_lon, target_region.lat, target_region.lon)
        dist_to_boundary = abs(dist_to_center - target_region.radius_m)

        if dist_to_boundary < min_dist_to_boundary:
            min_dist_to_boundary = dist_to_boundary
            best_node = node
            best_point = (node_lat, node_lon)

    return best_node, best_point
```

Şekil 4: Bölge sınırına temas koşulunu tanımlayan kod parçası.

2.4. Problem Temsili ve Mesafe Matrisi

Tanımlanan alanlar arasındaki ziyaret sırası, klasik TSP formülasyonuna uygun olacak şekilde alan merkezleri üzerinden temsil edilmiştir. Her alan için yol ağı üzerinde en yakın düğüm merkez noktası olarak kabul edilmiş ve alanlar arası mesafeler bu merkez düğümler arasındaki en kısa yol uzunlukları kullanılarak tanımlanmıştır. Bu temsil sayesinde TSPN problemi, çözücüler açısından klasik TSP yapısını koruyan bir mesafe matrisi üzerinden ifade edilmiştir. Alanlara giriş noktalarının hesaplanması ve gerçek tur uzunluğunun belirlenmesi ise problem tanımının bir parçası olarak ele alınmıştır.

```python
def build_center_distance_matrix(G: nx.MultiDiGraph, regions: List[RegionCircle]) -> Tuple[np.ndarray, List[int]]:
    """Distance matrix between region centers"""
    center_nodes = [int(ox.distance.nearest_nodes(G, r.lon, r.lat)) for r in regions]
    n = len(regions)
    D = np.zeros((n, n), dtype=float)
    for i in range(n):
        for j in range(n):
            if i == j:
                D[i, j] = 0.0
            else:
                d = safe_shortest_path_length(G, center_nodes[i], center_nodes[j], weight="length")
                D[i, j] = d
    return D, center_nodes
```

Şekil 5: Mesafe matrisini oluşturan kod parçası.

2.5. Çözücülerin Kullanımı

Bu çalışmada kullanılan Nearest Neighbor, OR-Tools ve Genetik Algoritma çözücüleri, önceki ödevlerdeki yapıları korunarak kullanılmıştır. Çözücüler, alan merkezleri üzerinden oluşturulan mesafe matrisi üzerinde çalışmakta ve yalnızca alanların ziyaret sırasını üretmektedir. TSPN problemine özgü alan tanımı ve giriş kriterleri, çözücülerin diğer ödevlerdeki işlevlerine müdahale edilmeden uygulanmıştır.

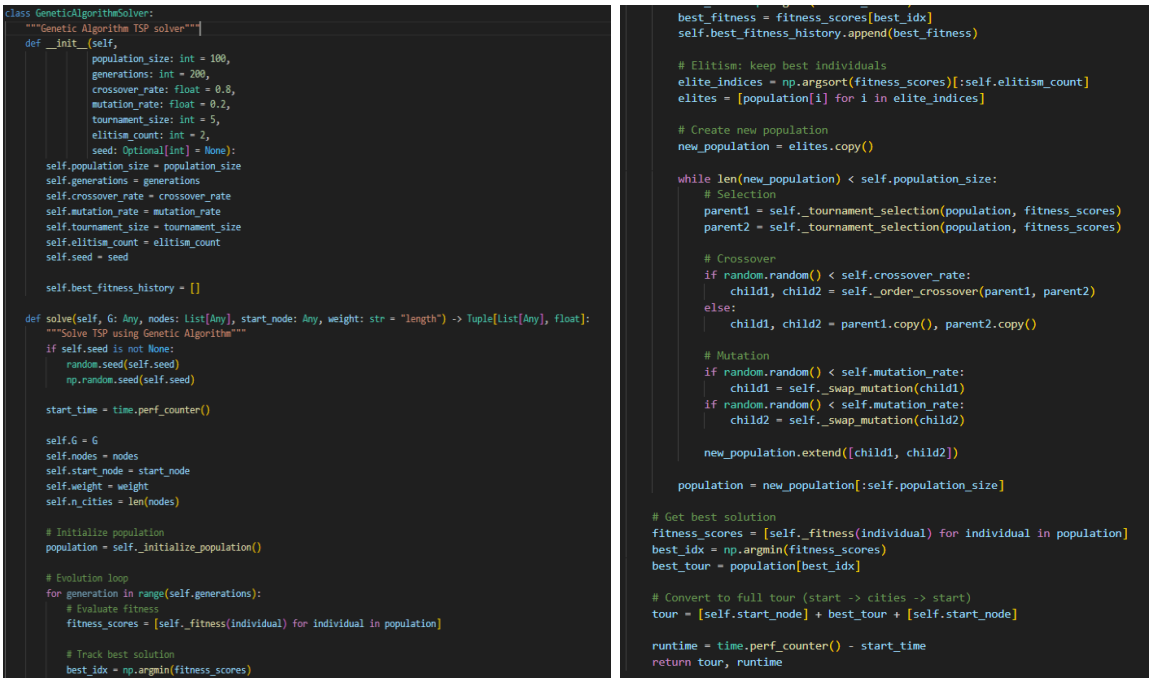
```python
class NearestNeighborSolver:
    """Greedy heuristic TSP solver"""

    def solve(self, G: Any, nodes: List[Any], start_node: Any, weight: str = "length") -> Tuple[List[Any], float]:
        """
        Solve TSP using Nearest Neighbor heuristic
        Returns: (tour, runtime_seconds)
        """
        start_time = time.perf_counter()

        unvisited = set(nodes)
        tour = [start_node]
        unvisited.remove(start_node)

        while unvisited:
            current = tour[-1]
            # Find nearest unvisited node (direct edge weight)
            nearest = min(unvisited, key=lambda node: G[current][node][weight])
            tour.append(nearest)
            unvisited.remove(nearest)

        tour.append(start_node)  # Return to start
        runtime = time.perf_counter() - start_time

        return tour, runtime
```

Şekil 6: Nearest Neighbor çözücüsü.

```python
class ORToolsSolver:
    def __init__(self, time_limit_s: int = 10):
        self.time_limit_s = time_limit_s

    def solve(self, G: Any, nodes: List[Any], start_node: Any, weight: str = "length") -> Tuple[List[Any], float]:
        """
        Solve TSP using Google OR-Tools
        Returns: (tour, runtime_seconds)
        """
        start_time = time.perf_counter()

        # Build node index mappings
        n = len(nodes)
        node_to_idx = {node: idx for idx, node in enumerate(nodes)}
        idx_to_node = {idx: node for idx, node in enumerate(nodes)}
        start_idx = node_to_idx[start_node]

        # Distance scale for integer precision
        scale = 1000.0

        # Create OR-Tools routing model
        manager = pywrapcp.RoutingIndexManager(n, 1, start_idx)
        routing = pywrapcp.RoutingModel(manager)

        # Distance callback using direct edge weights
        def distance_callback(from_index, to_index):
            i = manager.IndexToNode(from_index)
            j = manager.IndexToNode(to_index)
            if i == j:
                return 0
            u = nodes[i]
            v = nodes[j]
            # Direct edge weight from graph
            length = G[u][v][weight]
            return int(length * scale)
```

```python
transit_callback_index = routing.RegisterTransitCallback(distance_callback)
routing.SetArcCostEvaluatorOfAllVehicles(transit_callback_index)

# Search parameters
search_parameters = pywrapcp.DefaultRoutingSearchParameters()
search_parameters.first_solution_strategy = (
    routing_enums_pb2.FirstSolutionStrategy.PATH_CHEAPEST_ARC)
search_parameters.local_search_metaheuristic = (
    routing_enums_pb2.LocalSearchMetaheuristic.GUIDED_LOCAL_SEARCH)
search_parameters.time_limit.FromSeconds(self.time_limit_s)

# Solve
solution = routing.SolveWithParameters(search_parameters)

if solution:
    # Extract tour
    index = routing.Start(0)
    tour_indices = []
    while not routing.IsEnd(index):
        tour_indices.append(manager.IndexToNode(index))
        index = solution.Value(routing.NextVar(index))
    tour_indices.append(tour_indices[0])  # Close loop

    tour = [idx_to_node[idx] for idx in tour_indices]
else:
    # Fallback
    tour = [start_node, start_node]

runtime = time.perf_counter() - start_time
return tour, runtime
```

Şekil 7: OR-Tool çözücüsü.

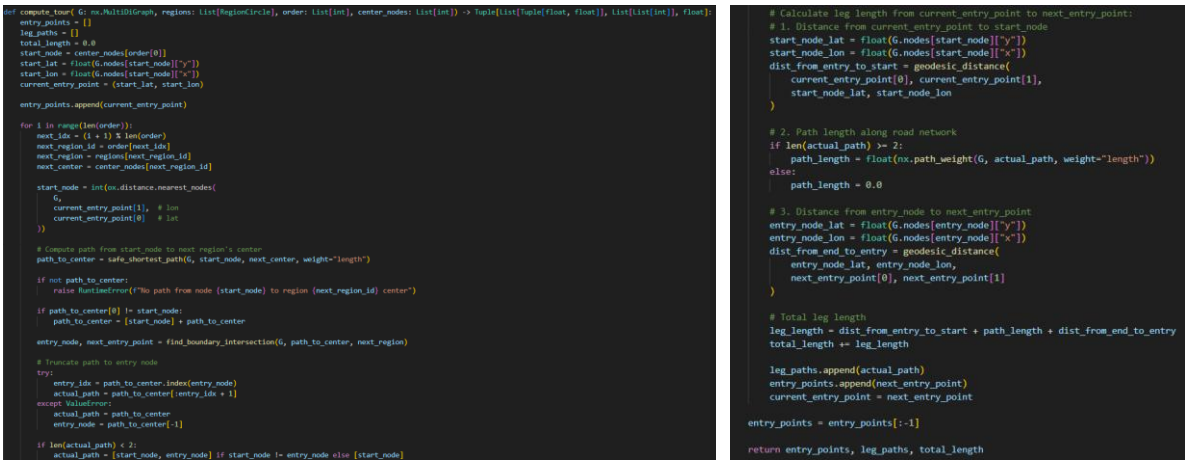


```python
class GeneticAlgorithmSolver:
    """Genetic Algorithm TSP solver"""
    def __init__(self,
                 population_size: int = 100,
                 generations: int = 200,
                 crossover_rate: float = 0.8,
                 mutation_rate: float = 0.2,
                 tournament_size: int = 5,
                 elitism_count: int = 2,
                 seed: Optional[int] = None):
        self.population_size = population_size
        self.generations = generations
        self.crossover_rate = crossover_rate
        self.mutation_rate = mutation_rate
        self.tournament_size = tournament_size
        self.elitism_count = elitism_count
        self.seed = seed

        self.best_fitness_history = []

    def solve(self, G: Any, nodes: List[Any], start_node: Any, weight: str = "length") -> Tuple[List[Any], float]:
        """Solve TSP using Genetic Algorithm"""
        if self.seed is not None:
            random.seed(self.seed)
            np.random.seed(self.seed)

        start_time = time.perf_counter()

        self.G = G
        self.nodes = nodes
        self.start_node = start_node
        self.weight = weight
        self.n_cities = len(nodes)

        # Initialize population
        population = self._initialize_population()

        # Evolution loop
        for generation in range(self.generations):
            # Evaluate fitness
            fitness_scores = [self._fitness(individual) for individual in population]

            # Track best solution
            best_idx = np.argmin(fitness_scores)
            best_fitness = fitness_scores[best_idx]
            self.best_fitness_history.append(best_fitness)

            # Elitism: keep best individuals
            elite_indices = np.argsort(fitness_scores)[:self.elitism_count]
            elites = [population[i] for i in elite_indices]

            # Create new population
            new_population = elites.copy()

            while len(new_population) < self.population_size:
                # Selection
                parent1 = self._tournament_selection(population, fitness_scores)
                parent2 = self._tournament_selection(population, fitness_scores)

                # Crossover
                if random.random() < self.crossover_rate:
                    child1, child2 = self._order_crossover(parent1, parent2)
                else:
                    child1, child2 = parent1.copy(), parent2.copy()

                # Mutation
                if random.random() < self.mutation_rate:
                    child1 = self._swap_mutation(child1)
                if random.random() < self.mutation_rate:
                    child2 = self._swap_mutation(child2)

                new_population.extend([child1, child2])

            population = new_population[:self.population_size]

        # Get best solution
        fitness_scores = [self._fitness(individual) for individual in population]
        best_idx = np.argmin(fitness_scores)
        best_tour = population[best_idx]

        # Convert to full tour (start -> cities -> start)
        tour = [self.start_node] + best_tour + [self.start_node]

        runtime = time.perf_counter() - start_time
        return tour, runtime
```

Şekil 8: Genetik Algoritma çözücüsü.

2.6. Rota Hesaplama

compute_tour fonksiyonu, çözücüler tarafından üretilen ziyaret sırasını temel alarak bu sıraya karşılık gelen gerçek turu yol ağı üzerinde oluşturmaktadır. Ardışık alanlar arasındaki her geçişte rota, mevcut konumdan hedef alan yönünde tanımlanan yol üzerinden ele alınmış ve yolun hedef alanın dairesel sınırına temas ettiği ilk noktada sonlandırılmıştır. Elde edilen temas noktası, bir sonraki rota parçasının başlangıcı olarak kabul edilerek tur, alan merkezleri yerine alan sınırları arasında giden bir rota şeklinde temsil edilmiştir. Tur uzunluğu ise yol ağına ait olan mesafeler dikkate alınarak tanımlanmıştır.



```python
def compute_tour( G: nx.MultiDiGraph, regions: List[RegionCircle], order: List[int], center_nodes: List[int]) -> Tuple[List[Tuple[float, float]], List[List[int]], float]:
    entry_points = []
    leg_paths = []
    total_length = 0.0
    start_node = center_nodes[order[0]]
    start_lat = float(G.nodes[start_node]["y"])
    start_lon = float(G.nodes[start_node]["x"])
    current_entry_point = (start_lat, start_lon)

    entry_points.append(current_entry_point)

    for i in range(len(order)):
        next_idx = (i + 1) % len(order)
        next_region_id = order[next_idx]
        next_region = regions[next_region_id]
        next_center = center_nodes[next_region_id]

        start_node = int(ox.distance.nearest_nodes(
            G,
            current_entry_point[1],  # lon
            current_entry_point[0]   # lat
        ))

        # Compute path from start_node to next region's center
        path_to_center = safe_shortest_path(G, start_node, next_center, weight="length")

        if not path_to_center:
            raise RuntimeError(f"No path from node {start_node} to region {next_region_id} center")

        if path_to_center[0] != start_node:
            path_to_center = [start_node] + path_to_center

        entry_node, next_entry_point = find_boundary_intersection(G, path_to_center, next_region)

        # Truncate path to entry node
        try:
            entry_idx = path_to_center.index(entry_node)
            actual_path = path_to_center[:entry_idx + 1]
        except ValueError:
            actual_path = path_to_center
            entry_node = path_to_center[-1]

        if len(actual_path) < 2:
            actual_path = [start_node, entry_node] if start_node != entry_node else [start_node]
```

```python
        # Calculate leg length from current_entry_point to next_entry_point:
        # 1. Distance from current_entry_point to start_node
        start_node_lat = float(G.nodes[start_node]["y"])
        start_node_lon = float(G.nodes[start_node]["x"])
        dist_from_entry_to_start = geodesic_distance(
            current_entry_point[0], current_entry_point[1],
            start_node_lat, start_node_lon
        )

        # 2. Path length along road network
        if len(actual_path) >= 2:
            path_length = float(nx.path_weight(G, actual_path, weight="length"))
        else:
            path_length = 0.0

        # 3. Distance from entry_node to next_entry_point
        entry_node_lat = float(G.nodes[entry_node]["y"])
        entry_node_lon = float(G.nodes[entry_node]["x"])
        dist_from_end_to_entry = geodesic_distance(
            entry_node_lat, entry_node_lon,
            next_entry_point[0], next_entry_point[1]
        )

        # Total leg length
        leg_length = dist_from_entry_to_start + path_length + dist_from_end_to_entry
        total_length += leg_length

        leg_paths.append(actual_path)
        entry_points.append(next_entry_point)
        current_entry_point = next_entry_point

    entry_points = entry_points[:-1]

    return entry_points, leg_paths, total_length
```

Şekil 9: Rota hesaplamasını gerçekleştiren ana fonksiyon.

2.7. Görselleştirme Yaklaşımı

Harita tabanlı görselleştirme için Folium kütüphanesi kullanılmıştır. ***visualize_tour*** fonksiyonu ile yol ağı üzerinde hesaplanan rotalar çizgi olarak, alanlar ise dairesel bölgeler olarak haritaya eklenmiştir. Görselleştirmede her alanın merkezi işaretlenmiş, başlangıç alanı diğerlerinden ayırt edilecek şekilde etiketlenmiştir. Alan sınırına temas edilen giriş noktaları ayrıca işaretlenerek, ziyaret kriterinin merkez değil sınır teması üzerinden tanımlandığı açık biçimde gösterilmiştir. Rota parçaları, ardışık geçişleri belirginleştirmek amacıyla farklı renklerle çizilmiş ve her çözücü için ayrı bir harita çıktısı üretilerek karşılaştırılabilir bir sunum elde edilmiştir.

2.8. Ana Program Akışı

main fonksiyonu, deneylerin genel akışını yöneten bölümdür. Program akışı içerisinde yol ağı yüklenmekte, alan tanımları oluşturulmakta ve tanımlanan problem yapısı çözücülere girdi olarak sunulmaktadır. Çözücüler tarafından üretilen ziyaret sıraları, rota hesaplama fonksiyonu aracılığıyla gerçek yol ağı üzerinde karşılık gelen turlara dönüştürülmüştür. Elde edilen sonuçlar, tur uzunluğu ve çalışma süresi gibi ölçütlerle birlikte saklanmış ve görselleştirme çıktıları oluşturulmuştur. Bu yapı, tüm çözüm sürecinin düzenli ve tekrarlanabilir biçimde yürütülmesini sağlamaktadır.

3. Deneysel Bulgular

3.1. Deney Kurulumu ve Parametreler

Deneyler, Nilüfer / Bursa bölgesine ait gerçek yol ağı üzerinde gerçekleştirilmiştir. Yol ağı verileri OpenStreetMap üzerinden elde edilmiş ve yalnızca araç trafiğine uygun yollar dikkate alınmıştır. Böylece rota hesaplamaları, gerçek bir ulaşım altyapısı üzerinde yürütülmüştür.

Deneylerde kullanılan temel parametreler aşağıdaki gibidir:

- Çalışma alanı: Nilüfer / Bursa
- Alan (neighborhood) sayısı: 22
- Alan yarıçapı: 200 metre
- Başlangıç alanı: İlk seçilen bölge

- Rastgelelik tohumu (seed): 6016
- Yol ağı türü: OpenStreetMap drive ağı
- Genetik Algoritma jenerasyon sayısı: 150
- Çözücüler: Nearest Neighbor, OR-Tools, Genetik Algoritma

Deneyler, aşağıdaki komut satırı parametreleri kullanılarak gerçekleştirilmiştir:

```
PS D:\Yüksek Lisans Dersler\Bilgisayar Oyunlarında Yapay Zeka\AI_in_Compter_Games_Assignments\Assignment-5> cd "d:\Yüksek Lisans Dersler\Bilgisayar Oyunların
da Yapay Zeka\AI_in_Compter_Games_Assignments\Assignment-5"; C:/Users/meteh/AppData/Local/Programs/Python/Python38/python.exe task5.py --regions 22 --radius-
m 200 --seed 6016 --ga-gen 150
```

3.2. Çözüm Kalitesi ve Tur Uzunlukları

Deneysel sonuçlar, kullanılan çözücülerin ürettiği toplam tur uzunlukları ve çalışma süreleri dikkate alınarak değerlendirilmiştir. Nearest Neighbor yöntemi, düşük hesaplama maliyeti sayesinde en kısa sürede çözüm üretmiş; ancak yerel karar mekanizması nedeniyle genellikle daha uzun turlar oluşturmuştur. Bu durum, algoritmanın global optimumdan uzak çözümler üretebilmesiyle ilişkilendirilmektedir.

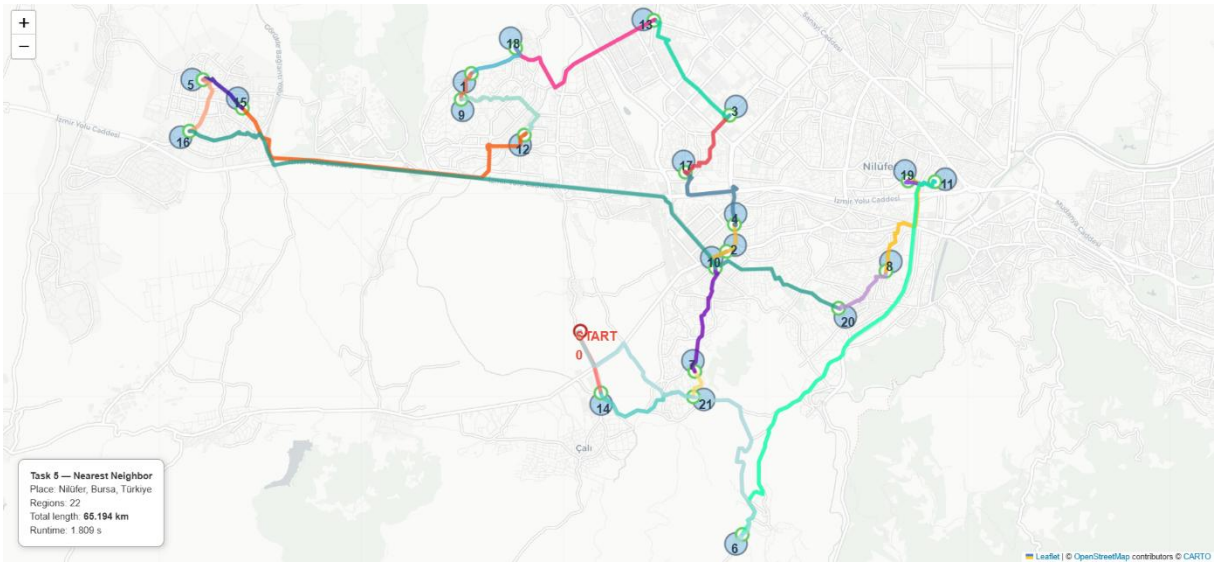
Genetik Algoritma, Nearest Neighbor'a kıyasla daha kısa turlar elde etmiş ve çözüm kalitesini artırmıştır. Bununla birlikte, iteratif ve popülasyon tabanlı yapısı sebebiyle çalışma süresi daha uzun olmuştur. OR-Tools tabanlı çözücü ise deneyde en kısa tur uzunluklarını üretmiş ve çözüm kalitesi açısından öne çıkmıştır. Çalışma süresi bakımından Nearest Neighbor'dan daha yavaş, Genetik Algoritma'dan ise daha hızlı sonuçlar üretmiştir. Genel olarak, OR-Tools çözüm kalitesi ile hesaplama süresi arasında daha dengeli bir yaklaşım sunmuştur. Sonuçlar Tablo 1'de gösterilmiştir.

Çözücü	**Toplam Uzunluk** (km)	**Çalışma Süresi** (s)
Nearest Neighbor	65.19	1.80
Genetic Algorithm	59.77	13.78
OR-Tools	56.72	3.57

Tablo 1: Çözücülerin deneysel sonuçları.

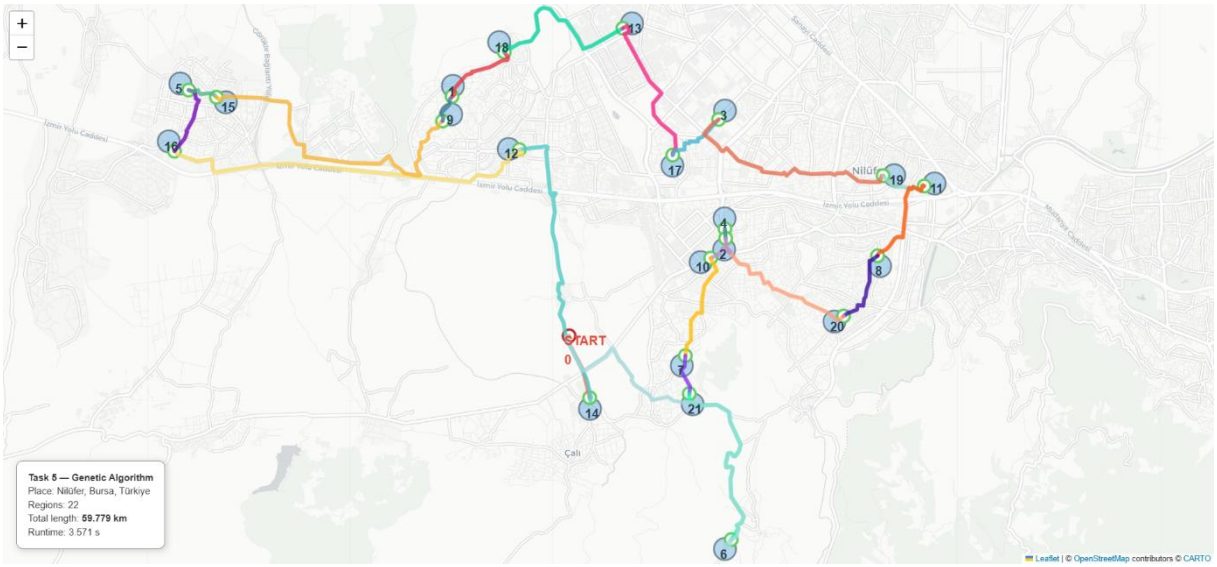
3.3. Rota Yapılarının Görsel Değerlendirilmesi

Harita tabanlı görselleştirmeler incelendiğinde, çözücülerin ürettiği rota yapıları arasında belirgin farklar olduğu gözlemlenmiştir. Nearest Neighbor yöntemi (Şekil 10), bazı bölgelerde gereksiz yön değişimleri ve dolaylı geçişler içeren rotalar oluşturmuştur.



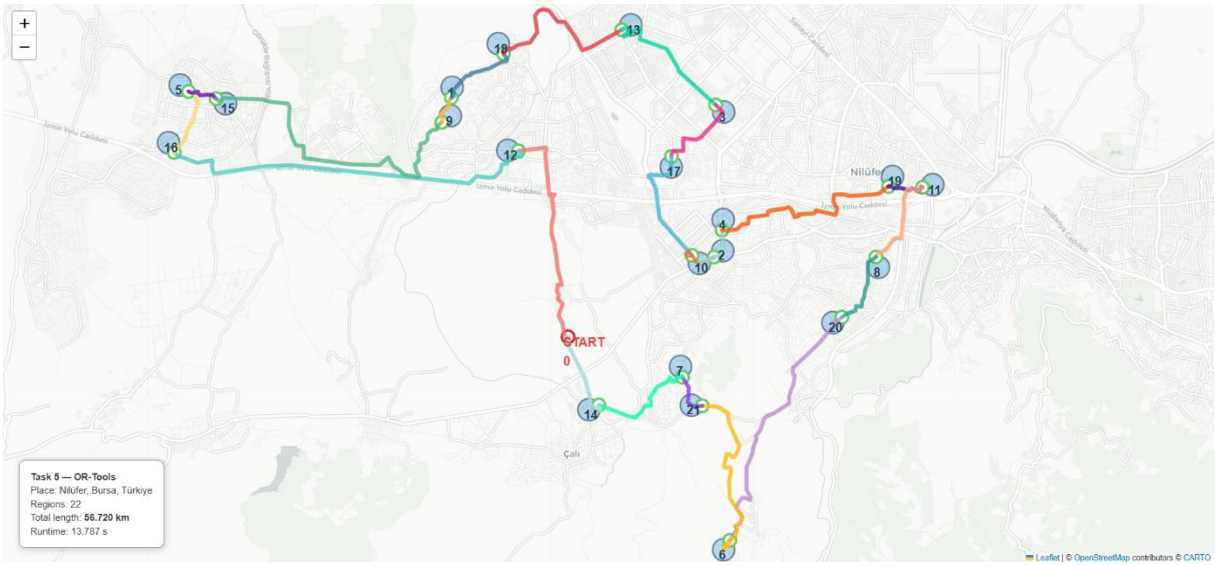
Şekil 10: Nearest Neighbor yöntemi ile elde edilen çıktı.

Genetik Algoritma tarafından üretilen rotalar (Şekil 11) ise daha dengeli bir yapı sergilemiş, ancak bazı durumlarda hâlâ optimal olmayan geçişler içermiştir.



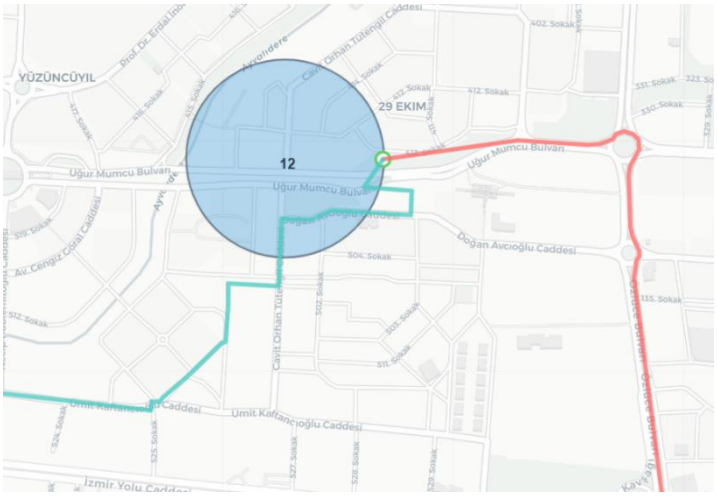
Şekil 11: Genetik Algoritma yöntemi ile elde edilen çıktı.

OR-Tools çözümleri (Şekil 12), genel olarak daha düzgün, daha kısa ve daha az kesişen rota yapıları ortaya koymuştur. Alan sınırlarına temas noktalarının dağılımı incelendiğinde, bu yöntemin alan temelli ziyaret mantığını daha verimli biçimde yansıttığı görülmüştür.



Şekil 12: Rota hesaplamasını gerçekleştiren ana fonksiyon.

Şekil 13'te rotanın ilgili alanın sınırına temas ettikten sonra alan merkezine yönelmeden bir sonraki hedefe doğru ilerlediğini görülmektedir. Bu durum, TSP with Neighborhoods yaklaşımında ziyaret koşulunun merkez noktaya ulaşmak yerine alan sınırına giriş üzerinden tanımlandığını göstermektedir. Böylece rota, alan içine girdikten sonra gereksiz bir merkez yönelimi oluşturmadan, yol ağına uygun ve süreklilik gösteren bir geçiş yapmaktadır.



Şekil 13: Rotanın alana girdikten sonra merkeze gitmeden diğer noktaya yönelmesi.

4. Sonuç ve Değerlendirme

Bu çalışmada odak noktası, klasik Gezgin Satıcı Problemi'nin TSP with Neighborhoods (TSPN) yaklaşımıyla genişletilmesi olmuştur. Şehirlerin tekil noktalar yerine alanlar olarak modellenmesi ve ziyaret koşulunun alan merkezine ulaşmak yerine alan sınırına temas edilmesi üzerinden tanımlanması, problemin gerçek yol ağı üzerinde daha esnek ve gerçekçi bir biçimde ele alınmasını sağlamıştır. Bu sayede rota yapısı, nokta merkezli bir optimizasyondan alan temelli bir yaklaşıma dönüştürülmüştür.

Harita tabanlı görselleştirmeler, alan sınırına temas noktalarının rota yapısını doğrudan etkilediğini ve TSPN mantığının görsel olarak da açık biçimde gözlemlenebildiğini göstermiştir. Alan temelli ziyaret yaklaşımı, klasik TSP çözümlerine kıyasla daha uygulanabilir bir rota modeli sunmakta olup, bu çalışmada geliştirilen yöntem TSPN probleminin gerçek yol ağları üzerinde ele alınmasına yönelik temel bir çerçeve ortaya koymaktadır.

Github Linki: https://github.com/BLM5026-AI-in-Computer-Games/hw5-metehansozenli