Bilgisayar Oyunlarında Yapay Zeka

Homework Series: "Progressive TSP Research"

Assignment 4 – AI Technique Integration

Ödev Raporu

Şevval Uzar – 25435004013

1. Giriş

Bu çalışmada, klasik TSP problemi sürekli bölgeler kullanılarak TSPN problemine genişletilmiş ve elde edilen problem üç farklı çözüm yaklaşımı ile ele alınmıştır. Yaklaşımlar, çözüm kalitesi ve hesaplama performansı açısından karşılaştırılarak değerlendirilmiştir.

Deneylerin tekrarlanabilirliğini sağlamak amacıyla tüm deneylerde seed = 42 kullanılmıştır.

2. Problem Tanımı

Bu ödev kapsamında problem aşağıdaki şekilde tanımlanmıştır:

- Her şehir sabit bir nokta yerine daire olarak modellenmiştir.
- Her bölgenin merkez koordinatı ve belirli bir yarıçapı bulunmaktadır.
- Amaç her bölgenin içinden bir nokta seçmek ve bu noktalar arasındaki toplam tur uzunluğunu en aza indirmektir.

Çalışmada Türkiye'den seçilen 10 şehir kullanılmıştır. Şehirler haritadaki koordinatları ile temsil edilmiş ve her şehir için dairesel bir komşuluk bölgesi tanımlanmıştır.

3. Kullanılan Şehirler

Deneylerde kullanılan şehirler aşağıda listelenmiştir:

- İstanbul, Ankara, İzmir, Bursa, Antalya, Adana, Konya, Gaziantep, Kayseri, Samsun

Her şehir, merkez koordinatı etrafında tanımlanan dairesel bir bölge ile temsil edilmiştir.

4. Yöntemler

4.1 Nearest Neighbour

Bu yöntemde önce şehirlerin ziyaret sırası belirlenmiş ve her bölgenin içinden rastgele bir nokta seçilerek tur oluşturulmuştur. Hesaplama maliyetinin düşük olması ve kolay uygulanabilirliği sayesinde hızlı sonuçlar üretmektedir. Ancak bölge içindeki nokta seçimi optimize edilmediği için sürekli yapı yeterince dikkate alınmamakta ve bu durum genellikle optimal çözümlerden uzak, daha uzun turlar elde edilmesine yol açmaktadır.

4.2 OR-Tools

Bu yaklaşımda, şehirlerin ziyaret sırası belirlendikten sonra her bölgeden rastgele bir nokta seçilerek tur oluşturulmuştur. Yöntem, düşük hesaplama maliyeti sayesinde hızlı ve pratik sonuçlar

sunmaktadır. Ancak nokta seçimi optimize edilmediği için bölgenin sürekli yapısı göz ardı edilmekte ve bu durum çoğu zaman optimalden uzak, daha uzun turların ortaya çıkmasına neden olmaktadır.

4.3 Genetik Algoritma

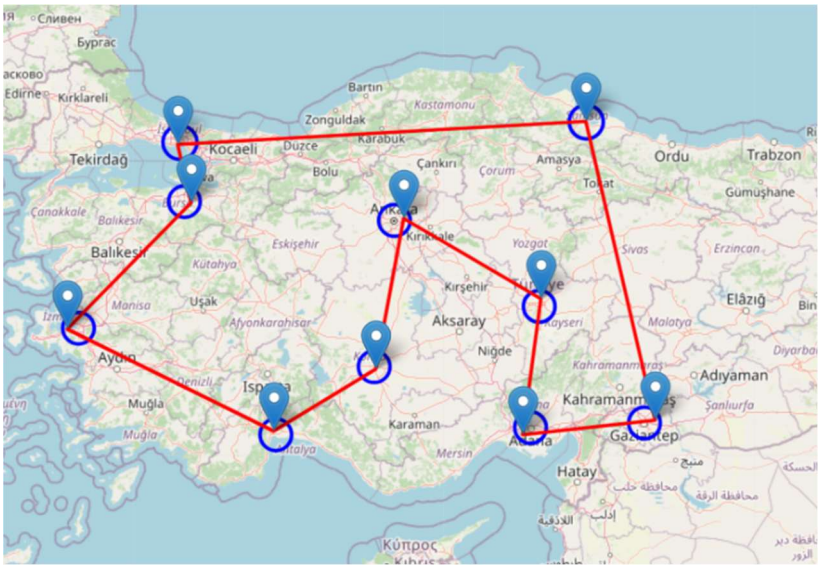
Şehirlerin ziyaret sırası bireyler üzerinden temsil edilmiş tur uzunluğu ise her bölgeden rastgele seçilen noktalarla hesaplanmıştır. Genetic Algoritma, nesiller ilerledikçe çözüm kalitesini artırabilme potansiyeline sahip olmakla birlikte, yüksek hesaplama maliyeti ve rastgele süreçlerden kaynaklanan sonuç tutarsızlığı yöntemin başlıca sınırlamalarıdır.

5. Sonuç

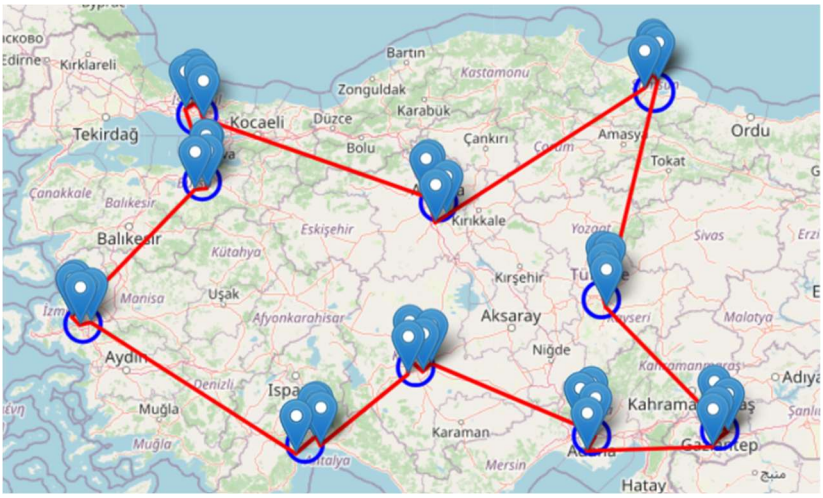
Sonuçlar incelendiğinde sezgisel yöntemin basit yapısı nedeniyle en uzun tur uzunluklarını ürettiği görülmektedir. OR-Tools yaklaşımı örnekleme sayesinde daha kısa ve daha kaliteli çözümler sunmuş ancak performansı örnekleme sayısına bağlı kalmıştır. Genetik algoritma ise genellikle en kısa veya OR-Tools ile benzer turlar üretmiş buna karşılık daha yüksek hesaplama süresi gerektirmiştir.

6. Harita

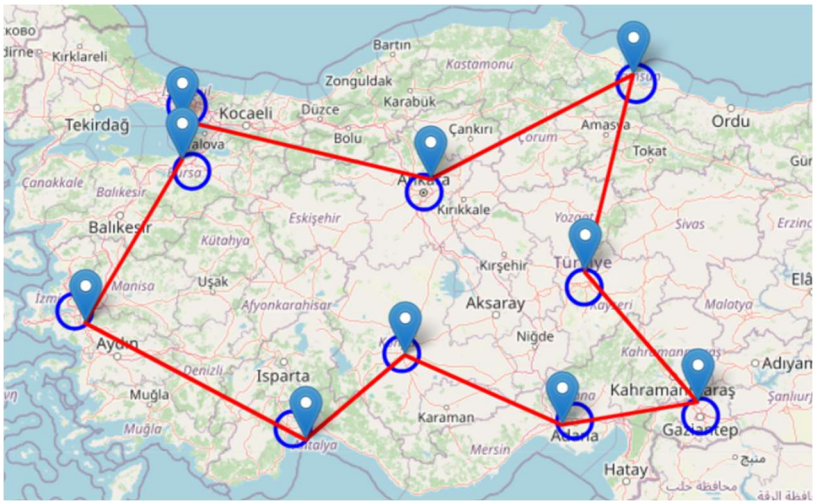
6.1 Heuristic



6.2 OR-Tools



6.3 Genetik Algoritma



7. Scripts

```
import math
import random
import folium

from ortools.constraint_solver import routing_enums_pb2
from ortools.constraint_solver import pywrapcp
from google.colab import files

SEED = 42
random.seed(SEED)

def euclid(a, b):
    return math.hypot(a[0] - b[0], a[1] - b[1])

def tour_length(points):
    return sum(
        euclid(points[i], points[(i + 1) % len(points)])
        for i in range(len(points))
    )


class Region:
    def __init__(self, name, center, radius):
        self.name = name
        self.center = center
        self.radius = radius

    def random_point(self):
        angle = random.uniform(0, 2 * math.pi)
        r = self.radius * math.sqrt(random.random())
        return (
            self.center[0] + r * math.cos(angle),
            self.center[1] + r * math.sin(angle)
        )


cities = [
    ("Istanbul", 41.0082, 28.9784),
    ("Ankara", 39.9334, 32.8597),
    ("Izmir", 38.4237, 27.1428),
    ("Bursa", 40.1950, 29.0600),
    ("Antalya", 36.8969, 30.7133),
    ("Adana", 37.0000, 35.3213),
    ("Konya", 37.8746, 32.4932),
    ("Gaziantep", 37.0662, 37.3833),
    ("Kayseri", 38.7312, 35.4787),
    ("Samsun", 41.2867, 36.3300),
]

regions = [
    Region(name, (lat, lon), radius=0.25)
    for name, lat, lon in cities
]

# 1.) Heuristic Solver
def nearest_neighbour(coords):
    unvisited = set(range(len(coords)))
    tour = [0]
    unvisited.remove(0)

    while unvisited:
        last = tour[-1]
        nxt = min(unvisited, key=lambda i: euclid(coords[last], coords[i]))
        tour.append(nxt)
        unvisited.remove(nxt)

    return tour

def heuristic_tspn(regions):
    centers = [r.center for r in regions]
    order = nearest_neighbour(centers)
    return [regions[i].random_point() for i in order]
```

```python
# Assignment 5 – TSP with Neighborhoods
# Şevval Uzar

import math
import random
import folium

from ortools.constraint_solver import routing_enums_pb2
from ortools.constraint_solver import pywrapcp
from google.colab import files

SEED = 42
random.seed(SEED)

def euclid(a, b):
    return math.hypot(a[0] - b[0], a[1] - b[1])

def tour_length(points):
    return sum(
        euclid(points[i], points[(i + 1) % len(points)])
        for i in range(len(points))
    )


class Region:
    def __init__(self, name, center, radius):
        self.name = name
        self.center = center
        self.radius = radius

    def random_point(self):
        angle = random.uniform(0, 2 * math.pi)
        r = self.radius * math.sqrt(random.random())
        return (
            self.center[0] + r * math.cos(angle),
            self.center[1] + r * math.sin(angle)
        )


cities = [
    ("Istanbul", 41.0082, 28.9784),
    ("Ankara", 39.9334, 32.8597),
    ("Izmir", 38.4237, 27.1428),
    ("Bursa", 40.1950, 29.0600),
    ("Antalya", 36.8969, 30.7133),
    ("Adana", 37.0000, 35.3213),
    ("Konya", 37.8746, 32.4932),
    ("Gaziantep", 37.0662, 37.3833),
    ("Kayseri", 38.7312, 35.4787),
    ("Samsun", 41.2867, 36.3300),
]

regions = [
    Region(name, (lat, lon), radius=0.25)
    for name, lat, lon in cities
]

# 1.) Heuristic Solver
def nearest_neighbour(coords):
    unvisited = set(range(len(coords)))
    tour = [0]
    unvisited.remove(0)

    while unvisited:
        last = tour[-1]
        nxt = min(unvisited, key=lambda i: euclid(coords[last], coords[i]))
        tour.append(nxt)
        unvisited.remove(nxt)

    return tour

def heuristic_tspn(regions):
    centers = [r.center for r in regions]
    order = nearest_neighbour(centers)
    return [regions[i].random_point() for i in order]
```

```python
# 2) OR-Tools (Sampling Approximation)
def ortools_tspn(regions, samples=4):
    sampled_points = []
    for r in regions:
        for _ in range(samples):
            sampled_points.append(r.random_point())

    n = len(sampled_points)
    dist = [
        [int(euclid(sampled_points[i], sampled_points[j]) * 1000)
         for j in range(n)]
        for i in range(n)
    ]

    manager = pywrapcp.RoutingIndexManager(n, 1, 0)
    routing = pywrapcp.RoutingModel(manager)

    def dist_cb(i, j):
        return dist[manager.IndexToNode(i)][manager.IndexToNode(j)]

    transit = routing.RegisterTransitCallback(dist_cb)
    routing.SetArcCostEvaluatorOfAllVehicles(transit)

    params = pywrapcp.DefaultRoutingSearchParameters()
    params.first_solution_strategy = routing_enums_pb2.FirstSolutionStrategy.PATH_CHEAPEST_ARC
    params.time_limit.seconds = 2

    sol = routing.SolveWithParameters(params)
    if sol is None:
        return []

    idx = routing.Start(0)
    tour = []
    while not routing.IsEnd(idx):
        tour.append(sampled_points[manager.IndexToNode(idx)])
        idx = sol.Value(routing.NextVar(idx))

    return tour

# 3.) Genetic Algorithm (AI Technique)
def genetic_tspn(regions, pop=80, gen=250, mut=0.2):
    n = len(regions)

    def individual():
        p = list(range(n))
        random.shuffle(p)
        return p

    def fitness(order):
        pts = [regions[i].random_point() for i in order]
        return tour_length(pts)

    population = [individual() for _ in range(pop)]

    for _ in range(gen):
        population.sort(key=fitness)
        elite = population[:int(0.2 * pop)]
        new_pop = elite[:]

        while len(new_pop) < pop:
            p1, p2 = random.sample(elite, 2)
            a, b = sorted(random.sample(range(n), 2))
            child = [-1] * n
            child[a:b] = p1[a:b]
            fill = [x for x in p2 if x not in child]

            k = 0
            for i in range(n):
                if child[i] == -1:
                    child[i] = fill[k]
                    k += 1
```

```
            if random.random() < mut:
                i, j = random.sample(range(n), 2)
                child[i], child[j] = child[j], child[i]

            new_pop.append(child)

        population = new_pop

    best = min(population, key=fitness)
    return [regions[i].random_point() for i in best]

def visualize(regions, tour, filename):
    m = folium.Map(location=[39, 35], zoom_start=6)

    for r in regions:
        folium.Circle(
            location=r.center,
            radius=r.radius * 100000,
            color="blue",
            fill=False,
            tooltip=r.name
        ).add_to(m)

    for p in tour:
        folium.Marker(location=p).add_to(m)

    folium.PolyLine(tour + [tour[0]], color="red").add_to(m)
    m.save(filename)

heuristic_pts = heuristic_tspn(regions)
ortools_pts   = ortools_tspn(regions)
ga_pts        = genetic_tspn(regions)

visualize(regions, heuristic_pts, "heuristic_turkey.html")
visualize(regions, ortools_pts, "ortools_turkey.html")
visualize(regions, ga_pts, "ga_turkey.html")

files.download("heuristic_turkey.html")
files.download("ortools_turkey.html")
files.download("ga_turkey.html")
```