Bilgisayar Oyunlarında Yapay Zekâ Ödev-4 Raporu

Metehan SÖZENLİ 25435004037

1. Giriş

Bu ödevde, Gezgin Satıcı Problemi için bir önceki ödevde uygulanan Nearest Neighbor (En Yakın Komşu) ve Google OR-Tools kütüphanesinin sunduğu TSP çözücüsü yaklaşımlarına ek olarak Genetik Algoritma yöntemi de sisteme dahil edilmiştir. Bu üç yaklaşım, araştırma odaklı bir deney düzeni kapsamında karşılaştırılmıştır. Ödevin temel amacı; söz konusu yöntemleri tur uzunluğu ve çalışma süresi açısından incelemek, en az 30 farklı problem topolojisi üzerinde elde edilen sonuçları grafiksel gösterimler ve istatistiksel analizler yardımıyla değerlendirmektir.

2. Yöntem

2.1. Test Verilerinin Oluşturulması

create_random_euclidean_graph fonksiyonu, iki boyutlu bir düzlemde rastgele konumlandırılmış düğümlerden oluşan tam bir graf üretmektedir. Her düğümün koordinatları belirlenen aralıkta rastgele seçilmekte, her düğüm çifti arasındaki kenar ağırlığı ise Öklid mesafesi üzerinden hesaplanmaktadır.

```python
def create_random_euclidean_graph(n: int, seed: int, area=(0.0, 100.0)) -> Tuple[Any, List[int]]:
    """Create complete Euclidean graph on n random points."""
    random.seed(seed)
    np.random.seed(seed)

    G = nx.Graph()

    # Generate random points
    for i in range(n):
        x = random.uniform(*area)
        y = random.uniform(*area)
        G.add_node(i, x=x, y=y, pos=(x, y))

    # Complete graph with Euclidean distances
    for i in range(n):
        for j in range(i+1, n):
            xi, yi = G.nodes[i]['x'], G.nodes[i]['y']
            xj, yj = G.nodes[j]['x'], G.nodes[j]['y']
            distance = float(np.hypot(xi - xj, yi - yj))
            G.add_edge(i, j, length=distance)

    return G, list(range(n))
```

2.2. Nearest Neighbor Çözücüsü

NearestNeighborSolver sınıfı, En Yakın Komşu heuristiğini doğrudan uygulamaktadır. Algoritma başlangıç düğümünden çıkarak her adımdaki en yakın komşuyu seçer ve bu işlemi tüm düğümler ziyaret edilene kadar sürdürür. Tur tamamlandığında tekrar başlangıç noktasına dönülür ve tur uzunluğu kenar ağırlıkları toplanarak hesaplanır. Bu yöntem hesaplama açısından son derece hızlıdır ancak ürettiği çözümler her zaman optimum olmak zorunda değildir.

```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
```

2.3. OR-Tools TSP Çözücüsü

ORToolsSolver sınıfı, Google OR-Tools altyapısını kullanarak TSP çözümü üretmektedir. *RoutingIndexManager* üzerinden her düğüme bir indeks atanır, mesafe fonksiyonu bir callback ile doğrudan grafın kenar ağırlıklarından alınır ve çözüm için *PATH_CHEAPEST_ARC* başlangıç stratejisi ile *GUIDED_LOCAL_SEARCH* yerel arama yöntemi kullanılır. Hesaplama on saniyelik bir süre limiti ile sınırlandırılmıştır. Bu yöntem özellikle çözüm kalitesi bakımından güçlüdür ancak NN'ye kıyasla çok daha maliyetlidir.

```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
```

2.4. Genetik Algoritma Çözücüsü

GeneticAlgorithmSolver sınıfı, TSP problemi için popülasyon tabanlı bir meta-sezgisel yaklaşım uygular ve turu permütasyon (sıralama) temsili ile kodlar. Bu çözümde bir birey (individual), başlangıç düğümü hariç tüm şehirlerin bir kez ziyaret edildiği bir sıralamadır. Nihai tur, raporda da izlenebileceği gibi, şu biçimde kurulmaktadır:

[start_node] + birey_permütasyonu + [start_node]. Amaç fonksiyonu, bu turun kenar ağırlıklarının toplamıdır ve minimizasyon yapılır (daha kısa tur = daha iyi fitness).

Algoritmanın akışı aşağıdaki temel bileşenlerden oluşur:

- **Başlatma (Initialization):**
  Popülasyon, başlangıç düğümü dışındaki şehirlerin rastgele karıştırılmasıyla üretilen permütasyonlardan oluşturulur. Böylece farklı başlangıç çözümleriyle arama uzayı geniş biçimde taranmaya başlanır.
- **Uygunluk (Fitness) Değerlendirmesi:**
  Her birey için tur uzunluğu hesaplanır. Bu sınıfta fitness, doğrudan tur uzunluğu olarak tanımlandığı için GA, klasik "maksimizasyon" yerine minimum tur uzunluğunu arar. Her jenerasyonda en iyi fitness değeri *best_fitness_history* listesine kaydedilerek yakınsama davranışı izlenebilir.
- **Seçim (Selection) – Tournament Selection:**
  Yeni nesli üretmek için ebeveynler turnuva seçimi ile belirlenir. Popülasyondan rastgele seçilen *tournament_size* adet birey arasında en kısa tura sahip olan birey "kazanan" seçilir. Bu yöntem, iyi çözümlere seçilim avantajı sağlarken çeşitliliği de tamamen yok etmez.
- **Çaprazlama (Crossover) – Order Crossover (OX):**
  crossover_rate olasılığıyla OX uygulanır. İki kesim noktası arasında bir ebeveynin parçası çocuğa kopyalanır; kalan pozisyonlar diğer ebeveynden göreli sıra korunarak doldurulur. Bu yaklaşım, TSP'de kritik olan "şehirlerin tekrarsız ziyaret edilmesi" kısıtını permütasyon yapısını bozmadan korur.

- **Mutasyon (Mutation) – Swap Mutation:**

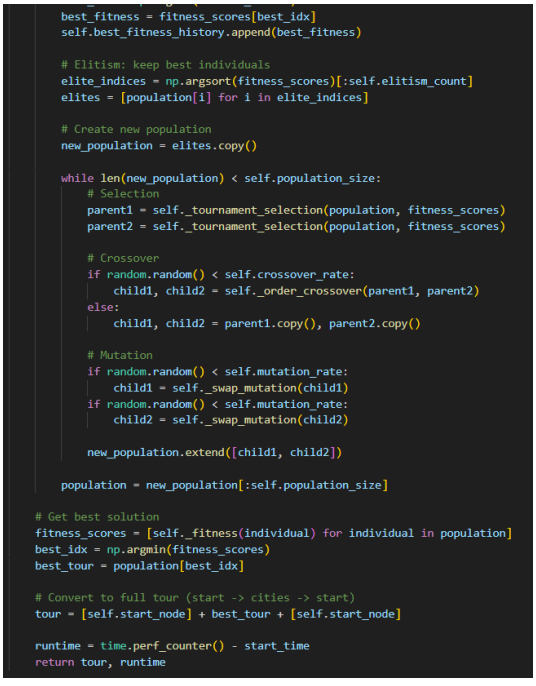
  mutation_rate olasılığıyla iki rastgele şehrin yeri değiştirilir. Swap mutasyonu, permütasyon temsiline uygun biçimde düşük maliyetli bir çeşitlilik mekanizması sağlar ve yerel minimumlara sıkışmayı azaltmaya yardımcı olur.

- **Elitizm (Elitism):**

  Her jenerasyonda en iyi *elitism_count* birey doğrudan yeni popülasyona aktarılır. Bu sayede o ana kadar bulunan en iyi çözümlerin kaybolması engellenir ve çözüm kalitesi sabit biçimde korunmaya çalışılır.

Bu yaklaşım, Nearest Neighbor'a kıyasla daha yüksek hesaplama maliyetiyle çalışsa da rastgelelik ve evrimsel operatörler sayesinde genellikle daha iyi kaliteye yakın çözümler üretebilir. OR-Tools ise çoğu durumda daha güçlü çözüm kalitesi sağlarken, GA'nın avantajı parametrik esneklik (popülasyon büyüklüğü, jenerasyon sayısı, çaprazlama/mutasyon oranları) ve yakınsama davranışının fitness geçmişi ile izlenebilmesidir.

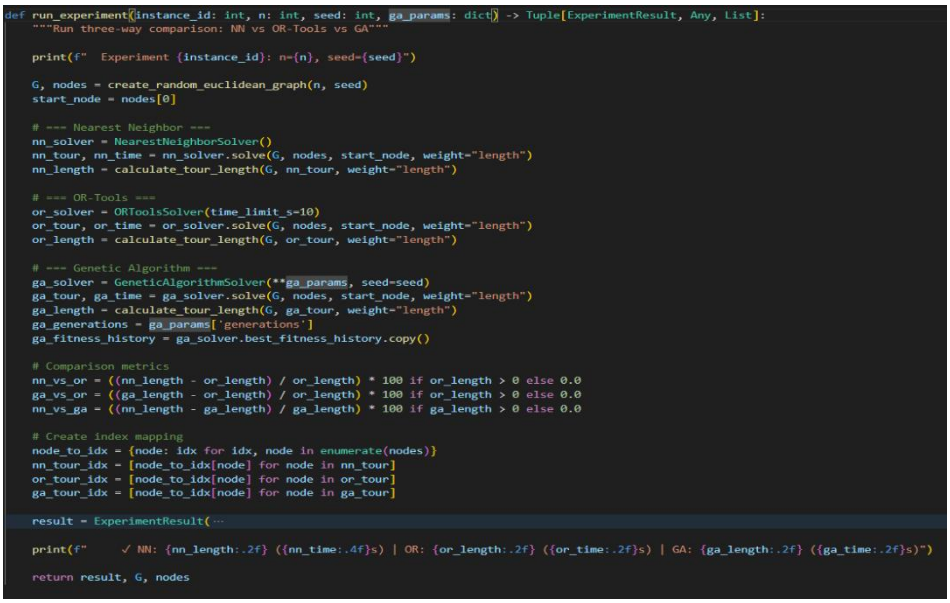
```python
class GeneticAlgorithmSolver:
    """Genetic Algorithm TSP solver"""
    def __init__(self,
                 population_size: int = 100,
                 generations: int = 200,
                 crossover_rate: float = 0.8,
                 mutation_rate: float = 0.2,
                 tournament_size: 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)
```



```python
            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
```

2.5. Deney Tasarımı

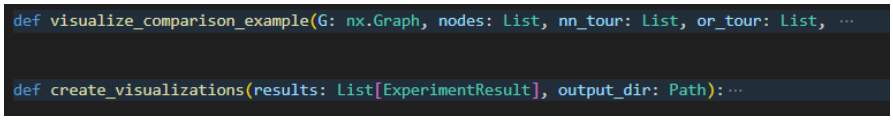
Her deney örneği ***run_experiment*** fonksiyonu ile yürütülmüştür. Bu fonksiyon, önce verilen seed ile rastgele bir Öklidyen graf üretir, ardından aynı grafik üzerinde hem *Nearest Neighbor* hem *OR-Tools* hem de *Genetik Algoritma* çözücülerini çalıştırır. Her üç algoritmanın ürettiği tur, toplam tur uzunluğu ve çalışma süresi hesaplanır. Tur uzunlukları arasındaki fark kullanılarak ayrıca bir kalite yüzdesi elde edilir. Tüm bu bilgiler bir *ExperimentResult* nesnesi içinde saklanarak ana programa geri gönderilir ve 30 farklı topoloji için aynı süreç tekrarlanır.



```python
def run_experiment(instance_id: int, n: int, seed: int, ga_params: dict) -> Tuple[ExperimentResult, Any, List]:
    """Run three-way comparison: NN vs OR-Tools vs GA"""

    print(f"  Experiment {instance_id}: n={n}, seed={seed}")

    G, nodes = create_random_euclidean_graph(n, seed)
    start_node = nodes[0]

    # === Nearest Neighbor ===
    nn_solver = NearestNeighborSolver()
    nn_tour, nn_time = nn_solver.solve(G, nodes, start_node, weight="length")
    nn_length = calculate_tour_length(G, nn_tour, weight="length")

    # === OR-Tools ===
    or_solver = ORToolsSolver(time_limit_s=10)
    or_tour, or_time = or_solver.solve(G, nodes, start_node, weight="length")
    or_length = calculate_tour_length(G, or_tour, weight="length")

    # === Genetic Algorithm ===
    ga_solver = GeneticAlgorithmSolver(**ga_params, seed=seed)
    ga_tour, ga_time = ga_solver.solve(G, nodes, start_node, weight="length")
    ga_length = calculate_tour_length(G, ga_tour, weight="length")
    ga_generations = ga_params['generations']
    ga_fitness_history = ga_solver.best_fitness_history.copy()

    # Comparison metrics
    nn_vs_or = ((nn_length - or_length) / or_length) * 100 if or_length > 0 else 0.0
    ga_vs_or = ((ga_length - or_length) / or_length) * 100 if or_length > 0 else 0.0
    nn_vs_ga = ((nn_length - ga_length) / ga_length) * 100 if ga_length > 0 else 0.0

    # Create index mapping
    node_to_idx = {node: idx for idx, node in enumerate(nodes)}
    nn_tour_idx = [node_to_idx[node] for node in nn_tour]
    or_tour_idx = [node_to_idx[node] for node in or_tour]
    ga_tour_idx = [node_to_idx[node] for node in ga_tour]

    result = ExperimentResult(...

    print(f"     ✓ NN: {nn_length:.2f} ({nn_time:.4f}s) | OR: {or_length:.2f} ({or_time:.2f}s) | GA: {ga_length:.2f} ({ga_time:.2f}s)")

    return result, G, nodes
```

2.6. Görselleştirme Fonksiyonları

Kodda yer alan görselleştirme fonksiyonları, deney sonuçlarını hem toplu hem de örnek bazında anlaşılır hâle getirmek için kullanılmaktadır. ***create_visualizations*** fonksiyonu, NN, OR-Tools ve Genetik Algoritma için tur uzunluklarını, kalite farklarını ve çalışma sürelerini karşılaştırmalı olarak sunan temel grafikler üretir. ***visualize_comparison_example*** fonksiyonu ise tek bir topoloji üzerinde üç algoritmanın rotalarını yan yana göstererek çözüm davranışlarını görsel olarak karşılaştırır. Bu iki fonksiyon birlikte, genel performans eğilimlerini ve örnek bazlı çözüm yapılarını açık biçimde ortaya koymaktadır.



```python
def visualize_comparison_example(G: nx.Graph, nodes: List, nn_tour: List, or_tour: List, ...

def create_visualizations(results: List[ExperimentResult], output_dir: Path):...
```

2.7. Ana Program Akışı

main fonksiyonu, deneylerin genel akışını yöneten bölümdür. Kullanıcıdan düğüm sayısı, deney adedi ve başlangıç seed'i gibi ayarları alır; ardından her örnek için *run_experiment* fonksiyonunu çağırarak tüm sonuçları bir liste içinde toplar. Deneyler tamamlandıktan sonra sonuçlar bir DataFrame'e dönüştürülür ve çözüm kalitesi, zaman–kalite dengesi ve kalite farkı gibi analizleri içeren tüm grafikler oluşturulur. Seçilen bir örnek topoloji için *NN*, *GA* ve *OR-Tools* rotalarını yan yana gösteren görsel de yine *main* içinde üretilir.

```python
def main():
    parser = argparse.ArgumentParser(description="Task 3: Comprehensive TSP Solver Comparison")
    parser.add_argument("--n", type=int, default=None, help="Number of nodes per instance (single value)")
    parser.add_argument("--n-values", type=str, default=None, help="Comma-separated n values for sweep (e.g., '10,15,20,25,30')")
    parser.add_argument("--instances", type=int, default=30, help="Number of test instances per n value (≥30 recommended)")
    parser.add_argument("--seed-start", type=int, default=100, help="Starting seed for reproducibility")
    parser.add_argument("--output-dir", type=str, default="results", help="Output directory")
    args = parser.parse_args()

    # Determine n values to test
    if args.n_values:
        n_values = [int(x.strip()) for x in args.n_values.split(',')]
    elif args.n:
        n_values = [args.n]
    else:
        n_values = [20]  # Default

    output_dir = Path(args.output_dir)
    output_dir.mkdir(exist_ok=True, parents=True)

    print("\nTask 3: TSP Solver Comparison - Comprehensive Analysis")
    print("=" * 70)
    print("Configuration:")
    print(f"  Node values       : {n_values}")
    print(f"  Instances per n   : {args.instances}")
    print(f"  Graph type        : Random Euclidean")
    print(f"  Starting seed     : {args.seed_start}")
    print(f"  Output directory  : {output_dir}")
    print()

    # Run experiments for each n value
    all_results = []
    example_data = None

    for n in n_values:
        print(f"\n{'='*70}")
        print(f"Running experiments for n={n} nodes...")
        print(f"{'='*70}")
        results_for_n = []
        seed_offset = 0

        for i in range(args.instances):
            instance_id = len(all_results) + i + 1
            seed = args.seed_start + seed_offset + i
            try:
                result, G, nodes = run_experiment(instance_id, n, seed)
                results_for_n.append(result)
                all_results.append(result)

                # Save an instance for example visualization (from first n value)
                if example_data is None and i == min(15, args.instances - 1):
                    example_data = (G, nodes, result)

            except Exception as e:
                print(f"      ✗ Failed: {e}")

        print(f"\nCompleted {len(results_for_n)} experiments for n={n}.")
        seed_offset += args.instances

    if not all_results:
        print("\n✗ No successful experiments!")
        return

    print(f"\n{'='*70}")
    print(f"Completed {len(all_results)} total experiments across {len(n_values)} n value(s).")
    print(f"{'='*70}")

    # Convert results to DataFrame for analysis/plots
    df = pd.DataFrame([r.to_dict() for r in all_results])

    # Generate standard visualizations (1-4) only for FIRST n value to avoid clutter
    print("\nGenerating visualizations...")
    first_n_results = [r for r in all_results if r.n_nodes == n_values[0]]

# Convert results to DataFrame for analysis/plots
df = pd.DataFrame([r.to_dict() for r in all_results])

# Generate standard visualizations (1-4) only for FIRST n value to avoid clutter
print("\nGenerating visualizations...")
first_n_results = [r for r in all_results if r.n_nodes == n_values[0]]
if len(n_values) == 1:
    # Single n value: use all results
    create_visualizations(all_results, output_dir)
else:
    # Multiple n values: use only first n for standard plots
    print(f"  (Using only n={n_values[0]} data for plots 1-4 to avoid clutter)")
    create_visualizations(first_n_results, output_dir)

# Generate n-sweep trend visualization if multiple n values
if len(n_values) > 1:
    print("\nGenerating n-value trend analysis...")
    create_n_sweep_visualization(all_results, n_values, output_dir)

# Generate example comparison visualization (from first n value)
if example_data:
    print("\nGenerating example comparison visualization...")
    G, nodes, result = example_data
    # Convert index-based tours back to actual node IDs
    nn_tour = [nodes[i] for i in result.nn_tour]
    or_tour = [nodes[i] for i in result.or_tour]
    visualize_comparison_example(···
```

3. Deneysel Sonuçlar ve Grafiksel Analiz

3.1. Deney Kurulumu ve Parametreler

Deneyler, rastgele üretilmiş tam bağlantılı Öklidyen grafikler üzerinde gerçekleştirilmiştir. Her bir problem örneğinde 20 şehirden (düğümden) oluşan bir TSP yapısı kullanılmış, şehirlerin koordinatları iki boyutlu düzlemde rastgele seçilmiş ve kenar ağırlıkları şehirler arasındaki Öklidyen mesafe olarak tanımlanmıştır. Toplamda 30 bağımsız topoloji üretilmiş ve her örnek için tüm algoritmalar aynı başlangıç düğümünden başlatılarak adil bir karşılaştırma sağlanmıştır.

Nearest Neighbor algoritması, herhangi bir ayarlanabilir parametre içermeden doğrudan uygulanmıştır.

OR-Tools tabanlı çözücüde, başlangıç çözümü için PATH_CHEAPEST_ARC stratejisi kullanılmış, çözüm kalitesini artırmak amacıyla GUIDED_LOCAL_SEARCH yerel arama yöntemi etkinleştirilmiş ve her bir örnek için çözüm süresi 10 saniyelik zaman limiti ile sınırlandırılmıştır.

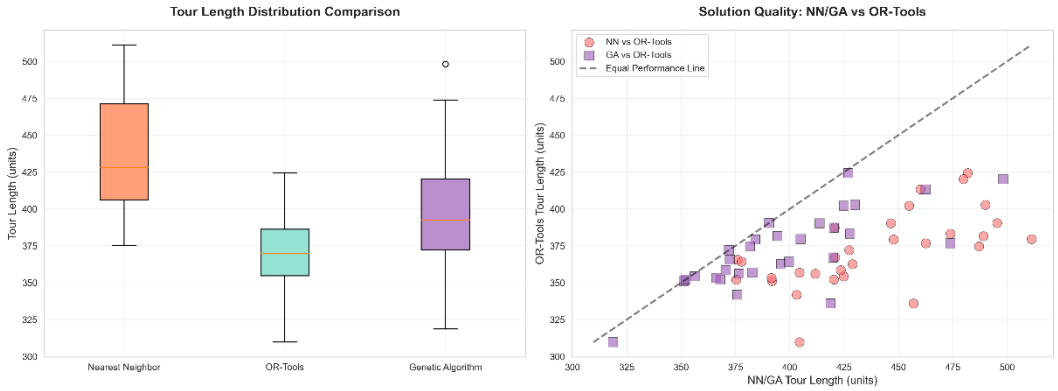
Genetik Algoritma çözümünde ise deneyler aşağıdaki parametreler ile yürütülmüştür:

- Popülasyon büyüklüğü: 300
- Jenerasyon sayısı: 1000
- Çaprazlama oranı (Order Crossover): 0.8 (varsayılan)
- Mutasyon oranı (Swap Mutation): 0.05
- Seçim yöntemi: Turnuva seçimi (turnuva boyutu = 5)
- Elitizm: En iyi 2 birey korunmuştur

3.2. Deneysel Sonuçlar

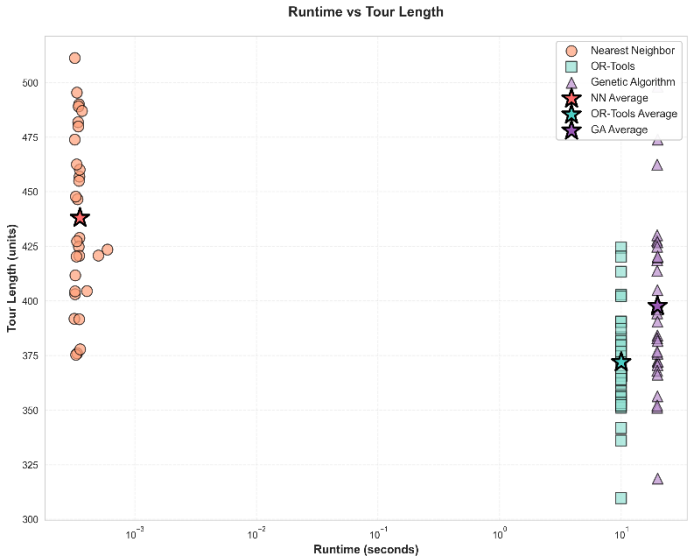
Şekil 3.2.1'deki kutu ve saçılım grafikleri, Nearest Neighbor, Genetik Algoritma ve OR-Tools'un performans dağılımlarını karşılaştırmalı olarak göstermektedir. Kutu grafiği, OR-Tools'un tur uzunluklarının diğer iki yönteme göre daha düşük seviyede ve daha dar bir aralıkta toplandığını ortaya koymaktadır. Bu durum, OR-Tools'un hem daha kısa turlar ürettiğini hem de sonuçlarda yüksek tutarlılık sağladığını işaret eder. Genetik Algoritma, NN'ye göre belirgin bir iyileşme gösterse de genel olarak OR-Tools'un gerisinde kalmıştır.

Saçılım grafiğindeki verilerin büyük kısmının eşitlik çizgisinin altında olması da OR-Tools'un sistematik başarısını doğrular niteliktedir.



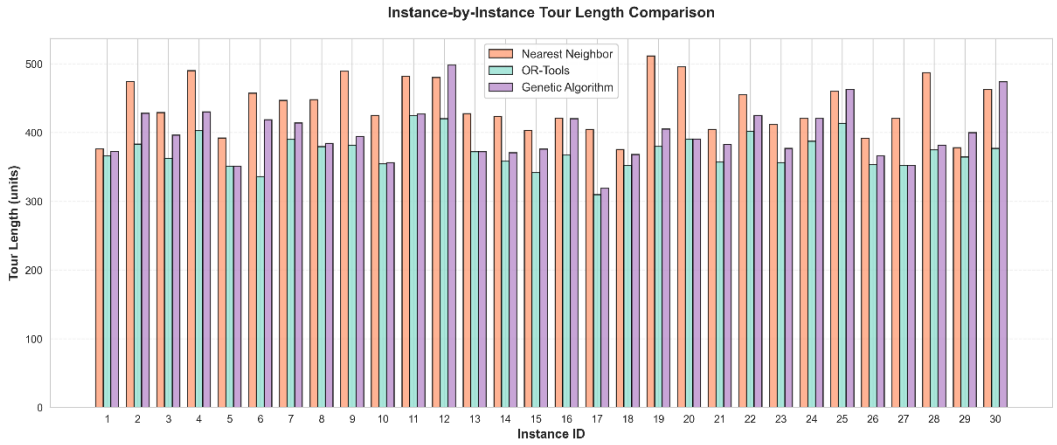
Şekil 3.2.1: Çözüm kalite karşılaştırması.

Çalışma süresi ve tur uzunluğu ilişkisi Şekil 3.2.2'de gösterilmektedir. Nearest Neighbor, çözümleri milisaniye düzeyinde üreterek en hızlı yöntemdir. OR-Tools, saniyeler içinde çalışan ve daha uzun hesaplama süresi gerektiren bir yöntem olmasına rağmen, en kısa tur uzunluklarını üretmektedir. Genetik Algoritma ise en uzun çalışma süresine sahip yöntemdir ve çözüm kalitesi açısından NN ile OR-Tools arasında yer almaktadır. Özetle; NN hız açısından avantajlı ancak çözüm kalitesi düşüktür, GA daha yavaş çalışmasına rağmen orta seviyede kalite sunmaktadır, OR-Tools ise daha kısa sürede daha yüksek kaliteli çözümler üretmektedir.

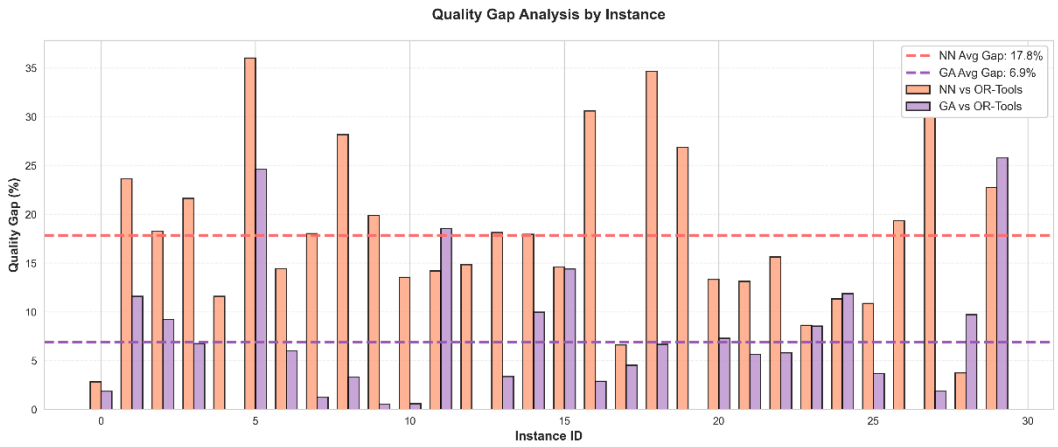


Şekil 3.2.2: Çözümlerin çalışma zamanı bakımından karşılaştırması.

Şekil 3.2.3'teki örnek bazlı karşılaştırmada, OR-Tools'un 30 örneğin neredeyse tamamında en iyi sonucu verdiği görülmektedir. GA, genellikle NN'den daha iyi performans gösterse de bazı örneklerde NN ile benzer veya daha kötü sonuçlar üretebilmektedir. Yine de genel tablo, GA'nın NN'ye göre anlamlı bir iyileşme sağladığı, ancak OR-Tools seviyesine çıkamadığı yönündedir. Bu durum Şekil 3.2.4'teki kalite farkı analiziyle de kanıtlanmıştır. OR-Tools referans alındığında, NN'nin ortalama kalite farkı %17,8 iken, GA'nın farkı %6,9 seviyesindedir. Bu veriler, GA'nın NN'den daha başarılı olduğunu ancak OR-Tools'un liderliğini koruduğunu gösterir.

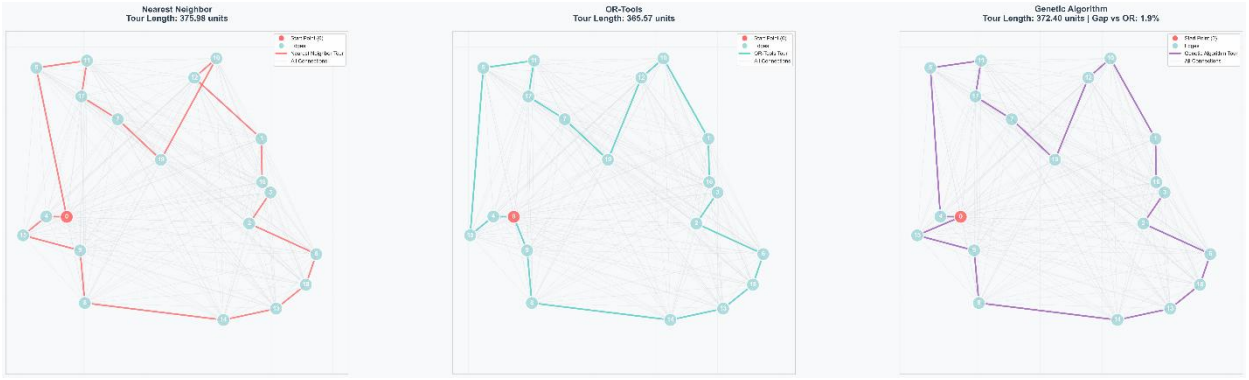


Şekil 3.2.3: Örnek bazlı tur uzunluğu karşılaştırması.



Şekil 3.2.4: Örnek bazında kalite farkı (quality gap) analizi.

Şekil 3.2.5'te ise tek bir örnek üzerindeki rota çizimleri karşılaştırılmıştır. NN 375.98, GA 372.40, OR-Tools ise 365.57 birimlik tur uzunluklarına sahiptir. GA, NN'ye göre daha düzgün bir rota çizse de OR-Tools'tan %1,9 daha uzun bir yol üretmiştir. Görsel incelemede NN'nin çapraz ve verimsiz bağlantılar kurduğu, GA'nın daha iyi ancak yine de kusurlu olduğu, OR-Tools'un ise geometrik olarak en düzgün ve kısa rotayı oluşturduğu net bir şekilde görülmektedir.



Şekil 3.2.5: Örnek tur görselleştirmesi.

4. Sonuç ve Değerlendirme

Bu çalışma kapsamında Nearest Neighbor, Genetik Algoritma ve OR-Tools tabanlı üç farklı TSP çözüm yaklaşımı deneysel olarak karşılaştırılmıştır. Elde edilen sonuçlar, OR-Tools'un çözüm kalitesi açısından açık bir üstünlük sağladığını; Genetik Algoritma'nın Nearest Neighbor'a kıyasla belirgin bir iyileşme sunmasına rağmen genel olarak OR-Tools seviyesine ulaşamadığını göstermektedir. Nearest Neighbor, çok kısa sürede çözüm üretebilmesi sayesinde zaman kısıtlı veya gerçek zamanlı uygulamalar için uygun bir yöntemdir. Genetik Algoritma ise en uzun çalışma süresine sahip olmasına rağmen, çözüm kalitesi açısından NN ile OR-Tools arasında yer almakta ve hız–kalite dengesi bakımından ara bir yaklaşım sunmaktadır. OR-Tools, Genetik Algoritma'ya kıyasla daha kısa sürede çalışarak en kısa ve en tutarlı turları üretmekte, bu yönüyle kalite odaklı uygulamalar için en uygun çözüm olarak öne çıkmaktadır. Yapılan deneysel analiz, kullanıcıların uygulama gereksinimlerine bağlı olarak hız ve çözüm kalitesi arasındaki dengeyi bilinçli biçimde değerlendirmelerine olanak tanımaktadır.

Github Linki: https://github.com/BLM5026-AI-in-Computer-Games/hw4-metehansozenli