Bilgisayar Oyunlarında Yapay Zekâ Ödev-3 Raporu

Metehan SÖZENLİ 25435004037

1. Giriş

Bu çalışmada Gezgin Satıcı Problemi için iki farklı çözüm yaklaşımı uygulanmış ve araştırma tarzı bir deney düzeni altında karşılaştırılmıştır. Birinci yaklaşım, basit ama hızlı bir heuristik olan Nearest Neighbor (En Yakın Komşu) algoritmasıdır. İkinci yaklaşım ise, Google OR-Tools kütüphanesinin sunduğu TSP çözücüsüdür. Amaç, bu iki yöntemi hem tur uzunluğu hem de çalışma süresi açısından inceleyerek en az otuz farklı problem topolojisi üzerinde karşılaştırmak ve elde edilen farkları hem grafikler hem de istatistiksel testler yardımıyla yorumlamaktır.

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. 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 de *OR-Tools* çözücülerini çalıştırır. Her iki algoritmanın ürettiği tur, toplam tur uzunluğu ve çalışma süresi hesaplanır. Tur uzunlukları arasındaki fark kullanılarak ayrıca bir kalite yüzdesi elde edilir. Bu değer, *NN*'in *OR-Tools*'a göre turu ne kadar daha uzun ürettiğini yüzde olarak gösterir. 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) -> Tuple[ExperimentResult, Any, List]:
    """Run one NN vs OR-Tools comparison and return result + graph for plotting."""

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

    # Create random Euclidean graph
    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")

    # Comparison metrics
    quality_gap = ((nn_length - or_length) / or_length) * 100 if or_length > 0 else 0.0
    speedup = (or_time / nn_time) if nn_time > 0 else 1.0

    # Create index mapping for simple IDs
    node_to_idx = {node: idx for idx, node in enumerate(nodes)}
    nn_tour_idx = [node_to_idx[node] if node in node_to_idx else -1 for node in nn_tour]
    or_tour_idx = [node_to_idx[node] if node in node_to_idx else -1 for node in or_tour]

    result = ExperimentResult(…

    print(f"    ✓ NN: {nn_length:.2f} ({nn_time:.4f}s) | OR: {or_length:.2f} ({or_time:.4f}s) | Gap: {quality_gap:.1f}%")

    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, tüm örneklerin tur uzunluklarını, algoritmaların kalite farkını, çalışma sürelerini ve örnek-bazlı karşılaştırmaları içeren dört temel grafik üretir; bunlar çözüm kalitesi kutu grafiği, NN–OR saçılım grafiği, zaman–kalite dengesi, instance-by-instance çubuk grafiği ve kalite farkı dağılımıdır. ***visualize_comparison_example*** fonksiyonu ise seçilen tek bir topoloji üzerinde NN ve OR-Tools rotalarını yan yana çizerek iki algoritmanın davranışını görsel olarak karşılaştırır. Bu iki fonksiyon birlikte hem genel performans eğilimlerini hem de bireysel bir örneğin çözüm yapısını açık ve tutarlı şekilde göstermektedir.

```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):…
```

2.5. Ana Program Akışı

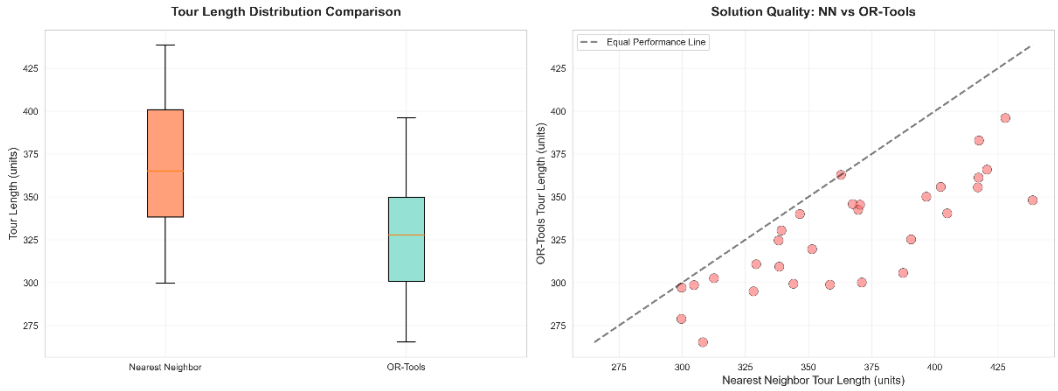
main fonksiyonu, deneylerin genel akışını yöneten bölümdür. Kullanıcıdan düğüm sayısı, deney adedi ve başlangıç seed'i gibi ayarları alır; ardından her örnek için *run_experiment* fonksiyonunu çağırarak tüm sonuçları bir liste içinde toplar. Deneyler tamamlandıktan sonra sonuçlar bir DataFrame'e dönüştürülür ve çözüm kalitesi, zaman–kalite dengesi ve

```
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(···
```

kalite farkı gibi analizleri içeren tüm grafikler oluşturulur. Seçilen bir örnek topoloji için *NN* ve *OR-Tools* rotalarını yan yana gösteren görsel de yine *main* içinde üretilir.

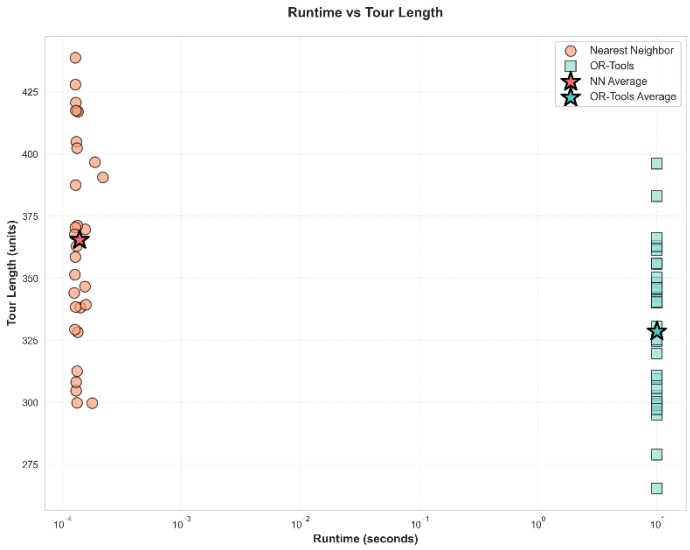
3. Sonuçlar ve Grafiksel Analiz

Şekil 3.1'de yer alan kutu grafiği ve saçılım grafiği, algoritmaların çözüm kalitesi açısından nasıl davrandığını ortaya koymaktadır. Kutu grafiğinde OR-Tools'un dağılımı NN'ye kıyasla belirgin biçimde daha aşağıdadır ve varyansı daha dardır. Bu durum OR-Tools'un çoğu örnekte daha kısa ve daha tutarlı turlar ürettiğini gösterir. Saçılım grafiğinde noktaların çoğunun eşit performans çizgisinin altında yer alması, her bir örnek bazında OR-Tools'un genel olarak daha iyi iş çıkardığını açıkça ortaya koymaktadır.



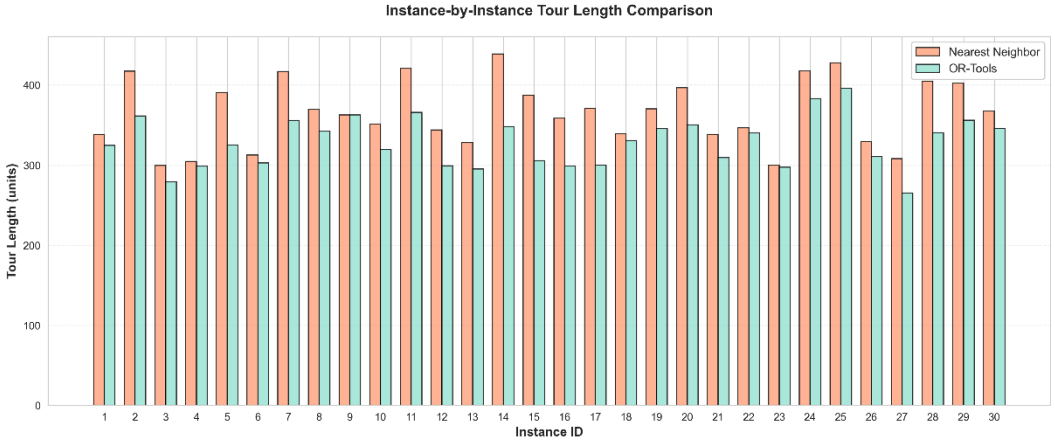
Şekil 3.1: Çözüm kalite karşılaştırması.

Çalışma süresi ile tur uzunluğu arasındaki ilişki Şekil 3.2'de karşılaştırmalı olarak gösterilmektedir. Grafikte Nearest Neighbor noktalarının neredeyse tamamının 10^{-4} saniye civarında yoğunlaştığı görülmektedir; bu durum NN'in hesaplama açısından son derece hızlı ve neredeyse anlık sonuç veren bir yöntem olduğunu göstermektedir. Buna karşılık OR-Tools çözümleri saniyeler seviyesine yayılan bir çalışma süresi gerektirmekte, ancak üretilen tur uzunlukları belirgin biçimde daha düşük bir aralıkta toplanmaktadır. Ortalama değerler de bu tabloyu desteklemekte; NN yüksek hız avantajı sağlarken çözüm kalitesi açısından tutarlı biçimde geride kalmakta, OR-Tools ise daha uzun hesaplama süresi karşılığında anlamlı derecede daha kısa turlar üretmektedir. Bu grafik, iki algoritma arasındaki hız–kalite dengesi farkını açık şekilde ortaya koymaktadır.



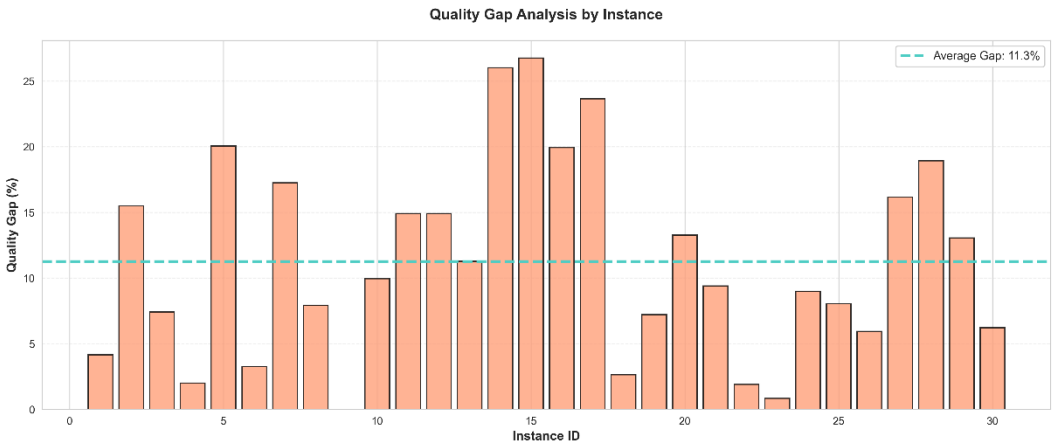
Şekil 3.2: Çözümlerin çalışma zamanı bakımından karşılaştırması.

Şekil 3.3'te her bir örnek için iki algoritmanın ürettiği tur uzunlukları yan yana gösterilmiştir. Çubuklar incelendiğinde, OR-Tools'un neredeyse tüm topolojilerde Nearest Neighbor'dan daha kısa turlar oluşturduğu açıkça görülmektedir. Bazı örneklerde fark oldukça sınırlı kalırken, belirli topolojilerde OR-Tools'un NN'ye göre yüzde yirmilere varan bir avantaj sağladığı dikkat çekmektedir. Bu gözlem Şekil 3.4'teki yüzde fark grafiğiyle de doğrulanmakta; değerlerin büyük kısmının pozitif olması NN'in çoğu durumda daha uzun turlar ürettiğini göstermektedir. Ortalama kalite farkının yaklaşık yüzde on bir olması ise



Şekil 3.3: Örnek bazlı tur uzunluğu karşılaştırması.

iki algoritma arasındaki performans farkının rastlantısal değil, sistematik bir eğilim olduğunu ortaya koymaktadır.



Şekil 3.4: Örnek bazında kalite farkı (quality gap) analizi.

Şekil 3.5'te ise seçilen bir örneğin üzerinde iki algoritmanın ürettiği rotalar yan yana gösterilmiştir. Bu örnekte Nearest Neighbor'ın tur uzunluğu yaklaşık 332.08 birim iken OR-Tools'un turu yaklaşık 263.39 birimdir. Aradaki fark yüzde yirmiye yaklaşmaktadır. Görsel incelendiğinde NN'nin çapraz geçişler içerdiği, OR-Tools'un ise noktalara daha dengeli ve geometrik olarak daha mantıklı bir düzen içinde ilerlediği görülmektedir.

Şekil 3.5: Örnek tur görselleştirmesi.

4. Sonuç ve Değerlendirme

Sonuç olarak, Nearest Neighbor ve OR-Tools tabanlı iki farklı TSP çözüm yaklaşımını kapsamlı biçimde değerlendirmiştir. Deneyler, OR-Tools'un ortalama %10 ila %12 arasında daha kaliteli çözümler ürettiğini göstermiştir. Nearest Neighbor çok hızlı çalıştığı için büyük ölçekli veya gerçek zamanlı uygulamalarda avantaj sağlayabilir; ancak ürettiği çözümler optimumdan belirgin şekilde uzaklaşabilmektedir. OR-Tools ise daha yavaş olmasına rağmen üretmiş olduğu turların kalitesi bakımından açık bir üstünlüğe sahiptir. Bu nedenle, hızın öncelikli olduğu senaryolarda NN tercih edilebilirken, kaliteye duyarlı uygulamalarda OR-Tools çok daha uygun bir çözümdür. Yapılan deneysel analiz, kullanıcıların bu iki yöntem arasında seçim yaparken zaman–kalite dengesini nasıl değerlendirmeleri gerektiğine dair somut bir rehber sunmaktadır.

Github Linki: https://github.com/BLM5026-AI-in-Computer-Games/hw3-metehansozenli