```
<routes>
    <vType id="car" accel="2.6" decel="4.5" sigma="0.5" length="5" maxSpeed="70" color="0.8,0.8,0.8"/>
    <vType id="ambulance" accel="4.0" decel="6.0" sigma="0" length="7" maxSpeed="90" vClass="emergency" color="1,0,0"/>

    <route id="start_amb_1" edges="e5"/>
    <route id="start_amb_2" edges="e10"/>
    <route id="start_amb_3" edges="e1"/>

    <vehicle id="amb_1" type="ambulance" route="start_amb_1" depart="0" />
    <vehicle id="amb_2" type="ambulance" route="start_amb_2" depart="0" />
    <vehicle id="amb_3" type="ambulance" route="start_amb_3" depart="0" />

    <flow id="bg_traffic" type="car" begin="0" end="1000" number="100" from="e6" to="e2"/>
</routes>
```

Hexagon.rou.xml

```
<configuration>
  <input>
    <net-file value="hexagon.net.xml"/>
    <route-files value="hexagon.rou.xml"/>
    <additional-files value="hospitals.add.xml ,patients.add.xml"/>
  </input>
  <time>
    <begin value="0"/>
    <end value="1000"/>
  </time>

  <output>
    <tripinfo-output value="tripinfo.xml"/>
  </output>
</configuration>
```

Hexagon.sumocfg

```
<additional>
    <poi id="P01" type="patient" color="red" layer="6" x="100" y="340"/>
    <poi id="P02" type="patient" color="red" layer="6" x="390" y="173"/>
    <poi id="P03" type="patient" color="red" layer="6" x="295" y="10"/>
</additional>
```

Patients.add.xml

```
import os
import sys
import traci
import heapq
import random
import xml.etree.ElementTree as ET
from collections import Counter

# --- GA CONFIGURATION ---
GA_CONFIG = {
    "population_size": 50,
    "generations": 30,
    "mutation_rate": 0.1,
    "crossover_rate": 0.8,
    "tournament_size": 5
}

# --- SIMULATION DATA ---
PATIENTS = {
    "P01": {"name": "Ravi Kumar", "condition": "Cardiac Arrest", "keywords": ["Cardiology"], "start_edge": "e6"},
    "P02": {"name": "Sita Devi", "condition": "Multiple Fractures", "keywords": ["Trauma Care", "Orthopedics"], "start_edge": "e1"},
    "P03": {"name": "Arjun Singh", "condition": "Severe Lacerations", "keywords": ["Emergency Care"], "start_edge": "e3_rev"},
}
HOSPITALS = {
    "H-01": {"name": "City_General", "specialties": ["General Medicine", "Emergency Care"], "available_beds": 12, "dest_edge": "e6"},
    "H-02": {"name": "Green_Heart", "specialties": ["Cardiology"], "available_beds": 5, "dest_edge": "e2"},
    "H-04": {"name": "Tumakuru_Trauma", "specialties": ["Trauma Care", "Orthopedics"], "available_beds": 3, "dest_edge": "e4"}
}
AMBULANCES = ["amb_1", "amb_2", "amb_3"]

# --- Dijkstra Pathfinding Class (UPDATED) ---
class DijkstraForSUMO:
    def __init__(self, net_file):
        self.net_file = net_file
        self.graph, self.edge_to_junctions, self.junction_pair_to_edge = {}, {}, {}
        self._build_graph()

    def _build_graph(self):
        tree = ET.parse(self.net_file)
        for edge in tree.getroot().findall('edge'):
            if edge.get('function') != 'internal':
                edge_id, from_node, to_node = edge.get('id'), edge.get('from'), edge.get('to')
                lane = edge.find('lane')
                if lane is not None:
                    speed = float(lane.get('speed'))
                    travel_time = float(lane.get('length')) / speed if speed > 0 else float('inf')
                    if from_node not in self.graph:
                        self.graph[from_node] = {}
```

```
          self.graph[from_node][to_node] = travel_time
          self.edge_to_junctions[edge_id] = (from_node, to_node)
          self.junction_pair_to_edge[(from_node, to_node)] = edge_id

  def find_shortest_path(self, start_edge, end_edge):
    if start_edge == end_edge:
      return [start_edge], 0
    if start_edge not in self.edge_to_junctions or end_edge not in self.edge_to_junctions:
      return None, float('inf')

    start_node = self.edge_to_junctions[start_edge][1]
    end_node = self.edge_to_junctions[end_edge][1]

    distances = {node: float('inf') for node in self.graph}
    distances[start_node] = 0
    previous_nodes = {node: None for node in self.graph}
    pq = [(0, start_node)]

    while pq:
      dist, current_node = heapq.heappop(pq)
      if dist > distances[current_node]:
        continue
      if current_node == end_node:
        break
      if current_node in self.graph:
        for neighbor, weight in self.graph[current_node].items():
```

```
                    distance = dist + weight
                    if distance < distances[neighbor]:
                        distances[neighbor] = distance
                        previous_nodes[neighbor] = current_node
                        heapq.heappush(pq, (distance, neighbor))

        path_nodes = []
        current = end_node
        while current is not None:
            path_nodes.insert(0, current)
            current = previous_nodes[current]

        if not path_nodes or path_nodes[0] != start_node:
            return None, float('inf')

        path_edges = [self.junction_pair_to_edge.get((path_nodes[i], path_nodes[i+1])) for i in range(len(path_nodes) - 1)]
        final_path = [start_edge] + [edge for edge in path_edges if edge]
        return final_path, distances[end_node]

    def find_shortest_path_time(self, start_edge, end_edge):
        _, time = self.find_shortest_path(start_edge, end_edge)
        return time

# --- Genetic Algorithm Functions ---
def create_chromosome():
    hospital_ids = list(HOSPITALS.keys())
    return [random.choice(hospital_ids) for _ in PATIENTS]

def calculate_fitness(chromosome, router):
    total_travel_time = 0
    penalty = 0
    available_ambulances = list(AMBULANCES)

    hospital_assignments = Counter(chromosome)
    for h_id, count in hospital_assignments.items():
        if count > HOSPITALS[h_id]["available_beds"]:
            penalty += 10000 * (count - HOSPITALS[h_id]["available_beds"])

    # This logic is complex because it has to find the best ambulance for each patient in the plan
    patients_to_assign = list(PATIENTS.keys())
    temp_ambulances = list(AMBULANCES)

    # Create a temporary assignment of ambulances to patients to calculate total time
    assignments = {} # patient_id -> ambulance_id
    for _ in range(len(patients_to_assign)):
        best_amb, best_pat, min_time = None, None, float('inf')
        for amb in temp_ambulances:
            for pat in patients_to_assign:
                time = router.find_shortest_path_time(traci.vehicle.getRoadID(amb),
PATIENTS[pat]["start_edge"])
                if time < min_time:
                    min_time, best_amb, best_pat = time, amb, pat
        if best_amb:
            assignments[best_pat] = best_amb
            patients_to_assign.remove(best_pat)
            temp_ambulances.remove(best_amb)

    for i, patient_id in enumerate(PATIENTS.keys()):
        hospital_id = chromosome[i]
        hospital_edge = HOSPITALS[hospital_id]["dest_edge"]
        patient_edge = PATIENTS[patient_id]["start_edge"]

        if patient_id in assignments:
            assigned_amb = assignments[patient_id]
            amb_edge = traci.vehicle.getRoadID(assigned_amb)

            time_to_patient = router.find_shortest_path_time(amb_edge, patient_edge)
            time_to_hospital = router.find_shortest_path_time(patient_edge, hospital_edge)
            total_travel_time += time_to_patient + time_to_hospital

    return total_travel_time + penalty

def selection(population, fitnesses):
    tournament = random.sample(list(zip(population, fitnesses)),
GA_CONFIG["tournament_size"])
    return min(tournament, key=lambda x: x[1])[0]
```

```
def crossover(parent1, parent2):
    if random.random() < GA_CONFIG["crossover_rate"]:
        point = random.randint(1, len(parent1) - 1)
        child1 = parent1[:point] + parent2[point:]
        child2 = parent2[:point] + parent1[point:]
        return child1, child2
    return parent1, parent2


def mutate(chromosome):
    for i in range(len(chromosome)):
        if random.random() < GA_CONFIG["mutation_rate"]:
            chromosome[i] = random.choice(list(HOSPITALS.keys()))
    return chromosome


# --- Main Simulation Logic ---
def run_simulation():
    sumo_cmd = [os.path.join(os.environ.get("SUMO_HOME", "."), "bin", "sumo-gui"), "-c", "hexagon.sumocfg", "--tripinfo-output", "tripinfo_results.xml"]
    traci.start(sumo_cmd)
    router = DijkstraForSUMO('hexagon.net.xml')
    plan_executed = False

    while traci.simulation.getMinExpectedNumber() > 0:
        traci.simulationStep()

        if not plan_executed and traci.simulation.getTime() >= 1:
```

```
print("--- Running Genetic Algorithm to Find Optimal Plan ---")

population = [create_chromosome() for _ in range(GA_CONFIG["population_size"])]

for gen in range(GA_CONFIG["generations"]):
    fitnesses = [calculate_fitness(chrom, router) for chrom in population]
    new_population = []
    for _ in range(GA_CONFIG["population_size"] // 2):
        parent1 = selection(population, fitnesses)
        parent2 = selection(population, fitnesses)
        child1, child2 = crossover(parent1, parent2)
        new_population.extend([mutate(child1), mutate(child2)])
    population = new_population
    print(f"Generation {gen+1}, Best Time: {min(fitnesses):.2f}s")

final_fitnesses = [calculate_fitness(chrom, router) for chrom in population]
best_plan = min(zip(population, final_fitnesses), key=lambda x: x[1])[0]

print("\n--- Optimal Plan Found! Dispatching Ambulances ---")

# Execute the best plan found by the GA
patients_to_assign = list(PATIENTS.keys())
temp_ambulances = list(AMBULANCES)

assignments = {}
for _ in range(len(patients_to_assign)):
```

```
            best_amb, best_pat, min_time = None, None, float('inf')
            for amb in temp_ambulances:
                for pat in patients_to_assign:
                    time = router.find_shortest_path_time(traci.vehicle.getRoadID(amb),
PATIENTS[pat]["start_edge"])
                    if time < min_time:
                        min_time, best_amb, best_pat = time, amb, pat
            if best_amb:
                assignments[best_pat] = best_amb
                patients_to_assign.remove(best_pat)
                temp_ambulances.remove(best_amb)

        for patient_id, amb_id in assignments.items():
            patient_index = list(PATIENTS.keys()).index(patient_id)
            hospital_id = best_plan[patient_index]

            current_amb_edge = traci.vehicle.getRoadID(amb_id)
            patient_edge = PATIENTS[patient_id]["start_edge"]
            hospital_edge = HOSPITALS[hospital_id]["dest_edge"]

            path_to_patient, _ = router.find_shortest_path(current_amb_edge, patient_edge)
            path_to_hospital, _ = router.find_shortest_path(patient_edge, hospital_edge)

            if path_to_patient and path_to_hospital:
                full_route = path_to_patient + path_to_hospital[1:] if len(path_to_hospital) > 1 else
path_to_patient
                traci.vehicle.setRoute(amb_id, full_route)
```

```
                    print(f"Dispatching '{amb_id}' to patient '{patient_id}' and then to hospital '{HOSPITALS[hospital_id]['name']}'.")
                else:
                    print(f"ERROR: Could not find a full path for '{amb_id}' to serve patient '{patient_id}'.")

            plan_executed = True

    traci.close()

if __name__ == "__main__":
    if "SUMO_HOME" not in os.environ:
        sys.exit("Please declare the environment variable 'SUMO_HOME'.")
    run_simulation()
```

traci_runner.py