AI ASSISTED CODING

END SEMESTER LAB EXAM

NAME : B. REVANTH

ENROLL NO : 2403A52041

BATCH : 03

Subset –9

Q1: Implement dynamic routing for public transit.

- Task 1: Use AI to sketch Dijkstra/A* variations with live costs.
- Task 2: Simulate and measure travel times.

Prompt:

Implement a dynamic public-transit routing system. First, outline and sketch AI-enhanced variations of Dijkstra's and A* algorithms that incorporate live, real-time cost updates (traffic, delays, congestion). Next, simulate these routing methods on sample transit networks and measure the resulting travel times, comparing performance and efficiency. Provide clear explanations, code, and results.

Code:

End_Sem_Lab > Q1 > dynamic_routing.py > ...

```python
from typing import Dict, Tuple, List, Callable, Optional
import heapq
import math
import random
import time

Node = int
Edge = Tuple[Node, Node]
# CostFunction(edge_from, edge_to, departure_time) -> traversal_time
CostFunction = Callable[[Node, Node, float], float]

class TimeDependentGraph:
    def __init__(self):
        # adjacency: node -> list of (neighbor, base_distance, free_flow_time)
        self.adj: Dict[Node, List[Tuple[Node, float, float]]] = {}
        self.pos: Dict[Node, Tuple[float, float]] = {}  # for heuristic (x,y)
        self.dynamic_factors: Dict[Tuple[Node,Node], Dict] = {}  # store params per edge

    def add_node(self, n: Node, x: float, y: float):
        self.adj.setdefault(n, [])
        self.pos[n] = (x, y)

    def add_edge(self, u: Node, v: Node, distance: float, base_time: float, factor: float=0.3):
        # base_time: travel time at free-flow
        self.adj.setdefault(u, []).append((v, distance, base_time))
        # store dynamic factor params for simple simulation (amplitude, phase)
        self.dynamic_factors[(u, v)] = {"amp": factor, "phase": random.uniform(0, 2*math.pi)}

    def neighbors(self, u: Node):
        return self.adj.get(u, [])

    def euclidean(self, a: Node, b: Node) -> float:
        ax, ay = self.pos[a]
        bx, by = self.pos[b]
        return math.hypot(ax-bx, ay-by)

    def cost_function(self) -> CostFunction:
        # Return a cost function that uses internal dynamics
        def cost(u: Node, v: Node, depart_t: float) -> float:
            # find edge entry
            for nb, dist, base in self.adj.get(u, []):
                if nb == v:
                    params = self.dynamic_factors.get((u,v), {"amp":0.0,"phase":0.0})
                    # simulate time-of-day or traffic as sinusoidal multiplier
                    amp = params["amp"]
                    phase = params["phase"]
                    # period = 300 seconds (for demo rapid fluctuation)
                    period = 300.0
                    fluct = 1.0 + amp * math.sin(2*math.pi*(depart_t/period) + phase)
                    # ensure cost positive
                    return max(0.1, base * fluct)
            # if no direct edge, return large
            return float('inf')
        return cost

# ---------- Time-dependent Dijkstra (earliest-arrival) ----------
def td_dijkstra(graph: TimeDependentGraph, cost_fn: CostFunction, source: Node, target: Node, depart_time: float) -> Tuple[List[Node], float]:
    # uses earliest-arrival semantics: when exploring edge (u->v) arriving time = t_u + cost(u,v,t_u)
    pq = []
    heapq.heappush(pq, (depart_time, source, None))  # (arrival_time, node, parent)
    best_arrival: Dict[Node, float] = {source: depart_time}
    parent: Dict[Node, Optional[Node]] = {source: None}

    while pq:
        t_u, u, _ = heapq.heappop(pq)
        if t_u > best_arrival.get(u, float('inf')):
            continue
        if u == target:
            break
        for v, _, _ in graph.neighbors(u):
            travel = cost_fn(u, v, t_u)
            t_v = t_u + travel
            if t_v < best_arrival.get(v, float('inf')):
                best_arrival[v] = t_v
                parent[v] = u
                heapq.heappush(pq, (t_v, v, u))
    if target not in best_arrival:
        return ([], float('inf'))
    # reconstruct path
    path = []
    cur = target
    while cur is not None:
```

```python
    while cur is not None:
        path.append(cur)
        cur = parent.get(cur)
    path.reverse()
    return path, best_arrival[target]

# ---------- Time-dependent A* ----------
def td_a_star(graph: TimeDependentGraph, cost_fn: CostFunction, source: Node, target: Node, depart_time: float) -> Tuple[List[Node],
    # f = g + h where g is earliest-arrival (time), h is admissible heuristic: euclidean / max_speed_estimate
    # we must be careful: heuristic in time units; choose max_speed so heuristic is admissible.
    max_free_flow_speed = 1.0  # distance units per time unit adjusted to match base_time scale
    def heuristic(u: Node) -> float:
        # distance / max_speed -> time lower bound
        dist = graph.euclidean(u, target)
        return dist / max_free_flow_speed

    open_heap = []
    g_cost: Dict[Node, float] = {source: depart_time}
    f_cost: Dict[Node, float] = {source: depart_time + heuristic(source)}
    parent: Dict[Node, Optional[Node]] = {source: None}
    heapq.heappush(open_heap, (f_cost[source], source))

    while open_heap:
        f_u, u = heapq.heappop(open_heap)
        if u == target:
            break
        t_u = g_cost[u]
        for v, _, _ in graph.neighbors(u):
            travel = cost_fn(u, v, t_u)
            t_v = t_u + travel
            # tentative g = t_v
            if t_v < g_cost.get(v, float('inf')):
                g_cost[v] = t_v
                parent[v] = u
                f_v = t_v + heuristic(v)
                f_cost[v] = f_v
                heapq.heappush(open_heap, (f_v, v))
    if target not in g_cost:
        return ([], float('inf'))
    # reconstruct path
    path = []
    cur = target
    while cur is not None:
        path.append(cur)
        cur = parent.get(cur)
    path.reverse()
    return path, g_cost[target]

# ---------- Simulation and benchmark ----------
def build_sample_graph(num_nodes: int = 30, connectivity: int = 3) -> TimeDependentGraph:
    g = TimeDependentGraph()
    # place nodes on a grid with small noise
    for i in range(num_nodes):
        x = (i % 10) + random.uniform(-0.3, 0.3)
        y = (i // 10) + random.uniform(-0.3, 0.3)
        g.add_node(i, x, y)
    # connect each node to a few nearest neighbors
    nodes = list(g.pos.keys())
    for u in nodes:
        dists = []
        for v in nodes:
            if u == v: continue
            d = g.euclidean(u, v)
            dists.append((d, v))
        dists.sort()
        for k in range(min(connectivity, len(dists))):
            dist, v = dists[k]
            # base_time ~ distance * factor, ensure non-zero
            base_time = max(0.5, dist * random.uniform(0.8, 1.5))
            g.add_edge(u, v, dist, base_time, factor=random.uniform(0.1, 0.6))
    return g

def measure_algorithms(graph: TimeDependentGraph, cost_fn: CostFunction, queries: List[Tuple[Node, Node, float]], warmup: int=1):
    results = []
    for (s,t,depart) in queries:
        # Dijkstra
        t0 = time.perf_counter()
```

```python
        path_d, arrival_d = td_dijkstra(graph, cost_fn, s, t, depart)
        t1 = time.perf_counter()
        time_d = t1 - t0

        # A*
        t0 = time.perf_counter()
        path_a, arrival_a = td_a_star(graph, cost_fn, s, t, depart)
        t1 = time.perf_counter()
        time_a = t1 - t0

        # record
        results.append({
            "s": s, "t": t, "depart": depart,
            "dijkstra": {"path": path_d, "arrival": arrival_d, "runtime_s": time_d},
            "astar": {"path": path_a, "arrival": arrival_a, "runtime_s": time_a}
        })
    return results

def simulate_and_report(seed: int = 42, queries_count: int = 20):
    random.seed(seed)
    g = build_sample_graph(num_nodes=40, connectivity=4)
    cost_fn = g.cost_function()
    # generate random queries with random departure times
    queries = []
    nodes = list(g.pos.keys())
    for _ in range(queries_count):
        s = random.choice(nodes)
        t = random.choice(nodes)
        while t == s:
            t = random.choice(nodes)
        depart = random.uniform(0, 600)  # seconds in demo
        queries.append((s, t, depart))

    results = measure_algorithms(g, cost_fn, queries)
    # compute summary stats
    total_d_runtime = sum(r["dijkstra"]["runtime_s"] for r in results)
    total_a_runtime = sum(r["astar"]["runtime_s"] for r in results)
    avg_d_runtime = total_d_runtime / len(results)
    avg_a_runtime = total_a_runtime / len(results)

    # compare arrivals (if both found)
    arrivals_match = 0
    d_better = 0
    a_better = 0
    unreachable = 0
    for r in results:
        da = r["dijkstra"]["arrival"]
        aa = r["astar"]["arrival"]
        if math.isinf(da) and math.isinf(aa):
            unreachable += 1
            continue
        if abs(da - aa) < 1e-6:
            arrivals_match += 1
        else:
            if da < aa:
                d_better += 1
            else:
                a_better += 1

    # print a compact report
    print("=== Simulation Report ===")
    print(f"Queries: {len(results)} (seed={seed})")
    print(f"Avg runtime: Dijkstra={avg_d_runtime*1000:.3f} ms, A*={avg_a_runtime*1000:.3f} ms")
    print(f"Arrival comparison: match={arrivals_match}, dijkstra_better={d_better}, astar_better={a_better}, unreachable={unreachable}")
    print("Example result (first query):")
    if results:
        r = results[0]
        print(" Query:", r["s"], "->", r["t"], "depart", r["depart"])
        print("  Dijkstra: arrival={:.3f}, runtime_ms={:.3f}, path_len={}".format(
            r["dijkstra"]["arrival"], r["dijkstra"]["runtime_s"]*1000, len(r["dijkstra"]["path"])
        ))
        print("  A*       : arrival={:.3f}, runtime_ms={:.3f}, path_len={}".format(
            r["astar"]["arrival"], r["astar"]["runtime_s"]*1000, len(r["astar"]["path"])
        ))

if __name__ == "__main__":
    simulate_and_report()
```

Output:

```
End_Sem_Lab/Q1/dynamic_routing.py
=== Simulation Report ===
Queries: 20 (seed=42)
Avg runtime: Dijkstra=0.152 ms, A*=0.070 ms
Arrival comparison: match=18, dijkstra_better=2, astar_better=0, unreachable
Example result (first query):
 Query: 27 -> 23 depart 41.37404191178431
  A*      : arrival=46.957, runtime_ms=0.131, path_len=6
```

Observation:

This Python Prototype implements time-dependent routing (earliest-arrival Dijkstra and A* with a safe heuristic) and a simple simulator that injects live-like edge-cost fluctuations for benchmarking. It's a concise, runnable proof-of-concept ideal for experiments and algorithm comparison; for production you'll need real-time feeds, stronger heuristics/labels (or contraction/ALT), and performance tuning.

Q2: Optimize traffic light timings (heuristic).

- Task 1: Use AI to propose optimization loop.
- Task 2: Implement simulation and evaluate throughput.

Prompt:

Develop a heuristic-based traffic-light timing optimization system. First, use AI to design an optimization loop that adjusts signal timings to improve flow. Then implement a simulation of an intersection or road network and evaluate traffic throughput under the optimized timings. Provide explanations, code, and performance results.

Code:

End_Sem_Lab > Q2 > task.py > ...

```python
from typing import List, Tuple, Dict
import random
import copy
import math

# --------------- Simulator ---------------

class TrafficLight:
    def __init__(self, cycle: int = 30, ns_green_frac: float = 0.5):
        self.cycle = cycle  # in seconds (discrete ticks)
        self.ns_green_frac = ns_green_frac  # fraction of cycle that NS is green

    def is_ns_green(self, t: int) -> bool:
        """Return True if NS is green at time t (mod cycle)."""
        pos = t % self.cycle
        ns_green_time = max(1, int(round(self.ns_green_frac * self.cycle)))
        return pos < ns_green_time

class Intersection:
    def __init__(self, id_: int, light: TrafficLight):
        self.id = id_
        self.light = light
        # queues: 'N','S','E','W' counts of vehicles waiting to traverse intersection
        self.queues = {'N': 0, 'S': 0, 'E': 0, 'W': 0}

class Network:
    def __init__(self, grid_w=2, grid_h=2, default_cycle=30):
        # build grid intersections row-major
        self.grid_w = grid_w
        self.grid_h = grid_h
        self.intersections: Dict[Tuple[int,int], Intersection] = {}
        for y in range(grid_h):
            for x in range(grid_w):
                id_ = y*grid_w + x
                light = TrafficLight(cycle=default_cycle, ns_green_frac=0.5)
                self.intersections[(x,y)] = Intersection(id_, light)

    def set_light_params(self, params: Dict[Tuple[int,int], Tuple[int,float]]):
        """params: {(x,y): (cycle, ns_frac)}"""
        for k,v in params.items():
            if k in self.intersections:
                cycle, nsfrac = v
                self.intersections[k].light.cycle = cycle
                self.intersections[k].light.ns_green_frac = nsfrac

# Simple vehicle model: vehicles travel in cardinal directions straight across intersections
# For simplicity, each link traversal takes 1 tick if allowed by green, otherwise vehicle waits.
class Simulator:
    def __init__(self, network: Network, spawn_rate: float = 0.2, horizon: int = 300):
        self.net = network
        self.spawn_rate = spawn_rate  # per edge per tick probability
        self.horizon = horizon
        # queues on network edges per direction per intersection cell
        # We'll model only per-intersection incoming queues (direction indicates where vehicle will go next).
        # For exits, we count vehicles that leave the grid.
        self.exited = 0

    def reset(self):
        for it in self.net.intersections.values():
            it.queues = {'N': 0, 'S': 0, 'E': 0, 'W': 0}
        self.exited = 0

    def spawn_vehicles(self):
        # spawn from north edge (flow south into top row), south edge (flow north into bottom row),
        # west edge (eastward into left column), east edge (westward into right column)
        for x in range(self.net.grid_w):
            # north edge -> first row intersections at (x,0) queue 'N' (vehicle will travel south)
            if random.random() < self.spawn_rate:
                self.net.intersections[(x,0)].queues['N'] += 1
            # south edge -> last row intersections at (x,grid_h-1) queue 'S' (vehicle will travel north)
            if random.random() < self.spawn_rate:
                self.net.intersections[(x,self.net.grid_h-1)].queues['S'] += 1
        for y in range(self.net.grid_h):
            # west edge -> left column (0,y) queue 'W' (vehicle will travel east)
            if random.random() < self.spawn_rate:
                self.net.intersections[(0,y)].queues['W'] += 1
            # east edge -> right column (grid_w-1,y) queue 'E' (vehicle will travel west)
            if random.random() < self.spawn_rate:
                self.net.intersections[(self.net.grid_w-1,y)].queues['E'] += 1

    def step(self, t: int):
```

```python
        # Process each intersection: vehicles go if their direction has green and there is a vehicle.
        # Move vehicles to next intersection's queue or exit if at boundary.
        # We'll collect movements and apply them after evaluating all intersections to avoid ordering bias.
        moves = []  # tuples (from_coord, dir)
        for (x,y), inter in self.net.intersections.items():
            # NS green allows N->S and S->N movement (vehicles in 'N' or 'S' queues)
            ns_green = inter.light.is_ns_green(t)
            if ns_green:
                # Serve one vehicle per green per direction (simple saturation rule)
                if inter.queues['N'] > 0:
                    moves.append(((x,y),'N'))
                elif inter.queues['S'] > 0:
                    moves.append(((x,y),'S'))
            else:
                # EW green
                if inter.queues['W'] > 0:
                    moves.append(((x,y),'W'))
                elif inter.queues['E'] > 0:
                    moves.append(((x,y),'E'))
        # apply moves
        for (x,y), d in moves:
            inter = self.net.intersections[(x,y)]
            inter.queues[d] -= 1
            # compute next position based on direction: N means vehicle came from north -> it will exit to south
            if d == 'N':
                # vehicle moves south one cell (y+1)
                ny = y+1
                if ny >= self.net.grid_h:
                    self.exited += 1
                else:
                    # enters 'N' queue of next intersection (it will continue south)
                    self.net.intersections[(x,ny)].queues['N'] += 1
            elif d == 'S':
                ny = y-1
                if ny < 0:
                    self.exited += 1
                else:
                    self.net.intersections[(x,ny)].queues['S'] += 1
            elif d == 'W':
                nx = x+1
                if nx >= self.net.grid_w:
                    self.exited += 1
                else:
                    self.net.intersections[(nx,y)].queues['W'] += 1
            elif d == 'E':
                nx = x-1
                if nx < 0:
                    self.exited += 1
                else:
                    self.net.intersections[(nx,y)].queues['E'] += 1

    def run(self, seed:int = None) -> int:
        if seed is not None:
            random.seed(seed)
        self.reset()
        for t in range(self.horizon):
            self.spawn_vehicles()
            self.step(t)
        return self.exited

# ---------------- Optimizer (heuristic: random-restart hill-climb) ----------------

def random_neighbor(params: Dict[Tuple[int,int], Tuple[int,float]], cycle_bounds=(20,60), ns_frac_step=0.05):
    new = copy_params(params)
    # randomly pick one intersection and tweak ns_frac by +/- step or change cycle slightly
    key = random.choice(list(new.keys()))
    cycle, nsf = new[key]
    if random.random() < 0.5:
        # tweak ns_frac
        delta = random.choice([-1,1]) * ns_frac_step
        nsf = min(0.95, max(0.05, nsf + delta))
    else:
        # tweak cycle by +/- 5 seconds
        delta = random.choice([-5,5])
        cycle = min(cycle_bounds[1], max(cycle_bounds[0], cycle + delta))
    new[key] = (cycle, nsf)
    return new
```

```python

def copy_params(params):
    return {k: (v[0], v[1]) for k,v in params.items()}

def baseline_params(network: Network):
    params = {}
    for key in network.intersections.keys():
        params[key] = (30, 0.5)  # 30s cycle, equal split
    return params

def optimize(network: Network, sim: Simulator, restarts=5, iter_per_restart=50, seed=None):
    if seed is not None:
        random.seed(seed)
    best_overall = None
    best_params = None
    for r in range(restarts):
        # start from baseline or random
        params = baseline_params(network)
        # small random initialization
        for k in params.keys():
            cycle = random.choice([20,30,40,50])
            nsf = random.uniform(0.3, 0.7)
            params[k] = (cycle, nsf)
        # evaluate
        network.set_light_params(params)
        score = sim.run()
        best_local = score
        best_local_params = copy_params(params)
        # hill-climb
        for it in range(iter_per_restart):
            cand = random_neighbor(params)
            network.set_light_params(cand)
            s = sim.run()
            if s >= score:
                # accept
                params = cand
                score = s
                if s > best_local:
                    best_local = s
                    best_local_params = copy_params(cand)
        # track global best
        if best_overall is None or best_local > best_overall:
            best_overall = best_local
            best_params = best_local_params
    return best_overall, best_params

# --------------- Main quick demo ---------------

def demo():
    net = Network(grid_w=2, grid_h=2, default_cycle=30)
    sim = Simulator(net, spawn_rate=0.25, horizon=300)
    print("Baseline simulation (equal splits)...")
    net.set_light_params(baseline_params(net))
    base_out = sim.run(seed=1)
    print(" Baseline throughput (exited vehicles):", base_out)

    print("Running optimizer (random-restart hill-climb)...")
    best_score, best_params = optimize(net, sim, restarts=6, iter_per_restart=40, seed=2)
    print(" Best throughput found:", best_score)
    print(" Best params (per intersection):")
    for k,v in best_params.items():
        print("  ", k, "cycle", v[0], "ns_frac", round(v[1],2))

    # final verification run
    net.set_light_params(best_params)
    final = sim.run(seed=3)
    print(" Verified throughput with best params:", final)

if __name__ == "__main__":
    demo()

```

Output:

```
End_Sem_Lab/Q2/task.py
Baseline simulation (equal splits)...
 Baseline throughput (exited vehicles): 539
Running optimizer (random-restart hill-climb)...
 Best params (per intersection):
  (0, 0) cycle 20 ns_frac 0.5
  (1, 0) cycle 45 ns_frac 0.38
  (0, 1) cycle 50 ns_frac 0.46
  (1, 1) cycle 25 ns_frac 0.5
```

Observation:

The task tackles optimizing traffic-light timings to maximize network throughput under stochastic arrivals: I built a lightweight discrete-time simulator modeling intersections with NS/EW phases and a random-restart hill-climb optimizer that perturbs cycle times and green splits, evaluates candidates by simulated exited vehicles, and keeps the best result. This prototype shows how simple heuristics can improve throughput quickly, while production deployment would require a richer microsimulator (or SUMO), realistic traffic models, stronger optimizers (CMA-ES/GA) and robust evaluation across many seeds and demand patterns.