Project

Project Goal • The goal of this project is to simulate and optimize traffic light management using OS scheduling algorithms, ensuring efficient, fair, and real-time traffic control. The system will be built with a Python backend and a React frontend, allowing users to visualize and interact with the traffic simulation than Simulating Traffic Management by implementing a four-way intersection where traffic flow is controlled using OS scheduling algorithms. Intial Milestones • Develop a lightweight Python API using Flask or FastAPI to handle vehicle input, process traffic rules, and update signals • Create an interactive UI where users can add vehicles, see real-time traffic light updates, and visualize vehicle movement. • Ensure traffic lights change based on vehicle density and scheduling logic, improving efficiency over fixed timer system. • Implement a priority system that allows ambulances, fire trucks, and police cars to pass first. INTRODUCTION TO SOFTWARE ENGINEERING(TBC-402) Traffic Schedule Manager Page 3 Project Approach System Design 1 Backend (Python + Flask/FastAPI) • Handles traffic logic & scheduling algorithms (FCFS, Round Robin, Priority Scheduling). • Receives vehicle input from different lanes via API. • Processes real-time signal changes and sends updates to the frontend. 1.1 Libraries Used: • Flask – For API development • Threading – To simulate real-time updates • collections (deque) – For queue-based vehicle management • Time, Random – For scheduling and test data 2 Frontend (React +CSS) • Interactive UI for a four-way intersection where users can add vehicles. • Displays real-time traffic lights based on backend responses. • Animates vehicle movement when lights turn green 2.1 Libraries Used: • React.js – For UI development • Axios – For API communication • CSS – For styling • Framer Motion (optional) – For smooth animations 3 Running Locally 3.1 Backend • Runs on localhost:5000 using Flask • Vehicles are added via API calls and processed using scheduling algorithms 3.2 Frontend • Runs on localhost:3000 using React.and CSS • Fetches real-time traffic updates from the backend and displays animations.

App.js

import React, { useState, useEffect } from 'react';

import './App.css';

import { motion } from 'framer-motion';

import axios from 'axios';

function App() {

  const [vehicles, setVehicles] = useState([]);

  const [inputValues, setInputValues] = useState({

    north: '', south: '', east: '', west: ''

  });

  const [isSimulating, setIsSimulating] = useState(false);

  const [lightStates, setLightStates] = useState({

    north: 'red', south: 'red', east: 'green', west: 'green'

  });

  // Helper functions

  const positionToViewport = (pos) => ({ x: `${pos.x}vw`, y: `${pos.y}vh` });

  const getDirectionFromOrigin = (origin) =>

    ({ north: 'down', south: 'up', east: 'left', west: 'right' }[origin] || 'down');

  const getDirectionFromDestination = (destination) =>

    ({ north: 'up', south: 'down', east: 'right', west: 'left' }[destination] || 'down');

  const getInitialPosition = (direction) =>

    ({ north: { x: 50, y: 90 }, south: { x: 50, y: 10 },

       east: { x: 90, y: 50 }, west: { x: 10, y: 50 } }[direction] || { x: 50, y: 50 });

  const getStopPosition = (direction) =>

    ({ north: { x: 50, y: 60 }, south: { x: 50, y: 40 },

       east: { x: 60, y: 50 }, west: { x: 40, y: 50 } }[direction] || { x: 50, y: 50 });

  const getEmoji = (type) =>

    ({ car: '🚗', bike: '🚲', ambulance: '🚑', bus: '🚌' }[type] || '❓');

  // Core simulation logic

  const fetchSimulationState = async () => {

    try {

      const response = await axios.get('http://localhost:5000/get\_state');

      setLightStates(response.data.lights);

      const processVehicle = (vehicle, isQueued) => {

        const position = vehicle.position

          ? { x: 50 + (vehicle.position.x / 2), y: 50 - (vehicle.position.y / 2) }

          : getInitialPosition(vehicle.originDirection || vehicle.direction);

        const atCenter = Math.abs(position.x - 50) < 5 && Math.abs(position.y - 50) < 5;

        const direction = atCenter

          ? getDirectionFromDestination(vehicle.destination)

          : getDirectionFromOrigin(vehicle.originDirection || vehicle.direction);

        return {

          ...vehicle,

          direction,

          position,

          stopped: vehicle.stopped,

          isQueued

        };

      };

      const queuedVehicles = Object.entries(response.data.vehicles.queued)

        .flatMap(([direction, vehicles]) =>

          vehicles.map(vehicle =>

            processVehicle({ ...vehicle, originDirection: direction }, true)

          )

        );

      const activeVehicles = Object.entries(response.data.vehicles.active)

        .map(([id, vehicle]) =>

          processVehicle({ ...vehicle, id }, false)

        );

      const newVehicles = [...queuedVehicles, ...activeVehicles];

      setVehicles(newVehicles);

    } catch (error) {

      console.error('Error fetching simulation state:', error);

    }

  };

 // Replace the getTargetPosition function with this:

const getTargetPosition = (vehicle) => {

  if (vehicle.isQueued) {

    return getStopPosition(vehicle.originDirection);

  }

  // Convert backend coordinates to frontend viewport coordinates

  const currentPos = {

    x: 50 + (vehicle.position?.x || 0) / 2,

    y: 50 - (vehicle.position?.y || 0) / 2

  };

  // If vehicle is stopped, use stop position

  if (vehicle.stopped) {

    return getStopPosition(vehicle.originDirection);

  }

  // If near center (intersection), proceed to destination

  if (Math.abs(currentPos.x - 50) < 5 && Math.abs(currentPos.y - 50) < 5) {

    return ({

      up: { x: 50, y: 0 },

      down: { x: 50, y: 100 },

      left: { x: 0, y: 50 },

      right: { x: 100, y: 50 }

    }[vehicle.direction] || { x: 50, y: 50 });

  }

  // Default to moving toward center

  return { x: 50, y: 50 };

};

// Update the processVehicle function to this:

const processVehicle = (vehicle, isQueued) => {

  const position = vehicle.position

    ? {

        x: 50 + (vehicle.position.x / 2),

        y: 50 - (vehicle.position.y / 2)

      }

    : getInitialPosition(vehicle.originDirection || vehicle.direction);

  const atCenter = Math.abs(position.x - 50) < 5 && Math.abs(position.y - 50) < 5;

  const direction = atCenter

    ? getDirectionFromDestination(vehicle.destination)

    : getDirectionFromOrigin(vehicle.originDirection || vehicle.direction);

  return {

    ...vehicle,

    direction,

    position,

    stopped: vehicle.stopped,

    isQueued

  };

};

  // Event handlers

  useEffect(() => {

    if (!isSimulating) return;

    const interval = setInterval(fetchSimulationState, 100);

    return () => clearInterval(interval);

  }, [isSimulating]);

  const handleInputChange = (direction, value) => {

    setInputValues(prev => ({ ...prev, [direction]: value }));

  };

  const startSimulation = async () => {

    setIsSimulating(true);

    setVehicles([]);

    const vehicleTypes = { c: 'car', b: 'bike', t: 'bus', a: 'ambulance' };

    for (const direction of ['north', 'south', 'east', 'west']) {

      const vehicles = inputValues[direction].replace(/\s/g, '').split(',').filter(Boolean);

      for (const char of vehicles) {

        const type = vehicleTypes[char.toLowerCase()];

        if (!type) continue;

        try {

          await axios.post('http://localhost:5000/add\_vehicle', { type, direction });

          await new Promise(resolve => setTimeout(resolve, 200));

        } catch (error) {

          console.error('Error adding vehicle:', error);

        }

      }

    }

  };

  // Render

  return (

    <div className="app">

      <div className="intersection">

        {/\* Roads and traffic elements \*/}

        <div className="road horizontal"></div>

        <div className="road vertical"></div>

        <div className="lane-markings horizontal"></div>

        <div className="lane-markings vertical"></div>

        {/\* Zebra crossings \*/}

        {['top', 'bottom', 'left', 'right'].map(pos => (

          <div key={pos} className={`zebra zebra-${pos}`}></div>

        ))}

        {/\* Traffic lights \*/}

        {['north', 'south', 'east', 'west'].map(direction => (

          <div key={direction} className={`traffic-light ${direction}`}>

            {['red', 'yellow', 'green'].map(color => (

              <div key={color} className={`light ${lightStates[direction] === color ? `${color} active` : 'off'}`}></div>

            ))}

          </div>

        ))}

        {/\* Direction inputs \*/}

        {[

          { dir: 'north', label: 'North (↓)' },

          { dir: 'south', label: 'South (↑)' },

          { dir: 'east', label: 'East (←)' },

          { dir: 'west', label: 'West (→)' }

        ].map(({ dir, label }) => (

          <div key={dir} className={`direction-input ${dir}`}>

            <label>{label}</label>

            <input

              type="text"

              value={inputValues[dir]}

              onChange={(e) => handleInputChange(dir, e.target.value)}

              placeholder="c,b,t,a"

              disabled={isSimulating}

            />

          </div>

        ))}

        <button className="start-button" onClick={startSimulation} disabled={isSimulating}>

          {isSimulating ? 'Simulating...' : 'Start Simulation'}

        </button>

        {/\* Vehicles \*/}

        {vehicles.map(vehicle => {

          const targetPos = positionToViewport(getTargetPosition(vehicle));

          const initialPos = positionToViewport(vehicle.position || getInitialPosition(vehicle.originDirection));

          return (

            <motion.div

              key={vehicle.id}

              className={`vehicle ${vehicle.type} ${vehicle.stopped ? 'stopped' : ''}`}

              initial={{ left: initialPos.x, top: initialPos.y }}

              animate={{ left: targetPos.x, top: targetPos.y }}

              transition={{ duration: vehicle.isQueued ? 3 : 5, ease: 'linear' }}

              style={{

                position: 'absolute',

                zIndex: vehicle.type === 'ambulance' ? 20 : 15,

                transform: 'translate(-50%, -50%)'

              }}

            >

              {getEmoji(vehicle.type)}

              {vehicle.type === 'ambulance' && <span className="siren">🚨</span>}

            </motion.div>

          );

        })}

      </div>

    </div>

  );

}

export default App;

App.css

.app {

  height: 100vh;

  width: 100vw;

  background-color: #2e8b57;

  overflow: hidden;

  position: relative;

}

.intersection {

  position: absolute;

  height: 100%;

  width: 100%;

}

/\* Roads - made wider to show more of the approach \*/

.road {

  position: absolute;

  background-color: #333;

  z-index: 1;

}

.road.horizontal {

  top: 50%;

  left: 0;

  width: 100%;

  height: 160px;

  transform: translateY(-50%);

}

.road.vertical {

  left: 50%;

  top: 0;

  height: 100%;

  width: 160px;

  transform: translateX(-50%);

}

/\* Lane markings - more visible \*/

.lane-markings {

  position: absolute;

  background-repeat: repeat;

  z-index: 2;

}

.lane-markings.horizontal {

  top: 50%;

  left: 0;

  width: 100%;

  height: 4px;

  background-image: repeating-linear-gradient(

    to right,

    white 0,

    white 20px,

    transparent 20px,

    transparent 40px

  );

  transform: translateY(-2px);

}

.lane-markings.vertical {

  left: 50%;

  top: 0;

  width: 4px;

  height: 100%;

  background-image: repeating-linear-gradient(

    to bottom,

    white 0,

    white 20px,

    transparent 20px,

    transparent 40px

  );

  transform: translateX(-2px);

}

/\* Zebra crossings - more visible \*/

.zebra {

  position: absolute;

  background: white;

  z-index: 3;

}

.zebra-top {

  top: calc(50% - 80px);

  left: calc(50% - 60px);

  width: 120px;

  height: 10px;

  background: repeating-linear-gradient(

    to bottom,

    white 0,

    white 10px,

    #333 10px,

    #333 20px

  );

}

.zebra-bottom {

  top: calc(50% + 70px);

  left: calc(50% - 60px);

  width: 120px;

  height: 10px;

  background: repeating-linear-gradient(

    to bottom,

    white 0,

    white 10px,

    #333 10px,

    #333 20px

  );

}

.zebra-left {

  left: calc(50% - 80px);

  top: calc(50% - 60px);

  width: 10px;

  height: 120px;

  background: repeating-linear-gradient(

    to right,

    white 0,

    white 10px,

    #333 10px,

    #333 20px

  );

}

.zebra-right {

  left: calc(50% + 70px);

  top: calc(50% - 60px);

  width: 10px;

  height: 120px;

  background: repeating-linear-gradient(

    to right,

    white 0,

    white 10px,

    #333 10px,

    #333 20px

  );

}

/\* Traffic lights - positioned at each approach \*/

.traffic-light {

  position: absolute;

  width: 30px;

  height: 90px;

  background-color: #111;

  display: flex;

  flex-direction: column;

  justify-content: space-between;

  align-items: center;

  padding: 8px 0;

  border-radius: 8px;

  z-index: 4;

  box-shadow: 0 0 10px rgba(0,0,0,0.5);

}

/\* Position each traffic light closer to intersection \*/

.traffic-light.north {

  top: calc(40% - 45px);  /\* Moved closer to intersection \*/

  left: 50%;

  transform: translateX(-50%);

}

.traffic-light.south {

  bottom: calc(40% - 45px);  /\* Moved closer to intersection \*/

  left: 50%;

  transform: translateX(-50%);

}

.traffic-light.east {

  top: 50%;

  right: calc(40% - 15px);  /\* Moved closer to intersection \*/

  transform: translateY(-50%);

}

.traffic-light.west {

  top: 50%;

  left: calc(40% - 15px);  /\* Moved closer to intersection \*/

  transform: translateY(-50%);

}

/\* Rest of your existing CSS... \*/

.traffic-light .light {

  width: 25px;

  height: 25px;

  border-radius: 50%;

  background-color: #444;

}

.traffic-light .red.active {

  background-color: red;

  box-shadow: 0 0 15px rgba(255,0,0,0.7);

}

.traffic-light .yellow.active {

  background-color: yellow;

  box-shadow: 0 0 15px rgba(255,255,0,0.7);

}

.traffic-light .green.active {

  background-color: #00ff00;

  box-shadow: 0 0 15px rgba(0,255,0,0.7);

}

.traffic-light .off {

  opacity: 0.2;

}

/\* Direction inputs \*/

.direction-input {

  position: absolute;

  background: rgba(255, 255, 255, 0.9);

  padding: 10px;

  border-radius: 6px;

  z-index: 10;

  display: flex;

  flex-direction: column;

  min-width: 160px;

  box-shadow: 0 2px 8px rgba(0,0,0,0.2);

}

.direction-input label {

  font-size: 14px;

  margin-bottom: 6px;

  color: #222;

  font-weight: bold;

}

.direction-input input {

  padding: 8px;

  border: 1px solid #ddd;

  border-radius: 4px;

  font-size: 14px;

}

.direction-input.north {

  top: 10px;

  left: 50%;

  transform: translateX(-50%);

}

.direction-input.south {

  bottom: 10px;

  left: 50%;

  transform: translateX(-50%);

}

.direction-input.east {

  top: 50%;

  right: 10px;

  transform: translateY(-50%);

}

.direction-input.west {

  top: 50%;

  left: 10px;

  transform: translateY(-50%);

}

.start-button {

  position: absolute;

  top: 50%;

  left: 50%;

  transform: translate(-50%, -50%);

  padding: 12px 24px;

  background: #4CAF50;

  color: white;

  border: none;

  border-radius: 6px;

  cursor: pointer;

  z-index: 20;

  font-weight: bold;

  font-size: 16px;

  box-shadow: 0 4px 8px rgba(0,0,0,0.2);

  transition: all 0.3s;

}

.start-button:hover:not(:disabled) {

  background: #45a049;

  transform: translate(-50%, -50%) scale(1.05);

}

.start-button:disabled {

  background: #aaa;

  cursor: not-allowed;

}

.vehicle {

  position: absolute;

  font-size: 28px;

  transform: translate(-50%, -50%);

  will-change: transform;

}

.vehicle.stopped {

  opacity: 0.7;

  filter: brightness(0.8);

  animation: pulse 1s infinite;

}

@keyframes pulse {

  0% { transform: translate(-50%, -50%) scale(1); }

  50% { transform: translate(-50%, -50%) scale(1.05); }

  100% { transform: translate(-50%, -50%) scale(1); }

}

.siren {

  position: absolute;

  font-size: 14px;

  top: -15px;

  left: 50%;

  transform: translateX(-50%);

  animation: blink 0.5s infinite alternate;

}

@keyframes blink {

  from { opacity: 0.3; }

  to { opacity: 1; }

}

.direction-input input:disabled {

  background: #eee;

  cursor: not-allowed;

}

Schedule.py

from flask import Flask, jsonify, request

from flask\_cors import CORS

from collections import defaultdict, deque

import time

import threading

import random

app = Flask(\_\_name\_\_)

CORS(app)

# Constants

INTERSECTION\_CENTER = (0, 0)

LANE\_LENGTH = 100  # Distance from intersection center to spawn point

VEHICLE\_SPEED = 20  # units per second

CROSSING\_TIME = 5  # seconds to cross intersection

# Traffic simulation state

simulation\_state = {

    'lights': {

        'north': 'red',

        'south': 'red',

        'east': 'green',

        'west': 'green'

    },

    'vehicles': {

        'queued': defaultdict(deque),

        'active': {}

    },

    'next\_id': 1,

    'intersection\_boundary': 15  # Increased boundary for smoother stopping

}

# Replace the entire Vehicle class with this updated version:

class Vehicle:

    def \_\_init\_\_(self, vid, vtype, origin, destination):

        self.id = vid

        self.type = vtype

        self.origin = origin

        self.destination = destination

        self.position = self.get\_spawn\_position(origin)

        self.path = self.calculate\_path()

        self.current\_segment = 0

        self.progress = 0

        self.speed = VEHICLE\_SPEED

        self.stopped = False

        self.has\_stopped\_at\_light = False

    def get\_spawn\_position(self, origin):

        positions = {

            'north': (0, LANE\_LENGTH),

            'south': (0, -LANE\_LENGTH),

            'east': (LANE\_LENGTH, 0),

            'west': (-LANE\_LENGTH, 0)

        }

        return positions.get(origin, (0, 0))

    def calculate\_path(self):

        path = []

        # From spawn to approach line

        path.append({

            'start': self.position,

            'end': self.get\_approach\_point(self.origin),

            'type': 'approach'

        })

        # Through intersection

        path.append({

            'start': self.get\_approach\_point(self.origin),

            'end': self.get\_exit\_point(self.destination),

            'type': 'intersection'

        })

        # To destination

        path.append({

            'start': self.get\_exit\_point(self.destination),

            'end': self.get\_spawn\_position(self.destination),

            'type': 'exit'

        })

        return path

    def get\_approach\_point(self, direction):

        boundary = simulation\_state['intersection\_boundary']

        return {

            'north': (0, boundary),

            'south': (0, -boundary),

            'east': (boundary, 0),

            'west': (-boundary, 0)

        }.get(direction, (0, 0))

    def get\_exit\_point(self, direction):

        return self.get\_approach\_point(direction)

    def distance(self, p1, p2):

        return ((p1[0]-p2[0])\*\*2 + (p1[1]-p2[1])\*\*2)\*\*0.5

    def get\_current\_position(self):

        segment = self.path[self.current\_segment]

        x = segment['start'][0] + (segment['end'][0] - segment['start'][0]) \* self.progress

        y = segment['start'][1] + (segment['end'][1] - segment['start'][1]) \* self.progress

        return (x, y)

    def should\_stop(self):

        # Only check when approaching the intersection (first segment)

        if self.current\_segment == 0:

            current\_pos = self.get\_current\_position()

            approach\_point = self.get\_approach\_point(self.origin)

            distance = self.distance(current\_pos, approach\_point)

            # Stop if close to intersection and light isn't green

            if (distance < simulation\_state['intersection\_boundary'] and

                simulation\_state['lights'][self.origin] != 'green' and

                self.type != 'ambulance'):

                return True

        return False

    def update(self, dt):

        # Update stopped state

        self.stopped = self.should\_stop()

        # Emergency vehicles ignore red lights

        if self.type == 'ambulance' and self.stopped:

            self.stopped = False

            self.speed = VEHICLE\_SPEED \* 0.8  # Slow down slightly

        # Don't move if stopped

        if self.stopped:

            return False

        # Calculate movement

        segment = self.path[self.current\_segment]

        distance = self.speed \* dt

        segment\_length = self.distance(segment['start'], segment['end'])

        # Normal movement

        self.progress += distance / segment\_length

        # Update position

        self.position = self.get\_current\_position()

        # Check if segment complete

        if self.progress >= 1:

            self.progress = 0

            self.current\_segment += 1

            if self.current\_segment >= len(self.path):

                return True  # Vehicle reached destination

        return False

def traffic\_light\_controller():

    while True:

        # North-South green, East-West red

        simulation\_state['lights'] = {

            'north': 'green',

            'south': 'green',

            'east': 'red',

            'west': 'red'

        }

        time.sleep(8)

        # North-South yellow

        simulation\_state['lights'] = {

            'north': 'yellow',

            'south': 'yellow',

            'east': 'red',

            'west': 'red'

        }

        time.sleep(2)

        # East-West green, North-South red

        simulation\_state['lights'] = {

            'north': 'red',

            'south': 'red',

            'east': 'green',

            'west': 'green'

        }

        time.sleep(8)

        # East-West yellow

        simulation\_state['lights'] = {

            'north': 'red',

            'south': 'red',

            'east': 'yellow',

            'west': 'yellow'

        }

        time.sleep(2)

def vehicle\_controller():

    last\_time = time.time()

    while True:

        current\_time = time.time()

        dt = min(current\_time - last\_time, 0.1)  # Cap dt to prevent large jumps

        last\_time = current\_time

        # Update all active vehicles

        for vid, vehicle in list(simulation\_state['vehicles']['active'].items()):

            try:

                reached\_dest = vehicle.update(dt)

                if reached\_dest:

                    del simulation\_state['vehicles']['active'][vid]

            except Exception as e:

                print(f"Error updating vehicle {vid}: {e}")

                continue

        # Spawn new vehicles from queues

        for direction in ['north', 'south', 'east', 'west']:

            queue = simulation\_state['vehicles']['queued'][direction]

            if queue and simulation\_state['lights'][direction] == 'green':

                vehicle\_data = queue.popleft()

                destinations = ['north', 'south', 'east', 'west']

                destinations.remove(direction)

                destination = random.choice(destinations)

                try:

                    vehicle = Vehicle(

                        vehicle\_data['id'],

                        vehicle\_data['type'],

                        direction,

                        destination

                    )

                    simulation\_state['vehicles']['active'][vehicle.id] = vehicle

                except Exception as e:

                    print(f"Error creating vehicle: {e}")

                    continue

        time.sleep(0.05)

@app.route('/add\_vehicle', methods=['POST'])

def add\_vehicle():

    data = request.json

    vehicle = {

        'id': simulation\_state['next\_id'],

        'type': data['type'],

        'direction': data['direction']

    }

    simulation\_state['next\_id'] += 1

    simulation\_state['vehicles']['queued'][data['direction']].append(vehicle)

    return jsonify({'success': True, 'vehicle\_id': vehicle['id']})

@app.route('/get\_state', methods=['GET'])

def get\_state():

    active\_vehicles = {}

    for vid, vehicle in simulation\_state['vehicles']['active'].items():

        pos = vehicle.position

        active\_vehicles[vid] = {

            'type': vehicle.type,

            'origin': vehicle.origin,

            'destination': vehicle.destination,

            'position': {'x': pos[0], 'y': pos[1]},

            'stopped': vehicle.stopped

        }

    return jsonify({

        'lights': simulation\_state['lights'],

        'vehicles': {

            'queued': {k: list(v) for k, v in simulation\_state['vehicles']['queued'].items()},

            'active': active\_vehicles

        }

    })

if \_\_name\_\_ == '\_\_main\_\_':

    # Start controller threads

    threading.Thread(target=traffic\_light\_controller, daemon=True).start()

    threading.Thread(target=vehicle\_controller, daemon=True).start()

    # Start Flask app

    app.run(port=5000, debug=True)