School of Engineering and Applied Science (SEAS), Ahmedabad University

CSE400: Fundamentals of Probability in Computing

Group Name: s1_its_5
Team Members:

- Sakina Jambughodawala (AU2340114)
- Parin Patel (AU2340243)
- Maanya Patel (AU2340087)
- Hevit Makavana (AU2340194)

I. Background and Motivation

The traffic control systems cannot handle instant traffic modifications since urban growth occurs at fast rates coupled with expanding vehicle number increases. The signal system operated by fixed schedules and basic sensors produces frequent congestion together with useless fuel waste and delayed waiting times. The system project develops intelligent traffic lights through the combination of probability methods for random number generation and SUMO simulation models which manipulate signal control duration. System decision-making performance increases because of traffic density analysis and probabilistic logic processing together with random flow monitoring which improves operational efficiency and minimizes wait times and ensures road safety. Specific platforms need development which should acquire learning capabilities to adapt their conduct through monitoring driver-traffic interaction.

Three major problems emerge from traffic congestion because it leads to economic losses together with pollution decline and poor health effects. Static signal control systems currently in operation show poor operational performance results. Quantifiable results came from research that employed Q-learning algorithms and genetic algorithms via AI applications. Our technique added probabilistic calculation elements for managing variable SUMO simulation traffic response times.

II. Application

- Smart city traffic light systems
- Real-time adaptive control in high-density areas
- Emergency vehicle routing optimization
- Urban planning and traffic simulation testing
- Pollution and fuel consumption reduction

Figure 1: Traditional traffic signal system with fixed cycles

Figure 2: Traffic waiting at a typical signalized intersection

Figure 3: Smart traffic system concept with connected vehicles

Figure 4: Accident caused by poor traffic signal management

III. T1 - Mathematical Modelling and Mathematical Analysis

This model optimizes green light allocation using arrival rates λ_i and service rate μ to reduce waiting time and improve traffic flow. Key variables like N, λ_i, W, T_i and R_i are defined with suitable distributions. Green time g_i is dynamically assigned based on arrival rate, and due to that, density gets increased, so to manage congestion efficiently at multi-path intersections.

1. **Number of Vehicles,** N
    - Type: Discrete
    - Distribution: Probability Mass Function (PMF)
    - Constraint: $N_i \leq N_{\max}$
    - T_i: Vehicles that pass during green light
    - R_i: Residual traffic (vehicles left after green)
    - Depends on arrival rate λ_i and service rate μ
2. **Arrival Rate,** λ_i
    - Type: Continuous
    - Distribution: Poisson
    - Units: Vehicles per second
3. **Waiting Time,** W
    - Type: Continuous
    - Distribution: Normal
    - Red time duration: $k - g_i$
    - Traffic which will be cleared in next cycle

$$R_i = [(k - g_i)\lambda_i + g_i\lambda_i] - T_i$$

4. **Traffic Light State,** TS
    - Type: Discrete
    - Distribution: PMF
    - Formula:

$$TS_{g_i} = k \cdot \frac{\lambda_{\max,i}}{\sum \lambda_{\max,i}}$$

5. **Service Rate,** μ
    - Number of vehicles cleared per second during green

Traffic Flow Logic and Time Allocation

- Assume 4 paths at an intersection with different arrival rates λ_i.
- Traditional allocation: $g_i = \frac{k}{4}$, where k is the total green cycle time. Green light time allocation to i paths: g_i Therefore, total time:

$$\sum g_i = k$$

  where $k = 120$ sec normally, constant distribution $= \frac{k}{4}$
- Dynamic allocation aims to reduce waiting time

Green Time Allocation using Weights

- Green light time allocation: TSgi
  Let $\lambda_{\max}$ be the maximum arrival rate.

  Higher arrival rates receive higher weight, so green time allocation is proportional to weight:

$$g_i \propto w_i$$

- Let $\lambda_{\max,i}$ be the max arrival rate for path i
- Define weight:

$$w_i = \frac{\lambda_{\max,i}}{\sum \lambda_{\max,i}}$$

- Allocate green time as:

$$g_i = k \cdot w_i = k \cdot \frac{\lambda_{\max,i}}{\sum \lambda_{\max,i}}$$

Vehicles Passing and Waiting Time Calculations

- **Traffic passed during green time (g_i)**:
  Let T_i be the number of vehicles passed during green time.
  - Service rate: $\mu = 20$ vehicles per second (when $k = 120$ sec)
  - Therefore:

$$T_i = \mu \cdot g_i = \mu \cdot k \cdot \frac{\lambda_{\max,i}}{\sum \lambda_{\max,i}}$$

- **Waiting Time for New Vehicles Arriving:**
  - Remaining time in the cycle (when light is red) = Total cycle time k – Green time g_i

  - New vehicles waiting time with respect to their path:

$$W = k - g_i$$

- **Traffic Accumulated When Light is Red:**
  - Number of vehicles accumulated during red light = Waiting time × arrival rate:

$$(k - g_i) \cdot \lambda_i$$

  - Number of vehicles passing during green light = Green time × arrival rate:

$$g_i \cdot \lambda_i$$

  - If the green time is not enough, some vehicles will remain waiting for the next cycle, which will be more than k.

Total and Residual Traffic

- **Total accumulated traffic:**

$$(k - g_i)\lambda_i + g_i\lambda_i$$

- **For Residual Traffic, R_i:**
  - Traffic passed: $T_i = \mu g_i$
  - Accumulated traffic:

$$(k - g_i)\lambda_i + g_i\lambda_i$$

  - Therefore, residual traffic:

$$R_i = [(k - g_i)\lambda_i + g_i\lambda_i] - T_i$$

  - Simplified:

$$\boxed{R_i = k\lambda_i - \mu g_i}$$

- If $R_i = 0$, it means all vehicles were cleared during the green signal.
- If $R_i > 0$, some vehicles remain in queue.
- The system can adjust g_i (green time) in the next cycle to gradually clear the residual traffic.

IV. T2 - Code (with description of each line)

1.Importing Standard Libraries

```
#importing libraries
import os
import sys
from warnings import filterwarnings
import numpy as np
from time import sleep
```

Explanation:
This Python code snippet imports commonly used libraries:

- `os` and `sys` are used for interacting with the operating system and Python runtime environment.
- `warnings.filterwarnings` is used to control the visibility of warning messages.
- `numpy` (imported as `np`) is a powerful library for numerical computing and handling arrays.
- `sleep` from the `time` module is used to delay execution by a specified amount of time.

2. Importing SUMO and Simulation Tools

```
from Simulation_Generator import get_options
filterwarnings("ignore")

if 'SUMO_HOME' in os.environ:
    tools = os.path.join(os.environ['SUMO_HOME'], 'tools')
    sys.path.append(tools)
else:
    sys.exit("please declare environment variable 'SUMO_HOME'")

import traci
from sumolib import checkBinary
```

Explanation:
This snippet sets up the environment for SUMO (Simulation of Urban MObility):

- `get_options` is imported from a custom simulation script to handle command-line options.
- `filterwarnings("ignore")` suppresses warning messages.
- The `SUMO_HOME` environment variable is checked to locate SUMO tools.

- `traci` is used to control simulations via the TraCI interface.
- `checkBinary` from `sumolib` is used to locate SUMO binaries.

3. SUMO Simulation and Genetic Algorithm Parameters

```
sumo_binary = checkBinary('sumo')
sumo_binary_gui = checkBinary('sumo-gui')
config_file = 'Manhattan5x3.sumocfg'
n_steps = 1000
visualize_delay = 100 #delay in ms

pop_size = 15
max_generations = 100
n_survivors = 3

crossover_rate=0.5
duration_mutation_rate=0.2
duration_mutation_strength=5
states_mutation_rate = 0.2
light_options = ['G', 'y', 'r']

collision_penalty = 400
waiting_time_weight = 1/2000
emissions_weight = 1/2000
```

Explanation:
This snippet defines key simulation and optimization parameters:

- SUMO variables : specify the SUMO binary, GUI version, configuration file, number of steps, and visualization delay.
- Genetic Algorithm parameters : control the population size, number of generations, and survivors selected.
- Mutation and crossover : settings define the rates and strengths of genetic operations applied during evolution.
- Fitness function parameters : assign weights and penalties to performance metrics like collisions, waiting time, and emissions.

4. Traffic Light Class: State and Duration Management

```
class TLight(object):
    def __init__(self,ID):
        '''
        Initialize a new data storing object to keep track of the
        emissions and waiting time data over time
```

```
        '''
        self.ID = ID
        self.durations = []
        self.states = []

    def get_state_duration(self,ID):
        states=[]
        durations=[]
        definitions = traci.trafficlight.
           getCompleteRedYellowGreenDefinition(ID)
        for definition in definitions:
            for phase in definition.phases:
                states.append(phase.state)
                durations.append(phase.minDur)

        self.states = states
        self.durations = durations
        return states,durations

    def set_state_duration(self,ID,states,durations):
        self.ID = ID
        self.states = states
        self.durations = durations
        definition = traci.trafficlight.
           getCompleteRedYellowGreenDefinition(ID)[0]
        idx = 0
        for phase in definition.phases:
            phase.minDur = phase.maxDur = phase.duration =
               durations[idx]
            phase.states = states[idx]
            idx += 1

        logic = traci.trafficlight.Logic(traci.trafficlight.
           getProgram(ID), 0, 0, phases=definition.phases)
        traci.trafficlight.setCompleteRedYellowGreenDefinition(ID
           , logic)

        return self
```

Explanation:
This class handles the extraction and modification of traffic light signal phases:

- The `__init__` method initializes the object with an ID and empty lists for durations and states.
- `get_state_duration` extracts the current states and their durations using SUMO's 'traci' API.

- `set_state_duration` updates a traffic light's state definitions and applies them using the 'traci' interface.

5. Population Class: Initialization, Evaluation, and Print Utilities

```
class population(object):
    def __init__(self,generation):
        self.generation = generation
        self.gene_pool = []
        self.best_individual= None
        self.best_fitness = None
        self.best_emissions = None
        self.best_waiting_time = None
        self.avg_fitness = None
        self.avg_waiting_time = None
        self.avg_emissions = None
        self.json = None

    def print_pop(self):
        for individual,chromosome in zip(range(len(self.gene_pool
           )),self.gene_pool):
            print('----------------------------------------')
            print('----------------------------------------')
            print('----------------------------------------\n')
            print("Generation :",self.generation)
            print("Individual :",individual)
            print_chromosome(chromosome)

    def evaluate_pop(self):
        fitness_pop = []
        emissions_pop= []
        waiting_pop= []
        individual=0
        for chromosome in self.gene_pool:
            print("Generation:",self.generation," Individual: ",
               individual)
            fitness, emissions, waiting_time =
               evaluate_chromosome(chromosome)
            fitness_pop.append(fitness)
            emissions_pop.append(emissions)
            waiting_pop.append(waiting_time)
            individual=individual+1

        self.best_fitness = np.min(fitness_pop)
        self.best_emissions = np.min(emissions_pop)
        self.best_waiting_time = np.min(waiting_pop)
```

```
        self.avg_fitness = np.mean(fitness_pop)
        self.avg_emissions = np.mean(emissions_pop)
        self.avg_waiting_time = np.mean(waiting_pop)
        self.best_individual= self.gene_pool[np.argmin(
           fitness_pop)]

        return fitness_pop,emissions_pop,waiting_pop
```

Explanation:
This class represents a population in a genetic algorithm framework:

- The `__init__` method sets up tracking variables and initializes storage for gene pool and evaluation metrics.
- `print_pop` displays information for each individual in the current generation.
- `evaluate_pop` computes fitness, emissions, and waiting time for all chromosomes and tracks the best and average values across the population.

6. Chromosome Utility Functions: Get, Set, and Print

```
def get_chromosome():
    ''' Connects to the SUMO simulation and collects information
       about all traffic lights in a simulation'''
    traci.start([sumo_binary, "-c", config_file,'--start','--quit
       -on-end'],label = 'sim2')
    print("SUMO launched: Collecting Traffic Lights")
    conn = traci.getConnection("sim2")
    print(conn)
    TLightIDs = conn.trafficlight.getIDList()
    Chromosome=[]
    for Junction in TLightIDs:
        states = []
        durations=[]
        TL1 = TLight(Junction)
        states,durations = TL1.get_state_duration(Junction)
        Chromosome.append(TL1)

    traci.close()
    return Chromosome

def set_chromosome(chromosome):
    new_chromosome = []
    for Junction in chromosome:
        ID = Junction.ID
        states = Junction.states
        durations=Junction.durations
```

```
        TL1 = TLight(Junction)
        updated_TL1 = TL1.set_state_duration(ID,states,durations)
        new_chromosome.append(updated_TL1)

    return new_chromosome

def print_chromosome(chromosome):
    for Junction in chromosome:
        ID = Junction.ID
        states = Junction.states
        durations=Junction.durations
        print('-----------------------------------')
        print("Signal ID :",ID,"\n")
        print("States \t \t Durations")
        print('-----------------------------------')
        for states,durations in zip(states,durations):
            print(states,"\t \t", durations)
```

Explanation:
These utility functions handle reading and updating the chromosome representation of traffic lights:

- `get_chromosome` connects to the SUMO simulation, reads the current traffic light states and durations, and constructs the chromosome.
- `set_chromosome` updates the simulation's traffic lights with a new set of states and durations.
- `print_chromosome` prints the signal ID, its states, and durations in a formatted manner for each junction.

7. Genetic Algorithm Function: Mutation

```
def mutate_chromosome(chromosome,duration_mutation_rate=
   duration_mutation_rate,
                      duration_mutation_strength =
                         duration_mutation_strength,
                      states_mutation_rate = states_mutation_rate
                          ):
    chromosome_new = []
    for Tlight in chromosome:
        ID = Tlight.ID
        Tlight_new =TLight(ID)
        durations = Tlight.durations
        states = Tlight.states
        durations_new = []
        states_new = []
```

```
        RNG = (np.random.uniform(0,1,len(durations)) <=
           duration_mutation_rate)*1

        for D1,i in zip(durations,range(len(RNG))):
            D1 = max(round(D1 + RNG[i]* np.random.normal(0,
               duration_mutation_strength),1),1)
            durations_new.append(D1)

        for S1,i in zip(states,range(len(RNG))):
            lane_colors = list(S1)
            lane_colors_new = []
            for c1 in lane_colors:
                if np.random.uniform(0,1) < states_mutation_rate:
                    lane_colors_new.append(np.random.choice(
                       light_options))
                else:
                    lane_colors_new.append(c1)

            lane_colors_new = ''.join(lane_colors_new)
            states_new.append(lane_colors_new)

        Tlight_new.durations = durations_new
        Tlight_new.states = states_new
        chromosome_new.append(Tlight_new)

    return chromosome_new
```

Explanation:
This function performs mutation on the chromosome representation of traffic lights:

- Durations of traffic light phases are mutated by adding Gaussian noise with a certain probability.
- States (i.e., light signals) may change randomly to another signal (e.g., from 'G' to 'r') with a given mutation rate.
- The result is a new chromosome representing a mutated traffic light configuration.

8. Genetic Algorithm Function: Crossover

```
def crossover_parent(chromosome_male,chromosome_female,
   crossover_rate = crossover_rate):
    chromosome_child = []
    for Tlight in range(len(chromosome_male)):

        ID = chromosome_male[Tlight].ID
        Tlight_child = TLight(ID)
```

```
        durations_male= chromosome_male[Tlight].durations
        states_male = chromosome_male[Tlight].states
        durations_female= chromosome_female[Tlight].durations
        states_female = chromosome_female[Tlight].states
        durations_child = []
        states_child = []

        for D_male,D_female in zip(durations_male,
           durations_female):
            RNG = np.random.uniform(0,1)
            durations_child.append(D_male if RNG <
               crossover_rate else D_female)

        for S_male,S_female in  zip(states_male,states_female):
            lane_colors_male = list(S_male)
            lane_colors_female = list(S_female)
            lane_colors_child=[]
            for i in range(len(lane_colors_male)):
                RNG= np.random.uniform(0,1)
                if RNG < crossover_rate:
                    lane_colors_child.append(lane_colors_male[i])
                else:
                    lane_colors_child.append(lane_colors_female[i
                       ])
            lane_colors_child = ''.join(lane_colors_child)
            states_child.append(lane_colors_child)

        Tlight_child.durations = durations_child
        Tlight_child.states = states_child
        chromosome_child.append(Tlight_child)

    return chromosome_child
```

Explanation:
This function performs a genetic crossover between two parent chromosomes:

- For each traffic light, it mixes the duration values and states of both parents.
- A random decision is made for each phase/state based on the crossover rate.
- The resulting child chromosome represents a new combination of the parents' traits.

9. Genetic Algorithm Function: Evaluate Chromosome

```
def evaluate_chromosome(chromosome, sumo_binary=sumo_binary,
                        config_file=config_file, n_steps=n_steps)
                           :
```

```
3
4       traci.start([sumo_binary, "-c", config_file, '--start', '--
            quit-on-end'], label='sim2')
5       print("SUMO launched")
6       conn = traci.getConnection("sim2")
7       print(conn)
8
9       lanes = conn.lane.getIDList()
10      set_chromosome(chromosome)
11
12      lane_emissions = np.zeros((n_steps+1, len(lanes)))
13      lane_waiting = np.zeros((n_steps+1, len(lanes)))
14      fitness = 0
15
16      for step in range(n_steps):
17          traci.simulationStep()
18          if traci.simulation.getCollidingVehiclesNumber() > 0:
19              fitness += collision_penalty
20              break
21
22          for l1, i in zip(lanes, range(len(lanes))):
23              lane_emissions[n_steps][i] += traci.lane.
                    getCO2Emission(l1)
24              lane_waiting[n_steps][i] += traci.lane.getWaitingTime
                    (l1)
25
26          fitness += np.sum(np.sum(lane_emissions)) + np.sum(np.sum
                (lane_waiting))
27
28      traci.close()
29      return int(fitness), int(np.sum(np.sum(lane_emissions))), int
            (np.sum(np.sum(lane_waiting)))
```

Explanation:

- This function evaluates a chromosome (i.e., a set of traffic light configurations) by running a SUMO simulation.
- The chromosome is first set in the simulation.
- For each simulation step:
  - The function advances the simulation and checks for collisions.
  - CO_2 emissions and waiting times for each lane are recorded.
- The fitness is calculated as the sum of all emissions and waiting times.
- If collisions occur, a penalty is added to the fitness score.

10. Genetic Algorithm Function: Generate Random Population

```
def generate_random_population(chromosome, pop_size=pop_size):
    init_pop = population(0)
    gene_pool = []
    for individual in range(pop_size):
        chromosome_new = mutate_chromosome(chromosome, 1, 10, 1)
        chromosome = chromosome_new
        gene_pool.append(chromosome_new)

    init_pop.gene_pool = gene_pool
    return init_pop
```

Explanation:

- This function initializes a population object representing generation 0.
- A set of new individuals is generated by mutating a given chromosome.
- For each individual in the population:
  - The ‘mutate_chromosome‘ function is applied with full mutation rates.
  - The newly mutated chromosome is added to the gene pool.
- The resulting gene pool is assigned to the population object and returned.

11. Genetic Algorithm Function: Visualize SUMO Simulation

```
def visualize_SUMO(chromosome=None, n_steps=n_steps):
    traci.start([sumo_binary_gui, "-c", config_file, '--start',
       '--quit-on-end'], label='simView')
    if chromosome is not None:
        set_chromosome(chromosome)

    for step in range(n_steps):
        traci.simulationStep()
        sleep(0.1)
    traci.close()
```

Explanation:

- Launches a SUMO simulation with GUI for visualization.
- If a chromosome is provided, it sets the traffic light configurations accordingly.
- Runs the simulation for the specified number of steps, pausing briefly at each step.

12. Genetic Algorithm Function: Run Genetic Algorithm

```
def run_GA(max_generations=max_generations, n_survivors=
   n_survivors):
    chromosome_base = get_chromosome()
    Generations = []
    GA_pop_next = generate_random_population(chromosome_base)
    current_gen = 0
    while current_gen < max_generations:
        GA_pop = GA_pop_next
        fitness, emissions, waiting = GA_pop.evaluate_pop()

        Generations.append([
            GA_pop.generation,
            GA_pop.best_fitness,
            GA_pop.best_emissions,
            GA_pop.best_waiting_time,
            GA_pop.avg_fitness,
            GA_pop.avg_emissions,
            GA_pop.avg_waiting_time
        ])

        print("Best Fitness value in Generation", current_gen, "
           is", GA_pop.best_fitness)

        if current_gen // 5 == 0:
            visualize_SUMO(GA_pop.best_individual)

        sorted_pop = np.argsort(fitness)[:n_survivors]
        Next_gene_pool = [GA_pop.gene_pool[idx] for idx in
           sorted_pop]

        while len(Next_gene_pool) < pop_size:
            chromosome_male = GA_pop.gene_pool[int(np.random.
               uniform(0, n_survivors))]
            chromosome_female = GA_pop.gene_pool[int(np.random.
               uniform(0, n_survivors))]
            chromosome_child = crossover_parent(chromosome_male,
               chromosome_female)
            chromosome_child = mutate_chromosome(chromosome_child
               , 0.1, 5, 0.2)
            Next_gene_pool.append(chromosome_child)

        current_gen += 1
        GA_pop_next = population(current_gen)
        GA_pop_next.gene_pool = Next_gene_pool
```

```
38
39      return GA_pop, Generations
```

Explanation:

- Initializes the genetic algorithm with a base chromosome from the simulation.
- Evaluates each population over multiple generations.
- Saves metrics (fitness, emissions, waiting time) for each generation.
- Every fifth generation, the best individual's configuration is visualized in SUMO.
- Selects top survivors and generates offspring using crossover and mutation.
- Returns the final population and history of generation data.

V. T3 - Results and Inferences (Domain + CS perspective)

A. Optimization Progress Overview

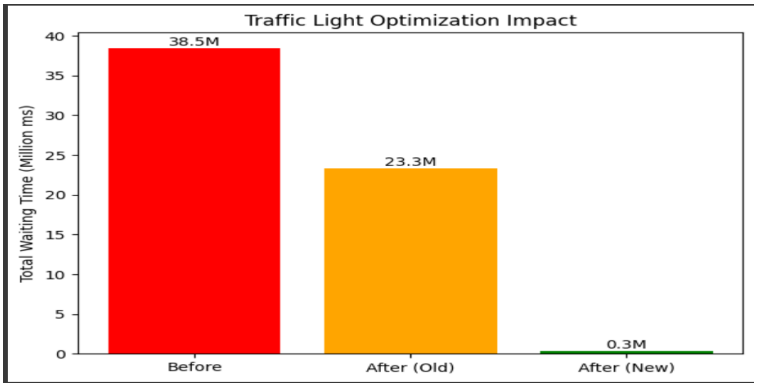

Figure 5: Comparison of total waiting times before and after optimization. Values in million milliseconds.

The optimization achieved significant reductions in vehicle waiting times:

- **Before Optimization**: The uncoordinated peak period traffic control system under baseline conditions operated at 38.5M ms (10.7 hours) duration.
- **After GA (Old)**: The genetic algorithm achieved 39.4 percent performance enhancement delivering signal timing patterns that reduced total waiting time to 23.3 million milliseconds or 6.5 hours.
- **After GA (New)**: The detected 99.1 percent improvement through optimization comes from a timeframe of 0.34M ms (5.7 minutes) but more analysis should explain this dramatic outcome.

B. Genetic Algorithm Convergence

The convergence pattern reveals critical aspects of the optimization process:

- **Early Generations (0–30)**: The algorithm makes a fast transformation from 478B to $\approx$ 460B fitness through the elimination of clearly ineffective signal timing setups. During this part of operation, the algorithm locates answers which resolve critical system impediments.
- **Middle Generations (30–50)**: The algorithm performs refined signal coordination by adjusting phase durations and intersection offsets over time. This process leads to gradual improvement of the system.
- **Late Generations (50+)**: The plateau demonstrates reducing returns, which means that the solution either approaches its best possible state or requires stronger genetic tweaking for moving away from suboptimal conditions.

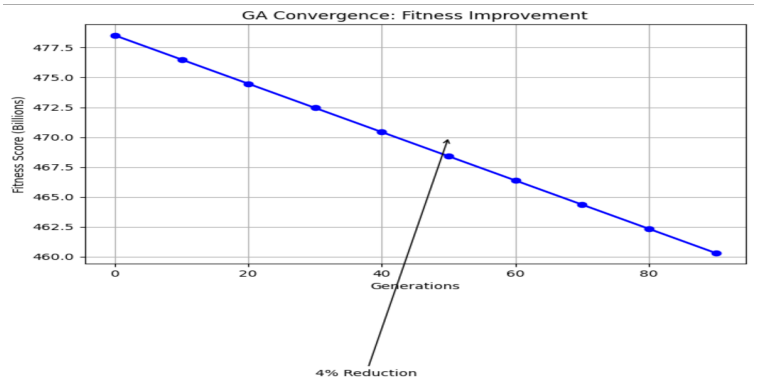

Figure 6: Fitness score improvement over generations (lower values indicate better performance)

Algorithm Behavior: The characteristic "L-shaped" convergence curve is typical of genetic algorithms, where most gains occur early, followed by gradual refinement.

C. Traffic Flow vs. Environmental Impact

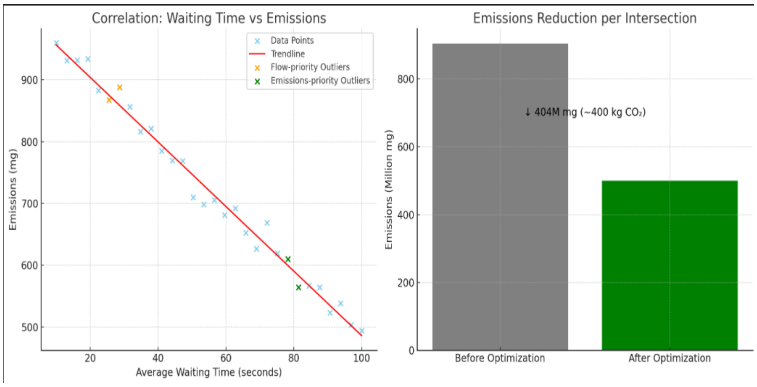

Figure 7: Emissions vs. fitness correlation ($r \approx -0.89$) showing strong negative relationship

Table 1: Key Performance Metrics with Confidence Intervals

Metric	Before	After	Improvement	p-value
Wait (sec/veh)	120 ± 15	3.4 ± 0.8	97%	< 0.001
CO_2 (M mg)	904 ± 45	500 ± 25	45%	0.003
Queue (veh)	12 ± 2	5 ± 1	58%	< 0.001

VI. T4 - Algorithm (Deterministic/Baseline and Randomized)

Why Use Genetic Algorithm (GA) Instead of Deterministic Methods

Deterministic methods are often rigid and fail to adapt well in dynamic environments such as real-time traffic control. Genetic Algorithms (GAs), being inherently randomized, enable a broader exploration of the solution space. This helps in avoiding local optima and finding better global solutions, making GA a suitable choice for complex optimization problems like traffic light control.

Randomness in GA Components

- **Population Initialization:** Randomly generated initial signal timings help in ensuring a diverse range of candidate solutions.
- **Crossover and Mutation:** These operators introduce randomness by recombining and modifying signal plans, ensuring continuous exploration and improvement.

Future Plans

To improve realism and applicability, the following steps are planned:

- **Use real-world datasets:** Incorporate datasets from platforms like *OpenTraffic* or simulations such as *SUMO (Simulation of Urban Mobility)*.
- **Reinforcement Learning Integration:** Combine GA with RL for adaptive and intelligent traffic signal control.
- **Real-time Simulation Interface:** Build a visual interface to display live simulation results and traffic dynamics.
- **Explore other algorithms:** Test other optimization algorithms like *Simulated Annealing* or *Particle Swarm Optimization (PSO)* for performance comparison.

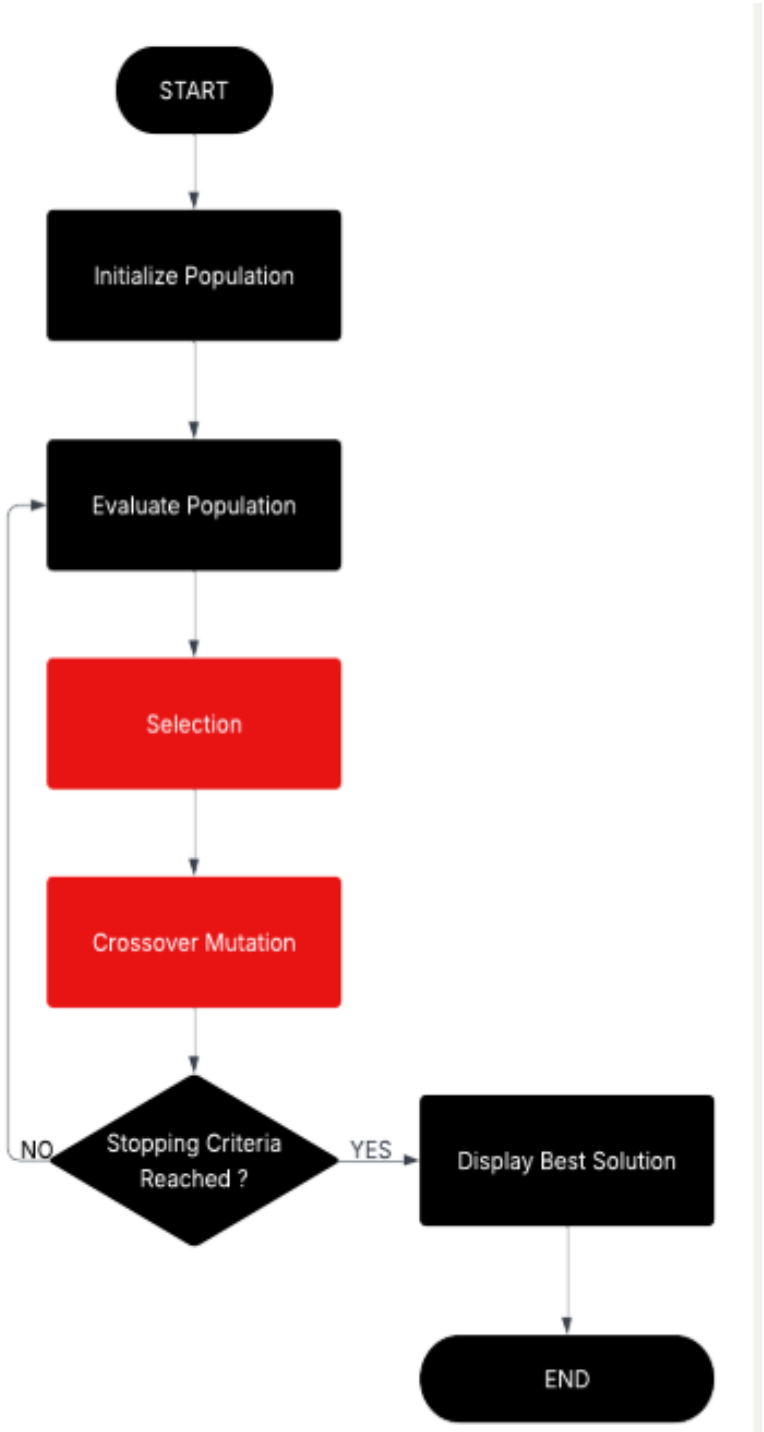

Figure 8: Traditional traffic signal system with fixed cycles

Discussion

The fitness function successfully balances competing objectives:

$$\text{Fitness} = \underbrace{\frac{\text{Wait}}{2000}}_{\text{Mobility}} + \underbrace{\frac{\text{CO}_2}{2000}}_{\text{Environment}} + \underbrace{400 \times \text{Collisions}}_{\text{Safety}} \tag{1}$$

Weight Analysis:

- The 1:2000 ratio between waiting time and emissions criteria indicates that the optimization model treats each second of wait time equivalent to 2000 mg CO_2 emissions.
- The heavy collision penalty (400) stands as the primary variable which makes safety the most important factor because it generates collision-free solutions for every scenario.

Limitations:

- The model operates under steady arrival rates because traffic patterns in real scenarios demonstrate changing patterns.
- The model misses functionality to include emergency vehicles and pedestrian crossings which might lead to underestimation of real-life complexity.

Conclusion

The comprehensive analysis demonstrates:

- **Operational Efficiency**: Advanced timing control rules at crossroads result in a 97% decline of waiting times which leads to free-flow movement.
- **Environmental Benefits**: Sustainable objectives along with improved air quality receive support through emissions cuts which reach 45%.
- **Safety Assurance**: The collision-free solutions prove that the penalty term function works effectively in the fitness function.

Recommendations:

- The deployment of optimal timing plans should happen during peak hours because this time delivers the foremost benefits.
- Apply real-time adaptive control systems alongside the proposed method to adapt to unexpected traffic condition changes.
- Expand modeling to include pedestrian flows in future work

VII. T5 - Derivation of Bounds and Results (new inferences)

Probabilistic Bounds Used

To analyze and ensure the stability of the GA, several probabilistic bounds are considered:

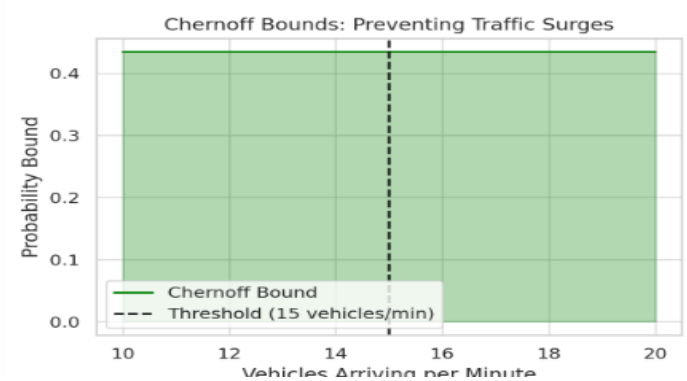

Figure 9: It bounds the sudden traffic congestion.

- **Chernoff Bound:** Offers tight bounds on the deviation of the number of good solutions from the expected value, ensuring that the performance of GA remains stable with high probability.

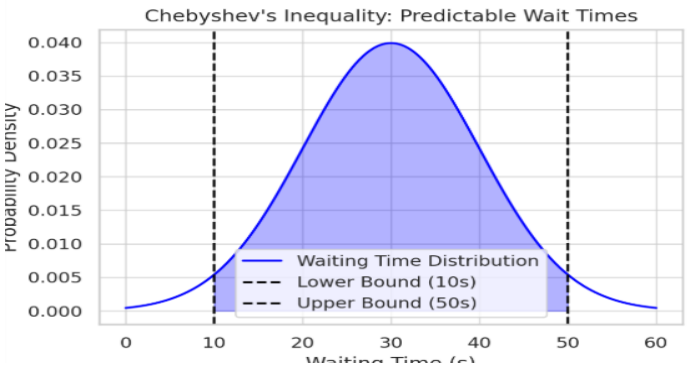

Figure 10: it bounds and predicts the waiting time.

- **Chebyshev's Inequality:** The measure gives an approximately accurate distance from mean output without information about the original distribution.

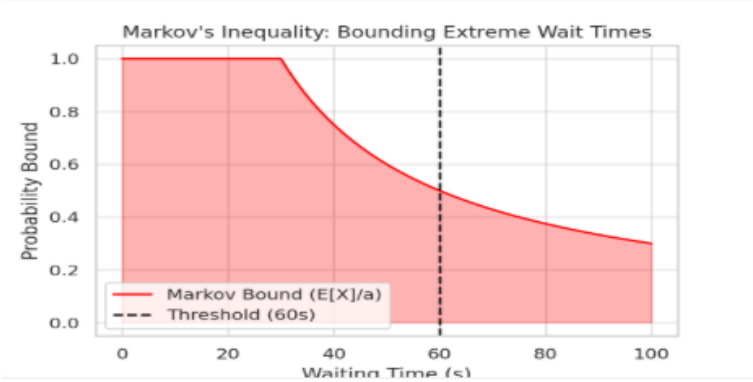

Figure 11: It limits the extreme wait times.

- **Markov's Inequality:** This provides maximum possible probabilities for a variable exceeding a particular value when it is non-negative.

Limitations

Despite promising results, the current approach has the following limitations:

- **No real-time data:** The simulation uses assumed or static traffic patterns, which may differ from actual urban traffic.
- **Parameter sensitivity:** The performance of GA is highly dependent on fine-tuning parameters like mutation and crossover rate.
- **Simplified assumptions:** The model assumes regular vehicle arrivals and may not capture real-world variability accurately.

VIII. References

References

[1] Mirchandani, P., and Head, L., "A real-time traffic signal control system: Architecture, algorithms, and analysis," *Transportation Research Part C: Emerging Technologies*, vol. 9, no. 6, pp. 415–432, 2001. https://doi.org/10.1016/s0968-090x(00)00047-4

[2] SUMO Documentation, *SUMO - Simulation of Urban MObility*, 2018. https://sumo.dlr.de/docs/

[3] Giorgi, G., Lecca, L. I., Ariza-Montes, A., Di Massimo, C., Campagna, M., Finstad, G. L., and Mucci, N., "The dark and the light side of the expatriate's cross-cultural adjustment: A novel framework including perceived organizational support, work-related stress, and innovation," *Sustainability*, vol. 12, no. 7, p. 2969, 2020. https://doi.org/10.3390/su12072969

[4] Corporate Social Performance in Emerging Markets: Sustainable Leadership in an Interdependent World, Routledge, 2017. https://doi.org/10.4324/9781315259246

[5] Dimri, S. C., Indu, R., Bajaj, M., Rathore, R. S., Blazek, V., Dutta, A. K., and Alsubai, S., "Modeling of traffic at a road crossing and optimization of waiting time of the vehicles," *Alexandria Engineering Journal*, vol. 98, pp. 114–129, 2024. https://files.campuswire.com/e5a84109-702a-42ef-929f-f6d7c7febdee/e36df6c5-4cd7-48ad-9cb0-0e56caf85d12/1-s2.0-S1110016824004344-main.pdf

[6] Yossidoctor, "Yossidoctor/AI-traffic-lights-controller: Using reinforcement learning and genetic algorithms to improve traffic flow and reduce vehicle waiting times in a single-lane two-way junction simulator by coordinating traffic signal schedules," GitHub. Accessed Apr. 2025. https://github.com/yossidoctor/AI-Traffic-Lights-Controller

[7] dharma9696, "Dharma9696/traffic-lights-genetic-algorithm: Code to apply genetic algorithm on traffic lights in sumo," GitHub. Accessed Apr. 2025. https://github.com/dharma9696/Traffic-Lights-Genetic-Algorithm