Traffic Signal Algorithm

Overview

The Traffic Signal Algorithm manages the flow of vehicles at an intersection with four lanes (North, East, South, and West). It prioritizes emergency vehicles and maintains a log of processed vehicles. The algorithm is implemented in C using the Kernighan & Ritchie (K&R) coding style for clarity and readability.

Features

- Vehicle Management: Add vehicles to lanes with specified priority and direction.
- Queue Management: Maintain queues for each lane to handle vehicle traffic efficiently.
- Emergency Vehicle Priority: Ensure that emergency vehicles are given precedence over regular vehicles.
- Traffic Log: Keep a log of all processed vehicles for future reference.
- User Interaction: Provide a menu for user interaction to create traffic, display vehicles, manage traffic, and print the traffic log.

Constraints and Edge Case Handling

- Vehicle ID: Must be a 3-digit number between 100 and 999.
- Priority: Must be either 0 (regular) or 1 (emergency).
- Direction: Must be one of 'N', 'E', 'S', or 'W'.
- Duplicate IDs: The system checks for and prevents the addition of duplicate vehicle IDs.
- Invalid Inputs: The algorithm handles invalid inputs and prompts the user to correct them.

Data Structures and Algorithms

This section details the data structures used in the traffic signal algorithm and their roles in the system.

Data Structures

1. Vehicle Structure

The `Vehicle` structure holds details about each vehicle, including its ID, priority, direction, and a pointer to the next vehicle. This structure is used to create nodes in the linked list for each lane's queue.

CODE

```
typedef struct Vehicle {
    int id;             // Vehicle ID
    int priority;       // 0 for regular, 1 for emergency
    char direction[4];  // "N", "E", "S", "W"
    struct Vehicle *next;
} Vehicle;
```

CODE

2. Queue Structure

The `Queue` structure is used to manage the vehicles in each lane. It is implemented as a simple linked list with pointers to the front and rear of the queue.

CODE

```
typedef struct {
    Vehicle *front;     // Pointer to the first vehicle in the queue
    Vehicle *rear;      // Pointer to the last vehicle in the queue
} Queue;
```

CODE

- Type: Simple Queue
- Implementation: Linked List
- Usage: Used for each of the four lanes (North, East, South, West).

3. Log Structure

The `Log` structure keeps a record of all processed vehicles. It is implemented as a linked list, where each node contains the details of a processed vehicle.

```
CODE
typedef struct Log {
    int id;             // Vehicle ID
    int priority;       // 0 for regular, 1 for emergency
    char direction[4];  // "N", "E", "S", "W"
    struct Log *next;   // Pointer to the next log entry
} Log;
CODE
```

Algorithms

1. Vehicle Enqueue

The `enqueue_vehicle` function adds vehicles to the appropriate lane's queue based on their direction and priority. Emergency vehicles are prioritized by maintaining an additional count.

2. Vehicle Dequeue

The `process_emergency_vehicles` function processes and dequeues emergency vehicles first, ensuring they are given precedence.

70% Criterion for Normal Cars Dequeue: If there are vehicles in other lanes, only up to 70% of the regular vehicles in the current lane will be dequeued. This ensures that traffic from other lanes also gets a chance to move, preventing starvation of any single lane.

Detailed Function and Variable Documentation

Vehicle Structure

Represents a vehicle with an ID, priority, direction, and a pointer to the next vehicle in the queue.

- id: Unique identifier for each vehicle.
- priority: Indicates the priority of the vehicle (0 for regular, 1 for emergency).
- direction: The direction of the vehicle ('N', 'E', 'S', 'W').
- next: Pointer to the next vehicle in the queue.

Queue Structure

Manages a list of vehicles in a lane with pointers to the front and rear vehicles.

- front: Points to the first vehicle in the queue.
- rear: Points to the last vehicle in the queue.

Log Structure

Records processed vehicle details including ID, priority, direction, and a pointer to the next log entry.

- id: Unique identifier for each vehicle.
- priority: Indicates the priority of the vehicle (0 for regular, 1 for emergency).
- direction: The direction of the vehicle ('N', 'E', 'S', 'W').
- next: Pointer to the next log entry.

Global Variables

- lanes: Array of four queues representing the lanes.
- traffic_log: Pointer to the head of the traffic log linked list.
- emergency_vehicle_count: Tracks the number of emergency vehicles in the queues.

Functions Documentation

`void create_lanes()`

Initializes the lanes as empty queues.

- lanes: Initializes the front and rear pointers of each lane to `NULL`.

`int is_empty(Queue *queue)`

Checks if a queue is empty.

- queue: The queue to check.

- Returns: 1 if the queue is empty, 0 otherwise.

`int get_lane_index(char direction[])`

Gets the lane index from the direction.

- direction: The direction of the vehicle ('N', 'E', 'S', 'W').

- Returns: The index of the lane (0 for North, 1 for East, 2 for South, 3 for West) or -1 for invalid direction.

`int vehicle_exists(int id)`

Checks if a vehicle with a given ID exists in any lane.

- id: The vehicle ID to check.

- Returns: 1 if the vehicle exists, 0 otherwise.

`void enqueue_vehicle(int id, int priority, char direction[])`

Adds a vehicle to the appropriate lane.

- id: Vehicle ID.

- priority: Vehicle priority (0 for regular, 1 for emergency).

- direction: Vehicle direction ('N', 'E', 'S', 'W').

- lane_index: The index of the lane.

- new_vehicle: Pointer to the newly created vehicle.

`void create()`

Creates traffic data by enqueuing vehicles based on user input.

- input: Input buffer for vehicle ID.

- id: Vehicle ID.

- priority: Vehicle priority.

- direction: Vehicle direction.

`void display()`

Displays all vehicles in all lanes.

- directions: Array of direction strings for display.

- temp: Temporary pointer to traverse the vehicle queue.

`int count_vehicles_in_lane(int lane_index)`

Counts the number of vehicles in a given lane.

- lane_index: The index of the lane.

- count: Counter for the number of vehicles.

- temp: Temporary pointer to traverse the vehicle queue.

`void add_to_log(int id, int priority, char direction[])`

Adds a vehicle to the log of processed vehicles.

- id: Vehicle ID.

- priority: Vehicle priority.

- direction: Vehicle direction.

- new_log: Pointer to the newly created log entry.

- temp: Temporary pointer to traverse the log linked list.

`void free_vehicles(Queue *lane)`

Frees the memory allocated for vehicles in a lane.

- lane: The queue to free.

- temp: Temporary pointer to traverse the vehicle queue.

`void process_emergency_vehicles(const char *directions[])`

Processes and dequeues emergency vehicles first.

- directions: Array of direction strings for display.

- emergency_count: Array to store emergency vehicle counts for each lane.

- total_emergency_vehicles: Total count of emergency vehicles.

- lane_order: Lane order array for prioritizing lanes.

- temp: Temporary pointer to traverse the vehicle queue.

- farthest_emergency: Pointer to the farthest emergency vehicle.

`void process_regular_vehicles(const char *directions[], int total_vehicles)`

Processes and dequeues regular vehicles.

- directions: Array of direction strings for display.

- total_vehicles: Total count of vehicles in all lanes.

- regular_vehicle_count: Count of regular vehicles in the current lane.

- other_lanes_have_vehicles: Indicates if other lanes have vehicles.

- allowed_to_dequeue: Number of vehicles allowed to dequeue.

- dequeued_count: Count of dequeued vehicles.

- temp: Temporary pointer to traverse the vehicle queue.

- dequeue_vehicle: Pointer to the dequeued vehicle.

`void manage_traffic()`

Manages overall traffic by coordinating emergency and regular vehicle processing.

- total_vehicles: Total count of vehicles in all lanes.

- directions: Array of direction strings for display.

`void print_traffic_log()`

Prints the log of processed vehicles.

- temp: Temporary pointer to traverse the log linked list.

`void free_log()`

Frees the memory allocated for the log.

- temp: Temporary pointer to traverse the log linked list.

`int main()`

Main function: provides a menu for user interaction.

- choice: User's menu choice.

- input: Input buffer for user's choice.

Conclusion

The Traffic Signal Algorithm is a robust and efficient solution for managing vehicle flow at intersections. By utilizing linked lists for queue management and prioritizing emergency vehicles, it ensures a smooth and orderly traffic system. The detailed documentation and adherence to the K&R coding style enhance the readability and maintainability of the code, making it accessible for further development and adaptation. The implementation considers various constraints and handles edge cases effectively, ensuring reliable operation in diverse traffic scenarios.