Navisense

Ride Matching & Optimization Algorithm

OVERVIEW: The Ride Matching & Optimization Algorithm connects users with suitable ride-sharing options by considering real-time traffic data and individual preferences. Its goal is to enhance user experience through efficient, cost-effective, and eco-friendly transportation solutions.

1. Data Collection:

- ➔ Traffic Data:
  - ◆ Integrate with traffic APIs (e.g., Google Maps, Waze) to gather real-time traffic conditions, including congestion levels and estimated travel times.
- ➔ User Preferences:
  - ◆ Create user profiles capturing preferences such as maximum fare, preferred travel time, and eco-friendliness (e.g., preference for electric vehicles).

2. Preprocessing:

- ➔ Data Normalization:
  - ◆ Normalize traffic data for consistency (e.g., converting all travel times to minutes).
- ➔ User Preference Weighting:
  - ◆ Assign weights to user preferences based on importance (e.g., cost may be prioritized over travel time for budget-conscious users).

3. Route Generation:

- ➔ Initial Route Calculation:
  - ◆ Calculate potential routes from the user's location to their destination using a shortest-path algorithm (e.g., Dijkstra’s or A *).
- ➔ Traffic Adjustment:
  - ◆ Adjust routes based on real-time traffic data to avoid congestion and minimize travel time.

4. Ride Matching:

- ➔ Available Rides Database:
    - ◆ Maintain a database of available rides, including driver location, estimated time of arrival (ETA), and user ratings.
- ➔ Matching Algorithm:
    - ◆ Implement a matching algorithm that considers proximity, ETA, and user preferences to calculate a match score for each ride.

5. Optimization:

- ➔ Multi-Criteria Decision Making:
    - ◆ Use a decision-making approach to rank available rides based on match scores and user preferences.
- ➔ Dynamic Re-Routing:
    - ◆ Continuously monitor traffic and user location, dynamically re-routing to better options as they become available.

6. User Notification:

- ➔ Real-Time Updates:
    - ◆ Notify users of their matched ride, including driver information, vehicle type, ETA, and cost.
- ➔ Feedback Loop:
    - ◆ Prompt users for feedback post-ride to improve future matching and optimization.

7. Machine Learning Integration:

- ➔ Predictive Analytics:
    - ◆ Implement machine learning models to analyze historical data and predict traffic patterns and ride availability.
- ➔ Continuous Learning:
    - ◆ Use user feedback to refine the matching algorithm, enhancing accuracy and satisfaction over time.

Conclusion: The Ride Matching & Optimization Algorithm leverages real-time data and user preferences to enhance the ride-sharing experience. Adapting to changing conditions and user feedback aims to provide efficient, cost-effective, and eco-friendly transportation options, contributing to a more sustainable urban mobility ecosystem.

Pseudocode for Navisense

1. Ride Matching Algorithm

```
FUNCTION matchRides(userLocation, destination, userPreferences):
   // Step 1: Gather real-time traffic data for the route
   trafficData = getRealTimeTrafficData(userLocation, destination)

   // Step 2: Retrieve available rides near the user's location heading to the destination
   availableRides = getAvailableRides(userLocation, destination)

   // Initialize variables to track the best ride found
   bestRide = null
   minTime = INFINITY  // Set initial minimum travel time to infinity
   minCost = INFINITY  // Set initial minimum cost to infinity

   // Step 3: Evaluate each available ride
   for each ride in availableRides:
      // Calculate the estimated travel time for the current ride considering traffic
      travelTime = calculateTravelTime(ride, trafficData)

      // Calculate the cost of the current ride based on user preferences
      rideCost = calculateRideCost(ride, userPreferences)

      // Step 4: Check if the current ride is better than the best found so far
      if (travelTime < minTime) and (rideCost <= minCost):
         // Update the best ride if the current ride is faster and within budget
         bestRide = ride
         minTime = travelTime  // Update minimum travel time
         minCost = rideCost    // Update minimum cost

   // Step 5: Return the best ride found
   return bestRide
```

2. Parking Assistance Algorithm

```
FUNCTION findParking(userLocation, destination):
   // Step 1: Gather parking data for the area around the user's location and destination
   parkingData = getParkingData(userLocation, destination)

   // Step 2: Filter the parking data to find available parking spots
   availableParkingSpots = filterAvailableParking(parkingData)

   // Initialize variables to track the best parking spot found
   bestParkingSpot = null
   minDistance = INFINITY  // Set initial minimum distance to infinity
```

```
    // Step 3: Evaluate each available parking spot
    for each spot in availableParkingSpots:
        // Calculate the distance from the user's location to the current parking spot
        distanceToParking = calculateDistance(userLocation, spot)

        // Step 4: Check if the current parking spot is closer than the best found so far
        if distanceToParking < minDistance:
            // Update the best parking spot if the current spot is closer
            bestParkingSpot = spot
            minDistance = distanceToParking  // Update minimum distance

    // Step 5: Return the best parking spot found
    return bestParkingSpot
```

3. Fuel-Efficient Navigation Algorithm

```
FUNCTION getOptimizedRoute(userLocation, destination, userPreferences):
    // Step 1: Gather real-time traffic data for the route
    trafficData = getRealTimeTrafficData(userLocation, destination)

    // Step 2: Gather road type data for the route
    roadData = getRoadTypeData(userLocation, destination)

    // Initialize variables to track the optimal route found
    optimalRoute = null
    minFuelConsumption = INFINITY  // Set initial minimum fuel consumption to infinity

    // Step 3: Retrieve all possible routes from the user's location to the destination
    possibleRoutes = getPossibleRoutes(userLocation, destination)

    // Step 4: Evaluate each possible route
    for each route in possibleRoutes:
        // Calculate the fuel consumption for the current route based on traffic and road data
        fuelConsumption = calculateFuelConsumption(route, trafficData, roadData)

        // Step 5: Check if the current route has lower fuel consumption than the best found so far
        if fuelConsumption < minFuelConsumption:
            // Update the optimal route if the current route is more fuel-efficient
            optimalRoute = route
            minFuelConsumption = fuelConsumption  // Update minimum fuel consumption

    // Step 6: Return the optimal route found
    return optimalRoute
```

❖ Main Algorithm for Navisense (Integrating all components)

```
FUNCTION main():
   // Step 1: Initialize the application
   initializeApp()

   // Step 2: Check user authentication
   IF userIsAuthenticated():
      displayDashboard()
   ELSE:
      displayLoginPage()

FUNCTION initializeApp():
   // Load necessary resources (CSS, JS, etc.)
   loadResources()
   // Set up event listeners for user interactions
   setupEventListeners()

FUNCTION userIsAuthenticated():
   // Check if the user is logged in
   RETURN checkSession()

FUNCTION displayLoginPage():
   // Show the login form
   renderLoginForm()
   // Handle login submission
   ON submitLoginForm:
      username = getInput("username")
      password = getInput("password")
      IF authenticateUser (username, password):
         redirectToDashboard()
      ELSE:
         showError("Invalid credentials")

FUNCTION displayDashboard():
   // Load user-specific data
   userData = fetchUser Data()
   // Render the dashboard with user data
   renderDashboard(userData)
   // Set up event listeners for dashboard interactions
   setupDashboardEventListeners()

FUNCTION fetchUser Data():
   // Make an API call to retrieve user data
   RETURN apiCall("GET", "/api/user/data")
```

```
FUNCTION authenticateUser (username, password):
   // Make an API call to authenticate the user
   response = apiCall("POST", "/api/auth/login", { "username": username, "password":
password })
   RETURN response.success

FUNCTION renderLoginForm():
   // Render the HTML for the login form
   DISPLAY loginFormHTML

FUNCTION renderDashboard(userData):
   // Render the dashboard with user data
   DISPLAY dashboardHTML WITH userData

FUNCTION apiCall(method, endpoint, data = null):
   // Make an HTTP request to the specified endpoint
   IF method == "GET":
      RETURN httpGet(endpoint)
   ELSE IF method == "POST":
      RETURN httpPost(endpoint, data)

FUNCTION setupEventListeners():
   // Set up event listeners for common actions
   ON click "loginButton":
      submitLoginForm()

FUNCTION setupDashboardEventListeners():
   // Set up event listeners for dashboard actions
   ON click "logoutButton":
      logoutUser ()

FUNCTION logoutUser ():
   // Make an API call to log out the user
   apiCall("POST", "/api/auth/logout")
   redirectToLoginPage()
FUNCTION redirectToDashboard():
   // Redirect the user to the dashboard page
   navigateTo("/dashboard")

FUNCTION redirectToLoginPage():
   // Redirect the user to the login page
   navigateTo("/login")

FUNCTION showError(message):
   // Display an error message to the user
   DISPLAY errorMessage(message)
```

❖ System Flow:

1. **User Input:** The user provides their current location and desired destination, along with preferences (e.g., fastest route, least expensive, eco-friendly).
2. **AI and IoT Data Gathering:**
    - The system gathers real-time data on traffic, available rides, parking spaces, and route information using external APIs and IoT sensors.
3. **Optimization:**
    - The algorithms use this data to suggest the best ride-sharing option, the most convenient parking space, and the most fuel-efficient route.
4. **Output:**
    - The user is presented with an optimized solution, which includes the best ride, parking spot, and route to take.