Direction – route planning

Jobs – task, setup & service time, (pickup, delivery, …), ~~amount~~

Vehicles – capacity, skills.

Specific for my use-case.

1. **BotFleetManager**: Manages the fleet of bots.
   * Attributes: list of bots, available material, etc.
   * Methods: assignRepairTask(), monitorProgress(), etc.
2. **RepairBot**: Represents a single bot in the fleet.
   * Attributes: vehicle profile, current location, material capacity, etc.
   * Methods: travelToLocation(), repairPothole(), refillMaterial(), etc.
3. **PotholeRepairTask**: Details of a pothole repair task.
   * Attributes: location coordinates, road defect type, defect dimensions, etc.
   * Methods: calculateMaterialRequired(), estimateRepairTime(), etc.
4. **RoutePlanner**: Utilizes OpenRouteService for route planning.
   * Attributes: ORS API credentials, current area, etc.
   * Methods: generateTravelRoute(), calculateTravelTime(), etc.
5. **MaterialManager**: Manages the filling materials.
   * Attributes: type of materials, quantity, etc.
   * Methods: checkMaterialAvailability(), refillBot(), etc.
6. **ReportingTool**: Generates reports and comparison charts.
   * Attributes: performance data, comparison metrics, etc.
   * Methods: generateTravelRoutesReport(), compareTimelines(), etc.

Relationships:

* **BotFleetManager** has a one-to-many relationship with **RepairBot**.
* **RepairBot** has a one-to-one relationship with **PotholeRepairTask**.
* **BotFleetManager** uses **RoutePlanner** for planning routes.
* **BotFleetManager** interacts with **MaterialManager** to manage material refills.
* **ReportingTool** uses data from **BotFleetManager** and **RepairBot**.

**Activity Diagram**

An activity diagram will depict the workflow of your system. Here's a high-level flow:

1. **Start**: The process begins.
2. **Input Pothole Details**: Including location, defect type, and dimensions.
3. **Plan Route using ORS**: RoutePlanner calculates the optimal route for ‘n’ bots.
4. **Assign Repair Task to Bot**: BotFleetManager assigns tasks to available RepairBots.
5. **Travel to Location**: RepairBot travels to the pothole location.
6. **Check Material Level**: RepairBot checks if a refill is needed.
7. **Repair Pothole**: RepairBot performs the repair task.
8. **~~Refill Material~~** ~~RepairBot travels back to station and gets refilled.~~
9. **Update Task Status**: RepairBot updates the task status upon completion.
10. **Generate Reports**: ReportingTool generates travel routes and comparison charts.
11. **End**: The process completes.

To leverage and extend ORS features, focus on integrating ORS more deeply into the RoutePlanner class. You might need to create custom features for route optimization based on bot locations, pothole locations, and repair priority. These custom features can be added to the ORS open-source project if they're not already available.

Remember, these diagrams are starting points. You might need to refine and expand them as your project progresses and as you get a better understanding of ORS's capabilities. For implementation, Python is a suitable choice, and you can use UML tools for creating these diagrams.

Logic – extend vehicle (& job) class, and create instances.

Blackbox\_Future\_scope:

Bot1\_loc = get\_sensor\_data(GPS)

Send\_to(mgr, Bot1\_loc)

Bot1.go\_to\_(GPS)

# SLAM algorithm: Simultaneous Localization and Mapping

Pothole\_2 – material needed – 5kg

Bot\_2 – material capacity – 20kg

Job1: Pothole\_1 – material needed – 45kg

Bot\_1 – material capacity – 20kg

Job1:

* + pothole1 – material needed: 20kg
  + pothole1 – material needed: 20kg
  + pothole1 – material needed: 5kg

[dig, pour, level]

1. **Problem Definition:**
   * Develop a software framework that enables UGVs to autonomously navigate through custom 2D static environments to automate pothole repairing process for citywide planning and maintenance.
2. **Data Collection:**
   * Download map for a selected postcode.
   * Use Government dataset (crowdsourced).
   * Filter potholes in that area.
3. Data Preparation & Resource Planning:
   * Calculate estimates:
     + material required,
     + time to fix,
     + no of bots.x
4. **Depot Setup**:
   * Establish a central depot or station where bots can start and return for maintenance, refilling, and recharging.
   * The depot should have facilities for asphalt-mix preparation, storage, and charging stations. – assume unlimited supply of material and fuel from station.
5. **Route Planning:**
   * Configure homogeneous fleet. (bot criteria & design)
     + Define the characteristics of your “pothole pro”-like bots (UGVs). Consider factors such as speed, payload capacity, digging mechanism, and energy source.
     + Add sensors: GPS (for real time updates), camera, LiDAR (for object detection and avoidance.
   * Plan the routes for each UGV bot.
   * Assign tasks.
6. **Execution**:
   * Report updates to fleet manager at regular intervals, and at start and end of every job.
7. **Simulation Environment**:
   * **Webots**: Ideal for simulating the physical behaviour of UGVs, including sensors, actuators, and interactions with the environment.
   * Set up the environment with the neighbourhood layout, roads, and pothole coordinates.
8. **Simulation Flow**:
   * Initialize the bot fleet at the depot.
   * Bots receive a list of pothole coordinates from the dataset.
   * Each bot follows these steps:
     + Move to the specified pothole location.
     + Use its digging mechanism to remove damaged asphalt.
     + Fill the pothole with prepared asphalt-mix.
     + Level the surface.
     + Move to the next pothole.
     + Repeat until all potholes are repaired.
     + Return to the depot when low on energy or asphalt.
   * Implement realistic constraints (speed, energy consumption, etc.) in the simulation.
   * **Continuous monitoring:**
     + Update location, battery status, job-status at regular intervals and at start and end of every repair job.
     + Monitor bot energy levels (and health) during the simulation.
     + Re-distribute workload if a bot goes down (dies).
9. **Data Collection and Metrics**:
   * Record repair times for each pothole.
   * Measure the fleet’s efficiency (number of potholes repaired per unit time).
   * ~~Evaluate the impact on traffic flow during repairs.~~
10. **Optimization and Scheduling**:
    * Optimize the fleet’s routes to minimize travel time.
    * Prioritize urgent potholes based on severity and location.
    * Schedule maintenance and refilling to minimize downtime.
11. **Visualization and Reporting**:
    * Visualize the fleet’s movements and repair actions.
    * Generate reports on repair progress, energy usage, and overall efficiency.

Remember that real-world deployment will involve additional challenges (e.g., sensor accuracy, weather conditions, traffic management). Simulations provide valuable insights, but field testing is essential for validation. Good luck with your project! 🛠️🚧🤖

Certainly! Let’s break down the steps to create your Webots simulation for the pothole-repairing fleet. We’ll use a **Publisher-Subscriber (Pub-Sub)** model for real-time monitoring and updates. I’ll provide a high-level overview, and you can adapt it to your specific needs.

1. **Setting Up Your Webots Environment**:
   * Download and install Webots if you haven’t already.
   * Create a new Webots project or open an existing one.
   * Import the OSM map file (e.g., map.osm) to create the neighbourhood environment.
2. **Bot Model and Behaviour**:
   * Design your “pothole pro”-like bot model in Webots. You can use a JCB-like 3D model or create a custom one.
   * Define the bot’s sensors (e.g., lidar, camera, GPS) and actuators (e.g., wheels, digging mechanism).
   * Implement the bot’s behaviour:
     + Navigate to potholes using GPS coordinates.
     + Detect potholes using sensors.
     + Repair potholes (dig-fill-level).
     + Return to the depot when needed.
3. **FleetManager (Publisher)**:
   * Create a FleetManager controller (Python or C++) that manages the fleet of bots.
   * The FleetManager:
     + Keeps track of bot states (idle, repairing, returning).
     + Publishes updates (e.g., bot status, repair progress) to subscribers.
     + Communicates with individual bots via custom messages.
4. **Bot Controllers (Subscribers)**:
   * Each bot has its own controller (Python or C++).
   * The bot controllers:
     + Subscribe to updates from the FleetManager.
     + Receive GPS coordinates of potholes.
     + Execute repair actions (dig, fill, level).
     + Report back to the FleetManager.
5. **Communication Channels**:
   * Use Webots’ built-in communication channels (e.g., Emitter, Receiver, Supervisor).
   * Set up a custom message format (e.g., JSON) for communication between the FleetManager and bots.
6. **Simulation Loop**:
   * In the FleetManager:
     + Load the OSM map and extract pothole coordinates.
     + Initialize bot positions and states.
     + Publish initial bot states.
   * In each bot controller:
     + Subscribe to the FleetManager.
     + Continuously:
       - Receive updates (e.g., new pothole coordinates).
       - Navigate to the pothole.
       - Repair the pothole.
       - Update status (e.g., “repairing,” “returning”).
       - Communicate back to the FleetManager.
7. **Sensor Data Collection**:
   * Each bot collects sensor data (e.g., lidar scans, camera images).
   * Use this data for pothole detection and navigation.
   * Publish relevant sensor data to the FleetManager.
8. **Real-Time Monitoring and Updates**:
   * The FleetManager receives bot updates and sensor data.
   * Monitor repair progress, energy levels, and bot health.
   * Visualize the fleet’s actions in Webots (e.g., highlight repaired potholes).
9. **Charging and Refilling**:
   * Implement logic for bots to return to the depot when low on energy or asphalt.
   * Refill asphalt-mix and recharge batteries as needed.
10. **Testing and Optimization**:
    * Run simulations with different fleet sizes and repair strategies.
    * Optimize bot routes to minimize repair time.
    * Evaluate the efficiency of your fleet.

Remember to adapt the code snippets below to your specific bot model and communication setup. Here’s a simplified example in Python:

# FleetManager.py

# Pseudo-code for FleetManager

class FleetManager:

def \_\_init\_\_(self):

# Initialize bot positions, states, and communication channels

# Load pothole coordinates from OSM map

def publish\_bot\_state(self, bot\_id, state):

# Publish bot state to subscribers

def handle\_sensor\_data(self, bot\_id, sensor\_data):

# Process sensor data (lidar, camera) from bots

# Detect potholes and update repair progress

# Other methods for managing the fleet

# BotController.py

# Pseudo-code for individual bot controller

class BotController:

def \_\_init\_\_(self, bot\_id):

# Initialize bot parameters and communication channels

def subscribe\_to\_fleet\_manager(self):

# Subscribe to FleetManager updates

def navigate\_to\_pothole(self, pothole\_coords):

# Use GPS data to navigate to pothole

def repair\_pothole(self):

# Execute repair actions (dig, fill, level)

* [Repair Planning](https://explorer.globe.engineer/?q=pothole+repair+process+strategizing+using+fleet+of+UGV+bots+in+a+neighbourhood#Repair%20Planning)
  + └── [Resource Allocation](https://explorer.globe.engineer/?q=pothole+repair+process+strategizing+using+fleet+of+UGV+bots+in+a+neighbourhood#Resource%20Allocation)
    - └── [UGV Fleet Deployment](https://explorer.globe.engineer/?q=pothole+repair+process+strategizing+using+fleet+of+UGV+bots+in+a+neighbourhood#UGV%20Fleet%20Deployment)
    - ├── [Material Logistics](https://explorer.globe.engineer/?q=pothole+repair+process+strategizing+using+fleet+of+UGV+bots+in+a+neighbourhood#Material%20Logistics)
  + ├── [Scheduling Optimization](https://explorer.globe.engineer/?q=pothole+repair+process+strategizing+using+fleet+of+UGV+bots+in+a+neighbourhood#Scheduling%20Optimization)
    - │ └── [Route Planning](https://explorer.globe.engineer/?q=pothole+repair+process+strategizing+using+fleet+of+UGV+bots+in+a+neighbourhood#Route%20Planning)
    - ├── [Real-Time Monitoring](https://explorer.globe.engineer/?q=pothole+repair+process+strategizing+using+fleet+of+UGV+bots+in+a+neighbourhood#Real-Time%20Monitoring)
* [Execution Phase](https://explorer.globe.engineer/?q=pothole+repair+process+strategizing+using+fleet+of+UGV+bots+in+a+neighbourhood#Execution%20Phase)
  + └── [UGV Operation](https://explorer.globe.engineer/?q=pothole+repair+process+strategizing+using+fleet+of+UGV+bots+in+a+neighbourhood#UGV%20Operation)
    - └── [Automated Pothole Filling](https://explorer.globe.engineer/?q=pothole+repair+process+strategizing+using+fleet+of+UGV+bots+in+a+neighbourhood#Automated%20Pothole%20Filling)
    - ├── [Quality Control Checks](https://explorer.globe.engineer/?q=pothole+repair+process+strategizing+using+fleet+of+UGV+bots+in+a+neighbourhood#Quality%20Control%20Checks)
  + ├── [Community Engagement](https://explorer.globe.engineer/?q=pothole+repair+process+strategizing+using+fleet+of+UGV+bots+in+a+neighbourhood#Community%20Engagement)
    - │ └── [Notification System](https://explorer.globe.engineer/?q=pothole+repair+process+strategizing+using+fleet+of+UGV+bots+in+a+neighbourhood#Notification%20System)
    - ├── [Feedback Collection](https://explorer.globe.engineer/?q=pothole+repair+process+strategizing+using+fleet+of+UGV+bots+in+a+neighbourhood#Feedback%20Collection)

**Framework:**

1. **Project Root Directory (e.g., PotholeRepairProject)**:

* **worlds/**:
  + **main\_world.wbt**: Main simulation world file with the initial setup.
* **protos/**:
  + **Pothole.proto**: Custom pothole representation.
  + **RepairBot.proto**: Custom robot PROTO for repairing.
* **controllers/**:
  + **repair\_bot\_controller/**: Controller script for individual repair bots.
    - **repair\_bot\_controller.py**: Main controller script for the bots.
  + **fleet\_manager\_controller/**: Controller script for the fleet manager.
    - **fleet\_manager\_controller.py**: Main controller script for managing the fleet.
* **data/**:
  + **neighbourhood\_map.osm**: OSM map file of the neighborhood.
  + **pothole\_data.csv**: Dataset with pothole locations and dimensions.
  + **Sensor\_data**
    - GPS (+ temporal)
    - LiDAR
    - Camera images – for evidence of job completion
* **scripts/**:
  + **pothole\_importer.py**: Import pothole data into Webots.
  + **map\_importer.py**: Import OSM map into Webots.
  + **route\_planner.py**: Interface with openrouteservice for route planning.
* **resources/**: Additional resources like textures, sounds, etc.
* **README.md**: Project documentation.
* **.gitignore**: For version control.

**Key Components and Implementation Details**

1. **Fleet Manager (Controller and Logic)**:
   * Implement **fleet\_manager\_controller.py** to assign tasks and manage the fleet.
   * Integrate with openrouteservice API for route planning.
   * Manage bots' payload capacity, material levels, and fuel/battery status.
   * Communicate with bots to send tasks and receive status updates.
2. **Repair Bots (Controller and Payload Management)**:
   * **repair\_bot\_controller.py** handles navigation to pothole locations and performing repair tasks.
   * Track and report payload capacity, material usage, and battery status to the fleet manager.
   * Handle instructions from the fleet manager, including task assignments and return-to-station commands.
3. **User Configuration**:
   * Allow user to configure the number of bots in the fleet and the starting point (station location) at the beginning of the simulation.
4. **Pothole Repair Logic**:
   * Calculate service time based on pothole size, with a constant setup time.
5. **Data and Map Handling**:
   * Use **pothole\_data.csv** for pothole locations and dimensions.
   * **map\_importer.py** and **route\_planner.py** scripts to process the OSM map and plan routes.
6. **Simulation Setup**:
   * The **main\_world.wbt** file should include the environment setup, bots, and the station.
7. **Documentation**:
   * Provide comprehensive instructions and descriptions in **README.md**.

Certainly! Designing a comprehensive framework for route planning, scheduling, and real-time monitoring of a fleet of Unmanned Ground Vehicles (UGVs) for pothole repair involves several key components. Let’s break it down:

1. **Load Planning and Scheduling**:
   * **Load Planning**: Optimize the available space within each UGV to maximize the number of pothole repair materials (such as asphalt, tools, and safety equipment) that can be carried.
   * **Scheduling**: Create efficient schedules for UGVs based on factors like repair urgency, location, and available resources. Consider real-time traffic conditions and road closures.
2. **Route Planning and Optimization**:
   * **Route Planning**: Determine the most efficient routes for UGVs to reach pothole repair sites. Consider factors like road conditions, traffic, and proximity to other repair locations.
   * **Real-Time Optimization**: Continuously update routes based on real-time data (e.g., traffic congestion, accidents) to minimize travel time and fuel consumption.
3. **Real-Time Monitoring**:
   * **Vehicle Tracking**: Use GPS and telematics to monitor UGV locations in real time. This helps ensure they follow planned routes and respond promptly to changes.
   * **Sensor Data**: Equip UGVs with sensors (e.g., lidar, cameras) to detect potholes, assess road conditions, and identify obstacles.
   * **Health Monitoring**: Monitor UGV health (battery levels, mechanical issues) to prevent failures during missions.
4. **Load Balancing and Redundancy**:
   * **Load Balancing**: Distribute repair tasks evenly among UGVs to optimize resource utilization. Avoid overloading any single UGV.
   * **Redundancy**: Have backup UGVs ready in case of failures. Automatically reassign tasks to available UGVs if one malfunctions.
5. **Communication and Coordination**:
   * **Central Control Center**: Establish a control center to manage UGV operations. Receive real-time data, adjust schedules, and communicate with UGVs.
   * **Inter-UAV Communication**: Enable UGVs to communicate with each other for collaborative repair efforts (e.g., sharing repair materials).
6. **Safety Measures**:
   * **Collision Avoidance**: Implement collision detection and avoidance systems to prevent accidents.
   * **Emergency Protocols**: Define protocols for handling emergencies (e.g., UGV breakdowns, accidents).
7. **Data Analytics and Reporting**:
   * **Performance Metrics**: Collect data on repair efficiency, travel time, and resource usage. Analyze this data to improve the process.
   * **Predictive Maintenance**: Use data to predict maintenance needs and prevent unexpected failures.

Remember that this framework should be adaptable to different road conditions, weather, and repair requirements. Collaborate with experts in robotics, transportation, and civil engineering to fine-tune the system and ensure successful pothole repair using UGVs. 🛣️🤖🚧

Certainly! Let’s break down the process of creating a Webots simulation for moving a car from point A to point B using GPS coordinates. Since you’re familiar with Python, we’ll integrate that as well. Here are the steps:

1. **Install Webots**:
   * If you haven’t already, download and install Webots from the official website: Webots Download.
2. **Create a New World**:
   * Open Webots and create a new world (File → New World).
   * Add a ground plane (Floor) and walls to define your arena.
3. **Add a Car Model**:
   * Import a car model into your world. You can use the built-in car models or create a custom one.
   * Position the car at point A in your world.
4. **Set Up GPS Sensor**:
   * Add a GPS sensor to the car model.
   * Configure the GPS sensor to provide latitude and longitude data.
   * Set the update rate to 5 seconds.
5. **Python Controller**:
   * Create a Python controller for the car.
   * In your Python script, you’ll need to:
     + Initialize the car model.
     + Define the target GPS coordinates (point B).
     + Calculate the distance and direction to move.
     + Implement a control loop to move the car towards point B.
     + Record GPS coordinates at regular intervals (every 5 seconds).
6. **Coordinate System**:
   * Decide whether you want to work with Webots’ Cartesian coordinate system or use latitude-longitude-altitude coordinates.
   * [You can set this in the gpsCoordinateSystem field of the WorldInfo node](https://cyberbotics.com/doc/reference/gps)[1](https://cyberbotics.com/doc/reference/gps).
7. **Simulation Loop**:
   * In your Python script, create a loop that:
     + Computes the distance and direction to the target.
     + Updates the car’s position based on the calculated movement.
     + Records GPS coordinates at the specified intervals.
8. **Testing and Debugging**:
   * Run the simulation and observe the car’s movement.
   * Debug any issues related to GPS accuracy, control, or calculations.
9. **Save Recorded Data**:
   * Store the recorded GPS coordinates in a data structure (e.g., a list or file).
   * You can later analyze this data or visualize it using external tools.
10. **Gradual Movement**:
    * To simulate gradual movement, consider using acceleration and deceleration profiles.
    * Adjust the car’s speed and steering angle accordingly.

Remember that this is a high-level overview, and you’ll need to dive into the specifics of Webots and Python to implement each step. [Feel free to explore Webots tutorials and documentation for more detailed guidance](https://cyberbotics.com/doc/reference/gps)[2](https://cyberbotics.com/doc/automobile/car-library?version=R2021b)[3](https://cyberbotics.com/doc/guide/tutorial-1-your-first-simulation-in-webots)[4](https://cyberbotics.com/doc/guide/tutorial-1-your-first-simulation-in-webots?tab-language=python)[5](http://www.cyberbotics.com/doc/guide/tutorials). Good luck with your simulation, and feel free to ask if you have any further questions! 🚗🌐

Explore

Great! Let’s proceed with importing a built-in car model into your Webots world. Here are the steps:

1. **Choose a Car Model**:
   * Webots provides several built-in car models. You can choose one that suits your simulation.
   * Common options include sedan cars, SUVs, or even futuristic vehicles.
2. **Insert the Car Model**:
   * In your Webots world, locate the scene tree (usually on the left side).
   * Right-click on the root node or any other suitable node where you want to insert the car model.
   * Select “Insert” → “Proto” → “Car” (or the specific car model you’ve chosen).
3. **Configure the Car Model**:
   * Once inserted, you’ll see the car model in your world.
   * Adjust its position, orientation, and scale as needed.
   * You can also modify other properties such as color, texture, and wheels.
4. **Run the Simulation**:
   * Save your world.
   * Run the simulation to check if the car appears correctly.
   * You can use the play button in the Webots interface to start the simulation.
5. **Fine-Tuning**:
   * If necessary, tweak the car’s parameters (e.g., mass, friction, steering) to match your requirements.
   * Test the car’s movement and interaction with the ground.
6. **Record GPS Coordinates**:
   * Since you want to move the car from point A to point B, you’ll need to implement the Python controller (as mentioned in step 5 of the previous response).
   * In your Python script, calculate the GPS coordinates corresponding to point B.
   * Use the GPS sensor data to track the car’s position during the simulation.

Remember that Webots provides flexibility, so you can customize the car model further or even create your own if needed. If you encounter any issues or need more guidance, feel free to ask! 🚗🌐