Creating and Evaluating a Simulation of an  
Autonomous Vehicle and Discussing its  
Potential as a Delivery System

Introduction

Autonomous vehicles have been dreamt of since the 1960s when science fiction writer Arthur C. Clarke devised the idea in his book “Profiles of the future” where he describes cars as able to choose their own route and decide on the fastest route to a set destination (Clarke, 2013). Autonomous vehicles have the potential to be applied in industrial, commercial and social sectors due to the variety of vehicles and machinery produced across the globe. In this project the focus will be on commercial uses of autonomous vehicles, specifically in delivering goods.

The “last mile delivery” refers to the last phase of the supply chain which is widely considered to be the most inefficient and most costly phase of the system (Slabinac, 2015). The inefficiency of this phase is increasing due to certain factors worsening, such as the spatial distribution, demand for more frequent and smaller deliveries, aims for better delivery times and more (Slabinac, 2015). The increase in use of e-commerce is causing problems for modern day last mile delivery due to the sustainability of environmental and social factors (Borghetti et al., 2022). The growing volume of parcels needing to be delivered in urban areas is requiring more delivery vehicles to cope with this increase (Boysen et al., 2020). The increase in delivery vehicles in turn is causing an increase in congestion, pollution and having a negative effect on the health of the population (Boysen et al., 2020). To mitigate these worsening conditions, a better system needs to be developed (Slabinac, 2015). Proposals for increasing the efficiency of the last mile include better automated systems and drone delivery systems, as well as the proposed systems by Mangiaracina et al: parcel lockers, crowdsourcing logistics, mapping the consumer at home and dynamic pricing policies (Mangiaracina et al., 2019).

The robot designed in this project is an autonomous delivery vehicle which will be able to calculate the shortest path to a specified destination and drive along this path until it reaches its destination. The vehicle will then stop outside its destination for a short time before autonomously returning to the warehouse. Automating the delivery vehicles will increase the efficiency of the delivery process by using algorithms to choose most efficient routes reducing cost of fuel and lowering emissions as well as helping with congestion issues in heavily populated areas. It will also remove the need for a driver meaning time can be used more efficiently and will not require specific delivery times during the day but can also run overnight.

The Design of the Robot

The robot is displaying its simulation on a custom-built map measuring 200m by 200m in length and width with a 5m high wall surrounding the map. The map was designed with a relatively simple road system with a roundabout and numerous corners the robot would have to navigate. The map also has 29 different buildings the robot will be able to navigate to which includes a built-up city area and a more rural suburban area.

The body of the robot is a simple cuboid measuring 1.5m width by 3.2m length by 1.2m height to simulate the body of a vehicle. Each component is attached to the body of the robot to give the basic model of a vehicle such as a car or van.

The robot has been given four hinge-joints set with their device to act as rotational motors which are connected to cylinders with height of 0.3m and radius of 0.5m. The cylinders have been orientated and attached to the lateral sides of the body to simulate wheels. Attaching the cylinders to rotational motors about the hinge-joints allowing the wheels to rotate and move the robot forwards, backwards and to turn left and right by giving the motors different rotational speeds and direction. To turn the vehicle to the left, the left wheels rotate in a backwards direction where the right wheels continue to rotate forwards, and vice versa to turn the vehicle to the right.

The robot has been given two distance sensors called the left sensor and right sensor. The left sensor is attached to the lower left side of the front of the body, and the right sensor is attached to the lower right side of the front of the body. The sensors are spherical in shape and are aimed outwards and forwards with respect to the front of the vehicle. The purpose of the sensors is to detect any changes in the distance between the floor and the vehicle. In doing this, the sensor can detect objects and elevated platforms such as kerbs. If the left sensor detects a change in distance, which in this simulation will be due to the elevated kerb, the vehicle will turn to the right, and vice versa for the right sensor. If no change is detected by either sensor, the car will either remain stationary or continue to travel along its set path.

The robot has been given a GPS which has been shaped as a small cylinder and attached to the roof of the body. The GPS can track the live location of the robot as it moves around the map, allowing the robot to geolocate and calculate its position relative to its destination along the designated path.

The robot has been given an IMU which has been shaped as a small cube and attached to the front centre of the roof of the body. The IMU is used to register the rotation of the robot around each axis, which allows for the robot to calculate and orientate itself towards the next point along the designated path.

The Control System of the Robot

The vehicle in this simulation has been designed fully autonomous, only requiring an input as a destination for the robot to travel to and from. Once the robot has been specified a destination by the user, it will carry out the following actions:  
 1. Calculate the shortest route along the roads from the warehouse to get to the destination.  
 2. Travel along the roads following the calculated shortest path to reach its destination.  
 3. Pause for a set amount of time to demonstrate a time allowance for delivery.  
 4. Calculate a return path from the destination back to the warehouse.  
 5. Return to the warehouse by travelling along the calculated reversed path.  
 6. Stop at the warehouse to simulate a completed delivery.

To calculate the shortest path to the destination, firstly each corner and junction of the road as well as each destination had to be mapped for their x and y coordinates. In doing this, Dijkstra’s algorithm was implemented so the robot can calculate every possible route to the destination and their respective distances, thus finding the shortest route. Once the route has been calculated, the robot will begin its journey. By coordinating the motors, sensors, GPS and IMU the robot will navigate along the shortest route whilst avoiding obstacles to safely arrive to the destination. Upon the robot’s arrival to its destination, the robot will stop at that location for a set period. This will simulate a given time the vehicle would need to stop to complete the delivery. Once the set time has passed, the robot will re-calculate its path back to the warehouse. It does this by reversing the route it took to get to the destination and returning via the same path, therefore also using the shortest route back, making the total distance travelled the shortest route possible according to Dijkstra’s algorithm. The simulation then terminates once the vehicle arrives back at the warehouse.

Limitations of the Robot

The first problem found when developing the robot was matching the weight of the vehicle to the density, gravitational force and the force applied by the motors to move the wheels. Due to the scaling of the vehicle, this greatly increased the mass of the vehicle which made the vehicle sink into the map and motors would not move the wheels. To correct this, the vehicle’s mass had to be changed to approximately 5kg and the softCFM was changed to x10-5. Although this rectified the issues above, this does not give a true representation of the correct size to mass ratio.

The physics of the world was difficult to get accurate when scaling the map up to real-life size which caused changes in the inertia of the vehicle when moving which made slowing, stopping and turning very difficult. This resulted in not being able to travel at very fast speeds to prevent the robot from sliding everywhere and going off track or breaking.

The robot has only been designed as a purpose to travel to and from a given destination via the shortest route and sticking within a simple road layout without lanes. The way the robot has been made has given it basic methods of avoiding obstacles and pathing capabilities. In real-life, the vehicle would have to stick to a driving lane whilst also avoiding other vehicles and obstacles along the path, as well as other real-life challenges.

Due to the linking of coordinates to create the path, the robot does not follow the smoothest path possible to its destination and back. It was also found if the coordinates were off or too close to the kerbs, this would throw off the pathing of the robot and cause the robot to either make an unnecessary turn or to break the path.

Even though Dijkstra’s algorithm is a well-renowned method of finding the shortest route, it does not scale well when scaling to real-life as this would involve many more junctions, meaning it would have to create a very large number of routes and compare all distances before finding the shortest route, making it inefficient. It is a very efficient method for this simulation due to the limited number of nodes used as coordinates (Fuhao & Jiping, 2009). A better algorithm to use would be the A\* algorithm as this takes a more heuristic approach to finding the shortest route and is much more efficient (Erke et al., 2020).

Robot Evaluation

One strength of the autonomous vehicle in this simulation is its obstacle avoidance capabilities. The robot uses the distance sensors to assess changes in the distance facing in a downward direction to detect when the vehicle is approaching an obstacle (in this case the kerbs of the road) which causes the robot to coordinate with the motors of the vehicle making it turn away from the obstacle to ensure it follows the road without collision. During testing, the vehicle showed excellent implementation of obstacle avoidance when only hitting the kerb once when reversing to turn around. This could be improved on by implementing reverse distance sensors on the rear of the vehicle.

The next strength of this robot is the successful implementation of Dijkstra’s algorithm to find the shortest route. This was done by mapping each corner and destination to a node and noting which nodes were connected to each other. This allows the robot to calculate each possible route and the total distance of each route using Pythagoras’s theorem, finding the shortest route and following this path. During testing, the robot was successfully able to navigate to each of the building locations via the shortest route and return to the warehouse after stopping. This demonstrated the robot being very reliable in navigating to any specified building on the map following the calculated shortest path.

Another strength displayed by the robot is the coordination displayed between the GPS coordinates and the IMU. This is a successful method of turning the vehicle towards the correct node to ensure the vehicle is travelling along the calculated path correctly.

The map design has proven to be a strength of the simulation due to it being a relatively complex road system with a variety of destinations with different corners and junctions, making testing much more robust.

One weakness of this simulation is the physics of the map. The coordination between scaling to the correct sizes, the mass of the robot, strength of the maps gravity and density of the floor were incredibly difficult to get in sync and therefore resulted in the mass of the vehicle having to be greatly reduced to allow the motors to successfully move the vehicle. Due to these changes, the velocity of the vehicle had to be greatly reduced to reduce the inertia of the vehicle, allowing for smoother movement between nodes.

The next weakness is the object detection of the vehicle. Although the distance sensors work well, they are only specialised to detect the changes in distance for when nearing the kerbs of the road. When applying this to real-life scenarios, the robot would require a variety of different sensors to ensure its ability to travel safely along actual roads. Such sensors would include object detection to enable the vehicle to slow or stop as well as turn away from the object, having the robot remain within its driving lane and not just on the road itself, and cameras to detect objects such as traffic lights and road signs. To improve upon this the robot would need a variety of sensors surrounding each of the robots’ sides, including top and bottom, to ensure as much data is collected as possible which the robot can react to accordingly.

Another weakness of this simulation is the accuracy of the nodes using the GPS system. Although the nodes have proven to work and achieve the aims of the robot, the robot will swerve along the road making it unsuitable for following lanes on a road. It was also noted if the nodes were not positioned in ideal locations (such as too close to the edge of the corner) that it can cause the robot to crash or make a wrong turning and having to correct itself. This can be improved on by making more accurate GPS nodes on the corners and using more GPS nodes along the roads to ensure better accuracy.

Existing Autonomous Delivery Robots

There are currently three main types of autonomous delivery robots (ADR’s) which are the sidewalk autonomous delivery robot (SADR), road autonomous delivery robot (RADR) and the truck and robot system (TRS) also known as the mothership van system (Hossain, 2022). One of the more popular autonomous delivery robots is the delivery drone which has become well-known since the proposal by Amazon.com of their own delivery drone (Sudbury & Hutchinson, 2016).

The SADR is a pedestrian-sized delivery robot which delivers parcels to the public and return to base, removing the need for a delivery person (Jennings & Figliozzi, 2019). It is currently most popular in delivery systems on college campuses and small neighbourhoods (Hassain, 2022). Implementing the use of SADR’s, especially in heavily populated areas such as cities, the SADR can substantially reduce the cost and time of delivery making the last mile delivery much more efficient (Jennings & Figliozzi, 2019). SADR’s are predominantly found in the U.S. and due to the need to travel along pavements, the SADR’s are seeing an increase in regulations by local agencies in the U.S (Jennings & Figliozzi, 2019).

The RADR is an autonomous delivery robot which travels via roads to set destinations alongside other vehicles, most commonly used to deliver parcels within relatively distant locations in a city (Hossain, 2022). This removes the need for a delivery person driving the vehicle and delivering the parcels again making the last mile delivery more cost and time effective (Hossain, 2022). However, due to the need to travel alongside other vehicles, this makes the system much less popular than the SADR regardless of its limitations, and as quoted by Hossain, 2022: “Customer acceptance of ADRs: Acceptance of ADRs by customers is key to their success”, meaning the system will not be effective if it is not an accepted method by the public.

The TRS is a system which relies on trucks to carry the autonomous robots for a portion of the journey and the autonomous robots then being used to cover the last short distance to deliver the parcels to their desired destination (Heimfarth et al., 2022). Using the TRS system has shown to be able to reduce the delivery costs by up to two thirds compared to the conventional method using trucks and delivery drivers used today (Ostermeier et al., 2021). Although this method has the potential to increase efficiency and reduce cost, the statistics used by Ostermeier et al. did not include the need for the truck drivers which may increase the cost and reduce its efficiency from the statistics proposed (Hossain, 2022). Each of these methods has shown potential in drastically reducing the cost and increasing the efficiency of the last mile delivery problem, whilst also helping to reduce congestion in cities and reduce pollution (Hossain, 2022). Although these systems are showing great promise in helping with commercial and environmental issues, the systems are still in their infancy and require further development before being integrated into the public sphere and will require more work in gaining the trust of the public (Hossain, 2022).

One proposed solution for the last mile delivery problem is the use of drones to deliver smaller parcels within populated areas such as cities (Yoo & Chankov, 2018). This is one of the leading delivery methods where drones loaded on a truck are used to deliver parcels within set locations (Imran et al., 2021). Drone delivery has the potential to assist with the increase in delivery demand, decrease the delivery time and will aid in the reduction of congestion and pollution in cities (Yoo & Chankov, 2018). Although showing promise to assist with the last mile delivery, there are several issues drones must overcome before safely being implemented in their use for delivery. Some of these problems include not having a specialised area to land, noise pollution produced by the drones, the safety and security of the population as well as public acceptance are the key issues discovered so far (Seidakhmetov & Valilai, 2022).

References

Borghetti, F., Caballini, C., Carboni, A., Grossato, G., Maja, R., & Barabino, B. (2022). The Use of Drones for Last-Mile Delivery: A Numerical Case Study in Milan, Italy. *Sustainability*, *14*(3), 1766.

Boysen, N., Fedtke, S., & Schwerdfeger, S. (2020). Last-mile Delivery concepts: a Survey from an Operational Research Perspective. *OR Spectrum*, *43*(1).

Boysen, N., Schwerdfeger, S., & Weidinger, F. (2018). Scheduling last-mile deliveries with truck-based autonomous robots. *European Journal of Operational Research*, *271*(3), 1085–1099.

Clarke, A. C. (2013). Profiles of the future. Hatchette UK.

Erke, S., Bin, D., Yiming, N., Qi, Z., Liang, X., & Dawei, Z. (2020). An improved A-Star based path planning algorithm for autonomous land vehicles. *International Journal of Advanced Robotic Systems*, *17*(5), 172988142096226.

Fuhao, Z. & Jiping, L. (2009). An algorithm of Shortest Path Based on Dijkstra for Huge Data. *2009 Sixth International Conference on Fuzzy Systems and Knowledge Discovery,* 244-247.

Heimfarth, A., Ostermeier, M., & Hubner, A. (2022). A mixed truck and robot delivery approach for the daily supply of customers. *European Journal of Operational Research,* 303(1), 401-421.

Hossain, M. (2022). Autonomous Delivery Robots: A Literature Review. *SSRN Electronic Journal*.

Imran, N. M., Mishra, S., & Won, M. (2021). Towards Fully Autonomous Drone-Based Last-Mile Delivery. *ArXiv:2103.04118 [Cs]*.

Jennings, D., & Figliozzi, M. (2019). Study of Sidewalk Autonomous Delivery Robots and Their Potential Impacts on Freight Efficiency and Travel. *Transportation Research Record: Journal of the Transportation Research Board*, *2673*(6), 317–326.

Mangiaracina, R., Perego, A., Seghezzi, A., & Tumino, A. (2019). Innovative solutions to increase last-mile delivery efficiency in B2C e-commerce: a literature review. *International Journal of Physical Distribution & Logistics Management*, 49(9), 901-920.

Ostermeier, M., Heimfarth, A., & Hübner, A. (2021). Cost‐optimal truck‐and‐robot routing for last‐mile delivery. *Networks*.

Seidakhmetov, A., & Valilai, O. F. (2022). *Drone based delivery system: Restrictions and limitations*. Www.econstor.eu; Berlin: epubli GmbH.

Slabinac, M. (2015). INNOVATIVE SOLUTIONS FOR A “LAST-MILE” DELIVERY – A EUROPEAN EXPERIENCE. *Business Logistics in Modern Management*.

Sudbury, A. W., & Hutchinson, E. B. (2016). A Cost Analysis Of Amazon Prime Air (Drone Delivery). *Journal for Economic Educators*, *16*(1), 1–12.

Yoo, H. D., & Chankov, S. M. (2018). Drone-delivery Using Autonomous Mobility: An Innovative Approach to Future Last-mile Delivery Problems. *2018 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*.

Controller Code

import com.cyberbotics.webots.controller.\*;

import java.util.\*;

import java.lang.Math;

public class CarController {

// Variables

Robot robot;

Motor[] motors = new Motor[4];

GPS gps;

InertialUnit imu;

DistanceSensor[] ds = new DistanceSensor[2];

String[] motor\_names = {"FR\_motor", "FL\_motor", "BR\_motor", "BL\_motor"};

Node home = new Node(44, "warehouse", 80, 10);

Node destination = new Node();

Node[] map = new Node[45];

int time\_step = -1;

double movement\_velocity = 2;

double turn\_velocity = 30;

int current\_time = 0;

double angle\_threshold = 0.05;

double distance\_threshold = 0.5;

// Constructor

public CarController() {

robot = new Robot();

ds[0] = robot.getDistanceSensor("L\_sensor");

ds[1] = robot.getDistanceSensor("R\_sensor");

gps = robot.getGPS("GPS");

imu = robot.getInertialUnit("IMU");

for(int i = 0; i < motor\_names.length; i++) {

motors[i] = robot.getMotor(motor\_names[i]);

}

for(int i = 0; i < motors.length; i++) {

motors[i].setPosition(Double.POSITIVE\_INFINITY);

motors[i].setVelocity(0);

}

time\_step = (int) Math.round(robot.getBasicTimeStep());

ds[0].enable(time\_step);

ds[1].enable(time\_step);

gps.enable(time\_step);

imu.enable(time\_step);

// Corner GPS coordinates

map[0] = new Node(0, "", 60, 10);

map[1] = new Node(1, "", 42, 10);

map[2] = new Node(2, "", 51, 4);

map[3] = new Node(3, "", 51, 19);

map[4] = new Node(4, "", 47, 71);

map[5] = new Node(5, "", -66, 71);

map[6] = new Node(6, "", -69, 30);

map[7] = new Node(7, "", 15, 27);

map[8] = new Node(8, "", 21, 13);

map[9] = new Node(9, "", -69, -20);

map[10] = new Node(10, "", 49, -18);

map[11] = new Node(11, "", 54, -40);

map[12] = new Node(12, "", 78, -45);

map[13] = new Node(13, "", 78, -70);

map[14] = new Node(14, "", -66, -70);

// Building GPS coordinates

map[15] = new Node(15, "motel", 51, -15); // motel reception

map[16] = new Node(16, "church", 78, -45); // church

map[17] = new Node(17, "suburban", 78, -70); // suburban house

map[18] = new Node(18, "manor", 63, -73); // small manor

map[19] = new Node(19, "modern", 54, -73); // modern house

map[20] = new Node(20, "small\_residential", 38, -73); // small residential building

map[21] = new Node(21, "simple", 28, -73); // simple two-storey house

map[22] = new Node(22, "medium\_residential", 12, -73); // medium residential building

map[23] = new Node(23, "modern\_suburban", -5, -73); // modern suburban house

map[24] = new Node(24, "big\_residential", -23, -73); // big residential building

map[25] = new Node(25, "garage", -38, -73); // house with garage

map[26] = new Node(26, "bungalow", -66, -70.5); // bungalow style house

map[27] = new Node(27, "composed", -69, -58); // composed house

map[28] = new Node(28, "auditorium", -69, 5); // auditorium

map[29] = new Node(29, "barn", -69, 22); // barn

map[30] = new Node(30, "hollow", -69, 50); // hollow building

map[31] = new Node(31, "old\_residential", -7, 30); // old residential building

map[32] = new Node(32, "hotel", 35, 10); // hotel

map[33] = new Node(33, "restaurant", 35, -18); // fast food restaurant

map[34] = new Node(34, "glass", 10, -19); // big glass tower

map[35] = new Node(35, "gas\_station", 51, 40); // gas station

map[36] = new Node(36, "carwash", 51, 64); // carwash

map[37] = new Node(37, "cyberbotics", 48, 70); // cyberbotics tower

map[38] = new Node(38, "small\_tower", 41, 73); // small residential tower

map[39] = new Node(39, "commercial", 30, 73); // commercial building

map[40] = new Node(40, "large\_tower", 15, 73); // large residential tower

map[41] = new Node(41, "building", -2, 73); // building

map[42] = new Node(42, "construction", -15, 73); // building under construction

map[43] = new Node(43, "museum", -50, 73); // museum

map[44] = home;

// Set adjacent points

map[0].setAdjacent(new int[]{2, 3, 44});

map[1].setAdjacent(new int[]{2, 3, 8, 32});

map[2].setAdjacent(new int[]{0, 1, 10, 15});

map[3].setAdjacent(new int[]{0, 1, 4, 35, 36});

map[4].setAdjacent(new int[]{3, 5, 35, 36, 37, 38, 39, 40, 41, 42, 43});

map[5].setAdjacent(new int[]{4, 6, 30, 38, 39, 40, 41, 42, 43});

map[6].setAdjacent(new int[]{5, 7, 9, 28, 29, 30, 31});

map[7].setAdjacent(new int[]{6, 8, 31});

map[8].setAdjacent(new int[]{7, 1, 32});

map[9].setAdjacent(new int[]{6, 10, 14, 27, 28, 29, 33, 34});

map[10].setAdjacent(new int[]{2, 9, 11, 15, 33, 34});

map[11].setAdjacent(new int[]{10, 12});

map[12].setAdjacent(new int[]{11, 13, 16, 18});

map[13].setAdjacent(new int[]{12, 14, 17, 18, 19, 20, 21, 22, 23, 24, 25});

map[14].setAdjacent(new int[]{9, 13, 19, 20, 21, 22, 23, 24, 25, 26, 27});

map[15].setAdjacent(new int[]{2, 10});

map[16].setAdjacent(new int[]{12});

map[17].setAdjacent(new int[]{13});

map[18].setAdjacent(new int[]{12, 13});

map[19].setAdjacent(new int[]{13, 14});

map[20].setAdjacent(new int[]{13, 14});

map[21].setAdjacent(new int[]{13, 14});

map[22].setAdjacent(new int[]{13, 14});

map[23].setAdjacent(new int[]{13, 14});

map[24].setAdjacent(new int[]{13, 14});

map[25].setAdjacent(new int[]{13, 14});

map[26].setAdjacent(new int[]{14});

map[27].setAdjacent(new int[]{9, 14});

map[28].setAdjacent(new int[]{6, 9});

map[29].setAdjacent(new int[]{6, 9});

map[30].setAdjacent(new int[]{5, 6});

map[31].setAdjacent(new int[]{6, 7});

map[32].setAdjacent(new int[]{1, 8});

map[33].setAdjacent(new int[]{9, 10});

map[34].setAdjacent(new int[]{9, 10});

map[35].setAdjacent(new int[]{3, 4});

map[36].setAdjacent(new int[]{3, 4});

map[37].setAdjacent(new int[]{4});

map[38].setAdjacent(new int[]{4, 5});

map[39].setAdjacent(new int[]{4, 5});

map[40].setAdjacent(new int[]{4, 5});

map[41].setAdjacent(new int[]{4, 5});

map[42].setAdjacent(new int[]{4, 5});

map[43].setAdjacent(new int[]{4, 5});

map[44].setAdjacent(new int[]{0});

}

// Main method

public static void main(String[] args) {

CarController controller = new CarController();

controller.ControlLoop(args[0]);

}

public void ControlLoop(String building) {

boolean valid = false;

while(!valid){

for(Node node : map){

if(building.toLowerCase().equals(node.getName().toLowerCase())){

destination = node;

valid = true;

break;

}

}

}

ArrayList<Node> path = findShortestPath(home, destination);

for(Node node : path){

}

int currentIndex = 0;

while (robot.step(time\_step) != -1) {

if(currentIndex+1 == path.size() && path.get(currentIndex).getName().toLowerCase().equals("warehouse")){

stop(motors);

double current\_time\_1 = robot.getTime();

double current\_time\_2= robot.getTime();

do {

stop(motors);

current\_time\_2 = robot.getTime();

robot.step(1);

} while(current\_time\_2 < (current\_time\_1 + 3));

return;

}

else if(currentIndex+1 == path.size()) {

double current\_time\_1 = robot.getTime();

double current\_time\_2= robot.getTime();

do {

stop(motors);

current\_time\_2 = robot.getTime();

robot.step(1);

} while(current\_time\_2 < (current\_time\_1 + 50));

Collections.reverse(path);

currentIndex = 0;

}

double remainingDistance = calculateDistance(new Node(-1, "", gps.getValues()[0], gps.getValues()[1]), path.get(currentIndex+1));

if(remainingDistance < distance\_threshold && currentIndex+1 < path.size()) {

currentIndex++;

}

if(currentIndex+1 == path.size()) {

continue;

}

if(ds[0].getValue() < 550) {

turnRight(motors);

}

else if(ds[1].getValue() < 550) {

turnLeft(motors);

}

else {

double currentOrientation = -1\* (Math.PI / 2) + imu.getRollPitchYaw()[2];

double desiredOrientation = calculateDesiredOrientation(path.get(currentIndex+1));

double angleDifference = desiredOrientation - currentOrientation;

// Find angle to next point

if (angleDifference > Math.PI) {

angleDifference -= 2 \* Math.PI;

} else if (angleDifference < -Math.PI) {

angleDifference += 2 \* Math.PI;

}

// Adjust angle of car to next point

if (Math.abs(angleDifference) > angle\_threshold) {

if (angleDifference < 0) {

turnRight(motors);

}

else {

turnLeft(motors);

}

}

else {

goForward(motors);

}

}

}

}

// Movement methods

private void goForward(Motor[] motors) {

for(int i = 0; i < motors.length; i++) {

motors[i].setVelocity(movement\_velocity);

}

}

private void stop(Motor[] motors) {

for(int i = 0; i < motors.length; i++) {

motors[i].setVelocity(0);

}

}

private void turnLeft(Motor[] motors) {

for(int i = 0; i < motors.length; i++) {

if(i % 2 == 0) {

motors[i].setVelocity(turn\_velocity);

}

else {

motors[i].setVelocity(-1 \* turn\_velocity);

}

}

}

private void turnRight(Motor[] motors) {

for(int i = 0; i < motors.length; i++) {

if(i % 2 == 1){

motors[i].setVelocity(turn\_velocity);

}

else {

motors[i].setVelocity(-1 \* turn\_velocity);

}

}

}

// Dijkstra's algorithm implementation

public ArrayList<Node> findShortestPath(Node start, Node destination) {

HashMap<Node, Double> distances = new HashMap<>();

for (Node node : map) {

distances.put(node, Double.POSITIVE\_INFINITY);

}

HashMap<Node, Node> previous = new HashMap<>();

PriorityQueue<Node> queue = new PriorityQueue<>(Comparator.comparingDouble(distances::get));

distances.put(start, 0.0);

queue.add(start);

while (!queue.isEmpty()) {

Node current = queue.poll();

// Found the destination, stop searching

if (current == destination) {

break;

}

// Calculate new distance through current node

for (Node neighbor : current.adjacent) {

double distance = distances.get(current) + calculateDistance(current, neighbor);

if (distance < distances.get(neighbor)) {

distances.put(neighbor, distance);

previous.put(neighbor, current);

queue.add(neighbor);

}

}

}

// Reverse path

ArrayList<Node> path = new ArrayList<>();

Node current = destination;

while (previous.containsKey(current)) {

path.add(current);

current = previous.get(current);

}

path.add(start);

Collections.reverse(path);

return path;

}

// Pythag implementation

private double calculateDistance(Node node1, Node node2) {

double dx = node2.position.getX() - node1.position.getX();

double dy = node2.position.getY() - node1.position.getY();

return Math.sqrt(dx \* dx + dy \* dy);

}

// Calculate desired orientation from current position to next node

private double calculateDesiredOrientation(Node nextNode) {

double dx = nextNode.getPosition().getX() - gps.getValues()[0];

double dy = nextNode.getPosition().getY() - gps.getValues()[1];

return Math.atan2(dy, dx);

}

// Position object

private class Position {

double x;

double y;

public Position() {}

public Position(double x, double y) {

this.x = x;

this.y = y;

}

public double getX() {

return this.x;

}

public double getY() {

return this.y;

}

public void setX(double x) {

this.x = x;

}

public void setY(double y) {

this.y = y;

}

}

// Nodes within path

private class Node {

int id;

Position position;

String name;

ArrayList<Node> adjacent = new ArrayList<Node>();

public Node() {}

public Node(int id, String name, double x, double y) {

this.id = id;

this.name = name;

this.position = new Position(x, y);

}

public int getID() {

return this.id;

}

public String getName() {

return this.name;

}

public Position getPosition() {

return position;

}

public void setAdjacent(int[] IDs) {

for(int id : IDs){

for(Node node : map) {

if(id == node.getID()) {

this.adjacent.add(node);

}

}

}

}

}

}