Table of Contents

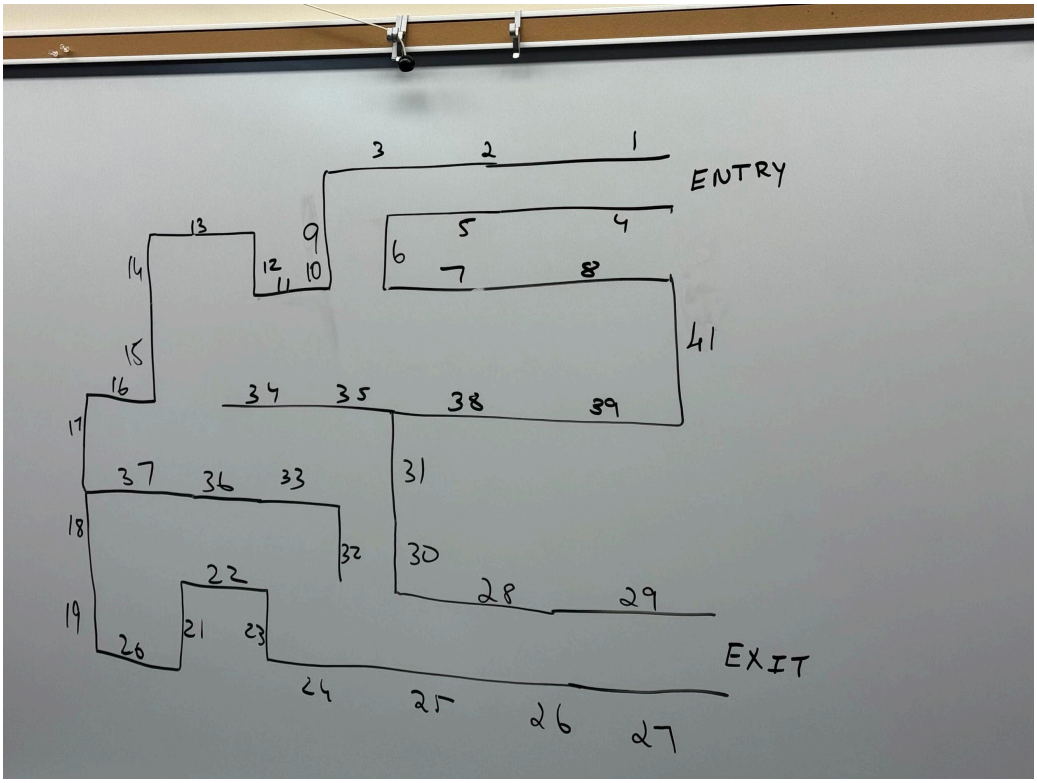
Physical

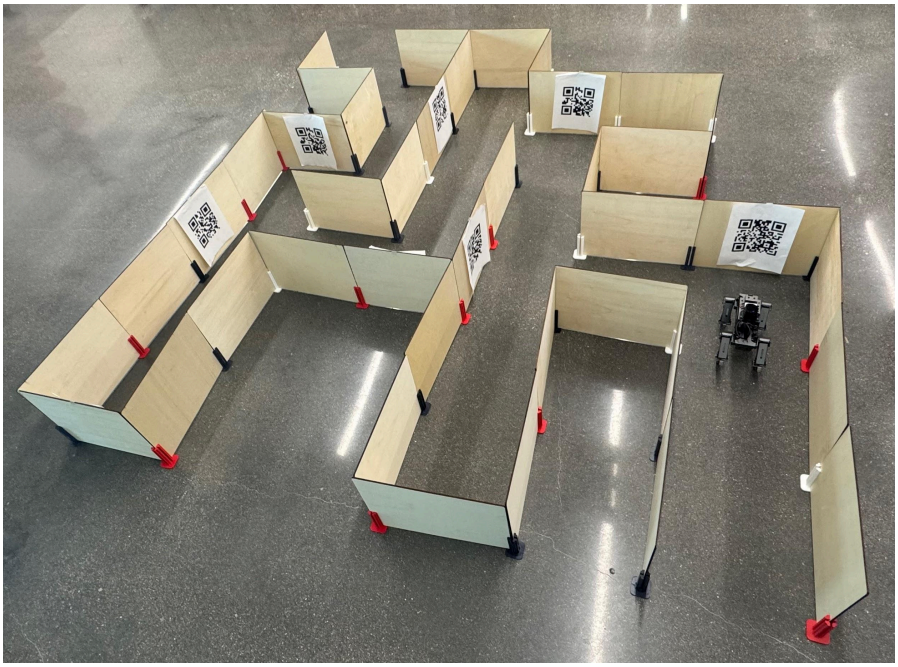
Things the robot can do

- Basic movements like move forward, backward, turn left/right, and crab walk left/right
- QR Code recognition, Face recognition, Colored object recognition
- Solve a physically built maze based on what the virtual robot sends to the physical robot
- Obstacle avoidance using Lidar sensors
- Follow-me mode: it can track and follow a person or object.
- Line-following capabilities: Useful in maze solving or navigation tasks.
- Can be controlled by a YahboomRobot app or the controller

Things to know / Problems

- Be mindful of the heat in the chassis as it might heat up the CPU faster as time goes on.
- Be mindful of the fact that the camera slows down the closer to death the dog is.
    - The battery health is about 1-2.5 hours depending on how intensive the tasks are
- The LIDAR data is incomprehensible on the physical side of things, so if you wish to use it be warned that you may not be able to read it in a usable way.
- When it comes to using the camera, VNC Viewer (VM) is required. Otherwise, the camera won't work. However, it is very laggy and really slow.
    - Need to constantly edit the file in the local computer, copy it to the robot, then test it from there, which can be time consuming and tedious.
    - You must be connected to the robot's WiFi in order to use VNC, but it might kick you off, so be ready to constantly reconnect to it to test the robot.
- Measurements for a maze if you would like to expand or continue a maze solver:
    - When figuring out measurements, use the dog length (21 cm)
    - MDF: 30 cm (h) by 40 cm (l) = 60 pieces
    - QR codes (Big and small):
        - Small size of QR Codes = 3 inches by 3 inches = 7.62 cm
        - Big size of QR Codes = 5 inches by 5 inches = 13.97 cm
    - Stands:
        - White: 12

        - Red: 12
        - Blue: 26
        - Total: 50
    - Example of maze structure:
    - 

    - 
- Both robots have the same IP address, so you can only use one at a time.

Recommendations

- Try to avoid breaking them or burning them
    - It may be tempting to add a fur to the dog, but don't recommend to do that
    - Do a health check about twice a month or maybe weekly (if necessary) to make sure that hardware issues won't occur as this may affect how the robot behaves even though the code is working
- Try to equally utilize both robots so wear is spread out amongst them.
    - When doing a turn left or turn right, make sure to even out both directions (i.e. don't just turn in one direction - turn left for the whole semester, and never or rarely used turn right).
- Don't leave the robots charging over breaks, and if you go into the lab and notice that they are fully charged, unplug them.
- Avoid overworking the robots and give them some rest to ensure their health is good
- Try to use a different VM that is compatible with viewing the camera for better responsiveness
- Use state machines to keep track of the movements when coding
- Use Github version control system to organize the code versions as it will be hard to test the code without connecting to the robot
    - Be mindful of how many versions of a program you keep on the dog, and remove unnecessary past iterations if they are no longer relevant.
- In terms of the physical maze, it is recommended to use MDF instead of cardboards so that it is strong and can hold it even if the robot kicks them
- If using a QR Code detection, use a printed paper because it is hard to detect it if using a mobile device QR code or a laminated QR Codes due to glossy reflections
- Clean camera lenses regularly to maintain image clarity, especially if the dog walks on dusty surfaces.
- Update software dependencies periodically, especially for Python libraries or OS packages.
- Document each test session (what you tried, what worked/failed) to save time for future debugging.
- Label charging cables and robot units to avoid confusion between Robot A and B.

- Mark maze with grid points for easier debugging and coordinate tracking.
- Have a recovery/reset procedure for when the robot behaves unexpectedly.
- Use overhead camera recordings during maze runs for post-run analysis.
- Track battery usage over time in a shared log to detect signs of battery degradation.
- Periodically inspect for any loose screws or motor wiring after repeated use.

Virtual

Problems for the next group to solve

This project was hard, and there was a decent amount of engineering done for this simulation. Here's the current problems with the robot:

- Movement is not perfectly straight. The math isn't perfect, and errors accumulate through things like friction and general simulation instability.
- Representing a maze through zero's and one's is hard to keep detail. How do you represent a tile with walls above it, with no walls to the sides? And then run that said representation through a traditional maze solving algorithm. The maze is generated in runtime, which is a big advantage for iteration however.
- Mazes with paths right next to each other cannot be represented in this style and therefore cannot be solved.
- The maze solver has a lot of magic PID constants.
- The maze solver cannot be sped up with Webot's time speed up function, the time checking logic does not properly increase with it.

Things the robot does

- Solves the maze given.
- Uses PIDs to balance movement and stay centered/avoid wall collisions.
- Receives most of the commands needed for movement, e.g. forward, left, right, back, as well as strafing and blending between movements.
- Uses LIDAR to detect wall depth.

- Procedurally generates movement steps and trajectories.

Things you should learn/know

- How LIDAR works in Webots.
- See Spot Implementation detail for details: https://github.com/LocalRobots/SpotImplementation
- How PID controllers work.
- What Inverse Kinematics (IK) is and how a basic trigonometry implementation of that works.
- General Webots and Python knowledge.
- AVOID ROS2 IF YOU VALUE YOUR TIME. It'll only be really usable on linux.