Autonomous Remote Intelligent Robot

Participants

Zachary - @Cutwell

Ivan - @luckychan12

Link to Video:

https://www.youtube.com/watch?v=iDN75vYgonQ

Project Description

We identified an emerging niche in autonomous robotics. Security and routine inspections of inaccessible facilities (due to distance or time of day) are often performed at the expense of human agents. We followed the state of the art developments from companies such as Boston Dynamics to develop our robot capable of autonomous behaviour and suitable for performing simple surveillance tasks.

What did we start with?

We started with a Thunderborg robot, with access to 4 driving motors, an HCSR04 ultrasonic sensor and an MPU6050 gyroscope/accelerometer sensor. Initial code for accessing the motors and sensor data was acquired from the Thunderborg forum and similar websites. While other artefacts in the field make use of expensive robotic components, the Thunderborg robot is relatively inexpensive and saves cost with the sensor components used. Our project focuses on maximising performance and overcoming these hardware-based constraints.

We also implemented a known algorithm (D* Lite) authored by Sven Koenig (https://aaai.org/Papers/AAAI/2002/AAAI02-072.pdf)
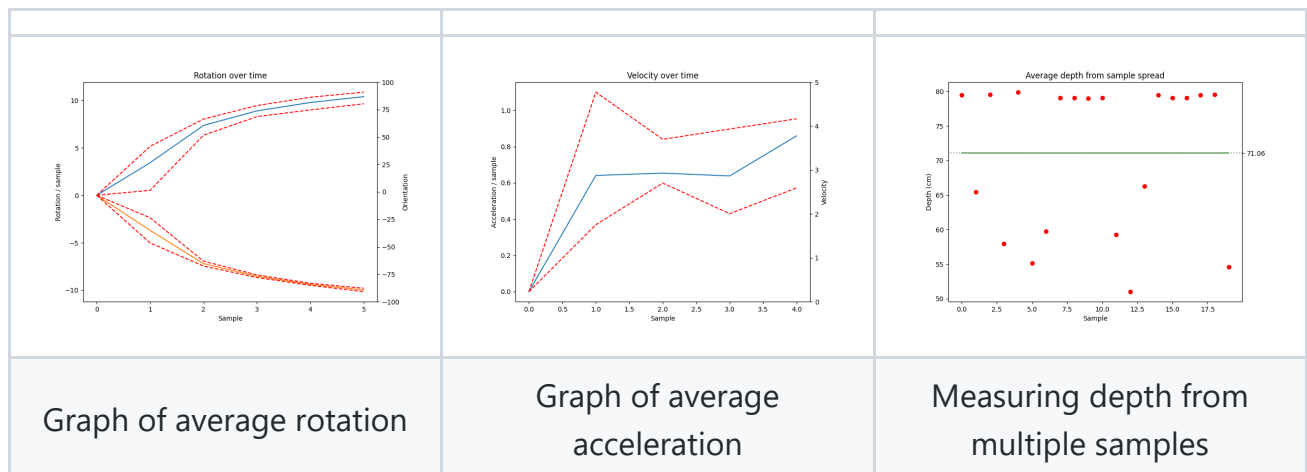
How the main program loop works:

- First, the program initialises the D* Lite Algorithm by parsing the initial environment.
- An input matrix is passed into the program, which details where any known obstacles are and how large the accessible environment is.
- This matrix is then converted into a graph of interconnected nodes.
- The program then initialises the weights of each node in the graph to represent any obstacles in the initial input matrix.
- After this, the program enters the main execution loop.
- First, it calculates the next node to be visited.

- The robot then checks to see if the next node is accessible using the ultrasonic sensors.
- If the node is not accessible, the robot marks and updates this within its internal graph before recalculating the following shortest path.
- If the next node to be visited is available, the robot drives to the following location and updates the algorithm of its current position in the graph.

Engineering

Software tree diagram	Fritzing hardware diagram
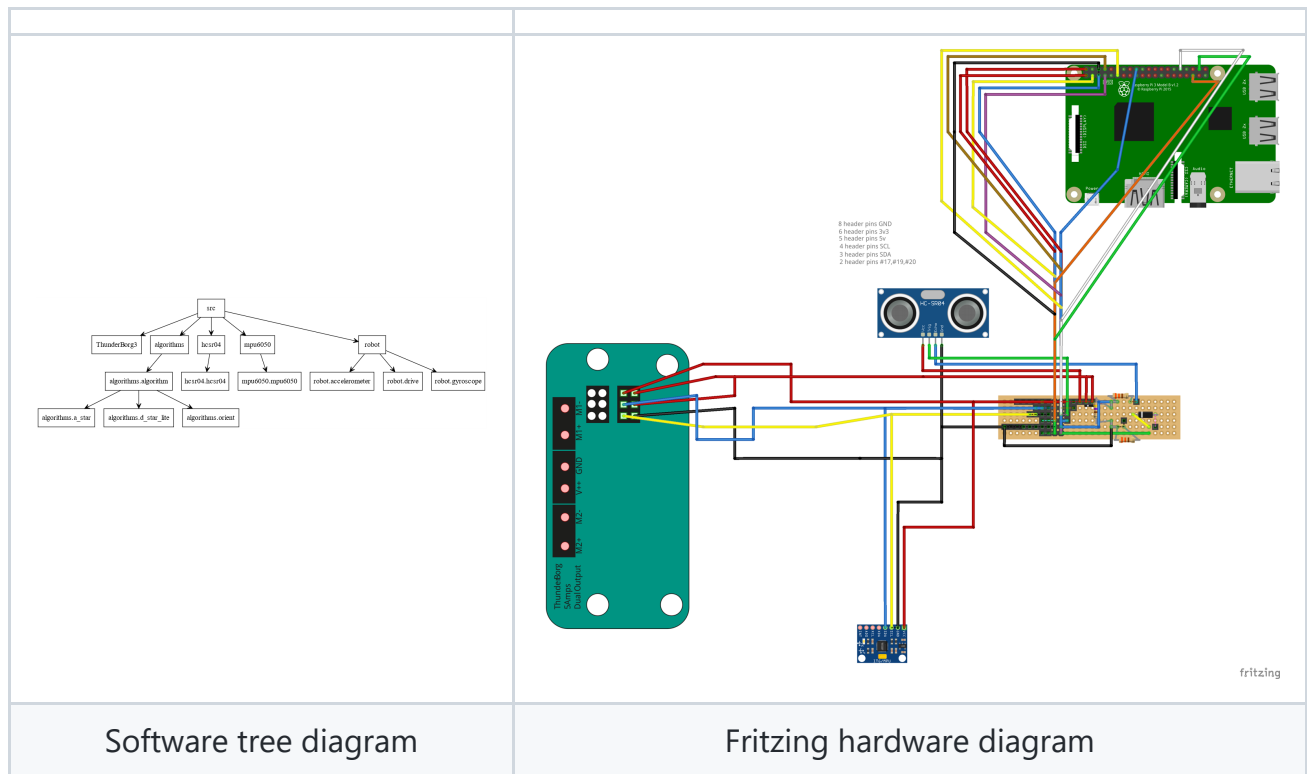
- We used a class-based structure to organise our codebase and expose individual functions.
- We used a raspberry pi 4 for our central controller, and collected internal and external data using a MPU6050 and HCSR04 sensor.

Graph of average rotation	Graph of average acceleration	Measuring depth from multiple samples

A significant engineering issue encountered during development was that the robot's motors were imbalanced. This resulted in the robot having a different rotational profile when rotating clockwise versus counterclockwise. Using the gyroscopic sensor and tracking rotation via rotational velocity, we countered this imbalance. We measured the rotational velocity at a high sample rate and tracked rotation over time. Once the robot was close to reaching the target rotation, we lowered the sample rate to reduce over or undershooting.

As the velocity over time was found to follow a similar logarithmic scale, we could utilise an identical algorithm for tracking movement between grid cells. Using an accelerometer sensor, we measured the robot's acceleration and calculated its velocity, which was used to track distance over time.

A further restriction of the MPU6050 sensor was the minimum sample rate. As each sample read data from the GPIO pins connected to the sensor, we could only get new updates as fast as the I2C communication protocol allowed. We established that sample rates lower than 6ms resulted in erroneous data in testing. A further issue was the reliability of the data. As the sensor was imperfect, it was liable to produce random results occasionally. As this data was being used directly in our rotation and drive calculations, it could lead to short term errors. This was mitigated by the high sample rate, as the error only influenced the function briefly; however it would impact the final result.

We utilised an ultrasonic sensor to identify obstacles in front of the robot or other utility tasks such as identifying if a door is open or closed. This sensor works by emitting a high-frequency wave reflected in the environment. The distance between the sensor and the closest object can be determined by the time between sending and receiving a pulse in seconds, multiplied by the speed of sound in the air. We collected several samples for each reading and utilised the more accurate nanosecond clock offered by the Python time module to collect accurate readings.

An engineering challenge was that the sensor often timed out or gave misleading results, which was combated by increasing our sample size and filtering out of bounds results, which worked well and resulted in more accurate readings. We took the mean result of valid samples, which performed well in testing; however, the median result could also be considered in further development.

Comparions

While other artefacts in the field make use of expensive robotic components, the Thunderborg robot is relatively inexpensive and saves cost with the sensor components used. Our project focuses on maximising performance and overcoming these hardware-based constraints.

Setup / usage

- To use this product, it is assumed that you have a Thunderborg robot, equipped with a Raspberry Pi, HCSR04 Ultrasonic Sensor, a MPU6050 Gyroscope/Accelerometer, and Python 3.X installed.

1. First, you must copy the `/src` directory into the robot.
2. Then you need to install the packages listed in `requirements.txt`.
3. You can then run `src/demo.py` to start the program.

The program parameters can be set in the `instructions.json` file. Here you can set the initial world dimensions and the locations of the start and goal positions.

Example instructions.json

```json
{
    "matrix": [ // define a long corridor environment
        [0,0,0,0,0],
        [0,0,0,0,0],
        [0,0,0,0,0],
        [0,0,0,0,0],
        [0,0,0,0,0],
        [0,0,0,0,0],
        [0,0,0,0,0],
        [0,0,0,0,0],
        [0,0,0,0,0],
        [0,0,0,0,0],
        [0,0,0,0,0],
        [0,0,0,0,0],
        [0,0,0,0,0]
    ],
    "unit_size": 1.5,      // unit size of 1.5 (approx 50cm)
    "start": "x2y12",      // starting coordinates (assumes oriented North)
    "instructions": [      // visit this nodes and perform a door open/close check
        {
            "goal": "x0y6", // visit this node
            "final_rotation": -90    // rotate -90 relative to North
        },
        {
            "goal": "x4y0", // visit this node
            "final_rotation": 90     // rotate 90 degrees relative to North
        }
    ]
}
```

Outcome of running example instructions.json

6	7	8	9	-
5	6	7	8	-
4	5	6	7	8
3	4	5	6	7
2	3	4	5	6
1	2	3	4	5
Goal	1	2	3	4
1	2	3	4	-
2	3	-	-	-
3	-	-	-	-
4	5	6	7	-
5	6	7	-	-
6	7	🚗	-	-

Packages Used:

See our `requirements.txt` for a list of third-party packages used.

We took inspiration from other peoples implementations of D* Lite. Some repositories we looked at include:

- https://github.com/robodhhb/Interactive-D-Star-Lite
- https://github.com/mdeyo/d-star-lite
- https://github.com/avgaydashenko/d_star

Conclusions

We have created an autonomous agent capable of navigating an unknown environment to reach multiple goals and perform surveillance actions. With further development, we would like to extend the instruction interface to a graphical interface accessible remotely from a web server. We would also consider implementing ultrasonic object detection in a constant thread instead of between actions. However, this was not considered in the original project scope due to complications when calculating wave travel distance from a moving sensor.

With access to more advanced hardware, we would upgrade our motors to a balanced set, removing the issues encountered with imbalanced profiles. We could also consider a laser or lidar range finder sensor to perform better object detection around the robot. The addition of a magnetometer would also significantly improve the project's autonomy. It would enable use to code self- localisation behaviour, removing the need for the robot to start from a known position and orientation.

Finally, with further development, we could consider the application of AI to improve the performance of our low-cost sensors. A network could identify sensor errors and perform regression to predict a more accurate value by performing error detection on sensor data.

To conclude, we consider this project worthwhile, exploring a practical application of robotic agents, and meaningful, considering how low-cost electronics can be optimised to deliver maximum performance.