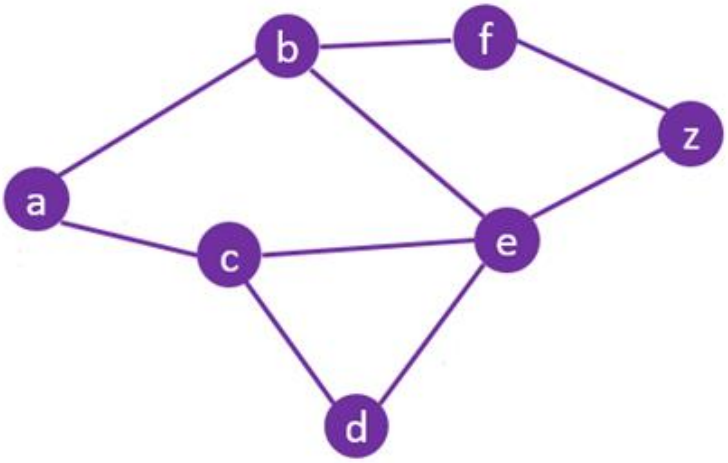

Camera-Guided Line Following Robot with A* Algorithm for Optimal Pathfinding

Practical Work

João Silva 16998

Paulo Macedo 17011

Intelligent Robotics

Master's in Applied Artificial Intelligence

2022/2023

Index

Figure Index

1 Introduction

In this work we implemented a line follower robot designed in a Webots simulation environment, employing camera-based line detection and the A* (A-star) algorithm for efficient pathfinding. The objective was to create an autonomous robot capable of accurately tracking lines and intelligently navigating our city-street-like environment. By leveraging computer vision techniques and the A* algorithm.

2 Webots

Webots is an open-source and multi-platform robot simulator used for developing and testing robotics applications. It provides a virtual environment where you can create and simulate various types of robots, ranging from simple wheeled robots to complex humanoid robots. Webots offers a comprehensive set of tools and features for designing, programming, and simulating robotic systems.

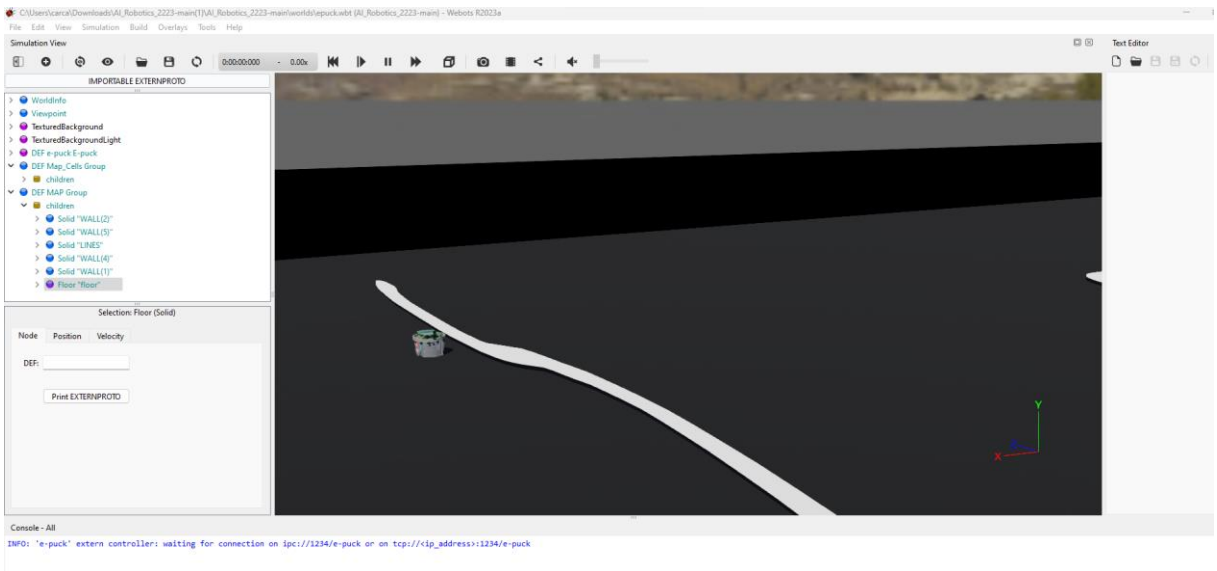

Figure 1 - Webots

2.1 Creating our environment

The virtual environment we created consists of a city-like environment, where we have roads, in our case white lines that our robot will follow, that lead into various intersections connecting

these roads to each other. These lines were drawn in TinkerCAD and exported as an OBJ file, where we later imported it into our virtual environment.

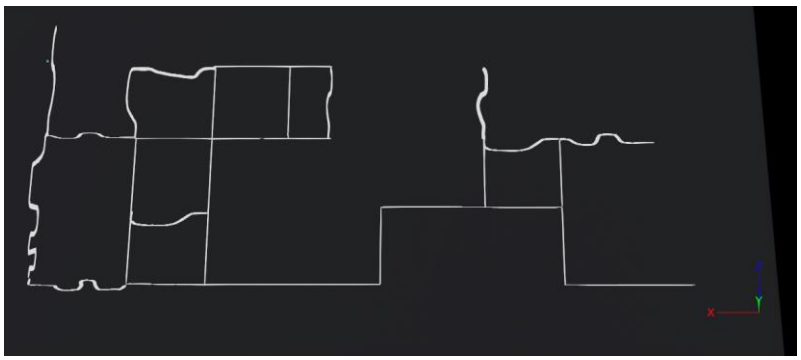

Figure 2 - Environment

Each intersection is marked as an object that contains its coordinates on the environment, since we need to know these to determine whether our robot has reached an intersection.

Figure 3 - Marked intersections

2.2 E-puck robot

The E-puck robot is commonly used in Webots projects and has all the essential features that are required for this work. The most important features we will use are:

- Two independently controlled wheels to be able to move around.
- A downward-facing camera that will be used for line detection. (640X240 resolution)
- A GPS system that enables us to know the robots coordinates in the environment.

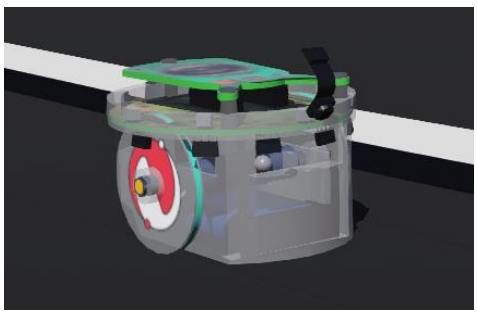

Figure 4 - E-puck robot

3 Line Following Robot

A Line Following Robot is an autonomous robot that is designed with the single purpose of following a line or path marked on the ground. The most common approach is to use light or color sensors. These sensors can detect and differentiate between different colors, allowing the robot to recognize the color of the line and distinguish it from the background.

In this work, instead of using color sensors we chose to utilize a camera that is present on the e-puck robot. Cameras equipped with computer vision algorithms can also be employed to detect and track lines.

3.1 Sensors vs Camera

There are several factors to consider the use of a camera instead of color sensors:

1. **Line Detection Accuracy –** Cameras provide a higher resolution and more detailed visual information compared to color sensors, resulting in improved line detection accuracy. Color sensors, on the other hand, are limited to detecting specific colors and may struggle with variations in lighting conditions.
2. **Smoother Line Following –** The higher resolution images captured by the camera allow for a more precise tracking of the line, resulting in smoother movements and transitions. A Color sensor approach, most of the time utilizes an on-off following method, leading to sporadic and choppy movements, where the robot switches between following the line and making corrections.
3. **Computational Requirements –** Cameras require more computational resources for image processing compared to color sensors. The processing of images in real-time requires more processing power and memory, meaning that if the computer has limited computing power, color sensors are more favorable.

3.2 Computer Vision

In this section, we will explore the key computer vision steps involved in the detection of the line.
The first step in processing the robot's camera image is to convert the RGB (Red, Green, Blue) image into grayscale. This conversion simplifies image analysis steps by reducing the image to a single channel and maintaining only the relevant intensity information without the complexities of color.

Figure 5 - Gray image

Image segmentation is the next step, this aims to partition the grayscale image into distinct regions based on differences in intensity. The segmentation technique used was thresholding. The process of thresholding involves deciding on a threshold value and classifying each pixel in an image as either belonging to the foreground or the background; in our example, this involves choosing which pixels are the line and which pixels are the floor.

The final product of image segmentation is a binary image where we either have white or black pixels. This binary representation simplifies subsequent analysis and allows for easy identification and extraction of the segmented regions.

Figure 6 - Binary image

With the segmented image, we can now try to detect the edges of our line. This involves two algorithms: the Canny edge detection and the Hough Transform.

Canny edge detection is primarily used to detect the edges in an image, it operates by finding regions of significant intensity gradient, which often corresponds to edges. This will also increase contrast and remove image noise.

Figure 7 - Original image compared to Edges image

Hough transform is then applied to determine whether those edges are lines or not and to extract information from those lines, like slope and intercept. Line slopes and intercepts are parameters that define a line equation in the form of $y = mx + b$, where '*m'* is our slope and *'b'* represents the y-intercept.

The Hough transform's effectiveness depends on the accumulated pixels being distinguishable, which means that there must be a clear distinction between a pixel and its neighbors; otherwise, no pixels would stand out as a line or circle. Although canny edge detection is not necessary, it can reduce the computational load for the following Hough transform. Since edge detection focuses on relevant information, the number of points that need to be processed by the Hough transform is significantly reduced.

Figure 8 - Hough Lines

Once we have our edge lines, we can calculate the center between these edges. One thing that needs to be taken into consideration is that the Hough transform algorithm is not perfect and multiple line segments can be detected at the same time, resulting in having more than one "center" point. To mitigate this, the average center point of all detected lines needs to be calculated and used as the center point that our robot will follow.

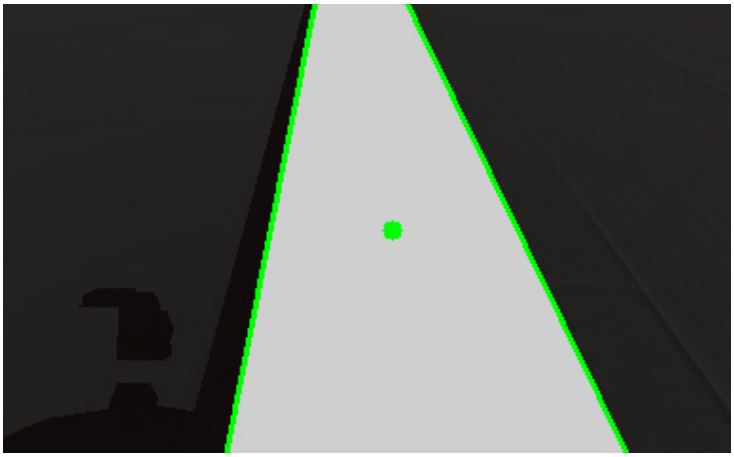

Figure 9 - Line Centroid

3.3 Movement and motors

Now that we have the center of our line, we need to determine the error value that represents the deviation of the line center from the desired position. Since the camera used has a resolution of 640x240, this means that our center point will have the coordinates (320,120). The error value wil the difference of the x coordinates of the calculated center of our line and the x coordinate of our center point (x=320).

To ensure the robot stays centered on the line, we utilize a proportional control gain for each wheel, allowing us to adjust the left and right motors accordingly. This control mechanism enables the robot to counteract any deviations by increasing the speed of the opposite wheel, effectively maintaining its position along the line.

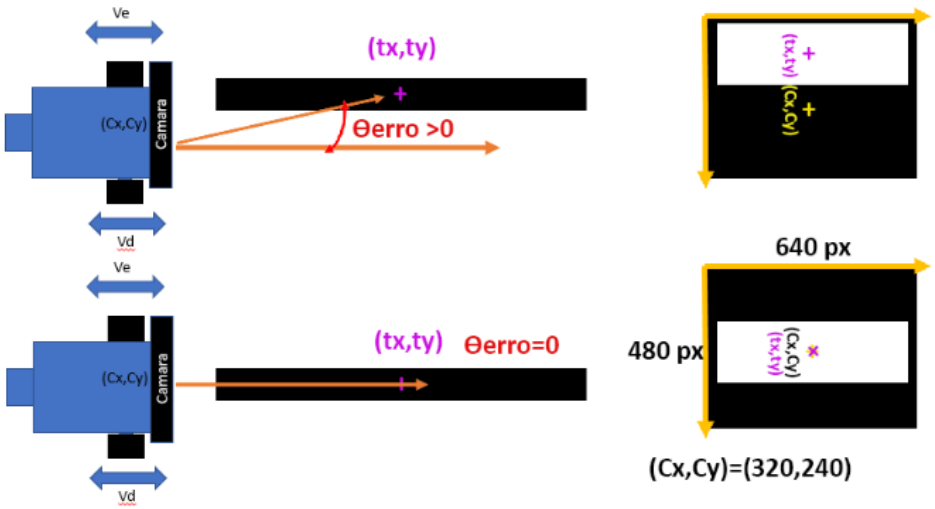

Figure 10 - Robot steering angle

```
left_wheel_speed = Kp +((Ke * error)/desired_position)
right_wheel_speed = Kp -((Ke * error)/desired_position)
```

Figure 11 - Wheel proportional equation

4 A* Search Algorithm

Since the best fit for our movement would be to get to certain positions following a line, our world was then represented by nodes. Having this in mind, we had 2 main options for our pathfinding algorithm: Dijkstra and A*.

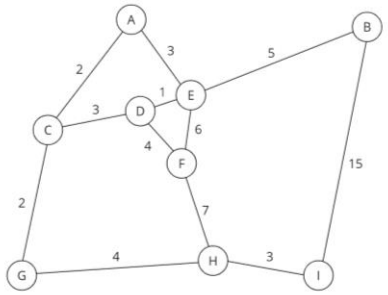

Figure 12 - Graph example

As represented above, our world is a graph that is not very dense and is “grid like”, with most connections forming squares, making the heuristic very approximate to the real distance. With those advantages in mind, we then chose to use the A* Search, since it is the most efficient and appropriate for our problem.

4.1 Implementation details

The A* Search algorithm is implemented as follows:

- **Initialization**: The algorithm begins by initializing the starting node as the current node and creates two lists: the open list and the closed list. The open list contains the nodes that are yet to be explored, while the closed list contains the nodes that have been visited.

- **Evaluation and Selection**: At each iteration, the adjacent nodes of the current node are evaluated. For each neighbor, it calculates two values: the g-cost and the heuristic(h-cost). These values are used to compute the total cost (f-cost = g-cost + h-cost).

- **Update Open List**: The algorithm updates the open list by adding the adjacent nodes not already in the closed list. If a node is already in the open list, it compares the new f-cost with the existing f-cost and if the new one is lower updates the g-cost, h-cost, and parent.

- **Selection of Next Node**: the node with the lowest f-cost is selected from the open list as the next current node.

- **Termination**: The algorithm repeats the above steps until it reaches the goal node or until there are no more nodes in the open list (no possible path).

4.2 Classes

Several classes were implemented to store, maintain, and alter the world representation:

- **Cell**: this class was created to represent a graph cell in the world, keeping track of its name (for representation purposes) and world position. It also stores information needed for the pathfinding algorithm such as the g-cost (value that represents how expensive it is to move since the beginning of the path until the correspondent cell), h-cost (the heuristic to the end objective, represented in our case by the Euclidean distance), the f-cost (sum of both costs) and a parent cell, with is the previous visited cell in the path.

- **Cells Graph**: responsible for creating and storing the initial cells of the graph and "connecting" them (no costs given, only during the pathfinding)

- **Pathfinding**: class responsible for the implementation of the search algorithm

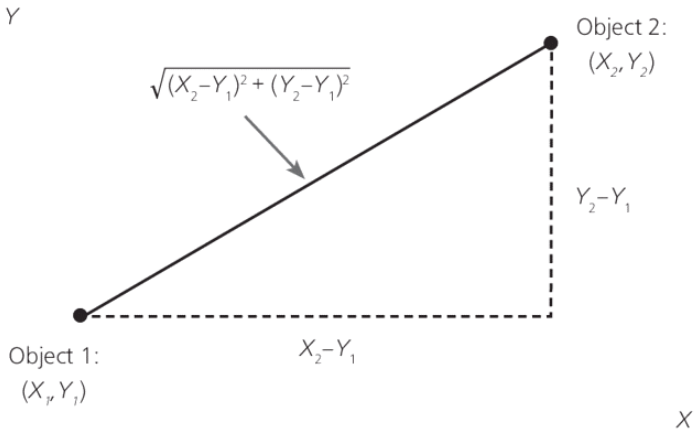

Figure 13 - Euclidean Distance

4.3 Nodes

The nodes have 2 main steps to be represented:

- Creation of transform nodes which have their global positions.
- Creation of cell instance which gets the preposition stored in that node.

5 Robot States

To control our robot, we defined two main states in which it can be: **MOVING** and **ROTATING**. Its movement follows the following steps:

- In the beginning the robot always starts by rotating towards the next point until the angle between its forward vector and the desired orientation (***nextCellPosition*** - ***currentCellPosition***) is smaller than a pre-defined amount.

- After facing the right direction, it then enters the **MOVING** state in which it moves itself using the line following movement.

- When the robot gets into a small area defined around the destination, it starts slowing down until it arrives at the supposed next cell, where it will stop.

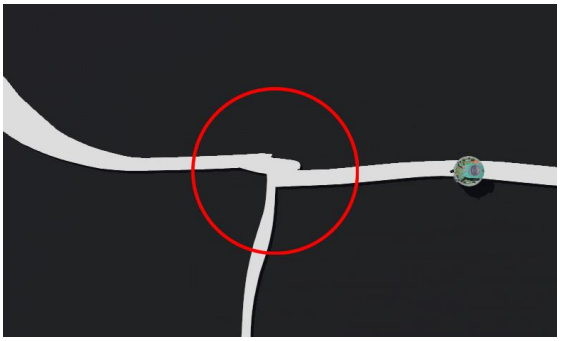

Figure 14 - Slowing down Area.

- When it stops it means it has reached the next point of the path, changing its current cell to the next cell, and going back to the **ROTATING** state, repeating the cycle until it has reached the destination.

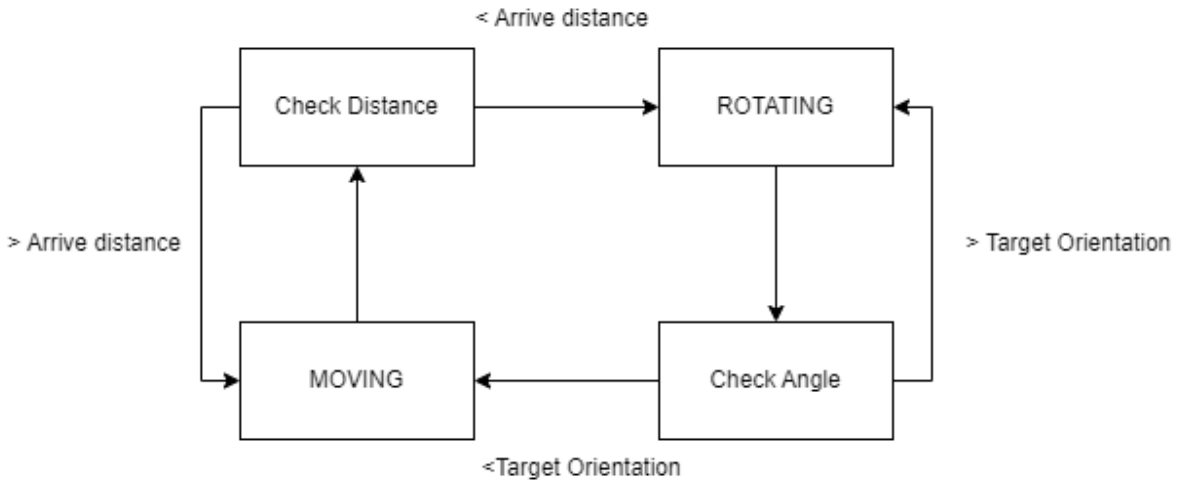

Figure 15 - Robot states Diagram

6 Conclusion

In conclusion, the implementation of our line follower robot was a success since it managed to execute the main goal of the work, which was to follow our lines and find the shortest path to

a destination, assisted by the A* algorithm. However, certain challenges were encountered during the project, highlighted by the learning curve involved in understanding and effectively utilizing the Webots simulation, errors in the calculation of the robot's orientation and the detection of more than one center points for a single line, which was overcome by always averaging the position of multiple lines.

7 Bibliography

https://learnopencv.com/hough-transform-with-opencv-c-python/

https://cyberbotics.com/doc/guide/epuck

https://stackoverflow.com/questions/9310543/whats-the-use-of-canny-before-houghlines-opencv

https://www.geeksforgeeks.org/a-search-algorithm/

https://www.youtube.com/watch?v=luyg3plGujg&list=PLbEU0vp_OQkUwANRMUOM00SXybYQ4TXNF