COSC343 Group 3 - Harry Pittar, Trent Lim, Lachlan Campbell, Shaun Liew

Algorithm Description

The robot's movement can be broken into two different steps; getting to the 'red zone' and searching for the tower. The red zone is the area in which the tower is located. As the robot starts a significant distance away from there, our first goal was finding a consistent way to this zone.

For finding the location, the robot uses its light sensor to detect a black tile. It is calibrated to update its tile count based on the direction it was moving. We mapped the directions based on the robots orientation in its starting position. Moving North would change the location by -15, South: +15, East: +1, West: -1. The methods; turnLeft() and turnRight(), turn the bot in their respective directions and update its direction. The robot is designed to ensure it is always going over black tiles and keeping its position known. Which is vital as this black tile count is used to make the robot turn and report the big tile number.

checkLeft() and checkRight()

Ideally the robot would be moving through the middle of every black tile. However, the robot often loses this line, either by having a poor orientation or being slightly off the middle line. Often it's a combination of the two. To counter this we implemented correction functions. These consist of the methods; correction, checkLeft() and checkRight().

When the robot moves onto a black tile, checkLeft() finds how far away the robot is from the left edge of the tile. It does this by turning slightly to the left before incrementally moving to the edge. This movement is broken into many tiny wheel turns. Each one of these turns is recorded to a variable called counter. A large value for this variable means that the robot is far away from the left edge of the black tile. This is important as it indicates the robot is likely to the right of the centre of the tile.

When the light sensor detects that it is moving off the black tile the robot stops and returns to the initial position. It then returns the counter value to the object that called it. checkRight() works the same way, only going right instead of left.

correction()

This is the main correction method for our robot. When initialised this method calls both checkLeft() and checkRight() in order to get both of their counter values. It then compares these values.

If the left counter is greater than the right counter then the robot is further to the right. To correct this the robot needs to head towards the left. The bot turns leftwards and moves a certain distance across the black tile. This distance is calculated by subtracting the left counter by the right counter. Therefore, this correction distance is variable. When the robot is quite far to the right it will move a relatively significant distance to the left. If the two counters are similar it will

barely readjust, if at all. Once it has moved back towards the centre of the tile, the robot returns it's direction back to the neutral angle, ready to find the next black tile. This obviously works vice versa when the right counter is greater than the left counter. The robot is closer to the left edge and will correct by moving rightwards.

These correction methods are called every time the bot reaches a black tile. This is due to the robots orientation often being of kilter. Resulting in it moving slightly to the side each time it's moving between black tiles. Having the correction() method in place helps ensure that the robot doesn't slowly move off the line of black tiles.

findNextBlack()
As the robot needs to use black tiles to keep its location, we designed the robot to seek out black tiles any time it moves a significant distance. If the bots light sensor detects it that it is one a white tile, it moves forward. The sensor works constantly, the bot continually moves forward while it detects its on a white tile. Once it has found a black tile the robot stops and then runs the correction() method. After the correction the bot updates its location. As mentioned above this value updates based on the direction of the robot. The location is then announced by the robot and the method is returned.

There are instances where findNextBlack() is called while the sensor is on a black tile. This tile would have already been found by a previous call of this method and therefore doesn't need to be found. To fix this the robot moves forward until it detects that it is off the tile. The method then calls itself. Starting another instance of this method but in the correct position.

findTower()
The robot searches for the tower by using its ultrasonic sensor. This sensor is used to locate the distance the robot is away from the tower, and alters its behaviour based on said distance. The tower had to be on one of 12 tiles in a 4x3 grid. As the tower had to be in one of four rows, the robot simply has to search each row until it senses the tower. Starting in the top left corner of the zone, the robot pivots leftwards and uses the ultrasonic sensor to try to detect the tower in that row. If the tower is not detected the robot will move downwards to the next row. This behaviour continues until the tower is picked up by the ultrasonic sensor.

FindFinalTile()
When the tower is detected, the bot locates the big tile number based on the distance between the two objects. When looking down the row, there are three possible distances that the sensor will recognise which corresponds to either tile 1, 2 ,3 or 4, 5, 6 etc. Then the robot announces the number and then begins traversing the row. The robot stops when its sensor indicates that it's right next to the tower.

initialize()
This is the main function of the project. Like its namesake it initializes the robot by combining previous functions and calling them accordingly. The position our robot starts and the position of the red zone are fixed. Since the tower must be in the red zone there is no need to search for it outside of this area. The first leg of the robot's journey is then navigating towards the sector, recording its position as it goes. As for the actual method of reaching this zone, we made the bot navigate across the board and then downwards once it reached a certain tile. The initial beginning of the first turn and the time to execute the next following turns all come from the initialize function. Firstly, the robot will move upwards towards Tile 1 and then turn right, moving along the upper rows of tiles. Once the robot had reached Tile 10 it would turn right again and navigate downwards. The tile count would then increase by 15 at each instance. After moving down 4 tiles it would reach a tile count of 55 and begin searching for the tower.

Problems we faced

Many of the problems that occurred were due to variability. Inconsistencies would occur between tests from the light sensor and the robot's movements. These problems were not always seen in the simulator, so we had to alter the robot's behaviour when transferring to the real world.

Inconsistent light levels and other visual interference caused our light sensor to not work optimally. The robot would often not identify black tiles when driven over. This leads to an incorrect tile count and, in some instances, total failure in finding the tower. As the robot used tiles to navigate and record its position, it was imperative to try to reduce or eliminate this issue. This was done by allowing for some variability in what qualifies as a black tile. Simply having the robot react when it saw the colour black wasn't enough. We added leniency to the colour sensor to try to accommodate for slight colour differences, light reflection and different ambient lighting effects. By being less strict on what is classified as a black tile, the amount of errors in our robot significantly dropped. The leniency on this classification is a balancing act however, as being too liberal with what constitutes as a black tile could result in false positives. There were instances where the darker tile grout would be mistaken for a black tile. Resulting in the robot believing it had driven much further than it actually had and us having to readjust the sensor's values. There still could be instances where a black tile is not detected, but we have to compare the risk of missing a tile and the risk of a false positive.

The perfect robot would be able to consistently pull off perfect 90° turns. What we found however is that there would be slight variability in the robots turning circle. Steering, dips between tiles and misalignment at the starting positions caused drift in our test runs. The robot's orientation would become slightly askew, sometimes causing it to miss tiles. Initially, this disorientation was causing the bot to lose its position and would require human intervention to re-right itself. Our goal then was to find a way for the robot to detect and counteract the drift. This was done by implementing the correction method above.

We found that our code needed to be reconfigured when transitioning from the simulator to the physical bot. When using the actual robot there was more variability, as alluded to above, but also aspects like momentum and the tile surface which affected the robots functions. This was not difficult to fix. Inputs like motor power just needed to be recalibrated to get the desired turning and distance. It did slow us down however, as tweaks to our code would take longer to set up and test on the physical bot compared to the simulator.

Variability drastically increased the difficulty in developing our robot. Straightforward methods and design ideas had to account for all these inconsistencies. Through trial and error however, we managed to develop strategies to persevere through randomness and create a fully independent robot.