Vehicle Design:
The initial design for our robot was the one that came with the how-to handout. That design was chosen because it was simple, fast, and the instructions to build were already there.

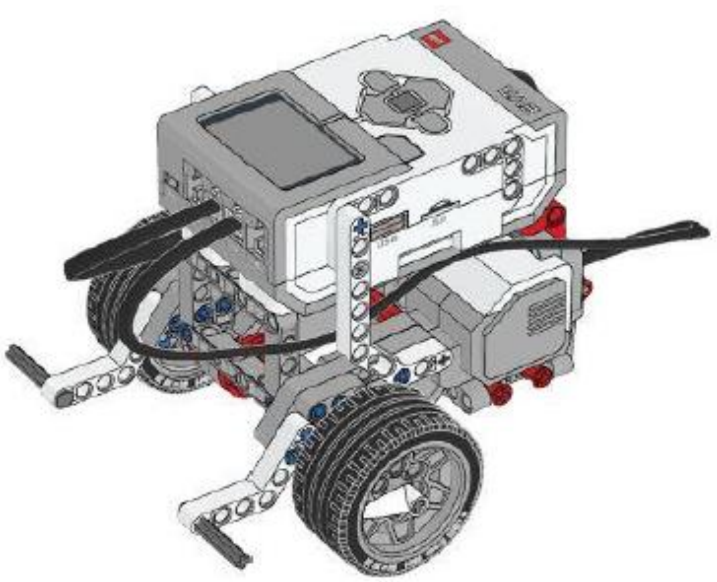

Fig: Initial design of the robot

However, during the final stages of testing, ball of the robot was getting stuck on the tapes on the ground while rotating. And after further testing, it was determined that the weight on the ball was more than what we needed. So, it was decided that the weight would have to be shifted towards the tire. It was also decided that gears were necessary for precision. So, necessary design changes were made, and the following design was achieved.

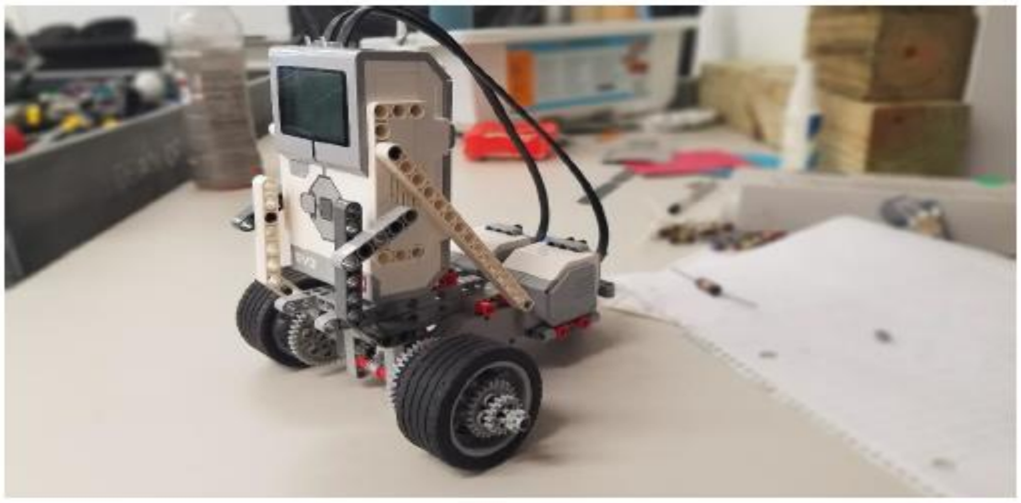

Fig: Final design of the robot

Map Design:
It was decided that the robot environment would be represented in a matrix form. Every element of the matrix represents the intersection point of the tiles. It was also decided that the real-world squares would be divided into 4 virtual squares. Doing so, the rows and columns of the matrix increased by a factor of 2. It also allowed the robot to have a buffer zone. Essentially, dividing the real-world square tiles to four virtual tiles allowed us to virtually increase the obstacles by half a tile. Since the project description defined the path to be two tiles wide, this allowed us to find a path with a padding of half a tile on both sides.

Path Planning
For pathfinding, it was decided that Dijkstra's algorithm would be used. So, the algorithm was used to populate the environment matrix with the cost to reach the goal from the starting position. Then, we used an algorithm that backtracks from goal position to start position. We make our start position as 0 distance and all other position as maximum distance. We explored positive X, positive Y, negative X, negative Y direction (up, down, left, right) from current location (Current location start with goal) and saved it in queue. We increment distance by one unit from its parent location. Once all neighbor of current location explored, we marked it as visited. We read top element from our queue and repeat the same step until we reached the goal state. Once we reached goal state, we backtrack our path using stack and generate the required path from start to goal.

Navigation
To simplify our control, we only allow the robot to perform either (1) moving forward a distance in millimeters or (2) turning clockwise robot. Once we have the path from our path planning algorithm, which is a list of coordinates in our environment matrix, we need to convert them to a series of instructions to the robot. We accomplish this by first choosing the original orientation of the robot to be in positive x-axis. At each step, we consider the current coordinates and orientation of the robot and the next coordinates. The number of turns needed and the distance (in millimeters) are computed and the instructions are sent to the robot.

Control
The rotate clockwise function, `void turnclockwise(int theta, int speed, int* deg_L, int* deg_R,int *clock_offset,` takes in the angle to rotate and the speed at which to rotate as input. All the other parameters are unused. The function takes the angle input, converts it to radians, and using the radius of the axle, determines the distance the tires need to travel. The robot then converts the distance to be travelled to the motor rotation and executes the motor functions provided by the c4ev3 library. Similarly, the move forward function, void `move_forward(int length,int speed, int* deg_L, int* deg_R,int *clock_offset)`, takes in the length to move and the speed at which to move. The other parameters are unused. This function takes in the length to travel in mm and converts it to the motor rotation count. The motors are then rotated until the rotation count is reached.