Team 12: Ethan Jobe, Yen Duyen Amy Le, Tim Tran
CSE 4360 001

Project 1: Autonomous Robots

Description of robot design

The main control factors for the robot involve two motors to propel the robot forward and steer, and a caster ball wheel to uphold the structure and allow smooth movement in all directions. As seen in Figure 1 below, both motorized wheels are set up side by side one another, making this robot act as a single axle vehicle, with the caster ball wheel set behind wheels. Note that depending on the caster ball distance from the front wheels, the arc rotation would change. So in the design, we moved the caster ball closer to the front wheels in order to minimize the arc. Overall, the robot maneuvers as a differential drive robot with a third non-actuator wheel.

When creating the robot, we encountered the problem that the robot would not balance on its own with only two wheels, so we were forced to find a way to use a caster. Creating a caster wheel did not have the fluidity required for returning, so we opted to use the caster ball. Also, when mounting the computer on the robot, we also found that mounting the computer on the back of the robot directly above the robot would be suboptimal, as the weight of the computer on the metal caster ball affected how well the ball would work. The ball would have too much weight and the friction would cause the ball to not move as well. Thus, we endeavored to find a way to distribute the computer's weight as evenly as possible.

Although there was an option to use a gyro, we were not aware of the option until the robot and algorithm were done, so we chose not to use it.

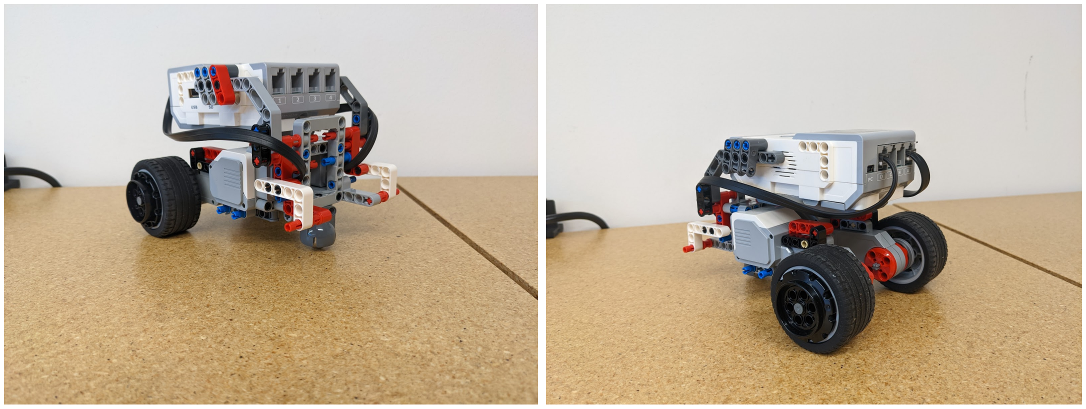

Figure 1

Description of navigation strategy

Our team decided to opt for a potential field implementation that used Manhattan distance to navigate the obstacles. Manhattan distance is a simpler potential field implementation compared to goal and obstacle potentials. In turn, this meant that our implementation was easier to create. Manhattan distance is the distance from the start point to the goal along a grid of cells. The grid generated by the strategy was implemented by following the tile sizes provided by the requirements. Due to the constraints of our robot, to implement the strategy, we also had to provide the orientation of the robot in order to have the robot go in the correct direction.

To rotate the robot, we rotated both wheels of the robot. This was in order to not have the robot pivot on one wheel and instead rotate in place about the center between the two wheels. To rotate 90 degrees, each motor was told to turn 180 degrees in opposite directions, depending on the turn. We measured the radius of the wheels to be 1 inch and used that to find that to go the distance of 1 tile, the wheel would need to turn 687.5493 degrees. We set the robot to move at a speed of 200 to reduce the amount of slipping that occurs when the wheels move too quickly.

Although we tried to account for the slip of the wheel when running the algorithm (hence the extra degrees added or subtracted from the wheel rotations in the code), there was still a large variation in the accuracy of the robot's movement. The robot would at times not turn enough and at times turn too much. These problems would occur on concurrent algorithm runs and we were forced to conclude that the floor created too much variation in the grip to run the robot consistently.

Description of algorithm

The algorithm is a greedy algorithm that produces the shortest Manhattan distance to the goal. Manhattan distance potential field navigation works by imposing a grid onto the workspace, and assigning each navigable cell the Manhattan distance of the cell from the cell(s) where the goal is located. To make it simple, the goal cells are assigned a distance of 0 (because they have zero distance from the goal), and the cells that contain an obstacle are assigned the max integer value (so that the large distance repels the robot from navigating into the obstacle cells). Then, the path is found by starting at the start cell, then navigating towards a grid cell with a lower value.

The workspace grid navigated by the robot is represented in the code by a two-dimensional matrix, where each cell represents a square floor tile in the workspace. To map the obstacles and goal to the matrix, the obstacles and goal coordinates were first scaled to the matrix by dividing the coordinates by the length of the tile. Because the given coordinates of the obstacles coincide with the intersection between four tiles, and each obstacle is the size of a square floor tile, each obstacle partially covers four tiles. To make it simple, any cell that contains any part of an obstacle is marked as an obstacle cell. For each obstacle, its scaled coordinates are used as the indices of the four obstacle cells that result from the obstacle, and those indices are used to mark the four cells as obstacle cells by assigning them the max integer value.

```
    obstacleScaled = []

    for idx, obs in enumerate(obstacle):
        if(obs[0] >= 0 and obs[1] >= 0):
            obstacleCenterX = round(obs[0] / tile)
            obstacleCenterY = round(obs[1] / tile)
            obstacleScaled.insert(idx, [obstacleCenterX,
obstacleCenterY])
```

```
    for idx, obs in enumerate(obstacleScaled):
        grid[obs[1]-1][obs[0]-1] = sys.maxsize
        grid[obs[1]-1][obs[0]] = sys.maxsize
        grid[obs[1]][obs[0]-1] = sys.maxsize
        grid[obs[1]][obs[0]] = sys.maxsize
```

Similarly, the goal also partially covers four tiles because the center of the goal, which is a circle with the diameter of two tiles, is located at the intersection between four tiles, thus the scaled goal coordinates were used as the indices of the four goal cells that result from the goal, and those indices are used to mark the four cells as goal cells by assigning them 0.

```
    goalScaled = [round(goal[0] / tile), round(goal[1] / tile)]

    grid[goalScaled[1]-1][goalScaled[0]-1] = 0
    grid[goalScaled[1]-1][goalScaled[0]] = 0
    grid[goalScaled[1]][goalScaled[0]-1] = 0
    grid[goalScaled[1]][goalScaled[0]] = 0
```

After the obstacle cells and goal cells are assigned their appropriate values, the rest of the cells in the workspace need to have their Manhattan distance from the goal cells calculated. The Manhattan distances are calculated starting from a chosen goal cell (out of four). The order of the cells to be assigned Manhattan distances is important because the Manhattan distance of a cell will be based on the Manhattan distance of a Manhattan neighboring cell instead of a direct calculation based on the goal cell. To maintain this order, a queue is implemented.

To determine which cell is the start cell (the first cell to be enqueued), the algorithm has to find a non-obstacle and non-goal cell that borders the chosen goal cell to enqueue as the start cell. A single goal cell that is to be used to find the start cell cannot be arbitrarily chosen. For example, if the goal is placed at the top right corner of the workspace, the four goal cells that can potentially be chosen to find the start cell are the four cells in the top right corner. However, the top right cell out of those four cells cannot be chosen as the goal cell to find the start cell, for out of the two cells that it is a Manhattan neighbor of, both are also goal-cells, but the start cell needs to be non-goal and non-obstacle cell.

```
def StartManhattan(goalGridIndices):
    start = None
    if goalGridIndices[1] < 15:
        if grid[goalGridIndices[0]][goalGridIndices[1] + 1] == None:
            start = (goal[0], goal[1] + 1)
    if goalGridIndices[1] > 0:
        if grid[goalGridIndices[0]][goalGridIndices[1] - 1] == None:
            start = (goalGridIndices[0], goalGridIndices[1] - 1)
    if goalGridIndices[0] < 9:
        if grid[goalGridIndices[0]+1][goalGridIndices[1]] == None:
            start = (goalGridIndices[0] + 1, goalGridIndices[1])
    if goalGridIndices[1] > 0:
        if grid[goalGridIndices[0]-1][goalGridIndices[1]] == None:
            start = (goalGridIndices[0] - 1, goalGridIndices[1])
    if start == None:
        return False
    else:
        queue.append(start)
        while len(queue) != 0:
            cell = queue.pop(0)
            Manhattan(cell[0], cell[1])
        return True
```

To overcome this problem, one of the goal cells is chosen to attempt to find the start cell until a valid start cell is found. If a valid start cell is not found after trying all goal cells (for example, the goal is completely surrounded by obstacle cells), then the algorithm ends concluding that no Manhattan path can be found from the start to the goal.

```
def StartManhattanHelper(goalCoordinates):
    goalGridIndices = [(goalCoordinates[1], goalCoordinates[0]),
(goalCoordinates[1], goalCoordinates[0]-1), (goalCoordinates[1]-1,
goalCoordinates[0]), (goalCoordinates[1]-1, goalCoordinates[0]-1)]
    for i in range(4):
        if StartManhattan(goalGridIndices[i]) == True:
            break
```

After enqueueing the start cell, the front cell in the queue is dequeued so that its Manhattan distance from the chosen goal cell can be assigned. Up to four cells can be the Manhattan neighbor of the dequeued cell. If any of those neighbor cells do not yet have a Manhattan distance assigned to it, then those specific cells cannot be used to determine the Manhattan distance of the dequeued cell, so those specific cells are enqueued so that the Manhattan distances of those cells can be found later. Out of the neighbor cells that do already have a Manhattan distance assigned to it, the neighbor cell with the minimum distance is identified such that the dequeued cell's Manhattan distance is the neighbor's cell plus 1. After the Manhattan distance of the dequeued cell is assigned, the front cell in the queue is dequeued again, repeating this process until the queue is empty i.e. every cell in the grid has an assigned value to it.

```
def Manhattan(i, j):
    min = sys.maxsize
    if i < 9:
        if grid[i+1][j] != None:
            if (grid[i+1][j] + 1) < min:
                min = grid[i+1][j] + 1
        else:
            queue.append((i+1, j))
    if i > 0:
        if grid[i-1][j] != None:
            if (grid[i-1][j] + 1) < min:
                min = grid[i-1][j] + 1
        else:
            queue.append((i-1, j))
    if j > 0:
        if grid[i][j-1] != None:
            if (grid[i][j-1] + 1) < min:
                min = grid[i][j-1] + 1
        else:
            queue.append((i, j-1))
    if j < 15:
        if grid[i][j+1] != None:
            if (grid[i][j+1] + 1) < min:
                min = grid[i][j+1] + 1
        else:
            queue.append((i, j+1))
    grid[i][j] = min
```

To start navigating, the algorithm has to find the cell the robot first navigates to (the robot starts at an intersection between four cells, not at the center of the cell). Out of the four cells associated with the starting coordinates, the algorithm chooses the cell with the lowest calculated Manhattan distance from the goal. The algorithm saves which quadrant the chosen cell is relative to the starting coordinates.

```
def FindStartCell(startCoordinates):
    startGridIndices = [(startCoordinates[1], startCoordinates[0]),
(startCoordinates[1], startCoordinates[0]-1), (startCoordinates[1]-1,
startCoordinates[0]), (startCoordinates[1]-1, startCoordinates[0]-1)]
    minValue = sys.maxsize
    minIndices = None
    minQuadrant = 0
    iteration = 1
    for indices in startGridIndices:
        if grid[indices[0]][indices[1]] < minValue:
            minValue = grid[indices[0]][indices[1]]
            minIndices = indices
            minQuadrant = iteration
        iteration += 1
    return(minIndices, minQuadrant)
```

After the start cell is chosen, the robot performs specific movements based on which quadrant the chosen cell is in order to get to the center of the start cell.

```
def GoToStartCell(quadrant):
    # robot placed facing right
    if quadrant == 3:
        print("quadrant 3")
        motor_left.run_angle(200, 344, wait = False) # go forward
        motor_right.run_angle(200, 344) # go forward
        motor_left.run_angle(200, 180, wait = False) # turn right
        motor_right.run_angle(200, -180) # turn right
        motor_left.run_angle(200, 344, wait = False) # go forward
        motor_right.run_angle(200, 344) # go forward
        motor_right.run_angle(200, 180, wait = False) # turn left
        motor_left.run_angle(200, -180) # turn left
    elif quadrant == 1:
        motor_left.run_angle(200, 344, wait = False) # go forward
        motor_right.run_angle(200, 344) # go forward
        motor_right.run_angle(200, 180 - 2, wait = False) # turn left
        motor_left.run_angle(200, -180 + 2) # turn left
        motor_left.run_angle(200, 344, wait = False) # go forward
        motor_right.run_angle(200, 344) # go forward
        motor_left.run_angle(200, 180, wait = False) # turn right
        motor_right.run_angle(200, -180) # turn right
    elif quadrant == 4:
        motor_left.run_angle(200, -344, wait = False) # go backwards
        motor_right.run_angle(200, -344) # go backwards
        motor_left.run_angle(200, 180, wait = False) # turn right
        motor_right.run_angle(200, -180) # turn right
        motor_left.run_angle(200, 344, wait = False) # go forward
        motor_right.run_angle(200, 344) # go forward
        motor_right.run_angle(200, 180, wait = False) # turn left
        motor_left.run_angle(200, -180) # turn left
    elif quadrant == 2:
        motor_left.run_angle(200, -344, wait = False) # go backwards
        motor_right.run_angle(200, -344) # go backwards
        motor_right.run_angle(200, 180, wait = False) # turn left
        motor_left.run_angle(200, -180)
        motor_left.run_angle(200, 344, wait = False) # go forward
        motor_right.run_angle(200, 344) # go forward
        motor_left.run_angle(200, 180, wait = False) # turn right
        motor_right.run_angle(200, -180)
```

Then, the algorithm checks if this cell is already a goal cell i.e. has Manhattan distance of 0. If the cell is not already a goal cell, then the algorithm checks the Manhattan neighbors of the cell (up to four) to find the cell with the lowest Manhattan distance. Then, the algorithm tells the robot to navigate into that cell, and the process of finding and navigating into the neighboring cell with the lowest Manhattan distance continues until the robot is in a goal cell.

```
def Navigate(start, orientation):
    NavigateFrom(start[0], start[1], orientation)

def NavigateFrom(i, j, orientation):
    if(grid[i][j] == 0):
        print("goal reached!")
        return
    minValue = sys.maxsize
    minIndices = None
    minOrientation = None
    if i < 9:
        if grid[i+1][j] != None:
            if (grid[i+1][j]) < minValue:
                minValue = grid[i+1][j]
                minIndices = (i+1, j)
                minOrientation = 90 # up
    if i > 0:
        if grid[i-1][j] != None:
            if (grid[i-1][j]) < minValue:
                minValue = grid[i-1][j]
                minIndices = (i-1, j)
                minOrientation = 270 # down
    if j > 0:
        if grid[i][j-1] != None:
            if (grid[i][j-1]) < minValue:
                minValue = grid[i][j-1]
                minIndices = (i, j-1)
                minOrientation = 180 # left
    if j < 15:
        if grid[i][j+1] != None:
            if (grid[i][j+1]) < minValue:
                minValue = grid[i][j+1]
                minIndices = (i, j+1)
                minOrientation = 0 # 'right'
    if minIndices == None:
        print("no path exists")
    else:
        print(minValue, minOrientation)
        Turn(orientation, minOrientation)
        GoForward()
        NavigateFrom(minIndices[0], minIndices[1], minOrientation)
```

Conclusion

In conclusion, the creation of our algorithm was successful, but the translation to real time runs was not. The variation created by the inconsistent floor grip created too many errors to accumulate over runs that could not be accounted for because the amount of grip changed each run. However, on runs that did not have a great amount of errors, the efficacy of our algorithm was properly showcased and the robot was able to somewhat reach the goal.

In the future, we would probably look deeper into the motor functionality and see if there was a way to keep track of the amount of slip that the robot suffered from to limit the accruement of errors. Doing so would eliminate most, if not all, of the problems that our real-world implementation suffers from and would showcase the algorithm to its fullest capabilities.