CSE 4360 Autonomous Robots
Project 1: Navigating a Known Obstacle Course

Team Members: Angelina Abuhilal, Soli Ateefa, Nebi Malik, Khoi Tran
Date: November 11, 2025

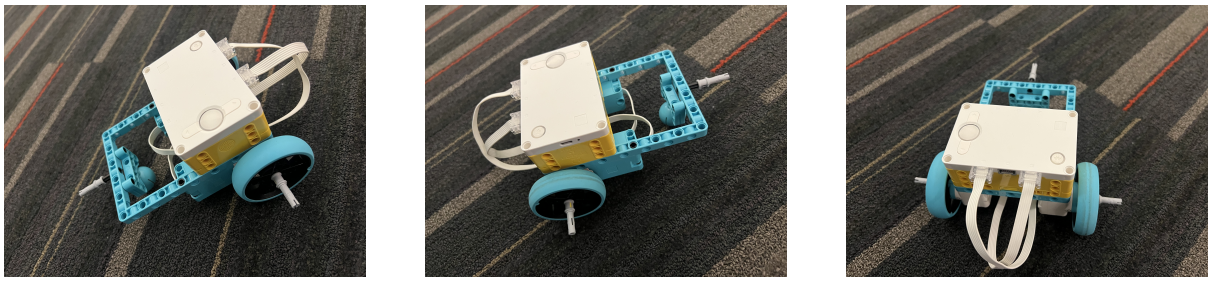
1. Robot Design

Our robot is a simple two-wheel differential drive built with the LEGO PrimeHub and two motors (Ports B and F). Both wheels share a common axle, and a small caster wheel in the back provides stability.

We used the motor encoders to measure how far the robot travels and to make 90° turns. The robot moves one tile (about 0.305 m) per step.

To improve accuracy:

- The robot moves slowly to reduce dead-reckoning errors.
- The chassis is flat and balanced to prevent drift while turning.



(a) Left Side View (b) Right Side View (c) Top View

Figure 1: Our LEGO robot with PrimeHub and dual-motor drive.

2. Navigation Strategy

We programmed the robot to move from a start point to a goal while avoiding obstacles. The workspace is a 16×10 grid where each tile represents one square foot.

Our navigation system uses two algorithms:

1. **Brushfire Algorithm:** Calculates the distance of each free tile to the nearest obstacle. This helps the robot stay centered in open paths.
2. **Wavefront Algorithm:** Starts from the goal and spreads values outward to represent distance-to-goal. The robot follows decreasing numbers to reach the goal efficiently.

Note: Obstacle inflation was removed because the inflate routine did not behave as expected in testing; the planner now uses the raw grid obstacles and relies on safe path selection and conservative motion speeds for safety.

After finding the path, we smooth it to remove unnecessary stops so the robot moves in straight lines between corners.

3. Code Summary

3.1 Path Planning

The main function combines both algorithms to produce a smooth path. (Inflation of obstacles was removed — the planner now operates on the original grid.)

Listing 1: Path Planning Function (no obstacle inflation)

```
def planPath(grid, start, goal, safetyMargin=0, return_raw=False)
   :
    H, W = len(grid), len(grid[0])
    assertPoint(start, W, H, "start")
    assertPoint(goal,  W, H, "goal")
    g = inflateObstacles(grid, safetyMargin)

    sx, sy = start; gx, gy = goal
    if g[sy][sx] == 1:
        raise ValueError("Start␣lies␣in␣inflated␣obstacle;␣reduce
           ␣safetyMargin␣or␣move␣start.")
    if g[gy][gx] == 1:
        raise ValueError("Goal␣lies␣in␣inflated␣obstacle;␣reduce␣
           safetyMargin␣or␣move␣goal.")

    clearance = obstacleDistance(g)
    wave = wavefrontGoal(g, goal)
    raw = extractedPath(start, goal, wave, clearance)
    if not raw:
        return [] if not return_raw else ([], [])
    smooth = smoothPath(raw)
    return (raw, smooth) if return_raw else smooth
```

3.2 Wavefront Algorithm

Here is the wavefront implementation we used (starts from the goal and floods outward).

Listing 2: Wavefront Algorithm

```
def wavefrontGoal(grid, goal):
    gx, gy = goal
    H, W = len(grid), len(grid[0])
    INF = 10**9
    wave = [[(-1 if grid[y][x] == 1 else INF) for x in range(W)]
       for y in range(H)]
    q = [(gx, gy)]
    wave[gy][gx] = 0
    NX, NY = (1, -1, 0, 0), (0, 0, 1, -1)
    while q:
        ux, uy = q.pop(0)
        for k in range(4):
            vx, vy = ux + NX[k], uy + NY[k]
            if not inb(vx, vy, W, H) or wave[vy][vx] == -1:
                continue
```

```
            cand = wave[uy][ux] + 1
            if cand < wave[vy][vx]:
                wave[vy][vx] = cand
                q.append((vx, vy))
    return wave
```

3.3 Motion Control

Movement commands are written in Python using Pybricks:

Listing 3: Basic Motion Primitives

```
def forward():
    leftMotor.run(-200)
    rightMotor.run(200)
    wait(3050)
    leftMotor.stop()
    rightMotor.stop()
    wait(100)

def turnRight():
    leftMotor.run(-100)
    rightMotor.run(-100)
    wait(1820)
    leftMotor.stop()
    rightMotor.stop()
    wait(100)

def turnLeft():
    leftMotor.run(100)
    rightMotor.run(100)
    wait(1820)
    leftMotor.stop()
    rightMotor.stop()
    wait(100)
```

Each command moves one tile or turns the robot 90 degrees.

3.4 Integration Steps

1. Load obstacle, start, and goal coordinates.
2. Build a 2D grid representation of the map.
3. Call `planPath()` to compute the route.
4. Use `forward()`, `turnLeft()`, and `turnRight()` to execute each step.

4. Testing

We tested the robot in three scenarios:

Test	Description	Result
1	No obstacles	Reached goal in straight line
2	One obstacle in middle	Took detour around obstacle
3	Multiple obstacles	Stayed within safe areas, reached goal

Table 1: Test scenarios and results.

The robot stayed within the workspace and avoided collisions. Lower speeds led to better accuracy and consistent performance.

5. Conclusion

Our robot successfully navigated a known obstacle course using Brushfire and Wavefront path planning. Because obstacle inflation produced incorrect results in testing, the planner operates on the raw grid and relies on the clearance computed by `obstacleDistance()` together with conservative motion speeds to maintain safety. Despite small dead-reckoning errors, the robot reached the goal consistently when moving slowly. This project demonstrated effective grid-based planning and safe motion control in a simple LEGO platform.