Name: Sim Justin

Matric No: A0257926N

Project 2: Path Planning

Initial Conditions

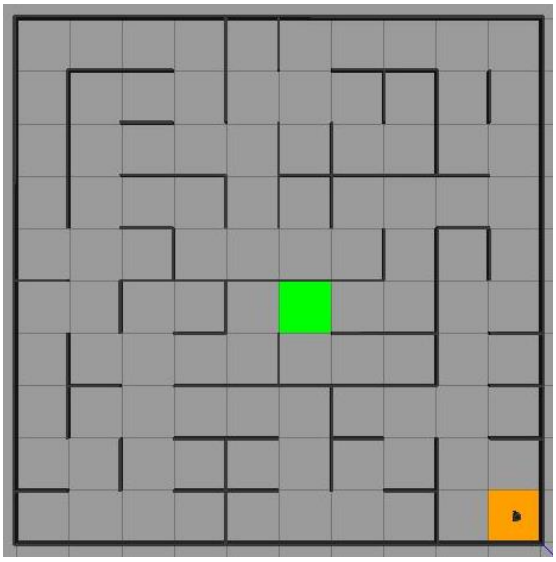

Figure I: Initial Maze Configuration

Starting Position: (0.5, 0.5)

Target Location: (4.5, 4.5)

Path Planning

How it Works

The path planning algorithm makes use of a BFS algorithm that explores all potential paths of the same length before exploring paths of a longer length.

This is done by forming a tentative path (stored in a queue) towards the goal with priority on translation along the x-axis followed by that along the y-axis. As the robot traverses along this tentative path, it will identify walls in its surroundings that will prevent it from following this path, updating the path based on this information until it reaches the goal.

This queue becomes empty when either (1) the robot reaches the goal or (2) the robot has navigated throughout all possible paths and found that there is no possible path to the goal.

Poll the Queue

This portion of the code identifies, based on the tentative path, the current cell of the robot and converts this to x, y indices.

Find the path if the goal is found and break the loop

This portion of the code checks if the robot's current cell is the goal cell. If it is in the goal cell, the robot halts the path planning algorithm and stores the path taken in an array.

Search neighbors and queue them if cheaper

Check if there are walls between the current cell and each neighbor. For each neighbor, check if there are shorter paths to this neighbor and update if so. The tentative path is updated based on this.

Path Planned

(1), In reverse

```
BFS Path: (4,4), (4,3), (4,2), (4,1), (4,0), (3,0), (2,0), (1,0), (0,0)
```

(2), In reverese

```
BFS Path: (4,4), (4,3), (4,2), (4,1), (3,1), (2,1)
```

(3) Final Path Traversed

(0,0), (1,0), (1,1), (2,1), (3,1), (4,1), (5,1), (4,1), (3,1), (2,1), (2,2), (2,3), (2,2), (1,2), (1,3), (0,3), (0,4), (1,4), (2,4), (2,5), (2,6), (2,7), (3,7), (4,7), (4,6), (4,7), (3,7), (3,6), (3,5), (4,5), (4,4)

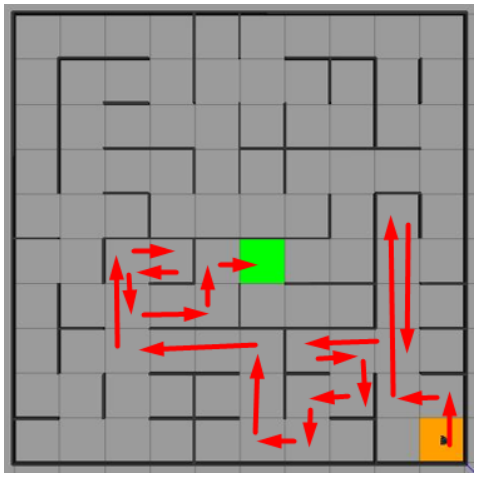

Figure II: Path Traversed

(3) Shortest Path Found

(0,0), (1,0), (1,1), (2,1), (2,2), (1,2), (1,3), (0,3), (0,4), (1,4), (2,4), (2,5), (2,6), (2,7), (3,7), (3,6), (3,5), (4,5), (4,4)

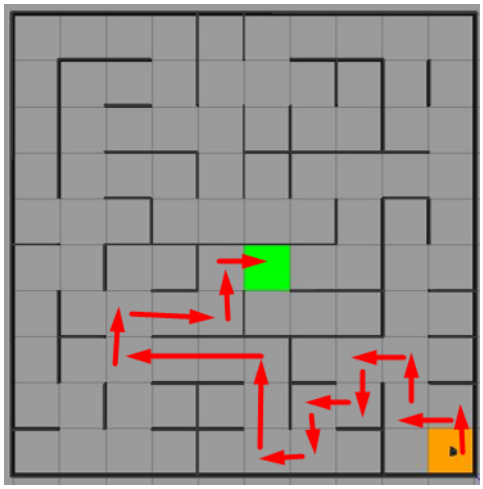

Figure III: Shortest Path Found

The paths are different as the path planned initially is tentative and does not account for the presence of the walls. As the robot traverses the maze, it can map out the position of the walls it encounters and replans the path based on this information, resulting in a different path being taken.

Navigation and PID Control

Code Added

```
// === (b) PID SUMS ===
// ENTER YOUR CODE HERE

p_term_x = Kp_x * error_pos;
p_term_a = Kp_a * error_heading;

if (p_term_x > -MAX_LINEAR_VEL && p_term_x < MAX_LINEAR_VEL)
  I_linear += (error_pos + error_pos_prev) / 2 * dt;
if (p_term_a > -MAX_ANGULAR_VEL && p_term_a < MAX_ANGULAR_VEL)
  I_angular += (error_heading + error_heading_prev) / 2 * dt;

D_linear = (error_pos - error_pos_prev) / dt;
D_angular = (error_heading - error_heading_prev) / dt;

vel_x = p_term_x + Ki_x * I_linear + Kd_x * D_linear;
vel_heading = p_term_a + Ki_a * I_angular + Kd_a * D_linear;
```

Figure IV: Code for PID Control

PID control was implemented as per in project 1. Trapezoidal Riemann Sum was used as opposed to Traditional Riemann Sum once again for improved accuracy of integral term.

Control Gains

```
Kp_a: 1.5
Ki_a: 0
Kd_a: 0.5

Kp_x: 1.5
Ki_x: 0.5
Kd_x: 0.5

dt: 0.1
```

Figure V: Control Gains Used

The control gains were tuned as per in project 1. It was decided to forgo the integral gain for the angular velocity as the target angle would constantly be changing as the robot is traversing the maze. Including an integral control would result in delays in error correction and inducing excessive overshoot, resulting in instability of the locomotion (and crashing into the wall).

Performance Enhancement

There are many avenues for the improvement of both the path planning algorithm and the locomotion algorithm. Changes to the locomotion algorithm and path planning algorithm were implemented and will be discussed in detail while other improvements to the path planning algorithm were not carried out and will only be briefly mentioned within this report.

(1) Locomotion: Heading Correction Priority

Unnecessary additional movements were noticed when making large heading corrections. A condition to slow down the linear velocity, by a factor of 4, while the robot is performing large heading corrections, of above 30 degrees in either direction, was added following the velocity saturation code. It was decided not to stop the robot (i.e., set the linear velocity to 0) to minimize the possibility of skidding which would induce overshooting as well. This was found to result in smoother negotiation of sharp corners, improving the overall locomotion of the robot as it traverses the maze.

```
if (error_heading < -M_PI/6 || error_heading > M_PI/6)
  vel_x /= 4;
```

Figure VI: Code for Heading Correction Priority

(2) Path Planning: Enabling Diagonal Path Generation

Changes were made to differentiate positions where walls are known to be (or not), and areas which have yet to be explored. This gives the robot a better understanding of the positions of walls in its surroundings. This does not affect navigation along the cardinal directions as position of walls would be updated as the robot enters the cell such that there would be no unknown walls from the current cell to the neighboring cells, but its effect on diagonal movement will be discussed shortly.

```
struct Cell
{
  // -1 for unknown, 0 for no wall, 1 for wall
  int walls[4] = {-1, -1, -1, -1};
};
```

```
bool north_wall = cur_cell->walls[0] == 1;
bool west_wall = cur_cell->walls[1] == 1;
bool south_wall = cur_cell->walls[2] == 1;
bool east_wall = cur_cell->walls[3] == 1;
```

Figure VII, VIII: Refactored Wall Detection Code

With the above changes, diagonal paths became a possibility for the robot, implemented as shown below. The need to differentiate known walls from unknown ones stems from the necessity of prior understanding of the position of walls surrounding the destination node. This means that the robot will only travel diagonally when traversing a path which it has already explored (i.e., backtracking). Cost of diagonal path was changed to reflect the actual distance travelled.

```
// === (e) check NORTHWEST neighbor ===
if (!north_wall && !west_wall && cur_x < MAP_MAX_X - 1 && cur_y < MAP_MAX_Y - 1)
{ // north wall does not exist AND west wall does not exist AND north cell exists AND west cell exists ==> northwest cell is accessible
  Cell *nw_cell = &cells[cur_x + 1][cur_y + 1];
  if (!nw_cell->walls[2] && !nw_cell->walls[3]) {
    Node *northwest_node = &nodes[cur_x + 1][cur_y + 1];
    int new_northwest_cell_cost = cur_cost + 0.707; // 0.707 here is sqrt(2)/2, which is the cost of moving diagonally
    int old_northwest_cell_cost = northwest_node->cost;
    if (old_northwest_cell_cost > new_northwest_cell_cost)
    { // northwest cell is cheaper to get there from current cell
      // update parent and cost information for northwest cell, and queue it
      northwest_node->cost = new_northwest_cell_cost;
      northwest_node->parent = cur_node;
      queue.push_back(northwest_node);
    }
  }
}
```

Figure IX: Code for Diagonal Path

Unfortunately, with the path taken by the robot as outlined by my task requirements, there was no effect on the results since there were no available diagonal paths as outlined from the constraints outlined earlier.

A way this could have improved to allow for diagonal paths in general would be to make changes to the scanning algorithm to collect data from all 8 directions instead of just 4, but this would have posed its own set of difficulties such as being unable to identify the exact wall being detected, e.g., sensing a wall in the North-West direction could correspond to either a South Wall or East Wall for the node in that direction.

The subsequent changes are discussed briefly as possible improvements but were not implemented.

(3) Path Planning: Heuristics-Based Path Planning

A well-known solution that would improve the maze solving capabilities of the robot would be to improve the path planning algorithm, implementing A* Search as opposed to the current BFS implementation. A* search will enable the robot to efficiently traverse the maze with arriving at the goal as its priority and help find the shortest possible path to the goal.

ROS Structure

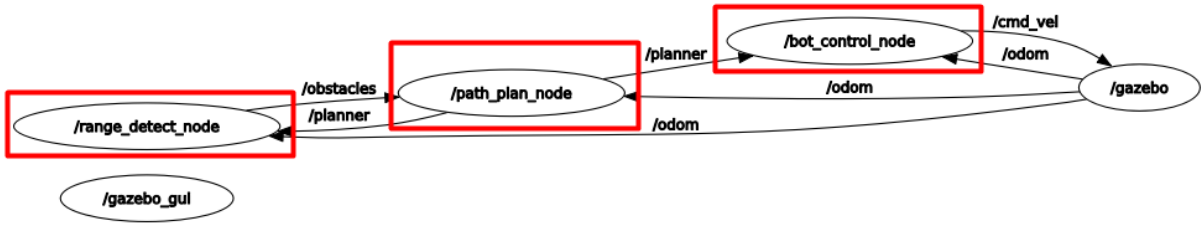

Figure X: rqt graph

The 3 main nodes are /range_detect_node, /path_plan_node and /bot_control_node, corresponding to the nodes for processing LIDAR data, path planning and generation and robot controls respectively.

The /range_detect_node node reads from the /odom topic published by /gazebo as well as the /planner topic published by the /path_plan_node node. It stores the position of walls within the maze and updates the position of walls around the node the robot is currently in, then publishes these positions on the /obstacles topic.

The /path_plan_node node reads from the /odom topic published by /gazebo as well as the /obstacles topic published by the /range_detect_node node. Based on the current robot position and maze conditions, it generates a tentative path (favouring x-axis then y-axis) towards the goal for the robot to traverse, which is published on the /planner topic.

The /bot_control_node node reads from the /odom topic published by /gazebo as well as the /planner topic published by /path_plan_node. It generates command outputs for the motors based on linear and angular velocities using a PID controller. These motor commands are then published on the /cmd_vel topic.