Path Planning

Group 17 || Aman Lohia | Teodor Mihai Tiuca | Rishab Sareen | Pavel Shering

***Abstract*—This report describes the implementation, simulation and experimental results of a ROS package which performs path planning and traversal using a TurtleBot robot.**

I. Introduction

THE lab focuses on the implementation of the Probabilistic Roadmap (PRM) algorithm. The PRM algorithm provides straight line paths between desired waypoints on a map. The other component of the lab is to traverse the planned path using a controller that navigates the Turtlebot through the environment. A static predetermined map is provided as well as some tools for visualization. Position data is available from the Indoor Positioning System (IPS). In terms of constraints, the lab allows only third party linear algebra packages, and libraries for creating random samples. The following sections describe the theory behind the algorithm, its implementation, and simulation and experimental results.

The maps used for simulation and live tests are shown in Fig. 1 and Fig. 2, respectively.

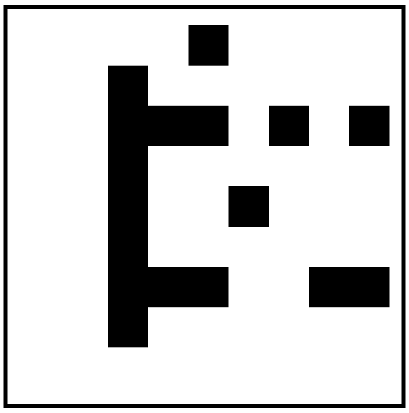
Fig. 1. Simulation map

The waypoints that the robot has to plan the path by for simulation and live are provided in Tables I and II, respectively. The waypoints also contain the desired orientation, however the minimum requirement is to hit the destinations, and the orientation is an additional challenge.

TABLE I
WAYPOINTS FOR SIMULATED PATH PLANNING IMPLEMENTATION

Waypoint #	**Position** $(x[m], y[m], \theta[rad])$
1	$(4.0, 0.0, 0.0)$
2	$(8.0, -4.0, \pi)$
3	$(8.0, 0.0, -\frac{\pi}{2})$

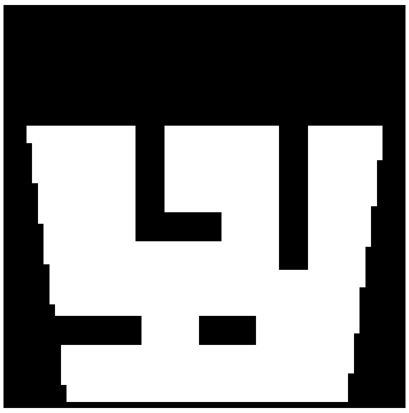
Fig. 2. Live map

TABLE II
WAYPOINTS FOR LIVE PATH PLANNING IMPLEMENTATION

Waypoint #	**Position** $(x[m], y[m], \theta[rad])$
1	$(5.0, 5.0, 0.0)$
2	$(9.0, 1.0, \pi)$
3	$(9.0, 5.0, -\pi)$

II. Theory

Probabilistic Roadmap: A PRM builds a probabilistic roadmap in the free configuration space of the robot by generating and interconnecting a large finite number of configurations of the robot. The random configuration or nodes can be randomized uniformly or with a weighted bias. The simplest is to use uniform random sampling (equal probability of any x-y coordinate). Uniform random sampling is used for this lab because the movable areas are large and open, and enough nodes can be generated to successfully path plan. Other sampling techniques are available, such as Latombe or Lavelle (which bias samples near walls), or Bridge Checking (which biases samples in the centre of thin hallways). Once the nodes are created, all are checked for collision with objects in the map and removed if colliding. Additionally, nodes up to half the robot width away from the objects are also not to be considered as these are locations that the robot cannot exist due to overlap with the object.

Once the nodes are sampled, a finite number of connections are created to other nearby nodes. The number of nearby nodes connected is a tunable parameter and the connections represent the possible motion paths. The Bresenham line algorithm is used to find the connections in collision with objects in the map. Once the connections in collision have been removed from the PRM, then only a map of possible connection paths will remain on the PRM graph.

Shortest Path Search: Once the PRM graph is generated, a shortest path algorithm can be run directly on the graph. Commonly used techniques introduced during the course are depth-first search, Wavefront (reverse breadth-first search), Dijkstra's algorithm, A*, or potential fields. Dijkstra's is an excellent compromise between performance and simplicity, and always provides the optimal path in a graph. However, the A* algorithm is similar implementation of the Dijkstra's algorithm with a heuristic bias. This allows to pick the "most promising" node at each step. Thus, A* is a much quicker path search process compared to Dijkstra's algorithm but might not be the shortest path, however, A* provides a very good approximation of the shortest path. Due to the significant reduction in computation required in implementation the A* path search algorithm is used in this Lab.

Controller: The controller designed for this Lab is a very simple controller that iteratively drives forward or corrects the heading of the robot. It is iterative in the sense that the controller constantly switches between heading correction and forward driving, and does not conduct both simultaneously. A P controller is used for both aligning robot heading and setting forward velocity.

III. Implementation

The code for both components of the lab is implemented inside a single ROS package and is detailed below.

Path Planning: The PRM algorithm should provide all necessary connections for traversing the map in one run through. Due to this, the PRM generation code only runs once every time the map callback function is triggered. This is implemented through flipping a Boolean in the map callback which gets checked in the main ROS loop.

The robot is assumed to have to traverse the waypoints in the order given, so no optimization was done through reordering waypoints.

Creating the PRM uses the C++ $rand()$ function to randomize X and Y coordinates of points. As a result, the numbers generated are pseudo-random because the code uses the same seed every run. The values used to generate the PRM could be made more random by running the $rand()$ function a certain number of times before beginning to generate the PRM. This can be changed manually every run through, therefore introducing a random seed. This was not done because it introduced predictability into the system. It is preferred for the points to be in the same place every run, rather than being truly random. This ensures that there are no edge cases hit while restarting. One such edge case would be having sections of the PRM graph not connected to each other. Although this would need to be checked for in a real system, it was ignored in this implementation due to ease of debugging.

To store the PRM, a C++ vector was used instead of a basic array. Storing the information as a vector allows for automatic re-sizing and the use of $vector.push\_back()$ and $vector.pop\_back()$ functions when inserting or removing points. There is also a $vector.erase()$ function that is used to remove points in the middle of the vector, saving a lot of looping and restricting which would happen in an array.

Built-in C++ function are used where possible to simplify code and improve efficiency. For example, when sorting the queue for A* path planning, the C++ $std :: sort()$ function is used with a custom comparison operator. Thus, a comparison operator for the total distance is the only piece of code that needs to be made, and the actual sorting is taken care of.

For the purpose of this Lab, a total 500 nodes and 20 connection per node is implemented. However, it must be considered that several nodes were removed due to overlapping with objects on the map.

Controller: The controller is implemented as a simple proportional controller with separate linear and angular gain values. The error in the heading controller is simply the angle difference between the robot's heading and where the robot's heading should be if it is to face the next node in the path. The error in the driving controller is simply the straight line distance to the next node. Both these error values are multiplied by constant gain values that are tuned through testing of the code in the simulation and live setups. This results in the final control values that are applied to the angular velocity about the z axis and linear x velocity components of the $geometry\_msgs :: Twist$ object that is published for the robot to listen to.

The decision to set the robot forward velocity to zero when correcting heading is done to simplify the operation of the controller. Allowing the forward velocity to continue unimpeded while correcting for heading causes the robot to travel in arcs and vastly increases the overshoot as well as decreases the stability of the designed controller increasing the possibility of a collision.

IV. Results and Discussion

The results of the implemented algorithms are discussed below based of simulation and live experiments.

Simulation Path Planning: Fig. 3 demonstrates the PRM creating the graph of possible paths (in green) and the A* algorithm selecting the shortest one (in white) to follow the waypoints given in Tab. I. It is assumed that the order of the waypoints must be exact. If the objective changed is for shortest time and shortest path traversal with visiting every waypoint the A* algorithm changes to sort the waypoints by distance to the original position before the original implementation stated in Sec. III.

Simulation Path Tracking: Fig. 4 demonstrates the TurtleBots navigation path in blue using the p-controller to reach the waypoints given in Tab. I in the order specified. The TurtleBot follows the path well with marginal error in the position seen. This error is due to the simulated robot in Gazebo colliding with the obstacles, even though the appropriate padding (robot diameter) is in effect. Thus, the behaviour of the robot is to keep trying to go forward while colliding and correcting heading after every collision. This can be fixed by eliminating

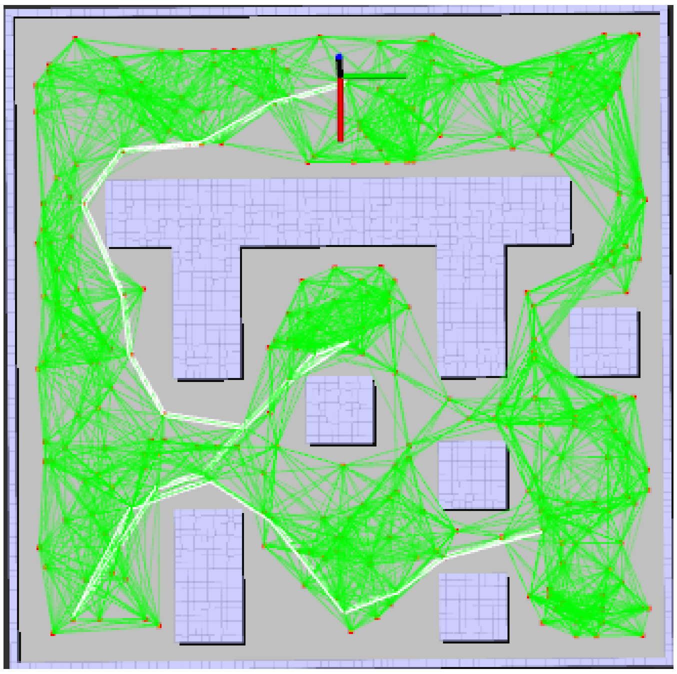

Fig. 3. Simulation of the planned path by the PRM and A* algorithms

the path edges that are in the padded space, and the easiest method to accomplish this is to pad the occupancy grid with the appropriate border (robot diameter). Additionally, the robot's traversal velocity can be increased by implementing a better controller. As mentioned before the P - controller has only an angular or linear velocity at any point it time, thus, creating a lot of stop periods to adjust the heading between path nodes.

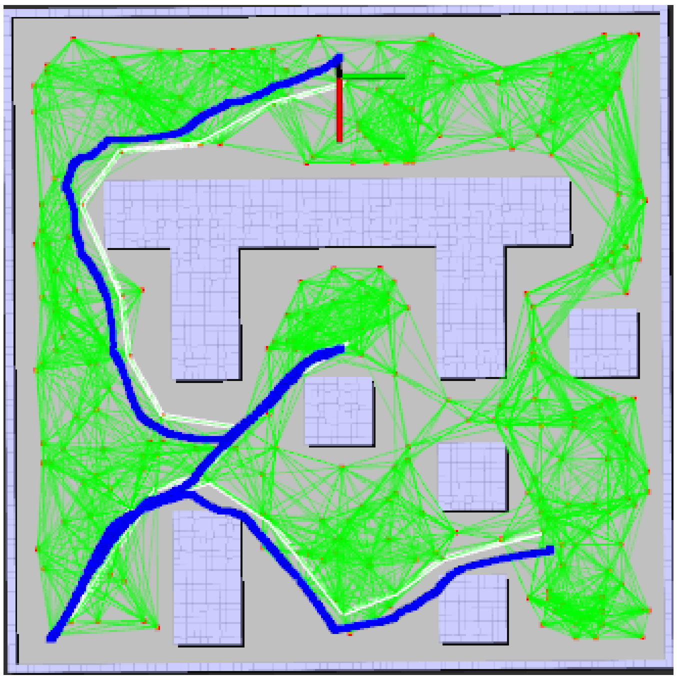

Fig. 4. Simulation Turtlebot navigating the planned path using the designed P-controller

A controller that can handle both linear and angular velocity changes allows for faster course completion. A carrot controller can be implemented to perform such task. This controller works by defining a "carrot" on a desired trajectory line (in this case along the shortest planned path made up for multiple nodes) with a fixed distance, ***R***, in front of the TurtleBot. The cross-track error as displayed in Fig. 5 is constantly minimized as the "carrot" travels along the planned path, with non zero linear speed. The ***R*** distance is chosen based on the characteristics that the TurtleBot requires. Creating a small R allows for no overshoot during turns however the max inside turning radius must be calculated to avoid collision with the objects (although that is unnecessary for appropriately padded maps). The carrot controller is an addition to the P-controller already implemented.

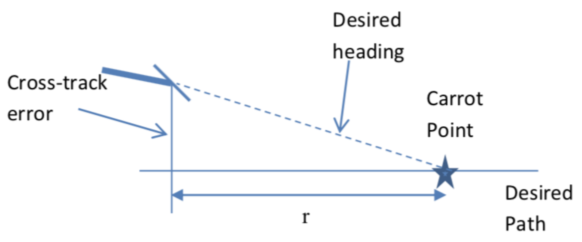

Fig. 5. Carrot controller implementation

Live Path Planning: The same PRM alogorithm is run for the live map, Fig. 6 demonstrates the PRM creating the graph of possible paths (in green) and the A* algorithm selecting the shortest one (in white) to follow the waypoints given in Tab. II.

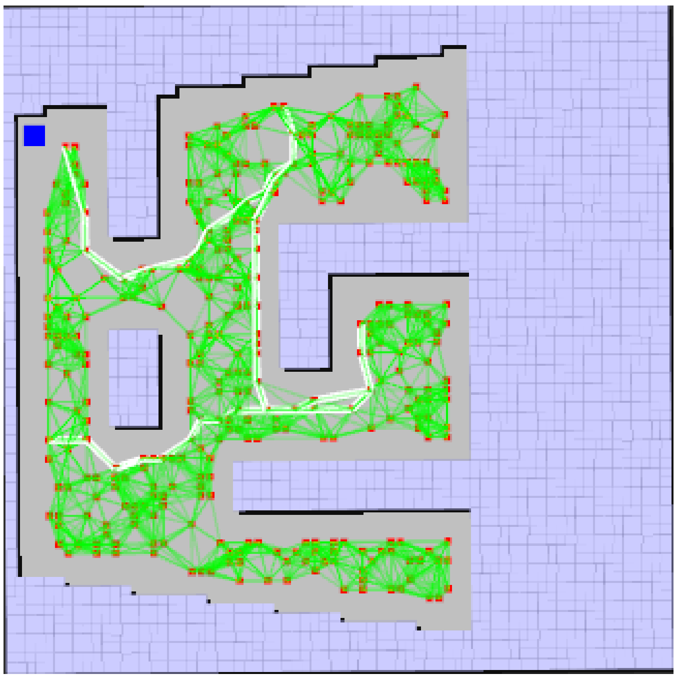

Fig. 6. Live planned path for the TurtleBot to navigate

Live Path Tracking: Due to unresolved issues with correctly interpreting and transforming the live pose data obtained from the indoor position system, the live path tracking results were poor and had an unknown angular offset between the IPS, map and baselink origins. For demonstration purposes, gazebo was used as a stand in, with the algorithm running on the live map provided. Fig. 7 shows the results of the same tracking algorithm running on the live path, with the robot's path shown in blue. Due to a better tuned padding size, the robot's path no longer cuts corners and does not collide with the walls.

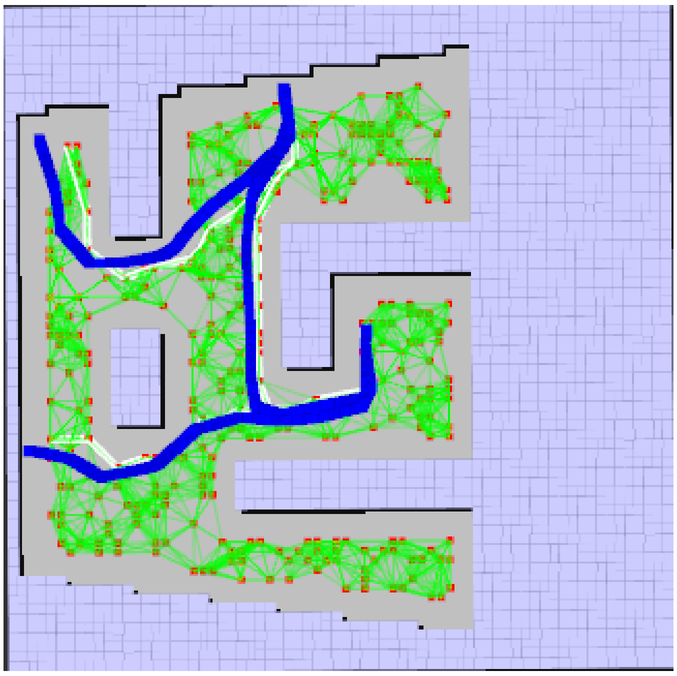

Fig. 7. Live Turtlebot navigating the planned path using the designed P-controller

Conclusion: Overall, the designed algorithms performed as expected and are successful in demonstrating a path planning and execution proof of concept. There are unresolved issues encountered with the live setup, but these are due to pose data interpretation and not an issue with the core algorithms developed and implemented.

V. SOURCE CODE

Listing 1. Path Planning and Traversal Controller Implementation

```cpp
// ///////////////////////////////////////////////////////////
//
// turtlebot_example.cpp
// This file contains example code for use with ME 597 lab 2
// It outlines the basic setup of a ros node and the various
// inputs and outputs needed for this lab
//
// Author: James Servos
// Edited: Nima Mohajerin
//
// //////////////////////////////////////////////////////////
#include <ros/ros.h>
#include <geometry_msgs/PoseStamped.h>
#include <geometry_msgs/Twist.h>
#include <tf/transform_datatypes.h>
#include <tf/transform_broadcaster.h>
#include <gazebo_msgs/ModelStates.h>
#include <visualization_msgs/Marker.h>
#include <nav_msgs/OccupancyGrid.h>
#include <vector>
#include <cmath>

#define WAYPOINT_THRESHOLD    0.1
#define THETA_THRESHOLD    0.15
#define P_CONTROL_LIN    0.75
#define P_CONTROL_ANG    0.25
#define MAP_DIM    10.0
#define MAP_RES    0.1
#define X_OFFSET   -1.0
#define Y_OFFSET   -5.0

const int MAP_SIZE = 100;
const int ROBOT_WIDTH = 3;
// const int NUM_POINTS = 1400;
const int NUM_POINTS = 500;
const int NUM_CONNECTIONS = 20;

using namespace std;
vector<int> shortestPath;

float line_segment_x = 0;
float line_segment_y = 0;
float line_segment_x_final = 0;
float line_segment_y_final = 0;
float dist_error = 0;
int current_index = 1;

int collisionMap[MAP_SIZE * MAP_SIZE];

bool mapReady = false;

struct Point {
  int x = 0;
  int y = 0;
  vector<int> nearestNeighbours;
  bool visited = false;
  int parentNodeIdex;
};

struct Neighbour {
  int pointIndex;
  double dist;
};

// ros::Publisher pose_publisher;
ros::Publisher marker_pub;

double ips_x;
double ips_y;
double ips_yaw;

short sgn(int x) { return x >= 0 ? 1 : -1; }

double random_gen() {
  return (double)(rand() % 10000) / 10000.0;
}

//Callback function for the Position topic (SIMULATION)
void pose_callback(const gazebo_msgs::ModelStates& msg) {
  int i;
  for(i = 0; i < msg.name.size(); i++) if(msg.name[i] == "mobile_base")
      ↪ break;

  ips_x = msg.pose[i].position.x - X_OFFSET;
  ips_y = msg.pose[i].position.y -   Y_OFFSET;
  ips_yaw = tf::getYaw(msg.pose[i].orientation);
  ROS_DEBUG("pose_callback␣X:␣%f␣Y:␣%f␣Yaw:␣%f", ips_x, ips_y, ips_yaw);

  //Create tf broadcaster
  ROS_DEBUG("sending␣broadcaster");
  static tf::TransformBroadcaster broadcaster_map;
  tf::Transform transform_map;
  transform_map.setOrigin( tf::Vector3(ips_x, ips_y, 0.0) );
  tf::Quaternion q;
  q.setRPY(0,0,ips_yaw);
  transform_map.setRotation(q);
  broadcaster_map.sendTransform(tf::StampedTransform(transform_map, ros::
        ↪ Time::now(), "base_link", "map"));

    //Create tf broadcaster
  ROS_DEBUG("sending␣broadcaster");
  static tf::TransformBroadcaster broadcaster_vis;
  tf::Transform transform_vis;
  transform_vis.setOrigin( tf::Vector3(ips_x, ips_y, 0.0) );
  tf::Quaternion w;
  w.setRPY(0,0,ips_yaw);
  transform_vis.setRotation(w);
  broadcaster_map.sendTransform(tf::StampedTransform(transform_vis, ros::
        ↪ Time::now(), "base_link", "visualization_marker"));
}

//Callback function for the Position topic (LIVE)
/*
void pose_callback(const geometry_msgs::PoseWithCovarianceStamped& msg)
{
  ips_x X = msg.pose.pose.position.x; // Robot X psotition
  ips_y Y = msg.pose.pose.position.y; // Robot Y psotition
  ips_yaw = tf::getYaw(msg.pose.pose.orientation); // Robot Yaw
  ROS_DEBUG("pose_callback X: %f Y: %f Yaw: %f", X, Y, Yaw);
}*/


//Callback function for the map
void map_callback(const nav_msgs::OccupancyGrid& msg) {
  ROS_INFO("Map␣Width␣:␣%d", msg.info.width);
  ROS_INFO("Map␣Height␣:␣%d", msg.info.height);

  for(int i = 0; i < MAP_SIZE * MAP_SIZE; i++) {
    collisionMap[i] = (int) msg.data[i];
  }

  ROS_INFO("Map␣Ready␣...");
  mapReady = true;
}

//Bresenham line algorithm (pass empty vectors)
// Usage: (x0, y0) is the first point and (x1, y1) is the second point.
      ↪ The calculated
//        points (x, y) are stored in the x and y vector. x and y should
      ↪ be empty
//    vectors of integers and shold be defined where this function is
      ↪ called from.
void bresenham(int x0, int y0, int x1, int y1, vector<int>& x, vector<int
      ↪ >& y) {

    int dx = abs(x1 - x0);
    int dy = abs(y1 - y0);
    int dx2 = x1 - x0;
    int dy2 = y1 - y0;

    const bool s = abs(dy) > abs(dx);

    if (s) {
        int dx2 = dx;
        dx = dy;
        dy = dx2;
    }

    int inc1 = 2 * dy;
    int d = inc1 - dx;
    int inc2 = d - dx;

    x.push_back(x0);
    y.push_back(y0);

    while (x0 != x1 || y0 != y1) {
        if (s) y0+=sgn(dy2); else x0+=sgn(dx2);
        if (d < 0) d += inc1;
        else {
            d += inc2;
            if (s) x0+=sgn(dx2); else y0+=sgn(dy2);
        }

        //Add point to vector
        x.push_back(x0);
        y.push_back(y0);
    }
}

/***************************************
PRM
***************************************/

vector<Point> pointMap;

void generateRandomMap() {
  for(int i = 0; i < NUM_POINTS; i++) {
    Point newPoint;
    newPoint.x = (int) (random_gen() * (MAP_SIZE - 2) + 1);
    newPoint.y = (int) (random_gen() * (MAP_SIZE - 2) + 1);

    pointMap.push_back(newPoint);
  }
}

bool inCollision(int index) {
  int min_x = index % 100 - ROBOT_WIDTH;
  int max_x = index % 100 + ROBOT_WIDTH;

  int min_y = (int)(index / 100) - ROBOT_WIDTH;
  int max_y = (int)(index / 100) + ROBOT_WIDTH;

  if(min_x < 0 || max_x > MAP_SIZE) { return true; }
  if(min_y < 0 || max_y > MAP_SIZE) { return true; }

  for(int x = min_x; x < max_x; x++) {
    for(int y = min_y; y < max_y; y++) {
      if(collisionMap[y*100 + x] > 0) {
        return true;
      }
    }
  }

  return false;
}

void removeCollisionPoints() {
  ROS_INFO("Number␣of␣points␣:␣%d", (int) pointMap.size());
  ROS_INFO("Removing␣points␣in␣collision....");

  vector<Point>::iterator p = pointMap.begin();

  while(p != pointMap.end()) {
    int index = p->y * 100 + p->x;

    if(inCollision(index)) {
      p = pointMap.erase(p);
    } else {
      p++;
    }
  }

  ROS_INFO("Final␣number␣of␣points␣:␣%d\n", (int) pointMap.size());
}

bool compareByDistance(const Neighbour &a, const Neighbour &b) {
  return a.dist < b.dist;
}

void connectNearestNeighbours() {
  for(int pIndex = 0; pIndex < pointMap.size(); pIndex++) {
    vector<Neighbour> allNeighbours;

    for(int nIndex = 0; nIndex < pointMap.size(); nIndex++) {
      if(nIndex == pIndex) { continue; }
      Neighbour n;
      n.pointIndex = nIndex;
      n.dist = sqrt(pow(pointMap[nIndex].x - pointMap[pIndex].x, 2) + pow(
          ↪ pointMap[nIndex].y - pointMap[pIndex].y, 2));
      allNeighbours.push_back(n);
    }

    sort(allNeighbours.begin(), allNeighbours.end(), compareByDistance);
    int iterations = NUM_CONNECTIONS;
    for(int i = 0; i < iterations; i++) {
      if(allNeighbours[i].dist > 1) {
        pointMap[pIndex].nearestNeighbours.push_back(allNeighbours[i].
            ↪ pointIndex);
      } else {
        iterations++;
      }
    }
  }
}

void removeCollisionPaths() {
  for(auto &point:pointMap) {
    vector<int>::iterator neighbour = point.nearestNeighbours.begin();

    while(neighbour != point.nearestNeighbours.end()) {
      vector<int> x;
      vector<int> y;

      bresenham(point.x, point.y, pointMap[*neighbour].x, pointMap[*
          ↪ neighbour].y, x, y);

      bool pathCollides = false;
      for(int i = 0; i < x.size(); i++) {
        if(collisionMap[MAP_SIZE*y[i] + x[i]] > 0) {
          pathCollides = true;
        }
      }

      if(pathCollides) {
        neighbour = point.nearestNeighbours.erase(neighbour);
      } else {
        neighbour++;
      }
    }
  }

}

/***************************************
A * Path Search
***************************************/

int findClosestPoint(int x, int y) {
  vector<Neighbour> allNeighbours;

  for(int nIndex = 0; nIndex < pointMap.size(); nIndex++) {
    Neighbour n;
    n.pointIndex = nIndex;
    n.dist = sqrt(pow(pointMap[nIndex].x - x, 2) + pow(pointMap[nIndex].y
        ↪ - y, 2));
    allNeighbours.push_back(n);
  }

  sort(allNeighbours.begin(), allNeighbours.end(), compareByDistance);

  return allNeighbours[0].pointIndex;
}

void prepareForTraversal() {
  for(auto & point:pointMap) {
    point.visited = false;
    point.parentNodeIdex = -2;
  }
}

struct PathPoint {
  int x;
  int y;
  int index;
  double cost;
  double heuristicCost;
};

void pushOntoStack(vector<PathPoint> * stack, int nodeIndex, double cost,
    ↪ int goalNode) {
  PathPoint p;

  p.x = pointMap[nodeIndex].x;
  p.y = pointMap[nodeIndex].y;
  p.index = nodeIndex;
  p.cost = cost;
  p.heuristicCost = sqrt(pow(p.x - pointMap[goalNode].x, 2) + pow(p.y -
      ↪ pointMap[goalNode].y, 2));

  pointMap[nodeIndex].visited = true;

  stack->push_back(p);
}

bool compareByFullCost(const PathPoint &a, const PathPoint &b) {
  return (a.cost + a.heuristicCost) > (b.cost + b.heuristicCost);
}

vector<int> getShortestPath(int startNode, int goalNode) {
  prepareForTraversal();

  vector<int> shortestPath;
  vector<PathPoint> stack;

  pointMap[startNode].parentNodeIdex = -1;
  pushOntoStack(&stack, startNode, 0.0f, goalNode);

  while(!stack.empty()) {
    sort(stack.begin(), stack.end(), compareByFullCost);
    PathPoint currentNode = stack.back();
    stack.pop_back();

    if(currentNode.index == goalNode) {
      shortestPath.push_back(currentNode.index);
      int parent = pointMap[currentNode.index].parentNodeIdex;
      do {
        shortestPath.push_back(parent);
        parent = pointMap[parent].parentNodeIdex;
      } while (parent != -1);
      return shortestPath;
    }

    for(auto & neighbour:pointMap[currentNode.index].nearestNeighbours) {
      if(!pointMap[neighbour].visited) {
        pointMap[neighbour].parentNodeIdex = currentNode.index;
        double interNodeCost = sqrt(pow(currentNode.x - pointMap[neighbour
            ↪ ].x, 2) + pow(currentNode.y - pointMap[neighbour].y, 2));
        pushOntoStack(&stack, neighbour, currentNode.cost + interNodeCost,
            ↪  goalNode);
      }
    }

  }

  ROS_INFO("Exited␣*****");
  shortestPath.clear();
  return shortestPath;
}

bool checkCollisionPaths(vector<int> shortestPathLocal, int p1, int p2) {
  vector<int> x;
  vector<int> y;
  ROS_DEBUG("HERE␣Check␣collision");


  bresenham(pointMap[shortestPathLocal[p1]].x, pointMap[shortestPathLocal[
      ↪ p1]].y, pointMap[shortestPathLocal[p2]].x, pointMap[
      ↪ shortestPathLocal[p2]].y, x, y);
  bool pathCollides = false;

  for(int i = 0; i < x.size(); i++) {
    if(collisionMap[MAP_SIZE*y[i] + x[i]] > 0) {
      pathCollides = true;
      ROS_DEBUG("COLLIDED");
    }
  }

  return pathCollides;
}

vector<int> optimizePath(vector<int> shortestPathLocal) {
  vector<int> optimizedPath;
  optimizedPath.push_back(shortestPathLocal.back());
  ROS_DEBUG("HERE␣optimize␣paths");
  int i = shortestPathLocal.size()-1;
  while (i != 0 ) {
    for (int j = 0; j < i; j++) {
      ROS_DEBUG("hello");
      if (!checkCollisionPaths(shortestPathLocal, i, j)) {
        optimizedPath.push_back(shortestPathLocal[j]);
        ROS_DEBUG("i␣=␣%d,␣j␣=␣%d", i , j);
        i = j;
        break;
      }
    }
  }

  return optimizedPath;
}


/****************************************
Robot Movement Controller
****************************************/

float theta_error(float line_segment_y, float line_segment_x) {
  float theta_ref = atan2(line_segment_y, line_segment_x);
  float theta_error = theta_ref - ips_yaw;

  if(theta_error > M_PI){
    theta_error = theta_error - 2*M_PI;
  }
  else if (theta_error < -M_PI) {
    theta_error = theta_error + 2*M_PI;
  }
  return theta_error;
}

int main(int argc, char **argv)
{
    //Initialize the ROS framework
    ros::init(argc,argv,"main_control");
    if( ros::console::set_logger_level(ROSCONSOLE_DEFAULT_NAME, ros::
          ↪ console::levels::Debug) ) {
        ros::console::notifyLoggerLevelsChanged();
    }
    ros::NodeHandle n;

    //Subscribe to the desired topics and assign callbacks
    // ros::Subscriber pose_sub = n.subscribe("/indoor_pos", 1,
          ↪ pose_callback);
    ros::Subscriber pose_sub = n.subscribe("/gazebo/model_states", 1,
          ↪ pose_callback);
    ros::Subscriber map_sub = n.subscribe("/map", 1, map_callback);

    //Setup topics to Publish from this node
    ros::Publisher velocity_publisher = n.advertise<geometry_msgs::Twist>(
          ↪ "/cmd_vel_mux/input/navi", 1);
    // pose_publisher = n.advertise<geometry_msgs::PoseStamped>("/pose",
          ↪ 1, true);
    marker_pub = n.advertise<visualization_msgs::Marker>("
          ↪ visualization_marker", 1, true);

    //Velocity control variable
    geometry_msgs::Twist vel;

    //Set the loop rate
    ros::Rate loop_rate(20);    //20Hz update rate

    visualization_msgs::Marker points;
    points.header.frame_id = "/map";
    points.header.stamp = ros::Time::now();
    points.ns = "points";
    points.action = visualization_msgs::Marker::ADD;
    points.type = visualization_msgs::Marker::POINTS;
    points.id = 1;
    points.points.clear();

    points.scale.x = 0.08f;
    points.scale.y = 0.08f;

    points.color.r = 1.0f;
    points.color.a = 1.0f;

    visualization_msgs::Marker collisionPoints;
    collisionPoints.header.frame_id = "/map";
    collisionPoints.header.stamp = ros::Time::now();
    collisionPoints.ns = "collisionPoints";
    collisionPoints.action = visualization_msgs::Marker::ADD;
    collisionPoints.type = visualization_msgs::Marker::POINTS;
    collisionPoints.id = 2;
    collisionPoints.points.clear();

    collisionPoints.scale.x = 0.12f;
    collisionPoints.scale.y = 0.12f;

    collisionPoints.color.r = 0.8f;
    collisionPoints.color.g = 0.8f;
    collisionPoints.color.b = 1.0f;
    collisionPoints.color.a = 1.0f;

    visualization_msgs::Marker waypointMarkers;
    waypointMarkers.header.frame_id = "/map";
    waypointMarkers.header.stamp = ros::Time::now();
    waypointMarkers.ns = "waypointMarkers";
    waypointMarkers.action = visualization_msgs::Marker::ADD;
    waypointMarkers.type = visualization_msgs::Marker::POINTS;
    waypointMarkers.id = 3;
    waypointMarkers.points.clear();

    waypointMarkers.scale.x = 0.18f;
    waypointMarkers.scale.y = 0.18f;

    waypointMarkers.color.r = 0.0f;
    waypointMarkers.color.g = 1.0f;
    waypointMarkers.color.b = 0.0f;
    waypointMarkers.color.a = 1.0f;

    visualization_msgs::Marker allPaths;
    allPaths.header.frame_id = "/map";
    allPaths.header.stamp = ros::Time::now();
    allPaths.ns = "allPaths";
    allPaths.action = visualization_msgs::Marker::ADD;
    allPaths.type = visualization_msgs::Marker::LINE_LIST;
    allPaths.id = 4;
    allPaths.points.clear();

    allPaths.scale.x = 0.03f;
    allPaths.scale.y = 0.03f;

    allPaths.color.r = 0.0f;
    allPaths.color.g = 1.0f;
    allPaths.color.b = 0.0f;
    allPaths.color.a = 0.1f;

    visualization_msgs::Marker traversedPaths;
    traversedPaths.header.frame_id = "/map";
    traversedPaths.header.stamp = ros::Time::now();
    traversedPaths.ns = "traversedPaths";
    traversedPaths.action = visualization_msgs::Marker::ADD;
    traversedPaths.type = visualization_msgs::Marker::LINE_LIST;
    traversedPaths.id = 5;
  traversedPaths.scale.x = 0.08f;
  traversedPaths.scale.y = 0.08f;

  traversedPaths.color.r = 1.0f;
  traversedPaths.color.g = 1.0f;
  traversedPaths.color.b = 1.0f;
  traversedPaths.color.a = 1.0f;


  ROS_DEBUG("pose_callback␣X:␣%f␣Y:␣%f␣Yaw:␣%f", ips_x, ips_y, ips_yaw);

  while (ros::ok())
  {
    loop_rate.sleep(); //Maintain the loop rate
    ros::spinOnce();   //Check for new messages
    static double waypoints[16] = {ips_x, ips_y, ips_yaw,  5.0, 5.0,
        ↪ 0.0,   9.0, 1.0, 3.14,    9.0, 5.0, -3.14,    8.0, 8.0,
        ↪ 3.14};

    if(mapReady) {
      generateRandomMap();
      removeCollisionPoints();
      connectNearestNeighbours();
      removeCollisionPaths();

      vector<int> waypointNodes;
      for(int i = 0; i < 16; i += 3) {
        int index = findClosestPoint((int) (waypoints[i] * 10), (int) (
            ↪ waypoints[i+1] * 10));
        waypointNodes.push_back(index);
      }


      for(int i = 0; i < waypointNodes.size()-2; i++) {
        vector<int> shortestPathLocal = getShortestPath(waypointNodes[i
            ↪ ], waypointNodes[i+1]);

        shortestPath.insert(shortestPath.begin(), shortestPathLocal.
            ↪ begin(), shortestPathLocal.end());

        if(shortestPath.empty()) {
          mapReady = false;
          ROS_INFO("Path␣not␣found");
          continue;
        }
        traversedPaths.points.clear();

        for(int j = 0; j < shortestPath.size() - 1; j++) {
          geometry_msgs::Point p1;
          geometry_msgs::Point p2;

          p1.x = (pointMap[shortestPath[j]].x)*MAP_RES + X_OFFSET;
          p1.y = (pointMap[shortestPath[j]].y)*MAP_RES + Y_OFFSET;
          traversedPaths.points.push_back(p1);
          p2.x = (pointMap[shortestPath[j+1]].x)*MAP_RES + X_OFFSET;
          p2.y = (pointMap[shortestPath[j+1]].y)*MAP_RES + Y_OFFSET;
          traversedPaths.points.push_back(p2);
        }
      }

      for(auto &point:pointMap) {
        geometry_msgs::Point p1;
        p1.x = (point.x)*MAP_RES + X_OFFSET;
        p1.y = (point.y)*MAP_RES + Y_OFFSET;
        for(auto &neighbour:point.nearestNeighbours) {
          geometry_msgs::Point p2;
          p2.x = (pointMap[neighbour].x)*MAP_RES + X_OFFSET;
          p2.y = (pointMap[neighbour].y)*MAP_RES + Y_OFFSET;
          allPaths.points.push_back(p1);
          allPaths.points.push_back(p2);
        }
      }

      mapReady = false;

      vector<geometry_msgs::Point> displayPointsVec;
      for(auto &point:pointMap) {
        geometry_msgs::Point p;
        p.x = (point.x)*MAP_RES + X_OFFSET;
        p.y = (point.y)*MAP_RES + Y_OFFSET;
        displayPointsVec.push_back(p);
      }
      points.points = displayPointsVec;

      vector<geometry_msgs::Point> displayCollisionPointsVec;
      for(int i =0; i < MAP_SIZE * MAP_SIZE; i++) {
        if(collisionMap[i] > 0) {
          geometry_msgs::Point p;
          p.x = (i%MAP_SIZE)*MAP_RES + X_OFFSET;
          p.y = ((int) (i /MAP_SIZE))*MAP_RES + Y_OFFSET;
          displayCollisionPointsVec.push_back(p);
        }
      }
      collisionPoints.points = displayCollisionPointsVec;

      reverse(shortestPath.begin(), shortestPath.end());
    }

    marker_pub.publish(allPaths);
    marker_pub.publish(traversedPaths);
    marker_pub.publish(points);
    marker_pub.publish(collisionPoints);
    marker_pub.publish(waypointMarkers);

    line_segment_x = pointMap[shortestPath[current_index]].x * MAP_RES -
        ↪ ips_x;
    line_segment_y = pointMap[shortestPath[current_index]].y * MAP_RES -
        ↪ ips_y;

    line_segment_x_final = pointMap[shortestPath[shortestPath.size()
        ↪ -1]].x * MAP_RES - ips_x;
    line_segment_y_final = pointMap[shortestPath[shortestPath.size()
        ↪ -1]].y * MAP_RES - ips_y;

    if ((abs(line_segment_x_final) <= WAYPOINT_THRESHOLD && abs(
        ↪ line_segment_y_final) <= WAYPOINT_THRESHOLD)) {
      ROS_DEBUG("DONE");
      vel.linear.x = 0.0;
      vel.angular.z = 0.0;
    }
    else {
      ROS_DEBUG("theta␣error:␣%f", theta_error(line_segment_y,
          ↪ line_segment_x));
      if (abs(theta_error(line_segment_y, line_segment_x)) <=
          ↪ THETA_THRESHOLD){
        vel.angular.z = 0.0;
        if (abs(line_segment_x) < WAYPOINT_THRESHOLD && abs(
            ↪ line_segment_y) < WAYPOINT_THRESHOLD) {
          float X = pointMap[shortestPath[current_index]].x * MAP_RES;
          float Y = pointMap[shortestPath[current_index]].y * MAP_RES;
          vel.linear.x = 0.0;
          current_index++;
        } else {
          float X = pointMap[shortestPath[current_index]].x * MAP_RES;
          float Y = pointMap[shortestPath[current_index]].y * MAP_RES;
          float dist_error = sqrt(line_segment_x*line_segment_x +
              ↪ line_segment_y*line_segment_y);
          vel.linear.x = P_CONTROL_LIN*dist_error;
        }
      } else {
        vel.linear.x = 0.0;
        vel.angular.z = P_CONTROL_ANG*theta_error(line_segment_y,
            ↪ line_segment_x);
      }
    }
    velocity_publisher.publish(vel); // Publish the command velocity
  }

  return 0;
}
```