Autonomous Robotics Project

Mohammad Rami Koujan and Yogesh Langhe
VIBOT
University of Girona

***Abstract*— This project report summarizes the work done by our group for the Autonomous Robotics project. It presents the implementation and test of two high-level controllers in terms of path planning algorithms (Tangent Bug and Rapidly Exploring Random Trees) for controlling a mobile robot in an environment with obstacles. The platform and robot software used in the implementation are TurtleBot and ROS (Robot Operating System) respectively. Furthermore, all the tests are conducted in Stage robot simulator.**

I. Introduction

Path planning or find-path problem is well known in robotics and it plays an important role in the navigation of autonomous mobile robots (artificial intelligent mobile robots). Navigation which is a process or activity to plan and direct a route or path is a task that an autonomous robot must do correctly in order to move safely from one location to another location without getting lost or colliding with other objects. The three general problems of navigation are localization, path planning and motion control. Between these three problems, it can be argued that path planning is one of the most important issues in the navigation process.

Accurate path planning enables autonomous mobile robots to follow or track an optimal collision free path from start position to the goal position without colliding with obstacles in the workspace area. In order to simplify the path planning problem and ensure that the robot run/move smoothly while avoiding obstacles in a cluttered environment, the configuration space must be matched with the algorithm being used. Hence the selection of algorithm for a given problem is an important issue. In computing, data structures play an important role and greatly influence the computational complexity and efficient implementations of algorithm .

In fact, two path algorithms have been selected in this project to be studied, analyzed, implemented, and tested on a TurtleBot platform. Those algorithms are Tangent Bug and Rapidly Exploring Random Trees (RRT).

II. RRT Algorithm

RRTs (LaValle and Kuffner 1999 [1]) were proposed as both a sampling algorithm and a data structure designed to allow fast searches in high-dimensional spaces in motion planning. RRTs are progressively built towards unexplored regions of the space from an initial configuration as shown in Figure 1. At every step a random q rand configuration is chosen and for that configuration the nearest configuration already belonging to the tree q near is found. For this a definition of distance is required (in motion planning, the euclidean distance is usually chosen as the distance measure). When the nearest configuration is found, a local planner tries to join q near with q rand with a limit distance. If q rand was reached, it is added to the tree and connected with an edge to q near. If q rand was not reached, then the configuration q new obtained at the end of the local search is added to the tree in the same way as long as there was no collision with an obstacle during the search. This operation is called the Extend step, illustrated in Figure 2. This process is repeated until some criteria is met, like a limit on the size of the tree.

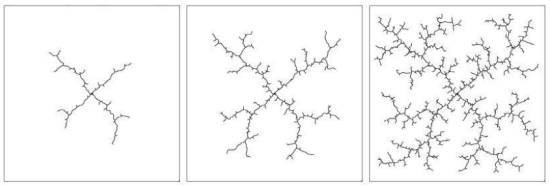


Fig. 1. Progressive construction of an RRT.

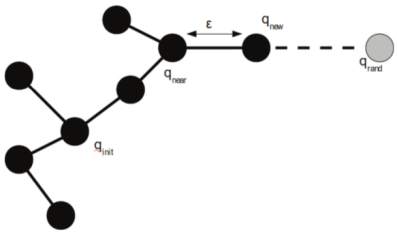


Fig. 2. Extend phase of an RRT.

Algorithm 1 gives an outline of the process.

Algorithm 1: Description of the building process of an RRT

Data: Search space S, initial configuration q_{init}, limit ϵ, ending criteria end
Result: RRT $tree$
begin
  $tree \longleftarrow q_{init}$
  while $\neg$ end **do**
    $q_{rand} \longleftarrow sampleSpace(S)$
    $q_{near} \longleftarrow findNearest(tree, q_{rand}, S)$
    $q_{new} \longleftarrow join(q_{near}, q_{rand}, \epsilon, S)$
    if $reachable(q_{new})$ **then**
      $addConfiguration(tree, q_{near}, q_{new})$
  return $tree$

Once the RRT has been built, multiples queries can be issued. For each query, the nearest configurations belonging to the tree to both the initial and the goal configuration are found. Then, the initial and final configurations are joined to the tree to those nearest configurations using the local planner and a path is retrieved by tracing back in the tree structure.

III. RRT Implementation

In this section the implemented RRT algorithm is explained in details. Python programming language is indeed used to implement this algorithm. The implementation starts by initializing a variable called "vertices", which used to store the coordinates of each vertex in the generated tree, with the value of the starting position of the robot. Then, for each sample of the total number of samples k that is defined, the following tasks are performed:

- A random number between 0 and 1 is drawn from the standard uniform distribution by the Python random number generator. This number is checked whether it is smaller than the probability variable "p" or not. If yes, which happens with probability p, q_rand is set to q_goal. Otherwise, q_rand takes random pair of numbers (x and y coordinates) in the navigated map.
- Next, the euclidean distance between each vertex in the "vertices" variable and q_rand is computed and the vertex that has the minimum distance is considered as q_near.
- Thereafter, q_new is set to the coordinates of the head vertex of the vector [delta_q, 0] after rotating and translating this vector by θ and q_near respectively, where θ is the angle between q_rand and q_near vertices, as shown in the following figure.

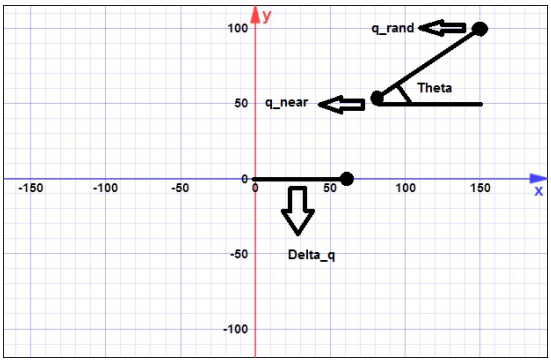


Fig. 3. The idea of computing q_new

- If the distance between (q_near, q_new) is greater than the distance between (q_near, q_rand), then q_new is given the value of q_rand since in this case the distance between q_near and q_rand is smaller than delta_q.
- The next step is to check if the map contains an obstacle at q_new coordinates. If yes, the program goes back to the step of generating a random number. Otherwise, the edge connecting q_near and q_new is tested to make sure that it does not pass through an obstacle. This is done by using an incremental strategy by which the distance between q_near and q_new is divided into e.g 10 and then this distance is increased gradually starting from q_near and each time the new vertex is checked weather it lies on an obstacle or not. In case it lies, the program goes back to the step of generating a new random number. Otherwise, q_new is stored in "vertices" variable and [index_of_q_new, index_of_q_near] is stored in "edges" variable.
- Finally, q_new is checked whether it is equal to q_goal or not. If it is equal, the "path" variable which should store the entire path between the start and goal vertex is filled in a reverse manner (traced back from the index of the goal back to the start node) and the program ends . Otherwise, new iteration starts. Basically, Path variable works as a pointer to the vertices variable and thereby it points to the free-space path that connects the start point of the robot to its destination. Therefore, the implemented function returns at the the end the set_of_goals variable which represents the vertices that are pointed to by the path variable. Nevertheless, when the program does not succeed in finding a solution during k iterations, it returns the vertices that have been explored during those iterations.

  As the generated set of nodes that the robot has to explore before reaching its final goal are randomly located on the map, a smoothing step should be taken into account in order to optimize this path in terms of the passed distance by the robot.

A. Path Smoothing

This section focuses on the implementation of a smoothing algorithm for obtaining a shorter and less noisy path. The idea is mainly to connect q_goal from q_start. If this fails, a closer position should be tried until it connects. Once q_goal is connected, the process starts again with its directly connected node until reaching the start node.

At the beginning, the following parameters are defined:

1) flip_path: contains the path from the starting vertex to the goal vertex and every element in this parameter represents the index of the corresponding path vertex.
2) v_end: stores initially the value of the goal vertex.
3) v_end_index: keeps initially the index of the last vertex in flip_path, the goal vertex index.
4) path_smooth: variable used to store the final smoothed path and it contains initially the goal point.

   A "while" loop is then used, which stops when the last element of the "path_smooth" variable

is equal to the index of the robot start location stored in the set_of_goals variable, to generate the smoothed path as follows:

- An index "i" is initialized to 1.
- Another loop is used to iterate from the i'th vertex of the flip_path parameter till the last checked vertex for connectivity, initially it is the goal vertex.
- A variable v_start is set to the coordinates of the i'th vertex of the flip_path parameter.
- A for loop is used to check incrementally if the path connecting v_start and v_end passes through an obstacle or not using the same principle stated in the RRT function (based on a transformation matrix and gradual increase by delta of the distance between the mentioned vertices). If yes, "i" is incremented by one and another iteration of the second nested loop starts. The purpose of increasing "i" is to check the connectivity with a closer vertex to v_end. Otherwise, the value of the flip_path at index "i" is used to update the path_smooth and v_end_index variables while v_end is updated with the coordinates of the vertex whose index is stored in the i'th position of the flip_path variable. Then, another iteration of the first loop starts.
- This process keeps repeating itself until the index of the start vertex is stored in path_smooth parameter, which causes the first loop to stop iterating.

IV. ROS Framework for testing RRT

The designed ROS (Robot Operating System) framework for testing the RRT algorithm on a TurtleBot is explained in this section. Since ROS works mainly with packages, we have created our own package for testing the implemented RRT algorithm. This package is called ar_project, and it consists of the following sections:

1) World folder: contains the configuration of the working environment (background map, size, pose, obstacles, etc.) and the robot itself (shape, pose, speed, sensors, and so on).
2) Source folder: contains the main node that is used to communicate with other nodes by publishing and subscribing to ROS messages and services. Furthermore, this folder contains the various functions that could be used by the node. Our source folder includes the following files:
   a) Main node file which has the responsibility of defining the required objects, organizing calls to different functions, and maintaining the general robot task.
   b) Driver file: this file contains a defined class of methods and data variables for controlling the movement behavior of the robot from one goal to another.
   c) RRT file: it contains the main RRT algorithm.
   d) Smooth file: this file is in charge of smoothing the robot path generated by the RRT algorithm.
3) Launch file: this is like the header of the package where it points to the main node of the package, calls the simulator, does default parameters allocation in the ROS server and initialization, and as its name implies it launches the package.

Nodes are in fact processes where computation is done. If it is desirable to have a process that can interact with other nodes, there is a need to create a node with this process to connect it to the ROS network. Usually, a system will have many nodes to control different functions. It is also recommended to have many nodes that provide only a single functionality, rather than a large node that makes everything in the system.

After introducing the general structure of the implemented package, the following steps explains how it works from the beginning till the end.

1) First, the launch file has to be called from the command line interface of the operating system as shown below:
   roslaunch ar_proj ar_proj.launch x:=-180 y:=320 theta:=π
   where the first name after the "roslaunch" command represents the package name and the second one is the launch file. x, y, and desired pose θ represent the coordinates of the goal in the simulator coordinate system and the required robot pose at the goal, where in the above example they are set to -180, 320, and π respectively. When any of the goal coordinates or robot final pose are not passed after the name of the launch file, the default value defined in the ROS parameters server will be used.
2) Calling the launch file will cause the simulator to run, and invoke the node file.
3) The node file creates initially an object of the "driver" class type. Defining this object will result in calling implicitly the constructor of the class which does parameters initialization and more importantly it subscribes to the topic ('/odom') on which the robot is sending (communicating) its current position, and it publishes a topic ('/cmd_vel') to which the robot is listening and waiting to receive velocity commands in order to move accordingly. Afterwards, the node calls a function defined inside the "driver" object which is load_goals().
4) load_goals function starts by loading the goal coordinate from the command line. Otherwise, the default value is loaded from the ROS parameters server. After that, this function calls the the RRT function with the goal passed as an input to the RRT and then the result of the RRT is passed to the smoothing function. Eventually, load_goals function stores the smoothed path inside the class data variables (self.x and self.y).

This means that, when the robot runs, it directly loads the map with the goal and starts executing RRT algorithm in order to know the set of goals to explore to get to the final goal.

5) The node function then calls a method from the defined object (driver). This method is called "drive" and it is responsible for keeping the driver object operating till the goal is reached or an interruption is occurred by the user. In the mean time the defined object is listening to the '/odom' topic waiting for an update about the robot current position. When an update is received, the robot prepares the differential velocity to send to the robot depending on the first goal from the set of goals and the robot current position. As soon as this differential velocity is ready, the driver object will publish it to the '/cmd_vel' topic so as to move the robot accordingly. This process of listening, preparing next goal, and publishing keeps repeating itself until the final goal is reached by the robot or the user interrupts the process. Figure 4 shows the graph that explains the communication between both the turtlebot (driver object) and Stage simulator. This graph is generated by rqt_graph program which is a Qt based framework that enables users to quickly build graphs or user interfaces.

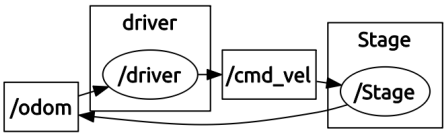


Fig. 4. rqt graph of the shared topics between the TurtleBot (driver object) and the stage simulator.

The following flowchart explains the followed steps by our framework in order to achieve the final task of planning the robot path from a start point to a goal point.

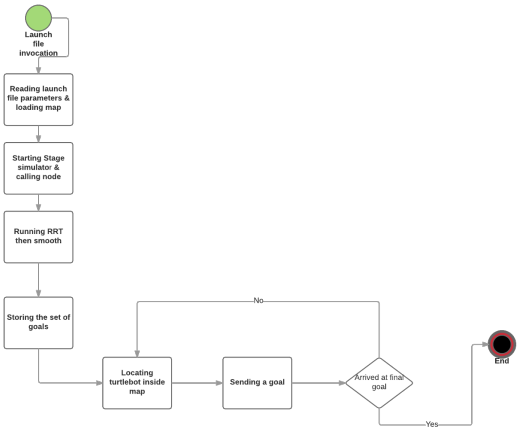


Fig. 5. Flowchart of the designed framework in ROS.

V. TANGENT BUG ALGORITHM

Bug algorithms are early approaches to path planning. Initial algorithms Bug1 and Bug2 are easy to implement but also exhaustive and greedy approaches to traverse to goal. Bug1 first looks for all choices and then traverse and Bug2 considers the first thing which is better and then traverse[1].

Tangent Bug algorithm is improved version of algorithms bug1,bug2 with range sensors and heuristic distance measurement. It is based on range sensor data acquisition and initialized by giving a goal position. Range sensor gives directed distance values of the rays emanating from the robot's current position and being scattered in a radially as shown in the figure 6. Range sensor can be modeled as $\rho = \mathbb{R}^2 X S^1 \rightarrow \mathbb{R}$ where $\rho(x, \theta)$ is the distance to the closest obstacle along the ray from x at an angle θ.

$$\rho(x, \theta) = \min_{\lambda \in [0, \infty]} d(x, x + \lambda[cos\theta, sin\theta]^T)$$

where

$$x + \lambda[cos\theta, sin\theta]^T \in \cup_i WO_i$$

For actual sensors, sensors have maximum range.

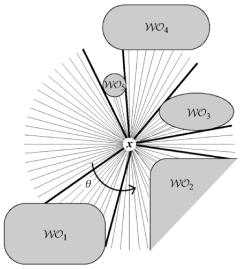


Fig. 6. Range Sensor Data Acquisition

Tangent bug algorithm follows two behaviors: move-to-goal and boundary-follow

- **Motion to Goal Behavior** First thing algorithm does is to move directly towards goal. This behaviour starts either when there is no obstacle sensed on the direction from robot to the goal or the robot can find a decreasing heuristic distance from a followed obstacle to the goal. This motion keeps executed while sensor readings yield maximum range distance result for the angle corresponding to the direction to the goal. After this, robot switches to the next part of the motion to goal behaviour wherein the robot detects points of discontinuity on the sensed obstacle. The robot then moves toward the O_i that maximally decreases a heuristic distance to the goal shown in figure 7. choose O_i that minimizes:

$$d(x, O_i) + d(O_i, q_goal)$$

- **Boundary following behaviour** Once obstacle is detected, we find discontinuities in range sensor data and find the shortest euclidean distance to the goal from the discontinuities present and move

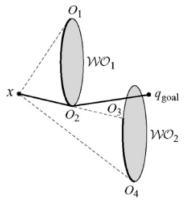


Fig. 7. Heuristic Movement

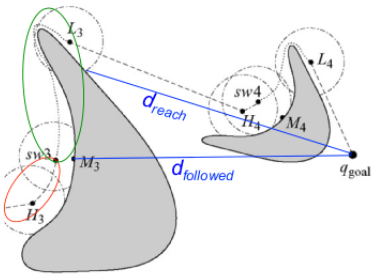


Fig. 8. Distance Measurements

towards the shortest discontinuity by following the boundary. If d_{reach} is less than $d_{followed}$ robot stops following boundary and switch back to step one that is move directly towards goal. These distance measurements can be seen in figure 8. If it detects already followed position, it exits with no solution. Pseudocode for tangentbug can be seen below[1].

Input: A point robot with a range sensor
Output: A path to the q_{goal} or a conclusion no such path exists
while True **do**
  repeat
    Continuously move toward the point $n \in (T, O_i)$ which minimizes $d(x, n) + d(n, q_{goal})$
  until
    the goal is encountered **or**
    The direction that minimizes $d(x, n) + d(n, q_{goal})$ begins to increase $d(x, q_{goal})$, i.e., the
  Chose a boundary following direction which continues in the same direction as the most recent
  repeat
    Continuously update d_{reach}, $d_{followed}$, and (O_i).
    Continuously moves toward $n \in (O_i)$ that is in the chosen boundary direction.
  until
    The goal is reached.
    The robot completes a cycle around the obstacle in which case the goal cannot be achieved.
    $d_{reach} < d_{followed}$
end while

VI. Tangent Bug Implementation

We have implemented Tangentbug algorithm using ROS and Python with stage simulator.

1) Initiating basic structure for simulation by creating robot and world frame which are defined in *frame.py*.
2) We have fixed maximum range of laserscan threshold to check for discontinuity. Laserscan returns array of distance measures, and we know the resolution of laserscan which is 0.08. Using these information and minimum angle from where scan starts which is $-180\,\mathrm{deg}$, we calculate $sensorIndex$ which points to the discontinuity.
3) In the function *tangentbug*, we first transforms the *odometry twist msgs* and calculate angle towards goal, $angle2goal = arctan(y/x)$. Using this angle we calculate $sensorIndex$ which points towards discontinuity through which robot directly points towards goal in next step.
4) We check if laser scan range of calculated index towards goal is greater than maximum range which is 3 as defined i.e no obstacle present in between and we move towards goal directly with fixed linear and angular velocity and minimum heuristics.
5) If previous step is false, that is we cannot see goal directly, we check for discontinuities using specified threshold, we calculate angle and distance measurements from the laserscan towards discontinuities as discussed in step 2. Laserscan will return distance measurement, we have to transform it to x,y coordinates to tell robot to move that position. We calculate Euclidean distance(heuristic) of that point towards goal and we know the angle towards that discontinuity. These are best heuristics.
6) We drive towards best heuristic or turn towards it if not facing using the angle we get from the previous step with constant linear and angular velocity.
7) We prioritize avoiding obstacles by keeping threshold(2 for this case) for the scan range data. We try to keep safe distance. Also, if obstacles are too close to the robot, we give a bit high threshold(2.5) as max range is 3, and try to move away with constant sharp angular velocity.
8) Stop moving if we reach goal.

VII. ROS Framework for testing Tangent Bug

As stated earlier, we are using ROS and Python with Stage Simulator for this algorithm. The structure of folders and nodes is almost same as discussed in section 5. We basically have four python source files,*frame, actionserver, goal, tangentbug*. In $frame$, we define robot and world frame structure. In $actionserver$, we check for robot movement and if it is reached or not. In $goal$, we continuously check current goals and publish them. $tangentbug$ contains the core algorithm to calculate heuristics. Also in world, we have defined robot details and we are using Hukuyo laser scan provided in Stage simulator. When launch file is called, first world file gets called and then we call all the four files that we have defined starting with frame. We can see the structure visually in figure 9.
To run the launch file, **roslaunch tbug tbug.launch** command is used.

VIII. Stage Simulator

Since the simulator that is used in this project to carry out all the tests for both of the implemented algorithms is Stage, this section briefly describes this simulator and discusses its features. Stage is a 2D physics-based simulation environment that can be used to simulate a variety of different robots (or sets of robots) using a variety of different sensors in a variety of different environments. Stage provides several different physics-based models for robot sensors and actuators. The following are some of the currently

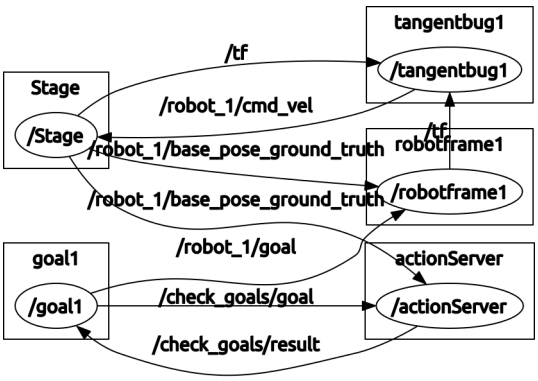


Fig. 9. ROS Framework for Tangentbug

supported models: sonar and infrared rangers, 2D scanning laser rangefinder, color-blob tracking, fiducial tracking, bumpers, grippers, and mobile robot bases with odometric and global localization, as shown in figure 6. The basic model simulates an object with basic properties; position, size, color, visibility to various sensors, etc. The basic model also has a body made up of a list of lines. Internally, the basic model is used as the base class for all other model types. Additionally, Stage is designed to support research into multi-agent autonomous systems, so it provides fairly simple, computationally cheap models of lots of devices rather than attempting to emulate any device with great fidelity[6].

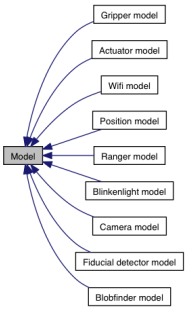


Fig. 10. Stage modules.

IX. Tests and Results

All the tests that are done on both algorithms in this project along with the results are demonstrated in this section. As stated previously, the platform of implementation is TurtleBot in ROS, the nodes, classes, and procedures are written in Python programming language, the used simulator is Stage, and the host operating system is Linux 14.04.4 LTS.

A. RRT Testing and Results

In order to test the first implemented framework, which is based on planning a path of the TurtleBot from start to goal points using RRT, different maps have been selected and various parameters were varied. Basically, there is a number of parameters that play a key role in determining the final result of this task. Those are:

1) The number of samples K. This parameter determines how many samples the program should generate at most to get the final result. The effect of changing the value of this variable is shown in figure 7 where for k=3000 the algorithm has not been able to reach the required goal. On the contrary, k=5000 is a sufficient number of samples for this start and goal pair. In all of the following figures, green points represent the nodes of the generated tree, blue lines are the tree branches, red path indicates the returned path by the RRT algorithm, and black path is the smoothed one, which is returned by the smoothing algorithm. Additionally, the start point is chosen to be the center of the image and the desired goal is a point at the top left area inside the small room.
2) The length of the generated tree branches (delta_q). This parameter controls the straight distance between the set of generated goals. This parameter is studied in figure 8 which reveals how increasing the value of this parameter affects the length of the branches of the tree and consequently makes the TurtleBot turn less during navigation.
3) The probability of generating the goal node as a random node. Figure 11 demonstrates distinct values of this parameter where as it is shown increasing this parameter a lot causes the algorithm to fail in reaching the final goal since it has less ability to explore the map well because of the high tendency toward goal. In fact, this analysis is also dependent on the map itself and the maximum number of allowed samples.
4) The step size in the incremental checking part of the RRT and smoothing algorithms. Figure 10 shows how increasing the value of this parameter highly affects the final result especially for maps with narrow edges which causes the algorithm to fail if this variable is larger than the width of the edge.

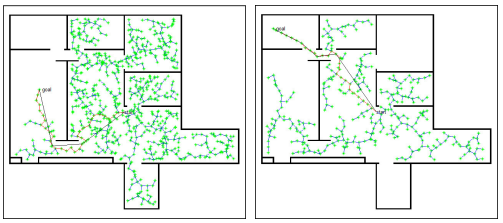


Fig. 11. Left: k=3000,p=0.3,delta=1, delta_q=20 - Right: k=5000,p=0.3,delta=1, delta_q=20.

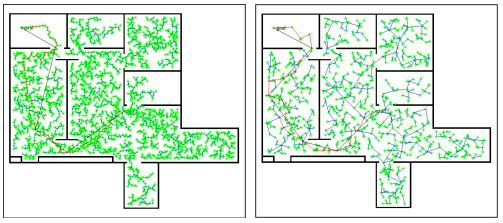


Fig. 12. Left: k=10000,p=0.3,delta=1, delta_q=10 - Right: k=10000,p=0.3,delta=1, delta_q=50.

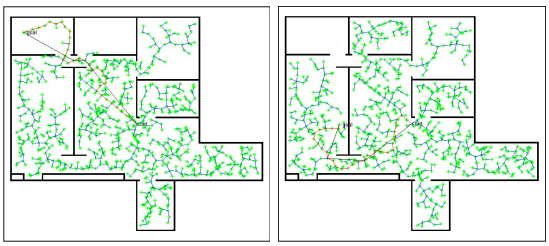


Fig. 13. Left: k=10000,p=0.5,delta=1, delta_q=20 - Right: k=10000,p=0.8,delta=1, delta_q=20.

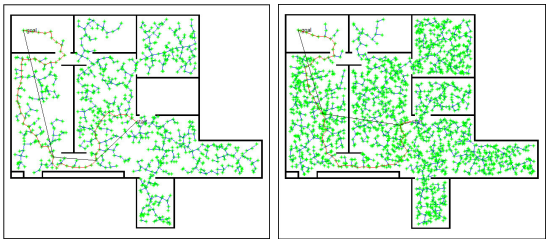


Fig. 14. Left: k=10000,p=0.3,delta=10, delta_q=20 - Right: k=10000,p=0.3,delta=20, delta_q=20.

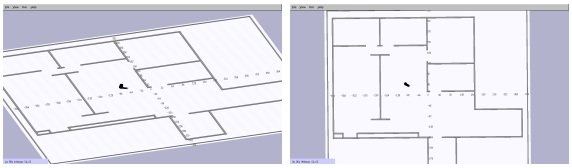


Fig. 15. Result from Stage Simulator.

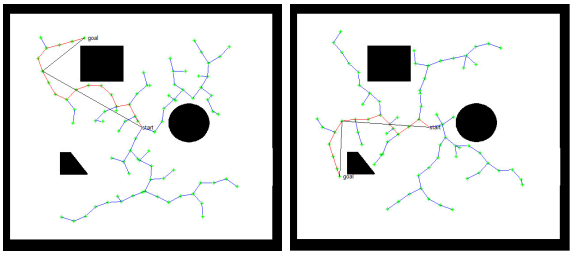
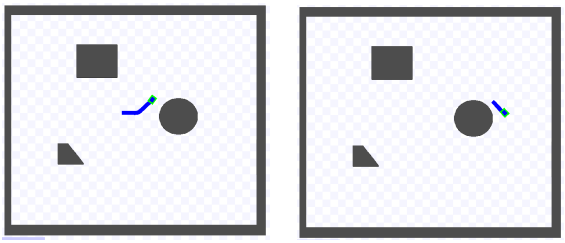


Fig. 16. Another map tested with two arbitrary set of parameters, Left: k=5000, p=0.3, delta=1, delta_q=40 - Right: k=2000, p=0.3, delta=1, delta_q=40.

Note: Results from figure 11 to 15 are plotted using Matlab to show tree lines and trajectory.

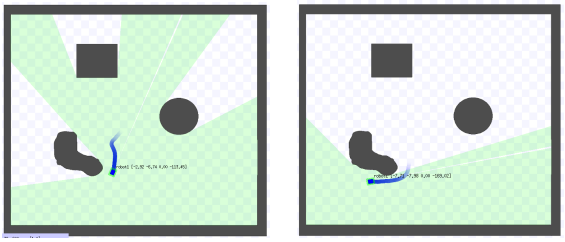
B. TangentBug Testing and Results

We checked our algorithm for finite sensor range and zero sensor range, also with different maps. To upload different image files of maps in world file to check the environment. Figure 18 shows finite sensor range response. While, in Figure 19 and Figure 20, we try different map with finite and zero sensor range. We can see that in zero sensor range, robot moves close to the boundary of obstacle to reach to the goal. As you can see from the Figure 20 that if the sensor range is high, the robot moves far away from obstacle to reach goal.



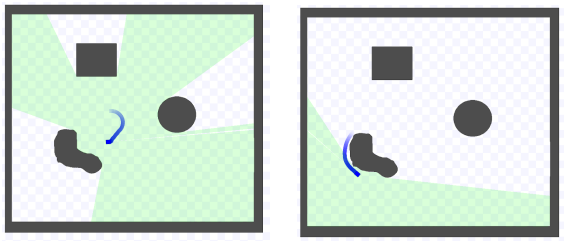
(a) Map 1 : Initial Robot Movement (b) Map 1 : Robot Movement Towards Goal

Fig. 17. Map 1



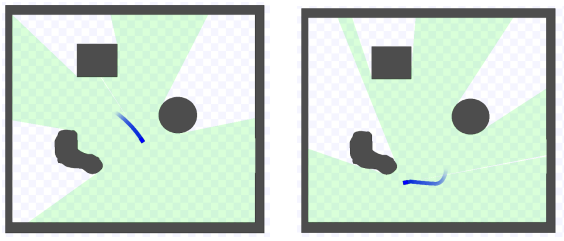
(a) Map 2 : Initial Robot Movement (b) Map 2 : Robot Movement Towards Goal

Fig. 18. Map 2 : Maxmimum Range of Sensor: 3



(a) Map 2 : Initial Robot Movement (b) Map 2 : Robot Movement Towards Goal

Fig. 19. Map 2: Effect of making maximum range of Sensor: 0



(a) Map 2 : Initial Robot Movement (b) Map 2 : Robot Movement Towards Goal

Fig. 20. Map 2: Effect of making maximum range of Sensor: 13

X. Conclusion

In this project, two high level robot control architectures, Delibrative and Behavioural, are designed, implemented, analysed, and tested on different maps using TurtleBot in ROS with Stage Simulator. The aim of these tow control architectures is to move a robot from a start point to a goal. The Deliberative approach uses RRT as a path planning algorithm which is widely

used and efficient with obstacles with differential constraints in which states are defined by path taken by the system. It also has a predictable behavior but a slow reaction. However, the Behavioral architecture uses the Tangentbug algorithm which works on sense and react methodology. For tangentbug, behaviour is random but it is faster than delibrative and there is no world model. This clearly demonstrates the need for a hybrid architecture which takes advantage of and accentuates the benefits of both approaches.

REFERENCES

[1] N. Sariff, N. Buniyamin. An Overview of Autonomous Mobile Robot Path Planning Algorithms. 4^{th} Student Conference on Research and Development (Scored 2006), June 2006.

[2] M. LaValle. Rapidly-Exploring Random Trees: A new Tool for Path Planning.

[3] V.Alcazar, M.Veloso, D.Borrajo. Adapting a Rapidly-Exploring Random Tree for Automated Planning. The Fourth International Symposium on Combinatorial Search (SoCS-2011)

[4] Lecture Notes by Prof. Marc Carreras.

[5] Robot Operating System. `http://wiki.ros.org/ /~uno/abcde.html`

[6] Stage Simulator. `http://playerstage.sourceforge.net/index.php?src=stage /~uno/abcde.html`