Active Slam and Path Planning for Mobile Robot Navigation

Saad Katrawala
ASU ID: 1213260514
skatrawa@asu.edu

Sai Krishna Bashetty
ASU ID: 1213376903
sbashett@asu.edu

Krishna Chaitanya Jinka
ASU ID: 1213246279
kjinka@asu.edu

Achyutha Sreenivasa Bharadwaj
ASU ID: 1213094192
asbhara2@asu.edu

Abstract- This paper performs a simulation study for SLAM in the ROS environment. It is modeled based on the paper[1] named Active SLAM-based algorithm for autonomous exploration with a mobile robot. This paper presents an algorithm for exploration and mapping of a fully autonomous, partially mapped indoor robot environment. This algorithm is based on the active SLAM. The mobile robot equipped with a laser sensor builds a map of an environment while keeping track of its current location. The paper implements an active slam approach for goal state selection in a real-world environment. Our simulation study involves implementing the benchmark algorithm and then extending it by incorporating recovery mechanisms and a fine-tuning parameter for better accuracy of the map and reduced computational time.

Index Terms—Active SLAM, dynamic environment, exploration, negative edge weight in a graph, path planning, simultaneous localization and mapping (SLAM).

1. INTRODUCTION

A model of the operative environment is an essential requirement for an autonomous mobile robot. The construction of this model requires the solution of at least three basic tasks for a mobile robot, namely localization, mapping and trajectory planning. The intersection of the first two tasks defines a key problem in modern robotics: Simultaneous Localization and Mapping (SLAM).

SLAM is a hugely popular and widely used technique in the robotics community for mapping an unknown, static or a slowly changing environment. There is not much research being done on designing offline algorithms that could map the whole area efficiently. Instead, recent works in this area are focused on designing online algorithms that estimate the current position and previous results and sets goal points in order to maximize the map coverage and accuracy. Some of the recent popular techniques include methods based on Kullback-Leibler divergence for evaluating the SLAM posterior approximations. Hierarchical Bayesian approach to determine what area should be visited next.Using model predictive control and an attractor to incorporate long-term goals; actively closing loops that are made during the exploration

Autonomous exploration under uncertain robot location requires the robot to use active strategies to a trade-off between the contrasting tasks of exploring the unknown scenario and satisfying given constraints, also known as Active SLAM.

The algorithm presented in this paper takes previous mapping results into account and combines them with cost maps that are used for autonomous navigation. In such a way, we can determine the “cheapest” goal point that will also increase the coverage and accuracy of our map. Goal points are chosen by combining multiple criteria. First, we have to make a distinction between the areas that lie outside the range of the sensor, and the areas that are close enough to be mapped but are hidden by an obstacle. Then we weight possible actions and pick the most favorable. Once goal points are set, we need to build an efficient path from the current position, and also to

provide safe motion so that the robot moves without colliding with static obstacles.

Recently, three-dimensional obstacle sensing is being used almost in all scenarios, usually in the form of an actuated laser range-finder or a stereo camera pair. Some sensors even combine laser and camera technology, in a pulsed or line-stripe fashion. The availability of such devices provides an opportunity for an indoor robot to sense and avoid almost all hazards that it might encounter. The challenge lies in interpreting, storing, and using the data provided by the three-dimensional sensors. This paper presents a robot navigation system that exploits the three-dimensional obstacle data to avoid even the smallest obstacles that can be perceived with the available sensors, yet drives through the tightest spaces that the robot can fit. The ROS environment provides an efficient technique for constructing, updating, and accessing a high-precision three-dimensional Voxel Grid. This structure encodes the robot's knowledge about its environment, classifying space as free, occupied, or unknown. Using this grid, the robot is able to plan and execute safe motions in close proximity to obstacles. In our work, we use the existing ROS libraries for the same.

2.PREVIOUS WORK

A wide variety of robots have demonstrated the ability to navigate successfully in human environments. A common theme is robotic museum tour guides. RHINO was the first robotic tour guide and was followed soon after by MINERVA which led tours through the Smithsonian Museum in 1998. Other robots have followed, including Mobots and Robox. These robots have dealt with environments that are often crowded with people and other obstacles with varying degrees of success. Most cite their primary difficulty as robust localization and have focused on creating localization systems that are reliable in these highly dynamic environments. In attempting to apply the navigation techniques used by these platforms to the PR2, however, another pressing issue emerged that was not addressed: the ability to reason about three-dimensional space, both occupied and unknown, in a principled manner.

Previous techniques for handling three-dimensional obstacles range from restricting the path of the robot to corridors known to be clear in the nominal case to building a three-dimensional octree of the environment and computing bounding boxes around obstacles The first approach fails whenever a difficult obstacle enters a navigation corridor, and the second approach requires the robot to stop for a prolonged period of time, take a full scan of the environment, create a plan, and then fall back on two-dimensional sensing if a dynamic or occluded obstacle is encountered while executing the plan. In cluttered environments such as offices, both approaches would be highly susceptible to collisions and have poor performance.

In the outdoor navigation domain, there has also been work on three-dimensional perception. Stanford's autonomous car Junior utilized a three-dimensional free space analysis from its Velodyne laser rangefinder. This free space analysis was performed radially in the ground plane and allowed dynamic obstacles recorded in the map to be cleared quickly as they moved to a new location. This approach is limited, however, by the fact that only the two-dimensional projections of obstacles were stored. For a sensor like the Velodyne, this works because the laser returns 360-degree scans of the world at a high rate.

3. IMPLEMENTATION

As we know probabilistic SLAM is broken down into two main subproblems. 1. Localization, which estimates the robot pose and 2. Mapping, which estimates the location of the landmarks. To address these problems, SLAM uses motion model **[2]** and the observation model **[3]** respectively as seen in the equations.

$P(x_k | x_{k-1}, u_k)$

$P(x_k, m | z_k) = P(x_k | z_k) * P(m | z_k)$

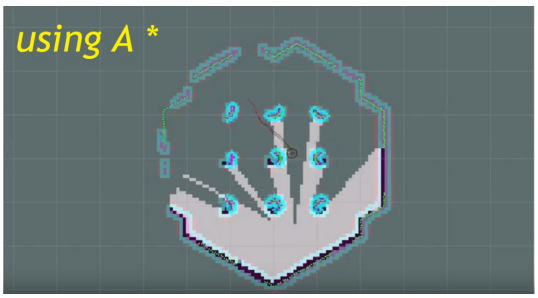

Fig. 1 Image showing mapping for A *

These models as we have seen require the observations from the sensors and the control signals for the robot's motion. The global planner and the local planner which decide which path is best to reach the goal state, provide us with the controls in the form of state vectors with acceleration and velocity. Keep in mind that this is assuming that we have a goal state, we'll discuss how we decide the goal state in the baseline implementation. Now that we have the control and the observations, the ROS calculates the probability distribution with an abstraction of a global cost map. These cost maps are indicative of the obstacle locations. These cost maps are fed to the goal planner, which will decide which state to go to next. This is where the active part of the active SLAM comes in.

3.1 Overview of the ROS architecture

The way ROS integrates all these components together is by using a publisher-subscriber mechanism. At the heart of it is a socket-based inter-process communication mechanism. To request information you subscribe to a node which is transmitting data called a publisher. Figure 2 below, is the communication between ROS nodes specific to our problem implementation.

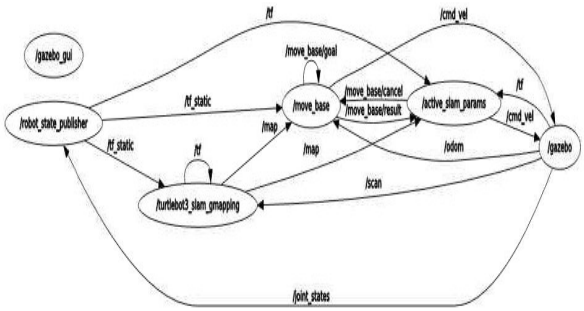

Fig. 2 ROS nodes following publisher-subscriber

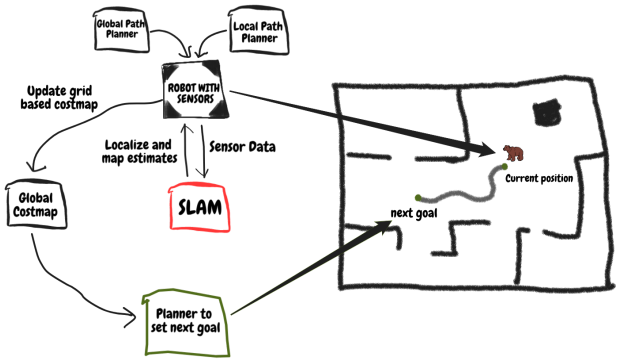

Fig. 3 Overview of Active SLAM system

The navigation stack gathers information from various sources. From those, the ones that are of particular interest to us are :

- Sensor sources: These give the point cloud extracted based on particle filtering. Thus helping us in getting information about the observation model.

- Odometry sources: These provide information about coordinates of the robots with passing time. Thus this helps in identifying the robot's pose.

3.2 The ROS/Environment setup used

The existing ROS packages we are using in this implementation are the turtlebot3_navigation, turtlebot3_gazebo, and turtlebot3_slam. These packages use a turtlebot3 simulation robot in a 3D simulator gazebo. The simulated robot is equipped with sensors and the default world provided by the packages is used as a testing environment. The laser and odometry data provided by the simulated sensors is used by the turtlebot3 slam node to localize and map the environment. The slam technique used is based on rao-blackwellized particle filters. The turtlebot3 navigation node uses the navigation stack to move to a specific location.

In general, a current pose estimate is required by the navigation stack and a predefined map is used along with the amcl ros package for monte carlo localization is used. But, in our case, the map is explored on the go and the robot needs to navigate

simultaneously. For this purpose, we use the rough pose estimate given by the slam algorithm and the results look promising with this approach.

Moreover, the default global planner used in ROS turtlebot packages is A* planner and the heuristics used are

H(n) : $|x1 - x2| + |y1 - y2|$

G(n): 1 if in the perpendicular straight line

1.414 if diagonal

This is the distance between the center of the two grids. Here (x1,y1) and (x2,y2) are the centers of the grid points and H(n) is the heuristic function, which is an estimation of the distance while G(n) is the actual distance and H(n) is considered monotonic if for every node and every successor n' of n generated by a,addition of the estimated cost of reaching the goal from n is no greater than the step cost of getting n' and the estimated cost of reaching the goal from n'. The heuristics is admissible and consistent.

H(n)<=C(n,a,n')+H(n')

3.3 Baseline Implementation

3.3.1 Scope

The implementation is based on the assumption that there is only a single bot in the map and can be scaled to a multi-robot system, with additional efforts. In multi-robot setup, proper care needs to be taken to combine the multiple maps generated by individual agents to form a single global map. This is due to the fact that cost map updates might change based on each robot's perception of the landmarks and different architectures such as master-slave mechanism might have to be used.

3.3.2 Algorithm

Goal State planner

The algorithm for classifying the gaps in the global cost maps into hidden states and frontier states. The paper classifies them based on the range of the sensor and the robot's orientation relative to the goal state in question. The frontier states as defined in the paper are the ones that lie on the edge of laser's range of effect and areas that are not mapped because they are hidden by another obstacle. This is done based on the data obtained from the global cost map that is used in A* search(or for that matter any global planner). Scan global occupancy matrix for gaps, and identify whether they are orthogonal within several degrees to robot position and within the distance sensor's range at the same time. If not, it marks such gaps as areas hidden behind an obstacle. Hidden areas are given priority over the edge frontiers, as they are closer and thus usually cheaper to explore. If it finds more hidden areas to consider, it picks the closest one.

These gaps are put on an exploration stack. From this stack the algorithm identifies a state and based on certain conditions, it removes the state from the exploration stack.

The way the algorithm manages the stack is described. It starts with hidden states on the stack. The algorithm chooses the nearest hidden state as the goal state. Once, it exhausts all the frontier states, it starts with the edge frontier states. The goal is to make robot orientation vector orthogonal to the edge so that we would cover as much area as possible with as little movement as possible. Each frontier is segmented, and then a plan for the trajectory to each segment is constructed. After that, it monitors the map and picks trajectory that would cover map segments with the highest covariance. This is done to reduce the computational time to explore the entire map. In this way, the robot does not have to revisit a specific area in order to increase accuracy multiple times. Once the algorithm picks a segment for exploring, the robot follows the trajectory that was already planned. If robot for any reason fails to reach the goal, it plans a new trajectory to the same spot and tries to reach the goal state. If the algorithm fails again, that spot is moved to the last place on the stack, so that we revisit it later, but still don't lose much time on retrying, as the algorithm is not bound to be complete, and spots that are not able to be explored are always possible. If the hidden area is successfully visited, but still not discovered completely (not very clearly defined, as to what discovery implies), it is moved to the last place on the stack and marked as already visited. If it is visited once again, but still not mapped correctly, the node is removed from the stack to prevent the algorithm to enter in an infinite loop. Thus the number of revisits that the robot makes is two.

```
Start
Initialize Stack
Run SLAM iteration, build global costmap
Procedure Fill stack:
Scan global costmap for gaps
for Each gap In global costmap do
    if gap is Not On Stack then
        if gap is Not orthogonal within 10 deg to the robot position And gap is inside sensor range then
            Set gap As Hidden
        else
            Set gap As Edge frontier
        end if
        Put gap On Stack
    end if
end for
end Procedure
```

Fig 4. Pseudo Code for state classification

```
Procedure Set current goal:
Find closes Hidden gap on Stack
if one exists then
    Set it as current goal
else
    for Each Edge frontier gap do
        Compute trajectory using A*
        Compare trajectory to the Map, calculate Sum of uncertainty
    end for
    Set Edge frontier gap with highest Sum of uncertainty ss current goal
end if
if current goal is Not set then
    end Procedure
end if
```

Fig 5. Pseudo Code for Stack Management

3.4 Our version of the Baseline

3.4.1 Setup

In contrast to the benchmark simulation environment which uses stage simulator, the current implementation uses the Gazebo simulator to implement the algorithm. This makes it difficult to provide a comparison with the actual benchmark results. Instead, we evaluate our version of the baseline algorithm and propose improvements to the same. The map that we chose, to begin with, is a basic map provided by the turtlebot environment. The turtlebot environment provides us with a basic configuration files of the sensors and other models required by ROS. Due to its open source nature and the ease of integration of the goal state planner, we chose to use that.

3.4.2 Algorithm changes in our baseline version

Due to some ambiguities and challenges faced in the implementation of the baseline algorithm in the paper, we had to modify some of the aspects.

Coverage area: Once the robot reaches a goal position, it checks whether the coverage is good in that position. For some reason, if the coverage is not proper, the state is marked to be visited for next time. The authors did not specify a clear metric to define the coverage area. We are checking the surrounding two layers of neighbouring grid locations and deciding the coverage.

Frontier goal selection: The authors mention that a frontier goal is selected by computing the trajectories to all frontier locations and choosing the one which has the highest sum of uncertainty. The concept of uncertainty in this aspect is vague and not clearly described. Additionally, we believe that calculating the trajectories at each step provides a computational overhead. From the basic frontier exploration methods, we found out that a random selection of goals from frontier states provides satisfactory results with little computational overhead.

3.4.3 Challenges with ROS implementation

We did not have the original algorithm implementation and we had to do it from scratch using the pseudo code.

Communication frequency: The ROS nodes need to communicate with each other at a specified frequency and if a particular message deviates from its frequency rate due to computational overhead, the robot misses the control loop and deviates from its regular behavior and most of the times it leads to improper localization and maps.

Global Planner cost factor: One of the important parameter we had to tune to make the algorithm work is the cost factor for the global planner. This parameter controls the quality of the path planned by the global planner. If it is set too high or too low, the path planned(not taking the robot dimensions into

consideration) is very close to obstacles and the robot gets stuck while crossing the obstacle. The value we used is 0.55.

Local Planner simulation time and goal tolerance: The Dynamic window approach local planner is used to avoid obstacles locally while trying to follow the global path. One of the major problems was that if the goal location is too close, the local planner is not able to plan an accurate path to reach the destination and just revolves around the goal. This is due to the number of time steps into future which the local planner plans or in a naive way, the length of path returned by the local planner. This behavior has been controlled by adjusting the sim time and the goal tolerance of the robot.

3.4.4 Results for baseline implementation

Map Resolution(m/grid)	Exploration time
0.05	23 min 16 sec
0.06	12 min
0.075	8 min

We ran the simulation to map a default turtlebot world. For a higher map resolution, the simulation tool a very long time to complete exploring and we had many problems for the grid locations at the corners of obstacles.
Even with lower map resolution the problem with very near goal locations is not completely resolved.
We used two different planners **A*** and **Dijkstra's** but there was not much change in results.

4. EXTENSION

To overcome the above problems and speed up the simulation, we improved some parts of the algorithm and experimented on certain parameters.

4.1 Changes to the algorithm

Hidden Goal State Selection: To avoid the problem of choosing very near goal locations and to improve the coverage with less number of movements, we introduced a threshold method to choose the next hidden goal state. To choose a particular goal state, we start computing the length of planned trajectories starting from the top most hidden goal position in the map. The lower threshold ensures that the goal state chosen is not very close to the robot. Also if we choose the farthest point we observed that the quality of the map reduces when the robot takes a single long path. Upper threshold controls this behavior. We select the first goal location found in the given limits.

Validate the path to the goal location: Before we finalize the goal location, we check whether the goal path is feasible to that location. This reduces a lot of unnecessary movements and saves time.

Recovery Behavior: Even though we validate a path before giving the goal location, sometimes due to the dynamic nature of environment or noise in sensing, the robot may not be able to reach the goal. For these situations, ROS has a recovery behavior but this induces unnecessary movements and slows the exploration process. Instead, we replaced that behavior with a simple but efficient method. If the robot is unable to reach a location, it simply goes back to its previous location. This method proved to be very efficient and also prevented being stuck in the corners of obstacles.

Rotating behavior: Additionally, after moving to a goal location, the robot rotates for n seconds to improve the coverage in the surrounding area and reduces the number of states to be explored.

4.2 Results and Experiments for extension

NOTE: All the following results are comparative within their specific categories. If other external parameters are changed, the results may vary.

Map Resolution(m/grid)	Exploration time
0.05	15 min 16 sec
0.06	8 min 23 sec
0.075	4 min 35 sec

We clearly observe an improvement compared to the values from benchmark implementation.

All the following experiments are conducted on a fixed map resolution of 0.075m/grid and the simulation is run until a visually good coverage is obtained. Other parameters might be slightly adjusted which are not mentioned here. Observe the comparative performance only.

Number of particles in SLAM:

No: of particles	**Exploration time**
100	5 min 45 sec
70	6 min 22 sec
40	8 min 54 sec

We can observe that as the number of particles decreases, the exploration time increases and with a decrease in map quality.

Laser Range of Sensor

Laser Sensor range	**Exploration time**
3m	4 min 24 sec
2.5m	5 min 29 sec
2m	8 min 54 sec

We can observe that as the laser sensor range decreases the time increases. Also for the last case, the quality of the map was not accurate.

Hidden Goal Selection Thresholds:

LT(cm)	**UT(cm)**	**Exploration time**
20	50	6 min 22 sec
10	40	4 min
20	40	3 min 25 sec
20	30	5 min 50 sec

LT → lower threshold
UT → Upper threshold

This is an **important** evaluation result which we performed and it shows that our extension can be further fine-tuned to reduce the exploration time. The reason for the above trend is vague though. It might be because of optimization between computing the number of trajectories and proper goal selection for improving the coverage in less number of movements.

Finally, we have tested the same on a bigger environment and the final result is shown below

It took around 20 minutes to explore the area.

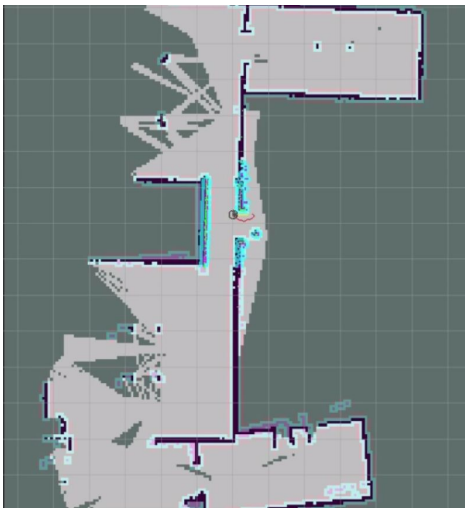

Fig. 6 map of a larger environment

5. CONCLUSION

Finally summarizing some important results of the experiment:

- By varying the threshold of the goal state selection for the hidden states we get different run times, by incorporating learning algorithms, these parameters can be fine-tuned for a particular application.
- We observed that with increasing the sensor range(within practical bounds), the time required to cover the entire map reduced.

- Map resolution of 0.075 takes less time to complete without any effects on the accuracy.
- Last, but no the least the number of particles in the point cloud affects the accuracy of map discovery and thus increases the time for exploring the entire graph.

REFERENCES

1. [Benchmark]https://ieeexplore.ieee.org/document/7125079/citations?tabFilter=papers#citations
2. https://ieeexplore.ieee.org/document/4522409
3. http://www.iri.upc.edu/people/jsola/JoanSola/objectes/curs_SLAM/SLAM2D/SLAM%20course.pdf
4. https://ieeexplore-ieee-org.ezproxy1.lib.asu.edu/stamp/stamp.jsp?tp=&arnumber=5509725https://answers.ros.org/question/205521/robot-coordinates-in-map/
5. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5876943
6. https://pdfs.semanticscholar.org/9afb/8b6ee449e1ddf1268ace8efb4b69578b94f6.pdf
7. https://ieeexplore.ieee.org/document/7125079/metrics#metrics
8. https://lamor.fer.hr/images/50020776/Mauro vic2017.pdf
9. http://msl.cs.illinois.edu/~lavalle/papers/TovGuiLav04.pdf
10. https://www.robots.ox.ac.uk/~mobile/Theses/rjs_thesis.pdf