Implementation of D* Lite using Turtlebot3

Hrishikesh Tawade
Robotics Engineering
A. James Clark School of Eng.
University of Maryland
College Park, USA
htawade@umd.edu

Siddhesh Rane
Robotics Engineering
A. James Clark School of Eng.
University of Maryland
College Park, USA
srane96@umd.edu

Niket Shah
Robotics Engineering
A. James Clark School of Eng.
University of Maryland
College Park, USA
sniket28@umd.edu

***Abstract*— Path planning is the most important aspect of navigation problems. Over the past years, many path planning algorithms were proposed. The most popular among these are A* and Dijsktras algorithm. They can efficiently find an optimal path between the start and the goal destination. However, their efficiency fails in dynamic or partially known environment as they recalculate the path from scratch whenever a change in the environment is encountered. It can be computationally costly in real time applications. Therefore, this paper introduces an efficient means of path planning in a dynamic environment using D* Lite algorithm which is a dynamic version of A*. Further, the application of this algorithm in a real-life scenario is demonstrated by implementing it on a Turtlebot3 robot to traverse in a lab environment.**

Key words- D* Lite, Path planning, Autonomous robots, ROS

I. INTRODUCTION

Several robots have been designed in recent times that assist humans in indoor environments, be it homes or labs or even factories. Their task is to assist humans in everyday tasks. The so-called Service Robots are designed to perform tasks that are dull, dirty, dangerous or repetitive, for humans [1]. The robots help in reducing the time and also increase the efficiency of the human. Path planning is the backbone of these tasks. The path planner algorithm, with the help of the visual information about the surroundings, generates a navigable path. The planner also needs to take care of dynamic obstacles, which in the indoor environments can be humans and also other robots. The planner must take into account such obstacles and generate a new path accordingly. Ideally, the path is generated using A*[2] or Dijkstras [3] algorithm using a static map initially generated. But these maps arent updated by the information of dynamic obstacles and hence the static maps are either invalid or suboptimal. The two approaches to resolve this issue would be to re-plan the path from scratch using the updated map or to fix the path more efficiently to maneuver the environment by avoiding the obstacles detected [4]. The algorithm D*[5] is a more dynamic version of A* which prioritizes computational efficiency over the optimality in the dynamic environments. It prefers using heuristics and reuses information from the previous searches, which is much faster than creating maps again from scratch. The applications of D* and its more simple version D* Lite is very popular in mobile and autonomous robots, including the Mars Rovers[6].
[7] The path planning algorithms can be essentially divided in four main categories: 1) off-line algorithms, 2) on-line algorithms, 3) incremental algorithms, and 4) soft-computing algorithms. The off-line planning algorithms implement and output the solutions before execution, whereas in on-line search algorithms couple the planning and execution phases and hence the agent repeatedly plans and executes the next movement. In dynamic or unknown environments, the off-line planning algorithms suffer from execution time, on the other hand, the on-line algorithms result in sub-optimal solutions in terms of path length. Incremental algorithms combine the advantages of both the off-line and on-line counterparts, resulting in a better execution time with optimum solutions. They reuse the past data to reduce the number of iterations. Soft computing is usually used in multi-objective shortest path finding.

Various different variants of D* algorithm were studied during the course of the projects, having their use cases in different domains such as video gaming as well as warfare. [8]Game worlds are often dynamic, meaning the environments can constantly change, demanding a faster replanning. The replanning can be faster with the use of data from previous searches. But, in gaming scenarios, there are several moving agents and other highly dynamic factors. Replanning to adapt to these changes in infeasible, and thus the games generally use a local planner or steering algorithm to avoid these obstacles. This paper [8] discusses about their modification to D* Lite algorithm, introducing an additional priority queue V, to store all the nodes that potentially need to be updated for the current start node. Also, unlike D* Lite, the algorithm updates all h-values each time the agent moves in order to minimize the computations of edge costs. [7]Consider the scenario of a warfare setting, wherein the navigation of an UAV is to be controlled in a 3-D terrain. The navigation algorithm's task is to find the shortest but also the safest path considering the opponent forces, but such information cannot be detected by the UAV's sensors. So, the paper talks about a variant of D* Lite which is MOD* (Mutliobjective D* Lite) which is an incremental path planning algorithm.

II. METHOD

This section will discuss our approach towards achieving the goal discussed in the previous section. We will start by

discussing the algorithm and simulations using it. We will move towards the implementation in real-world.

A. D* Lite Algorithm

Algorithm 1 D*Lite with Human Assistance

```
    procedure CalculateKey(s)
1:  return [min(g(s),rhs(s))+h(s_start,s)+k_m;min(g(s),rhs(s))];

    procedure Initialize()
2:  U = ∅;
3:  k_m = 0;
4:  for all s ∈ S rhs(s) = g(s) = ∞;
5:  rhs(s_goal) = 0;
6:  U.insert(s_goal,CalculateKey(s_goal));

    prodecure UpdateVertex(u)
7:  if (u ≠ s_goal) rhs(u) = min_{s'∈Succ(u)}(c(u,s')+g(s'));
8:  if (u ∈ U) U.remove(u);
9:  if (g(u) ≠ rhs(u)) U.insert(u,CalculateKey(u));

    prodecure ComputeShortestPath()
10: while (U.TopKey() <CalculateKey(s_start) OR rhs(s_start ≠ g(s_start))
11:     k_old = U.TopKey();
12:     u = U.Pop();
13:     if (k_old <CalculateKey(u))
14:       U.insert(u,CalculateKey(u));
15:     else if (g(u) > rhs(u))
16:       g(u) = rhs(u);
17:       for all s ∈ Pred(u) UpdateVertex(s);
18:     else
19:       g(u) = ∞;
20:       for all s ∈ Pred(u) ∪ {u} UpdateVertex(s);

    procedure CalculatePathSize()
21: P = ∅;
22: s = s_start;
23: while (s ≠ s_goal)
24:     P.insert(s);
25:     s = min_{s'∈Succ(s)}(c(s,s')+g(s'));
26: return |P| * cell_size /* in centimeters */;

    procedure Main()
27: s_last = s_start;
28: Initialize();
29: ComputeShortestPath();
30: while (s_start ≠ s_goal)
31:     /* if (g(s_start) = ∞) then there is no known path */
32:     s_start = arg min_{s'∈Succ(s_start)}(c(s_start,s')+g(s'));
33:     Move to s_start;
34:     Scan graph for changed edge costs;
35:     if any edge costs changed
36:       k_m = k_m + h(s_last,s_start);
37:       s_last = s_start;
38:       for all directed edges (u,v) with changed edge costs
39:         Update the edge cost c(u,v);
40:         UpdateVertex(u);
41:       p_size = CalculatePathSize();
42:       ComputeShortestPath();
43:       p'_size = CalculatePathSize();
44:       if (|p'_size − p_size| > min_deviation)
45:         Ask for Help;
46:         timeout = |p'_size − p_size| * average_speed;
47:         while (timeout > 0)
48:           Scan graph for changed edge costs;
49:           if any edge costs changed
50:             k_m = k_m + h(s_last,s_start);
51:             s_last = s_start;
52:             for all directed edges (u,v) with changed edge costs
53:               Update the edge cost c(u,v);
54:               UpdateVertex(u);
55:             ComputeShortestPath();
56:             break;
```

Despite several advances in robotics, robots still have limitations at the perception, cognition, and execution levels. For example, the robot we used in our experiments lacks arms so it could never move blocking objects in its way. However, some of the robots limitations are strengths for humans who coexist with robots in the environment. In this sense, humans can clean robots paths by lifting and moving objects, while robots are navigating in the environment in order to execute tasks for humans.[4]. Therefore an extension of D* Lite is introduced in the following section which integrate human assistance functionality in the new trajectory generation section of the original algorithm. In order to understand D* Lite algorithm, it is necessary to first understand the node structure the algorithm uses.

D* Lite algorithm is an incremental heuristic heuristic search algorithm derived from Lifelong planning A* algorithm. In this algorithm, the parent node is called the 'successor' and it's children are called it's 'predecessors'. Edges are directed from predecessors to their successor. All nodes have two cost parameters g(X) and rhs(X). For a given node X, g(X) is the objective function value known as g-cost which represents the start distance and rhs(X) is the one step look-ahead cost value calculated using its successor's g-cost g(S) added with path cost c(X,S) between them.

$$rhs(X) = g(S) + c(X,S) \tag{1}$$

The algorithm starts from the given goal node, and backtracks to the start node by minimizing the rhs value. Furthermore, this algorithm is driven by the concept of the 'local inconsistency'. A node X is locally consistent if g(X) = rhs(X) and it is locally inconsistent if g(X) $\neq$ rhs(X). Unlike traditional path planning algorithms, open list (priority queue) is stored with the nodes which are inconsistent. Based on values of g-cost and rhs, there are two types of inconsistencies:

1) Under-consistent: $g(X) < rhs(X)$
2) Over-consistent: $g(X) > rhs(X)$

An under-consistent path means that the path to the node was made expensive. This can happen if a node was previously in free space and now obstructed by an obstacle making it unreachable.Similarly, an over-consistent path means that the path to the node was made less expensive. This can happen if a previously obstructed node is now free from obstacles.

This algorithm uses a value known as 'key' to sort the open list (priority queue). The key/priority of a node X on the open list is the minimum of g(X) and rhs(X) plus a focusing heuristic h(X). The first term is used as the primary key. The second term is used as the secondary tie-breaking key.

$$key = [min(g(X); rhs(X)) + h(X); min(g(X), rhs(X))] \tag{2}$$

Fig(1) shows the pseudo-code of the algorithm that is implemented in this project. First key is generated as explained above (line 1). Next, initialization process is called in which an empty priority queue is created. The values of rhs and g-cost of all the nodes except the goal node is set to ∞ initially. For the goal node, rhs is set equal to zero. (line 2-6). Since, goal node is inconsistent as $rhs(goal) \neq g(goal)$, it is added to the priority queue U. Function Compute ShortestPath() (line 10-20) initiates by popping the minimum element from the open list using the key obtained from the Calculatekey(). This node is now the currently active node. If active node is over-consistent $g(X) > rhs(X)$ then $g(X) = rhs(X)$. Next UpdateVertex() (line 7-9) is called on each predecessor of the

currently active node (line 20). This process is repeated until start node is expanded i.e. it is made consistent.

Once a path is created from start to goal node, the gradient of g values is followed to reach the goal node. If an obstacle is detected during the path traversal, new path is calculated (line 35-43). If this path is greater than some predefined maximum threshold (line 44), robot calls for the human help (line 45).

B. Simulation in Gazebo and Custom GUI

1) D Lite Simulation:* Initially we simulated the D*Lite algorithm on a custom OpenCV GUI wherein we used the map of Robotics Realization Lab's environment and created the obstacle space using half planes and algebraic methods and Minowski sum. We can see the exploration as shown in 1. Here the exploration starts from the goal and move towards the start. The region in red are consistent node and the region in green are in open list. Here we have considered the action space to be the adjacent 8 neighbours with diagonal neighbour cost as $\sqrt{2}$.

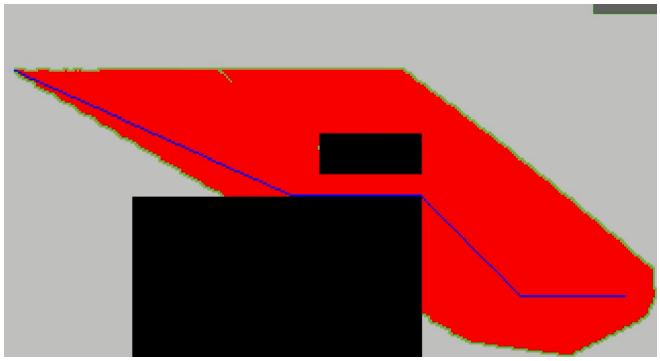

Fig. 1. Node exploration using D* Lite

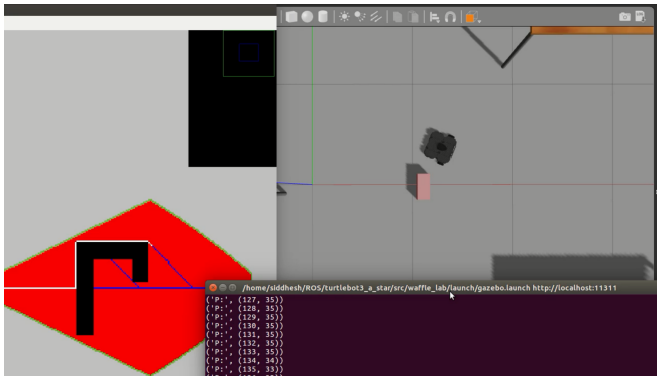

Fig. 2. Static Obstacle avoidance in Gazebo

To implement D* Lite in Gazebo we have used the turtlebot3 waffle. The environment we used was the RRL lab environment. D* Lite algorithm generated the trajectory for the robot to follow. We implemented a Proportional closed loop controller which sends linear and angular velocity commands on the cmdvel topic. When the robot is traversing n the trajectory it planned earlier it discovers the obstacle as shown in theo figure 2. It re-plans the path but discovers the same obstacle again since its quite big. The robot then re-plans again and then starts moving towards the goal. You can also see the snippet of turtlebot3 avoiding the obstacle in the figure 2, which is a cardboard box, and moving towards the goal.

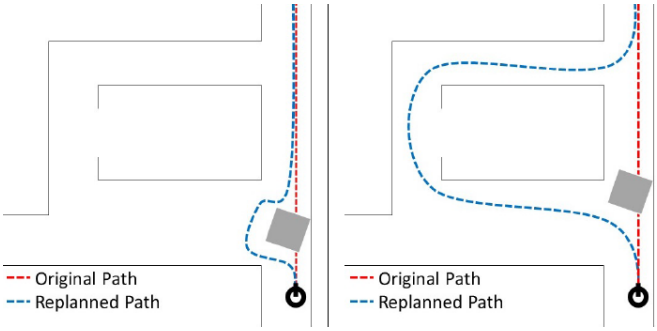

Fig. 3. Replanning [Image source:[4]]

2) D Lite with Human Assistance Simulation:* [4] has introduced an extension to the D* Lite algorithm in case the re-planned paths computed by the D* Lite algorithm as explained in the previous section, are too long. When the robot is moving to its next states changes might occur in the environment which are detected by the laser sensor scans of the robot. If it is detected any change on edge costs of the map (line 35), the algorithm updates the parameter km (line 36), and sets again s_last as been the current state s_start (line 37). The parameter km is used to calculate states keys and prioritize the OPEN list U.

For all edges with changed costs, the procedure UpdateVertex() is called to update states potentially affected by the changes as well as their membership in the priority queue U (lines 38-40). The variable p'_{size} receives the size of the current path (line 41), calculated by the procedure CalculatePathSize() (lines 21-26) which computes the path from the current state to the goal state and returns its size.

Once again, the algorithm calls ComputeShortestPath() to replan the path given the changes detected (line 42). The variable psize receives the size of the path after replanning (line 43). The algorithm proceeds by comparing the size of both paths, original and replanned (line 44).

If the difference is less than the predefined threshold min deviation, the robot executes the replanned path as the deviation can be done quickly.This scenario is depicted by 3 (a). The original path, dashed in red, is blockled by a gray box. The replanned path, dashed in blue, shows the deviation necessary to avoid the object. As this is a small deviation, the robot decides to perform the replanned path rather than waiting for human assistance.

But as you can see for the 3 (b) the re-planned path is much longer then the previously planned path and it requires a deviation which is much longer and to the other side. Also there might be a condition wherein the introduction of the obstacle in the environment has completely blocked the path of the robot. In such cases the robot notifies the human.

In our case the robot will notify us on its output terminal. However, it can not wait indefinitely because maybe there is nobody around to help. Thus, we compute a timeout based on the difference between both paths and the average speed of the robot (line 46). During this time, if anyone comes to clean at least part of the original path, the robot can reach the destination faster than if it takes the replanned path.

C. Real-world Implementation

To test the concept on real robot it was first necessary to create a setup. The figure 4 shows our setup wherein all the objects are static obstacles or Long term featureLTF.

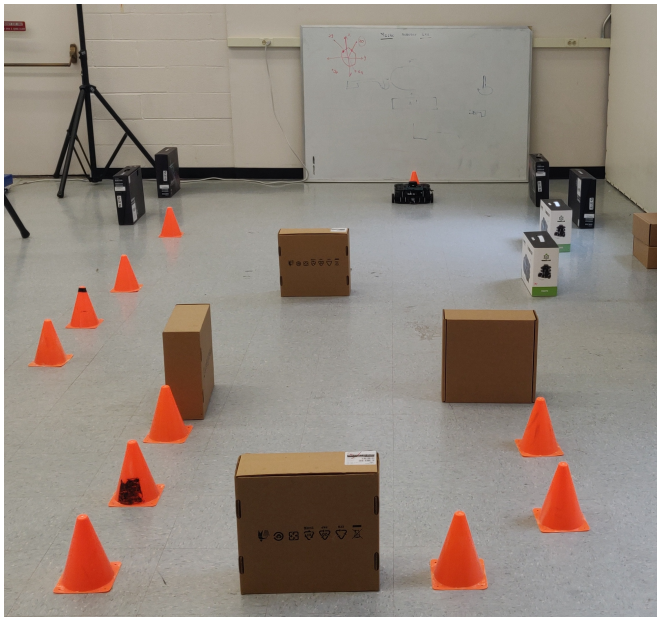

Fig. 4. Our Setup

1) Map creation using Simultaneous Localization and Mapping: To traverse in this setup we would first require a map which we created using SLAM algorithm. SLAM has been carried out using the gMapping package available for ROS Kinetic.Further is an overview of the methodology used in this package.

- gMapping uses Laser Scan Matcher (LSM) which is a form of visual odomtery that is calculated by the help of matching consecutive laser scans. The result from this visual odometry is fed into the mapping algorithm for faster convergence. With the help of the LSM the gMapping algorithm selects optimum initial start point.
- Further gMapping uses RBPF (Rao-Blackwellized Particle Filter) which internally uses grid based mapping with multiple particles. Every particle stores its own belief of the robot's previous positions and constructs its own map. Further every new laser iteration will update the belief of this particle. These beliefs consist of the position and orientation of the robot.

gMapping algorithm [9] is as follows:

1) Measurement: Obtain a new laser scan
2) Scan Matching: In this stage the previous and current scans are matched.
3) Sampling: Representation of the new particles $x_t^{(i)}$ are calculated from the distribution $\pi(x_t|z_{1:t}, u_{0:t})$ using previous particles $x_{t-1}^{(i)}$.
4) Weighting: Every particle is then weighted based on (1).

$$w^{(i)} = \frac{p(x_t^{(i)}|z_{1:t}, u_{0:t})}{\pi(x_t^{(i)}|z_{1:t}u_{0:t})} \tag{3}$$

5) Re-sampling: Then better particles with higher weights replace the particles with lower weights.
6) Mapping: Map $m_t^{(i)}$ is calculated based on every sample $x_t^{(i)}$ and all observations $p(m_t^{(i)}|x_{1:t}, u_{0:t})$

gMapping further uses the occupancy grid metric model to generate occupancy grid maps which allows fusion of different sensors. The occupancy here is defined as the ratio the total number of times the grid is seen full to the total of number of times the grid is seen by the sensor.

$$p(x, y) = \frac{\#occupied}{\#occupied + \#empty} \tag{4}$$

The map created of the setup is shown in figure 5.

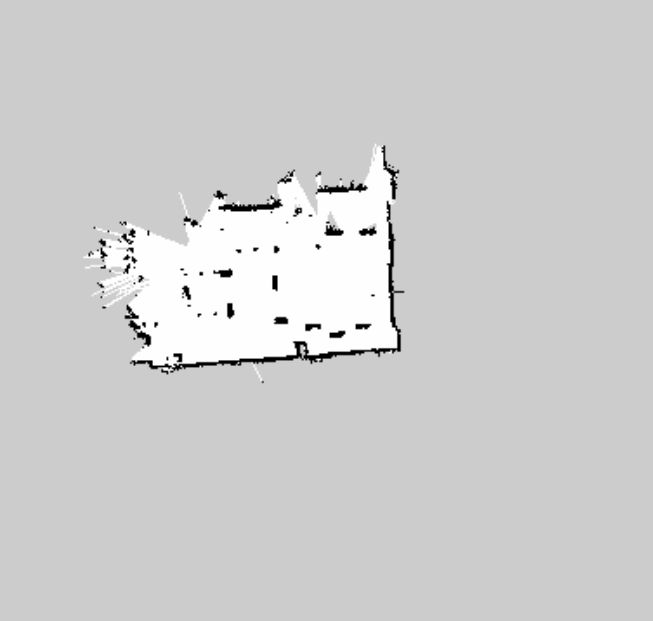

Fig. 5. Map created using gMapping

2) Localization: For the robot to know its source we need to know its location in the map. For that we have used the Adaptive Monte Carlo localization package in ROS. Adaptive Monte Carlo localization is a variant of Monte Carlo localization which is improved by sampling the particles in an adaptive manner based on an error estimate using the KullbackLeibler divergence (KLD). Initially it uses a large M due to the need to cover the entire map with a uniformly random distribution of particles. However, when the particles have converged around the same location, maintaining such a large sample size is computationally wasteful. [12]

KLD sampling is a variant of Monte Carlo Localization where at each iteration, a sample size M_x is calculated. The sample size M_x is calculated such that, with probability $1 - \delta$, the error between the true posterior and the sample-based approximation is less than ϵ. The variables δ and ϵ are fixed parameters [11].

When the robot is placed in the setup we need to give the 2D pose estimate initially so that AMCL converges faster. Then we teleop the robot for few seconds till the AMCL particles converge and then the pose is passed on as source to the D* Lite Algorithm. The figure 6

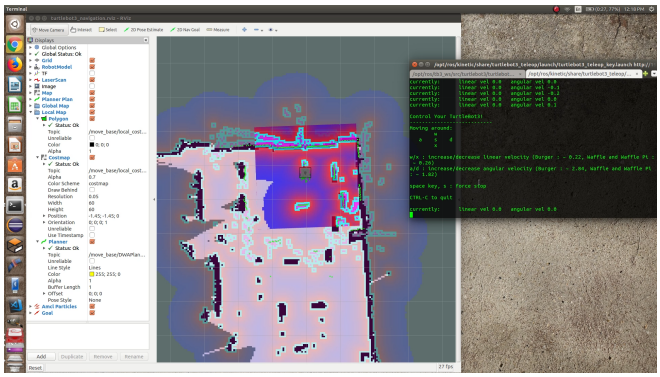

Fig. 6. Localization using AMCL

3) Obstacle Detection:

- Turtlebot3 waffle comes with the 2D laser distance scanner (LDS) capable of sensing 360 degree.
- This sensor publishes distance readings for each angle on the scan topic.
- For the purpose of obstacle detection, we use the minowski clearance and transfer the lidar ranges to the neighbours in terms of robot coordinates.
- When an obstacle is detected below 32cm radius the robot stops its forward motion and calls the re-planning module to generate new path.

4) Obstacle Placement: For the demonstration the robot is given a goal which is straight in front of it about 2 meter distance from it. When the robot was travelling we suddenly place a box of cardboard in front of it. The figure 7 shows the placement of obstacle by one of the authors.

Fig. 7. Placement of Short term feature

5) Obstacle Avoidance: The robot then detects the obstacle and marks the nodes in obstacle space as described in section before. The robot then re-plans the path to the goal and moves on the new trajectory. The figure 8

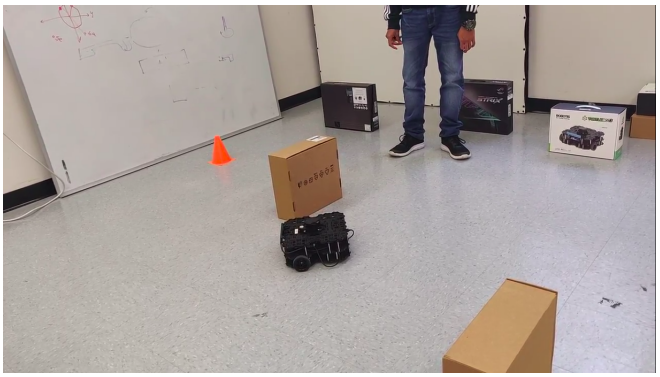

Fig. 8. Placement of Short term feature

III. Reference Paper Results

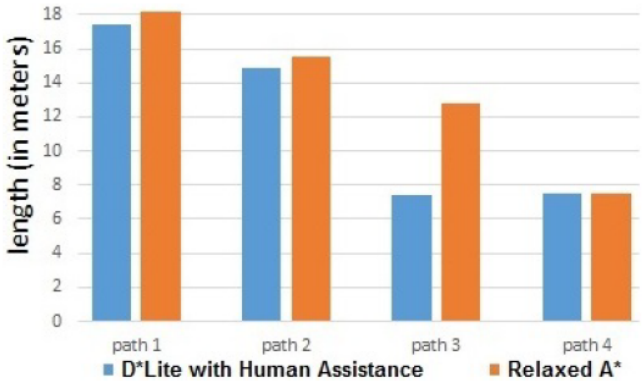

Fig. 9. Comparison of D* Lite and Relaxed A* for different path lengths

R. Alves et. al. implemented D* Lite with Human Assistance [4] in virtual and real environments. They generated different environments with different grid sizes, like 10x10, 20x20, 30x30 cells. The start and goal nodes were provided with different scenarios. Two algorithms were used to generate paths: A* and D* Lite. After the path generation and execution of path, they introduced new obstacles in to observe the generation of new paths. The D* Lite algorithm rapidly generates new path but A* takes time because it starts the search process from scratch. The OPEN list is a queue maintained by the search algorithms, which maintains states to be explored during the search. To test the performance of both the algorithms, R. Alves et. al. compared the count of states stored in the OPEN lists while the algorithms were replanning. Fig.10 shows the number of states in OPEN list plotted against the number of iterations during the search. As we can clearly observe, the number of states in D* Lite are quite less compared to A*, meaning A* requires more time compared to D* Lite for replanning. A* needs to explore as

many states as necessary to find the optimal path, which is computationally more expensive, which is not desired in real world applications.

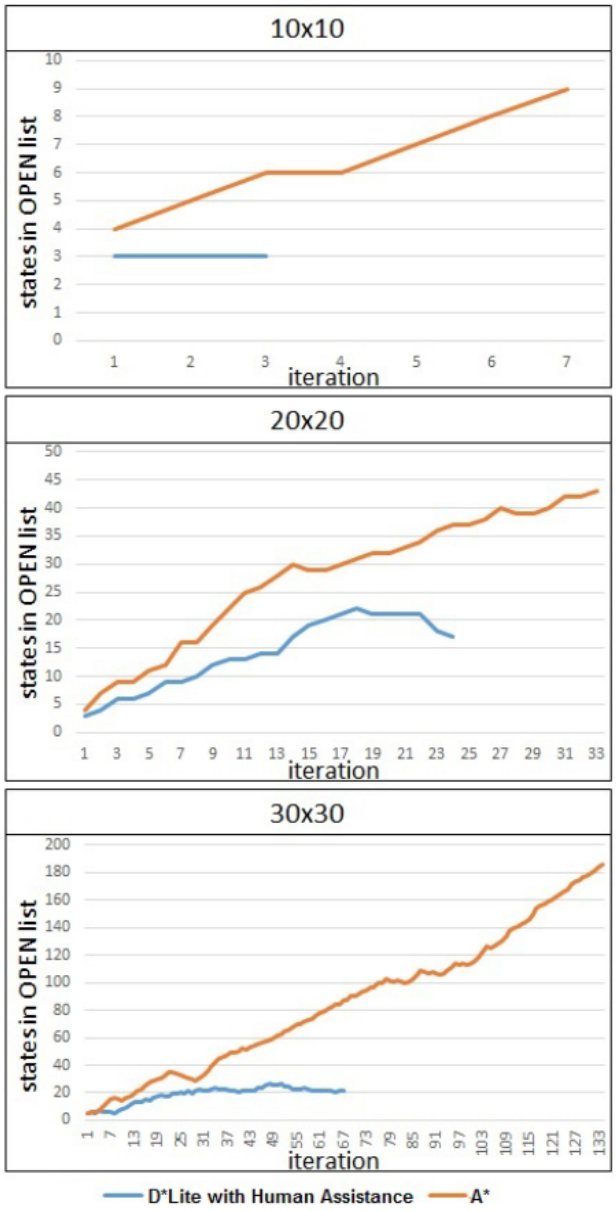

Fig. 10. OPEN List vs number of iterations for different grid sizes

Another study done by R. Alves et. al. shows the evaluation of performance of Relaxed A* vs D* Lite with Human Assistance. The Relaxed A* algorithm doesn't need to keep a tab on all the possible paths to goal state, but only stores the best path based on the cost function. The team implemented both algorithms on Robot Operating System (ROS) on a Turtlebot in Gazebo. The fig.9 shows that the optimal path lengths for different goal states and the results show that for smaller paths, the path lengths is comparable but for longer distances, D* Lite provides a shorter path.

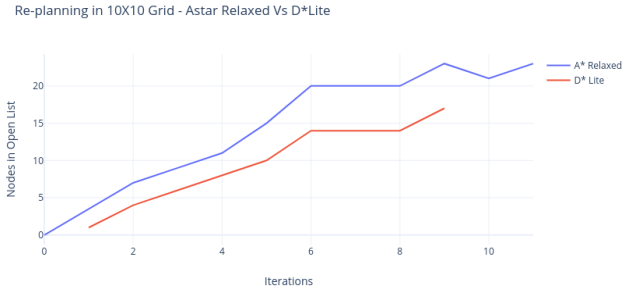

Fig. 11. States in open list for D* Lite and Relaxed A* against the number of iterations for 10x10 grid

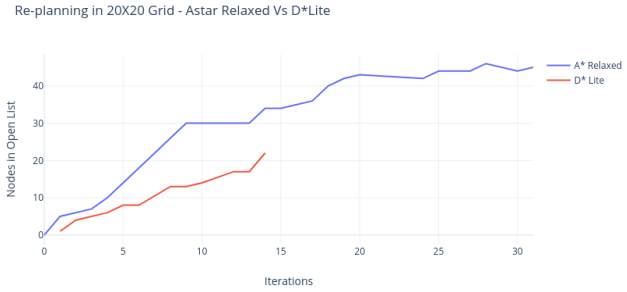

Fig. 12. States in open list for D* Lite and Relaxed A* against the number of iterations for 20x20 grid

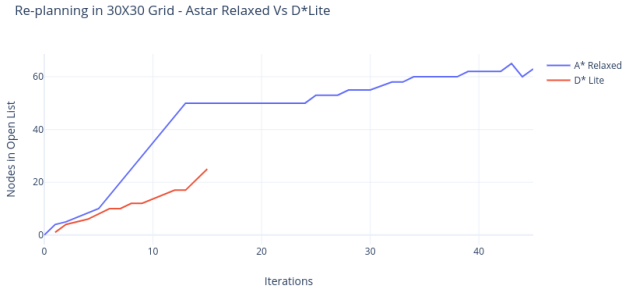

Fig. 13. States in open list for D* Lite and Relaxed A* against the number of iterations for 30x30 grid

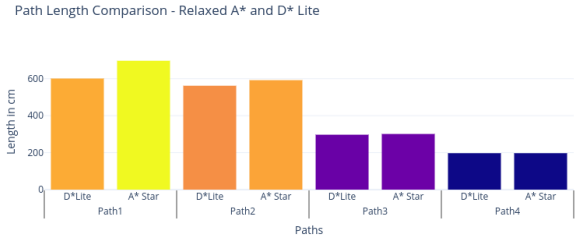

Fig. 14. Comparision for path lengths for different goals for Relaxed A* and D* Lite

IV. RESULTS

The figures 11, 12, 13 are the plots of the number of open states versus the number of iterations during re-planning for Relaxed A* and D* Lite algorithms, for different grid sizes. We can conclude that the number of states in open list for Relaxed A* is more than the that of D* Lite. The reason being Relaxed A* is not an incremental search algorithm, so it starts the re-planning from scratch, hence it needs to explore a lot of new nodes. But in case of incremental search algorithms like D* Lite, the re-planning uses previous search data and updates the nodes that are now in obstacle space and reconnects to the previous path without much exploration.
The figure 14 shows the different path lengths for different goal nodes in case of Relaxed A* and D* Lite algorithms. The path lengths are comparable when the goal node is close-by, but the difference is significant when the goal node is far away from the start node.

V. CONCLUSIONS

Robots functioning in environments with humans working around need to adjust to the constant changes in their workspace. The traditional path planning algorithms do not account for these changes and hence we need to go for a sophisticated algorithm such as D* Lite, which is an incremental hueristic search algorithm.D* Lite reuses the past search data to find the re-planned path, which is more efficient. The results show the performance of D* Lite against Relaxed A* and we can clearly observe that the D* Lite produces the re-planned path much faster than its counterpart.

By adding the extra feature of human assistance, the robots which use D* Lite for their planning can save time to reach the goal by avoiding longer re-planned paths. Human assistance also further help the robot in going to regions where the robot can't reach by any other means and removing the obstacle towards such regions is the only way. As a future scope we can certainly do obstacle avoidance for dynamically moving obstacles and re-plan on the run.

VI. ACKNOWLEDGMENT

We would like to thank Professor Reza Monfaredi and Teaching Assistants Ashwin Goyal and Utsav Patel for there constant support and guidance. We are grateful for having him as a mentor and for helping us in every aspect of this project right from project selection to its Python and Gazebo simulation. They have provided useful incites on improving performance of the project and in making it more robust.

REFERENCES

[1] Z. Teresa, History of service robots, in Concepts, Methodologies, Tools, and Applications. IGI Global, 2014, pp. 114. [Online]. Available: http://dx.doi.org/10.4018/978-1-4666-4607-0.ch001

[2] N. J. N. P. E. Hart and B. Raphael, A formal basis for the heuristic determination of minimum cost paths, IEEE Transactions on Systems, Science, and Cybernetics, vol. SSC-4, no. 2, pp. 100107, 1968

[3] E. W. Dijkstra, A note on two problems in connexion with graphs, NUMERISCHE MATHEMATIK, vol. 1, no. 1, pp. 269271, 1959

[4] Raulcezar Alves, Josue Silva de Morais, Carlos Roberto Lopes, Indoor Navigation with Human Assistance for Service Robots using D*Lite, 2018 IEEE International Conference on Systems, Man, and Cybernetics, 10.1109/SMC.2018.00696.

[5] S. Koenig and M. Likhachev, D*lite, in Eighteenth National Conference on Artificial Intelligence. Menlo Park, CA, USA: American Association for Artificial Intelligence, 2002, pp. 476483. [Online]. Available: http://dl.acm.org/citation.cfm?id=777092.777167

[6] D. T. Wooden, Graph-based path planning for mobile robots, Ph.D. dissertation, Georgia Institute of Technology, 2006.

[7] Tugcem Oral and Faruk Polat, "MOD* Lite: An Incremental Path Planning Algorithm Taking Care of Multiple Objectives", IEEE TRANSACTIONS ON CYBERNETICS, VOL. 46, NO. 1, January 2016.

[8] J. Kauko, and V.-V Mattila, Mobile Games Pathfinding, Proceedings of the 12th Finnish Artificial Intelligence Conference STeP 2006, pp 176 182, Helsinki, Finland

[9] E. Uslu, F. akmak, M. Balclar, A. Aknc, M. F. Amasyal and S. Yavuz, "Implementation of frontier-based exploration algorithm for an autonomous robot," 2015 International Symposium on Innovations in Intelligent SysTems and Applications (INISTA), Madrid, 2015, pp. 1-7

[10] Imen Chaari, Anis Koubaa, Hachemi Bennaceur, Adel Ammar,Maram Alajlan, Habib Youssef, "Design and performance analysis of global path planning techniques for autonomous mobile robots in grid environments", International Journal of Advanced Robotic Systems, March-April 2017: 115 ,10.1177/1729881416663663

[11] Sebastian Thrun, Wolfram Burgard, Dieter Fox. Probabilistic Robotics MIT Press, 2005. Ch. 8.3 ISBN 9780262201629

[12] Sebastian Thrun, Dieter Fox, Wolfram Burgard, Frank Dellaert. "Robust monte carlo localization for mobile robots." Artificial Intelligence 128.1 2001, 99-141