# A Comparative Study of D* Lite, A*, and Dijkstra Algorithms for Dynamic Obstacle Avoidance

Phanindrateja Thammi
*Department of Mechanical Engineering*
*Texas A&M University*
College Station, TX, USA
phanith12@tamu.edu

*Abstract*— ***In the world of robotics, Path Planning is the most important aspect of robot navigation. There were several researchers have done and algorithms were proposed such as Depth first search, Breadth first search, A*, Dijkstra etc., The most popular among these are A* and Dijkstra algorithms. Incremental heuristic search methods use heuristics to focus their search and reuse information from previous searches to find solutions to series of similar search tasks much faster than is possible by solving each search task from scratch. However, all these algorithms fail in a dynamic partially known environment, they recalculate the path from scratch whenever there is a change in the environment. This process can be computationally costly in real world applications. In this Paper an attempt is made to study & implement the D*- Lite Algorithm for autonomous navigation avoiding dynamic obstacles considering a unique cost equation giving importance in the form of the weighting factors to time & safety while replanning the path and compare it with the existing algorithms like A* & Dijkstra.***

***Keywords—D*-Lite, Autonomous path Planning, ROS, Obstacle Avoidance***

## I. Introduction

There are several robots in the market designed to assist humans in various tasks indoor and outdoor [1]. Taking help from the robots increase the efficiency and reduce the time to complete the tasks for humans. To perform all these tasks, robots require an efficient navigation system which leads to a path planning problem. This planner should also consider the dynamic obstacles in real time and generate a new path accordingly. Generally, we leverage the power of the A*[2] or the Dijkstra [3] algorithms for path generation using a static map obtained initially. But these algorithms don't consider updated maps when a obstacle is introduce while the robot is navigating. Hence, the static maps become invalid. There are two ways to solve this issue would be to provide the updated maps to the algorithms like A* or Dijkstra and get the updated path or leverage an algorithm which can replan and fix the path more efficiently to avoid the dynamic obstacles detected.[4]. One such algorithm is D* which is a more dynamic version of the A* which prioritizes computational efficiency over optimality in dynamic environments. D* replans the path using heuristics and reuses the information from the previous search much faster than creating maps again from scratch. The applications of D* and its simpler version D* Lite is very popular in mobile and autonomous robots, including the Mars Rovers [5].

Path planning algorithms play an important role in navigating settings and are classified into four types [6]: 1) off-line algorithms, 2) on-line algorithms, 3) incremental algorithms, and 4) soft-computing algorithms. Off-line algorithms create solutions prior to execution, whereas online algorithms merge the planning and execution phases, forcing agents to plan and execute each movement repeatedly. Off-line planning algorithms encounter execution time issues in dynamic or unpredictable contexts, whilst on-line algorithms may produce sub-optimal path length solutions. Incremental algorithms, which bridge the gap between off-line and on-line approaches, offer improved execution speed and optimal solutions by reusing past data to reduce iterations. Soft computing is often used in multi-objective shortest path discovery. This paper [7] presents the basic algorithm of implementing the D*-Lite algorithm, provides its properties and demonstrates experimentally the advantages of combining incremental and heuristic search for the applications studied. There are various versions of the D* algorithms as it has the wide use cases in different domains such as videogaming, warfare [8], unknown terrain navigation. In this modern world, research has been done even integrating robots to the blockchain providing them power to do monetary transactions as humans. [9]

## II. Motivation

Consider a scenario in which a goal-directed robot is traversing through unknown terrain. The robot constantly evaluates the traversability of its eight nearby cells, with each traversable cell having a movement cost of one. Starting from the start cell, the robot attempts to reach the goal cell by consistently computing the shortest path, assuming that cells with unknown blockage status are also traversable.

The computed path is followed by the robot until it either successfully achieves the goal or meets an impassable cell. D* Lite effectively addresses this difficulty by recognizing and recalculating just those objective distances that have changed or were not previously computed, focusing on the required modifications for recalculating the shortest path. It recalculates the shortest path from its present cell to the target in the latter scenario.

Robots are generally expensive, and everyone wants their robot to be most efficient and avoid any type of damage to the robot. In such cases, ideally everyone wants their robot to maintain a certain distance from the obstacle while maneuvering. Considering this aspect of safety and still reach the goal in the least time possible, in this paper, I have introduced an unique cost equation giving percentage of importance for robot safety and the time to reach the goal.

## III. PROBLEM STATEMENT

The problem statement that is given as a part of the course is:
"1. Navigate an autonomous robot, *TurtleBot* in the *Gazebo* simulated environment from a start point to the goal point avoiding dynamic obstacles, additionally change the cost equation while replanning the path to maintain more distance from the obstacle while navigation for the safety of the robot.
2. Compare the out result with implementing the A* and Dijkstra algorithms in the same *Gazebo* environment on *TurtleBot*.
3. Provide critiques and views after implementation in the simulation.
4. The reference for doing this project is from here.

## IV. APPROACH

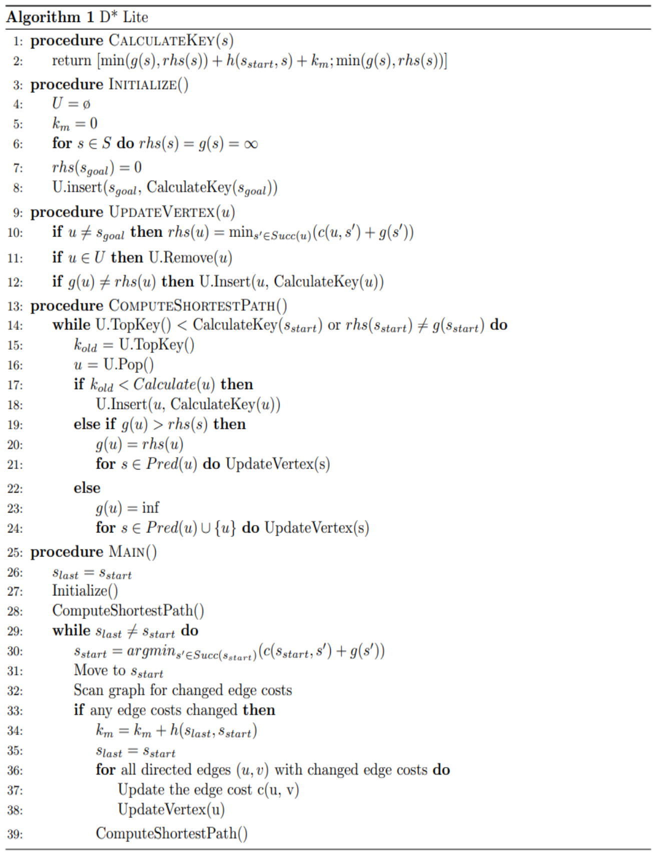
This section will discuss our approach towards achieving the goal discussed in the previous section. We will start by discussing the algorithm and simulations using it. We will move towards the implementation in *Gazebo* simulation environment.

The D* Lite algorithm is a Lifelong Planning A* algorithm-derived incremental heuristic search algorithm. Within this algorithm, the parent node is known as the 'successor' and its children are known as the 'predecessors.' Edges are transferred from predecessors to successors. Two cost parameters, g(X) and rhs(X), characterize all nodes. G(X) denotes the objective function value, also known as the g-cost, representing the distance from the start for a given node X. Meanwhile, rhs(X) is the one-step look-ahead cost value calculated by adding the g-cost (g(S)) of its successor to the path cost (c(X, S)) between them.

$$Rhs(X) = g(S) + c(X, S) \qquad (1)$$

The algorithm begins its exploration at the provided goal node and returns to the start node by minimizing the rhs value. Notably, the driving force behind this algorithm is the concept of 'local inconsistency.'

Figure 1



**Algorithm 1** D* Lite

```
procedure CALCULATEKEY($s$)
    return $[\min(g(s), rhs(s)) + h(s_{start}, s) + k_m; \min(g(s), rhs(s))]$
procedure INITIALIZE()
    $U = \emptyset$
    $k_m = 0$
    for $s \in S$ do $rhs(s) = g(s) = \infty$
    $rhs(s_{goal}) = 0$
    U.insert($s_{goal}$, CalculateKey($s_{goal}$))
procedure UPDATEVERTEX($u$)
    if $u \neq s_{goal}$ then $rhs(u) = \min_{s' \in Succ(u)}(c(u, s') + g(s'))$
    if $u \in U$ then U.Remove($u$)
    if $g(u) \neq rhs(u)$ then U.Insert($u$, CalculateKey($u$))
procedure COMPUTESHORTESTPATH()
    while U.TopKey() < CalculateKey($s_{start}$) or $rhs(s_{start}) \neq g(s_{start})$ do
        $k_{old}$ = U.TopKey()
        $u$ = U.Pop()
        if $k_{old} < Calculate(u)$ then
            U.Insert($u$, CalculateKey($u$))
        else if $g(u) > rhs(s)$ then
            $g(u) = rhs(u)$
            for $s \in Pred(u)$ do UpdateVertex(s)
        else
            $g(u) = \inf$
            for $s \in Pred(u) \cup \{u\}$ do UpdateVertex(s)
procedure MAIN()
    $s_{last} = s_{start}$
    Initialize()
    ComputeShortestPath()
    while $s_{last} \neq s_{start}$ do
        $s_{start} = argmin_{s' \in Succ(s_{start})}(c(s_{start}, s') + g(s'))$
        Move to $s_{start}$
        Scan graph for changed edge costs
        if any edge costs changed then
            $k_m = k_m + h(s_{last}, s_{start})$
            $s_{last} = s_{start}$
            for all directed edges $(u, v)$ with changed edge costs do
                Update the edge cost c(u, v)
                UpdateVertex(u)
            ComputeShortestPath()
```

When g(X) = rhs(X), a node is considered locally consistent and if g(X) ≠ rhs(X), it is considered locally inconsistent. Unlike traditional path planning algorithms, an open list (priority queue) is kept only for nodes that exhibit inconsistency. Based on the values of g-cost and rhs, two types of inconsistencies, there are two types of inconsistencies:

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

An under-consistent path signifies an increase in the cost of the path to a node. This circumstance arises when a node, once situated in free space, becomes obstructed by an obstacle, making it unreachable. Similarly, an over-consistent path indicates a reduction in the cost of the path to a node. This can occur when a node, previously obstructed by obstacles, is now free.

This algorithm employs a parameter known as the 'key' to organize the open list (priority queue). The key or priority of a node X in the open list is determined as the minimum value between g(X) and rhs(X) added to a focusing heuristic h(X). The primary key is derived from the minimum of g(X) and rhs(X), while the second term serves as the secondary tie-breaking key.

$$Key = [\min(g(X); rhs(X)) + h(X); \min(g(X), rhs(X))]$$

### *A. Unique Cost Equation formulation:*

While replanning the path, the algorithm should consider the safety of the robot and consider the least time to reach the goal.

Inorder to accommodate this criterion, I have modified the cost equation while replanning as below.

$$c(X,S) = \alpha * Time + \beta * Safety$$

where Time is the least time possible to reach the goal and Safety is the distance to the nearest obstacle or the amount of clearance around the robot. Higher this number, the robot maintains more distance around the obstacles.
$\alpha$ and $\beta$ are weighting factors

Figure 1 shows the pseudo code of the algorithm that is implemented in this project. Initially, a key is generated as explained above. Then an empty priority list 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. Since, goal node is inconsistent as $rhs(goal) \neq g(goal)$, it is added to the priority queue U. Function Compute ShortestPath() (lines 13-24) 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 9-12) 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 the obstacle is detected during the path traversal, a new path is calculated considering the unique cost equation, the robot then navigates to the goal maneuvering around the obstacle to maintain certain distance.

## B. *Robot Kinematics:*

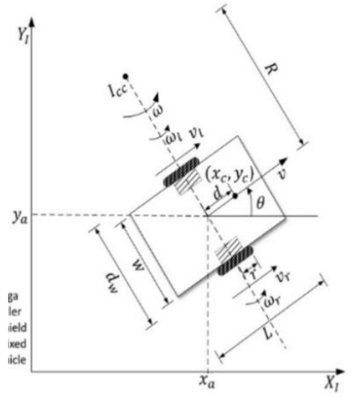


Figure 2

The State variables considered in the system are given below.
$X$ is the x-coordinate of the TurtleBot's position.
$Y$ is the y-coordinate of the TurtleBot's position.
Ɵ is the Orientation of the TurtleBot's in radians.
Input variables considered are given below.
V is the Linear velocity of the robot.
$\dot{\omega}$ is the Angular velocity of the robot.

Action space for the TurtleBot will rollout to be [0, 1], where zero represents to stop and the 1 represents the motion of the TurtleBot.

The Robot's position should be within the boundaries of the environment created in the simulation. This is considered as the robot's state space constraints. Figure 3 shows the TurtleBot model-Waffle used in the simulation.

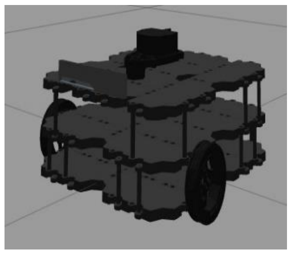


Figure 3

## C. *Robot Operating System (ROS)*

In this modern world, all the Autonomous robots perform complex tasks, but it requires navigation from point A to point B to finish the task. ROS framework facilities to practically implement the planned navigation and establish the communication between the simulation environment and the real testing robotics agent. ROS is widely used in the robotics community and has strong support from open-source robotics developers. Also, the ease of interfacing with various types of robots such as TurtleBot3 [10] Universal Robot [11] and the algorithm packages availability such as Gmapping Simultaneous Localization and Mapping (SLAM) [12] for Light Detection and Ranging (LiDAR) sensors motivated us to use ROS as the robot communication and control layer in this work. This framework has a unique way of communicating. It has nodes and topics. All the nodes communicate through the topics. There will be two types of nodes, publisher nodes and the subscriber nodes. If you want to give a command to the robot such as provide velocity, you can create a node, create a publisher node and send the msg to a /cmd_vel topic then the robot runs as per the instructions. If you want to read data such as odometer reading from the robot, you can create a subscriber node and subscribe to the /odom topic, and you will get the robot's pose.

## D. *Simulation in Gazebo & Rviz*

Initially, I built a custom lab environment in the Gazebo and created the launch files to spawn a TurtleBot Waffle robot into the simulation environment. Then mapped the entire space running the SLAM algorithm and saved the environment file in the world folder as a pgm file. This file will be used to provide the robot when spawned in the Rviz tool. Figure 4 shows the complete Gazebo environment of the lab and highlights the obstacles and TurtleBot. In the navigation, I have provided the position constraints to the TurtleBot which means it should not navigate outside the boundary, in this simulation, the robot's position constraints are the walls and should avoid the obstacles while navigating from given start point to the goal point.

Figure 5 shows the pgm file obtained after performing the SLAM algorithm to map the Gazebo simulation environment. The boundary regions, wall or the obstacles are generally colored black. And the grey color part is the region where the TurtleBot can freely navigate.

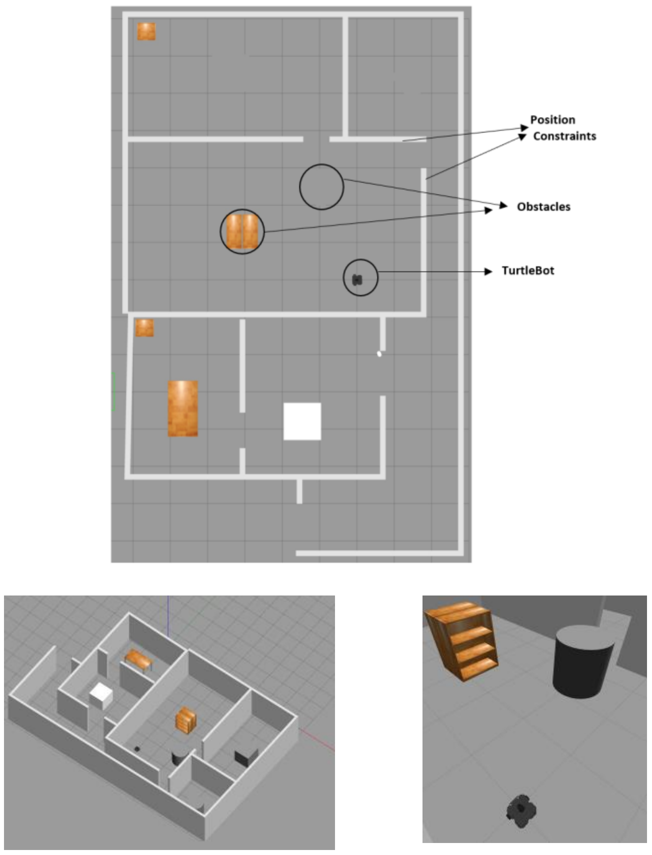


Figure 4

Given the fact that we know that the TurtleBot consists of a LIDAR which we will use while navigation to detect the obstacles. Below Figure 6 shows how the LIDAR of the TurtleBot can see, the information of the region that is shaded in the grey in map whether there is an obstacle or not is known to TurtleBot and the Grey region can be seen only upto certain extent due to the limitations of the range of LIDAR. We can always make changes to the LIDAR range of visibility.

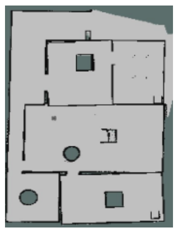


Figure 5

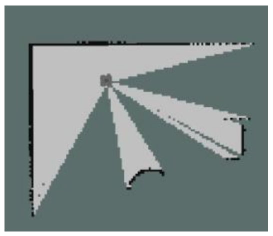
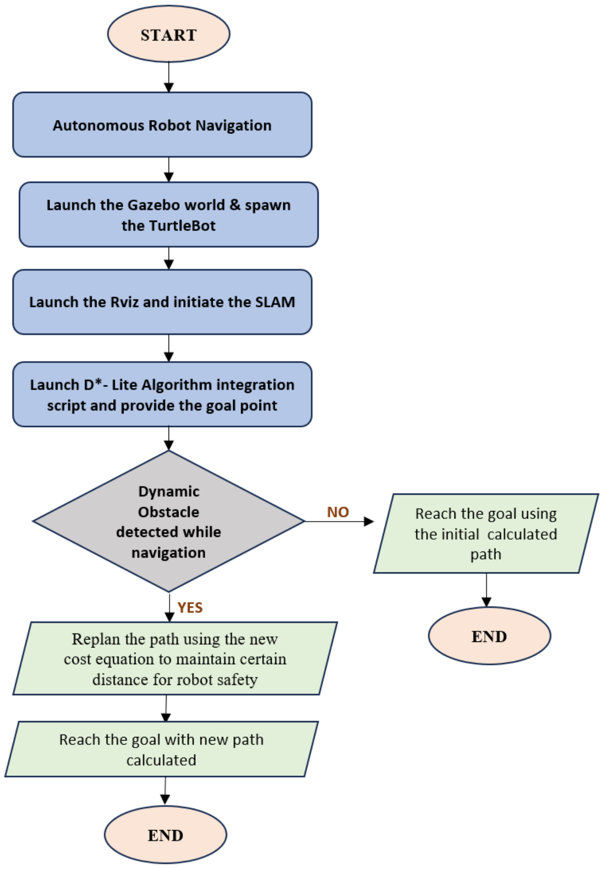


Figure 6

## E. *Algorithm FlowChart*


START
Autonomous Robot Navigation
Launch the Gazebo world & spawn the TurtleBot
Launch the Rviz and initiate the SLAM
Launch D*- Lite Algorithm integration script and provide the goal point
Dynamic Obstacle detected while navigation
NO
Reach the goal using the initial calculated path
END
YES
Replan the path using the new cost equation to maintain certain distance for robot safety
Reach the goal with new path calculated
END

## F. *Implimentation*

For implementation of the concept in the simulation, one might follow the instructions below.

1) Initially launch the world file with the appropriate command ‘roslaunch ros_world Turtlebot3_world_RL.launch’

2) Next, the Turtlebot is spawned in the Rviz world, this world supports for robot navigation. With the commnad *roslaunch global_path_planning turtlebot3_world_RL.launch*. Figure 5 pops up when the above command is executed.

3) The real SLAM which enables the LIDAR in the robot is launched by the follwing command.
*roslaunch turtlebot3_slam turtlebot3_gmapping_RL.launch* *Figure 6 will be launched when the above command is executed.*

4) Now, the algorithm, a python file will be launcheed using the command below.
“Python3 path_planning.py”.

5) After the server status got updated in the Rviz terminal which was launched in the 2nd step, we can provide the goal point to the turtleBot by clicking at the “2D_nav_goal”.

6) After providing the goal, the robot calculates the shortest path to reach the goal. Once the path is calculated, the tobot travels to the goal with the specified constant velocity and reduce velocity while manuevering.

7) Figure 7 shows the robot navigating to the goal point using the path calculated by D*-Lite algorithm when there are no obstacles.

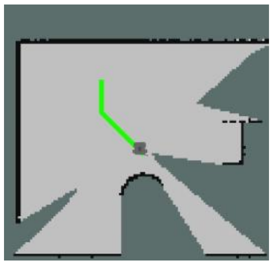


Figure 7

## V. Results

For the simulation setup mentioned in above sections, the start and goal points were provided with different scenarios. Total of three algorithms, A*, Dijkstra and D*-Lite were used to generate the paths and for comparing results. The results show that the D*lite algorithm generates the path more rapidly whereas the other two algorithms will replan the path by search process from scratch. Figure 8 shows comparison result between the three algorithms with respect to the number of nodes present in the open list while replanning. We can see that the number of nodes are much more in case of both the algorithms except D*-Lite. The reason behind this is that other

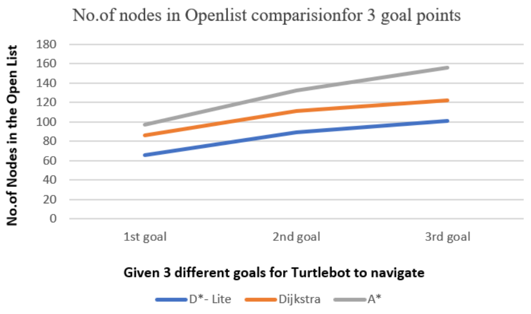


Figure 8

Two algorithms are not incremental search algorithms.so they start the replanning from scratch which results in exploring new nodes. Unlike the other two algorithms, in D*-Lite algorithm, the re-planning uses the previous saved search data and updates only the nodes which are in the obstacle space and reconnects the previous path without exploration.

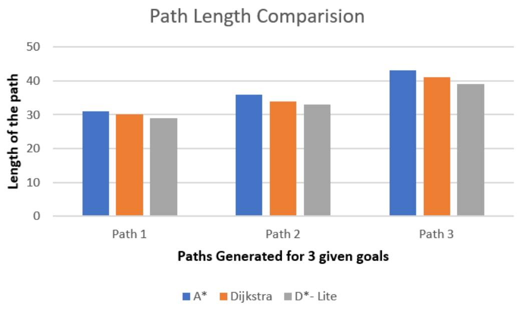


Figure 9

Figure 9 shows the comparison of algorithms based on the length of the path produced for three different goal points. The path is almost the same when the goal node is given close to the robot's start but there is a significant difference when the goal is given far away from the start.

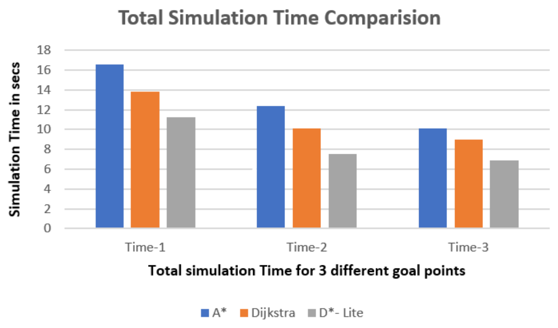


Figure 10

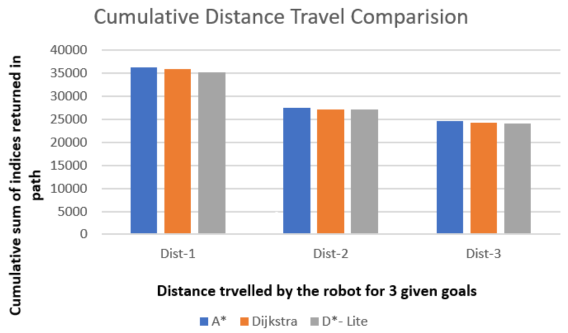


Figure 11

Figure 10 illustrates that D* Lite will complete the computation in way less than the other algorithms during the replanning. Figure 11 shows the cumulative addition of indices that are returned in the path. Sum will be almost the same as if we provide the updated map to the A* & Dijkstra, calculates the same path but the time is the term which differentiate algorithms efficiency.

## VI. Conclusion

The traditional path planning algorithms doesn't account for the dynamic obstacles whereas the D*-lite algorithm resolve this issue of path planning by replanning the path from the searched space. The results show the performance of the D*-Lite algorithm against the A*, Dijkstra. We can clearly see that D*-Lite algorithm replans the path much faster than the other two algorithms. By adding the unique cost equation while replanning, the robot maintains a certain mentioned threshold distance from the obstacle while maneuvering. The reference paper clearly defines the algorithm for the replanning but in this paper, I have introduced the new cost equation for robot safety. For the future scope we can do obstacle avoidance for dynamically moving obstacles.

## VII. 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.ch00.

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

[6] 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.

[7] Koenig, S., & Likhachev, M. (2005). Fast replanning for navigation in unknown terrain. *IEEE Transactions on Robotics*, *21*(3), 354–363.

[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] S. Mallikarachchi, B. Ho, I. Kanj, O. Seneviratne, and I. Godage, “Decentralized data collection via swarm contracts: Study on crowdsourced google maps,” in IEEE International Conference on Conference on Control, Automation and Robotics (ICCAR). IEEE, 2023, p. accepted.

[10] turtlebot3 Developers. turtlebot3. [Online]. Available: https://emanual. robotis.com/

[11] Universal Robots GmbH. universal-robots. [Online]. Available: https: //www.universal-robots.com/

[12] X. Zhang, J. Lai, D. Xu, H. Li, and M. Fu, “2d lidar-based slam and path planning for indoor rescue using mobile robots,” Journal of Advanced Transportation, vol. 2020, 2020.

[13] https://github.com/srane96/D-Star-Lite-Implementation-on-Turtlebot3-Waffle-/tree/master