Comparative Analysis of Lifelong Planning D* Lite

Zuoan He
Arizona State University
Tempe, AZ, USA
zuoanhe@asu.edu

Wenzhe Zheng
Arizona State University
Tempe, AZ, USA
wzheng41@asu.edu

Mahidhar Reddy Dwarampudi
Arizona State University
Tempe, AZ, USA
mdwaram1@asu.edu

Apoorva Verma
Arizona State University
Tempe, AZ, USA
averma63@asu.edu

***Abstract*—Developing search algorithms has always been an integral part of learning Artificial Intelligence. Typically A* search is used to solve path finding problems in a known environment. But most of the times, an autonomous agent will find itself in an unknown environment, which is when the performance of A* search is not as good. In this paper, we study and analyse D* lite algorithm [1], which is also an incremental version of the A* search, to explore and find the shortest path when in such an environment. We provide with a comparative analysis of D* lite in the pacman domain with other search algorithms like DFS, BFS, UCS and LPA* based on the criteria such as path cost, nodes explored, time taken, etc. which shows that D* lite either outperforms or is atleast as good as the other algorithms, even with a limited view of its environment.**

***Index Terms*—D*Lite, Shortest path, Lifelong Planning A* Search, Pacman World, rhs-values, g-values, locally inconsistent**

I. Introduction

Mobile robot-based applications have increasingly penetrated all aspects of life, and path planning is one of the essential requirements in this aspect. In recent years, with the growth of large-scale applications of mobile robots, the general search algorithms for finding paths from a start state to the goal state no longer meet the requirements in an unknown environment. The agent can only constantly adjust its plan through available location information. However, the importance of this problem has generated a significant amount of work over the past few years. Mobile robot path planning is to find a collision-free path that is optimal in time, distance, or space from the starting position to the target position in a given environment with unknown obstacles. This paper is a detailed study of an incremental search algorithm, D* Lite. Incremental search methods build upon the desired shortest path by reusing the information from their previous searches, thereby guaranteeing faster response than starting the whole search process from scratch.

The problem with using the existing uninformed search algorithms like Breadth First Search (BFS), Depth First Search (DFS), Uniform Cost Search (UCS), or informed search strategies like A* search or Greedy search is that they have been developed to account for the fact that the agent has complete information of its environment and the obstacles present in it. D* Lite was designed to incorporate the fact that the agent does not have comprehensive knowledge about its environment in most real-world problems and often ends up replanning based on its previous findings of the domain.

On a high level, in D* Lite works by maintaining a priority queue with heuristic information, similar to A* search, such that the most promising vertices are expanded first. The priority of each node in the A* search is calculated based on the following cost function,

$$f(n) = g(n) + h(n)$$

where g(n) is the path cost from the start node to the current node, and h(n) is the heuristic cost from the current node to the goal node. We are going to use a modified version of this cost function in D* Lite to assign priorities to the nodes in the queue. The priority of a node in D* Lite is determined by a key-value tuple (k_1(s), k_2(s)), where k_1(s) and k_2(s) corresponds to the f-value and g-value in the A* search respectively. The next section gives a detailed view of how to compute these key-values. An important feature of the D* lite algorithm is that it searches for the path in the reverse direction, i.e., from the goal node to the start node. This is done in order to conserve the key-values and only the nodes relevant to the replanning task.

In the following sections, we will go through the technical details about the implementation of D* Lite, a comparison of running it in the Pacman domain with other search techniques, along with a detailed discussion of the results obtained. And finally, an analysis of our findings with the help of the student's t-test.

II. TECHNICAL APPROACH

A. D* Lite Overview

The D* lite algorithm was developed to extend the Lifelong planning A* (LPA*) algorithm for applications that require path replanning. The main idea of D* lite is to compute the shortest path from the current node to the goal node, thereby making the current node the new start node at every step. Thus, in D* lite, the path is calculated in the reverse direction, i.e., from the goal node to the start node, and the agent only has a limited field of view.

Pseudocode for the D* lite algorithm works as follows:

1) Initialise all nodes as unexpanded.
2) Best first search until start node is consistent with neighbours and expanded.
3) Move to the next best vertex.
4) If any edge costs change :
   - Track how heuristics have changed.
   - Update source nodes of changed edges.

5) Repeat from step 2.

In order to understand the D* lite algorithm implementation in depth, let us first introduce some important terms.

- S is a set of all the nodes in the graph.
- $Succ(s) \subseteq S$ is a set of all the successors of a node $s \in S$.
- $Pred(s) \subseteq S$ is a set of all the predecessors of a node $s \in S$.
- g(s) is the distance from the start node to the current node s.
- rhs(s) is the minimum value of the set consisting of the sum of g-value of node's parents s' and the cost from each parent node s' to node s.
- c(s',s) is the cost of moving from node s' to node s.
- Key modifier k_m to limit the reordering of the priority queue.

According to the D* lite algorithm, a node within the graph can be in three different states :

- Case 1: Locally consistent, g(s) == rhs(s)
- Case 2: Locally overconsistent, $g(s) > rhs(s)$
- Case 3: Locally underconsistent, $g(s) < rhs(s)$

Case 2 and 3 above are a result of edge cost changes in the graph. For case 2, when g(s) is greater than rhs(s), we update g(s) to the new path cost, which is lower than the current g-value, and propagate this update to all its predecessors. For case 3, we set g(s) to infinity and again propagate this update to all its predecessors. So we maintain a priority queue to store all the locally inconsistent nodes that will remain there until their next best rhs(s) cost value is found. The priority of a node is dependent on a k-value tuple which is defined as follows.

$$k(s) = \begin{bmatrix} k1 \\ k2 \end{bmatrix} = \begin{bmatrix} min(g(s), rhs(s)) + h(s_{start}, s) + k_m \\ min(g(s), rhs(s)) \end{bmatrix}$$

And the rhs(s) updates are based on the following criteria:

$$rhs(s) = \begin{cases} 0 & \text{if } s = s_{goal} \\ min_{s' \in Succ(s)}(g(s') + c(s', s)) & \text{otherwise} \end{cases}$$

B. Problem Setup and Environment:

The D* lite algorithm has been implemented in the search.py file. We make use of the Pacman world domain to test our approach. The heuristic used to calculate the cost is Manhattan distance. The obstacles to the Pacman agent are the walls in the domain, which are unknown to it until it bumps into one. And that is when replanning takes place.
All the essential methods used in the algorithm are as follows:

1. CalculateKey() : This method returns a priority value to be assigned to the nodes for ordering in the priority queue.
2. Initialise() : It initialises all the several components in the algorithm. The priority queue is set up with the goal node added to it. rhs and g-values for all the nodes is set to infinity, and $rhs(s_{goal})$ is set to 0.
3. UpdateVertex() : This method calculates the rhs-values for the nodes, checks for local inconsistency by comparing rhs(s) with g(s), and adds or removes the node to/from the priority queue if found inconsistent or consistent, respectively.
4. ComputeShortestPath() : This method computes the shortest path from the top node in the priority queue to the start node with the help of CalculateKey() and UpdateVertex() methods and propagates the updated rhs and g-values to the node neighbours.

C. Implementation and Optimization

Several code-level optimizations have been applied to the D* Lite algorithm. Although, we haven't made any changes to the global workflow of D* Lite as described in [1].

1) PriorityQueue: The way priorityQueue worked was slightly tweaked, so that update function will both update the node, if it is present or add it, if it is not present. Similarly, which is used to get the top key. Instead of popping the element and pushing it back again, we simply access the first element of the index. Finally, remove function, updates the node to put is in the top and pop's it, instead of searching and deleting it.
2) UpdateVertex(): Most of the functionality related to updating a node, changing rhs, ghs values and updating the node position in priority queue is put on UpdateVertex. Here one major change is instead of always checking is the node is in the priorityQueue, we update the node's rhs value and pop remove it out the Queue and only add it back, if ghs and rhs value is not equal, this reduces the number of comparisons done on this function.
3) ComputeShortestPath(): Whenever there is an overestimate, instead of removing and the node and adding it back to the queue, it is simply updated, priority queue takes care of these changes efficiently.
4) Driver Code: Some minor changes have been made in the driver code to efficiently adapt the actual D* lite to our scenario. Instead of updating the state costs for all actions, we only update when the cost between 2 nodes is less than the old cost. D * lite only tracks the states where it is expanding, we are storing this sequence of states in a global variable. After the search terminates, we compared every pair of consecutive states and determined the action, that pacman was supposed to take. This step added significant overhead, especially, when the maze was large. Here we leveraged the base code provided and stored the actions simultaneously, along with states. Finally, changes were made to track the walls, the getSuccessors function was modified in such a way that it returned walls when required (for DFS, BFS and UCS, etc.) and didn't return the walls for D* lite. Whenever we hit a wall, we recompute the path after the wall is added to a list of obstacles, with a very high cost, infinity in our case, so that Pacman never chooses this path.

III. RESULTS AND DISCUSSION

In order to compare the advantages and disadvantages of these algorithms, we measured the algorithms from three different dimensions: No. of nodes explored, Runtime and Pathlength. Runtime refers to the total time required by the Pacman agent to find beans under the test algorithm. Path-length refers to the length of the total path taken by the Pacman to find beans. Nodes Explored refers to the number of nodes expanded in the path while finding the beans. At the same time, to measure the completeness of our comparison metrics, the researcher also chose five different layouts of mazes to test, namely: TinyMaze, SmallMaze, Medium Maze and BigMaze. The size, complexity and scale of these mazes are different, which can better reflect the difference in behavior and performance between these algorithms. Because our project mainly focused on the difference between D* lite and LPA*, we will use paired Student's T-test to statically analyze our result. For the Student's T-test, our null hypothesis is D* lite performs the same as LPA* in our test environments and the alternative hypothesis is that D* lite performs better than LPA* in our test environments. We will collect samples from multiple runs of these algorithms with different sizes and complexity of environments, then calculate the P-value and compare with the commonly used significance level of 0.05 to decide whether to reject the null hypothesis.

a. No. of nodes explored

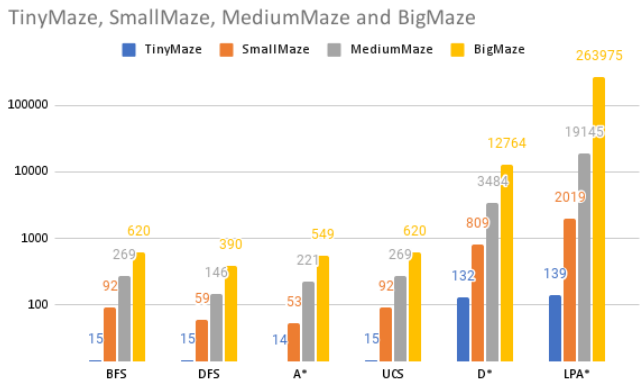

Fig. 1: No. of nodes explored(Log values)

We can see from figure 1 that the rankings of Nodes explored in different algorithms from good to bad are: DFS, A*, BFS, UCS, D* and LPA*. When facing simple mazes, D* and LPA* are not much different from other algorithms in terms of nodes expanded. But when facing larger/more complex mazes, D* and LPA* nodes begin to grow significantly. This is because these two algorithms are accustomed to re-planning owing to the change in edge costs and newly discovered information about obstacles in the domain.

	D* Lite	LPA*
Mean	10969.61905	518918.9048
Variance	114646622.6	888732147277
Observations	21	21
df	20	
t Stat	-2.492257566	
P(T¡=t) one-tail	0.01079365461	
t Critical one-tail	1.724718243	

TABLE I
STUDENT'S T-TEST FOR NODE EXPLORED; T-TEST: PAIRED TWO SAMPLE FOR MEANS

By applying paired Student's T-test on nodes explored between D* lite and LPA* (figure 2), we got P-value of 0.01079 (Table I) which is smaller than the significance level of 0.05. Hence, we can reject the null hypothesis of D* lite's performance similar to LPA* and conclude that D* lite performs better compared to LPA* in terms of nodes explored with an increasing complexity of the environment.

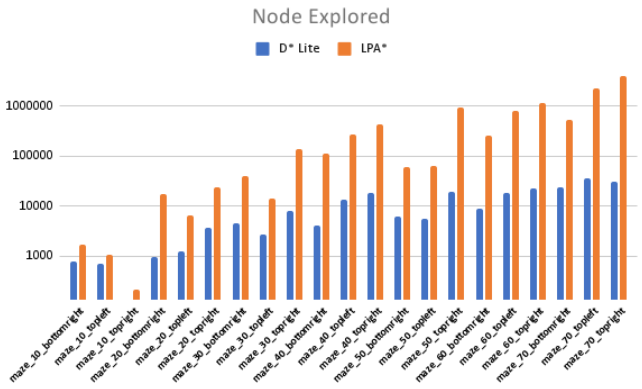

Fig. 2: Node Explored D* Lite vs. LPA*(Log value)

Another observation that we noticed while running the two algorithms is that the Pacman agent hits more obstacles near the goal node in LPA* contrary to D* Lite in which it hits more obstacles near the start node. This is because the two algorithms start their traversal from opposite directions (D* Lite from the goal state and LPA* from the start state).

b. Running Time

Search	TinyMaze	SmallMaze	MediumMaze	BigMaze
BFS	0.002250	0.0114320	0.022340	0.042632
DFS	0.008135	0.006536	0.025913	0.036426
A*	0.002588	0.013089	0.036486	0.041766
UCS	0.003546	0.018893	0.027859	0.048962
D*	0.001680	0.017308	0.075947	0.559965
LPA*	0.007784	0.037578	0.319673	5.274131

TABLE II
RUNTIME

We can see from figure 3 and Table II that the least to longest running time is: DFS, A*, BFS, UCS, D* and LPA*. But we also need to take this into consideration the fact that the no. of nodes explored generated by D* Lite and LPA* in the big maze is very large (due to changing edge costs). Based on this observation, we can see that the search time of

D* Lite in an unknown environment is comparable enough to that of other search strategies in a known environment. We can also see from the search time of D* in TinyMaze that it only took 0.001680 seconds, and the time of LPA* increased significantly as the maze increased compared with D* Lite.

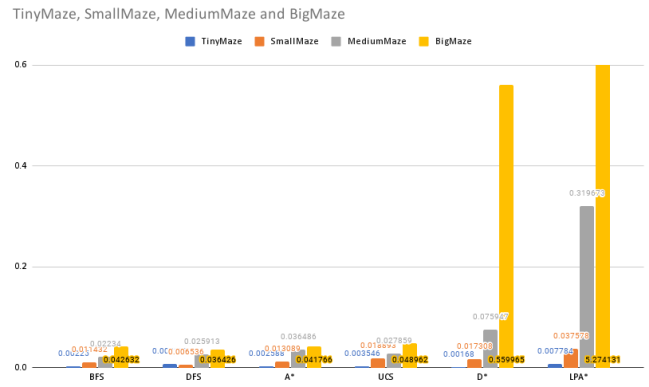

Fig. 3: Runtime

By applying paired Student's T-test on the running time between D* lite and LPA*(figure 4), we got P-value of 0.02028(Table III) which is smaller than the significance level of 0.05. Hence, we can reject the null hypothesis that D* lite performs similar to LPA* and conclude that D* lite is faster in terms of running time when compared to LPA* in increasing complexity of environment.

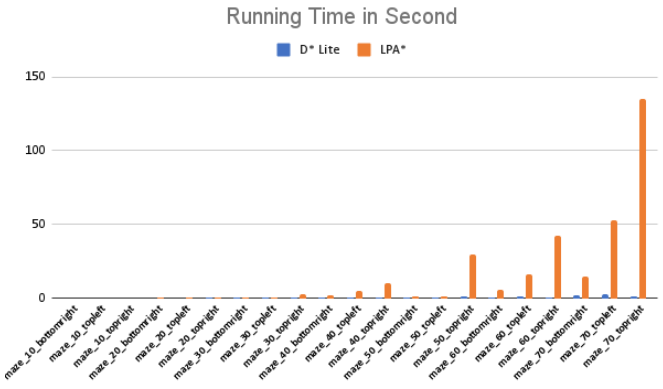

Fig. 4: Running Time D* Lite vs. LPA*

	D* Lite	LPA*
Mean	0.5985638095	15.28305514
Variance	0.5988288304	970.6062454
Observations	21	21
df	20	
t Stat	-2.189923272	
P(T$_i$=t) one-tail	0.02027601165	
t Critical one-tail	1.724718243	

TABLE III
STUDENT'S T-TEST FOR RUNNING TIME; T-TEST: PAIRED TWO SAMPLE FOR MEANS

c. PathLength

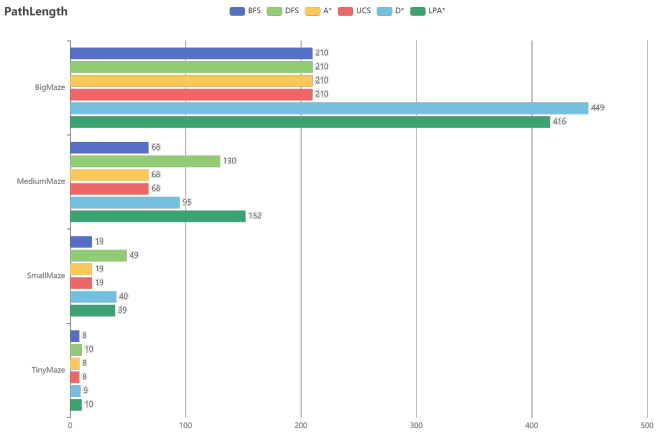

Fig. 5: PathLength

Search	TinyMaze	SmallMaze	MediumMaze	BigMaze
BFS	8	19	68	210
DFS	10	49	130	210
A*	8	19	68	210
UCS	8	19	68	210
D*	9	40	95	449
LPA*	10	39	152	416

TABLE IV
PATH LENGTH

In Figure 5, the arrangement of Pathlength from minimum to maximum is: BFS, A*, UCS, D*, DFS, LPA*. Among them, the Pathlength of the first four search algorithms in complex mazes is the same, but for D* and LPA*, the path length for big maze is significantly higher than the others. An explanation for this is that because we are running them in a partially observable environment, they use backtracking for the purpose of replanning.

IV. CONCLUSION

Based on our observations from section 3, we can safely conclude that D* lite search in a partially observable environment is as efficient as all the other search techniques in a fully observable environment. Even though D* lite is an extension over the LPA* algorithm, the reason for a reduction in the performance metrics of the latter in more complex environments is because it always replans from the start node with changing path costs, whereas D* lite uses its current location as the start location while replanning. This saves D* lite quite a lot of computations, contrary to LPA* search. And as the size of the environment increases, the number of nodes expanded in LPA* will increase, which is why it takes longer to execute in bigger environments. Hence, of all the algorithms discussed in this paper, D* lite should be suitable for solving goal-directed robot navigation tasks in an unknown terrain that requires constant replanning.

REFERENCES

[1] Sven Koenig, Maxim Likhachev, "D* Lite," .AAAI-02