Life-Long Planning

CSE 571 – Artificial Intelligence

Deepthi Reddy Obulareddy Gari
Arizona State University
Tempe, Arizona
dobulare@asu.edu

Harish Paul Thavisi
Arizona State University
Tempe, Arizona
hthavisi@asu.edu

Krishna Sree Gottumukkala
Arizona State University
Tempe, Arizona
kgottumu@asu.edu

Sreshta Chowdary Kampally
Arizona State University
Tempe, Arizona
skampall@asu.edu

***Abstract* – In a known or unknown environment, a robot must face many challenges and maximize the utility to reach the goal. Finding path to reach goal is one task where it uses the robot's knowledge of environment. For the robot to act fast in the partially observable environment, incremental heuristic search technique is useful as it utilizes the previous knowledge rather than starting the search from the start. We have implemented the Lifelong planning A* (D* lite) algorithm for the agent to track the history in an unknown environment and use it to reach the goal. The D* Lite algorithm is designed to use the existing knowledge by reducing the overhead and increasing utility which helps the robot to avoid performing the repetitive actions. In this project, we verify the results of D* Lite by using the cumulative path search in the pacman problem and compare the results with other baseline methods such as A*, BFS, DFS by performing the test analysis in different environments.**

***Keywords* –** *D* Lite, BFS, DFS, A*, Incremental Heuristic, Partially Observable Environment.*

I. INTRODUCTION

In the scenario that a task environment is provided, the primary objective of an intelligent agent would be to figure out the compact path beginning from a start location to the goal location. However, the agent faces an obstacle in the form of the lack of information it has regarding the environment. With respect to the pacman problem, an assumption has been made that the agent is in a moderately observable environment and is cognizant of the following: a) the local environment which comprises its current location and information corresponding to its four neighbors in all the directions, b) start location, goal location, and bounds of the terrain, c) memorizes the past observations and computations in the brain. The end goal of the agent is to be able to figure out the path as and when it observes information that is new in the domain and to also save that information for future use in turn steering clear of redundant calculations. The inspiration is to maximize the effectiveness of the agent by minimizing computation times and cost overheads. We take a glance at informed search algorithms such as A*[2] which utilize heuristics to enhance the search and deep dive into the implementation of A*[2] and its variations to examine the performance.

When we consider an agent navigation task in a terrain that is unknown, the agent begins at the start location and further moves to the end location (goal). On the basis of present knowledge of obstacles in the local environment, it determines the shortest path from its present location to its goal location. The path is then followed till the goal location is attained, or an obstacle could come across in its new local environment, during which the shortest path is recalculated from its present location to the goal.

We have other search algorithms like Replanning based A*, UCS, BFS, DFS and D* Lite which are different in their approach and performance. The end objective of our project corresponds to the application of these algorithms for the pacman problem, to figure out the shortest path beginning at the start location to the goal. The backdrop of the above-mentioned algorithms and the domain surrounding the pacman will be discussed in Section II of this report. The implementation of these algorithms will be further talked through in Section III of this report along with its application to the pacman problem. The evaluation of these algorithms' performance will be then talked through in Section IV of this report. Last but not the least, Section V and Section VI correspond to the conclusion and discussion, and team contributions respectively.

II. TECHNICAL APPROACH / METHODOLOGY

A. *Re-planning Search Algorithm*

The Re-planning search algorithm helps to re-route the path from current state to goal state when it encounters an unexpected obstacle in a partially observable environment. This replanning can be implemented along with different baseline search algorithms, namely Breadth First Search (BFS), Depth First Search (DFS), Uniform Cost Search (UCS) and A* Search. These baseline methods usually calculate the path from current state to goal state using various different mechanisms to get a path with minimum cost. For instance, the strategy of Breadth First Search (BFS) is to explore all children of a node expanding any new nodes and it stops when it reaches the *goal* state. While A* search selects the path that gives the minimum value of *g(nextNode)* + *h(nextNode)*, here *g(d)* is the minimum cost of the path from *start node* to node *d*, and *h(d)* is a heuristic function which calculates the minimum cost of the path from node *d* to *goal* state. Similarly DFS and UCS have different mechanisms to compute the minimum cost path to the *goal* state.

Whereas, in Re-planning search algorithm the agent chooses the best route using a baseline algorithm, say using A* search, and proceeds along this path until the goal is reached. But it stops if it encounters any obstacle in its way which is previously unknown. Now, the agent restarts the procedure and runs the baseline search algorithm, here A* search again. This next search process does not have any prior knowledge of the computations done previously. This results in a substantial re-computation.

B. D Lite Search Algorithm*

D* Lite is an incremental search algorithm. It is a combination of both LPA* and D* algorithms and was written and implemented by Sven Koenig and Maxim Likhachev. Generally, in the above Re-planning algorithm we saw that all the information except the path is discarded once we replan the path when encountered by an obstacle. However, D* Lite keeps all this information and calculates some extra information. So that, when we see some new obstacle, we can update our path instead of recalculating it entirely. D* Lite can efficiently recalculate the shortest path to the goal by updating only those nodes in which a change is observed and are relevant to recalculate the shortest path to the goal.

D* Lite is similar to LPA* but it differs in a way that LPA* algorithm searches from the start vertex to the goal vertex, that is, *g(s)* here denotes the estimated distance from *start* node to *s*. Whereas, D* Lite algorithm searches from the goal vertex to the start vertex, that is, *g(s)* here denotes the estimated distance from g*oal* node to *s*. D*Lite updates nodes along the route in the process of reaching from current node to the goal state, in contrast to traditional Lifelong Planning A*. In order to re-plan, D* Lite additionally keeps a priority offset k_m. Every time the start value is updated, the key modifier—which is initially set to 0—increases.

III. Implementation Details

We implemented D* Lite algorithm to compare its performance with the results obtained by running the Replanning algorithm on BFS, DFS, UCS and A* respectively. These algorithms are run in partially observable environments, and hence initially the agent has knowledge of only the boundaries of the environment, the start state, and the goal state. As we run the algorithms the agent traverses the path, updates and stores any new information that it observed in the environment along the path, that is, in case it encounters any obstacle that is previously unknown, it updates the information it has on the environment.

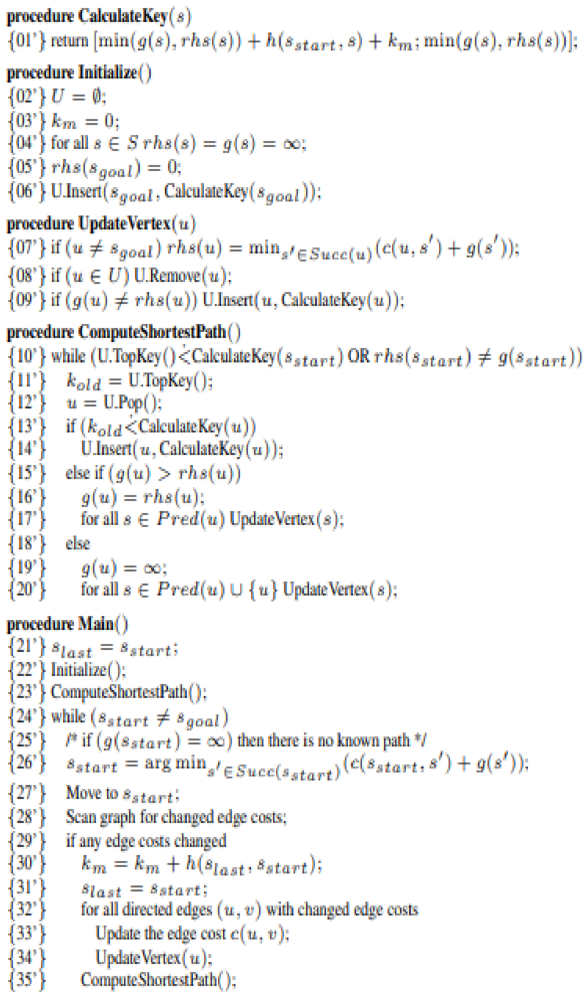
A. Re-planning Search Algorithm

This algorithm takes a baseline search algorithm function as an input along with the heuristic and the problem. It initially runs the baseline search algorithm to get a least cost path from start point to the goal. It then follows along this path while updating the information it can get on its surroundings along this path. Here we check if there are any walls surrounding the current point and store this information. If the agent comes across any obstacle, it stops and the agent re-plans the entire path using the baseline search function from scratch, but this time it has some extra information about the path and hence we get a different path to the goal. This process continues until we reach the goal.

We can see in the above implementation that the agent discards all the other information except for the path. Therefore, it does not utilize any of calculations we calculated previously and ends up recalculating the paths from the same points multiple times. Which takes more time, especially when the environment we have is big.

B. D Lite Search Algorithm*

The drawback in the above Re-planning Algorithm is resolved using D* Lite. This is an incremental search algorithm which when encountered by an obstacle, updates only those nodes whose edge costs are changed and are required to recalculate the least cost path. The implementation of the D* Lite algorithm is done using to the pseudocode shown in Figure 1.



```
procedure CalculateKey($s$)
{01'} return [$\min(g(s), rhs(s)) + h(s_{start}, s) + k_m; \min(g(s), rhs(s))$];
procedure Initialize()
{02'} $U = \emptyset$;
{03'} $k_m = 0$;
{04'} for all $s \in S$ $rhs(s) = g(s) = \infty$;
{05'} $rhs(s_{goal}) = 0$;
{06'} U.Insert($s_{goal}$, CalculateKey($s_{goal}$));
procedure UpdateVertex($u$)
{07'} if ($u \neq s_{goal}$) $rhs(u) = \min_{s' \in Succ(u)}(c(u, s') + g(s'))$;
{08'} if ($u \in U$) U.Remove($u$);
{09'} if ($g(u) \neq rhs(u)$) U.Insert($u$, CalculateKey($u$));
procedure ComputeShortestPath()
{10'} while (U.TopKey() $\dot{<}$ CalculateKey($s_{start}$) OR $rhs(s_{start}) \neq g(s_{start})$)
{11'}    $k_{old}$ = U.TopKey();
{12'}    $u$ = U.Pop();
{13'}    if ($k_{old} \dot{<}$ CalculateKey($u$))
{14'}       U.Insert($u$, CalculateKey($u$));
{15'}    else if ($g(u) > rhs(u)$)
{16'}       $g(u) = rhs(u)$;
{17'}       for all $s \in Pred(u)$ UpdateVertex($s$);
{18'}    else
{19'}       $g(u) = \infty$;
{20'}       for all $s \in Pred(u) \cup \{u\}$ UpdateVertex($s$);
procedure Main()
{21'} $s_{last} = s_{start}$;
{22'} Initialize();
{23'} ComputeShortestPath();
{24'} while ($s_{start} \neq s_{goal}$)
{25'}    /* if ($g(s_{start}) = \infty$) then there is no known path */
{26'}    $s_{start} = \arg\min_{s' \in Succ(s_{start})}(c(s_{start}, s') + g(s'))$;
{27'}    Move to $s_{start}$;
{28'}    Scan graph for changed edge costs;
{29'}    if any edge costs changed
{30'}       $k_m = k_m + h(s_{last}, s_{start})$;
{31'}       $s_{last} = s_{start}$;
{32'}       for all directed edges $(u, v)$ with changed edge costs
{33'}          Update the edge cost $c(u, v)$;
{34'}          UpdateVertex($u$);
{35'}       ComputeShortestPath();
```

Fig 1: D* Lite Algorithm

In the above pseudocode we can see that there are four functions we used for implementing D* Lite. Here, *g(s)* is the distance between node *s* and the *goal* state, *h(s,s')* is the heuristic which is an approximate of the distances between the vertices *s* and *s'*, *c(s,s')* is the cost of moving from *s* to *s'*. We define rhs(s) as follows:

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

A vertex is called locally consistent iff its g-value equals its rhs-value, otherwise it is called locally inconsistent.

The function CalculateKey(s) returns a key which is the priority of a vertex. This key is a vector with two components as shown in Figure 1. In the function Initialize() all vertices except the goal node have their *rhs(s)* and *g(s)* set to ∞. The goal node's *rhs* is zero (see Figure 1[lines 03'-05']). Goal state will be the only locally inconsistent vertex initially and is added to the empty priority queue shown in Figure 1[line 06'].

The function UpdateVertex(s) updates the rhs(s) and priority of s in the queue only if that vertex is affected by the changed edge costs. It also removes the node from the priority queue if it becomes locally consistent.

The function ComputeShortestPath() recurrently expands inconsistent vertices in the order of their priorities. It first eliminates the vertex with the lowest priority k_{old} = U.TopKey() (U.TopKey() returns the smallest priority of all vertices in priority queue U) from the priority queue as shown in Figure 1[line 12']. It now calculates its new priority using the CalculateKey() method. The deleted vertex is inserted back into the priority queue with the priority determined by CalculateKey() if k_{old} < CalculateKey(u). Else, if k_{old}< CalculateKey(u), check if the vertex is over-consistent (An inconsistent vertex is called over-consistent if g(s) > rhs(s) and under-consistent if g(s) < rhs(s)). If it is over-consistent, its g-value is to its rhs-value else if it is under-consistent, its g-value is set to infinity.

D*Lite's main method Main() first calls Initialize() to initialize the search problem. It then calls the ComputeShortestPath() function to continue down the path until an obstacle is found (Since the only state in the priority queue at this point is goal state). Now, only those edges whose edge costs have changed are updated, and it calls UpdateVertex() is on those vertices. Lastly, the least cost path is recalculated by calling ComputeShortestPath() function.

IV. RESULTS

We compared the nodes expanded, time to execute and score of the pacman game generated by the D* lite algorithm with various baseline algorithms implemented previously during the coursework. The previous baseline algorithms are modified to run with the replanning algorithm with an adapter function called – ReplanningSearch. This function enable us to compute the baseline metrics to retrieve for the BFS, UCS, A* and DFS algorithms.

Table 1 represents the results of D* lite and baseline algorithms prescribed above on various layouts for the Nodes expanded and the Score obtained on the game. Apart from the Mazes (Medium / Tiny / Big and Small) we have created custom layouts called custom2022 of size 20x22 and custom3035 with size 30x35. These two mazes are designed in such a way to showcase the power of D* lite algorithm and how it outperforms the other baseline algorithms we have discussed before.

Table 2 represents the results of D* lite and baseline algorithms prescribed above on various layouts for the Time it took to execute the algorithm which are measured in Milliseconds and the path cost of the pacman that it took to solve the game.

The results in the table 1 and 2 clearly notifies the strength and uniqueness in its computation for the large mazes using recomputation. As the nodes expanded for BigMaze using D* lite are 4199 which is around 20 times less compared to that of other base lines (BFS / DFS / UCS) and nearly 7 times lesser compared to that of the A* algorithm. Time of computation is atleast half of any other baseline methods. Also with increase in layout size and complexity D* lite tends to outperform other baseline algorithms.

Whereas the other baseline algorithms performs similar or better than D* lite in our tinyMaze layout and complexity as the algorithm that is designed for D* lite tends to linearly grow in computation time and nodes expanded with the increase of the layout size and complexity but the other algorithms grow exponentially in time taken and nodes expanded with the growth of layout sizes. This is the reason for the outperformance of other baseline algorithms in small layouts. The other factor that mainly contributes for the discrepancy of time in tinyMaze is because of the number of walls that is encountered is less in tinyMaze. The number of wall interactions is the major reason why all the replanning algorithms take the most of the time. As the algorithms do not maintain the state of the layout and the time of computation rockets up with the increase in walls encountered or complexity increase.

The interesting find in the table 1 is also the data about the Replanning UCS and BFS which provides the same results for both the Nodes expanded and the score obtained through the algorithm whereas the table 2 depicts the time taken is varying by a lot for the same algorithms. This is since the BFS algorithms provides us with the same results in the less time of computation as compared to the replanning UCS.

Also the Replanning DFS provides results of scores in negative as the results generated from this algorithm are repeating over the layout whenever the pacman encounters the wall. Due to this repetative behavior of the algorithm the time taken to reach the destination will be longer and the score results in negative in the large mazes. Whereas the time of the computation of the algorithm is very less compared to that of the BFS or UCS.

And the Replanning A* algorithm performs comparatively better to the other baseline algorithms in terms of time to compute and score. This is due to the algorithm is triggered less in terms of walls encounter which is since the algorithm for every wall interaction tries to change the approach of the pacman to the goal in a optimized approach with every new wall interaction. This makes this algorithm little better to the other baseline methods. But although the baseline algorithms except DFS takes different time to computes the score and cost of the pacman in the algorithms are same nearly. DFS takes the longest route and repetitive route as given in the previous paragraph for explanation. Hence the discrepancy for this algorithm.

T-test is an inferential statistic method used to find if there is any significant difference between two independent variables whose variances are unknown. The t-test scores provided in the table 3 are the calculated between the D* lite algorithm and each of the baseline algorithms (Replanning applied). Higher the T-value indicates the significant difference between the algorithms compared. From the results we can infer that there is a significant improvement in the nodes expanded and the time taken to reach the goal state whereas the score / path cost are slightly improved or not changed.

Layout Type	**Layout Size (w x h)**	**Nodes Expanded and the score by the different algorithms (nodes expanded, score)**									
		D* Lite		**Replanning A***		**Replanning UCS**		**Replanning BFS**		**Replanning DFS**	
		Nodes	**Score**	**Nodes**	**Score**	**Nodes**	**Score**	**Nodes**	**Score**	**Nodes**	**Score**
BigMaze	37 x 37	4199	70	29662	78	95884	70	95884	70	93822	-2894
Custom3035	30 x 35	1711	351	24315	351	64189	351	64189	351	15645	-659
MediumMaze	18 x 36	820	426	6269	426	15831	426	15831	426	6143	-762
Custom2022	20 x 22	875	404	5278	404	14170	404	14170	404	3001	50
SmallMaze	10 x 22	224	469	452	469	1609	469	1609	469	1023	301
TinyMaze	7 x 7	31	502	46	502	83	502	83	502	65	488

Table 1

Layout Type	**Layout Size (w x h)**	**Time taken and Path cost by the different algorithms (in milli seconds)**									
		D*Lite		**Replanning A***		**Replanning UCS**		**Replanning BFS**		**Replanning DFS**	
		Time	**Cost**	**Time**	**Cost**	**Time**	**Cost**	**Time**	**Cost**	**Time**	**Cost**
BigMaze	37 x 37	162.62	440	294.80	432	3831.53	440	838.00	440	690.00	3404
Custom3035	30 x 35	52.83	159	246.03	159	2249.48	159	566.62	159	152.71	1169
MediumMaze	18 x 36	25.58	84	64.14	84	586.82	84	136.71	84	52.44	1272
Custom2022	20 x 22	24.38	106	50.33	106	363.89	106	122.03	106	23.75	460
SmallMaze	10 x 22	7.16	41	5.23	41	30.67	41	13.84	41	8.10	209
TinyMaze	7 x 7	1.83	8	0.65	8	1.28	8	0.95	8	0.81	22

Table 2

T-test scores of the different baseline algorithms compared to that of the D* lite	**Nodes Expanded**	**Score**	**Path cost**	**Time**
A*	1.84999	0.01494	0.01494	1.12079
BFS	1.91935	0	0	1.64653
DFS	1.24553	1.85894	1.85894	0.97105
UCS	1.91935	0	0	1.79508

Table 3

V. Conclusion

We have tested D* Lite and baseline search algorithms (BFS, DFS, A*, UCS) with replanning on different layouts of varying terrain, size and complexity that includes the custom layouts designed by us.

We have executed our pacman on various layout sizes and complexities. The more complex the layout is the more walls or obstacles the pacman faces. We can conclude from the above results that with the increase in layout size and complexity or no of walls the pacman has come across, D*lite algorithm outperforms all the other baseline algorithms mainly in terms of the Number of nodes expanded and the time taken to compute the path.

As we know that the D* lite algorithm works with nodes in a different manner compared to the other baseline metrics where here it keeps track of the nodes visited and always make the informed decision before visiting a node but the other baseline methods recalculate the whole path whenever faced with an obstacle. The other baseline methods only can outperform the D* lite when the layout is small and least complex.

VI. References

[1] Sven Koenig and Maxim Likhachev. D* lite. Aaai/iaai, 15, 2022.
[2] T-Test Calculator for 2 Independent Means https://www.socscistatistics.com/tests/studentttest/
[3] T-Test: What It Is With Multiple Formulas and When To Use Them. Hayes. https://www.investopedia.com/terms/t/t-test.asp
[4] D*. https://en.wikipedia.org/wiki/D*
[5] D* pathfinding algorithm. Cybrosys. https://www.gamedev.net/forums/topic.asp?topic_id=452231&whichpage=1&#2988568