Life-Long Planning

***Abstract*—In this report, we are implementing A* lifelong planning and D* lite algorithm described in the paper: "D* Lite" by Sven Koenig and Maxim Likhachev, AAAI 2002. The Algorithms are implemented on pacman world domain. Assuming that the pacman knows size of the enviornment and can observe the local environment only. It is also assumed that the pacman is aware of its starting location and is also able to maintain the knowledge of the environment it has observed. After implementing the algorithms they were tested on different layouts which give us insights on the behavior of the pacman agent. The outputs of the 3 algorithms namely D* Lite, A* Life Long and A* baseline was compared based on the number of nodes expanded, Computation Time and the pacman score achieved. The results derived from the comparison took us to a conclusion that D* Lite is better than the other two algorithms in most of the aspects. Graphical representations showing the results gave us insights on the caveats of the algorithm when compared to A* baseline. This helped us conclude on why D* lite is practically preferred over others in practical use.**

***Index Terms*— locally inconsistent, rhs, g- value**

I. INTRODUCTION

In computer science a search algorithm is any algorithm which is used to solve the search problem namely to retrieve information which is stored within some data structure or calculated in a search space of a problem domain.[3] There are many methodologies to implement search algorithms and, in this project, we decided to compare the performance of 3 of them which are A* baseline, A* Life Long and D* Lite.

A* baseline is a type of informed search algorithm which is basically a combination of the cost to reach the nodes[g(n)] and the cost to reach the goal from the node. [h(n)][1]

$$\text{i.e. } f(n) = g(n) + h(n)$$

A* baseline search algorithm tries to minimize the total estimated cost of the solution. A* search is both complete and optimal. In A* the tree search is optimal if h(n) is admissible. The graph search is optimal if h(n) is consistent. In A* if h(n) is consistent the value of f(n) along any path are non-decreasing. When a node in A* is selected for expansion it means that the optimal path to that node is already found. f(n) is the true cost of the goal node and all the other nodes expanded will have at least that cost. If C* is the cost of optimal solution path, then A* will expand all the nodes with f(n) < C* and it will also expand some more nodes around the goal node before fixing the goal node. The time complexity of A* is dependent on the heuristic and in the worst case of an unbounded search space is exponential in the depth of the solution which is b^d where b is the branching factor and d is the depth. When A* assumes infinite state space, the algorithm will not terminate and run indefinitely.[3]

According to Koenig & Likhachev Life-Long A* is an algorithm that generalizes both DynamicSWSF-FP and A*. [1] It is due to this approach that it uses two different techniques to reduce its planning time. A* Life Long is an incremental version of A* algorithm. It is specifically applicable to finite graph search problems with known graphs whose edge cost increases or decreases over time period. LPA* always defines the shortest path from a given start vertex s start to any predecessor s' which minimizes g(s') + c(s',s) until the start state is reached. Life-Long A* does not make all the vertices locally consistent after some edge costs have changed but it uses the heuristics and focuses only on the g – calues that are relevant for computing a shortest path and updates them. Life-Long A* knows both the topology of the graph as well as the edge cost currently.[4] In Life-Long A* g*(s) is considered to be the distance from the start state to the current state which correspond to the g-values of a A* search. In A* Life-Long similar to A* heuristics is used to calculate the distance from the current state to the goal state and the heuristic should be consistent. In Life-Long A* a second set of values called the rhs are also maintained which are one-step lookahead values based on the g-values and thus are potentially more informed as compared to the g-values.[1]

D*Lite algorithm is developed from A* Life-Long algorithm that repeatedly determines shortest paths between the current vertex the agent is in and the goal vertex as the cost of the edge in the graph changes with the movement of the agent towards the goal vertex. [1] The D*Lite has no assumptions about how the cost of the edge changes. It is used to solve the goal directed navigation problems in the unknown terrain. In D* Lite algorithm we first need to switch the search direction as compared to Life-Long A*.[5] unlike Life-Long A* the D* Lite searches from the goal vertex to the start vertex and thus its g-values are estimates of the goal distances. D*Lite is basically A*Life-Long in which the start and the goal vertex are exchanged, and all the edges are reversed.[4] Unlike A* Life-Long D*Lite uses only one tie breaking criterion when the priorities are compared. After computation one can follow the shortest path from the start state to the goal state by always moving from the current vertex s to any successor s' that minimizes the value for c(s, s') + g(s') until the goal is reached.[1] In D*Lite the heuristics are derived by relaxing the search problems. D*Lite uses a method derived from D* to avoid the reordering of the priority queue. To avoid re-ordering, we maintain the variable and incrementally increase its value when robot moves towards the goal state. As the terrain gets larger, D* lite algorithm expands fewer nodes than A* and LPA* algorithms while being far more efficient than the both

of them. One of its main uses is in the domain of greedy mapping.[5]

II. Technical Approach

We are already given with the PACMAN world environment where the PACMAN knows the size of the environment and can observe local environment only. Our implementation of these algorithms is based off on modification of search.py and searchAgent.py files to write the algorithm logic and helper functions and making minor changes to the util.py file to implement some of the new utility functions that we needed as well.

To implement the A* baseline algorithm, we have written two functions "astarRoute" and "aStarBase" in search.py file. In order to implement the baseline search we initiated the start state of the pacman agent and the goal state which is the location of the food palette, then we ran a loop while we did not reach the goal state where we calculated the route from the current state to the goal state using heuristic values. The heuristic used is Manhattan heuristic implemented using Manhattan distance. The route from the current state does not consider walls as the agent is unaware of the walls in the environment so we keep on checking for walls and adding them to the knowledge passed to the agent so that he is aware of the walls as they are found. This is repeated until we reach the goal state after which we pass the instructions to the pacman agent to start its journey to the goal state.

Distance from the start state to the current state is denoted by g(s) and the heuristic distance between the current state to the goal state is denoted by h(s). The optimized path is found using the below formula.

$$F(s) = g(s) + h(s)$$

This algorithm belongs to the informed type of algorithm and it is admissible in nature as we can calculate the goal distance from each and every node by using heuristic and while trying to optimize the path of the agent towards the goal. Hence, in order to find the shortest path, all the nodes are explored in A* baseline algorithm.

In order to implement the A* Life-Long algorithm we have implemented the function "aStarSearch" in search.py file. In this algorithm the heuristic used is also the Manhattan heuristic. We have initiated the priority queue to maintain the vertices which are not locally consistent. The priority of any vertices in this queue is dependent on its key value which comprises of a vector of two components $[k_1(s), k_2(s)]$ so that

$$k_1(s) = \min\big(g(s), rhs(s)\big)\ and$$
$$k_2(s) = min(g(s), rhs(s)).$$
$$rhs\,(s) = 0 \quad if\ s = start$$
$$else$$
$$rhs(s) = min_{s' \epsilon Pred(s)}(g(s') + c(s', s))\ otherwise$$

Initially, we initialize the rhs, g and the key values and compute the shortest path. [1] The value for the g and rhs is initially defined as infinite for all the states and "initialize" function is used to perform this allotment of infinite to all the states except for the start state where it is considered to be zero due to the local inconsistency, this node is pushed into the priority queue under the initialize function. Then we compared the g and rhs values to check for overestimation and underestimation. Further we find the successors for the current state and then update the vertices stored in the rhs values and g values accordingly.

During the implementation of the algorithm we keep checking for changes in the costs of the edge. If there is change in any edge cost, then "UpdateVertex" is used to update the rhs value and the key value of the node that are affected and the new vertex cost is then updated to the priority queue depending on the local consistency and inconsistency. Further "computeShortestPath" was used to repeatedly expand all the vertices that are locally inconsistent depending on their priority in the queue. The implementation of A* lifelong acts as a steppingstone in the development of the D* lite algorithm which was far less resource hungry compared to A* Lifelong algorithm.

For the implementation of D* Lite search algorithm we have implemented a function "dStarSearch". in D* Lite similar to A* Life-Long a priority queue is initiated in order to store the inconsistent vertices. We introduce a key modifier to track and make lesser changes to the queue while planning. "calculateKey" function is implemented to update the key modifier variable at that particular state. In the "calculateKey" function the value is obtained by using the below given formula:[1]

$$(min(g[s], rhs[s]) + util.manhattanDistance(s, start) + keymod\ , min(g[s], rhs[s]))$$

In D*Lite like the A* Life-Long the values of g and rhs are infinite at the start. "initialize" function is implemented to assign these infinite values of g and rhs for all the states except for the goal state where the rhs value is zero. As in the goal state the rhs value is zero and the value for g is infinite it becomes inconsistent and so the goal node is pushed into the priority queue. Further in order to update the value of g and rhs on the path of the agent as it moves ahead, we have implemented the "UpdateVertex" function.[1] If the agent observes any obstacle on its path, then the edge values of all the vertices attached to it changes and becomes infinite which in turn affects its neighbouring vertices. The "UpdateVertex" function is implemented to update the g and rhs values of the affected vertices. After the completion of the updating process it is checked if the value of g and rhs are same for any of the node and if it is same then that node is removed from the priority queue using the "remove" function. After checking if it is found that the values are not equal, and the nodes are inconsistent then it is pushed into the priority queue. For finding the shortest path from the start state to the goal state "computeShortestPath" function is implemented. In the D* Lite search algorithm the agent is navigated in a terrain which is unknown a loop is run till the goal state and then the agent is moved as per the results of the "computeShortetPath" function. In accordance with the D* Lite algorithm the heuristic value needs to be calculated every time the agents move from one state to another state. To avoid these multiple calculations the keymod value is incremented every time the agent makes a move. As a result of

this we obtain a path that is optimized to navigate the agent towards the goal.

III. Results, Analyses and Discussions

For comparison of the search algorithms we choose 5 different layouts namely: counterMaze, smallMaze, mediumMaze, bigMaze and customMaze. The layouts are different based on their complexity and size. The counterMaze is a basic open layout with just a pacman and a food palette whereas the complexity and the size of these mazes are increased in the small, medium and big mazes, respectively. The custom maze is like the big maze in terms of its size but is a slightly more complex than the others with changes to the pacman start location and the goal food palette.

The findings of these experiments in different graphs are judged on different criteria which reflect the performance, the optimality and the efficiency of these algorithms. We use number of nodes expanded to find the computation needed for each of the algorithms to solve the maze. The score obtained tells us on the efficiency of the algorithm and give us insights on the tradeoff between computation time and better routing of the path. The time complexity tells us about the overall time needed for algorithm execution. All these together help us understand the pros and the cons of these algorithms and better understand their use case in practicality.

Our findings are as follows:

a. Number of Nodes Explored

As per the observations in the *Figure 1* and *Table 1*

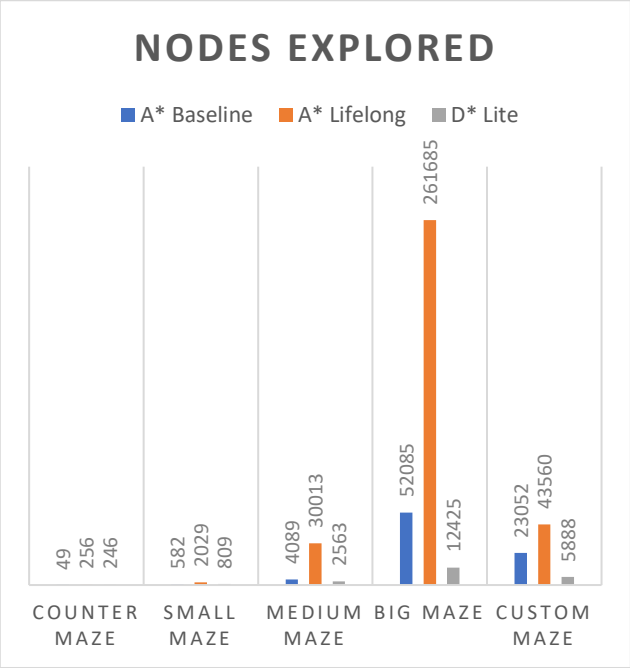

Figure 1:Nodes Explored

	Counter Maze	Small Maze	Medium Maze	Big Maze	Custom Maze
A* baseline	49	582	4089	52085	23052
A* Life-Long	256	2029	30013	261685	43560
D*Lite	246	809	2563	12425	5888

Table 1:Nodes Explored

We can say that the nodes explored by A* baseline for a simple and less complex maze will be similar or better than the D* lite algorithm. With a higher complexity medium maze, the performance of the D* lite improves in terms of the nodes explored which reaches a stage of a massive improvement with the introduction of big and custom big maze.

b. Computation Time

As per the observations in the *Figure 2* and *Table 2*

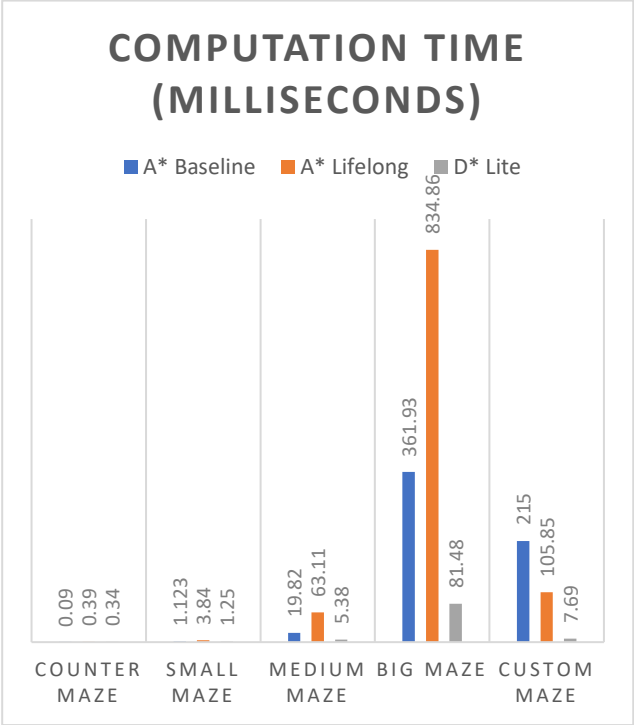

Figure 2: Computation time in milliseconds

	Counter Maze	Small Maze	Medium Maze	Big Maze	Custom Maze
A* baseline	0.09	1.123	19.82	361.93	215
A* Life-Long	0.39	3.84	63.11	834.86	105.85
D*Lite	0.34	1.25	5.38	81.48	7.69

Table 2: Computation time in milliseconds

For the analysis of computation time of the algorithms we used a machine with *8 gigabytes of RAM, a 64-bit 4 core intel 6200U CPU, Intel HD 520 graphics running on Ubuntu 18.04 LTS.* According to our observations, given the optimality of D* lite as seen before in terms of the nodes explored we can say that the computation time of D* lite is a lot less when compared to A* base and A* lifelong for complex mazes but shows similar computation time for simple mazes. It was also observed that the computational time of A* lifelong and A* baseline increased drastically as compared to D* lite.

c. Pacman Score

As per the observations in the *Figure 3* and *Table 3*

The pacman scores were calculated by the project environment that was provided to us. Based on the results we can say that when the mazes were simple and less complex the pacman

scores were same for all the 3 algorithms but as the complexity increased the A* baseline and the D* lite algorithms outperformed the A* lifelong algorithm by a small margin generally. The pacman scores however does not give us a proper quantitative analysis of the algorithm but only portray the tradeoff between node exploration and computation time.

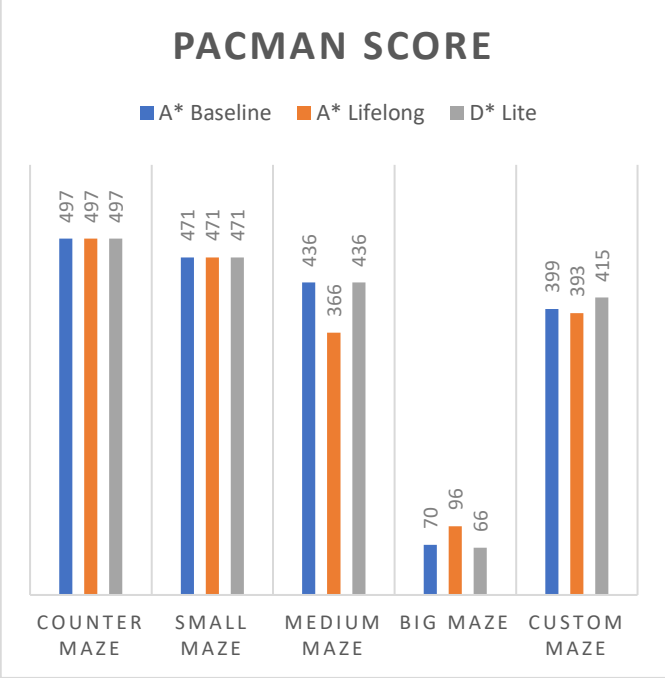

Figure 3: Pacman Score

	Counter Maze	**Small Maze**	**Medium Maze**	**Big Maze**	**Custom Maze**
A* baseline	49	582	4089	52085	23052
A* Life-Long	256	2029	30013	261685	43560
D*Lite	246	809	2563	12425	5888

Table 3: Pacman Score

From the above-mentioned results, we come to a final result that A* lifelong algorithm results even though similar for simple tasks are not as good as A* baseline algorithm for complex mazes so the improved implementation of the D* lite algorithm as mentioned in the paper out outperformed both the other algorithms in a more complex environment while having minimum to no tradeoff between its performance. The D* lite algorithm showed improved results, better to similar scores, a drastically reduced computation time and far less nodes explored compared.

IV. CONCLUSIONS

We can conclude that the number of nodes explored, and the computation time improves for the D*Lite algorithm as the complexity of the environment increases. We can observe that the number of nodes explored improves drastically when we search for the shortest path from the goal state in D*Lite as compared to that from the start state in A* Life-Long. The computation time required by the D*Lite algorithm is also less as compared to the A* Baseline and A* Life-Long algorithms for complex environment. The score obtained by the pacman was same for all the algorithms till the environment was simple and as the complexity increases the scores obtained by the pacman was higher for D*Lite and A* baseline as compared to A* Life-Long. From all the results derived in this environment we can conclude that D*Lite performs better as compared to A*baseline and A*Life-Long with the increase in complexity of the environment. In order to use this D*Lite algorithm for the environments having multiple food pallets and where ghosts are present, we need to extend the logic.

V. REFERENCES

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

[2] Stuart Russell, Peter Norvig, *Artificial Intelligence A Modern Approach Third Edition.*

[3] Soh Chin Yun, Velappa Ganapathy, Tee Wee Chien, "*Enhanced D* Lite Algorithm for Mobile Robot Navigation*" , IEEE Symposium on Industrial Electronics and Applications (ISIEA 2010).

[4] Xiaowu Liu, Riqiang Deng, Jinwen Wang, Xunzhang Wang, "COStar: A D-star Lite-based dynamic search algorithm for codon optimization," Journal of Theoretical Biology 344 (2014) 19–30

[5] Shaoyang Dong ,Hehua Ju and Hongxia Xu, "An Improvement of D* lite Algorithm for Planetary Rover Mission Planning", International Conference on Mechatronics and Automation August 7 - 10, Beijing, China.