Life-long Planning for Path-Finding in Pacman Domain

Lei Zhang, Yi Chen, Siddhartha Cheruvu, Aaron Arul Maria John

***Abstract*—The process of developing search algorithms has always been a hot-spot of problem-solving due its wide range of applicability. For the path planning problems of mobile robots in an unknown environment, regular path search algorithms (including uninformed search and informed search) have been found to yield sub-optimal results as the agent's knowledge of the environment evolves continuously. When autonomous agents traverse an unknown environment, their route-planning should adopt information available from their surroundings to try different solutions, and adjust the next set of actions to achieve the optimal path. In this report, we will discuss and compare the strengths, weaknesses and applicability of three different path planning methods: Life-long Planning A* (LPA*), D* Lite and Simple Replanning A*. Based on their relative effectiveness in path planning, conclusions are derived and presented at the end of this report.**

***Index Terms*—path planning, shortest path problem, robot learning**

I. Introduction

The process of identifying a sequence of actions which leads the agent to a goal state is called *'Search'*. This process holds a very important position in the field of AI. A typical search algorithm takes problem as the input and returns a sequence of actions to achieve the goal. Search algorithms are classified mainly into two types, namely, uninformed search and informed search. Uninformed search strategies do not have any additional information of states apart from the problem definition. Common algorithms include Breadth-First Search(BFS), Uniform Cost Search(UCS), and Depth-First Search(DFS). Informed search is to use specific knowledge beyond the definition of the problem itself. This method can solve the problem more efficiently than the uninformed search. Common algorithms in this category are Greedy Search and A* Search. The latter uses backward cost and heuristic function to find the optimal path. In recent years, with the growth of large-scale applications of mobile robots, the general search algorithms can no longer conform to the requirement of finding the optimal path in an unknown environment. Since the environment is unknown, the robot can only continue to try and adjust through limited surrounding information to achieve path planning. Therefore, how to efficiently deal with the unknown environment has become a problem that is needed to be solved by search algorithm. In this paper we will discuss and analyze the three classic search algorithms (Lifelong Planning A* (LPA*), D* Lite and Simple Replanning A*) in the application of an environment only observable surrounding an agent, which is often the situation in robot navigation. The results will provide us a deeper understanding of these algorithms. The contents of this paper are separated into: 1) Introduction, 2) Technical Approach, 3) Results and Discussion, 4) Conclusion.

II. Technical Approach

In this section, a brief introduction of the aforementioned search algorithms is provided.

A. Simple Replanning A*

Simple Replanning A* Search is a simplified path planning algorithm based on the classic A* Search, which allows the agent to make decisions based only on its surrounding environment.

Classic A* search calculates the priority of each node through the following function:

$$f(n) = g(n) + h(n), \tag{1}$$

where $g(n)$ is the path cost form the start node to node n, and $h(n)$ is the estimated cost of the cheapest path from n to the goal, which is the heuristic function of the A* Search.

In Simple Replanning A* Search, the only perceptible information for the agent is the four adjacent units in the maze environment where it is located. Therefore, the agent will formulate an optimal path based on these details. Until one of the following two situations occur, the agent will use A* Search to re-plan its path: 1) it has reached the goal or 2) it has encountered unobserved obstacles. When the latter situation occurs, the agent marks the encountered obstacle as closed circuit from initial open path, and execute A* Search with the current position as the start node based on updated knowledge of the environment.

B. Lifelong Planning A*

Lifelong Planning A* search (LPA*) is an improved version of A* search based on incremental search. This method deals with the shortest path from a given start node to a known goal node in a dynamical environment. Unlike Simple Replanning A*, LPA* can execute a new search based on the information obtained from the original path-finding calculation. Therefore, LPA* makes decision based on the relevant information of the tuple including $g(s)$ and $rhs(s)$ of each node. Intuitively, $g(s)$ is the path cost form the start node to node s, and its definition is consistent with $g(n)$ in A*. $rhs(s)$ is the minimum value of the set consisting the sum of g value of node's parents s' and the cost from each parent node s' to node s. The specific calculation formula is shown below.

$$rhs(s) = \begin{cases} 0 & if s = s_{start} \\ min_{s' \in Pred(s)}(g(s') + c(s', s)) & otherwise \end{cases} \tag{2}$$

Compared to Simple Repalnning A*, the queuing strategy of LPA* is more complicated. The Agent has three states at node s: locally consistent $g(s) = rhs(s)$, locally overconsistent $g(s) > rhs(s)$ and locally underconsistent $g(s) < rhs(s)$.

The agent makes policy judgments based on the key tuple composed of $g(s)$, $rhs(s)$ and $h(s)$. The smaller the key value, the higher its priority, and the corresponding node will be searched first.

$$k(s) = \begin{bmatrix} k_1 \\ k_2 \end{bmatrix} = \begin{bmatrix} min(g(s), rhs(s)) + h(s, s_{goal}) \\ min(g(s), rhs(s)) \end{bmatrix} \quad (3)$$

LPA* sets up the $rhs(s)$ of the start node to zero in the initial state, while the $g(s)$ (including the start node) and $rhs(s)$ of other nodes are infinite. The agent can pop possible nodes orderly from the priority queue according to the key values, and make $g(s) = rhs(s)$, keep these nodes locally consistent, and finally find the best path. If the algorithm expands a locally overconsistent node, it sets the g-value equal to its rhs value making the node locally consistent. On the other hand, if it expands a locally underconsistent node, the agent will set its g-value to infinity making it locally overconsistent and insert it to the priority queue again. In this way, these nodes will become locally consistent after they are popped from priority queue the next time. When all the the nodes are locally consistent, the agent starts to calculate the shortest path.

LPA* repeatedly plans the shortest path between the start and the goal node. Because the start node is fixed, the planning path after the environmental information changes is not the optimal for the agent at the current time.

C. D* Lite

D* Lite is an efficient algorithm designed specifically for path re-planning applications, derived from LPA*. One of the core ideas of D* Lite is to treat the current node as a new start node, and iteratively calculate the shortest path between the goal node and the new start node. Therefore, modest changes to LPA* are necessary. However, LPA*'s core replanning approach is still maintained.

LPA* runs from the start to the goal node, and ensures that all the traversed nodes reach locally consistency, so the optimal path can be found from these nodes. However, D* Lite adopts the opposite method, which is from the goal to the start node. In addition, the difference between D* Lite and LPA* lies in processing the heuristic function value of the node when changes in the environment are discovered.

As mentioned earlier, the LPA* solves the optimal path search problem with a fixed start and goal node. The heuristic function value of the node s does not change dynamically. On the other hand, D* Lite considers the problem as a mission composed of fixed goal node and variable start node. The value of the heuristic function of node s is variable with the change of the start node.

Similar to LPA*, D* Lite also prioritizes nodes based on key tuples defined by $g(s)$, $rhs(s)$, and $h(s)$. Besides, it also maintains the offset k_m, which can limit the propagation of re-planning, thereby ensuring that D* Lite has better performance than LPA*.

$$rhs(s) = \begin{cases} 0 & if s = s_{goal} \\ min_{s' \in Succ(s)}(g(s') + c(s', s)) & otherwise \end{cases} \quad (4)$$

$$k(s) = \begin{bmatrix} k_1 \\ k_2 \end{bmatrix} = \begin{bmatrix} min(g(s), rhs(s)) + h(s_{start}, s) + k_m \\ min(g(s), rhs(s)) \end{bmatrix} \quad (5)$$

The framework of D * Lite is as follows: it first attempts to calculate the shortest path from the start to the goal node, then it takes the steps along this path, calculates the rhs of all subsequent nodes, and selects the nodes corresponding to the smallest rhs value as the new start point. After that, it scans the map to update the rhs and g values. If the rhs and g values of the successor of the latest node do change, then setting the latest node as the new start node, and traversing all the nodes with variable rhs and g values, and updating the rhs and g values of these nodes. Then, using this new method to calculate the shortest path information and repeat. Continuing the foregoing method until the goal state is reached.

D. Pacman Domain Implementation

The implementation of our search algorithm is written in Python 2, and the environment that we will be testing our algorithm would be in the Pacman domain. The PEAS of the Pacman environment are discussed in this section.

Classic Pacman game provides a great testing platform for general algorithms of path planning. The tester can explore the path planning under different algorithms by adjusting the game's setting parameters, such as the presence of ghosts, the location of walls, the density and location of food, and other environmental parameters.

Under the basic settings of the Pacman game, the agent is assumed to move in four different directions in this maze: East, South, West and North. Besides this, the maze is surrounded by four insurmountable obstacles. In addition, the agent does not know whether there are other obstacles in the maze, but the agent can judge whether it encounters obstacles in the four locations around itself. These settings simulate an unknown environment where constant replanning is required.

III. Results and Discussion

A. Performance Evaluation Metrics

The four indicators that we are using to evaluate the performance of the search algorithm are path length, number of expanded nodes, memory usage and running time. Path length refers to the length of the total path Pacman takes to find the goal. This path is the same as the path visualized in the Pacman environment. The number of nodes expanded refer to the number of nodes that are expanded to find the path to the goal.

This number is related to the complexity of time and space in the way that the more nodes are expanded, longer the search time, and more memory is needed to store these nodes. Memory usage describes the total memory used by the Pacman Python process to execute the algorithm. Command $valgrind$

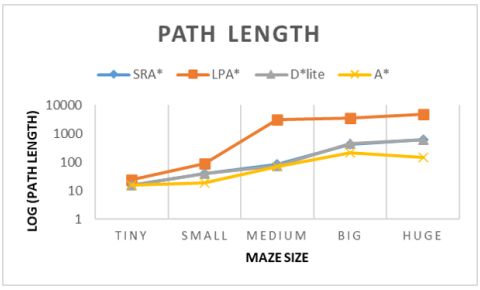

Fig. 1. Path Length vs Maze Size

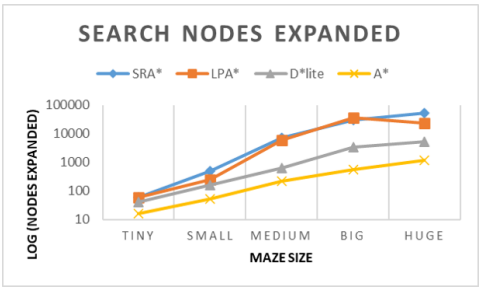

Fig. 2. Number of Search Nodes Expanded vs Maze Size

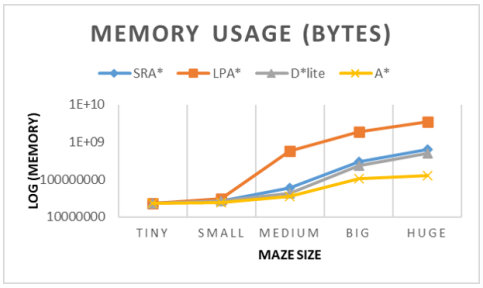

Fig. 3. Memory Usage vs Maze Size

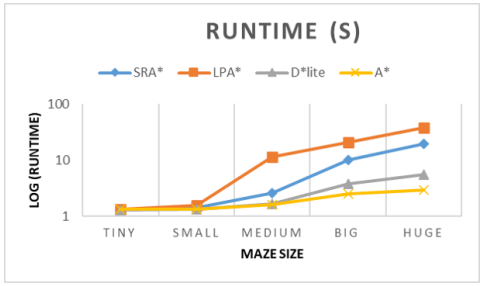

Fig. 4. Runtime vs Maze Size

is used to capture memory usage, and command $time$ is used to capture the run time, which reflects the time needed to get Pacman to the goal.

All four index parameters are measured for each type of maze. Please note that there is no direct linear relationship between the size of the maze and the search complexity, because the properties of the maze may not be affected by its size, such as complexity or wall number. However, the size of the maze is related to the complexity of the search. In general, the larger the size of the maze, more complicated the search problem.

A* search is included as a baseline comparison. Note that in this case the A* algorithm has a fully observable environment, while the other three algorithms only have a partially observable environment, that is, the four adjacent pixels surrounding Pacman's current location.

We also tested each algorithms in mazes with different complexity level. We rank the complexity by the amount of walls are added to them, thus creating number of detours that would likely occur during the search. The mazes were tested with A* search to confirm the rank of complexity.

B. Discussion

1) Varying Maze Size:

We executed each of the four algorithms in five different maze sizes (tinyMaze [7x7], smallMaze [22x10], mediumMaze [36x18], bigMaze [37x37] and hugeMaze [47x47]) in the Pacman domain and evaluated the four performance metrics for comparison. The results are shown in the figures below.

In all of the algorithms, the path length increases with the size of the maze (see Figure 1). A* search has the best performance as we expected, followed by simple planning A* and D* Lite, and finally LPA*. The reason for the low performance of LPA* is that the agent will keep backtracking when executing the algorithm, which will increase the path cost.

The number of nodes expanded in different algorithms are shown in figure 2. The rank from the best to the worst is as expected: A*, D* Lite, LPA* and then Simple Replanning A*. The large amount of nodes expanded by Simple Replanning A* is due to the fact that after the agent encounters obstacles, it will use the current node as the new start node to execute A* search again for path planning, leaving the knowledge of all previously explored nodes from the queue. Both LPA* and D* Lite have a solution by reusing the previous planned path to avoid this situation during the search.

In figure 3, the memory usage is compared against the maze size. The amount of memory used from low to high is A*, D* Lite, Simple Replanning A* and LPA*. The low performance of LPA* is due to the backtracking nature. Looking back at the comparison of path length in figure 1, we found that the path length of LPA* is also the largest among the four algorithms. Therefore, it can concluded that the memory usage is directly related to the path length.

Lastly, we compare the algorithms with their run time in figure 4. A* and D* Lite finished the search with the least time.

This result indicates that even if the agent has only the partial information of the environment, by using the correct strategy (in this case, D * Lite), agent's search time can be

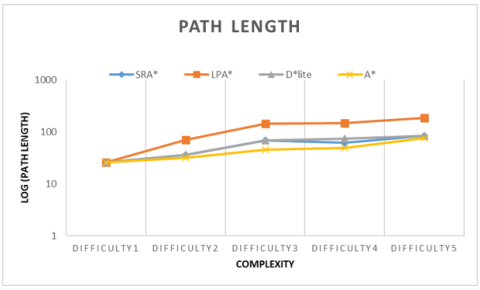

Fig. 5. Path Length vs Maze Complexity

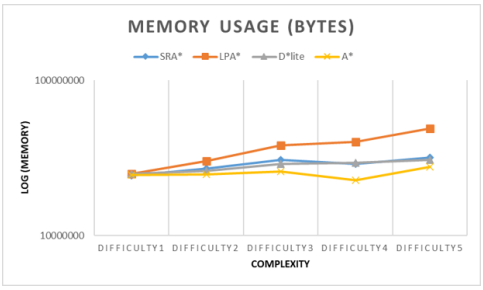

Fig. 7. Memory Usage vs Maze Complexity

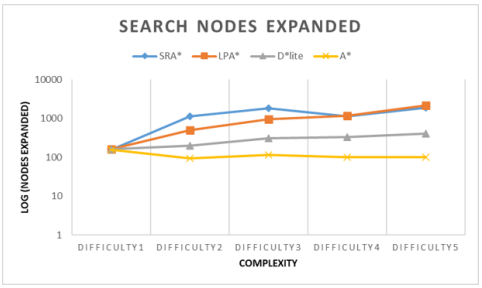

Fig. 6. Number of Search Nodes Expanded vs Maze Complexity

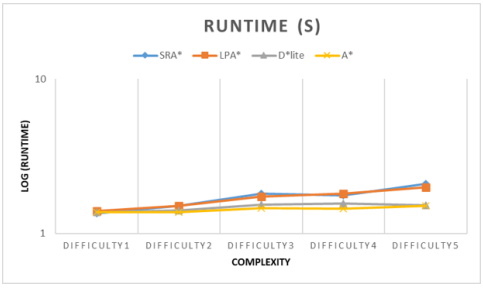

Fig. 8. Runtime vs Maze Complexity

brought relatively closer to that of the agent with a fully observable environment (A*). The other two algorithms have a longer running time due to the nature of re-planning or backtracking mechanisms.

In summary, D* Lite is the most suitable algorithm for path planning in partially observable environment, especially in terms of run time, which has direct implications in the real world where conditions could be constantly changing. The performance of LPA* is not as competitive as expected, mainly due to it requiring a backtracking process whenever obstacle is detected and re-planning is required.

2) Varying Complexity:

For the purpose of assessing the effectiveness of each algorithm in worlds with varying complexity, the small size the maze [22x10] is chosen for creating worlds with various complexity. Five different variations of this world are created by different number of obstacles in the path to reach the goal.

The trends in path length, number of search nodes expanded, memory usage and run time with varying complexity are similar to that of varying maze sizes.

LPA* performs relatively poorly in all these departments due to its backtracking nature. A* remains the best performing algorithm in observable environments. D* lite is found to be the best option in the case of an unknown and partially observable environment. This is because it takes specific measures to only re-evaluate the nodes that will have an effect on the final path. Simple Replanning A* outperforms LPA* in all the metrics except for the number of search nodes expanded. This is due to the fact that Simple Replanning A* algorithm doesn't use the key values of the visited nodes whenever an obstacle is met. Instead it reruns the A* search from the agent's current location and repeats the process.

IV. Conclusion

In this project, we implemented three different search methods, LPA*, D* Lite and Simple Replanning A* and then compared their performance against each other inside a partially-observable Pacman domain. As a datum, an A* search agent in a fully observable environment is used to assess the remaining algorithms. The Pacman domain is chosen due to its resemblance with many robotics search applications, where the agent is assigned to reach a goal given an unknown environment. Based on the results, we can conclude that D* Lite search algorithm is the most suitable especially in mobile robot exploration activities in an unknown environment where constant re-planning is required. It is an interesting observation to note that its performance is even comparable to A* search agent from an observable environment.

V. Team Contributions

The project responsibilities shared by each team member are provided in this section.

Siddhartha and Lei were in charge of writing the A* and D* Lite algorithms and also creating the priority queue utility. Yi and Aaron were in charge of writing and debugging the LPA* and Simple Replanning A* codes. Lei and Yi prepared the project report whereas Aaron and Siddhartha collected and organized the data for visualization.

References

[1] Sven Koenig and Maxim Likhachev, "D* Lite", 2002, AAAI
[2] Stuart Russell Peter Norvig, "Artificial Intelligence - A Modern Approach, Third Edition"