Comparative Study of D*Lite & Naive Re-Planning A* Algorithm in Partially Observable Terrains

Neha Tandon
Arizona State University
ntandon3@asu.edu
#1216798336

Atin Singhal
Arizona State University
asing274@asu.edu
#1217358454

Shivani Shah
Arizona State University
ssshah35@asu.edu
#1215350435

Amarnath Tadigadapa
Arizona State University
atadigad@asu.edu
#1215129955

***Abstract*—An autonomous agent faces many challenges in an environment, maximizing utility while accomplishing a task is one of them. Path finding is one such task that is integral to most applications and utilizes the agents sensibilities regarding the environment. In a partially observable domain the agent has to be agile in its decision making. D*Lite[1] requires the agent to compute the path using information as it becomes available. To minimize overhead and maximize utility, the agent must not perform redundant calculations and instead re-use information from the past. In this report, we study incremental path finding using D* lite[1] and Naive Re-planning A*[2] algorithms in the pacman domain and perform a comparative analysis to generate performance based results.**

***Index Terms*—Incremental Path Finding, Partially-observable environment, D* Lite, Naive Re-planning A*.**

I. Introduction

A rational agent is expected to find the shortest path from start to goal location in a given task environment. The challenge for the agent becomes the information that is known to it regarding the environment. In the pacman domain, we assume that the agent is in a partially observable environment and is aware of the following: i) its local environment that includes its current location and information of the neighbours in the four directions, ii) start location, goal location, boundary of the terrain, and iii) keeps past observations and computations in its mind. The aim of the agent is finding the path as it observes new information in the domain along with storing that information in order to utilize it in the future to avoid re-calculations. The motivation lies in maximising utility of the agent by minimizing overhead and time of computation. We look at informed search algorithms like A*[2] that uses heuristics to optimise the search and study the implementation of A*[2] and its modifications to analyze performance.

In an agent navigation task in unknown terrain, the agent starts at the start location and moves to its goal location. It calculates the shortest path from its current location to the goal location based on its current knowledge of obstacles in its local environment. It then follows the path until it reaches the goal location, or it encounters an obstacle in its new local environment, in which case it recomputes the shortest path from its current location to the goal.

The search algorithms D* Lite[1] and NRA*(Naive Re-planning A*) differ in their approach and performance, the goal of our project lies in implementing these algorithms for pacman agent to find shortest path from the start to goal. In section II of the report, we discuss the background of the above mentioned algorithms and the pacman domain. In section III, we present our implementation of the algorithms and integrate them to the pacman world. In section IV, we discuss the performance evaluation of the algorithms. In section V, we present the conclusion and discussion followed by section VI which includes the contributions of the team.

II. Background

This section contains a brief technical discussion for Naive Re-planning A*[2] and D* Lite[1] Algorithms.

A. Naive Re-planning A*

The Naive Re-planning A*[2] algorithm is an extension of the A* search algorithm to an environment which is partially observable. A*[2] search algorithm uses a heuristic estimate to rank nodes and aims to find a path to a given goal node having the smallest cost. The path which minimizes the value of $f(n) = g(n) + h(n)$ is selected by A* algorithm, where n represents the next node, $g(n)$ represents the cost of the path from the start node to node n, and $h(n)$ represents a heuristic function that estimates the cost of the shortest path from node n to the goal node.

In NRA*(Naive Re-planning A*), the agent can only perceive the boundaries of the assumed grid-world environment. The agent finds an optimal path and follows it until it has either reached the goal node or encountered a new obstacle. If it encounters a previously unobserved obstacle, the agent re-runs the A*[2] search, and repeats the process. Any previous knowledge that might have been computed earlier doesn't carry over to the next search, resulting in a significant re-computation.

B. D* Lite

With A*[2] and a good heuristic, we choose a straight line in path planning. If an obstacle occurs, we have to re-plan the

sections that are beyond the obstacle. With incremental path planning, we have an efficient search that goes around the path we had already planned. Anything before the obstacle wouldn't be affected by the obstacle so it's useless to calculate it again using A*.

D* Lite[1] is a search algorithm based on incremental heuristic search. Unlike Naive Re-planning A*, D* Lite is able to use information from previous computations to inform new searches. D* Lite keeps track of $g(s)$ and $rhs(s)$. $g(s)$ represents the previously calculated g-values, and $rhs(s)$ is a look-ahead value that is based on the g-values and is equal to the sum of the cost of the parent node and the cost to travel to that node. It can be thought of as corrected value of g. Initially, only the start state is locally inconsistent. Both rhs and g values are updated constantly. The $g(s)$ in-turn relaxes down to the $rhs(s)$ value. When the values become equal, we can be sure there aren't any inconsistencies.

Unlike classic Lifelong Planning A*[1], D*Lite[1] updates nodes on the path from the in-progress current position to the goal state. D* Lite also maintains a priority offset k_m, which allows it to re-plan. The key modifier is initialized as 0 at the start and increases every time we update the start value.

III. Implementation Details

To compare the performance of A*[2] and D*lite[1] algorithms, we are using partially observable environment in which the agent only knows about the boundaries, start and goal locations of the environment. It can only sense its surrounding as and when it travels. As it comes across walls, it updates its information. A* algorithm and D*Lite algorithm handle these situations differently.

A. Naive Re-planning A*

In Naive Re-planning A*[2] algorithm, initially,the agent finds the shortest path from the start to the goal using A* algorithm, after it gets the shortest path direction from the initial A* algorithm it traverses this path . As it traverses the path when ever it finds a wall or an obstacle, it updates its internal representation of the environment with this information. Since the previous strategy failed, the agent re-plans the entire strategy from scratch using A* algorithm with the start node as the current node and the updated environmental representation as a modified graph.

This methodology is classified "naive" on the grounds that it doesn't re-utilize any of the past planning calculations for later re-planning, which can prompt re-exploration of numerous states, and pointlessly copied calculation. Furthermore, in spite of the fact that A* is ensured to locate the best course given the known natural data, NRA* isn't, this is on the grounds that it works with missing data, and can lead the agent down a local minima path.

B. D* Lite

We have implemented this algorithm taking the psuedo-code in Figure 1 as reference. D*Lite[1]search algorithm is an incremental search algorithm that updates only those nodes that observe a change in edge cost and need not re-plan every time it sees a new obstacle like A*[2], in our case the new walls it comes across.

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()$<$CalculateKey(s_{start}) OR $rhs(s_{start}) \neq g(s_{start})$)
{11'} k_{old} = U.TopKey();
{12'} u = U.Pop();
{13'} if ($k_{old}<$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 pseudo-code Image Source: Figure 3 in [1]

Whenever the search is done from Start to Goal (for example the heading of search utilized in Naive A* algorithm(let's call this S-G), the "start distances" i.e. g values in the A* algorithm are contradicting each time the agent moves.

To avoid this, D*Lite searches in the opposite direction that is from goal to start node(let's call this as direction G-S). In this algorithm the g values represents the distance from the goal. If the agent comes across a wall that it didn't see beforehand, the nodes that should be updated for re-planning will be the ones between the current node and the wall. In the S-G situation, it is the nodes between the wall and the goal that are required to be updated are much more than that when compared to G-S and are likely un-observable and requires future updates as the agent discovers new walls.

The node's priority in the priority queue that depends on the goal and start-distances (g and h values) is useless once the agent moves, because the preceding values of h are

obsolete in the D*lite search. The agent has moved away from them which further leads to increase of actual "start distances", or h values, and visa-versa for the nodes which are currently nearer to the agent. Hence, few of the nodes will be having a very high priority. When the high priority nodes are encountered, the values can be corrected with the actual ones. However, the nodes with too-low-priority would be an issue. This is because the corrected(decreased) priority nodes' priority may be still greater than the too-low-priority nodes' priority.

When updating a node from the queue, if its current queue-priority is the same as its true queue-priority, it must be the highest priority node. The difference between these priorities is at most the heuristic distance between the current position of the agent and the position in which it was earlier, when the queue-priorities were determined (say $h(s, s')$). So to update the queue-priorities of all nodes in the queue to the lower-bound of their real queue-priorities,the authors of [1] suggested that a constant difference ($h(s, s')$) should be added to all newly calculated queue-priorities which is an accumulation of the $h(s, s')$ values from previous steps Figure 1[line 1'].

This is done by $ComputeShortestPath()$ Figure 1. After $ComputeShortestPath()$ has removed a vertex u with the smallest priority $k_{old} = U.TopKey()$ from the priority queue Figure 1[line 12'], $CalculateKey()$ calculates the new priorities of the nodes Figure 1[lines 1'] If k_{old} is less then $CalculateKey(u)$ then it reinserts the removed vertex with the new priority calculated by $CalculateKey()$ into the priority queue Figure 1 [lines 13'-14']. If $k_{old} \geq CalculateKey(u)$, then it holds that $k_{old} = CalculateKey(u)$ since k_{old} was a lower bound of the value returned by $CalculateKey()$. In this case, $ComputeShortestPath()$ expands vertex u (by expanding a vertex, executing Figure 1 [lines 15'-20'].)

The main function $Main()$ Figure 1 of D*Lite first calls $Initialize()$ to initialize the search problem. $Initialize()$ sets the g values of all vertices to infinity and sets their rhs-values to infinity except the goal node. The rhs value of the goal node is 0 Figure 1[lines 03'-05']. Thus, initially goal state is the only locally inconsistent vertex and is inserted into the empty priority queue Figure 1[line 06']. This initialization guarantees that the first call to $ComputeShortestPath()$ performs an A*[2] search and follows the path until a wall is reached. The key modifier is updated every time the agent moves according to line 38 in pseudo-code Figure 1.

If some edge costs change, it updates the edge cost in its information stored about the environment and calls $UpdateVertex()$ Figure 1[lines 07'-09']. $UpdateVertex()$ updates the rhs-values and keys of all the vertices that are affected by the changed edge costs as well as their presence in the priority queue if they become locally consistent or inconsistent, and finally recalculates a shortest path Figure 1[line 19'] by calling $ComputeShortestPath()$, that repeatedly expands locally inconsistent vertices in the order of their priorities.

A locally inconsistent vertex is called locally over-consistent if $g(s) > rhs(s)$. When $ComputeShortestPath()$ expands a locally over-consistent vertex Figure 1 [line 15'], then it relaxes the g-value of the vertex to its rhs-value 1 [line 16'], making the vertex locally consistent. A vertex is locally under-consistent if $g(s) < rhs(s)$. When $ComputeShortestPath()$ expands a locally under-consistent vertex 1[lines 18'-19'], it sets the g-value of the vertex to infinity 1[line 15']. This makes the vertex either locally consistent or over-consistent. If the expanded vertex was locally inconsistent, then the change of its g-value can affect the local consistency of its successors thus, $ComputeShortestPath()$ updates rhs-values of all these vertices, checks their local consistency, and adds them to or removes them from the priority queue accordingly Figure 1 [lines 06'- 08']. $ComputeShortestPath()$ expands vertices until start state is locally consistent and the key of the vertex to expand next is no less than the key of start state.

IV. Results

We have used various layouts with just the information of the goal and start locations and analyzed the performance of D*Lite[1] in which we only consider the vertices whose cost is changed in the current path to recompute the shortest path and Naive A*[2] algorithm in which we consider all the vertices from the current node to the goal for recomputing the shortest path when ever we observe an obstacle in the path.

Table I presents the results of D*Lite and Naive Re-planning A*[2] on different terrains where the first column of the table represents the width and height of the terrain, the second column represents the total number of nodes expended to find the shortest path using D*lie algorithm, the third column gives the actual time in milliseconds taken by the D*lite algorithm to find the shortest path in the terrain and fourth and fifth columns gives the values of the nodes expanded and time taken by Naive A* algorithm to find the shortest path in the terrain. We have observed that D* lite outperforms Naive A* algorithm in factors like the total number of nodes that were expanded during the search process and also in the total amount of time taken to find the shortest path from start state to the fixed goal state. we also observe that with the increase in the size and complexity of the terrain the difference in the performance of both the algorithm further increases and D*Lite performs much better when compared to Naive A* algorithm.

V. Conclusion

We tested D*Lite[1] and Naive A*[2] algorithms on different terrains of varying sizes and complexities. Complexity is defined as the number of obstacles present in the terrain. In case of Pacman, this is defined as the total number of walls that are discovered as the Pacman agent traverses.

Terrain Size(w,h)	Nodes Expanded in D* Lite	Time Taken in D* Lite (msec)	Nodes Expanded in Naive A*	Time Taken in Naive A* (msec)
37, 37	4198	686	95884	40856
36, 18	779	128	18381	3365
22, 10	168	62	1949	359
20, 17	217	109	1897	156
20, 20	154	47	1618	136
7, 7	31	0	83	16

TABLE I: Number of nodes expanded & time taken for D* Lite[1] and Naive A*[2] on different Partially-observable Terrains

Based on the results, we observe that the most important factor that affects the performance of both the algorithms is the complexity of the terrains. As the number of obstacles in the terrain increase, we see an increase in the cost of recomputing the shortest distance in the Naive A*star algorithm. This increase in cost is because A* does not carry over any previous knowledge that might have been computed and simply starts from the start node again upon discovering new obstacles.

The size of the terrain also plays a major role in the performance of both the algorithms as this directly impacts the number of nodes expanded in the algorithm. In all the cases D* Lite outperforms Naive A* algorithm in attributes like the total time taken by the algorithm to find the shortest path and total number of nodes expanded by the algorithm.

We know that D* Lite expands each node at max twice. If we compare this to Naive A*, which needs to recalculate everything from the start if an obstacle is encountered, we see that A* methods will have to do a lot more computation when compared to D* Lite. Naive A* performs better only when the obstacles or the changes encountered are close to the start node or if there are large changes in the graph. In all other cases, D* Lite performs better as it updates the previous search results, saving re-computation. We can see that the results we obtained also show the same results.

VI. Team Contributions

Amarnath and Atin were responsible for implementation of D* lite and Naive Re-planning A* algorithms while Shivani and Neha covered the compilation of the report.

Acknowledgment

We would like to thank Dr. Yu Zhang for giving us this opportunity and guiding us in our research and implementation of this project.

References

[1] Sven Koenig and Maxim Likhachev. D^* lite. *Aaai/iaai*, 15, 2002.
[2] Parth Mehta, Hetasha Shah, Soumya Shukla, and Saurav Verma. A review on algorithms for pathfinding in computer games. 03 2015.