Dynamic Path Planning

Abhianshu Singla
Department of Computer Science
University of Central Florida
Orlando, Florida 32816, USA
abhianshu@knights.ucf.edu

Abstract

Path Planning is an important technique in both static and dynamic worlds. It can be considered a trivial task in static environments; however, it is very difficult to plan in a dynamically changing world with constrained time limits and limited trajectory costs. Path Planning through A* is better than Dijkstra's or Best-First Search Algorithms for known environments. In this paper, we propose several variants of A* algorithms for known, unknown, static and dynamic environments. We then provide a detailed analysis and comparison of these search algorithms. The experimental results show that Adaptive A* Algorithms such as Real-Time Adaptive A* is able to compute path in real time while Incremental Heuristic Search Methods such as D* Lite computes the path by exploring less number of nodes compared to other search methods.

Introduction

Robot Navigation is an important aspect in both static and dynamic environments. If the environment is static, the task of path planning becomes trivial. All the obstacles and free space are known to robot a priori and robot can plan the path in advance using any of the path planning algorithms like Dijkstra's Shortest Path Algorithm, Best-First Search, A*. However, in the real-world scenario, it is very rare that the world is static. Therefore, the robot is required to plan in a Dynamic environment. Dynamic Path Planning is a difficult task as static or dynamic obstacles are added, deleted or their location is changed. When the robot aims to reach a certain goal position, it starts navigating the world with the local information. It also stores the information of the path traversed till now and re-plans whenever an obstacle is encountered in the map or an information mismatch of new information with the acquired information is detected.

Breadth-First Search Method (BFS) is a simple search method in which all successive states of a current state are explored before the next-level of successive states. BFS is guaranteed to provide the minimum trajectory cost path from the current position to the goal position. BFS has an assumption of same edge cost from one state to the adjacent state, which limits its usage in real-world applications. Dijkstra's Algorithm eliminates the limitation of same edge cost. In Dijkstra's search algorithm, the shortest path is computed from start node to every other node. From start node, the cost of every successive node is computed and the node with the minimum cost is explored further and the process continues till it reaches the goal node. Although BFS and Dijkstra's algorithms are beneficial in unknown environments, they are time inefficient. On the other hand, if the environment is known, then a fast informed search can be done using Best-First Search Method. It is a greedy sub-optimal algorithm, which always explores the most promising node based on a heuristic rule like Euclidean Distance, Manhattan Distance, etc. Similarly, A* performs an informed search from a source/current location to a target/goal location. Thus, A* provides an optimal solution which takes the advantages of both the Dijkstra's Algorithm (cost of current state from start state) and Best-First Search (admissible heuristic cost from goal state).

Several variants of these search algorithms have been proposed in the literature for the robot navigation problem in a dynamic environment. A simple intuitive approach is to initially plan a path using A* search and then re-plan whenever the robot encounters an obstacle on the planned path. Based on this idea, we propose five search algorithms namely Forward A*, Real-Time Adaptive A* (RTAA*), Learning Real-Time Adaptive A* (LRTA*), Lifelong Planning A* (LPA*) and D* Lite. The key difference between these algorithms lies in the way they update their heuristics after each unsuccessful attempt to reach the goal. In Forward A* search, the robot plans the initial path from start to goal using known world information. After planning a path, the robot starts traversing the planned path. Whenever any new obstacles or changes in the planned path are encountered, the robot discards the previous path and re-plans from the current location to the goal location. This method always provides a minimum-cost trajectory possible in the given map. Unfortunately, this search method is time-inefficient as the robot completely ignores the previous heuristic information. However, methods based on Incremental Heuristic search technique considers the previous search heuristic to re-plan the path and are thus faster than Forward A* search. The two main algorithms based on Incremental Heuristic search are LPA* and D* Lite. The first search of LPA* algorithm is similar to A* search but the consecutive searches are much faster than A*. D* Lite is a short version of D* with only one tie-breaking condition, making it simple to implement. D* Lite is based on the free-space assumption when planning for the robot navigation in an unknown world. Another

approach is to plan a short trajectory from the start towards the goal and then extend this trajectory after reaching a certain stage. Planning multiple small trajectories is efficient than planning long trajectories after every unsuccessful attempt. RTAA* and LRTA* are anytime algorithms which plan a short path using only local heuristic information. These methods are most suitable for real-time path planning applications. However, the main drawback of these methods is that they don't guarantee a minimum cost path. All of the above-mentioned methods have their own advantages and disadvantages. The choice of an algorithm for path planning problem is based on the trade-off between execution time, the number of explored nodes and cost required to reach the goal.

Related Work

The most popular approach for path planning is to discretize the continuous space into a grid space and then perform a Grid-based search. In literature, A* and D* methods are still among the notable path planning algorithms. However, it is difficult to discretize a robot having high degrees of freedom and complex configuration. Hence, randomized sampling-based algorithms such as Probabilistic Roadmaps (PRMs) and Rapidly-exploring Random Trees (RRTs) are widely used path planning methods in Configuration Space. Connell and La 2017 proposed the Dynamic Planning and Replanning for Mobile Robots using RRT*. Sampling-based search is fast but is unable to provide an asymptotically optimal solution, i.e. it fails to converge many times or to detect whether a path to goal exists.

Another search technique is to use artificial potential fields in which robot is considered as a point in Configuration space and it is attracted towards goal and repelled from obstacles. Hwang and Ahuja 1992 proposed the potential field approach to path planning. Minimal computation is required to compute the path which can be determined through the trajectory in C-space. However, the robot tends to fall (trap) in local minima of potential fields. A completely different method based on Reward-based planning or Reinforcement Learning is also widely used and one of the state-of-the-art algorithms. Li and Zhang proposed Q-Learning based method of Adaptive Path Planning for mobile robots. Some of the Hybrid approaches use a combination of Sampling-based, Geometric-based or Reinforcement Learning methods to provide better results by taking into account the advantages of more than one technique.

Our implementation of the variants of A* algorithm provides an in-depth analysis of the planning problem in a discrete state space or Grid World. Many of the Artificial Intelligence and Robotics applications use A* or D* Lite because of their simplicity and ease of implementation. Also, they provide a strong foundation for further research and development of advanced path planning algorithms.

Method

Forward A*

A* is an algorithm which is used to find the minimum cost path from a start state to a goal state in a graph. It combines

Algorithm 1 A*

```
procedure ASTAR(start)
    Initialize lists open, closed, action, policy
    f = g + h
    Add [f[start], g[start], h[start], start] in open
    while not found and not resign do
        if empty open then
            Return resign
        end if
        Inverse Sort open
        Pop minimum value tuple from open
        if tuple is goal then
            Set found to True
        end if
        Explore neighbours of tuple not in closed
        Add neighbours to open
        Add tuple to closed
        Save action for neighbours
    end while
    if  thenfound
        Back-propagate from goal until start
        Save policy in each back-propagate iteration
        Return policy
    end if
end procedure
```

Figure 1: A* Algorithm

Algorithm 2 Forward A*

```
procedure FORWARDASTAR
    found = False
    policy = ASTAR(start)
    while not found do
        Follow policy until a change is detected at current node
        if change detected then
            policy = ASTAR(current)
        else
            found = True
        end if
    end while
end procedure
```

Figure 2: Forward A*

the estimation of the goal distance from the current state (heuristic) and distance of current node from the start state (path cost). The heuristic should be an admissible heuristic, i.e., the estimated cost *h[current]* should be less than or equal to the optimal cost of current node to the goal node. A* maintains two costs with every node actual distance from start state *(g[s])* and a summation of actual cost and estimate of goal distance *(f[s] = g[s] + h[s])*. A* also maintains two lists - open and closed, for tracking the explored nodes. Figure 1 shows the working of A*: It starts by inserting the start node in the open list with key *[f[s], g[s], h[s]]*. It pops the node with minimum *f[s]* from open list and adds the unexplored successor nodes to the open list and the current node to the closed list. It then keeps on repeating this step until the goal state is reached or the open list becomes empty in case of failure (no possible solution). Action/Policy can be computed by back-propagating from the goal node to the start node. The back-propagation can be done following the g-cost values from goal to start as this cost will continuously decrease in every step from goal to start node. Forward A* is an iterative version of A* Algorithm. Figure 2 shows the working of Forward A*. In Forward A* Search, the robot plans the initial path from the start state to goal state using known world information. After path planning,

robot start traversing on the path and whenever a new obstacle or a change is encountered in the planned path, robot plans from the current location to goal location from scratch. The search is repeated until the goal state is reached. Forward A* always follows a the minimum cost trajectory in the given map.

Real-Time Adaptive A*

In Real-Time Adaptive A*, heuristics is made more informative through already planned information, thus speeding up the future A* searches. RTAA* plans a short trajectory from the start towards the goal and then extends this trajectory after reaching a certain stage. The main idea behind RTAA* is as follows: An admissible estimate of goal state *gd[s]* is non-negative and should not be greater than the actual minimum cost path to the goal, i.e., the distance of the current state (start state of the small trajectory) to the goal distance is not greater than the start distance via intermediate state s plus the distance of the state s to the goal state. The goal distance of the current state is equal to the f-value of the state s' which was about to expand when A* terminates. After every A* termination, the heuristics of nodes in the closed list are updated with the difference of the f-value of the current state s' and the cost of state in the closed node. The below equations shows the relationship between the current node and the node in the closed list.

$$gd[s_{curr}] \leq g[s] + gd[s]$$
$$gd[s] \geq gd[s_{curr}] - g[s]$$
$$gd[s] \geq f[s'] - g[s]$$

The pseudo-code of Real-Time Adaptive A* is shown in Figure 3. The first search is a simple A* search similar to Forward A* search, but the consecutive searches are different and faster than Forward A*. After initial A* search, the robot starts moving towards the goal state till a change is encountered at state s' where the A* search terminates. The heuristics of all the nodes in closed state are updated using $h[s_{closed}] = g[s'] + h[s'] - g[s_{closed}]$. A* search is called again with start state equal to current state s' and the modified heuristics. This process is repeated until a goal state is reached. In our implementation, we have assumed a look-ahead of infinity in order to compare the results with other Dynamic Path Planning Algorithms. The value of look-ahead determines the behavior and performance of RTAA*. The look-ahead of one means that the robot only travels one step towards the goal and senses the changed environment, which is highly inefficient. However, a look-ahead of infinity means that the robot plans all the way to the goal before starting to navigate towards the goal, which is efficient in small maps. All the properties of RTAA* (which is similar to A*) holds true for any value of look-ahead. Properties of RTAA* are as followed,

- Every state is expanded at most once in A* search.
- The heuristics remain consistent.
- The heuristics become more informed as the time elapses because the heuristics of the same state are monotonically non-decreasing over time. Thus, $f[s] \leq f[s']$ holds true for every expanded node and $f[s'] \leq f[s]$ holds true for nodes in open list which are yet to be explored.
- The agent always reaches a goal, whenever there is a possible path.
- The number of times that the agent doesn't follow the minimum cost trajectory path from the source state to the goal is upper bounded by a constant iff the changed costs are lower bounded by another constant.

Algorithm 3 Real-Time Adaptive A*

```
procedure RRTAstar
    policy = Astar(start)
    found = False
    while not found do
        Follow policy until a change is detected at current node
        if change detected then
            for nodes in closed do
                h[nodes] = g[current] + h[current] - g[nodes]
            end for
            policy = Astar(current)
        else
            found = True
        end if
    end while
end procedure
```

Figure 3: Real-Time Adaptive A*

Learning Real-Time A*

LRTA* is similar to RTAA* except the way the heuristic is updated. In LRTA*, heuristics of the nodes in the closed list are updated to the minimum value obtained by taking the sum of the shortest distance between the closed and open node and the heuristic of the open node. Figure 4 illustrates the pseudo-code of LRTA*. We used Breadth-First Search Method to calculate the shortest distance between the two nodes as the cost of moving from one node to the adjacent node is constant. Any search method (like Dijkstra's shortest path) can be used in place of BFS. The properties of RTAA* holds true for LRTA* as well. Additionally, with a look-ahead of one, RTAA* behaves exactly same as LRTA* given identical tie-breaking. However, if they have larger look-ahead, then LRTA* is more informed than RTAA*. The more informed nature of LRTA* is suppressed by the fact that it takes more time to update the heuristic values and it is also difficult to implement. So, evaluation of both the methods is required to have an in-depth knowledge about the trade-off between the fast solution and the minimum cost trajectory.

Lifelong Planning A*

Lifelong Planning A* (a variant of incremental A*) is the combination of two different search techniques - A* and Dynamic SWSF-FP. Like in A*, the heuristics need to be consistent and admissible, i.e., the heuristic of a state s via state s' is not greater than the sum of the heuristic of state s' and the distance between the two states: $h[s] \leq cost(s, s') + h[s']$. Along with the *g-values*, LPA* maintains another estimate cost (*rhs*) from the start state for every node in the graph. The *rhs* cost is one-step look-ahead value based

Algorithm 4 Learning Real-Time Adaptive A*

```
procedure LRTAstar
    policy = Astar(start)
    found = False
    while not found do
        Follow policy until a change is detected at current node
        if change detected then
            for nodes1 in closed do
                temp = Infinity
                for nodes2 in open do
                    dist = BFS(nodes1,nodes2)
                    temp = min(temp, dist + h[nodes2])
                end for
                h[nodes1] = temp
            end for
            policy = Astar(current)
        else
            found = True
        end if
    end while
end procedure
```

Figure 4: Learning Real-Time A*

Algorithm 5 Lifelong Planning A*

```
procedure CalculateKey(s)
    Return [min(g[s],rhs[s]) + h[s], min(g[s],rhs[s])]
end procedure
procedure Initialize
    Initialize U to null and g, rhs to infinity,
    rhs[start] = 0
    U[start] = CalculateKey(start)
end procedure
procedure UpdateVertex(u)
    if u != start then
        among all predecessors s' of u rhs[u] = min(g[s'] + cost)
    end if
    if u in U then U.remove(u)
    end if
    if g[u] != rhs[u] then U[u] =CalculateKey(u)
    end if
end procedure
procedure ComputeShortestPath
    Sort U with priority in values
    while U.TopKey != CalculateKey(goal) or rhs[goal] != g[goal] do
        u = U.pop()
        if g[u] > rhs[u] then
            g[u] = rhs[u]
            UpdateVertex(u') for all successor nodes u' of u
        else
            if u not an obstacle then
                g[u] = infinity
                UpdateVertex(u') for u and all successor nodes u' of u
            end if
        end if
    end while
end procedure
procedure LifelongPlanningAstar
    Initialize( )
    found = False
    while not found do
        found = ComputeShortestPath( )
        Follow policy until a change is detected at current node
        if change detected then
            for u in changed nodes do
                for adjacent nodes v to u do
                    g[v] = infinity
                    UpdateVertex(v)
                end for
            end for
        else
            found = True
        end if
    end while
end procedure
```

Figure 5: Lifelong Planning A*

on the *g-values*, thus more informed than *g-values* alone.

$$rhs(s) = min(g(s') + c(s', s))\forall s' \in Pred(s)$$

A node is inconsistent if value *g(s)* is not equal to the value *rhs(s)*. After a change in edge cost, LPA* makes only those nodes consistent which are relevant for computing the shortest path, making the search faster. The *g-value* of a node gives the distance of the node from the start distance, iff all the nodes are consistent. LPA* maintains a priority list *U*, which contains the list of all the inconsistent nodes. Like in A*, LPA* also has *g-costs*, *h-costs* and *f-costs (g(s) + h(s))*. It also expands the node with the minimum *f-cost* first and uses smallest *g-cost* node in order to break the ties between the nodes with the same *f-cost*. The pseudo-code of LPA* is shown in Figure 5. LPA* works as follows: It first calls the *initialize* function which initializes all the variables required in the algorithm and sets the *rhs* of start node to 0, making the start node inconsistent and adds it to the priority list *U*. The first search of LPA* by calling the *ComputeShortestPath* function is same as A* search, starting from the start node. If there is no change in the edge costs, then it terminates with the A* path, otherwise, it updates the changed vertex and the adjacent vertices. This process continues till the agent reaches the goal state. In *ComputeShortestPath*, vertices in priority list are updated till the node with minimum *f-cost* becomes greater than or equal to the *f-value* of the goal state or the *rhs-cost* of goal state becomes equal to *g-cost*. If a node with minimum *f-cost* becomes greater than or equal to the *f-value* of the goal state, then there is no path possible from the start state to the goal state. A vertex is locally over-consistent if and only if $g(s) > rhs(s)$ and under-consistent if and only if $g(s) < rhs(s)$. If a vertex becomes over-consistent, set its *g-cost* to *rhs-cost* and update all the successive nodes by calling *UpdateVertex*. If a vertex becomes under-consistent, set its *g-cost* to *infinity* and update all the successive nodes and the current node. After all the updates, the agent can trace back from the goal state to the start state by following the *g-values* in decreasing order, similar to A*. LPA* repetitively calculates the shortest path between the start node and the goal node.

D* Lite

D* Lite is like D* except in using only one tie-breaking scheme for simplification. D* Lite is built on the LPA* algorithm and repetitively calculates the shortest path between the current node and the goal node. In LPA*, the search starts from the start state to the goal state, but in D* Lite, the search starts from the goal state to the start state and the *g-costs* are estimates of the goal distance. D* Lite is implemented by reversing the search direction and edge directions of LPA*. The pseudo-code of D* Lite is depicted in Figure 6. LPA* is computationally very expensive if we have a very large number of nodes in the map because the priority queue needs to be sorted every time a change is detected in the path. In contrast, D* Lite adds the new priorities in the list with modified value in the first component. In this way, the priority list does not require sorting because the new priorities are larger and do not interfere with already added nodes in the priority list. Also, it does not require back-propagation from goal state

Algorithm 6 D* Lite

```
procedure CalculateKey(s)
    Return [min(g[s],rhs[s]) + h(s,start) + k_m, min(g[s],rhs[s])]
end procedure
procedure Initialize
    Initialize U = null, k_m = 0 and g, rhs = infinity,
    rhs[goal] = 0
    U[goal] = CalculateKey(goal)
end procedure
procedure UpdateVertex(u)
    if u != start then
        among all successors s' of u rhs[u] = min(g[s'] + cost)
    end if
    if u in U then U.remove(u)
    end if
    if g[u] != rhs[u] then U[u] =CalculateKey(u)
    end if
end procedure
procedure ComputeShortestPath
    Sort U with priority in values
    while U.TopKey != CalculateKey(start) or rhs[start] != g[start] do
        k_old = U.topKey()
        u = U.pop()
        if k_old < CalculateKey(u) then
            U[u] = CalculateKey(u)
        else if g[u] > rhs[u] then
            g[u] = rhs[u]
            UpdateVertex(u') for all predecessor nodes u' of u
        else
            if u not an obstacle then
                g[u] = infinity
                UpdateVertex(u') for u and all predecessor nodes u' of u
            end if
        end if
    end while
end procedure
procedure DstarLite
    last = start
    Initialize( )
    found = ComputeShortestPath( )
    while not found do
        Follow policy until a change is detected at current node
        if change detected then
            k_m = k_m + h(last, start)
            last = start
            for u in changed nodes do
                for adjacent nodes v to u do
                    g[v] = infinity
                    UpdateVertex(v)
                end for
            end for
            found = ComputeShortestPath( )
        end if
    end while
end procedure
```

Figure 6: D* Lite

to start state as we can directly compute the path greedily moving from start to goal state.

Evaluation

Illustration of Forward A*, RTAA* and LRTA* Searches

A simple example to illustrate the working of Forward A*, RTAA* and LRTA* is shown in Figures 7. Figure 7a depicts the actual world model, where the flag represents the goal state and walls represent obstacles, which are currently unknown to the robot. In this example, we assume that the robot can sense the obstacles on four adjacent blocks only. Figure 7b shows the two iterations of Forward A* searches. Every block has three values inside it-top value is the f-cost, middle value is the g-cost and the bottom value shows the

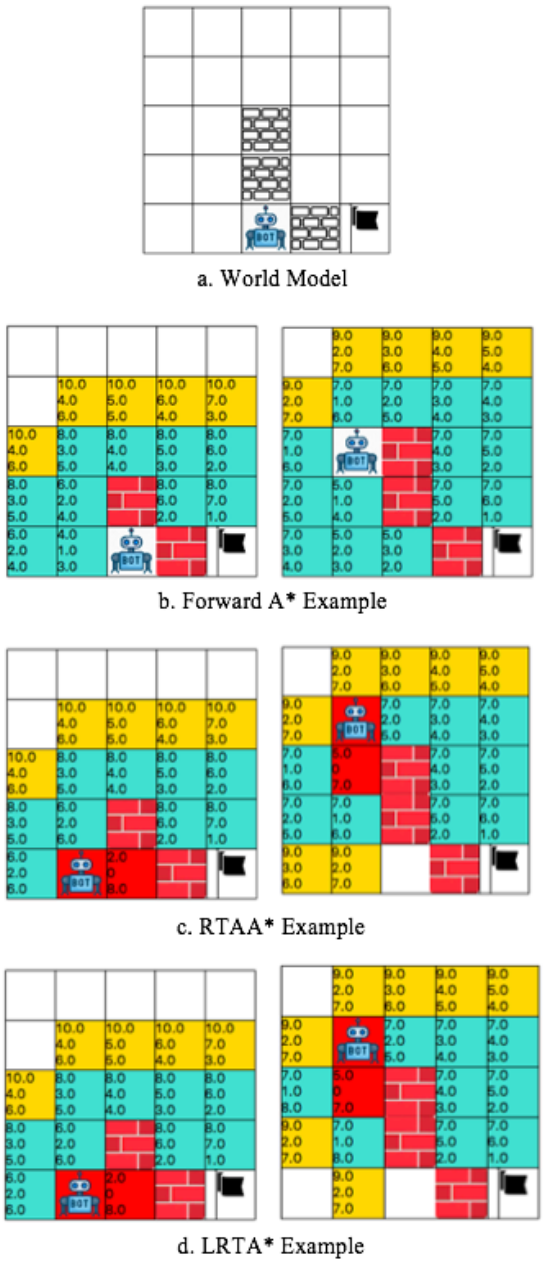

Figure 7: Grid World Example

h-cost. In the first iteration, the robot plans the path by sensing only two adjacent obstacles. After planning, the robot starts moving towards the goal but finds a new obstacle. After sensing the new obstacle, the robot plans again from the scratch and the total number of explored nodes in Forward A* reaches 24. Figure 7c shows the RTAA* searches and Figure 7d shows the LRTA* searches. All three algorithms have exactly the same first search and expand identical 10 nodes. However, RTAA* expands 21 nodes in total and LRTA* expands a total of 20 nodes. The bottom values in the blocks of LRTA* and RTAA* shows the up-

dated heuristic costs. If more than one cell has the same key, then the preference order for expansion is up, left, bottom and right. RTAA* and LRTA* has less number of expanded nodes than Forward A* because RTAA* and LRTA* perform an informed search after sensing the new obstacle. Also, LRTA* has less expanded nodes than RTAA* because the updated heuristic of LRTA* has a higher value (more informed) than the updated heuristic in RTAA*. To compare and evaluate the performance of the search algorithms, we will consider execution time, the number of explored nodes and the trajectory cost to reach the goal as our metrics. Let us now discuss each of them in the following section.

Time Complexity

Grid Size	Dynamic Path Planners			
	10	*20*	*50*	*100*
Forward A*	0.41	8.3	42.1	83.7
RTAA*	0.32	1.42	24.3	73.6
LRTA*	0.45	7.79	26.2	78.1
LPA*	0.35	6.67	27.2	79.4
D* Lite	0.28	1.85	16.6	62.2

Figure 8: Time Complexity on Grid Size

All the experiments are conducted on a discrete world map where the grid size varies from 10*10 to 100*100. The robot has no information about the world and can sense only adjacent blocks. Obstacles are added in the grid world randomly and the same generated grid is used for all the five algorithms to compare the results. As the size of the grid increases, the time taken by the Dynamic A* variants also increase. Figure 8 shows that Forward A* is the most inefficient algorithm among the five algorithms in terms of time complexity. This is justified by the fact that Forward A* performs the uninformed search. Comparison in terms of the time complexity reveals that LPA* performs slightly better than the Forward A*, LRTA* is better than LPA* while D* Lite outperforms all of them. We have used RTAA* with the look-ahead value of infinity which suppresses the performance of RTAA*. In practice, RTAA* gives the result in real-time with only some specific look-ahead value which is determined empirically based on the application domain. If the domain knowledge is provided in advance, then the look-ahead value for RTAA* can be determined to make it run in real-time. Our experimental results show that D* Lite is the most efficient algorithm based on the time complexity.

Explored Nodes

Figure 9 shows the quantitative results with the number of nodes explored by each algorithm for different grid sizes. With the increase in the number of explored nodes, the space complexity also increases as we need to store the nodes in the priority list. In Forward A*, explored nodes increase exponentially with the increase in grid size. In other search methods, the increase in the number of explored nodes is still polynomial with the grid size in terms of space complexity. D* Lite, followed by LRTA* is highly effective in the number of explored nodes. LRTA* updates the heuristic value to a larger value which makes it more informed. Also, D* Lite uses the variable k_m which adds the heuristic cost from current state to the goal state to the first component of the updated edge key. We can see the direct relationship between the time complexity and the number of explored nodes.

Grid Size	Dynamic Path Planners			
	10	*20*	*50*	*100*
Forward A*	472	1988	30669	155443
RTAA*	227	1924	30440	92595
LRTA*	373	1918	14905	69331
LPA*	354	1965	16791	82533
D* Lite	445	1853	11331	41566

Figure 9: Explored Nodes

Trajectory Costs

Trajectory Cost is the cost incurred while going from the start state to the goal state via intermediate states. Trajectory cost should be minimum for an algorithm to be efficient. An algorithm is expensive w.r.t. to the trajectory cost if the difference between the computed and the optimal trajectory cost is large. Forward A* always computes the shortest path, followed by LPA*. However, RTAA* algorithm with a look-ahead of infinity performs the worst among all the algorithms. The trajectory cost of RTAA* can be reduced with a smaller look-ahead value.

Grid Size	Dynamic Path Planners			
	10	*20*	*50*	*100*
Forward A*	29	71	273	402
RTAA*	31	77	294	450
LRTA*	29	71	283	423
LPA*	29	71	273	428
D* Lite	29	77	273	423

Figure 10: Trajectory Costs

Conclusion

In this paper, we have implemented and compared the performance of five different dynamic path planning algorithms - Forward A*, RTAA*, LRTA*, LPA* and D* Lite. As shown, RTAA* and LRTA* plans multiple short trajectories towards the goal while D* Lite starts its search from the goal towards the start in order to make the heuristics more informed. Our experimental results show that RTAA* is a good choice in time-critical applications whereas the Forward A* always provides the minimum cost trajectory. Note that these search algorithms cannot be evaluated solely based on a single evaluation criterion. We, therefore, provided a comprehensive comparison of these algorithms based on the three different evaluation metrics. Later it was observed that D* Lite outperforms the rest of the dynamic path planning algorithms explored in this paper. Path planning in a dynamic environment is a vast area of research.

This paper is presented with the hope to provide a strong foundation for further research and development in the area of Dynamic Path planning.

References

S. Koenig; and M. Likhachev. 2005. Adaptive A*. In *Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems*, 1311-1312.

S. Koenig; and M. Likhachev. 2002. D* Lite. In *Proceedings of the National Conference on Artificial Intelligence*, 476-483.

S. Koenig; and M. Likhachev. 2006. Real-Time Adaptive A*. In *Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems*.

D. Connell; and H. Manh La. 2017. Dynamic Path Planning and Replanning for Mobile Robots using RRT*. *IEEE International Conference on Systems, Man and Cybernetics*.

J. van den Berg; D. Ferguson; and J. Kuffner. 2006. Anytime path planning and replanning in dynamic environments. In *Proceedings of the IEEE International Conference on Robotics and Automation*.

Y.K. Hwang; and N. Ahuja. 1992. A Potential Field Approach to Path Planning. *IEEE Transactions on Robotics and Automation*.

C.W. Warren. 1989. Global path planning using artificial potential fields. In *Proceedings of IEEE International Conference on Robotics and Automation*.

Y. Li; C. Li; and Z. Zhang. 2006. Q-Learning Based Method of Adaptive Path Planning for Mobile Robot. *IEEE International Conference on Information Acquisition*.

B. Zuo; J. Chen; and L. Wang. 2014. A reinforcement learning based robotic navigation system. *IEEE International Conference on Systems, Man and Cybernetics*.

J. Park; J. Kim; and J. Song. 2007. Path Planning for a Robot Manipulator based on Probabilistic Roadmap and Reinforcement Learning. *International Journal of Control Automation and Systems*.

X. Zhang; Y. Zhao; N. Deng; and K. Guo. 2016. Dynamic Path Planning Algorithm for a Mobile Robot Based on Visible Space and an Improved Genetic Algorithm. *International Journal of Advanced Robotic Systems*.

G. Ramalingam; and T. Reps. 1996. An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms*.

S. Koenig; and M. Likhachev. 2001. Incremental A*. In *Proceedings of the Neural Information Processing Systems*.

V. Bulitko; and G. Lee. 2005. Learning in real-time search: A unifying framework. *Journal of Artificial Intelligence Research*.

P. Hart; N. Nilsson; and B. Raphaelv. 1968. A formal basis for the heuristic determination of minimum cost paths. In *IEEE Transaction on Systems Science and Cybernetics*, 100-107.

S. Koenig. 2004. A comparison of fast search methods for real-time situated agents. In *Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems*, 864-871.

R. Korf. 1990. Real-time heuristic Search. *Artificial Intelligence*, 253-279.

A. Stenz. 1995. The focused D* Algorithm for real-time replanning. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1652-1659.

S. Koenig; C.Tove; and Y. Smirnov. 1990. Performance bounds for planning in unknown terrain. *Artificial Intelligence*, 253-279.