Intro to Algorithmic Robotics: D* Lite

Zihan Li and Karahm Kim

I. INTRODUCTION

For our EECS 498: Algorithmic Robotics final project, we chose to implement a D* Lite algorithm for plath planning. This path planning algorithm makes use of the idea of incremental search methods. The nature of incremental search algorithms takes advantage of previous searches in order to find solutions to similar search tasks instead of having to start over from scratch, which would be required for other methods such as A* with modifications to account for collision. The D* Lite algorithm itself is based off of Lifelong Planning A* *LPA*$*$ and D*, making use of characteristics from both algorithms in order to produce an algorithm that is substantially shorter to implement than D*, while just as efficient, if not better.

A. MOTIVATION

With robotic applications to the real world, there are a number of issues that arise with some of the path planning algorithms that we have explored throughout the class. With algorithms such as probabilistic road mapping or A*, it is possible to efficiently and quickly find a solution to go from a starting configuration to an ending one. However, it requires either perfect knowledge of the environment, some means of perfectly calculating whether or not a given configuration would cause a collision in the environment, or a perfect model of the environment to sample from and use to calculate a path. However, it is not always possible to fulfill these requirements, provide the necessary information, or it would not be efficient or possible to create a model of the situation to sample and discover a path from a given start configuration to a goal configuration. The strength of the D* Lite algorithm is displayed in this case. For D* Lite, it assumes imperfect knowledge of the world from the start. Given that there is some way to interpret the start configuration and the goal configuration, it works under the basis of a ”cell” or ”grid” coordinate system. This system is used to interpret the environment, taking into account known obstacles in the environment. After this, it assumes that all cells of unknown status are traversable and follows the shortest path until it reaches the goal. However, if it comes across an observes an untraversable cell, it recalculates the new shortest path from it's current cell to the goal cell. This makes it possible for a robot to dynamically learn the environment and find a path from its' starting configuration to its' goal configuration given some method of scanning the environment for viable configurations. This is useful as it also allows one to find a path through a dynamically changing environment, which is not accounted for in algorithms such as A*. This algorithm can be applied to a number of different uses such as path planning for a robot. It could be adapted for use in exploring and learning an environment through setting a series of goal points and having a robot move through the terrain to collect data.

II. IMPLEMENTATION

D* Lite is an algorithm that applies the ideas from D* to create an extension of the Lifelong Planning A* algorithm in order to perform similarly, if not better than D*. For our specific implementation, we implemented a graph of nodes and edges between adjacent nodes. For each node, we keep use metrics of configuration distance to the goal, a one step lookahead distance, along with the a along with the cost to move from one node to the next for adjacent nodes. In our implementation, we defined adjacency as a four connected graph with respect to our coordinate system. Our implementation can be split up into three major sections - the priority queue, the graph, and the D* Lite algorithm implementation itself.

A. Priority Queue

Per the requirements of the D* Lite algorithm, it was necessary for us to implement our own priority queue, as the one provided by python did not fulfill our needs. To fit out needs, we implemented a heap-based priority queue. One requirement for our priority queue was a custom comparator to determine the priority. We calculated the priority according to keys that were generated for each node in the graph.

```
def calculateKey(node):
   return [min([node.g, node.rhs]) +
       heuristic(grid.start_n_no_change,
       node) + k_m, min([node.g, node.rhs])]
```

The keys contain two components. The first component is calculated based on a $min(g(s), rhs(s)) + h(s_{start}, s) + k_m$, where s is a node in the graph, as outlined in the function above. $g(s)$ is an estimate of the shortest path from the given s_{goal} node to the current node s. $h(s_{start}, s)$ is a heuristic that estimates the difference between the current node s and the original start node s_{start}. The variable k_m is a global variable that assists in guiding the robot along an unblocked path towards the goal. While moving towards the goal, when a path that was previously known as traversable is found to be untraversable, or an untraversable path is found to be traversable, the value of k_m is incremented. This variable is used to push the path seeking algorithm away from local minimums. Similar to how algorithms such as potential fields can get stuck in a local minimum, the same is possible for the path planning algorithm. However, this variable prevents that by pushing it away from these local minimums when an obstacle is found. For our implementation, we used the following heuristic to calculate the approximate distance from one node to another.

```
def heur(n1, n2):
   return sqrt( (n1.x - n2.x)^2 + (n1.y -
       n2.y)^2 + min(abs(n1.theta -
       n2.theta), 2 * pi - abs(n1.theta -
       n2.theta))^2)
```

A few other functions were required for managing the priority queue. For our priority queue U, we needed functionality provided by the following methods:

- $U.Top()$, which returned an array containing the key associated with a node in position 0 and the node itself in position 1.
- $U.Pop()$, which removes the top node and its corresponding key from the priority queue
- $U.Insert(s,k)$, which inserts node s into the priority key with the corresponding key k according to its priority in the queue
- $U.Update(s,k)$, which updates the node s with the priority of key k and updates the order of the priority queue correspondingly.
- $U.Remove(s)$, which removes node s from the priority queue.

The custom comparator that we used for ordering the priority queue is outlined below.

```
def pqComparator(lhs,rhs):
   if lhs[0][0] > rhs[0][0]:
      return True
   elif lhs[0][0] == rhs[0][0]:
      return lhs[0][1] > rhs[0][1]
   else:
      return False
```

Since there is no priority queue implemented in Python that contains all the required functionality, we implemented our own priority queue. We implemented a binary priority queue with time complexity of $U.Pop()$, $U.Insert(s,k)$, $U.Update(s,k)$, $U.Remove(s)$ all as $O(log(n))$.

B. Graph

To simplify our implementation, we created an alternate coordinate system. To deal with floating point errors in python and for ease of use, we converted everything to a 3D, one based coordinate "grid" and placed our goal location as $(0,0,0)$. Every change of 1 in the x and y coordinates was equivalent to an change of 0.1 within the x and y degrees of freedom for our robot. Every change in one for the z coordinate was equivalent to a change of $\frac{\pi}{2}$ radians for each configuration. We also bounded this value from 0 to 3 in order to bound $\theta \in [0, 2\pi)$, where θ is the rotation of the robot. As a result of this configuration, we stipulated that adjacency between two coordinates wraps around for the third coordinate, meaning the $(x,y,0)$ and $(x,y,3)$ would be considered as adjacent nodes. Our implementation of the graph was used to to conveniently hold all of the nodes that were used in our algorithm, as well as to allow for easy access to nodes based on our translated coordinate system through a python dictionary. With our graph structure, we also included another dictionary that took in two sets of ordered pairs that mapped to a scalar value that represented the cost from moving from one node to the next. The cost was calculated similarly to the heuristic used to calculate the distance between two configurations. We used the value of ∞ to represent when there was an obstacle on the map, preventing that configuration from being able to be achieved.

C. Node

For each node that was located in the graph, each node had a g values and an rhs value that was kept track of. The value of $g(s)$ denotes the distance from node s to the goal node, while $rhs(s)$ is a one step

lookahead that checks adjacent values and minimizes the cost, providing potentially better informed values than the g-values. For our specific implementation, we chose to implement a 4-connected algorithm to improve efficiency and speed of the algorithm by reducing the number of calculations that need to be made for each node. We chose this to mimic a robot in an unknown environment with only a proximity sensor to detect collision, as this was what we thought would be the most interesting method to see the behavior for. The method for calculating the *rhs*-value is shown below.

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

The default initialization for each node was $rhs(s) = g(s) = \infty$.

D. D* Lite Algorithm

```
def scanEdges():
    edgeChange = False
   for neighbor in
       current_s_start.getNeighbors():
          if edgeHasChanged: # (Coll and
              cost(s_start,neighbor) != inf)
              or (NoColl and
              cost(s_start,neighbor) == inf)
            if not edgeChange:
              edgeChange = True
               k_m +=
                   heur(initial_s_start,s_last)
             for changeNeighbor in
                 neighbor.getNeighbors():
              update cost(changeNeighbor,
                  neighbor) # inf if
                  collision or configuration
                  distance if no collision
               updateVertex(s_start)
    return edgeChange

def calculateKey(s):
   return [min(g(s),rhs(s)) + h(s_start,s) +
       k_m, min(g(s),rhs(s)]
def initialize():
   U = PriorityQueue()
    k_m=0
    rhs(s_goal)=0
    U.insert(s_goal,calculateKey(s_goal)):
def updateVertex(s):
   if u != s_goal:
     rhs(u) = min([c(u,s) + g(s) for s in
         u.getNeighbors()])
    if U.exists(u):
     U.Remove(u)
    if g(u) != rhs(u):
     U.Insert(u, calculateKey(u))
def computeShortestPath():
   while(U.Top()[0] < calculateKey(s_start)
       or rhs(s_start) != g(s_start):
     k_old = U.Top()[0]
      u = U.pop()
      if k_old < calculateKey(u):
       U.Insert(u, calculateKey(u))
      else if g(u) > rhs(u)
       g(u) = rhs(u)
          for neighbor in u.getNeighbors():
            updateVertex(neighbor)
      else
       g(u) = infinity
          for neighbor in u.getNeighbors():
            updateVertex(neighbor)
def Main():
   s_last = s_start
    initialize()
    computeShortestPath()
    while(s_start != s_goal):
      s_start = argmin([c(s_start,s) + g(s)
          for s in s_start.getNeighbors()])
       Move to s_start
       newEdges = scanEdges(s_last)
       if newEdges:
        s_last = s_start
          computeShortestPath()
```

The D* Lite algorithm that we implemented is outlined in pseudo code above, which is adapted from the pseudo-code outline in the original D* Lite paper[1]. Our implementation follows the pseudo-code outlined above, with a few changes in it according to our specific implementation of the algorithm and helped functions that were used for convenience.

III. RESULTS

Now we want to compare A* and our implementations of D* Lite. We used the experiment setup as shown in Fig. 1. Starting point of the robot was where the robot was standing, and goal point was close to Table B. As shown in Fig. 2, we changed the the position of table A and B immediately after the robot started to move. Table B will block the robot's way to goal as both algorithm will start by going straight to the goal position. In our graph, red points are nodes that have collisions, black points are the path of the robot, and yellow points are nodes whose value got updated.

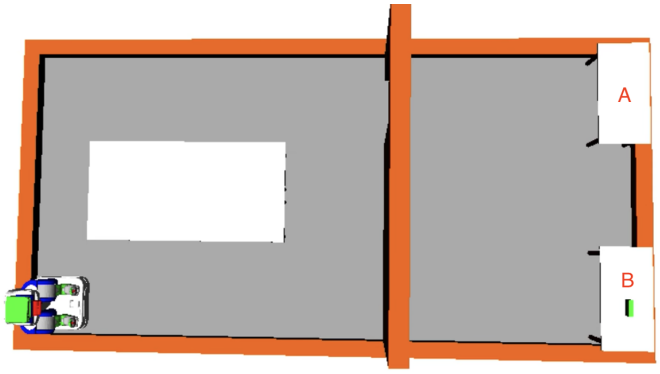

Fig. 1. Initial setup of the environment in OpenRave

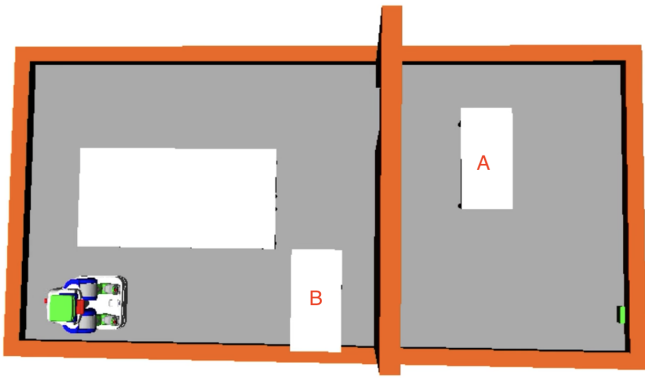

Fig. 2. We changed the the position of table A and B immediately after the robot started to move.

Since A* algorithm does not have the ability to update path, we can see the robot collided with table B in Fig. 3.

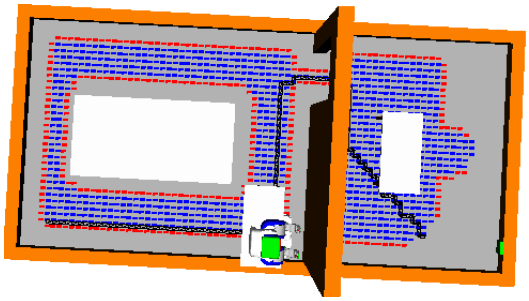

Fig. 3. Robot collided with table A.

We compared the performance of our implementations of D* Lite. While both of them successfully found the path to the goal as we change the configuration of the environment, grid implementation was 8 times faster than graph implementation. But the grid method requires a predefined map area, while the graph method does not. There is a trade-off between efficiency and scalability in the implementation.

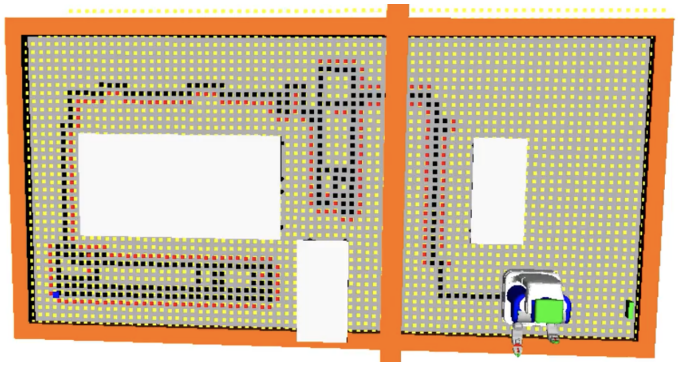

Fig. 4. D* Lite successfully found the path.

After that we constructed a more complicated experiment to test our algorithm: After the robot successfully got away from the local minimum created by table B, we moved table B on top of the goal so that the robot can never get to the goal, as shown in Fig. 5 and Fig. 6. D* Lite will update where was originally marked as collisions during it's search as shown in Fig. 8 and Fig. 9. And the algorithm finally found the path as shown in Fig. 9.

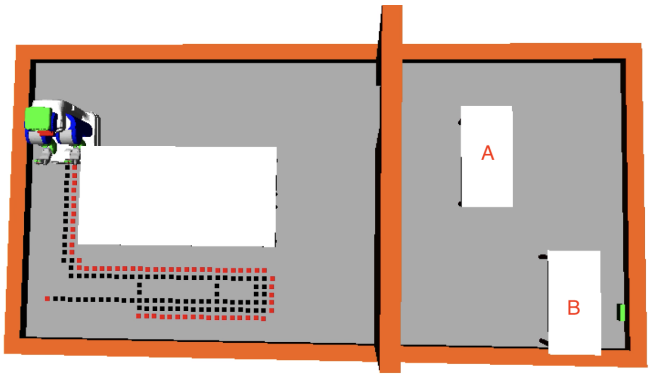

Fig. 5. Moved table B on top of the goal

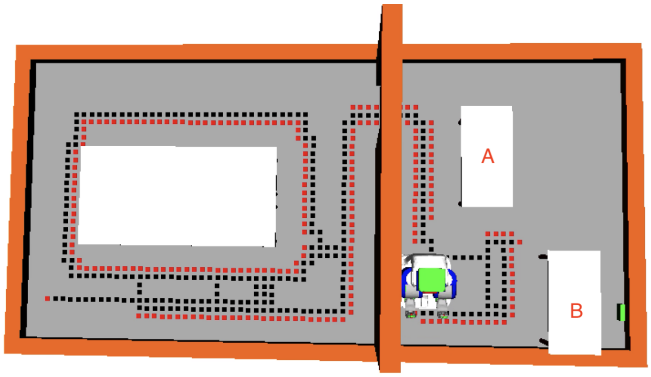

Fig. 6. Robot cannot get to the goal point

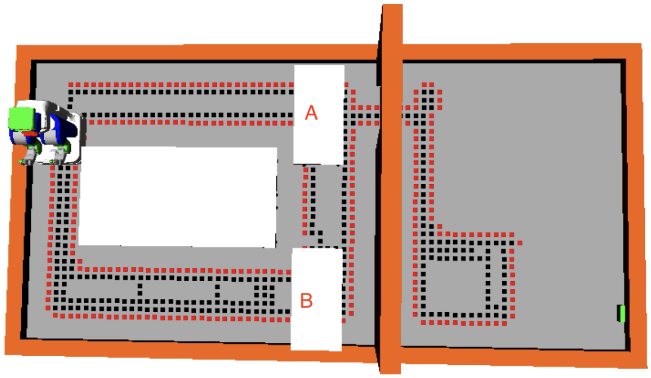

Fig. 7. Moved table A and B back to block the robot's way to goal

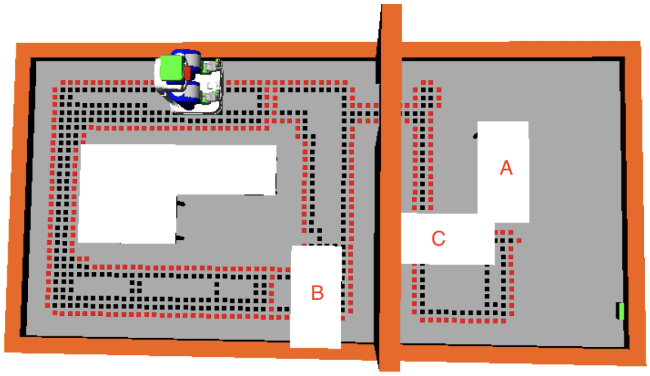

Fig. 8. Moved table A and C to block the found way of getting closer to goal.

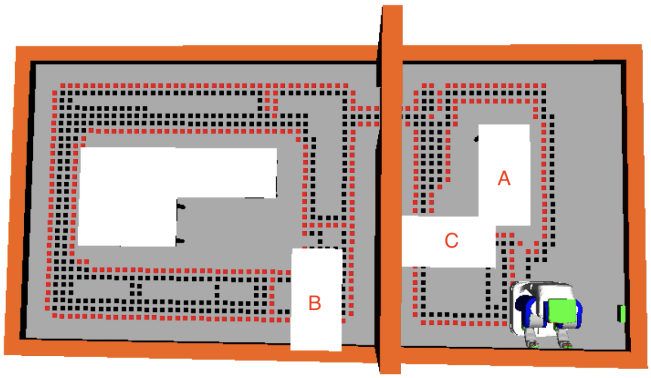

Fig. 9. D* Lite successfully found the path.

IV. DISCUSSION

As the first version, we implemented d-star by using a grid based system. However, we found that while it significantly improved the speed of the algorithm, it was rigid and inflexible to an increase in expected map size, which would be an issue in extremely large spaces. Due to this, we chose to change our implementation to one that is dictionary based. This caused a slow down of speed, but it added much more flexibility to the algorithm. There was another algorithm outlined in the D* Lite paper called D* Lite Optimized, which provided a number of optimizations to speed up the functionality of the algorithm. We were unable to tackle implementing this algorithm due to time constraints however, so we were not able to test the amount of improvement that it would provide overall.

APPENDIX

Here is a video demo: https://www.youtube.com/watch?v=yh8xjHrITVE.
GitHub Repo: https://github.com/lizihan021/d-star.

REFERENCES

[1] S. Koenig and M. Likhachev. D* Lite. In Proceedings of the AAAI Conference of Artificial Intelligence (AAAI), 476-483, 2002.