**An Implementation of D\* Lite**

Jerry Feng and Albert Yang

ME132b Final Project

# **1. Introduction**

In the field of robotic path planning, a major challenge lies in autonomous navigation of unknown territory. One approach to the problem of navigating in an unknown room is the D\* family of algorithms. This family of algorithms works by iteratively generating a map, or occupancy grid, which breaks down the environment into unoccupied, occupied, and unknown cells. The algorithms then perform graph searches to generate the minimum-cost paths from origin to target. The robot can then travel along that path, all while dynamically updating the map and therefore the path along the way.

D\* Lite is different from D\* primarily in the method of path search. In the original D\* algorithm, cells in an occupancy grid store references to its predecessor and the cost from the current cell to the goal. D\* uses a raise and lowering operator for each set of blocks within the occupancy grid. This process, known as expansion, iteratively selects open nodes and evaluates their cost values. All changes made to a node are then recursively propagated to its neighbors.

The D\* Lite algorithm forgoes this and uses an implementation of Lifelong Planning A\*. After the D\* Lite algorithm generates an occupancy grid and then performs the Lifelong Planning A\* algorithm to find the optimum path to the target. By dynamically updating its occupancy grid, and running the pathfinding algorithm A\* over all known free spaces, the algorithm is guaranteed to find a solution to the goal provided one exists. D\* Lite has several advantages over D\*, namely cost efficiency and simplicity in implementation. For this reason, we implement a practical application of D\* Lite to be used on a Kobuki mobile base station with differential velocity drive.

In this final report, we will discuss our implementation of D\* Lite and the issues that had to be addressed when going from Koenig’s original virtual implementation to a real-world implementation.

# **2. Implementation**

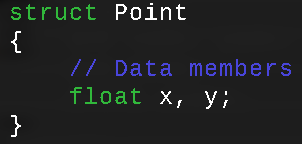
## 2.1. Code

Our implementation of D\* Lite is written in C++. Below we give a brief description of the central class (Map) and the function call flow of the implementation:

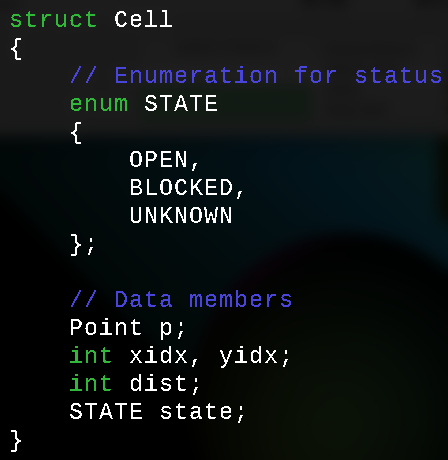
Map

1. A vector of Cells (map) of size m\*n that is the representation of the occupancy grid with dimensions m x n.
2. A vector of Points (path) is a list of waypoints for the robot to navigate.
3. A sequence of four floats (xmin, ymin, xmax, ymax) that define the outer boundaries of the occupancy grid.
4. A float (res) that defines the “resolution” (or granularity) of the map. Each grid block in the map is res x res meters in size.
5. A pair of ints (Nx, Ny) that store the number of cells in the occupancy grid in the x and y directions respectively.
6. An int maxDist that stores the maximal distance a cell is from the goal point.
7. A Point (goal) that represents the goal point in real-world coordinates
8. A Point (goal\_idx) that represents the goal point in occupancy grid coordinates.

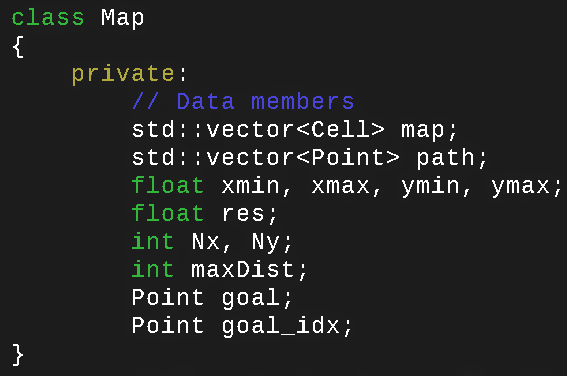
where the Point structure is a 2D vertex (with members x and y), and the Cell structure is a representation of a single member of the occupancy grid (which stores the member’s state, real-world coordinates, occupancy grid coordinates, and distance from goal.



*Figure 1: Structure definition of Point.*



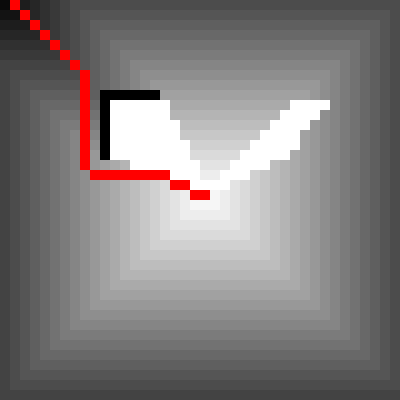
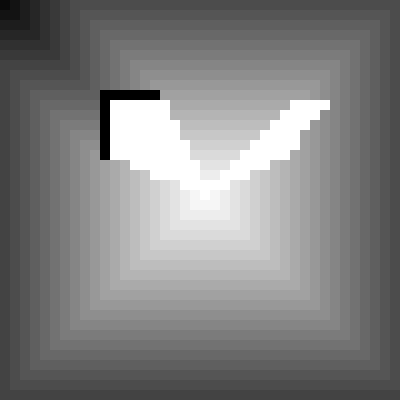
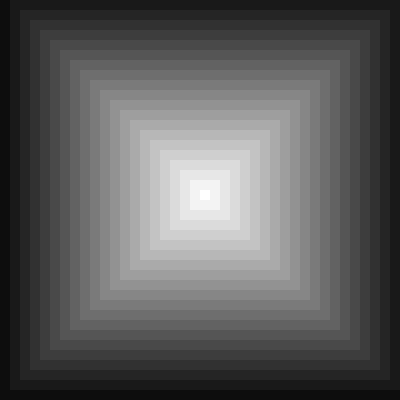
*Figure 2: Structure definition of Cell.*



*Figure 3: Class definition of Map.*

To use map, we wrote the following member functions, which are designed for easy interfacing with the robotic navigation code:

1. **init(Point)** which takes the goal as its input and initializes the cells in the occupancy grid with their indices and distance to goal.
2. **setBlocked(Point, Point)** and **setOpen(Point, Point)** which are used to update the occupancy grid. Both functions take source and destination points use Bresenham’s Line Algorithm to set cells along the resulting pixelated line to the open state. While this is where setOpen ends, setBlocked also sets the endpoint to blocked at the end.
3. **recalculate()** which takes the stored goal point in the map and recalculates the distances of the cells to the goal, taking into account the presence of obstacles
4. **AStar(Point)** which takes the current position of the robot and uses the map’s current occupancy grid to find the shortest path to the goal and stores the result into the map class’s path member.
5. operator<< (ofstream&, Map), which outputs the occupancy grid as an image. Occupied cells are drawn black. Open cells are drawn white. Unknown cells are drawn grey, scaled to the distance to the goal (darker is further). Finally, the waypoint path is outlined in red.



*Figure 4: Intermediary steps in between running the D\* Lite algorithm. On the left is the occupancy grid immediately following initialization. In the middle is a the updated grid after some example calls to setBlocked and setOpen and the changes in the the distances of the cells are reflected. On the right is the path found by AStar(), which avoids pathing immediately next to the obstacle.*

Thus, the flow of code should first call the map constructor, then initialize it with a user-defined start point. Then, in the robotic navigation loop, update the occupancy grid based on the rangefinder readings. When all readings are accounted for, recalculate the occupancy grid and run A\* on the robot’s current position. Finally, extract the waypoints from the map and navigate the robot towards the waypoints given.

## 2.2. Implementation decisions

A variety of compromises had to be made due to the constraints of the robot with which we were working.

**2.2.1. Incomplete LPA\***

Firstly and most notably was the implementation of an incomplete LPA\*. In LPA\*, only the cells potentially affected by occupancy grid updates are considered for recalculation of distance to goal. However, we used a full recalculation of the map. This function was originally used to show that the following A\* implementation was functional. However, due to time constraints, we were unable to get the update rule as described by LPA\* working -- running the robot in simulation would cause the recalculation of the occupancy grid to run into an infinite loop. We suspect that this error is caused by improper handling of the goal when updating the map, resulting in distances of cells to be unresolvable. Because we were unable to successfully debug the function before the lab session, we fell back to the code we previously wrote. Our attempt at implementing the update rule of LPA\* can be found as the function updateDistances() found in map.cpp. However, settling for this compromise means that our computation time takes a large hit.

As a consequence of the unoptimized LPA\* and the limited processor speed, the algorithm runs too slowly to be feasible in real time, which is defined to be matching the update rate of the robot (30 Hz). Currently, the occupancy grid update takes on the order of .1 seconds, and the graph search takes on the order of .5 seconds. As a result, it is unreasonable to run both algorithms at every possible point.

To adjust for the operation time of the algorithm, a series of further modifications were made to improve performance in the lab.

**2.2.2. Occupancy Grid Voting Removed:**

A voting system which used an exponential function to calculate probabilities of occupancy at each section of the grid was computationally expensive. Since it had to be run many times over the course of the laser scan, it was not feasible to implement this. As a result, it was removed.

An added bonus of this approach is that the program will be able to handle moving obstacles and vanishing obstacles with ease, as it will not take a significant number of measurements to overcome the previously established voting system. The primary downside of this adjustment is that the rangefinder data will be significantly more prone to noise.

**2.2.3. Rangefinder Data Sampling Rate:**

The rangefinder comes with roughly 512 angular measurements. Rather than using all 512 measurements, every 4th, 6th, or 8th was taken, depending on the block size of the occupancy grid. This would drastically reduce the amount of data the processor would have to handle, and would increase the program’s speed.

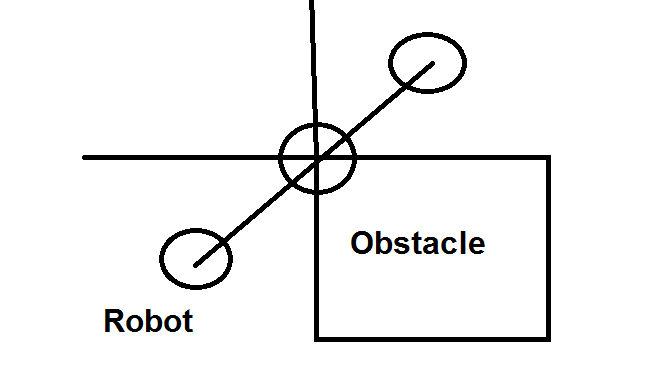
In addition, several changes were made that were not mentioned in the original literature. It is believed that in practice, measures like these are necessary because of issues that could appear in practice that do not occur in simulations:

**2.2.4. Pathfinding Updates:**

The path was updated every time a new waypoint was reached. This was to make sure that the path would not vacillate between two equally good choices. Because of rangefinder error, what the robot would do be unsure, pick one, and then change its mind after the map is updated. In addition, if the map is updated because of the presence of scan noise and odometry error, the robot will end up travelling in a circle. By forcing the pathfinder to choose one path, versus another, the robot will certainly pick a way to the end.

**2.2.5. Graph Search Blocked Neighbors:**

The translation between occupancy space and real space is not perfect. Because of this, there are cases where the robot could be navigating between two open blocks in the occupancy grid, but the path will intersect an obstacle. Furthermore, this approach serves to ensure that the robot does not get too close to any obstacles when looking for the optimal path to the goal. This is demonstrated in Figure .5



*Figure 5: Robot traveling between two blocks in occupancy space intersects obstacle in real space*

# **3. Evaluation**

## 3.1. Simulation performance

Our implementation of D\* Lite has proven to be reasonably robust. Without fail, in our simulations, the robot was able to navigate a wide variety of obstacle courses. Below, we provide several links to videos of the occupancy grid and path updating during simulation as well as simulations themselves. The individual trial runs can be identified by the code name provided before the link.

**3.1.1. ROS simulation visualizations**

ScatteredObstacles <https://youtu.be/t90OHF9P5yI>

Circles <http://youtu.be/YvfcsExg6DI>

EasyMaze <https://youtu.be/nTOjl-fsR5U>

SmallObstacles <https://youtu.be/jc1GRM9ZnTg>

LargeObstacles <https://youtu.be/RzCOWTKEaHo>

**3.1.2. Occupancy grid generation and path finding videos.**

ScatteredObstacles <https://youtu.be/rKnqE6huI6s>

Circles <https://youtu.be/ASLmk0fMjjE>

EasyMaze <https://youtu.be/mJdHbK1EZgM>

SmallObstacles <https://youtu.be/6WZadf3x85M>

LargeObstacles<https://youtu.be/fenoiA4zW5w>

**3.1.3. Lab Trials**

Furthermore, in our seven lab trials, six successfully navigated to the end goal. The robot navigated a series of three static obstacles in four trials, and in all four trials successfully managed to go from the origin (0, 0, 0) to the goal, specified as (5, 5). Because (5, 5) itself was not a block center, the nearest one was used, and this ended up being (5.1, 5.1).

In one case, three static obstacles of small size and a large wall obstacle were utilized to generate the robot obstacle map. The robot was able to recognize the presence of an impassable obstacle due to the wall, and as a result, changed its path and headed in the other direction to avoid the dead-end. This indicated the success of the planning algorithms, allowing the robot to conclude the path was a dead end before reaching the end and seeing no solution. It beats the bug planner considerably in this aspect.

In another similar case, three static objects of small size and a large wall obstacle were used again. However, the wall and the nearest obstacle had a small gap that the robot could have feasibly fit through. This led to an interesting edge case of failure due to noise in the laser scanner’s readings. The laser scanner had a decent amount of noise, and as a result, the feasibility of the robot being able to navigate through the gap would change frequently. Because the robot updated its path on each waypoint completed, the robot would change the status of the gap, and would turn around. However, after more updates were made, the robot would again change its mind on the feasibility of the path and then turn back to head towards it due to the significant amount of minimization in length. This would continue in a cycle, and the robot would just go back and forth in a circle.

This, being the only observed case of failure of the robot in the lab, requires some thought as to how to fix. One proposed solution would be to implement a block voting system. This is difficult in that due to the required number of updates, the computation required would be extensive. As a result, updates couldn’t be run nearly as often, and this would be extremely problematic. Another issue with implementing voting is that because of the reduced frequency of updates, it would be significantly harder for the robot to handle cases such as moving objects. It is also possible that, given the scenario and the equipment at hand, there does not exist software capable of resolving the laser rangefinder noise, which comes to play a large role in the performance of the system.



We were also able to test some behaviors that are not available in simulation. In one test, a moving obstacle was tested. Jerry stood in the lab space and the robot was commanded to move to the goal. Three static obstacles were also present. Because the path updates every third waypoint, the first test ended up running over Jerry because Jerry was not factored in when the path was being calculated. However, when Jerry stood in the way before the robot updated, the robot was able to see Jerry during the update and avoid him. When Jerry moved out of the way, the robot adjusted its path to move through where Jerry was once standing, indicating that the occupancy grid was dynamically updating to optimize the path to the target.

# **4. Future Work**

If we had more time, the first thing we would attempt to fix would be the code inefficiencies. Most importantly, the full LPA\* algorithm should be implemented in order to facilitate real-time input analysis. The full LPA\* algorithm will significantly improve computation time, allowing for more frequent updates of the path. The full LPA\* algorithm updates a quarter of the map at best, and is as slow as the current configuration at its worst. In addition, some form of voting must be used to reduce the laser rangefinder noise. A significant number of issues stem from the fact that the rangefinder is so prone to noise. This is because the last angle reading is the one that matters, so even if 6 angle readings read it as closed, and one reads it as open, the occupancy grid will indicate it as open, and could plot a course through that block, which would lead to a collision.

In conclusion, we have developed a rigorous implementation of the D\* Lite algorithm. In simulation, the D\* Lite algorithm was capable of navigating complex maps with many obstacles. In lab, the D\* Lite algorithm succeeded in six cases, and failed in one case that was essentially designed for the algorithm to fail on. The algorithm can be made more robust through changes in our implementation, but as it stands the current version is an excellent model for future implementations of navigation algorithms in robotics.

# **5. References**

<http://web.mit.edu/eranki/www/tutorials/search/>

<http://pub1.willowgarage.com/~konolige/cs225b/dlite_tro05.pdf>