ME/CS 133B FINAL PROJECT
(Winter Quarter, 2018)

Guanya Shi, UID 2073344
Botao Hu, UID 2073825
Yukai Liu, UID 2072577

June 2, 2018

1 Introduction

Motion planning is a fundamental research area in robotics, which received intensive focus as the whole robotics field grows[1]. This concept considers the problem where a robot needs to automatically, in limited time, determine the shortest path from some starting point to the goal, while avoiding all the obstacles in the environment. Early efforts were spent in developing motion planning techniques in deterministic environments; but in many occasions robot needs to operate in dynamic surroundings, so later on, sensor-based motion planning took on the trend. Specifically, we have mapping methods which decompose the complex work space into graph data structure that robot is able to understand, such as visibility graphs[2], Voronoi diagrams[3], occupancy graph and configuration space[4]. Correspondingly, researchers proposed complete and efficient graph search algorithms such as bug algorithms; Dijskstra[5] and A*[6] for static maps; and D*[7] and D* lite[8] for dynamic sensed maps.

In this project, we first did simulation of D* lite algorithm. After that, we plan to implement this algorithm on Turtlebot through Robotics Operating System (ROS). However, due to the limit of time, we finished up our project with the implementation of bug algorithms.

As a result, we successfully simulated D* lite algorithm with high accuracy, high efficiency and illustrative visualizations. Moreover, we managed to realize bug algorithm on the turtlebot and achieved satisfying performance. For details of our approaches, please see section 2; for results and demos, please refer to section 3.

2 Technical Approaches

2.1 Simulation of D* Lite

- **The Algorithm**

  D* Lite (as shown in algorithm 1) is an advanced and simplified version of D* algorithm. It implements the same behavior as the original D* or Focused D*, but is based on Lifelong Planning A*. The basic idea is to search in reverse starting from the goal and attempting to work back to start. Just like LPA* or even A*, it uses current optimal path and heuristic estimations to greedily expand every node. When environment changes, it will first update all directly affected

Algorithm 1 D* Lite

```
1: procedure CalculateKey($s$)
2:     return $[\min(g(s), rhs(s)) + h(s_{start}, s) + k_m; \min(g(s), rhs(s))]$
3: procedure Initialize()
4:     $U = \emptyset$
5:     $k_m = 0$
6:     for $s \in S$ do $rhs(s) = g(s) = \infty$
7:     $rhs(s_{goal}) = 0$
8:     U.insert($s_{goal}$, CalculateKey($s_{goal}$))
9: procedure UpdateVertex($u$)
10:     if $u \neq s_{goal}$ then $rhs(u) = \min_{s' \in Succ(u)}(c(u, s') + g(s'))$
11:     if $u \in U$ then U.Remove($u$)
12:     if $g(u) \neq rhs(u)$ then U.Insert($u$, CalculateKey($u$))
13: procedure ComputeShortestPath()
14:     while U.TopKey() $<$ CalculateKey($s_{start}$) or $rhs(s_{start}) \neq g(s_{start})$ do
15:         $k_{old}$ = U.TopKey()
16:         $u$ = U.Pop()
17:         if $k_{old} < Calculate(u)$ then
18:             U.Insert($u$, CalculateKey($u$))
19:         else if $g(u) > rhs(s)$ then
20:             $g(u) = rhs(u)$
21:             for $s \in Pred(u)$ do UpdateVertex(s)
22:         else
23:             $g(u) = \inf$
24:             for $s \in Pred(u) \cup \{u\}$ do UpdateVertex(s)
25: procedure Main()
26:     $s_{last} = s_{start}$
27:     Initialize()
28:     ComputeShortestPath()
29:     while $s_{last} \neq s_{start}$ do
30:         $s_{start} = argmin_{s' \in Succ(s_{start})}(c(s_{start}, s') + g(s'))$
31:         Move to $s_{start}$
32:         Scan graph for changed edge costs
33:         if any edge costs changed then
34:             $k_m = k_m + h(s_{last}, s_{start})$
35:             $s_{last} = s_{start}$
36:             for all directed edges $(u, v)$ with changed edge costs do
37:                 Update the edge cost c(u, v)
38:                 UpdateVertex(u)
39:             ComputeShortestPath()
```

nodes, put them into the queue, and then update other indirectly affected nodes greedily and iteratively until we have found a stable estimation of the current node.

In this process, there are two important elements for each node. g is the "confident" estimation of the cost from a specific node to the goal. rhs is a temporary estimation of the cost from a specific node to the goal computed from g of its predecessors.

1. If $g = rhs$, this node is in stable state, which means we are sure that all estimates about this node is the optimal result.
2. If $g < rhs$, this node is underconsistent, which means our previous estimate about this node does not work anymore, so to be conservative, we have to set g to be infinity.
3. If $g > rhs$, this node is overconsistent, which means our previous estimate about this node can be optimized.

Following this definition, every time when a node is affected, we will update its rhs. If this influence makes our previous estimate inaccurate, we need to put it into a queue and deal with it later. And g is only updated when it is popped out of a queue.

- **Simulation Method**

  Our simulation is implemented in a Maze environment. We randomly generate a 100×100 maze as the global_map, where 0 stands for empty space and 1 stands for a wall, and we choose a starting and a target point. Our robot starts at the starting point, with its own sensed_map initialized with all zeros. It has a 7×7 view, here for simplicity we assume the robot can look through the wall and see everything within this range regardless of any obstacles that may block its sight. Every step, it first looks around and updates its sensed_map using the gathered information. Then, it updates all affected nodes and run the motion planning algorithm to find an optimal path. Next, it takes one step along the optimal path. This process is repeated until it reaches the target or reports the target is unreachable.

2.2 Turtlebot implementation of Bug algorithm

In class, Joel introduced two basic sensor-based motion planning algorithms, Bug I and Bug II. For each obstacle that the robot encounters, Bug I performs an exhaustive search to find the optimal leave point. This requires that Bug I surround the entire perimeter of the obstacle, it is certain to have found the optimal leave point. In contrast, Bug II uses an opportunistic approach. When Bug II finds a leave point that is better than any it has seen before, it commits to that leave point, so Bug II is greedy. When the obstacles are simple, the greedy approach of Bug II gives a quick payoff, but when the obstacles are complex, the more conservative approach of Bug I often yields better performance.

In this project, we applied Bug II algorithm on the TurtleBot system, and tested it on some simple demos.

- **The Algorithm**

  The Bug II algorithm pseudocode is shown as algorithm 2. There are two modes in this algorithm: Motion-to Goal (MTG) mode and Boundary-Following (BF) mode. In MTG mode, robot will move along M-line until it hits some obstacle and then switch to BF mode, or reaches the goal. In BF mode, the robot will follow the boundary of the obstacle, until it reaches the goal, or re-reached M-line.

Algorithm 2 Bug II

```
1: Record $M$-line as the straight line connecting $s_{start}$ to $s_{goal}$
2: $i = 1$
3: $s^L_{i-1} = s_{start}$
4: $state$ = moving
5: while True do
6:     if $state \neq$ moving then
7:         Stop the robot
8:         break
9:     while True do
10:        Move robot from $s^L_{i-1}$ toward $s_{goal}$ along $M$-line
11:        if $s_{goal}$ is reached then
12:            $state$ = success
13:            break
14:        if An obstacle is met at $s^H_i$ then
15:            break
16:    if $state \neq$ moving then
17:        Stop the robot
18:        break
19:    while True do
20:        Turn right and follow boundary
21:        if $s_{goal}$ is reached then
22:            $state$ = success
23:            break
24:        if $M$-line is re-reached at $s$ then
25:            if $s = s^H_i$ then
26:                $state$ = failure
27:                break
28:            else
29:                if $d(s, s_{goal}) < d(s_{goal}, s^H_i)$ and Safe to move toward $s_{goal}$ then
30:                    $s^L_i = s$
31:                    Increment $i$
32:                    break
```

- **TurtleBot Implementation Method**

  To implement Bug II algorithm in a real TurtleBot system, we need to consider (1) how to sense an environment with incomplete information, (2) how to localize the TurtleBot and (3) how to control the TurtleBot. In practice, we used two subscribers (which takes data returned by on-body sensors in real-time) and one publisher (which publishes live motion commands from control terminal to the TurtleBot) to solve these issues.

  The first subscriber is `Odometry`, which will collect odometry data (returned by IMU and wheel encoder modules) from `/odom` and send it to a function called `location_callback`. In this function, we can get x and y positions from the `position` information in the odometry data, and we can also calculate the orientation information from the quaterion in odometry data.

  The second subscriber is `LaserScan`, which will collect lidar data from `/scan` and send it to a function called `sensor_callback`, where we can get 512 range values (i.e., the distance of obstacle in 512 directions with approximately 180 degree angle range). We will use these range values to estimate the distance between the TurtleBot and obstacles/goal.

  The publisher will publish `Twist` (motion) commands to `cmd_vel_mux/input/navi`, and then the TurtleBot would follow these commands to go straight, turn left or turn right.

3 Results and Demonstration

3.1 D* Lite Simulation

- **Challenges in D* Lite Simulation**

  - **Data Structure**

    We created a `class` object in Python for the whole D* lite procedure, with attributes such as g, rhs, k_m, maps, start point, goal, and priority queue. This `class` also has corresponding methods in the pseudocode - `CalculateKey(s), Initialize(), UpdateVertex(u), ComputeShortestPath()`.

    We used a matrix to present the occupancy graph of the map, with values in free grids as `0`, and values in obstacle grids as `infinity`. All the coordinates in the algorithm are reported as indices of the map matrix. Meanwhile, g and rhs are also stored as matrices with the same size of the map, where the value corresponds to the g or rhs value of a certain grid. The advantage of this data structure is that it is convenient for map updates and index calls (see next bullet for detail), and the disadvantage is that it needs more code to fetch successors and predecessors (select every element indices nearby).

    The data structure of priority queue is realized by `heapq` package. This package is able to convert `list` object into a heap and return the smallest element in $O(1)$ time. The elements inside the priority queue belong to an `Element` class, with three attributes: coordinate of the point as `key`, and two `values` returned by `CalculateKey(s)`. The `equal` and `compare` built-in methods are overloaded, such that one can find `Element` inside the priority queue by its `key`, and one can compare two `Element` objects by its `values` in lexicographical order. For specific implementations, please refer to our code.

  - **Sensor Update**

In D* lite algorithm, the robot will update edge costs when encounters a mismatch between the `global_map` (the real world landscapes) and `sensed_map` (the world the robot thinks it is). This update is hard to code since other than map matrix, we need extra space to record costs. As a modification of the algorithm, here we let the `sensed_map` matrix to be zero matrix in `Initialize()`, which means robot initially thinks the workspace is pretty smooth before sensing. Whenever a mismatch is noticed, the robot simply update the `sensed_map` rather than maintaining a large cost record.

  - **Debugging**

    Debugging is an essential part of code simulation. We first tested the A* version of this algorithm, that is, the robot knows everything about the workspace, and only needs to compute the shortest path. For testing maps, we designed small maps by hand, and used a random maze generator for large maps. After that, we tested the D* lite algorithm, and examine through g, rhs, `sensed_map`, and the actual path in every step taken by the robot. This process also helped us to deepen our understanding of the algorithm since we are looking at a vivid example.

- **Simulation Results and Visualization**

  - **Success Rate**

    Using the random maze generator, we tested the D* lite algorithm on various sizes of the workspace map and various maze structures, and achieved 100% success rate of reaching the goal.

  - **Efficiency**

    This algorithm is very efficient in time. For a 100×100 map matrix with maze structure (see figure 1 below, where the robot starts from bottom left corner, and ends at upper right corner), the robot takes less than 3 minutes to find the goal.

  - **Visualization**

    Using MATLAB, we created an animation for D* lite algorithm in the same maze as figure 1. Please see either attachment or YouTube channel `https://youtu.be/h6H3n0BNXi8` for the video.

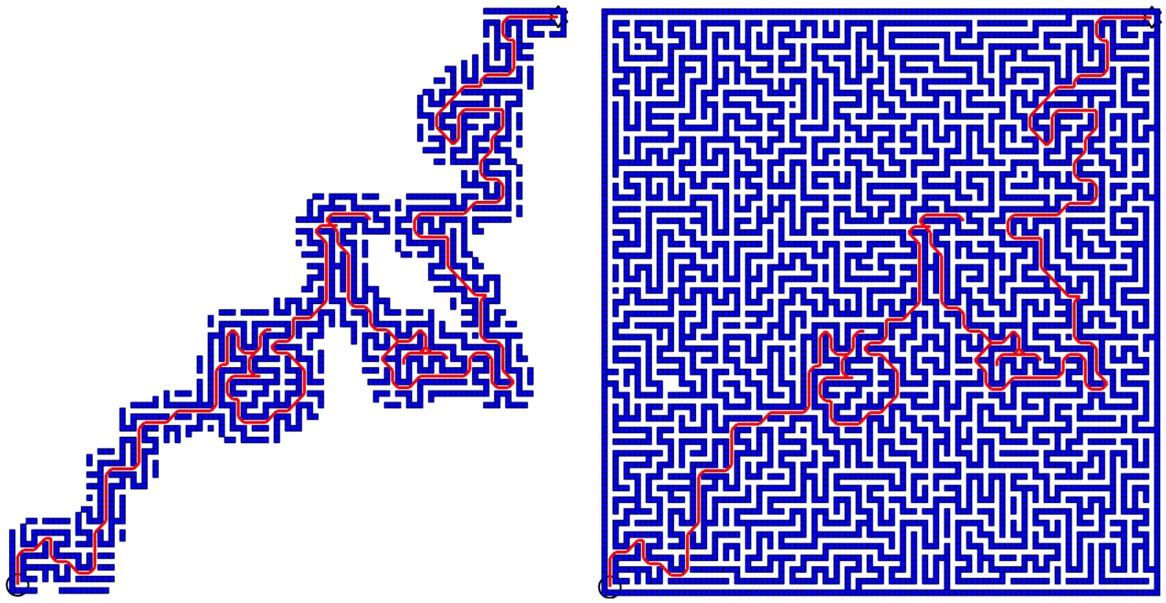
3.2 Turtlebot implementation of Bug algorithm

- **Challenges in Bug Implementation**

  One thing we learned from this project is that implementations in a real system are much more complicated than the corresponding theoretical framework. Although Bug II is a quite easy and fundamental algorithm in sensor-based motion planning, we met many challenges.

  - **Configuration Space**

    For Bug II algorithm, we assume that the workspace is converted to configuration space, where the robot is a point and the obstacle boundaries are calculated considering the outer shape of the robot. However, in practice, the TurtleBot is a fairly big cylinder and we are dealing with real obstacle boundaries. Moreover, the Laser sensor on the TurtleBot is at the front part of the TurtleBot, so that the range in sensor data would contain a position bias, which is also an issue for our implementation.



(a) `sensed_map` (b) `global_map`

Figure 1: D* lite algorithm trajectory in 100 × 100 maze

To solve this issue, we spent a lot time on threshold parameter validations, such as setting the threshold distance from robot to the obstacle, which makes the TurtleBot switch from motion-to-goal mode to boundary-following mode; or adjusting the angle window range to tell the robot where is its "front" direction, or "left" direction; and so on.

- **Noise in Odometry and LaserScan Data**

  The odometry data is from IMU and wheel encoder. Without closed loop control, the data is quite noisy, and the estimation error tends to accumulate with the operation time. The Laser scanner on the TurtleBot also introduces noise, due to various reasons such as limited sensing range or uncertainty in angle. Moreover, in this project, we just took a naive way to calculate the distance between the TurtleBot and the obstacle - simply taking the smallest range value inside a front angle window as the distance. These three factors jointly caused inaccuracy in running the bug algorithm. If we have more time on this project, maybe the Kalman Filter or other estimation algorithms could come to rescue the TurtleBot's performance.

- **"Dead Locations"**

  Sometimes when these two problems above come together, the TurtleBot may just get stuck at some special locations, we say "Dead Locations". For example, a common problem occurs when the TurtleBot is in boundary-following mode and almost ready to leave the wall. At that moment, the Laser scanner (at the front part of the TurtleBot) reports that there is no seen obstacle from its current location to the goal. The bug algorithm would immediately command TurtleBot to "leave the wall", that is, change its orientation towards the goal, and then go straight. However, as the robot turns its orientation, the position of Laser Scanner is also changed. As a consequence, the obstacle - initially out of Laser's sight - comes back to the Laser's view. The poor Laser has to again report "obstacle ahead", and force the TurtleBot back to boundary-following mode. All the efforts we made end in vain, and the robot could never get out of this position. In this project, due to the limit of time, we just use some brute force methods to directly solve this issue, such as tuning threshold parameters, asking TurtleBot to take some actions to leave that weird state, etc. If we have more time in the future, we think the more reasonable way is to transfer laser scanner data into TurtleBot's own body frame.

There are many other issues we have to deal with in real experiments. Robot is a complicated system, where the motion planning algorithm is only a small section. Probably a better strategy would be to divide the whole system into modules and come up with a complete plan so that we could know what we are doing and we can move forward step by step.

- **Results and Visualization**

  Figure 2 shows the movement of the TurtleBot when it avoided an obstacle, by Bug II algorithm. This experiment was implemented in the lecture hall in GTL, and the obstacle is a platform. For videos, please see attachment or YouTube channel `https://youtu.be/XoQ6hDP7jcQ`.

4 Debriefing

If we have more time, we would improve the project in the following ways.

- **D* lite Simulation**

  - **Interface**

    Develop a more interactive interface, where the user can set the start and goal point, add obstacles, and adjust sensing range. Functionalities such as step-wise monitoring, reset and pause are also in our will list.

  - **Environment**

    We would not confine ourselves in `(0, infinity)` representation of the environment. We could add variety such as sandy ground, icy ground, and even some soft obstacles. Moreover, a dynamic environment where `global_map` keeps changing is more realistic to simulate.

  - **Visualization**

    To better demonstrate our simulation and help algorithm learners, we would consider showing g and rhs values in real-time inside grids, while displaying k_m value and priority queue near the map.

- **TurtleBot Implementation**

  - **Systematic solution**

    As we've mentioned above, for most of the problems we met in our experiments, we need to think about and deal with them in a more systematic way. For example, we should write a data processing module that process data and transfer them into some form that

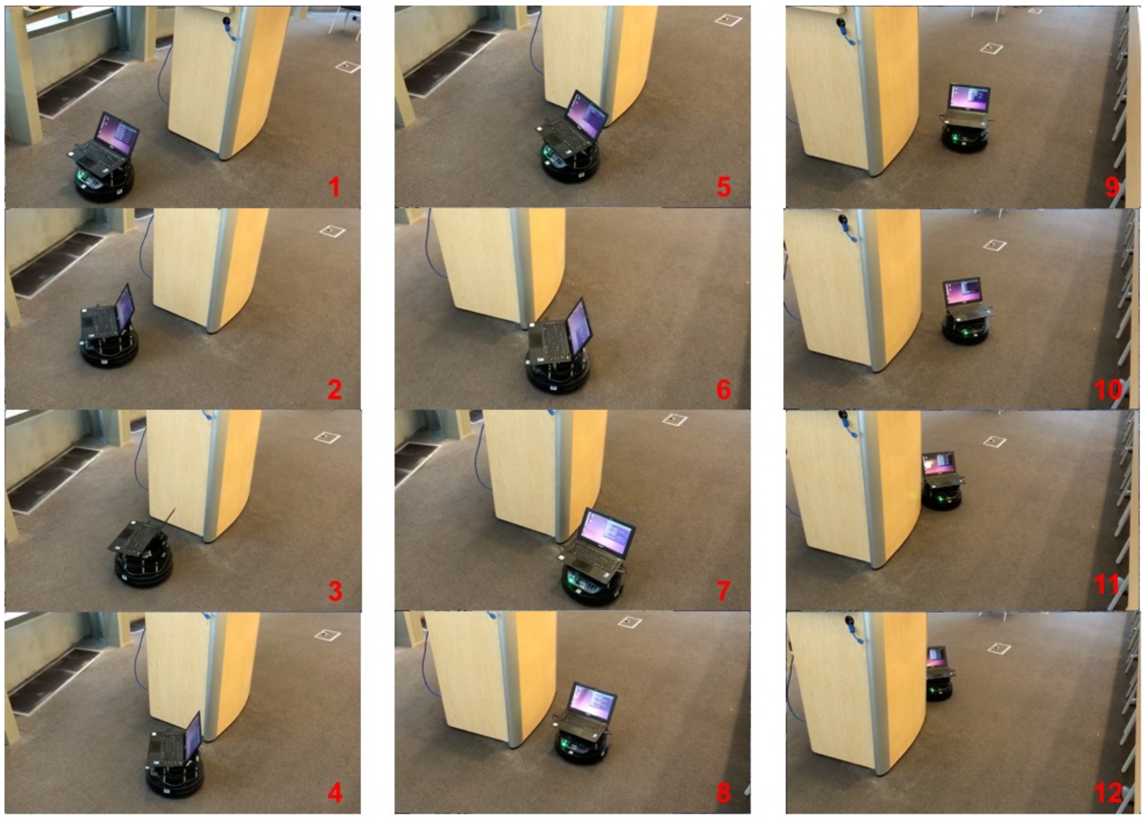


Figure 2: The Bug II algorithm implementation on TurtleBot

the TurtleBot can directly use. We should also add a sensor calibration part that tells us the real working range and space. Furthermore, converting workspace into configuration space is definitely very important to us. There are still lots of stuff we can do even without extending the objective of this project.

- **RGB-D Camera**

  This time we used laser scanner to implement the algorithm, next time if we have time we could also use RGB-D camera to complete this task. Unlike laser scanner, RGB-D camera does not directly give us distance information to any obstacle, but on the other hand, it provides more information about its surroundings. So we need to either convert images into distances or figure out how to more efficiently use this information.

- **D* Lite Implementation**

  Of course, implementing D* Lite was the original goal of our project. We can imagine there will be much more difficulties we may meet compared with this simple Bug algorithm, like how to grid the world, what information we need to save and how to manage those stuff, etc. That will be challenging but also very interesting.

References

[1] M. Elbanhawi and M. Simic, Sampling-Based Robot Motion Planning: A Review, in IEEE Access, vol. 2, pp. 56-77, 2014.

[2] T. Asano, T. Asano, L. Guibas, J. Hershberger, and H. Imai, Visibilitypolygon search and euclidean shortest paths, in Proc. 26th Annu. Symp. Found. Comput. Sci., Oct. 1985, pp. 155164.

[3] J. Canny, A Voronoi method for the piano-movers problem, in Proc. IEEE Int. Conf. Robot. Autom., Mar. 1985, pp. 530535.

[4] R. A. Brooks and T. Lozano-Perez, “A subdivision algorithm in conguration space for ndpath with rotation,” IEEE Trans. Syst., Man Cybern., vol. 15, no. 2, pp. 224233, Mar./Apr. 1985.

[5] E. W. Dijkstra, A note on two problems in connexion with graphs, Numer. Math., vol. 1, no. 1, pp. 269271, Dec. 1959

[6] P. E. Hart, N. J. Nilsson, and B. Raphael, A formal basis for the heuristic determination of minimum cost paths, IEEE Trans. Syst. Sci. Cybern., vol. 4, no. 2, pp. 100107, Jul. 1968.

[7] A. Stentz, Optimal and efcient path planning for unknown and dynamic environments, Int. J. Robot. Autom., vol. 10, no. 3, pp. 89100, 1995.

[8] Sven Koenig and Maxim Likhachev, D* lite, `http://idm-lab.org/bib/abstracts/papers/aaai02b.pdf`