ME/CS 134 Final Project

Implementation of D^*-lite Algorithm on Turtlebot

(Spring Quarter, 2018)

Guanya Shi, UID 2073344
Botao Hu, UID 2073825
Yukai Liu, UID 2072577
Yan Wu, UID 2048228
Yu-Wei Wu, UID 2072499

June 16, 2018

1 Introduction

When a robot needs to plan a collision-free path from certain starting point to the goal position, this seemly simple path planning problem is actually computational hard[1]. When it comes to real robot, additional constraints originated from mechanical and sensor limitations, such as uncertainties and feedback, also need to be considered. Normally, the path planning problem is approached by one of the following three categories: search-based, sampling-based, or combinatorial, as shown in figure 1

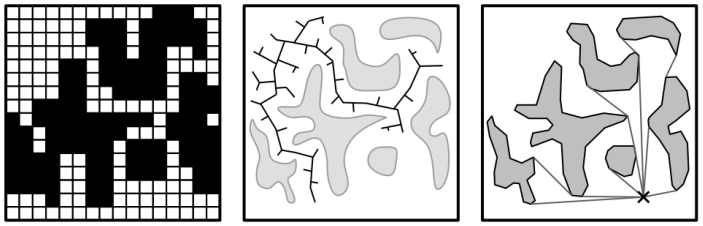

Figure 1: Three planning methods [3]

Search-based planning, the dominating approach in field robotics, generates a graph representation of the planning problem, and as its name suggests, searches the graph for a solution. One example of 2D (X,Y) search-based planning is shown in figure 2.

Once a graph is given, the next step is to search it for a path in an known, partially known, or unknown environment, while bearing in mind operation cost, completeness, space complexity, and time complexity. We have encountered in ME134 Labs from Dijkstras Algorithm, which is simply an uninformed breadth-first search, to A*, a popular method that uses an admissible heuristic to narrow

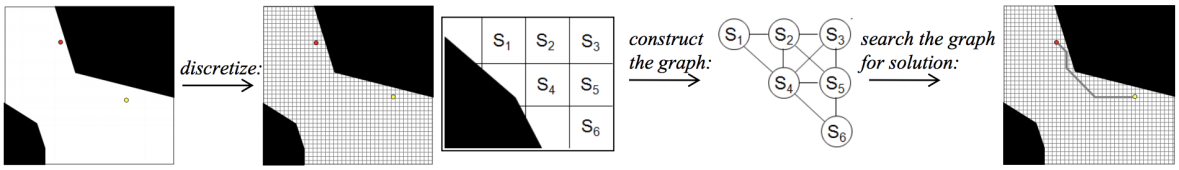

Figure 2: 2D grid-based graph representation for 2D search-based planning [2]

the search. The problem we had with A* during experiment was that, A* is a static algorithm, which means that when the configuration space changes (such as when an obstacle changes or in our case, an unexpected obstacle that was not observed in the beginning actually existed and stayed in the way), the old path is invalidated and everything needs to be start over again[4].

D*, the first dynamic search-based algorithm, adds new information to its map when encounters changes or previously unknown environment. It plans the shortest path in real time by incrementally repairing paths to the robots state. Focused D* addresses the time complexity limitation of the original D*, by focusing repairs to significantly reduce the total time required.[5]

Among all the variants of A* and D* algorithms, D* lite is particular interesting, as it implements the same navigation strategy as Focused D* does, but is much easier to code and maintain.[6]

Last term we successfully performed D* lite simulation (YouTube video). In this project, we first implemented D* Lite in global planner of ROS navigation stack in Ros_Indigo simulator. After that, we planned to implement this algorithm on Turtlebot through Robotics Operating System (ROS). However

2 Technical Approaches

2.1 D* Lite 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 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.

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 = ∅
5:     k_m = 0
6:     for s ∈ S do rhs(s) = g(s) = ∞
7:     rhs(s_goal) = 0
8:     U.insert(s_goal, CalculateKey(s_goal))
9: procedure UpdateVertex(u)
10:    if u ≠ s_goal then rhs(u) = min_{s'∈Succ(u)}(c(u, s') + g(s'))
11:    if u ∈ U then U.Remove(u)
12:    if g(u) ≠ rhs(u) then U.Insert(u, CalculateKey(u))
13: procedure ComputeShortestPath()
14:    while U.TopKey() < CalculateKey(s_start) or rhs(s_start) ≠ 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 ∈ Pred(u) do UpdateVertex(s)
22:        else
23:            g(u) = inf
24:            for s ∈ Pred(u) ∪ {u} do UpdateVertex(s)
25: procedure Main()
26:    s_last = s_start
27:    Initialize()
28:    ComputeShortestPath()
29:    while s_last ≠ s_start do
30:        s_start = argmin_{s'∈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()
```

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.

2.2 Realization

2.2.1 Global Planner

Our first idea was simply replace the A* code in the original ME134 global planner with D* lite algorithm. This attempt failed as we found out, theoretically, move_base as shown in figure 3 will only call `global_planner` once for each given start and goal. It is totally compatible with A*, since the map and the path are determined in the beginning. However it won't work with D* or any dynamic search-based algorithms, as the map is constantly updated and the path is repaired incrementally. One interesting thing we noticed is that, in reality move_base will visit `global_planner` more than once during one path, which makes our plan seem doable. But since we don't know why this happens and we cannot predict when and how frequently `global_planner` is visited, it's unlikely to get consistent and reasonable results.

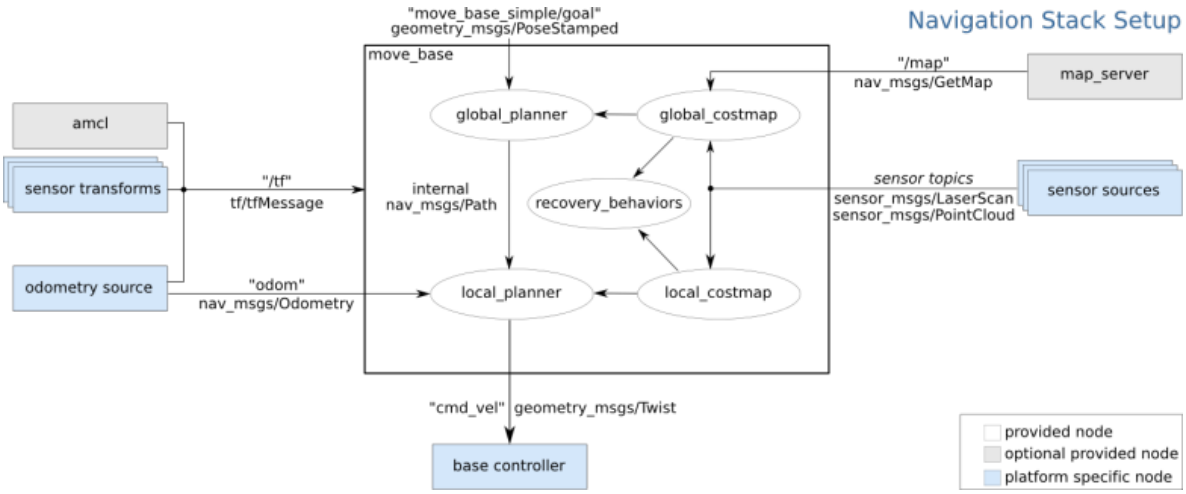

Figure 3: Navigation stack setup

After some trial and error, the final solution we came up with is combining the idea of frontier_explorer in ME134 Lab3 and a naive `global_planner` which only takes straight path. In this design, explorer will actually execute D* lite for path planning, and it will select a point on the path, which satisfies 1) it can be reached by a straight path 2) all the way to reach the point has been explored, as the goal and send it to `global_planner`. Once it reaches that goal, explorer will update the map and rerun D* lite searching and send out another temporary goal to `global_planner`. In this way, instead of planning the whole path all at once (as the yellow dot line shown in figure 4), we break down the ultimate goal into a series of temporary goals which can be safely achieved. Figure 4 shows a general idea of such design, with more details of modified D* lite included in following subsection.

Important to note that in this setting, since the global planner in navigation stack only commits to a straight line, the actual path of the turtlebot is completely determined by our "D* Lite Explorer".

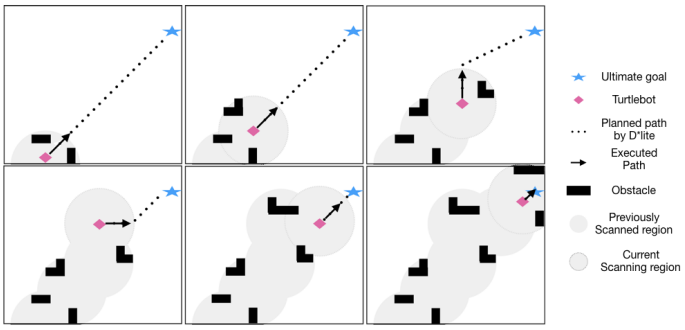

Figure 4: D* lite realization with modified explorer and `global_planner`

Following this tutorial, we wrote our own `global_planner` as plugin in ROS. Briefly, we first create a new path planner class with the straight-path-only behavior , and adhere the planner to `nav_core::BaseGlobalPlanner` C++ interface. This was done by exporting our global planner class as plugin and register it for `nav_core::BaseGlobalPlanner` for `move_base`. Then the plugin was registered with ROS package system in order to be used by `move_base`. When running this plugin on Turtlebot, we need first export it to the Turtlebot ROS environment and then modify `move_base` configuration to specify that the new planner will be used.

2.2.2 D* lite Explorer

We implement our D* Lite Explorer based on the Explorer class defined in Lab3, which listens to "map" and sends goal to navigation stack.

- Structure and Pipeline

  In the explorer class, we keep a D* Lite object which saves our ultimate goal, sensed map, g matrix, rhs matrix and does all the computation for us. Each time when the explorer is called, we first update the latest sensed map in D* Lite object, then run standard D* Lite algorithm. The algorithm will return a path from its current position to the goal based on its knowledge of the environment, where unexplored area are treated as free spaces. We then select a temporary goal from this path and send it to the navigation stack and wait for the next call.

- Goal Selection

Given a path from current position to the goal, our explorer always selects the temporary goal which has already been observed and can be reached by moving along a straight line in the path. A visualized example is shown in figure 4.

In real implementation, let's assume we have a path = $\{(x_1, y_1), (x_2, y_2), (x_3, y_3), ..., (x_T, y_T)\}$, where (x_1, y_1) is our current position, and (x_T, y_T) is the ultimate goal. We start from (x_2, y_2), for each (x_i, y_i), first check if this point is observed, then check if $(x_i - x_{i-1},\ y_i - y_{i-1}) = (x_{i-1} - x_{i-2},\ y_{i-1} - y_{i-2})$. If both conditions are true, we update the temporary target to be this point.

2.2.3 Improvements taken

- Reuse computed shortest path
  As mentioned above, our explorer selects the temporary goal as the point that has already been observed and can be reached by moving along a straight line in the path. However, this temporary goal may be very close to the current starting point and finding the whole path from current starting point to the goal becomes a waste. Therefore, we modify the algorithm to reuse the computed shortest path by continuously selecting temporary goal from the path that can be reached in straight line, until we reached a point that hasn't been explored before.

- "Configuration Space"
  Since the Turtlebot itself has a certain volume, and the D* lite algorithm tends to find path along the wall, we have to make sure that the Turtlebot wouldn't be too close to the wall so that it wouldn't bump into it. We achieve this by doing dilation on the sense map, which is equivalent to dilating the wall by a certain amount. As a result, the Turtlebot will always be kept away from the wall by a certain distance.

- Move away from the wall when recomputing
  Following the previous point, even if we do dilation on the wall, there are still circumstances that a Turtlebot might move into the dilation space of a wall that it hasn't detected yet. In order to deal with this, before finding the shortest path from the Turtlebot's current point to the goal, we will check if the robot is in the dilated wall, and move it away from the closest wall.

- Autoscale map
  While testing our algorithm with Turtlebot in real world, we discovered a problem that the sense map will change size occasionally in the beginning stage of process. This is because the Turtlebot will update its detected boundary point of the map while scanning around the environment. This causes a problem to our algorithm since the D* lite algorithm deals with relative location of points in the map instead of absolute location. We resove the problem by shifting and updating the size of the old maps to whenever its size is different from the new maps.

3 Results, demonstration and discussion

3.1 Simulation

We firstly tested our D* lite implementation in simulation. We took `me134_explorer` package in Lab 3 as the base, run our code with the launch file `explore_in_stage.launch`, and recorded the Turtlebot's trajectory as well as figures plotted in the way.

The map is shown in the top left of figure 5. The Turtlebot starts from the bottom left, with the goal at the top right of the map. This map well characterizes the D* lite algorithm: the goal and some obstacles are initially invisible to the Turtlebot, so the planner has to frequently update its map and replan as the Turtlebot moves, which is fundamentally different from static path planners such as Dijkstra or A*.

The stepwise plot of the Turtlebot is shown in subfigure 1 - 11 of figure 5. In this figure we can see that the Turtlebot initially had a limited view of the map (subfigure 1 and 2). According to our implementation, the next step must be chosen within the seen area and can be achieved by our naive global planner, until it sees a broader range of view (in subfigure 3 and 4). The Turtlebot then made a big progress in subfigure 5, but immediately in subfigure 6, it found a new obstacle on the right. The Turtlebot took a few small steps to observe this obstacle (in subfigure 7, 8 and 9), and replanned the path. As a result, it successfully avoids this obstacle and reached the goal in subfigure 10 and 11.

The screenshots taken from `RVIZ` is shown in figure 6. In subfigure 1, 2 and 3, from the thick trajectory point cloud, we can tell that the Turtlebot moved conservatively in order to observe a wider view of the map. Also, in subfigure 7 and 8 we see the Turtlebot took a small turn in the front of new obstacle, which indicates that map was updated and path was replanned.

We completed this simulation repeatly in small amount of time, which also proved that the algorithm is stable and time-efficient. In conclusion, our implementation performed well in this simulation, which gave us confidence to test it in real Turtlebot and practical scenes.

3.2 Experiment 1

We did our first experiment in classroom 115, GTL. The obstacle setup is shown in subfigure 1 of figure 8, where the Turtlebot has to make its way between two trash bins, and get to the goal near the wall. We set the situation to be rather simple, that the Turtlebot can have a glance of each obstacle parts at the beginning.

Figure 7 shows the stepwise plot of the Turtlebot and the map. The Turtlebot started with partial information (it can only observe some parts of the obstacles), updated the map during its way, and finally reached the goal following D* lite algorithm.

Figure 8 shows the real performance and path of the Turtlebot. In subfigure 1 - 5, Turtlebot initialized itself by turning around, and recording its first glance of the world. Then, in subfigure 6 and 7, the Turtlebot took its step conservatively to have a better view of two blocking trash bins. In subfigure 8 and 9, the Turtlebot carefully made its way between two trash bins (our algorithm computes shortest path in configuration space, so the Turtlebot must maintain a safe distance from obstacles). After that, the Turtlebot can have a better view of its goal (by turning around again), and D* lite planner updated the map and replanned. Finally, in subfigure 10, 11 and 12, the Turtlebot reached the goal following the replanned path.

In conclusion, our implementation worked successfully in real, simple situation, where the Turtlebot can see partial information of all obstacles. Apart from simulation, a lot of boundary issues were taken into consideration during the test phase, and computation needed more time as the dimension of the map was significantly increased. For reference, a video demonstration is uploaded to YouTube. Link: `https://youtu.be/ttc8bM89Alk`.

3.3 Experiment 2

In this experiment, We move even further from the previous one: initially, the goal and some obstacles are *completely* hidden from Turtlebot. We did this experiment in the living room of graduate dorm in Catalina housing. Also, since the dorm has limited space and gaps between obstacles are rather narrow, this situation requires our implementation to have very precise obstacle avoidance strategies.

The environment setup is shown in the top left of figure 9. We can see that the Turtlebot's view is blocked by a series of obstacles, so it can't see another obstacle (a black trash bin and a wooden board), behind which the goal is set. Also, the bottom right of figure 9 shows the configuration space where the D* lite path planner is working on. Due to the limited space, the aisles in configuration space is pretty narrow. The path planner always gives the shortest path, which means that the Turtlebot would not "stand" in the center of the aisles, but would make its way along the boundary of configuration space obstacles. In this case, if localization system has a small error, the Turtlebot would find itself stuck "inside" the configuration space obstacles, and our planner cannot "break the wall" and find a path toward the goal. To solve this, we ask the Turtlebot to check its surroundings when stuck into obstacles. If there are some free spaces nearby, the Turtlebot would make a short move in the direction which goes further from the obstacle, and return to the free space.

Other subfigures in figure 9 shows the stepwise plot of the Turtlebot and the map. Clearly the Turtlebot could't see the next obstacle in subfigure 1 and 2, but it found that obstacle in subfigure 3. After map update and replan, the Turtlebot carefully moved towards the next obstacle in subfigure 4, 5 and 6. When it reached behind that obstacle, the Turtlebot finally can see the goal, and the planner's replan told the Turtlebot to go straight to the goal.

Figure 10 shows the real performance and path of the Turtlebot. The top left subfigure provides an overview. In subfigure 1-5, the Turtlebot turned around to see its initial surrondings, and had no idea of the next blocked obstacle. Then, the Turtlebot tried to detour around the first obstacle in subfigure 6, 7 and 8, but due to the localization error, it found itself stuck into the configuration space obstacle. So, the Turtlebot had to move further from that obstacle to get back into free space, shown in subfigure 9 and 10. After that, the Turtlebot had a better view of the next obstacle and tried to detour, in subfigure 11, 12 and 13. However, the Turtlebot got stuck again into configuration space of the black trash bin, so it had to rescue itself in subfigure 14 and 15. At last, the Turtlebot successfully saw the goal and moved straight towards the goal in subfigure 16 and 17.

Our success in Experiment 2 is remarkable, since it proved the stability of our implementation, which worked well in narrow spaces and complicated, hidden obstacle setup. The overall process is a bit long, as the Turtlebot keeps rescuing itself and doing replan, and this time issue can be improved if we had more time on this project (explained in the next section). For reference, a video demonstration is uploaded to YouTube. Link: `https://youtu.be/q3fGvxpixrg`.

4 Challenges and Debriefing

This project consists of simulation and hardware experiments. For simulation, the main challenges are about algorithm and structure design. On the one hand, we should think about how to combine our D* Lite algorithm with its original navigation stack. On the other hand, when we implement a theoretical algorithm on a turtlebot, we need to take into account its shape, size and noise. Also the map size is changing in this process. Generally speaking, in this stage, these challenges are hard but very clear. We know where the issues are and we can try our best to find away to solve them.

In the second stage, when we want to do real experiments, things become much more complicated.

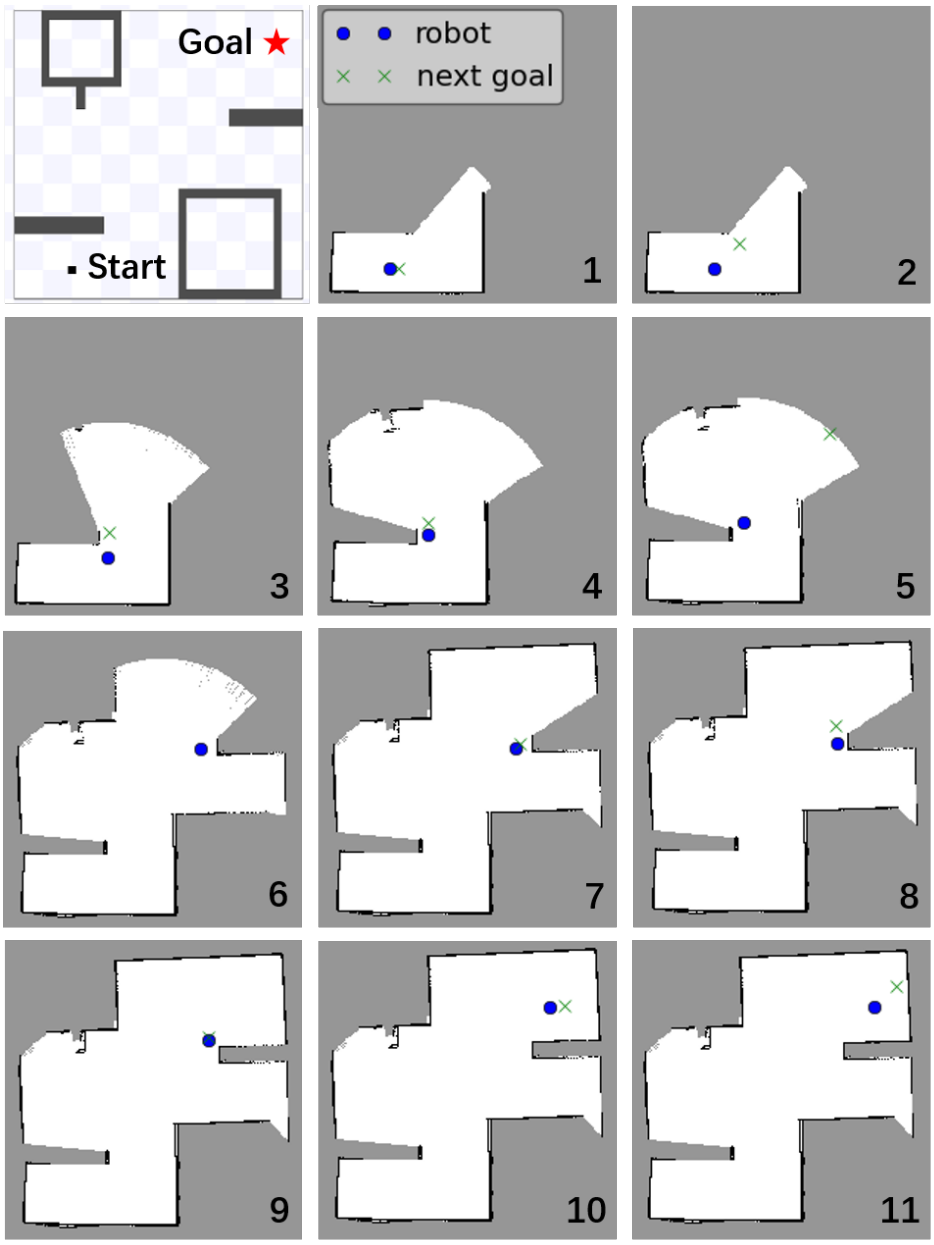

Figure 5: [Top left] the overall map setup, [1 - 11] the stepwise plot of Turtlebot from start to goal, in simulation

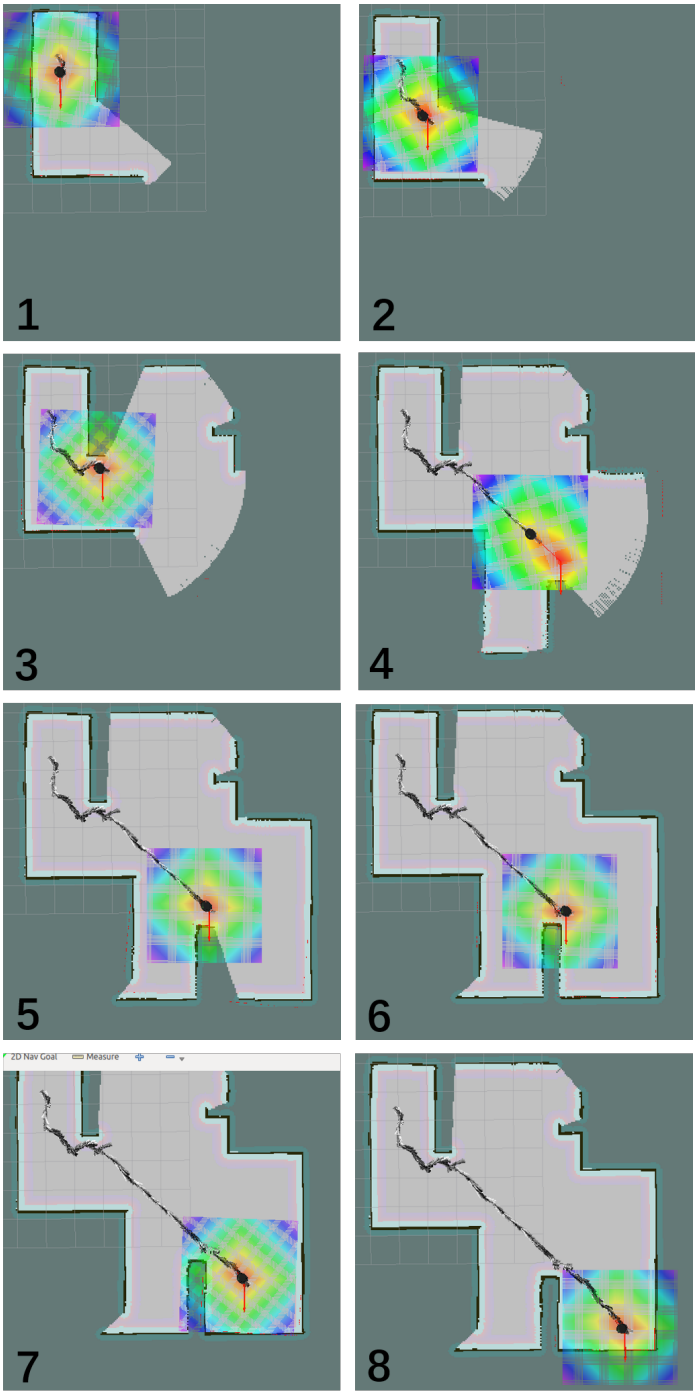

Figure 6: A series of screenshots taken from `RVIZ` while Turtlebot is approaching the goal, in simulation

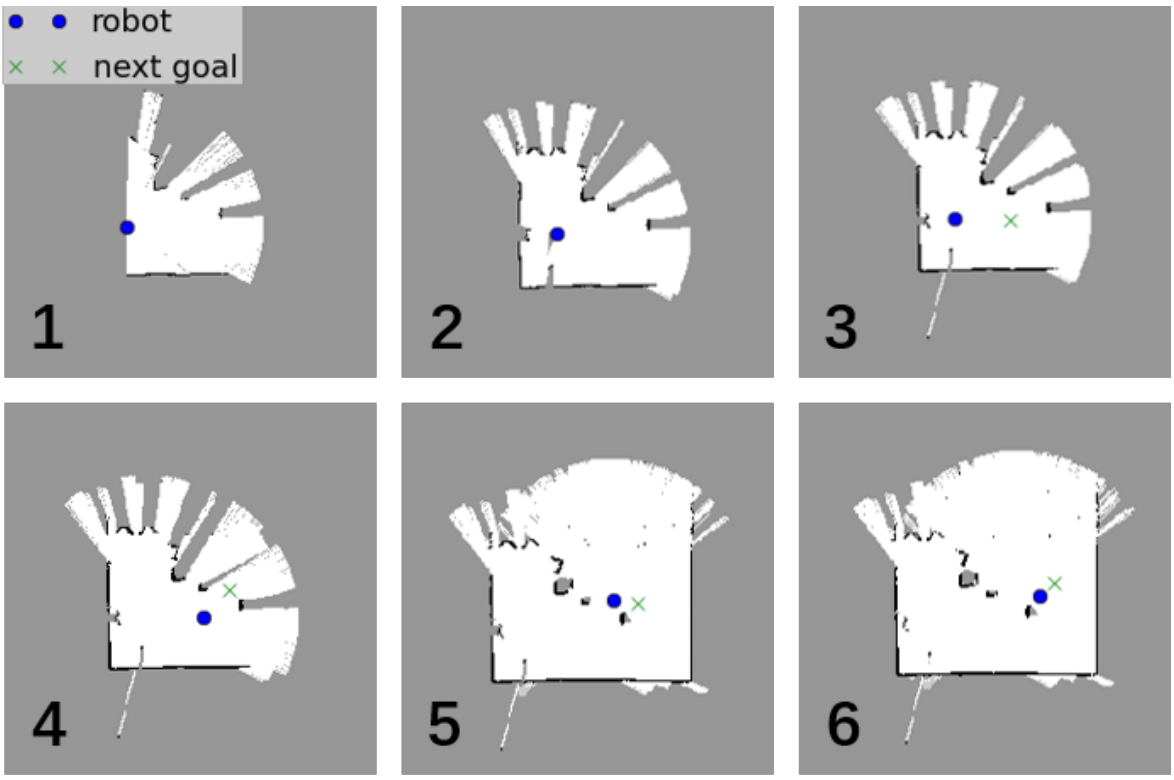

Figure 7: The stepwise plot of Turtlebot from start to goal, in experiment 1

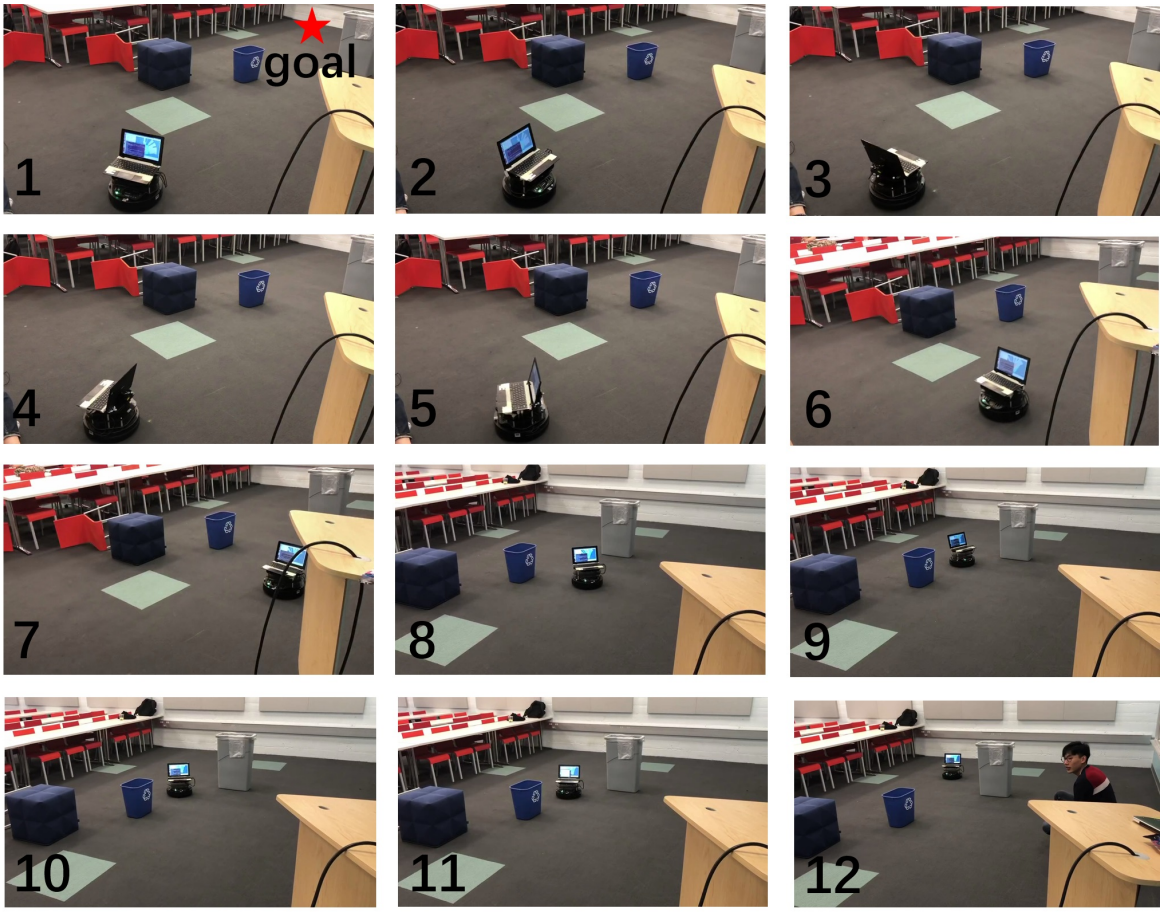

Figure 8: The photo record of Turtlebot's real path, in experiment 1

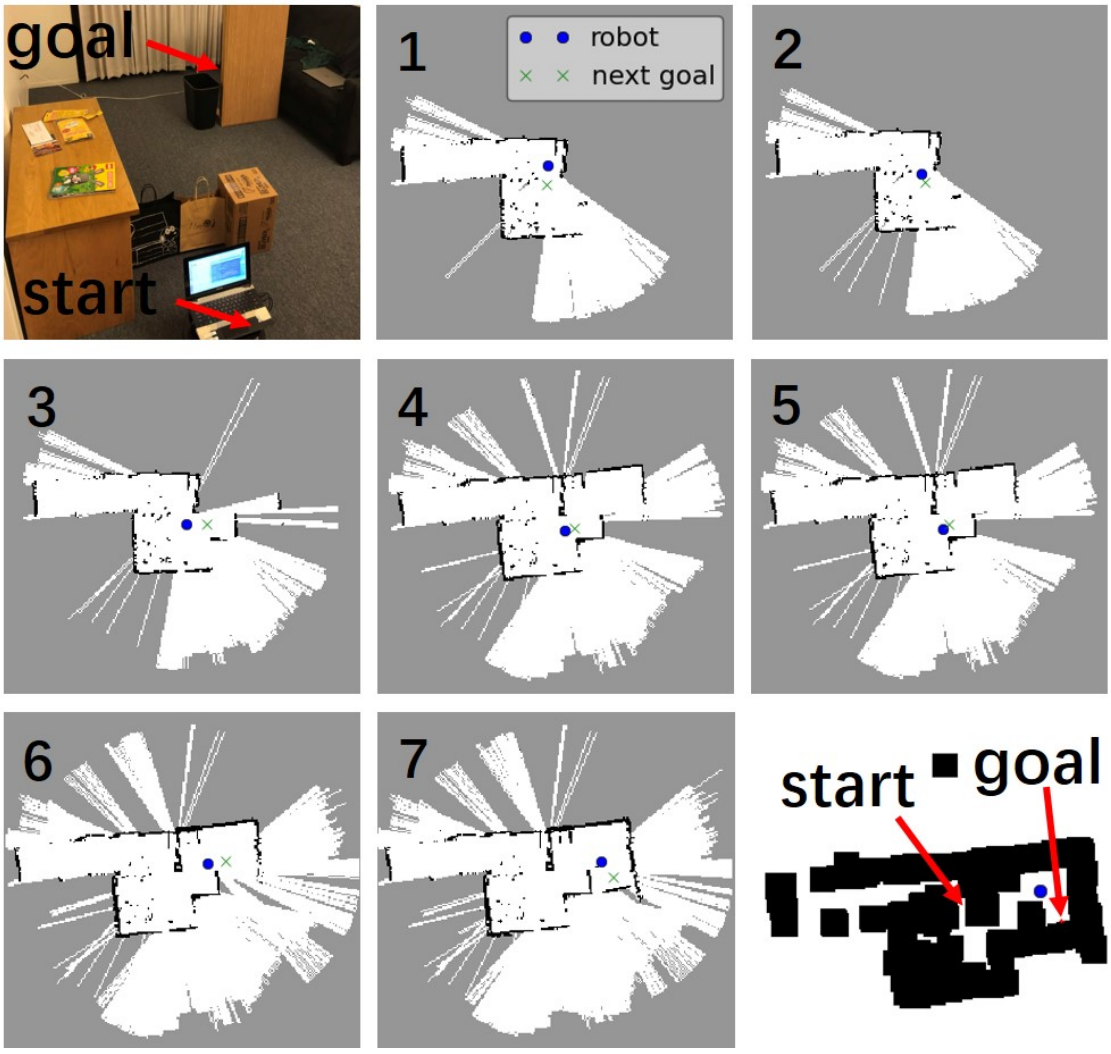

Figure 9: The stepwise plot of Turtlebot from start to goal, in experiment 2

Figure 10: The photo record of Turtlebot's real path, in experiment 2

Hardware does not work as we expected. Sensors and self-localization are more inaccurate than simulation. For example, even though we plan the path in configuration space, when we reach the temporal target, because of action errors, it may still go into some spaces which are considered as walls by the configuration map. Furthermore, even the simplest ultimate target selection becomes a big issue in real experiments, because the experiment space is very narrow and distance measure is not that easy. And computation speed becomes another limitation. Every attempt may take 10 mins or more.

In the future, if we have time to further improve the algorithm. We may try to implement the algorithm using C or C++ instead of Python to improve the running speed. Also, in this project, in order to make the algorithm stable, we have many edge tests and self-rescue modules to keep it in the best state, like the "Move away from the wall when recomputing" trick as described in section 2.2.3, which are a little bit hacky. In the future, we could change the algorithm structure in a more fundamental way.

5 Conclusion

In this project, we have successfully achieved dynamic path-planning using D*lite based on frontier explorer class and a modified global-planner plugin. We have shown our system is working both in simulation and on real Turtlebot, and it enables path replanning even in narrow space with complicated obstacles which cannot be seen at the beginning. If time allows, further work will be done in reducing the time complexity.

References

[1] J. H. Reif, Complexity of the movers problem and generalizations, In Proceedings IEEE Symposium on Foundations of Computer Science, pages 421427, 1979.

[2] Maxim Likhachev, Search-based Planning with Motion Primitives, Carnegie Mellon University

[3] David T. Wooden, Graph-based Path Planning for Mobile Robots, ECE, Georgia Institute of Technology, 2006

[4] Hart, P., Nilsson, N., and Raphael, B., A formal basis for the heuristic determination of minimum cost paths, IEEE Transactions on Systems Science and Cybernetics, pp. 100-107, 1968.

[5] Stentz, A., The focussed D* algorithm for real-time replanning, in Proc. of the Intl Joint Conf. on Artificial Intelligence, 1995.

[6] Koenig, S. and Likhachev, M., Fast replanning for navigation in unknown terrain, Transactions on Robotics and Automation, 2005.

[7] 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.

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

[9] 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.

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

[11] 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.

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

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