Botlab 3: Implementing Planning and Exploration

Max Ernst
University of Michigan
maxernst@umich.edu

Andrew Greer
University of Michigan
alogreer@umich.edu

Dhillon Patel
University of Michigan
dhipatel@umich.edu

John Pye
University of Michigan
jepye@umich.edu

***Abstract*—In the third and final phase of MBot development, we set about enabling truly autonomous operation. To this end, we first set about implementing A* path planning - to enable obstacle-aware motion - and then developed a autonomous exploration and navigation algorithm. These new features were tested in both simulated and real-world maze environments.**

I. Introduction

Part 3 of Botlab involves combining the SLAM features implemented in Botlab Part 2 with a path planning algorithm to allow for autonomous navigation and exploration of an environment. This requires the creation of an obstacle distance grid based off of the SLAM map, the development of A* path planning to navigate to waypoints in that map, and a method of autonomously determining waypoints to explore unexplored regions.

II. Obstacle Distance

Using SLAM, the robot is able to obtain a map of its environment with a high degree of accuracy. That map is divided between known obstacle spaces, known free spaces, and unknown spaces. In order to navigate the environment, it would be helpful to know how far the robot is from an obstacle at any time so it is able to maintain a safe distance from any obstacles. To do this, we implement an obstacle distance grid. This is a cell-based map in which each cell contains its distance from the nearest obstacle.

This map is populated using the Brushfire algorithm. This algorithm starts by initializing all obstacle cells with a distance of 0 and adding them to a queue. Each cell in the queue is expanded to the cells around it, and the distance of each nearby cell that has not yet been assigned a cell is incremented by 1 and is added to the queue. This is essentially a breadth-first search of cells around any obstacles, so that their obstacle distances may be efficiently obtained.

Algorithm 1 initializeDistances
```
0: for all Cell do
0:     if logOdds(Cell) $< 0$ then
0:         distance(Cell) = $max(float)$
0:     else
0:         distance(Cell) = 0
```

With a map populated with accurate obstacle distances, we are then able to set a "radius", or a maximum acceptable distance the robot can get to any obstacle. This radius can be enforced simply by checking the obstacle distance at any x and y coordinate. As can be seen in Figure 2, this radius leads to the creation of a "safe" set of cells, depicted in white, and an "unsafe" set of cells (through this the MBot may not traverse) depicted in red.

Algorithm 2 setDistances
```
0: for all Cell do
0:     if logOdds(Cell) $< 0$ then
0:         distance(Cell) = $max(float)$
0:     else
0:         distance(Cell) = 0
```

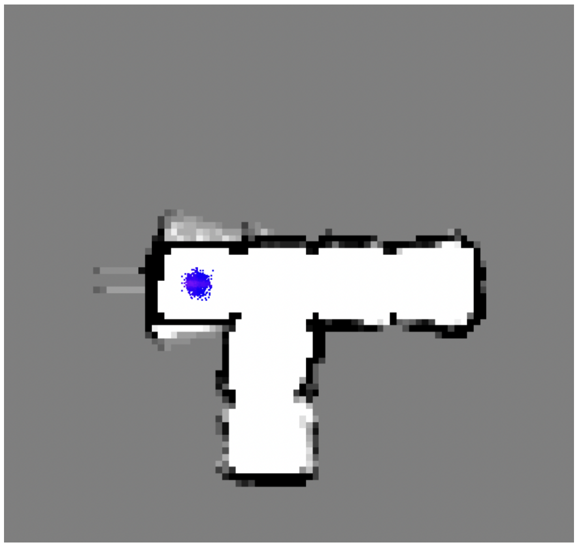
Fig. 1. SLAM Map

III. A* Path Planning

A* path planning is an obstacle-aware path planning algorithm with an acceptable level of optimality, balanced against a reasonably quick compute time.

Once implemented, the A* path planning algorithm was tested in a maze-like environment, to ensure obstacles were appropriately avoided, and the path followed. Figure 3 depicts one path taken by the MBot in green, with the unknown frontier shown in magenta. We can see that the path attempts to maintain distance from obstacles, and it is closely followed by the MBot.

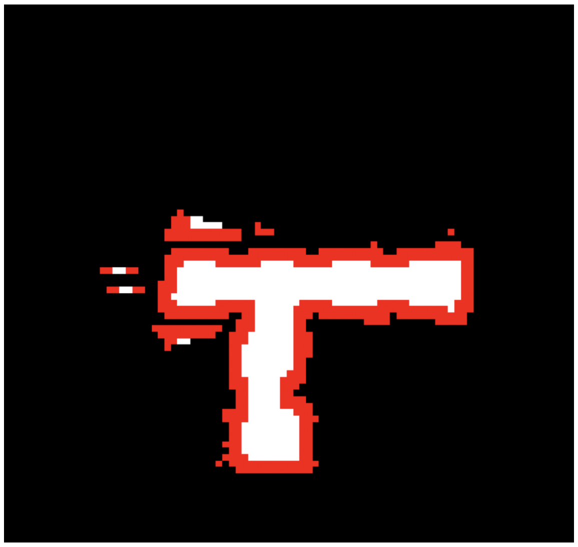
Fig. 2. Distance Grid Applied to SLAM Map

Algorithm 3 A*

```
0: Input: start, goal(n), h(n), expand(n)
0: Output: path
0: if goal(start) = true then return makePath(start)
0: end if
0: open ← start
0: closed ← ∅
0: while open ≠ ∅ do
0:    sort(open)
0:    n ← open.pop()
0:    kids ← expand(n)
0:    for each kid  kids do
0:       kid.f ← (n.g + 1) + h(kid)
0:       if goal(kid) = true then return makePath(kid)
0:       end if
0:       if kid ⋂ closed = ∅ then open ← kid
0:       end if
0:    end for
0:    closed ← n
0: end while
   return ∅
   =0
```

Finally, we set about characterizing the performance of the A* path planning algorithm in a variety of provided test environments, as dpicted in Figures 4 through 9.

We are relatively satisfied with the level of optimality and speed shown in Tables 1 and 2. We note that, on the "narrow constriction" failed trial, the abnormally large maximum planning time is due to timeout. We chose a relatively large number to ensure we did not discard any promising paths. We do not believe this long processing time to be a significant issue; any situation which would lead to a time-out is one in with there is simply no safe path anywhere on the map, and so there is no time criticality. Across the rest of the trials, we see relatively quick median trials, normally falling between 0.01 and 0.1 seconds which - given our limited re-planning logic, activated only when a current path is completed or revealed to be unsafe - is effective. In terms of optimality, Figures 3 and 8 show completed path which appear relatively optimal by visual inspection. Paths are direct, but respectful of obstacles.

Over larger horizons of operation - on the scale of hours, or days - we would likely opt to intensify investigation into path optimality. As it stands, in our small operational environments rich with obstacles, we do not believe an intensive focus on optimality to be warranted.

IV. Map Exploration

For map exploration, the MBot identifies frontiers in its environment and uses those frontiers to plan a path for

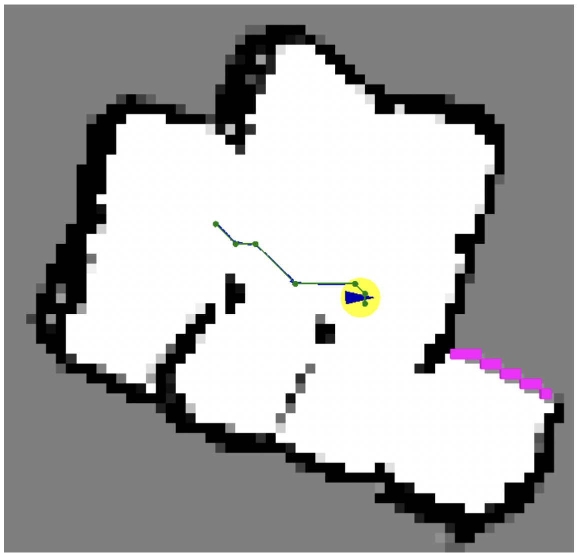
Fig. 3. Real-Life A* Path Executed

Test	Min(us)	Mean(us)	Max(us)	Median(us)	Std Dev(us)
convex	612	612	612	0	0
empty	3808	45692.3	124234	124234	55578.3
maze	1729	7836.5	17218	17218	5880.01
narrow constriction	4182	9590.5	14999	0	5408.5
wide constriction	7903	13184	19189	12460	4635.84

TABLE I
SUCCESSFUL PATH PLANNING EXECUTION TIMES

Test	Min(us)	Mean(us)	Max(us)	Median(us)	Std Dev(us)
convex	100	118	141	100	17.1075
empty	110	138.5	167	0	28.5
maze	102	129	190	110	32.5208
narrow constriction	135	333854	1.00127 e+06	157	471935
wide constriction	139	139	139	0	0

TABLE II
FAILED PATH PLANNING EXECUTION TIMES

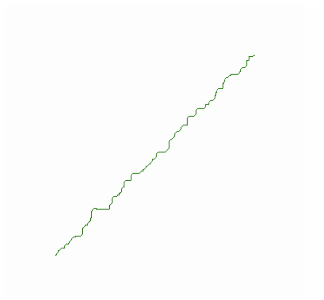
Fig. 4. Test 0: Blank

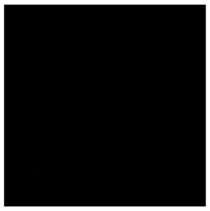
Fig. 5. Test 1: Solid Obstacle

Fig. 6. Test 2: Thin Gap

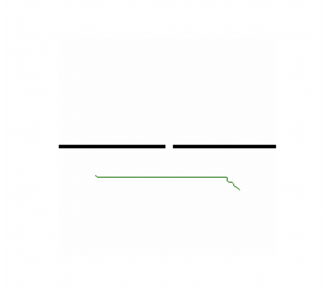
Fig. 7. Test 3: Slightly Less Thin Gap

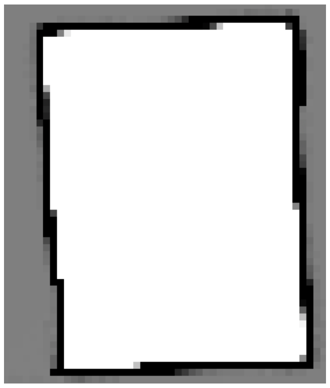
Fig. 8. Test 4: Convex Arena

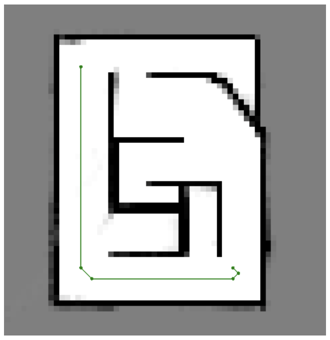
Fig. 9. Test 5: Maze

exploration. Frontiers are borders for unknown areas on a map that haven't been determined to be a wall, an obstacle, or open space, and thus requires further investigation. The algorithm used is a basic Breadth First Search (BFS) to find all the frontiers. Once the initial state of frontiers is known, the MBot follows two methods for planning a path for exploring the frontiers: a check over centroid candidates and a radial search. To plan the path, we go through the centroids candidates to see if they qualify as a possible path, if they do not, we move on to conducting a radial search from each centroid. If both methods can't find a valid path, the search has failed.

V. Map Localization with Unknown Starting Position

Part of map localization is placing the MBot in an unknown location within an environment and it needing to localize itself. For this portion of the lab, the MBot is intended to move about the environment without collisions and without the use

Algorithm 4 plan_path_to_frontier

```
0: Input: frontiers, robotPose, map, planner
0: Output: path
0: for each frontiers do $find\_centroids\_by\_distance()$ $sort(centroids)$
0: end for
0: for each centroid do bool path_to_centroid = $check\_path\_to\_centroid()$
0:    if !path_to_centroid then find_suitable_frontier_cells() and radial_search()
0:    end if
0: end for
0: distance(Cell) = $max(float)$
0:
0: distance(Cell) = 0 =0
```

of A* planner, until the distribution has been reduced enough that the MBot can accurately know where it is. However, due to our high-speed SLAM algorithm, with LIDAR updates at least 10 times per second, we are able to largely avoid the complications of unknown starting positions by simply activating SLAM first, and exploration second. The few-second delay provides ample time to localize the robot, and centralize the estimation particles. Furthermore, our exploration logic continuously re-evalutes paths for their safety and progress. SHould a path be discovered to be unsafe - through, for example, a better map - the path is re-planned with the updated map. This logic further abrogates the need for more specialized map convergence monitoring.

VI. Conclusion

After completing the third and final part of the Botlabs, we now have a fully functioning autonomous robot. We are able to start in an unknown position, map our surroundings, characterize the boundaries of known terrain, and map them. In its present state, the MBot is a functional, if simplistic, robot. Our next steps are now to extend on the MBot's functionality in the independent design phase of the class, to apply the solid foundations of its autonomous operation to a useful purpose.

References

[1] EECS 467 WN24 Slides, Google Drive Folder. Available online: https://drive.google.com/drive/folders/1k2LsNlQYenkD3fTVSE6PTB4Cy1tz1eGF?usp=drive_link (accessed on February 7, 2024).

[2] Thrun, Sebastian; Burgard, Wolfram; Fox, Dieter. The MIT Press, 2006. ISBN-13: 978-0-262-20162-9.