Robotics: Exploration of Unknown Environments

Nikolaos Manginas
Ilyas Ibragimov

April 29, 2018

Abstract

This report outlines the methods followed in order to create a fast exploration algorithm for application in robotic systems. The explorer runs with a reactive controller which is used to manage sudden changes in the state of the environment, therefore allowing for a smooth exploration process in completely unknown environments. The framework used for simulation is ROS(Robotics Operating System) and the software was developed in Python within the Unix Operating System. Code used for completion can be accessed by cloning the repository available on `https://github.com/nickmagginas/Comp313p_CW2`

1 Investigation of the Mapping System

In the first part of this report, we are tasked with investigating the properties of a robot's mapping system in ROS and exploring several changes that can be made to improve its performance. In this section, a robot is instructed to travel to several pre-defined waypoints in a known environment. As the robot travels, it uses its sensors to attempt to recreate its observed environment in an internal map. We wish to investigate the difference in the robot's mapping quality as several parameters are changed. Further, we examine a technique called 'scan matching' that is commonly used in robotic mapping systems to increase mapping performance.

In order to investigate the mapping system we are provided with two launch files that although are very similar in function, have markedly different performances. After studying the function of both files, part_1_1_mapping.launch and part_1_2_mapping.launch, we observe that the former has boolean enable_change_mapper_state set to true, while the latter sets it false. This has an effect in comp313p_mapper.mapper_node.py where self.enableMapping in turn is set to the truth value of enable_change_mapper_state. Then if self.enableMapping is false, the function laserScanCallback is disabled and the robot is unable to process the scans and update the map accurately. Following, we ran both files to completion to compare their performance with an expectation that part_1_2 would have inaccurate mapping.

Indeed as expected, the robot from part 1 had no issues mapping the factory environment, with a very accurate recreation shown in figure 1. However, as shown in figure 2, the robot from the second launch file with a disabled laser scan has issues with accurately mapping many of the obstacles. We see many 'shadows' of obstacles appear on

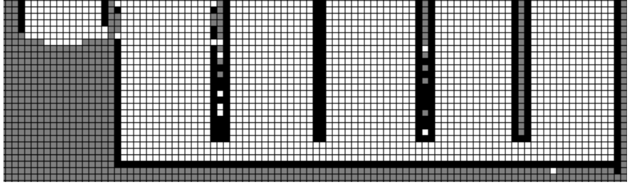


Figure 1: Part_1_1_mapping.launch mapping of the known environment after traversing a set of waypoints.

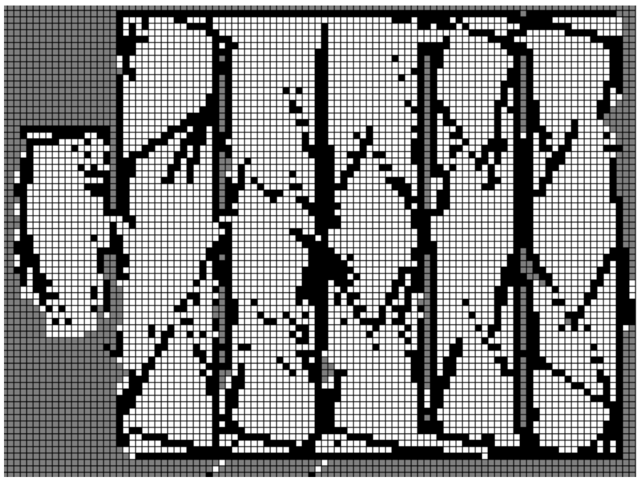


Figure 2: Part_1_2_mapping.launch mapping of the known environment after traversing a set of waypoints with standard speed.

the map where there are none. These are caused by obstacles being detected that are further than the accurate sensor detection range. We found that increasing the linear velocity of the robot by a factor of 1.5 from 10 m/s to 15 m/s in move2goal_controller, drastically reduced the mapping errors with the second launch file. Figure 3 illustrates that the sped up robot with a disabled laser scan seems to even outperform the robot with laser scanning enabled in mapping quality.

A common technique in a full SLAM (Simultaneous Localization and Mapping) system to reduce mapping errors is called 'scan matching'. Scan matching attempts to find a transformation that aligns two point sets that may have different translational and rotational offsets. In our case, it aligns a map to a scan or a scan to a scan. Effectively, this allows us to obtain a rough image of the environment discovered by the robot without having to update the map. For example, if we store several consecutive laser scans that were taken at different poses of the robot, we can perform scan matching to align them together to get a representation of our occupancy grid without ever having updated the map at the time of a scan. Although the result is referred to as a 'rough' image of the map, scan matching usually results in a significant increase in mapping quality as many of the algorithms help eliminate scanning errors during alignment [1] [2] [3]. The most widely used scan matching algorithms are Iterative Closest Point (ICP) and its variations. As the name implies, it attempts to iteratively improve the transformation between two point sets, where one point set is kept fixed and the other transformed to best match the fixed set based on an error metric, usually distance between the points in each set. This iterative minimization is achieved by repeatedly assuming the closest points in each set correspond to each other and performing an appropriate transformation. When the starting positions of both point sets are close enough, eventually the algorithm converges and aligns the point sets [4]. This is illustrated in figure 4 where the initial closest points do not align the point sets together accurately but in figure 5, after several successive iterations the point sets are close enough together to obtain an accurate transformation.

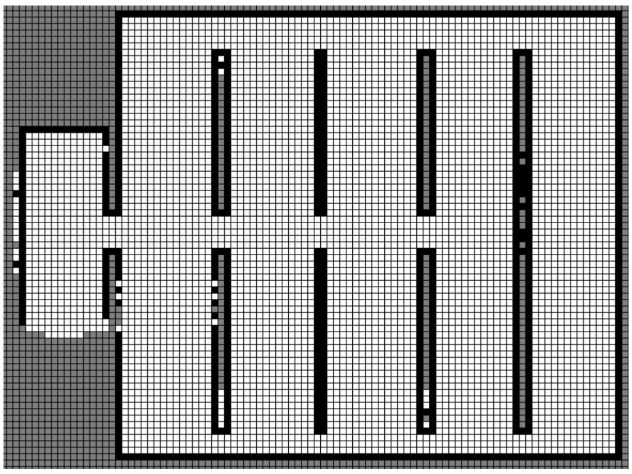


Figure 3: Part_1_2_mapping.launch mapping of the known environment after traversing a set of waypoints with a sped up robot controller.

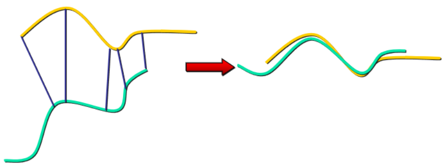


Figure 4: Iterative Closest Point (ICP) algorithm alignment and transformation visualization step 1. The transformation will not be accurate until the two point sets are closely aligned [4].

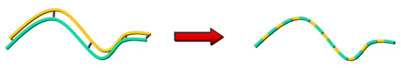


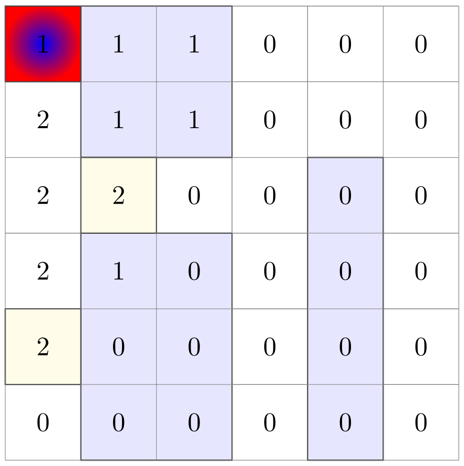
Figure 5: Iterative Closest Point (ICP) algorithm alignment and transformation visualization step 2. The two point sets are closely aligned and the algorithm converges to the accurate transformation [4].

2 Exploration System Implementation

Exploration is crucial for the vast majority of robotic applications. It is rare for the environment in which a robot acts to be completely known and even if it is, unpredictable changes to it make it so that an exploration algorithm that can detect obstacles and update the state of the environment is usually necessary.

The primary aim of an exploration system is to be able to completely explore a map, with a small amount of time and a small distance traveled being the secondary objectives when designing the algorithm which governs the exploration. Therefore, an ideal explorer would explore the entirety of an environment in the smallest time possible (fast operation) and the smallest distance traveled (power-efficiency). The aim is to adhere to these designs constraints while constructing a simple and dense algorithm.

The environment, from now on referred to as the map, can be expressed as a grid like object containing a finite amount of cells. Each cell with coordinates (x, y) can be either explored and known to be open, or explored and known to be closed or unexplored. We can arbitrarily assign numerical values to each of the above states. We can denote unexplored cells with a zero, explored and open cells with a two and explored and closed cells with a one. A small map could then be expressed as grid-like object similar to that of figure 6.



1	1	1	0	0	0
2	1	1	0	0	0
2	2	0	0	0	0
2	1	0	0	0	0
2	0	0	0	0	0
0	0	0	0	0	0

Figure 6: 6×6 map depicting an exploration process

Let us assume that the robot starts at the origin of Figure 6. The position is colored in the figure above. The current position of the robot is $(0, 0)$ and is of type 1. It is an explored cell that is closed. Note, that this only holds true if the robot is in this exact position. If not, the state of the cell would be 2, that is explored and open. Assume that we start the simulation and the scans from the robot are registered and analyzed. The obstacles, which we denote with blue in the figure are registered now as explored and of type 1, that is they are not accessible but are now known. Cells that are within the range of the scans and are open are labeled as 2 since they are known and can be accessed. All other cells which the scans do not process are still of type 0. The aim is to be able to explore all cells on the map, or using this representation we should have no cell that is denoted with a zero. To be able to do that we use a cell type, which we call a frontier cell. A frontier cell is any cell that is open and has at least one unknown neighbor. Frontier cells are shown in the figure with yellow fill. Using this representation, a frontier cell must be of type 2 and must have at least one neighbor of type 0.

The most important task of the explorer is to be able to continually update the frontier cells and from these choose the one that is the "best" to go to. Obviously, there is no absolute truth as to whether a cell is the "best" since that is strongly dependent on the rest of the map that is still unknown. Therefore, we must use some sort of score to order the frontier cells in such a way that we pick an optimal cell at each iteration. To determine what cells are optimal we don't have to look further than the shortcomings of an explorer that randomly selects cells. Identifying the problems in the process with which this explorer filters the map will allow us to understand what features a selection strategy must have.

The default planner as provided in the Github repository picks the first frontier cell that it sees. That is a pseudo-random process which obviously is under-performing. The larger problem associated with the planner is that it generally goes back and forth a lot. That is it could follow the following path:

$$path = \{8,8\} \rightarrow \{8,24\} \rightarrow \{8,7\}$$

Naturally that is suboptimal and it would be a lot more reasonable to follow the path in the order of $\{8,8\} \rightarrow \{8,7\} \rightarrow \{8,24\}$. Therefore we should definitely use distance as a criterion when shorting the frontiers. Cells that are closer should generally be rated higher than those that are far away. Having considered distance, another criterion that could be taken into account is some sort of measure of the amount of unexplored cells in the vicinity of each frontier cell. Generally, it is optimal to consider frontiers that have many unexplored cells around them first. Considering the above we can calculate a score for each frontier cell.

Algorithm 1: Choose Frontier

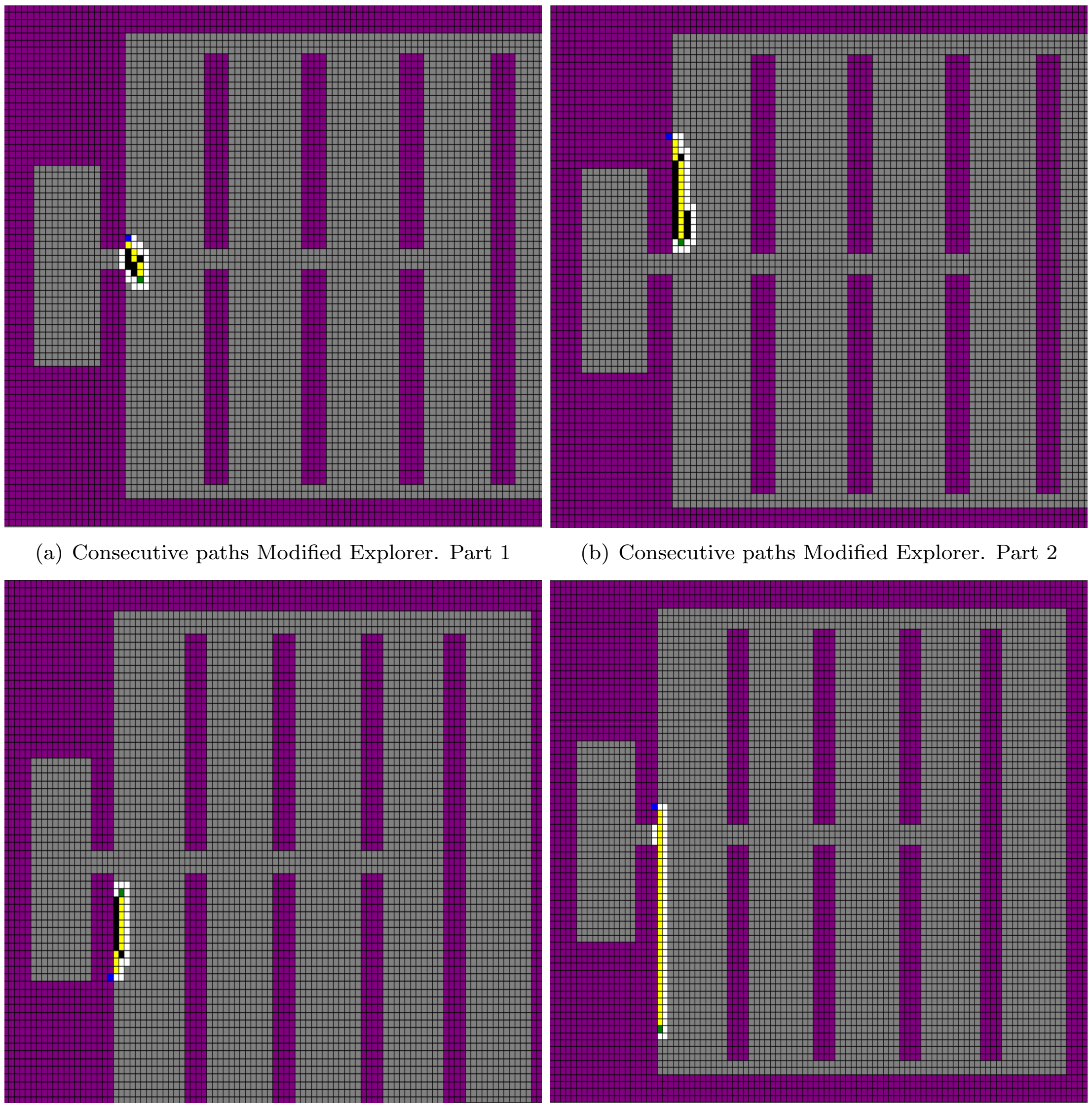
```
1:  function CHOOSE_NEW_DESTINATION($frontier\_list$)
2:      for $i$ in $frontier\_list$ do
3:          $distances.append(eucledian\_distance(current\_cell, i))$
4:          $neighbours.append(check\_unexplored\_neighbours(i))$
5:      end for
6:      $neighbours = neighbours * max(distances)/max(neighbours)$
7:      $score = average(neighbours, distances)$
8:      $frontier\_list = sort(frontier\_list, score)$
9:      for $i$ in $frontier\_list$ do
10:         if $i$ not in $blacklist$ then
11:             $target = i$
12:             $blacklist.append(target)$
13:             break
14:         end if
15:     end for
         return target
16: end function
```

The frontier selection strategy as implemented to improve the performance of the default exploration system is described in the algorithm above. As mentioned before, the selection is based on both the distance of each frontier cell to the current position as well as the amount of unexplored space in the direct vicinity of a frontier cell quantified by the number of unexplored neighbors within some distance of each frontier. More specifically, we can construct two arrays of which the former contains the distance of each frontier from the current position and the latter the number of unexplored neighbors in the vicinity of each frontier. The number of unexplored neighbors is in reality represented as a negative number. This is done because we wish to minimize distance and maximize the number of unexplored neighbors, but it is easier to minimize both of them and therefore we express the number of neighbors as a negative number. After we construct both the arrays we scale one relative to the other to achieve an equal weighting between the two. We then compute a score for each of the frontiers by taking the average of distance and number of neighbors and sort the frontier array by score in ascending order(smallest scores first). The first element that is not in the blacklist is finalized as the next target.

Note: The blacklist is used to denote cells we never wish to choose as targets, either because we have already been there or because we know them to not be accessible.

This approach solves the problems faced when using the default pseudo-random controller. As mentioned above those include going back and forth, shown in Figure 7, as well as prioritizing cells which have a small amount of unexplored neighbors. As can be seen in Figure 7 the improved controller generally prioritizes closer cells while the default one is pseudo-random and thus this back and forth behavior occurs. Implementing this frontier selection strategy results in a large improvement in performance. The total runtime for the default explorer to achieve full exploration of the map was 1655 seconds while only 1000 seconds were necessary when using the above heuristics to govern the search. The distance traveled was also less for the modified explorer, from 2048 to 1279 meters, and the number of waypoints also decreased from 855 to 512. The angle turned remained almost constant increasing by a very small increment from 424 to 458 radians.

On the whole, using heuristics to allow for sorting of the list of available frontiers makes it so that the explorer has a large increase in performance. However, in order for the algorithm to function correctly we also need to manage the frontier cells. In order to do that we iterate though the map and if a cell meets the criteria for a frontier cell we append it to the frontier list. To minimize problems we also use a blacklist explained above in which we add cells that we have been to as well as cells which we know we cannot get to for any reason. This function runs whenever the map updates, i.e. when new cells are explored.



(a) Consecutive paths Modified Explorer. Part 1 (b) Consecutive paths Modified Explorer. Part 2

(c) Consecutive paths Default Explorer. Part 1 (d) Consecutive paths Default Explorer. Part 2

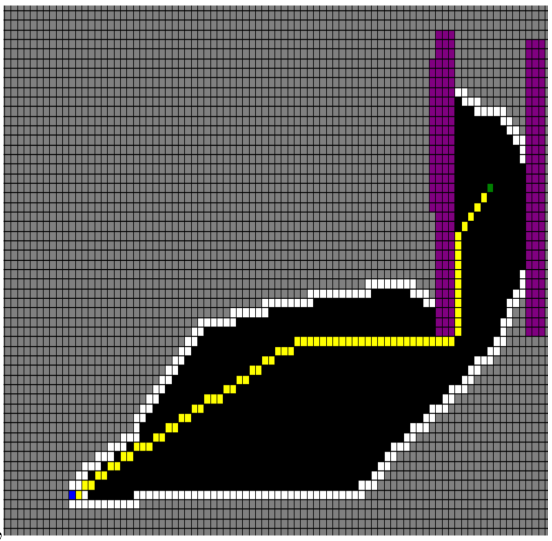
Figure 7: Improved selection eliminates back and forth movement

3 Reactive Planner Implementation

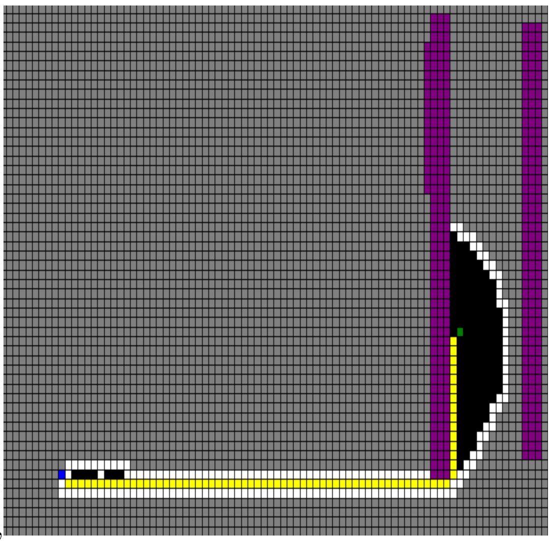
In this section we will be outlining the reactive planner implementation for our robot which uses feedback (closed loop) from the mapping system to adjust its pathing dynamically in order to react to obstacles. In this case, we assume that we do not have prior knowledge of our environment and we want to try to reach several predefined waypoints in our warehouse map. This is known as a Simultaneous Localisation and Map Building (SLAM) problem, where we need to keep track of both the robot position and environment. Our robot stores an internal map of the environment that is updated based on its sensors as it explores the space. This map stores either the known state of the cell (occupied or unoccupied) if explored or the probability that the cell is occupied if it is unexplored.

Using the information gathered in its internal map, the robot plans a potential path to its destination. While following this path and updating the map as it goes, the robot continuously detects whether the path is blocked by an obstacle. If the current planned path is found to be blocked by an obstacle based on a recent map update, a different path is planned with the new information. This process repeats until either the destination is reached or the map is sufficiently explored so that it is clear there is no path possible to that point. As long as we are not in a dynamic environment, when the entire map is explored once, this becomes a known environment problem and the reactive elements aren't used.

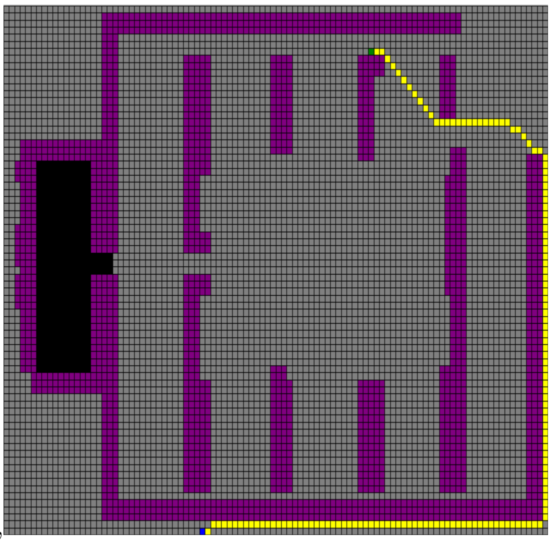
In order to finish the reactive planner implementation as specified in part 3 of the assignment, we implement the checkIfPathCurrentPathIsStillGood function in the ReactivePlannerController. As the robot drives along a planned path and discovers more of the environment, this function checks whether any of the waypoints on the path are now known to be occupied (i.e. by an obstacle), and if so it stops the robot and recomputes the planned path considering the new information. Figures 8(a) and 8(b) visualize this iterative process and the way the planner reacts to new obstacles. After instructing the robot to drive to several accessible waypoints, in figure 8(a) we see the robot attempting to drive on a path that is blocked by obstacles. In figure 8(b) we see that after the robot discovers that the previous path is blocked, a new and now possible path is recomputed with the updated map representation. When we instruct the robot to travel to a waypoint that is clearly inaccessible, in figure 8(c) we observe the robot exploring the entire relevant portion of the map as it tries to find a non-blocked path, then ending once it can see there is no possible path to that point, shown in figure 8(d).


,

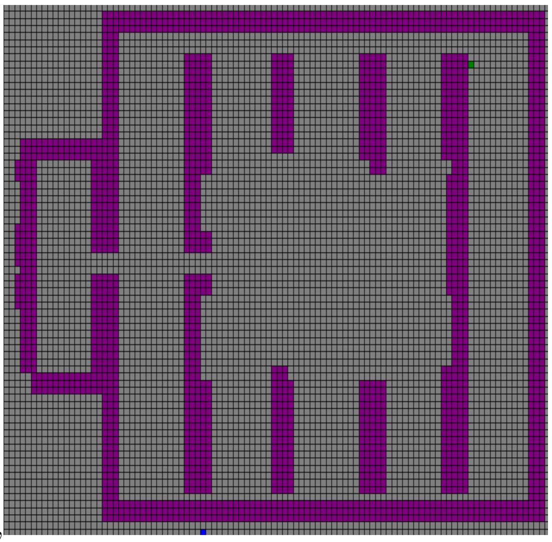
(a) Reactive planner implementation for part 3 attempting to drive to a way point on the currently unknown factory scenario map. If the robot continues to follow this path it will find that it is blocked by obstacles and recalculate the path as shown in figure 8(a)


,

(b) Reactive planner implementation for part 3 after attempting to follow the path in figure 8(a). It can be seen that the path has now changed in reaction to the newly discovered obstacles. This path in turn is not blocked (as visible by comparison with the factory map) and will not be recomputed.


,

(c) Reactive planner map representation for part 3 during an attempt to navigate to an inaccessible point outside of the factory borders. It can be seen that almost the entire factory map has been explored at this point.


,

(d) Reactive planner map representation for part 3 after attempting to navigate to an inaccessible point. The map is now sufficiently explored and it is clear that there is no path to the point outside of the factory, so the process ends.

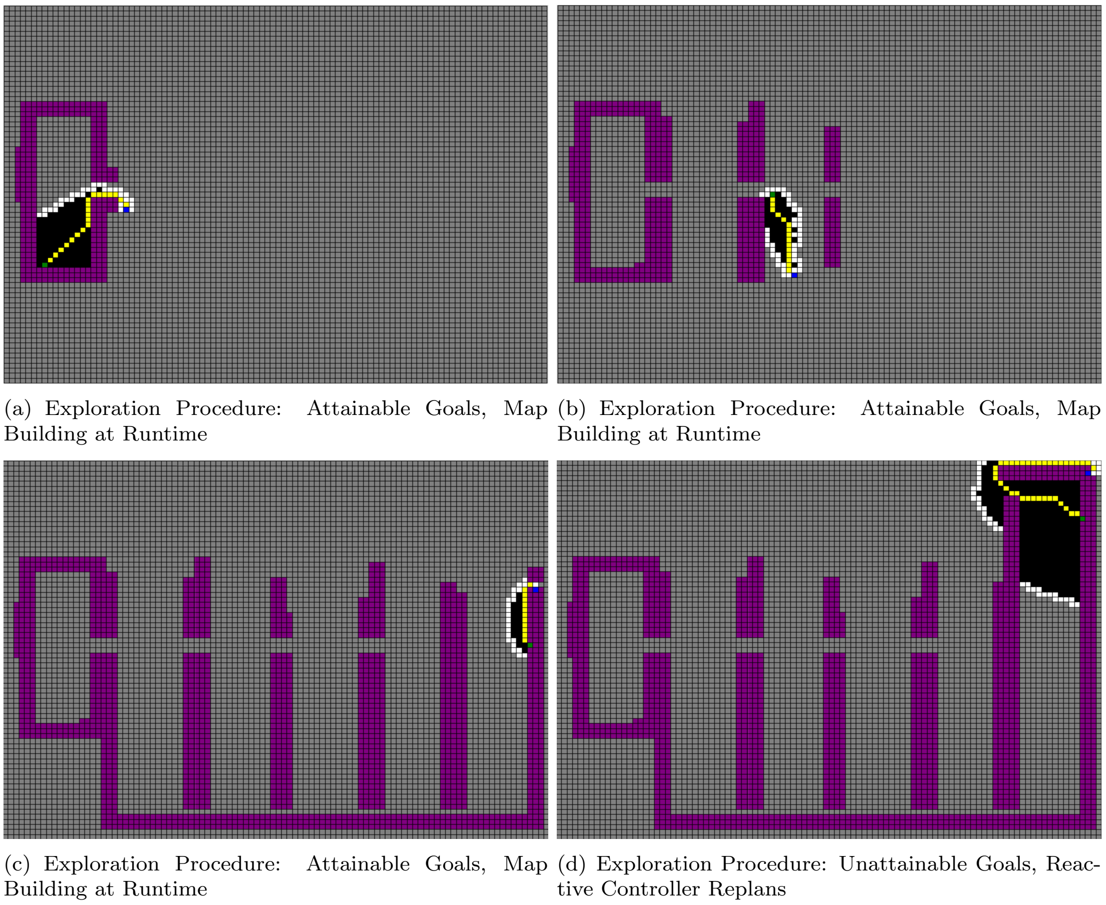
Figure 8: Reactive Planner Controller Behavior

In our reactive planner implementation, each time we re-plan the path we use the A* algorithm, which requires significant processing power for a large environment and would result in long idle times during computation. A* is inefficient here because on a map update where the path needs to be recalculated, only part of the planned path needs to change, but A* recalculates it in its entirety. D* Lite is a novel solution that extends A* to solve this problem by reducing unnecessary computations. D* lite and similar algorithms only recalculate the parts of the planned path that need to change in order to maintain the shortest path. This is accomplished by keeping track of a 'consistency condition' for each explored cell in the map, which checks whether the cost to get to the node has changed (with infinite cost representing an unreachable node). In turn, the D* lite algorithm is able to only operate on the nodes with changed costs, saving extensive computation time [5].

Although D* Lite offers a substantial reduction in computation costs for a reactive planner, it can still require significant processing power. With our previous approach to reactive planning in an unknown environment, we plan as if we know the world perfectly and re-plan when necessary. Further methods exist called Markov Decision Processes (MDP's) that model the uncertainty present in our unexplored environment from the beginning, with no assumption of perfect world knowledge. This uncertainty is modeled as unknown actions which nature can take. Building on these possible nature actions, the actions space of the robot, that is all legal actions it can take at a given state are weighed not only using some sort of arbitrary heuristic and from that computing a relevant cost function but also by taking into account the possible nature actions. By incorporating this uncertainty into the decision process, performance tends to increase.

4 Part 4: Putting it All Together

In this section we assess the performance of the systems developed above, more specifically the exploration systems and the reactive controller, by combining them and trying to explore a completely unknown map. Although, the explorer was developed with those circumstances in mind, it was tested using a controller which was fully aware of the complete map and therefore could make decisions regarding whether a path was possible. The planner is this case does not have full knowledge of the map and we therefore need the reactive controller to make certain that the robot does not collide by calling the explorer for a new path whenever the current path cannot be executed.


(a) Exploration Procedure: Attainable Goals, Map Building at Runtime

(b) Exploration Procedure: Attainable Goals, Map Building at Runtime

(c) Exploration Procedure: Attainable Goals, Map Building at Runtime

(d) Exploration Procedure: Unattainable Goals, Reactive Controller Replans

Figure 9: Explorer and Reactive Controller Acting together for correct determination of the states of an unknown map

On the whole, this system seems to function correctly. It replans when necessary thus avoiding any collisions. The exploration systems still remains the same using distance and unexplored cell density for frontier selection. It explored almost the entirety of the map in about 1000 seconds, which is significantly better performance than the default benchmark explorer. The top left part of the map was not explored as the controller rendered paths towards there impossible, however that left only about 10 cells unexplored which is not of extreme significance compared to the large dimensions of the map.

5 Conclusions

The sections above outlined the gradual development of a reactive exploration algorithm for use in unknown environments. The final system is able to dynamically construct a map at runtime while being reliant exclusively on readings from its sensors. The theory behind the development of the systems was analyzed as well as specific functional blocks necessary for effective implementation. Certain algorithms which could be used to increase the performance of such a system, specifically D* lite and MDP's, were also briefly introduced in sections above. The final code is available at `https://github.com/nickmagginas/Comp313p_CW2`.

References

[1] "Estimate robot pose with scan matching." *https://uk.mathworks.com/help/robotics/examples/estimate-robot-pose-with-scan-matching.html.*

[2] J. L. Blanco, "Iterative closest point (icp) and other matching algorithms," Oct 2013.

[3] C. Stachniss, "Robot mapping: Scan matching in 5 minutes." *http://ais.informatik.uni-freiburg.de/teaching/ws12/mapping/pdf/slam12-scanmatching-mini.pdf.*

[4] R. Gvili, "Iterative closest point." *www.cs.tau.ac.il/ dcor/Graphics/adv-slides/ICP.ppt.*

[5] S. Koenig and M. Likhachev, "D*lite," in *Eighteenth National Conference on Artificial Intelligence*, (Menlo Park, CA, USA), pp. 476–483, American Association for Artificial Intelligence, 2002.