LONDON'S GLOBAL UNIVERSITY

COMP0037

Report

Exploration in Unknown Environments

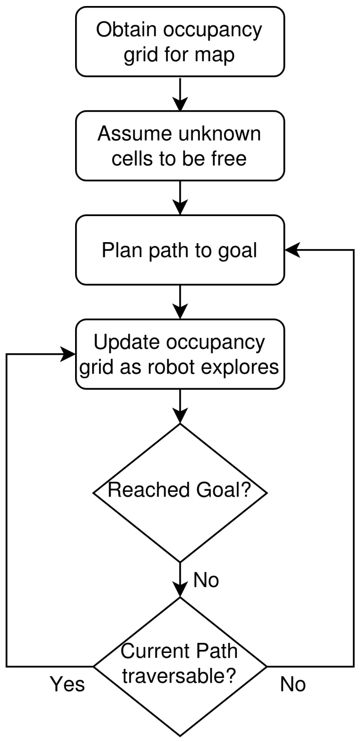
Group AS

Student Name	**Student number**
Arundathi Shaji Shanthini	16018351
Dmitry Leyko	16021440
Tharmetharan Balendran	17011729

Department:	Department of Electronic and Electrical Engineering
Submission Date:	31st of March 2020

Contents

1 Reactive Planner



Obtain occupancy grid for map
Assume unknown cells to be free
Plan path to goal
Update occupancy grid as robot explores
Reached Goal?
No
Current Path traversable?
Yes
No

Figure 1: Flowchart for the reactive planner

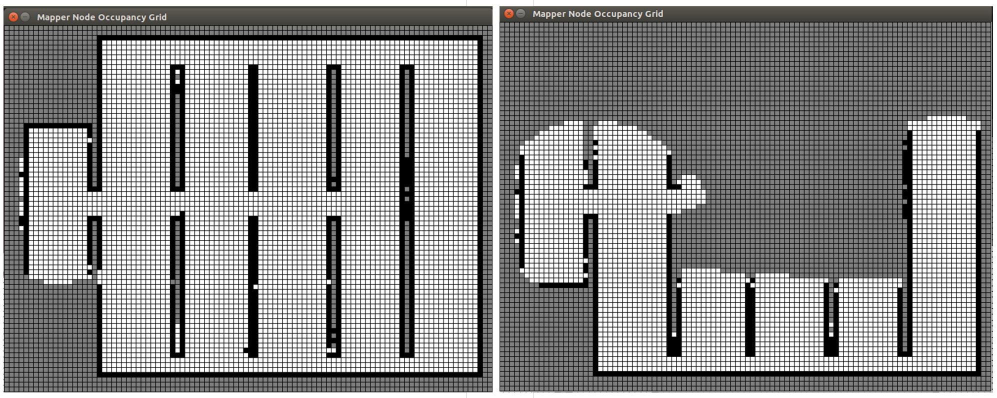
1.1 Our Implementation of a Reactive Planner

A reactive planner is able to adapt its path based on the information it obtains about the environment as it explores it. The high-level functionality of a general reactive planner is described in the flowchart in Fig.1. The reactive planner initially utilizes the available occupancy grid and computes a search grid from this grid assuming any unknown cells to be free. The computation of the search grid instead of utilizing the occupancy grid directly, will prevent the robot from ending up too close to the walls. First, the planner plans a path using this assumption and starts to traverse the path. The robot will explore the environment as it traverses the path and whenever the map is updated the robot checks to see if the originally planned path is still traversable. If the new information suggests that the currently planned path is no longer traversable, then a new path is planned using the new occupancy grid and search grid. This process is repeated until the robot reaches its goal.

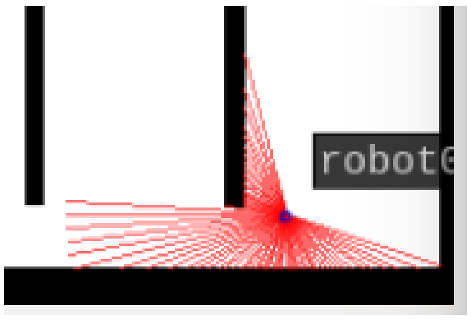
Our implementation of a reactive planner can be found in the file: *reactive_planner_controller.py* in which it can be seen that the callback function *mapUpdateCallback()* is called each time the map gets updated. *mapUpdateCallback()* invokes the function *checkIfPathCurrentPathIsStillGood()* which checks if the cells on the planned path have the label *OBSTRUCTED* in the updated map. If this is True, then *controller.stopDrivingToCurrentGoal()* is called and the path to goal is re-planned with the updated map information. If not, it continues on the same path and the process repeats.

To test this implementation, the robot was set to visit a list of goals on the factory map. The final mapper node occupancy grid is shown in Fig.2a. There were 2 launch scripts provided to us. The one on the left had reachable waypoints whereas the second launch script used an unreachable goal this is why only half the map was explored before the planner realised that it cannot reach the goal. So the process finished cleanly with the message for the second launch script with a message that it cannot find path to goal whereas for the first launch script it ran through all given waypoints. It is clear from the map that there are inaccuracies in the mapping of the world as well as some areas which have still not been explored completely. The inaccuracies occur as a result of rounding errors and other noise due to timing mismatch (caused by networking delays), these errors are mostly corrected as the robot re-explores an area.

The map is updated using the laser range finder that the robot is equipped with. Depending on the reflections of this laser, the environment surrounding the robot can be mapped. Fig.2b depicts an instance in time of the robot. The red lines represent the individual laser beams used to map the surrounding environment. It is clear to see that the laser beam is stopped (and reflected) by the opaque walls. By measuring the time taken for the reflected beam to return, the distance to the opaque object can be calculated. In addition to this, it can also be seen that some of the beams stop despite not having hit any opaque objects. This is due to the maximum measurable range of the laser range detector.



(a) Mapper Node occupancy grid on launch script 1 (left) and 2 (right)



(b) Robot Laser Range Finder

Figure 2: a) Mapper occupancy grid at the end of traversing the set of goals. b) The robot traversing the map with the laser range finder visualized by the red lines. The maximum range as well as the opaque walls preventing the laser from passing through can also be seen.

1.2 Improving the performance of the Reactive Planner

Our implementation of a reactive planner is quite inefficient and may be computationally intense. The current global planner that is used to plan the paths uses an implementation of Dijkstra's algorithm to plan the path. One way of improving the performance of the planner itself is by utilizing the A* algorithm. This way fewer cells will be considered when planning the path resulting in a lower computational cost.

Another method of improving efficiency is to prevent the algorithm from having to re-plan the whole path. In a dynamic environment, obstacles may move in and out of the robot's path. Therefore, sometimes it may be sufficient to take a remedial action to overcome/avoid the obstacle. This may involve simply waiting for the obstacle to pass or in more complicated cases it may involve the robot moving the obstacle out of the way. However, not all obstacles may be overcome in such a way. For example, a newly discovered wall will neither move on its own nor is it possible for the robot to move the wall. In such a case, a local planner may be utilized to plan a path around the obstacle. This involves planning a path to another point on the proposed path and thereby avoiding the obstacle. These local planners are very responsive and generally utilize gradient-based techniques which use physics-based approximations. However, despite being fast and responsive they generally get stuck in local minima.

2 Frontier-Based Exploration System

2.1 Frontier Exploration

A frontier is a region or a group of cells which marks the boundary between the known open space and the unknown/unexplored space of the environment being explored. For a cell to be classified as a frontier cell, it's state must be known to be free and it must have an adjacent cell of which the state is not known. Two frontier cells are considered to be part of the same frontier (i.e. the same boundary) if the adjacent cell that was previously scanned was also a frontier cell. [1]
An example of a frontier can be found in Fig 3. Frontiers are useful in exploring unknown environments as it provides a systematic approach to decide which cell to go to next to continue exploring the unknown portion of the map with a considerable coverage. There are multiple algorithms that involve detecting frontiers and choosing the optimal next destination to allow the robot to maximise it's knowledge of the surrounding. Some of these approaches will be discussed in this chapter.

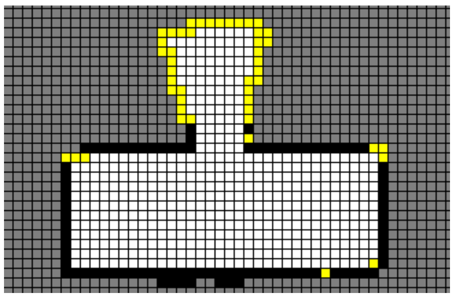


Figure 3: A portion of a partially explored map. Cells marked white are known to be free while cells marked black are known to occupied while cells marked grey are unknown. The cells marked yellow are known to be free and have neighbouring unknown cells (i.e. they are frontier cells).

2.2 Frontier Detection

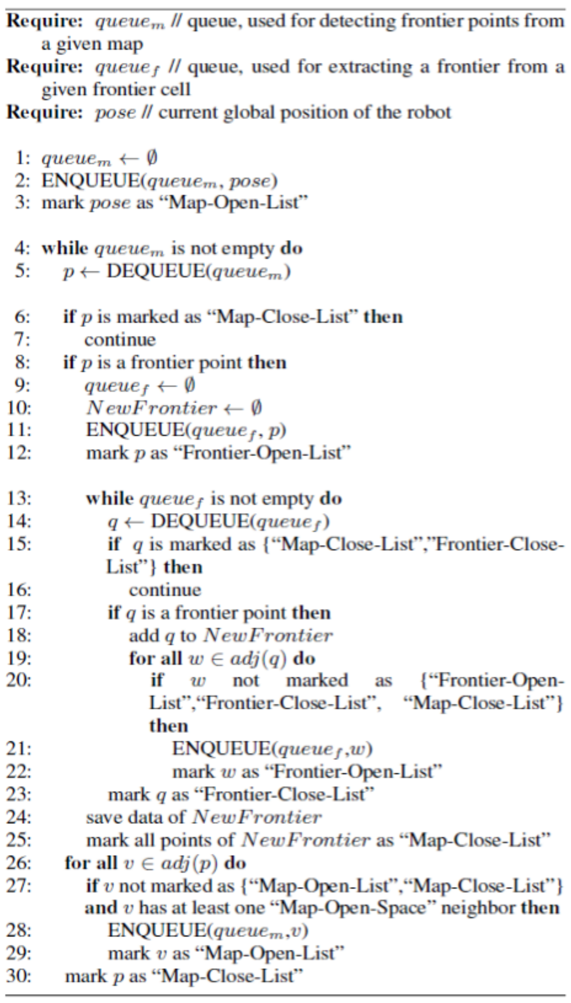
Two algorithms for frontier detection will be discussed. One of them is Wave-front Frontier Detection (WFD) algorithm and the other is Fast Frontier Detection (FFD) algorithm. As the name implies the FFD algorithm is less computationally intense, faster and scales better to larger problems due to its higher efficiency. However, both of these algorithms provide the same result in identifying frontiers.

2.2.1 Wave-front Frontier Detection

The WFD algorithm works by implementing two nested Breadth First Search (BFS) algorithms. This search algorithm utilizes a First In First Out (FIFO) queue to manage the cells that are searched over. The starting cell is set as the current pose of the robot. From this cell, the outer BFS algorithm traverses to its neighbouring cells and checks if each one of them is a frontier cell. If a frontier cell is found, the cell is added to the FIFO queue of the inner BFS algorithm and the neighbouring cells of this cell are checked to see if they are frontier cells as well. If they are, then they are added to the inner queue and this continues until all frontier cells in the cluster are found. Once no more cells are on the queue, we break out of the inner BFS and continue on the outer BFS until another frontier cell is encountered. The pseudocode for this algorithm can be found in Fig. 4a [2]. In this pseudocode the cells are assigned to 5 different states: **Unvisited**, **Map-Open List**, **Map-Close List**, **Frontier-Open List** and **Frontier-Close List**. Initially all cells are unvisited. If enqueued by the outer BFS, the cell is labelled as *'Map-Open List'* and once dequeued by the outer BFS it is labelled as *'Map-Close List'*. Similarly if enqueued by the inner BFS, the cell is labelled as *'Frontier-Open List'* and once dequeue it is labelled as *'Frontier-Close List'*.

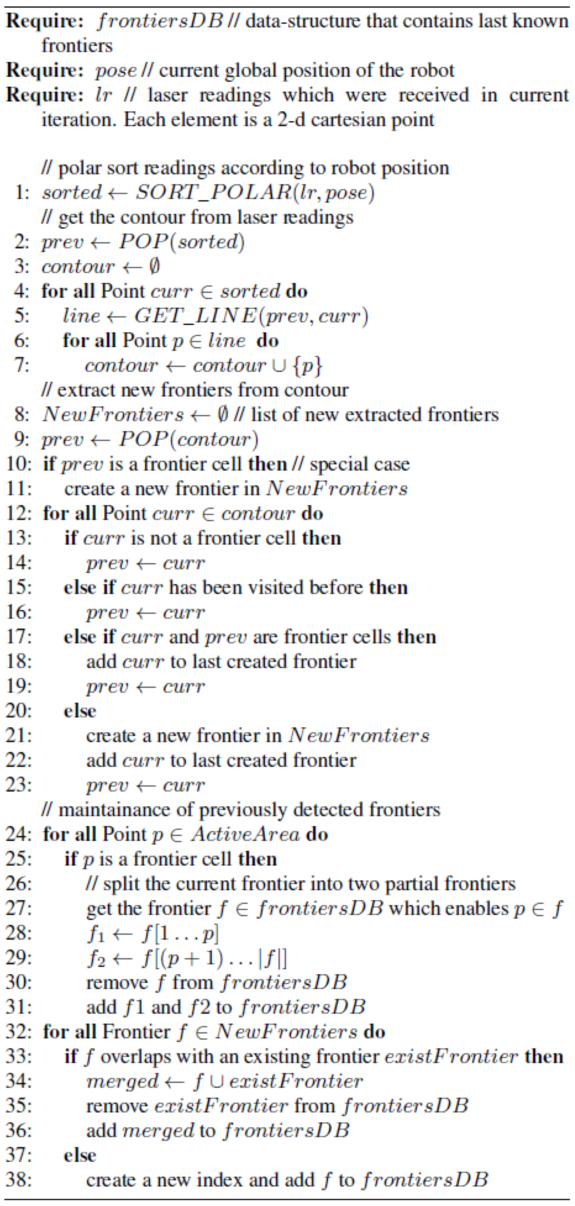
2.2.2 Fast Frontier Detection

The FFD algorithm is an extension to the WFD algorithm. The WFD algorithm on its own identifies frontiers over the whole map every time the map is updated. This means that frontiers that have already been identified that didn't change will still have to be re-identified by the algorithm. FFD overcomes this inefficiency by computing the frontiers of only the cells identified by the robot's laser readings. By doing so, the number of cells that are traversed in the search is greatly reduced, especially if the environment is of a greater size than considered in this coursework (e.g. 3D with environment broken discretized using a higher resolution). Once the algorithm has computed the frontiers of the cells identified by the laser scan, the old frontiers and the new frontiers from the laser scan are compared. The pseudocode for this algorithm is shown in Fig. 4b. This pseudocode tackles the identification of cells from raw laser readings as well which will not be discussed. The main focus is the section of code labelled "maintenance of previously detected frontiers". This section of the code aims to gain information about the frontiers of the whole map by using the old frontiers on the map and the frontiers of the laser scan. In this step multiple checks are performed: if a newly identifier frontier already exists, to see if an old frontier is no longer a frontier and to see if previously separate frontiers are now joined.



```
Require: $queue_m$ // queue, used for detecting frontier points from
    a given map
Require: $queue_f$ // queue, used for extracting a frontier from a
    given frontier cell
Require: $pose$ // current global position of the robot

 1: $queue_m \leftarrow \emptyset$
 2: ENQUEUE($queue_m$, $pose$)
 3: mark $pose$ as "Map-Open-List"

 4: while $queue_m$ is not empty do
 5:   $p \leftarrow$ DEQUEUE($queue_m$)

 6:   if $p$ is marked as "Map-Close-List" then
 7:     continue
 8:   if $p$ is a frontier point then
 9:     $queue_f \leftarrow \emptyset$
10:     $NewFrontier \leftarrow \emptyset$
11:     ENQUEUE($queue_f$, $p$)
12:     mark $p$ as "Frontier-Open-List"

13:     while $queue_f$ is not empty do
14:       $q \leftarrow$ DEQUEUE($queue_f$)
15:       if $q$ is marked as {"Map-Close-List","Frontier-Close-
          List"} then
16:         continue
17:       if $q$ is a frontier point then
18:         add $q$ to $NewFrontier$
19:         for all $w \in adj(q)$ do
20:           if $w$ not marked as {"Frontier-Open-
            List","Frontier-Close-List", "Map-Close-List"}
            then
21:             ENQUEUE($queue_f$,$w$)
22:             mark $w$ as "Frontier-Open-List"
23:       mark $q$ as "Frontier-Close-List"
24:     save data of $NewFrontier$
25:     mark all points of $NewFrontier$ as "Map-Close-List"
26:   for all $v \in adj(p)$ do
27:     if $v$ not marked as {"Map-Open-List","Map-Close-List"}
        and $v$ has at least one "Map-Open-Space" neighbor then
28:       ENQUEUE($queue_m$,$v$)
29:       mark $v$ as "Map-Open-List"
30:   mark $p$ as "Map-Close-List"
```

(a) Wave-front Frontier Detection Pseudocode



```
Require: $frontiersDB$ // data-structure that contains last known
    frontiers
Require: $pose$ // current global position of the robot
Require: $lr$ // laser readings which were received in current
    iteration. Each element is a 2-d cartesian point

    // polar sort readings according to robot position
 1: $sorted \leftarrow SORT\_POLAR(lr, pose)$
    // get the contour from laser readings
 2: $prev \leftarrow POP(sorted)$
 3: $contour \leftarrow \emptyset$
 4: for all Point $curr \in sorted$ do
 5:   $line \leftarrow GET\_LINE(prev, curr)$
 6:   for all Point $p \in line$ do
 7:     $contour \leftarrow contour \cup \{p\}$
    // extract new frontiers from contour
 8: $NewFrontiers \leftarrow \emptyset$ // list of new extracted frontiers
 9: $prev \leftarrow POP(contour)$
10: if $prev$ is a frontier cell then // special case
11:   create a new frontier in $NewFrontiers$
12: for all Point $curr \in contour$ do
13:   if $curr$ is not a frontier cell then
14:     $prev \leftarrow curr$
15:   else if $curr$ has been visited before then
16:     $prev \leftarrow curr$
17:   else if $curr$ and $prev$ are frontier cells then
18:     add $curr$ to last created frontier
19:     $prev \leftarrow curr$
20:   else
21:     create a new frontier in $NewFrontiers$
22:     add $curr$ to last created frontier
23:     $prev \leftarrow curr$
    // maintainance of previously detected frontiers
24: for all Point $p \in ActiveArea$ do
25:   if $p$ is a frontier cell then
26:     // split the current frontier into two partial frontiers
27:     get the frontier $f \in frontiersDB$ which enables $p \in f$
28:     $f_1 \leftarrow f[1 \ldots p]$
29:     $f_2 \leftarrow f[(p+1) \ldots |f|]$
30:     remove $f$ from $frontiersDB$
31:     add $f1$ and $f2$ to $frontiersDB$
32: for all Frontier $f \in NewFrontiers$ do
33:   if $f$ overlaps with an existing frontier $existFrontier$ then
34:     $merged \leftarrow f \cup existFrontier$
35:     remove $existFrontier$ from $frontiersDB$
36:     add $merged$ to $frontiersDB$
37:   else
38:     create a new index and add $f$ to $frontiersDB$
```

(b) Fast Frontier Detection Pseudocode

Figure 4: Pseudocode for the two frontier detection algorithms discussed. [1–3]

2.3 Waypoint Selection

Once the frontiers are identified the robot needs to decide which point, on which frontier, it should travel to next. The next destination should be chosen to maximize the amount of new information that can be obtained. For this there are two main methods, heuristic based approach and information based approach. Using these, the robot will be able to pick the next waypoint to move to.

2.3.1 Heuristic Approaches

To decide where to go to next to continue exploration, heuristics can be used. Two heuristics that are very commonly adopted involve making a decision based on the size of the frontier and the distance to the frontier from the robots currents position i.e. the larger frontier and a closer frontier. Based on the the next point to travel to can be chosen as:

- A cell on a frontier closest to the robot's current position.
- A cell on the largest frontier

- The cell closest to the robot on the largest frontier etc.

The largest frontier heuristic chooses the frontier with the largest number of frontier cells as the frontier to travel to next. Which cell of this frontier to travel to may be decided by the closest cell (smallest Euclidean distance to the robot) or the middle cell of the frontier. The idea of choosing the largest frontier as the next frontier is based on the idea that the largest frontier will give the robot the largest field of view to explore the unknown cells. We implemented WFD using this heuristic such that it chooses the middle cell of the largest frontier as we believe this will provide the largest field of view.

Another heuristic approach can be based on the distance to the next point on the frontier purely based on the time taken to explore. In this case, the frontier cell with the smallest Euclidean distance to the robot in the same direction will be chosen. This approach puts less emphasis on the information gained and just aims to improve the time taken to explore the whole map. This is specially chosen in cases where energy is limited and moving forwards and finishing exploration can be though of as better than stopping to decide where to go next.The frontier size heuristic approach may result in the robot going back and forth between two large frontiers and thereby result in a change in direction or even make the robot halt for a bit as it decides. In such cases, the closest distance heuristic approach is likely to result in the robot covering map well as quickly as possible.

2.3.2 Information Based Approach

This method aims to compute the uncertainty in the map and chooses a point that maximizes the amount of information that can be gained about the surrounding environment. The approach utilizes a probabilistic approach and the concept of entropy. The entropy of a singular cell is computed by the equation given in Eq. 1. This approach will be discussed in more detail in later chapters.

$$H(\mathbf{c}) = -(1-p(\mathbf{c}))\, ln\,(1-p(\mathbf{c})) - p(\mathbf{c})\, ln\, p(\mathbf{c}) \tag{1}$$

[4] discusses how different view-point selection strategies affect different factors of exploration like time taken to explore, path length etc.

2.4 Reference Exploration Algorithm

The implementation of a reference exploration algorithm as provided was purposefully inefficient. To choose a new destination, the algorithm traverses over all the cells and identifies if they are frontier cells or not. The search does not implement a queue but rather starts from the coordinate $(0, 0)$ and uses two for loops to iterate over columns and rows. The chosen destination is set to be the frontier cell closest to the point $(0, \frac{Y}{2})$ where Y is the height of the grid. This uses the assumption that the robot will start off in the central left position (close of the stated coordinate). Therefore, it will initially explore cells that are close to it and slowly work its way to the right. However, due to the fact that this algorithm has no notion of frontier size or current position of the robot, the choice of the next waypoint is suboptimal. The plot in Fig. 6 shows how the exploration time and coverage compare to our own implementation of an explorer algorithm. It is clearly visible from the plot that the reference algorithm is relatively inefficient. It should be noted that the coverage that is computed can never be 100% in the given map. This is due to the fact that the robot cannot gain information about the cells that are enclosed by walls on all its sides or are outside of the traversable area confined by boundaries.

Note that the reference implementation function name has been changed from *chooseNewDestination()* to *chooseNewDestinationOld()*.

2.5 Improving the Exploration Algorithm

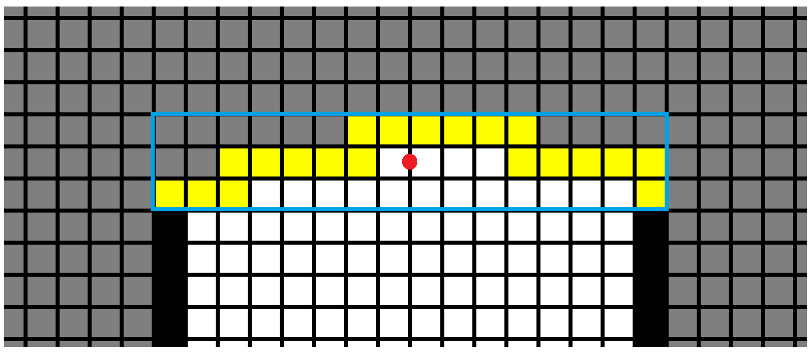


Figure 5: Visualization of the algorithm that is to compute the middle cell of a frontier. The blue rectangle identifies the extremes of the frontiers and is characterized by the minimum and maximum x and y coordinates of the frontier. The red dot is the middle of the rectangle and the middle cell is chosen as the one that is closest to the red dot.

Our implementation of the exploration algorithm utilizes the WFD algorithm to detect the frontiers of the map. The algorithm stores the cells of the current frontier of the map in a priority queue. The priority value is the number of cells in the frontier and the largest frontier is popped first. Every time the map is updated, the WFD algorithm re-identifies the frontiers and updates the list. When a new destination is to be chosen, a frontier is popped from the priority queue and the middle" cell is chosen. This notion of "middle" is computed by considering the maximum and minimum x and y coordinates. An illustration of this is shown in Fig. 5. The cell closes to the centre of the rectangle outlining the frontiers is chosen. If the algorithm chose a waypoint that was not reached by the robot, this cell is added to a black-list to prevent looping and other bad behaviour. Once black-listed the cell will no longer be chosen as a destination to travel to. The performance of this exploration algorithm is shown in Fig. 6 alongside the performance of the baseline reference algorithm. It can be seen that our implementation is quicker at exploring the whole map.

The code implementing this part can be found in *explorer_node.py* and the algorithm is implemented by function *chooseNewDestination()* which then intrinsically invokes the other functions - getFrontiers(), updateFrontiers(), getGoodCells(), getmiddleCell() that together implement WFD which uses the middle cell of the largest frontier as the next point to continue exploration.

In addition to this, we decided to disable the validation of the start cell. The robot sometimes found its way into a place where the occupancy grid was free but the search grid wasn't. Under this circumstance it meant that the validation of the start cell would return false and all possible plans will be unsuccessful. This results in all of the remaining frontiers being black-listed and the algorithm terminating without having explored the map completely.

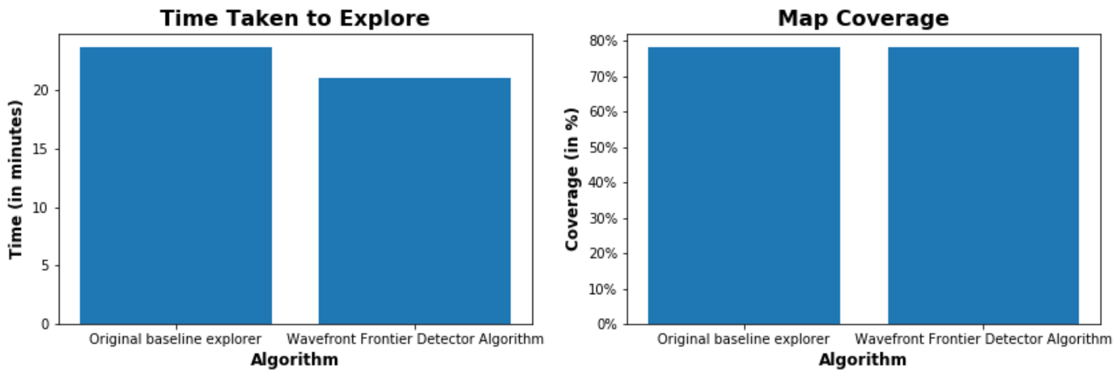


Figure 6: Graphs showing how the exploration time and coverage differ for the reference(baseline) algorithm and our implementation of the explorer algorithm utilizing WFD.

3 Mapping the Environment

The algorithm results obtained in the last chapter utilized a search grid that had knowledge of the real map when planning a route. However, in this example, the planner and the occupancy grid are both set to be unknown when the robot starts. This will test both the reactive planner and the frontier detection in conjunction. Previously, the planner could decide if a destination is reachable, but now the reactive planner will be used to handle this. As previously discussed, the validation of the start cell is disabled to prevent early termination of the planner. The results of the exploration are presented below in Fig. 7.

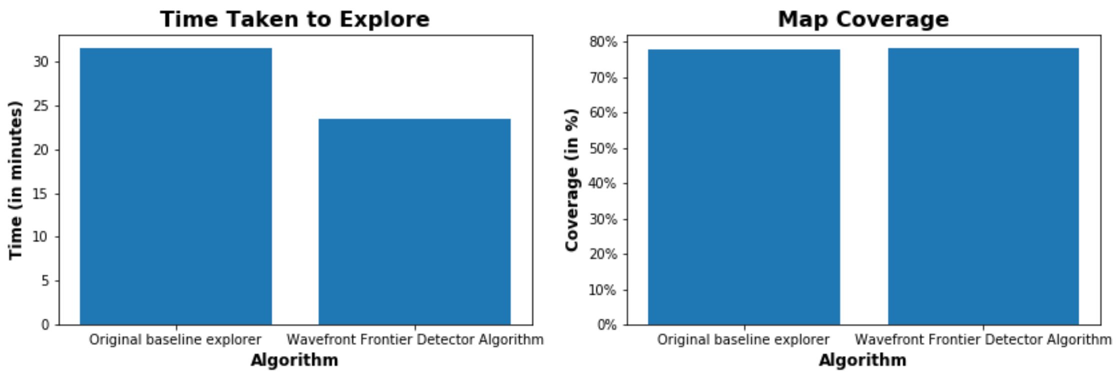
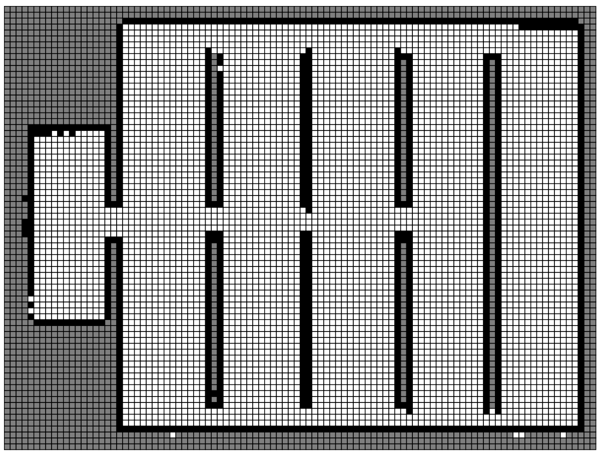


Figure 7: Graphs showing how the exploration time and coverage differ for the reference(baseline) algorithm and our implementation of the explorer algorithm utilizing WFD in the case when the true search grid is not known to the planner.

The final occupancy grids for both algorithms are pretty similar as can be seen in Fig. 8. Both maps are not 100% accurate and have some noise even in the free space that the robot was able to explore. This is most likely due to noise in the sensors and rounding errors in the code. The unknown cells that the robot couldn't get to (within cells and outside of the boundary) were also not explored. In addition to this, both maps feature frontiers. The frontiers in the final map are most likely cells that are black-listed meaning that the robot previously attempted to plan a path to these points and was unable to do so. This is as most the frontiers are single cells with occupied cells on either side. This means that when the search grid is built from the occupancy grid, these frontier cells are also marked as occupied. As the final cell is now occupied the planner returns false and the explorer black lists the cell.


(a) Final occupancy grid with the baseline exploration algorithm

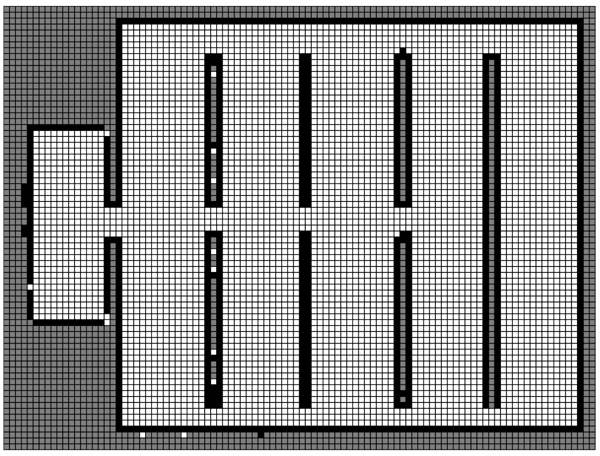

(b) Final occupancy grid with the WFD exploration algorithm

Figure 8: The final occupancy grid for both exploration algorithms

4 Information-Theoretic Path Planning

While the heuristic approach provides a systematic method/rules to explore a map, the Information-Theoretic approach treats a planning exploration problem as an optimisation problem. This way an optimal solution is guaranteed unlike in the previous approach. An optimisation problem usually uses a cost and attempts to minimise it. The cost measure that is minimised here is entropy.

4.1 Description of Entropy and Mutual Entropy

The cost measure for a planning exploration problem has to analyse the explored-ness of the given map. This can also be thought of as the uncertainity in the map. Entropy is a metric that quantifies the uncertainty and hence information theorotic approach involves using entropy as the cost measure that is minimised. For a single cell the entropy would be given by the equation shown in Eq. 2. In this equation, **c** represents a cell and c represents the possible states of the cell. Therefore entropy is a sum over all of the possible states of the cell.

$$H(\mathbf{c}) = -\sum_{c} p(\mathbf{c} = c) \ln p(\mathbf{c} = c) \tag{2}$$

In our example, cells are either known to be occupied/free or unknown. Additionally, the state of a cell can either be occupied or free. Therefore, the above mentioned sum has only two terms. Furthermore, given that the probability of a cell being occupied and the probability of a cell being free sum to 1, the sum depends on only one probability. This simplified version of the entropy of a cell can be found in Eq. 1. In this equation, the probability $p(\mathbf{c})$ represents the probability of the cell being occupied. The occupancy grid for our simplified case consisting of cells with probabilities equal to 0, 0.5 or 1 can be shown to have cells of entropy as follows:

$$\begin{aligned} p(\mathbf{c}) = 0 &: H(\mathbf{c}) = -(1)\text{ln}(1) - \text{ln}(0) = 0 \\ p(\mathbf{c}) = 0.5 &: H(\mathbf{c}) = -(0.5)\text{ln}(0.5) - 0.5\text{ln}(0.5) = \text{ln}2 \\ p(\mathbf{c}) = 1 &: H(\mathbf{c}) = -(0)\text{ln}(0) - 1\text{ln}(1) = 0 \end{aligned} \tag{3}$$

To compute the entropy of the whole map the expression in Eq. 4 needs to be evaluated. This equation, **M** represents possible map realizations (i.e. specific assignments of occupied/free to cells). Here, the entropy is the sum over all possible map realizations.

$$H(\mathbf{C}) = -\sum_{\mathbf{M}} p(\mathbf{M}) \ln p(\mathbf{M}) \tag{4}$$

To compute $p(\mathbf{M})$ we introduce the notation $p_{\mathbf{C}}(\mathbf{M})$ which signifies the probability that an occupancy grid $\mathbf{C}$ can take a specific map realization $\mathbf{M}$. Assuming independence between cells this can be expressed as shown in Eq. 5.

$$p_{\mathbf{C}}(\mathbf{M}) = \prod_i \prod_j p(c_{ij} = m_{ij}) \tag{5}$$

This product can be broken down into two groups of cells: the group of known cells and the group of unknown cells. As this is mutually exclusive, there is no intersection between the two sets. We can therefore rewrite Eq. 5 as a product of the known and unknown probabilities. The probability of the known portion can easily be computed, If the map realization $\mathbf{M}_K$ is equal to the current known map realization $\mathbf{M}_K^{known}$ then the probability is 1 otherwise the probability is 0. This can be described by a delta function which we shall label $\delta_{\mathbf{C}_K}(\mathbf{M}_K)$. Using these the following equations are formulated:

$$\begin{aligned} p_{\mathbf{C}}(\mathbf{M}) &= p_{\mathbf{C}_K}(\mathbf{M}_K) \cdot p_{\mathbf{C}_U}(\mathbf{M}_U) \\ p_{\mathbf{C}}(\mathbf{M}) &= \delta_{\mathbf{C}_K}(\mathbf{M}_K) \cdot p_{\mathbf{C}_U}(\mathbf{M}_U) \end{aligned} \tag{6}$$

By exploiting the nature of the delta function the map probability can be written as the following piecewise function:

$$p_{\mathbf{C}}(\mathbf{M}) = \begin{cases} 0 & \text{if } \mathbf{M}_K \neq \mathbf{M}_K^{known} \\ p_{\mathbf{C}_U}(\mathbf{M}_U) & \text{if } \mathbf{M}_K = \mathbf{M}_K^{known} \end{cases}$$

This piecewise function can be simplified further by considering the simple case studied in this coursework. In this example, cells are limited to 2 different states and if unknown they have the same probability of being occupied or free. Therefore the probability $p_{\mathbf{C}_U}(\mathbf{M}_U)$ can be rewritten as shown in Eq. 7. In this equation, $\mid \mathbf{C}_U \mid$ represents the number of unknown cells.

$$\begin{aligned} p_{\mathbf{C}_U}(\mathbf{M}_U) &= \prod_i p(c_{U,i} = m_{U,i}) \\ &= (0.5)^{|\mathbf{C}_U|} \end{aligned} \tag{7}$$

Now to compute the entropy of the map we consider Eq. 4. Any map realizations where the known portion of the map doesn't match what is already known, will result in a probability of 0 and a 0 sum term. Therefore, these map realizations can be ignored. The only map realizations that we need to consider are those, where the known portion matches what is already known about the world. Given 2 possible states and $\mid \mathbf{C}_U \mid$ unknown cells, there are $2^{|\mathbf{C}_U|}$ different realizations of the unknown portion of the map. Additionally, due to the fact that in this simple case, the unknown cells have equal probability of being occupied and free, the probability of each of these realizations is equal.This results in Eq. 4 reducing to the form shown in Eq. 8.

$$\begin{aligned} H(\mathbf{C}) &= -\sum_{\mathbf{M}} p(\mathbf{M}) \ln p(\mathbf{M}) \\ &= -2^{|\mathbf{C}_U|} \cdot (0.5)^{|\mathbf{C}_U|} \ln (0.5)^{|\mathbf{C}_U|} \\ &= -2^{|\mathbf{C}_U|} \cdot (2)^{-|\mathbf{C}_U|} \ln (0.5)^{|\mathbf{C}_U|} \\ &= - \mid \mathbf{C}_U \mid \ln (0.5) \\ &= \mid \mathbf{C}_U \mid \ln (2) \end{aligned} \tag{8}$$

It can be seen from this that the entropy in our case is directly proportional to the number of unknown cells. Now that we have a way of computing the current entropy of the map, we can utilize this in conjunction with another quantity to decide where the robot should travel to next. For this we introduce the mutual entropy.

The mutual entropy is a measure of how much information is gained and it is this quantity that is to be maximized when choosing the next waypoint. To describe the mutual entropy, let us first assume that the robot travels to a point $\mathbf{x}$ and observes the cells $\mathbf{z}$. The new entropy of the map can be written as $H(\mathbf{C} \mid \mathbf{x}, \mathbf{z})$. As we don't know the actual map, the observation $\mathbf{z}$ that the robot makes is unknown to us. Therefore, we utilize the probability distribution of the unknown cells to produce an expected realization of the cells that are covered by the robot's lasers. This can be computed by producing a probability weighted sum of the Map Entropy over all the possible map realizations(as shown in Eq. 9). This will be used to estimate the entropy of the map once the robot has travelled to the point $\mathbf{x}$.

$$H(\mathbf{C} \mid \mathbf{x}) = \sum_{\mathbf{C}} p_{\mathbf{C}_k}(\mathbf{M}) H(\mathbf{C} \mid \mathbf{x}, \mathbf{z}(\mathbf{M})) \tag{9}$$

The mutual entropy is the information gained by travelling to a point $\mathbf{x}$ and is given by the expression in Eq. 10

$$I(\mathbf{x}) = H(\mathbf{C}) - H(\mathbf{C} \mid \mathbf{x}) \tag{10}$$

The optimal cell to travel to $\mathbf{x}^*$ is then given by the following maximization problem:

$$\mathbf{x}^* = \underset{\mathbf{x}}{\operatorname{argmax}} \, I(\mathbf{x}) \tag{11}$$

4.2 Evolution of Entropy over Time

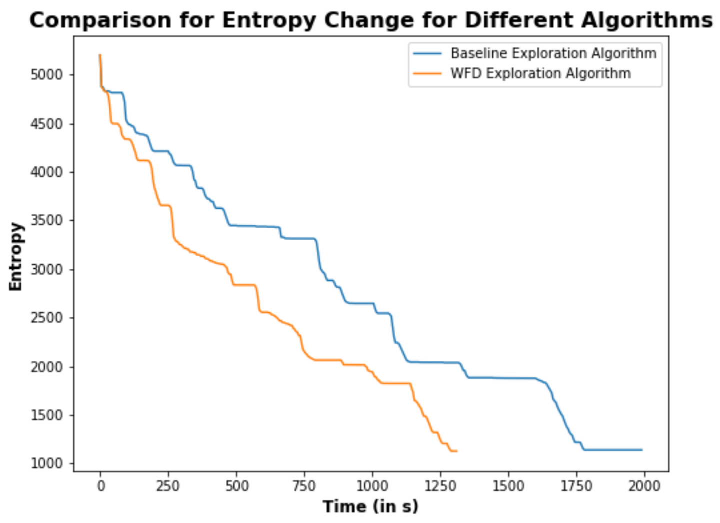


Figure 9: The evolution of entropy over time for two different exploration algorithms.

Using the equation for the entropy from Eq. 8, the entropy of the map was computed at 5 second intervals. This was done for both the baseline algorithm and our implementation of the WFD algorithm. This was done by modifying run() function in *mapper_node.py* to call the function *computeMapEntropy()* every 5s. *computeMapEntropy()* then invokes *computeCellEntropy()* which calculates and returns the entropy of each cell based on Eq. 8. *computeMapEntropy()* then sums the entropy of individual cells and returns it.

The entropy values calculated were then stored in a csv file and were plotted. The results are shown in Fig. 9. There are some distinct features in this plot. We can observe that both plots decrease as time evolves. This makes sense as when the explorer gains more knowledge about the environment, the uncertainty in the map decreases resulting in a lower entropy. In addition to this, a step like behaviour can be observed. This can be explained by the fact that the robot will not be gaining new information always. For example, if a robot travels through a known region to get to a frontier, no new information will be gained for the time it travels through the known region. This results in no change in the map entropy.

We can also see that both explorers reach a similar final entropy. This is backed up by Fig. 7 where both explorers have a similar final coverage. Due to the fact that entropy is proportional to the number of unknown cells, a similar coverage will result in a similar final map entropy. A notable difference between the two traces is that despite reaching the same final entropy value, our WFD implementation reached this value much quicker in comaprison to the baseline algorithm. In addition to this, the gradient of the entropy plot is higher for our WFD implementation than the baseline algorithm. This suggests that our WFD implementation is able to gain information about the environment quicker than the baseline algorithm. Therefore, we are able to conclude with confidence that our WFD implementation is better than the baseline explorer algorithm.

References

[1] M. Keidar, E. Sadeh-Or, and G. A. Kaminka, "Fast frontier detection for robot exploration," in *International Conference on Autonomous Agents and Multiagent Systems*. Springer, 2011, pp. 281–294.

[2] A. Topiwala, P. Inani, and A. Kathpal, "Frontier based exploration for autonomous robot," *arXiv preprint arXiv:1806.03581*, 2018.

[3] M. Keidar and G. A. Kaminka, "Robot exploration with fast frontier detection: theory and experiments," in *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 1*, 2012, pp. 113–120.

[4] C. Stachniss and W. Burgard, "Exploring unknown environments with mobile robots using coverage maps," in *IJCAI*, vol. 2003, 2003, pp. 1127–1134.