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: 17th of March 2020

Contents

1 Reactive Planner **2**
1.1 Our Implementation of a Reactive Planner 2
1.2 Improving the performance of the Reactive Planner 3

2 Frontier-Based Exploration System **3**
2.1 Frontier Exploration 3
2.2 Frontier Detection 4
2.2.1 Wave-front Frontier Detection 4
2.2.2 Fast Frontier Detection 4
2.3 Waypoint Selection 5
2.3.1 Heuristic Approaches 5
2.3.2 Information Based Approach 6
2.4 Reference Exploration Algorithm 6
2.5 Improving the Exploration Algorithm 6

3 Mapping the Environment **7**

4 Information-Theoretic Path Planning **8**
4.1 Description of Entropy and Mutual Entropy 8
4.2 Evolution of Entropy over Time 9

Appendices **11**

1 Reactive Planner

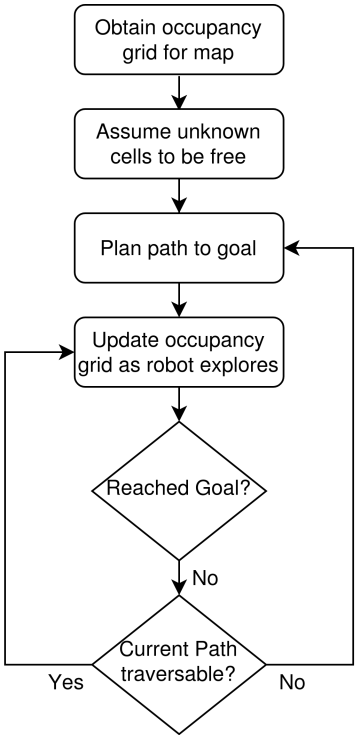


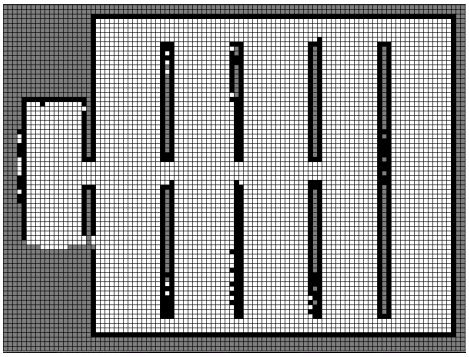
Figure 1: Flowchart for the reactive planner

1.1 Our Implementation of a Reactive Planner

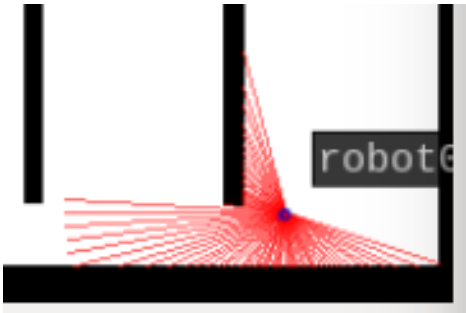
A reactive planner is able to adapt its path based on the information it obtains about the environment as it explores it. The high-level functionality of a general reactive planner is described in the flowchart in Fig.1. The reactive planner initially utilizes the available occupancy grid and computes 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. The planner then plans a path using this assumption and starts to traverse the path. The robot will explore the environment as it traverses the path and if new information suggests that the currently planned path is no longer traversable, a new path is planned using the new occupancy grid and search grid. This process is repeated until the robot arrives at the goal.

As a means of testing the code 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. It is clear to see 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 way the map is updated is 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



(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 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. How-ever 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. 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 group of cells which marks the border between the known portion of the world and the unknown portion of the world. For a cell to be classed 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. An example of a frontier can be found in Fig 3. Frontiers are useful in exploring unknown environments as travelling to these cells will allow the robot to find out more about the unknown portion of the map. There are multiple algorithms for 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.

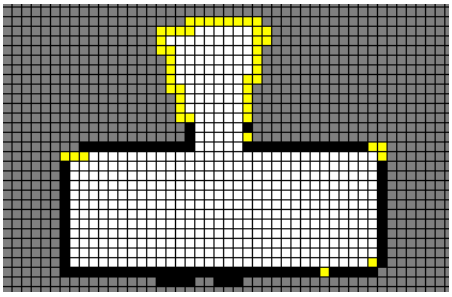


Figure 3: A portion of a partially explored map. Cells marked white are known to be free while cells marked black are known to occupied while cells marked grey are unknown. The cells marked yellow are known to be free and have neighbouring unknown cells (i.e. they are frontier cells).

2.2 Frontier Detection

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

queue_m ← ∅
ENQUEUE(queue_m, pose)
mark pose as "Map-Open-List"

while queue_m is not empty do
    p ← DEQUEUE(queue_m)

    if p is marked as "Map-Close-List" then
        continue
    if p is a frontier point then
        queue_f ← ∅
        NewFrontier ← ∅
        ENQUEUE(queue_f, p)
        mark p as "Frontier-Open-List"

        while queue_f is not empty do
            q ← DEQUEUE(queue_f)
            if q is marked as {"Map-Close-List","Frontier-Close-
            List"} then
                continue
            if q is a frontier point then
                add q to NewFrontier
                for all w ∈ adj(q) do
                    if w not marked as {"Frontier-Open-
                    List","Frontier-Close-List", "Map-Close-List"}
                    then
                        ENQUEUE(queue_f,w)
                        mark w as "Frontier-Open-List"
            mark q as "Frontier-Close-List"
        save data of NewFrontier
        mark all points of NewFrontier as "Map-Close-List"
    for all v ∈ adj(p) do
        if v not marked as {"Map-Open-List","Map-Close-List"}
        and v has at least one "Map-Open-Space" neighbor then
            ENQUEUE(queue_m,v)
            mark v as "Map-Open-List"
    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
sorted ← SORT_POLAR(lr, pose)
    // get the contour from laser readings
prev ← POP(sorted)
contour ← ∅
for all Point curr ∈ sorted do
    line ← GET_LINE(prev, curr)
    for all Point p ∈ line do
        contour ← contour ∪ {p}
    // extract new frontiers from contour
NewFrontiers ← ∅ // list of new extracted frontiers
prev ← POP(contour)
if prev is a frontier cell then // special case
    create a new frontier in NewFrontiers
for all Point curr ∈ contour do
    if curr is not a frontier cell then
        prev ← curr
    else if curr has been visited before then
        prev ← curr
    else if curr and prev are frontier cells then
        add curr to last created frontier
        prev ← curr
    else
        create a new frontier in NewFrontiers
        add curr to last created frontier
        prev ← curr
    // maintainance of previously detected frontiers
for all Point p ∈ ActiveArea do
    if p is a frontier cell then
        // split the current frontier into two partial frontiers
        get the frontier f ∈ frontiersDB which enables p ∈ f
        f_1 ← f[1 ... p]
        f_2 ← f[(p + 1) ... |f|]
        remove f from frontiersDB
        add f1 and f2 to frontiersDB
for all Frontier f ∈ NewFrontiers do
    if f overlaps with an existing frontier existFrontier then
        merged ← f ∪ existFrontier
        remove existFrontier from frontiersDB
        add merged to frontiersDB
    else
        create a new index and add f to frontiersDB
```

(b) Fast Frontier Detection Pseudocode

Figure 4: Pseudocode for the two frontier detection algorithms discussed.

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

Heuristics can be used to choose the next frontier to travel to. Two main heuristics exist: the largest frontiers and the closest frontier. 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. The other heuristic is the distance. In this case the frontier with the smallest Euclidean distance to the robot will be chosen. This

approach puts less emphasis on the information gained and aims to improve the time taken to explore the whole map. This is due to the fact that where the size heuristic approach may result in the robot going back and forth between two large frontiers, the distance heuristic approach is likely to result in the robot covering all frontiers as it travels.

2.3.2 Information Based Approach

This method aims to compute the uncertainty in the map and choses 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.

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

2.4 Reference Exploration Algorithm

The algorithm implemented out of the box 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 closes 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 a explorer algorithm. it is clearly visible 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 within walls or outside of its confined range.

2.5 Improving the Exploration Algorithm

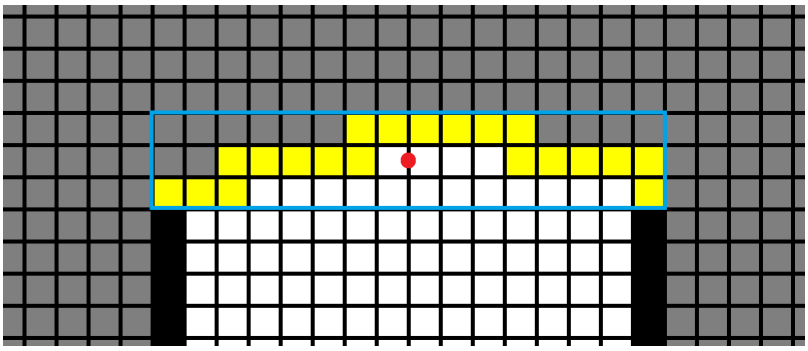


Figure 5: Visualization of the algorithm that is to compute the middle cell of a frontier. The blue rectangle identifies the extremes of the frontiers and is characterized by the minimum and maximum x and y coordinates of the frontier. The red dot is the middle of the rectangle and the middle cell is 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 has a store of the current frontiers 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 2middle" 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.

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.

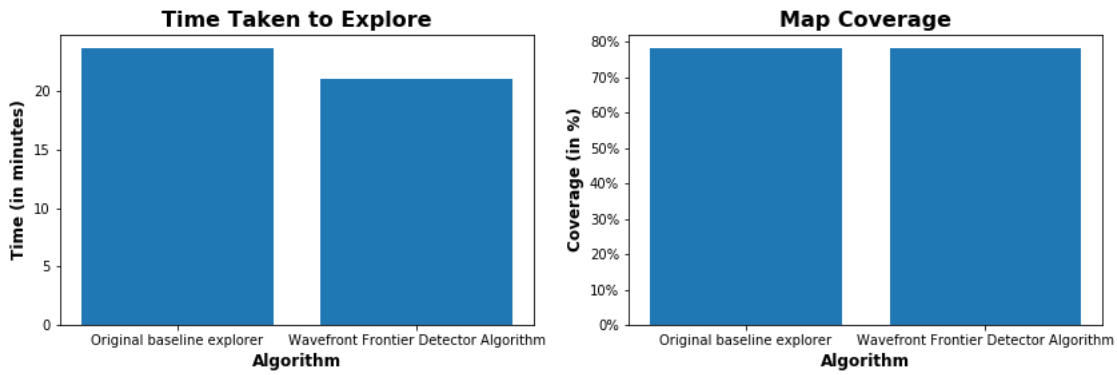


Figure 6: Graphs showing how the exploration time and coverage differ for the reference(baseline) algorithm and our implementation of the explorer algorithm utilizing WFD.

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. Whereas previously, the planner could decide if a destination is reachable, 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.

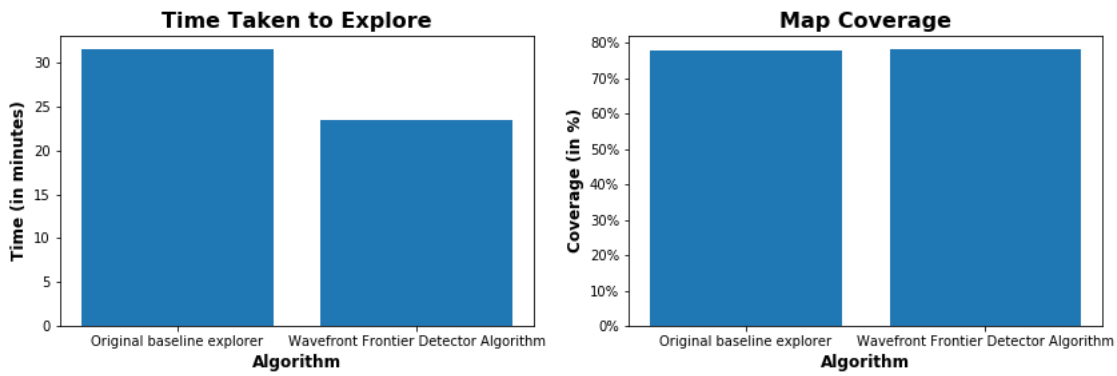


Figure 7: Graphs showing how the exploration time and coverage differ for the reference(baseline) algorithm and our implementation of the explorer algorithm utilizing WFD in the case when the true search grid is not known to the planner.

4 Information-Theoretic Path Planning

4.1 Description of Entropy and Mutual Entropy

The information-Theoretic approach to path planning utilizes the concept of entropy. Entropy represents the uncertainty in the map. 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:

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

To compute the entropy of the whole map the expression in Eq. 4 needs to be evaluated. in this equation, **M** represents possible map realizations (i.e. specific assignments of occupied/free to cells). 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 **C** can take a specific map realization **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 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 know, 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 out case is directly proportional to the number of unknown cells.

4.2 Evolution of Entropy over Time

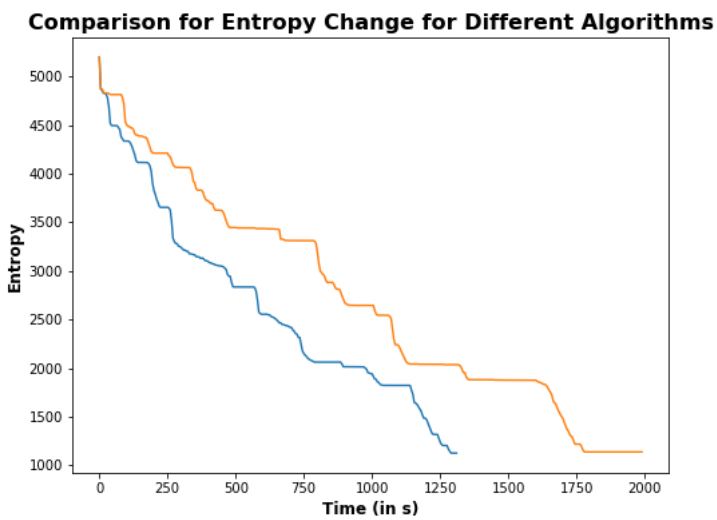


Figure 8: asd

References

Appendices