ME/CDS/EE 234b Advanced robotics **June, 2023**

Final Project

Name Kevin Gauld and Neil Janwani **Late Submission:** No

1 Introduction: Exploration

Much of classical robotics is framed as a start-goal problem. These problems are aimed at guiding a robot in a partially or fully known environment to a pre-determined start *and* goal–ultimately becoming a graph-search problem. However, these problems do not encompass all the scenarios encountered by robots. For instance, robots designed for search and rescue may need to plan in unknown environments where random variables dictate the reliability of robot sensors. Thus, arguably an equally important problem in robotics is *exploration*. In other words, given an unknown map and unreliable sensor readings, can a robot follow a somewhat efficient trajectory to build the correct map?

1.1 Lab 3

Notably, this project is a follow-up to the pre-lab completed during class. While, indeed, our exploration algorithm was able to explore the given `12x12` CAST arena, we chose to test the exploration algorithm in much larger environments. A consequence of this is that simulating MPC trajectories for an exploration scheme of that size is computationally expensive. Thus, we opted to implement our algorithm using `matplotlib` rather than in ROS (`rviz`). However, note that our algorithm has built-in measures to deal with rounding and map Cartesian offsets. This makes our system viable as a high-level node to be added to the current lab 3 ROS stack.

2 Methods

2.1 Problem Formulation

We assume that our robot, R, is working in a non-dynamic (i.e. obstacles do not move/appear/disappear) occupancy grid, G, and has starting position x_0 which is in freespace. Furthermore, we allow our sensor readings to be unreliable; they are corrupted with a probability of $(1 - p_{true})$; we assume $p_{true} = 0.9$ in our simulations. In other words, with probability $(1-p_{true})$ if the correct sensor reading for a particular cell $(i, j) \in G$ is *not* freespace R's sensor will detect that (i, j) is actually freespace. Note that $c_{i,j}$ and (i, j) are used interchangeably and both refer to a specific cell with indices (i, j). Furthermore, we allow each cell $c_{i,j} \in G$ to have probability $p_{i,j}$ of being *not* freespace (an obstacle).

Allow G_T to be the true occupancy grid. Then R's task is to explore G and update probabilies $p_{i,j}$ for each cell $c_{i,j} \in G$ such that G most closely matches G_T. We initialize G so that each cell $c_{i,j}$ has $p_{i,j} = 0.5$. Consequently, since R needs to be able to traverse its way through the map, we need to be able to mark certain cells as freespace (traversable) when $p_{i,j}$ is below a certain number. For this, we defined $\alpha_{FREE} = 0.05$. Thus, we allow $c_{i,j}$ to be treated as freespace when

$$c_{i,j} < \alpha_{FREE}$$

and to be treated as an obstacle/unknown when

$$c_{i,j} \geq \alpha_{FREE}$$

Furthermore, we need to define our stopping criterion such that G matches G_T with confidence α_{FREE}. Thus, we amend our bounds slightly to allow an obstacle to be certified when $p_{i,j} > 1 - \alpha_{FREE}$. Thus, our stopping criterion formally becomes

$$p_{i,j} \in (1 - \alpha_{FREE}, 1] \cup [0, \alpha_{FREE}) \;\; \forall \; (i,j) \in G$$

which can be interpreted as "when all cells in G become identified as freespace or as obstacles with confidence α_{FREE}".

Now that we have determined our problem inputs and when our problem is solved, we need to define how measurements are corrupted and taken. Note, we mentioned earlier that a measurement $o^k_{i,j}$ for cell $(i,j) \in G$ taken at timestep k is corrupted with probability $(1 - p_{true})$. Furthermore, R takes sensor updates of the given form

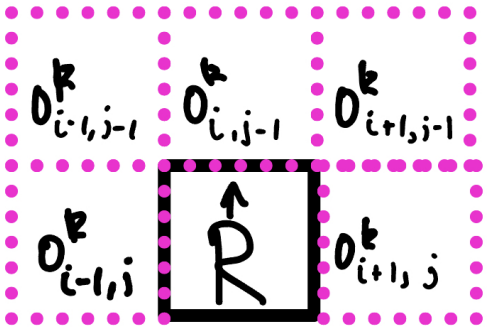

Figure 1: Robot sensing approach. The robot is able to see one cell in front and one cell to the side of its center as indicated by the image. Note that each of these sensor readings is corrupted with probability $(1 - p_{true})$ (not the entire scan). We chose this sensing model to best simulate a poor, unreliable sensor.

2.2 Exploration update

Allow $(i,j) \in G$ to represent an arbitrary cell. We need to be able to update any $p_{i,j}$ with sensor reading $o^k_{i,j}$ for timestep k. Notably, depending on how much we trust our sensor reading $o^k_{i,j}$, we want $p_{i,j}$ to be updated in different ways (strongly for high p_{true} and vice versa otherwise). Note that this formulation was derived in lecture 7 on exploration. First, let's define the following

$$p^o_{i,j} \coloneqq p_{true} p_{i,j} + (1 - p_{true})(1 - p_{i,j})$$

$$p^n_{i,j} \coloneqq p_{true}(1 - p_{i,j}) + (1 - p_{true}) p_{i,j}$$

where the correct occupancy (i.e from the truth cell $(i,j) \in G_T$) is assessed by the above model. Notably, see that $p^o_{i,j}$ can be viewed as the expectation over p_{true} of whether (i,j) is an occupied cell. The reverse is true for $p^n_{i,j}$ (an *unoccupied* cell). Using these terms, we can now define a generalized Bayesian measurement update:

$$p(c^{k+1}_{i,j} | o^{k+1}_{i,j}) = \frac{p(o^{k+1}_{i,j} | c^{k+1}_{i,j}) p(c^{k+1}_{i,j} | o^k_{i,j})}{p(o^{k+1}_{i,j} | o^k_{i,j})}$$

While this notation seems difficult to parse at first, note that $p(o^{k+1}_{i,j} | c^{k+1}_{i,j})$ is simply p_{true} and $p(c^{k+1}_{i,j} | o^k_{i,j})$ is simply $p_{i,j}$ (as it is the cell probability given the last sensor reading). Finally, see that $p(o^{k+1}_{i,j} | o^k_{i,j}) =$

$p^o_{i,j}$ if the sensor reading from timestep $k+1$ is occupied and $p^n_{i,j}$ otherwise since

$$p(o^{k+1}_{i,j}|o^k_{i,j}) = \sum p(o^{k+1}_{i,j}|x^{k+1}_{i,j})p(x^{k+1}_{i,j}|o^k_{i,j})$$

which, as mentioned earlier, can be seen as the expectation over p_{true} given the last sensor reading. Thus, we may finally write our occupied update as

$$p'_{i,j} = \frac{p_{true}p_{i,j}}{p^o_{i,j}}$$

Our unoccupied update differed slightly from slides. We were required to subtract the update included on the slides from 1 as shown below

$$p'_{i,j} = 1 - \frac{p_{true}(1-p_{i,j})}{p^n_{i,j}}$$

We noted that this made sense as the probability that a cell is freespace is the complement of the probability that it is an obstacle. To confirm our suspicion, we also plotted these updates by initial cell probability $p_{i,j}$:

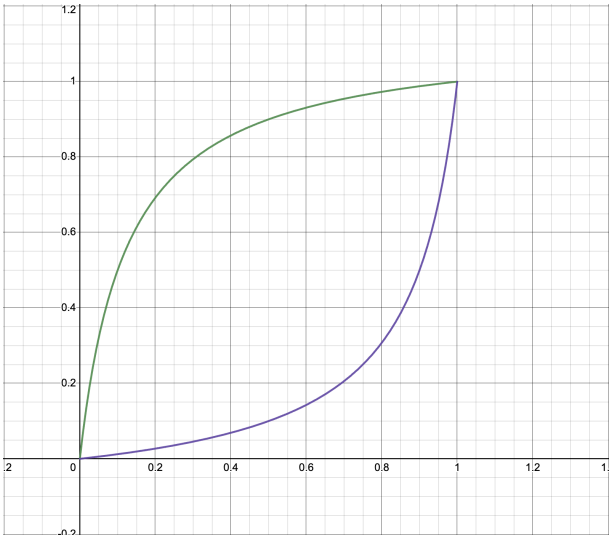

Figure 2: Update rules plotted by new cell probability (y-axis) against initial cell probability $p_{i,j}$ (x-axis). The green line represents an occupied update and the purple line represents an unoccupied update

See that both curves are mirrored over the line $y = x$, which is expected. Thus, we may use this update rule to alter probabilities given new sensor readings. Furthermore, recall that each sensor reading consists of reading five total cells (in a U-shape around the robot). Thus, this update must be done *for each cell* such that we utilize maximum information from each sensor reading.

2.3 Expected Information Gain

In order to write an efficient planner, it is required that one must derive a useful cost. In this case, we propose (as in the slides) to use EIG (expected information gain) which uses log-probabilities to determine the gain in *entropy* (or information) a given update will produce. The equation for this is given in the slides, as is shown below for convenience

$$H^o_p(c_{i,j}) = -p^o_{i,j}\log(p^o_{i,j}), \quad H^n_p(c_{i,j}) = -p^n_{i,j}\log(p^n_{i,j})$$

$$EIG(c_{i,j}) = H_p(c_{i,j}) - \mathbb{E}[H_{p'}(c_{i,j})]$$

This EIG can be calculated across possible actions $\{u_1, u_2, \ldots, u_n\}$, from which the highest EIG can be chosen as a goal action for a standard path planner. This will be elaborated and expanded on in the following algorithm sections. Furthermore, note that as for the cell probability updates this EIG metric should be calculated for every cell in a given robot sensor reading from a certain robot pose.

2.4 EIG Exploration Algorithm

We can now define our fundamental exploration algorithm. Note that we represent a very high-level overview here and exact code is included at the end.

```
def EIG_exploration_algorithm(grid, start):
   path ← []
   current_pose ← start
   while True:
      poses ← list of all freespace poses sorted by EIG
      while path is empty
         path ← Dijkstra path planner (current pose to highest EIG pose)
         if path is not [] # a path was planned
            break
         else
            poses = poses[1:] # discard the first pose as we could not find a path to it
            continue # need to generate another path using a different goal
      current_pose = path[0]
      path = path[1:]
      if sensor area has any cells not marked with freespace or obstacle
         path = []
         corrupted_measurement() # (take a corrupted measurement)
```

Notice that while we are in an infinite loop, R will stop moving when all cells are known and we stop taking measurements. In this case, we will never generate a new path taking R to another location in the graph (please see video for this behavior). Furthermore, notice that when the area in front of the robot (the sensor reading) is completely known, the robot does not need to replan (as the current plan has already taken into account the current map). Thus, the path is simply iterated. Otherwise, a replanning step occurs as `plan` is set to the empty list.

2.5 Smart Exploration Algorithm

In the smart exploration algorithm, we provide an incentive for R to explore areas of information gain first even if they are not positions of globally maximum EIG. Thus, our cost metric factors EIG by inverse distance as so

$$\text{smart\_cost}(x) = EIG(c_{i,j}) \cdot \frac{1}{||c_{i,j} - c_c||_2}$$

Where c_c indicates R's current position. Our algorithm follows `EIG_exploration` closely; we only need to change the cost metric as according to `smart_cost` above:

```
def EIG_exploration_algorithm(grid, start):
   path ← []
   current_pose ← start
   while True:
      poses ← list of all freespace poses sorted by smart_cost
      while path is empty
         path ← Dijkstra path planner (current pose to highest EIG pose)
         if path is not [] # a path was planned
            break
         else
            poses = poses[1:] # discard the first pose as we could not find a path to it
            continue # need to generate another path using a different goal
```

```
13       current_pose = path[0]
14       path = path[1:]
15       if sensor area has any cells not marked with freespace or obstacle
16          path = []
17          corrupted_measurement() # (take a corrupted measurement)
```

2.6 Environments

For the map environment, we use Perlin noise generation, similar to that used in Kevin & Prashanth's planning project from 234a. Perlin noise is a gradient noise, creating a smooth texture value on a 2D grid region. Since this texture is smooth, "cross sections" resulting from thresholding the value of the noise on the grid will result in obstacles which are contiguous, substantially sized, and have smooth, continuous edges. We create our obstacles by thresholding at the 90^{th} percentile of the generated noise. The higher the percentile, the more obstacles present in the final map. Below is the pseudocode for environment generation. Environments were generated at various square sizes, primarily 20x20, 30x30, 50x50, and 100x100. We also used the basic map given in the last lab as a standard testing map.

```
1 def generate_map(M, N, risk_t = 0.9):
2    noise ← Perlin Noise Generator
3    risk ← N x M array
4    For each point in risk:
5       risk[point] ← noise[point]
6    risk ← normalize(risk)
7    risk ← round(risk-risk_t) # Threshold the risk value
8    Return risk
```

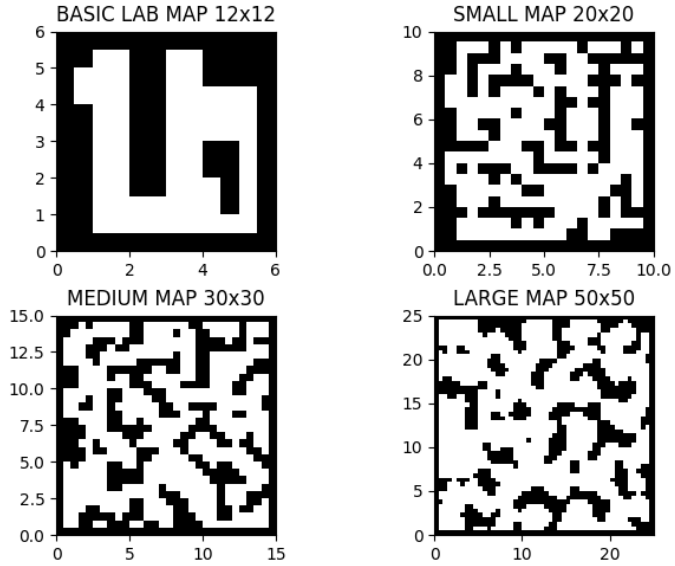

Figure 3: Plots of the four main maps used as environments for testing. Each image is labeled with the pixel grid size, and is plotted with a cell width/height of 0.5

2.7 Validation

Finally, we need a metric to calculate the percent at which our map has been explored. For this, we propose taking the mean over the following series for all $(i, j) \in G$

$$\{\bar{p}_{i,j}\} : \text{where } \bar{p}_{i,j} \text{ is 0 if it is less that } \alpha_{FREE} \text{ and 1 if it is greater than } 1 - \alpha_{FREE}$$

3 Results and Analysis

See that our EIG exploration algorithm is able to fully explore the given lab space from prelab 3 as shown below in Figure 4. Thus, it is our hope that on real hardware this exploration algorithm would similarly perform

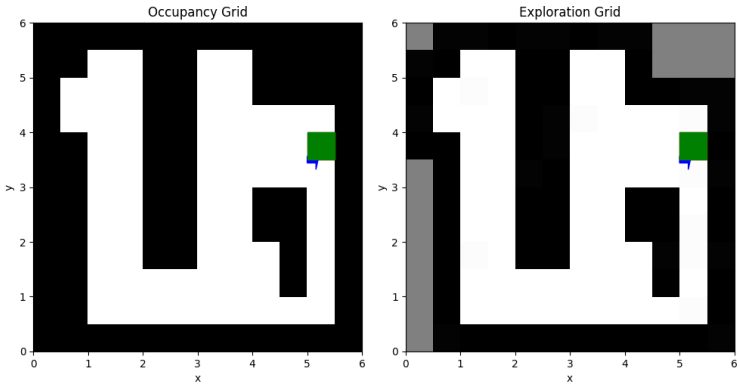

Figure 4: EIG exploration planner on given lab map (12x12)

See that in certain cases, however, like in Figure 5, where the globally maximal EIG position is far away from the current position, the robot can take inefficient routes (as opposed to "finishing up" the exploration where it currently is). This lengthens the exploration time significantly, and while visiting the maximal EIG at each iteration may initially allow for fast exploration, this property can quickly drop off as the map becomes more explored (and high information gain spaces begin to disappear).

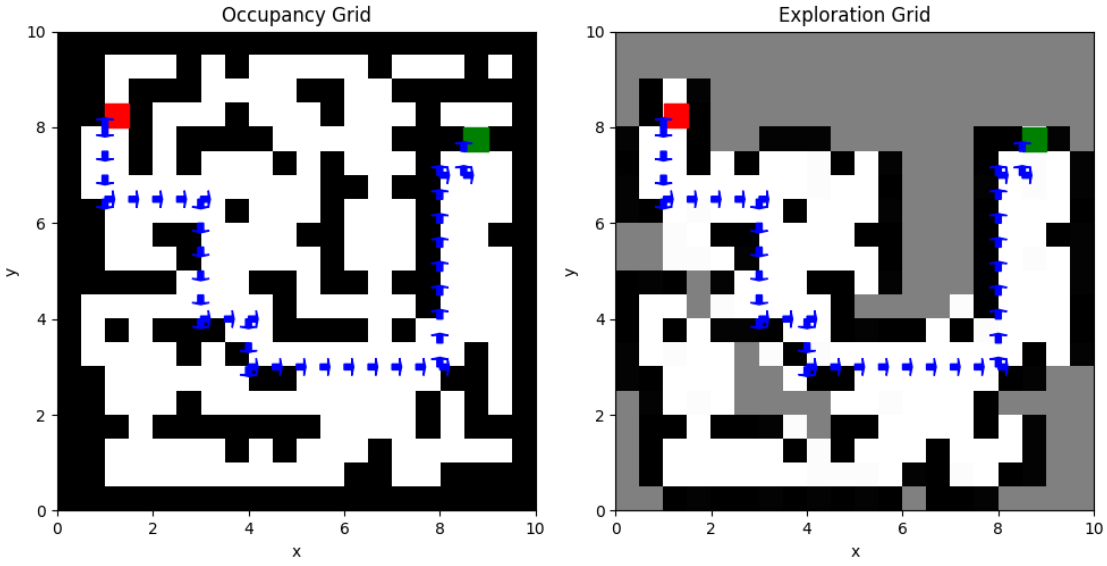

Figure 5: EIG exploration planner on map 20x20

Thus, we introduce the smart exploration planner, which incentivized R to explore nodes closest to it if they provided a sufficient information gain. See below in Figure 6 that our algorithm is able to prioritize "finishing up" exploring before moving on to higher information gain areas.

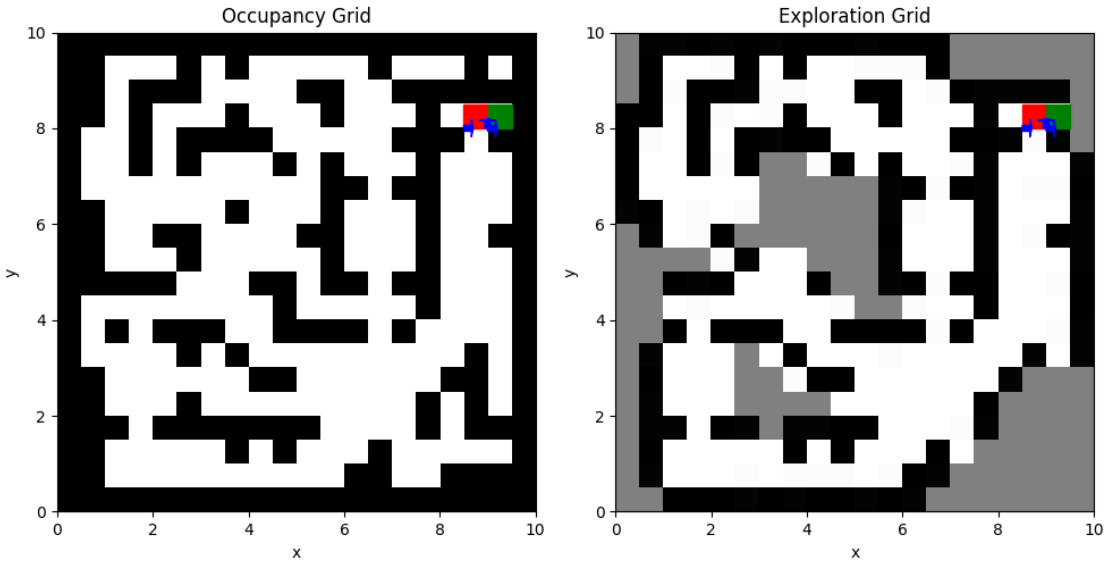

Figure 6: Smart exploration planner on map 20x20

Thus, overall, we see that the EIG explore is successful in increasing the information within the system, but is significantly less efficient than the smart exploration. This is shown in both Figures 8 and 9, where the smart explore is able to pull away from the EIG explore, more quickly reaching its maximal exploration percentage.

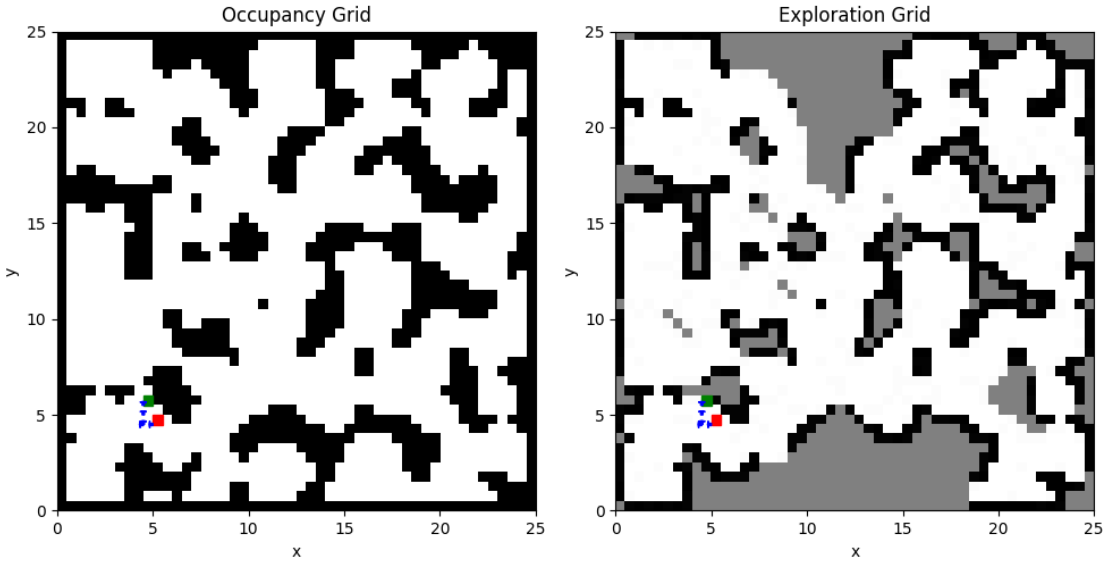

Figure 7: Smart exploration planner on map 50x50. The map is eventually fully explored (please see video)

We can see the efficiency of the smart exploration algorithm on a `50x50` grid. Such a map would have taken an impractical amount of time (as discovered through testing) to fully explore as seen in Figure 7.

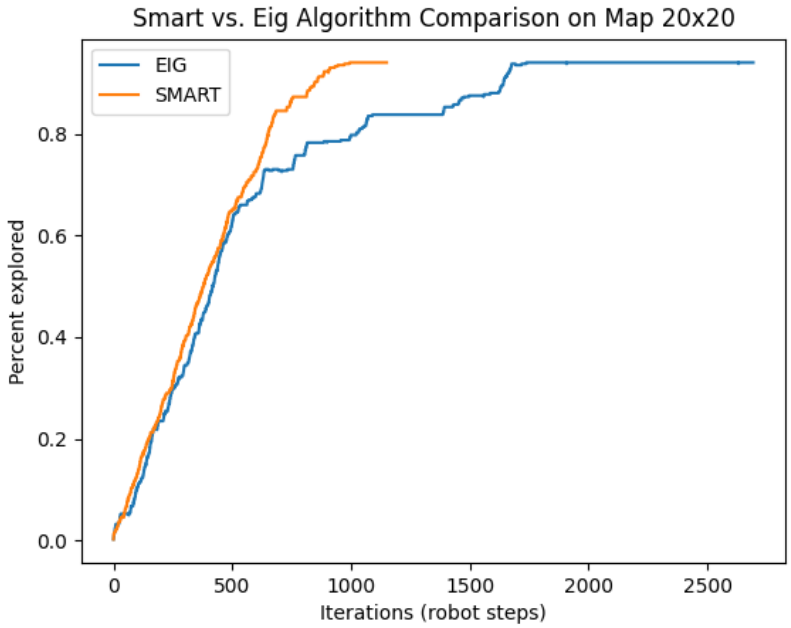

Figure 8: Comparison between the smart and EIG algorithms on a single map. As shown, the two algorithms are comparable for the beginning, but smart explore pulls ahead at the end when EIG travels long distances towards maximum information gain.

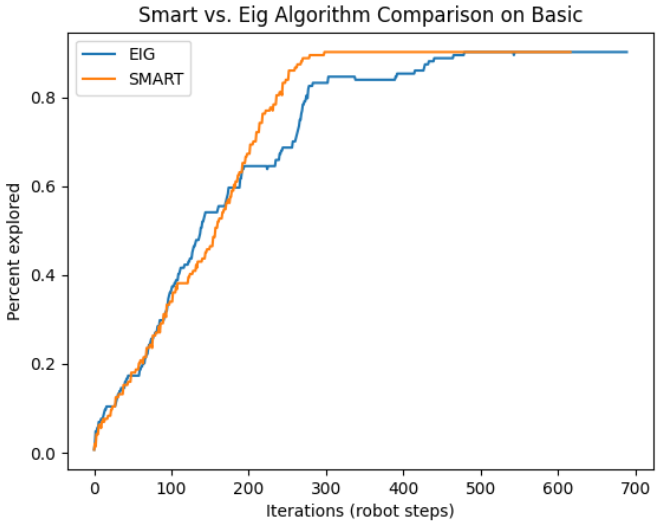

Figure 9: Comparison between smart and EIG algorithms on the basic map. As seen in figure 8, the two algorithms behave similarly then diverge, with the smart explore outperforming EIG in terms of exploration speed after many iterations.

We also see that, while smart explore performs well, it scales poorly with size. Based on Figure 9, we clearly see polynomial gain in the number of iterations taken to explore the full map space. However, we do not have perfect polynomial scaling, as the corruption of measurements causes extra computation

which happen more times the larger the grid size.

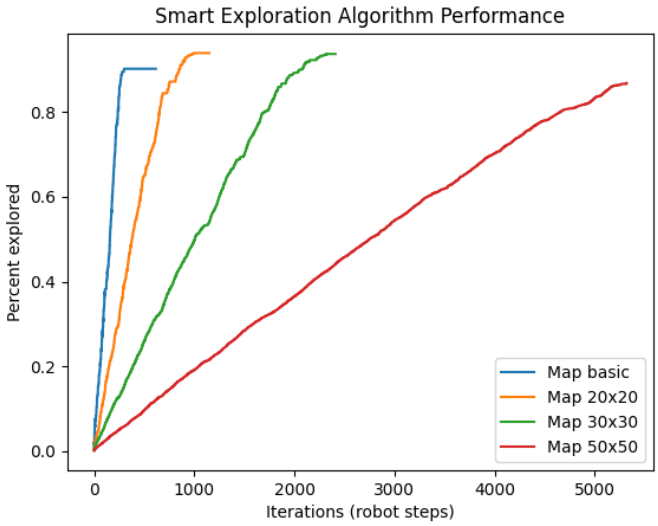

Figure 10: Performance of the smart exploration time for four different map sizes. As shown, the time taken to maximize exploration grows quadratically with map size.

One of the most important cases solved by the smart explorer, and the trait that makes this algorithm more viable over EIG, is its resistance to planning far across the grid. For each iteration in the algorithm, the robot is able to take one step. For EIG, there is no disincentive for the robot to spend those iterations in a long movement across the entire map to the next point. This wastes a significant amount of iteration time, as this movement from known point to known point does not uncover as many unexplored nodes as a movement into a primarily unexplored area. As such, the smart explorer's prioritization of nearby unexplored nodes, even if they aren't globally maximized for EIG, allows the planner to choose much more time/iteration optimal paths.

4 Next steps

There are various ways in which this project can be expanded on in the future. One such method is by expanding the scope from exploratory motion to goal-based motion with a directed explore implementation. A directed explore would deprioritize the EIG, prioritizing movement towards a goal within unknown space. This would allow for navigation between waypoints in an unknown map, which is important for handling increasingly large maps.

We could also implement a reachability criterion. As it stands, there is no way for the robot to check if the inner areas within explored boundaries are explored or unexplored. In the future, these points could be found, then a path could be planned from the inner point to the robot, with unknown areas treated as freespace. If no path is found, then it is known that there is a blockage in every possible path to the inner point, so the robot can never explore the point, meaning the point is not freespace. This can also be implemented by checking if the map has been fully explored, in which case all points which are uncertain are known to not be freespace.

Also, we may implement a measurement cost, which penalizes the act of taking a measurement and adds it as a movement step in the planning. As it stands, there is currently no penalty for remaining stationary in a single location and repeatedly taking measurments to drop uncertainty. Also, measure-

ments are taken at every movement automatically. Making the act of taking measurements deliberate would be an interesting addition to the project, making the planner significantly more complex.

This has significant potential to be implemented in hardware. As mentioned prior, the stack is built with the same architecture as in the mapping lab, allowing us to feasibly be able to port this code onto GVR-Bot. This can also be introduced to other hardware related systems, as the architectures behind the smart and EIG explore are generalizable to a wide variety of robotic motion planning systems.

Appendix

VIDEO LINK

CODE LINK