Authors:
Kaloyan Rusev, Tania Turdean, Darius Filip

COMP0037 2021/22 Coursework 1

Question 1

a. This problem can be described as a bandit problem with 4 arms. There are 4 possible actions it can take: A = {1,2,.,k}, k=4. These actions represent each of the 4 charging stations. The charging rate represents the reward, which is drawn from a Gaussian distribution. The goal of the robot is to pick the best charging location, so to maximise the expected reward by measuring the charging rate each time (pulling handles) and averaging the observations.

Epistemic uncertainty is the uncertainty of the model. In this case, too few pulls, or not charging enough times, means the robot won't be able to learn the best mean of the charging stations.

Aleatoric uncertainty is the uncertainty of the measurement which in our case might be caused, for example, by humidity or the static electricity of the robot - the same charging rate might differ because of these external conditions.

b. A bandit environment with 4 bandits and the specified means and standard deviations was created. The function *run_bandits* was defined: it is looping through the bandits and number of steps (2000) and pulling the bandit arm each time to create an array of the observed rewards of the charging stations. At the end it is printing the mean of the rewards for each bandit. At first, smaller values were tried for the number of steps, but 2000 steps were used for the means to converge.

The code was modified to store the rewards for the charging stations in a dataset. Their distribution is displayed in the violin plot below.

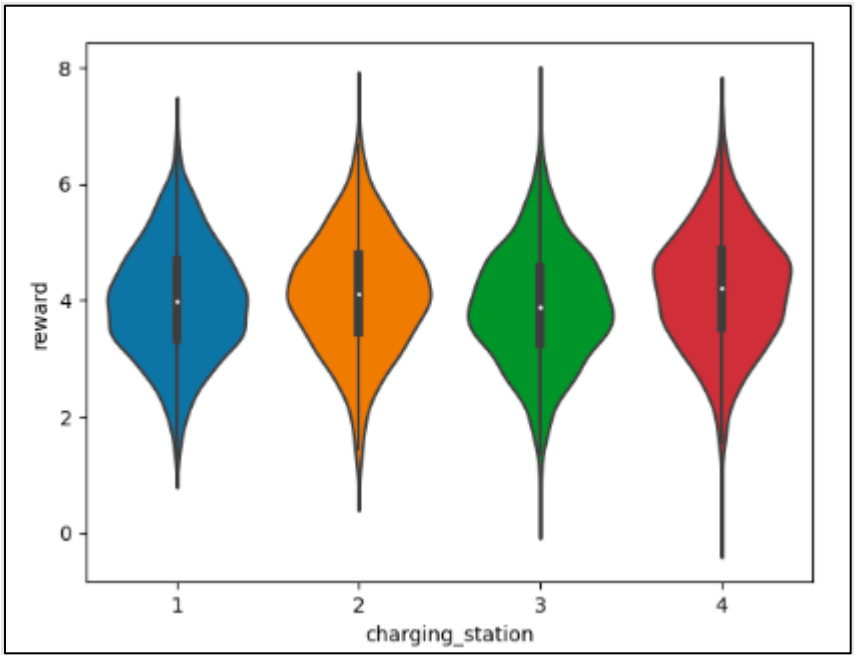

Figure 1 – Violin Plots of the Reward for each Charging Station

See *1b.py*

c. Regret measures the difference between the expected reward the agent would get for the optimal action, and the reward the agent got using the actions it took. Usually the regret is non-negative, but it could become negative on an individual run. The formula used to calculate regret is presented below (equation 1).

$$\rho = TE[R_t(a \cdot)] - \sum_{t=1}^{T} E_\pi[R_t(A_t)] \qquad (1)$$

Regret is used because it is a better evaluation method than calculating the percentage of correct actions. For example, in this case, the correct action is station 4 as it has the biggest reward.

We can use regret in this problem because the reward distributions (which are the mean charging rates) are known. However, for problems with unknown reward distributions, we would be unable to calculate the regret at each "t" since we would only know the optimal reward after trying all the arms.

d. The e-greedy strategy is used for addressing epistemic uncertainty by mixing exploration and exploitation. The **e**-greedy strategy makes the agent pick the best action, ($\text{argmax}_a\ Q_t(a)$) with probability 1-**e** and a random action with probability **e**. We initialize Qt(A) (the estimate of reward) using “try them all once”, then we dynamically update it using the following formula:

$$Qt(a) = \frac{sum\ of rewards\ when\ a\ taken\ prior\ to\ t}{number\ of\ times\ a\ taken\ prior\ to\ t} \qquad (2)$$

Long-term probability of visiting the optimal station:

$1 + (1/C - 1)\ \epsilon$, where C is the number of stations

$1 + (1/C - 1)\ \epsilon = (1 - \epsilon\ ) + (\frac{1}{C}\epsilon\ )$ (3)

Let us say $b \in \{1,2,3,\ldots,C\}$ and b is the optimal action. The long-term probability to visit b:

$P(b) = \epsilon\ \frac{1}{C} + (1 - \epsilon\ )\ [\ P\ (b = \arg\max_a Q_t(a))]$ (4)

*Visiting a random action with probability ϵ and for each action 1/C
Visiting the best action with probability $1 - \epsilon$ and the probability that b is the optimal action up until t

To prove (3) and (4) are equal we need to prove that:

$P\ (b = \text{argmax}_a Q_t(a)) = 1$ (5)

When t is high enough (long-term), after a lot of trials, the estimated values tend to converge to the real mean rewards as epistemic uncertainty is reduced, so the equation above is correct.

$0 < P\ (b = \text{argmax}_a Q_t(a)) < 1$ for any t (7)

For very short runs (small t), P (b = argmax $_a$ Q $_t$ (a)) will be much lower. Therefore, the probability of correctly pulled handles, P(b), will be lower, as it follows the equation:

ϵ + (1 - ϵ) [P (b = arg max $_a$ Q $_t$ (a))] (8)_

and we know that [P (b = argmax $_a$ Q $_t$ (a))] is lower for lower t.

For very short runs, the epistemic uncertainty was no reduced yet, especially in the case of hard to separate rewards.

e. A "try them all agent" was created in *try_them_all_agent.py.* Using this agent, the *monte_carlo.py* was created with 100 number of tries and 10000 number of steps. More values for the hyperparameters were tried until these were found to converge to the correct action (bandit 4 with mean 4.2).

The *epsilon_greedy_agent.py* was created to implement the epsilon-greedy approach. In *epsilon_greedy_equation.py*, 4 values of epsilon were tested: [0.01, 0.05,0.5, 1.0], for 5000 steps. For each value of epsilon, the equation in d) was calculated, where C=4 and epsilon are the current value of epsilon. Also, we created the plot of percentage of correct actions against the number of trials for each epsilon. For every epsilon value, for larger values of trials, the percentage of correct actions (or the probability of the correct action to be visited) converges to the equation value. For smaller values, the probability is much lower. Therefore, it appears that the equation is correct.

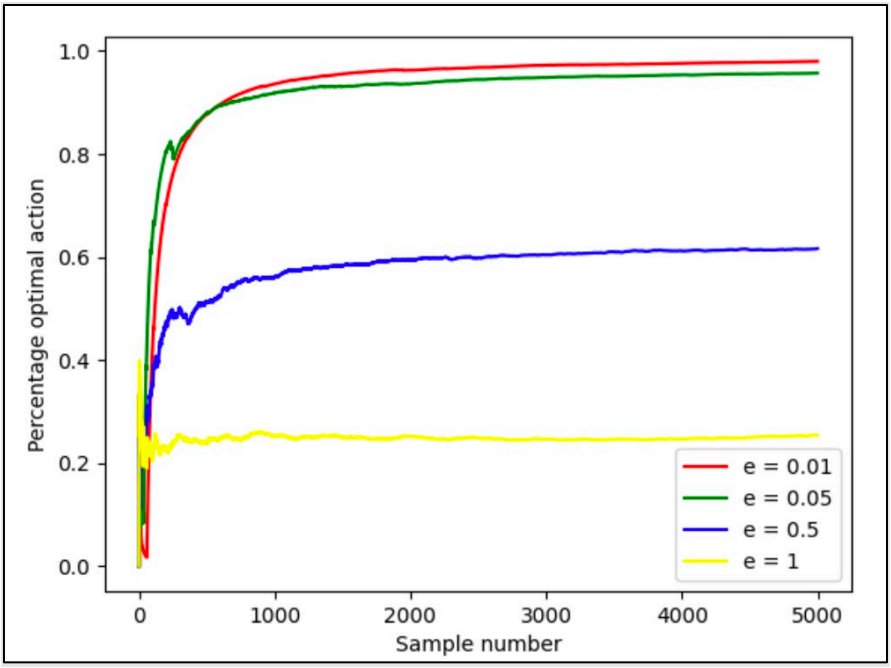

Figure 2 – Percentage Optimal Actions for Different Epsilon Values For Epsilon-Greedy

Printed the equations for epsilon values:

Epsilon = 0.01 Equation=0.9925 Epsilon = 0.5 Equation=0.625

Epsilon = 0.05 Equation=0.9625 Epsilon = 1 Equation=0.25

f. One of the commonly available methods to improve the efficiency of the ϵ - greedy algorithm is tempering the schedule for exploration (i.e.: damp or change the exploration over time). Tempering makes the exploration probability a non-increasing function of time. Mathematically, this can be described as follows:

$$A_t = \begin{cases} argmax_a Q_t(a) & with\ probability\ 1 - \epsilon_t \\ a\ random\ action & with\ probability\ \epsilon_t \end{cases}$$
$$\epsilon_t \leq \epsilon_t - 1$$

As a result, the more the focus shifts towards finding the optimal solution, the percentage of time used for exploration is decreased (the more is known, the less is explored). Tempering can be achieved using an exponential equation, such as equation x:

$$\epsilon_t = \epsilon \times e^{-0.05t} \qquad (9)$$

The *damped_epsilon_greedy_ageny.py* was created to implement the tempering the schedule for exploration approach. We tested it with the same values of epsilon as the ones in for 1e: [0.01, 0.05, 0.5, 1], for 5000 steps – see *test_dampering_greedy.py*.

For each value of epsilon, we created the plot of percentage of correct actions against the number of trials and the cumulative regret plot.

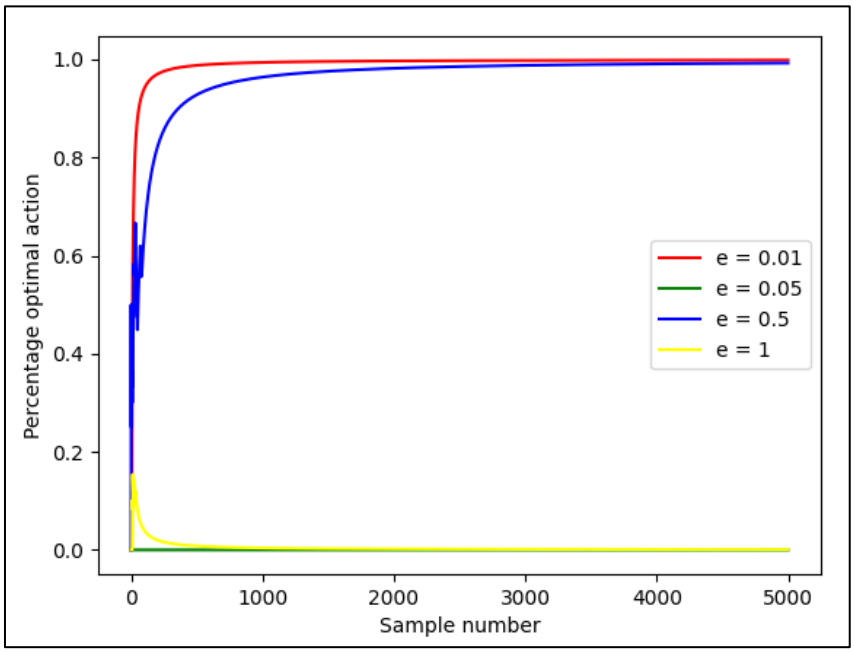

Figure 3 - Percentage Optimal Actions for Different Epsilon Values For Damped Epsilon-Greedy

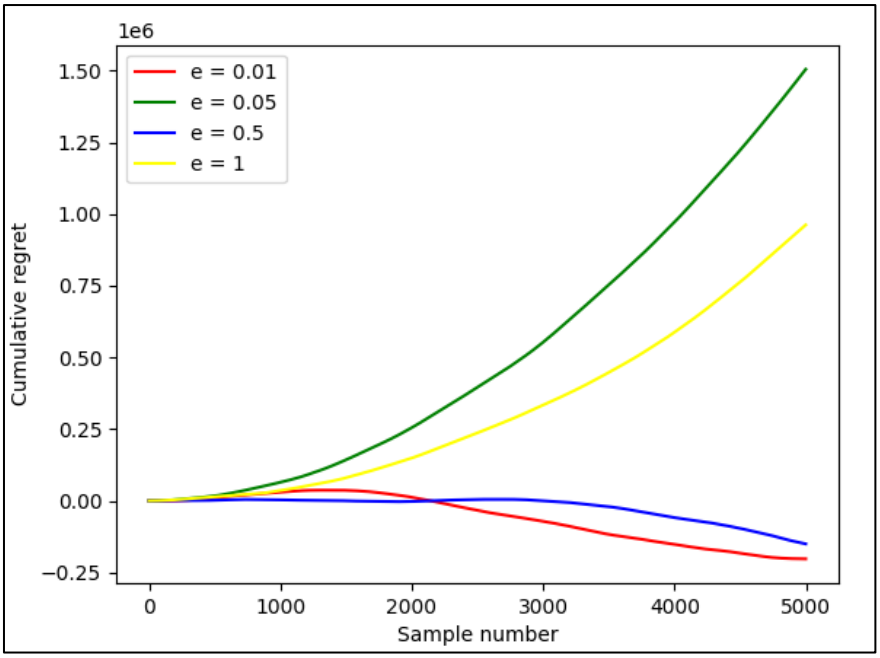

Figure 4 – Cummulative Regret for Different Values of Epsilon

As can be seen from the flattening of the curves for percentage of optimal actions, this method explores much less than the basic ϵ- greedy and either manages to find the optimal action in this period or not. In these graphs, the optimal action is found to be such only for ϵ equal to 0.01 and 0.5. This is because with the equation (9) that we use for decreasing ϵ, it decreases quite quickly, and respectively, the times the algorithm explores and not exploit, and as it is also not guaranteed to find the correct mean for each action, it finds other actions to be the most optimal. The cumulative regret also converges to 0 for these values. However, it becomes negative as the sample number gets too large.

g. Upper-Confidence-Bound (UCB) Action Selection

The main feature of this algorithm is that instead of selecting arbitrary actions chosen with a probability that remains constant in time, it balances exploration and exploitation as it learns more about the environment. In the early steps it focuses primarily on exploration, and actions that have been taken the least number of times are prioritised. Over time, it starts concentrating on exploitation instead, and actions with the highest estimated reward are preferred. Hence, the reason why this algorithm was proposed is the fact that it requires less memory in the long run, thus being less computationally expensive. The action taken at each time step is formulated in equation (10):

$$A_t = \underset{a\in[1,\dots,k]}{\arg\max}[Q_t(a) + C_t(a)]; \qquad (10) \qquad C_t(a) = c\sqrt{\frac{\log t}{N_t(a)}}; \qquad (11)$$

Where:

- $Q_t(a)$ is the estimated value for action ‘a’ at time ‘t’, responsible for exploiting;
- $C_t(a)$ is the upper bound confidence value, responsible for exploring;
- $N_t(a)$ is the number of times the actio has been selected before time ‘t’;
- ‘c’ is the value that controls the level of exploration.

h. UCB was implemented in *upper_confidence_agent.py.* In *test_ucb.py*, the percentage of optimal actions and the cumulative regret plots were created for 6 different choices of the degree of exploration: C: [0.1, 0.2, 0.5, 1, 2, 4]. For c=0.5, the percentage of optimal actions taken converges the quickest to 100%. For c=0.1 and c=0.2, the optimal action is not found even for an exceptionally large sample number. The cumulative regret is the worse for c=0.2. After the value of 0.5, as c increases, the performance gets worse. In conclusion, for smaller values of c, the optimal action is not found. The best-found value of c is 0.5. When c gets larger than this, the optimal action is found after more trials and the cumulative regret is much larger.

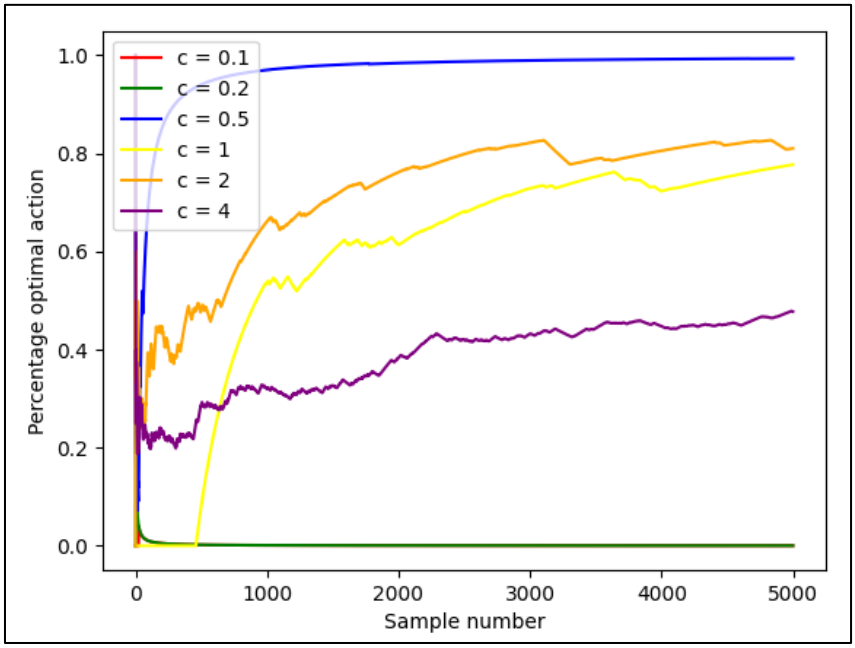

Figure 5 –UCB – Percentage of Optimal Actions for Different Values of C

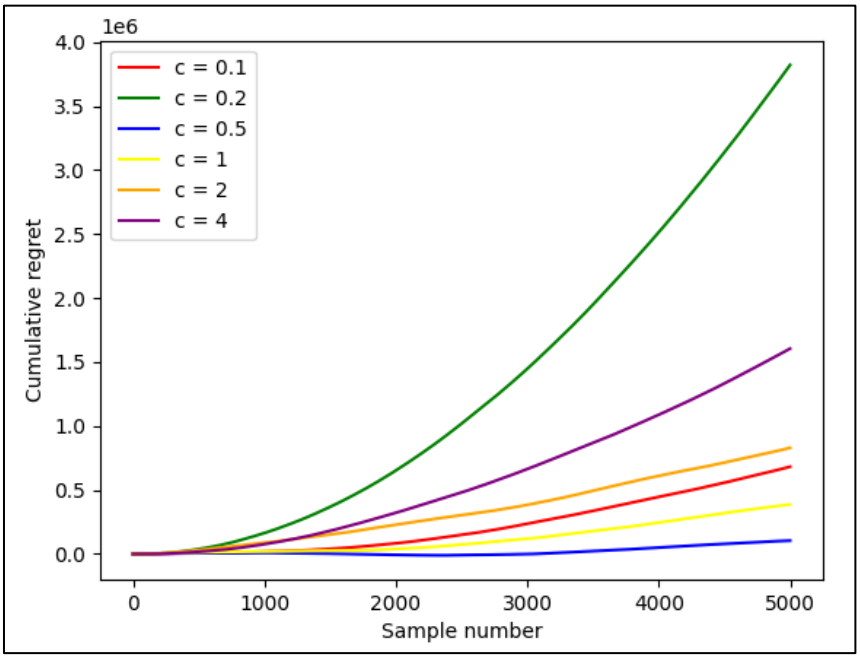

Figure 6 - UCB – Cummulative Regret for Different Values of C

The variance of each charging rates is 1. However, if we come back to the distributions of each action from b), we can see that the variance looks different for different actions. The white dot represents the mean Qt. The idea of UCB is to look at the probability of producing the best estimate - Ct(a). Therefore, we draw the black line to find the largest area after the upper bound. The probability of producing the best estimate is the same as the means Epsilon = 1, Equation=0.25. In this case, the upper bound confidence value is:

$$C_t(a) = c\sqrt{\frac{\log t}{Nt\,(a)}} \qquad (12)$$

As c value varies, the upper confidence bound will get higher or lower so we will look at different parts of the distributions with different variances.

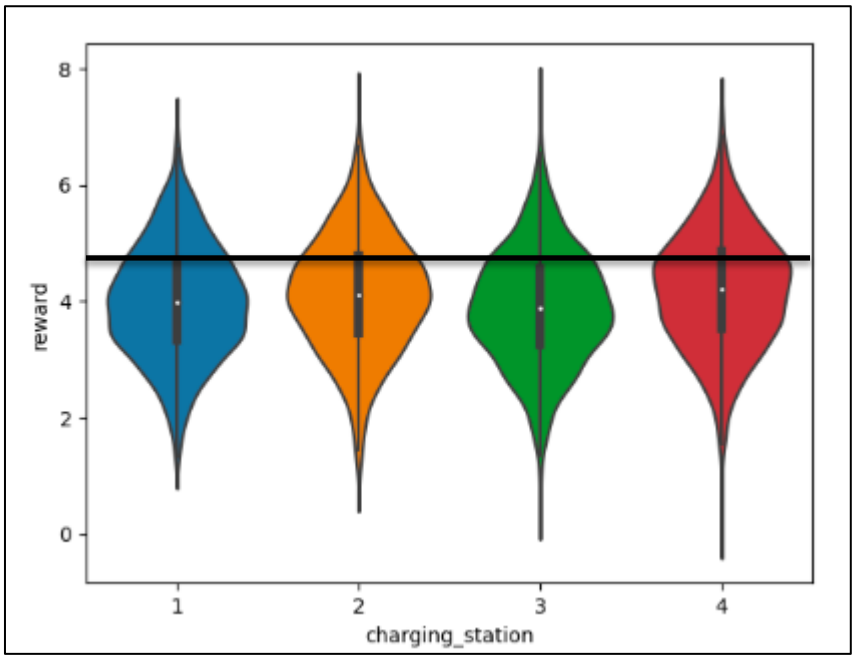

Figure 7 - Violin Plots of the Reward for each charging station

Question 2

a. Breadth First Search vs Depth First Search:

The Breadth-First Search algorithm visits and marks all the nodes in a graph from the starting cell to the goal cell, until it found the goal cell – this means it will always find a solution if it exists. It uses a breadth first queue, which means that the *Insert* function puts new elements at the tail of the queue and the *GetFirst* function takes the first element from the head of the queue. This is called first-in first-out.

The Depth-First Search algorithm visits and marks all the nodes in a graph from the starting cell to the goal cell (until it found the goal cell). It uses a depth first queue, which means that the *Insert*

function puts new elements at the tail of the queue and the *GetFirst* function takes the last element from the tail of the queue. This is called last-in first-out.

Advantages and Disadvantages:

The advantage of the breadth first search is that it is always optimal in a sense that it optimizes the solution with the minimum number of edges between the start node and the goal. However, this represents a disadvantage because it gives the shortest path when the cost is the same on all edges. Let us say that moving down has a lower cost than moving down-right. In this case, the breadth first search will not find the optimal path.

The advantage of the depth-first search is that it requires less memory to run, since only nodes on the current chosen path are stored (as opposed to breadth first, where all the trees generated thus far must be stored). As a result depth-first might be able to find a solution without having to visit many of the nodes in the search space. However, the disadvantage is that sub-optimal solutions could be discovered, and without a depth bound, the algorithm might be unable to find a solution, even if it exists.

b. The breadth-first search algorithm always manages to find the optimal path to each of the rubbish bins (according to the set path cost). However, it will often explore many cells that are not on the desired path. DFS, in contrast, is quite dependent on the order in which the algorithm explores. Often it does not manage to find the most optimal route, however, DFS will usually either explore almost no cells that are not on the desired path (when the path follows its exploration strategy), or will explore many cells, not on the path (when the path doesn't follow its strategy).

BFS:	DFS:
- total_cost: 580.5	- total_cost: 7538.8
- total_cells: 13050	- total_cells: 32296

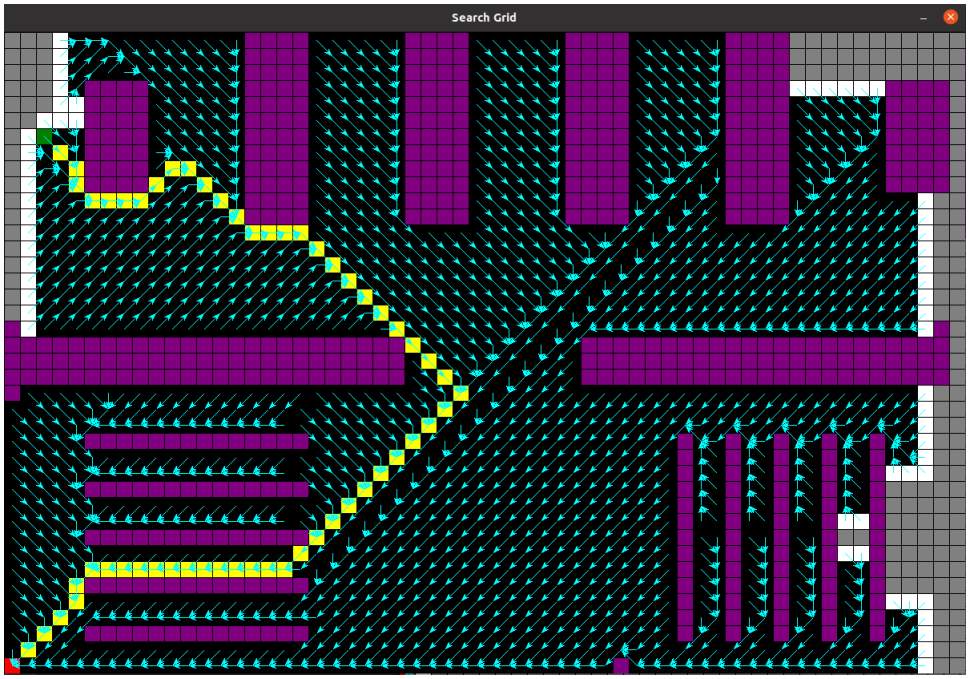

Figure 8 - Example 1: BFS

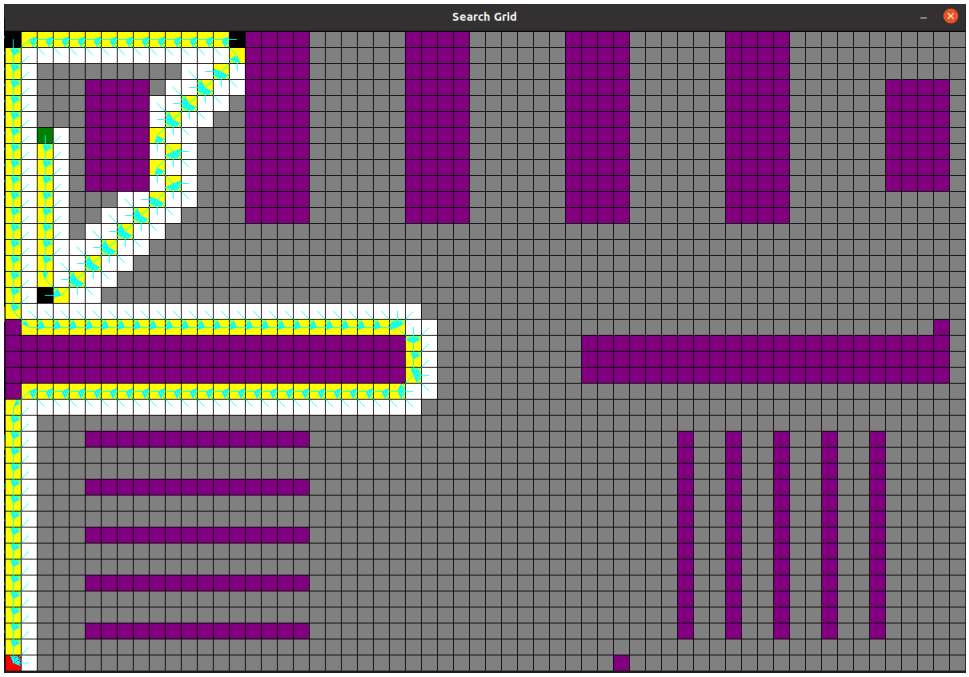

Figure 9 - Example 1: DFS

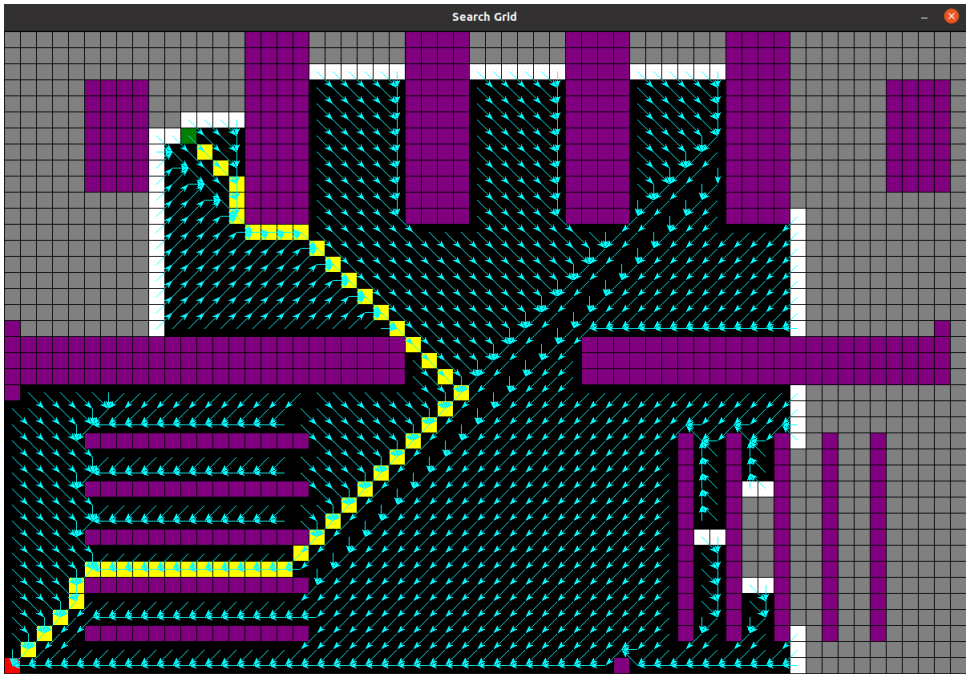

Figure 10 - Example 2: BFS

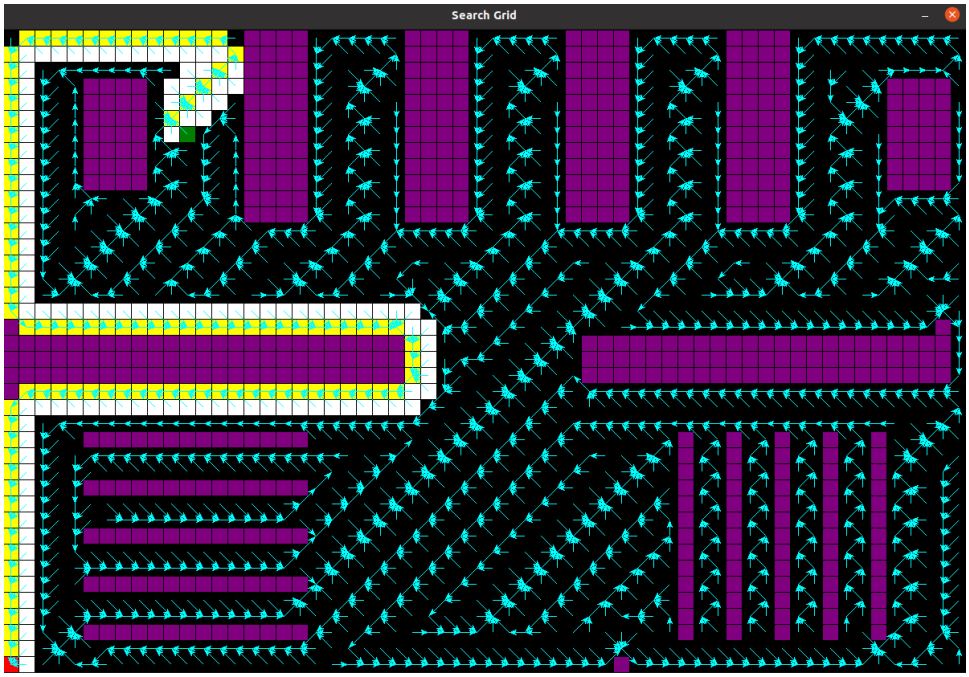

Figure 11 - Example 2: DFS

The total cost and total number of cells visited, when finding a path between each rubbish bin in turn, doesn't really give a sense of the difference in the optimality of the routes and the number of cells explored in the two algorithms as DFS just seems quite less optimal. However, when looking in the pictures, we can see that although BFS always manages to find a better path than DFS (or at least as good), there are cases in which DFS can be much more efficient in terms of memory and complexity as it explores smaller number of cells but can also be much less efficient in the cases when it explores more cells.

c. 2 key changes compared to BFS:
Dijkstra's algorithm is a path planning algorithm that uses forward search, similar to Breadth-First Search. In fact, BFS can be viewed as a special case of Dijkstra algorithm on unweighted graphs. There are two key differences between the two algorithms:

- Dijkstra uses a priority queue as opposed to bread first queue (the purpose of this is that in each step, it visits the node with the lowest cost, instead of the node closest to the source node).
- Dijkstra will compute the optimal path to all cells it encounters and resolves duplicates by choosing the path with the shortest length.
- Dijkstra gives an optimal solution for both weighted and unweighted graphs. BFS can only give optimal solutions on unweighted graphs or weighted graphs with equal weights.

d. To implement the Dijkstra algorithm, we made use of the PriorityQueue module in python that sorts the entries into ascending order of the given key, the distance travelled from the start point in our case (this distance is saved into the cell object as we will need it when resolving duplicates). When popping it returns the first entry with the smallest distance. When resolving duplicates, we compare the distance travelled so far + the Euclidean distance to the next cell with the distance already saved in the next cell and if the former is smaller, we update the parent and the saved distance of the next cell.

Total_cells: 13074
Total_cost: 550.6

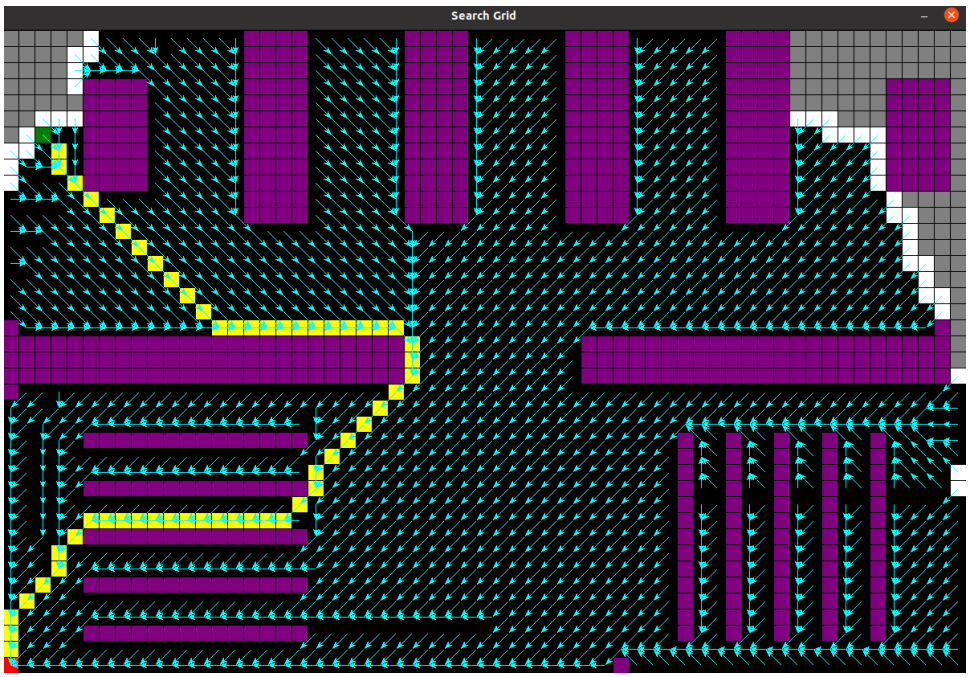

Figure 12 - Example 1: Dijkstra

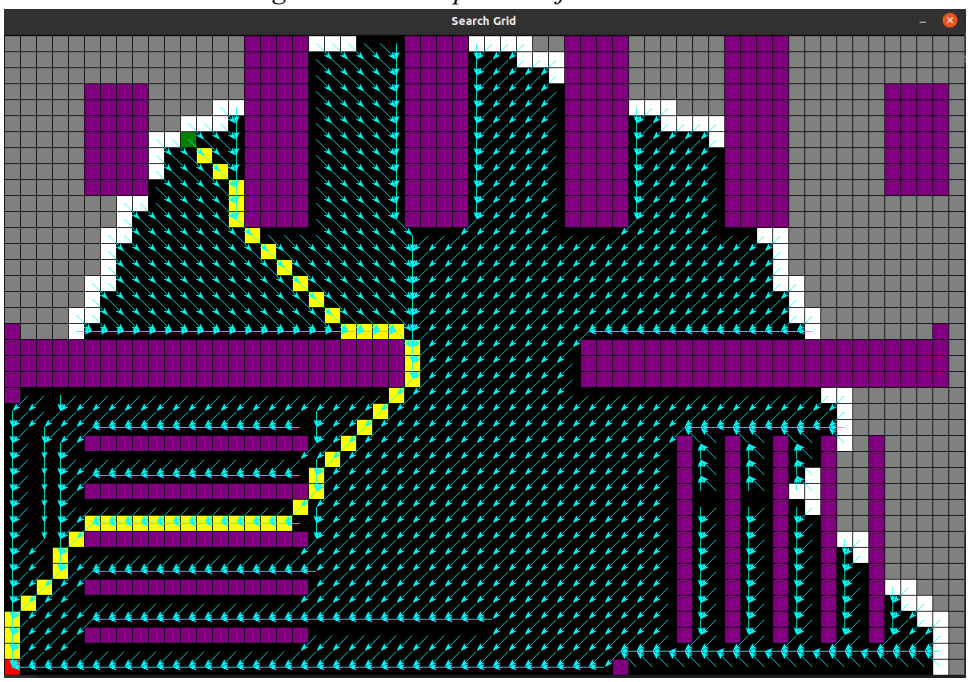

Figure 13 - Example 2: Dijkstra

Compared to BFS Dijkstra explores cells in a more circular motion, which in some cases leads to less cells explored and in some cases leads to more cells explored, the total for all rubish bins is a bit more than that of BFS. However, always taking the optimal path in terms of Euclidean distance rather than just number of cells, leads to more optimal solutions.

e. To implement the needed change, we just changed the returned value of L from the function to be 100 times bigger if the cell in current_coords (not the one in last_coords) is in the customs area, and to be 5 times bigger if the cell is in the secret door area.

What the change in the code did is that now, the algorithm will try to avoid exploring after going through the customs at almost all cost and is unlikely to explore after going through the secret door. This means that its performance improves quite a bit when the 2 rubbish bins are on the same side of the customs as it will not try to explore the other side but worsens in the cases when the two rubbish bins are on the two opposite side of the customs and the shortest path is through them, as now it will have to explore more and go through the secret door.

The improve of performance can be seen from the smaller number of total cells explored, but the total cost increased as none of the routes got shorter, but some of them got longer.

Total_cells: 12172
Total_cost: 659.2

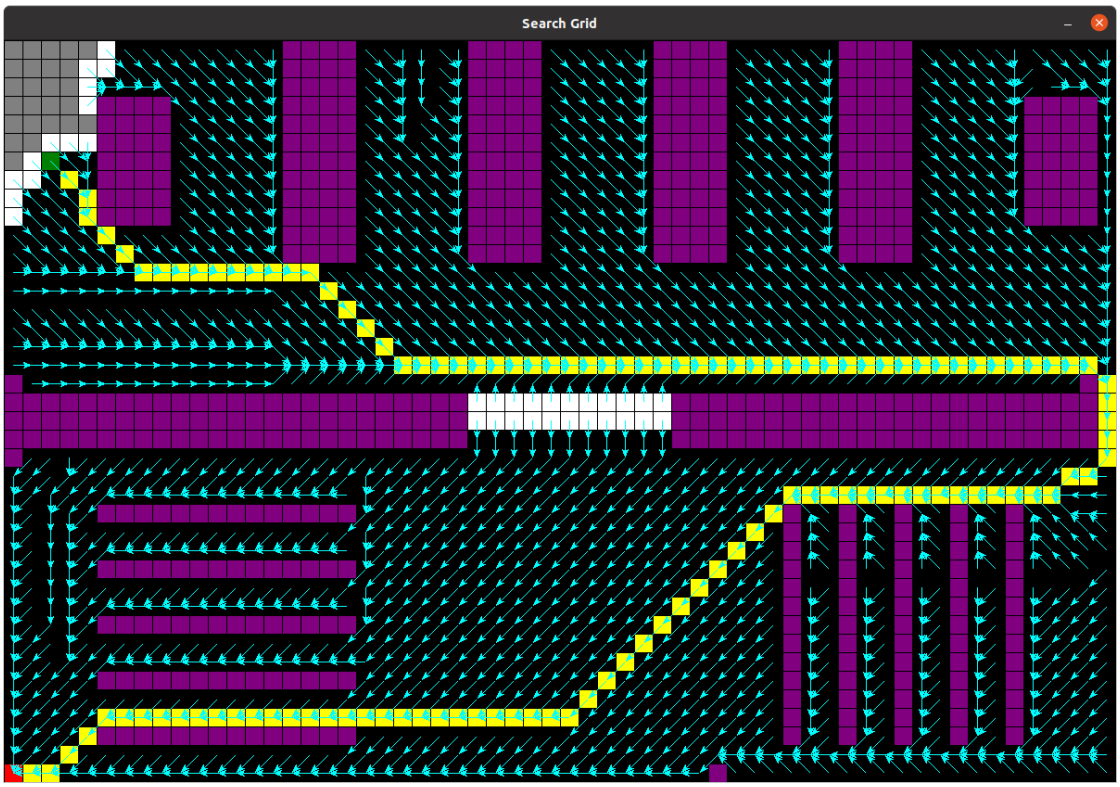

Figure 14 - Example 1: Dijkstra

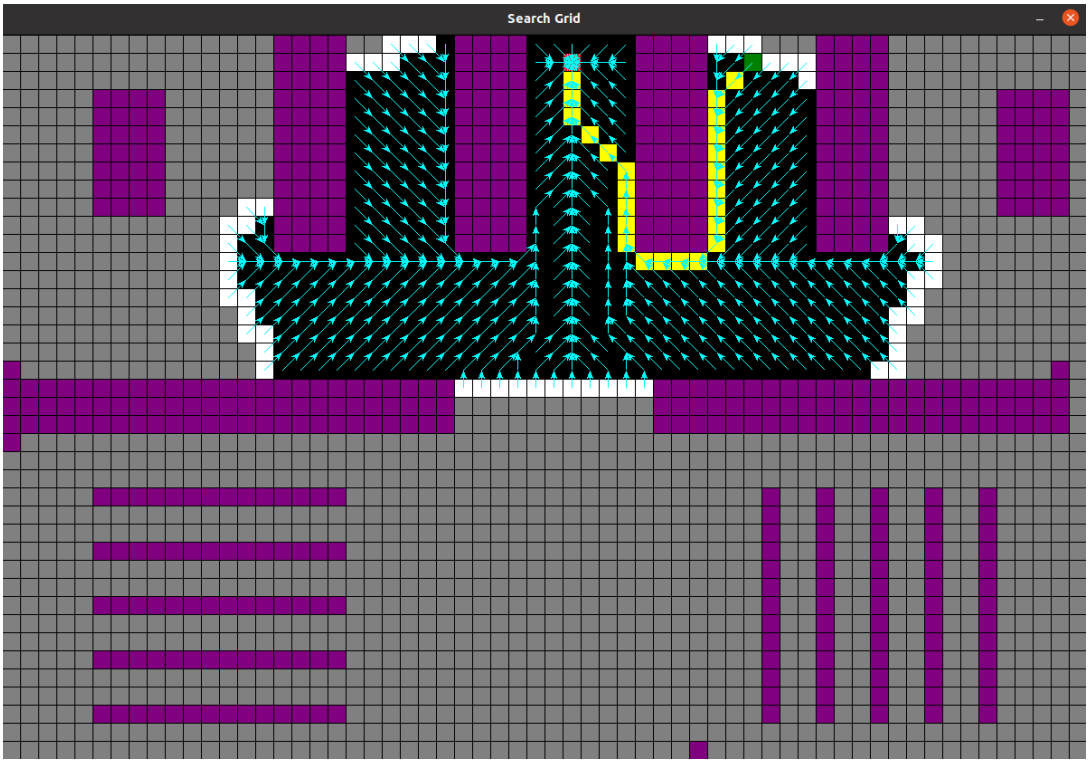

Figure 15 - Example 2: Dijkstra

f. The A* algorithm is an improvement from Dijkstra's Algorithm, achieved by employing a heuristic function. A* uses the same shortest path rewiring technique as Dijkstra but is able to perform a more directed search. Its search order is also a priority queue, but nodes are ordered based on the shortest predicted total path cost. A heuristic function is admissible if for every node, it is positive and it does not overestimate the cost to reach the goal (i.e., it is optimistic).

$$0 \leq \widehat{G_t}(s_t; \pi_T) \leq G_t(s_t; \pi_T) \qquad (13)$$

The Euclidean distance is a metric that gives the shortest path between any two nodes and the cost it expresses will always be smaller than or equal to the real cost to the goal, making it an admissible heuristic. The squared Heuristic distance is not a metric and might indicate a path with a higher cost, therefore overestimating the cost to reach the node, which makes it inadmissible heuristic. This can be best exemplified in the figure below, by trying to determine the path with the lowest cost from point A to point C.

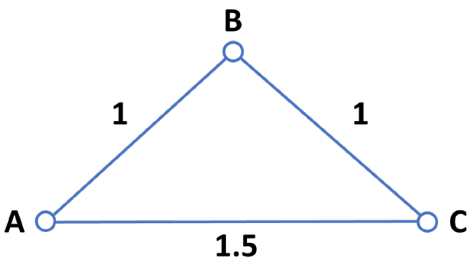

Figure 16 – Cost of Path from A to C

Euclidean Distance:
$\mathbf{1 + 1 > 1.5} \implies \boldsymbol{A \rightarrow C}$

Squared Euclidean Distance:
$\mathbf{1^2 + 1^2 < 1.5^2} \implies \boldsymbol{A \rightarrow B \rightarrow C}$

g. A more optimal heuristic would be a function that minimizes the exploration time outside of the optimal path. To that extent, the Euclidean distance would be an optimal heuristic because it always indicates the shortest path. However, this might not always be applicable in practice, because the robot is restricted to moving in 8 directions (↑↓←→↗↖↘↙). Therefore, the robot is not physically capable (because of the way it is programmed) to follow the shortest path. This situation is represented in Figure 17:

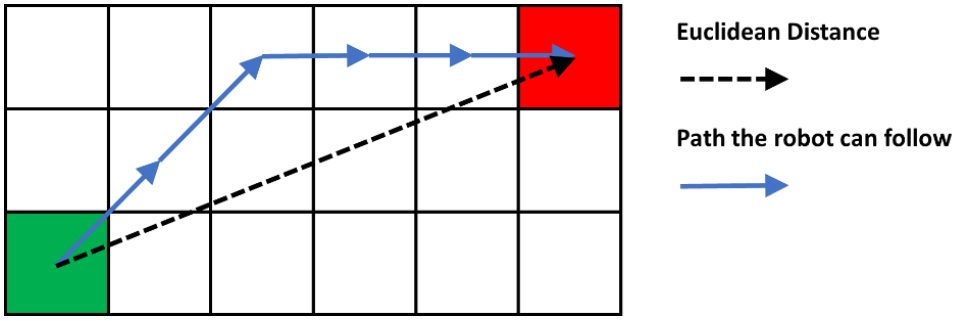

Figure 17 – Euclidean Distance vs Path the robot can follow

One particular heuristic commonly used in path planning algorithms when the robot can move in eight directions on the grid is the octile distance heuristic. The resulting motion of the robot will be a combination of straight-line motion and diagonal motion (just like the blue arrows in Figure X). One advantage of this over the Euclidean heuristic is that sometimes it can be less computationally expensive. The expression of the octile heuristic is as follows:

$$\widehat{G_t}(s_t;\pi_T) = \max(|x_t - x_G|, |y_t - y_G|) + (\sqrt{2} - 1)\min(|x_t - x_G|, |y_t - y_G|) \quad (14)$$

h. The A* algorithm that we implemented is almost identical to the Dijkstra algorithm. The only difference is what is used as the key in the priority queue. It is no longer just the distance that is already travelled from the start cell, but a heuristic value – the predicted distance from the current cell to the goal set (Euclidean distance) is added to the distance travelled so far. This causes the algorithm to explore much less and, in many cases, when the two rubbish bins are not on the two opposite sides of the customs, it goes almost straight to the goal cell. This improvement can be seen from the following examples, as well as the difference in the number of total cells explored, while the same most optimal paths are found (the total cost is identical).

Total_cells: 4941
Total_cost: 659.2

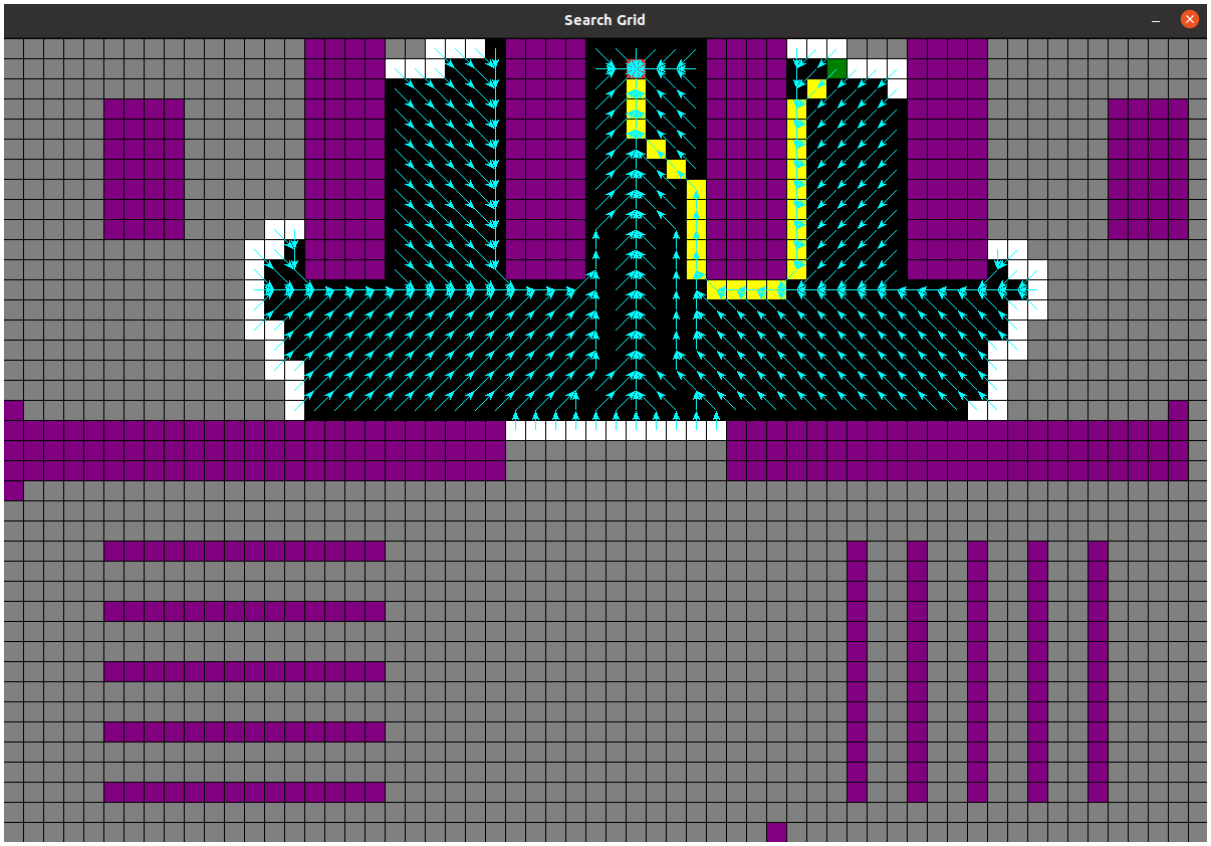

Figure 18 - Example 1: Dijkstra

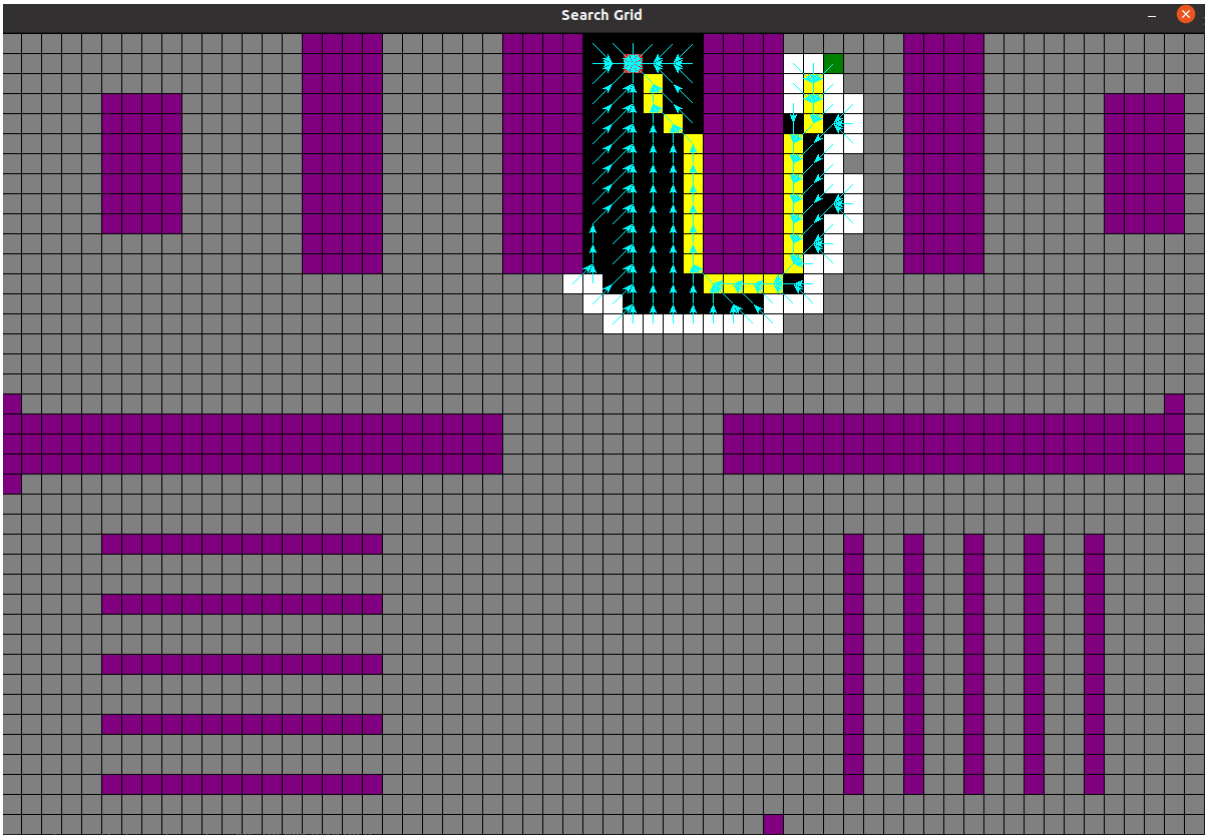

*Figure 19 - Example 1: A**

i. To select the optimal charging station the robot decides it needs to recharge at, we need to compute the optimal policy for the whole state space π_*. This means we need to compute the optimal action for any state in *airport_map*, or the optimal reward for any state/cell. However, we disregarded a few types of cells which the robot cannot get to: unknown, wall, baggage claim, chair and toilets.

We are given the mean and covariance of each station. We also know the action value function for policy π which reveals the expected return which can be obtained from a value of an action of any state following a policy of π: $q_\pi(s,a) = \mu_a - l(s, s_a)$. In order to maximise the action value function, the selected action must ensure that the difference between the charging station mean and the path cost to travel to the charging station s must be maximised. So $q_*(s,a) = \max q_\pi(s,a)$. For this we need to check if policy π is better than or same as policy π which can be done by calculating and comparing state-value function $\upsilon(\pi)$. The optimal state value function is defined as $\upsilon_*(s) = \max \upsilon\pi(s)$.

j. To implement the algorithm, we need to iterate through every cell on the map (except the ones we disregarded before) as each of them is a separate state and calculate the best action by maximising the difference between the mean charging rate and path cost to get to the station. We do this by looping through each of the 4 stations to compare the policies resulting from choosing each of them.

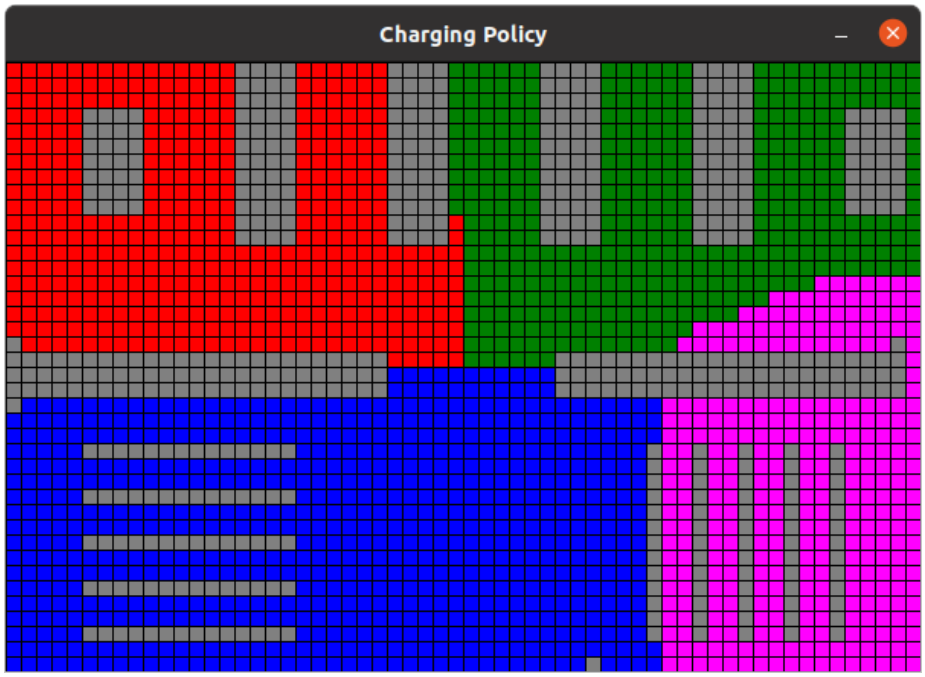

Figure 20 – Charging Policy with Euclidean Distance

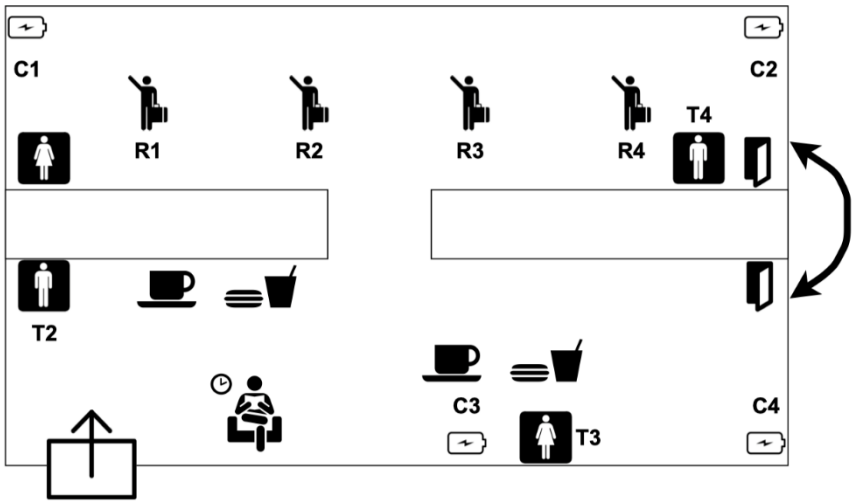

Figure 21 – Airport Layout

The means of the 4 charging stations are: $\mu_1 = 15$, $\mu_2 = 15$, $\mu_3 = 30$, $\mu_4 = 40$. The figures above show the learned policy and the locations of the charging stations. The results are expected because charging stations C1 and C2 are located in corners and have equal means, so the split is even. The location of C3 is almost in the middle of the lower part so the blue cells represent around 75% of the lower side of the map. Because $\mu_4 = 2\ \mu_2$, the pink cells are also distributed in the upper right side of the map.

k. The squared Euclidean distance was picked as the inadmissible heuristic for the A* algorithm. The inadmissible heuristic is previously described in point f). The new policy was computed by temporarily modifying the function *compute_transition_cost* in *airport_map.py* to look at squared Euclidean distance instead of Euclidean distance. The new learned policy is displayed below. The use of the inadmissible heuristic seems to have placed a higher path cost for moving to C4 from the green zone. This means that the fact that $\mu_4 = 2\ \mu_2$ has less weight in the learned policy.

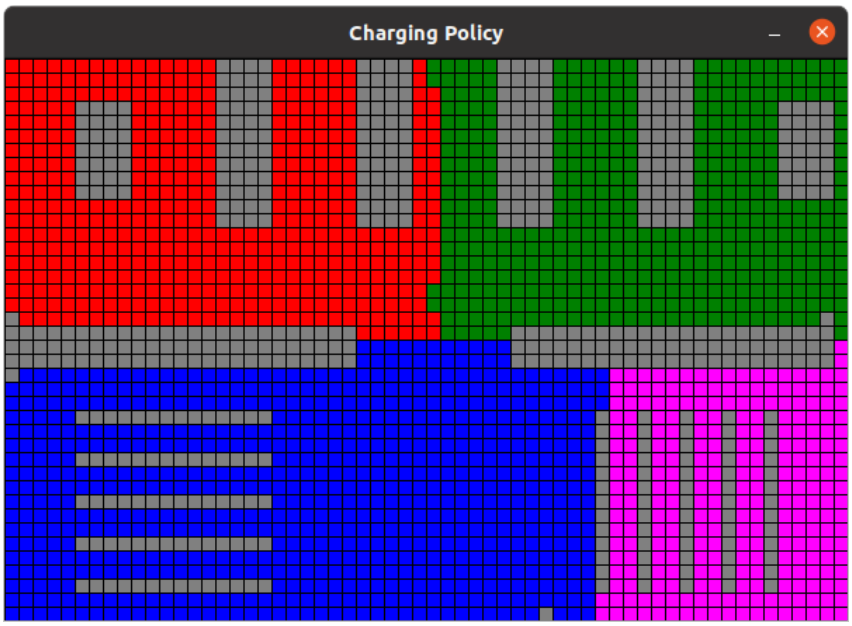

Figure 22 - Charging Policy with Squared Euclidean Distance

l. If the means and covariances for each charging station (μ_i, σ_i^2) were not known we can turn learning them into a k-arm bandit problem, that needs to be run before computing the best policy. However, pulling each arm in order will be quite expensive as the robot needs to travel to the arm it wants to pull, which would mean many circles around the whole airport. Instead, the robot can make only one circle around the airport, going through each charging station just once, but it will test it many times while it is there (so reducing the cost of travelling) and will estimate the mean and covariance this way.

Question 3

a. Finite Markov Decision Processes (FMDP) are mathematical models used in reinforcement learning tasks. They are an extension of multi-armed bandit problems, and as such, they aim to maximize the total reward by deciding an action at every time step. However, the main differentiator of FMDPs is that they have states.

An FMDP consists of $\begin{cases} S - Finite\ set\ of\ states \\ A - Finite\ set\ of\ actions \\ P - State\ transition\ probabilityes \\ R - Reward\ function \\ \gamma - Discount\ factor\ \in [0,1] \end{cases}$

- The states and actions are discrete and similar to those used previously in graph search. The fact that they are discrete makes the Markov Decision Processes (MDP) finite.
- The states are members of the state space:

$$|S| < \infty; \qquad S_t \in S; \qquad S_1 \neq S_2$$

- The actions are part of a state-dependent action space:

$$A_t \in A(s)$$

- The uncertainty in the process is modelled as a state transition probability:

$$p(s_t|s_{t-1}, a_{t-1}) = Pr\{S_t = s_t|S_{t-1} = s_{t-1}, A_{t-1} = a_{t-1}\} \qquad (15)$$

- The reward function gives real values which depend on the states and actions, based on a probability distribution.

$$p(r_t|s_{t-1}, a_{t-1}) = p\{R_t = r_t|S_{t-1} = s_{t-1}, A_{t-1} = a_{t-1}\} \qquad (16)$$

- The discounted returns multiply the terms in the series by a discount factor γ. This affects how much influence future rewards have on the computer return. The smaller the γ, the less weight is attributed to later rewards.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \qquad (17)$$

b. The roaming robot case study can be modelled as an FMDP problem, and the following components ca be identified:

- States: The states of the robot are represented the cells the robot is able to travel to, within the environment (open space, customs area, secret door, rubbish bins).
- Actions: The robot can remain stationary in the current cell or move in any neighbouring cells in one of the 8 directions, in order to reach the target.

- State transition probability: This is the probability that the robot will move to a certain cell on the grid, based on its previous action and state. The state transition probability is described in the equation below:

$$p = Pr\{S_t = Nominal\ Directon \mid S_{t-1} = s_{t-1}, A_{t-1} = a_{t-1}\} \quad (18)$$
$$q = 0.5(1-p) = Pr\{S_t = Deviated\ Direction \mid S_{t-1} = s_{t-1}, A_{t-1} = a_{t-1}\} \quad (19)$$

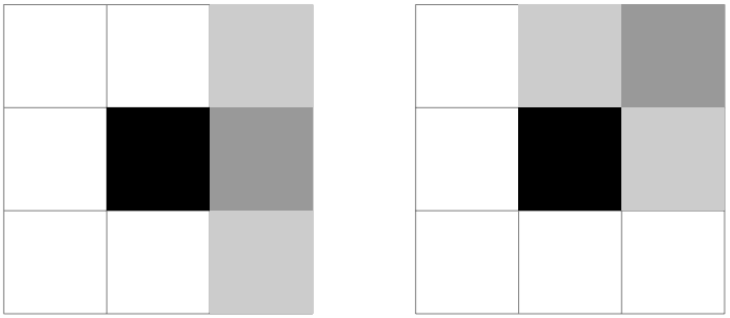

Figure 23 – Examples of the errors in the process model

If the robot does not move, it will remain stationary in its current cell with a probability $p = 1$.

- Reward function: This varies according to the cells visited by the robot. The total reward for a path that goes from one cell to a neighbouring one is equal to the Euclidean distance between the two cells, multiplied by (-1) times the penalty factor of the cell, as follows:

  - 5 for cells which are part of the secret door;
  - 100 for cells in the customs area;
  - 1 for any other cell.

  Additionally, some actions of the robot will be penalised by reducing the reward:
  - -10 for colliding with the luggage area;
  - -1 for colliding with any other cell.

- Discount factor: γ can take various values depending on the type of exploration preferred. For example, a low γ value would mean that early actions could be prioritised which could lead to sub-optimal actions on short-term, whereas a high γ value would add more influence to future steps, which could lead to more optimal actions on long-term.

c. The code in *q3_cef.py* was used to apply the policy iteration algorithm to the system. The first figure shows the rewards when $\gamma = 0.1$. It did not converge after 100 steps. As the discount factor is 1, the evaluation will keep on looking infinitely many states ahead, as 1^n is equal to 1 for all n. However, when we make $\gamma = 0.1$, it will get relatively small quickly and the evaluation will not look after the n-th state, when γ^n gets low enough (in this case it converges after just 7 steps).

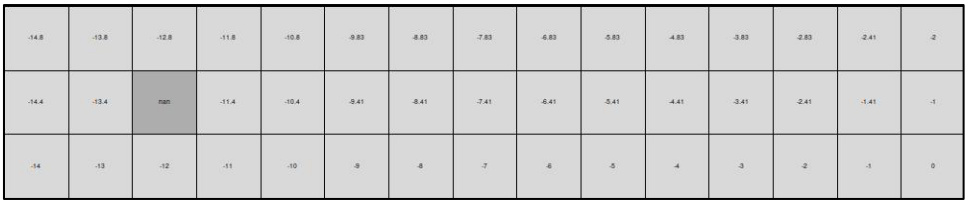

Figure 24 – Value function when $\gamma = 1$

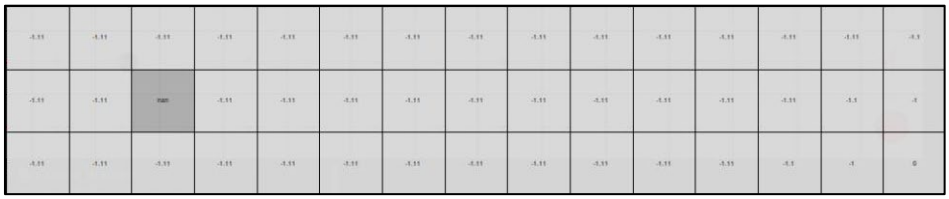

Figure 25 – Value function when $\gamma = 0.1$

d. For this part we modified the improvement step in the PolicyIterator class to iterate over every state and choose the best actions for it out of the 8 possibilities, according to the value function, using this equation:

$$\pi(s) \leftarrow argmax_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')] \qquad (20)$$

For the evaluation we decided to use γ=1 (this value will be used for all runs in this question, beside the one with γ=0.1 in part c), so the evaluation function looks at more states ahead (it will not converge but it will end after it reaches the maximum of 100 iterations or the terminal state).

e. Below we plotted the policies and value functions for each of the four values of p.

Case 1: P = 1

In this case, there is 100% probability that the robot will move in the nominal direction. It can be seen from the policy learned that the robot would reach the goal cell by performing the actions with the lowest penalty, such as avoiding the obstacle, not colliding, and moving straight (avoiding diagonals when it is possible).

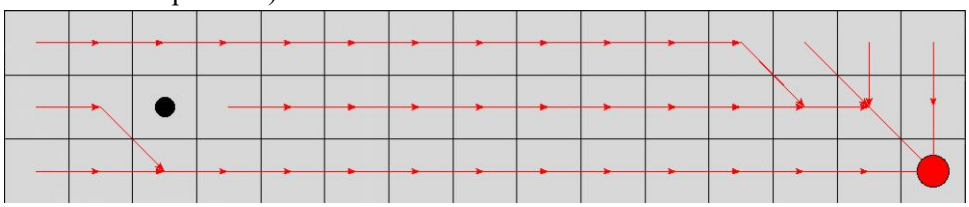

Figure 26 – Policy Learned when P=1

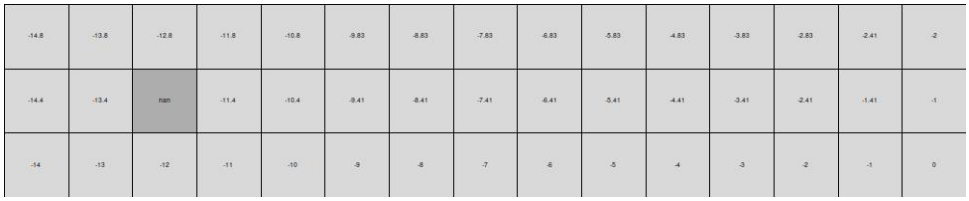

Figure 27 – Value Funciton when P=1

Case 2: P = 0.9

In this case, the robot will move in the nominal direction with probability 0.9 and one square on either side of the nominal direction with probability 0.05. Because of the probability to move to the wrong direction, there are a few differences in the learned policy, compared to when p=1. For

example, in the circled cells. The robot decides to move diagonally instead of continuing to the right, to avoid the probability of 0.05 of colliding with the wall.

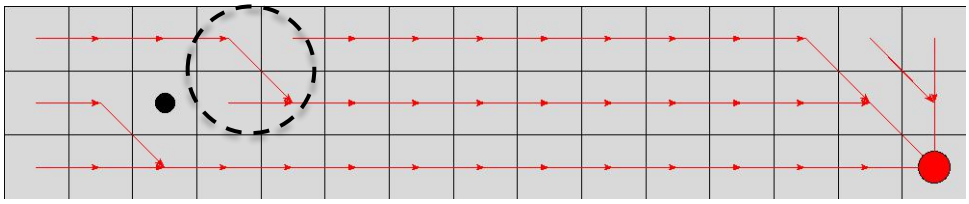

Figure 28 - Policy Learned when P=0.9

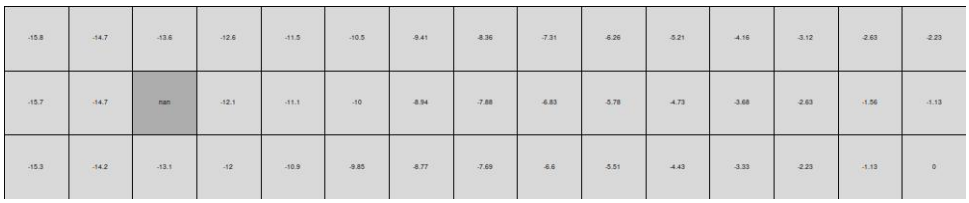

Figure 29 - Value Funciton when P= 0.9

Case 3: P = 0.6

In this case, the robot will move in the nominal direction with probability 0.6 and one square on either side of the nominal direction with probability 0.2. The probability of moving in the wrong direction is larger than before, but still substantially lower than the one-off moving in the nominal direction, and so there are not many changes in the learned policy compared to the one with P=1. This is because a collision with the wall might occur only in the first and third row, but it has a penalty of just 1. This means that there isn't much point in moving to the middle row, where there is no chance of collision but moving there will increase the cost of the path and there is still the probability of moving in the wrong direction and increasing it even further.

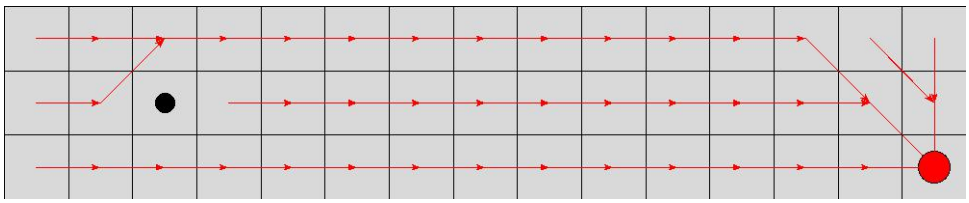

Figure 30 - Policy Learned when P=0.6

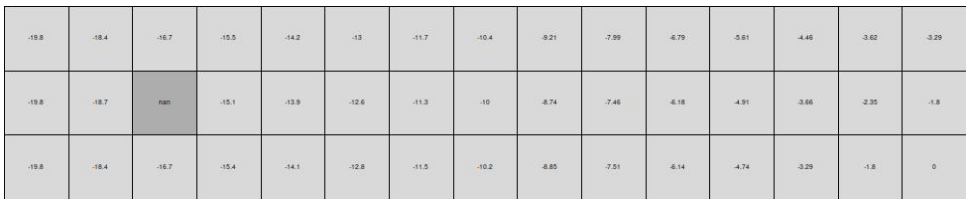

Figure 31 - Value Funciton when P=0.6

Case 4: P = 0.3

In this case, the robot will move in the nominal direction with probability 0.3 and one square on either side of the nominal direction with probability 0.35. The probability of moving on either side of the nominal direction is larger than the nominal direction. In contrast to the previous case, here there is a point of moving into the middle row (when the robot is in the third row) as the increase in the path cost is justified by the fact that following the nominal direction has the least chance, so staying in the third row would mean higher chance of collision or moving to the second row again.

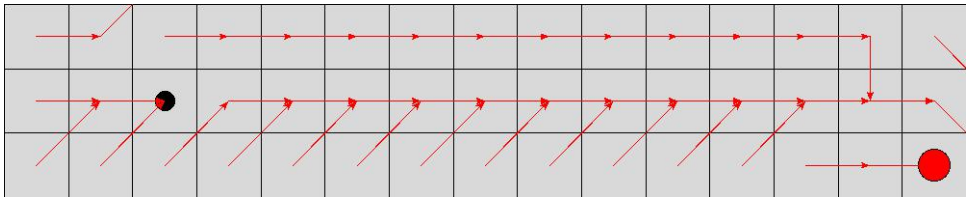

Figure 32 - Policy Learned when P=0.3

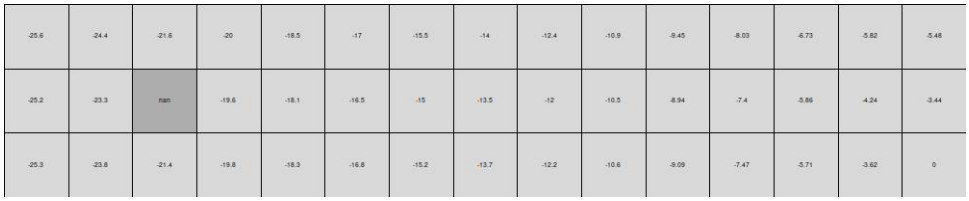

Figure 33 - Value Funciton when P=0.3

f. For this part we added 3 variables in the PolicyIterator class that keep track of the number of iterations during each evaluation step as well as the total number of evaluation iterations and the total number of improvement iterations.

As can be seen from the results below, the average number of iterations in the evaluation step doesn't decrease very much when P changes from 1 to 0.9 (just by around 7) as the probability of the nominal direction is still very high, though not 1, which helps it converge just a bit faster. The total number of iterations in the evaluation step increases when p is changed to 0.6 and 0.3, whereas the total number of improvement steps decreases just when P is changed to 0.6. However, in both cases the average evaluation iterations per improvement step increase substantially. This is because in these cases, the possibility to move not in the nominal directions is substantial (especially the case with p=0.3 in which the nominal direction is the least likely) and the evaluation needs to check not only the nominal directions but what happens if it moves in the error ones as well.

P=1
Total evaluation iterations = 205
Total improvement iterations = 5

P=0.6
Total evaluation iterations = 204
Total improvement iterations = 4

P=0.9
Total evaluation iterations = 170
Total improvement iterations = 5

P=0.3
Total evaluation iterations = 255
Total improvement iterations = 4

g. The *q3_g.py* was run to apply the alghorithm to the full problem. Displayed bellow are the generated policy (*Figure 34)*, the value function (*Figure 36*), and the policy iteration alghorithm statistics (4278 total evaluation iterations and 71 total improvement iterations).

A value of p=0.8 was used, meaning that there is an 80% probability of choosing the nominal action and a 10% probability of moving one square on either side of the nominal direction. To see the impact of the uncertainty on the learned policy, we compare the learned policy when p=0.8 (*Figure 34*) to the learned policy when p=1 (*Figure 35*). One example in the learned policy can be seen in the highlighted green area above the customs

area on the right. To avoid the cost of coliding with probability 0.1 when moving in the wrong direction, the policy learned when p=0.8 is moving diagonal instead of to the left. There are also several more examples of this behaviour in the learned policy.

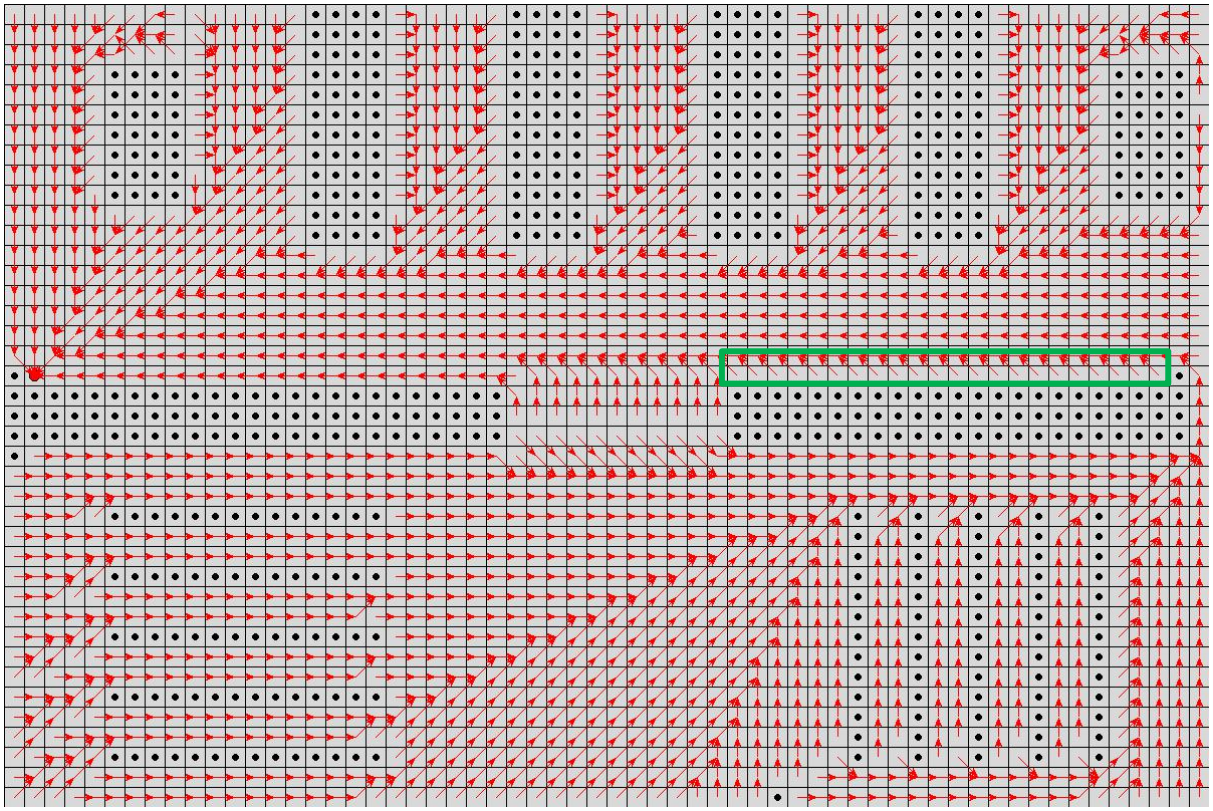

Figure 34 – Policy Learned when P=0.8

Figure 35 – Policy Learned when P=1

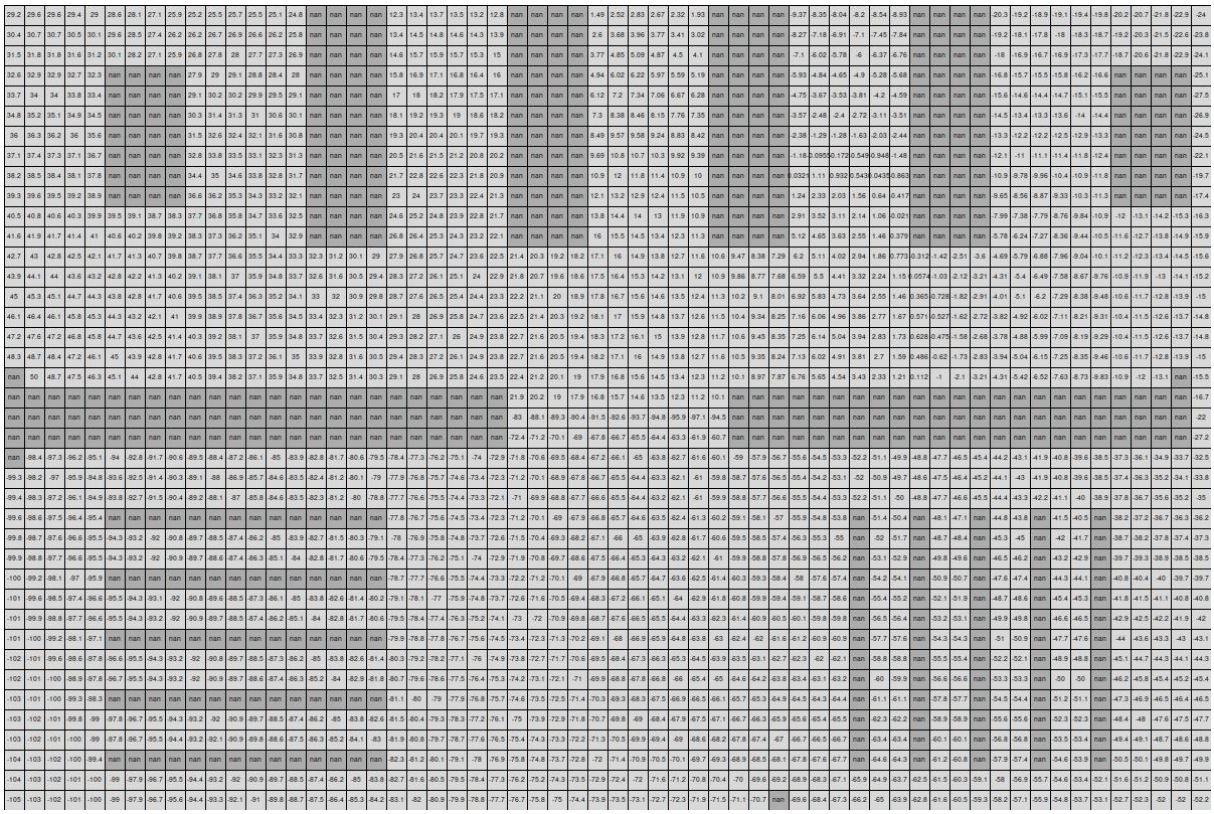

Figure 36 – Value Function when P=0.8

Total evaluation iterations = 4278
Total improvement iterations = 71

h. To implement the _ compute_optimal_value_function function we reused the code from the evaluation function in the PolicyIterator, as it is almost the same computation. The only change that needed to be made was to find the value for each possible action for each state by iterating through them, and record the best one, until **Δ** converges.

   To implement the _extract_policy function we reused the code from the improvement step function in the PolicyIterator class as we need basically the same functionality – to find which action maximises the value function (for which action do we get the value that was calculated in the _ compute_optimal_value_function).

i. *Q3_I.py* was modified to run the value iteration on the full map. Below we compare the policy iteration and the value iteration when p=0.8 - *Figure 37* (learned policy from value iteration) and *Figure 38* (learned policy from policy iteration). The learned policies are identical. However, it can be noticed from the table that the difference in the number of iterations is significant: 114 iterations for value iteration and 4278 iterations for policy iteration. This means that performing policy iteration is computationally much more expensive than value iteration, which in many cases will also give us the optimal policy, as

is in this case, or at least one that is very close to it. For the reasons mentioned above, we recommend the value iteration algorithm.

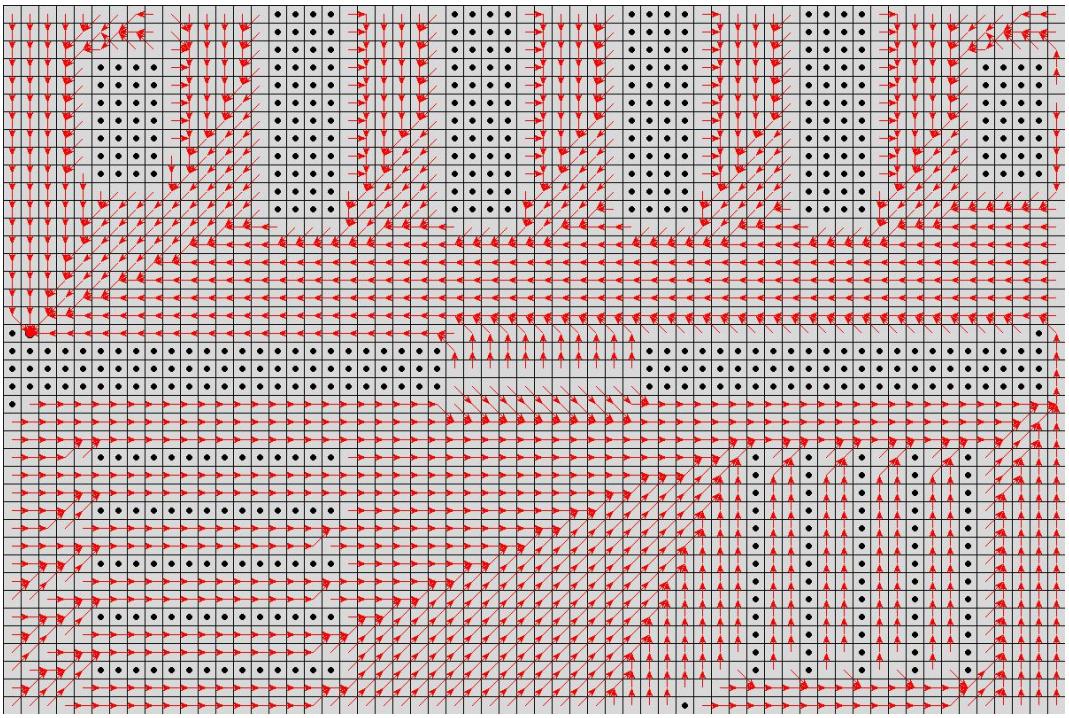

Figure 37 – Value Iteration Learned Policy

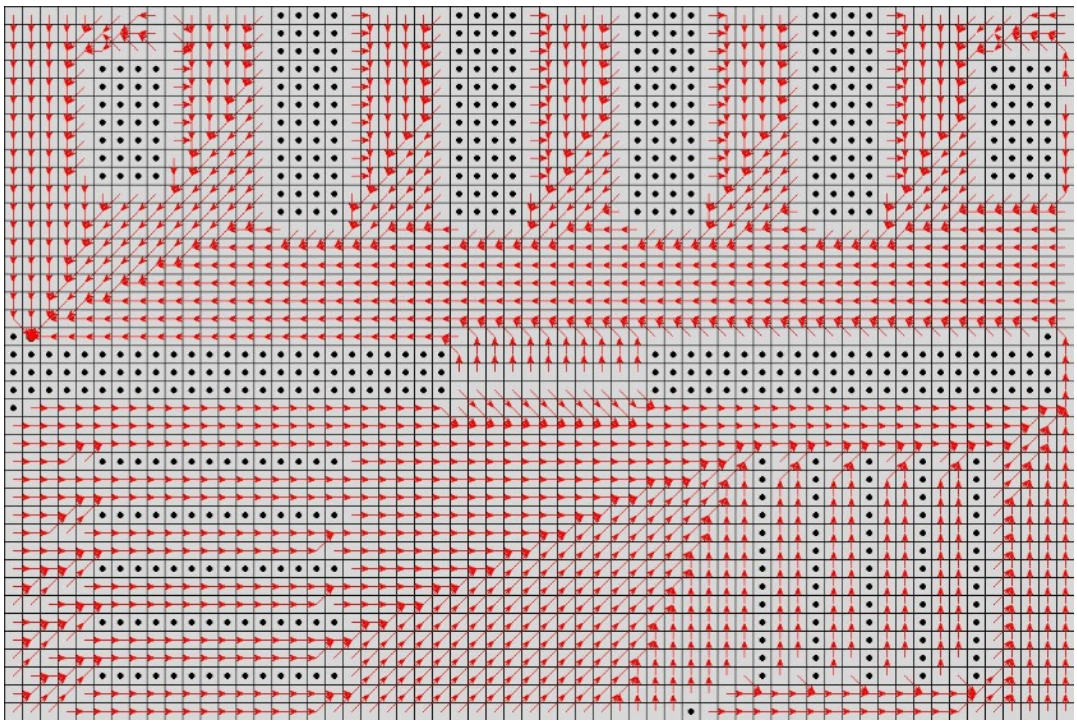

Figure 38 – Policy Iteration Learned Policy

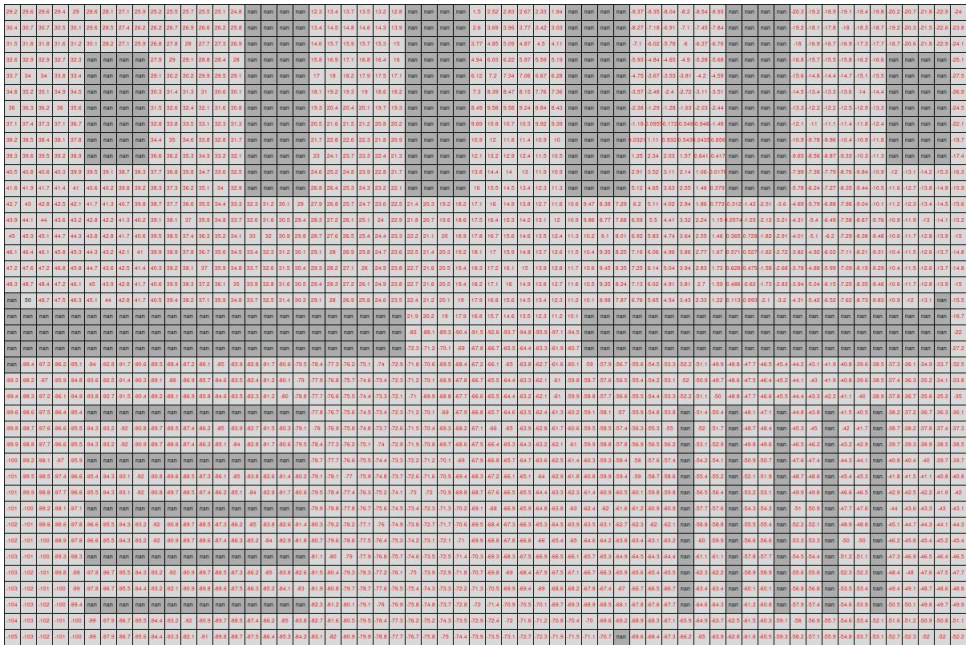

Figure 39 – Value Function from Value Iteration

	Policy iteration	Value iteration
Total iterations	4278	114

Question 4

- States: The states, just like in Q3, are represented by the cells that the robot is able to visit (cleaning sites, charging stations, open space, customs area, secret door, rubbish bins). Additionally, each state will have a battery level associated to it.
- Actions: Similarly, the actions the robot can take is remaining stationary or moving in any of the 8 directions in order to reach the next cleaning site or the charging station. The robot will consider the current state in order to decide what action to take, and therefore, it takes into account both the Euclidean distance and the battery level. For example, if it does not have enough charge to reach the next cleaning site and then a charging station, it will decide to first go to the nearest charging station in the sequence.
- State transition probability: This is the probability that the robot will move to a certain cell on the grid, based on its previous action and state. The state transition probability is described in the equation below:

$$p = Pr\{S_t = Nominal\ Directon \mid S_{t-1} = s_{t-1}, A_{t-1} = a_{t-1}\} \quad (21)$$

$$q = 0.5(1-p) = Pr\{S_t = Deviated\ Direction \mid S_{t-1} = s_{t-1}, A_{t-1} = a_{t-1}\} \quad (22)$$

- Reward function: Once again, this varies according to the cells visited by the robot. The total reward for a path that goes from one cell to a neighbouring one is equal to the Euclidean distance between the two cells, multiplied by (-1) times the penalty factor of the cell, as follows:

  - 5 for cells which are part of the secret door;
  - 100 for cells in the customs area;
  - 1 for any other cell.

Additionally, some actions of the robot will be penalised by reducing the reward:

- -10 for colliding with the luggage area;
- -1000 if the battery becomes flat (or other very large value);
- -1 for any other cell.

The rewards provided by the recharge stations can be represented as Gaussian-distributed bandit arms. Each time the robot recharges, it pulls that arm. The mean and variance of the arms are unknown, so the robot will estimate the mean by pulling each arm multiple times. The rewards are just the current estimation of the mean of that arm.

- Discount factor: In this scenario, a higher value of γ should be selected in order to emphasize the importance of future steps. This will enable the robot to avoid situations that would end up with a flat battery.

- Policy and Value Iteration:
  When it comes to Policy Iteration, in each cell the policy evaluation is run multiple times until it converges, and then the policy is updated. Value Iteration, on the other hand, only performs a single iteration of policy evaluation for every visited cell. This is reinforced by the trend observed in Q3 i), where it takes significantly more steps for the policy iteration to converge (4278) compared to value iteration (114). Therefore, Value Iteration outperforms Policy Iteration, and because of the rapid-changing nature of the airport environment, an algorithm that finds the optimal state value function using less steps to is preferred.