COMP0037 CW 1 2022/23

Group Name: `Group 0`
Gui Cheng Tong, Jeffrey Li, Youmna Srour

February 15, 2023

Additional Packages

On top of the standard Python libraries and coursework required libraries, we have also used the **joblib** library for saving and loading data for more efficient experimentation.

To install **joblib**, use the command **pip install joblib**.

Question 1

(a) The breadth-first search (BFS) algorithm uses a queue structure which means that the first node in is the first one out (FIFO). The depth-first search (DFS) algorithm uses a stack structure so that the last one in is the first one out (LIFO). The advantages of BFS are:

- It is complete, it is guaranteed that if a solution exists, it will find it.
- Assuming the costs are constant, if multiple solutions exist, the most optimal one is found.

Disadvantages of BFS are:

- BFS is more time-consuming compared to DFS on average.
- If the solution is at the end of the map, it will search all the cells before reaching it
- Consumes more memory since it stores all the cells and its neighbours that need to be visited.

The advantages of DFS are:

- Memory requirement depends on the number of nodes and is less compared to BFS.

Disadvantages of DFS are:

- If the graph is infinite, it may get stuck in an infinite loop.
- Not always optimal, it may not always find the shortest path.

(b)

Planner	Sum of all path costs	Sum of all cells visited
Breadth-First Search	580.47	13050
Depth-First Search	7538.8	32296

Table 1: BFS and DFS performance values

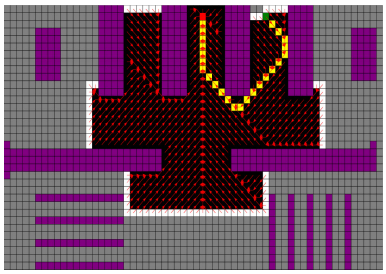

Figure 1: Inefficient path using BFS

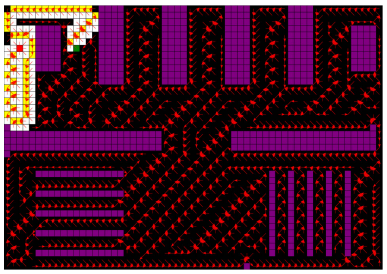

Figure 2: Inefficient search using DFS

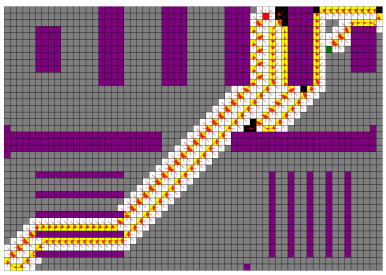

Figure 3: Non-optimal path using DFS

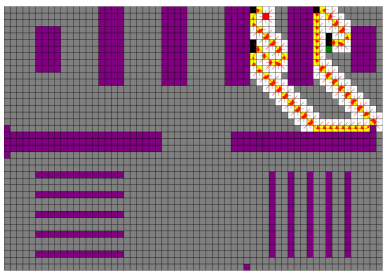

Figure 4: DFS path with new cell exploration order

As we can tell from the results found in Table 1, the breadth-first search is the better-performing algorithm, since the path costs and the number of cells visited are significantly lower compared to the depth-first search algorithm. Even though breadth-first seems to have significantly better results than depth-first search, it's still in some cases not showing the most optimal path (Figure 1). In Figure 1, the path is not optimal and instead it's going too far to the right. The reason for that is since duplicates are not resolved the planner uses the first path found. Additionally, the breadth-first planner doesn't consider path cost, which means the path found may not be the shortest.

You can tell by looking at Figures 2 and 3 that depth-first search is highly dependent on the start and goal locations as well as the order it's exploring the next cells. If the goal happens to be in a different direction than the planner, it may result in a large search and a less optimal path.

Much of the unexpected behaviour for DFS is due to the order of getting the next cell. As can be seen in Figure 3, the path goes all the way to the bottom left of the map, before going towards the upper right. If we change the order of the cell exploration the path of Figure 3 will change drastically, as seen in Figure 4, the path is more optimal since the order of the cell exploration prioritized searching in the direction of the goal

```
FORWARD_SEARCH
1   Q.Insert(x_I) and mark x_I as visited
2   while Q not empty do
3       x ← Q.GetFirst()
4       if x ∈ X_G
5           return SUCCESS
6       forall u ∈ U(x)
7           x' ← f(x, u)
8           if x' not visited
9               Mark x' as visited
10              Q.Insert(x')
11          else
12              Resolve duplicate x'
13  return FAILURE
```

Figure 5: LaValle's discrete forward planner pseudocode

(c) Dijkstra's algorithm is implemented by using a priority queue in LaValle's discrete forward planner pseudocode. The modifications are on lines 1, 3, 9, 10 and 12. Lines 1 and 10: Q.Insert() adds the cell into the priority queue with the path cost to the cell as the key and then sorts the queue, so the key with the lowest value is at the head. Line 3: `Q.GetFirst()` removes the item at the head and returns it. Line 9: In addition to marking the cell as visited, additional information is added to the cell, such as its parent cell and the path cost (the cost-to-come). Line 12: When resolving duplicates, the current path cost of the cell is compared with the path cost if x' is reached through x (path cost of x + additional cost to reach x'). If the latter is smaller, the key for that cell in the priority queue is updated, and the parent cell is updated as well.

(d) Modifications made:

The functions `push_cell_onto_queue`, `is_queue_empty`, `pop_cell_from_queue`, `resolve_duplicate`, and `_update_priority_queue` were implemented.

`push_cell_onto_queue`: This reflects the modification for `Q.Insert()`. The `put()` method of the `PriorityQueue()` object is used to put a tuple of `(path_cost, cell)` into the priority queue. `path_cost`

is computed as the path cost of the parent cell plus the additive cost (computed using `compute_l_stage_additive_cost`).

`is_queue_empty`: This uses the `empty()` method of the `PriorityQueue()` object. It returns a Boolean indicating if the queue is empty

`pop_cell_from_queue`: This reflects the modification for `Q.GetFirst()`. The `get()` method of the `PriorityQueue()` object is used. It returns the cell with the lowest path cost.

`resolve_duplicate`: This reflects the modification for line 12. As mentioned, it compares the current path cost of the cell with the cost of the path taken through x instead, x being the current cell examined. If the cost of the path taken through x is lower, the queue is updated via `_update_priority_queue`.

`_update_priority_queue`: This helper function is used to modify an existing item in the priority queue. It works by removing all items from the priority queue until it is empty and adding it to a new and updated priority queue. If it encounters an item with the same coordinates as the cell given, it will replace the path cost and then add it into the new priority queue. Then, the priority queue is changed to the new and updated one.

(e)

Planner	Sum of all path costs	Sum of all cells visited
Breadth-First Search	580.47	13050
Depth-First Search	7538.8	32296
Dijkstra's	550.65	13104

Table 2: BFS, DFS and Dijkstra's performance values

As can be seen from the above table (Table 2), Dijkstra's algorithm has the lowest sum of all path costs. This is due to Dijkstra's algorithm being able to produce the optimal path with respect to the cost function. The sum of all cells visited is more than BFS, showing that the method of exploring cells with the lowest path costs could cause it to explore more cells. However, the increase is not huge (an increase of 54 for about 13000 cells visited).

The statistics for BFS are very similar to Dijkstra's algorithm – the sum of all path costs is 580 for BFS compared to 550 and the sum of all cells visited is 13050 compared to 13104. This could be due to the similarity in the behaviour of the algorithms. BFS works by exploring cells with the least number of moves required to reach them (a move here refers to going to an adjacent cell); Dijkstra's algorithm works by exploring cells with the lowest path cost required to reach them.

For DFS, the sum of all path costs and the sum of all cells visited are very high compared to Dijkstra's algorithm. The sum of all path costs for DFS is approximately seven times that of Dijkstra's and the sum of all cells visited is more than twice that of Dijkstra's. This shows that DFS is very inefficient – it explores many cells and does not yield paths that are close to optimal.

(f)

Metric	Dijkstra's (D)	Dijkstra's with T (DT)
Sum of all path costs (Eucl)	550.65	635.23
Sum of all path costs (traversability)	1308.68	659.23
Sum of all cells visited	13104	12172

Table 3: Dijkstra's performance values with and without traversability

To complete Dijkstra's to consider the traversability costs, the function `compute_transition_cost()` in `airport_map.py` was modified. When the flag for using traversability costs is turned on, we check if the cell type is either SECRET_DOOR or CUSTOMS_AREA and return the Euclidean distance with the appropriate multiplier according to the transition cost.

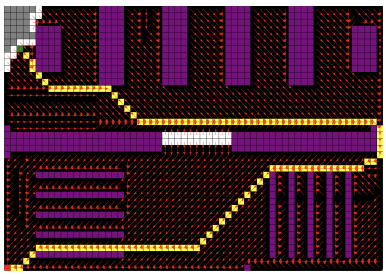

Figure 6: Dijkstra with compute cost

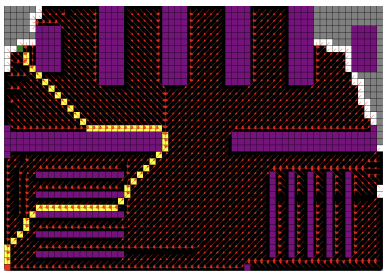

Figure 7: Dijkstra without compute cost

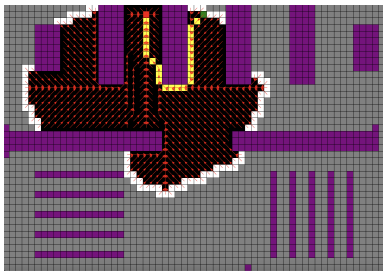

Figure 8: D visits past customs

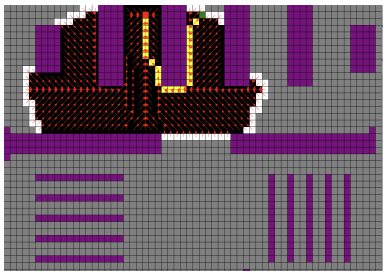

Figure 9: DT does not visit past customs

As shown in Figure 5, the path is much longer, but it avoids the customs area and uses the secret door as desired.

Table 3 shows the performance of Dijkstra with and without using traversability costs (D and DT). The Euclidean path cost for D is lower than DT, however, after considering traversability costs, DT has much better performance. An interesting point is that DT visits fewer cells than D, and this is due to the high cost of going through customs reducing needless exploration, as shown in Figures 7 and 8.

Question 2

(a) The A star algorithm would be implemented like the Dijkstra algorithm. Q would be implemented as a Priority Queue, where Q.GetFirst() retrieves the state with the highest priority.

In this case, the priority is determined by an estimate of the minimal total path length from the current state to the goal states. The estimate is calculated by adding the actual Cost to Come from the start state to that state with an estimated Cost to Go to the goal state calculated using a heuristic and is inserted along with the state using Q.Insert().

Resolving duplicate is done in the same fashion as in Dijkstra, where the path with the lower cost to get to that state is chosen, the parent of the cell is updated to the parent which gives that lower path cost, and the corresponding cell within the Priority Queue is updated with that lower path cost.

(b) When a heuristic is admissible, it means that it never gives an over-estimate to the actual cost-to-go value from the current state to the goal and is non-negative. An example of an admissible heuristic is the Euclidean distance because firstly it is never negative due to the addition of the inner squares, and it never overestimates the cost to go from the current state to the goal state as the path between two points can only be as little as the straight-line distance between them. An example of an inadmissible heuristic is a heuristic which yields the Euclidean distance but perceives one or more of the states on the only actual optimal path as having a very high cost (e.g., in extreme cases higher than the maximal total path length of the given problem) because although it is always positive, this heuristic yields an extensive over-estimate to the cost to go of the current state to the goal.

This reason why admissible heuristics are important is because taking the two functions described above when using the inadmissible heuristic, the A* algorithm could reach the goal state before fully exploring the actual optimal path due to the heuristic's perceived high cost on the optimal states (which would have the lowest priority within the priority queue). This means that the algorithm would not provide an optimal path with minimal cost to the goal and is undesired. On the other hand, admissible heuristics such as Euclidean distance would allow the A* algorithm to find an optimal path because the heuristic is at most the actual cost to go. As the agent gains more realistic information through deeper searches and discovering the actual cost to come to the path, the estimation of the total path cost (actual cost to come

+ heuristic cost to go) would also be refined and closer to its actual cost. A path may seem promising at first giving a low sum of cost to come plus heuristic value, but as the agent explores more and more along this path, it might turn out to have higher costs than other paths' estimated total cost, and this gives the agent a chance to switch and explore alternative paths which may possibly provide a better path. In this case, the heuristic act more as a way to guide the agent's exploration but does not stop the agent from finding the optimal path.

An example of an inadmissible heuristic which still guarantees to provide an optimal solution is an admissible function with an additive constant, such as the Euclidean distance f(x) with a constant c = maximal path length of the given problem. Hence the total estimated path cost becomes `cost_to_come + h(x) + c`. The constant wouldn't affect the way in which the algorithm explores as it wouldn't change the priority of the states, however, it makes the heuristic inadmissible as the heuristic now yields an overestimate of the path.

(c) The optimal heuristic that A* can use is the actual cost to go from current state to the goal. Could it be used in practice? No because this information to the actual cost to go in most cases would not be known by the agent though good estimations of the actual cost to go could be generated. For example, in the current airport scenario, the cost to go would not be known by the agents due to many factors, such as walls and traversibility costs. This is more so the case for stochastic environments. One possible way to find the actual cost to go might be using an algorithm that searches through the entire state space, generating the optimal path distance from each state to the goal, store this information and then use A* to do the search again using this "heuristic" information, however this defeats the purpose of using A* in the first place.

(d) The implementation of A* is built upon the implementation of Dijkstra, where the only addition is that when pushing new cells to the Priority Queue, on top of the cost to come (as in Dijkstra), an estimated cost to go generated by a Euclidean distance heuristic is also added.

(e)

Planner	Sum of all path costs	Sum of all cells visited
Breadth-First Search	580.47	13050
Depth-First Search	7538.8	32296
A*	550.65	3415

Table 4: Results without traversability Costs

Planner	Sum of all path costs	Sum of all cells visited
Breadth-First Search	1420.51	13050
Depth-First Search	22747.43	32296
Dijkstra's	659.23	5167

Table 5: Results with traversability Costs:

As shown in Table 4, for an environment that doesn't consider the traversability costs, A* has the lowest sum of all path costs, and it is also worth noting that it has the same sum of path cost as the Dijkstra's algorithm as an admissible Euclidean distance heuristic is used, hence the algorithm would provide the optimal path solution.

In terms of quality, BFS produced a very similar sum of path costs as to A* and is only 30 units extra. In general, BFS could produce optimal solution given that the cost from one state to another is constant as the nature to its "radiating search" schedule, however this isn't the case with the current scenario with the existence of diagonal traversals, hence resulted in a less optimal solution.

This issue is emphasized in scenarios that has traversability costs associated with them (see Table 5 above). This is a factor to the path in which the BFS algorithm does not consider as it is not designed to be an optimal path finding algorithm. An example would be the penalty factor added to the current problem for traversing customs area, and we would see a more distinctive difference in sum of path costs between BFS and A*, with BFS performing way worse than A*.

In terms of quantity, BFS visited significantly more cell than A* (about 4 times more), due to the "radiating search" behaviour of this algorithm, which searches at increasing radius and only moves to the higher radius when all states at current radius are searched.

DFS on the other hand is nowhere near the performance of the two other algorithms, both qualitatively and quantitatively, with about 14 times the path cost and 9 times the total number of cells visited to the A* algorithm.

Qualitatively, DFS also suffers that same issue as with BFS that they don't consider traversability costs and does not yield optimal paths. In the airport scenario, DFS could also perform a lot worse as it may find paths that has successfully reached the goal, but traverses back and forth from customs multiple times, resulting in an enormous penalty.

Quantitatively, the reason why it has so many cells visited is because of how DFS would search all the way down one path before backtracking and explore alternative paths. Although it could be lucky and get to the goal at its first couple of full path searches, which may yield a low number of cells searched, but that is highly unlikely in most scenarios.

Question 3

(a) To model the system as an FMDP, it must be expressed in terms of the components in an FMDP: $\langle S, A, P, R, \gamma \rangle$.

Finite set of states S: The set of cells. Each state can be expressed as a 3-tuple of the x coordinate, y coordinate, and the type of cell. The terminal state is the known goal location.

Finite set of actions per state A: The set A for each cell is the set of actions that the robot can choose to take to move an adjacent cell. It can be expressed as a 2-tuple (x, y) where x indicates the change in the x-coordinate of the robot and y indicates the change in the y-coordinate of the robot. The values of x and y are limited to $\{-1, 0, 1\}$ and (x, y) cannot be $(0, 0)$. The maximum size of A for any cell is 8, which are the 8 cells surrounding a cell in the middle of the grid. If a cell is on the boundary, the set A is smaller since the robot has less valid cells to move to.

State transition probabilities P: This models the randomness in the robot movement. In this process, for any action a, the robot will either move in the direction specified in a with probability p or in an adjacent direction with probability $0.5(1 - p)$.

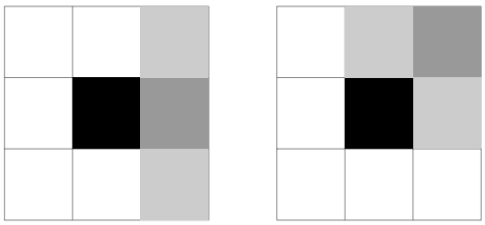

Figure 10: Dark grey indicates the action a, light grey indicates a movement in an adjacent direction

Reward function R: This models the cost associated with taking a certain action at a certain state. The costs for movement and collision can be modelled by the reward function. Each action the robot takes incurs a penalty depending on the Euclidean distance and the type of cell that the robot will end up in. If the robot ends up in a cell of type "secret door", the cost incurred is $5 \times$ Euclidean distance. If the robot ends up in a cell of type "customs area" the cost incurred is $100 \times$ Euclidean distance. If the robot ends up in a cell of type "luggage reclaim area" the cost incurred is Euclidean distance + 10. If the robot ends up in a cell of type indicating other objects, it incurs a cost of Euclidean distance + 1. To ensure that the robot reaches the goal there should be a large positive reward (e.g., 10000) if the robot ends up in the terminal state.

Discount factor γ: The discount factor is used when calculating the return. In this case, the task is episodic and thus the return will always be finite. γ can be set to 1 in this system so that the return accurately reflects the cumulative sum of rewards until the end of task.

(b) γ represents the discounting factor; this is a value between 0 and 1 that affects the importance of future rewards. The lower the value of gamma is the lower the importance future rewards have.

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{1}$$

If $\gamma = 1$, then the reward would be scaled by 1^k, which will cause the discount factor to be insignificant.

The initial policy is to always move to the right, given the trajectories generated by this policy, there will always be a "ending state" that collides with an object causing a penalty cost of either –10 or –1 depending on if it collides with a reclaim belt, other objects or off the edge. The penalties will propagate back along the same trajectories, causing the rewards of the states to be the same as the "ending state" of its trajectory. In this case, the state values would decrease at a constant rate since gamma is 1, hence the delta being the maximum change in state values would be constant, and therefore never less than theta, causing the evaluation loop to never terminate.

A suggested solution would be to lower the value of gamma using the setter function provided. With gamma lower than 1, with each iteration, the change in state values will decrease and eventually cause delta to become less than theta, and the evaluation iteration stops, causing the algorithm to converge.

Figure 11 shows that the algorithm doesn't converge when gamma is set to 1 and the maximum number of iterations is reached instead. In Figure 12 you can see that the algorithm converges when setting gamma to values lower than 1.

```
Maximum number of iterations exceeded
Finished policy evaluation iteration 1
Finished policy evaluation iteration 2
Finished policy evaluation iteration 3
Finished policy evaluation iteration 4
Finished policy evaluation iteration 5
Finished policy evaluation iteration 6
Finished policy evaluation iteration 7
Finished policy evaluation iteration 8
Finished policy evaluation iteration 9
Finished policy evaluation iteration 10
```

Figure 11: Policy iteration with gamma = 1

```
Finished policy evaluation iteration 6
Finished policy evaluation iteration 7
converges
Finished policy evaluation iteration 1
converges
Finished policy evaluation iteration 1
converges
```

Figure 12: Policy iteration with gamma = 0.7

(c)

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Loop:
   $\Delta \leftarrow 0$
   Loop for each $s \in \mathcal{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))[r + \gamma V(s')]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   old-action $\leftarrow \pi(s)$
   $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
   If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

Figure 13: Policy iteration in Pseudocode

The policy improvement step was implemented using the pseudo-code provided in the lecture slides (Figure 13). The algorithm works by iterating through the cells of the grid to find the action that produces the highest reward given the state of the transitive probabilities, then it sets the chosen action as its policy. It first obtains the environment's dynamics of state and action pair using the

`self._environment.next_state_and_reward_distribution(state, action)` method, which returns a list of the next states (`s_prime`), a list of rewards for each corresponding action, and a list of probabilities for transitioning to the corresponding `s_prime`. The new action value is then computed by applying Bellman's equation. The aim of the algorithm is to find the action that maximises the summation in Bellman's formula:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \tag{2}$$

The new action value is then compared with the previously highest calculated action value, if its higher then `argmax_action` is assigned the current action, and the highest calculated action value is updated. After the iteration over the action space, the algorithm then checks if the new policy is the same as the old policy, if it is then the `policy_stable` flag is set to `True`, meaning that the optimal policy has been found and the policy iteration stops. If the new policy is not the same as the old policy, then `policy_stable` is set to False, and the algorithm moves onto the policy evaluation and repeats the process again.

(d)

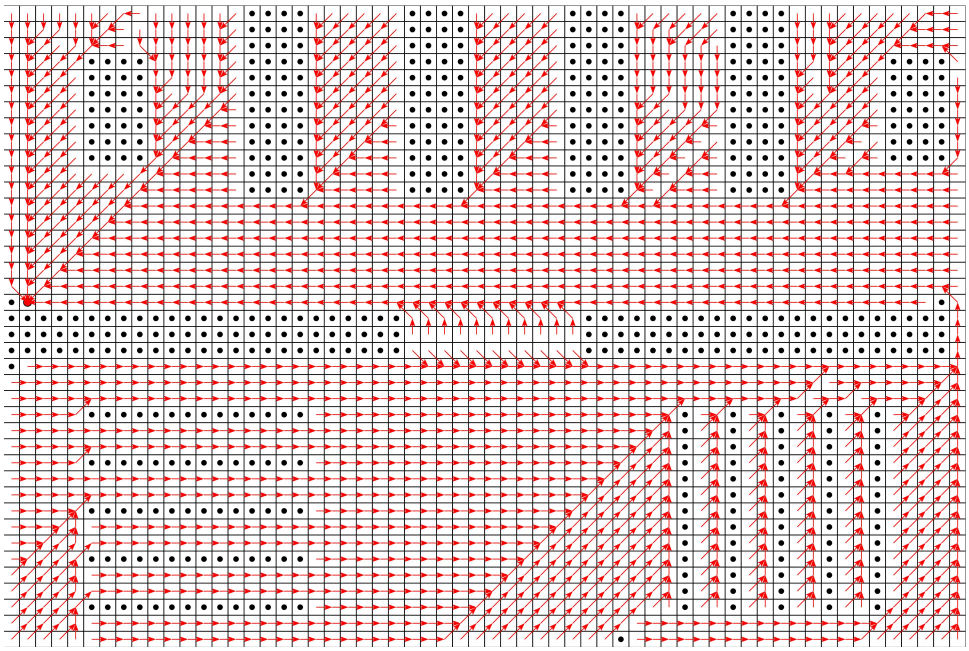

Figure 14: Policy iteration with p = 1

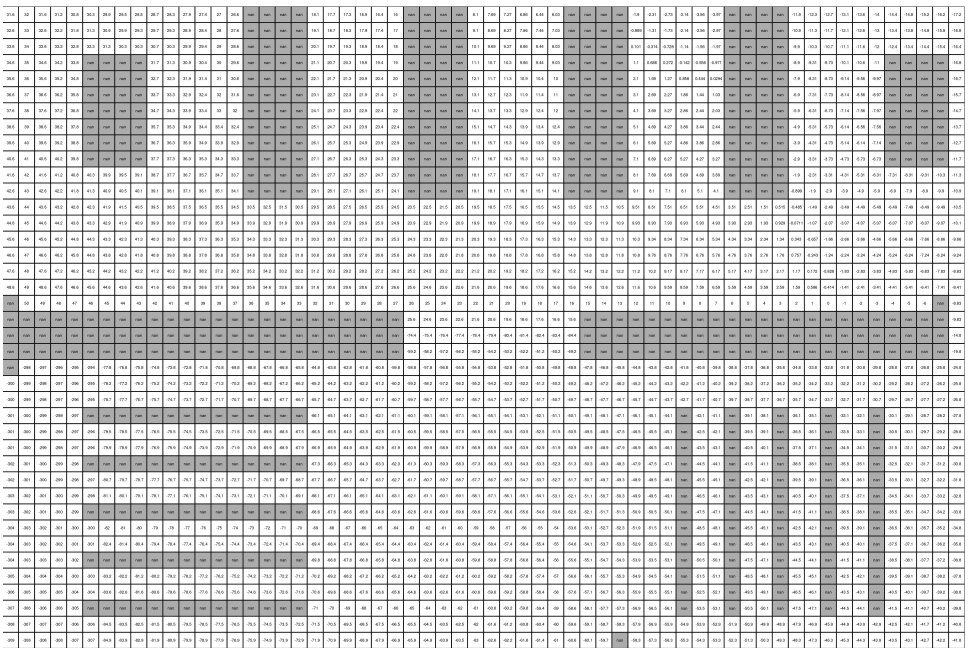

Figure 15: Value Function with p = 1

In general, as p decreases, the values in the environment decreases and also for p less than 1, values are higher for states that are slightly away from states that may lead to high penalty such as the baggage reclaim area. In Figures 14 and 15, when p=1 the policy and value function are deterministic, meaning there's only one action and the corresponding value for each state that can be taken, making it very straightforward. The policy always selects the action that would lead to the highest reward, and you can also see it reflected in the value function values of the cells.

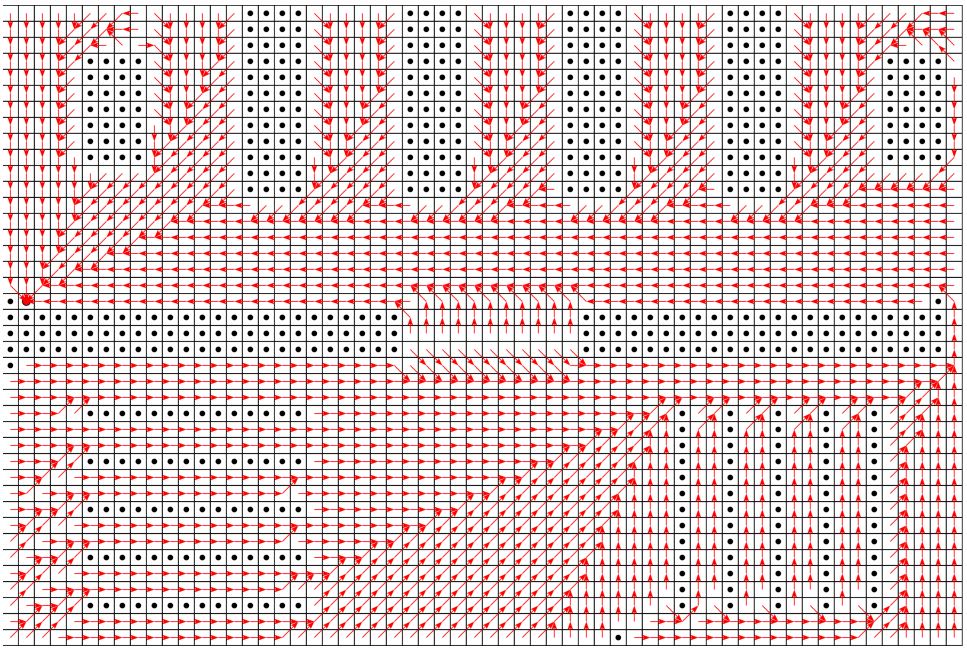

Figure 16: Policy iteration with p = 0.9

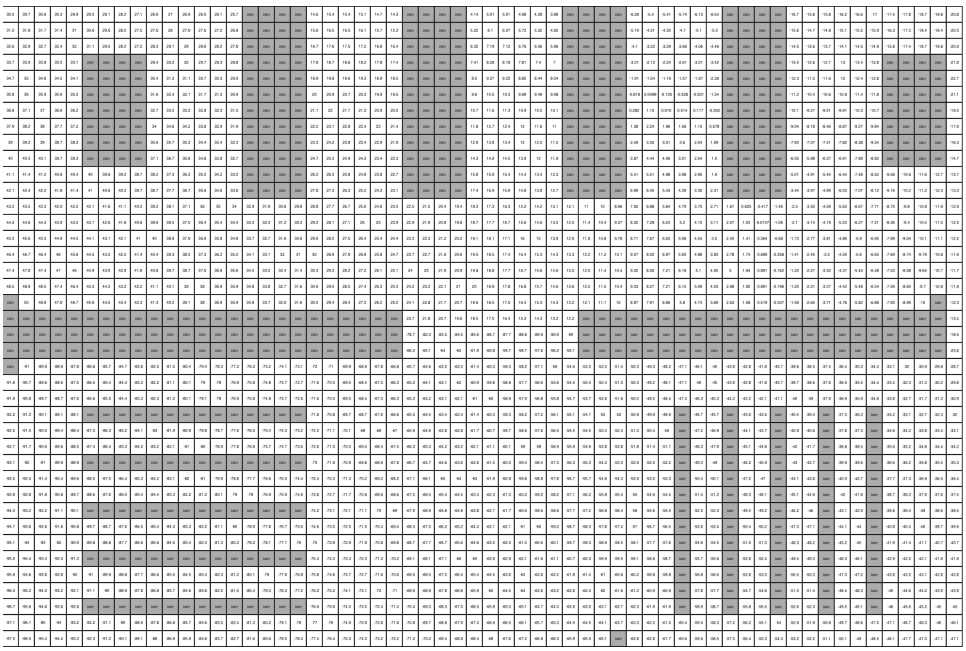

Figure 17: Value Function with p = 0.9

When p=0.9, there is a small uncertainty in the resulting action that will be carried out at each state, and the value function would converge to expected values by taking into account of this uncertainty. A noticeable difference between the policies of p=1 and p=0.9 is at the right side of the baggage reclaim areas, whereas in p=1 the policy is just going down until reaching the bottom of the reclaim area, whereas for p=0.9 you would notice that the policies are going down right. This is because the uncertainty would mean going down might result in hitting the baggage reclaim area, and would receive a -10 penalty, but going down right would avoid this penalty while also having the possibility of moving in downward direction. This is also reflected by its value function, which has less value in this area than p=1 to account for this uncertainty.

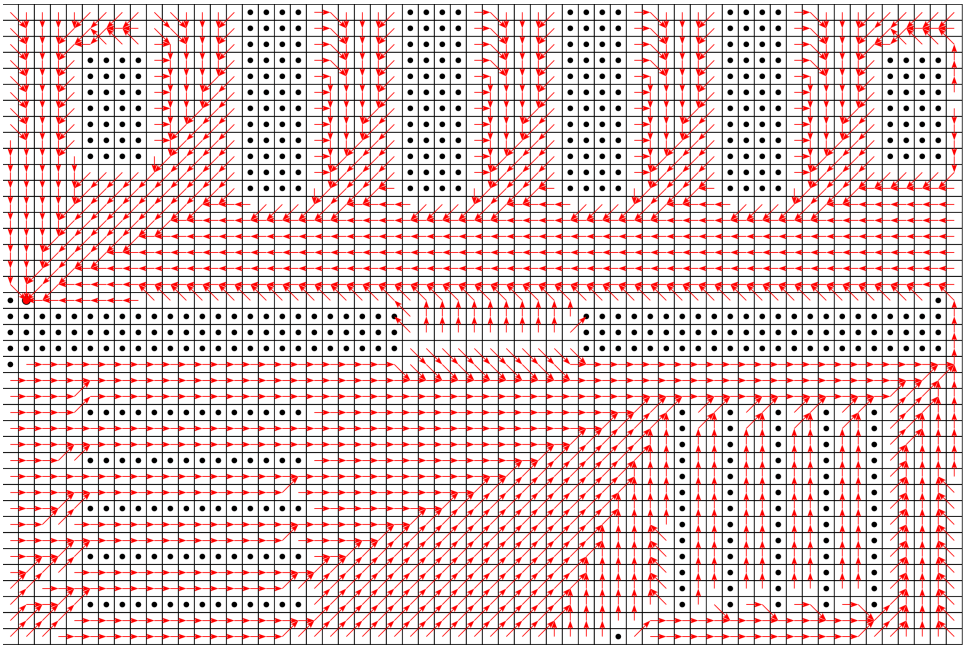

Figure 18: Policy iteration with p = 0.6

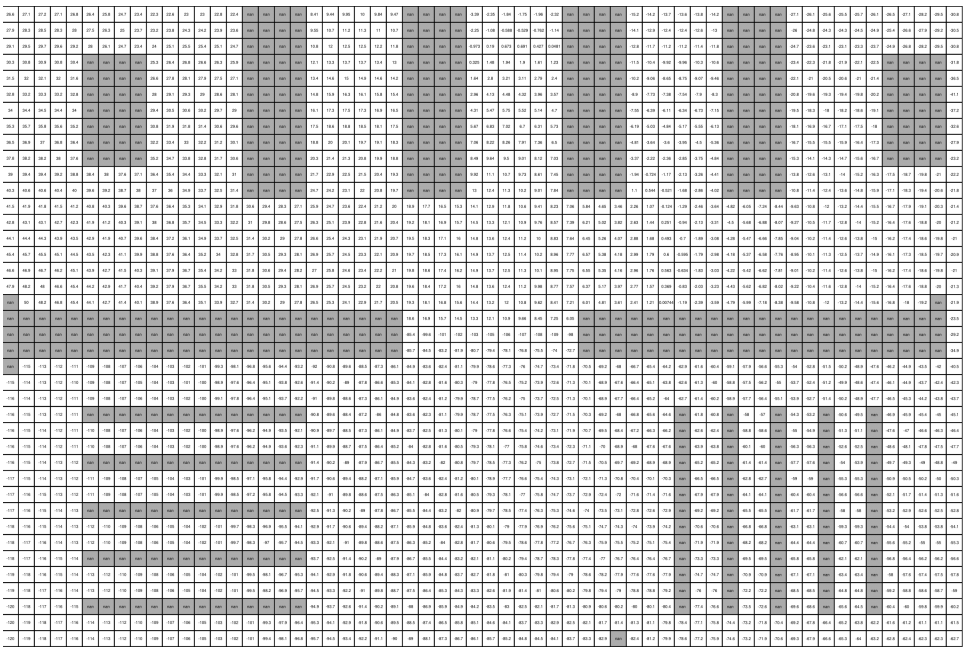

Figure 19: Value Function with p = 0.6

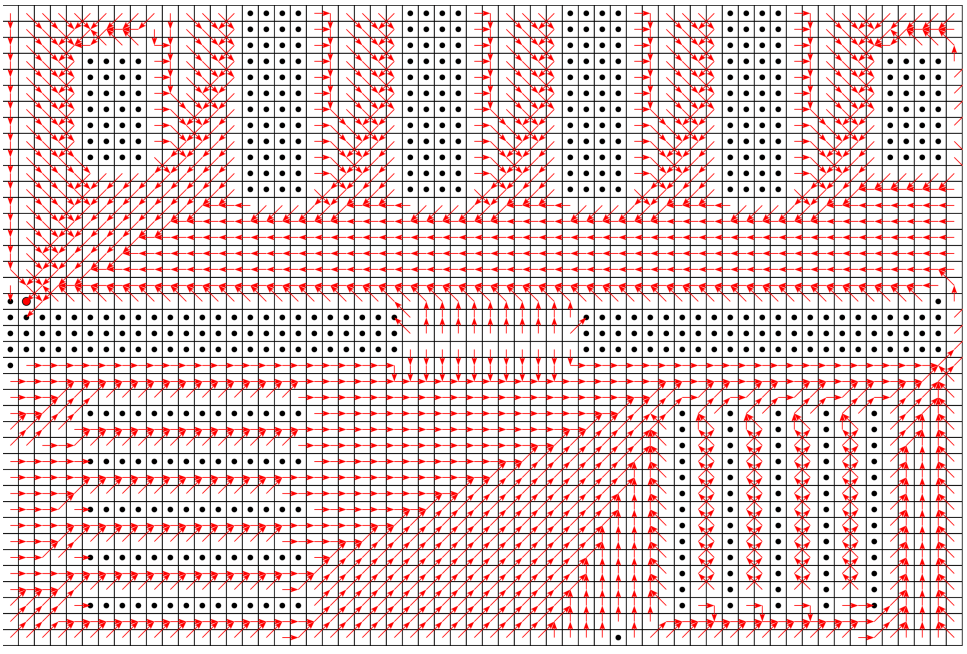

Figure 20: Policy iteration with p = 0.3

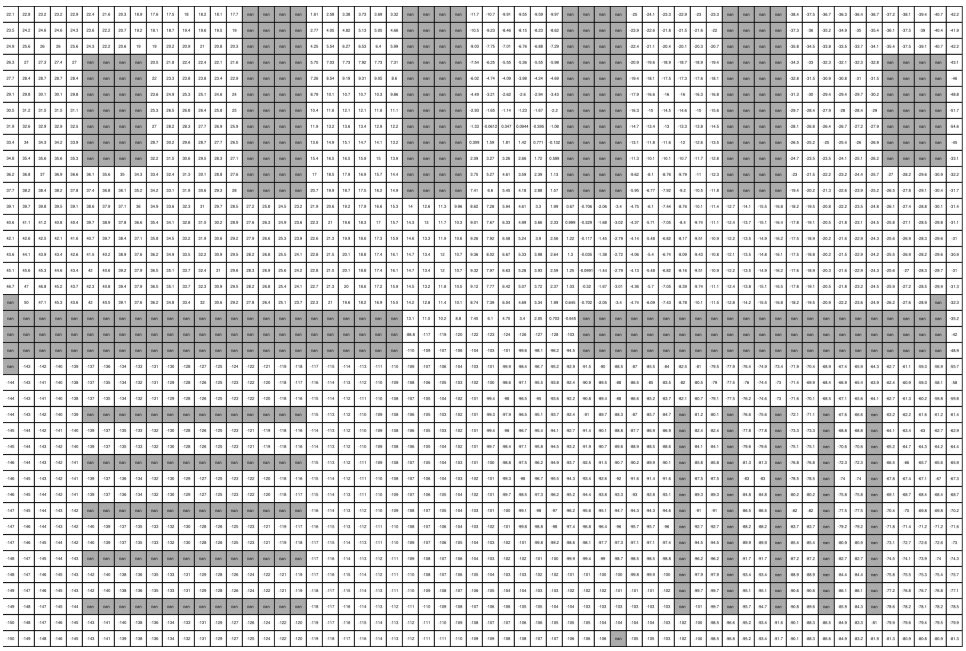

Figure 21: Value Function with p = 0.3

When p=0.6, there is a greater probability of moving in either side of the nominal direction. relating back to the baggage reclaim area example described above, the policy for p=0.6 is moving to the right to escape from the baggage reclaim area to avoid its penalty. This "escaping" effect is exaggerated for p=0.3, where at the baggage reclaim areas, the agent would attempt harder to get further away from the baggage reclaim areas to avoid the penalty. Interestingly, in the middle between baggage reclaim areas, the policy forms several crossings that are moving downwards. This crossing behaviour is also noticed at the bottom right seating areas, and the reason for this because with a 0.7 probability of moving either side of the nominal direction, with crossing policy, the agent would either result in the other side of the cross at the same horizontal level or would successfully move in the intended direction diagonally. Either way, the penalty for hitting the seats or the baggage reclaim area on either side can be avoided.

We have also noticed that there are cases where the agent would attempt to move diagonally into the sides of the environment (see top right corner) to avoid the baggage reclaim area, which is a good strategy as the penalty is only -1 for hitting the edge and it is also rather likely (0.7) that the agent would result in the desired direction, either going up or going down.

(e) Trivially, one could precompute an optimal value function, set that as the initial value function for the policy iteration algorithm, and the evaluator would only run once. But that would defeat the goal of reducing the computational cost required while still obtaining the optimal solution.

There are three parameters that we observed in this algorithm which could have an effect on reducing the number of time that the policy evaluator is ran: Gamma, Maximum policy evaluation steps per iteration (MPESPI), and theta, which we experimented with independently. Gamma and theta are default parameter values in the policy iteration algorithm. MPESPI on the other hand is an integer value to decide the number of iterations allowed for calculating the value function of current policy in policy evaluation, which means that the value function generated on exit of the iteration may not provide the exact value function of the current policy being evaluated.

The experiments are carried out as follows: We first decided on some initial default experiment settings:

- MPESPI = 10
- Gamma = 1
- Theta = 10e-6

For experiments on each of the parameters, only that parameter would be changed, and other parameters will be kept as their default experiment setting values. A wide range of parameter values are searched and for each parameter value, we would conduct 5 repetitions to get the mean of policy evaluation runs:

- MPESPI = 1, $5 \times k$ (k being integer ranging from 1-20)
- Gamma = 0.8 - 1.0 (0.01 intervals)

- Theta = 1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1, 2, 4, 6, 8, 10, 12, 14, 16

Parameter values are set using the corresponding attribute setter The number of times that the policy evaluator ran for each parameter value is counted and stored, and after searching through the range of parameter values, a graph is generated using matplotlib.

Gamma Experiments The following figure shows the results obtained for varying gamma values. As shown in the graph, lower gamma does indeed result in a smaller number of times that the policy evaluator is ranked.

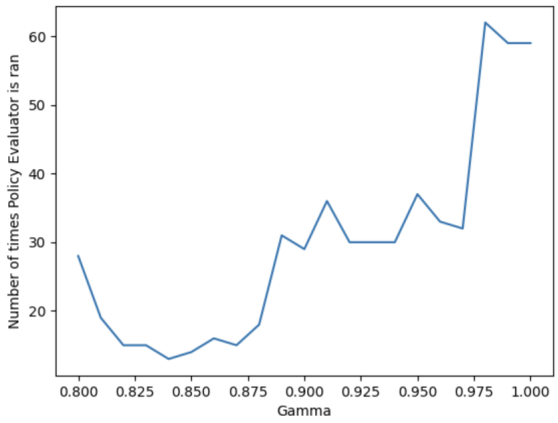

Figure 22: Gamma experiment

However, it is also worth noting that through experimenting, we have noticed that lowering gamma to less than or equal to 0.974 does not guarantee finding the optimal policy as the agent's search becomes bounded by its theta value. The policy generated by these agents will look like the following, which is obviously not "globally optimal" though it is optimal based on its given parameters.

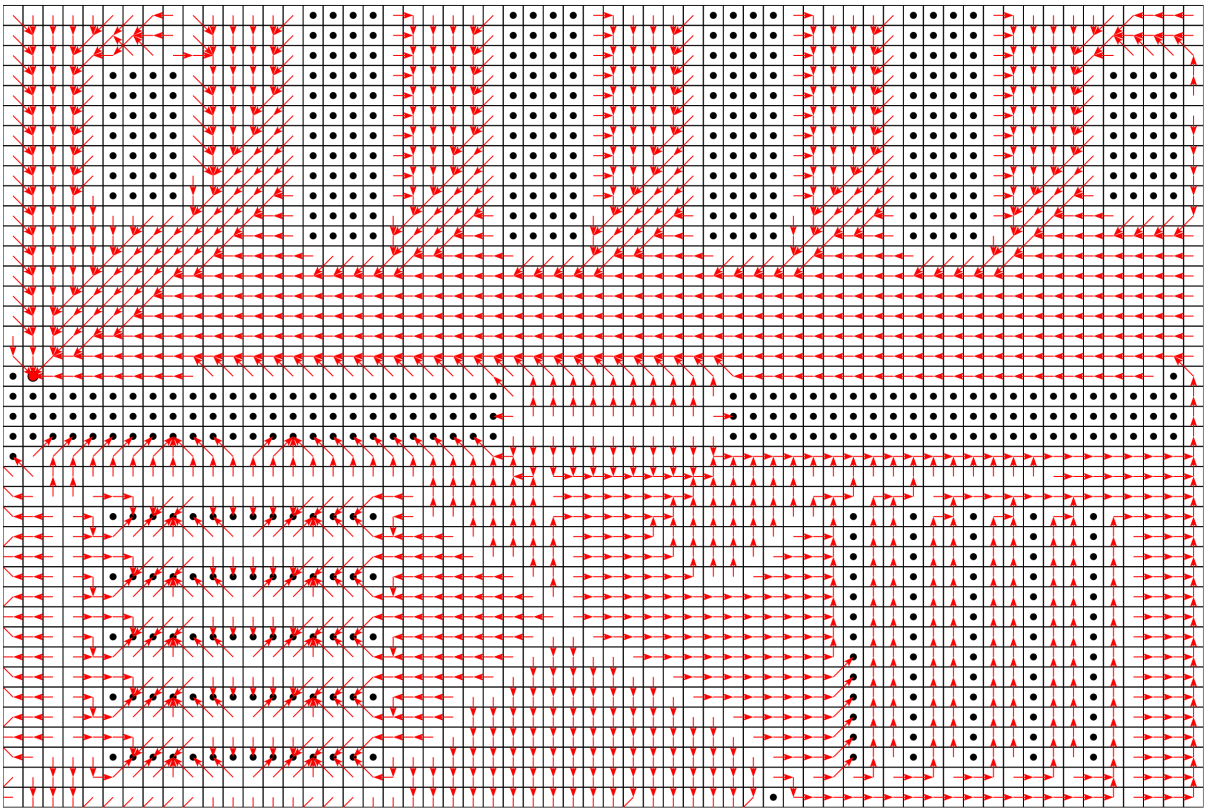

Figure 23: Policy iteration with gamma = 0.975

Although theoretically the policy iteration algorithm would guarantee finding the optimal policy, but that is based on the assumption that theta is set to 0, where in our experimental case, theta is only set to a rather small number. Since this task is episodic, it is not necessary to use a gamma of less than one, also shown by the graph, there is no significant advantage in having a gamma value $0.975 < \gamma < 1$, and it would be better to keep gamma to 1 for finding the optimal policy.

MPESPI Experiments The following figure shows the results obtained for varying MPESPI values. As shown in the graph, MPESPI value of 1 would result in the policy evaluator running for an extremely high number of times. This may be due to the agent taking too little steps (see figure) towards approaching the optimal value function causing a greater number of evaluation-improvement iterations.

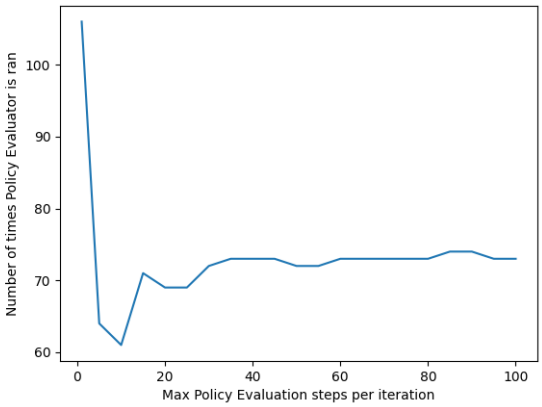

Figure 24: MPESPI experiment

Increasing MPESPI increased the number of times the policy evaluator is ran and plateaus at around 73 times, this may be because as MPESPI increases, the iterations within Policy Evaluation would already terminate before reaching the MPESPI value, hence further increase in MPESPI would not affect the quality of the value function that is evaluated. Conclusively, the algorithm performs optimally when MPESPI is 10 where the number of times the policy evaluator is ran is around 62 times.

Increasing MPESPI increased the number of times the policy evaluator is ran and plateaus at around 73 times, this may be because as MPESPI increases, the iterations within Policy Evaluation would already terminate before reaching the MPESPI value, hence further increase in MPESPI would not affect the quality of the value function that is evaluated. Conclusively, the algorithm performs optimally when MPESPI is 10 where the number of times the policy evaluator is ran is around 62 times.

Theta Experiments The following figure shows the results for varying the theta values. We observed that changing theta at very low values (1e-6 to 1e-2) has no significant effect on number of times the policy evaluator is ran. However, there seems to be a sudden drop between Theta 1-4, with 2 being optimal of having only 56 policy evaluation runs. Why exactly is it 2 is the matter of tuning the parameters, with theta of 2 being the best theta that works along other default parameter values.

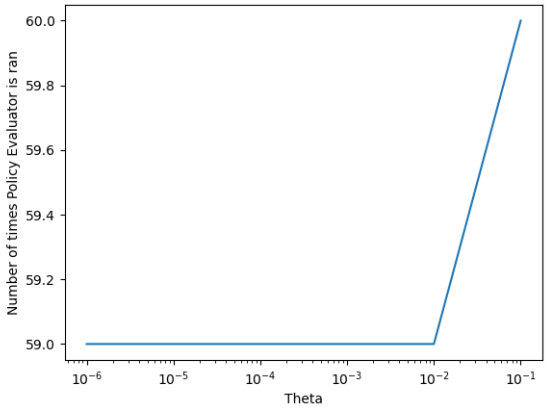

Figure 25: Theta experiment part 1 (X-axis in log scale)

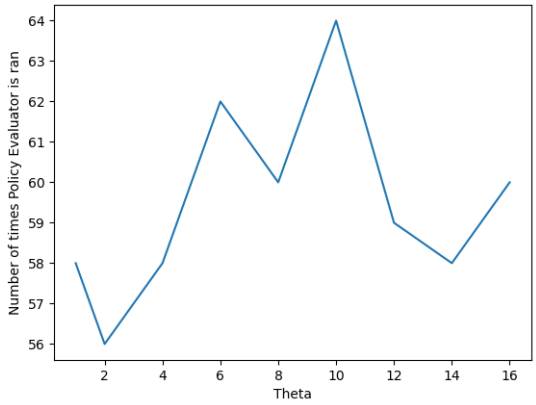

Figure 26: Theta experiment part 2

Due to limitations in computational power, pair wise and triplet parameter searches cannot be achieved, which may further optimise the performance of the algorithm in this scenario. In the current case, we have concluded that the possible optimal parameters to use would be:

- MPESPI = 10
- Gamma = 1
- Theta = 2

(f) There are two parts to the value iteration algorithm, the part to find the optimal value function, and the part to use that optimal value function in order to greedily generate an optimal policy. The optimal value function is found using the `_compute_optimal_value_function` method and the policy is generated by the `_extract_policy` method. Also, a helper `q` method is used to calculate the action values.

`q` method is implemented as follows. The method takes in two parameters, state and action. It first obtains the environment's dynamics of the state and action pair using the

`_environment.next_state_and_reward_distribution(state, action)` method, which returns a list of next states(s'), list of rewards for corresponding rewards and a list of probabilities for transitioning to the corresponding s'. Then the action value of the state action pair is calculated by applying the Bellman equation. Since in this case, the policy is deterministic, no explicit sum over the probability of taking the action is needed.

`_compute_optimal_value_function` is implemented as the following; it consists of a loop which only terminates when the maximal change in values of the value function at the current iteration is less than a pre-set value `self.theta`. For each iteration, there is another iteration that iterates through all states (x,y coordinates pairs) of the airport, and within this inner iteration, there is another loop which iterates

through all available actions at the current state. Since the action space is the same across all states, all states would iterate through the same set of actions. For each action, its action value is calculated using the q method, and at the end of the action iteration, the value of the current state is updated to the highest action value at that state. Also, the maximal difference between the values is noted to be used for checking the stopping condition.

`_extract_policy` is implemented by iterating through all states (x,y coordinates pairs of the airport) and by iterating through the action space and calculated using the `q` method, the method set the action to be selected at that state to the action that gives the highest action value at that state.

An edge case that is worth mentioning is when `self._v.value(x, y)` returns `nan`, which happens when the cell is a wall, baggage claim, chair or toilet. If this happens, then all operations described above wouldn't be carried out, and that state would be ignored.

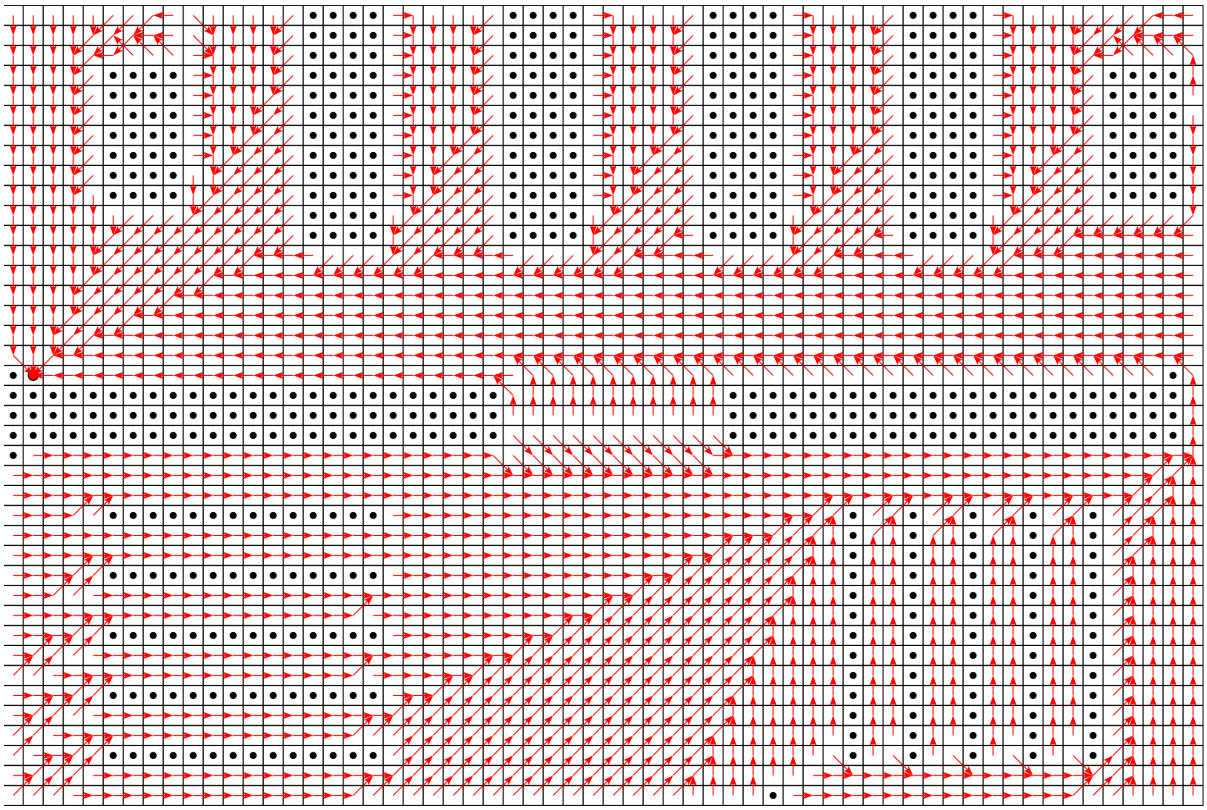

Figure 27: Policy Iteration Policy

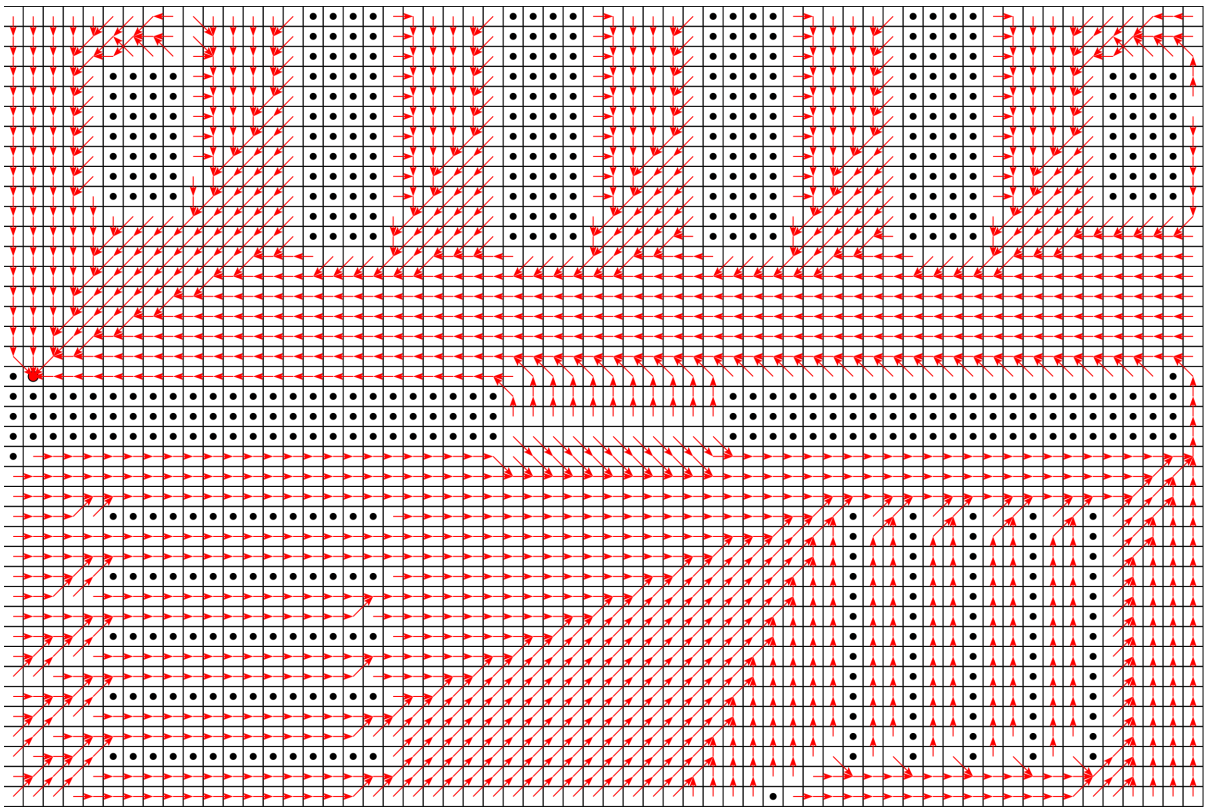

Figure 28: Value Iteration Policy

(g) In terms of policy generated by the two algorithms, as shown by the figures above there is no difference, and both resulted in the same optimal policy, which is expected. But in terms of total number of iterations, the results are the following with both algorithms being tuned to their optimal performing parameters (to our best abilities):

Algorithm	Sum of iterations	Sum of Action iterations
Policy Iteration	4488804	836864
Value Iteration	4844664	1615032

Table 6: Results with total iterations

Surprisingly, Policy iteration has a significantly smaller number of iterations (approx. 400000 iterations less) which means that this algorithm converges quicker to the same optimal policy compared to value iteration. This could be because as suggested by their names, Policy iteration iterates over policies and value iteration iterate over value function. A single policy can represent many, many value functions hence as Policy iteration iterates from one policy to another, it is essentially discarding the process of going through these value functions that represented the same policy, of which value iteration in the worst case might iterate through.

In addition, for policy iteration, the algorithm only iterate through the action space and greedily selects the best action in each state during the policy improvement stage. On the other hand value iteration would need to update the values until convergence to the optimal value function and then generates a greedy policy, where in every update for each state, the algorithm would iterate through that state's action space. In the case where the action space is rather big (10 actions in current scenario), this may have an impact of the speed of convergence. This is shown in the table above with Value iteration having two times the number of iterations through the actions than Policy Iteration, which once again verified the efficiency of Policy Iteration over Value iteration in terms of convergence speed. This result is also supported by many literature such as Sutton and Barto's Reinforcement Learning: An Introduction that Policy iteration in fact converges surprisingly quick.

Conclusively, I would recommend policy iteration in the current scenario as this algorithm took way smaller number of iterations but produced the same optimal policy as value iteration with both algorithms being tuned to their optimal performing parameters to our best abilities.

Question 4

An FMDP for this system would look something like this:

Set of finite states S: Each state is a 4-tuple of x-coordinate, y-coordinate, battery percentage, and cell type. The set of terminal states are the states with x and y coordinates which are the same as the next cleaning site.

Set of actions for each state A: The set of actions from a state is a change in x and y coordinates, which is the same as in 3a.

State transition probabilities P: For any action, a, the change in x and y coordinates is the same as in 3a, where the x and y coordinates have probability p to move in the direction of the action a, and probability 0.5(1-p) to move in an adjacent direction. Battery percentage change is a decrease which is proportional to the Euclidean distance from the change in x and y coordinates.

The only exceptions are when the x and y coordinates of the previous cell type belong to the recharge station or when the resulting battery percentage is 0. In the former case, the resulting battery percentage will always be 100%. In the latter case, it is assumed that the robot will be picked up and moved to the closest recharge station, so the resulting x and y coordinates will be the same as the recharge station, and the battery percentage will be 100%.

Reward function R: The reward function is the same as in 3a, but with some additional components: Euclidean distance with a multiplier and cost to recharge.

The multiplier is inversely proportional to the battery percentage. If the battery is at 100%, the multiplier is at a minimum, and if the battery goes to 0, the multiplier is at a maximum (running out of battery then incurs a massive cost).

The cost to recharge is a Gaussian distribution according to the Gaussian-distributed rewards at each recharge station. This is primarily to reflect the time taken to recharge.

Discount factor γ: The discount factor can be set to 1 to let the return accurately reflect the cumulative sum of rewards.

Value iteration can then be applied as in Sutton and Barto's pseudocode: Initialise the value of all states to 0, and iterate, updating the value of each state using the reward function from the FMDP, until it converges (since the task is episodic, this is guaranteed to converge). Then the deterministic policy π can be obtained by taking the action with the optimal value for each state.