

Data Gathering

D-G

Mr. Muawiya Al Danaf

Date: 6/7/2024

**Contents**

[**1.** **Introduction** 2](#_Toc125641319)

[**2.** **Problem Statement** 3](#_Toc125641320)

[**.3** **Methods and Pseudocode** 5](#_Toc125641322)

[**.4** **Configuration and Parameters** 6](#_Toc125641323)

[**.5** **Evaluation Graphs and Boxplot** 8](#_Toc125641325)

# **Introduction**

In recent years, the integration of drones, or Unmanned Aerial Vehicles (UAVs), in various fields such as agriculture, disaster management, and environmental monitoring has revolutionized data collection processes. These UAVs can cover large areas efficiently and access hard-to-reach locations, making them indispensable in scenarios requiring real-time data collection and transmission. However, one of the significant challenges in deploying UAVs for such tasks is optimizing their communication and data relay strategies to ensure minimal delay and high packet delivery ratios.

Reinforcement Learning (RL), a subset of machine learning, offers promising solutions to optimize decision-making processes in dynamic environments. By employing RL algorithms, UAVs can autonomously learn and adapt their data transmission strategies to improve overall system performance. This project focuses on developing a framework using Q-Learning, a popular RL algorithm, to guide UAVs in a field containing multiple sensors that collect crucial information. Each UAV operates within a predetermined path of waypoints and decides the best recipient for the data at each waypoint, choosing between a Base Station, other UAVs, or no transmission.

# **Problem Statement**

The primary challenge addressed in this project is the optimization of data transmission from UAVs to the intended recipients to minimize communication delay and maximize the packet delivery ratio. In a field densely populated with sensors, UAVs must efficiently collect and relay data to ensure timely and reliable information flow. The problem is multifaceted and involves the following key aspects:

1. **Waypoint Navigation:** Each UAV must navigate through a specific path of waypoints within the field. At each waypoint, the UAV needs to make an informed decision on data transmission.

2. **Recipient Selection:** The UAVs must choose the optimal recipient for the data they carry. The options include sending data to a Base Station, other UAVs, or deciding not to send data at all. The choice impacts overall network performance, including latency and delivery reliability.

3. **Reinforcement Learning Application:** Implementing the Q-Learning algorithm to enable UAVs to learn the best choes to choose the optimal recipient.

4. **Simulation Environment:** Creating a robust simulation environment using Python and libraries like pygame to model the UAVs, their paths, and the decision-making processes.

By addressing these aspects, the project aims to enhance the efficiency and effectiveness of UAV-based data collection systems, ultimately contributing to improved performance in real-world applications.

# **Methods and Pseudocode**

**Algorithm 1: Main**

|  |
| --- |
| **Input:**  1. Command-line arguments  **Output:**  1. Different actions based on the run type argument, such as training Q-learning agents, testing an environment, plotting results, etc.  **Start Algorithm:**  1. **Define `get\_algorithm` function:**  1.1. Input: `name` (string)  1.2. Output: ForwardingAlgorithm instance  1.3. Steps:  1.3.1. If `name` is 'none', return `NoForwarding(env)`  1.3.2. If `name` is 'random', return `RandomForwarding(env)`  1.3.3. If `name` is 'greedy', return `GreedyForwarding(env)`  1.3.4. If `name` is 'dqn', raise `NotImplementedError`  1.3.5. If `name` is 'ql', get Q-learning forwarding agents and return `QLearningForwarding(env, agents=agents)`  1.3.6. Else, raise `NotImplementedError`  2. **Define `get\_args` function:**  2.1. Input: None  2.2. Output: Parsed command-line arguments  2.3. Steps:  2.3.1. Create an argument parser  2.3.2. Add arguments for `run\_type`, `solution`, `id`, `algorithm`, `repeat`, `chunk\_size`, `visualizer`, `load\_from`, `episodes`, `steps`, `with\_log`, `log\_behavior\_freq`, `title`, `grid\_size`, `num\_of\_uavs`, `num\_of\_sensors`, `width`, `height`  2.3.3. Return parsed arguments  3. **Define `get\_ql\_forwarding\_agents` function:**  3.1. Input: `load\_from\_file` (boolean)  3.2. Output: List of QLearningForwardingAgent instances  3.3. Steps:  3.3.1. Initialize an empty agents list  3.3.2. If `load\_from\_file` is True:  3.3.2.1. For each UAV in the environment:  3.3.2.1.1. Get the path to the saved model  3.3.2.1.2. Assert the model exists  3.3.2.1.3. Load the agent from the file and append to the agents list (**Algorithm 3)**  3.3.2.2. Return the agents list  3.3.3. Get the action size  3.3.4. For each UAV in the environment:  3.3.4.1. Calculate `q\_table\_size` based on other UAVs' paths  3.3.4.2. Append (has data) parameter and action size to `q\_table\_size`  3.3.5. For each UAV in the environment:  3.3.5.1. Create `QLearningForwardingAgent` instance with `q\_table\_size` and `action\_size` (**Algorithm 3)**  3.3.5.2. Append the agent to the agents list  3.3.6. Return the agents list  4. **Define `init\_environment` function:**  4.1. Input: None  4.2. Output: Environment instance  4.3. Steps:  4.3.1. Assert `solution` argument is provided  4.3.2. Create a `FileManager` instance (**Algorithm 6**)  4.3.3. Configure logger  4.3.4. Set current timestamp (**Algorithm 2)**  4.3.5. Return loaded environment from file (**Algorithm 2**)  5. **Main logic:**  5.1. Parse arguments using `get\_args`  5.2. If `run\_type` is 'train-dqn':  5.2.1. Raise `NotImplementedError`  5.3. Else if `run\_type` is 'train-ql':  5.3.1. Initialize the environment (**Algorithm 2)**  5.3.2. Get Q-learning forwarding agents (**Algorithm 3)**  5.3.3. Create `QLearningAgentsController` instance (**Algorithm 4)**  5.3.4. Define agent sequence **(it is the revers of the agents order)**  5.3.5. Train Q-learning agents (**Algorithm 4)**  5.4. Else if `run\_type` is 'test':  5.4.1. Initialize the environment  5.4.2. Repeat for specified times:  5.4.2.1. Get forwarding algorithm based on argument  5.4.2.2. Create `EnvironmentController` instance (**Algorithm 5)**  5.4.2.3. Run controller with visualizer  5.4.2.4. Reset environment (**Algorithm 2)**  5.5. Else if `run\_type` is 'plot':  5.5.1. Create `Plotter` instance  5.5.2. Call plot method  5.6. Else if `run\_type` is 'plot-epsilon':  5.6.1. Call `Plotter.plot\_epsilon\_decay` method  5.7. Else if `run\_type` is 'generate':  5.7.1. Create `InputGenerator` instance  5.7.2. Call generate method  **End Algorithm** |

**Algorithm 2:** **Environment**

|  |
| --- |
| **Input:**  1. `land\_width` (float): Width of the land.  2. `land\_height` (float): Height of the land.  3. `uavs` (List of `UAV` objects): List of UAVs in the environment.  4. `sensors` (List of `Sensor` objects): List of sensors in the environment.  5. `base\_stations` (List of `BaseStation` objects): List of base stations in the environment.  6. `run\_until` (int): Number of time steps to run the simulation.  **Output:**  1. Packet delivery ratio (PDR).  2. End-to-end delay.  3. Energy consumption.  4. Performance metrics (PDR, delay).  **Start Algorithm:**  1. **Initialize Environment:**  1.1. Set environment attributes (`land\_width`, `land\_height`, `uavs`, `sensors`, `base\_stations`, `run\_until`).  1.2. Initialize `time\_step` to 0.  1.3. Store a deep copy of the initial state.  2. **Define `get\_sensors\_in\_range` function:**  2.1. Input: `uav` (UAV object)  2.2. Output: List of sensors in range of the UAV  2.3. Steps:  2.3.1. Initialize an empty list `sensors`.  2.3.2. For each sensor in the environment:  2.3.2.1. If the sensor is within the UAV's coverage radius, add it to the `sensors` list.  2.3.3. Return the `sensors` list.  3. **Define `get\_uavs\_in\_range` function:**  3.1. Input: `uav` (UAV object)  3.2. Output: List of UAVs in range of the UAV  3.3. Steps:  3.3.1. Initialize an empty list `uavs`.  3.3.2. For each other UAV in the environment:  3.3.2.1. If the other UAV is not the current UAV and is in range, add it to the `uavs` list.  3.3.3. Return the `uavs` list.  4. **Define `get\_base\_stations\_in\_range` function:**  4.1. Input: `uav` (UAV object)  4.2. Output: List of base stations in range of the UAV  4.3. Steps:  4.3.1. Initialize an empty list `base\_stations`.  4.3.2. For each base station in the environment:  4.3.2.1. If the base station is in range, add it to the `base\_stations` list.  4.3.3. Return the `base\_stations` list.  5. **Define `step` function:**  5.1. Input: None  5.2. Output: None  5.3. Steps:  5.3.1. Increment `time\_step` by 1.  5.3.2. For each UAV in the environment:  5.3.2.1. Reset UAV's `has\_collected` flag to False.  5.3.2.2. Get the UAV's collection point.  5.3.2.3. Initialize `sensors\_in\_range` and `can\_move`.  5.3.2.4. If the UAV has a collection point:  5.3.2.4.1. Get sensors in range of the UAV.  5.3.2.4.2. If there are sensors in range, set `can\_move` to False and collect data.  5.3.2.5. If the UAV has a forwarding target:  5.3.2.5.1. Set `can\_move` to False and forward data.  5.3.2.6. If `can\_move` is True, move the UAV.  5.3.3. For each sensor in the environment, generate data.  6. **Define `calculate\_pdr` function:**  6.1. Input: None  6.2. Output: Packet delivery ratio (float)  6.3. Steps:  6.3.1. Initialize `generated\_packets` to 0.  6.3.2. For each UAV in the initial state, sum the collection rates to `generated\_packets`.  6.3.3. If `generated\_packets` is 0, return 0.  6.3.4. Sum the data packets in all base stations to `arrived\_packets`.  6.3.5. Return the ratio of `arrived\_packets` to `generated\_packets`.  7. **Define `calculate\_end\_to\_end\_delay` function:**  7.1. Input: None  7.2. Output: Average end-to-end delay (float)  7.3. Steps:  7.3.1. Initialize `end\_to\_end\_delay` to 0.  7.3.2. Sum the data packets in all base stations to `arrived\_packets`.  7.3.3. If `arrived\_packets` is 0, return 0.  7.3.4. For each base station:  7.3.4.1. For each packet in the base station's data packets:  7.3.4.1.1. Calculate the delay and add it to `end\_to\_end\_delay` if it is within the packet's lifetime.  7.3.5. Return the ratio of `end\_to\_end\_delay` to `arrived\_packets`.  8. **Define `calculate\_energy\_consumption` function:**  8.1. Input: None  8.2. Output: Total energy consumption (float)  8.3. Steps:  8.3.1. Sum the consumed energy of all UAVs and return it.  9. **Define `get\_performance\_matrices` function:**  9.1. Input: None  9.2. Output: Performance metrics (tuple: PDR, delay)  9.3. Steps:  9.3.1. Calculate PDR using `calculate\_pdr`.  9.3.2. Calculate delay using `calculate\_end\_to\_end\_delay`.  9.3.3. Return PDR and delay.  10. **Define `has\_ended` function:**  10.1. Input: None  10.2. Output: Boolean indicating if the simulation has ended  10.3. Steps:  10.3.1. For each UAV, check if it is done.  10.3.2. If all UAVs are done, return True.  10.3.3. Else, return False.  11. **Define `reset` function:**  11.1. Input: None  11.2. Output: None  11.3. Steps:  11.3.1. Reset `time\_step` to 0.  11.3.2. Reset UAVs, sensors, and base stations to their initial state.  **End Algorithm** |

**Algorithm 3: Q-learning forwarding**

|  |
| --- |
| **Input:**  1. `uav` (UAV object): The UAV to be controlled by the agent.  2. `env` (Environment object): The environment in which the agent operates.  3. `action\_size` (int): The number of possible actions.  4. `q\_table\_size` (tuple): The size of the Q-table.  5. `solution\_id` (int): An identifier for the solution.  6. `gamma` (float, default=0.90): Discount factor for future rewards.  7. `alpha` (float, default=1): (Unused in current code, placeholder for potential future use).  8. `lr` (float, default=0.01): Learning rate.  9. `epsilon` (float, default=0.99): Initial exploration rate.  10. `epsilon\_min` (float, default=0.12): Minimum exploration rate.  11. `epsilon\_decay` (float, default=0.00009): Decay rate for exploration.  **Output:**  1. Updated Q-table.  2. Performance metrics (e.g., episode rewards).  **Start Algorithm:**  **1. Initialize QLearningForwardingAgent:**  1.1. Set agent attributes (`uav`, `env`, `action\_size`, `q\_table\_size`, `solution\_id`, `gamma`, `alpha`, `lr`, `epsilon`, `epsilon\_min`, `epsilon\_decay`).  1.2. Initialize `steps` to 0.  1.3. Initialize `episodes\_rewards` as an empty list.  1.4. Initialize `episode\_return` as an empty list.  1.5. Initialize `q\_table` as an empty dictionary.  1.6. Initialize `log\_enabled` to False.  1.7. Initialize `log` as an empty dictionary.  **2. Define `calculate\_return` function:**  2.1. Input: `target` (UAV/BaseStation), `available\_targets` (List of targets).  2.2. Output: Reward (int).  2.3. Steps:  2.3.1. If `target` is a `BaseStation`, return 100.  2.3.2. If any available target is a `BaseStation`, return -100.  2.3.3. For each available target:  2.3.3.1. If the target is in range of a `BaseStation` and equals `target`, return 10.  2.3.4. If `target` is the agent's UAV, return 0.  2.3.5. Return -10 otherwise.  **3. Define `get\_available\_targets` function:**  3.1. Input: None.  3.2. Output: List of available targets (UAVs and BaseStations).  3.3. Steps:  3.3.1. If the UAV has no data, return an empty list.  3.3.2. Get base stations in range of the UAV.  3.3.3. Return the list of base stations and UAVs in range.  **4. Define `get\_available\_actions` function:**  4.1. Input: None.  4.2. Output: List of available actions, List of forward targets.  4.3. Steps:  4.3.1. Get available targets.  4.3.2. Initialize `actions` list.  4.3.3. For each target in available targets:  4.3.3.1. If the target is a `BaseStation`, append its action index to `actions`.  4.3.3.2. Else, append the UAV's index to `actions`.  4.3.4. Append the agent's UAV index to `actions`.  4.3.5. Return `actions` and available targets.  **5. Define `choose\_action` function:**  5.1. Input: None.  5.2. Output: Chosen action, List of available targets.  5.3. Steps:  5.3.1. Get available actions and targets.  5.3.2. If the maximum available action index is greater than or equal to the number of UAVs, return it.  5.3.3. Else, randomly choose an action from available actions.  **6. Define `choose\_best\_action` function:**  6.1. Input: `state` (current state).  6.2. Output: Best action, List of available targets.  6.3. Steps:  6.3.1. If the state is not in `q\_table`, initialize it with zeros.  6.3.2. Get Q-values for the state.  6.3.3. Get available actions and targets.  6.3.4. Initialize `q\_value` and `action` with a very small number.  6.3.5. For each available action:  6.3.5.1. If the Q-value for the action is greater than `q\_value`, update `q\_value` and `action`.  6.3.6. Return the best action and available targets.  **7. Define `choose\_epsilon\_greedy\_action` function:**  7.1. Input: `state` (current state).  7.2. Output: Chosen action, List of available targets.  7.3. Steps:  7.3.1. If a random number is less than `epsilon` or `time\_step` is less than or equal to 10:  7.3.1.1. Get available actions and targets.  7.3.1.2. Randomly choose an action.  7.3.2. Else, choose the best action using `choose\_best\_action`.  **8. Define `get\_target` function:**  8.1. Input: `action` (chosen action).  8.2. Output: Target (UAV or BaseStation).  8.3. Steps:  8.3.1. If `action` is greater than or equal to the number of UAVs:  8.3.1.1. Return the corresponding `BaseStation`.  8.3.2. Else, return the corresponding UAV.  **9. Define `take\_forwarding\_action` function:**  9.1. Input: `action` (chosen action).  9.2. Output: None.  9.3. Steps:  9.3.1. Get the target using `get\_target`.  9.3.2. If the target is not the agent's UAV, set it as the UAV's forwarding target.  9.3.3. Call `env.step`.  10. **Define `update\_epsilon` function:**  10.1. Input: None.  10.2. Output: None.  10.3. Steps:  10.3.1. If `epsilon` is greater than or equal to `epsilon\_min`, update `epsilon` using decay.  11. **Define `update\_q\_table` function:**  11.1. Input: `state`, `action`, `reward`, `next\_state`.  11.2. Output: None.  11.3. Steps:  11.3.1. If `state` or `next\_state` are not in `q\_table`, initialize them with zeros.  11.3.2. Update the Q-value for the state-action pair using the reward and max Q-value of the next state.  12. **Define `get\_current\_state` function:**  12.1. Input: None.  12.2. Output: Current state.  12.3. Steps:  12.3.1. Get the current point index for each UAV.  12.3.2. Append the data flag for the agent's UAV.  12.3.3. Return the state.  13. **Define `step` function:**  13.1. Input: None.  13.2. Output: None.  13.3. Steps:  13.3.1. Get the current state.  13.3.2. Choose an epsilon-greedy action.  13.3.3. Get the target using the chosen action.  13.3.4. Perform the forwarding action.  13.3.5. Get the next state.  13.3.6. Calculate the reward.  13.3.7. Update the Q-table.  13.3.8. Append the reward to `episode\_return`.  13.3.9. Update `epsilon`.  **End Algorithm** |

**Algorithm 4: QLearningAgentsController**

|  |
| --- |
| **Input:**  1. `id` (int): Identifier for the controller.  2. `forwarding\_agents` (List[QLearningForwardingAgent]): List of Q-learning agents.  3. `env` (Environment): The environment in which the agents operate.  4. `max\_steps` (int): Maximum number of steps per episode.  5. `num\_of\_episodes` (int): Number of training episodes.  6. `solution\_id` (int): Identifier for the solution.  7. `log\_behavior\_freq` (int): Frequency of logging agent behavior.  **Output:**  1. Trained agents with updated Q-tables.  2. Logs of training results and behaviors.  3. Saved model files for each agent.  **Start Algorithm:**  1. **Initialize QLearningAgentsController:**  1.1. Set controller attributes (`id`, `forwarding\_agents`, `env`, `max\_steps`, `num\_of\_episodes`, `solution\_id`, `log\_behavior\_freq`).  2. **Define `train\_agents` function:**  2.1. Input: `agent\_sequence` (List of agent indices to train).  2.2. Output: None.  2.3. Steps:  2.3.1. Initialize an empty list `trained\_agents`.  2.3.2. For each index `idx` in `agent\_sequence`:  2.3.2.1. Select the agent from `forwarding\_agents` using `idx`.  2.3.2.2. Print training message for the selected agent.  2.3.2.3. For each episode in the range of `num\_of\_episodes`:  2.3.2.3.1. Reset the environment.  2.3.2.3.2. For each UAV and corresponding agent:  2.3.2.3.2.1. Reset `steps` to 0.  2.3.2.3.2.2. Reset `episode\_return` to an empty list.  2.3.2.3.2.3. Assign the UAV to the agent.  2.3.2.3.2.4. If the episode number is a multiple of `log\_behavior\_freq`, enable logging.  2.3.2.3.3. Initialize `steps` to 0.  2.3.2.3.4. While the environment has not ended and `steps` is less than or equal to `max\_steps`:  2.3.2.3.4.1. Print progress message.  2.3.2.3.4.2. Increment `steps`.  2.3.2.3.4.3. Perform an environment step.  2.3.2.3.4.4. For each agent:  2.3.2.3.4.4.1. If the agent has collected data:  2.3.2.3.4.4.1.1. If the agent is the selected agent, perform the agent step.  2.3.2.3.4.4.1.2. If the agent is in `trained\_agents`, choose the best action and take it.  2.3.2.3.4.4.1.3. Else, choose an action and take it.  2.3.2.3.5. Append the mean of `episode\_return` to `episodes\_rewards` of the selected agent.  2.3.2.3.6. Set the `forward\_value` of the UAV to the maximum of `episodes\_rewards`.  2.3.2.4. Append the selected agent to `trained\_agents`.  2.3.3. Call `log\_results`.  2.3.4. Call `save\_models`.  **3. Define `save\_models` function:**  3.1. Input: None.  3.2. Output: None.  3.3. Steps:  3.3.1. Get the current working directory.  3.3.2. Create the experiment directory path.  3.3.3. Create the model directory path.  3.3.4. If the model directory does not exist, create it.  3.3.5. For each agent in `forwarding\_agents`:  3.3.5.1. Open a file for the agent in write-binary mode.  3.3.5.2. Dump the agent into the file using pickle.  4. **Define `log\_results` function:**  4.1. Input: None.  4.2. Output: None.  4.3. Steps:  4.3.1. Get the current working directory.  4.3.2. Create the experiment directory path.  4.3.3. If the experiment directory does not exist, create it.  4.3.4. Create the file path for agent rewards.  4.3.5. For each agent in `forwarding\_agents`:  4.3.5.1. Round each reward in `episodes\_rewards` to three decimal places.  4.3.6. Transpose the list of rewards for all agents.  4.3.7. Open the agent rewards file in append mode.  4.3.8. Write the header row with agent identifiers.  4.3.9. For each row of transposed rewards, write the row to the file.  4.3.10. Create the behavior log directory path.  4.3.11. For each agent in `forwarding\_agents`:  4.3.11.1. If the behavior log directory does not exist, create it.  4.3.11.2. For each episode and log in the agent's log:  4.3.11.2.1. Create the file path for the episode log.  4.3.11.2.2. Open the episode log file in append mode.  4.3.11.2.3. Write the header row for the log.  4.3.11.2.4. For each row in the log, write the row to the file.  **End Algorithm** |

**Algorithm 5: EnvironmentController**

|  |
| --- |
| **Input:**  1. `env` (Environment): The environment in which the UAVs operate.  2. `forwarding\_algorithm` (ForwardingAlgorithm): The algorithm used to determine the forwarding target for each UAV.  **Output:**  1. Log file containing the performance metrics (Packet Delivery Ratio and delay) for the given solution.  **Start Algorithm:**  1. **Initialize `EnvironmentController`:**  1.1. Set controller attributes (`env`, `forwarding\_algorithm`).  **2. Define `control` function:**  2.1. Input: None.  2.2. Output: None.  2.3. Steps:  2.3.1. For each UAV in `env.uavs`:  2.3.1.1. Set `uav.forward\_target` to the result of `forwarding\_algorithm(uav)`.  **3. Define `log\_results` function:**  3.1. Input: `solution\_id` (str): Identifier for the solution.  3.2. Output: None.  3.3. Steps:  3.3.1. Get performance metrics (`pdr`, `delay`) from the environment.  3.3.2. Get the current working directory.  3.3.3. Create the file path for the performance log.  3.3.4. Open the performance log file in append mode.  3.3.5. Write the performance metrics (`pdr`, `delay`) to the file.  **4. Define `get\_visualizer` function:**  4.1. Input: `visualizer` (str): The type of visualizer to use.  4.2. Output: An instance of the specified visualizer.  4.3. Steps:  4.3.1. If `visualizer` is 'pygame':  4.3.1.1. Return an instance of `PygamePresenter`.  4.3.2. Log that no visualizer is used.  4.3.3. Return an instance of `EnvironmentPresenter`.  **5. Define `run` function:**  5.1. Input: `visualizer` (str): The type of visualizer to use. `solution\_id` (str): Identifier for the solution.  5.2. Output: None.  5.3. Steps:  5.3.1. Get the visualizer by calling `get\_visualizer` with the `visualizer` argument.  5.3.2. While the environment has not ended:  5.3.2.1. Capture events using the visualizer.  5.3.2.2. Render the environment using the visualizer.  5.3.2.3. Perform an environment step.  5.3.2.4. Call the `control` function to set forwarding targets for the UAVs.  5.3.3. Call `log\_results` with the `solution\_id` to log the performance metrics.  **End Algorithm** |

**Algorithm 6: FileManager**

|  |
| --- |
| **Input:**  1. `solution\_id` (int): Identifier for the solution to load the data from.  **Output:**  1. Environment object initialized with the UAVs, sensors, and base stations loaded from the corresponding CSV files.  **Start Algorithm:**  **1. Initialize `FileManager`:**  1.1. Set the `solution\_id` attribute.  1.2. Set `input\_dir` to the path of the input directory for the given solution.  **2. Define `read\_data\_frame` function:**  2.1. Input: `name` (str): Name of the CSV file to read.  2.2. Output: A Pandas DataFrame containing the data from the CSV file.  2.3. Steps:  2.3.1. If `name` does not end with '.csv', append '.csv' to `name`.  2.3.2. Read the CSV file from `input\_dir` using Pandas and return the DataFrame.  **3. Define `load\_basic\_variables` function:**  3.1. Input: None.  3.2. Output: A tuple containing `width`, `height`, and `run\_until` values for the environment.  3.3. Steps:  3.3.1. Read the `environment\_basics` DataFrame.  3.3.2. Ensure the DataFrame contains only one row.  3.3.3. Extract `width`, `height`, and `run\_until` from the row and return as a tuple.  **4. Define `load\_sensors` function:**  4.1. Input: None.  4.2. Output: A list of `Sensor` objects.  4.3. Steps:  4.3.1. Read the `sensors` DataFrame.  4.3.2. For each row in the DataFrame:  4.3.2.1. Create a `Vector` object for the sensor position.  4.3.2.2. Create a `Sensor` object with the appropriate attributes.  4.3.2.3. Generate initial data for the sensor.  4.3.2.4. Append the `Sensor` object to the `sensors` list.  4.3.3. Return the `sensors` list.  **5. Define `load\_base\_stations` function:**  5.1. Input: None.  5.2. Output: A list of `BaseStation` objects.  5.3. Steps:  5.3.1. Read the `base\_stations` DataFrame.  5.3.2. For each row in the DataFrame:  5.3.2.1. Create a `Vector` object for the base station position.  5.3.2.2. Create a `BaseStation` object with the appropriate attributes.  5.3.2.3. Append the `BaseStation` object to the `base\_stations` list.  5.3.4. Return the `base\_stations` list.  **6. Define `load\_uavs` function:**  6.1. Input: None.  6.2. Output: A list of `UAV` objects.  6.3. Steps:  6.3.1. Read the `uavs` DataFrame.  6.3.2. Read the `way\_points` DataFrame.  6.3.3. For each row in the `uavs` DataFrame:  6.3.3.1. Create a `Vector` object for the UAV position.  6.3.3.2. Create a `UAV` object with the appropriate attributes.  6.3.3.3. Append the `UAV` object to the `uavs` list.  6.3.4. For each row in the `way\_points` DataFrame:  6.3.4.1. Create a `Vector` object for the way point position.  6.3.4.2. Find the corresponding UAV by ID.  6.3.4.3. Append the way point position to the UAV's path.  6.3.4.4. Append the collection rate to the UAV's collection rates.  6.3.5. Return the `uavs` list.  **7. Define `load\_environment` function:**  7.1. Input: None.  7.2. Output: An `Environment` object.  7.3. Steps:  7.3.1. Call `load\_basic\_variables` to get `height`, `width`, and `run\_until`.  7.3.2. Call `load\_uavs` to get the list of UAVs.  7.3.3. Call `load\_sensors` to get the list of sensors.  7.3.4. Call `load\_base\_stations` to get the list of base stations.  7.3.5. Create and return an `Environment` object with the loaded UAVs, sensors, and base stations, and the basic variables.  **End Algorithm** |

# **Configuration and Parameters**

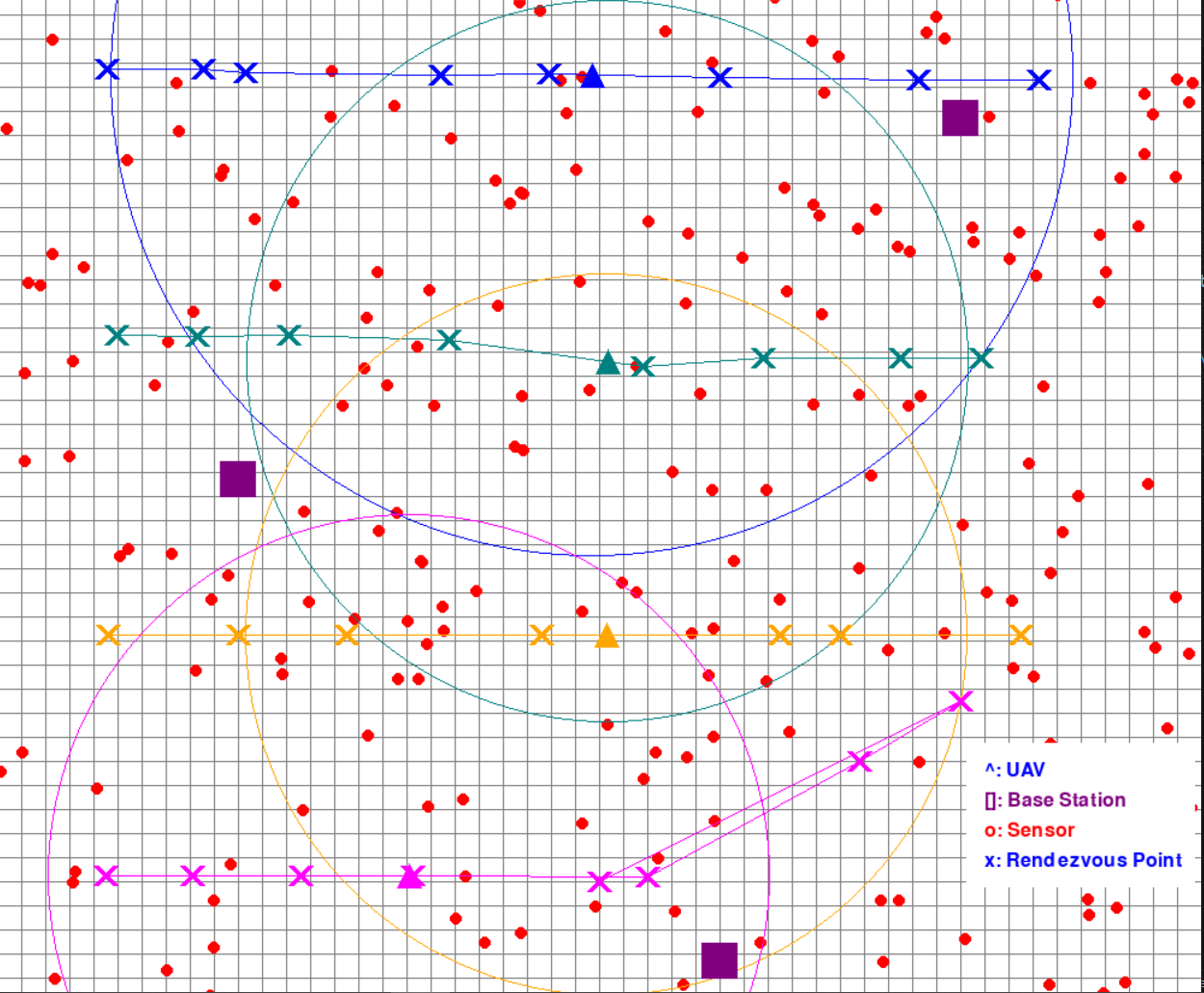
**Experiment 1:**

|  |  |
| --- | --- |
| Parameters Name | Parameters Value |
| RL Algorithm name | Q-Learning |
| Number of Agents(UAVs) | 4 |
| Number of Base Station | 3 |
| The packet life time for each base station | 1000 |
| The Size of the Grid(Field) | 1000 × 1000 |
| Number of the Sensors in the field | 250 |
| The Coverage radius for each UAV | 400, 300, 300, 300 |
| Initial data packets for each Sensor | 20 |
| Sampling rate for each Sensor | 3 |
| Capacity for each Sensor | 200 |
| Collection rate for each Sensor | 20 |
| Number of way points for each UAV | 8, 8, 7, 8 |
| Number of Episodes | 300 |
| Number of steps during each episodes | 100 |
| Gama | 0.90 |
| Learning rate | 0.01 |
| Epsilon | 0.99 |
| Epsilon min | 0.12 |
| Epsilon decay | 0.00009 |
| State Size | 3 |
| Action Size | Number of UAVs × Number of base stations + 1 |

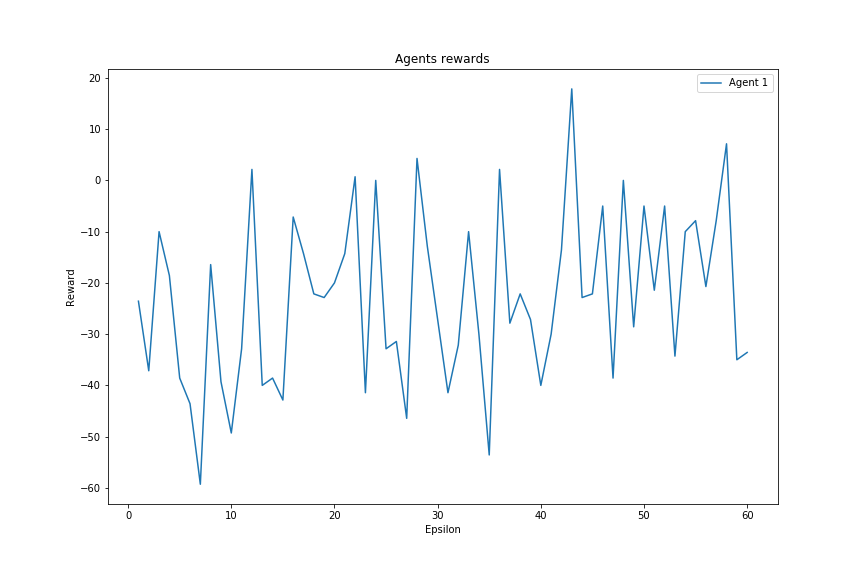
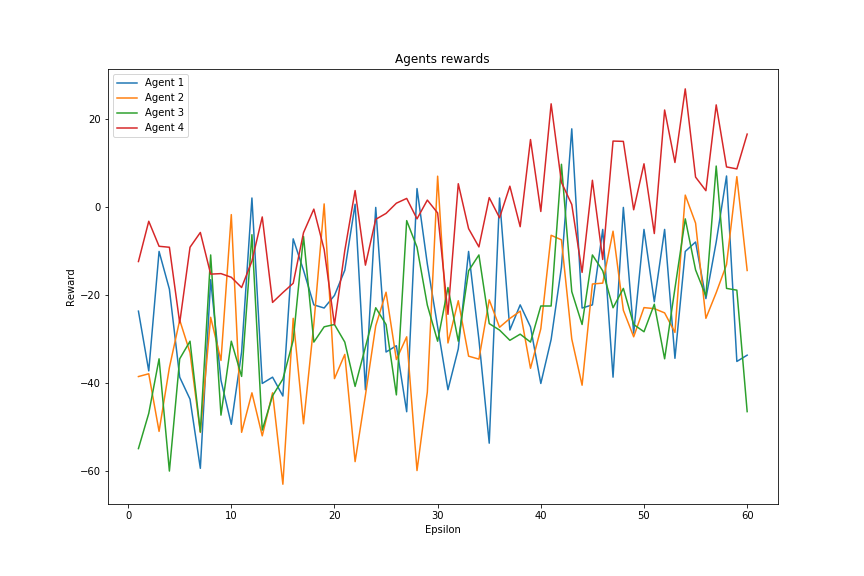
# **Evaluation Graphs and Boxplot**

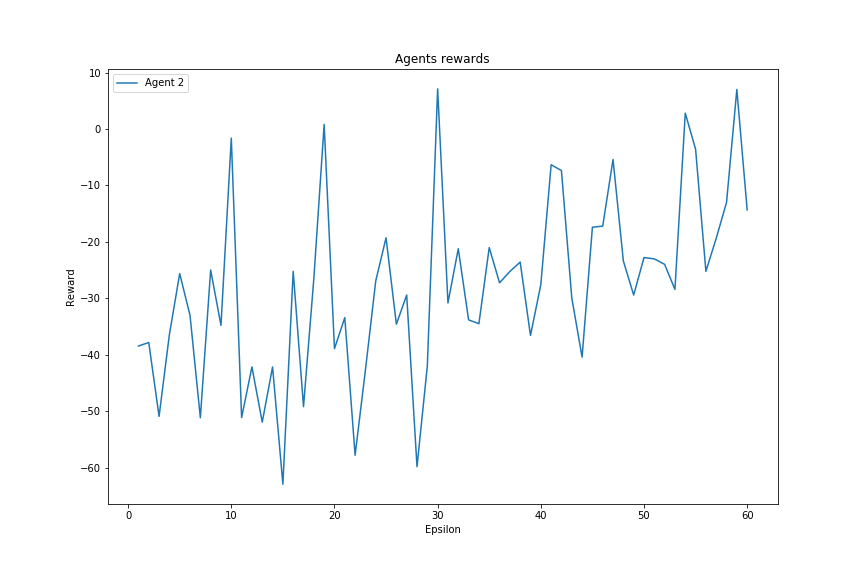
**Experiment 1:**

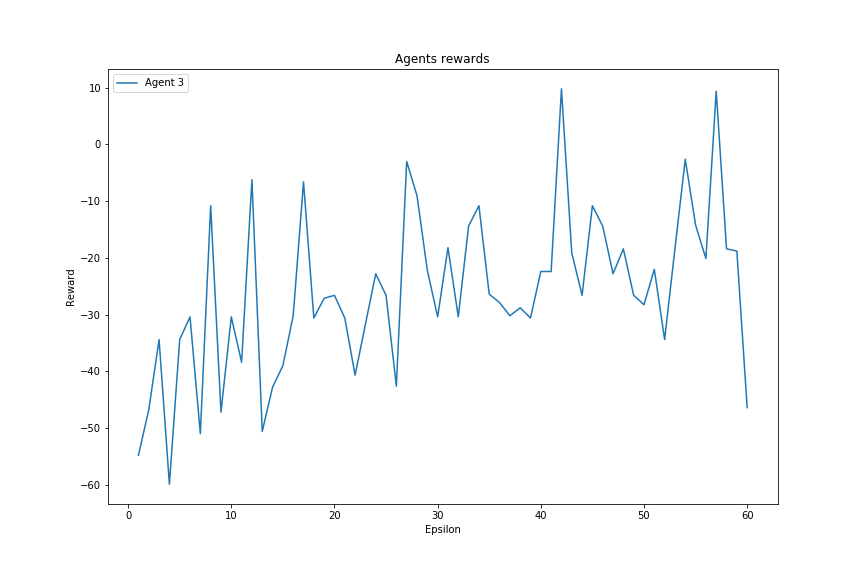
**The Environment (A snapshot):**

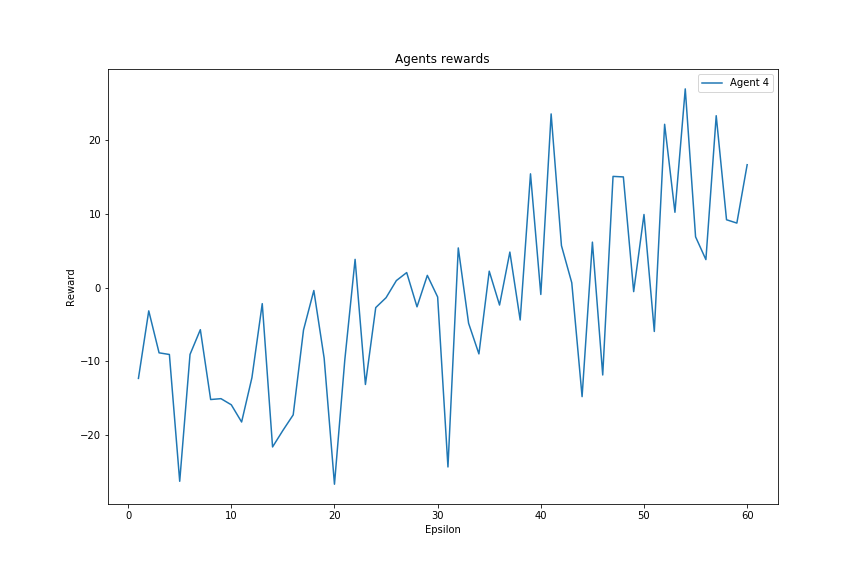
****

**The reward conversion for the agents:**

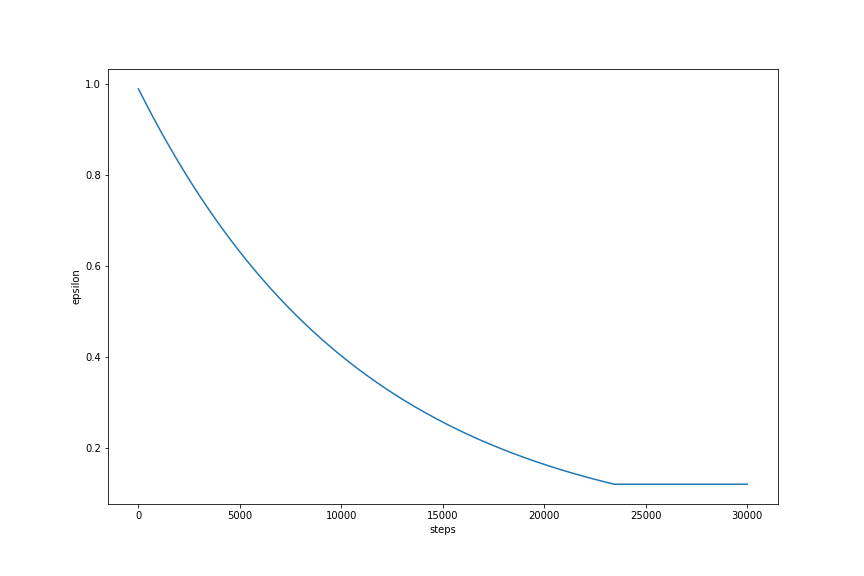
****







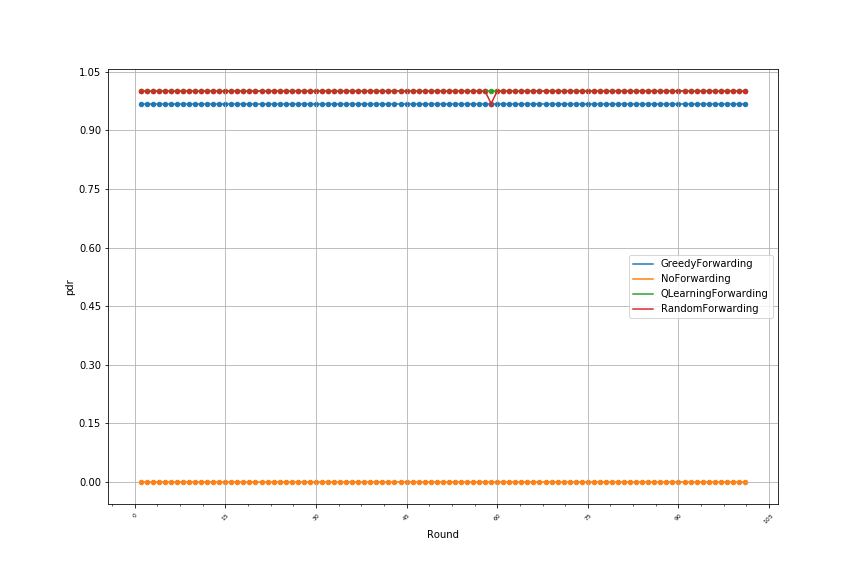
**The Epsilon:**

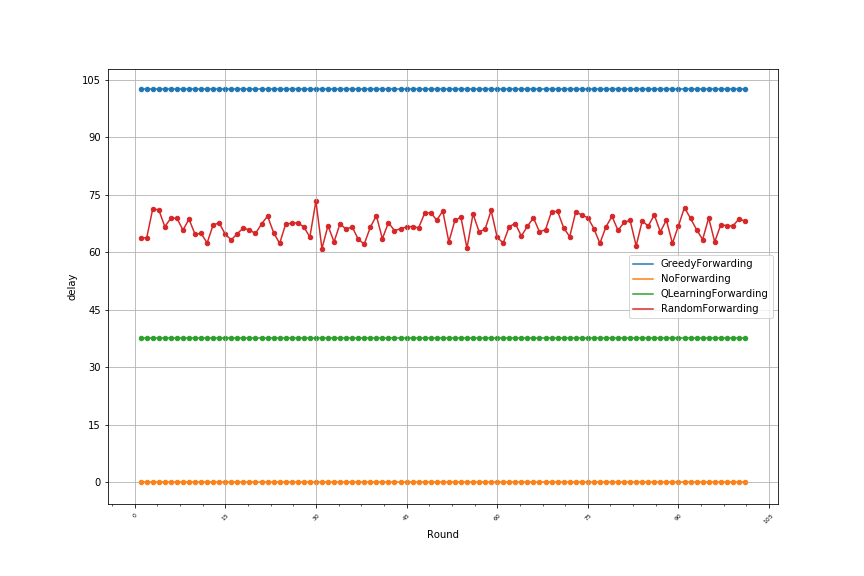


**Networking performance:**

Comparing the reinforcement learning algorithm (Q-Learning) performance in routing with a random routing algorithm, a greedy routing algorithm, and no routing in the term of packet delivery ratio and dilay.

***Hint: round mean running the simulation until the UAVs done their paths***





**The Conclusion:**

The previous results show the superiority of QL over the rest of the algorithms in terms of delay, while in terms of packet delivery rate, it outperforms everyone except the random algorithm, where they are almost equivalent.