Evaluating Fixed Communication Strategies for Multi-Agent Search-and-Rescue Efficiency

Name: Jacob Abraham Palakunnathu

Student ID: 20700939

Department of Computer Science, University of Nottingham, Nottingham, UK

Email: psxjp9@nottingham.ac.uk

Github: https://github.com/iamjacob97/Multi-Agent-SAR-RL.git

Abstract

Autonomous agents operate in a simulated disaster zone, each with limited line-of-sight sensing and online reinforcement learning. Mission time is constrained, so communication is introduced to reduce search overlap and accelerate victim recovery. Four protocols are examined: (1) **No-Communication** (benchmark), (2) **Periodic Communication** - transmission at fixed intervals, (3) **Selective Communication** - transmission restricted to new discoveries, rescues or after extended periods without updates, and (4) **Continuous Communication** - Transmission in real time. Performance is assessed by total rescue time, mean victims recovered, redundancy, and communication load (bytes sent). Experiments map the efficiency–cost frontier and isolate the protocol that maximizes rescue performance per communicated byte.

Through this coursework I am trying to answer the question: "What is the optimal balance of communication to maximize efficiency while minimizing redundant searches?"

1. Introduction and Related Works

Reinforcement learning (RL) is a real-time, trial-and-error approach to optimal control that approximates solutions from dynamic programming (**Bellman, 1957**). It frames decision-making problems as a discrete-time, finite-state Markov Decision Process (MDP), where each state-action pair yields an immediate reward and transitions the agent to a new state (**Van Otterlo & Wiering, 2012**). The goal of the agent is to discover a policy that maximizes the expected sum of discounted future rewards over time.

In contrast to classical search algorithms such as A* or Dijkstra's, reinforcement learning does not require prior knowledge of the entire environment. Instead, it builds optimal behaviours through repeated interactions, making it particularly well-suited for dynamic or partially observable settings. This property is especially advantageous in search-and-rescue (SAR)

scenarios where agents operate with limited information and must adapt to changing environments.

In this study, I use a tabular, one-step Q-learning approach, a model-free algorithm that iteratively updates a table of action-value estimates, known as the Q-table. The standard Q-learning update rule is:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \times \max_{a'} Q(s',a') - Q(s,a)]$$

Where:

- $Q(s,a)$: Current estimate of the expected reward for taking action a in state s
- α: Learning rate (how much new experiences affect the current estimate)
- r: Reward received after taking action α
- γ: Discount factor for future rewards (between 0 and 1)
- s′: New state resulting from action α
- $\max_{a'} Q(s',a')$: Maximum estimated value for the next state across all possible actions

The Q-learning update rule is a stochastic approximation of the Bellman optimality equation, which defines the optimal action-value $Q^*(s,a)$ as the expected immediate reward for taking action a in state s, plus the discounted maximum Q-value of the next state s', under the assumption of optimal future behaviour. Since the agent typically lacks prior knowledge of the environment's dynamics or the true Q-values, Q-learning approximates this relationship iteratively using sampled experience. This process enables the agent to incrementally refine its value estimates, effectively balancing short-term rewards with long-term potential, and gradually converging toward an optimal policy.

Watkins & Dayan (1992) formalized Q-learning as an off-policy algorithm, proving that under the right conditions (sufficient exploration and proper learning rate decay) it converges to the optimal policy. Their framework underpins the reliability of Q-learning in both theoretical and applied settings.

While Q-learning traditionally assumes a stationary environment (as in single-agent settings), **Tan (1993)** explored its use in multi-agent systems, where each agent's learning alters the environment's dynamics. He showed that:

- This non-stationarity can be treated as part of the environment,
- Agents can still converge to useful policies through trial and error,
- Cooperative strategies like shared episodes and policy averaging enhance learning speed and coordination.

Tan's work illustrates that although theoretical convergence is not guaranteed in MARL, Q-learning can still perform well empirically, especially when cooperation is built into the system design.

A key advantage of Q-learning lies in its simplicity and interpretability, making it effective for analysing agent behaviour and decision rationale. As shown by **Mahadevan & Connell (1991**),

decentralized behaviour-based Q-learning is effective in robotic domains requiring spatial navigation, obstacle avoidance, and task recovery. In this coursework, Q-learning enables agents to act based on local observations, while selectively integrating shared information from other agents, simulating various communication strategies in a SAR context.

2. Environment and Agent Design

To investigate the impact of communication strategies on search efficiency and redundancy, I designed a grid-based disaster zone simulation in which multiple autonomous agents operate within a partially observable environment. The environment is structured as a finite 10x10 2D grid where the core world state is stored in NumPy arrays, but a few Python containers round out the bookkeeping. Each cell may be empty, contain an obstacle, a target (victim), or an agent. Obstacles are randomly placed to simulate structural debris, and targets are scattered throughout the grid, mimicking real-world unpredictability in disaster scenarios.

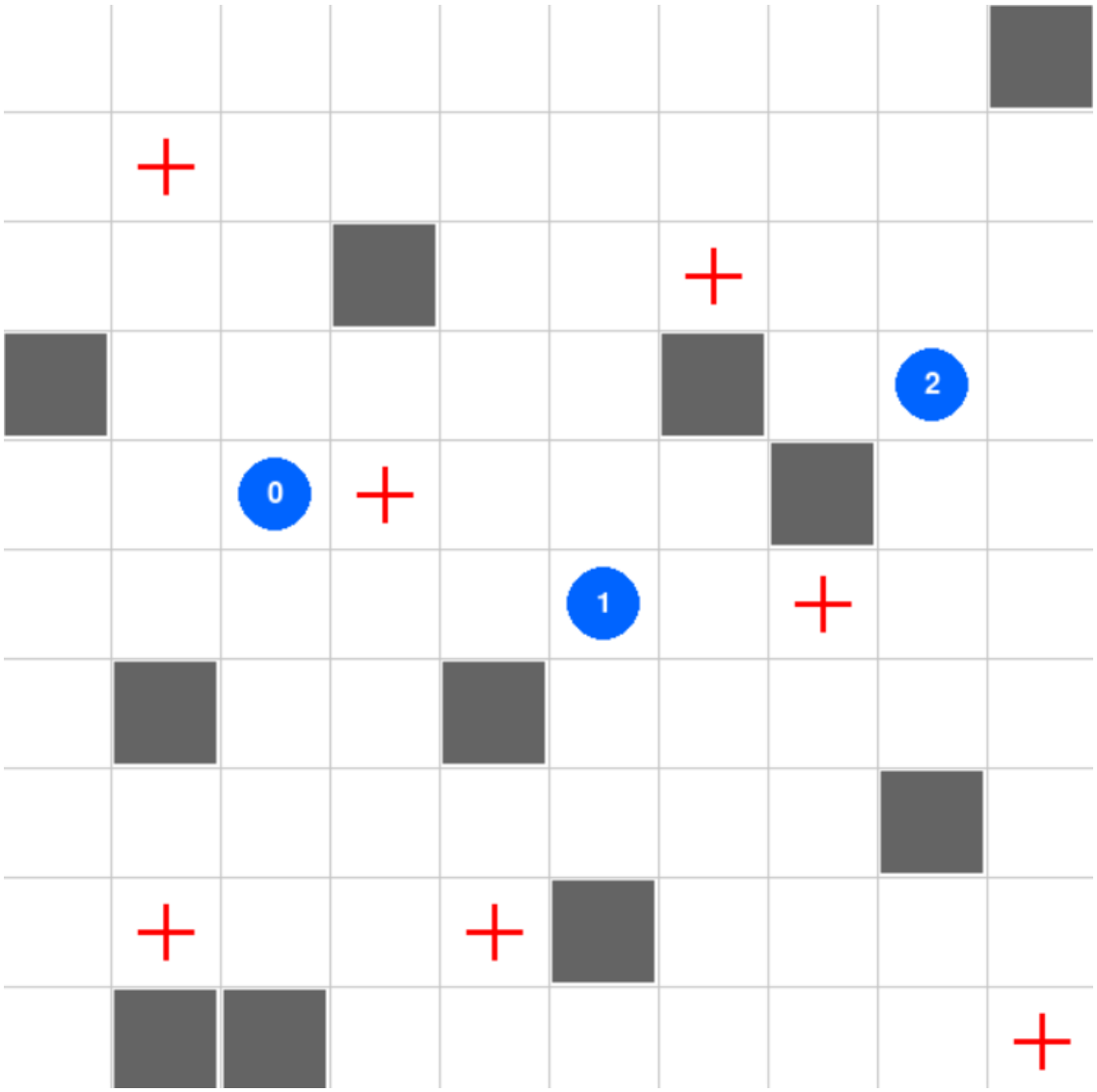

Figure 1: 10 × 10 grid (SAR Visualiser)

Each agent is modelled as an independent Q-learning entity, operating under limited local perception, typically a small fixed-radius view of adjacent grid cells. This constraint ensures that agents cannot see the entire environment and must rely on exploration and, when permitted, communication with teammates.

To systematically evaluate communication strategies, four agent types were implemented:

1. **Independent Agents** - no communication allowed.
2. **Limited Communication Agents** - share information at fixed time intervals (5 steps).
3. **Selective Communication Agents** - communicate only for new rescues/discoveries (also if they haven't communicated in a long time (11 steps)).
4. **Full Communication Agents** - continuously broadcast all discoveries to teammates.

All communicative agents send the same type of information, differing only in the frequency of the transmissions. Each agent maintains a Q-table indexed by local state features and actions and updates it using standard tabular Q-learning. Rewards are given for exploration and rescuing victims, penalizing redundant searches or revisiting explored areas. This is the current reward structure:

1. *Base Movement Reward*:

- -0.1 for each step taken (small penalty to encourage efficiency)

2. *Victim-Related Rewards*:

- +0.17 for moving closer to a known victim
- +10.0 for rescuing a victim

3. *Exploration Rewards*:

- +0.37 for discovering a completely new cell (not visited by any agent)
- -0.23 for revisiting a cell that the agent has already visited
- -0.1 * min(visits, 3) for visiting a cell that has been globally visited(scales with number of visits, up to -0.3)

4. *Communication Costs*:

- -lambda_cost * len(payload) for each message sent (where lambda_cost = 0.01).

5. *Invalid Move Penalty*:

- -1.7 for attempting to move into an obstacle or out of bounds

The environment and agents were custom-built from the ground up and are structured to follow the standard reinforcement learning framework, as illustrated in Figure 2.

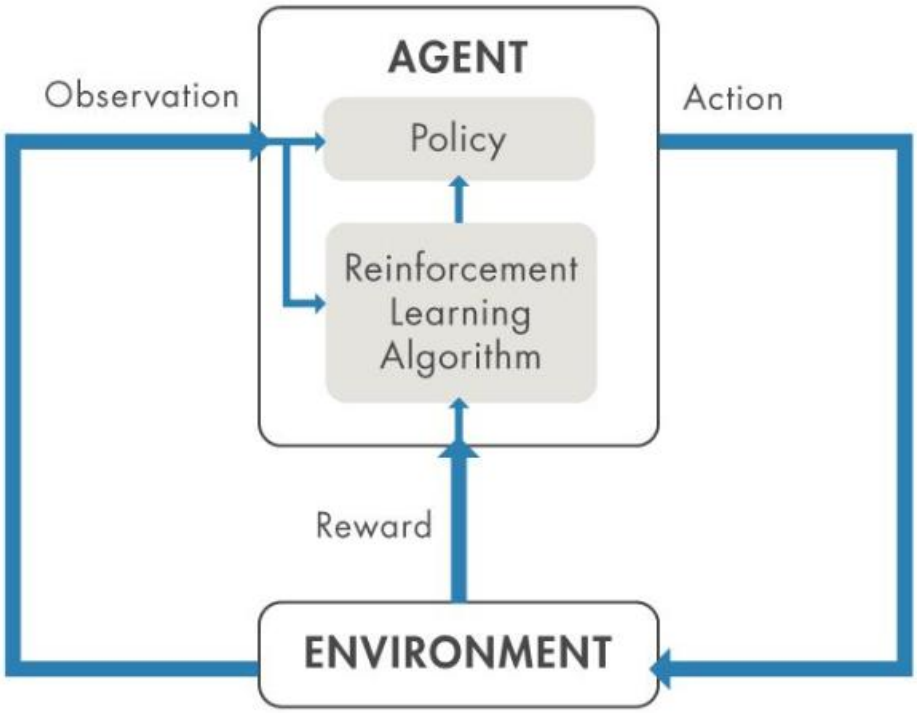

Figure 2: Reinforcement Learning Overview Diagram (Source: mathworks.com)

Each agent faces the classic reinforcement learning trade-off between exploration **(**searching unfamiliar areas to discover victims) and exploitation (revisiting known high-reward paths based on learned experience). To balance this, agents use the ε-greedy action selection strategy. At each decision point, an agent selects a random action with probability ε (exploration), and with probability 1−ε, it chooses the action with the highest estimated Q-value (exploitation).

$$a_t = \begin{cases} rand\{0,1,2,3\}, & \text{w.p. } \varepsilon_t \\ \text{argmax}_a Q(s_t, a), & \text{w.p. } (1-\varepsilon_t) \end{cases}$$

where ε_t can be found by the equation:

$$\varepsilon_t = \max(\varepsilon_{min},\ \varepsilon_0 \,.\, \delta_\varepsilon^t), \qquad \varepsilon_0 = 0.97,\ \varepsilon_{min} = 0.05$$

The factor δ_ε is chosen so that ε reaches its floor after 77% of the total episodes:

$$\delta_\varepsilon = \left(\frac{\varepsilon_{min}}{\varepsilon_0}\right)^{1/\mathrm{N}_{decay}} = \left(\frac{\varepsilon_{min}}{\varepsilon_0}\right)^{1/0.77\mathrm{N}_{total}}$$

ε is initialised with a high value to encourage exploration in the early stages of learning and is gradually decayed over time to favour exploitation as the agent becomes more knowledgeable about the environment.

α, the learning rate determines how much new information (temporal difference (TD) errors) overrides old estimates. A higher α allows the agent to adapt quickly to new experiences but may cause instability, while a lower α results in slower but more stable learning. In the coursework, the α value is linearly decayed from an initial value of 0.73 to a floor value of 0.05, with the decay occurring predictably over the first 77% of the total training episodes. α_t can be found by the equation:

$$\alpha_t = \max(\alpha_{min},\ \alpha_0 - t \times \frac{\alpha_0 - \alpha_{min}}{N_{decay}})$$

with $\alpha_0 = 0.73$, $\alpha_{min} = 0.05$ and $N_{decay} = 0.77N_{total}$

Both exploration (ε) and learning (α) slow together, letting the agent settle into its learned policy smoothly.

The discount factor γ controls the weight of future rewards relative to immediate ones. A value close to 1 encourages long-term planning, which is crucial in multi-agent search-and-rescue, where actions may only yield rewards after extended exploration or coordinated effort. Conversely, a lower γ would prioritize short-term gains, which help agents learn more quickly by reinforcing actions that lead to immediate positive outcomes.

Maintaining γ at 0.95 throughout ensures that both immediate shaping/penalties and delayed rescue rewards are balanced consistently, while the decays of α and ε orchestrate a smooth transition from broad exploration to stable exploitation. This configuration proved effective across all four communication protocols, isolating communication as the primary variable in our efficiency study.

3. Technologies Used and Challenges Faced

I used a very basic tech stack; these are my dependencies:

- **Python**: core language for environment, agents, and scripts
- **NumPy**: core array and mask operations
- **Pandas:** Dataframe manipulation, CSV logging and result aggregation
- **Matplotlib & Seaborn**: plotting learning curves, bar charts, and Pareto scatter plots
- **Scipy**: Statistical Testing
- **Multiprocessing**: Parallel Processing
- **Pygame**: Grid Rendering
- **Git**: Version Control

I had quite a few implementation challenges, with these being the most notable ones,

1. **State Representation Complexity/ State Space Explosion**
   At the start I tried to give each agent all the information it could possibly need:

   - *Absolute position*:
     (row, col) ∈ $\{0, \dots, 9\}^2 = 100$ possibilities

   - *Complete local grid (3×3 patch):*
     $\{0, 1, 2, 3\}^2 = 4^9 \approx 262{,}000$ patterns

   - *Known-victims:*
     Up to 7 victims, each could lie in any of 100 cells = 700 slots

- *Visited mask (3×3)*
  A bit per cell = $2^9 = 512$ patterns

That's $100 \times 262{,}000 \times 700 \times 512 \approx 9 \times 10^{12}$ distinct states which is clearly intractable for tabular Q-learning. Careful refactoring and reiterations helped me boil the observations down to exactly the features needed for decision-making, keeping the key both compact and Markov.

Feature	Encoding & Size
Position	(row,col) $\in \{0, \dots, 9\}^2$ [100 positions]
Directional beacon	(dx,dy) $\in \{0, \dots, 8\}^2 + (9, 9)$ [$9^2 + 1 = 82$ possibilities]
Obstacle mask	9-bit int = up to 512
Visited mask	9-bit int = up to 512

Table 1: Refined 6-Tuple State Key: (*row, col, dx, dy, obstacle_mask, visited_mask*)

The directional beacon encodes the relative direction of the nearest discovered victim as a compact integer pair, enabling agents to make informed movement decisions despite their limited local perception. To construct this feature, the Manhattan distance to each known victim is computed, and the offset to the closest one is clipped to the range [−4, +4] to constrain the state space. These values are then mapped to [0, 8] range by adding 4. If no victims are known, a sentinel value of 9 is assigned to both axes. Obstacle mask and visited mask are similar, where their respective 3×3 binary grids are flattened and converted from 9-bit binary to decimal, ensuring that all spatial features are compactly and consistently represented within the Q-table.

Total reachable states are $100 \times 82 \times 512 \times 512 \approx 2 \times 10^9$. Obstacle positions were fixed per random seed, reducing the number of unique obstacle-mask configurations to approximately 85. Combined with around 120 observed distinct visited-mask patterns, this results in an estimated state space of $100 \times 82 \times 85 \times 120 \approx 83 \times 10^6$ possible state-action pairs. However, due to effective epsilon decay and the natural sparsity of exploration, the actual Q-table size in practice plateaued at around 230,000 unique states. Empirical convergence tests showed stable Q-value updates over time, indicating that the learning process successfully converged within this reduced operational state space (More on this later). It is also important to note that the action space was reduced

from five actions (which included a "stay" action) to the current four-directional movement actions. This reduction significantly decreased the number of Q-values the agent needs to learn and update, thereby improving learning efficiency and convergence speed. In rare cases where two agents attempt to move into the same cell, updates are processed sequentially: the first agent claims the cell, while the second receives a penalty for an invalid move. Since such collisions occur infrequently, their impact is negligible over the course of hundreds of thousands of episodes and does not introduce significant bias into the learned policies.

2. **Maintaining the Markov property**
   An MDP requires that the probability of the next state and reward depends only on the current state and action, not on any earlier history.

$$P(S_{t+1}, R_t \mid S_t, A_t)$$

   This condition is essential for the correctness of tabular Q-learning. Adding duplicate-visit penalties based on a global counter risked hidden history. I kept the environment fully Markov by having the environment compute all penalties and shaping before handing the agent its local 6-tuple. Agents never store trajectory history in the Q-key itself. Every time I made changes to the state key encoding, I had to be careful not to break the Markov property.

3. **Reward Shaping**
   To preserve convergence guarantees, reward shaping must not alter the optimal policy. Initial reward structure I designed provided sparse rewards which were insufficient to guide early exploration. However, naively adding dense shaping rewards resulted in agents over-optimizing local movement metrics without focusing on the actual rescue objective. Balancing the two reward sources required iterative adjustment of shaping weights.

4. **Reproducibility across Agents and Seeds**
   To enable a fair comparison between different communication strategies, it was essential to maintain identical environment configurations across all runs and agent types. Without strict control over the seeds across the entire simulation stack, including obstacle placement, victim distribution, and agent spawn positions, the results could exhibit significant variance, with certain agent types appearing to perform better simply due to favourable or “lucky” environment configurations, rather than superior policy effectiveness. This variability risked obscuring the true effects of communication on agent performance. To address this, each environment instance initializes its own seeded random number generator using:

```
self.rng = np.random.default_rng(seed)
```

This guarantees deterministic sampling and ensures that, for a given seed, the same sequence of random numbers, and thus the same initial conditions are reproduced. Each agent type was evaluated over 9 distinct seeds, with each seed running 357,951 episodes of 33 steps. As a result, all environment elements remained consistent across all episodes for a given seed, allowing for controlled, reproducible comparisons between agent policies.

5. **Training times for huge batch runs**
   Training reinforcement learning agents over hundreds of thousands of episodes, especially across multiple agent types and random seeds introduced substantial computational overhead. With each experiment involving 357,951 episodes per seed, across 9 seeds and 4 communication strategies, the total number of training runs exceeded 12 million episodes. Executing this sequentially was infeasible within practical time constraints on my ten-year-old Ryzen 5, GTX 1650 laptop configurations. Initial runs exposed several bottlenecks:

   - Long wall-clock times for single-agent training loops.
   - Resource underutilization on multi-core systems.

   To overcome this, we adopted Python's multiprocessing module to parallelize training across CPU cores. Each experiment (defined by a unique combination of seed and agent type) was run as an independent process. This approach provided:

   - *Core-level parallelism*: Leveraging multiple CPU cores reduced total wall time drastically.
   - *Seed-level isolation*: Since each process had its own environment, RNG, and agent instance, reproducibility and state separation were preserved.
   - *Modular scaling*: The batch runner could queue and execute dozens of experiments in parallel, with dynamic throttling based on core availability.

   Each process logged training results independently, and a master thread aggregated results at the end of execution. This infrastructure enabled us to train all agent types across all seeds within a tractable timeframe, facilitating statistical comparisons between communication strategies.

4. Experimental Setup

The core of the experiments follows the standard reinforcement learning (RL) paradigm. The environment and agents are first initialised and at the beginning of each episode, they are reset. The environment provides the agents with their initial observations via the _get_observation() method. Each episode runs within a while loop, which continues until either all victims have been rescued, or a predefined maximum number of steps (33) is reached.

Within each step of the loop, every agent selects its next action using the act() method, which implements the ε-greedy policy for balancing exploration and exploitation. The collective actions are then passed to the environment's step() method, which executes the actions, updates and processes all messages (information sent by agents) in its inbox, updates the environment state, and computes rewards accordingly.

The step() function returns four key components:

- The next list of agent observations,
- A list of rewards,
- A done flag indicating episode termination,
- An info dictionary containing auxiliary data.

This output is used both for logging key metrics (e.g., cumulative reward, number of victims rescued, steps taken, map coverage, bytes sent, duplicate visits, and Q-table size) and for updating each agent's Q-values via temporal difference learning using the update() method. The newly returned observations then become the current observations for the next iteration of the loop.

This cycle continues until the episode terminates, after which results are logged, and a new episode begins under the same environment seed until the total count (357,951 episodes per seed) is reached.

Before running the main experiments, I ensured that all functionalities of the different agent types were working as expected. A smoke test was first conducted using the following configuration:

- GRID_SIZE = (5, 5)
- NUM_AGENTS = 2
- NUM_VICTIMS = 3
- NUM_OBSTACLES = 5
- MAX_STEPS = 17
- SEED = 27
- NUM_EPISODES = 10000

The smoke test is designed to quickly check if anything 'catches fire' - if the environment and all agent types are functioning correctly. It is not meant for full-scale training or evaluation, but rather for sanity checking the codebase and agent-environment interactions. The smoke test revealed the following results:

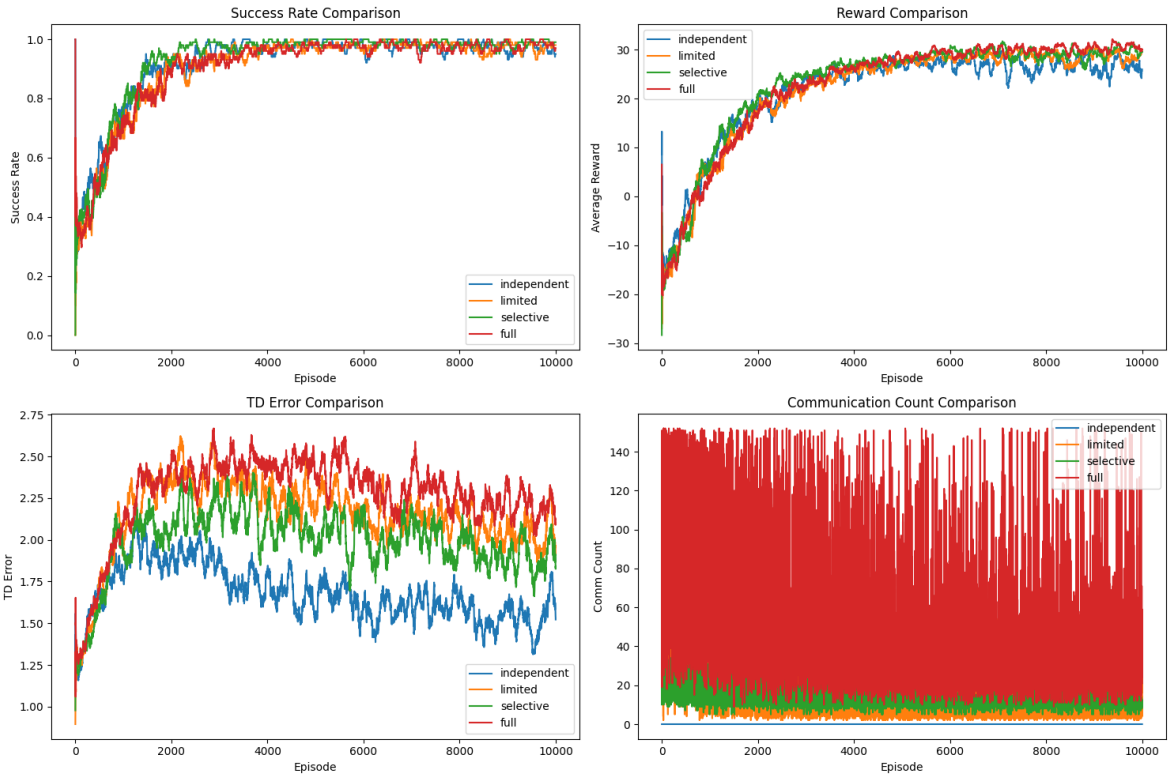

Figure 3: Smoke test results

The plots in Figure 3 show that success rates and rewards steadily improve over episodes, indicating it's move towards convergence, but TD error trends are highly noisy and unstable, likely due to the low episode count and agents frequently encountering new states. This noise makes it difficult to confirm convergence. To address this, a convergence test was performed to more reliably assess whether the agents' policies were stabilizing over time (see next paragraph). Communication counts align with expectations for each strategy, confirming that message handling works as intended. Overall, the results demonstrate that the agents are learning effectively, the environment dynamics are consistent, and the reward and communication mechanisms are operating correctly.

Note: smoke_test.py was later refactored into a streamlined testing script designed to provide real-time feedback during training

The convergence test is designed to evaluate how quickly and effectively the agents learn over time to reach a stable policy. It tracks key metrics like reward, victims rescued, coverage, Q table size and TD error to determine if the learning process is converging. Convergence test is conducted using the original experimental configurations to assess the effectiveness of the learning setup. The test is conducted over one fixed seed, with the default parameters:

- NUM_AGENTS = 3
- NUM_VICTIMS = 7
- NUM_OBSTACLES = 11
- GRID_SIZE = (10, 10)
- TOTAL_EPISODES = 357951

- MAX_STEPS = 33
- SEED = 42

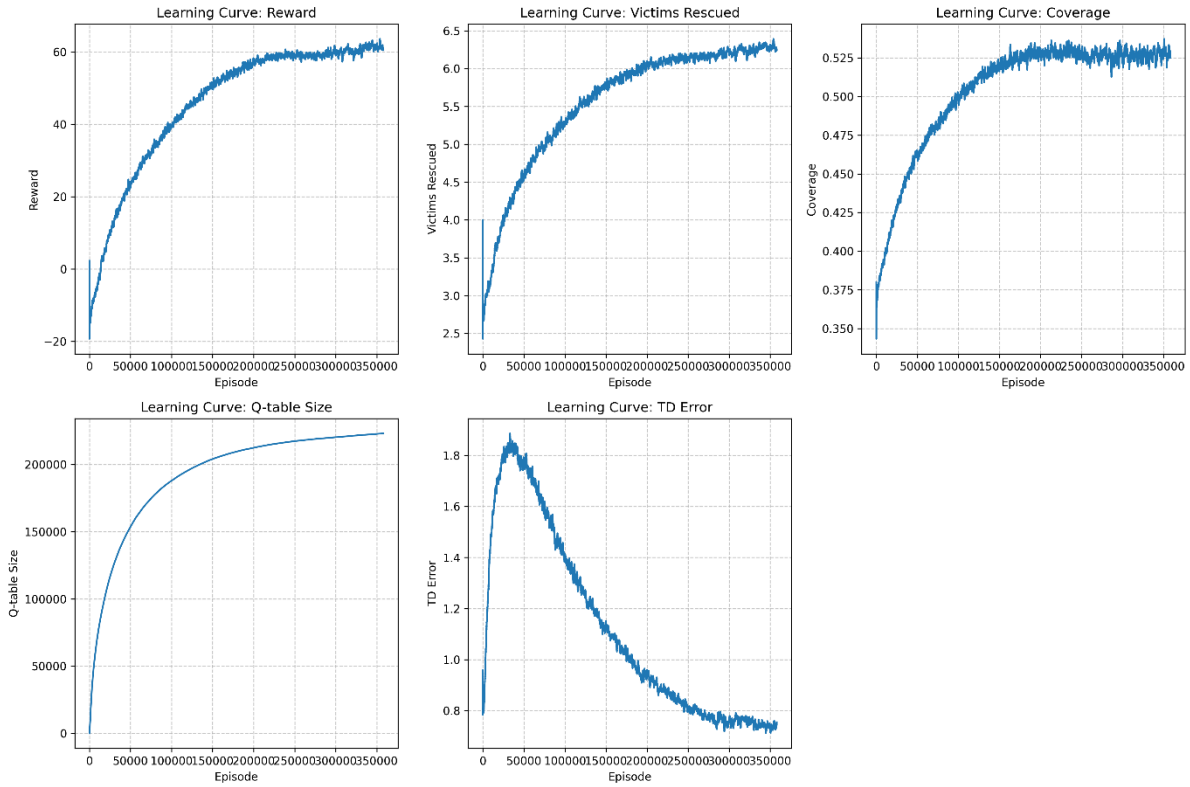

Figure 4: Convergence test results

Over 357,951 episodes, key metrics such as reward, victims rescued, and coverage show clear upward trends followed by long plateaus, suggesting that the agent has learned a reliable policy. The Q-table size initially grows rapidly and then levels off, indicating that the agent is no longer frequently encountering new state-action pairs, a hallmark of convergence in tabular learning. Most importantly, the TD error peaks early and steadily decrease over time, eventually stabilizing at a low value. This decline confirms that the temporal difference updates are shrinking, meaning the agent's value estimates are settling. Together, these trends validate that the agent has converged to a stable (close to optimal) policy under the tested configuration.

The main experiment uses the same configuration as the convergence test but is run across 9 distinct random seeds {1,3,5,7,9,11,13,15,17} to ensure statistical robustness and account for variability in environment layouts and agent initialization. All key parameters were carefully tuned through multiple trial-and-error iterations to ensure convergence and meaningful performance differentiation across communication strategies. For instance, max_steps was set to 33, allowing agents to explore only 99/100 total positions on the map without redundancy, creating time constraints. Training and evaluation were handled separately to ensure unbiased performance assessment. During training, agents operated under an ε-greedy policy with decaying exploration, allowing them to balance exploration and exploitation while updating Q-values. Metrics such as rewards, victim rescues, and Q-table growth were tracked continuously. In contrast, evaluation was conducted every 1,000 episodes using a fully greedy policy (ε = 0),

meaning agents exploited their learned policies without any exploration. Each evaluation phase ran for 100 episodes and was used strictly for benchmarking (Q-tables were not updated during this phase). This separation ensured that performance metrics reflected stable policy behaviour and not transient exploration noise. All training and evaluation data were logged to CSV files, and automated plots were generated to visualize learning curves, communication efficiency, and performance trade-offs. Statistical comparisons between strategies were made using paired t-tests and effect size measures, with results summarized in tables. Visualizations were produced using Matplotlib and Seaborn. Additionally, each agent-type seed pair runs parallelly in separate processes for efficiency.

5. Results

The main batch run revealed the following results:

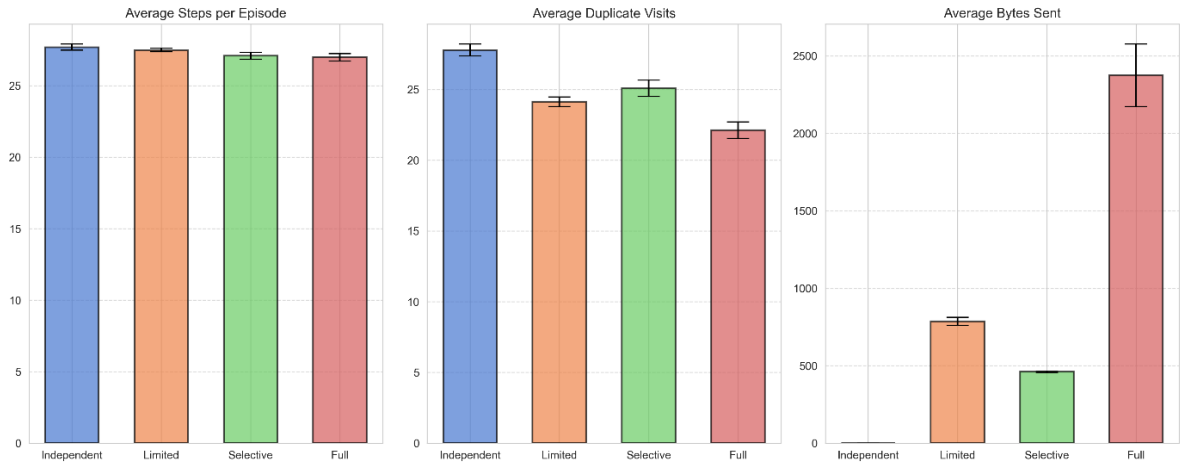

Figure 5: Efficiency bars – Rescue Time. Redundancy, Bytes Sent

Strategy	Reward	Steps	Duplicates	Bytes	Coverage	Success Rate
Independent	61.57	27.69	27.78	0.00	0.53	0.53
Limited	64.20	27.49	24.13	786.97	0.56	0.56
Selective	63.00	27.09	25.08	460.24	0.54	0.58
Full	67.78	26.98	22.12	2375.10	0.58	0.61

Table 2: Mean Performance Metrics by Strategy (±SEM omitted for clarity)

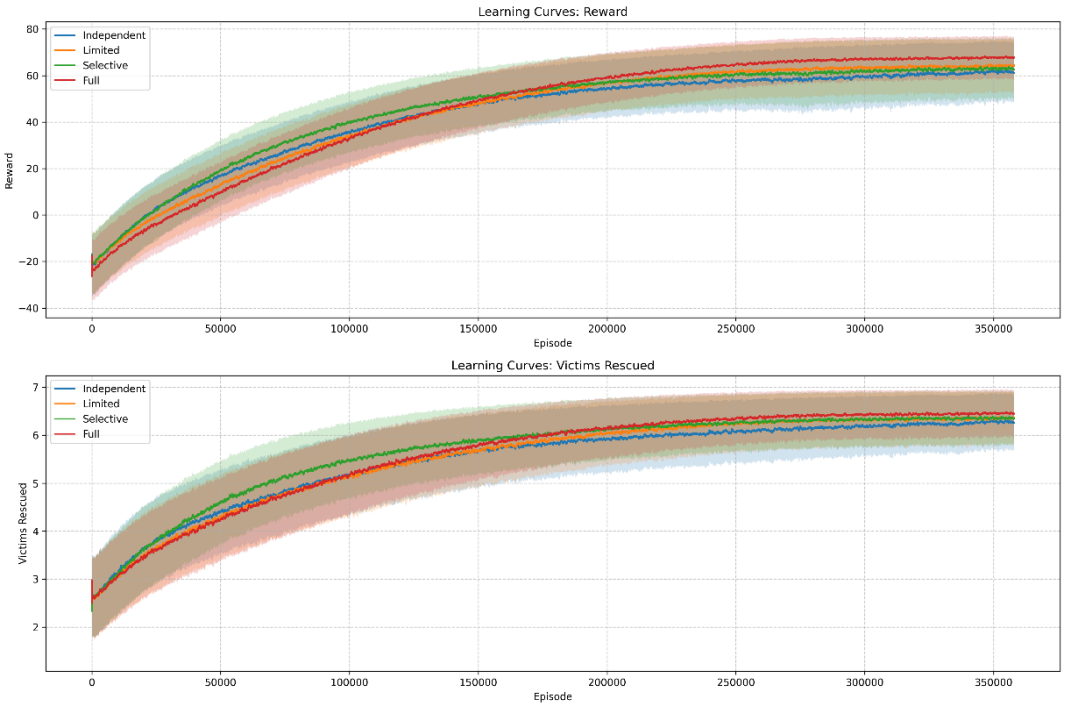

Figure 6: Learning Curves – Reward & Victims Rescued

Comparison	Reward	Steps	Duplicates	Bytes	Coverage	Success Rate
Ind vs Lim	+3.28 **	-0.73	-6.25 **	+27.91 **	+5.76 **	+1.15 *
Ind vs Sel	+1.20 *	-1.68 **	-3.50 **	+90.73 **	+1.57 **	+1.33 *
Ind vs Full	+5.21 **	-1.93 **	-7.25 **	+10.92 **	+10.68 **	+2.25 **
Lim vs Sel	-1.23 *	-1.32 *	+1.30 *	-11.41 **	-3.81 **	+0.48
Lim vs Full	+3.66 **	-1.62 **	-2.74 **	+7.24 **	+6.04 **	+1.61 **
Sel vs Full	+3.64 **	-0.28	-3.32 **	+8.80 **	+8.64 **	+0.94

Table 3: Statistical Significance – Effect Size (Cohen's d) Interpretation Table

Values indicate how many standard deviations apart the two group means are. A positive value implies $\mu_{\text{strategy2}} > \mu_{\text{strategy1}}$, while a negative value implies $\mu_{\text{strategy1}} > \mu_{\text{strategy2}}$. Asterisks (*) denote statistical significance based on *p*-values:

- No * indicates $p > 0.05$ (not significant),
- * indicates $p < 0.05$ (statistically significant),
- ** indicates $p < 0.01$ (highly significant),

All significance levels are reported with a 95% confidence interval.

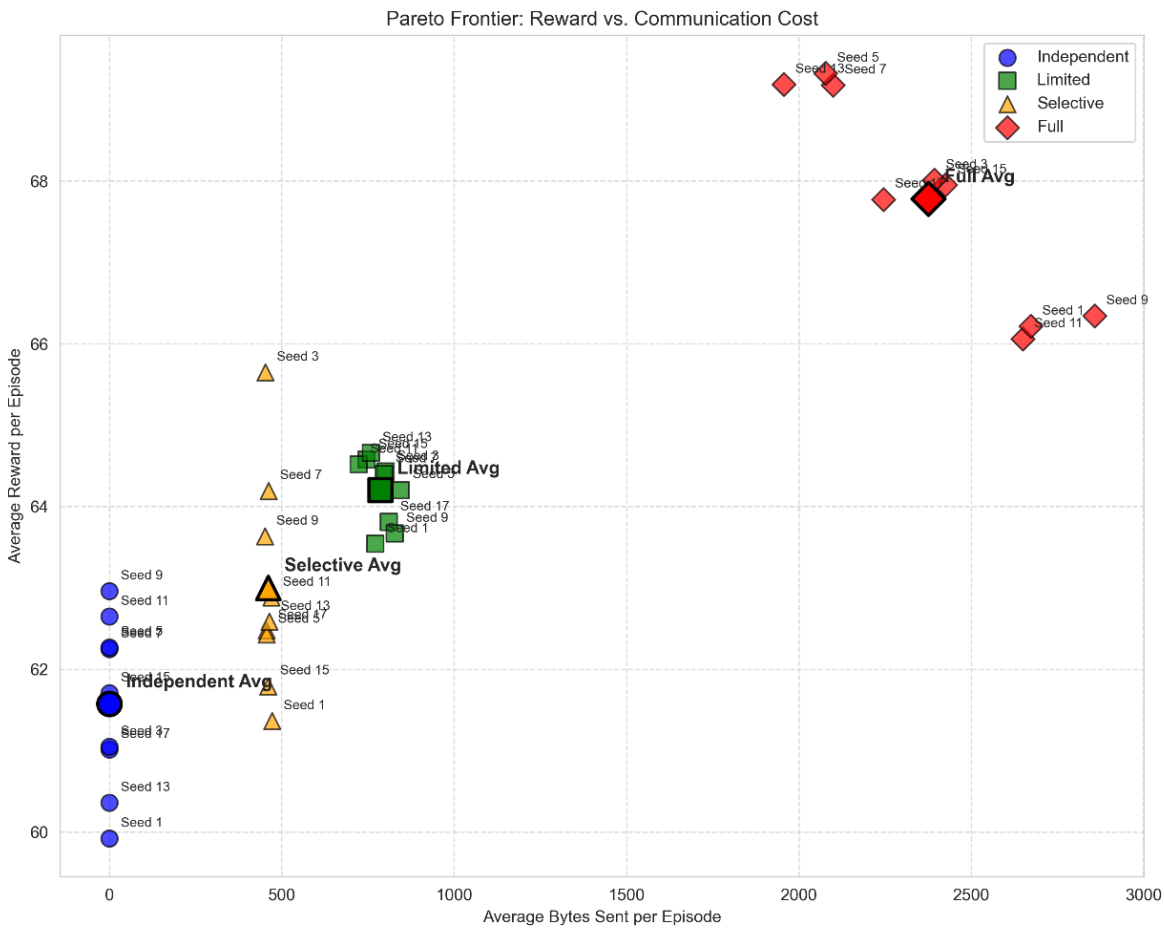

Figure 7: Pareto Frontier – Reward vs Communication Cost

6. Discussion

The main experiment results confirm that communication significantly influences learning efficiency and coordination in multi-agent search-and-rescue. Efficiency bars show that communication, especially full, reduces redundancy and improves efficiency, but at a steep communication cost. Learning curves reveal that all agents learn, but those with more communication converge to better policies, rescuing more victims and earning higher rewards. Although full and limited outperform selective and independent, it is worth noting that selective agents seem to learn faster and achieve similar performance to limited at a lower communication cost. The Pareto frontier clearly highlights the trade-off between performance and communication:

- Full Communication is optimal for reward but very costly.
- Selective/Limited strategies offer a strong balance – good reward for much less communication.
- Independent is cheapest but least effective.

Statistical analysis reinforces these findings, with differences that are statistically significant ($p < 0.05$, often $p < 0.001$) and exhibit large effect sizes (Cohen's $d > 0.8$).

7. Conclusion and Future Work

To answer the question "What is the optimal balance of communication to maximize efficiency while minimizing redundant searches?", it's a hard choice between Selective and Limited agents from the results, but in limited bandwidth scenarios I would lean towards selective agents.

Future works include implementing Dynamic Communication Policies, Function Approximation (DQN), Multi-Agent Coordination via Joint Policies (QMIX) and completing the graphical SARVisualiser plugin in utils/graphics.py using PyGame.

References

Bellman R. E. (1957) *Dynamic Programming,* Princeton University Press, Princeton, NJ

Watkins, C. J. C. H., & Dayan, P. (1992). *Technical Note: Q-Learning*. Machine Learning, Kluwer Academic Publishers

Mahadevan S., Connell J. (1991), *Automatic Programming of Behaviour Based Robots using Reinforcement Learning*, In Proceedings of AAAI-91

Tan, M. (1993). *Multi-Agent Reinforcement Learning: Independent vs. Cooperative Agents*. In Proceedings of the Tenth International Conference on Machine Learning, Morgan Kaufmann

Van Otterlo, M., & Wiering, M. (2012). *Reinforcement Learning and Markov Decision Processes*. In *Reinforcement Learning: State-of-the-Art* , Springer.

Reinforcement Learning, Q Learning, Markov Decision processes – Youtube videos from Computerphile, CodeEmporium, Steve Brunton, Freecodecamp