Containing Information Contagions A Comparative Study of Heuristic, Planning, and Deep Reinforcement Learning Strategies

Sagnik Pal(B2530058)*, Silajeet Banerjee(B2530059)*

Abstract

The spread of misinformation through social networks presents a complex challenge that requires intelligent, resource-aware intervention strategies. This project introduces an interactive simulation framework for modeling and controlling rumor propagation over large-scale networks. The underlying social structure is represented using a Connected Caveman graph to reflect the scale-free properties of real-world networks. Rumor diffusion is modeled as a stochastic process in which infected nodes probabilistically transmit misinformation to neighboring nodes over discrete time steps. The task is formulated as a sequential decision-making problem where an agent is constrained by a limited daily budget of interventions, such as inoculating or fact-checking selected nodes, with the objective of minimizing total infection while accounting for time costs. The environment supports multiple agent architectures, including a heuristic degree-based agent, a Monte Carlo Tree Search (MCTS) planning agent, deep q-learning reinforcement learning agent, a graph neural network based learning agent and a random agent. A graphical user interface enables real-time visualization, manual user control, and systematic comparison of agent strategies. The simulator incorporates episode termination conditions, scoring mechanisms, and detailed logging to facilitate experimental evaluation. Overall, this project provides a flexible platform for studying rumor containment under uncertainty and serves as a foundation for further research in graph-based planning and reinforcement learning.

Code — https://github.com/Rumor-Spread-Inoculation-Agent/NetGuard_AI.git

Introduction

The rapid spread of misinformation across online social platforms has emerged as a critical societal and technological challenge. Unlike traditional information diffusion, rumor propagation in social networks is driven by complex interaction patterns, heterogeneous node influence, and stochastic transmission dynamics. Empirical studies have shown that real-world social networks often exhibit scale-free structures, where a small number of highly connected nodes play a disproportionate role in shaping global information flow. These structural properties make rumor containment particularly difficult, as interventions must be carefully targeted to prevent cascades while operating under limited resources and incomplete information. Consequently, designing effective strategies for mitigating misinformation requires principled modeling of both network dynamics and decision-making processes.

From the standpoint of artificial intelligence, rumor control constitutes a rich and challenging sequential decision-making problem. At each time step, an intelligent agent observes the current state of a dynamic graph representing the social network, where nodes may be susceptible, infected, or cured. The agent must then select a subset of nodes to intervene upon, subject to strict budget constraints, before the rumor further propagates. The environment evolves stochastically as infected nodes transmit misinformation to their neighbors with some probability, introducing uncertainty and delayed consequences for actions. This setting naturally aligns with the formalism of Reinforcement Learning, where the state space is large, combinatorial, and graph-structured, and the action space grows rapidly with network size. These characteristics make exact solutions intractable and motivate the exploration of approximate planning, heuristic methods, and learning-based approaches.

Reinforcement learning and planning techniques provide powerful tools for addressing such problems, particularly in domains where long-term consequences must be balanced against short-term costs. Heuristic strategies, such as prioritizing high-degree or centrally located nodes, offer interpretable baselines but may fail to adapt to changing dynamics. Planning methods like Monte Carlo Tree Search enable deeper lookahead under uncertainty but are computationally constrained by the size of the state space. More recent advances in deep reinforcement learning and graph neural networks promise scalable representations that can generalize across different network structures, making them especially relevant for this domain. Studying rumor containment within a unified simulation framework therefore offers valuable insight into the strengths and limitations of these AI paradigms.

The objective of this project is to develop an interactive, extensible simulation environment for modeling and controlling rumor propagation on large-scale networks. The environment represents social interactions using Connected

*These authors contributed equally.

Caveman graphs to capture realistic connectivity patterns and simulates rumor diffusion as a probabilistic process over discrete time steps. The framework supports multiple agent architectures, including heuristic, Monte Carlo Tree Search, reinforcement learning–based and Graph Nural Network based agents enabling systematic comparison under consistent conditions. A graphical user interface facilitates real-time visualization, manual intervention, and detailed logging, enhancing both interpretability and experimental rigor. As an academic course project, this work aims to integrate theoretical concepts from artificial intelligence and reinforcement learning with a practical, socially motivated application, while providing a foundation for further experimentation in graph-based planning, learning under constraints, and decision-making in complex networks.

Problem Formulation and Representation

This project formulates the task of controlling rumor propagation in a social network as a sequential decision-making problem under uncertainty. The objective is to design intelligent agents that can strategically allocate limited intervention resources over time in order to minimize the spread of misinformation. The problem combines elements of graph theory, stochastic processes, and artificial intelligence planning and learning, making it a suitable abstraction for studying decision-making in complex, dynamic environments.

Problem Domain

The problem domain consists of a social network modeled as a graph, where nodes represent individuals and edges represent social connections through which information can spread. To capture the structural properties observed in real-world social networks, the underlying graph is generated using the Connected Caveman preferential attachment model, resulting in a scale-free topology with heterogeneous node degrees. This choice reflects the presence of highly influential individuals who can significantly impact rumor diffusion.

Rumor spread is modeled as a stochastic contagion process that evolves over discrete time steps. Each node in the network can be in one of three states: susceptible (unaware of the rumor), infected (actively spreading misinformation), or inoculated (immune due to intervention, such as fact-checking or awareness). At each time step, infected nodes probabilistically transmit the rumor to their susceptible neighbors. The agent interacts with this environment by selecting a subset of nodes to inoculate before the next diffusion step occurs. The domain is therefore dynamic, partially controllable, and inherently uncertain.

State, Actions, and Constraints

State Representation The state of the environment at any given time step is defined by the current graph structure, the status of each node, and the elapsed time. Formally, the state can be represented as a tuple **(G,s,t)**, where G denotes the network graph, s is a vector encoding the state of each node (susceptible, infected, or inoculated), and t represents the current time step or day. For learning-based agents, this state is further encoded into structured representations such as adjacency matrices and node feature vectors to enable efficient processing.

Actions / Operators An action corresponds to selecting a subset of nodes to inoculate at the current time step. Inoculating a node transitions it from the susceptible state to the inoculated state, preventing future infection. The action space is combinatorial, as the agent may choose multiple nodes simultaneously, and its size grows with the number of susceptible nodes in the network. Different agents generate actions using different strategies, including degree-based heuristics, randomized selection, planning via Monte Carlo Tree Search, or learning-based policies.

Constraints / Rules The primary constraint is a fixed daily intervention budget that limits the maximum number of nodes the agent can inoculate at each time step. Additional hard constraints include the inability to inoculate already infected or inoculated nodes and the irreversible nature of state transitions (e.g., inoculated nodes remain immune). The rumor propagation itself follows probabilistic rules governed by an infection probability, introducing stochasticity into the environment. These constraints collectively enforce realism and prevent trivial or unconstrained solutions.

Goal Test The goal of the agent is to minimize the overall spread of the rumor over time. An episode is considered complete when no susceptible nodes remain or when no further infections are possible. Performance is evaluated using a scoring function that balances the number of inoculated nodes, the number of infected nodes, and the time taken, thereby capturing both effectiveness and efficiency. This formulation allows the problem to be evaluated both as an optimization task and as a benchmark for comparing different AI and reinforcement learning strategies.

Environment

The environment provides a controlled yet realistic abstraction of rumor propagation in social networks, enabling systematic evaluation of different artificial intelligence and reinforcement learning agents. It encapsulates the network structure, rumor diffusion dynamics, intervention mechanisms, and evaluation criteria, and serves as the interface through which agents perceive the state of the system and apply actions. The environment is designed to support both algorithmic experimentation and interactive visualization, making it suitable for academic analysis as well as practical demonstration.

Environment Description

The environment models a social network as an undirected graph, where nodes correspond to individuals and edges represent social interactions through which rumors can spread. The network structure is typically generated using the Connected Caveman preferential attachment model, resulting in a scale-free graph that captures the heterogeneity and hub-based connectivity observed in real social systems. Each

node exists in one of three discrete states: susceptible, infected, or inoculated.

Observation Space At each time step, the agent observes the current state of the environment, which includes the network topology, node-level states, and the current time index. For computational agents, this observation is encoded as a structured representation consisting of an adjacency matrix describing graph connectivity and a node-state vector indicating the infection or inoculation status of each node. This representation allows both classical planning agents and learning-based agents to reason over the graph structure and its evolving dynamics.

Action Space The action space consists of selecting a subset of nodes to inoculate at the current time step. Actions are constrained by a fixed daily intervention budget, which limits the maximum number of nodes that can be selected. Only susceptible nodes may be inoculated, and each action is applied before the rumor diffusion process for that time step. Due to the combinatorial nature of node selection, the action space grows rapidly with network size, making the problem computationally challenging and suitable for approximate planning and learning methods.

Reward Structure The reward (or scoring) mechanism is designed to encourage effective and timely containment of rumors. Agents are rewarded for successfully inoculating nodes and penalized for allowing nodes to become infected, as well as for prolonged episode duration. This trade-off reflects real-world constraints, where interventions are costly and delayed responses can amplify misinformation spread. The reward structure enables evaluation of both short-term and long-term decision quality and supports reinforcement learning formulations.

Episode Structure and Termination Conditions Each episode proceeds over discrete time steps. At every step, the agent applies an intervention action, after which rumor diffusion occurs probabilistically. An episode terminates when no susceptible nodes remain, when the rumor can no longer spread, or when all nodes are either infected or inoculated. These termination conditions ensure finite episodes and provide clear criteria for evaluating agent performance.

Working of the Simulation

The simulation operates by repeatedly generating, evolving, and visualizing instances of a rumor propagation process over a social network. At the start of an episode, an environment instance is created by generating a graph using the Connected Caveman preferential attachment model. This model produces a scale-free network that reflects realistic social connectivity patterns, where a small number of nodes act as highly connected hubs. The number of nodes, attachment parameters, infection probability, and daily intervention budget are configurable and may be adjusted by the user through the graphical interface. Initial rumor sources are selected at the beginning of each episode, typically by randomly infecting a small subset of nodes, while all remaining nodes are initialized as susceptible.

Once initialized, the simulation proceeds in discrete time steps. At each step, the selected agent (heuristic, planning-based, learning-based, or user-controlled) chooses a set of nodes to inoculate, subject to the predefined budget constraint. These inoculations are applied immediately, after which the rumor spreads stochastically: each infected node attempts to infect its susceptible neighbors with a fixed probability. The environment updates node states accordingly and advances the simulation clock. This cycle continues until a termination condition is reached, such as the absence of infected nodes or exhaustion of susceptible nodes.

The simulation is tightly integrated with an interactive graphical user interface (GUI), which serves both as a visualization tool and as a control panel for experimentation. The central area of the GUI displays the social network graph, where each node is visually encoded according to its current state. Susceptible nodes are shown in blue, infected nodes in red, and inoculated (protected) nodes in green, enabling immediate visual interpretation of rumor spread and containment. Node sizes and edge layouts remain consistent across time steps to preserve spatial continuity and facilitate comparison of different simulation runs.

The GUI includes a set of control buttons that allow the user to interact with the environment. A Start/Run button advances the simulation using the currently selected agent, while a Reset Environment button restores the current graph to its initial state, resetting all node statuses without changing the network structure. A separate New Environment button generates an entirely new graph instance with fresh initial conditions. Agent selection controls enable switching between different automated strategies, such as heuristic, MCTS, reinforcement learning agents, or a manual user agent, allowing systematic comparison under identical conditions.

Parameter configuration elements within the GUI allow the user to modify key simulation variables, including infection probability, intervention budget, and initial infection settings, without restarting the application. When the user agent is selected, the interface supports direct interaction with the graph, allowing users to manually choose nodes for inoculation and observe the resulting effects in real time.

A dedicated log panel records detailed step-by-step information throughout the simulation. This log includes the current time step, nodes selected for intervention, newly infected nodes, and summary statistics such as total infected and inoculated counts. The log can be cleared during runtime or saved for offline analysis, supporting reproducibility and systematic evaluation of agent behavior. Together, the simulation engine and GUI provide a coherent framework for experimenting with rumor containment strategies, combining algorithmic rigor with intuitive visual feedback.

Methodology

This project adopts a multi-paradigm artificial intelligence methodology to study and compare different decision-making strategies for controlling rumor propagation on networks. The task is framed as a sequential decision-making problem under uncertainty, allowing the use of classical

planning, heuristic search, and reinforcement learning techniques. By implementing multiple agents within a common environment, the framework enables systematic evaluation of algorithmic trade-offs between computational cost, decision quality, and adaptability to stochastic dynamics.

Algorithms Used

Degree-Based Heuristic Algorithm The degree-based heuristic algorithm selects nodes for inoculation based on their connectivity in the network. At each time step, the agent identifies all susceptible nodes and ranks them according to their degree in the graph. The top B nodes, where B denotes the daily intervention budget, are selected for inoculation.

Algorithm: Degree-Based Heuristic

1. Identify all nodes with susceptible status.
2. Compute the degree of each susceptible node.
3. Sort nodes in descending order of degree.
4. Select the top B nodes for inoculation.
5. Apply inoculation and advance the simulation by one time step.

This algorithm is deterministic, computationally efficient, and does not require learning or simulation-based planning.

Monte Carlo Tree Search (MCTS) Algorithm Monte Carlo Tree Search is employed as a planning-based approach to evaluate the long-term effects of inoculation decisions. MCTS incrementally constructs a search tree in which each node represents a state–action pair. The algorithm balances exploration and exploitation using the Upper Confidence Bound for Trees (UCT) criterion and estimates action values through stochastic rollouts. In practice, MCTS plans inoculations greedily by selecting one node at a time and repeating the planning process until the daily budget is exhausted.

Algorithm: Monte Carlo Tree Search

Input: Current environment state s, simulation budget N, planning horizon H

Output: Action a, representing a set of nodes to inoculate

1. Initialize the root node corresponding to the current state s.
2. For each simulation $i = 1$ to N:
   (a) **Selection:** Traverse the tree by selecting child nodes according to the UCT criterion:
   $\frac{Q(v)}{N(v)} + c\sqrt{\frac{\ln N(\text{parent}(v))}{N(v)}}$
   where $Q(v)$ is the cumulative reward, $N(v)$ is the visit count of node v, and c is an exploration constant.
   (b) **Expansion:** If the selected node is non-terminal and has unexpanded actions, add a new child node corresponding to one such action.
   (c) **Simulation (Rollout):** From the expanded node, simulate rumor spread for up to H steps using a default policy.
   (d) **Backpropagation:** Propagate the rollout reward back through the visited nodes, updating visit counts and value estimates.
3. Select the child of the root node with the highest visit count.
4. Return the corresponding inoculation action.

This planning-based approach enables limited look-ahead reasoning under uncertainty and explicitly models stochastic transitions and delayed rewards without requiring a learned policy.

Reinforcement Learning (Deep Q-Learning) The reinforcement learning agent uses Deep Q-Learning to learn an intervention policy directly from interaction with the environment. The action–value function $Q(s, a)$ estimates the expected cumulative reward of performing action a in state s and is approximated using a neural network to handle high-dimensional state representations.

- DQN learns a node-level Q-function.
- Multi-node inoculation is approximated by selecting the top-B nodes by Q-value.
- Training treats each inoculation as an independent action.

Algorithm: Deep Q-Learning (DQN) for Rumor Containment

Input: Learning rate α, discount factor γ, exploration rate ϵ, replay buffer size M

Output: Learned policy i.e., $\pi(s) = \arg\max_a Q(s, a)$

1. Initialize Q-network parameters θ and target network parameters θ^-.
2. Initialize an experience replay buffer D.
3. For each episode:
   (a) Reset the environment and observe the initial state.
   (b) For each time step:
      i. Select an action using an ϵ-greedy policy.
      ii. Execute the action and observe the reward and next state.
      iii. Store the transition (s, a, r, s') in the replay buffer.
      iv. Sample a mini-batch from D and update the Q-network by minimizing the temporal-difference loss: $\left(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta)\right)^2$
   (c) Periodically synchronize the target network parameters.

The learned Q-function is used as the control policy. Deep Q-Learning allows the agent to learn effective intervention strategies without requiring an explicit model of rumor dynamics.

Graph Neural Network (GNN)–Based Representation Graph Neural Networks are integrated with reinforcement learning to directly approximate node-level action-value functions, enabling topology-aware intervention policies. Through iterative message passing between neighboring nodes, GNNs generate node embeddings that capture both local and global structural properties of the network. These embeddings can be integrated with reinforcement learning methods to produce topology-aware intervention policies.

To incorporate topological information directly into the decision-making process, a Graph Neural Network

(GNN)–based reinforcement learning agent is employed. The agent uses a GraphSAGE architecture to approximate node-level action-value functions, enabling topology-aware inoculation strategies.

Node Feature Representation Each node $v \in V$ is represented by a fixed-dimensional feature vector:

$$\mathbf{x}_v = [s_v,\ d_v,\ b_v]$$

where s_v denotes the node status (susceptible, infected, or inoculated), d_v is the normalized degree of the node, and b_v is the normalized betweenness centrality. Betweenness centrality is precomputed once during environment initialization to reduce runtime overhead.

GraphSAGE Architecture The policy network is implemented using a two-layer GraphSAGE model with max aggregation. Given node features $\mathbf{X}$ and edge index $\mathcal{E}$, the forward propagation is defined as:

$$\mathbf{H}^{(1)} = \text{ReLU}(\text{SAGEConv}(\mathbf{X}, \mathcal{E}))$$

$$\mathbf{H}^{(2)} = \text{ReLU}(\text{SAGEConv}(\mathbf{H}^{(1)}, \mathcal{E}))$$

$$Q_v = \mathbf{W}\mathbf{H}_v^{(2)}$$

where Q_v denotes the estimated action-value of inoculating node v. The output layer produces a scalar value per node, representing its relative priority for intervention.

Action Selection At each time step, the agent selects nodes using an ϵ-greedy strategy. With probability ϵ, a random subset of susceptible nodes is selected. Otherwise, the agent selects the top B susceptible nodes with the highest predicted Q_v values, where B is the daily inoculation budget. Nodes that are not susceptible are masked by assigning their Q-values to $-\infty$.

Learning Procedure Training follows a temporal-difference learning framework. Transitions of the form (s, a, r, s', d) are stored in a replay buffer. The loss for a transition is computed as:

$$\mathcal{L} = \left(Q(s,a) - \left[r + \gamma \max_{v'} Q(s', v')\right]\right)^2$$

where γ is the discount factor. A separate target network is used to stabilize learning and is periodically synchronized with the policy network.

Reward Signal The reward function penalizes both current and newly infected nodes, with a large terminal bonus assigned when the rumor spread is fully contained. This encourages policies that minimize long-term infection rather than greedy short-term gains.

This GNN-based agent enables end-to-end learning of intervention strategies that explicitly exploit graph structure, allowing the learned policy to generalize across different network configurations.

Justification

Heuristic Agent The heuristic agent leverages the scale-free characteristics commonly observed in social networks, where high-degree nodes serve as influential hubs for information diffusion. Targeting such nodes provides an effective and interpretable baseline strategy that requires minimal computational overhead, making it suitable for rapid intervention and benchmarking.

MCTS Agent MCTS is well suited to environments characterized by uncertainty and delayed effects, both of which are fundamental aspects of rumor propagation. By simulating future outcomes, MCTS enables reasoning about the long-term consequences of current actions. Its ability to operate without a fully specified transition model makes it appropriate for complex stochastic settings.

Reinforcement Learning Agent Reinforcement learning naturally aligns with the sequential and reward-driven nature of the rumor containment problem. The agent must balance short-term containment actions against long-term spread reduction. Deep Q-Learning further enables generalization across large and dynamically evolving state spaces.

Graph Neural Network Agent GNNs are justified by the inherently relational structure of social networks. Unlike flat feature representations, GNNs preserve topological dependencies and enable learned policies to generalize across graphs of varying sizes and structures, supporting scalable and transferable rumor control strategies.

Mathematical / Formal Description

The rumor containment problem is formulated as a Markov Decision Process (MDP):
$\mathcal{M} = (S, A, T, R, \gamma)$

State Space (S) A state consists of the network graph $G = (V, E)$, a node status function
$\sigma : V \to S, I, R$
indicating whether each node is susceptible, infected, or recovered, along with the current time step.

Action Space (A) Actions correspond to selecting a subset of susceptible nodes for inoculation, subject to a daily budget constraint B.

Transition Function (T) State transitions are stochastic. After inoculation, infected nodes transmit the rumor to neighboring susceptible nodes with probability p.

Reward Function (R) The reward function balances containment effectiveness, infection penalties, and temporal cost. In learning-based agents, the reward is instantiated as a negative function of newly infected and currently infected nodes, with optional terminal bonuses for complete containment.

Discount Factor (γ) The discount factor controls the relative importance of future rewards compared to immediate outcomes.

This formulation provides a unified decision-theoretic framework that encompasses heuristic, planning-based, and learning-based approaches to rumor containment.

Implementation

This section describes the practical implementation details of the proposed rumor containment framework, including the software tools used, learning hyperparameters, and training hardware configuration.

Tools and Libraries

The entire system was implemented in **Python**, making use of widely adopted scientific computing, graph processing, deep learning, and visualization libraries. The key tools and libraries used are summarized below:

- **Python**: Core programming language used for environment simulation, agent implementation, training, and evaluation.
- **NumPy**: Used for numerical operations, array manipulation, state representation, and efficient computation of node features.
- **NetworkX**: Used to construct and manipulate graph structures, including the generation of community-based (connected caveman) graphs and computation of graph statistics such as node degree and betweenness centrality.
- **PyTorch**: Used to implement deep reinforcement learning models, including Deep Q-Networks (DQN), optimization routines, and loss computation.
- **PyTorch Geometric**: Used to implement the Graph Neural Network agent, specifically the GraphSAGE architecture with message passing over graph edges.
- **Matplotlib**: Used for plotting training curves, evaluation results, and comparative performance graphs.
- **PyQt5**: Used to develop the interactive graphical user interface (GUI) for visualizing rumor spread, agent actions, and simulation controls.
- **tqdm**: Used to display progress bars during training and evaluation loops.

No external reinforcement learning frameworks (such as OpenAI Gym) or game engines were used; instead, a custom environment was implemented to provide full control over the rumor propagation dynamics and intervention process.

Hyperparameter Values and Tuning Strategy

Hyperparameters were chosen empirically based on stability and learning performance. No automated hyperparameter optimization was performed; instead, values were manually tuned through repeated experimentation.

Deep Q-Learning (DQN) Agent

- Learning rate: 0.001
- Discount factor (γ): 0.99
- Replay buffer capacity: $10{,}000$
- Mini-batch size: 64
- Initial exploration rate (ϵ): 1.0
- Minimum exploration rate: 0.01
- Exploration decay rate: 0.995
- Target network update frequency: every 10 episodes
- Maximum episode length: 50 time steps
- Training episodes: 500

During training, domain randomization was applied by varying the infection probability between 0.05 and 0.30 to improve robustness across different rumor spreading conditions.

Graph Neural Network (GNN) Agent

- Architecture: GraphSAGE
- Number of GraphSAGE layers: 2
- Hidden dimension size: 64
- Aggregation function: max aggregation
- Learning rate: 0.002
- Discount factor (γ): 0.99
- Replay buffer capacity: $10{,}000$
- Mini-batch size: 32
- Initial exploration rate (ϵ): 1.0
- Final exploration rate: 0.05
- Exploration decay: exponential decay over 150 episodes
- Target network update frequency: every 10 episodes
- Training episodes: 300
- Infection probability range during training: 0.15–0.30

The GNN agent was trained on community-structured graphs and learns node-level action-value estimates using temporal-difference learning with a target network for stability.

Training Duration and Hardware Details

All experiments were conducted on CPU-based systems without GPU acceleration. Two laptops were used during development, training, and evaluation:

- **Laptop 1**: AMD Ryzen 3 (7000 series), 8 GB DDR5 RAM, 512 GB SSD.
- **Laptop 2**: AMD Ryzen 5 3450U, 8 GB DDR4 RAM, 1 TB SSD.

Training times varied depending on the agent. The DQN agent required five minutes to complete 500 training episodes. The GNN agent required longer training time of around thirty minutes due to graph-based message passing and batch-level learning but remained feasible on CPU-only hardware. Despite hardware constraints, the system successfully trained learning-based agents and supported real-time visualization through the GUI.

Results

This section presents the experimental results of the proposed rumor containment framework. The performance of different agents is evaluated using quantitative metrics, training dynamics, and qualitative observations derived from the simulation environment. The compared agents include a random baseline, a degree-based heuristic agent, a Deep Q-Learning (DQN) agent, and a Graph Neural Network–based reinforcement learning (GNN-RL) agent.

Evaluation Metrics

The primary evaluation metric used across experiments is the **average number of final infected nodes**, where lower values indicate better rumor containment. This metric directly captures the effectiveness of intervention strategies in limiting misinformation spread.

For reinforcement learning agents, additional evaluation metrics include:

- **Total episodic reward**, reflecting cumulative containment performance,
- **Training reward curves**, indicating learning progress and stability,
- **Exploration rate** (ϵ) **and loss curves**, providing insight into learning dynamics and convergence behavior.

Comparative Performance of Agents

Figure 1 compares the average final number of infected nodes for different agents on a community-structured graph with infection probability $p = 0.15$.

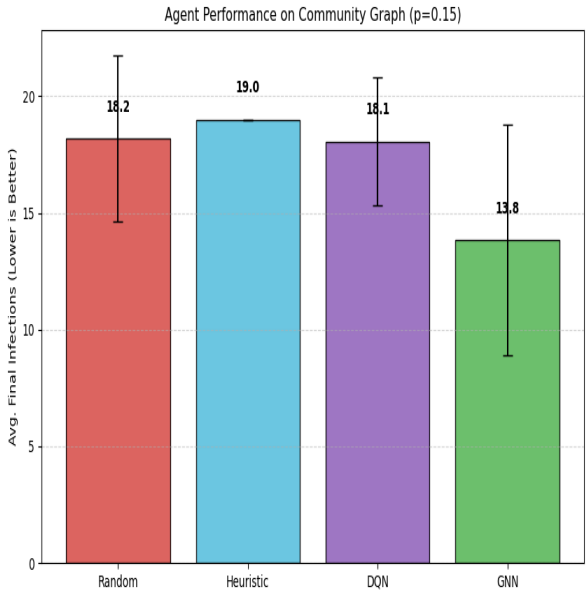


Figure 1: Average final number of infected nodes for different agents on a community graph ($p = 0.15$). Lower values indicate better rumor containment.

Observations

- The **random agent** performs worst, resulting in the highest average number of infected nodes.
- The **degree-based heuristic** improves containment by targeting highly connected nodes.
- The **DQN agent** achieves a substantial reduction in final infections compared to both baselines.
- The **GNN-based agent** matches or slightly improves upon DQN performance, while exhibiting higher variance.

Overall, learning-based agents consistently outperform heuristic and random strategies.

Reinforcement Learning Training Dynamics

Deep Q-Learning (DQN) Figure 2 shows the total reward per episode during DQN training and the decay of the exploration parameter ϵ.

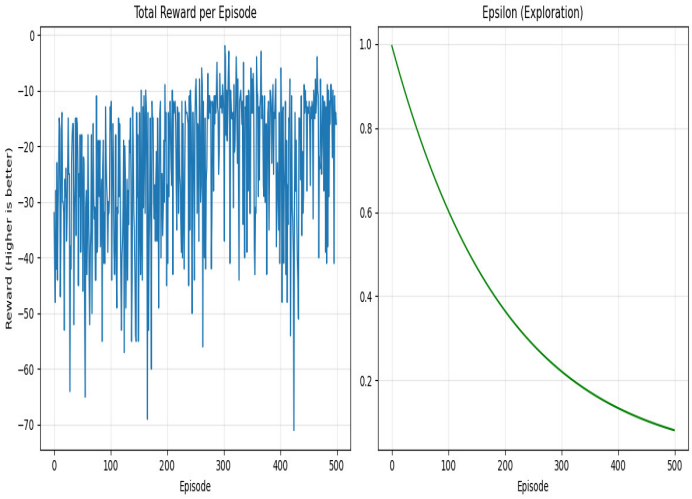


Figure 2: Total episodic reward during DQN training.

Early training is characterized by high variance due to extensive exploration. As ϵ decays, the agent increasingly exploits learned policies, resulting in improved and more stable rewards.

GNN-Based Reinforcement Learning Agent Figure 3 presents episodic training rewards for the GNN-based agent.

Figure 9 illustrates the reward history, loss evolution, and exploration decay during GNN training.

The loss curve shows a general downward trend, indicating stabilization of the learned Q-value estimates. The exploration rate decays smoothly, allowing the agent to gradually shift from exploration to exploitation. Compared to DQN, the GNN agent exhibits higher reward variance due to graph-based message passing and structural sensitivity.

Qualitative Analysis and Interface Observations

In addition to quantitative evaluation, qualitative observations from the graphical user interface provide insight into agent behavior. Learning-based agents prioritize structurally important nodes, often blocking inter-community rumor transmission early. Heuristic strategies, in contrast, rely solely on degree information and may overlook critical bridge nodes.

The interface visualization confirms that GNN-based agents adapt their actions based on both node status and graph topology.

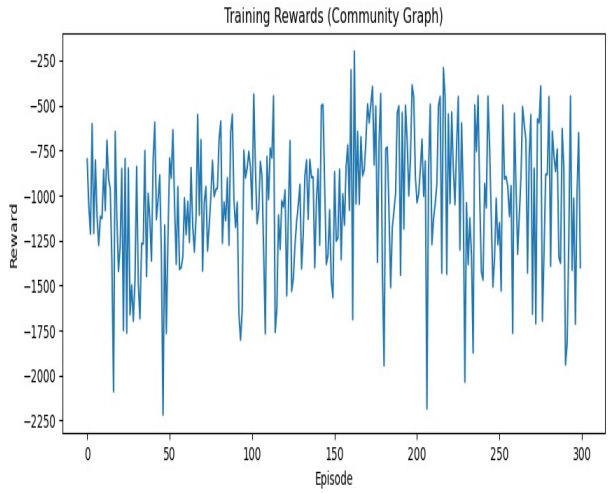


Figure 3: Training rewards per episode for the GNN-based reinforcement learning agent.

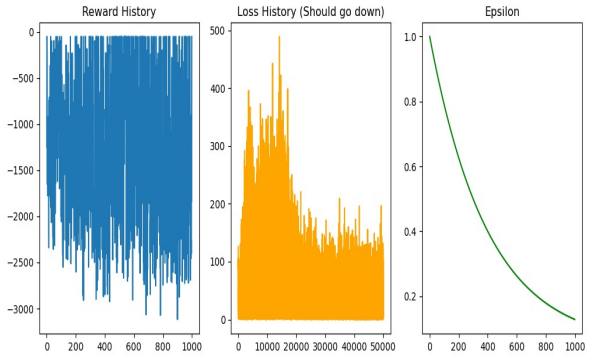


Figure 4: Training dynamics of the GNN-based agent: reward history (left), loss curve (center), and exploration decay (right).

Summary of Results

The experimental results demonstrate that:

- Reinforcement learning agents significantly outperform random and heuristic baselines.
- DQN provides stable and competitive performance with relatively low training complexity.
- GNN-based reinforcement learning further improves containment by exploiting graph structure. And This model provides the best result.
- Training curves confirm effective learning and convergence under CPU-only hardware constraints.

These results highlight the importance of combining sequential decision-making with structural awareness for effective rumor containment in social networks.

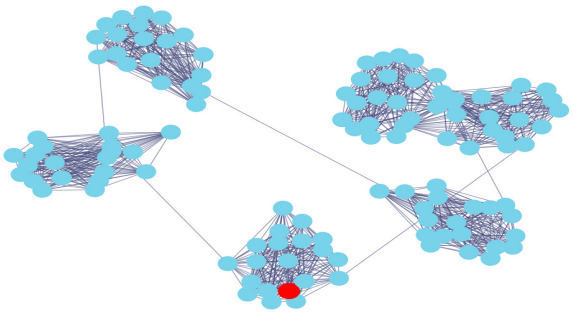


Figure 5: Example Problem Graph

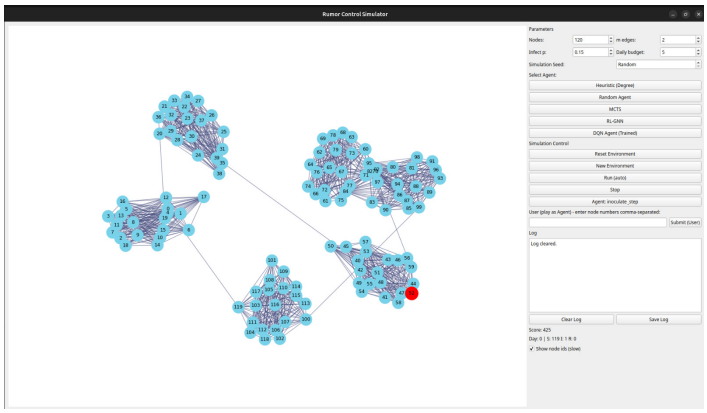


Figure 6: GUI

Discussion

This section reflects on the implementation process, interprets the experimental results, and discusses unexpected behaviors, limitations, and practical challenges encountered during development and evaluation.

Observations During Implementation

Several important insights emerged during the implementation and experimentation phase. One notable observation was that **curriculum learning**, which initially appeared promising, proved counterproductive in this setting. Training agents first on easier rumor transmission rates resulted in the emergence of *lazy policies* that failed to generalize to the target environment. When exposed to higher infection rates during evaluation, these agents reacted too slowly, allowing the rumor to spread uncontrollably. In contrast, agents trained directly at the target difficulty level learned urgency-aware behaviors that were better aligned with the dynamics of rapid misinformation spread.

Another key observation concerned the **computational sensitivity of Monte Carlo Tree Search (MCTS)**. While MCTS occasionally produced near-optimal solutions, its performance was highly inconsistent under stochastic transitions and limited simulation budgets. Increasing reliabil-

ity at higher infection probabilities would require orders of magnitude more simulations, making MCTS impractical for real-time or interactive settings within the given hardware constraints.

Interpretation of Results

The experimental results clearly demonstrate the advantage of learning-based approaches over heuristic strategies in community-structured social networks. While degree-based heuristics performed reasonably well, their effectiveness was fundamentally limited by assumptions that do not hold in modular graphs.

Both the Deep Q-Network (DQN) and the GNN-based agent converged to nearly identical average infection rates. This convergence strongly suggests that the agents discovered the **theoretically optimal containment strategy** for the given network topology. Since the simulation begins with an already infected node, it is impossible to fully prevent infection in the initial community. The optimal policy therefore consists of immediately isolating the infected cluster while preventing inter-community spread. The agreement between two distinct learning architectures provides strong empirical evidence that this strategy represents the optimal solution.

Anomalies and Unexpected Outcomes

A particularly revealing anomaly was the failure of the **degree-based heuristic in community networks**, despite its known effectiveness in scale-free graphs. This phenomenon, referred to as the *hub fallacy*, arises because high-degree nodes in modular networks are often influential only within their own communities. In such cases, targeting hubs merely slows local diffusion while allowing the rumor to propagate across communities through structurally critical but low-degree nodes.

This led to the crucial insight that **bridge nodes**, rather than hubs, are the most important targets for global containment. These nodes often have unremarkable degrees, making them invisible to traditional heuristics, but exhibit high betweenness centrality. The superior performance of the GNN-based agent can be directly attributed to its ability to identify and prioritize these bridge nodes, thereby blocking the main transmission pathways between communities.

Another unexpected but instructive outcome was the sensitivity of the GNN agent to **feature engineering**. Early experiments using only degree and clustering coefficient failed to produce meaningful learning. The inclusion of betweenness centrality immediately enabled successful training, highlighting that Graph Neural Networks are not inherently capable of discovering all structural roles without informative input features. This underscores the importance of domain knowledge in designing effective graph-based learning systems.

Stability, Variance, and Risk Trade-offs

The error bars observed in the results provide important insight into the trade-off between stability and performance. The heuristic agent exhibited low variance due to its deterministic nature, consistently producing mediocre outcomes. In contrast, the GNN-based agent achieved significantly better average performance but with higher variance.

This variance reflects the *fragility of optimal containment strategies*. The learned policy relies on precisely identifying and blocking a small number of critical bridge nodes. If even one such node is missed due to stochastic effects, the rumor can escape, leading to a sudden spike in infections. This behavior highlights the high-risk, high-reward nature of precision intervention strategies and suggests that robustness should be a key consideration in real-world deployments.

Limitations and Practical Challenges

Despite promising results, several limitations must be acknowledged. First, all experiments were conducted on synthetic graph structures, which, while informative, may not fully capture the complexity of real-world social networks. Second, training and evaluation were performed on CPU-only hardware, constraining the scale of experiments and limiting the feasibility of computationally intensive methods such as large-budget MCTS.

Additionally, the action space was simplified by selecting nodes independently rather than optimizing over full subsets, which may restrict the expressiveness of learned policies. Finally, the reliance on carefully engineered features for GNN success suggests that performance may degrade if such features are unavailable or noisy in practical scenarios.

Summary

Overall, the discussion highlights that effective rumor containment in social networks requires both **sequential decision-making** and **structural awareness**. Learning-based approaches, particularly those incorporating graph structure, offer substantial advantages over fixed heuristics. However, their success depends on appropriate feature design, sufficient training at realistic difficulty levels, and careful consideration of stability–performance trade-offs.

Conclusion

This project investigated the problem of rumor containment in social networks through the lens of sequential decision-making under uncertainty. By modeling rumor spread as a stochastic process on graphs, we designed and evaluated multiple intervention strategies, ranging from simple heuristics to planning-based and learning-based artificial intelligence methods.

The experimental results demonstrate that while degree-based heuristics provide a fast and interpretable baseline, their effectiveness is limited in community-structured networks. Planning-based approaches such as Monte Carlo Tree Search exhibit strong potential but suffer from high variance and computational inefficiency under tight simulation budgets. In contrast, reinforcement learning agents, particularly those augmented with graph structural information, consistently achieved superior containment performance. The convergence of both DQN and GNN-based agents to the same infection limit further suggests that the learned policies approximate the theoretical optimum for the given network topology.

The project objectives were largely achieved. A flexible simulation environment was successfully implemented, enabling fair comparison across heuristic, planning, and learning paradigms. Learning-based agents demonstrated the ability to adapt their strategies over time and outperform static approaches, validating the suitability of reinforcement learning for rumor control. Additionally, the integration of graph neural networks highlighted the importance of structural awareness, especially in identifying critical bridge nodes that govern inter-community diffusion.

Despite these successes, several avenues for future work remain. The current experiments are limited to synthetic graph structures and CPU-only training, and future studies could explore larger-scale or real-world social networks with richer dynamics. Incorporating robustness-aware objectives, such as risk-sensitive or worst-case optimization, may reduce the variance observed in high-performing but fragile policies. Extensions to multi-agent coordination, partial observability, and dynamic intervention budgets also present promising directions. Finally, combining learning-based approaches with lightweight planning or rule-based constraints could offer a practical balance between performance, stability, and computational cost.

Overall, this work highlights the effectiveness of combining reinforcement learning with graph-aware representations for rumor containment and provides a foundation for further research into intelligent intervention strategies for complex networked systems.

References

Sutton, R. S., and Barto, A. G. 2018. *Reinforcement Learning: An Introduction*. MIT Press.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518(7540): 529–533.

Browne, C. B.; Powley, E.; Whitehouse, D.; et al. 2012. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games* 4(1): 1–43.

Kipf, T. N., and Welling, M. 2017. Semi-supervised classification with graph convolutional networks. In *Proceedings of ICLR*.

Hamilton, W.; Ying, Z.; and Leskovec, J. 2017. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Paszke, A.; Gross, S.; Massa, F.; et al. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*.

Fey, M., and Lenssen, J. E. 2019. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

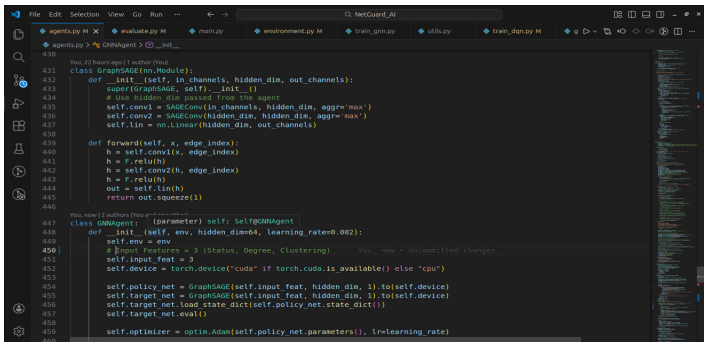
Hagberg, A.; Swart, P.; and S Chult, D. 2008. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference*.

Russell, S.; Abbeel, P.; and Klein, D. 2023. CS 188: Introduction to Artificial Intelligence. University of California, Berkeley. Course lecture notes and assignments.

Bishop, C. M. 2006. *Pattern Recognition and Machine Learning*. Springer.

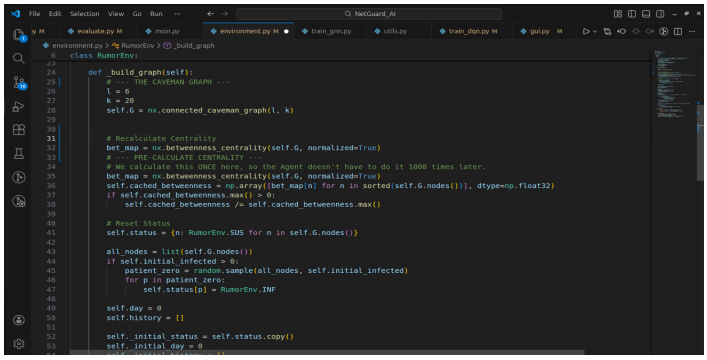
Appendix

A few code snippets are presented here.



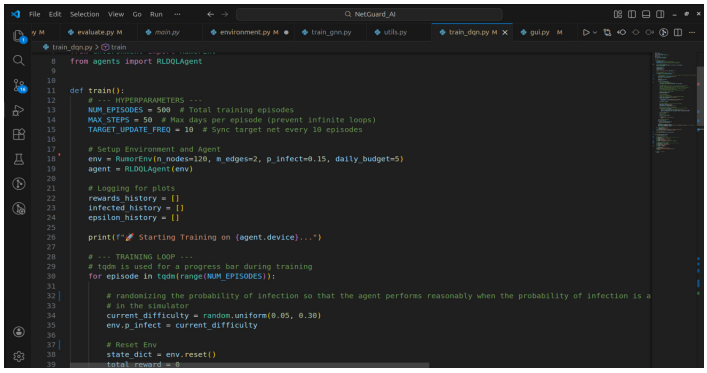
```python
class GraphSAGE(nn.Module):
    def __init__(self, in_channels, hidden_dim, out_channels):
        super(GraphSAGE, self).__init__()
        # Use hidden_dim passed from the agent
        self.conv1 = SAGEConv(in_channels, hidden_dim, aggr='max')
        self.conv2 = SAGEConv(hidden_dim, hidden_dim, aggr='max')
        self.lin = nn.Linear(hidden_dim, out_channels)

    def forward(self, x, edge_index):
        h = self.conv1(x, edge_index)
        h = F.relu(h)
        h = self.conv2(h, edge_index)
        h = F.relu(h)
        out = self.lin(h)
        return out.squeeze(1)

class GNNAgent:    (parameter) self: Self@GNNAgent
    def __init__(self, env, hidden_dim=64, learning_rate=0.002):
        self.env = env
        # Input features = 3 (Status, Degree, Clustering)
        self.input_feat = 3
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

        self.policy_net = GraphSAGE(self.input_feat, hidden_dim, 1).to(self.device)
        self.target_net = GraphSAGE(self.input_feat, hidden_dim, 1).to(self.device)
        self.target_net.load_state_dict(self.policy_net.state_dict())
        self.target_net.eval()

        self.optimizer = optim.Adam(self.policy_net.parameters(), lr=learning_rate)
```

Figure 7: Snippet-1



```python
class RumorEnv:

    def _build_graph(self):
        # --- THE CAVEMAN GRAPH ---
        l = 6
        k = 20
        self.G = nx.connected_caveman_graph(l, k)


        # Recalculate Centrality
        bet_map = nx.betweenness_centrality(self.G, normalized=True)
        # --- PRE-CALCULATE CENTRALITY ---
        # We calculate this ONCE here, so the Agent doesn't have to do it 1000 times later.
        bet_map = nx.betweenness_centrality(self.G, normalized=True)
        self.cached_betweenness = np.array([bet_map[n] for n in sorted(self.G.nodes())], dtype=np.float32)
        if self.cached_betweenness.max() > 0:
            self.cached_betweenness /= self.cached_betweenness.max()

        # Reset Status
        self.status = {n: RumorEnv.SUS for n in self.G.nodes()}

        all_nodes = list(self.G.nodes())
        if self.initial_infected > 0:
            patient_zero = random.sample(all_nodes, self.initial_infected)
            for p in patient_zero:
                self.status[p] = RumorEnv.INF

        self.day = 0
        self.history = []

        self._initial_status = self.status.copy()
        self._initial_day = 0
```

Figure 8: Snippet-2



```python
from agents import RLDQLAgent


def train():
    # --- HYPERPARAMETERS ---
    NUM_EPISODES = 500  # Total training episodes
    MAX_STEPS = 50  # Max days per episode (prevent infinite loops)
    TARGET_UPDATE_FREQ = 10  # Sync target net every 10 episodes

    # Setup Environment and Agent
    env = RumorEnv(n_nodes=120, m_edges=2, p_infect=0.15, daily_budget=5)
    agent = RLDQLAgent(env)

    # Logging for plots
    rewards_history = []
    infected_history = []
    epsilon_history = []

    print(f"🚀 Starting Training on {agent.device}...")

    # --- TRAINING LOOP ---
    # tqdm is used for a progress bar during training
    for episode in tqdm(range(NUM_EPISODES)):

        # randomizing the probability of infection so that the agent performs reasonably when the probability of infection is a
        # in the simulator
        current_difficulty = random.uniform(0.05, 0.30)
        env.p_infect = current_difficulty

        # Reset Env
        state_dict = env.reset()
        total_reward = 0
```

Figure 9: Snippet-3