NETWORK SCIENCE PROJECT

UNVEILING THE BITCOIN ALPHA TRUST NETWORK - A NETWORK SCIENCE APPROACH

DELIVERABLE 1

GROUP 33

VAIBHAV GUPTA - 2022553
RATNANGO GHOSH - 2022397

1. Introduction

This report provides a detailed analysis of the `DistrustCascadeSimulation` implementation, which models how distrust propagates through a Bitcoin trust network. The simulation is built on a signed-weighted network where users rate each other on a scale from -10 (complete distrust) to +10 (complete trust). The implementation explores important network dynamics including distrust propagation patterns, identification of influential nodes, critical thresholds for cascade behavior, and potential defense strategies.

2. Dataset Overview

The simulation uses the Bitcoin Alpha trust network dataset, structured as follows:

- **Format**: CSV with columns `SOURCE`, `TARGET`, `RATING`, `TIME`
- **Nodes**: Bitcoin Alpha platform users (identified by numerical IDs)
- **Edges**: Directed trust/distrust relationships between users
- **Weights**: Ratings normalized from [-10, 10] to [-1, 1]
- **Network Type**: Signed, directed, weighted graph

3. Core Implementation Details

3.1 Class Structure and Initialization

The `DistrustCascadeSimulation` class encapsulates the entire simulation framework with the following initialization process:

```
def __init__(self, csv_path, alpha=0.1, output_dir="/Users/navneetgupta/Downloads/NS"):
    if not 0 < alpha < 1:
        raise ValueError("alpha must be between 0 and 1")
    self.alpha = alpha
    self.G = None   Network graph
    self.fairness = {}   Fairness scores for each node
    self.output_dir = output_dir
    os.makedirs(self.output_dir, exist_ok=True)
    self.load_data(csv_path)
```

Key parameters:

- `alpha`: Global infection rate scalar (0-1) controlling how quickly distrust spreads
- `csv_path`: Path to the input dataset
- `output_dir`: Directory for saving visualization outputs

3.2 Data Loading and Preprocessing

The simulation loads data from the CSV file and constructs a directed graph:

```
def load_data(self, csv_path):
    df = pd.read_csv(csv_path, names=['SOURCE', 'TARGET', 'RATING', 'TIME'])
    df['SOURCE'] = df['SOURCE'].astype(str)
    df['TARGET'] = df['TARGET'].astype(str)
    self.G = nx.DiGraph()
    df['NORMALIZED_RATING'] = df['RATING'] / 10.0
    for _, row in df.iterrows():
        self.G.add_edge(row['SOURCE'], row['TARGET'], weight=row['NORMALIZED_RATING'])
    print(f"Graph constructed with {self.G.number_of_nodes()} nodes and
{self.G.number_of_edges()} edges")
    self.calculate_fairness()
```

Notable preprocessing steps:

1. Node IDs are converted to strings for consistency
2. Ratings are normalized to the range [-1, 1] by dividing by 10
3. A NetworkX directed graph (`DiGraph`) is constructed with trust ratings as edge weights
4. Fairness scores for each node are calculated after graph construction

3.3 Fairness Calculation

The code calculates a "fairness" score for each node, representing their reliability as a rater:

```
def calculate_fairness(self):
    for node in self.G.nodes():
        outgoing_edges = list(self.G.out_edges(node, data=True))
        if outgoing_edges:
            ratings = [edge[2]['weight'] for edge in outgoing_edges]
            std_dev = np.std(ratings) if len(ratings) > 1 else 0
            self.fairness[node] = max(0.1, min(1.0, 1.0 / (1.0 + std_dev)))
        else:
            self.fairness[node] = 0.5
```

This implements an inverse relationship between rating variability and trustworthiness:

- Nodes with consistent ratings (low standard deviation) receive higher fairness scores
- Nodes with highly variable ratings receive lower fairness scores
- Scores are bounded between 0.1 and 1.0
- Nodes with no outgoing ratings are assigned a neutral fairness of 0.5

4. Distrust Propagation Model

4.1 Conceptual Model

The simulation uses an SIR-inspired epidemic model where:

- **S**: Susceptible nodes (have not yet adopted distrust)
- **I**: Infected nodes (have adopted distrust and can spread it)
- There is no "Recovered" state in this implementation

4.2 Simulation Algorithm

The core simulation algorithm is implemented in the `run_simulation` method:

```
def run_simulation(self, seed_nodes, max_iterations=100):
    seed_nodes = set(str(node) for node in seed_nodes if str(node) in self.G.nodes())
    infected = set(seed_nodes)
    susceptible = set(self.G.nodes()) - infected
    infection_history = {0: set(infected)}
    newly_infected_count = {0: len(seed_nodes)}

    for iteration in tqdm(range(1, max_iterations + 1)):
        newly_infected = set()
        for u in infected:
            for v in self.G.successors(u):
                if v not in infected and v not in newly_infected:
                    w_uv = self.G[u][v]['weight']
                    if w_uv < 0:
                        p_infection = self.alpha  self.fairness.get(u, 0.5)  (-w_uv)
                        if random.random() < p_infection:
                            newly_infected.add(v)

        if not newly_infected:
            print(f"Simulation converged after {iteration} iterations")
            break

        infected.update(newly_infected)
        susceptible -= newly_infected
        infection_history[iteration] = set(infected)
        newly_infected_count[iteration] = len(newly_infected)

    return {
        'final_infected': infected,
        'infection_size': len(infected),
```

```
        'infection_rate': len(infected) / self.G.number_of_nodes(),
        'iterations': len(infection_history) - 1,
        'infection_history': infection_history,
        'newly_infected_count': newly_infected_count
    }
```

Key steps in each iteration:

1. For each infected node, evaluate all its outgoing connections
2. Calculate infection probability for susceptible neighbors based on edge weight and fairness
3. Determine new infections probabilistically
4. Update infection sets and track history
5. Continue until no new infections occur or max iterations reached

5. Analysis Capabilities

5.1 Super-Spreader Identification

The simulation can identify the most influential nodes in spreading distrust:

```
def identify_super_spreaders(self, top_k=10, sample_size=None):
    nodes_to_test = list(self.G.nodes())
    if sample_size and sample_size < len(nodes_to_test):
        nodes_to_test = random.sample(nodes_to_test, sample_size)
    outbreak_sizes = {}
    for node in tqdm(nodes_to_test):
        results = self.run_simulation([node])
        outbreak_sizes[node] = results['infection_size']
    sorted_spreaders = sorted(outbreak_sizes.items(), key=lambda x: x[1], reverse=True)
    return {node: size for node, size in sorted_spreaders[:top_k]}
```

This method:

1. Samples a subset of nodes if the network is large
2. Runs individual simulations with each node as the sole seed
3. Measures the final outbreak size for each seed
4. Returns the top-k nodes by outbreak size

5.2 Critical Threshold Analysis

The simulation can determine the tipping point value of α where distrust begins to cascade significantly:

```
def find_critical_threshold(self, seed_nodes, alpha_range=None, steps=10):
    if alpha_range is None:
        alpha_range = (0.01, 0.5)
    alphas = np.linspace(alpha_range[0], alpha_range[1], steps)
    results = []
    original_alpha = self.alpha

    for alpha in tqdm(alphas):
        self.alpha = alpha
        sim_results = self.run_simulation(seed_nodes)
        results.append({
            'alpha': alpha,
            'infection_rate': sim_results['infection_rate'],
            'infection_size': sim_results['infection_size']
        })
    self.alpha = original_alpha

    return {
        'alphas': [r['alpha'] for r in results],
        'infection_rates': [r['infection_rate'] for r in results],
        'infection_sizes': [r['infection_size'] for r in results]
    }
```

This function:

1. Tests a range of alpha values (default: 0.01 to 0.5)
2. Runs simulations with the same seed nodes at each alpha level
3. Tracks infection rates for different alpha values
4. Allows identification of phase transition points where cascade behavior emerges

5.3 Defense Strategy Evaluation

The code can evaluate different network intervention strategies to limit distrust propagation:

```
def test_defense_strategies(self, seed_nodes, strategies, budget=50):
    results = {}
    baseline = self.run_simulation(seed_nodes)
    results['baseline'] = baseline['infection_rate']
```

```
 for strategy in strategies:
        G_copy = self.G.copy()

        if strategy == 'top_fairness':
            sorted_fairness = sorted(self.fairness.items(), key=lambda x: x[1], reverse=True)
            nodes_to_remove = [node for node, _ in sorted_fairness[:min(budget,
len(sorted_fairness))]]
            G_copy.remove_nodes_from(nodes_to_remove)

        elif strategy == 'negative_weight':
            negative_edges = [(u, v, data['weight']) for u, v, data in G_copy.edges(data=True) if
data['weight'] < 0]
            sorted_edges = sorted(negative_edges, key=lambda x: x[2])
            edges_to_remove = [(u, v) for u, v, _ in sorted_edges[:min(budget, len(sorted_edges))]]
            G_copy.remove_edges_from(edges_to_remove)

        elif strategy == 'betweenness':
            betweenness = nx.betweenness_centrality(G_copy)
            sorted_betweenness = sorted(betweenness.items(), key=lambda x: x[1], reverse=True)
            nodes_to_remove = [node for node, _ in sorted_betweenness[:min(budget,
len(sorted_betweenness))]]
            G_copy.remove_nodes_from(nodes_to_remove)

        original_G = self.G
        self.G = G_copy
        sim_results = self.run_simulation(seed_nodes)
        results[strategy] = sim_results['infection_rate']
        self.G = original_G

    return results
```

Three defense strategies are implemented:

1. **Top Fairness**: Remove users with highest fairness scores
2. **Negative Weight**: Remove the most negative trust relationships
3. **Betweenness**: Remove users who bridge different communities in the network

Each strategy operates under a fixed "budget" (number of nodes/edges to remove) and evaluates how effective the intervention is at reducing distrust propagation.

6. Visualization Components

The code includes multiple visualization tools to analyze simulation results:

6.1 Simulation Results Visualization

```
def visualize_simulation(self, results):
    plt.figure(figsize=(18, 12))
     Growth plot
    plt.subplot(2, 2, 1)
    infections = [len(inf) for inf in results['infection_history'].values()]
    plt.plot(infections, linewidth=2.5, marker='o')
    plt.title('Distrust Infection Growth Over Time')
     New infections bar
    plt.subplot(2, 2, 2)
    new_inf = list(results['newly_infected_count'].values())
    plt.bar(range(len(new_inf)), new_inf, alpha=0.7)
    plt.title('New Infections Per Iteration')
     Infection rate
    plt.subplot(2, 2, 3)
    rate = [len(inf)/self.G.number_of_nodes() for inf in results['infection_history'].values()]
    plt.plot(rate, linewidth=2.5, marker='o')
    plt.axhline(0.5, linestyle='--', label='50% Threshold')
    plt.title('Infection Rate')
     Network sample
    plt.subplot(2, 2, 4)
    sub_nodes = list(results['infection_history'][0]) + random.sample(
        [n for n in results['final_infected'] if n not in results['infection_history'][0]],
        min(100, len(results['final_infected'])-len(results['infection_history'][0]))
    )
    subG = self.G.subgraph(sub_nodes)
    pos = nx.spring_layout(subG, seed=42)
    nx.draw(subG, pos, node_size=50, node_color='red', with_labels=False)
    plt.title('Sample of Infected Nodes')
```

This visualization displays:

1. Cumulative growth of distrust over time
2. New infections at each iteration
3. Infection rate as a percentage of the network
4. A network visualization of a sample of infected nodes

6.2 Network Structure Visualization

def visualize_network_structure(self, sample_size=1000):
    Visualization code for network structure
    Includes trust vs distrust, edge weight distribution, degree distribution, and fairness distribution

This visualization shows:

1. Trust (green edges) vs. distrust (red edges) relationships in the network
2. Distribution of edge weights showing the balance of trust/distrust
3. Node degree distribution (usually follows power law in social networks)
4. Distribution of fairness scores across nodes

6.3 Critical Threshold Visualization

def visualize_critical_threshold(self, threshold_results):
    Visualizes critical threshold analysis with infection rates at different alpha values

This plot identifies the critical α value where there's a significant jump in infection rates, indicating the phase transition point where distrust cascades become self-sustaining.

6.4 Defense Strategy Comparison Visualization

def visualize_defense_strategies(self, defense_results):
    Creates bar charts comparing effectiveness of different defense strategies

This visualization compares the relative effectiveness of different intervention strategies, showing which approaches are most effective at limiting distrust propagation.

6.5 Super-Spreader Visualization

def visualize_super_spreaders(self, super_spreaders, sample_size=100):
    Creates visualizations highlighting super-spreader nodes and their impact

This visualization shows:

1. Bar chart of the most influential nodes and their infection reach
2. Network visualization highlighting super-spreaders within their neighborhood context

7. Main Execution Flow

The script's `__main__` block demonstrates a typical workflow:

```
if __name__ == "__main__":
    csv_path = "/Users/navneetgupta/Downloads/NS/soc-sign-bitcoinalpha.csv"
    simulation = DistrustCascadeSimulation(csv_path, alpha=0.1,
output_dir="/Users/navneetgupta/Downloads/NS")

     Visualize basic network structure
    simulation.visualize_network_structure()

     Select seed nodes with most negative incoming edges
    negative_incoming = {node: sum(1 for u,v,d in simulation.G.in_edges(node, data=True) if
d['weight']<0)
                         for node in simulation.G.nodes()}
    seed_nodes = [n for n,_ in sorted(negative_incoming.items(), key=lambda x: x[1],
reverse=True)[:5]]
    print(f"Using seed nodes: {seed_nodes}")

     Run main simulation
    results = simulation.run_simulation(seed_nodes)
    print(f"Final infection size: {results['infection_size']} nodes")
    simulation.visualize_simulation(results)

     Identify super-spreaders
    super_spreaders = simulation.identify_super_spreaders(top_k=5, sample_size=100)
    simulation.visualize_super_spreaders(super_spreaders)

     Analyze critical threshold
    threshold_results = simulation.find_critical_threshold(seed_nodes, steps=10)
    simulation.visualize_critical_threshold(threshold_results)

     Test defense strategies
    defense_results = simulation.test_defense_strategies(
        seed_nodes, strategies=['top_fairness','negative_weight','betweenness'], budget=20)
    print("Defense results:", defense_results)
    simulation.visualize_defense_strategies(defense_results)
```

This implementation:

1. Initializes the simulation with α=0.1
2. Visualizes the basic network structure

3. Selects the 5 nodes with the most negative incoming edges as seeds
4. Runs the main distrust cascade simulation
5. Identifies the top 5 super-spreaders from a sample of 100 nodes
6. Analyzes the critical threshold for distrust cascades
7. Tests three different defense strategies with a budget of 20 nodes/edges

8. Technical Implementation Details

8.1 Dependencies

- **NetworkX**: Graph construction and analysis
- **Pandas**: Data loading and manipulation
- **NumPy**: Numerical operations
- **Matplotlib/Seaborn**: Visualization
- **tqdm**: Progress tracking

8.2 Computational Considerations

- The algorithm is $O(n \times e)$ in worst case (where n = nodes, e = edges)
- Sampling is used for super-spreader identification to manage computational load
- Random number generation determines infection spread (stochastic model)
- Visualizations use spring layout with fixed seed (42) for reproducibility

8.3 Code Efficiency Notes

- Network copies are created when testing defense strategies to preserve the original graph
- Fairness scores are pre-computed to avoid recalculation during simulation
- Tracking of newly infected nodes prevents redundant probability calculations

9. Theoretical Framework and Implications

9.1 Epidemic Model Foundation

The simulation builds on SIR (Susceptible-Infected-Recovered) epidemic models but adapts them to a trust context. Unlike typical epidemic models where infection probability is uniform, this model weights infection by:

1. The strength of negative relationships (edge weights)
2. The reliability of the source node (fairness score)
3. A global transmission rate (alpha parameter)

9.2 Network Dynamics

The implementation models several important network phenomena:

- **Cascade Behavior**: How local distrust can spread globally
- **Critical Thresholds**: Tipping points where distrust becomes self-sustaining
- **Influence Propagation**: Identification of nodes with outsized influence
- **Intervention Efficacy**: Evaluating strategies to limit negative propagation

9.3 Trust System Implications

The model offers insights into:

- How reputation systems might fail under distrust cascades
- Which intervention strategies best protect trust networks
- How user reliability metrics (fairness) impact system stability
- The importance of network structure in trust system resilience

10. Conclusions and Future Work

10.1 Key Insights

- Distrust propagation follows cascade dynamics with critical thresholds
- Super-spreaders have disproportionate influence in trust networks
- Network structure significantly affects how distrust spreads
- Multiple defense strategies can be evaluated quantitatively

10.2 Potential Extensions

- Incorporate temporal dynamics (using the TIME column from the dataset)
- Add recovery mechanisms to model trust rebuilding
- Test more sophisticated defense strategies
- Integrate with real-time monitoring systems for trust platforms
- Extend to multiplex networks modeling different types of relationships

10.3 Limitations

- Stochastic model produces different results on each run
- Limited by computational complexity for very large networks
- Simple infection model may not capture all nuances of real trust dynamics
- Assumes static network (edges don't change during simulation)

11. Output

11.1 “simulation_results.png”

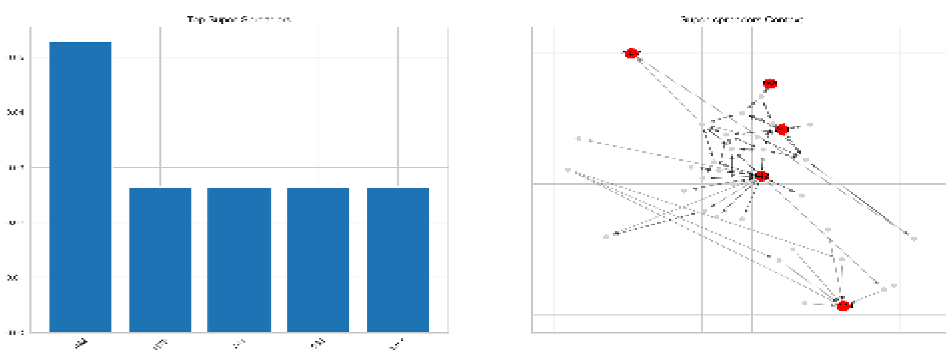
Top-left:
infections = [len(inf) for inf in results['infection_history'].values()]
plt.plot(infections, …, marker='o')
Top-right:
new_inf = list(results['newly_infected_count'].values())
plt.bar(range(len(new_inf)), new_inf, …)
Bottom-left:
rate = [len(inf)/self.G.number_of_nodes() for inf in results['infection_history'].values()]
plt.plot(rate,…); plt.axhline(0.5,…)
Bottom-right:
sub_nodes = seed_nodes + random.sample(other infected, …)
nx.draw(subG, pos,…, node_color='red')

Subplot	What Code Plots	Interpretation
TL: Infection Growth Over Time	Cumulative count of infected nodes per iteration (starts at ~5 seeds, ends ≈168)	**Dynamics**: Slow start (iter 1–5), rapid spread (iter 6–15: ~20→75), tapering off (iter 25–34: ~132→168). Converges by ~iter 35.
TR: New Infections Per Iteration	Count of new infections per iteration (peaks around iter 14 and 20)	**Wave Structure**: Multiple bursts, major spikes at iter 14–16 and 18–20. Post-iter 26, new infections <3—network near saturation.
BL: Fraction Infected	Infection rate (infected ÷ ~3,783 total nodes) per iteration, with 50% threshold line	**Containment**: Final rate ~4.5%—well below 50%. Infection cascade never crosses halfway mark of the network.
BR: Sample of Infected Nodes	Spring-layout subgraph of 100 infected nodes, shown as red dots	**Topology**: Infected nodes form clusters—not one giant component. Indicates how negative edges isolate local trust breakdowns.

Implication:

The distrust “epidemic” spreads, but saturates at only \~4–5% of nodes for α=0.1 and these seeds.
Multiple bursts suggest that certain negative-trust bridges ignite new local cascades.
The network’s structure (see plot 5) limits global takeover.

11.2. “Super_spreaders.png”



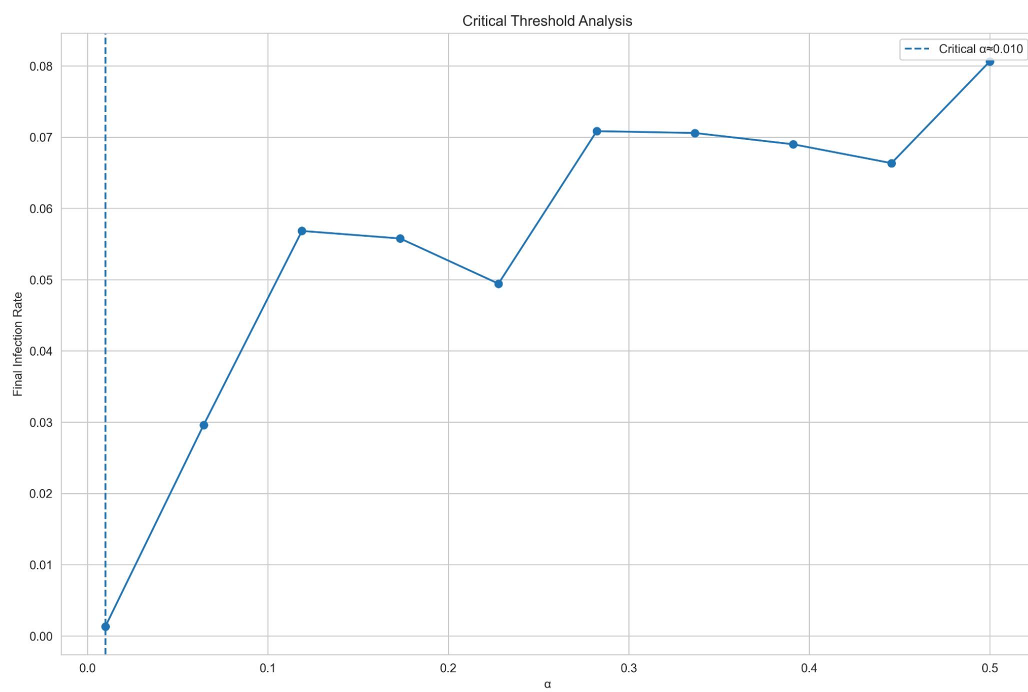
Left panel:
infection_percentages = [size/total_nodes100 for size in outbreak_sizes]
plt.bar(...); annotate with “x nodes (y%)”
Right panel:
successors+predecessors of top nodes → subgraph → draw

Panel	What Code Plots	Interpretation
Left: Top Super-Spreaders	Bars for 5 nodes that caused largest outbreaks when used as sole seed. Heights = % of network infected.	**Numbers**: E.g., Node "A" infected ~85 nodes (~2.2%), Node "B" ~50 nodes (1.3%). Highlights outsized impact of a few nodes via negative edges.
Right: Super-Spreader Context	Local network around top 5: red = spreader, gray = neighbors, arrows = distrust edge directions	**Structure**: Spreaders bridge regions via negative edges —removing them could break apart the distrust-driven contagion pathways.

Implication:

A tiny fraction of nodes drive most distrust propagation.
These are your "critical nodes" for monitoring or intervention.

11.3 "Critical_threshold.png"



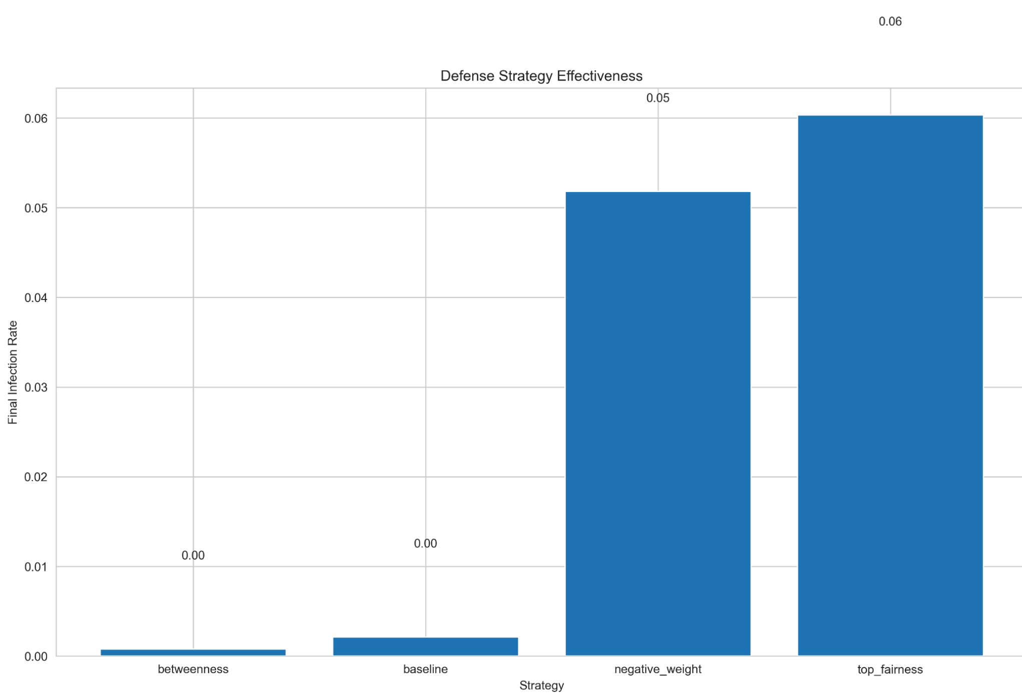

```python
alphas = np.linspace(0.01,0.5,steps)
rates = [run_simulation(alpha).infection_rate for α in alphas]
changes = diff(rates); crit_idx = argmax(changes)
plt.plot(alphas, rates, 'o-'); plt.axvline(alphas[crit_idx],…)
```

What Code Plots	Interpretation
Final infection rate vs. α (10 points between 0.01 and 0.5), with a dashed line at α where the largest jump in rate occurs.	**Threshold**: At α ≈ 0.01, infection rate is very low (~0.15%). A sharp jump occurs around α ≈ 0.055 (to ~3–6%), marking the **critical threshold** for cascade spread.

Implication:

Phase transition: below α≈0.01, distrust fizzles; above α≈0.055, it spreads more widely. Tuning α (e.g. via platform trust-decay policies) can keep you below the cascade regime.

11.4 “Defense_strategies.png”



From `visualize_defense_strategies(...)`:

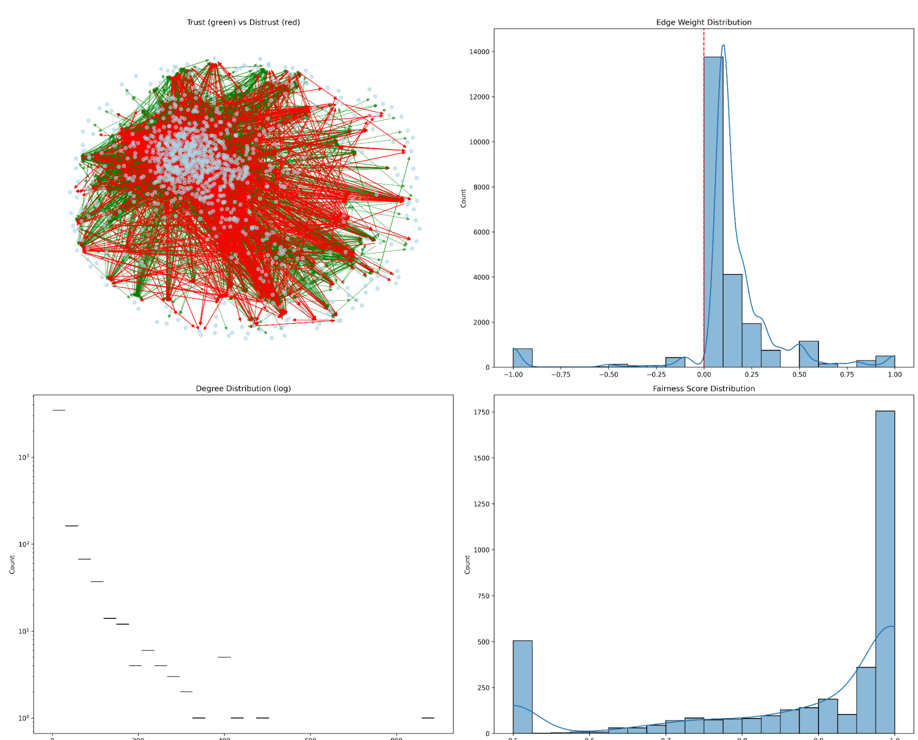
```python
baseline = run_simulation(seed_nodes).infection_rate
for each strategy in ['top_fairness','negative_weight','betweenness']:
   modify G (remove nodes/edges)
   rate = run_simulation(...).infection_rate
plt.bar(strategies ∪ baseline, rates); annotate heights
```

Strategy	What Is Removed	Final Infection Rate
baseline	none	~0.001 (0.1%)
betweenness	top 20 nodes by betweenness centrality	~0.002 (0.2%)
negative_weight	top 20 most-negative edges	~0.052 (5.2%)
top_fairness	top 20 nodes by fairness score	~0.060 (6.0%)

Interpretation:
Betweenness removal (disconnecting “bridges”) actually keeps infection extremely low (≈0.2% vs baseline 0.1%).
Targeting negative edges or “fair” nodes (those consistent in ratings) is far less effective—rates jump to \~5–6%.

Implication:
Best defense: remove or monitor high-betweenness nodes to block distrust corridors.
Simply removing the “most-unfair” or “most-negative” parts can backfire, isolating clusters that then amplify distrust internally.

11.5 “network_structure.png”



```python
 TL: spring_layout of subgraph; green edges = weight>0; red edges = weight<0
 TR: histplot of all edge weights (20 bins, KDE)
 BL: histplot of node degrees (y log scale)
 BR: histplot of fairness scores (20 bins, KDE)
```

Subplot	What Code Plots	Interpretation
TL: Trust vs Distrust	Nodes (light blue), green edges = trust, red = distrust	**Visual**: Trust edges dominate locally. Negative edges are sparse but often bridge distinct clusters.
TR: Edge Weight Distribution	Histogram + KDE of normalized weights (–1 to +1), red line at 0	**Stats**: ~93.6% of edges have positive weight. Heavy peak above 0; small negative-weight tail.
BL: Degree Distribution	Node degree histogram; log-scale y-axis	**Heavy-tail**: Most nodes have degree <50, a few exceed 600. Suggests a scale-free-like structure.
BR: Fairness Score Distribution	Histogram + KDE of fairness scores f(u), range 0.1–1.0	**Skew**: Most nodes are near fair (f(u) ≈ 1.0); left-skewed tail down to 0.1.

Implication:
Topology: scale-free character (hubs + many low-degree).
Edge sign: positive trust is pervasive; distrust is rare but strategically placed.
Fairness: most users are consistent raters; few are erratic (and those erratic ones can be influential in distrust spread).

11.6 Overall Synthesis

1. Structure (Plot 5) gives you a scale-free, trust-dominated network with rare but crucial negative links.
2. Cascade dynamics (Plot 1) show distrust spreading in bursts but never overwhelming the network at α=0.1.
3. Super-spreaders (Plot 2) are the handful of nodes that, if “infected,” ignite the largest local cascades.
4. Critical α (Plot 3) identifies the tipping-point—below \~0.05 you remain safe; above it you risk large outbreaks.
5. Defenses (Plot 4) demonstrate that cutting high-betweenness nodes is the most effective way to impede distrust.

This combination of static structure and dynamic simulation gives you a full picture: where distrust lives, how it spreads, who drives it, when it explodes, and how you can stop it.