TrustRank Algorithm for Fraud Detection in Payment Networks Using Pregel Framework

Sakshi Badole CS24MTECH11008
Laveena Herman CS24MTECH11010
S Anjana Shankar CS24MTECH14015

February 28, 2025

Abstract

This report presents an implementation of the TrustRank algorithm for fraud detection in payment networks. The implementation leverages Google's Pregel computational model for distributed graph processing. The algorithm identifies potentially fraudulent entities by propagating distrust scores through a payment transaction network, using known fraudulent entities as seed nodes. We discuss the algorithm design, implementation details, and analyze the results using various visualization techniques.

1 Introduction

Fraud detection in payment networks is a critical application of graph analytics. By modeling payment transactions as a directed weighted graph, we can identify potentially fraudulent entities by their connection patterns to known bad actors. The TrustRank algorithm, originally developed for identifying spam web pages, has been adapted here for fraud detection in payment networks.

The implementation presented in this report uses the Pregel computational model, a distributed graph processing framework that follows a vertex-centric approach. This enables efficient processing of large-scale payment networks through parallel computation.

2 Algorithm Overview

TrustRank operates on the principle that fraudulent entities are more likely to be connected to other fraudulent entities. The algorithm starts with a set of known fraudulent entities (seed nodes) and propagates distrust scores through the network based on transaction patterns.

3 Implementation Architecture

Our implementation consists of three main components:

3.1 Vertex Class

The Vertex class represents each entity in the payment network. It maintains:

- Entity identifier
- Current distrust score
- Outgoing connections (recipients of payments)
- Edge weights (payment amounts)
- Computation state (active/inactive)
- Message queues for communication

Each vertex processes incoming messages during the update phase and generates outgoing messages during the propagation phase:

Algorithm 1 TrustRank for Fraud Detection

Input: Graph $G(V, E)$, seed set $S \subset V$, damping factor α
Output: Distrust scores for all vertices
Initialize distrust scores: $d(v) = \begin{cases} \frac{1}{|S|} & \text{if } v \in S \\ 0 & \text{otherwise} \end{cases}$
while not converged **do**
  for each vertex $v \in V$ **in parallel do**
    $d_{new}(v) = 0$
    for each incoming edge $(u, v) \in E$ **do**
      w_{uv} = weight of edge (u, v)
      W_u = sum of all outgoing weights from u
      $d_{new}(v) += d(u) \cdot \frac{w_{uv}}{W_u}$
    end for
    $d_{new}(v) = \alpha \cdot d_{new}(v) + \frac{(1-\alpha)}{|S|} \cdot 1_S(v)$
  end for
  Handle sink nodes by redistributing to seed nodes
  $d \leftarrow d_{new}$
end while
Normalize scores to sum to 1.0
return d

```python
def update(self):
    """Updates the distrust score of
        this vertex."""
    alpha = 0.85  # Damping factor

    if self.superstep == 0:
        self.propagate_scores()
    elif self.superstep < 100:  # Max
        iterations
        propagation_score = sum(score
            for (_, score) in self.
            incoming_messages) if self.
            incoming_messages else 0
        damped_score = alpha *
            propagation_score
        teleport_score = (1.0 - alpha) /
             self.total_seed_count if
            self.is_seed else 0

        self.value = damped_score +
            teleport_score
        self.propagate_scores()
    else:
        self.active = False
```

3.2 Worker Class

The Worker class enables parallel processing by distributing vertices across multiple threads:

```python
class Worker(threading.Thread):
    """Worker thread that processes
        vertices in a superstep."""
    def __init__(self, vertices):
        threading.Thread.__init__(self)
        self.vertices = vertices

    def run(self):
        for vertex in self.vertices:
            if vertex.active:
                vertex.update()
```

3.3 Pregel Class

The Pregel class coordinates the computation by:

- Partitioning vertices among workers

- Executing supersteps
- Redistributing messages between supersteps
- Handling sink nodes
- Normalizing final scores

```python
def run(self, max_iterations=100):
    """Runs the Pregel instance until convergence or max iterations."""
    current_iteration = 0

    while self.check_active() and current_iteration < max_iterations:
        self.superstep()
        self.redistribute_messages()
        current_iteration += 1

    total_score = sum(v.value for v in self.vertices)
    print(f"Iteration {current_iteration}: Sum of scores = {total_score:.6f}")

    # Only normalize after all iterations are complete
    self.normalize_final_scores()
```

4 Data Processing Pipeline

The implementation follows a structured pipeline:

4.1 Graph Construction

The payment network is constructed from transaction data:

- Entities (senders and receivers) become vertices
- Transactions become weighted edges
- Known fraudulent entities are marked as seed nodes

A key aspect of the implementation is the edge direction. In TrustRank for fraud detection, edges flow backward from receivers to senders. This ensures that entities receiving payments from fraudulent entities inherit distrust scores.

4.2 Algorithm Execution

The TrustRank algorithm is executed using the Pregel framework with a specified number of worker threads. The computation proceeds in supersteps until convergence or a maximum number of iterations.

4.3 Result Analysis

After execution, the results are analyzed to identify potential fraudulent entities:

- Entities are ranked by distrust score
- Percentile thresholds (90th, 95th) are calculated
- Entities above thresholds are flagged as potentially fraudulent

5 Handling Special Cases

5.1 Sink Nodes

Sink nodes (entities that only send payments but don't receive any) can cause score leakage in the network. Our implementation handles this by redistributing sink node scores equally among the seed nodes:

```python
def redistribute_messages(self):
    """Redistributes messages between vertices after a superstep."""
    for vertex in self.vertices:
        vertex.superstep += 1
        vertex.incoming_messages = []

    if self.has_sinks and self.seed_vertices:
        # For sink nodes, redistribute their value equally to seed nodes
        for sink in self.sink_vertices:
```

```
            for seed in self.
                seed_vertices:
                seed.incoming_messages.
                    append(
                    (sink, sink.value /
                        len(self.
                        seed_vertices)))

    for vertex in self.vertices:
        for (receiving_vertex, message)
            in vertex.outgoing_messages:
            receiving_vertex.
                incoming_messages.append
                ((vertex, message))
```

5.2 Score Normalization

To ensure the final scores sum to 1.0, any difference is distributed to seed nodes:

```
def normalize_final_scores(self):
    """Ensures the final scores sum to
        1.0 by distributing any
        difference to seed nodes."""
    total_score = sum(vertex.value for
        vertex in self.vertices)
    print(f"Final sum before
        normalization: {total_score:.6f}
        ")

    if abs(total_score - 1.0) > 0.01:
        difference = 1.0 - total_score

        if difference != 0 and self.
            seed_vertices:
            print(f"Distributing
                difference of {
                difference:.6f} to {len(
                self.seed_vertices)}
                seed nodes")
            # Distribute the difference
                equally among seed nodes
            seed_adjustment = difference
                / len(self.
                seed_vertices)
            for vertex in self.
                seed_vertices:
                vertex.value +=
                    seed_adjustment
```

6 Visualization and Analysis

Several visualization techniques are used to analyze the results:

6.1 Entity Scatter Plot

This visualization shows the distribution of distrust scores across entities, with known bad senders highlighted in red and threshold lines marking potential fraud detection boundaries.

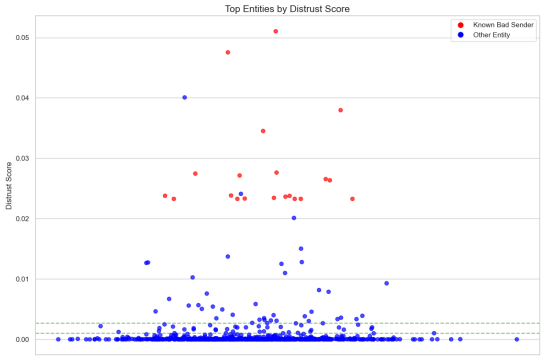

Figure 1: Top entities by distrust score with threshold indicators

6.2 Cumulative Distribution Function

The CDF plot shows the distribution of scores across all entities and helps identify appropriate threshold values for flagging potential fraud.

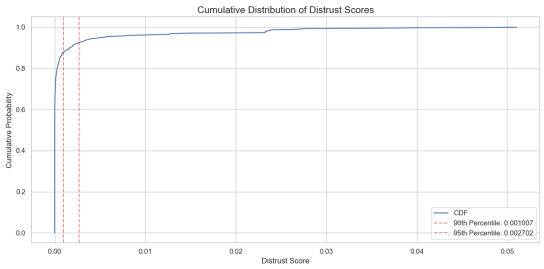

Figure 2: Cumulative distribution of distrust scores

6.3 Log-Scale Histogram

This visualization reveals the score distribution across multiple orders of magnitude, which is typical in fraud detection scenarios where most entities have very low scores while a few have significantly higher scores.

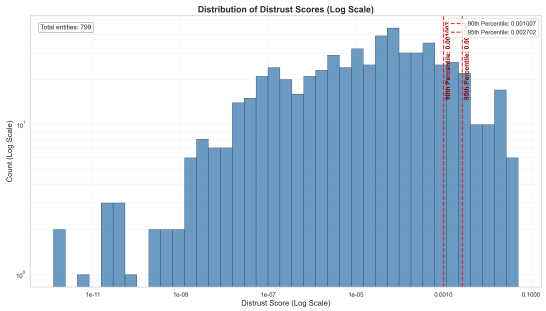

Figure 3: Distribution of distrust scores (log scale)

7 Implementation Highlights

7.1 Weighted Edge Propagation

The implementation accounts for transaction amounts as edge weights. Entities that receive larger payments from fraudulent entities inherit proportionally larger distrust scores:

```
def propagate_scores(self):
    """Propagate scores to outgoing
       vertices based on weighted edges
       ."""
    self.outgoing_messages = []

    if not self.out_vertices:
        return  # Sink nodes handled in
            EnhancedPregel.
            redistribute_messages

    # Get total outgoing weight
    total_weight = sum(self.out_weights)

    if total_weight > 0:
        # Distribute score
            proportionally to edge
            weights
        self.outgoing_messages = [
            (vertex, self.value * (
                weight / total_weight))
            for vertex, weight in zip(
                self.out_vertices, self.
                out_weights)
        ]
    else:
        # If all weights are zero,
            distribute equally
        score_per_edge = self.value /
            len(self.out_vertices)
        self.outgoing_messages = [(
            vertex, score_per_edge) for
            vertex in self.out_vertices]
```

7.2 Parallel Processing

The Pregel model enables parallel processing of vertices, making the implementation scalable for large networks:

```
def superstep(self):
    """Executes a single superstep
       across workers."""
    workers = []
    for vertex_list in self.
        partition_vertices().values():
        worker = Worker(vertex_list)
        workers.append(worker)
        worker.start()
    for worker in workers:
        worker.join()
```

8 Experimental Results

The algorithm successfully identifies potentially fraudulent entities in the payment network. The results include:

- Known bad senders (seed nodes) receive high distrust scores
- Entities frequently transacting with known bad senders also receive elevated scores

- Threshold analysis identifies new potentially fraudulent entities

8.1 Statistical Thresholds

Two statistical thresholds were used to identify potential fraud:

- 90th percentile: Entities with distrust scores above the 90th percentile are flagged as suspicious
- 95th percentile: Entities with distrust scores above the 95th percentile are flagged as highly suspicious

Entities identified through these thresholds can be prioritized for further investigation.

9 Conclusion

The TrustRank algorithm implemented using the Pregel computational model provides an effective approach for fraud detection in payment networks. By propagating distrust scores from known fraudulent entities through the transaction network, the algorithm identifies other entities that may be involved in fraudulent activities.

Key advantages of this approach include:

- Ability to process large-scale payment networks
- Incorporation of transaction amounts as edge weights
- Identification of potential fraud based on network connections
- Parallel processing for computational efficiency

Future work could include incorporating additional features such as transaction frequency, temporal patterns, and entity attributes to enhance the fraud detection capability.