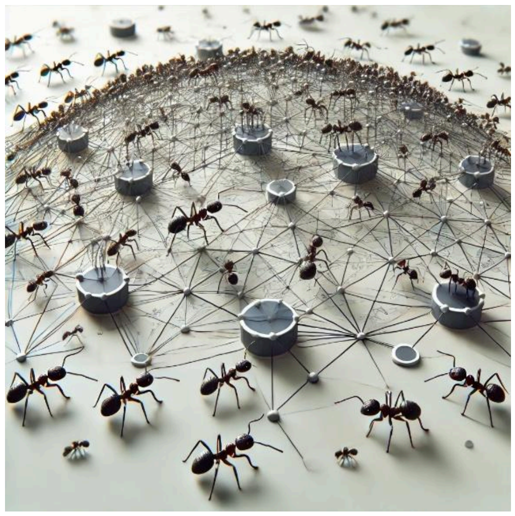
Citation Prediction using Ant Colony based Multi-Level Network Embedding



Programmer Manual

Version 1.0

28/01/2025

Table of Contents

1. Introduction

Welcome to the **Developer Manual** for the **Citation Prediction using Ant Colony-based Multi-Level Network Embedding (ACE)** system. This manual provides essential guidance for understanding, maintaining, and extending the system.

The ACE system combines Ant Colony Optimization (ACO) and graph embedding techniques to analyze and improve citation relevance in academic networks. This document offers a high-level overview of the architecture, core algorithms, and performance optimization strategies, ensuring developers can work efficiently with the system.

Whether you're debugging, adding features, or optimizing performance, this manual equips you with the knowledge needed to support and enhance the ACE system effectively.

2. System Overview

2.1 High-Level Architecture

The system is designed to address citation prediction using **Multi-Level Network Embedding (MLNE)** and **Ant Colony Optimization (ACO)**. It consists of modular components that handle distinct responsibilities, making it easy to maintain and extend.

Key Components:

1. **UI Layer**:
   - UI.py provides a user-friendly interface for configuring parameters, loading datasets, and triggering the evaluation process.
   - It communicates with the backend logic in a separate thread while maintaining UI responsiveness.
2. **Logic Layer**:
   - Core algorithms and processes are executed in main_code.py and its supporting modules (evaluation_plan.py, research_plan.py, etc.).
   - Handles the heavy lifting, including graph coarsening, embedding, and similarity calculations.
3. **Config Layer**:
   - The Config module centralizes parameter management, providing flexibility for customization and experimentation.

2.2 Overview of Multi-Level Network Embedding (MLNE)

The **MLNE** algorithm constructs a hierarchical graph representation by coarsening the input graph into a multi-level pyramid structure. At each level, embeddings are generated using techniques such as Node2Vec. These embeddings are concatenated across layers to create a unified representation of the graph.

Key Steps:

1. **Graph Coarsening**:
   - Groups nodes into clusters based on the strength of their relationships, identified through ACO-based random walks.
   - Reduces graph complexity by creating a sequence of progressively simplified graphs (layers).
2. **Multi-Scale Embedding**:

   - Embeddings are generated at each coarsened layer using Node2Vec, capturing both local and global graph structures.

3. **Concatenation**:
   - Embeddings from all layers are combined to produce the final multi-level embedding matrix.

2.3 Ant Colony Optimization (ACO) and Graph Coarsening

The **Ant Colony Optimization (ACO)** algorithm simulates the behavior of ants depositing pheromones as they explore paths. In this system:

- **Ant Colony Walking (Algorithm 2)**: Identifies strong connections in the graph by detecting short loops with high pheromone levels.
- **Adaptive Threshold Selection (Algorithm 3)**: Dynamically determines thresholds to classify edges as "strong" or "weak."
- **Graph Clustering Pyramid (Algorithm 4)**:
  - Merges strongly connected nodes into clusters, forming a coarsened graph at each level.
  - This process is repeated until the graph reaches the desired level of abstraction.

2.4 Threading Structure

The system operates on a **two-thread model** to separate user interaction from backend processing.

Interaction Between UI.py and main_code.py:

- UI.py acts as the entry point, where users configure parameters, load datasets, and trigger processes.
- A new thread is created to execute main_code.py, which handles all computational tasks without blocking the UI.

Responsibilities of UI and Logic Threads:

1. **UI Thread**:
   - Ensures smooth user interaction.
   - Displays progress and allows users to stop or restart processes.
2. **Logic Thread**:
   - Executes the Evaluation Plan and Research Plan algorithms.
   - Manages computationally intensive tasks like graph coarsening, embedding generation, and similarity calculation.

2.5 Core Workflow

UI Initialization and Input Gathering

- Users interact with UI.py to:
  - Select datasets (edge list and label list).
  - Configure hyperparameters such as embedding dimensions, thresholds, and iteration counts.
  - Trigger the evaluation process.

Execution of Evaluation and Research Plans

1. **Evaluation Plan**:
   - Randomly adds edges to the graph for testing purposes.
   - Executes the Research Plan and calculates the success rate.
2. **Research Plan**:
   - Coarsens the graph and generates embeddings using MLNE.
   - Constructs similarity and statistical matrices.
   - Refines the graph by removing edges with weak relationships.

At the end of the workflow, results are saved (e.g., plots, histograms, and success rates), and the user can visualize the process output through the UI.

3. Environment Setup

This section provides instructions for setting up the environment to run the system effectively, including hardware requirements, installation steps, and configuration management.

3.1 System Requirements

To ensure smooth execution, especially for resource-intensive tasks like graph coarsening and embedding, the following system specifications are recommended:

Minimum Requirements:

- **Memory**: 8 GB RAM
- **Storage**: 10 GB free disk space
- **GPU**: Not required, but may lead to slower execution.

Recommended Hardware:

- **Memory**: 16 GB RAM or higher
- **Storage**: SSD with at least 20 GB free space
- **GPU**: NVIDIA CUDA-enabled GPU with at least 4 GB VRAM (e.g., NVIDIA GTX 1660 or higher).

3.2 Dependencies and Installation

To set up the environment, follow these steps:

1. Open a terminal or command prompt.
2. Clone the repository from GitHub:
```
$ git clone https://github.com/yoni4600/Final-Project.git
```
3. Open the project in Python IDE with python version 3.9.0:
```
$ cd src
$ pip install -r requirements.txt
```
4. Build:
```
$ python setup.py build_ext --inplace
```
5. Open the UI:
```
$ python UI.py
```

3.3 Configuration File

The **Config class** in the config.py module centralizes the configuration parameters used throughout the system. These parameters control key aspects of the algorithms, including thresholds, dimensions, and iteration counts.

Default Configuration Parameters

Below are some important default parameters defined in the Config class:

- **DIMENSION**: Embedding vector size (default: 512)
- **THRESHOLD 1**: Threshold for similarity matrix classification (default: 0.9)
- **THRESHOLD 2**: Threshold for graph refinement (default: 30%)
- **PERCENTAGE**: Percentage of edges to randomly add/remove (default: 30%)
- **K**: Number of iterations for the Research Plan (default: 30)
- **NODE2VEC_ITERATIONS**: Iterations for Node2Vec embedding (default: 3)
- **PYRAMID_SCALES**: Number of coarsening scales (default: 8)
- **ALPHA**: Scaling factor for ACO pheromones (default: 0.5)

4. **Codebase Structure**

This section provides an overview of the project's codebase, highlighting the responsibilities of each major file and module, and explaining how different components interact.

4.1 UI.py: Handles the User Interface and Interaction

UI.py serves as the entry point for the application and provides a user-friendly interface for interacting with the system.

Key Responsibilities:

- Displays input fields for selecting datasets, configuring hyperparameters, and starting the evaluation process.
- Creates a separate thread to execute the core logic in main_code.py, ensuring the UI remains responsive.
- Provides visual feedback during execution, such as progress updates and error messages.
- Includes **'?' tooltips** next to configurable options for quick guidance.

4.2 main_code.py: Executes Core Logic and Algorithms

main_code.py orchestrates the system's computational workflows. It is executed in a separate thread, initiated by UI.py, to handle resource-intensive tasks.

Key Responsibilities:

- Loads datasets (edge list and labels) from the specified paths.
- Initializes and invokes the **EvaluationPlan**, which executes the evaluation and research workflows.
- Calculates success rates and logs outputs (e.g., plots, histograms, and conclusions).

4.3 Key Supporting Modules

evaluation_plan.py:

The evaluation_plan.py module contains the logic for evaluating the system's performance.

EvaluationPlan Class:

- Purpose: Manages the overall evaluation process by simulating noisy data and assessing the system’s ability to refine it.
- Attributes:
    - g: Input graph (NetworkX Graph object).
    - d: Dimension of embedding vectors.
    - t1, t2: Threshold values for similarity classification and graph refinement.
    - p: Percentage of edges to add/remove.
    - K: Number of iterations for evaluation.
- Key Method:
    - EvaluationPlanAlg():
        - Simulates noisy data by adding random edges using the AddingEdges function.
        - Invokes the ResearchPlan to perform graph coarsening, embedding, and refinement.
        - Calculates the success rate by comparing refined graphs with the original.
        - Handles potential errors during visualization or output storage using exception handling.

research_plan.py:

The research_plan.py module implements the **Multi-Level Network Embedding (MLNE)** algorithm and handles graph refinement processes.

- **ResearchPlan Class**:
    - **Purpose**: Executes the MLNE process to generate embeddings and refine graphs.
    - **Attributes**: Similar to EvaluationPlan (e.g., g, d, t1, t2, p, K).
    - **Key Method**:
        - ResearchPlanAlg():
            - Iteratively applies Ant Colony Optimization (ACO) for graph coarsening.
            - Generates node embeddings using Node2Vec.
            - Computes similarity matrices based on cosine similarity.

    - Produces a refined graph by analyzing statistical matrices over multiple iterations.
  - Highlights:
    - Handles hierarchical graph coarsening (Algorithm 4).
    - Applies Ant Colony Walking (Algorithm 2) for clustering nodes.

config.py:

The config.py module centralizes configuration management, making it easier to customize parameters across the system.

- **Key Features**:
  - Stores all hyperparameters, such as DIMENSION, THRESHOLD 1, and PYRAMID_SCALES.
  - Provides methods to dynamically save (save_to_json) or load (load_from_json) configuration settings in JSON format.
- **Use Case**:
  - Allows for tuning parameters like embedding dimensions and thresholds, optimizing the system for different datasets.

4.4 How Threads Communicate

The system employs a **two-thread model** to ensure a responsive user experience while executing computationally heavy tasks.

Communication Flow:

1. **Thread 1 (UI)**:
   - Launched by UI.py and interacts with the user.
   - Collects inputs such as datasets and hyperparameters.
   - Starts a separate thread for executing the core logic.
2. **Thread 2 (Logic)**:
   - Initiated by the UI to run main_code.py.
   - Handles the Evaluation Plan and Research Plan, including MLNE execution.
   - Reports progress and outputs (e.g., logs and results) back to the UI.

Synchronization:

- The threads operate independently, but progress updates and error handling are passed back to the UI thread to keep the user informed.

5. **Core Algorithms**

This section provides an in-depth explanation of the key algorithms implemented in the system, including the Evaluation Plan, Research Plan, and embedding processes.

5.1 Evaluation Plan

The **Evaluation Plan** tests the effectiveness of the system by adding and removing edges in the graph, executing the Research Plan, and calculating success rates based on refined graphs.

Adding and Removing Edges

- **AddingEdges(g, p)**:
  - Randomly adds a percentage (p) of edges to the graph g.
  - Simulates the presence of noisy or "fake" edges to evaluate the system's ability to filter them out.
- **RemoveEdges(g, p)**:
  - Randomly removes a percentage (p) of edges from the graph.
  - Used to simulate graph degradation and test the robustness of the embedding and coarsening processes.

Similarity Calculation

- **CalculateCosineSimilarity(embedding_matrix, t1)**:
  - Computes a similarity matrix by calculating cosine similarity between all node embeddings.
  - Values above the threshold t1 are marked as "similar," while others are marked as "dissimilar."
  - Outputs a binary similarity matrix.

Statistical Matrix Creation and Graph Refinement

- **Statistical Matrix**:
  - Aggregates similarity matrices from multiple iterations of the Research Plan.
  - Calculates the percentage of iterations where nodes are classified as similar.
- **Graph Refinement**:
  - **RefineGraph(g, M_stat, t2)**:
    - Removes edges from the graph based on the statistical matrix.

        ▪ Edges with similarity percentages below the threshold t2 are considered weak and are removed.

5.2 Research Plan

The **Research Plan** implements the **Multi-Level Network Embedding (MLNE)** algorithm, which coarsens the graph into hierarchical layers, generates embeddings for each layer, and refines the graph.

Multi-Level Network Embedding (MLNE) Overview

The MLNE algorithm captures the global and local structures of a graph through hierarchical coarsening and multi-scale embedding:

1. **Graph Coarsening**: Simplifies the graph by merging nodes into clusters, reducing complexity while preserving structure.
2. **Embedding Generation**: Creates embeddings at each layer of the graph pyramid.
3. **Embedding Concatenation**: Combines embeddings from all layers to produce a comprehensive multi-level representation.

Graph Coarsening (Algorithm 4)

- Iteratively simplifies the graph by:
    1. Identifying strongly connected nodes through **Ant Colony Walking (Algorithm 2)**.
    2. Merging these nodes into clusters, forming coarser graphs at higher levels.
- Stops when the number of nodes or edges drops below a threshold.

Ant Colony Walking (Algorithm 2)

- Simulates the behavior of ants exploring paths to identify strong relationships in the graph.
- **Process**:
    - Ants perform random walks and scatter pheromones on loops they traverse.
    - Edges with higher pheromone levels represent stronger connections.
- Outputs a pheromone matrix that highlights the strength of relationships between nodes.

Adaptive Threshold Selection (Algorithm 3)

- Dynamically determines the threshold for classifying edges as "strong" or "weak."
- **Process**:
    1. Sorts edges by pheromone levels.

2. Identifies an "elbow point" in the sorted data, which represents the optimal threshold for separating strong and weak edges.

5.3 Embedding Process

The embedding process uses Node2Vec to generate feature vectors for nodes and combines these embeddings across multiple scales.

Node2Vec Integration

- Node2Vec is used to generate embeddings for each layer of the coarsened graph.
- Key parameters include:
  - **Walk Length**: Number of steps in a random walk.
  - **Number of Walks**: Number of random walks per node.
  - **Hyperparameters (p, q)**: Control the balance between exploring local and global graph structures.

Multi-Scale Embedding and Graph Pyramid

- Embeddings are generated for each layer of the graph pyramid.
- These embeddings are concatenated to form a unified feature vector for each node, capturing both local and global structures of the graph.