



ALMA MATER STUDIORUM UNIVERSITÀ DI BOLOGNA

Gene Ontology Analysis System Report

Team members

- Elnaz Nilooftary
- Luna López-Aparicio Rabasco
- Momchil Petkov
- Alessandra Zoli

Professor: Andrea Giovanni Nuzzolese

Date: February 2026

Abstract

This report details the design and implementation of a Gene Ontology (GO) analysis system. The software provides a web-based interface for researchers to upload standard GO data formats (OBO and GAF), perform statistical analysis, and calculate semantic similarity between genes and terms using multiple algorithms (Jaccard, Wu-Palmer, Resnik). The system is built using Python and Flask, featuring a flexible and modular design that allows researchers to easily switch between different analysis algorithms and expand the tool in the future. Key features include a recursive annotation checking, similarity score calculations, searching genes or terms, and pathfinding between ontology terms.

Introduction

Gene Ontology (GO) is a major bioinformatics initiative to unify the representation of gene and gene product attributes across all species. Analyzing this data requires robust tools capable of handling complex Directed Acyclic Graph (DAG) structures and large annotation datasets.

Problem Statement: Researchers need a way to easily explore the relationships between genes and functions, specifically finding common ancestors and checking how similar two genes are based on their function.

Proposed Solution: We developed a web application that parses standard OBO (structure) and GAF (annotation) files. The system allows users to search for genes, see term neighborhoods, and compute similarity scores. The backend uses appropriate data structures to handle calculations, while the frontend provides a user-friendly dashboard.

System Architecture

The software is organized into three main layers, allowing the system to handle data storage, analysis, and user interaction separately.

1. Data Handling Layer (The "Brain")

ontology.py: This component loads the Gene Ontology structure (the tree/graph of terms). It understands how terms are related (parents/children) and allows the system to find paths between biological terms, like molecular functions, biological processes and cellular components.

repository.py: This manages the gene annotation data. It acts like a digital library, allowing the system to quickly look up which genes belong to which terms (and vice versa) and handle the complex associations found in GAF files.

models.py: Defines the basic biological objects in the system: a *Gene* (with its symbol and annotations) and a *GOTerm* (with its ID, name, and definition).

2. Analysis Core (The "Calculator")

analysis.py: This contains the mathematical algorithms used to compare terms or genes. It can calculate semantic similarity scores (like Jaccard or Resnik) to determine how functionally similar two genes or terms are.

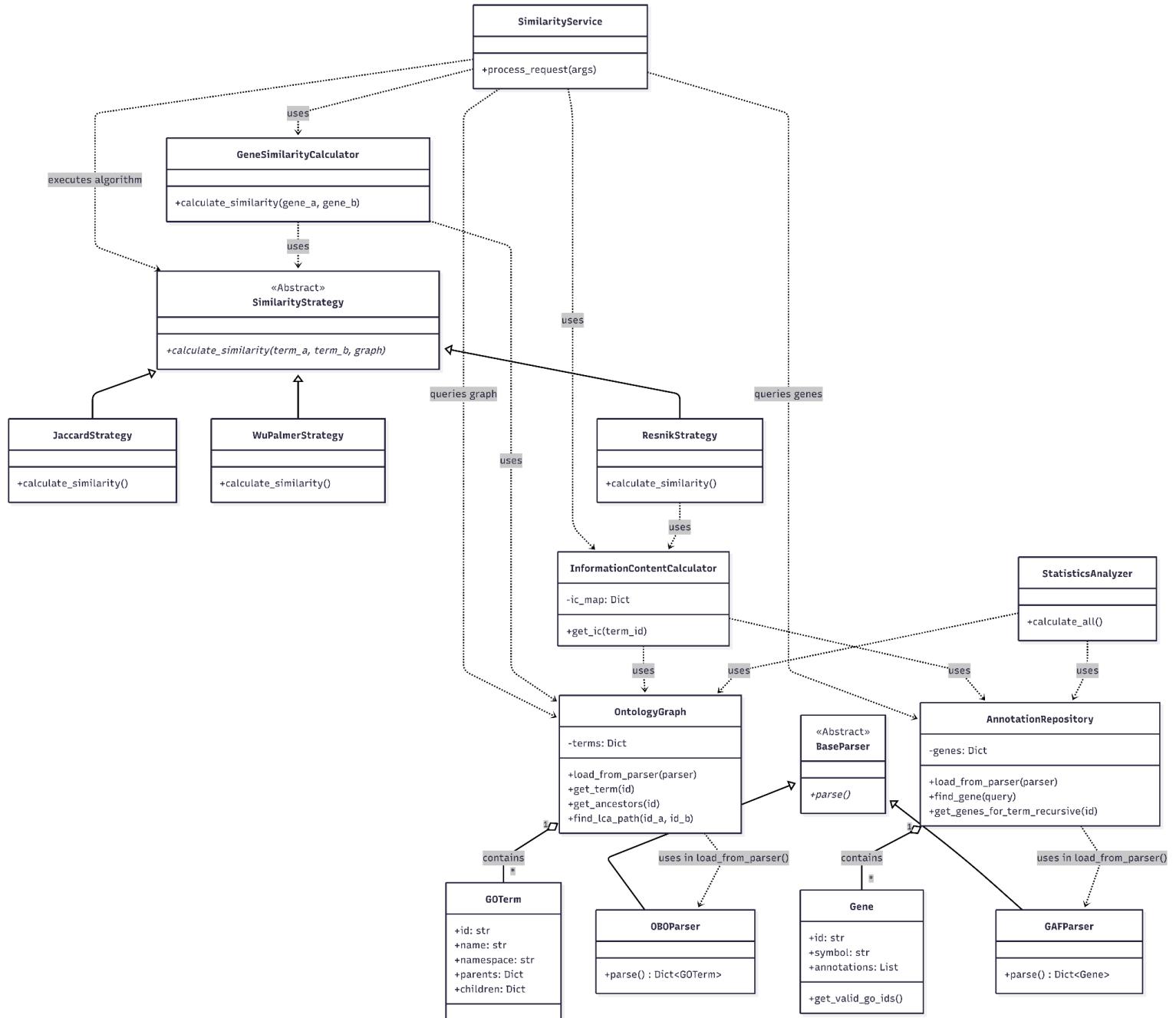
statistics.py: This module analyzes the loaded data to generate summary statistics, such as the distribution of annotations across the three GO aspects (Process, Function, Component), top annotated terms and many other metrics.

3. User Interface (The "Dashboard")

main.py: The central control script that runs the web server. It receives user inputs (like a search query or file upload) and asks the Data and Analysis layers for the results.

templates: These provide the visual interface as HTML files, displaying results as interactive tables, heatmaps, and path visualizations in the web browser.

UML class diagram



CRC Cards

Core Data Classes

Class: Gene	Superclass:	Subclasses:
Responsibilities	Collaborations	
Store biological data (ID, Symbol, Name)		
Store a list of GO annotations		
Filter annotations (e.g., exclude "NOT" relations)		
Count total annotations		

Class: GOTerm	Superclass:	Subclasses:
Responsibilities	Collaborations	
Store ontology term details (ID, Name, Namespace)		
Maintain a list of parent and child relationships.		
Store metadata like definitions and synonyms.		

Management & Logic Classes

Class: OntologyGraph	Superclass:	Subclasses:
Responsibilities	Collaborations	
Manage the entire graph of GOTerm objects	GOTerm	
Load data using a parser	OBOParser	
Calculate term depth and find ancestors or descendants		
Find paths between terms (LCA, Shortest Path)		

Class: AnnotationRepository	Superclass:	Subclasses:
Responsibilities	Collaborations	
Load data using a parser	Gene	
Manage the collection of Gene objects	GAFParser	
Perform gene searches by symbol or name.	OntologyGraph	
Build a reverse index (Term ID →Genes)		

Class: SimilarityService	Superclass:	Subclasses:
Responsibilities	Collaborations	
Facade for all similarity operations	OntologyGraph	
Process web requests and format results	AnnotationRepository	
Initialize and manage different SimilarityStrategy instances.	SimilarityStrategy	

Analysis & Strategy Classes

Class: SimilarityStrategy (Abstract)	Superclass:	Subclasses: JaccardStrategy WuPalmerStrategy ResnikStrategy
Responsibilities	Collaborations	
Define the interface for calculating similarity between two terms	OntologyGraph	
Enforce implementation of calculate_similarity		

Class: JaccardStrategy	Superclass: SimilarityStrategy	Subclasses:
Responsibilities	Collaborations	
Calculate similarity using the Jaccard score	OntologyGraph	
Identify intersection and union of ancestor sets		

Class: WuPalmerStrategy	Superclass: SimilarityStrategy	Subclasses:
Responsibilities	Collaborations	
Calculate similarity using the Wu-Palmer method.	OntologyGraph	
Find the Lowest Common Ancestor (LCA)		
Compute term depths relative to the root		

Class: ResnikStrategy	Superclass: SimilarityStrategy	Subclasses:
Responsibilities	Collaborations	
Calculate semantic similarity using Information Content (IC)	OntologyGraph	
Find the maximum IC among common ancestors	InformationContentCalculator	
Provide a more specific, probability-based measure		

Class: GeneSimilarityCalculator	Superclass:	Subclasses:
Responsibilities	Collaborations	
Calculate similarity between two Genes	Gene	
Compute the "Best Match Average" score	SimilarityStrategy	
Generate N×N similarity matrices for lists of genes		

Class: InformationContentCalculator	Superclass:	Subclasses:
Responsibilities		Collaborations
Calculate the probability of occurrence for every GO term		OntologyGraph
Compute Information Content (IC) values		AnnotationRepository
Provide IC lookup for the Resnik algorithm.		

Utility Classes

Class: BaseParser	Superclass:	Subclasses: OBOParser GAFParser
Responsibilities		Collaborations
Enforce a common interface (parse method) for all file readers		GOTerm
Handle basic file validation (check if file path exists)		
Act as a parent class for OBOParser and GAFParser		

Class: OBOParser	Superclass: BaseParser	Subclasses:
Responsibilities		Collaborations
Open and read raw text files line-by-line		GOTerm
Parse OBO syntax format files		
Create GOTerm objects from text data		

Class: GAFParser	Superclass: BaseParser	Subclasses:
Responsibilities		Collaborations
Open and read raw text files line-by-line		Gene
Parse GAF tabular format files		
Create Gene objects from text data		

Class: StatisticsAnalyzer	Superclass:	Subclasses:
Responsibilities		Collaborations
Aggregate data from the graph and repository		AnnotationRepository
Calculate global stats (e.g., Average Annotations per Gene)		OntologyGraph
Identify top-ranked genes and terms		

OOP Principles

- **Abstraction**

We used the `BaseParser` and `SimilarityStrategy` abstract base classes. This hides the complex implementation details of file reading and mathematical formulas from the main application flow. Also the `OntologyGraph` class encapsulates the complex graph traversal logic (BFS algorithm), term depth calculation and finding term ancestors and descendants. External classes simply call `get_ancestors()` without needing to know how the graph is traversed internally.

- **Encapsulation**

The `Gene` and `GOTerm` classes demonstrate encapsulation by hiding their internal data. Attributes like `_id`, `_name`, and `_annotations` are made private, preventing direct external modification. Access is controlled through public methods and properties. For instance, the `Gene` class forces all annotation additions through the `add_annotation()` method, which contains logic to handle duplicates and ensures data consistency.

- **Inheritance**

`JaccardStrategy` and `ResnikStrategy` inherit from the parent `SimilarityStrategy`. This ensures they both implement the required `calculate_similarity` method, enforcing a strict contract. The same goes for `OBOParser` and `GAFParser` classes, which are inherited from the abstract class `BaseParser`.

- **Polymorphism**

The `GeneSimilarityCalculator` can accept any strategy object. This is possible because concrete classes like `JaccardStrategy`, `WuPalmerStrategy`, and `ResnikStrategy` all inherit from the abstract `SimilarityStrategy` and provide their own implementation of the same method. It calls `.calculate_similarity()` on the object without knowing which specific algorithm is being used, allowing the algorithm to be swapped at runtime.

Design Decisions

1. Modular Similarity Algorithms

In bioinformatics, there is no single "best" way to measure functional similarity. Some analyses benefit from simple graph overlaps (Jaccard), while others require statistical weighting and checking how rare terms are (Resnik). To address this, we designed the similarity engine to be modular.

Instead of using a single formula, the system treats each algorithm as an interchangeable component or "strategy." This allows researchers to switch between Jaccard, Wu-Palmer, and Resnik methods instantly to compare results. Furthermore, this design future-proofs the software. If a new similarity metric is developed in the scientific community, it can be easily used as a new class without needing to rewrite the core application logic.

2. Integration of Pandas and NumPy

Pandas is used in `statistics.py` and `repository.py` to handle large tabular data from GAF files. It allows for vectorized operations (like `groupby` and `value_counts`), which are significantly faster than standard Python loops for generating reports and statistical summaries.

NumPy is used in `InformationContentCalculator` to perform element-wise logarithmic calculations on array data, which is essential for the Resnik similarity metric.

3. Caching & Optimization

To improve performance, the `OntologyGraph` implements a caching mechanism for term depth. Since calculating the depth of a node in a DAG is recursive and expensive, we store the result after the first calculation (`__depth_cache`), making subsequent lookups O(1).

Conclusion

This project successfully implements a comprehensive tool for Gene Ontology analysis. By strictly adhering to Object-Oriented principles like Encapsulation and Polymorphism, we created a system that is both robust and easy to extend. The modular architecture allows for the easy addition of new features, such as new file parsers or similarity metrics, without disrupting the core functionality. The integration of efficient data structures like Pandas DataFrames ensures the system can handle real-world biological datasets effectively.