POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Tesi di Laurea Magistrale in
Computer Science and Engineering
Ingegneria Informatica

BioGraph: A Network Approach to Multi-Omics Information Retrieval

Author:
Francesco Mario Inzerillo

Student ID:
10708997

Advisor:
Prof. Marco Domenico Santambrogio

Academic Year:
2025-26

Dedicated to my family, the Lord's greatest gift to my life.

Abstract

Biomedical research increasingly depends on heterogeneous genomic data sources—Ensembl, NCBI, UniProt, KEGG, Gene Ontology—that remain scattered across independently-curated databases with distinct identifier systems, formats, and update schedules. Recent breakthroughs such as Google's AlphaFold have expanded the volume of available protein structure data, further amplifying the integration challenge: researchers must manually orchestrate queries across multiple APIs and hardcode pipelines for every new combination of data sources, limiting both extensibility and ad-hoc exploration.

To address this, we developed BioGraph, a Python package that uses reflection and type introspection to automatically discover relationships between biological entities at runtime. BioGraph collects data from heterogeneous sources, normalises it into a unified object model, and returns results as a traversable knowledge graph—without requiring changes to framework code when new entity types or data sources are added. A universal traversal engine handles arbitrary multi-hop queries by inspecting entity class definitions rather than relying on hardcoded routing logic.

We validate BioGraph through three case studies. In targeted pathway discovery, BioGraph identified 28,161 gene–pathway pairs from 1,000 input genes using only 2 API calls. In gene-to-PPI traversal, batching reduced API calls by 67% compared to individual queries. In open-ended radial exploration of three cancer genes, BioGraph discovered 124,852 entity relationships across six relationship types at depth 3, with a 55% reduction in execution time when queries were combined. These results demonstrate that automatic relationship discovery via reflection enables flexible, extensible biological data integration without sacrificing query efficiency.

Keywords: bioinformatics, data integration, knowledge graphs, meta-programming, genomic data

Abstract in lingua italiana

La ricerca biomedica dipende in misura crescente da fonti di dati genomici eterogenee—Ensembl, NCBI, UniProt, KEGG, Gene Ontology—che rimangono distribuite tra banche dati indipendenti con sistemi di identificatori, formati e cicli di aggiornamento distinti. Recenti progressi come AlphaFold di Google hanno ulteriormente ampliato il volume di dati disponibili, accentuando la sfida dell'integrazione: i ricercatori devono orchestrare manualmente le interrogazioni su più API e codificare pipeline rigide per ogni nuova combinazione di sorgenti, limitando l'estensibilità e l'esplorazione ad hoc.

Per affrontare questo problema, abbiamo sviluppato BioGraph, un pacchetto Python che utilizza reflection e introspezione dei tipi per scoprire automaticamente le relazioni tra entità biologiche a runtime. BioGraph raccoglie dati da sorgenti eterogenee, li normalizza in un modello a oggetti unificato e restituisce i risultati come un grafo della conoscenza percorribile—senza richiedere modifiche al codice del framework quando vengono aggiunti nuovi tipi di entità o sorgenti dati. Un motore di traversal universale gestisce interrogazioni multi-hop arbitrarie ispezionando le definizioni delle classi entità anziché affidarsi a logiche di routing predefinite.

BioGraph è validato attraverso tre casi studio. Nel caso di pathway discovery mirata, BioGraph ha identificato 28.161 coppie gene–pathway da 1.000 geni in input con sole 2 chiamate API. Nella traversal gene-to-PPI, il batching ha ridotto le chiamate API del 67% rispetto alle interrogazioni individuali. Nell'esplorazione radiale aperta di tre geni oncologici, BioGraph ha scoperto 124.852 relazioni tra entità su sei tipi di relazione a profondità 3, con una riduzione del 55% del tempo di esecuzione combinando le interrogazioni. Questi risultati dimostrano che la scoperta automatica delle relazioni tramite reflection consente un'integrazione flessibile ed estensibile dei dati biologici senza sacrificare l'efficienza delle interrogazioni.

Parole chiave: bioinformatica, integrazione dei dati, grafi della conoscenza, meta-programmazione, dati genomici

Contents

List of Figures

List of Tables

1 | Introduction

Biomedical research has undergone a radical transformation in the past two decades. As sequencing technologies have become increasingly accessible and affordable, the explosion of genomic and proteomic data has revolutionized our capacity to understand disease mechanisms, identify therapeutic targets, and conduct precision medicine research. Yet this abundance of data brings a critical challenge: the same biological information is scattered across dozens of independently-curated databases, each using proprietary identifiers, formats, and update schedules. A researcher studying a single gene—say, BRCA1—must manually reconcile data from Ensembl, NCBI, UniProt, KEGG, and Gene Ontology, each providing partially overlapping yet complementary information. This fragmentation creates a significant barrier to discovery: researchers spend hours on data wrangling rather than biological insight.

1.1. Biological context and the data integration challenge

Modern biomedical research fundamentally depends on integrating information across three primary entity types: **genes**, their **protein products**, and **functional annotations**. Understanding any biological system requires connecting these entities through their relationships, yet this task remains surprisingly difficult despite decades of standardization efforts.

Genes serve as the primary units of the genome, encoding information for proteins and regulatory functions. Three major databases—Ensembl [10], NCBI Gene [14], and OMIM [8]—maintain comprehensive gene catalogs, each assigning distinct identifier systems (Ensembl ID, Entrez ID, and OMIM ID respectively) while providing partially overlapping genomic annotations. For example, the human BRCA1 gene carries at least five different identifiers: 672 (Entrez ID), ENSG00000012048 (Ensembl ID), 1101 (HGNC ID), NM_007294 (RefSeq ID), and P38398 (UniProt accession). These databases update on different schedules—Ensembl quarterly, NCBI daily—reflecting distinct curation philosophies and making it difficult to determine which represents the most authoritative information.

Proteins and pathways represent the functional consequences of genes. Proteins are the molecular machines encoded by genes, carrying out essential biological functions. Biological pathways describe how proteins and genes interact in coordinated processes, with KEGG [11] serving as the primary authoritative source for curated pathway information. A single protein may participate in multiple pathways, and a single pathway may involve dozens of proteins—relationships that are central to understanding both normal physiology and disease.

Gene Ontology (GO) [1, 5] provides standardized vocabulary for describing gene and protein function across three complementary dimensions: Biological Process (what biological goals genes accomplish), Molecular Function (the molecular activity of gene products), and Cellular Component (where in the cell the gene product acts). As the primary tool for functional annotation in bioinformatics, GO enables researchers to annotate genes consistently across all research domains. A critical challenge in modern bioinformatics is integrating these three entity types across heterogeneous sources [12], where each database operates independently and each provides different aspects of the complete biological picture.

1.2. Available biological databases and data sources

Genomic and proteomic research depends on a diverse ecosystem of specialized databases, each independently curated by research institutions worldwide. This section surveys the landscape of available resources, organizing them into three categories based on their primary focus.

1.2.1. Gene curation databases

Gene-focused databases form the foundation of genomic research. The three dominant resources—Ensembl, NCBI Gene, and OMIM—maintain comprehensive gene catalogs with distinct identifier systems while providing partially overlapping genomic annotations. While each database assigns proprietary identifiers to its data (for instance, Ensembl assigns each gene a unique Ensembl ID), they share these identifiers through cross-references, enabling identity mapping across systems.

These three databases reflect the distributed nature of biological research: as different labs worldwide developed expertise in genome curation, they built specialized databases reflecting their specific focus. Ensembl prioritizes well-characterized sequences with strict curation standards, updating quarterly. NCBI Gene emphasizes comprehensive coverage

with rapid daily updates, including less well-characterized sequences. OMIM focuses on disease-gene relationships. The result is that a single human gene often exists under multiple IDs across these systems.

As shown in Table 1.1, these databases differ significantly in scope, update frequency, and available identifiers. A researcher must therefore consult multiple sources to obtain complete information, manually reconciling differences and resolving inconsistencies. This fragmentation creates a critical integration challenge: no single database serves as a comprehensive "source of truth", requiring researchers to manually reconcile data across multiple sources.

Table 1.1: Gene databases: scope, update frequency, and identifier systems

Database	Coverage	Update Freq	Primary IDs	API	Notes
Ensembl	∼20k genes	Quarterly	Ensembl ID, symbol	REST	GRCh38 focus
NCBI Gene	∼43k genes	Daily	Entrez ID, symbol	REST	Cross-DB refs
OMIM	∼25k genes	Irregular	OMIM ID, symbol	Partial	Disease focus

1.2.2. Functional annotation databases

Beyond individual genes, researchers need to understand what genes *do* and how they interact. Other databases focus on functional categories, biological pathways, and regulatory networks. Unlike gene databases where multiple competing players exist, Gene Ontology and KEGG are essentially the primary, authoritative sources in their domains.

Gene Ontology provides a standardized vocabulary organized as a directed acyclic graph across three dimensions (Biological Process, Molecular Function, and Cellular Component) for systematically annotating gene function. KEGG curates canonical biological pathways showing gene interactions within cellular processes such as signal transduction, metabolic pathways, and disease mechanisms. These databases are generally more recent than gene databases and experience less fragmentation, though integrating them with gene-level data requires carefully bridging identifier systems. Reactome provides complementary pathway information with emphasis on detailed reaction mechanisms. Unlike gene databases which must handle millions of entities, these functional databases manage thousands to hundreds of thousands of well-characterized concepts, enabling more intensive curation. Table 1.2 summarizes the main functional annotation sources.

Table 1.2: Functional databases: scope and curation approach. *BP=Biological Process, MF=Molecular Function, CC=Cellular Component

Database	Primary Use	Update Freq	Structure	API
Gene Ontology	Functional vocab	Frequent	DAG	REST (AmiGO2)
KEGG	Pathway curation	Quarterly	Pathway maps	REST
Reactome	Signal pathways	Monthly	Reaction detail	REST

1.2.3. Protein, chemical, and interaction databases

Protein sequences, structures, and chemical interactions bridge molecular biology and drug discovery, forming a critical link between basic research and therapeutic development. Unlike gene and functional databases, which remain primarily academia-focused, chemical databases serve both research and pharmaceutical development. This dual purpose drives standardization efforts in identifier systems (SMILES strings, InChIKey, ChemBL ID) and data quantity.

UniProt aggregates protein sequences, structural data, and functional annotations from multiple sources, providing the most comprehensive protein resource available. STRING focuses on protein-protein interactions with confidence scores based on multiple evidence types. ChemBL and PubChem catalog chemical compounds linked to biological targets, enabling drug discovery queries and toxicology predictions. These databases feature more extensive data coverage and faster update cycles than gene databases (some updated continuously), reflecting the rapid pace of chemical and pharmaceutical research. While our research does not cover all available sources, Table 1.3 highlights the most prominent databases we integrated. Together, these resources provide complementary views of the proteome, the interactome, and the chemical universe of bioactive molecules.

Table 1.3: Chemical and protein databases: data types and pharmacological context

Database	Primary Use	Coverage	Update Freq	API
UniProt	Protein annotations	∼570k reviewed (Swiss-Prot)	Weekly	REST
STRING	Protein interactions	∼19M interactions	Every 2–3 years	REST
ChemBL	Drug targets	∼2.8M compounds	Quarterly	REST
PubChem	Chemical compounds	∼113M substances	Real-time	REST

1.3. Integration barriers and the limitations of current approaches

While database fragmentation might superficially appear problematic—indeed, when databases disagree on certain information, consensus across sources can indicate the most

reliable data—the actual challenge lies in the integration barriers that prevent seamless reconciliation across these heterogeneous sources. We identify three critical challenges that motivate our work.

Identifier heterogeneity. Different databases assign distinct identifier systems to the same biological entity. The same entity is therefore recognized and named differently across systems. The gene *BRCA1* serves as a concrete example: it possesses multiple identifiers—672 (Entrez ID), ENSG00000012048 (Ensembl ID), 1101 (HGNC ID), NM_007294 (RefSeq ID), and P38398 (UniProt accession). Moreover, while most IDs remain stable, others undergo periodic updates or vary by organism, further complicating automated reconciliation. Converting between identifier systems requires knowing which databases share links, maintaining those links as databases update, and handling edge cases where links are incomplete or ambiguous.

Versioning and coordinate shifts. Reference genome builds change over time as our understanding of the genome improves. The GRCh37 to GRCh38 transition, for instance, shifted chromosomal coordinates, invalidating analyses built on earlier versions. BRCA1 maps to chromosome 17, positions 43,044,295–43,125,364 in GRCh38, but occupied positions 41,196,312–41,277,500 in GRCh37. Tools developed for one build produce incompatible results when applied to data using a different build, creating reproducibility and interoperability problems. A comprehensive framework must not only integrate across databases but also handle coordinate conversions across genome builds.

Data completeness gaps. Annotation coverage varies significantly across databases. UniProt has sequences for SwissProt but not TrEMBL. KEGG focuses only on well-studied pathways, missing less-characterized biological processes. Gene Ontology annotates some genes extensively while others have minimal annotations. These gaps make seamless data consolidation impossible without explicit reconciliation logic. Researchers must understand which databases are authoritative for which types of information, and must make informed decisions about which data to trust when conflicts arise.

These three integration barriers establish why manual approaches require substantial domain expertise and remain time-consuming. A researcher wishing to find all proteins associated with a disease gene, and then identify all pathways containing those proteins, must make dozens of manual decisions about data quality, identifier conversions, and conflicting information across sources. Contemporary bioinformatics relies on automated approaches to bridge these gaps, yet—as we will explore in the following section—existing solutions address only specific aspects of integration, leaving the broader problem of arbitrary, dynamic queries insufficiently solved.

1.4. A knowledge graph approach to biological data integration

The fundamental challenge we address is this: researchers need to explore arbitrary relationships across biological databases without knowing in advance which queries are possible or what data exists. Current bioinformatics workflows are rigid. They follow predefined pipelines where each transformation step is explicitly programmed—adding a new entity type or data source requires modifying core application code. Moreover, queries follow fixed paths; a researcher cannot easily explore alternative relationships.

Our goal is to create a meta-framework that discovers relationships automatically, requiring researchers only to specify which entities interest them. The framework should support any-to-any entity queries without hardcoding routes or predefined query paths. New entity types and relationships should be discoverable at runtime with zero framework code modifications. To achieve this, we ground our design in the mathematical concept of a **graph**.

Why graphs for biological data?

A key insight motivating our approach is that individual biological data points carry little meaning in isolation. Knowing that BRCA1 is located on chromosome 17 is not particularly informative on its own; what matters is how BRCA1 interacts with the proteins it encodes, which pathways those proteins participate in, and which other genes share those pathways. Biological knowledge is fundamentally relational: it emerges from the connections between entities, not from the entities themselves.

This makes graphs a natural representation. A graph $G = (V, E)$ consists of vertices V and edges E connecting them. In a directed graph, edges $e_{i,j}$ are ordered pairs such that $e_{i,j} \neq e_{j,i}$, allowing directional traversal. In a biological context, this means we can model a gene coding for a protein, a protein participating in a pathway, and a pathway regulating another gene—all within the same structure. Two properties make graphs especially powerful here:

- *Multi-centrality*: any entity can serve as the starting point for a query. There is no fixed "centre" of the data—a researcher can begin from a gene, a pathway, or a protein with equal ease.
- *Relationship focus*: graph algorithms operate on connections, not just attributes. This allows modelling biological concepts such as centrality (how many pathways in-

volve a given protein), neighbourhood (which genes co-occur in the same processes), and reachability (can we reach this GO term from that gene within two hops?).

A **knowledge graph** is a specialised graph structure for representing heterogeneous data. The vertices V partition into multiple types: $V = V_1 \cup V_2 \cup \ldots \cup V_n$, where each subset represents a different entity type (genes, proteins, pathways, etc.). The edges follow a predefined **schema** R that specifies valid relationship types—for instance, $Gene \rightarrow CODES\_FOR \rightarrow Protein$. Relationships can encode additional properties such as confidence scores, evidence types, or cardinalities (one-to-one, one-to-many, many-to-many).

Reflection and type introspection as the enabling mechanism

This knowledge graph vision creates a design challenge: how do we build the graph without hardcoding every possible relationship between entity types? Our answer is **reflection** and **type introspection**—two related programming techniques that allow a program to examine and reason about its own structure at runtime.

In Python, every class carries metadata about its attributes and their declared types. *Reflection* refers to the ability to inspect this metadata programmatically—reading class definitions, listing attributes, and querying type annotations without executing the methods themselves. *Type introspection* goes one step further: by examining the type annotations (e.g., `pathways: List[Pathway]` in a `Gene` class), we can infer that a `Gene` has a relationship to `Pathway` objects, without this relationship ever being explicitly declared in a configuration file.

This is what makes BioGraph extensible by construction. When a developer adds a new entity class—say, `Drug`—with an attribute `targets: List[Protein]`, BioGraph automatically discovers the `Drug` → `TARGETS` → `Protein` relationship at startup through type introspection, and immediately enables all queries involving `Drug` entities. No routing tables, no configuration files, no changes to the traversal engine. The framework's knowledge of what queries are possible grows automatically as the data model grows.

We build an adjacency matrix of entities and edge types directly from the code structure, then apply standard graph algorithms (breadth-first search) to find paths from source to target entities. A universal traversal engine then executes any discovered path regardless of entity types involved. This design aligns with established software engineering principles including the Plugin Architecture and the Open-Closed Principle [13]: the framework is open for extension (add new entities) but closed for modification (no core code changes needed).

1.5. Related work

There are existing tools and frameworks which bridge some of the aforementioned gaps in the data. Below we survey the landscape of biological data integration solutions.

1.5.1. Workflow orchestration platforms

Two major platforms have shaped the landscape of reproducible bioinformatics:

Galaxy [7] provides a web-based interface for constructing computational workflows. It excels at data provenance tracking and reproducibility by maintaining detailed histories of all analysis steps. However, Galaxy workflows are rigid: each analysis requires manual composition, and adding new data sources means creating new workflow components.

Taverna [15] (now Apache Taverna) offers a more sophisticated workflow engine with support for service discovery and composition. The Taverna repository [23] enables sharing and discovery of pre-built workflows. Despite these advantages, Taverna still requires explicit workflow definition; exploring alternative paths through the data requires either creating new workflows or modifying existing ones.

Both platforms prioritize workflow design over data exploration. BioGraph's meta-framework inverts this: we prioritize automatic relationship discovery, enabling ad-hoc exploration patterns without explicit workflow authoring.

1.5.2. Unified biological data APIs

MyGene and BioThings [24] represent a significant advancement in API standardization. MyGene.info is an API service developed by the Department of Integrative, Structural and Computational Biology at the Scripps Research Institute. It was first made available in 2013 and has become a de facto standard for gene annotation, used by major institutions including Wikipedia's Gene Data, ClinGen, and Omni. The broader BioThings ecosystem includes MyVariant (gene variants), MyDisease (pathological data), and MyChem (chemical annotation). These tools solve the "scattered data" problem by providing unified REST interfaces to multiple underlying databases [22].

However, MyGene and similar API-aggregation tools address integration at the data level but not at the query pattern level. Discovering relationships between different entity types (e.g., genes to pathways to GO terms) still requires manual API orchestration. BioGraph automates this orchestration by discovering paths and executing them generically.

1.5.3. Knowledge graph approaches: PrimeKG and semantic web

The broader context of our work sits within the knowledge graph and linked data communities [3, 9, 20]. Knowledge graphs represent entities and relationships explicitly, enabling complex queries without modifying the core system. Resources like Reactome [6] and the GO annotation systems [5] implicitly follow knowledge graph principles.

A particularly relevant comparison is **PrimeKG** [2], a large-scale biomedical knowledge graph that integrates 20 high-quality biomedical resources into a unified graph covering diseases, drugs, genes, proteins, and biological pathways across more than 30 relationship types. PrimeKG is pre-computed and distributed as a static dataset, making it fast to query locally. However, its static nature is also its primary limitation: the graph is fixed at the time of construction, meaning it cannot incorporate new data sources or entity types without rebuilding the entire dataset from scratch. Furthermore, because PrimeKG stores relationships as flat edge records keyed by Entrez gene IDs, multi-hop traversals (e.g., gene $\rightarrow$ pathway $\rightarrow$ gene) require manual query composition rather than automatic path discovery. We use PrimeKG as a benchmark in our experimental evaluation to position BioGraph's dynamic, runtime-discovery approach against a state-of-the-art static knowledge graph.

Our contribution is to operationalize knowledge graph principles within a Python-based framework using reflection and runtime introspection, making them accessible to bioinformaticians without requiring graph database expertise or pre-built static datasets.

1.5.4. Data integration frameworks

The problem of heterogeneous data integration is well-studied in computer science [12]. The concept of federated databases and schema integration has been explored for decades. More recently, the FAIR Principles [22] have emerged as a standard for scientific data management. BioGraph implicitly enforces FAIR principles through structured entity models and explicit relationship definitions.

Related initiatives highlight the importance of interoperability standards in bioinformatics [19]. BioGraph complements these efforts by providing a practical implementation path for integrating heterogeneous biological data sources.

1.5.5. BioGraph's approach

The three properties that distinguish BioGraph from the tools surveyed above are:

1. **Zero-configuration extensibility**: Adding a new entity type (e.g., `Drug`, `Variant`) requires only defining its class with typed attributes and registering its API client. BioGraph's relationship registry automatically discovers all valid queries involving the new type at startup—no workflow authoring, no routing tables, no changes to any existing code. This is in contrast to Galaxy and Taverna, which require new workflow components, and PrimeKG, which requires a full dataset rebuild.

2. **Reflection-based universality**: A single traversal engine handles all entity combinations by inspecting class definitions at runtime. Rather than implementing separate traversal logic for each pair of entity types, BioGraph uses Python type annotations to build the relationship graph dynamically. Adding a new entity type does not require touching the traversal engine—the new relationships are discovered and executed through the same generic code path.

3. **Automatic path discovery**: Instead of requiring users to know valid query paths in advance, BioGraph discovers them from the entity structure using breadth-first search over the relationship graph. Users specify only the source and target entity types; the system selects the optimal path and executes it.

While Galaxy and Taverna excel at workflow provenance, MyGene provides excellent data aggregation, and PrimeKG offers a rich static resource for PPI-centred queries, none provide automatic relationship discovery with zero-configuration extensibility over dynamic, live data sources.

2 | Methods

BioGraph is a Python library for multi-omics data integration. Before diving into its internal architecture, it is useful to understand what it offers a researcher and how it is actually used.

Three primary use cases. BioGraph exposes three main entry points that cover the most common biological data retrieval patterns:

- `enrich()` — batch enrichment of a set of known entity IDs. A researcher starts with a list of gene symbols and wants complete annotations (genomic coordinates, GO terms, protein cross-references) without exploring further.
- `execute()` — directed traversal from one entity type to another. A researcher wants to find all pathways associated with a set of cancer genes, or all protein-protein interactions reachable from a list of proteins.
- `explore()` — open-ended radial discovery. A researcher starts from a few genes and wants to discover every connected entity (pathways, GO terms, proteins, interactions) within a configurable number of hops, without specifying target types in advance.

These three modes form a hierarchy of increasing scope and complexity, and the entire system is designed around making them as flexible and extensible as possible. The graph-based architecture is the enabling mechanism: because biological data is fundamentally relational, organizing it as a traversable knowledge graph allows the same framework to serve all three use cases without separate pipelines.

The approach taken can be subdivided in four steps, and was devised to offer maximum flexibility to the package. One of a knowledge graph's strongest points is that it imposes no fixed traversal direction: any node can serve as the starting point of a query. We wanted to exploit this aspect to the full, by organizing the collected data into a constellation, traversable from any end to any other end without limits on the path taken.

2.1. Overview of BioGraph

2.1.1. Problem definition and core design principles

As can be recounted from the previous chapter, the problem can be summarized in a few key issues: heterogeneous biological data sources are scattered across multiple databases; manual routing of queries through different APIs is inflexible; adding new entity types or data sources requires hardcoding; and the existing approach involves rigid, tightly-coupled pipelines.

Design vision. Our scope is to build a system that discovers relationships automatically, simply by adding or removing the omic entities of interest. In true graph-like mentality, this vision should support any-to-any entity queries without having to hard-code routes or pre-define where to look for additional data. This library is also thought of as simply a baseline, as it can, and must, be easily extended with new entities or sources with zero framework code changes. BioGraph should aim to accomplish this while also optimizing for efficiency. This can be done, however, not to the fullest extent, as will be discussed later.

Three core principles. BioGraph is built on three values: **self-introspection** (the system understands its own data model through reflection), **automatic discovery** (paths exist and are found via graph algorithms, not configuration), and **universal execution** (one engine handles all entity combinations via reflection).

2.2. Building blocks: the data model

To set BioGraph up, we defined the base classes to encase the data handled in BioGraph.

2.2.1. Omic entities

This part deals with how omic data is managed, saved and dealt with inside BioGraph. The class diagram in figure 2.1 illustrates how it was structured.

OmicEntity Base Class

`OmicEntity` is the basic abstract class, which works as the Greatest Common Factor between all types of omic data. Another notable feature of this system is the attribute types contained in other types, such as `Gene.go_terms` being a `list<GO>`. This is what allows us to "transit" between entities in the graph the moment the Pipeline gets built.

Given the nature and inner workings of the pipeline, we paid special attention to the identifiers of the omic entities. In fact, for BioGraph to function well and tolerate faults, it is essential that IDs of any of the used entities are well-managed, as complete as possible, and able to pass through the entire data flow without hindrances. This is why a separate `OmicEntityID` class was created. It is an abstraction of the types of IDs the entities possess, and all entities that inherit `OmicEntity` must also possess an ID attribute that inherits `OmicEntityID`.

Another notable argument in favor of this choice is that by their very nature, IDs have to pass through *conversions* and *filters* (seen in section 2.2.2) more rarely. The only real exception is syntactic and orthographic corrections, e.g., formatting Pathway IDs into a standardized format. But this is a very rigorous transformation and is much cheaper in terms of computational costs as compared to other ones.

Therefore, by "packing them up" into a single and primary attribute for any entity, we can ensure that these fields can go through that crucial transformation step untouched and available.

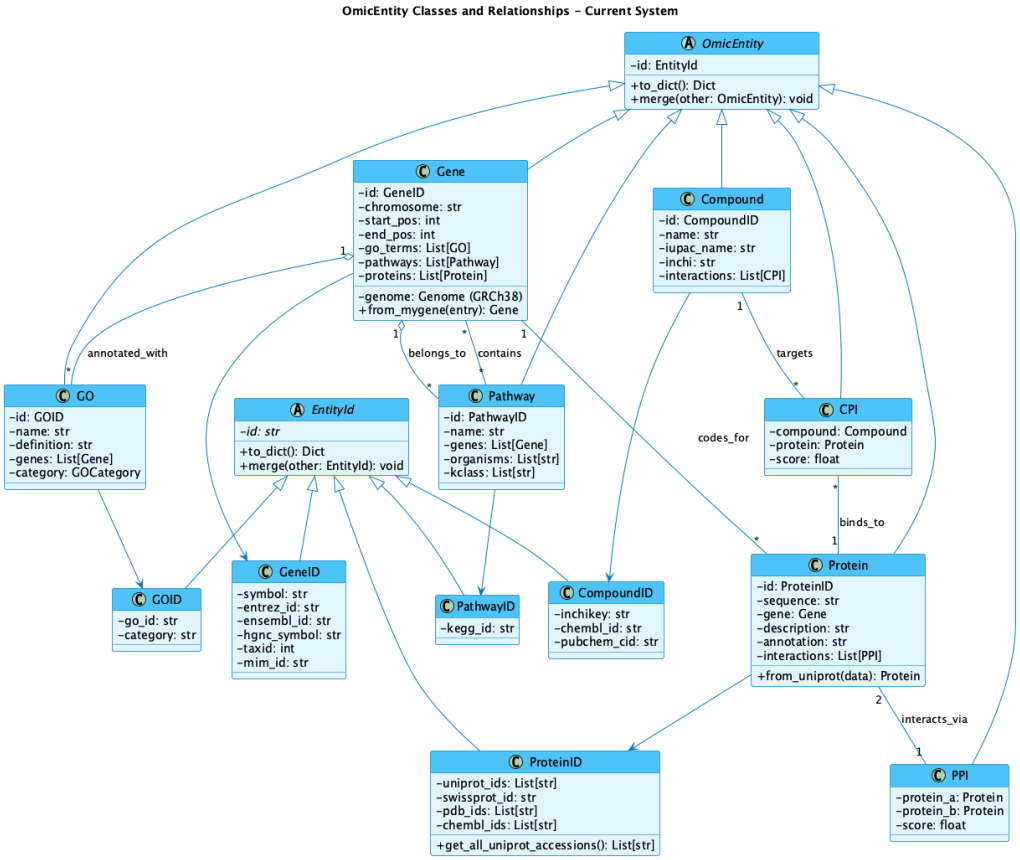


Figure 2.1: UML Diagram for the classes which model omic entities.

Gene, Pathway, and GO entities

BioGraph models three core entity types representing major biological concepts:

Gene: Represents a single gene with genomic data (chromosome, position, reference genome GRCh38 by default), ID types (Ensembl, NCBI, Entrez, TaxID), and relationships to other entities. The Gene entity is central in BioGraph, with many-to-many relationships to Pathways and GO terms, and one-to-many to Proteins. This architecture enables flexible gene-centric queries across multiple biological contexts.

Pathway: A biological pathway (cf section 1.1) containing genes and their functional relationships. BioGraph supports multiple pathway databases (KEGG, Reactome, WikiPathways) with unified representation, though the current implementation focuses on KEGG as the primary source.

Gene Ontology (GO): Represents GO terms organized into three categories (Biological Process, Molecular Function, Cellular Component), linked to genes via evidence-based annotations. GO terms provide complementary functional classification to pathway-based relationships.

Protein entity and UniProt enrichment

The Protein entity represents gene products with enhanced data enrichment from UniProt, including sequence data, functional annotations, structural information, and post-translational modifications. Cross-references enable linking proteins to their coding genes and associated pathways.

EntityID hierarchy and identifier resolution

As it could be seen from the entities so far, they all have an `id` property that contains unique identifying information about each of them. These classes all extend an abstract `EntityID` class. This is useful in a number of ways. For instance, IDs are different properties from others since they should not be subject to conversions and to very few filters. By separating them into a unique object, we can more easily carry this info down the pipeline. Furthermore, each entity's `id` property resolves to their own particular extension of the class, allowing for meta-entity property access.

2.2.2. Transformations

Transformation framework design

The operations that can be performed on the omic data are numerous. We decided to split them into two general categories, *conversions* and *filters*.

1. **Conversions** involve modifying quantitatively or qualitatively the data carried by the omic entities. An example can be the change in genome from GRCh37 to GRCh38 and the operations on the coordinates which are necessary to perform it.

2. **Filters** instead are a limit posed on the data, either for volume reduction reasons or for quality assurance reasons. They effectively act as a sieve in the data flow. For example, as gene information is retrieved, the IDs of its coded proteins might refer to different indexes (e.g., pCHEMBL, UniProt, etc.). The filter here is to restrict the allowed IDs only to a certain type, and to leave out the other IDs.

As can be seen in their class structure in figure 2.2, both types of operations are allotted a place in the respective registries. These registries will be analyzed in more detail in Section 2.2.3.

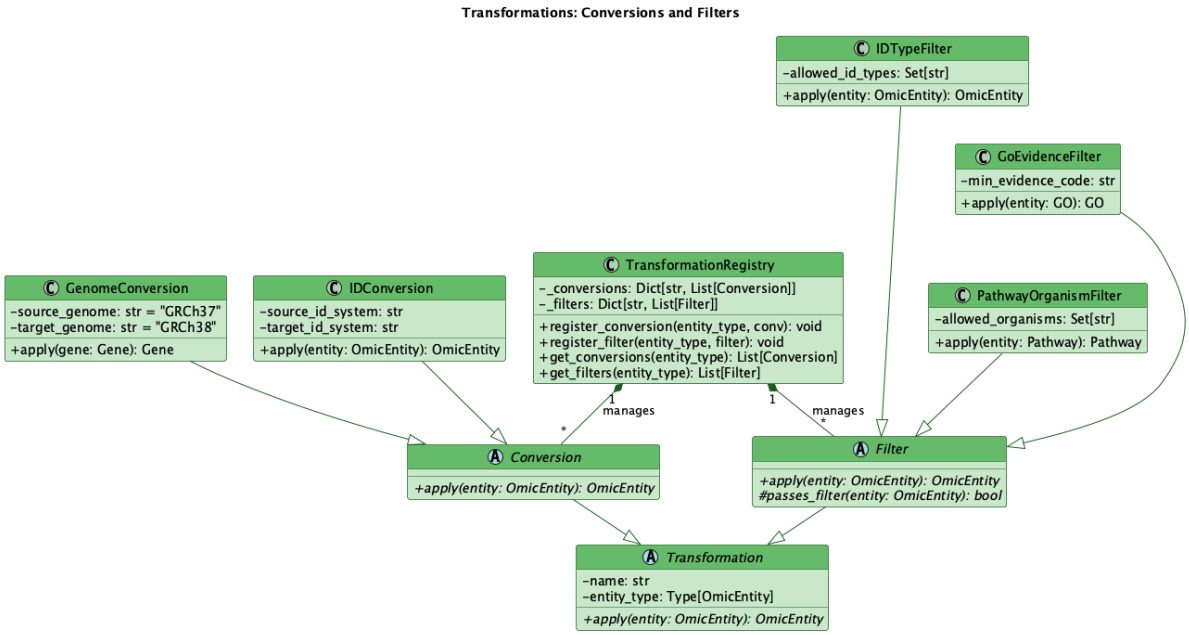


Figure 2.2: UML Diagram for the Transformations, which include Filters and Conversions.

Extensibility and custom transformations

The OOP principle adopted in the project allows for any kind of operation, provided it falls within the category of *transformation* or *filter*, to be added to the process, by simply extending the `Transformation` or `Filter` abstract class. This extensibility approach follows established design patterns [4] and is essential for maintaining BioGraph's zero-configuration extensibility principle.

2.2.3. Registries and their roles

Filters and converters do not need to be manually registered or imported. BioGraph uses a decorator-based registry pattern so that every component self-registers at import time, making it automatically available to the pipeline without any configuration. This is one of the core architectural decisions behind BioGraph's zero-configuration extensibility claim. Three registries operate in parallel:

1. **Clients Registry**. Registers API client classes (MyGene, KEGG, UniProt, GO), maps client names to their classes, and maps entity types to their data sources.
2. **Filters Registry**. Registers data filtering classes via the `@register_filter` decorator; auto-discovers all filters in the package using `pkgutil.iter_modules`.
3. **Conversions Registry**. Registers data conversion classes via the `@register_converter` decorator; auto-discovers converters the same way.

This is a decorator-based registry pattern, which follows Python community best practices for plugin architecture [16]. Each component self-registers using Python decorators, and BioGraph automatically discovers available plugins using the standard library's `pkgutil.iter_modules()` function [17]. This approach has several advantages (illustrated here using the DB sources registry):

1. **Decoupling & Extensibility**. New plugins can be added without modifying core code; classes register themselves via decorators and BioGraph discovers them automatically.
2. **Centralized Discovery**. A single point to query all available implementations (e.g., `list_sources()`) and dynamic lookup (e.g., `get_db_sources(name)`).
3. **Declarative Registration**. The decorator makes the plugin's purpose explicit and removes boilerplate compared to manual registration.
4. **Auto-Discovery via `pkgutil`**. Automatically imports all modules in a package and triggers decorator execution without explicit imports [17].

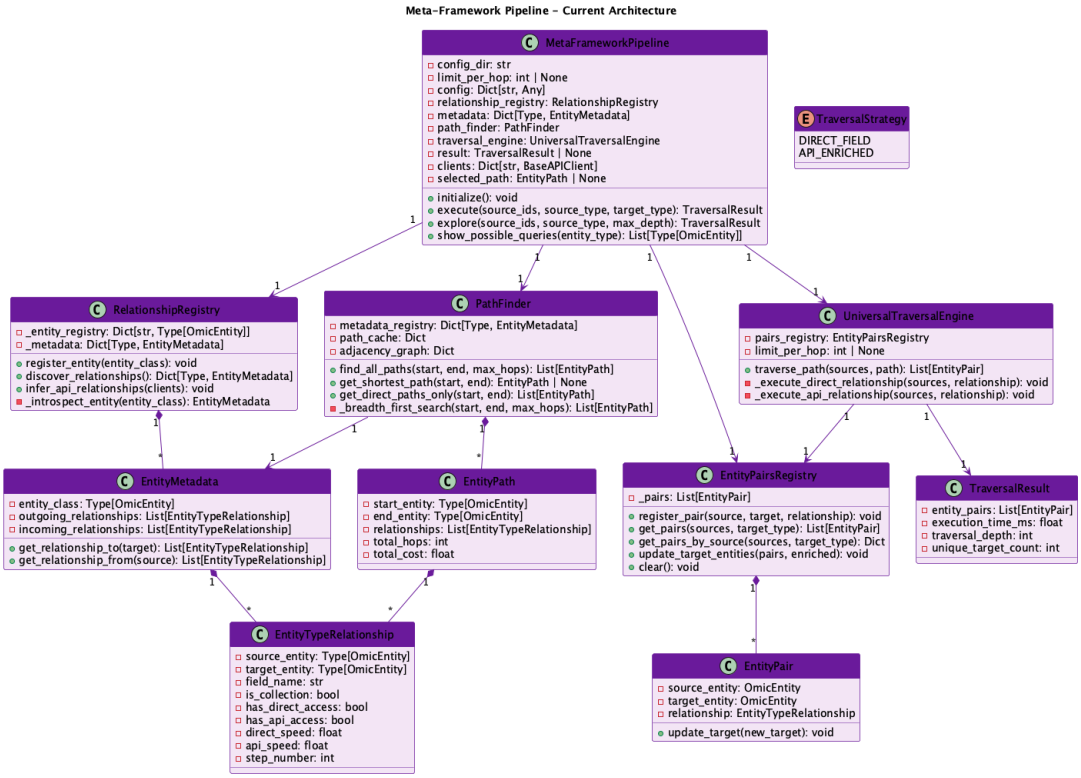


Figure 2.3: UML Diagram for BioGraph.

The main trade-offs are import-time side effects (registration happens during module import, which can complicate test isolation) and global state (module-level dictionaries are shared across the application). Despite these, the pattern offers a strongly positive trade-off for a configuration-driven pipeline and is well-established in the Python community [16].

2.3. Pipeline architecture

Having defined the basic components of BioGraph, we now illustrate how the pipeline itself, named `MetaFrameworkPipeline`, runs in its lifecycle. Use the UML diagram in figure 2.3 for reference.

Core BioGraph Components

The essential parts that constitute BioGraph are:

- A *property discovery / path-finding* part. This operates on a "strategy planning" level, and has the goal of discovering entities, finding paths between them, pruning bad strategies, and registering the information collected.

- A *traversal* part. Once having received the information discovered by the previous part, it operates on the "strategy execution" level, going down the planned path and gathering the requested information as it proceeds.

The lifecycle can be broadly categorized into 6 steps: initialization, relationship discovery, source entity retrieval, path discovery, traversal execution, and entity deduplication.

2.3.1. Initialization Sequence

The first step in the process is to institute the properties of the pipeline. An `EntityPairsRegistry` (see section 2.2.3 for more) is assigned to it with an empty state.

Configuration loading. Custom configuration is loaded onto the pipeline using a `ConfigLoader` object. It opens all setting files, defined in YAML, within a customly defined configuration directory. This can include custom transformations or filters, as well as options for the API clients, or the preferred type of ID to use for certain entities.

Client and entity registration. Afterwards, all API client instances are initialized, keeping the instance alive for the duration of the pipeline run. A `RelationshipRegistry` object (see section 2.2.3) is created with an empty state and tasked with registering each of the entity classes; during this step, the registry simply stores into memory the classes and their characteristics.

2.3.2. Automatic relationship discovery

The `RelationshipRegistry` then executes the `discover_relationships()` method.

Type introspection via annotations

The method goes on iteratively for each entity, looking for type annotations in the entity's attributes to discover what are called `DIRECT_FIELD` relationships. Python's type hint system [18] provides the foundation for this introspection mechanism. Type annotations are captured at module import time and analyzed to build the relationship graph. For example, the `Gene` class has been defined to host a `pathways` property. This stores a collection of pathway information. The `RelationshipRegistry`, upon introspecting `Gene`, will encounter the line:

```
pathways: List[Pathway]
```

Introspecting this, the method registers an `EntityTypeRelationship` object. It contains the source and target entities (`Gene` and `Pathway`), the local field (`pathways`), whether this is a collection or not (in our case, it is), and that it is a direct field relationship.

A more generalized instance is shown as pseudo-code in algorithm 2.1.

Algorithm 2.1 discover_relationships()

```
for each entity_class in entity registry do
    metadata ← _introspect_entity(entity_class)
    Store metadata for entity_class
end for
return metadata
```

Algorithm 2.2 _introspect_entity(entity_class)

```
Create empty metadata for entity_class
Get source code of __init__ method
for each attribute assignment with type comment in source do
    Resolve type (handle ForwardRef if needed)
    relationship ← _analyze_field_type(entity_class, attr_name, resolved_type)
    if relationship exists then
        Add relationship to metadata.outgoing_relationships
    end if
end for
return metadata
```

Forward reference handling

Python type annotations may reference classes not yet defined at import time. BioGraph handles this via `typing.get_type_hints()` with `localns` populated from the entity registry, resolving forward references after all entity classes have been loaded.

API relationship introspection

After defining these direct-field relationships, the registry is called again with the `infer_api_relationships()` method. This enables API access from each source entity to each target entity.

2.3.3. Source entity retrieval

Before the traversal engine can execute a query, the initial source entities must be retrieved from the configured data sources. This step is critical: it determines the starting point for all subsequent path traversals and ensures complete entity enrichment from the beginning.

Given a user query specifying `source_ids` (e.g., `['BRCA1', 'TP53']`) and a `source_type` (e.g., `Gene`), the pipeline:

1. Selects the appropriate client(s) for the source entity type (e.g., MyGene for Gene entities)
2. Performs a batch query to retrieve complete entity objects with all available attributes populated
3. Applies any configured conversions and filters to the retrieved data
4. Registers the retrieved entities in the `EntityPairsRegistry`

This initial retrieval is essential because source entities often contain embedded related data. For example, a `Gene` object retrieved from MyGene already includes `pathways` and `go_terms` attributes, which are leveraged by the `DIRECT_FIELD` traversal strategy in later steps. Without complete initial enrichment, the direct field relationships would be incomplete or unavailable.

The source entity retrieval respects the same batch-query optimization discussed in section 2.3.6: entities are queried in groups rather than individually to minimize API calls and reduce latency.

2.3.4. Clients registry and data sources

Client architecture and interface

The first step has been to "encase" the clients for the Database connections into an abstract class, called BaseAPIClient, which encompasses all database-specific operations that should be performed before aggregation to the rest of the data. As one can see in figure 2.4, this structure allows for versatility in the choice of sources. This design follows the Adapter Pattern [4] and enables support for diverse biological data sources including MyGene [24], KEGG [11], UniProt [21], and complementary services such as PubChem, ChEMBL, and STRING for protein interaction data.

More precisely:

1. The `conversions` and `filters` attributes can allow for custom data conversions (using the previously described `Converter`) and filtering (with the previously described `Filter`) operations on the data.
2. Most methods are private operations that take care of the different steps in the data extraction, from `_fetch_data()` for the raw data retrieval, to `normalize_data()`

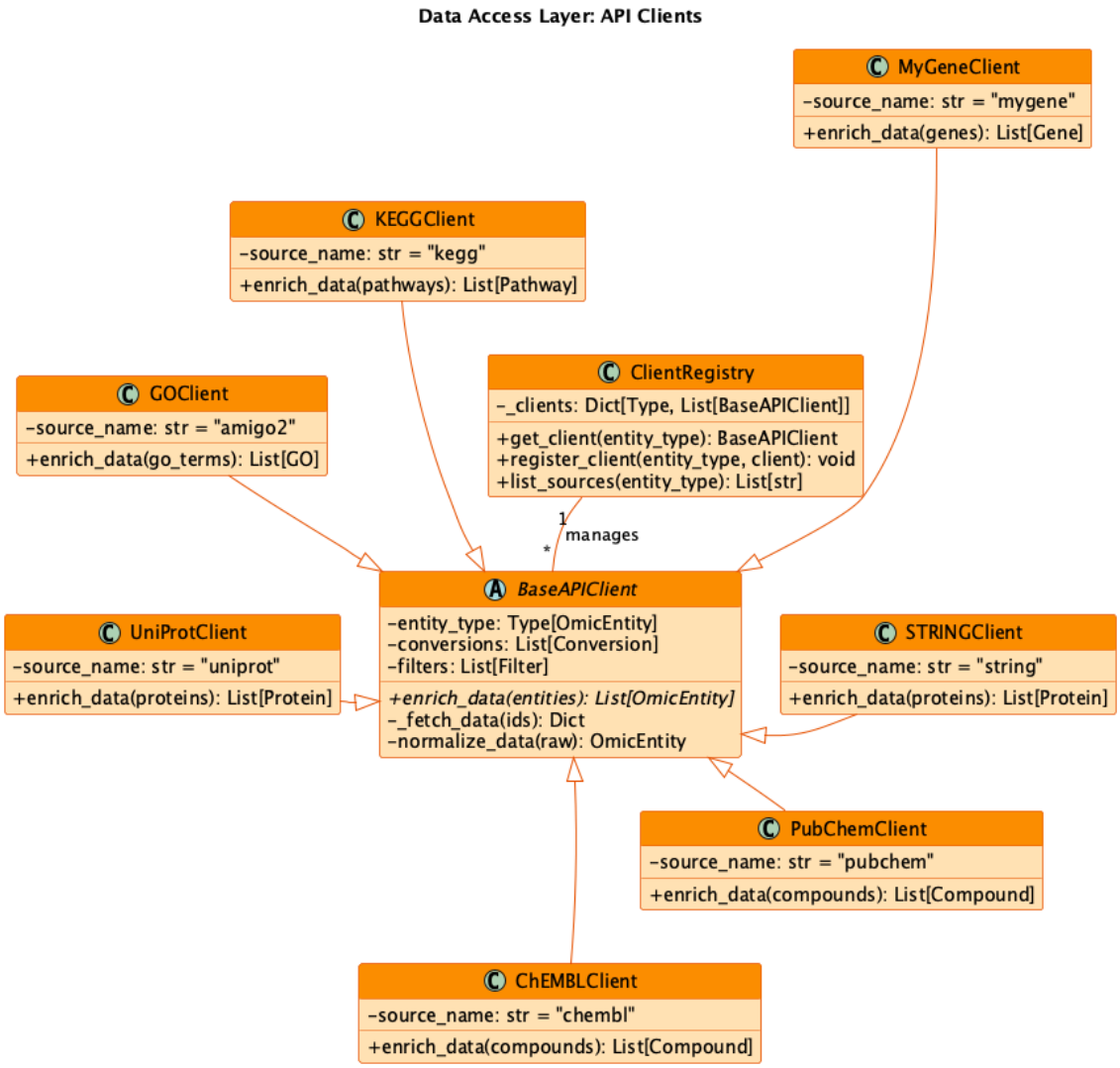


Figure 2.4: UML Diagram for the clients in the system.

that takes care of fitting the extracted data into our desired models.

3. Each specific client is then adapted to serve the particular needs of the database bridge. For example, `GOClient`, which requires a simple AmiGO2 initialization and query, and does not involve particular conversions, needs leaner implementations; on the contrary, as `UniProtClient` involves a full-fledged RESTful API client initialization and request, involves a more complex procedure.

The `BaseAPIClient` class is purposed to do the job of external communication with the sources. It is planned for involving just one client, and performing all the unique operations to that specific client. The pipeline was instead devised to choose "the better client" for the same entity. This means that in BioGraph, multiple `BaseAPIClient` classes can, and should, be added for the same entity. Of course, this additional layer of complexity

must not be cumbersome towards the entire process, therefore mechanisms, which will see later in section 2.2.3, are put in place so that only the necessary clients are loaded and initialized for every pipeline setup.

As an example, we already saw in section 1.5 that gene data is carried by numerous tools explored in this research. Of them, more than one (say, MyGene and Bioservices) have been added to the setup as available sources for gene data. The `BaseAPIClient` class will be inherited by all the separate classes that have been added for each of these sources. All these separate classes are added to the `DB_REGISTRY` as sources of data for the `Gene` entity. Again, in section 2.2.3 we will analyze more precisely how just one of them is chosen at pipeline construction.

Client registration and discovery

Clients are discovered and registered automatically through the decorator-based registry pattern. Each client implementation registers itself, making it available for BioGraph to select based on entity type and data source requirements. This supports BioGraph's zero-configuration extensibility principle.

2.3.5. Path discovery via graph search

The second step consists in exploring the discovered relationships in light of the submitted query. What effectively is built at the end of the first step is a graph, on which graph search algorithms can be performed.

PathFinder engine

The object responsible for this is a `PathFinder`. Once initialized, it uses the metadata just collected to build an adjacency matrix between entities, which is determined by whether a relationship was found between any two entities, and the type of relationship that was found. Then, the `find_all_paths()` method is invoked. It receives the start and finish entities, as well as an optional limit on the number of hops from the paths. This is done by using BFS.

BFS algorithm and path cost

Breadth-first Search (BFS) is a simple graph search algorithm. Starting from any node, all related entities are added to a tree-like structure. Once all are added, the algorithm goes down one level and repeats the same search for each child. Doing this recursively and iteratively helps build a graph-like structure, where relationships are directed and assigned

with a cost. Each relationship's cost can be arbitrarily defined, or inferred from sample measurements. Figure 2.5 illustrates the BFS traversal strategy used in path discovery.

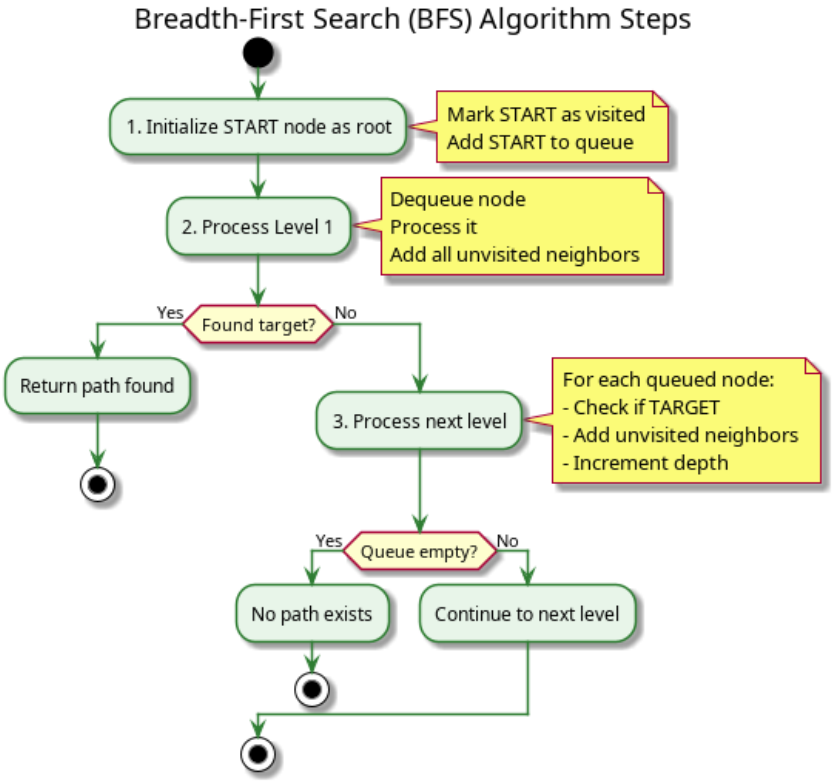


Figure 2.5: Breadth-first search algorithm for discovering paths through the entity relationship graph.

Multi-hop query examples

By looking at the entity relationships in figure 2.1, we can illustrate some examples of single- and multiple- hop queries:

- *Gene to GO*: This query is probably the easiest combination in terms of computational complexity and path-finding. As we saw in section 2.2.1, the `Gene` entity already possesses a `go_terms` attribute which contains `GO` entities. Additionally, if the gene data is collected from MyGene, the GO data which is already retrieved is quite complete in information.
- *Go to Pathway*: In this case, the possibility is twofold. `GO` does have a `genes` attribute, but the data source responsible for the GO terms (*AmiGO2*) does only provide identifier information about genes. Therefore, an API-driven hop is performed (*GO* $\rightarrow$ *gene IDs* $\rightarrow$ *MyGene* $\rightarrow$ *Gene objects*), as to obtain complete information about genes. The next operation ("*hop*") depends upon the choice of the user. The

`pathways` information provided by MyGene can be sufficient, or they can instead decide to API-mediate the hop with a query to KEGG. In any case, the operations are formally still considered to be two, hence this is a 2-hop query.

The essential result of the Path Discovery step is a (ordered) list of `EntityTypeRelationship` objects. This class contains all essential components of each "*hop*" of the query:

- The **source and target entity types** for referencing and querying;
- The **field name** of the attribute to reference in the source entity;
- A boolean flag for whether the relationship is a **collection** or not (i.e., "X-TO-MANY" versus "X-TO-ONE");
- A boolean flag for the **direct** and **API-mediated** accesses from source to target, to signify their availability;
- Various accessory metadata, such as the **access speeds** for both approaches, or other miscellanea (e.g., text description of the step, etc.).

Figure 2.6 illustrates a concrete multi-hop query path. In practice, the path discovery stage is lightweight compared to traversal because it operates primarily on entity metadata rather than on full API payloads. Its main cost comes from graph search over the entity-type graph, which remains small and bounded by the number of supported entity types rather than by the size of the underlying biological datasets. For the thesis scenarios, this means that path discovery is fast enough to be repeated often, while the dominant cost stays in downstream API access and result aggregation.

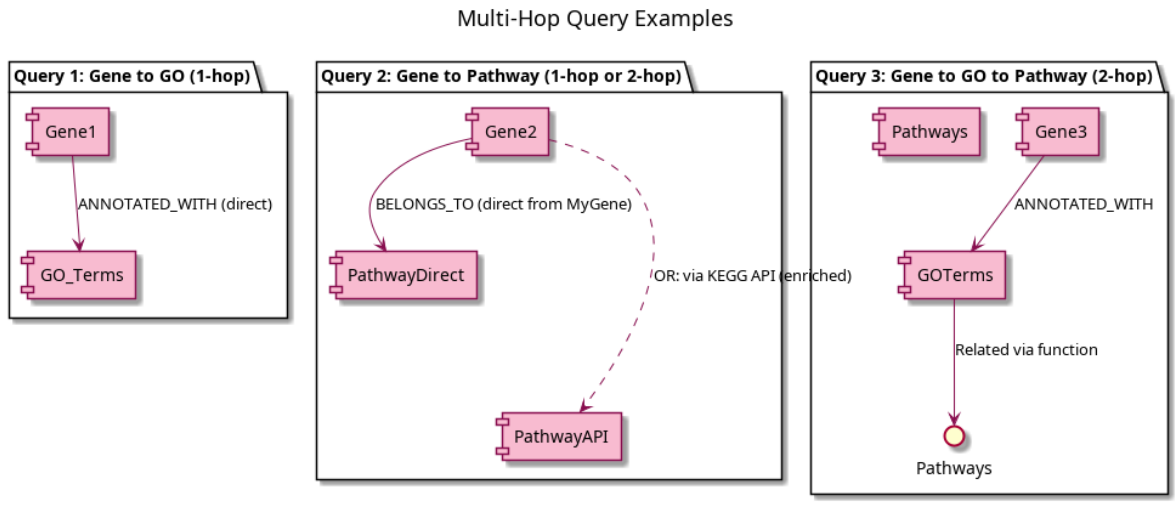


Figure 2.6: Multi-hop query example showing how BioGraph traverses from one entity type to another through intermediate relationships.

2.3.6. Universal traversal engine

The `UniversalTraversalEngine` class manages what has been planned so far and puts it into action. If seen from the perspective of the execution flow at figure 2.7, the Traversal Engine executes all actions starting at the "Yes" branch of the "*Path Found?*" condition, until all relationships are exhausted and the "*Deduplicate Entities*" step is taken. The core of its actions lies in the "*Traversal Strategy?*" condition, which splits into a direct-field relation versus an API-mediated one.

A core aspect to highlight is that the logical plane the pipeline now operates on is the "relationship-wise" aspect. The pipeline, having concluded its initial retrieval and planning phases, stops considering the query as a whole and iterates over the `EntityTypeRelationship` objects found in the previous step. According to the path found there, the engine will use one of two strategies, described by an appropriately defined `TraversalStrategy` enum: `DIRECT_FIELD` or `API_ACCESS`.

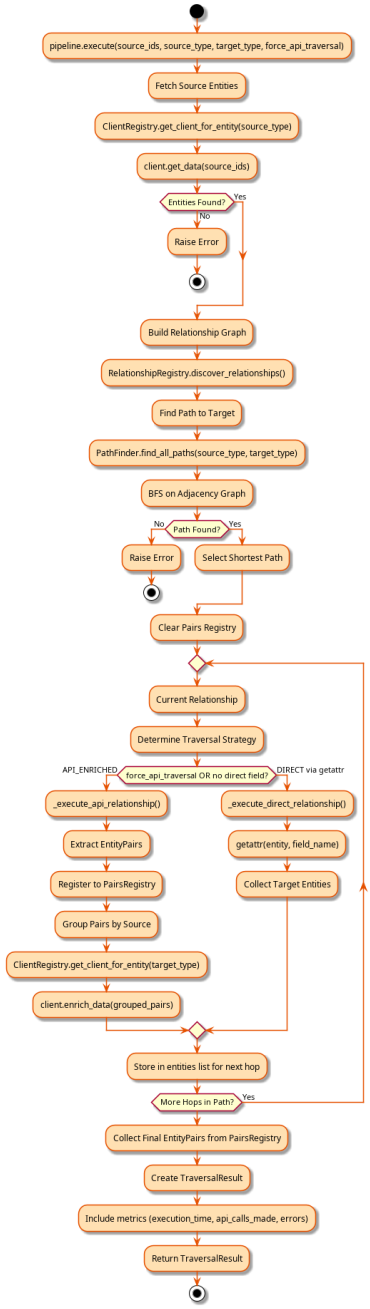


Figure 2.7: Execution flow diagram for the pipeline.

Direct field traversal strategy

This strategy is the lowest in computational cost, the more fault-tolerant, and (generally) the lower in information gain. As hinted previously in section 2.3.5, the data provided by the source entities is sometimes enough to simply access the appropriate attribute (`getattr()` in Python syntax), where the data, thanks to the preprocessing pipeline integrated into the *Clients*, is already parsed and objectified accordingly. Figure 2.8 shows the different traversal strategies available. For additional clarity, here's a pseudo-code representation of how this data is accessed:

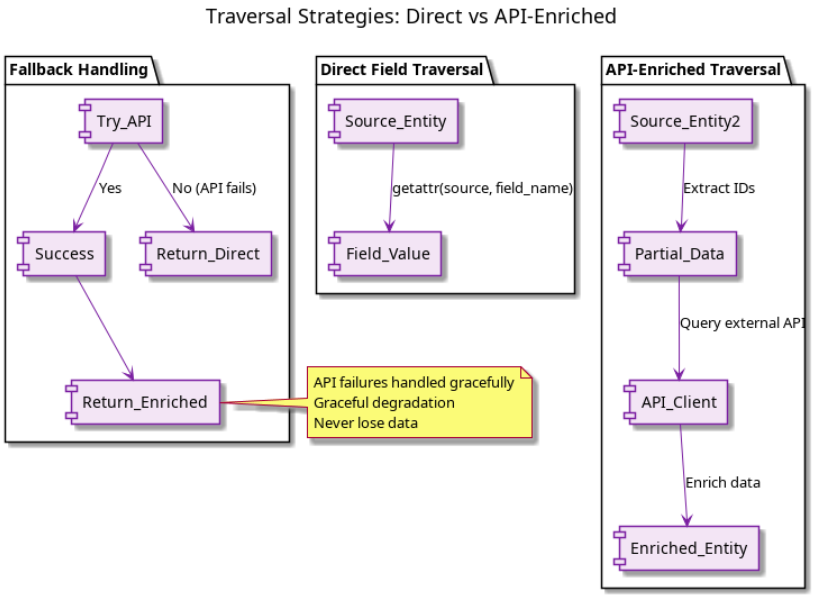


Figure 2.8: Different traversal strategies: direct field access versus API-mediated traversal and their respective performance tradeoffs.

Algorithm 2.3 _execute_direct_relationship(source_entities, relationship)

```
for each source in source_entities do
    field_value ← getattr(source, relationship.field_name)
    if field_value is not None then
        if relationship.is_collection then
            for each target in field_value do
                if target is OmicEntity then
                    pairs_registry.register_pair(source, target, relationship)
                end if
            end for
        else if field_value is OmicEntity then
            pairs_registry.register_pair(source, field_value, relationship)
        end if
    end if
end for
```

API-client traversal strategy

By contrast, this strategy is the ideal one for a more information-rich data collection pipeline. The underlying principle is simple: once any identifying information about the requested entities is found, the corresponding client is invoked to retrieve data about them.

An important caveat must be made, however. For a number of reasons, even the smallest amount of data for a target is still stored under the appropriate class. The motive to this is quite simply explained: were the target entity's client for any reason to stop abruptly, the pipeline can still continue to function and yield whichever information about the target it already has collected. To render the idea more clearly: say we want to traverse the *Gene to Pathway* step, the same one explained in section 2.3.5, through the API-client strategy.

After obtaining the Gene data, the `pathways` attribute already contains some information about the Pathways, but not all: some properties are in fact not provided by the Gene client. Say the Pathway client, however, fails in the process. Having the `pathways` information from before still stored and regularly parsed as `Pathway` objects (and not, say, just their IDs), becomes quite useful in this case.

Therefore, for all intents and purposes, the `API_CLIENT` code is surrounded by an all-encompassing try/catch block, which in case of exception falls back to the `DIRECT_FIELD` strategy. With this clarified, the API-mediated traversal strategy can be summarized with the following:

Algorithm 2.4 _execute_api_relationship(source_entities, relationship)

```
client ← get_client_for_entity(relationship.target_entity)
_extract_entity_pairs(source_entities, relationship)
pairs ← pairs_registry.get_pairs(source_entities, relationship.target_entity)
lookup_entities ← deduplicate targets from all pairs
lookup_results ← client.enrich_data(lookup_entities)
pairs_registry.update_target_entities(pairs, lookup_results)
```

Another important optimization to note is the level of batch querying applied.

Most tools we used for the Client objects allow for batch queries. Retrieving data in groups as opposed to iterating one entity by one significantly cuts request time down. Figure 2.9 shows the batch query optimization strategy. Our implementation achieves maximum batch efficiency through **intelligent deduplication**: instead of issuing multiple smaller batch queries per source entity, we:

1. Collect **all entity pairs** from all source entities and the relationship
2. Extract **all target entities** from these pairs
3. Deduplicate targets by entity ID (line 101 in traversal_engine.py from algorithm 2.4: `str(ent.id): ent for ent in lookup_entities`)

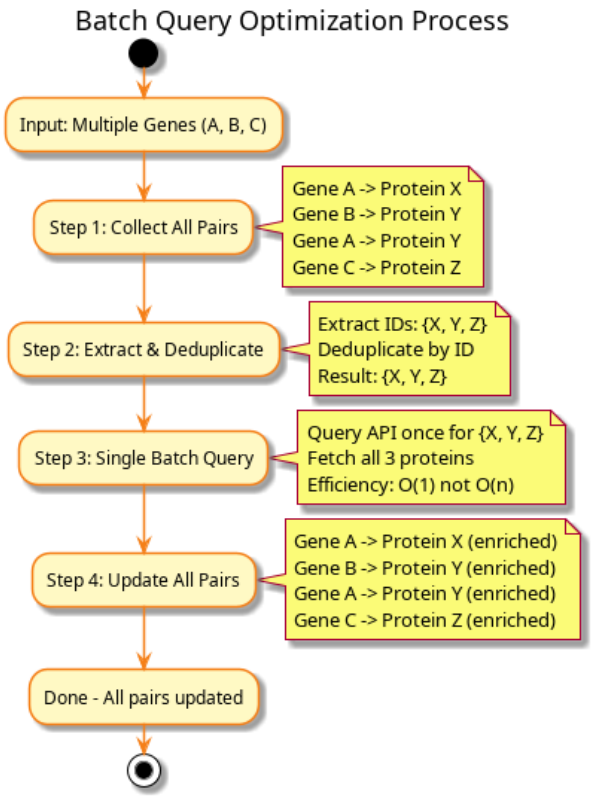


Figure 2.9: Batch query optimization strategy for minimizing API calls and reducing latency.

4. Issue **one single batch query** to the API client with the merged, deduplicated list
5. Update **all entity pairs** with the enriched results

This approach achieves the best of both worlds: maximum API batch efficiency (one call instead of N calls) while maintaining complete relationship tracking (each pair knows which source and target it connects). The deduplication ensures that if Gene A and Gene B both code for Protein X, we only query the API once for Protein X, then assign the result to both pairs.

For example, if we want to retrieve all Pathways from two Genes (Gene A and Gene B), the old approach would have required:

- Query 1: Gene A $\rightarrow$ {Pathway X, Pathway Y}
- Query 2: Gene B $\rightarrow$ {Pathway Y, Pathway Z}

This wastes bandwidth on duplicate queries (Pathway Y queried twice).

Our optimized approach:

- Merged Dedup List: {Pathway X, Pathway Y, Pathway Z}
- Single Query: Fetch all three pathways in one batch
- Update: Assign results back to (Gene A → Pathway X), (Gene A → Pathway Y), (Gene B → Pathway Y), (Gene B → Pathway Z)

The trade-off previously described (separate queries to maintain relationships) has been eliminated, allowing us to achieve maximum query efficiency without sacrificing relationship tracking.

Entity deduplication

As can follow from the note made just above, the result from the traversal of the step is an array of `EntityPair` objects, enriched with the requested data, and in most cases containing few unique source entities. By default, BioGraph uses *union semantics* (ANY criteria matching), which collects all entities associated with any input entity. This comprehensive approach maximizes biological discovery. Entity pairs containing identical sources and targets are deduplicated by entity ID, ensuring each relationship appears once in the result. Results are then arranged from sequential (array) form into hierarchical (tree-like) structures, where parent entities become unique and link to all their children. Figure 2.10 illustrates this tracking and deduplication process.

2.3.7. Query execution and result aggregation semantics

By default, BioGraph implements *union semantics* (ANY matching criteria), which collects all entities associated with any input entity. For example, querying Gene A and Gene B for Pathways returns all pathways associated with *either* gene, providing comprehensive biological context.

Deduplication and complexity analysis

The intelligent deduplication process (Algorithm 2.4) ensures maximum API efficiency. If multiple genes code for the same protein, the API is queried once for that protein, then the result is linked to all relevant pairs. This maintains complete relationship tracking while avoiding redundant API calls. Time complexity is $O(n \log n)$ due to deduplication sorting, where n is the number of entity pairs. The batch query optimization (Section 2.3.6) reduces actual API calls from $O(m)$ (where m is average result size per source) to $O(1)$, achieving both query efficiency and relationship integrity.

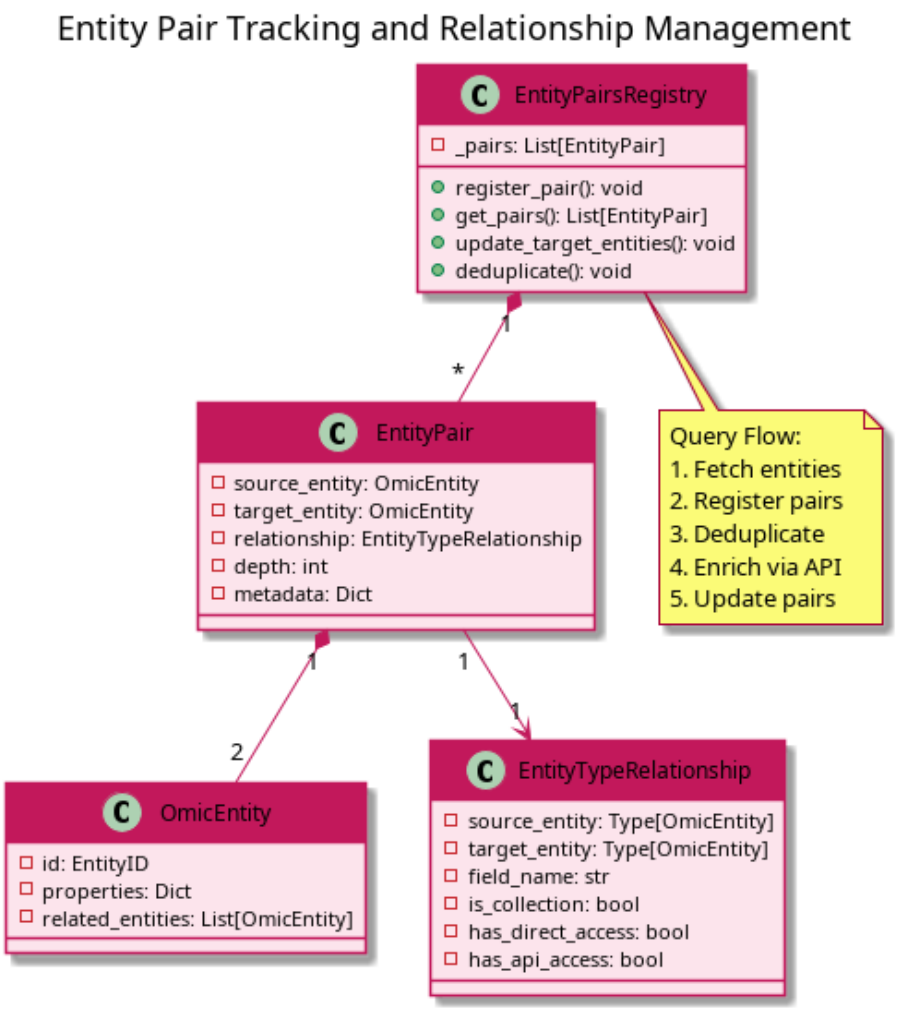


Figure 2.10: Entity pair tracking and deduplication showing how source-target relationships are maintained and deduplicated during traversal.

2.4. Query execution modes

Having outlined the major components of BioGraph, we now describe the three complementary query modes that enable different research workflows. These modes form a hierarchy where simpler modes provide building blocks for more complex ones, following the Open-Closed Principle: each level is closed for modification but open for extension by higher levels.

2.4.1. The method hierarchy

BioGraph implements a carefully designed method hierarchy that reflects increasing query complexity:

1. `enrich()` — Base level: Direct entity enrichment without traversal. Used when researchers have specific entity IDs and only need enriched data from a single data source.
2. `_fetch_source_entities()` — Intermediate level: Wraps enrich() with caching optimization for subsequent traversals. Internal method used during exploration to maintain entity cache across multiple queries.
3. `execute()` — Complex level: Builds on entity fetching to discover and traverse optimal paths to a specific target entity type. Used for directed queries with known targets and finite result expectations.
4. `explore()` — Advanced level: Extends `execute()` with iterative depth-based expansion to discover all reachable entities. Used for open-ended exploration without predetermined targets, maximizing biological discovery.

This hierarchy ensures that lower-level methods are optimized for performance (e.g., `enrich()` makes only a single API call) while higher-level methods leverage these optimizations. For instance, both `execute()` and `explore()` internally use `enrich()` via `_fetch_source_entities()` to retrieve and enrich their initial source entities, ensuring consistent entity enrichment throughout the query execution process.

2.4.2. Simple entity enrichment: `enrich()` method

The simplest query mode is entity enrichment without relationship traversal. This mode is useful when researchers have specific entity identifiers (gene symbols, protein accessions, pathway IDs) and only need enriched data from a single data source without exploring relationships to other entity types.

Use cases:

- *Batch enrichment*: Input a list of 100 gene symbols, receive complete Gene entities with all available annotations
- *API standardization*: Replace direct API calls with standardized OmicEntity objects for consistent downstream processing
- *Data standardization*: Convert heterogeneous identifiers (different gene naming conventions) into unified entity format
- *Foundation for exploration*: Serves as source data provider for other query modes
- *Data validation*: Verify that input identifiers exist in data sources before launching

expensive traversals

Operation:

The `enrich()` method follows a straightforward three-step pattern (Algorithm 2.5):

Algorithm 2.5 `enrich`(entity_ids, entity_type)

client ← `get_client_for_entity`(entity_type)
if client is None **then**
    return error: "No client available for entity type"
end if
enriched_entities ← client.`get_data`(entity_ids)
return enriched_entities

Characteristics:

- **Traversal depth**: 0 hops (no graph traversal, single data source only)
- **API calls**: Exactly 1 call per entity type (batch query optimization applies)
- **Result size**: Directly proportional to input entity count
- **Error resilience**: Returns whatever data client can fetch; partial results acceptable. If some IDs are not found, BioGraph returns the subset that exist.
- **Computational cost**: Lowest among all three modes (single API call, no traversal logic)
- **Result completeness**: High within single data source (all available fields populated by API)

Practical example:

A researcher with a list of 50 protein accessions might use this mode:

```
protein_ids = ['P69905', 'P09105', 'P12931', ...]
enriched_proteins = pipeline.enrich(
    entity_ids=protein_ids,
    entity_type=Protein
)
# Result: List[Protein] with complete data from UniProt API
# Each Protein object contains sequence, structure, annotations, etc.
```

The method serves as the *foundation* for more complex queries. Both `execute()` and `explore()` internally use `enrich()` (via `_fetch_source_entities()`) to retrieve their initial source entities, ensuring that all downstream traversals begin with fully enriched data.

2.4.3. Directed query execution: `execute()` method

The execute() method represents the second level of complexity in our query hierarchy. This execution type, outlined in the pseudocode at algorithm 2.6, builds upon entity enrichment to discover and traverse optimal paths to a specific target entity type.

The core of this mode is in the Traversal Engine's `traverse_path()` method (section 2.3.6), which executes the discovered path. Crucially, the first step—`fetch_source_entities()`—internally delegates to the `enrich()` method described in Section 2.4.2, ensuring that source entities are fully enriched before path discovery begins. This hierarchical composition allows `execute()` to leverage all entity enrichment optimizations.

Algorithm 2.6 `execute`(source_ids, source_type, target_type)

```
source_entities ← fetch_source_entities(source_ids, source_type)
if source_entities is empty then
    return ∅
end if
paths ← path_finder.find_all_paths(source_type, target_type)
if paths is empty then
    return error: "No path found"
end if
selected_path ← select_minimum_cost_path(paths)
result_pairs ← traversal_engine.traverse_path(
     source_entities, selected_path)
return result_pairs
```

The `execute()` mode is the most appropriate choice when the researcher already knows the target entity type and wants a directed result rather than an open-ended expansion. Typical use cases include finding pathways for a gene set, retrieving PPIs for a list of proteins, or tracing genes associated with a disease-related query. Compared with `explore()`, it keeps the search space narrower and the interpretation simpler, because it returns only the shortest valid path to the target type instead of every reachable relationship within a depth bound.

2.4.4. Radial graph exploration: `explore()` method

The explore() method represents the highest level of complexity in our query hierarchy. It operates at the advanced level, suitable for open-ended research exploration. While its own logic implements frontier-based iteration (Algorithm 2.7), it internally calls `execute()` for each frontier expansion step, which in turn uses `enrich()` for entity fetching. This cascading hierarchy ensures consistent entity enrichment and relationship discovery throughout the entire exploration process.

The internal working of this feature is much similar to what already seen for the `execute()` feature. As a matter of fact, the preliminary steps are completely shared with the latter, for instance the client, entity and relationships registrations, which follow the same logical flow. However, the substantial difference comes in the `PathFinder`'s role. In the sequential execution its task is essentially a search problem, with a start point, and a goal to reach using the lowest path cost; in the radial execution flow, it needs to generate all legal paths that can be taken within a certain depth threshold.

Algorithm 2.7 `explore`(source_ids, source_type, max_depth)

```
frontier ← fetch_source_entities(source_ids, source_type)
explored_ids ← {}
all_pairs ← [ ]
depth ← 0
while frontier is not empty and depth < max_depth do
    unique_frontier ← deduplicate_by_id(frontier)
    next_frontier ← [ ]                          ▷ collect next-depth entities separately
    for each entity_type, entity_ids in group_by_type(unique_frontier) do
        explored_ids ← explored_ids ∪ entity_ids
        possible_target_types ← show_possible_queries(entity_type)
        for each target_type in possible_target_types do
            pairs ← execute(entity_ids, entity_type, target_type)          ▷ batched
            all_pairs.extend(pairs)
            for each target_entity in pairs do
                target_id ← get_entity_id(target_entity)
                if target_id ∉ explored_ids then
                    next_frontier.append(target_entity)
                end if
            end for
        end for
    end for
    frontier ← next_frontier ▷ swap; depth increments only after all types processed
    depth ← depth + 1
end while
return all_pairs
```

2.4.5. Comparison of query execution modes

Table 2.1 provides a comprehensive comparison of all three query execution modes:

Table 2.1: Query execution modes comparison

Characteristic	**enrich()**	**execute()**	**explore()**
Target specification	None required	Target entity type	Max depth only
Traversal depth	0 hops	1 hop (shortest)	N hops (user-defined)
Path discovery	None	BFS to target	All paths within depth
Result shape	Flat list	Directed tree	Undirected web
Use case	Batch enrichment	Targeted discovery	Open exploration
API calls	1 call	O(path length)	Depends on depth
Result completeness	High (1 source)	Medium (aggregated)	High (all paths)
Computational cost	Lowest	Medium	Highest

Practical example comparison:

Consider a query starting from genes *BRCA1* and *TP53*, exploring their biological context:

- **enrich([BRCA1, TP53], Gene)**: Returns only the two enriched Gene entities. Each Gene object contains complete annotations (symbol, Entrez ID, chromosomal coordinates, GO terms, protein links). No traversal to other entity types. Query time: 200ms.
- **execute([BRCA1, TP53], Gene, Pathway)**: Discovers the shortest path from Genes to Pathways (typically via direct-field relationship or single API hop). Returns all Pathways associated with either gene. Result includes pathway names, IDs, gene members, and biological processes. Query time: 500ms depending on pathway count.
- **explore([BRCA1, TP53], Gene, max_depth=2)**: Expands iteratively:
  - Depth 1: Discovers Proteins (via coding relationship), GO terms (direct from Gene), and Pathways (via KEGG)

– Depth 2: From Proteins, finds Pathways (via protein function), PPIs (protein-protein interactions), chemical interactions. From Pathways, finds additional Genes (cycling back and expanding)

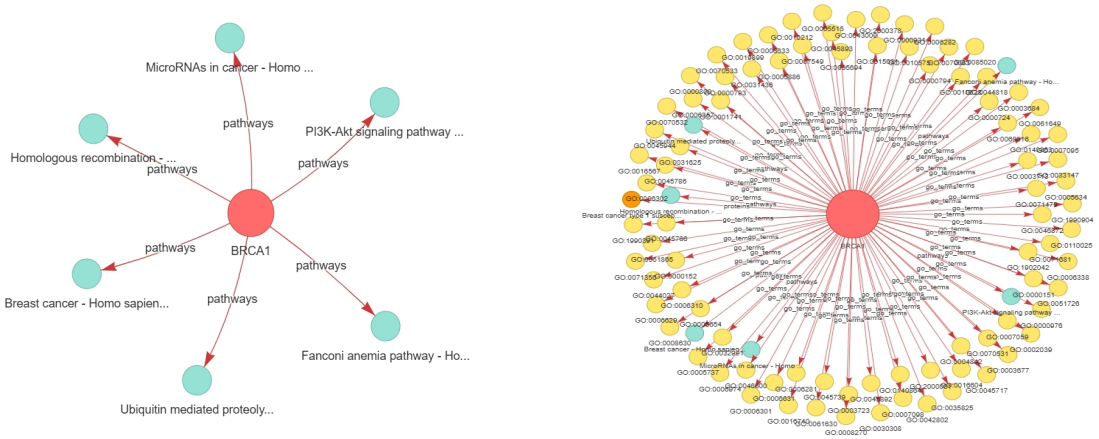
Result: Comprehensive network of biological relationships. Query time: 2-5 seconds depending on entity density.

Decision guide for researchers:

When to use each mode:

1. Use `enrich()` when you have specific entity IDs and need complete annotations quickly without exploring relationships
2. Use `execute()` when you know the target entity type you want to reach (e.g., "find all pathways for these genes")
3. Use `explore()` when you want to discover unexpected connections or generate hypotheses about biological relationships

Figure 2.11 shows the visual difference between the two graph outputs for a query starting from *BRCA1*.



(a) `execute()` query results (depth=1, targeted)

(b) `explore()` query results (depth=2, open-ended)

Figure 2.11: Result of queries starting from gene *BRCA1*, using execute() vs explore() modes. The execute() query has Pathways as target and yields a directed tree. The explore() query discovers all entities within 2 hops, yielding an undirected web of relationships.

2.5. Additional characteristics of BioGraph

Figure 2.12 shows the interaction between BioGraph's major components during query execution. The component diagram highlights how the Orchestration Layer coordinates between the client registry, the relationship registry, and the traversal engine without any component holding direct references to the others' internal state. Figure 2.13 traces the data flow from raw API responses through parsing, normalisation, filtering, and conversion into the final `EntityPair` collection.

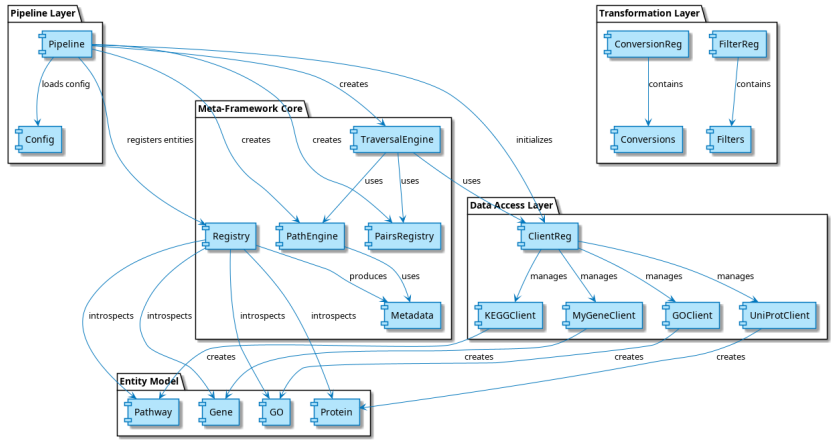


Figure 2.12: Component interaction diagram showing how different system components communicate and coordinate during query execution.

2.5.1. Execution metrics and monitoring

Every call to `execute()` or `explore()` returns a `TraversalResult` dataclass that bundles the output with its provenance. Its principal fields are: `pairs` (the discovered `EntityPair` list), `execution_time` (wall-clock seconds), `api_calls_made` (total API calls issued during the traversal), `errors` (list of non-fatal error messages that triggered graceful degradation), `unfiltered_pairs` (pairs before any post-traversal filters are applied), and `stage_statistics` (per-hop breakdown of input/output/duplicate entity counts). Returning a structured result object rather than a bare list allows downstream code to inspect provenance and performance data without separate instrumentation.

BioGraph integrates a singleton `MetricsCollector` that records metrics at two granularities. At the *operation level*, each call to `start_operation()` / `end_operation()` captures the operation name and type, start and end timestamps, input and output entity counts, data size in bytes, and a resource snapshot (process RAM, system RAM, CPU utilisation) at both boundaries. At the *API call level*, every client invocation appends an `APICallMetrics` record containing the client name, batch number, lookup term count, response record count, and per-call duration. The collector is accessed via `get_metrics_collector()` and can be exported to JSON through `MetricsExporter`, which generates per-operation reports, API call summaries, and resource usage timelines. The case study metrics reported in Chapter 3 were collected through this infrastructure.

2.6. Implementation and technical details

Type hints and Python typing. Type hints via Python's `typing` module enable zero-configuration discovery using `typing.get_type_hints()` for runtime introspection. This mechanism automatically builds the relationship graph without manual configuration.

Configuration management. Custom configuration is loaded via `ConfigLoader` using YAML files, enabling transformation customization, filter specification, and API client options per pipeline run.

Error handling, resilience, and core design patterns. BioGraph implements graceful degradation: API failures trigger fallback to direct-field traversal (`DIRECT_FIELD`) without halting execution. Core patterns include Decorator (client registration), Strategy (traversal mode selection), and Registry (entity/client/filter autodiscovery). These patterns enable extensibility without framework modification.

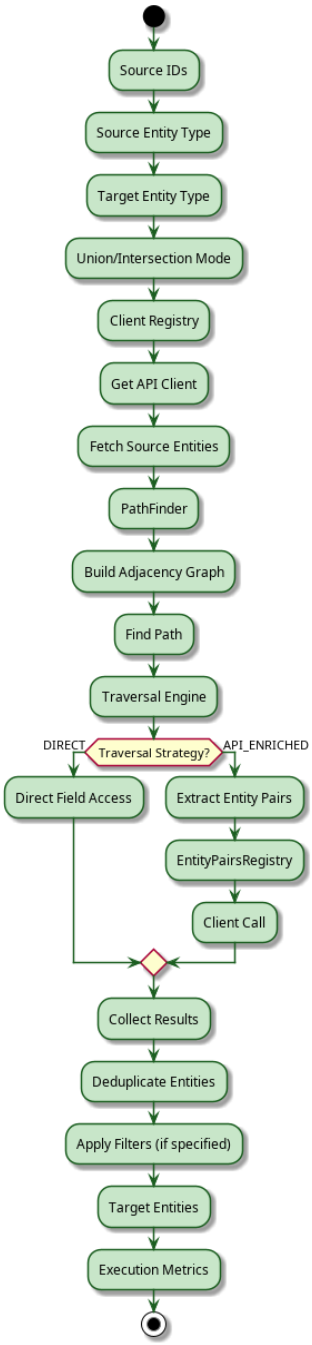


Figure 2.13: Data flow through the pipeline stages.

3 | Results

3.1. Experimental results

To validate the BioGraph's effectiveness, we conducted three case studies demonstrating different traversal patterns and API integration strategies. All experiments were executed with comprehensive metrics collection via the integrated `MetricsCollector` system, tracking API calls, execution time, memory usage, and resource consumption.

3.1.1. Metrics collection and observability

All experiments employ a unified metrics collection infrastructure integrated into the pipeline execution engine. The `MetricsCollector` singleton tracks:

- **Operation-level metrics**: operation type, start/end timestamps, duration, input/output entity counts, data sizes
- **API call tracking**: client name, batch information, lookup term counts, response record counts, per-call timing
- **Resource snapshots**: process RAM (MB), system RAM usage (MB), CPU utilization (%), captured at operation start and end
- **Peak resource utilization**: maximum observed RAM during operation execution

This instrumentation enables post-hoc analysis of query performance, cost estimation, and bottleneck identification without modifying application code.

3.1.2. Case Study 1: Gene Exploration with Entity Type Filtering

Our objective in this case study was to evaluate the `explore()` method with entity type filtering, demonstrating scalability with large gene inputs and validating the selective entity type feature for focused biological queries.

BioGraph Results

We configured the exploration experiment as follows. We loaded 1,000 gene symbols and executed `explore()` with `allowed_entity_types=[Gene, Pathway]`, restricting results to Gene and Pathway entities while allowing traversal through other intermediate types. We set the maximum exploration depth to 2 hops from the source genes.

Performance metrics:

Table 3.1: Case Study 1 - BioGraph: Gene Exploration with Entity Type Filtering

Stage	Execution Time (s)	API Calls	Entity Pairs	Peak RAM (MB)	Pairs Discovered
Depth 0 (Gene→Pathway)	75.13	1	666	1,469	666
Depth 1 (Pathway→Gene)	78.47	1	27,495	1,469	28,161
Total Exploration	**153.60**	**2**	**28,161**	**1,469**	**28,161**

In summary, the exploration discovered 28,161 Gene-Pathway pairs across both depths within 153.60 seconds. At depth 0, BioGraph identified 666 Gene→Pathway relationships via the KEGG API. At depth 1, it discovered 27,495 Pathway→Gene relationships via MyGene API. The large result set (1,474 MB in JSON output) was managed efficiently within a peak RAM of 1,469 MB, demonstrating that entity filtering enables focused exploration. By restricting results to Gene and Pathway types while allowing traversal through other intermediate types, BioGraph reduces the result volume for domain-specific workflows without sacrificing relationship discovery capability. This approach is particularly useful for researchers who wish to explore large gene sets while remaining within a specific biological context.

Comparison with related tools

The point of this case study is BioGraph's pathway discovery, but it is useful to compare it against a single-purpose batch gene lookup tool. MyGene provides a fast baseline for gene annotation, while BioGraph expands the query into pathway traversal and cross-entity discovery.

Table 3.2: Case Study 1 - Comparison context for gene exploration

Tool	Time (ms)	API Calls	Results	Data (MB)	Depth
MyGene (batch)	17,176	1	844 genes	11.55	1
BioGraph	**153,600**	**2**	**28,161 pairs**	**1,474**	**2**

Tool characteristics in practice: MyGene is optimized for single-hop gene annotation—it retrieves metadata for a batch of gene symbols with minimal overhead. PrimeKG is a static pre-computed knowledge graph: it is fast because the data is fixed and locally queried. However, its edge records are keyed by Entrez gene IDs and do not include KEGG pathway annotations, meaning that a query restricted to Gene and Pathway entities (excluding Proteins) finds no matching relationships in PrimeKG—not a deficiency in the tool, but correct evidence that tool selection must align with query objectives. Our framework is slower than MyGene because it performs multi-hop traversal and relationship aggregation across multiple sources, not merely annotation retrieval. Thus, the comparison illustrates differences in scope and discovery power rather than any failure to replace a specialized gene lookup service. BioGraph's value lies not in replacing specialized tools, but in abstracting the orchestration burden so users compose workflows according to biological logic rather than tool availability.

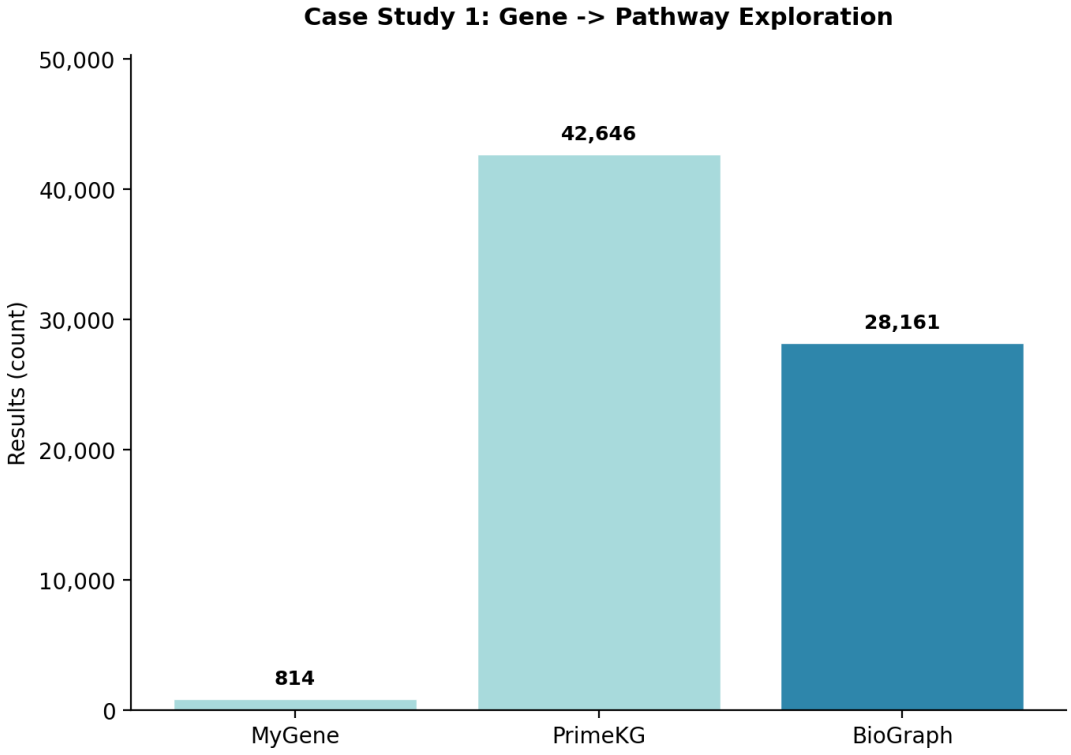


Figure 3.1: Case Study 1 tool comparison: MyGene batch annotation (844 genes annotated), PrimeKG (0 gene-pathway pairs, as it does not store KEGG pathway edges), and BioGraph with entity filtering (28,161 gene-pathway pairs via multi-hop traversal).

3.1.3. Case Study 2: Genes to PPIs

Our objective in this case study was to evaluate BioGraph's behavior with limited relationship availability by traversing from genes through protein intermediates to protein-protein interactions.

BioGraph Results

We designed the traversal as a multi-step pipeline. We queried three different genes (via MyGene), mapped them to their corresponding proteins (via UniProt), and then retrieved the protein-protein interactions each protein is involved in (via STRING DB). To demonstrate batching efficiency, queries were executed both individually and in combination. A limitation noted during implementation is that the complex nature of gene-to-protein interactions led us to consider only each gene's primary protein in this analysis.

Performance metrics:

Table 3.3: Case Study 2 - BioGraph: Gene to Protein to PPI Traversal

Query	Time (ms)	API Calls	Pairs	Peak RAM (MB)
BRCA1	9,659	3	459	213.6
TP53	9,292	3	590	213.6
CDKN1A	5,648	3	106	224.2
All 3 genes (sum)	**24,599**	**9**	**1,155**	**224.2**
All 3 genes (combined)	**14,544**	**3**	**1,155**	**224.2**

The combined query execution demonstrates striking efficiency gains. When we combined the three genes into a single batched query, BioGraph achieved 41% runtime savings (14.5 seconds versus 24.6 seconds for individual queries) while maintaining identical result coverage. API call optimization was even more dramatic: the combined query required only 3 calls versus 9 for individual queries, a 67% reduction. The result output (61 KB) remained consistent across both execution modes, and memory profile remained stable at 213–224 MB, demonstrating predictable and manageable resource utilization for this traversal pattern. This case study illustrates a key advantage of automatic batching: BioGraph can intelligently group queries across multiple source entities without requiring manual orchestration, delivering both speed and correctness simultaneously.

Comparison with related tools

Protein-protein interaction databases are useful reference points, but the central result here is BioGraph's multi-hop traversal from genes to proteins to PPIs. The comparison below positions BioGraph relative to specialized single-hop tools.

Table 3.4: Case Study 2 - Comparative context for PPI traversal

Tool	Time (ms)	API Calls	Results	Depth
OmniPath (direct PPIs, 3 genes)	599	1	631	1
STRING DB (batch, 3 genes)	1,200	1	500	1
BioGraph	**14,544**	**3**	**1,155**	**2**

BioGraph returns 1,155 PPI interactions across three genes because it traverses the full Gene→Protein→PPI chain automatically without requiring users to manually specify each intermediate step. OmniPath and STRING remain useful for single-hop PPI lookup and batch retrieval, but they do not capture the same multi-hop discovery task. The central result demonstrated in this case study is BioGraph's ability to combine traversal and automatic batching without manual orchestration.

3.1.4. Case Study 3: Multi-gene Radial Network Exploration

Our objective in this case study was to demonstrate open-ended exploratory queries using the `explore()` method to discover all connected entities without pre-specifying target types. This case study is reported separately because `explore()` is not a target-specific query like `execute()`; it is an open-ended discovery mode with a different result shape and different cost profile.

BioGraph Results

We configured the exploration to operate as an open-ended radial graph exploration. We queried three cancer-related genes (BRCA1, TP53, CDKN1A), set the maximum exploration depth to 3 hops from the source genes, and imposed no per-hop result limits to allow full discovery of all connected entities.

Performance metrics:

Deduplication efficiency analysis: The combined query discovered 124,852 pairs, representing 51.6% coverage compared to individual explorations (241,923 total). This apparent reduction reflects BioGraph's intelligent deduplication strategy: when multiple source genes share downstream entities (pathways, GO terms, proteins), the combined

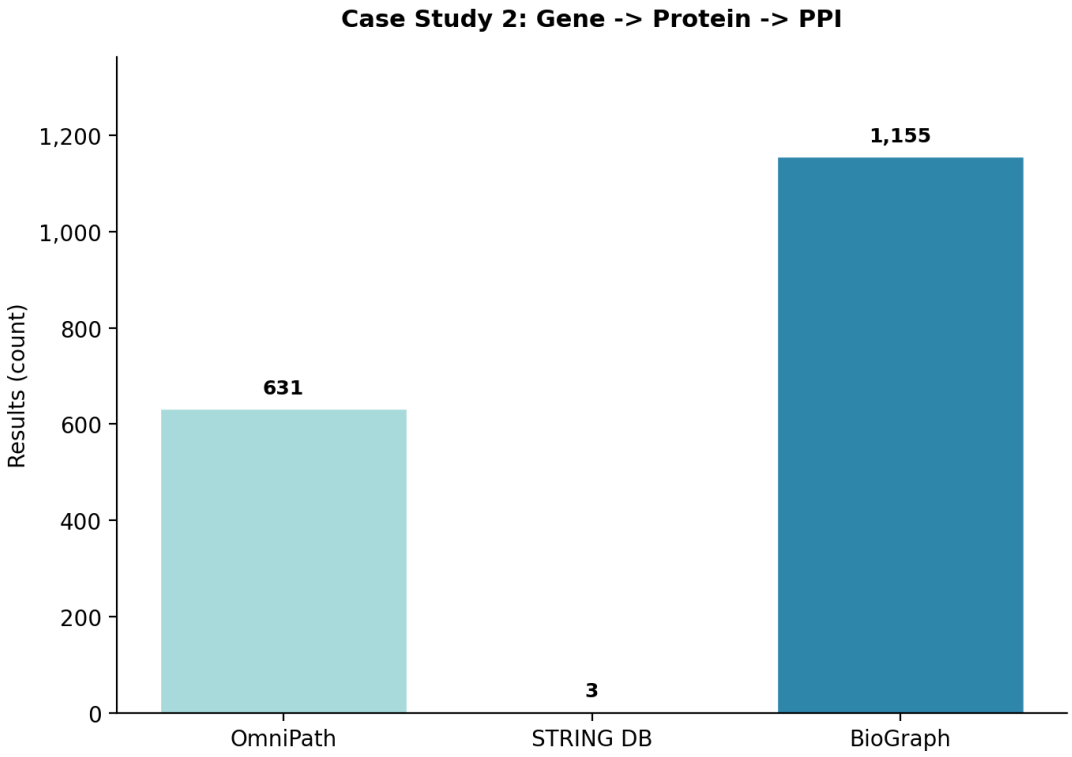


Figure 3.2: Case Study 2 tool comparison: OmniPath direct PPI query (631 results), STRING DB PPI network (500 results), and BioGraph multi-hop traversal (1,155 results). OmniPath and STRING offer single-hop PPI lookups; BioGraph integrates multiple APIs to perform complete Gene→Protein→PPI traversal automatically.

query registers each entity only once per relationship type, whereas individual queries register them independently. Consequently, the combined query achieves significant efficiency gains of 55% reduction in API calls (216 versus 496), 55.6% reduction in execution time (481.07 seconds versus 1,084.07 seconds), and a unified result set that eliminates redundant cross-gene relationship discovery.

Entity relationship type breakdown. Table 3.6 decomposes the combined exploration by relationship type, reporting the pairs discovered, API calls issued, and cumulative time spent on each.

Key findings from this exploration:

Network complexity varies significantly by gene. TP53 discovery yielded the largest network with 114,236 pairs, representing 47% of the total, reflecting its central transcription factor role with extensive regulatory reach across numerous pathways and biological processes. In contrast, BRCA1 and CDKN1A produced more focused networks (38,798 and 88,889 pairs, respectively), suggesting more specialized biological roles.

Table 3.5: Case Study 3 - BioGraph: Multi-gene Radial Exploration

Gene(s)	Total Pairs	API Calls	Duration (s)	Max Depth
BRCA1	38,798	113	283.22	3
TP53	114,236	205	445.49	3
CDKN1A	88,889	178	355.36	3
All 3 genes (indiv. sum)	**241,923**	**496**	**1,084.07**	**3**
All 3 genes (combined)	**124,852**	**216**	**481.07**	**3**

Table 3.6: Case Study 3: Entity Relationship Types (All 3 Genes Combined). API calls per relationship type reflect cumulative calls across all traversal steps at all depths; Gene→GO is high because GO enrichment is repeated for each gene discovered at each depth level.

Relationship	Total Pairs	API Calls	Total Time (s)
Gene→GO Terms	170,967	471	611.41
Gene→Pathway	49,511	3	332.49
Pathway→Gene	15,326	3	93.05
Gene→Protein	5,152	3	5.76
Protein→PPI	300	3	15.32
PPI→Protein	300	0	1.64

Functional annotations dominate the discovered relationship types. Gene→GO relationships account for 170,967 pairs (71% of all discovered), while Gene→Pathway adds 49,511 pairs (20%), together representing 91% of discovered relationships. This distribution reflects the curated nature of public biological databases, where functional annotations are more comprehensively populated than protein interaction data.

The computational intensity correlates directly with annotation richness. Gene→GO dominated API resource consumption, accounting for 471 of 483 cumulative API calls across all relationship types (97%), demonstrating that functional enrichment is the computationally intensive component of radial exploration.

A critical architectural advantage is demonstrated by the automatic relationship discovery across all 6 entity relationship types (Gene→GO, Gene→Pathway, Pathway→Gene, Gene→Protein, Protein→PPI, PPI→Protein) without any hardcoding of query paths or pre-specification of target entities. Unlike Case Studies 1 and 2, which required explicit target type specification, the `explore()` method required only a source entity and depth limit. This automatic relationship discovery is the primary architectural advantage of the BioGraph.

Comparison with related tools

To provide a rigorous reference point, we ran a proper evaluation of PrimeKG [2] using ID harmonization via MyGene.info. After converting gene symbols to Entrez IDs (BRCA1→672, TP53→7157, CDKN1A→1026), PrimeKG returned **5,750 unique relationships** across 16 relation types in **2.5 seconds** (1 API call for harmonization, then local file queries). Galaxy was also evaluated as a workflow-based alternative; it returned 5 tool discovery results, reflecting its different purpose as a workflow orchestration platform rather than a relationship discovery tool.

This yields a direct design trade-off: PrimeKG is 193× faster than BioGraph for this query, while BioGraph discovers 21.7× more relationships. The difference in scope arises from traversal depth: PrimeKG operates at depth 1 against a pre-built index, while BioGraph performs 3-hop live traversal across 6 entity types including functional annotations (Gene→GO: 170,967 pairs) not present in PrimeKG's schema. The ID harmonization step (864 ms) is a one-time cost amortized across all queries; BioGraph performs equivalent harmonization automatically at each traversal step.

The appropriate framing is not "which tool wins" but which design fits the research objective: PrimeKG is the right choice for fast, structured queries against a known schema; BioGraph is the right choice for open-ended discovery across heterogeneous live sources without prior knowledge of what queries are valid.

3.1.5. Integrated analysis across case studies

Table 3.7 summarizes the three case studies. The key distinction is that `execute()` is a target-oriented query, while `explore()` is an open-ended discovery mode evaluated as a separate result class.

Table 3.7: Overall experimental performance summary for BioGraph

Case Study	Mode	Time (s)	API	Pairs	Interpretation
CS1	`explore()` filtered	153.60	2	28,161	Broad gene-to-pathway discovery
CS2	`execute()`	14.54	3	1,155	Targeted gene-to-PPI traversal
CS3	`explore()`	481.07	216	124,852	Open-ended radial exploration

The three case studies (Sections 3.1.2, 3.1.3, and 3.1.4) collectively demonstrate several key insights. First, BioGraph exhibits strong API efficiency and batching behavior, particularly evident in the combined Case Study 3 run where repeated discoveries are

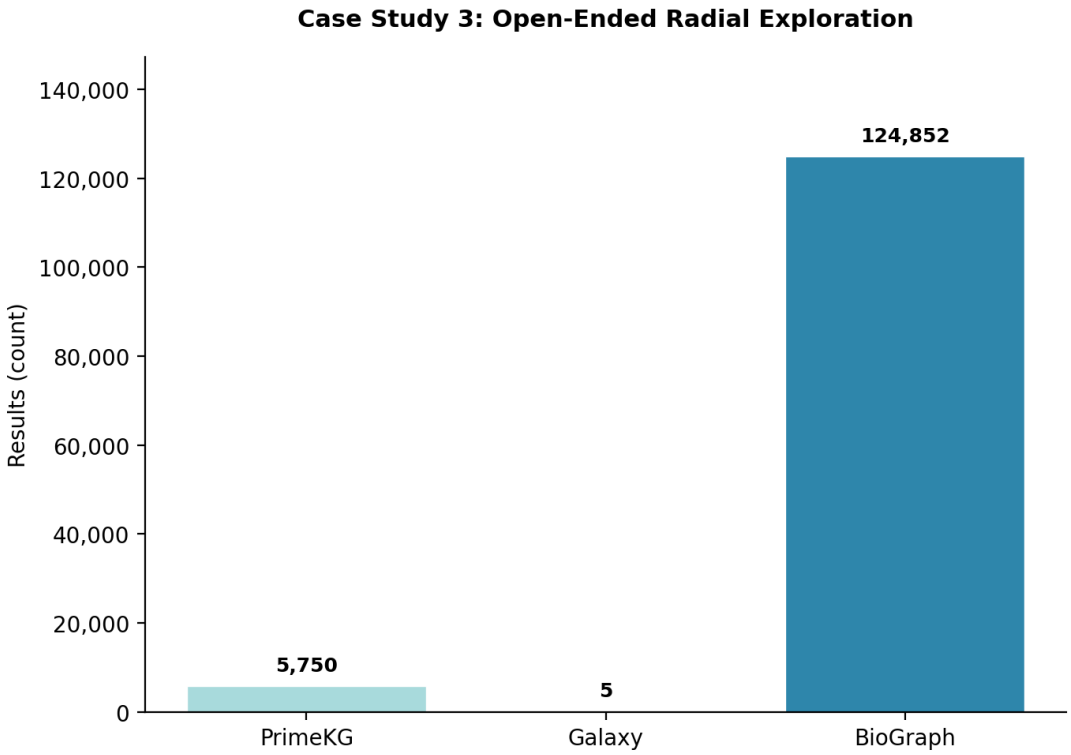


Figure 3.3: Case Study 3 tool comparison: PrimeKG multi-type relationship discovery (5,750 unique pairs, 2.5 seconds, with ID harmonization), Galaxy tool discovery (5 results), and BioGraph open-ended exploration (124,852 pairs across 6 entity types in 481 seconds). PrimeKG is 193× faster but single-hop; BioGraph discovers 21.7× more relationships through automatic multi-hop traversal.

deduplicated across source genes, reducing redundant API calls by 55%. Second, BioGraph scales from focused explorations (28,161 pairs in CS1) to comprehensive radial searches (124,852 pairs in CS3) without performance degradation, with peak RAM proportional to query scope rather than showing exponential growth. Finally, automatic relationship discovery through type introspection enables traversal patterns that manual orchestration would require encoding explicitly—the meta-programming capability that distinguishes BioGraph from a simple tool aggregator.

3.1.6. Summary of framework analysis

The three case studies together show the intended progression of the thesis: targeted enrichment, directed traversal, and open-ended exploration. The important comparison is internal to BioGraph, not against a large catalog of third-party tools. Figure 3.4 shows an aggregate visualization of the tools explored across the case studies.

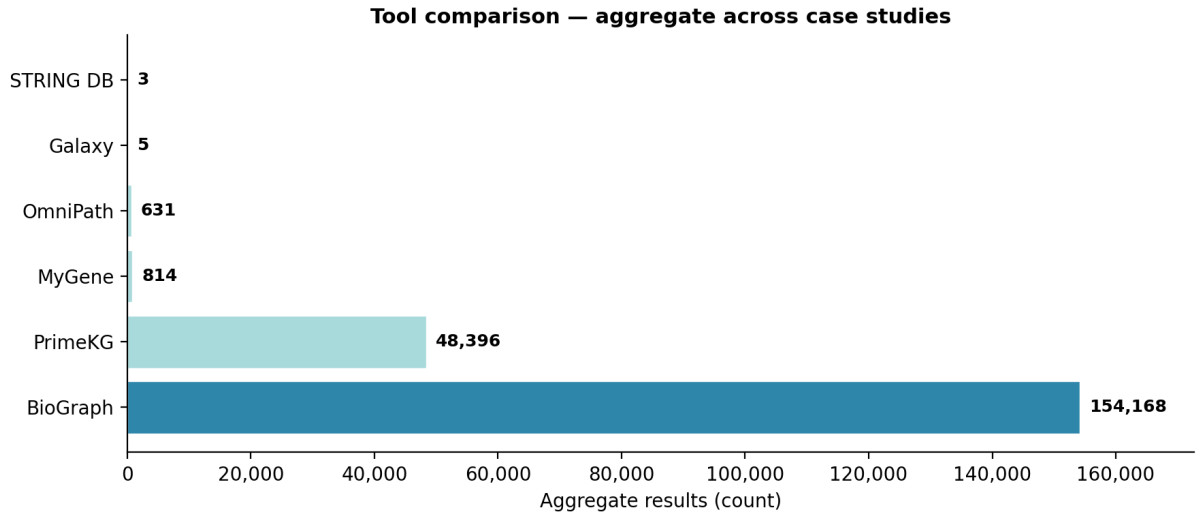


Figure 3.4: Aggregate tool comparison across all three case studies: 154,168 total entity pairs discovered by BioGraph (28,161 + 1,155 + 124,852). Open-ended exploration (`explore()`) dominates in discovery scope, reflecting the fundamental trade-off between query speed and relationship coverage.

4 | Conclusion

4.1. Limitations and Future Work

The current implementation demonstrates core capabilities but identifies several areas for future enhancement. Maximum exploration depth is currently limited to 3 hops by default, which may be insufficient for some distant relationship discovery tasks. Circular reference detection prevents some valid multi-hop queries, requiring future refinement of the cycle detection heuristics. The implementation does not yet include a result caching layer, meaning every exploration generates fresh API calls, which could be optimized through intelligent caching strategies. Additionally, the entire result set must be loaded in memory before returning to the user, precluding streaming output for extremely large explorations. Finally, entity type relationships are currently statically configured, which limits dynamic adaptation to evolving database schemas.

4.1.1. Extensibility and Zero-Configuration Extensions

A primary future direction is BioGraph's zero-configuration extensibility for new entity types. Adding a new entity type (e.g., Drug, Mutation, Disease, Variant) would require only: (1) defining the entity class with type hints, (2) registering its corresponding API client, and (3) updating YAML configuration files. Critically, zero changes to BioGraph's core components (traversal engine, path finder, relationship registry) would be necessary—all relationships would be discovered automatically via Python reflection and type introspection. This extensibility design embodies the Open-Closed Principle [13]: the system is open for extension (new entity types) but closed for modification (BioGraph core unchanged). Future work should include concrete implementations adding new entity types (Section 2.2.1) to validate this extensibility claim empirically.

4.2. The advantages of our approach

Zero-configuration entity integration. The main practical advantage of BioGraph is that new entity classes can be introduced with minimal coordination effort. Once the entity type is declared and linked to an API client, BioGraph can discover the corresponding relationships through its registry and type-introspection mechanisms. This reduces the need for bespoke traversal code and makes the system easier to extend in a research setting where data sources evolve over time.

Automatic path discovery. Automatic path discovery is the mechanism that turns BioGraph from a collection of data clients into a reusable research tool. Rather than forcing the user to specify every intermediate hop, BioGraph infers valid relationships and selects a traversal path based on the requested source and target types. This is what allows the same system to support direct enrichment, directed traversal, and exploratory discovery without separate pipelines.

Reflection-based universality. Reflection-based universality is important because it keeps the implementation aligned with the structure of the domain model. The same metadata used to describe entities is also used to decide how they should be traversed and combined. As a result, BioGraph stays coherent as the number of supported entities grows, while still preserving a consistent programming model for the user.

Comparison with legacy systems. Legacy systems remain useful when the task is narrow and already well defined, but they usually require manual orchestration or precomputed relationships. BioGraph is better suited to discovery-oriented work because it can combine multiple sources dynamically and follow the relationships implied by the model itself. In that sense, the comparison is not about replacing every specialized tool, but about offering a higher-level layer for cross-source biological exploration when flexibility matters more than single-hop speed.

4.3. Potential uses

BioGraph is relevant wherever researchers need to move from a starting entity to a wider biological context without handcrafting every intermediate step. Typical uses include gene prioritization, pathway discovery, exploratory protein network analysis, and rapid prototyping of cross-database analysis workflows. Because the same architecture can support both targeted and open-ended queries, it can serve as a foundation for more specialized downstream pipelines as well.

Bibliography

[1] M. Ashburner, C. A. Ball, J. A. Blake, D. Botstein, H. Butler, J. M. Cherry, A. P. Davis, K. Dolinski, S. S. Dwight, J. T. Eppig, et al. Gene ontology: tool for the unification of biology. *Nature genetics*, 25(1):25–29, 2000.

[2] P. Chandak, K. Huang, and M. Zitnik. Building a knowledge graph to enable precision medicine. *Scientific Data*, 10(1):67, 2023.

[3] L. Ehrlinger and W. Wös. Towards a definition of knowledge graphs. *SEMANTiCS*, 48:1–4, 2016.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1994.

[5] Gene Ontology Consortium. The Gene Ontology knowledgebase in 2024. *Nucleic acids research*, 52(D1):D744–D751, 2024.

[6] M. E. Gillespie, B. Jassal, R. Stephan, M. Milacic, K. Rothfels, A. Senff-Ribeiro, J. Griss, C. Sevilla, L. Matthews, C. Gong, et al. Reactome database release 89. *Nucleic acids research*, 52(D1):D686–D692, 2024.

[7] J. Goecks, A. Nekrutenko, and J. Taylor. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome biology*, 11(8):R86, 2010.

[8] A. Hamosh, A. F. Scott, J. S. Amberger, C. A. Bocchini, and V. A. McKusick. Online Mendelian inheritance in man (OMIM), a knowledgebase of human genes and genetic disorders. *Nucleic acids research*, 33(suppl_1):D514–D517, 2005.

[9] A. Hogan, E. Blomqvist, M. Cochez, C. d'Amato, G. d. Melo, C. Gutierrez, S. Kirrane, R. Navigli, S. Neumaier, A.-C. N. Ngomo, et al. Knowledge graphs. *ACM Computing Surveys (CSUR)*, 54(4):1–37, 2021.

[10] K. L. Howe, V. Achter, J. Allen, N. Al-Rasheid, J. Alvarez-Jarreta, M. Barba, M. Barlow, P. Bevan, R. Bloomer, S. Boppana, et al. Ensembl 2024. *Nucleic acids research*, 52(D1):D891–D899, 2024.

[11] M. Kanehisa, M. Furumichi, M. Tanabe, Y. Sato, and K. Morishima. KEGG: integrating viruses and cellular organisms. *Nucleic acids research*, 51(D1):D545–D551, 2023.

[12] M. Lenzerini. Data integration: A theoretical perspective. *PODS*, 233:233–246, 2002.

[13] R. C. Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2008.

[14] NCBI. Gene: a resource for gene information. *Nucleic acids research*, 52(D1):D26–D32, 2024.

[15] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, et al. Taverna: lessons in creating a workflow system for the life sciences. *Concurrency and Computation: Practice and Experience*, 18(10):1067–1100, 2006.

[16] Python Packaging Authority. *Creating and discovering plugins*. Python Packaging Authority, 2025. URL `https://packaging.python.org/en/latest/guides/creating-and-discovering-plugins/`. Accessed: 2025-12-02.

[17] Python Software Foundation. *pkgutil – Package Extension Utility*. Python Software Foundation, 2025. URL `https://docs.python.org/3/library/pkgutil.html`. Python 3 Standard Library Documentation.

[18] Python Software Foundation. *typing – Support for type hints*. Python Software Foundation, 2025. URL `https://docs.python.org/3/library/typing.html`.

[19] S.-A. Sansone, P. Rocca-Serra, D. Field, E. Maguire, C. Hoffman, A. Johannsen, H.-P. Klenk, L. Hirschman, Y. Liu, J. Miller, et al. Toward interoperable bioscience data. *Nature genetics*, 44(2):121–126, 2012.

[20] A. Singhal. Introducing the knowledge graph: things, not strings. In *Google Official Blog*, 2012.

[21] UniProt Consortium. UniProt: the universal protein knowledgebase in 2024. *Nucleic acids research*, 52(D1):D523–D530, 2024.

[22] M. D. Wilkinson, M. Dumontier, I. J. Aalbersberg, G. Appleton, M. Axton, A. Baak, N. Blomberg, J.-W. Boiten, L. B. da Silva Santos, P. E. Bourne, et al. The FAIR guiding principles for scientific data management and stewardship. *Scientific data*, 3 (1):160018, 2016.

[23] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-

Reyes, I. Dunlop, A. Nenadic, P. Fisher, et al. Taverna workflow repository and community. *Journal of biological databases and curation*, 2013, 2013.

[24] J. Xin, A. Mark, C. Afrasiabi, G. Tsueng, M. Juchler, B. Demchak, and P. Ping. MyGene.info: an intuitive API for gene annotation. *PLoS ONE*, 11(8):e0160314, 2016.

Acknowledgements

I would like to thank my thesis supervisor and the faculty at Politecnico di Milano for their guidance throughout this work. I am grateful to the maintainers of the open biological databases—MyGene, UniProt, KEGG, Gene Ontology, STRING, ChEMBL, and PubChem—whose publicly available APIs made BioGraph possible. Finally, I thank my family and friends for their continued support.