POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Executive Summary of the Thesis

BioGraph: a Network Approach to Multi-Omics Information Retrieval

Laurea Magistrale in Computer Engineering - Ingegneria Informatica

Author: Francesco Mario Inzerillo

Advisor: Prof. Marco Domenico Santambrogio

Co-advisor: Leonardo De Grandis

Academic year: 2025-2026

1. Introduction

Biomedical research has undergone a radical transformation in the past two decades. The explosion of sequencing technologies and high-throughput experiments has produced an unprecedented volume of genomic and proteomic data, enabling new insights into disease mechanisms, drug targets, and precision medicine. Yet this abundance of data brings a critical challenge: the same biological information is scattered across dozens of independently-curated databases, each using proprietary identifier systems, distinct data formats, and separate update schedules. A researcher studying a single gene 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.

The problem is not merely one of data volume. Three structural barriers make automated reconciliation genuinely hard. The first is **identifier heterogeneity**: the gene BRCA1, for instance, carries at least five different identifiers across major databases (Entrez ID 672, Ensembl ID ENSG00000012048, HGNC ID 1101, RefSeq NM_007294, and UniProt P38398), and these systems update on different schedules—Ensembl quarterly, NCBI daily—making it difficult to keep cross-database mappings consistent. The second is **coordinate shifts**: reference genome builds change chromosomal positions. BRCA1 maps to chr17:43,044,295–43,125,364 in GRCh38 but to different coordinates in GRCh37, invalidating analyses built on earlier builds. The third is **data completeness gaps**: no single database is authoritative for all information types. KEGG focuses only on well-studied pathways, Gene Ontology annotations are sparse for less characterised genes, and UniProt's manually reviewed Swiss-Prot set covers only ∼570k proteins out of millions in the broader TrEMBL set.

Contemporary bioinformatics relies on automated approaches to bridge these gaps, yet existing solutions remain incomplete. Workflow platforms such as Galaxy [2] and Taverna [4] support reproducible multi-step analysis pipelines but require explicit authoring for every query pattern: adding a new data source means creating new workflow components and hardcoding the routing logic that connects them. Unified API aggregators such as MyGene [5] reduce the burden of querying individual sources, but they address the data access problem rather than the cross-

entity traversal problem. Static pre-computed knowledge graphs, most notably PrimeKG [1], integrate up to 20 biomedical resources and offer fast local queries, but their fixed schemas and pre-built indices cannot incorporate new data sources without a full rebuild, and they operate at traversal depth 1.

The core gap is a framework that combines three properties none of the existing tools provide simultaneously: automatic discovery of valid query paths without manual configuration; execution of arbitrary multi-hop queries across live heterogeneous APIs; and the ability to incorporate new entity types and data sources with zero changes to framework code. Table 1 summarises this positioning.

Table 1: Positioning of BioGraph relative to existing tools

Property	Galaxy	MyGene	PrimeKG	**BioGraph**
Multi-hop queries	✓	–	–	✓
Live data	✓	✓	–	✓
Auto path discovery	–	–	–	✓
Zero-config extension	–	–	–	✓

This work presents **BioGraph**, a Python library that uses reflection and type introspection to automatically discover and traverse relationships between biological entities at runtime, without requiring any configuration changes when new entity types or data sources are added.

2. Knowledge Graphs as a Biological Data Model

A key insight motivating BioGraph's design is that biological knowledge is fundamentally relational: the significance of a gene emerges not from its sequence alone but from which proteins it encodes, which pathways those proteins participate in, and which other genes share those pathways. Individual data points carry little meaning in isolation.

This relational nature makes graphs a natural representation. A knowledge graph $G = (V, E)$ partitions its vertices into multiple types $V = V_1 \cup \ldots \cup V_n$—genes, proteins, pathways, GO terms—and defines a schema R of valid typed edges between them, for example Gene→CODES_FOR→Protein. Two properties make this structure especially powerful. First, *multi-centrality*: any entity can serve as the starting point for a query, with no fixed root. A researcher can begin from a gene, a pathway, or a compound with equal ease. Second, *relationship focus*: graph algorithms operate on connections rather than attributes alone, enabling queries about reachability, neighbourhood membership, and path structure—"which GO terms are reachable from BRCA1 within two hops through its associated pathways?"—that are cumbersome or impossible to express in a flat database model.

The construction challenge is how to build the graph schema without hardcoding every possible relationship. BioGraph's solution is to derive the schema automatically from the Python type system: a `Gene` class that declares an attribute `pathways: List[Pathway]` is simultaneously a domain model definition and an edge type declaration in the knowledge graph. The data model and the graph schema are the same artefact; extending one automatically extends the other.

3. BioGraph: Design and Implementation

3.1. Reflection-based relationship discovery

At startup, BioGraph's *Relationship Registry* calls `typing.get_type_hints()` on every registered entity class and inspects each attribute's type annotation. An attribute typed as `List[Pathway]` or `Optional[Protein]` declares a directed relationship from the enclosing class to the target class, along with the cardinality (one-to-many or one-to-one) and the field name through which the relationship can be traversed directly. These inferred relationships are stored in an adjacency graph that the PathFinder queries at execution time.

When a developer adds a new entity class—say `Drug(targets: List[Protein])`—BioGraph discovers the Drug→Protein relationship automatically and immediately enables all query paths that pass through it, including Gene→Protein→Drug and any deeper traversal. No changes to the traversal engine, path finder, client registry, or any other framework component are required. This design embodies the Open-Closed Principle [3]: open for extension by adding new classes, closed for modification of

existing framework code.

3.2. Architecture

Figure 1 shows the five-layer architecture. The *Application Layer* is the user-facing API. The *Orchestration Layer* handles pipeline initialisation, configuration loading, and coordination between components. The *Meta-Framework Layer* contains the three core algorithmic components: the Relationship Registry (schema discovery), the PathFinder (BFS over the entity graph), and the Universal Traversal Engine (hop execution). The *Data Model Layer* defines the seven entity types currently implemented—Gene, Protein, Pathway, GO term, Compound, PPI, and CPI—each paired with a typed ID class (`GeneID`, `ProteinID`, etc.) that carries all available identifiers separately from the entity's data attributes. This separation ensures that identifier resolution does not interfere with data-level filtering and conversion. The *Data Access Layer* encapsulates one API client per data source (MyGene, KEGG, UniProt, STRING, ChEMBL, PubChem, Gene Ontology), each self-registering at import time via the `@register_db_source` decorator and auto-discovered at startup through `pkgutil.iter_modules()`.

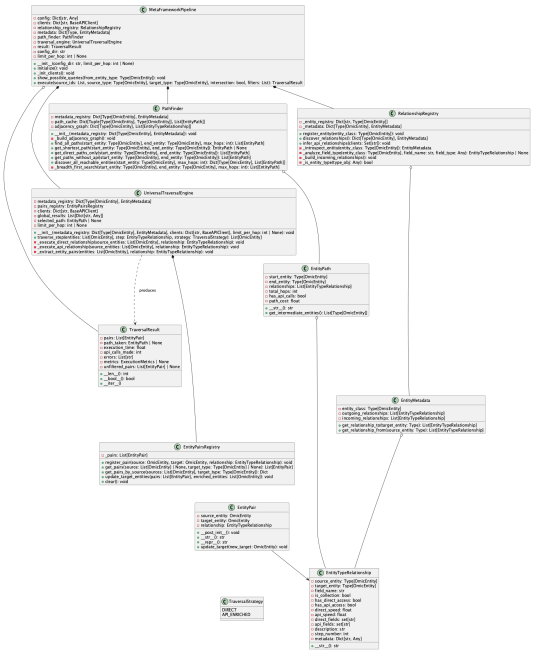

Figure 1: BioGraph class diagram showing the five-layer architecture.

3.3. Path discovery and traversal strategies

Given source and target entity types, the *PathFinder* applies BFS over the relationship adjacency graph and returns the lowest-cost path as an ordered list of `EntityTypeRelationship` objects. Cost is assigned by relationship type: direct-field relationships (zero API calls) cost less than API-mediated ones, so BFS naturally prefers paths that reuse already-fetched data. The Universal Traversal Engine executes each hop using one of two strategies selected per relationship:

- **Direct-field**: reads an already-populated attribute on source entities (e.g. `gene.go_terms` populated by MyGene at fetch time). Zero additional API calls; richness limited to what the source client provides.
- **API-mediated**: invokes the target entity's registered client for a richer, complete response. Falls back gracefully to direct-field if the API call fails, ensuring the pipeline continues rather than halting.

A critical optimisation precedes every API-mediated hop. *Intelligent deduplication* collects and deduplicates all target entity IDs from all source pairs before issuing any request, then submits a single batch query for the merged list. The results are assigned back to all relevant pairs. If Gene A and Gene B both encode Protein X, the API is called once for Protein X, not twice. This reduces redundant requests proportionally to how much source entities share downstream neighbours—a property that becomes increasingly valuable as the number of source entities grows.

3.4. Query execution modes

BioGraph exposes three query modes that form a strict reuse hierarchy, each level implemented on top of the one below.

`enrich(ids, type)` is the base level. It selects the registered client for the given entity type and issues exactly one batch API call, returning a list of fully populated entity objects. It serves as the foundation for all higher-level modes and is useful on its own when only single-source annotation is needed.

`execute(ids, source, target)` is the directed traversal level. It calls `enrich()` internally to retrieve source entities, then invokes PathFinder to discover the shortest path to the target type, then traverses it hop by hop using the Traversal Engine. It returns a list of `EntityPair` objects each capturing a (source, target, relationship) triple.

`explore(ids, type, max_depth)` is the open-ended discovery level. It iterates `execute()` at each depth level, maintaining a frontier of newly discovered entities and a set of already-explored entity IDs to prevent cycles. No target type needs to be specified. The framework discovers all reachable entity relationships up to the given depth automatically, making it suitable for hypothesis generation and biological network characterisation where the query space is not known in advance.

Every call returns a `TraversalResult` that bundles the discovered entity pairs, wall-clock execution time, total API call count, and per-relationship timing and count statistics, enabling post-hoc performance analysis without additional instrumentation.

4. Experimental Validation

Three case studies validated BioGraph's correctness, efficiency, and scalability. All experiments ran with a unified `MetricsCollector` integrated into the pipeline execution engine, tracking API calls, execution time, and RAM usage at both operation and per-call granularity.

4.1. Case Study 1: Gene-to-pathway exploration at scale

The objective was to evaluate `explore()` with entity-type filtering on a realistic gene expression dataset and validate that focused queries remain efficient at scale. We loaded 1,000 gene symbols from a gene-expression spreadsheet and executed `explore()` with `allowed_entity_types=[Gene, Pathway]` at depth 2. This configuration traverses through intermediate entity types during path discovery but restricts the result set to Gene and Pathway entities, enabling focused gene–pathway analysis without sacrificing the multi-hop traversal capability.

BioGraph discovered 28,161 gene–pathway pairs using only 2 API calls. Peak RAM was 1,469 MB and remained stable across both depths, demonstrating that the deduplication strategy prevents exponential growth in intermediate state. By comparison, a MyGene batch query over the same 1,000 genes returns 844 gene annotations in approximately 17 seconds but discovers no cross-entity relationships whatsoever.

Table 2: CS1: BioGraph performance

Stage	Time (s)	API	Pairs
Depth 0 (G→Pathway)	75.1	1	666
Depth 1 (P→Gene)	78.5	1	27,495
Total	**153.6**	**2**	**28,161**

4.2. Case Study 2: Multi-hop gene-to-PPI traversal

The objective was to evaluate `execute()` on a three-hop Gene→Protein→PPI traversal and quantify the efficiency gains from intelligent batching. We queried BRCA1, TP53, and CDKN1A individually and then combined, using MyGene for gene retrieval, UniProt for protein enrichment, and STRING for protein–protein interaction data.

Table 3: CS2: individual vs. combined query

Query	Time (ms)	API	Pairs
BRCA1	9,659	3	459
TP53	9,292	3	590
CDKN1A	5,648	3	106
Sum (individual)	24,599	9	1,155
Combined	**14,544**	**3**	**1,155**

Combining the three genes into a single query reduced API calls by **67%** and execution time by **41%** while returning identical results. This result arises directly from the deduplication optimisation: BRCA1, TP53, and CDKN1A share many protein interaction partners in the STRING network, so the combined query resolves protein IDs once and reuses the result for all three genes rather than issuing redundant requests.

4.3. Case Study 3: Open-ended radial exploration

The objective was to demonstrate fully automatic relationship discovery with no target-type spec-

ification, and to evaluate the efficiency of the combined query strategy at depth 3. We ran `explore()` on the same three cancer genes with no entity-type restrictions and no per-hop result limits.

Table 4: CS3: relationship breakdown, combined run

Relationship	Pairs	API calls
Gene→GO Terms	170,967	471
Gene→Pathway	49,511	3
Pathway→Gene	15,326	3
Gene→Protein	5,152	3
Protein→PPI	300	3
PPI→Protein	300	0
Total	**124,852**	**216**

BioGraph discovered 124,852 entity relationships across six relationship types automatically. No query paths were specified in advance; the framework inferred all valid traversals from the data model. The combined three-gene query reduced API calls by **55%** and execution time by **55.6%** compared to running each gene individually (496 calls, 1,084 s), again due to shared downstream entities. TP53 produced the largest sub-network (114,236 pairs, 47% of the total), consistent with its role as a central transcription factor with extensive regulatory reach across hundreds of biological processes. Gene→GO annotations dominated (71% of all pairs, 97% of API calls), reflecting the depth of functional annotation coverage in public databases compared to the relative sparsity of curated protein–protein interaction data.

A direct evaluation of PrimeKG with MyGene-based ID harmonisation (BRCA1→672, TP53→7157, CDKN1A→1026) returned 5,750 unique relationships in 2.5 seconds—193× faster than BioGraph, which discovered 21.7× more relationships. PrimeKG queries a fixed pre-built index at depth 1; BioGraph performs live multi-hop traversal at depth 3, including Gene→GO annotations (170,967 pairs) that are entirely absent from PrimeKG's schema. The trade-off is architectural rather than a quality difference: PrimeKG is the appropriate choice when the query structure is known and latency is critical; BioGraph is the appropriate choice for open-ended discovery where the valid query space itself is not known in advance.

5. Discussion

Batching efficiency scales with shared neighbours. The deduplication optimisation yields reductions proportional to the overlap between source entities' downstream neighbourhoods. In Case Study 2 the three cancer genes share many STRING interaction partners, giving 67% fewer API calls. In Case Study 3 the effect compounds across depths as shared pathways and GO term ancestors accumulate, sustaining 55% reductions even at depth 3. For source sets with no shared neighbours the savings would approach zero; for highly connected hub genes in co-expression networks the savings would be substantially larger.

Memory is predictable and proportional to result size. Peak RAM was 1,469 MB in Case Study 1 (28,161 pairs) and 213–224 MB in Case Study 2 (1,155 pairs at depth 2), proportional to discovered pair count and stable across depths. This predictability follows from the deduplication strategy, which prevents exponential intermediate state growth. The current limitation—full in-memory loading—means that very large explorations require sufficient available RAM. Streaming result output, where pairs are emitted and consumed incrementally rather than accumulated, is the natural remedy and a planned future addition.

The depth–latency trade-off is tunable. Case Study 2 (`execute()`, depth 2, 14.5 s) and Case Study 3 (`explore()`, depth 3, 481 s) bound the operating range. The `allowed_entity_types` filter demonstrated in Case Study 1 provides a middle ground: open-ended traversal through the full entity graph with controlled result scope. Researchers with a known target entity type should use `execute()` for efficiency; researchers performing hypothesis generation or phenotype-driven network characterisation benefit from `explore()` despite the higher latency.

6. Conclusions

BioGraph demonstrates that Python's type annotation system provides sufficient structural information to automate the construction and multi-hop traversal of a biological knowledge graph over live, heterogeneous data sources. Three novel properties distinguish it from existing tools:

zero-configuration extensibility (new entity types discovered automatically from class declarations), automatic path discovery (users specify source and target types; the framework infers valid traversal paths via BFS), and efficient batching (entity deduplication before API calls reduces redundant requests proportionally to shared downstream neighbours).

Experiments confirm 41–67% API call reductions from batching, 55% efficiency gains from combined multi-gene exploration, and the ability to discover 124,852 multi-type biological relationships across six entity categories from three cancer genes without any hardcoded query logic. Comparison with PrimeKG clarifies the positioning: BioGraph trades latency for live data access, greater traversal depth, and a dynamically extensible schema.

Future directions include adding Drug, Variant, and Disease entity types to empirically validate the extensibility claim; introducing a result caching layer for repeated queries over stable databases; and implementing streaming result output to remove the current in-memory limitation for very large explorations.

Acknowledgements

The author thanks Prof. Marco Domenico Santambrogio and Leonardo De Grandis for their guidance, and the maintainers of the public biological databases—MyGene, UniProt, KEGG, Gene Ontology, STRING, ChEMBL, and PubChem—whose open APIs made BioGraph possible.

7. Bibliography

The Executive Summary should contain the very essential bibliography of your study. It is suggested to use the BibTeX package [**?**] and save the bibliographic references in the file `bibliography.bib`.

8. Conclusions

A final section containing the main conclusions of your research/study have to be inserted here.

9. Acknowledgements

Here you might want to acknowledge someone.

References

[1] Payal Chandak, Kexin Huang, and Marinka Zitnik. Building a knowledge graph to enable precision medicine. *Scientific Data*, 10(1):67, 2023.

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

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

[4] Tom Oinn, Mark Addis, Justin Ferris, David Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R Pocock, Anil 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.

[5] Jianchao Xin, Adriano Mark, Cyrus Afrasiabi, Ginger Tsueng, Michael Juchler, Barry Demchak, and Peng Ping. MyGene.info: an intuitive API for gene annotation. *PLoS ONE*, 11(8):e0160314, 2016.