УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Александар Н. Вељковић

СЕМАНТИЧКО ОБЈЕДИЊАВАЊЕ И ПРЕТРАЖИВАЊЕ БИОИНФОРМАТИЧКИХ БАЗА ПОДАТАКА КОРИШЋЕЊЕМ МЕТОДА ИСТРАЖИВАЊА ПОДАТАКА

Докторска дисертација

Београд, 2023

UNIVERSITY OF BELGRADE
FACULTY OF MATHEMATICS

Aleksandar N. Veljković

SEMANTIC UNIFICATION AND SEARCHING OF BIOINFORMATICS DATABASES USING DATA MINING METHODS

Doctoral dissertation

Belgrade, 2023

Подаци о ментору и члановима комисије

Ментор

проф. др Ненад Митић, редовни професор, Математички факултет, Универзитет у Београду

Чланови комисије

проф. др Саша Малков, ванредни професор, Математички факултет, Универзитет у Београду

др Јована Ковачевић, доцент, Математички факултет, Универзитет у Београду

др Александар Картељ, ванредни професор, Математички факултет, Универзитет у Београду

проф. др Наталија Половић, редовни професор, Хемијски факултет, Универзитет у Београду

проф. др Јурij Орлов, професор Руске академије наука, Институт за дигитално здравље, Први московски државни медицински универзитет И.М.Сеченов, Москва, Русија

Датум одбране:

Advisor and Committee members

Advisor:

dr Nenad Mitić, full professor, University of Belgrade, Faculty of Mathematics

Committee:

dr Saša Malkov, associate professor, University of Belgrade, Faculty of Mathematics

dr Jovana Kovačević, assistant professor, University of Belgrade, Faculty of Mathematics

dr Aleksandar Kartelj, associate professor, University of Belgrade, Faculty of Mathematics

dr Natalija Polović, full professor, University of Belgrade, Faculty of Chemistry

dr Yuriy Orlov, PhD, DrSci, Professor of the Russian Academy of Sciences, The Digital Health Institute, I.M. Sechenov First Moscow State Medical University of the Ministry of Health of the Russian Federation, Moscow, Russia

Date of the defense:

Подаци о докторској дисертацији

Наслов дисертације: Семантичко обједињавање и претраживање биоинформатичких база података коришћењем метода истраживања података

Резиме: Биоинформатика као наука будућности суочава се са проблемима обраде велике количине података која се свакодневно увећава. Поред проблема складиштења података, изазов представља и анализа података и разумевање скривених веза између биолошких појмова које се уочавају тек након обједињавања података из различитих извора. Ова дисертација дефинише нови модел података за обједињавање хетерогених података из различитих биоинформатичких база података и дизајн архитектуре система за имплементацију софтверског система заснованог на предложеном моделу података. Додатно, дисертација дефинише аутоматизовани протокол за откривање нових веза семантичке сличности заснованих на методама истраживања података коришћењем података добијених из дефинисаног модела. Модел података, софтверска архитектура и аутоматизовани протокол су тестирани над подацима из пет биоинформатичких база података. Резултати показују високу флексибилност модела и високу ефикасност софтверског система имплементираног према описаном дизајну архитектуре.

Кључне речи: семантика, обједињавање биоинформатичких база, кластеровање, правила придруживања, граф знања

Научна област: Рачунарство

Ужа научна област: Истраживање података, Биоинформатика

УДК број:

Dissertation Data

Dissertation title: Semantic unification and searching of bioinformatics databases using data mining methods

Abstract: Bioinformatics as a science of the future faces the problems of processing a large amount of data that is increasing every day. In addition to the problem of data storage, the challenge is also data analysis and the understanding of hidden relations between biological entities that are observed only after unifying data from different data sources. This thesis proposes a novel data model for the unification of heterogeneous data from multiple bioinformatics databases and a system architecture design for implementing software systems based on the proposed data model. Additionally, the thesis defines an automated pipeline for discovering new semantic similarity relations based on data mining methods using the data found in the proposed data model. The data model, software architecture, and automatic pipeline are evaluated using data from five real-world bioinformatics databases. The results demonstrate a high flexibility of the data model and the high efficiency of the software system implemented following the proposed architecture design.

Keywords: semantics, bioinformatics database unification, clustering, association rules, knowledge graph

Scientific field: Computer Science

Scientific discipline: Data mining, Bioinformatics

UDC number:

Contents

1 Introduction **1**
- 1.1 Proximity measures . . . 3
  - 1.1.1 Proximity measures of simple attributes . . . 3
  - 1.1.2 Proximity measures of data objects . . . 4
- 1.2 Semantic similarities . . . 7
- 1.3 Related work . . . 11
- 1.4 Related topics . . . 12

2 Cluster analysis and association rules mining **13**
- 2.1 Cluster analysis . . . 13
  - 2.1.1 Cluster types . . . 14
  - 2.1.2 Properties of clustering methods . . . 14
  - 2.1.3 Categorization of clustering methods . . . 15
  - 2.1.4 Measuring quality of clustering results . . . 20
- 2.2 Association rules mining . . . 23
  - 2.2.1 Apriori algorithm . . . 24
  - 2.2.2 FP-growth algorithm . . . 25

3 Searching bioinformatics databases **27**
- 3.1 Primary and secondary databases . . . 27
- 3.2 Identifiers of bioinformatics data . . . 28
- 3.3 Storing bioinformatics data . . . 29
- 3.4 Accessing and searching bioinformatics data . . . 29

4 New Data Joining Model Proposal **33**
- 4.1 BioGraph data model . . . 34
  - 4.1.1 Entity objects . . . 35
  - 4.1.2 Identifiers . . . 36
  - 4.1.3 Data objects . . . 37
  - 4.1.4 Entity relations . . . 37
  - 4.1.5 Duplicate entries . . . 38
  - 4.1.6 Data updates . . . 38
  - 4.1.7 Mapping BioGraph model to graph and relational database . . . 38

4.1.8 Efficient indexing . . . 40
4.2 Generalized method for deriving semantic relations . . . 40
4.2.1 Selecting a subset of relations . . . 42
4.2.2 Generating relation matrix . . . 42
4.2.3 Deriving semantic similarity relations . . . 43
4.2.4 Automated method for deriving semantic similarity relations . 44

5 Model implementation and validation 51
5.1 Software architecture . . . 51
5.1.1 Data importers . . . 51
5.1.2 Core service . . . 52
5.1.3 Indexers . . . 53
5.1.4 Database adapters . . . 53
5.1.5 HTTP REST API . . . 54
5.1.6 Internal query language . . . 54
5.1.7 Data flows . . . 54
5.2 Material . . . 56
5.2.1 DisProt dataset . . . 57
5.2.2 HGNC dataset . . . 58
5.2.3 IEDB dataset . . . 58
5.2.4 Tantigen 2.0 dataset . . . 59
5.2.5 DisGeNET dataset . . . 59
5.3 Deriving new semantic similarity relations in BioGraph data model . 60
5.4 User interface . . . 61

6 Results and discussion 65
6.1 Comparison with the existing data unification and querying systems . 66
6.2 Advantages and disadvantages . . . 68
6.3 Examples of biomedical applications . . . 68
6.3.1 Genes related to Parkinson's disease . . . 68
6.3.2 Genes related to pancreatic cancer . . . 73

7 Conclusion 77

Bibliography 85

A Database importers 87
A.1 DisProt data importer . . . 87
A.2 HGNC data importer . . . 90
A.3 DisGeNET data importer . . . 96
A.4 IEDB data importer . . . 98
A.5 DisGeNET data importer . . . 105

List of Figures

4.1 A schema of the BioGraph model 35
4.2 Network of objects from five different sources 36
4.3 Diagram of the BioGraph data model mapped to a relational model. . 39
4.4 General pipeline for automated deriving of new semantic relations. . . 45
4.5 Relation deriving substeps of the pipeline for automated deriving of new semantic relations based on clustering method. 45
4.6 Relation deriving substeps of the pipeline for automated deriving of new semantic relations based on association rules mining. 46
4.7 Example of selecting relations as inputs for deriving similarity relations. 47
4.8 Example of constructed data matrix based on the selected relations and their weights. 48
4.9 Parallel overview of clusters (a), hyperedges (b), and cluster entity objects (c). 48

5.1 Example of an internal BioGraph query in JSON format. The query fetches all genes and related proteins where the protein disorder content is 0.9 or higher. 55
5.2 Diagram representing the architecture of the system which implements BioGraph data model. 56
5.3 Diagram representing metadata from DisGeNET dataset record mapped to BioGraph model. 57
5.4 Diagram representing spectral clustering substeps for relation deriving. 60
5.5 Example of a graphical query drawn using BioGraph Web interface . 62
5.6 List of results received when querying the genes which are transcribed into highly disordered proteins 63
5.7 Details of a single gene entity, displaying information collected from multiple data sources. 64
5.8 Searching diseases using keyword “pneumonia”. 64

6.1 Blank canvas of the BioGraph Web interface. 69
6.2 Steps required for the user to create the disease entity node. 70
6.3 Example of the process of associating the disease node to a specific disease. 70

6.4 Steps required for the user to create the gene entity node. 71
6.5 Creating relation between gene and disease nodes. 72
6.6 Query for extracting all genes highly related to Parkinson's disease. . 73
6.7 Results of the query for extracting all genes highly related to Parkinson's disease and details of the PLA2G6 gene. 73
6.8 Query for extracting all genes highly related to pancreatic cancer . . 74
6.9 Results of the query for extracting all genes highly related to pancreatic cancer 75

List of Tables

4.1 Basic types of the relations between entity objects in BioGraph data model. 41

6.1 Comparison between the BioGraph system and currently existing solutions 67
6.2 Genes found to be related to pancreatic cancer 74

Chapter 1

Introduction

In order to obtain a general understanding of an object or phenomenon, it is important to observe it from multiple angles and put it in a wider context. Biological data comes from a wide variety of experiments and analyses, adding context to potentially overlapping sets of biological entities. Crystallography experiments provide insight into the structure of proteins, while functional analysis of the same proteins may give a deeper understanding of their roles and detect interaction networks among them. Additional research may further extend the knowledge of proteins by noting their presence and activity in different conditions and diseases. The holistic view, achieved by observing multiple pieces of information, is the only way to understand complex biological concepts and interactions among the entities. Although many of the analyses add bits of information to the same subsets of entities, the data that they produce is structured following different data formats and stored in different data silos. Often the only link connecting information on the same entity from different datasets is the common identifier of the entity. Unique identifiers, such as gene symbols or sequence accession numbers, are assigned to biological entities. These identifiers can be shared among multiple datasets, but some identifiers are strictly local and are not recognized in other datasets, which makes it challenging to link similar or identical data in different datasets.

Although identifiers play an important role in connecting the data from different datasets, they represent only one part of the solution. Entities with different identifiers, which are not explicitly connected, can have multiple common properties, making them similar in a certain context. Such values can even be seen through unstructured text descriptions of the entities, where the similarity is observed through the similarity of texts written using natural language. Another complex case of similarity is the similarity that can be seen only by analyzing similar patterns of behavior or interaction between groups of entities. Detecting such similarities helps to recognize patterns in biological processes but also enables a powerful semantic search that goes beyond the identifier-based lexical search and overcomes the problem of unmatched identifiers.

A sentence is an array of characters until the semantics give meaning to it. Arrays of characters become words and the sentence becomes an array of words. Words have their individual meaning, which can be found in a dictionary, but a different ordering of words and their forms give a different meaning to the sentence. Semantics is the science of meaning and it is an important part of data search. A keyword search, based on matching the exact keywords to a range of documents, may prove useful, but exact keyword matching is not always applicable. Processing a query "protein similar to P53"[50] would require a deeper understanding of the concept of similarity. Two genes may have similar roles, can be involved in similar biological processes, or even be linked to similar diseases. The concept of similarity is difficult to capture using keyword-based search only. Structured queries, like the ones used in relational database systems, provide the ability to establish relations by following the keys between tables. Such queries are well-defined and can be used to find semantically similar data but the cost is paid in the complexity of the query itself. An average user is unable to properly construct the query or know in advance all possible entity types and relations that could be utilized in query construction. The third option is natural language search, which enables writing queries in an unstructured, natural language form. This way of creating semantic queries may be the easiest for the user but adds complexity in parsing the query and introduces potential ambiguity in the intentions of the user, due to ambiguities in natural languages.

A direct approach for finding similarities in data is to apply some data mining algorithms on data specifically preprocessed for the given task. Finding similarities in the data from different datasets may require a different approach to preprocessing or even a different data mining method. The problem with the direct approach is inflexibility, especially when combining data from multiple sources. Designing a unified data model for storing heterogeneous biological data and generalizing the methods for analyzing the similarities between entities on different levels could enable a unified semantic search over biological data from multiple data sources. Such ability would provide deeper insight into biological and biochemical processes and become a powerful tool in biomedical domains, such are gene therapy, drug discovery, immunology, and many more.

There are different approaches to solving the problem of the unification of data and semantic searches. The key data structures for supporting unified data are graphs. The generic structure of a graph, consisting of nodes and edges is useful when the possible set of relation types between data nodes is not known in advance. Adding a new edge of a previously unknown type to a populated graph does not change the underlying structure nor introduce new foreign keys like it is the case with relational databases. Additionally, the graph databases handle traversals by following links in constant time, while relational databases show lower performance when handling join queries [48]. A knowledge based on interconnected data, consisting of entities and links between the entities that represent semantic relations, organized in a graph

structure is called a knowledge graph.

1.1 Proximity measures

Similarity and dissimilarity measures evaluate the degree of association between objects. Both types of measures are commonly referred to as proximity measures. Values of the similarity measures are higher when two objects are more look alike, and lower otherwise. Dissimilarity measures behave in the opposite way, resulting in lower values when the objects are more similar.

1.1.1 Proximity measures of simple attributes

Dissimilarity measures are also known as distance measures, commonly taking values in the interval $[0, +\infty)$. A value of 0 indicates no dissimilarity or equivalence and higher values indicate greater differences between the objects. Similarity measures commonly take values from the interval $[0, 1]$, where a value of 0 indicates no similarity and a value of 1 indicates complete similarity or equivalence. Material listed in this section is based on the descriptions found in [81].

Data types of the compared values highly influence the choice of the similarity and dissimilarity measures. In some cases, it is more intuitive to define a dissimilarity measure and transform it into a similarity measure, or the other way around. Similarity and dissimilarity measures will be presented for simple attributes of nominal, ordinal, and interval types.

Nominal attributes

Nominal attributes can only be compared with equality relation, resulting in a similarity measure s defined as:

$$s(x, y) = \begin{cases} 1, x = y \\ 0, otherwise \end{cases} \tag{1.1}$$

Inversely, a dissimilarity measures d defined for the nominal attributes is defined as:

$$d(x, y) = \begin{cases} 1, x \neq y \\ 0, otherwise \end{cases} \tag{1.2}$$

or

$$(x, y) = 1 - s(x, y). \tag{1.3}$$

Ordinal attributes

Ordinal attributes are discrete attributes with ordinal relations defined between the values. A dissimilarity measure for ordinal data can be defined as:

$$d(x, y) = \frac{|x - y|}{n - 1} \tag{1.4}$$

where n is a number of values of the ordinal attribute and attribute values are mapped to integers 0 to $n - 1$. A similarity measure can be derived from the given dissimilarity measure:

$$s(x, y) = 1 - d(x, y). \tag{1.5}$$

Interval attributes

Interval attributes are continuous attributes and a dissimilarity measure for such attributes can be defined as:

$$d(x, y) = |x - y| \tag{1.6}$$

Values of the defined dissimilarity measure are in the interval $[0, max_{val})$, where max_{val} is the maximum value of the attribute. There are many similarity measures that can be derived from the given dissimilarity measure. Some of those similarity measures that return values from the interval $[0, 1]$ are:

- $s(x, y) = e^{-d(x,y)}$
- $s(x, y) = \frac{1}{1+d(x,y)}$
- $s(x, y) = 1 - \frac{d(x,y)-min_d}{max_d-min_d}$

Where min_d and max_d respectively are minimum and maximum values of the dissimilarity function d.

1.1.2 Proximity measures of data objects

A similarity or dissimilarity measure that perfectly fits all data types does not exist. However, for specific objects' data types, certain measures prove better than others. This section will cover some examples of the similarity and dissimilarity measures that were used in further work and discuss their advantages and weaknesses.
A distance measure d is called a metric when the following conditions are satisfied:

- $d(x, x) = 0$
- Positivity: $x \neq y \Rightarrow d(x, y) \geq 0$
- Symmetry: $d(x, y) = d(y, x)\ \forall x, y$
- Triangle inequality: $d(x, y) + d(y, z) \geq d(x, z)\ \forall x, y, z$

Various similarity measures often do not satisfy all the conditions to be called metric, but commonly satisfy two conditions:

- Simmetry: $s(x, y) = s(y, x)\ \forall x, y$
- $s(x, y) = 1 \Leftrightarrow x = y\ \forall x, y$; i.e. $s = 1$ iff $d = 0$, the consequence of Positivity

where s is a similarity measure function.

Minkowski distance

An example distance measure in multidimensional space is Minkowski distance [81] defined as:

$$Minkowski(x, y) = (\sum_{i=1}^{n} |x_i - y_i|^p)^{\frac{1}{p}} \tag{1.7}$$

where p is the parameter and n is the size of vectors x and y. Commonly used values of p are:

- $p = 1$; Manhattan distance [81] or L_1 norm

$$Manhattan(x, y) = \sum_{i=1}^{n} |x_i - y_i|. \tag{1.8}$$

  A common example is Hamming distance [14] where a distance between binary data objects can be computed by counting attributes for which the two compared objects have different values. This is the approach of Hamming distance measure, formally defined as:

$$Hamming(x, y) = \sum_{i=1}^{n} d(x_i, y_i) \tag{1.9}$$

  where d is a distance measure between nominal attributes. Although the Hamming distance is easily computed, the measure can provide misleading values when computed on sparse vectors. The measure equally treats matches between two ones and two zeros, meaning any two sparse vectors will have a low distance value due to a large number of matched zero attribute values.

- $p = 2$; Euclidean distance or L_2 norm [81]

$$Euclidean(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}. \tag{1.10}$$

- $p = \infty$; Supremum or L_∞ norm [81], defined as

$$Supremum(x, y) = \lim_{p \to \infty} (\sum_{i=1}^{n} |x_i - y_i|^p)^{\frac{1}{p}} = \max_{i=1 \ldots n} |x_i - y_i|. \tag{1.11}$$

Minkowski distance, for any given value of the parameter p, in general, is not suitable for high-dimensional data. However, Minkowski distance is commonly used in clustering algorithms for low-dimension data, such as DBSCAN mentioned on page 17.

Jaccard coefficient

The Jaccard coefficient [58] is a similarity measure that considers binary vectors as sets of items included in a transaction. Each vector coordinate corresponds with an item where the item is included in the transaction if its respective attribute value is non-zero. This measure ignores matches between two zero values, solving the issue recognized in the case of Hamming distance. The Jaccard coefficient for sets A and B is defined as:

$$Jaccard(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{1.12}$$

or, specifically for binary vectors

$$Jaccard(x, y) = \frac{f_{11}(x, y)}{f_{01}(x, y) + f_{10}(x, y) + f_{11}(x, y)} \tag{1.13}$$

where $f_{ab}(x, y)$ are count functions counting the number of attributes where vector x has the attribute value equal to a and vector y has a value of b for the same attribute.

The measure does not take into account the frequency values of the items in the set, just their presence in both compared transactions.

Tanimoto coefficient

Tanimoto coefficient [72], or extended Jaccard coefficient, uses the idea of the Jaccard coefficient but is modified to be more suitable for non-binary vectors. The coefficient is defined as:

$$Tanimoto(x, y) = \frac{x^T y}{||x||^2 + ||y||^2 - x^T y} \tag{1.14}$$

The measure applied to binary data results in the same values as those computed using the Jaccard coefficient. Both Jaccard and Tanimoto coefficients can be used in deriving new semantic similarity relations explained on page 47.

Cosine similarity

The cosine similarity measure [36] is among the most commonly used similarity measure between vectors, specifically document-term vectors. The measure is defined as a cosine of the angle between the normalized input vectors x and y:

$$CosineSimilarity(x, y) = \frac{x \cdot y}{||x||\ ||y||} \tag{1.15}$$

where $\cdot$ denotes the vector dot product. As the measure computes only the cosine value of the angle, the vector intensities are ignored, which can result in the highest similarity value of one even if the vectors are not equal but one vector is a scalar multiple of the other.

Correlation

Correlation is a statistical measure that computes a linear relationship between vectors. The vectors can represent the values of two variables or two objects. The definition of the correlation measure is not unique. One specific correlation measure is Pearson's correlation [20] defined as:

$$Pearson(x, y) = \frac{\Sigma(x, y)}{\sigma(x) \cdot \sigma y} \tag{1.16}$$

where Σ is a covariance function and σ is a standard deviation function. The values of Pearson's correlation are in the range $[-1, 1]$, where the value of 1 indicates a perfect positive correlation, while the value of -1 indicates a perfect negative correlation. The value of 0 indicated no correlation between the vectors. Another approach to computing the correlation between the variables is using Kendall's rank correlation [45]. Kendall's rank correlation, or Kendall's τ coefficient, is a non-parametric statistic test that does not require variables to come from the normal distribution. The only requirement for the tested variables is to have ordinal attribute values. The τ value is calculated as:

$$\tau = \frac{C - D}{C + D} \tag{1.17}$$

where C and D are, respectively, the numbers of concordant and disconcordant pairs of ranks of the tested variables. The values of Kendall's rank correlations are also in the range $[-1, 1]$ and their interpretation is analogous to the interpretation of the Pearson's correlation value. Correlation values computed by any of the two listed correlation coefficients can also be used in deriving new semantic similarity relations explained on page 47.

1.2 Semantic similarities

As metadata that describes objects in a database is mostly textual, it is important to find semantic similarities among these descriptions in order to connect various objects together. The similarity in the sense of semantics relies on the measures defined in the previous section but requires a proper transformation of the objects to a vector format suitable for the computation of the similarity measures. There is no single way of transforming the data. Transformation depends on the properties of the objects, such as object type, available object attributes, additional annotations, and the availability of ontology information.

The ontology is a graphical structure containing objects and semantic relations between them, representing a basic knowledge about the observed objects and their relations. Besides observed objects, the ontology may contain concepts and categories that can define the hierarchy or taxonomy between the objects. Without knowing any of the objects' attributes, the similarity between two objects within a defined ontology can be derived from the number of edges on the shortest path connecting the two objects in the ontology graph. This measure is called *path* [69] and is defined as:

$$PathSimilarity(x, y) = \frac{1}{MinPathLength(x, y) + 1} \tag{1.18}$$

where $MinPathLength$ is a function that returns the length of the shortest path between the objects x and y in the ontology graph.

Observing the ontology graph constructed from the organism taxonomy relations, the shortest path between two taxons is also the only path between them, due to the tree structure of the ontology graph. It is expected that the similarity is higher between two mammals than between a mammal and bacteria. The length of the path in the ontology graph is inversely proportional to the similarity, thus the *path* similarity reflects the true nature of the semantic similarity relation.

Analogous to counting the edges on the path, the distance between nodes can be seen as the number of edges on the paths from the two objects to their least common subsumer (LCS), also known as the lowest common ancestors (LCA) in tree-like graphs [8]. The number of edges on the shortest path between the objects and the number of edges from the objects to their LCS is equal, but the reformulated description of the problem introduces another similarity measure derived from the distance between the objects and their LCS. This similarity can be defined as the relation between the distance of the LCS from the root node and the sum of distances from the objects to the root. The distance between the root node and other objects represents the *depth* of the objects in the ontology graph. Denoted as *wup* measure, from the names of the authors that first introduced it, this measure was first introduced in the context of text terms ontology [93] but can be generalized to any hierarchical ontology. For two objects x and y in the ontology graph, *wup* similarity measure is defined as

$$WupSimilarity(x, y) = 2 \cdot \frac{Depth(LCS(x, y))}{Depth(x) + Depth(y)} \tag{1.19}$$

Another similarity measure obtained by combining both the shortest path distance and node depth distance [52] can be used for determine similarity between two objects x and y:

$$CombSimilarity(x,y) = e^{-\alpha \cdot MinPathLenght(x,y)} \cdot \frac{e^{\beta \cdot Depth(LCS(x,y))} - e^{-\cdot Depth(LCS(x,y))}}{e^{\beta \cdot Depth(LCS(x,y))} + e^{-\beta \cdot Depth(LCS(x,y))}} \tag{1.20}$$

where $\alpha \geq 0$ and $\beta \geq 0$ are user-defined parameters. Parameter α controls the influence of the minimal path length while the parameter β controls the influence of the node depth in the ontology tree.

When the ontology information is not available, a large corpus of documents containing information about the objects can be used to infer similarities between the objects based on their mutual appearances. An example of such an approach is tf-idf (term frequency-inverse document frequency) vectorization of the text documents and computing cosine similarity between the document vectors.

$$tf(d,t) = \frac{f(d,t)}{\sum_{t' \in d} f(d,t')} \tag{1.21}$$

$$idf(D,t) = \log(\frac{N}{|t \in d : d \in D|}) \tag{1.22}$$

$$tf\text{-}idf(D,d,t) = tf(d,t) \cdot idf(D,t) \tag{1.23}$$

where tf represents the relative frequency of the term t in a single document d, scaled by the total number of different terms in the same document. Function $f(d,t)$ is the frequency of the term t in document d. Function idf is the inverse document frequency, having a lower value when the term is present in a higher number of documents. D represents a corpus of documents while N is the number of documents in the corpus D.

The tf-idf vectorization is based on the term frequencies in the documents. The term that is frequent in some documents, but not so frequent in others, is a good representative of the group of documents where it is frequent. On the other side, if the term is frequent in all documents, such a term does not provide any distinction between the documents and its significance is low. Special cases of frequent words without special meaning are stopwords[1]. Those terms are removed before the transformation. The terms may come in different forms, singular and plural nouns, present and past tenses, and similar. To account for the different forms of the same term, the stemming procedure is performed, reducing all terms to their base form. The values of the tf-idf vectors are an example of information content (IC), where the IC of a concept is defined as the information derived from the concept when it appears in context [18]. A formal definition of the IC is:

$$IC(c) = -log(p(c)) \tag{1.24}$$

[1]Stopwords are defined specifically for each language. Some examples of stopwords in the English language are: the, a, an, is, has...

where $p(c)$ is frequency concept c in a given corpus.

If the ontology is also known, the IC value of LCS of the terms can be used for computing similarity between the objects:

$$IC_{LCS}Similarity(x, y) = IC(LCS(x, y)). \tag{1.25}$$

Analogous to the relation between the *path* and the *wup* similarities, the similarity based on the IC of the LCS can be modified to account for IC of individual objects [54].

$$IC_{LCS}SimilarityModified(x, y) = 2 \cdot \frac{IC(LCS_(x, y))}{IC(x) + IC(y)} \tag{1.26}$$

Combining the IC value for the LCS with the *path* similarity adds support for weighted edges to *path* similarity measure. The resulting measure is denoted as *wpath* [99]:

$$WPathSimilarity(x, y) = \frac{1}{1 + minPathLength \cdot k^{IC(LCS(x,y))}} \tag{1.27}$$

where the value k is a configurable parameter dependent on the knowledge graph implementation and data domain. [18].

tf-idf matrix of a set of documents can be used as input for further analysis, revealing hidden topics within the documents and the relations between the topics and words that are the representatives of the topics. The method used for extracting the topics from the *tf-idf* matrix is the latent semantic analysis (LSA) [24], which represents the singular value decomposition (SVD) applied to the text document data. The decomposition performs a dimensionality reduction, reducing the individual words to topics, thus uncovering the latent topics found in documents. The similarity between two documents can then be computed as the cosine similarity between two document vectors with values representing weights of the topics in the given documents, computed using the LSA. LSA transformation starts with a bag-of-words representation of text documents, where the counts are computed for each word, regardless of the order of words in sentences. The text is then vectorized to a count matrix representation, or *tf-idf* and the LSA transformation attempts to decompose the text matrix into three matrices, U, Σ and V, where Σ is a diagonal matrix of singular values of the text matrix and U and V are respectively matrices of left and right singular vectors of the text matrix. Interpretation of the matrix U is the weight of the relation of the concepts found in text documents (corresponding to rows of the matrix U) in relation to individual original documents. Matrix V reflects how important individual words in all documents are in different topics found in the text. The *tf-idf* matrix can be used for efficient keyword semantic searches mentioned on page 51.

$$LSA(D_C) = U \cdot \Sigma \cdot V^T \tag{1.28}$$

where D_C is the document count matrix or $tf\text{-}idf$ matrix, U is the dimensionally reduced matrix of document-topic weights, Σ is the covariance matrix and V^T is the matrix of term-topic weights.

If the documents are represented as sets of words, the similarity can be computed using the Tanimoto coefficient. This approach is useful for performing a keyword search against a set of documents.

The defined similarities can be combined to derive new similarity measures. The similarities between the objects can be materialized as weighted edges representing similarity relations between the object nodes in the knowledge graphs.

1.3 Related work

Current trends show that knowledge graphs are becoming more popular and widely used in many domains where the semantic search can be applied [41]. Using knowledge graphs for interconnecting data from biological data sources is not a novel idea. Knowledge graphs are the foundational structure for intelligent health care [92].

There is a broad range of knowledge graphs in the biomedical domain. The Biolink model [84] defines a data model for biological entities and is used in some open-source solutions, like ROBOKOP (Reasoning Over Biomedical Objects linked in Knowledge-Oriented Pathways) [12]. ROBOKOP system offers a knowledge graph of biomedical data that uses the Biolink model to represent high-level schemas. Monarch [77] is another open-source initiative with a knowledge graph solution based on relations between genotype and phenotype from different species and data sources. The system provides data querying by identifiers and neighborhood relations but does not enable querying patterns in graph data. Data loading is done using the Koza data transformation framework [49] maintained by the Monarch initiative [77]. GeneCard.org [29] provides tools for searching human gene-centric data aggregated from more than 190 different data sources. Elsevier's Biology Knowledge Graph [25] is a commercial, closed-source software solution designed to link biological data from multiple sources through manual annotation, automatic generation of links between data, and natural language processing of research papers.

Natural language processing enables detecting similarities between entities described using unstructured text. It also enables extracting knowledge from research documents, classification, and preprocessing text data to create a knowledge base based on semantic similarities. Transformers [85], deep learning models relying on attention mechanism [46], are one of the most powerful methods for dealing with vectorization of text data. BioBERT [51] is a biomedical language representation model designed for biomedical text mining tasks. It is a domain-specific adaptation of

the BERT (Bidirectional Encoder Representations from Transformers) [23] model. The semantic similarities between entities discovered using natural language processing can be used for constructing relation edges in knowledge graphs and further enriching the knowledge base beyond simple matched identifier links.

1.4 Related topics

The following chapters will introduce the basic premises required for understanding the bioinformatics data, problems of data unifications, and the proposed solution for semantic unification and searching of bioinformatics databases.

Chapter 2 will introduce clustering and association rules mining algorithms that can be utilized to recognize patterns in data and help in deriving semantic similarity relationships between data objects.

Chapter 3 will provide an overview of some of the most representative bioinformatics databases and address the problems of searching the bioinformatics data due to heterogeneous data schemas, differences in ways of accessing data, and problems that arise from the existence of multiple different identifiers for the same entities.

Chapter 4 will provide a theoretical insight into the proposed data joining model which will be a foundation for unifying bioinformatics data and developing the system for deriving new semantic similarity relationships between data objects

Chapter 5 will provide details on how the proposed model can be implemented, along with the system for deriving new semantic similarity relationships from data.

Chapter 2

Cluster analysis and association rules mining

Due to their importance for finding semantic relations, a separate chapter is assigned for cluster analysis and association rules mining. This chapter aims to provide an overview of different cluster analysis and association rules mining methods as well as metrics for evaluating the quality of the created clusters and association rules. The clustering methods used for extracting semantic relations in the following chapters will be explained in more detail.

2.1 Cluster analysis

Cluster analysis is the process of grouping data objects based on information found only in the data that describes the objects and their relationships. There are two broad goals of clustering: clustering for understanding and clustering for a given utility [81]. Clustering for understanding provides insight into natural groups found by observing similarities between data objects. On the other side, clustering for utility groups data in a certain way that provides a representation (or abstraction) of multiple objects in a dataset using clusters that contain them, for a given utility. Often the key property that guides cluster formation is having higher similarities between the objects in the same cluster in comparison to similarities between the objects in different clusters [81]. The choice of similarity and dissimilarity measures highly influences the quality of clustering results. As no additional user inputs are required, besides data values or precomputed similarity or dissimilarity values between the objects, clustering methods are examples of unsupervised learning methods. Additionally, new object instances can be assigned to existing clusters based on similarities to other objects from the cluster, thus performing an unsupervised classification.

2.1.1 Cluster types

The clusters are formed in a way that serves the original purpose of clustering. Due to differences in the purposes of clustering, there are different types of clusters [81]:

- Well-separated clusters; Elements of a cluster are closer to each other than to any element from other clusters.
- Prototype-based clusters; Distances between elements of a cluster and a prototype that defines the cluster are smaller than the distances to the prototypes of other clusters. Examples of cluster prototypes are centroids, for continuous attributes, or medoids, for categorical attributes.
- Graph-based clusters; Data objects are represented as graph objects interconnected with given relations. Clusters represent induced subgraphs of highly connected data objects.
- Density-based clusters; Clusters represent dense regions of data objects separated by regions of lower density.
- Conceptual clusters; Clusters define groups of objects that share certain properties. For example, pixels of a black-and-white image can be done based on the proximity of the pixels of the same color, where the color represents a shared property.

2.1.2 Properties of clustering methods

There are multiple characteristics of clustering methods that can be used for grouping the methods. Based on the existence of a hierarchy between computed clusters, clustering methods are divided into partitional and hierarchical methods. Partitional methods divide the elements into non-overlapping subsets, while hierarchical clustering methods create clusters that can contain nested clusters where a data object may be a member of multiple hierarchically organized clusters.

Some methods can create clusters that are not exclusive but also not hierarchical, This is the case with fuzzy clustering methods which assign each element to each cluster with a certain degree of membership. The degree of membership can be seen as the degree of truth of the fact that an element belongs to a cluster with a value in the range $[0, 1]$.

While other methods try to assign every element to some cluster, there are cases where not all data objects can be assigned naturally to any group. Such outlier elements can be detected and specifically labeled by some clustering methods. Based on those properties, the clustering methods can be divided into partial and complete clustering methods. Complete methods create results where each element has

to belong to some cluster, while with partial methods elements do not have to be assigned to any cluster.

2.1.3 Categorization of clustering methods

There is no unique categorization of clustering methods. This section will follow the categorization provided by Dongkuan Xu and Yingjie Tian [94]. Additionally, hard distinctions between different categories of clustering methods are not always possible, since some algorithms share properties of multiple categories. Clustering methods are divided according to the basic algorithm into 9 categories

- partition;
- hierarchy;
- fuzzy theory;
- distribution;
- density;
- graph theory;
- grid;
- fractal theory;
- model.

Partition-based algorithms

Algorithms based on partition create clusters by grouping points based on their proximity to a cluster center. Examples of partition-based algorithms are K-Means [34] and K-Medoids [64]. The time complexity of these algorithms is relatively low, but they can get stuck in a local optimum. They are also sensitive to outliers' presence and require a number of clusters to be set as a parameter.

Hierarchy-based algorithms

Algorithms based on hierarchy create hierarchical relationships between clusters. Construction of hierarchical clusters can be performed bottom-up (agglomerative), where initially each cluster contains a single object and they are merged until only one cluster containing all objects remains. Another approach is a top-down (divisive) method which begins with one cluster containing all elements and clusters are divided until each element is contained in an individual cluster. Examples of

hierarchy-based algorithms are BIRCH [98], ROCK [30] and Chameleon [43]. Algorithms that belong to this category generally have a high time complexity but they are not sensitive to differences in the shapes of clusters.

The clusters generated using the hierarchical methods are represented using a tree-like structure called a dendrogram, where the clusters are connected to their parent clusters using branches and the length of each branch reflects the distance between the connected clusters.

Agglomerative hierarchical clustering iteratively merges the currently closest clusters into a new cluster until one cluster remains. The distance between the two objects can be computed using some of the distance metrics defined in the previous sections. There are several ways of selecting which two clusters should be merged for being the closest, based on the linkage type. There are five commonly used linkage types:

- single linkage;
- complete linkage;
- average linkage;
- centroid linkage;
- ward linkage. [91]

Single linkage merges clusters C_A and C_B by the smallest minimum distance between the elements in the two clusters:

$$\min\{dist(x_i, y_j)|x_i \in C_A, y_j \in C_B)\}. \tag{2.1}$$

Complete linkage merges the clusters by the maximum distance between the elements in the two clusters:

$$\max\{dist(x_i, y_j)|x_i \in C_A, y_j \in C_B)\}. \tag{2.2}$$

Average linkage selects two clusters with the minimum average distance between the elements in the clusters:

$$\frac{\sum_{x_i \in C_A, y_j \in C_B} dist(x_i, y_j)}{|C_A||C_B|}. \tag{2.3}$$

Centroid linkage merges clusters with the smallest distance between the cluster centroids:

$$dist(c_A, c_B) \tag{2.4}$$

where c_A and c_B are respectively centroids of the clusters C_A and C_B.

The Ward linkage method merges clusters in a way that optimizes a selected target function of the resulting cluster. A frequently chosen criterion is the variance criterion where the target function is the variance of the merged cluster, thus the clusters

are selected for merging in a way that gives the smallest variance of the resulting cluster.

An example method that performs agglomerative hierarchical clustering is UPGMA (Unweighted pair group method with arithmetic mean) [59] which uses unweighted average linkage for merging the clusters.

Divisive hierarchical clustering or DIANA (Divisive analysis) [44] starts with a single cluster containing all objects of the dataset and iteratively divides the clusters so that the resulting clusters have the least similarity with each other. The division process stops when each cluster contains a single object.

Algorithms based on fuzzy theory

Algorithms based on fuzzy theory assign all objects to all clusters with a given membership level in the range of $[0, 1]$. Examples of algorithms belonging to this category are Fuzzy c-means (FCM) [11], Fuzzy c-shells (FCS) [22], and mountain method clustering (MM) [95]. The time complexity of these algorithms is low on average, but they require a preset number of clusters and are highly sensitive to parameter selection.

Distribution-based algorithms

Algorithms based on distributing group objects by the distribution from which they most likely originate. An example algorithm from this category is the Gaussian Mixture Model (GMM) [70], where it is assumed that points originated from several Gaussian distributions. These algorithms have a reasonably high time complexity but are very sensitive to the selection of the base parameters and the underlying distribution assumptions. However, the results of these algorithms may reflect the true probabilities of elements belonging to different clusters.

Density-based algorithms

Densely colocated points are recognized as clusters by algorithms based on density. A data point is considered to belong to the same cluster as the points within its same-density neighborhood. Examples of the algorithms belonging to this category are DBSCAN [26] and OPTICS [4]. The main problem with these algorithms is high sensitivity to different density clusters and parameter selection which may result in bad clustering results. Their time complexity is reasonably high but their memory complexity can become very high. The good side of these algorithms is their adaptability to data of arbitrary shape.

Algorithms based on graph theory

Algorithms based on graph theory consider data objects as graph nodes where the edges between them represent relationships between the nodes, such as high-similarity relations or closest neighbors. Some of the algorithms belonging to this category are spectral clustering [55], which will be described in detail, MST-based (minimum spanning tree) clustering algorithms [40], and CLICK [7]. These algorithms produce very good clustering results but the time complexity quickly grows as the graph becomes more complex.

The spectral clustering algorithm is a graph-based clustering algorithm, suitable for high-dimensional data and clusters of arbitrary shapes. The main idea of the algorithm is mapping data objects to a lower-dimensional embedding, also known as spectral embedding, and performing the K-Means algorithm on the mapped values.

The algorithm starts with a sparse graph adjacency matrix of data objects. The graph is commonly constructed as a neighbor graph, where each node, representing a data object, has edges connected to its k nearest neighbors following a selected proximity measure. A value in the adjacency matrix at index (i, j) indicates proximity between objects i and j. Let A be a symmetric adjacency matrix of a graph:

$$A = \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nn} \end{bmatrix} \tag{2.5}$$

Elements a_{ij} are weights of the relations between nodes i and j, where diagonal elements a_{ii} are equal to zero. With diagonal matrix D, defined as:

$$D = \begin{bmatrix} \sum_{j=1}^{n} a_{1j} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \sum_{j=1}^{n} a_{nj} \end{bmatrix} \tag{2.6}$$

$$D_{i,j} = \begin{cases} 0, i \neq j \\ \sum_{k=1}^{n} a_{i,k}, i = j \end{cases} \tag{2.7}$$

a Laplacian matrix L of the adjacency matrix A is computed as:

$$L = A - D = \begin{bmatrix} -\sum_{j=1}^{n} a_{1j} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & -\sum_{j=0}^{n} a_{1j} \end{bmatrix} \tag{2.8}$$

For the matrix L, eigenvalues λ and eigenvectors v are computed. The eigenvalues and eigenvectors satisfy the equation:

$$Lv = \lambda v. \tag{2.9}$$

The Laplacian matrix L is symmetric, positive semi-definite, and all eigenvalues are non-negative. Eigenvalues sorted in ascending order represent a spectrum and eigenvectors sorted in ascending order by their respective eigenvalues represent embeddings of the original data values. The first k eigenvectors are selected to construct a matrix V. Finally, the K-Means algorithm is applied to matrix V for finding k clusters in new reduced space.

Grid-based algorithms

Algorithms based on a grid map the data objects to a multi-resolution grid, quantizing the attribute into predefined grid fields. Examples of the algorithms from this category are STING (Statistical Information Grid in Data Mining) [90] and CLIQUE (Clustering in Quest) [2]. The time complexity of these algorithms is lower and their execution can be parallelized, but the grid resolution has a high impact on the quality of the clustering, requiring a proper balance between the quality of the results and time complexity.

Algorithms based on fractal theory

Fractal-based clustering algorithms utilize the fractal properties where subsets can share the same properties as a whole set. An example of an algorithm that belongs to this category is FC [5] algorithm. The time complexity is low for fractal-based algorithms with low sensitivity to outliers and adaptability to arbitrary shapes of clusters. However, for the best efficiency of the algorithm, data has to satisfy the fractal property. Fine-tuning algorithm parameters is also a critical requirement.

Model-based algorithms

Algorithms based on models are based on the assumption that different clusters are formed following different underlying models. These methods can be divided into methods based on statistical learning methods and those based on neural network learning methods. Statistical learning methods are closely related to distribution-based methods, which is why the GMM method is also a member of this category. Self-organizing maps (SOM) [47] is a model based on unsupervised neural network methods and associates data objects and clusters with nodes of a neural network. A wide variety of model-based methods suitable for various specific use cases are available, although their time complexity is generally high and they highly depend on the choice of parameters.

2.1.4 Measuring quality of clustering results

All clustering methods can discover some clusters or outliers in a given dataset even if there are no naturally defined groups in the data. That is why it is essential to define metrics that can be used to analyze the quality of clustering results and determine if data has a clustering tendency in the first place.

Determining the clustering tendency of data can be performed by analyzing the data distribution. If the data objects are distributed randomly, following a uniform distribution, the clustering tendency is very low. This is the core assumption of a Hopkins statistic [35] that compares data distribution against the uniform distribution. To compute the statistic, k points are generated and randomly distributed in the space of the data objects. For each randomly generated point x_i, value u_i represents the distance between x_i and the closest neighbor object from the original dataset. A value w_i is defined as the closest distance between objects y_i from a k-size sample of the original dataset and their closest neighbors in the original dataset. Finally, Hopkins statistic H is defined as:

$$H = \frac{\sum_{i=1}^{k} w_i}{\sum_{i=1}^{k} u_i + \sum_{i=1}^{k} w_i} \tag{2.10}$$

Values of the Hopkins statistic close to 0.5 indicate a low clustering tendency, while values close to 0 or 1 indicate a highly clustered dataset and data that are regularly distributed in the data space, respectively. Evaluating the statistic multiple times, by generating different sets of random objects, and computing the average H value gives more stable results.

There are two types of techniques for evaluating clustering quality:

- unsupervised techniques;
- supervised techniques.

Unsupervised techniques for evaluation of clustering quality

Unsupervised techniques evaluate the quality of clustering based on the properties of data and do not require external inputs. A general approach to evaluating clustering quality estimates how close the objects in clusters are in relation to distances between the points in different clusters. Smaller intracluster distances and bigger intercluster distances are good indications of higher-quality clustering. A quality measure that relies on intracluster distances is cohesion, while a measure that relies on intercluster distances is separation. The overall quality of the clustering can be expressed as a weighted sum of the validity metrics for each cluster, where a validity metric can be cohesion, separation, or a combination of both. If data is represented as a graph, where data objects are nodes and proximities between data objects are expressed as

weighted edges, the cohesion of a cluster C_i is defined as:

$$cohesion(C_i) = \sum_{x \in C_i, y \in C_i} proximity(x, y) \tag{2.11}$$

while the separation between two clusters C_i and C_j is defined as:

$$separation(C_i, C_j) = \sum_{x \in C_i, y \in C_j} proximity(x, y) \tag{2.12}$$

If the clusters are prototype-based clusters, with prototypes $c_1, c_2, ..., c_k$ respective for each of the k clusters, the cohesion of a cluster C_i is defined as:

$$cohesion(C_i) = \sum_{x \in C_i} proximity(x, c_i) \tag{2.13}$$

and the separation between two clusters C_i and C_j is defined as:

$$separation(C_i, C_j) = proximity(c_i, c_i) \tag{2.14}$$

An overall separation value of all prototype-based clusters can be defined as:

$$separation(C_i) = proximity(c_i, c) \tag{2.15}$$

where c is a prototype of the entire dataset.

For points in Euclidean space, a commonly used cohesion metric for a cluster C_i is a sum of squared errors (SSE), defined as:

$$SSE(C_i) = \frac{1}{2|C_i|} \sum_{x \in C_i, y \in C_i} dist(x, y)^2 \tag{2.16}$$

This metric is a good choice for globular clusters in Euclidean space, but not appropriate for clusters of arbitrary shapes, like the ones found using density-based clustering methods. The SSE metric can also be used for visual estimation of the number of clusters in data. The clustering algorithm is performed for every value of the parameter setting the target number of clusters in range $[2, M]$, where M is a user-defined parameter. For each clustering result, the SSE metric is computed and the result is plotted as a point on a chart with coordinates $(k, SSE(C^k))$, where k is the number of clusters in the current clustering and $SSE(C^k)$ is the average value of the SSE metric for clusters in the current clustering. A k coordinate of a point on a chart, visually recognized as an "elbow" point, where the decrease of SSE values is noticeably lower compared with the previous points is a good estimation of the number of clusters in data. The method is subjective and not always applicable but gives a good base estimation.

Another commonly used metric is the silhouette coefficient [44], which combines both cohesion and separation by computing the ratio between intracluster distances

and intercluster distances for each point. Let a_i be the average distance between object x_i and every other object belonging to the same cluster as x_i and let b_i be the average distance between the object x_i and all other objects from other clusters. Silhouette coefficient for the object x_i is defined as:

$$s_i = \frac{(b_i - a_i)}{max(a_i, b_i)} \tag{2.17}$$

Values of s_i are in a range $[-1, 1]$, where values close to 1 indicate that the object x_i is close to other objects from its own cluster and distant from the objects from other clusters. Negative values indicate a bad cluster label of the object x_i. The silhouette coefficient for the entire clustering can be expressed as an average silhouette score for each object in the dataset.

In the case of hierarchical clustering, a cophenetic distance measure can be used for evaluating the quality of clustering. Cophenetic distance between two objects, clustered using agglomerative hierarchical clustering, is the proximity at which the algorithm adds the objects to the same clusters the first time. If two clusters are merged at distance d, then the cophenetic distance between any objects in one cluster and any objects in the other cluster is d. A matrix of all pairwise cophenetic distances between objects can be used to compute the cophenetic correlation coefficient (CPCC) [27] as a correlation between the cophenetic distances matrix and the distance matrix used for performing the hierarchical clustering.

Supervised techniques for evaluation of clustering quality

Supervised techniques are techniques that require external inputs to accompany data, such as class labels. If the correct labels are known, clustering quality evaluation represents evaluating the overlapping degree between assigned cluster labels and true labels. More homogeneous clusters, in relation to true labels of objects within them, indicate better clustering.

Entropy is a measure that evaluates the level of disorder. If clusters are seen as sets of true labels of data objects contained within them, entropy can be used to evaluate the homogeneity of the cluster. For a cluster C_i and L possible class labels, entropy is defined as:

$$e(C_i) = -\sum_{j \in L} p_{ij} log_2(p_{ij}) \tag{2.18}$$

where p_{ij} is the probability of class j in cluster C_i. A perfect clustering should result in the best entropy score for each cluster. More precisely, the entropy of the entire clustering is evaluated as the sum of entropy values for each individual cluster.

Purity measure uses the notion of probability p_{ij} of a label j within a cluster C_i and evaluates the homogeneity of a cluster as:

$$purity(C_i) = max_j p_{ij} \tag{2.19}$$

The purity of clustering with K clusters is defined as a weighted sum:

$$\sum_{i=1}^{K} \frac{|C_i|}{|D|} purity(C_i), \tag{2.20}$$

where $|D|$ is the total number of objects in the dataset.

Precision and recall measures, defined as:

$$precision(C_i, j) = p_{ij}, \tag{2.21}$$

$$precision(C_i, j) = \frac{|\{x \in C_i | label(x) = j\}|}{\sum_l^K |\{y \in C_l | label(y) = j\}|} \tag{2.22}$$

and

$$recall(C_i, j) = \frac{m_{ij}}{m_j} \tag{2.23}$$

where m_j is the number of objects in class j, can be combined into F measure defined as:

$$F(C_i, j) = \frac{2 \cdot precision(C_i, j) \cdot recall(C_i, j)}{precision(C_i, j) + recall(C_i, j)}. \tag{2.24}$$

The F measure is suitable for evaluating the quality of hierarchical clustering as a weighted sum:

$$F(C_i, j) = \sum_{j}^{L} \frac{|C_j|}{|D|} max_i F(C_i, j) \tag{2.25}$$

where $|D|$ is the number of objects in the entire dataset.

In special cases, where there are two clusters and two external labels, the Jaccard coefficient, defined in the section on similarity measures, can also be used for evaluating the quality of clustering, comparing the ratio of matched cluster-label pairs against the overall number of non-0-0 matches.

2.2 Association rules mining

With data containing information about items included in different transactions, it is often needed to determine which items are often found together in the same transactions. The described problem is commonly described as determining which products are commonly found together in shopping carts, but it can be generalized to analyzing different types of itemsets.

Association rules are defined as expressions $X \longrightarrow Y$ where X and Y are itemsets. The two itemsets X and Y are disjoint. Itemset X is called antecedent and itemset Y is called consequent. The interpretation of the rule can be described as: "If the itemset X is a subset of a transaction itemset, then the itemset Y is probably a subset of the transaction itemset". A frequent itemset that can not be extended

without reducing its frequency is called a maximal frequent itemset.

For a rule $X \longrightarrow Y$ to be labeled as *interesting*, the union of itemsets X and Y needs to be frequent. This means that the items found in the union are frequently found together in the same transactions. The measure of the frequency of the rule itemsets is called support, defined as:

$$support(X \longrightarrow Y) = \frac{|X \cup Y|}{|T|}, \tag{2.26}$$

where T is the set of all transactions. Not all frequent itemsets are considered interesting. If the itemset X is present in all transactions in the dataset, many rules in a form $X \longrightarrow Y$ may be statistically common but there is nothing unexpected in them to make them interesting. However, if X is a common itemset and itemset Y is frequent in most of the transactions along X, then $X \longrightarrow Y$ may be considered as interesting. A measure that expresses this property numerically is called confidence, defined as a percentage of transactions containing itemset X and Y in relation to all transactions where the X is present:

$$confidence(X \longrightarrow Y) = \frac{support(X \cup Y)}{support(X)} \tag{2.27}$$

Another way to label a rule as interesting is by evaluating the probability of a rule $X \longrightarrow Y$ being randomly generated from the available itemsets X and Y if the itemsets were independent. The measure of this property is called lift, defined as:

$$lift(X \longrightarrow Y) = \frac{support(X \cup Y)}{support(X) \cdot support(Y)} \tag{2.28}$$

Finding the frequent itemsets as candidates for the interesting rules can be done by generating all possible itemsets and computing their support using a brute force method. However, there are 2^n possible subsets of a set with n elements and the direct approach for enumerating all possible subsets is not computationally feasible for bigger sets. The algorithms that search for frequent subsets use minimum support thresholds to identify and discard infrequent itemsets and their supersets.

2.2.1 Apriori algorithm

The Apriori algorithm [1] for finding frequent itemsets is based on the apriori principle, which states that if some itemset is frequent, then every subset of this itemset is also frequent. As a consequence, if some itemset is not frequent, then any superset of this itemset is not frequent.

The parameter that specifies the support threshold defines a lower limit of support for an itemset to be considered frequent. The algorithm generates a graph structure called a lattice starting with an empty itemset and iteratively generating the lattice

level by level. Items in every itemset in the lattice are sorted in lexicographic order. The itemsets on level $k+1$ of the lattice are generated by joining the itemsets from the level k which have equal first $k-1$ items in itemsets.

If the new itemset fails the minimum support criterion, neither the itemset nor any of its direct supersets are not included in the lattice. Another condition that could be set for the algorithm to focus and speed up the search is the maximum allowed size of the itemset.

The traversal of the lattice performed by the apriori algorithm resembles the breath-first search (BFS) traversal of the graph. To reduce the cost of storing the frequent itemsets, only maximal frequent itemsets should be stored.

2.2.2 FP-growth algorithm

The FP-growth (Frequent pattern growth) algorithm [32] skips the candidate generation using the lattice, as seen in the apriori algorithm. The algorithm consists of two steps where the first step generates an auxiliary tree structure, called an FP-tree, for storing frequent items. The items are collected from all transaction itemsets found in the dataset. Individual items with support lower than the minimum support threshold are discarded. Transactions containing only subsets of the remaining items are collected and their itemsets, represented as lists, are sorted descendingly by the global frequency of the items in the itemset.

Collected itemsets are mapped to the FP-tree tree nodes, starting from the empty root and adding items from the transactions in the sorted order as new nodes on a path from the root. When a new node is added, a count value associated with the node is initialized to one. Each time an existing node is visited during the mapping process of the itemsets to nodes, the visited node's count value is increased by one. The tree nodes are linked to other nodes in the tree representing the co-occurrence of the items in the itemsets.

The second step of the algorithm is collecting the frequent itemsets by traversing the FP-tree starting from the root in a DFS manner. The frequent itemsets represent individual paths from the root to leaves in the FP tree.

Chapter 3

Searching bioinformatics databases

Bioinformatics databases are mutually very heterogeneous. The differences between databases can be found in data source domains, data structures for representing data, sets of data attributes and their data types, formats for storing and delivering the data, and ways of accessing the data. All these differences make data searching challenging, especially when the data needs to be retrieved from multiple databases and have the query results combined.

3.1 Primary and secondary databases

Bioinformatics data primarily comes from biomedical domains. Each domain favors a different set of domain-specific databases, resulting in a large number of databases with little or no connections between them. Currently, there are more than 6000 bioinformatics databases cataloged in Database Commons [56] archive. The databases can be classified into two broad groups: primary and secondary databases. The primary databases act as repositories of raw data from individual experiments, while secondary databases contain data that is created based on the analyses and annotations of the data entries in primary databases. Each domain has a group of referent primary databases for depositing the raw data from experiments. Commonly used primary databases for storing nucleotide sequences are GenBank [9], DNA Data Bank of Japan (DDBJ) [82], Ensembl [37], and EMBL [42]. Primary protein sequence records can be found in UniProt [21] database while data regarding protein structure can be found in Protein Data Bank (PDB) [17]. However, the UniProt database also contains secondary protein data. Primary gene expression data can be found in NCBI Gene Expression Omnibus (GEO) [6] and gene expression data specifically from immune cells can be found in Database of Immune Cell EQTLs (DICE) [31]. Secondary databases can combine entries from multiple primary databases and provide a broader view of a certain domain. Signal pathway databases, such as Some other examples of secondary databases are the Database of Disordered Proteins [89] (DisProt), containing disorder information of

intrinsically disordered proteins, Immune Epitope Database [39] (IEDB), containing information on immune epitopes, and STRING database [80], containing data on protein-protein networks.

3.2 Identifiers of bioinformatics data

Biological entities stored in separate databases may still be identified using shared identifiers. Proteins are commonly identified using the accession identifiers from UniProt [21] database or using a gene identifier of the protein's source gene. Genes are often identified using their symbols or identifiers in one of the referent databases such as NCBI Gene [15], EMBL [42] or Ensembl [37]. UniProt accession identifiers are unique identifiers assigned to each protein sequence included in the UniProt database. As one protein may be sequenced in multiple experiments, each sequence is assigned a unique identifier, even though the primary sequence of the protein is the same. Therefore, one protein entity may have multiple UniProt accession identifiers. Gene symbols can identify genes but do not differentiate between the orthologs in different organisms. NCBI Gene identifiers are different for each gene from each organism. Ideally, each database containing gene entity data should provide the NCBI Gene identifier to remove any ambiguity, but, unfortunately, that is not the case. Diseases can be identified using Concept Unique Identifiers (CUI), as well as other, non-disease concepts. Overall, even if there are identifiers that are shared between multiple databases, it does not mean they are the only or even unique, identifiers for individual entities. Nucleotide and protein sequences are identified using accession numbers in their respective databases. There are identifiers that encode additional information, rather than being unique random values. Ensembl identifiers are commonly used for identifying genomic sequences in gene expression experiments, but can also be used for identifying proteins, transcripts, or exons. Determining the entity type behind the Ensembl identifier can be achieved using the fact that the Ensembl identifier follows a strict structure, where one letter of the identifier encodes the entity type. For example, the letter G, found on the fourth position of the identifier ENSG00000141510 encodes gene entity type while identifier ENST00000509496.1 represents a transcript (T) identifier. Additionally, the Ensembl identifiers of objects that are of non-human origin also encode the species reference using three-letter codes. The identifier of a gene sequence (G) of the TP53 gene from species *Canis lupus familiaris* (CAF) has a format of ENSCAFG00000016714.1. Unfortunately, Ensemble identifiers are used only in the domain of protein and genetic sequences and not as general-purpose identifiers for arbitrary biological entities.

3.3 Storing bioinformatics data

There are no strict rules on how bioinformatics data should be stored. The way of storing and organizing the data is highly dependent on the data type, size, and intended way of accessing the data. It is important to note that the underlying database used for data storage does not necessarily induce the format of the retrieved data. A database management system for storing data may be a relational database, but the data can be retrieved in JSON or even CSV format. Some databases provide REST (REpresentational State Transfer) API[1] [61] for accessing and querying the data, while others may allow downloading whole datasets in CSV or TSV file format. On the other hand, there are databases that do not explicitly allow data downloading but can display data on a web page that can only be downloaded in raw HTML format scraped, requiring additional processing before being ready for querying and analyzing. Another commonly used format for the internal representation of databases, especially for knowledge databases supporting semantic search are RDF [57] triplets, consisting of subject, relation, and object. Besides general-purpose data formats, such are JSON, CSV, and TSV, there are specialized data formats that require specialized parsers. Examples of such specialized formats are PDB (Protein DataBase) text format for representing the 3D structure of proteins, and SOFT (Simple Omnibus in Text Format) and MINiML (MIAME Notation in Markup Language) formats that are used for describing gene expression data found in NCBI GEO (Gene Expression Omnibus) [6]. Gene and protein sequences can be stored in FASTA and GenBank text formats, where FASTA is the basic format for text sequence representation while GenBank format supports additional annotations of the sequence segments as well as metadata regarding the sequence source, authors, and related organism taxonomy. Similar to FASTA, the FASTQ format is used to represent raw reads from sequencers. Specialized searching methods have been developed for searching DNA and protein sequence databases. The most popular searching method is BLAST [3], which performs efficient alignments between query sequences and sequences stored in the database.

3.4 Accessing and searching bioinformatics data

Many bioinformatic tools, such as NCBI E-utilities [73], can be run as command-line applications. A more user-friendly way of accessing data from bioinformatics databases is using a graphical user interface exposed through the database website. Users navigate through the interface using a Web browser and input queries in the input fields of the interface. For collecting small amounts of data for specific purposes, the graphical interface is the simplest way of accessing and querying the data. The queries can be keyword-based, where the user specifies a list of keywords that

[1] Application programming interface

are independently matched with the database records, and the results are sorted based on the number of matches. This approach is useful when searching by entity identifiers. Another type of querying is by using structured queries, where the user specifies the query using strictly defined language rules. One of the structured languages used in bioinformatics databases is SPARQL [66]. Finally, some databases support natural language queries, where the user inputs the query in free form, using natural language.

Manual searching for data from multiple databases and combining the results is a difficult and time-consuming task. Automation of the data access is enabled using database APIs. Databases often expose the APIs for searching the data using HTTP requests which enables implementing applications in arbitrary programming languages with the ability to access data from a remote application using a general-purpose protocol. In some cases, API responses may be difficult to parse, due to specificity in the response structure and data format of the received payload. To help with programmatical access to individual bioinformatics databases and interpreting specialized data formats, programming libraries were developed for some of the most commonly used programming languages in the bioinformatics domain. BioPython [19] library enables parsing and processing various bioinformatics data formats using Python programming language and also enables API access to services such are BLAST and Exonerate [79]. Entrezpy [16] library for Python programming language allows programmatical access to Entrez [74] databases. Ensembl provides Perl language API [96] for accessing their databases. BlasterJS [13] is a JavaScript-based library for interactive visualization of BLAST alignment results.

Research articles also represent a data source that can be used for extracting useful information. The articles are often used as inputs for text mining and natural language processing algorithms for extracting knowledge from text data. Data from the articles can be seen as a meta-source utilizing summarized results and comments from multiple analyses in different biomedical domains without interacting with the raw data of the individual experiments. Text descriptions of biological entities and their interactions derived from the articles give meaning to the biological entities mentioned in texts in various circumstances, observed from many angles. The acquired information, in the form of relations between the entities in different contexts, can be used for constructing a knowledge base, giving a more generalized view of the mechanisms where the entities are involved, instead of analyzing the entities on the individual level. Such relations can also be discovered not only from the text attributes but also from general metadata related to the entity using a variety of data mining algorithms.

Searching data from multiple databases is possible only by querying each individual database and combining the results into a final, joint result. To enable such simultaneous searching and seamless joining of the query results, one approach is to

create a generalized searching method that can be mapped to each data querying method of the individual databases. The first problem with such an approach is the continuous maintenance of the query method adapters, which may become obsolete with every update of the database API, thus breaking the system until the new adapter is developed. The second problem is the lack of joint data indexing and cross-database queries that would enable query optimizations, which can result in significantly slower queries due to separate steps of data fetching and data joining without efficient use of indexes.

Chapter 4

New Data Joining Model Proposal

Bioinformatics data stored in separate databases, with heterogeneous data schemas and formats, are not suitable for direct use in semantic searches. The bioinformatics databases also include metadata, or data about data, which provides additional context to the data entries. The bioinformatics data represent results from biological experiments, nucleotide and protein sequences, expression profiles, and similar, while metadata contains information such as collection date, location, descriptions of the analysis steps, additional notes, and comments. This thesis will primarily focus on utilizing metadata for semantic searching but also providing the ability to access the data itself.

The key motivation for this decision is creating a lightweight indexing network for finding relations between data objects that are stored in the original databases, instead of collecting all data. The second reason to choose metadata is the richness of the information contained in metadata valuable for detecting semantic similarities, in relation to data. Information on how a certain gene interacts with other genes is more valuable for understanding the semantic similarities between genes than the raw nucleotide sequence. And finally, the third, practical, reason to focus more on metadata instead of data is the differences in their size in the general case. The data objects, such as nucleotide and protein sequences may have a size of hundreds of megabytes or more, as well as high-resolution images from biomedical domain or crystallography experiments. On the other side, metadata size is commonly expressed in hundreds of kilobytes or a few megabytes. Because of that asymmetry in size, maintaining the metadata database in a local environment on a workstation computer is much more feasible than storing copies of all data objects.

Efficient searching of unified metadata requires a specialized database for storing the unified metadata. Such a database also requires a specific data model. Such a model should be as general as possible to properly map and store metadata from diverse data schemas found in the original databases. Besides storing metadata, the model needs to provide a foundation for efficient searching, by having a structure that allows efficient use of indexing. Finally, the model should be database agnostic

so that it can be implemented by an arbitrary DBMS.

Biological data contains information about biological entities and their properties. Those entities were recognized as the primary building blocks of the new model called BioGraph. An important remark is that the name of the model, unintentionally, matches the name of an unrelated project in a different domain [53]. Entities are not isolated objects but are highly interconnected in many ways. The relations between the entities have various origins, from biological processes to taxonomical relations and mutual similarities. The association of entities from multiple datasets is enabled by connecting relation paths among them. As many types of entities exist, creating specific model schemas for each biological entity and their relations is inefficient on the level of model definition. It violates the requirement of unified representation, and there is no feasible way of predicting all future properties and relations associated with every entity type. That is why the decision was made to design a model that enables storing metadata of arbitrary entity types in a generic form. Details regarding entity type-specific information are delegated to higher-level data schemas. It is essential to mention that the model links metadata related to biological entities in a way that can be used for efficient data location and retrieval from the original data sources. An example of gene metadata might be its identifier, while the data is a DNA sequence representing the gene. The gene sequence is not handled by the model but can be easily retrieved directly from the external source database, such as the NCBI gene, using the metadata linked in the model. The model structure and its implementation are published in [88]

4.1 BioGraph data model

The proposed model consists of three object types – entity objects, identifier objects, data objects, and relations that connect the objects. Model objects and their individual relationships are shown in Figure 4.1. Metadata is loaded from arbitrary data formats, from which entities, identifiers, and data objects are extracted along with respective relations. The extracted objects and relations form a unified knowledge graph where all relations are represented as directed edges of the graph. An example of a network of biological objects from five different sources (DisProt[89], DisGeNET[68], Tantigen[97], IEDB[39], and HGNC[75]) represented using the BioGraph model elements is displayed in Figure 4.2.

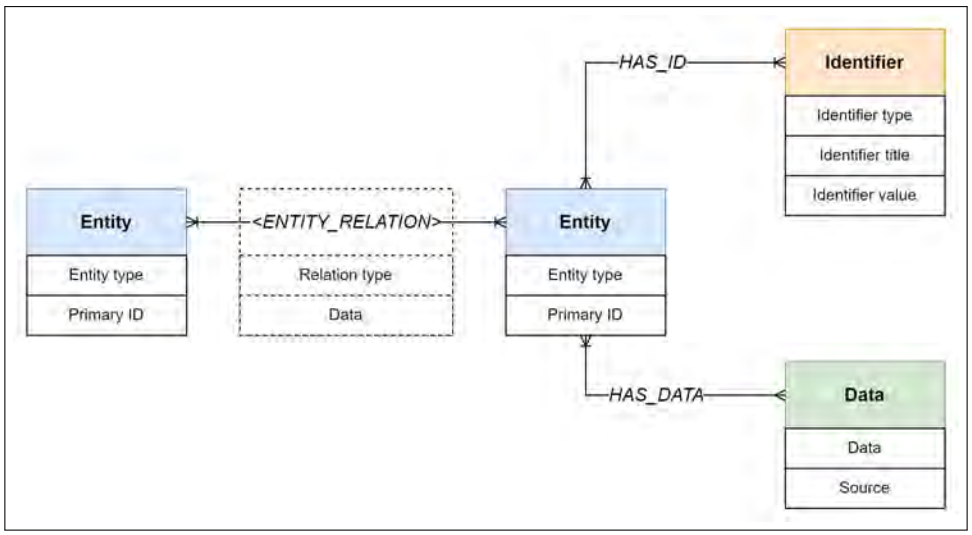

Figure 4.1: A schema of the BioGraph model. Entities of specified types, identified using primary identifiers, have connections with all of their identifiers and data objects, as well as with other entities. The identifier objects contain information about the nature of the identifier and the identifier values, while data objects store metadata collected from datasets along with the label of the dataset source. Relations between entities are defined using the relation type and additional data that can further explain the nature and strength of the relationship.

4.1.1 Entity objects

Biological entities have different types. Some are associated with biological functions, like genes or proteins, while others can be more general, like habitats and coverage areas. Regardless of the type, at least one unique identifier can be assigned to each entity, denoted as a primary identifier. When multiple unique identifiers exist, an identifier shared among most biological databases is selected as the primary identifier. An example of a primary identifier for a gene entity is a gene name. It is important to note that the data model is not restricted to human genetic data, but can also support data from arbitrary biological domains regardless of the data source. Entity type names and primary identifiers were used to construct entity objects in the BioGraph data model. An entity object represents a single biological entity. The same entity objects can be found in different databases. For each entity, idealy, only one entity object is created. In some cases, no identifiers in a dataset are used in any other database. In those cases, a new entity object is created, which can be subsequently connected to the other objects representing the same entity using "IS EQUAL TO" relation. Figure 4.2 shows concrete examples of biological objects (sourced from DisProt (Disorder Protein database) [89], DisGeNET (Disease Gene Network) [68], Tantigen 2.0 [97], IEDB (Immune Epitope Database) [39], and HGNC (HUGO Gene Nomenclature Committee) [75] datasets) and their relations. Five different entities are in blue boxes along with their type label and primary identifiers. Those entities, with their respective primary identifiers, are gene

"CDKN1A", protein "P38936", disease "C0038356", antigen "Ag002102", and epitope with the sequence "FAWERVRGL".

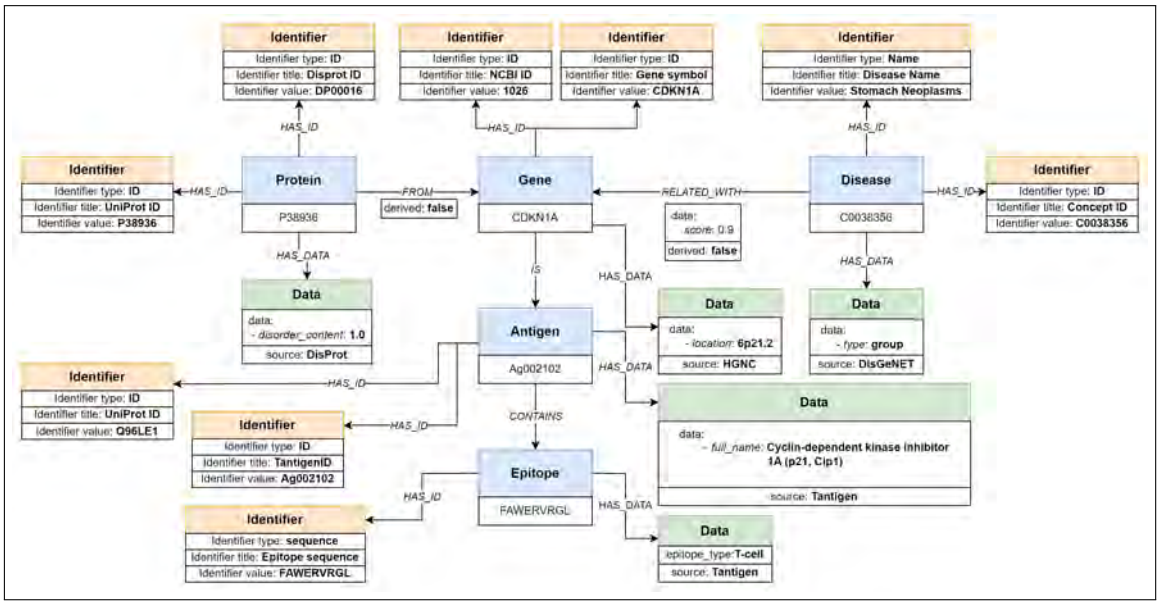

Figure 4.2: Network of objects from five different sources (DisProt, DisGeNET, Tantigen 2.0, IEDB, and HGNC) represented using the BioGraph model. The example shows protein data collected from DisProt dataset, with identifiers assigned by DisProt and UniProt along with disorder content value. Gene data, collected from HGNC dataset, is connected with protein data from DisProt dataset assigning the gene NCBI identifier. Disease data from DisGeNET is connected to the corresponding gene with an assigned relation score from DisGeNET dataset. Tantigen 2.0 and IEDB datasets add context to the antigen nature of the gene, including information on the epitope of the antigen.

4.1.2 Identifiers

All available entity identifiers, including the primary identifier, are collected and represented in the BioGraph model using identifier objects. The identifier objects are connected to their respective entity objects using identifier relations labeled as "HAS ID" relations. Each identifier contains information about the identifier type, title, and value. The identifier type defines the nature of the identifier. The model proposes three types of identifiers – a name, a URL, or a generic identifier without any specific type. The identifier title is used to determine the meaning of the identifier, such as "gene name" or "NCBI ID", which can also be useful when an external application displays the data from the model. Finally, the third component of the identifier contains a value of the identifier stored as a string data type. One identifier can be shared between multiple entities, so one identifier object can be connected to multiple entity objects. Additionally, one entity object may also have multiple identifiers. Identifiers do not list the source database from which they originated, as one identifier can originate from multiple databases but represented using only one identifier object. Figure 4.2 shows six identifiers in yellow boxes, connected to

five entities. The example protein entity is connected with two identifiers, where "P38936" is the identifier of the protein in the UniProt database (also used as a primary identifier) and "DP00016" is the identifier of the same protein in DisProt database.

4.1.3 Data objects

Besides identifiers, datasets may contain additional metadata about the entities, like protein regions, location coordinates, or gene positions on chromosomes. Those metadata values can be stored in data objects, along with the source label so that the metadata can be easily tracked and verified against the original dataset. Data objects, which contain entity metadata other than identifiers, are associated with their respective entities using data relations labeled as "HAS DATA". One entity object can contain multiple data objects, with metadata originating from different databases. Also, one data object can be shared between multiple entity objects. A unified representation of a data object, compatible with most of the DBMS software is a key-value representation, where the keys are attribute names and values are the values of the corresponding attributes. In relational database management systems, key-value pairs can be stored in a data object table or in attributes with JSON data type. Example in Figure 4.2 shows data objects that provide additional information to entities. Data object connected to protein entity contains disorder content value, gene metadata contains gene location on chromosome, disease metadata contains disease type annotation, antigen metadata contains the full name of the antigen, and epitope data contains type annotation for the epitope.

4.1.4 Entity relations

Relations can also exist between entities. A protein can be associated with its source gene, and a gene can be associated with the chromosome. As the entity relations are various, they are defined for each specific use case, but a general structure of an entity-entity relation is supported by the model. All relations in the proposed model share the same structure. The structure of a relation contains relation type, "derived" flag indicating if the relation is a similarity relation derived from data using data mining algorithms (explained in the following sections) and key-value pairs of metadata associated with that relation. A relation between genes and diseases has a certain score, which is a value associated with the relation and not with the individual entities. Therefore, the disease relation score for a gene has to be stored in a key-value pair of the gene-disease relation. The basic relationships between entities are shown in Table 4.1. The relations between entities are dynamic, meaning that new relations can be defined and added in updates without affecting the existing data. A pair of entities can share multiple relations.

4.1.5 Duplicate entries

Duplicate entries can have a negative impact on search results and overall performance. An object or relation is considered a duplicate when its entire content matches the content of an object or relation that is already stored in a database. An entity object contains a primary identifier and an entity type. If two entity objects share the same primary identifier, there are two possibilities: the two objects also share the same entity type – it is a duplicate object and only one representation should be stored in the database; the two objects have different entity types – both objects should be stored in the database. The entity objects in the example should also be connected with "HAS ID" relations to the respective identifier objects storing the primary identifier value. As the identifier objects are duplicate entries, in this case, only one identifier object should be stored while both entity objects should point to it with their "HAS ID" relations.

To detect duplicate entries, the proposed model assigns specific identifiers to objects and relations based on their content. The method for generating such identifiers is called content addressing. Data stored in an object or relation is serialized to a string representation, containing all object and relation information, and mapped to a value from a large interval using a hash function, such as SHA256[33]. In the example of an entity object, both entity type and primary identifier are used in hashing. The probability of having multiple different data objects mapped to the same value, also known as hash collision, using an industry-standard hash function is very low. If the same object is serialized and hashed multiple times, it will always result in the same value. The processing of hash collision where the objects are different is not currently implemented in the system, as it is a highly unlikely event, but the proposed solution includes adding specific salt values to objects which are by default set to 0 value and incremented for the objects that are in collision with the existing objects in the database.

4.1.6 Data updates

Thanks to an efficient handling of duplicate entries, updating the data can be easily performed by inserting new data objects and relations. All duplicate entries will be ignored while the relations will still seamlessly connect to all newly stored or already existing objects in the database.

4.1.7 Mapping BioGraph model to graph and relational database

The BioGraph data model is designed to be database agnostic, meaning that it can be implemented using any database management software. Although graph database systems are the most suited for the model implementation, this section

will describe the versatility of the model by mapping it to a relational database model, but a similar pattern is utilized for implementing the data model using a graph database where tables are replaced with nodes of different types. The model objects (entities, identifiers, and data objects) can be represented as graph nodes in a graph database system while the relations are naturally represented using graph edges. The relational database can also be used to represent graph-like structured data by having tables "Node" and "Relation". As there are three different types of nodes, each type is represented using individual tables related to the "Node" table as a specialization of the generic nodes. Attributes of the entity, identifier, and data nodes are defined in their respective tables while the identifier attribute "id" in the specialized tables matches the identifier of the generic node in the "nodes" table. A diagram of the proposed mapping is shown in Figure 4.3. Data node attribute containing key-value pairs is represented using JSON-type attribute. Relations have two foreign keys, "from_node_id" and "to_node_id", representing identifiers of the end nodes of the directed relation.

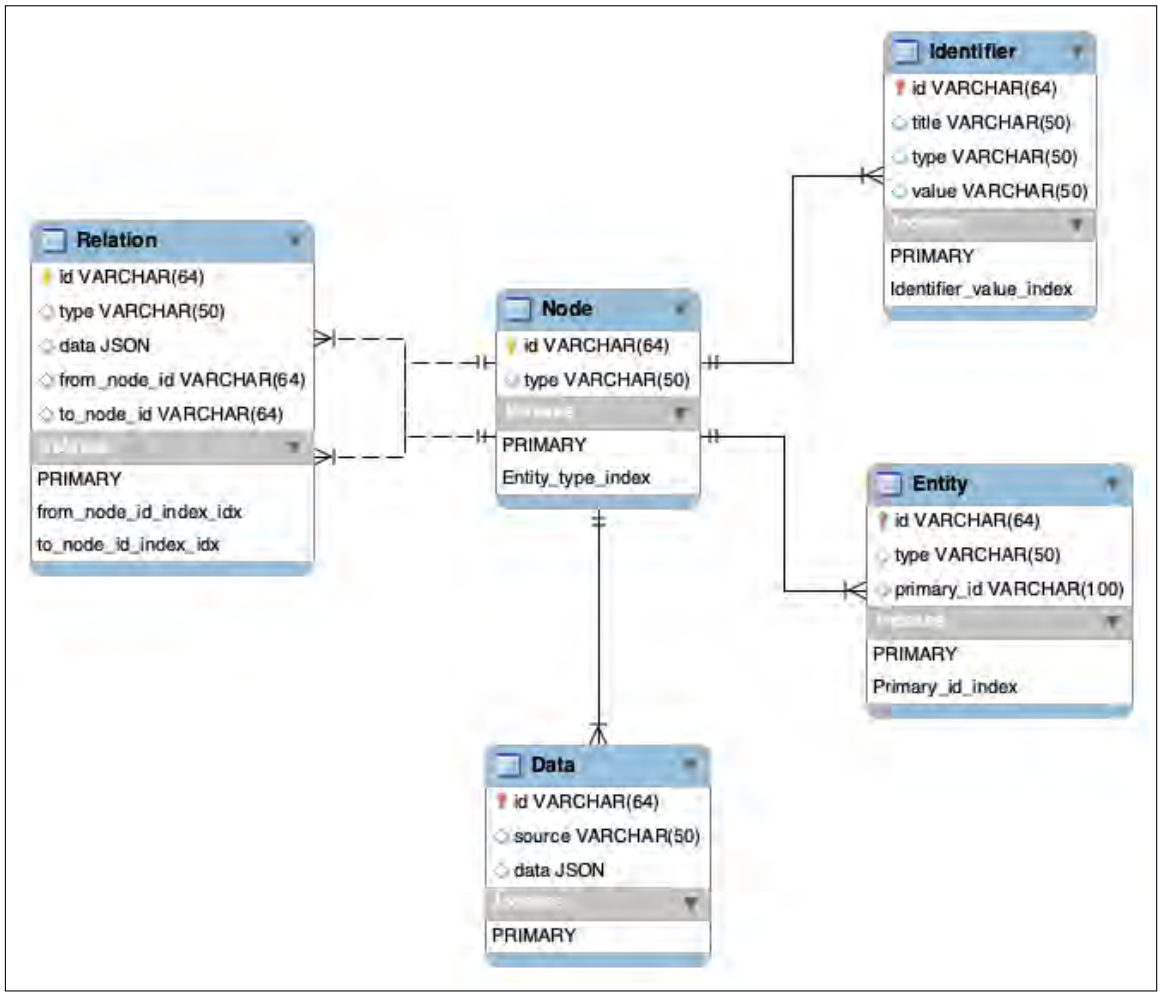

Figure 4.3: Diagram of the BioGraph data model mapped to a relational model.

4.1.8 Efficient indexing

Proper indexing is crucial for the efficiency of data loading and querying operations especially when the join queries are heavily utilized for querying paths in the graph represented in the described way using relational database tables. Indexing should be applied to attributes that play a crucial role in connecting tables, such as primary table identifiers, but also those that are heavily used in searching data, such as the "identifier_value" attribute of the identifier node and "primary_id" and "type" attributes of the entity node.

4.2 Generalized method for deriving semantic relations

As discussed before, semantic similarities can be used for adding relations to the knowledge graphs. The goal was to design a generic method for extracting new semantic relations from the unified data knowledge graph represented using the proposed data model. The designed method enables deriving new relations in a generic way, regardless of the analyzed entities, based on the existing relations found in the knowledge graph. The new relations are added to the knowledge graph and ready to be used along with the previously stored relations for generating new sets of semantic relations in an iterative manner.

Two entities are represented as nodes of the knowledge graph. A similarity between them can be estimated based on matching metadata values of the mutual attributes, found in their respective attribute key-value pairs stored in data objects. This approach can generate similarity relations, but the heterogeneity of the data objects makes the process difficult to generalize. Matching the identifiers and text descriptions can also be a foundation for computing the similarity. One of the issues with this approach is the lack of text descriptions for all entities found in original databases. The second problem is the generic form of identifiers where two identifiers may have very similar values, like "P01234" and "P01235", but the entities identified by the values may be very different.

The approach to computing semantic similarity between the entities is based on the relations connecting the observed entities and other entities in the graph. The motivation for this approach can be seen in the fact that two similar entities would express similar behavior, have similar interactions with other entities, or be related to similar concepts. For example, genes that produce cytokine proteins, which represent messages emitted by the immune system cells, may be involved in the inflammation processes and thus have relations with similar sets of diseases. The relation-based similarity can be specified in a certain context, as is the case with gene-disease relations, or be a general similarity between entities, where the sets of all relations of the compared entities are used in the computation.

Table 4.1: Basic types of the relations between entity objects in BioGraph data model.

Relation	**Description**	**Example**
IS EQUAL	Relation representing equality between objects, where object A, on one side of the relation can also be represented as B in general or specific circumstances. The relation can contain details specifying the equality relation.	Protein A IS EQUAL TO Protein B, where two proteins are the same proteins represented using different entity objects not unified at the time of importing.
IS INSTANCE	Relation between objects where one of the objects is an instance of a larger class.	TP53 protein in humans IS INSTANCE of TP53 protein.
IS VARIANT	Representing relation between objects where one object is an isoform of the other.	Antigen A IS VARIANT of Antigen B.
FROM	Describes the connection between an entity object and another entity object that symbolizes its source.	Protein A FROM gene B. Gene C FROM organism D.
CONTAINS	Represents the relation between the object and its part, like a composition in object-oriented programming.	Antigen A CONTAINS Epitope B.
HAS ROLE	Relation between entity objects where one object represents a functional definition of the other entity.	Gene A is HAS ROLE of antigen A1.
RELATED WITH	General relationship between objects. A weight, or relation score, of the relation can be defined in relation parameters.	Gene A is RELATED WITH disease B, with a relation score of 0.9. The relation score parameter is a user-provided relation parameter.
SIMILAR TO	Semantic similarity relation between the objects in a given context.	Gene A is SIMILAR TO gene B, within the context of relations with diseases.

The process of deriving new relations based on the existing relations found in the proposed data model consists of five steps:

1. Selecting a subset of relation types that would be used for computing similarities between selected entity types.
2. Generating relation matrix from the selected relations and connected entities.
3. Applying data mining algorithms to extract new relations.
4. Selecting relevant relations, for example, similarity scores values above a given threshold.
5. Generating similarity relations based on the selected values and storing them in the knowledge graph.

4.2.1 Selecting a subset of relations

Although the most confident similarity results would be obtained by observing all available relation types, extracting the full set of relations and performing computations on it is a highly computationally intensive task, requiring significant resources. Relations between genes, proteins, and antigens would not be significant for evaluating the similarity between genes in the context of similar subsets of related diseases. The choice of the relation subsets should reflect the aimed context of the similarity.

4.2.2 Generating relation matrix

The selected set of relation types contains information about the connected entity types, on both ends and, potentially, a relation weight. All relations of the given types, among the selected types of entities, are collected from the data model and used for creating a vectorized representation of the entities. The vectorization is performed by constructing a matrix with rows representing instances of the entity types, for which the similarity is computed, and columns representing the adjacent entities connected to the selected entities using the relations from the set. The values in the matrix correspond to the relation weights between the row and column entities. The matrix can be seen as a biadjacency matrix, where row entities and column entities represent nodes of a bipartite graph. If the relations do not contain the weight information, the matrix is a binary matrix, where 1 represents the presence of the relation between the row and column entities and 0 otherwise. It can be noticed that such a matrix may become large due to a large number of entities and relations used for the construction, so careful selection of the relations in the first step is essential. The resulting matrix will be in the rest of the text referred to as the relation matrix.

4.2.3 Deriving semantic similarity relations

Direct extraction of the similarity relations between the pairs of objects can be achieved by computing pairwise cosine similarities between the rows of the relation matrix. The semantic similarity measures, defined in the introduction, are also applicable in this case, as the ontology information is implicitly stored in the knowledge graph. However, the relation matrix can be further utilized for deriving new concepts and new semantic relations between the entities and concepts. Different data mining algorithms can uncover different types of semantic relations. A greater focus will be on unsupervised methods, that do not require any additional information besides the data found in the knowledge base.

Deriving semantic similarity relations using clustering

Clustering algorithms organize data objects into groups based on the distances, or similarities, between the objects. The basic properties of good clustering are small distances between the objects inside the same cluster and larger distances between the objects in different clusters. A cluster found in the relation matrix can indicate the presence of a new concept. Membership of an object in a cluster represents the semantic relation between the object and the concept represented by the cluster [10]. New concepts can be added to the knowledge base along with the semantic relations of the objects that belong to the concept. The concept implicitly induces semantic similarity between all entities that are part of the concept.

Nearest neighbor graphs (NNG)

Nearest neighbor graphs are graphs constructed using the distances between the objects and their k-closest neighbors [63]. The relation matrix provides the vector representation of the entities suitable for constructing neighbor graphs using the cosine distance metric. Algorithms for community detection applied to the NNG constructed from the relation matrix can discover densely connected groups of nodes that can represent new concepts. The community detection algorithms are special cases of clustering algorithms, adapted for graphical data.

Deriving relations using association rules mining

Association rules mining is used for deriving common groups of items that are commonly involved in transactions. Applying association rules mining algorithms to the relation matrix is possible by observing entities from one axis of the matrix as the transaction and the entities from the other axis as item entities [76]. For example, the relation matrix consisting of gene rows and disease columns can also be seen as a list of diseases with potentially overlapping sets of related genes. Genes that are commonly found as jointly related to a significant number of diseases uncover new

semantic similarities between the genes. Different sets of genes may be considered interesting depending on the parameters of the algorithm.

Deriving relations using latent semantic analysis (LSA)

As mentioned in the introduction, LSA [24] performs decomposition of the term-document matrix which results in a matrix with a reduced number of dimensions where individual word columns are substituted by the topics found in the documents. The algorithm can be applied to the non-weighted, binary relation matrix, where entities of one axis are analogous to documents and the entities of the other axis are analogous to the terms. In the example of the gene-disease relation matrix, a natural analogy would be the one where the diseases are analogous to documents while genes are analogous to terms. Deriving topics in the relation matrix using LSA discovers new concepts and semantic relations between the genes and the concepts, but also the relations between the concepts and the diseases.

4.2.4 Automated method for deriving semantic similarity relations

A novel method is designed for the automated deriving of new semantic similarity relations, based on data from the BioGraph data model that utilizes clustering and association rules mining to derive new semantic similarity relations. The key property of the approach is the automation of the process that includes extraction of required data from the data model, using the data for deriving new semantic similarity relations using different data mining techniques, and importing the relations back to the model. The approach can be represented as a pipeline starting with stored BioGraph model data and ending as a new set of semantic relations ready to be imported back into the BioGraph data model. The goal of the automated method is to use minimum user inputs at the beginning of the process and automatically analyze stored data and output new semantic similarity relations between the objects. The general pipeline of the automated method consists of 5 steps:

1. context definition;
2. data extraction;
3. data preprocessing;
4. relation deriving;
5. importing relations.

A diagram of the general pipeline is shown in Figure 4.4. The method for automated deriving of new semantic relations can use two approaches for finding the relations – clustering-based and approach based on association rules mining. The

two approaches follow the steps of the general pipeline but have different relation-deriving steps. Clustering-based relation deriving substeps are shown in Figure 4.5, while the substeps for the approach based on the association rules mining are shown in Figure 4.6

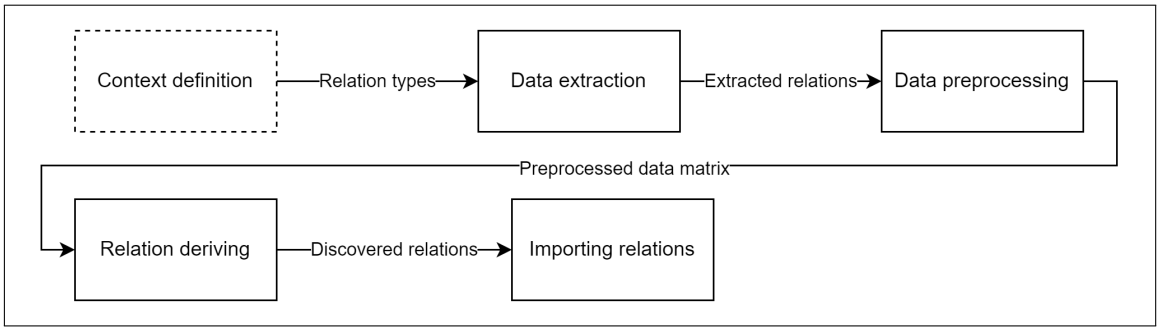

Figure 4.4: General pipeline for automated deriving of new semantic relations.

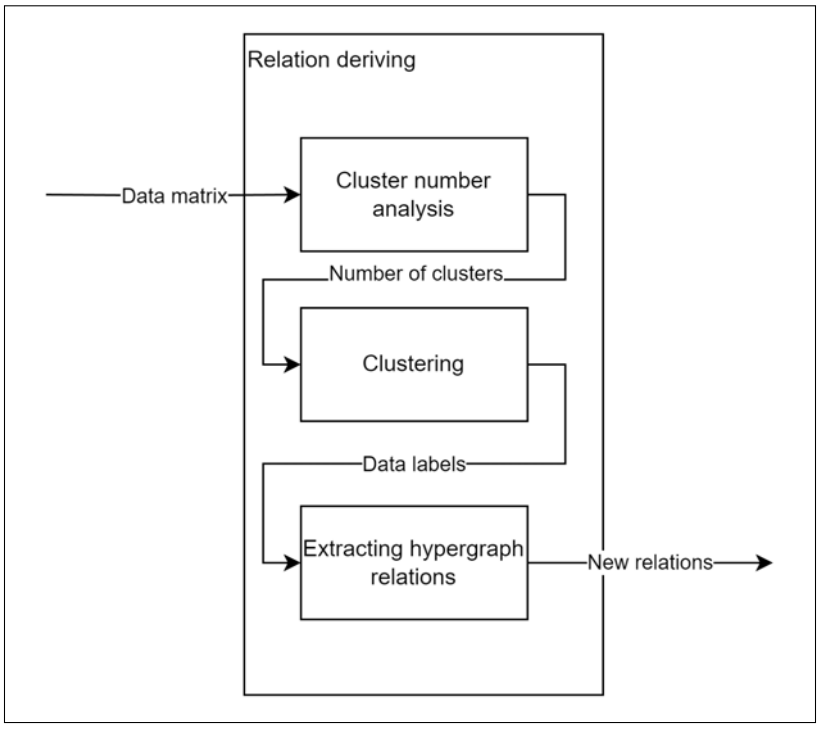

Figure 4.5: Relation deriving substeps of the pipeline for automated deriving of new semantic relations based on clustering method.

Context definition step

Two entity objects can be similar in some contexts, while very dissimilar in others. For this reason, it is essential to define a context in which semantic similarity relations will be sought. Two genes may be similar in the context of mutual involvement in similar sets of diseases. In that case, relations and their weights between gene

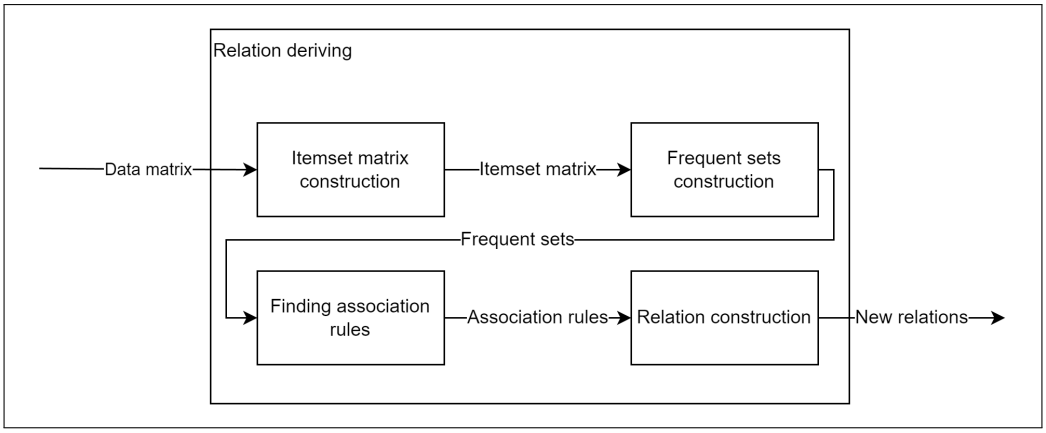

Figure 4.6: Relation deriving substeps of the pipeline for automated deriving of new semantic relations based on association rules mining.

and disease entity objects can be used to evaluate the similarity between genes. In the general case, the similarity between entity objects can be based on a set of relations between entity objects of a given type and entity objects of any other type. The context is defined as a set of relation types that are selected as important for deriving new semantic similarity relations.

Data extraction step

When the context is defined using the relations types on which the similarity will be computed, the data needs to be extracted from the database. Graph-friendly structure of the BioGraph data model enables efficient finding of relations, of the given types. The implemented method for automatic data extraction requires only a list of relation types as input and outputs the list of found relations. An example showing a selection of "RELATED WITH" type relations between genes and diseases, omitting other available relations, is shown in Figure 4.7.

Data preprocessing step

Data preprocessing uses extracted relations to construct a data matrix which will be used as input for the relation deriving step. The data matrix rows represent entity objects between which the relation-deriving algorithm will attempt to find new semantic similarity relations. Data matrix cells contain weights of the relations extracted in the previous step. If the relations don't have explicitly defined weights, or the relation deriving step will use association rule mining, the matrix is a binary matrix where a value of 1 in a cell (i, j) indicates the existence of the relation between objects i and j. Conversion of weighted relations to binary ones is done by thresholding the values, having values above a certain threshold transformed to 1, otherwise to 0. An example of a constructed data matrix based on the selected "RELATED WITH" relations between genes and diseases and the "score" attribute

selected as weights is shown in Figure 4.8.

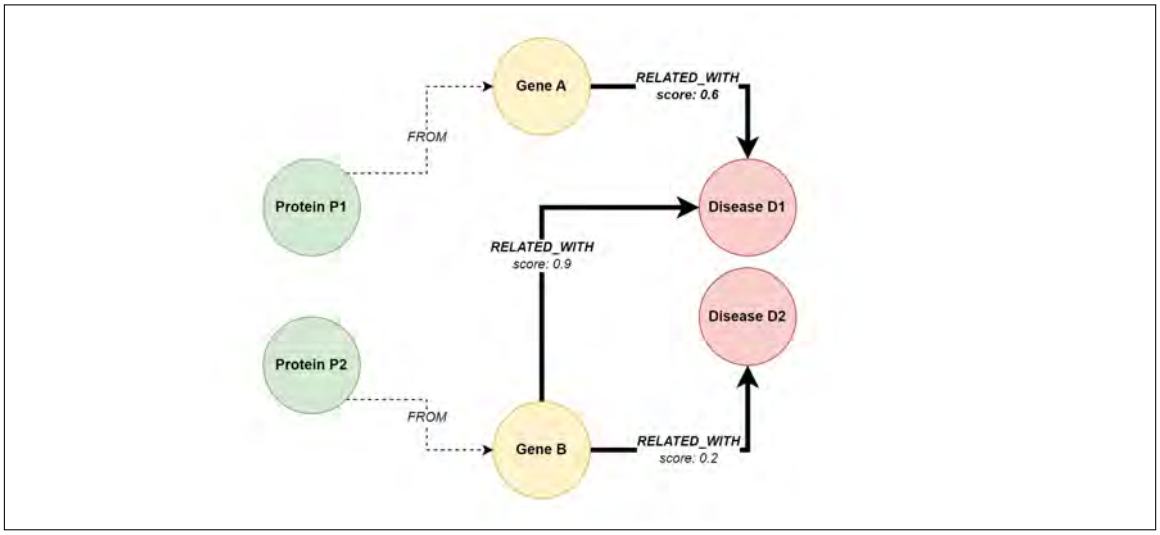

Figure 4.7: Example of selecting relations as inputs for deriving similarity relations. The example shows selecting "RELATED WITH" type relations between genes (black lines) and diseases, omitting the other available relations (dashed lines).

Relation deriving step

In the relation deriving step, the data matrix is used to find new semantic similarity ("IS SIMILAR TO" type) relations in data using clustering or association rules mining method. Direct deriving of semantic similarity relations can be done by computing various similarity measures, appropriate for high-dimensional data, on the rows of the constructed data matrix. However, such results may require user-input parameters that are not always objective enough to testify quality of the relations. That is why this chapter focuses on two general-purpose methods using clustering and association rules mining.

Deriving relations based on clustering

The clustering-based method for relation deriving uses a previously computed data matrix for finding clusters in data. A cluster found in the data matrix composed of the extracted relations represents a hyperedge of a hypergraph connecting all mutually similar entity objects. As most graph database systems do not support hyperedges, the cluster hyperedges are represented using special cluster entity objects connecting all entities linked with the hyperedge. Cluster entity objects are created for each cluster. Parallel overview of clusters, hyperedges, and cluster entity objects are shown in Figure 4.9. The clustering algorithm used in this step is not fixed and may require additional algorithm-specific data preprocessing before it can be used. The number of clusters in the data needs to be estimated and evaluated

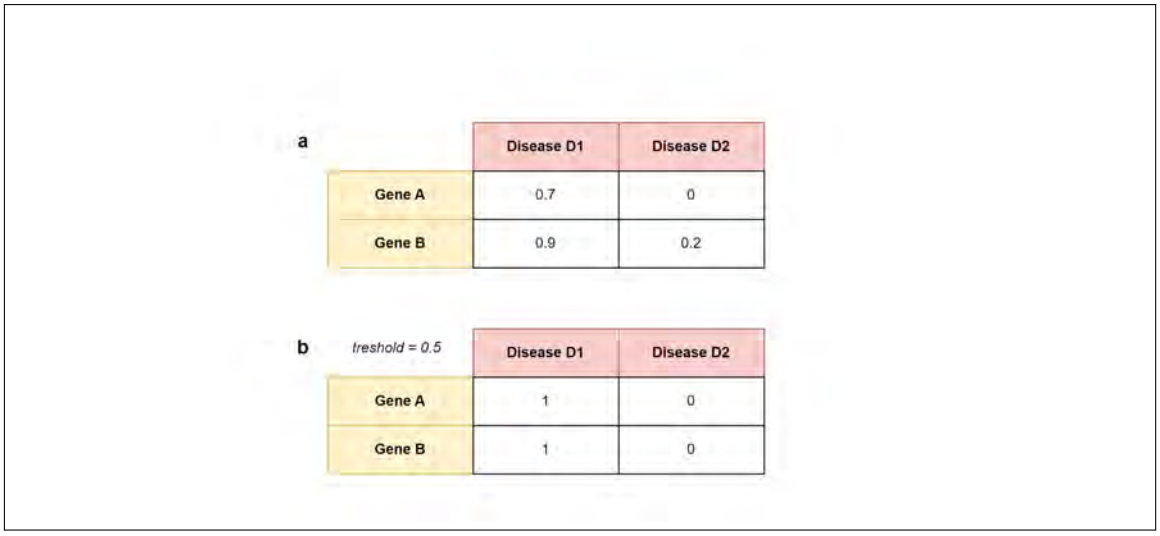

Figure 4.8: Example of constructed data matrix based on the selected relations and their weights. The example shows values of the "score" attribute of "RELATED WITH" relations between genes and diseases, selected as weight (a) and a matrix with thresholded values using a threshold value of 0.5 (b).

and the best value is selected for the final output of cluster labels. For each cluster label a new cluster entity object is created along with the edges connecting the cluster entity with all elements belonging to the cluster.

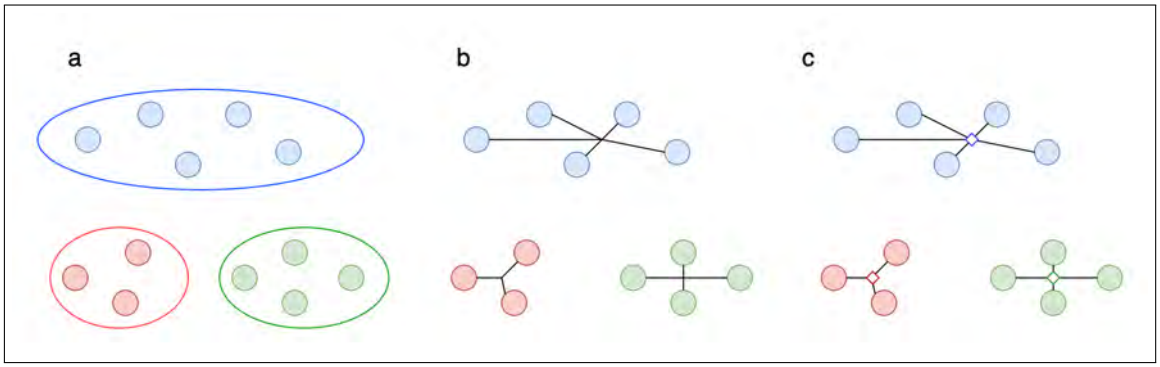

Figure 4.9: Parallel overview of clusters (a), hyperedges (b), and cluster entity objects (c).

Deriving relations based on association rules mining

The method for deriving semantic similarity relations based on association rules mining uses a binary data matrix, constructed in the previous step, and views data as a set of transactions. Depending on the use case, the data matrix may need to be transposed before further analysis. There are no restrictions on which algorithm should be used for finding frequent sets or association rules, but the values of minimum support, lift, and confidence should be provided as user inputs.

The derived association rules follow the structure of $X \longrightarrow Y$, interpreted as if elements of a set X are present then elements from a set Y are also present in the same transaction. These facts can be used to extract semantic similarity relations. Association rules having set X of cardinality 1 reveal semantic similarity relations between an element in X and all elements in Y.

Importing relations

All relations generated in the previous step are transformed and prepared for importing back into the BioGraph data model, or individual analysis using any external software. The new relations can be further combined with the relations already stored in the data model for extracting additional relations using the same proposed pipeline. All derived relations imported back to the BioGraph data model have a ”derived“ attribute set to “true”.

Chapter 5

Model implementation and validation

For the purposes of validating the proposed BioGraph model, a BioGraph software system was implemented. The implementation utilizes the BioGraph data model for unifying metadata from external datasets and semantic search using the semantic relations found in metadata as well as new relations derived by the system. The system collects metadata from the external databases and stores the extracted metadata according to the BioGraph model in graph data storage. Additionally, in order to provide information about data imports and efficient indexing for keyword searches, the data is recorded in a ledger. The implemented system provides data searching methods exposed through an HTTP REST API which processes queries written using an internal query language designed specifically for the BioGraph system. The system is created using Javascript programming language and NodeJS v19 runtime environment [62].

5.1 Software architecture

The BioGraph software system consists of five key components:

- data importers;
- core service;
- indexers;
- graph database adapters;
- HTTP REST API.

5.1.1 Data importers

The entry points for data into the BioGraph system are the data importers. Importers transform data from the original formats of their external databases into

objects in the BioGraph data model. A specialized importer is assigned to each external database. The initial list of importers can be extended to support data from an arbitrary number of external databases. Importers fetch data from the external source by sending API requests, download files, or even scrape Web pages, and load the data for further processing. The entities, identifiers, data nodes, and relations are recognized and extracted from the loaded using methods provided by the core service. The outputs of the importers are arrays of BioGraph data model objects prepared for storing in a local database. The scripts for importing data from all supported databases are listed in the appendix.

5.1.2 Core service

Core service is the central component of the BioGraph system. The role of the core service is to provide the importer service interface for creating BioGraph data model objects and relations and preparing data received from the importers for indexing and recording in the ledger. Additionally, the core service provides data storage services by utilizing database adapters. Core service maintains a transactional way of data inserts in the graph database and the ledger to maintain data consistency. There are six core methods exposed from the core service and provided to the importers for labeling and importing the metadata:

- `beginImport`

  Method for starting new import. The method starts new transactions in both the graph and ledger database and initializes a new import by assigning it a unique identifier.

- `createEntityNode`

  Method for creating entity model object based on the entity type and its primary id.

- `createIdentifierNode`

  Method for creating identifier model object based on the identifier type, title, and value.

- `createDataNode`

  Creating data object of the BioGraph data model. A certain attribute can be noted as a description attribute so the description text value can be indexed with a proper index for keyword searches.

- `createEntityEdge`

  Method for creating relations of a given type, with a given payload, between entity objects.

- `finishImport`

  Method for finalizing the active import. The data is prepared for storage in both the graph and ledger database and the transactions are committed.

5.1.3 Indexers

Indexers provide services for storing and indexing import data in the ledger database. There are four types of indexers:

- import indexer;
- entity indexer;
- identifier indexer;
- description indexer.

The import indexer stores information about the imports, such as the import timestamp or type of importer used for importing the data. The entity indexer logs all entity types and primary identifiers contained within a given import, preventing duplicate entries. The identifier indexer logs all entity identifiers, including the primary identifier, and creates an index structure for efficient entity search based on the identifier values. The description indexer stores text descriptions of the entities using an indexed structure designed for efficient keyword searches. The internal text index was designed as an inverted index, a list of words where individual words refer to a list of documents where they are mentioned.

5.1.4 Database adapters

Multiple database systems have been tested for storing metadata objects. Although the model was successfully mapped to a relation database, querying speeds were not satisfying, the decision was made to use a graph database system as the underlying database. However, as the system was designed as database agnostic, it is possible to use an arbitrary database management system for data storage. The graph database adapters enable the connection between the BioGraph system and arbitrary graph database systems. Each graph adapter exposes the same interface facing the BioGraph system and transforms system requests to specific database implementation calls. For the initial implementation of the BioGraph system, the decision was to use the Neo4J [60] database management system, as it provides high efficiency for performing data storage and retrieval of graph data structures.

A relational database was used as a ledger database to store the import metadata of all objects and relations, preprocessed text metadata for semantic searches, and entity identifiers for quicker entity lookups. Both graph and ledger databases are kept synchronized, and data modification is done exclusively in transaction mode. For

the specific implementation, we used the MySQL relational database management system, but support is provided for connecting different relational databases.

5.1.5 HTTP REST API

HTTP Representational State Transfer (REST) [61] API enables querying the data from the BioGraph system using user-implemented programs. The API supports two types of queries:

- graph queries;
- keyword queries.

The graph queries are queries where the input is a pattern that should be matched from the graph database and the results are represented as subgraphs of the stored knowledge graph. The format of the graph queries follows the internal query language schema.

Keyword queries are queries that fetch entities of a given from the graph database type by matching the given list of keywords against the text descriptions and identifiers of the stored entities. Data indexes are heavily utilized with both types of queries.

5.1.6 Internal query language

To remain database-agnostic, a simple and generic internal query language was designed that can be easily transformed into any of the native languages of the underlying database systems to support easy and efficient searches through metadata following the structure of the proposed model. The search is performed using query objects in JSON [67] format, divided into two segments – "match" and "params". The "match" segment lists the relations between the searched entities, while the "params" segment lists the identifiers and attributes of the searched objects and relations. The query language was inspired by the Cypher [28] query language used with the Neo4J database management system. An example of a query using internal BioGraph query language is shown in Figure 5.1.

5.1.7 Data flows

To give a complete picture of the BioGraph system, Figure 5.2 shows enumerated steps for data transformation and retrieval, from importing data into the graph and ledger database to fetching the query results in the Web user interface. The central group of services, shown in Figure 5.2, enclosed in a blue dashed rectangle, is the core of the BioGraph system. Importers, enclosed in a green dashed rectangle, are

```
{
    "match": [
        "(A:Protein)-[geneProteinRelation:FROM]-(B:Gene)"
    ],
    "params": {
        "A": {
            "data": [
                {
                    "field": "disorder_content",
                    "op": "GTE",
                    "value": 0.9,
                    "isNumber": true
                }
            ]
        }
    }
}
```

Figure 5.1: Example of an internal BioGraph query in JSON format. The query fetches all genes and related proteins where the protein disorder content is 0.9 or higher.

dynamic services, that can be modified to fit any external data source. The first step of the process (step 1) is the collection of raw data from external databases.

The raw data are collected from external data sources (step 1) and a subset of the collected data is labeled and transformed into BioGraph model elements using importers specialized for the selected external data source. The list of importers shown in the figure is not final, it represents the current state of the system and can be easily extended for different data sources. Labeled metadata and relations (step 2) are then sent to the BioGraph core service, which prepares graph nodes and edges based on the BioGraph model elements and sends them to indexers for storing in the ledger database (step 3). Indexers individually log all objects into the ledger database (step 4), preventing duplicate entries. New, non-duplicate, entries are sent to the graph database adapter (step 5) which converts graph elements into storage queries in the native language of the underlying graph database and executes them (step 6). External applications, like BioGraph Web UI, communicate with the BioGraph services using exposed API (step 7). The applications send queries in the form of the internal JSON query language. API sends parsed queries to either the indexers (step 8a), in case of keyword queries, or the graph database adapter (step 8b), in case of graph queries. Graph queries are executed on the graph database (step 9a) while keyword queries are executed on the ledger database (step 9b). New graph database adapters can be implemented for many of the existing graph database management systems.

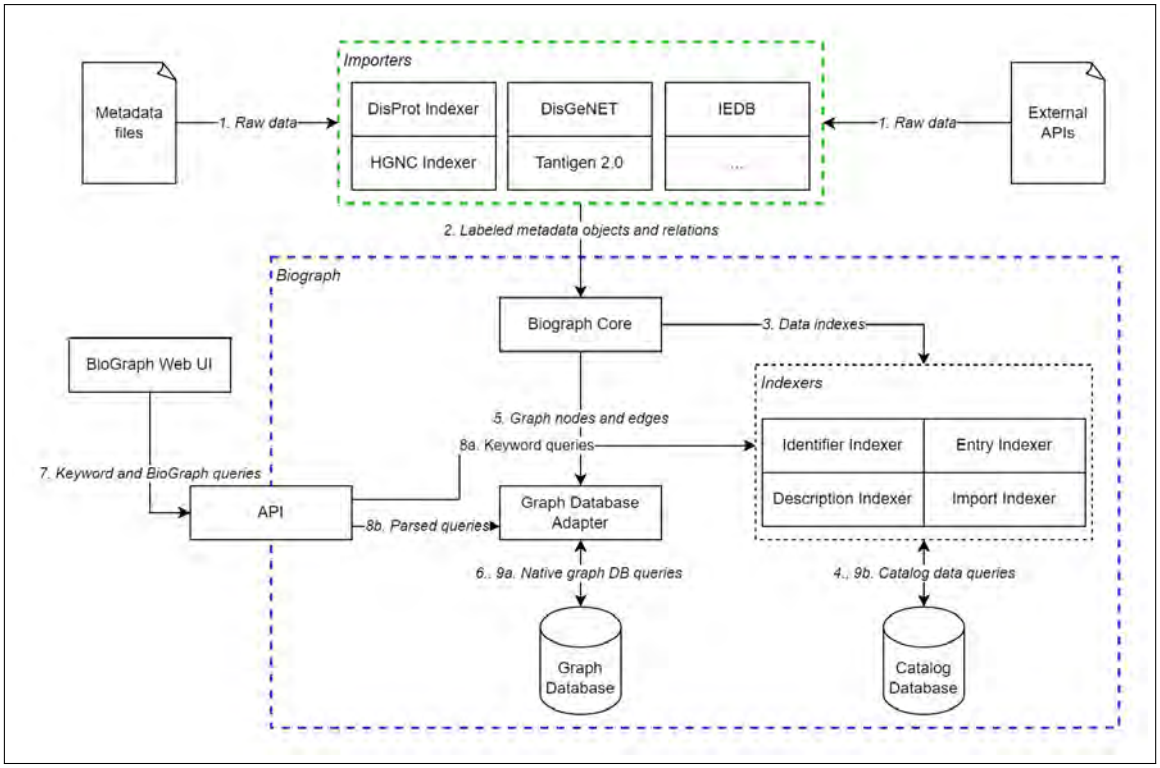

Figure 5.2: Diagram representing the architecture of the system which implements BioGraph data model. The data is downloaded using APIs of the external data sources, FTP access or even scraping the Web pages (step 1). The subset of collected data is labeled within the import service and mapped to BioGraph objects and relations. In the next step (step 2), labeled objects and relations go to the BioGraph Core service where the objects are first logged in the ledger database using indexers (step 3). Identifiers, descriptions, and general import information are indexed individually and stored in the ledger database (step 4). Duplicate entries are skipped, while non-duplicate objects and relations are sent to the graph database adapter (step 5). The graph database adapter transforms objects and relations into native storage queries of the underlying graph database and executes the queries on the graph database (step 6). Storing both graph and ledger data is done in transaction mode, to prevent data inconsistencies.

5.2 Material

For validating the model, we collected metadata from five different data sources:

- DisProt;
- HGNC;
- IEDB;
- Tantigen 2.0;
- DisGeNET.

Metadata from the DisProt database was collected from DisProt API in JSON format, and metadata from the HGNC and DisGeNET databases were downloaded as JSON documents from the website. Metadata from the IEDB database were downloaded as CSV documents while metadata from Tantigen 2.0 database were collected by scraping the website in HTML format. All collected metadata were successfully transformed, connected, and imported into the BioGraph system. The extracted metadata contained more than 16 million model objects, of which more than 2,500,000 individual entity objects, interconnected with more than 21 million relations. An example of the mapping of metadata between gene A1BG and Adenocarcinoma disease from the DisGeNET dataset is shown in Figure 5.3.

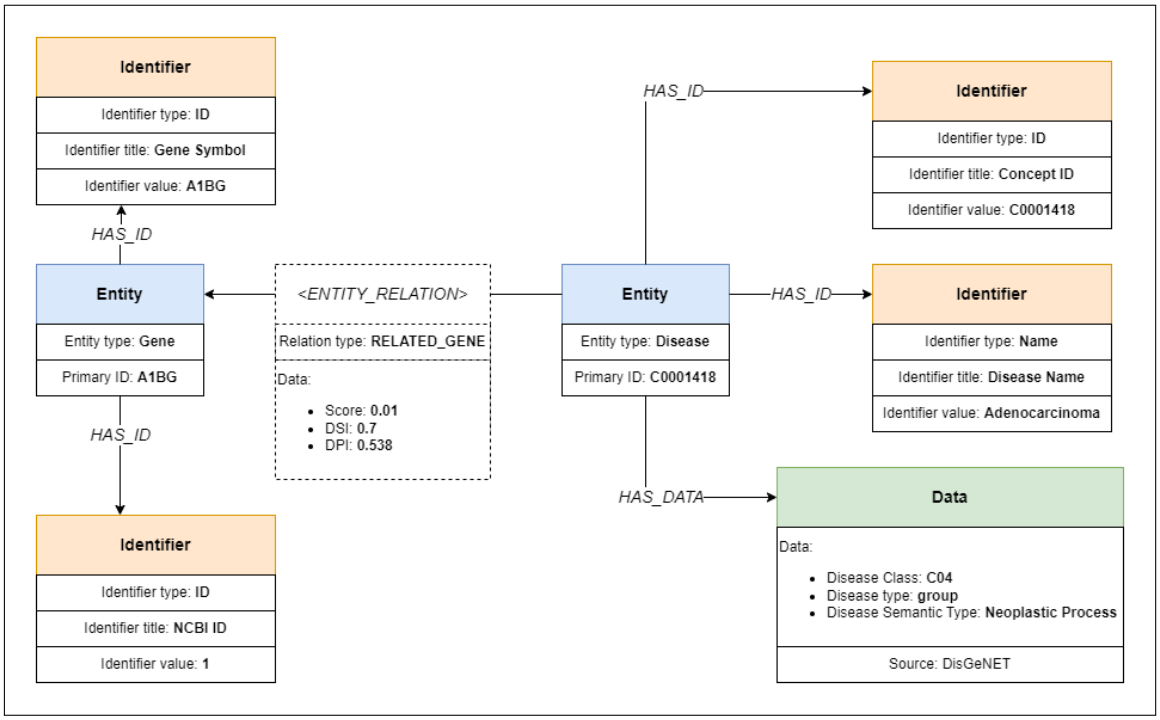

Figure 5.3: Diagram representing metadata from DisGeNET dataset record mapped to BioGraph model.

5.2.1 DisProt dataset

DisProt (Database of Disordered Proteins) dataset contains data on intrinsically disordered proteins from different species. The dataset currently consists of over 2,300 protein entries, containing information on:

- protein sequence;
- entry curator information;
- identifiers in DisProt and UniProt dataset;

- source organism taxonomy information;
- source genes;
- computed disorder content percentage;
- structural annotations and evidence for the annotations in the form of references to research publications confirming the annotations.

The importer dedicated to processing metadata from the DisProt dataset collects protein identifiers, disorder content information, source organism taxons, and genes related to proteins. Gene data contains primary symbols of the genes but also the alias symbols, which are all included by the importer. The data from the DisProt dataset is collected as a complete dataset in JSON format. The current size of the dataset in JSON format (version 2023_06) is 19.1 MB, containing data on 2649 proteins. Disordered regions were not included in the metadata imported with the current version of DisProt importer but there are no technical restrictions for adding the support for disordered regions in the following updates.

5.2.2 HGNC dataset

The HGNC (HUGO Gene Nomenclature Committee, where HUGO stands for Human Genome Organization) dataset contains information on gene symbols, identifiers, gene chromosome locations, and their respective protein references for all known human genes. The current dataset (data available on 08.2023) contains 43,621 entries of human genes and also includes the identifiers of the ortholog genes found in mouse and rat genomes. Our importer collects all available identifiers, including the orthologs, and UniProt protein references, and associates them with the proper organism taxons.

5.2.3 IEDB dataset

IEDB (Immune Epitope Database) dataset contains data on immune epitopes from different organisms. Each epitope is associated with its source antigen, including identifier references to both protein and non-protein antigens, and source organism taxon. Although the dataset contains more information, like those related to diseases, we decided for our model verification to use only the subset of information containing the relations between epitopes and antigens. The dataset is downloaded by the importer from the website https://www.iedb.org/database_export_v3.php in TSV format. The size of the data depends on the data that is present in the database at the time of download.

5.2.4 Tantigen 2.0 dataset

Tantigen 2.0 dataset contains data on human tumor antigens containing HLA ligands and immune T-cell epitopes. The data contains:

- antigen accession ID;
- antigen sequence;
- protein reference from UniProt dataset;
- source gene reference from NCBI data set;
- antigen full name and synonyms;
- links to the antigen mutation and isoform entries;
- list of T-cell epitopes and HLA ligands of the antigen.

The database website does not provide API access or downloadable files. The importer scrapes the content of the website and transforms the collected HTML pages into JSON documents as a preprocessing step. The information collected from the Tantigen 2.0 and transformed into BioGraph model objects includes all available information except the antigen sequences. The current total number of antigen entries in the Tantigen 2.0 dataset (valid version date 09 March 2017) is 4,297.

5.2.5 DisGeNET dataset

DisGeNet (Disease Gene Network) dataset contains information on the relations between genes and human diseases collected from multiple sources. The dataset consists of 1,134,942 gene-disease relation entries. Each relation contains information on:

- gene symbol;
- disease concept identifier;
- dPI - Disease pleiotropy index;
- dSI - Disease specificity index;
- disGeNET gene-disease association score.

The data are downloaded as a TSV file by the importer. All the information available in the file is collected by the importer and transformed into BioGraph model objects.

5.3 Deriving new semantic similarity relations in BioGraph data model

The pipeline for deriving new semantic similarity relations, described in the previous chapter, is implemented following both the clustering-based method and the method based on association rules mining. The spectral clustering algorithm was selected for the clustering-based method, as it provides good flexibility for clusters of arbitrary shapes and is a natural choice when dealing with graph data. The input for spectral clustering is a graph, so a neighbor graph was computed using the data matrix. For each entity object, k nearest neighbors were found and connected in the neighbor graph. As the data matrix is a sparse matrix, cosine similarity was used as a proximity measure. The resulting graph was represented using the affinity matrix.

The affinity matrix was then used for computing the graph Laplacian matrix for which the eigenvalues and eigenvectors were computed. The eigenvectors were sorted by their respective eigenvalues in ascending order and used as inputs for the K-means clustering algorithm. To estimate the number of clusters, the elbow method was used with the Sum Squared Error (SSE) measure. Finally, the K-means algorithm with the selected optimal number of clusters was used to label data objects. Cluster labels were used for creating cluster entity objects with relations to all data objects within the respective cluster. Implemented clustering method substeps of the relation deriving step are shown in Figure 5.4. Another clustering algorithm that can be used for deriving semantic relations within hierarchical data is agglomerative clustering.

The relation deriving based on association rules mining starts with a data matrix

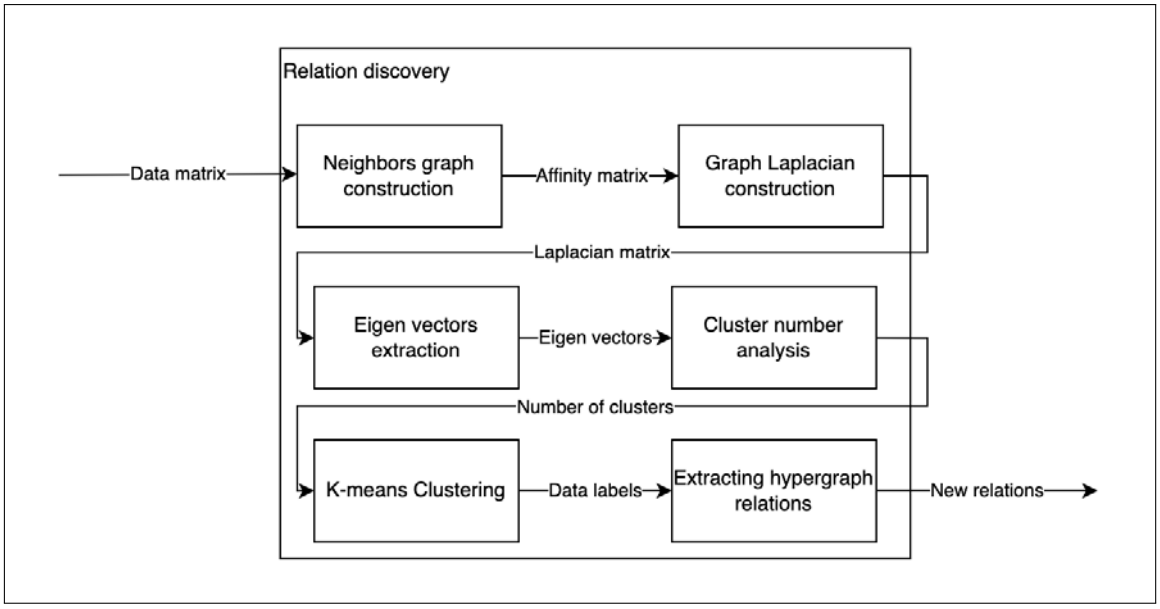

Figure 5.4: Diagram representing spectral clustering substeps for relation deriving.

which is transformed into an itemset matrix. The set of frequent items was computed using the FP-Growth algorithm. Association rules were then extracted from the frequent set using selected minimum support, confidence, and lift parameters.

All steps of the pipeline were implemented as individual Python language scripts. The outputs of each script were written in temporary output files and the parameters were set as command-line arguments. All scripts were listed in the particular order as a pipeline in a shell script.

5.4 User interface

Users with technical experience can write their own applications to submit queries to BioGraph and use BioGraph as a backend system. Even though the integrated query language is simple and intuitive for users with technical backgrounds, it still may not be friendly enough for users with little or no technical experience. That is why we have also implemented a Web-based graphical user interface, developed using ReactJS [71] framework, that communicates with BioGraph and can be used to create graph and keyword queries using the graphical interface. It enables users to select predefined entity types and relations between the selected entities to draw a pattern that will be matched against the metadata graph. Graphical queries allow users without significant technical experience to intuitively create complex queries and discover indirect links between entities. Users can further explore the metadata of the entities from the query results and navigate through the graph following relations to neighboring entities. Additionally, the Web interface offers example queries and a help section to enable quicker onboarding of new users and a better understanding of the data retrieval process using graphical queries. All executed queries and results can be exported to files for further reuse and analysis. Queries can be exported in JSON, while the results can be exported in CSV format.

An example of a graphical query and the query results can be seen in Figure 5.5. The graphical query describes a pattern connecting genes and related diseases with a relation score greater than 0.5, where the genes are also related to proteins with a disorder content percentage greater than 0.9. The relation score between the gene and the disease is the DisGeNET gene-disease association score [68]. Additionally, the requirement was that genes are also tumor antigens, and to match all epitopes related to those tumor antigens. All matched patterns for the given graphical query are shown in Figure 5.6 displays. As the patterns are often not linear, the result patterns were decomposed into linear disjoint paths. The results of the query are shown in Figure 5.6, where the first results match the example shown in Figure 4.2.

Writing this type of query can be challenging in native database languages while drawing a graphical pattern that the query should match is easy and intuitive.

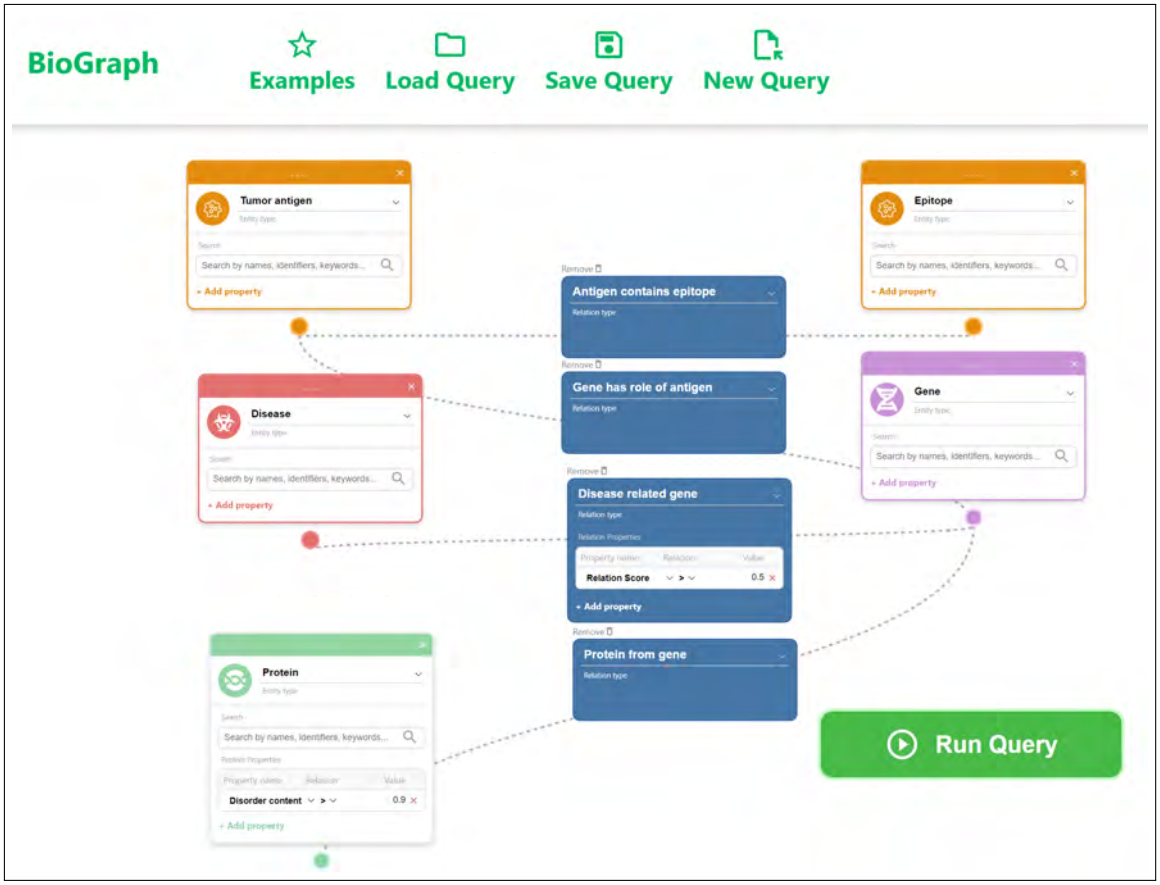

Figure 5.5: Example of a graphical query drawn using BioGraph Web interface. The data is fetched from imported metadata from the currently supported data sources. The query attempts to find links between Diseases and Genes with a DisGeNET relation score of 0.5 and greater, where genes are transcribed into proteins, with disorder content of 0.9 or higher and genes are also tumor antigens. Protein metadata comes from the DisProt dataset, gene metadata comes from the HGNC dataset, and disease metadata comes from the DisGeNet database. Epitopes, from Tantigen 2.0 and IEDB are also fetched for the matched tumor antigens. The query pattern matches the structure of the example shown in Figure 4.2

.

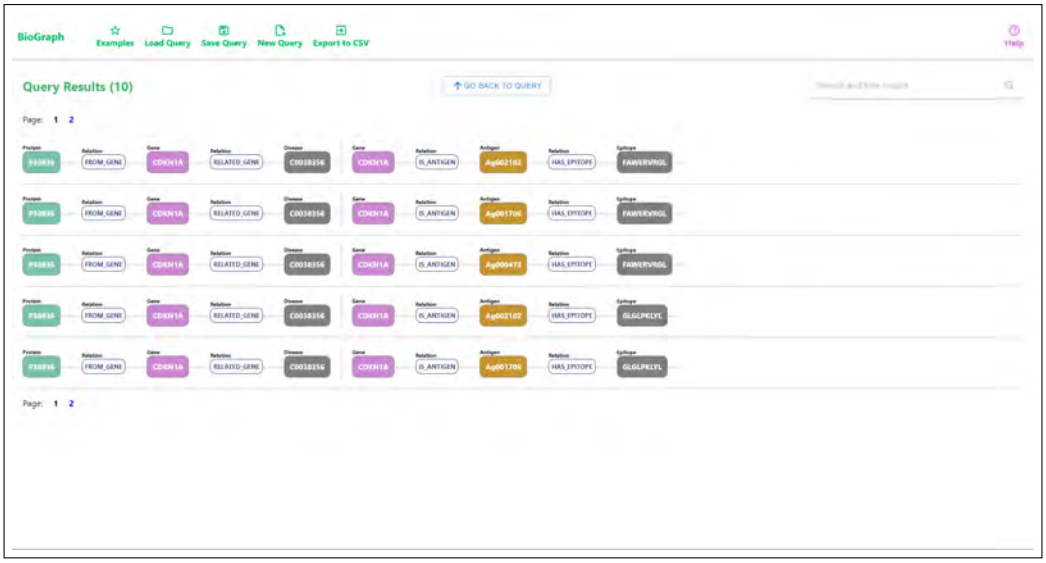

Figure 5.6: List of results received when querying the genes which are transcribed into highly disordered proteins (disorder content greater than or equal to 0.9) and linked with diseases with DisGeNET relation score of 0.5 and greater. It was requested that the genes in the results are also tumor antigens and fetch all epitopes related to the antigens. The results are displayed as matched paths in the knowledge graph. The example shown in Figure 4.2 matches the first result in the list.

The results show paths from the matched patterns. Selecting any node from any path displays details about the specific entity and lists all the connected data and identifiers from all datasets. By following the relations listed in the entity, the user can easily traverse the graph and track the relations between the entities. An example of details for a specific gene is displayed in Figure 5.7. Besides using pattern matching, entities can also be searched using keyword searches through input fields located in entity nodes. An example of searching disease entity matching the keyword “pneumonia” is displayed in Figure 5.8.

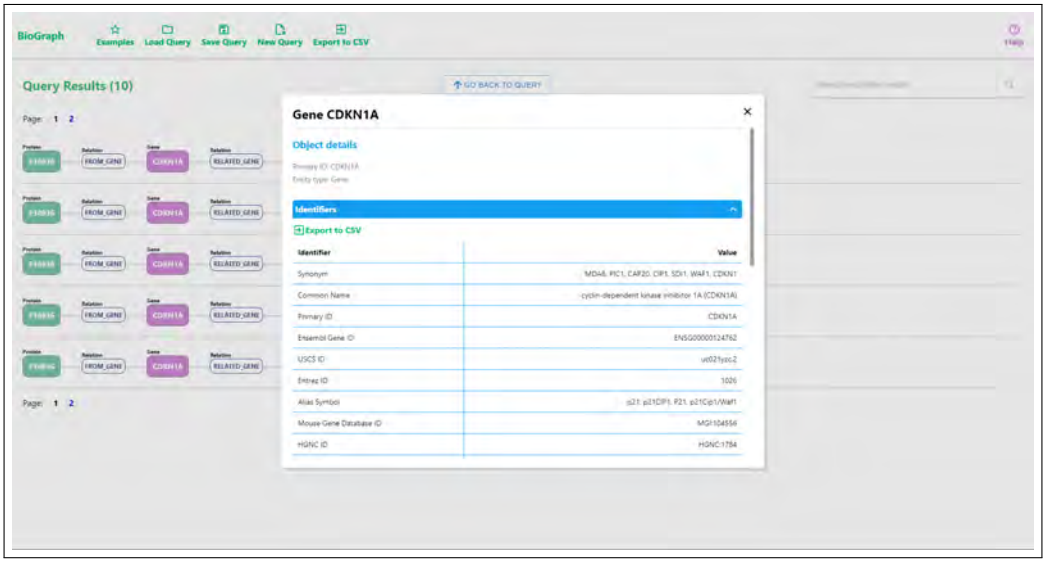

Figure 5.7: Details of a single gene entity, displaying information collected from multiple data sources.

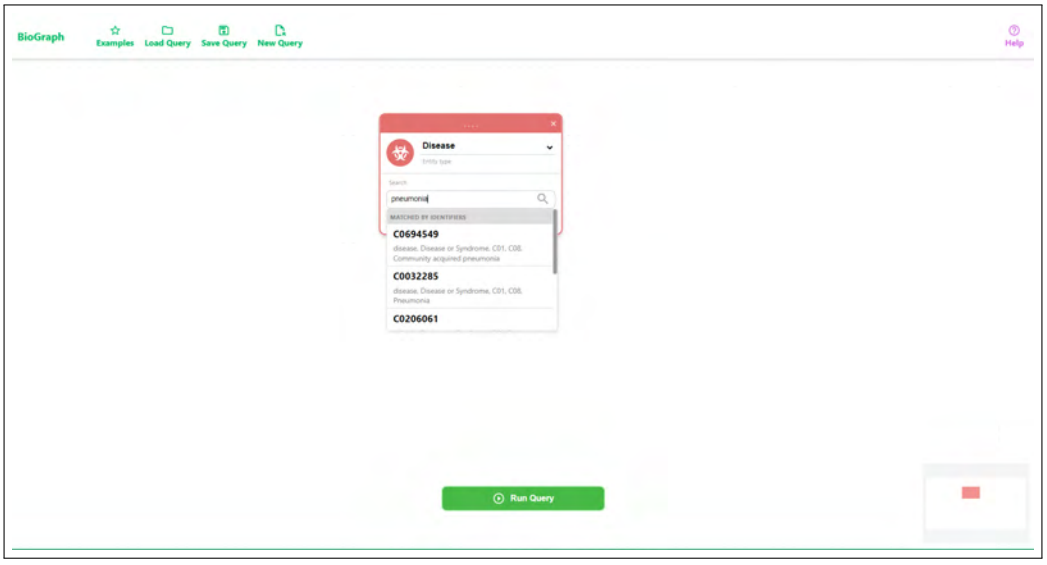

Figure 5.8: Searching diseases using keyword “pneumonia”.

Chapter 6

Results and discussion

The thesis presents a formal definition of the novel BioGraph data model, a proposed software architecture that enables the implementation of a system based on the data model, and a generalized automatic pipeline for extracting new semantic similarity relations between objects from the proposed data model.

The described data model is a general-purpose data model, suitable for various use cases, but primarily designed for biological data. It enables the unification of heterogeneous data from various external sources as well as a unified semantic search over the unified data. The model specifies three core object types that form the backbone of the model which can be used to represent objects and their relations from an arbitrary domain. All domain-specific definitions are specified in higher-level schemas without modification of the model or the search functions. The model's ability to represent heterogeneous data was tested by unifying data from five different databases - DisProt, HGNC, IEDB, Tantigen 2.0, and DisGeNET, testifying to the flexibility and expressiveness of the model. The resulting graph included more than 17 million nodes of which 2.5 individual biological entities with over 21 million relationships. The model specification and the system architecture proposal were published in [88] and [83] and also presented at [86], and [87]

The system was designed as a modular, database-agnostic system easily adaptable to any underlying database management system. The list of importers used for data loading is extensible, allowing the system to support importing data from any data source by adding new import scripts using only several methods exposed from the core library. Content addressing and multiple indices indexing enables the detection of duplicate data and quick data searching using complex graph queries. The keyword queries utilize a reverse lookup index to quickly retrieve identifiers matching partial query strings.

Discovering semantically similar objects in big datasets is a difficult challenge, due to the large volume of data and the requirement for using or writing custom software for a specific data analysis. On the other side, being able to find new relations in the base knowledge provides the ability to extract new knowledge which reveals hidden

patterns in data and gives a wider context to the domain. The pipeline for the automated discovery of new semantic similarity relations, introduced in this thesis, is an unsupervised process that finds new semantic similarity relations in data stored in the BioGraph data model. It is implemented in a way that requires only a small number of parameters, depending on the selected analysis algorithm. The output of the pipeline is a set of relations and objects prepared for importing back into the data model and ready to be used in a new analysis.

6.1 Comparison with the existing data unification and querying systems

For any software system to be accepted by the research community, the availability of the system is the key property. Open-source and free-to-use systems are reaching a wider audience and providing their utility at a much higher degree than payware systems. On the other hand, deploying a sophisticated system on a platform available to researchers worldwide is unfeasible without any financial support. The decision was to maintain the free availability of the system and its open-source properties by enabling the users to deploy the system components locally on their machines easily. The source code for the system can be found at https://github.com/aleksandar-veljkovic/biograph and the deployed version of the BioGraph Web UI can be found at http://andromeda.matf.bg.ac.rs:54321/. The Monarch [77] and ROBOKOP [12] systems also follow the idea of open-source availability, while the Elsevier Biology graph [25] is a closed-source software.

The following important property for the broader adoption of a software system in the research community is simplicity. The intention was to create a simple, easy-to-use system that allows users with almost no technical know-how to express their data searching requests in the simplest possible way, enabled by the simplicity of the underlying data model. Simple data searching is enabled by graphical queries, where users draw the patterns between entities while also having the ability to run keyword searches to find specific entities that should be included in the search. Monarch allows users to run keyword searches but not pattern queries. ROBOKOP system allows drawing queries in graphical form but in a generalized and unintuitive way, making the system's usage difficult for regular users. GeneCards.org [29] provides keyword-based search tools and does not support matching patterns between multiple entities. It also has support for queries using natural language. The decision was not to support natural language queries in the initial version of the querying interface. Natural language queries, while being easy to use by the users, introduce unnecessary ambiguity in queries and uncertainty in results.

Extensibility is another key property. The easy extensibility of the data model and the overall software system is necessary for continuous development and upgrades of

the system, not only by the authors but also by the research community members. The proposed system and data model, being open-source, along with the easy addition of new import scripts, allow users to develop new scripts for importing data from countless data sources. Besides developing import scripts and other upgrades for themselves, developers can share their upgrades with other community members worldwide and further expand the adoption. Both Monarch and ROBOKOP, as open-source systems, enable the extensibility of their systems.

It is often the case that the results of one research are inputs for the others. That is why saving the search results is also a feature of our system. Besides saving the results, the BioGraph system also allows saving and loading queries, so the researchers can share their queries and run them on different machines instead of transferring the results, which are orders of magnitude larger than the queries. ROBOKOP system also allows saving and loading queries from files, while Monarch does not provide those features.

A comparison between existing solutions and the BioGraph model, with the corresponding querying system, is presented in Table 6.1.

Table 6.1: Comparison between the BioGraph system and currently existing solutions. The BioGraph system fulfills all the given criteria, except natural language queries when compared to similar systems. The question marks in the table represent the states where the criteria could not be evaluated.

Criterion	**BioGraph**	**ROBOKOP**	**Monarch**	**GeneCards**	**BKG[1]**
Open-source	✓	✓	✓	✗	✗
Local deployment	✓	✓	✓	✗	✗
Pattern querying	✓	✓	✗	✗	?
Graphical queries	✓	✓	✗	✗	?
Extensibility	✓	✗	✗	?	?
Natural Language Queries	✗	✓	✗	✗	?
Loading and Storing Queries	✓	✓	✗	✗	?
User-friendly	✓	✗	✓	✓	?

[1]Elsevier Biology Knowledge Graph

6.2 Advantages and disadvantages

The BioGraph system presents a compelling framework for semantic data unification in the domain of bioinformatics, offering several notable advantages, as well as encountering certain challenges. One of the primary benefits of the BioGraph system lies in its capacity to unify heterogeneous data sources, a longstanding issue in bioinformatics. Structuring the metadata within a knowledge graph, the system enables the integration and normalization of information from diverse bioinformatics databases. This property mitigates issues created using disconnected data silos and enables a more holistic understanding of biological systems. The result is heightened research efficiency, as it enables the analysis of data originating from a multitude of sources, thereby providing deeper insights. The BioGraph UI application provides a seamless querying functionality for metadata across various bioinformatics databases. Researchers can access and retrieve metadata without the need to learn the intricacies of each database's interface, thus significantly streamlining the research process. Automated process for deriving new semantic similarity relations enables discovery of new knowledge with minimal user intervention, hidden in large heterogeneous datasets.

However, the flexibility and generality of the BioGraph system come with a cost. Adding metadata from a new database requires implementing a new import script, created specifically for the new database without the ability to reuse the script in general cases for additional databases. Luckily, import scripts are easily implemented with the help of simple API exposed by the core service. Another challenge of the BioGraph system is the time required for loading the initial data, which depends on the hardware resources of the machine that performs the import and can take several hours to complete. Similar time requirements apply for the automated deriving of new relations, especially when a large set of the existing relations is selected for deriving new semantic similarity relations. Although the system is designed as database-agnostic, different implementations of the system on different database systems may require careful analysis of the choice of indexing techniques and attributes, as the efficiency of the querying system heavily relies on proper indexing and data joining and traversal methods.

6.3 Examples of biomedical applications

6.3.1 Genes related to Parkinson's disease

To show the usefulness of the BioGraph model for researchers in the field of bioinformatics the following example will describe the process of finding all genes that are closely related to Parkinson's disease [65] using the BioGraph web interface. The gene data is extracted from the HGNC [75] database and diseases along with their

associations with the genes are extracted from the DisGeNET [68] database. A user can query the system in a unified manner, without the knowledge of the specific database from which the data originates. The scripts used for importing data from the two databases into the BioGraph data model are listed in the appendix.

When the BioGraph web interface is loaded, the user is presented with a blank canvas where a graph query should be created. The start screen with a blank canvas is shown in Figure 6.1.

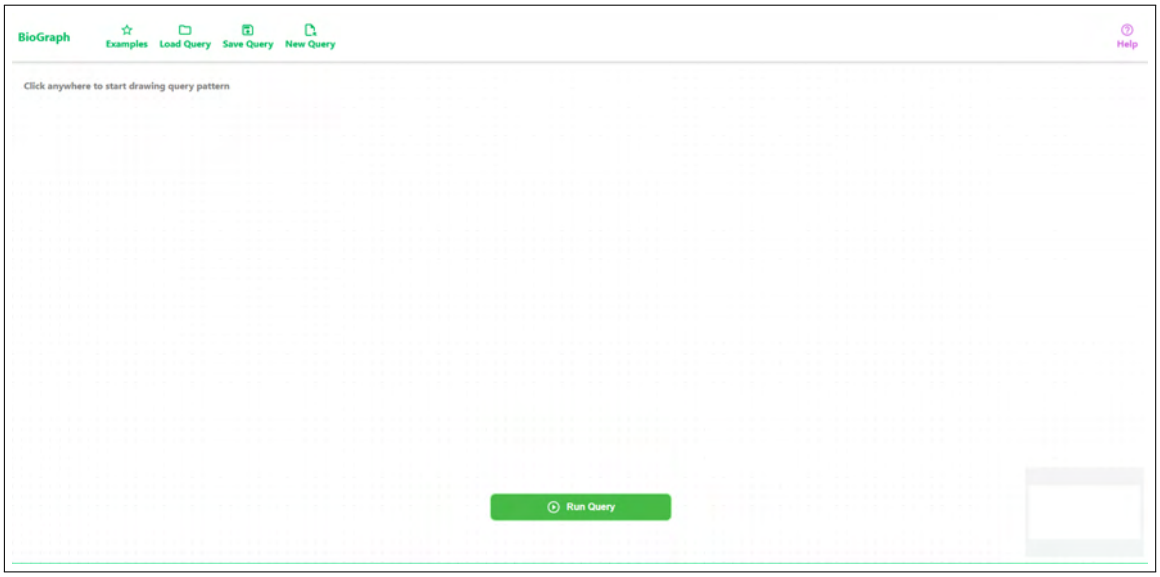

Figure 6.1: Blank canvas of the BioGraph Web interface waiting for the user to design a graph query.

A user can create nodes of the graph query by clicking on the canvas using a left mouse click. As a result, a generic node will appear on the canvas waiting for the user to select the entity type that the node will represent. For this example, the first node to be created will be a disease entity node that will represent Parkinson's disease. The steps required for the user to create the disease entity node are shown in Figure 6.2.

In this state, the created disease node represents any disease found in any of the underlying databases. To specify that the node represents Parkinson's disease, the user can enter the name of the disease in the search field or directly list the disease id in the node properties. When the user enters even the part of the disease name in the search input field of the disease node, a list of found diseases matching the keyword query appears under the search input field. The user then selects the item in the list and the disease node now represents only the selected disease. An example of the process of associating the disease node to a specific disease is shown in Figure 6.3.

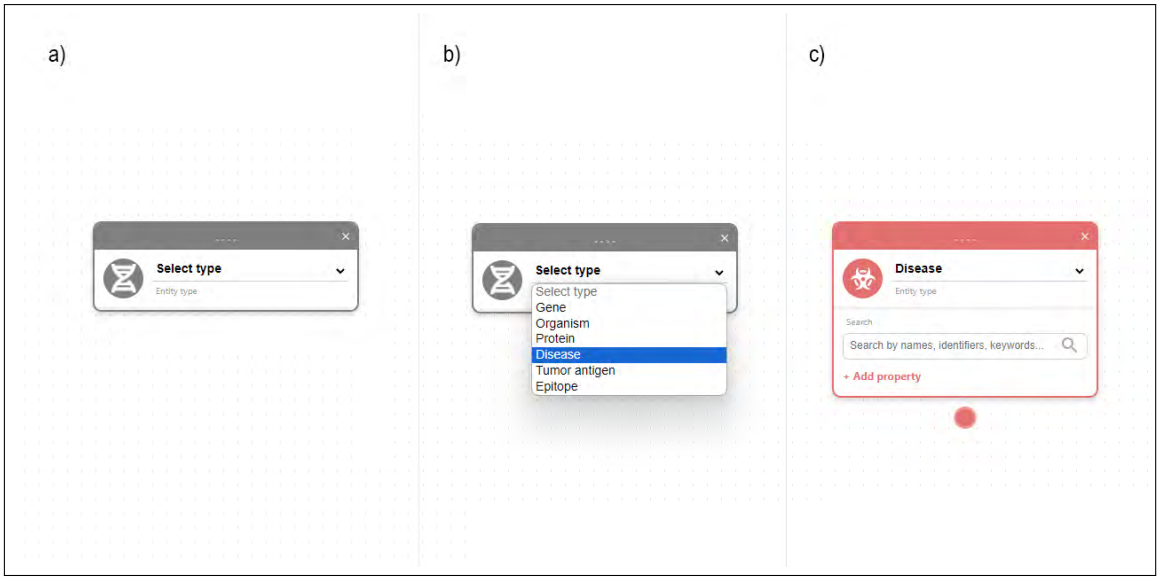

Figure 6.2: Steps required for the user to create the disease entity node. The user clicks on the blank part of the canvas using a left mouse click and a generic node appears (a). The user then selects the required entity type from the "Entity type" dropdown of the generic node (b), in this case, the "Disease" type. Finally, the generic node is transformed into a disease entity node (c)

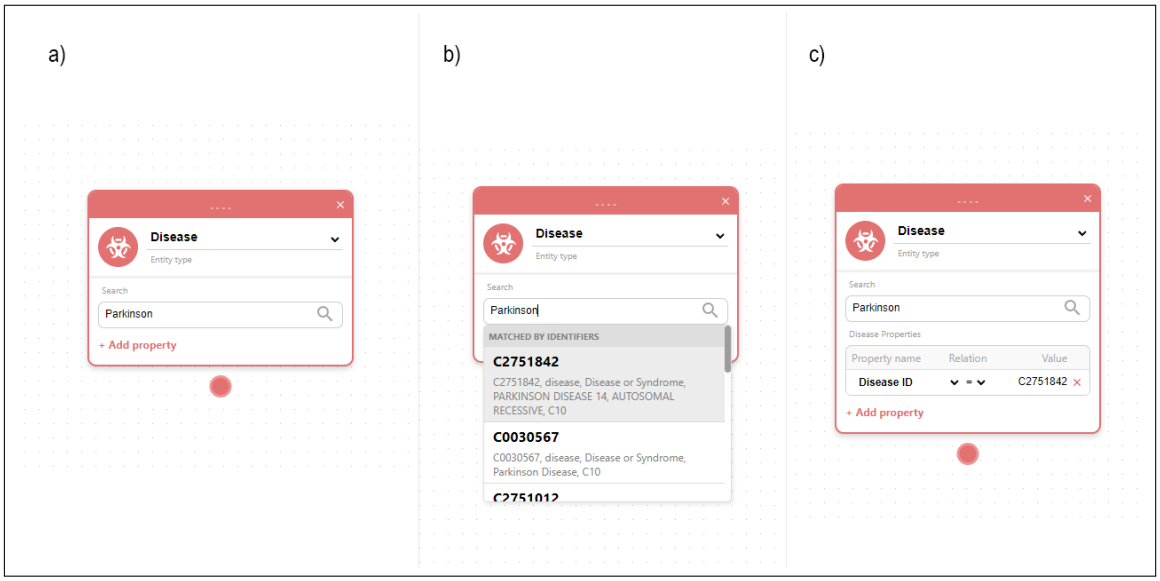

Figure 6.3: Example of the process of associating the disease node to a specific disease. The user inputs part of the disease name into the search input field, in this case, the word "Parkinson" (a). A list of matched diseases with their identifiers is shown in the list below the input field (b). The user selects the item on the list and the node is now associated with Parkinson's disease (c).

When the disease node is associated with Parkinson's disease, the next step is getting genes associated with the disease. To do that, the user creates a gene entity node following the same flow described for the disease entity node, with the

exception that the entity type is now gene. An example of this process is shown in Figure 6.4. The user does not know which genes will be represented by the node as that is to be answered by the query results, no identifier is specified for the gene node.

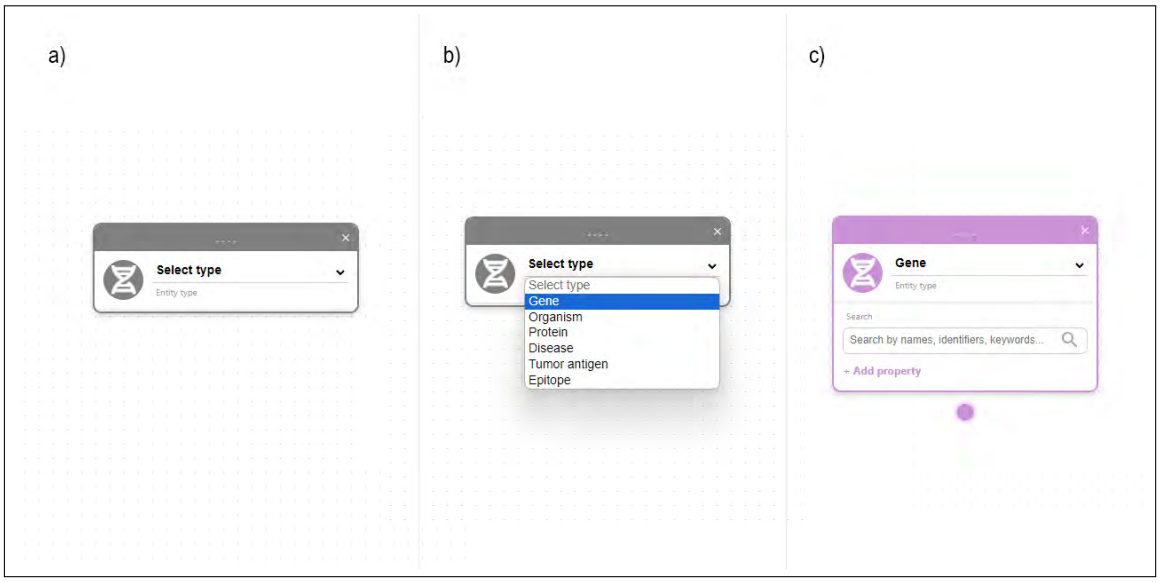

Figure 6.4: Steps required for the user to create the gene entity node. The user clicks on the blank part of the canvas using a left mouse click and a generic node appears (a). The user then selects the required entity type from the "Entity type" dropdown of the generic node (b), in this case, the "Gene" type. Finally, the generic node is transformed into a gene entity node (c)

The requested association between the genes and the disease needs to be explicitly listed in the query. In the case of this example, the query should return all genes that have a high association score with Parkinson's disease. To do this, the user connects the gene and the disease node using a proper relation. The relation between the nodes is created by holding a left mouse click over the relation handle circle below the one entity node and dragging it to the relation circle below the other entity node. The generic relation will appear between the two nodes. The user selects the relation type from the relation's dropdown list. In this case, the relation type will be "Disease gene relation". This relation type has a parameter "Relation score" which indicates the strength of the association between the genes and the disease expressed as the DisGeNET score. The input parameters for the relation properties are displayed by clicking the "Add property" label below the relation type dropdown. For this example, the user sets the relation score property to be greater than or equal to 0.7. The relation score value in this case corresponds to the DisGeNET association score between genes and diseases. An example showing the creation of the relations between the gene and disease nodes is shown in Figure 6.5

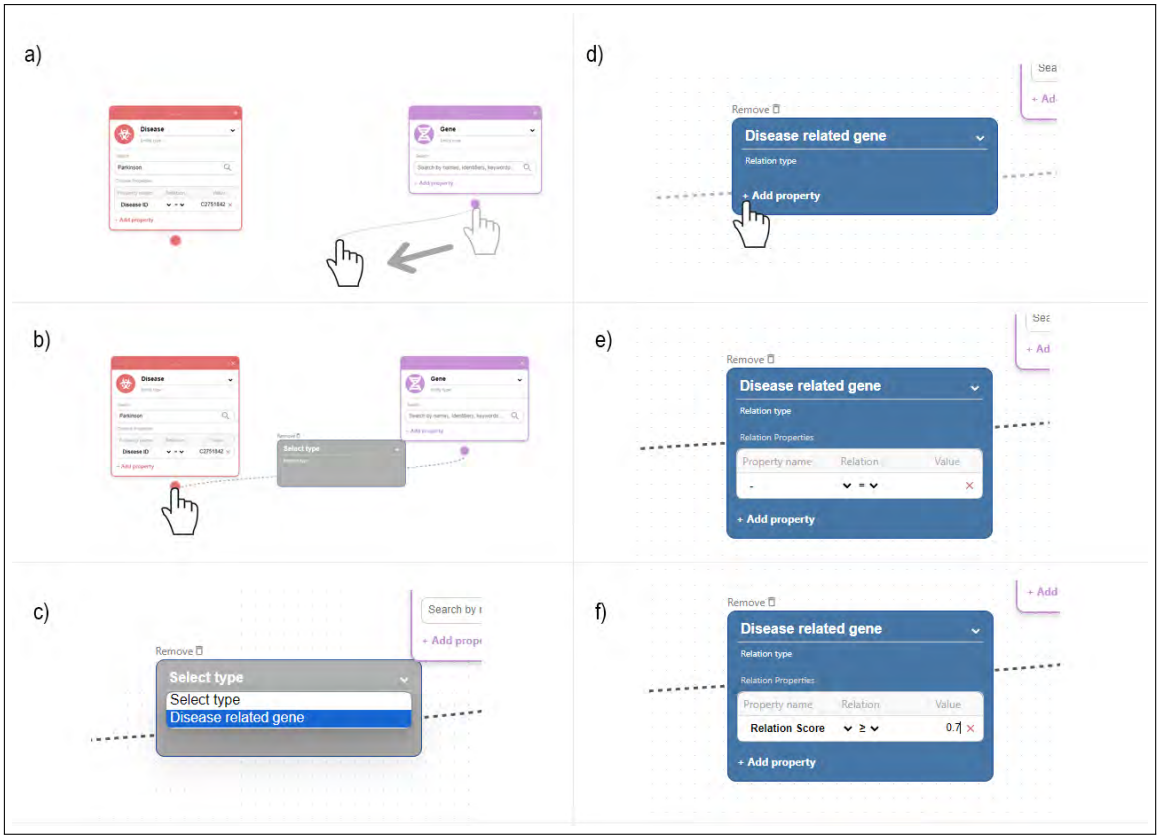

Figure 6.5: Creating relation between gene and disease nodes. The user creates the relation by dragging the relation line from the relation circle handle of one node and dropping it on the circle relation handle of the other node (a and b). The user then selects the relation type (c). Relation properties are added by clicking on the "Add property" label (d) and the property list inputs are displayed (e). The user selects from the inputs the "Relation score" property, the DisGeNET association score between genes and diseases, and sets it to greater than or equal to 0.7.

The constructed query is now ready to be executed by clicking the "Run query" button located at the bottom of the screen. The final query is displayed in Figure 6.6. The results of the query are shown in the table below the query and, for this example, they include one specific result – a gene highly associated with Parkinson's disease. The result shows that the gene highly associated with Parkinson's disease is PLA2G6. The validity of this result is verified in [78]. Clicking on the label PLA2G6-9606 in the results, where the suffix "-9606" signals that the gene came from the Homo Sapiens organism (taxon identifier 9606), more details about the gene PLA2G6 are displayed. Figure 6.7 shows the results table for the given query and details of the PLA2G6 gene. The gene details include all identifiers and metadata collected from all available databases as well as the relations between the PLA2G6 and other entity objects.

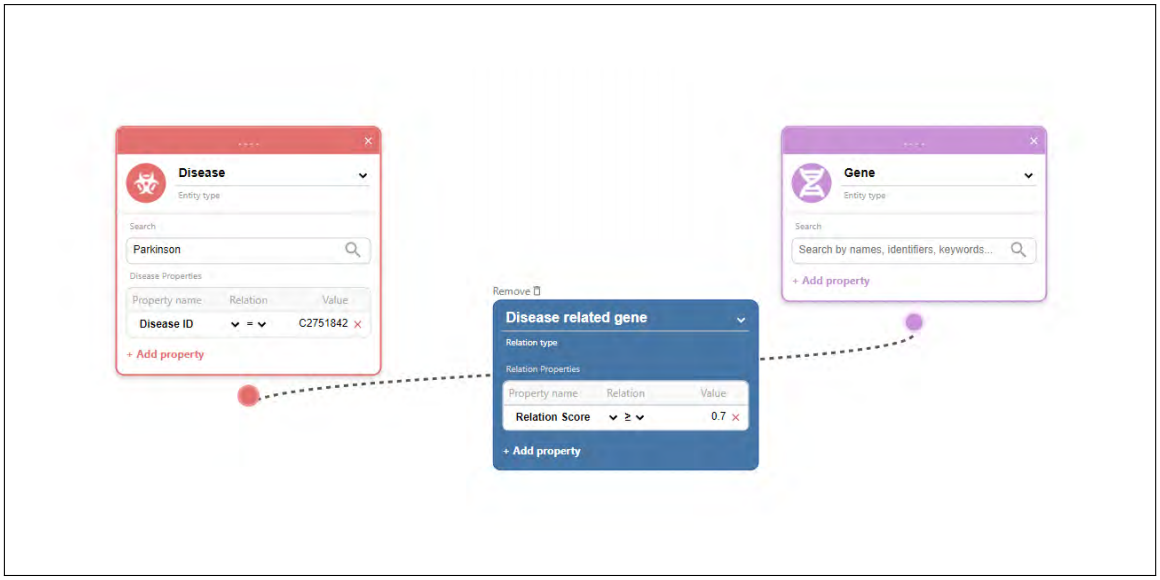

Figure 6.6: Query for extracting all genes highly related to Parkinson's disease.

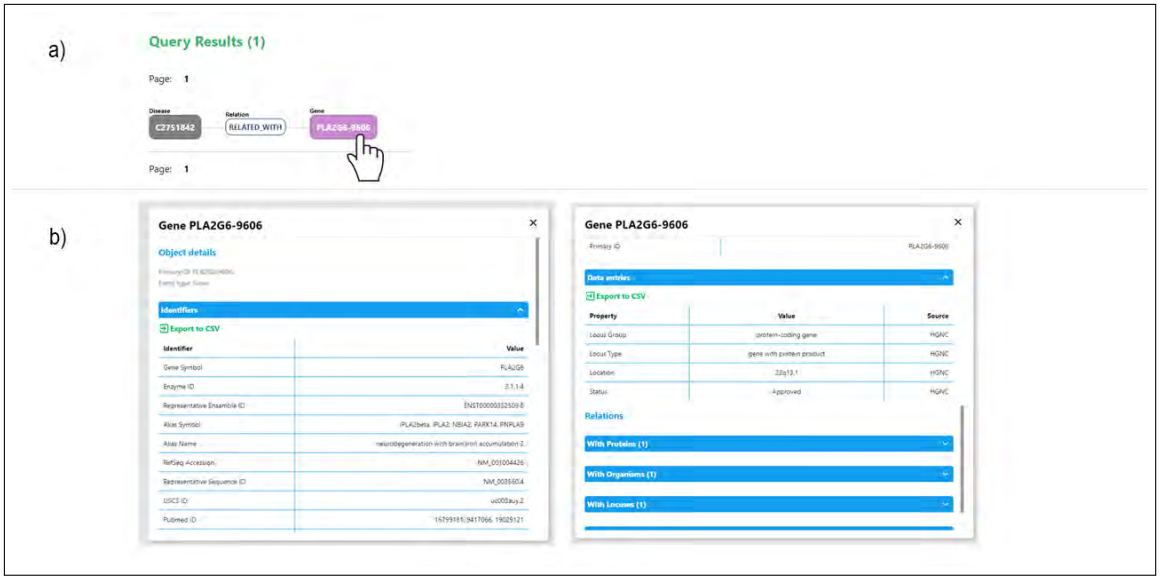

Figure 6.7: Results of the query for extracting all genes highly related to Parkinson's disease (a) and details of the PLA2G6 gene (b).

6.3.2 Genes related to pancreatic cancer

Another example that demonstrates the usability of the BioGraph system is finding the genes highly associated with specific types of tumors, such as pancreatic cancer. A query is constructed in a similar way as it was the case with genes related to Parkinson's disease, with the exception that the relation score threshold between genes and pancreatic cancer disease is set to greater than or equal to 0.3. The query is shown in Figure 6.8.

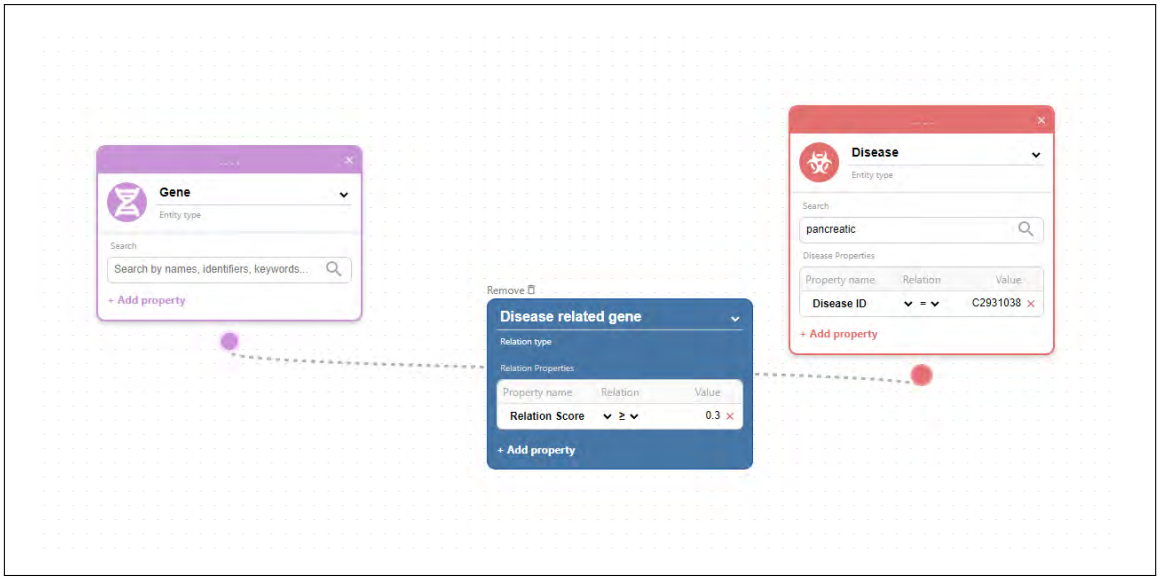

Figure 6.8: Query for extracting all genes highly related to pancreatic cancer.

The results for the given query contain 9 different genes: RABL3, BRCA1, BRCA2, CDKN2A, KRAS, SMAD4, TP53, PALLD, and PALB2. All genes are scientifically proven to be related to pancreatic cancer [38]. The results for the given query are shown in Figure 6.9. If the threshold value for association between genes and diseases is omitted, the system finds 43 related genes. All 43 genes are shown in Table 6.2.

Table 6.2: Genes found to be related to pancreatic cancer.

MAGT1	H3P10	H3P8	GTF2H5	RABL3
IDO2	ALPP	ALPI	ATM	BRCA1
BRCA2	CD47	CDK2	CFTR	CDKN2A
CDKN1A	CDH10	CPB1	ERCC2	FANCC
FANCG	MSH6	IAPP	KRAS	LCN2
SMAD4	MSH2	PKHD1	RNASEL	S100A9
SPINK1	STK11	TIMP1	TP53	CDK2AP2
SUB1	RPP14	CHEK2	PALLD	
PALD1	LAMTOR2	SF3B6	PALB2	

Figure 6.9: Results of the query for extracting all genes highly related to pancreatic cancer.

Chapter 7

Conclusion

This thesis presents challenges of unification and search of bioinformatics data and proposes solutions for the given challenges. The most significant contributions of the thesis are the definition and implementation of the BioGraph data model for the unification of metadata from heterogeneous bioinformatics databases and the automated pipeline for deriving new semantic similarity relations based on a set of existing relations found in the unified data.

The unification of heterogeneous data from many data sources provides a holistic view of intricate relationships between entities observed collectively instead of individually in isolated databases. The knowledge graphs provide the backbone structure for supporting such data unification but also unified searching of the unified data using complex graph query patterns. Implementation of a scalable knowledge graph system with a robust and flexible data model that supports efficient querying was a challenge that the work presented in this thesis attempted to solve. The results of the work are encouraging and provide motivation for further efforts for improvement and wider adoption of the BioGraph system in the scientific community.

The pipeline for automated discovery of semantic similarity relations is a promising base point for the development of knowledge-based bioinformatics systems with the ability to perform unsupervised learning of new relations between the objects stored in the knowledge database. Such a system would provide great assistance in many domains, but most significantly in the biomedical domain. Discovering hidden relations between diseases and various clinical parameters would enable the detection of new molecular pathways that could be used for drug discovery purposes.

The technical disadvantages associated with the BioGraph system, as discussed in the previous chapter, undoubtedly present a significant challenge to its broader adoption. However, it is important to view these limitations as a necessary step on the path toward developing a more efficient and comprehensive system. Having the system with an open source code allows the assistance of a broad community in solving the challenges but also proposing new features and applications of the system.

As there is currently no repository of community-developed importers, a reasonable assumption is that different community members may develop importers with slight differences in attribute naming or entity object selection for metadata originating from the same databases, resulting in multiple variations of essentially the same importers. Solving this issue will require standardization of the import script development process with precise guidelines and the creation of a central repository of import scripts.

The steps following the presented research include further improvements in the efficiency of the software architecture, testing the model for unification of data from even more data sources, improving the automated pipeline for detecting more types of semantic relations, and exploring possibilities of designing additional software systems that would use the BioGraph core system and its powerful data model in a vast number of applications in the bioinformatics domain.

Bibliography

[1] R. Agarwal, R. Srikant, et al. "Fast algorithms for mining association rules". In: *Proc. of the 20th VLDB Conference*. Vol. 487. 1994, p. 499.

[2] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. "Automatic subspace clustering of high dimensional data for data mining applications". In: *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*. 1998, pp. 94–105.

[3] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. "Basic local alignment search tool". In: *Journal of molecular biology* 215.3 (1990), pp. 403–410.

[4] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander. "OPTICS: Ordering points to identify the clustering structure". In: *ACM Sigmod record* 28.2 (1999), pp. 49–60.

[5] D. Barbará and P. Chen. "Using the fractal dimension to cluster datasets". In: *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*. 2000, pp. 260–264.

[6] T. Barrett, S. E. Wilhite, P. Ledoux, C. Evangelista, I. F. Kim, M. Tomashevsky, K. A. Marshall, K. H. Phillippy, P. M. Sherman, M. Holko, et al. "NCBI GEO: archive for functional genomics data sets—update". In: *Nucleic acids research* 41.D1 (2012), pp. D991–D995.

[7] A. Ben-Dor and Z. Yakhini. "Clustering gene expression patterns". In: *Proceedings of the third annual international conference on Computational molecular biology*. 1999, pp. 33–42.

[8] M. A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin. "Lowest common ancestors in trees and directed acyclic graphs". In: *Journal of Algorithms* 57.2 (2005), pp. 75–94.

[9] D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, and D. L. Wheeler. "GenBank". In: *Nucleic acids research* 31.1 (2003), p. 23.

[10] A. Bertels and D. Speelman. "Clustering for semantic purposes: Exploration of semantic similarity in a technical corpus". In: *Terminology. International Journal of Theoretical and Applied Issues in Specialized Communication* 20.2 (2014), pp. 279–303.

[11] J. C. Bezdek, R. Ehrlich, and W. Full. "FCM: The fuzzy c-means clustering algorithm". In: *Computers & geosciences* 10.2-3 (1984), pp. 191–203.

[12] C. Bizon, S. Cox, J. Balhoff, Y. Kebede, P. Wang, K. Morton, K. Fecho, and A. Tropsha. "ROBOKOP KG and KGB: Integrated Knowledge Graphs from Federated Sources". In: *Journal of Chemical Information and Modeling* 59.12 (2019). PMID: 31769676, pp. 4968–4973.

[13] A. Blanco-Míguez, F. Fdez-Riverola, B. Sánchez, and A. Lourenço. "BlasterJS: A novel interactive JavaScript visualisation component for BLAST alignment results". In: *PLoS One* 13.10 (2018), e0205286.

[14] A. Bookstein, V. A. Kulyukin, and T. Raita. "Generalized hamming distance". In: *Information Retrieval* 5 (2002), pp. 353–375.

[15] G. R. Brown, V. Hem, K. S. Katz, M. Ovetsky, C. Wallin, O. Ermolaeva, I. Tolstoy, T. Tatusova, K. D. Pruitt, D. R. Maglott, et al. "Gene: a gene-centered information resource at NCBI". In: *Nucleic acids research* 43.D1 (2015), pp. D36–D42.

[16] J. P. Buchmann and E. C. Holmes. "Entrezpy: a Python library to dynamically interact with the NCBI Entrez databases". In: *Bioinformatics* 35.21 (2019), pp. 4511–4514.

[17] S. K. Burley, H. M. Berman, G. J. Kleywegt, J. L. Markley, H. Nakamura, and S. Velankar. "Protein Data Bank (PDB): the single global macromolecular structure archive". In: *Protein crystallography: methods and protocols* (2017), pp. 627–641.

[18] D. Chandrasekaran and V. Mago. "Evolution of semantic similarity—a survey". In: *ACM Computing Surveys (CSUR)* 54.2 (2021), pp. 1–37.

[19] B. Chapman and J. Chang. "Biopython: Python tools for computational biology". In: *ACM Sigbio Newsletter* 20.2 (2000), pp. 15–19.

[20] I. Cohen, Y. Huang, J. Chen, J. Benesty, J. Benesty, J. Chen, Y. Huang, and I. Cohen. "Pearson correlation coefficient". In: *Noise reduction in speech processing* (2009), pp. 1–4.

[21] U. Consortium. "UniProt: a hub for protein information". In: *Nucleic acids research* 43.D1 (2015), pp. D204–D212.

[22] R. N. Dave and K. Bhaswan. "Adaptive fuzzy c-shells clustering and detection of ellipses". In: *IEEE Transactions on Neural Networks* 3.5 (1992), pp. 643–662.

[23] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv preprint arXiv:1810.04805* (2018).

[24] S. T. Dumais. "Latent semantic analysis". In: *Annual Review of Information Science and Technology (ARIST)* 38 (2004), pp. 189–230.

[25] Elsevier. *Biology knowledge graph*. `https://www.elsevier.com/solutions/biology-knowledge-graph`. Accessed: 2022-12-28.

[26] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, et al. "A density-based algorithm for discovering clusters in large spatial databases with noise." In: *kdd*. Vol. 96. 34. 1996, pp. 226–231.

[27] J. S. Farris. "On the cophenetic correlation coefficient". In: *Systematic Zoology* 18.3 (1969), pp. 279–285.

[28] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. "Cypher: An evolving query language for property graphs". In: *Proceedings of the 2018 international conference on management of data*. 2018, pp. 1433–1445.

[29] *GeneCards®: The Human Gene Database*. `https://www.genecards.org/`. accessed on 10 April 2023.

[30] S. Guha, R. Rastogi, and K. Shim. "ROCK: A robust clustering algorithm for categorical attributes". In: *Information systems* 25.5 (2000), pp. 345–366.

[31] B. Ha, J. A. Greenbaum, B. J. Shmiedel, D. Singh, A. Madrigal, A. G. Valdovino-Gonzalez, B. M. White, J. Zapardiel-Gonzalo, G. Altay, G. McVicker, et al. "Database of Immune Cell EQTLs, Expression, Epigenomics". In: *The Journal of Immunology* 202.1_Supplement (2019), pp. 131–18.

[32] J. Han, J. Pei, and Y. Yin. "Mining frequent patterns without candidate generation". In: *ACM sigmod record* 29.2 (2000), pp. 1–12.

[33] H Handschub and H. Gilbert. "Evaluation report security level of cryptography–sha-256". In: *(Issy-les-Moulineaux) Technical Report* (2002).

[34] J Hartigan. "The k-means algorithm". In: *Clustering algorithms* 4 (1975).

[35] B. Hopkins and J. G. Skellam. "A new method for determining the type of distribution of plant individuals". In: *Annals of Botany* 18.2 (1954), pp. 213–227.

[36] A. Huang et al. "Similarity measures for text document clustering". In: *Proceedings of the sixth new zealand computer science research student conference (NZCSRSC2008), Christchurch, New Zealand*. Vol. 4. 2008, pp. 9–56.

[37] T. Hubbard, D. Barker, E. Birney, G. Cameron, Y. Chen, L Clark, T. Cox, J Cuff, V. Curwen, T. Down, et al. "The Ensembl genome database project". In: *Nucleic acids research* 30.1 (2002), pp. 38–41.

[38] S. Idachaba, O. Dada, O. Abimbola, O. Olayinka, A. Uma, E. Olunu, and A. O. J. Fakoya. "A review of pancreatic cancer: epidemiology, genetics, screening, and management". In: *Open access Macedonian journal of medical sciences* 7.4 (2019), p. 663.

[39] *IEDB*. Retrieved from `http://www.iedg.org`. Accessed: 2023-02-06.

[40] A. K. Jain, M. N. Murty, and P. J. Flynn. "Data clustering: a review". In: *ACM computing surveys (CSUR)* 31.3 (1999), pp. 264–323.

[41] S. Ji, S. Pan, E. Cambria, P. Marttinen, and P. S. Yu. "A Survey on Knowledge Graphs: Representation, Acquisition, and Applications". In: *IEEE Transactions on Neural Networks and Learning Systems* 33.2 (2022), pp. 494–514.

[42] C. Kanz, P. Aldebert, N. Althorpe, W. Baker, A. Baldwin, K. Bates, P. Browne, A. van den Broek, M. Castro, G. Cochrane, et al. "The EMBL nucleotide sequence database". In: *Nucleic acids research* 33.suppl_1 (2005), pp. D29–D33.

[43] G. Karypis, E.-H. Han, and V. Kumar. "Chameleon: Hierarchical clustering using dynamic modeling". In: *computer* 32.8 (1999), pp. 68–75.

[44] L. Kaufman and P. J. Rousseeuw. *Finding groups in data: an introduction to cluster analysis*. John Wiley & Sons, 2009.

[45] M. G. Kendall. "A new measure of rank correlation". In: *Biometrika* 30.1/2 (1938), pp. 81–93.

[46] Y. Kim, C. Denton, L. Hoang, and A. M. Rush. "Structured attention networks". In: *arXiv preprint arXiv:1702.00887* (2017).

[47] T. Kohonen. "The self-organizing map". In: *Proceedings of the IEEE* 78.9 (1990), pp. 1464–1480.

[48] P. Kotiranta, M. Junkkari, and J. Nummenmaa. "Performance of graph and relational databases in complex queries". In: *Applied Sciences* 12.13 (2022), p. 6490.

[49] *Koza*. `https://koza.monarchinitiative.org/`. accessed on 28 December 2022.

[50] D. P. Lane. "p53 and human cancers". In: *British Medical Bulletin* 50.3 (Sept. 1994), pp. 582–599. eprint: `https://academic.oup.com/bmb/article-pdf/50/3/582/7292468/50-3-582.pdf`.

[51] J. Lee, W. Yoon, S. Kim, D. Kim, S. Kim, C. H. So, and J. Kang. "BioBERT: a pre-trained biomedical language representation model for biomedical text mining". In: *Bioinformatics* 36.4 (2020), pp. 1234–1240.

[52] Y. Li, Z. A. Bandar, and D. McLean. "An approach for measuring semantic similarity between words using multiple information sources". In: *IEEE Transactions on knowledge and data engineering* 15.4 (2003), pp. 871–882.

[53] A. M. Liekens, J. De Knijf, W. Daelemans, B. Goethals, P. De Rijk, and J. Del-Favero. "BioGraph: unsupervised biomedical knowledge discovery via automated hypothesis generation". In: *Genome biology* 12.6 (2011), pp. 1–12.

[54] D. Lin et al. "An information-theoretic definition of similarity." In: *Icml*. Vol. 98. 1998. 1998, pp. 296–304.

[55] J. Liu and J. Han. "Spectral clustering". In: *Data clustering*. Chapman and Hall/CRC, 2018, pp. 177–200.

[56] L. Ma, D. Zou, L. Liu, H. Shireen, A. A. Abbasi, A. Bateman, J. Xiao, W. Zhao, Y. Bao, and Z. Zhang. "Database Commons: A Catalog of Worldwide Biological Databases". In: *Genomics, Proteomics Bioinformatics* (2022).

[57] B. McBride. "The resource description framework (RDF) and its vocabulary description language RDFS". In: *Handbook on ontologies*. Springer, 2004, pp. 51–65.

[58] A. H. Murphy. "The Finley affair: A signal event in the history of forecast verification". In: *Weather and forecasting* 11.1 (1996), pp. 3–20.

[59] M. Nei and S. Kumar. *Molecular evolution and phylogenetics*. Oxford University Press, USA, 2000.

[60] *Neo4j Graph Database*. Retrieved from: `https://neo4j.com/product/neo4j-graph-database/`. Accessed: 2023-02-06.

[61] A. Neumann, N. Laranjeiro, and J. Bernardino. "An analysis of public REST web service APIs". In: *IEEE Transactions on Services Computing* 14.4 (2018), pp. 957–970.

[62] *NodeJS*. Retrieved from: `https://nodejs.org/`. Accessed: 2023-02-06.

[63] A. Panchenko, S. Adeykin, A. Romanov, and P. Romanov. "Extraction of semantic relations between concepts with knn algorithms on wikipedia". In: *Concept Discovery in Unstructured Data Workshop (CDUD) of International Conference On Formal Concept Analysis, Belgium*. Citeseer. 2012, pp. 78–88.

[64] H.-S. Park and C.-H. Jun. "A simple and fast algorithm for K-medoids clustering". In: *Expert systems with applications* 36.2 (2009), pp. 3336–3341.

[65] J. Parkinson. "An essay on the shaking palsy". In: *The Journal of neuropsychiatry and clinical neurosciences* 14.2 (2002), pp. 223–236.

[66] J. Pérez, M. Arenas, and C. Gutierrez. "Semantics and complexity of SPARQL". In: *ACM Transactions on Database Systems (TODS)* 34.3 (2009), pp. 1–45.

[67] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč. "Foundations of JSON schema". In: *Proceedings of the 25th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee. 2016, pp. 263–273.

[68] J. Piñero, R.-A. Juan Manuel, F. R. Josep Saüch-Pitarch, F. S. Emilio Centeno, and L. I. Furlong. "The DisGeNET knowledge platform for disease genomics: 2019 update". In: *Nucl. Acids Res.* (2019).

[69] R. Rada, H. Mili, E. Bicknell, and M. Blettner. "Development and application of a metric on semantic nets". In: *IEEE transactions on systems, man, and cybernetics* 19.1 (1989), pp. 17–30.

[70] C. Rasmussen. "The infinite Gaussian mixture model". In: *Advances in neural information processing systems* 12 (1999).

[71] P. Rawat and A. N. Mahajan. "ReactJS: A modern web development framework". In: *International Journal of Innovative Science and Research Technology* 5.11 (2020), pp. 698–702.

[72] D. J. Rogers and T. T. Tanimoto. "A Computer Program for Classifying Plants: The computer is programmed to simulate the taxonomic process of comparing each case with every other case." In: *Science* 132.3434 (1960), pp. 1115–1118.

[73] E. Sayers. "A General Introduction to the E-utilities". In: *Entrez Programming Utilities Help [Internet]. Bethesda (MD): National Center for Biotechnology Information (US)* (2010).

[74] G. D. Schuler, J. A. Epstein, H. Ohkawa, and J. A. Kans. "[10] Entrez: Molecular biology database and retrieval system". In: *Methods in enzymology*. Vol. 266. Elsevier, 1996, pp. 141–162.

[75] R. L. Seal, B. Braschi, K. Gray, T. E. M. Jones, S. Tweedie, L. Haim-Vilmovsky, and E. A. Bruford. "Genenames.org: the HGNC resources in 2023". In: *Nucleic Acids Research* (Oct. 2022).

[76] P. Sethi and S. Alagiriswamy. "Association rule based similarity measures for the clustering of gene expression data". In: *The open medical informatics journal* 4 (2010), p. 63.

[77] K. A. Shefchek, N. L. Harris, M. Gargano, N. Matentzoglu, D. Unni, M. Brush, D. Keith, T. Conlin, N. Vasilevsky, X. A. Zhang, et al. "The Monarch Initiative in 2019: an integrative data and analytic platform connecting phenotypes to genotypes across species". In: *Nucleic acids research* 48.D1 (2020), pp. D704–D715.

[78] T. Shen, J. Hu, Y. Jiang, S. Zhao, C. Lin, X. Yin, Y. Yan, J. Pu, H.-Y. Lai, and B. Zhang. "Early-onset Parkinson's disease caused by PLA2G6 compound heterozygous mutation, a case report and literature review". In: *Frontiers in neurology* 10 (2019), p. 915.

[79] G. S. C. Slater and E. Birney. "Automated generation of heuristics for biological sequence comparison". In: *BMC bioinformatics* 6 (2005), pp. 1–11.

[80] D. Szklarczyk, A. L. Gable, K. C. Nastou, D. Lyon, R. Kirsch, S. Pyysalo, N. T. Doncheva, M. Legeay, T. Fang, P. Bork, et al. "The STRING database in 2021: customizable protein–protein networks, and functional characterization of user-uploaded gene/measurement sets". In: *Nucleic acids research* 49.D1 (2021), pp. D605–D612.

[81] P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to data mining*. Pearson Education India, 2016.

[82] Y. Tateno, T. Imanishi, S. Miyazaki, K. Fukami-Kobayashi, N. Saitou, H. Sugawara, and T. Gojobori. "DNA Data Bank of Japan (DDBJ) for genome scale research in life science". In: *Nucleic acids research* 30.1 (2002), pp. 27–30.

[83] "The 7th Congress of Biophysicists of Russia - conference proceedings". In: *Biophysical Reviews* 15.5 (2023), pp. 1877–1877.

[84] D. R. Unni, S. A. Moxon, M. Bada, M. Brush, R. Bruskiewich, J. H. Caufield, P. A. Clemons, V. Dancik, M. Dumontier, K. Fecho, et al. "Biolink Model: A universal schema for knowledge graphs in clinical, biomedical, and translational science". In: *Clinical and Translational Science* (2022).

[85] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017).

[86] A. N. Veljković and N. S. Mitić. "Semantic unification and search of bioinformatics databases". In: *Belgrade BioInformatics Conference*. 2023, p. 63.

[87] A. N. Veljković, Y. L. Orlov, and N. S. Mitić. "BioGraph: Data Model for Linking and Querying Diverse Biological Metadata". In: *World congress Systems theory, algebraic biology, artificial intelligence: mathematical foundations and applications*. 2023.

[88] A. N. Veljković, Y. L. Orlov, and N. S. Mitić. "BioGraph: Data Model for Linking and Querying Diverse Biological Metadata". In: *International Journal of Molecular Sciences* 24.8 (2023), p. 6954.

[89] S. Vucetic et al. "DisProt: a database of protein disorder". en. In: *Bioinformatics* 21.1 (2004), pp. 137–140.

[90] W. Wang, J. Yang, R. Muntz, et al. "STING: A statistical information grid approach to spatial data mining". In: *Vldb*. Vol. 97. 1997, pp. 186–195.

[91] J. H. Ward Jr. "Hierarchical grouping to optimize an objective function". In: *Journal of the American statistical association* 58.301 (1963), pp. 236–244.

[92] X. Wu, J. Duan, Y. Pan, and M. Li. "Medical knowledge graph: Data sources, construction, reasoning, and applications". In: *Big Data Mining and Analytics* 6.2 (2023), pp. 201–217.

[93] Z. Wu and M. Palmer. "Verb semantics and lexical selection". In: *arXiv preprint cmp-lg/9406033* (1994).

[94] D. Xu and Y. Tian. "A comprehensive survey of clustering algorithms". In: *Annals of Data Science* 2 (2015), pp. 165–193.

[95] R. R. Yager and D. P. Filev. "Approximate clustering via the mountain method". In: *IEEE Transactions on systems, man, and Cybernetics* 24.8 (1994), pp. 1279–1284.

[96] A. Yates, K. Beal, S. Keenan, W. McLaren, M. Pignatelli, G. R. Ritchie, M. Ruffier, K. Taylor, A. Vullo, and P. Flicek. "The Ensembl REST API: Ensembl data for any language". In: *Bioinformatics* 31.1 (2015), pp. 143–145.

[97] G. Zhang, L. Chitkushev, L. R. Olsen, D. B. Keskin, and V. Brusic. "TANTIGEN 2.0: a knowledge base of tumor T cell antigens and epitopes". In: *BMC bioinformatics* 22.8 (2021), pp. 1–8.

[98] T. Zhang, R. Ramakrishnan, and M. Livny. "BIRCH: an efficient data clustering method for very large databases". In: *ACM sigmod record* 25.2 (1996), pp. 103–114.

[99] G. Zhu and C. A. Iglesias. "Computing semantic similarity of concepts in knowledge graphs". In: *IEEE Transactions on Knowledge and Data Engineering* 29.1 (2016), pp. 72–85.

Appendix A

Database importers

The import scripts for importing data from the original databases into the BioGraph data model are listed here.

A.1 DisProt data importer

```
1  const axios = require('axios');
2
3  class DisprotImporter {
4      constructor(bg) {
5          this.bg = bg;
6          this.importer = 'disprot';
7          this.importerVersion = '1.0';
8          this.dataSource = 'DisProt';
9          console.log('Disprot importer loaded');
10     }
11
12     async run() {
13         const { bg } = this;
14
15         // Start new import
16         await bg.beginImport(this.importer, this.importerVersion,
               this.dataSource);
17
18         console.log('Importing data from DisProt...');
19
20         // DisProt URL
21         const url = 'https://disprot.org/api/search?release=current
               &show_ambiguous=false&show_obsolete=false&format=json'
22
23         // Acc list of all imported proteins from disprot
24     const proteinAcc = [];
25
26     console.log('Downloading DisProt data...');
27     const { data, size } = (await axios.get(url)).data;
28     console.log('Download complete, preparing data...');
29
30         let i = 0;
31
```

```
32          console.log('Parsing import data');
33          for (const protein of data) {
34        i += 1;
35
36              // Logging progres
37        if (i > 0 && i % 100 == 0) {
38          console.log(`DisProt: ${i}/${data.length}`);
39        }
40
41              const {
42          acc,
43          disprot_id,
44          name,
45          genes,
46        } = protein;
47
48              proteinAcc.push(acc);
49
50              let uniref50 = null;
51        if (protein.uniref50 != null) {
52          uniref50 = protein.uniref50;
53        }
54
55        let uniref90 = null;
56        if (protein.uniref90 != null) {
57          uniref90 = protein.uniref90;
58        }
59
60        let uniref100 = null;
61        if (protein.uniref100 != null) {
62          uniref100 = protein.uniref100;
63        }
64
65              const proteinEntityId = await bg.createEntityNode('
                    Protein', acc);
66              await bg.createIdentifierNode(proteinEntityId, 'id', '
                    DisProt ID', disprot_id);
67        await bg.createIdentifierNode(proteinEntityId, 'id', 'UniProt
              ID', acc);
68        await bg.createIdentifierNode(proteinEntityId, 'url', '
             UniProt URL', `https://uniprot.org/uniprot/${disprot_id}`)
             ;
69
70              if (uniref50 != null) {
71          await bg.createIdentifierNode(proteinEntityId, 'id', '
                UniRef 50', uniref50);
72        }
73
74        if (uniref90 != null) {
75          await bg.createIdentifierNode(proteinEntityId, 'id', '
                UniRef 90', uniref90);
76        }
77
78        if (uniref100 != null) {
79          await bg.createIdentifierNode(proteinEntityId, 'id', '
                UniRef 100', uniref100);
```

```
80      }
81
82            await bg.createIdentifierNode(proteinEntityId, 'url', '
                 DisProt URL', `https://disprot.org/${disprot_id}`);
83      await bg.createIdentifierNode(proteinEntityId, 'name', '
           Protein Name', name);
84
85
86            // Taxonomy
87      // =========================
88
89      const taxonId = `${protein.ncbi_taxon_id}`;
90
91      const organismName = protein.organism;
92
93      const organismEntityId = await bg.createEntityNode('Organism'
           , taxonId);
94
95      await bg.createIdentifierNode(organismEntityId, 'id', 'Taxon
           ID', taxonId);
96            await bg.createIdentifierNode(organismEntityId, 'id', '
                 Taxon Name', organismName);
97
98      // Gene
99      // =========================
100
101     if (genes != null) {
102       for (const gene of genes) {
103         let geneEntityId = null
104         let rootGeneEntityId = null
105         if (gene.name != null) {
106           const geneSymbol = gene.name.value;
107           rootGeneEntityId = await bg.createEntityNode('Gene',
                 geneSymbol);
108           geneEntityId = await bg.createEntityNode('Gene', `${
                 geneSymbol}-${taxonId}`);
109
110           await bg.createEntityEdge(proteinEntityId, geneEntityId
                 , 'FROM', {});
111           await bg.createEntityEdge(geneEntityId,
                 organismEntityId, 'FROM', {});
112           await bg.createEntityEdge(geneEntityId,
                 rootGeneEntityId, 'IS_INSTANCE', {});
113           await bg.createIdentifierNode(geneEntityId, 'id', 'Gene
                  symbol', geneSymbol);
114           await bg.createIdentifierNode(rootGeneEntityId, 'id', '
                 Gene symbol', geneSymbol);
115         }
116
117         if (geneEntityId != null && gene.synonyms != null) {
118           const geneSynonyms = gene.synonyms.map(el => el.value);
119           for (const synonym of geneSynonyms) {
120             await bg.createIdentifierNode(geneEntityId, 'id', '
                    Synonym', synonym);
121             await bg.createIdentifierNode(rootGeneEntityId, 'id',
                     'Synonym', synonym);
```

```
            }
          }
        }
      }

            await bg.createDataNode(organismEntityId, 'DisProt', {
               species: taxonId });

      await bg.createEntityEdge(proteinEntityId, organismEntityId,
         'FROM', {});

            // Protein data
      // ========================

      const disorderContent = parseFloat(protein.disorder_content);
      const regionsCounter = parseInt(protein.regions_counter);

            await bg.createDataNode(proteinEntityId, 'DisProt', {
        disorder_content: disorderContent,
        regions_counter: regionsCounter,
      });
        }

        console.log(`DisProt: ${size}/${size}`)

        // Finish and commit the import
        await bg.finishImport();
        console.log('Disprot import complete');
    }
}

module.exports = DisprotImporter;
```

Listing A.1: DisProt data importer script

A.2 HGNC data importer

```
const axios = require('axios');

class HGNCImporter {
    constructor(bg) {
        this.bg = bg;
        this.importer = 'hgnc';
        this.importerVersion = '1.0';
        this.dataSource = 'HGNC';
        console.log('HGNC importer loaded');
    }

    async run() {
        const { bg } = this;

        // Start new import
        await bg.beginImport(this.importer, this.importerVersion,
           this.dataSource);
```

```
17
18          console.log('Importing data from HGNC...');
19
20          // HGNC URL
21          const url = 'http://ftp.ebi.ac.uk/pub/databases/genenames/
                hgnc/json/hgnc_complete_set.json'
22
23          console.log('Downloading data...');
24      const { docs: data }  = (await axios.get(url)).data.response;
25          const size = data.length;
26      console.log('Download complete, preparing data...');
27
28          let i = 0;
29
30          console.log('Parsing import data');
31          const organismEntityId = await bg.createEntityNode('
                Organism', `9606`);
32          await bg.createIdentifierNode(organismEntityId, 'id', 'NCBI
                ID', '9606');
33
34
35          for (const gene of data) {
36        i += 1;
37
38              if (i % 100 == 0) {
39                  console.log(`HGNC: ${i}/${data.length}`);
40              }
41
42              // Logging progres
43        if (i > 0 && i % 2000 == 0) {
44          console.log(`HGNC: ${i}/${data.length}`);
45                  await bg.closeBatch();
46        }
47
48              const {
49                  symbol,
50                  alias_symbol: alias_symbols,
51                  alias_name: alias_names,
52                  name,
53                  omim_id: omim_ids,
54                  rgd_id: rgd_ids,
55                  ucsc_id: ucsc_id,
56                  entrez_id,
57                  ensembl_gene_id,
58                  gene_group: gene_groups,
59                  gene_group_id: gene_group_ids,
60                  location,
61                  locus_type,
62                  locus_group,
63                  status,
64                  vega_id,
65                  hgnc_id,
66                  ncbi_id,
67                  uniprot_ids,
68                  refseq_accession: refseq_accessions,
69                  pubmed_id: pubmed_ids,
```

```
70                  gene_family_id: gene_family_ids,
71                  gene_family: gene_families,
72                  ccds_ids,
73                  mgd_id: mgd_ids,
74                  enzyme_id: enzyme_ids,
75                  mane_select,
76              } = gene;
77
78              const rootGeneEntityId = await bg.createEntityNode('
                    Gene', symbol);
79              const geneEntityId = await bg.createEntityNode('Gene',
                    `${symbol}-9606`);
80              await bg.createEntityEdge(geneEntityId,
                    rootGeneEntityId, 'IS_INSTANCE', {});
81              await bg.createEntityEdge(geneEntityId,
                    organismEntityId, 'FROM', {});
82
83              await bg.createIdentifierNode(geneEntityId, 'name', '
                    Gene Symbol', symbol);
84              await bg.createIdentifierNode(rootGeneEntityId, 'name',
                     'Gene Symbol', symbol);
85
86              if (name != null) {
87                  await bg.createIdentifierNode(geneEntityId, 'name',
                         'Gene Name', name);
88                  await bg.createIdentifierNode(rootGeneEntityId, '
                        name', 'Gene Name', name);
89              }
90
91              if (entrez_id != null) {
92                  await bg.createIdentifierNode(geneEntityId, 'id', '
                        Entrez ID', entrez_id);
93                  await bg.createIdentifierNode(rootGeneEntityId, 'id
                        ', 'Entrez ID', entrez_id);
94              }
95
96              if (ensembl_gene_id != null) {
97                  await bg.createIdentifierNode(geneEntityId, 'id', '
                        Ensembl Gene ID', ensembl_gene_id);
98                  await bg.createIdentifierNode(rootGeneEntityId, 'id
                        ', 'Ensembl Gene ID', ensembl_gene_id);
99              }
100
101             if (vega_id != null) {
102                 await bg.createIdentifierNode(geneEntityId, 'id', '
                        Vega ID', vega_id);
103                 await bg.createIdentifierNode(rootGeneEntityId, 'id
                        ', 'Vega ID', vega_id);
104             }
105
106             if (hgnc_id != null) {
107                 await bg.createIdentifierNode(geneEntityId, 'id', '
                        HGNC ID', hgnc_id);
108                 await bg.createIdentifierNode(rootGeneEntityId, 'id
                        ', 'HGNC ID', hgnc_id);
109             }
```

```
110
111             if (ncbi_id != null) {
112                 await bg.createIdentifierNode(geneEntityId, 'id', '
                        NCBI ID', ncbi_id);
113                 await bg.createIdentifierNode(rootGeneEntityId, 'id
                        ', 'HGNC ID', hgnc_id);
114             }
115
116             if (enzyme_ids != null) {
117                 for (const enzyme_id of enzyme_ids) {
118                     await bg.createIdentifierNode(geneEntityId, 'id
                            ', 'Enzyme ID', enzyme_id);
119                     await bg.createIdentifierNode(rootGeneEntityId,
                            'id', 'Enzyme ID', enzyme_id);
120                 }
121             }
122
123             if (pubmed_ids != null) {
124                 for (const pubmed_id of pubmed_ids) {
125                     await bg.createIdentifierNode(geneEntityId, 'id
                            ', 'Pubmed ID', pubmed_id);
126                     await bg.createIdentifierNode(rootGeneEntityId,
                            'id', 'Enzyme ID', pubmed_id);
127                 }
128             }
129
130             if (omim_ids != null) {
131                 for (const omim_id of omim_ids) {
132                     await bg.createIdentifierNode(geneEntityId, 'id
                            ', 'OMIM ID', omim_id);
133                     await bg.createIdentifierNode(rootGeneEntityId,
                            'id', 'OMIM ID', omim_id);
134                 }
135             }
136
137             if (rgd_ids != null) {
138                 for (const rgd_id of rgd_ids) {
139                     await bg.createIdentifierNode(geneEntityId, 'id
                            ', 'Rat Gene Database ID', rgd_id);
140                     await bg.createIdentifierNode(rootGeneEntityId,
                            'id', 'Rat Gene Database ID', rgd_id);
141                 }
142             }
143
144             if (ucsc_id != null) {
145                 await bg.createIdentifierNode(geneEntityId, 'id', '
                        USCS ID', ucsc_id);
146                 await bg.createIdentifierNode(rootGeneEntityId, 'id
                        ', 'USCS ID', ucsc_id);
147             }
148
149             if (alias_symbols != null) {
150                 for (const alias_symbol of alias_symbols) {
151                     await bg.createIdentifierNode(geneEntityId, 'id
                            ', 'Alias Symbol', alias_symbol);
152                     await bg.createIdentifierNode(rootGeneEntityId,
```

```
                        'id', 'Alias Symbol', alias_symbol);
153                 }
154             }
155
156             if (alias_names != null) {
157                 for (const alias_name of alias_names) {
158                     await bg.createIdentifierNode(geneEntityId, 'id
                            ', 'Alias Name', alias_name);
159                     await bg.createIdentifierNode(rootGeneEntityId,
                            'id', 'Alias Symbol', alias_name);
160                 }
161             }
162
163             if (mgd_ids != null) {
164                 for (const mgd_id of mgd_ids) {
165                     await bg.createIdentifierNode(geneEntityId, 'id
                            ', 'Mouse Gene Database ID', mgd_id);
166                     await bg.createIdentifierNode(rootGeneEntityId,
                            'id', 'Mouse Gene Database ID', mgd_id);
167                 }
168             }
169
170             if (ccds_ids != null) {
171                 for (const ccd_id of ccds_ids) {
172                     await bg.createIdentifierNode(geneEntityId, 'id
                            ', 'Consensus CDS ID', ccd_id);
173                     await bg.createIdentifierNode(rootGeneEntityId,
                            'id', 'Consensus CDS ID', ccd_id);
174                 }
175             }
176
177             if (refseq_accessions != null) {
178                 for (const refseq_accession of refseq_accessions) {
179                     await bg.createIdentifierNode(geneEntityId, 'id
                            ', 'RefSeq Accession', refseq_accession);
180                     await bg.createIdentifierNode(rootGeneEntityId,
                            'id', 'RefSeq Accession', refseq_accession)
                            ;
181                 }
182             }
183
184             if (gene_families != null) {
185                 for (const gene_family of gene_families) {
186                     await bg.createIdentifierNode(geneEntityId, 'id
                            ', 'Gene Family', gene_family);
187                     await bg.createIdentifierNode(rootGeneEntityId,
                            'id', 'Gene Family', gene_family);
188                 }
189             }
190
191             if (gene_family_ids != null) {
192                 for (const gene_family_id of gene_family_ids) {
193                     await bg.createIdentifierNode(geneEntityId, 'id
                            ', 'Gene Family ID', gene_family_id);
194                     await bg.createIdentifierNode(rootGeneEntityId,
                            'id', 'Gene Family ID', gene_family_id);
```

```
195                 }
196             }
197
198             if (mane_select != null) {
199                 const [ens_id, rep_seq_id] = mane_select;
200                 await bg.createIdentifierNode(geneEntityId, 'id', '
                        Representative Ensamble ID', ens_id);
201                 await bg.createIdentifierNode(geneEntityId, 'id', '
                        Representative Sequence ID', rep_seq_id);
202                 await bg.createIdentifierNode(rootGeneEntityId, 'id
                        ', 'Representative Ensamble ID', ens_id);
203                 await bg.createIdentifierNode(rootGeneEntityId, 'id
                        ', 'Representative Sequence ID', rep_seq_id);
204             }
205
206             const locusEntityId = await bg.createEntityNode('Locus'
                    , location);
207             await bg.createDataNode(locusEntityId, 'HGNC', {
                    location });
208             await bg.createIdentifierNode(locusEntityId, 'id', '
                    Location', location);
209             await bg.createEntityEdge(geneEntityId, locusEntityId,
                    'FROM', {});
210
211             const geneData = {
212                 location,
213                 locus_type,
214                 locus_group,
215                 status,
216             }
217
218             await bg.createDataNode(geneEntityId, 'HGNC', geneData)
                    ;
219
220              // UniProt ids
221              if (uniprot_ids != null) {
222                 for (const uniprot_id of uniprot_ids) {
223                     const proteinEntityId = await bg.
                            createEntityNode('Protein', uniprot_id);
224                     await bg.createIdentifierNode(proteinEntityId,
                            'id', 'UniProt ID', uniprot_id);
225                     await bg.createEntityEdge(proteinEntityId,
                            geneEntityId, 'FROM', {});
226                     await bg.createEntityEdge(proteinEntityId,
                            organismEntityId, 'FROM', {});
227                 }
228             }
229
230
231         }
232
233         console.log(`HGNC: ${data.length}/${data.length}`)
234
235         // Finish and commit the import
236         await bg.finishImport();
237         console.log('HGNC import complete');
```

```
238     }
239 }
240
241 module.exports = HGNCImporter;
```

Listing A.2: HGNC data importer script

A.3 DisGeNET data importer

```
1  const { TSV } = require('tsv');
2  const fs = require('fs');
3
4  class DisGeNetImporter {
5      constructor(bg) {
6          this.bg = bg;
7          this.importer = 'disgenet';
8          this.importerVersion = '1.0';
9          this.dataSource = 'DisGeNet';
10         console.log('DisGeNet importer loaded');
11     }
12
13     async run() {
14         const { bg } = this;
15
16         // Start new import
17         await bg.beginImport(this.importer, this.importerVersion,
               this.dataSource);
18
19         console.log('Importing data from DisGeNet...');
20
21
22
23         // Load data
24         const tsvFile = fs.readFileSync('./importers/local-data/
               disgenet.tsv','utf8');
25         const parser = new TSV.Parser("\t", { header: true });
26     const data = parser.parse(tsvFile);
27
28         const organismEntityId = await bg.createEntityNode('
               Organism', `9606`);
29         await bg.createIdentifierNode(organismEntityId, 'id', 'NCBI
               ID', '9606');
30
31         let i = 0;
32         for (const disease of data) {
33             i += 1;
34
35             if (i % 1000 == 0 && i > 0) {
36                 console.log(`DisGeNet: ${i}/${data.length}`);
37             }
38
39       if (i % 10000 == 0 && i !== 0) {
40                 await bg.closeBatch();
41       }
```

```
42
43        const {
44          diseaseId,
45          diseaseName,
46          diseaseType,
47          diseaseClass,
48          diseaseSemanticType,
49        } = disease;
50
51        // Create disase
52              const diseaseEntityId = await bg.createEntityNode('
                    Disease', diseaseId);
53              await bg.createEntityEdge(diseaseEntityId,
                    organismEntityId, 'FROM', {});
54
55              await bg.createIdentifierNode(diseaseEntityId, 'id', '
                    Condition ID', diseaseId);
56              await bg.createIdentifierNode(diseaseEntityId, 'name',
                    'Disease Type', diseaseType);
57              await bg.createIdentifierNode(diseaseEntityId, 'name',
                    'Disease Semantic Type', diseaseSemanticType);
58              await bg.createIdentifierNode(diseaseEntityId, 'name',
                    'Disease Name', diseaseName);
59
60
61              await bg.createDataNode(diseaseEntityId, 'DisGeNet', {
62          disease_name: diseaseName,
63        });
64
65        if (diseaseClass != null) {
66          for (const dclass of diseaseClass.split(';')) {
67                      await bg.createIdentifierNode(diseaseEntityId,
                            'id', 'Disease Class', dclass);
68                      const diseaseClassEntityId = await bg.
                            createEntityNode('Disease_Class', dclass);
69                      await bg.createIdentifierNode(
                            diseaseClassEntityId, 'id', 'Disease Class',
                             dclass);
70                      await bg.createEntityEdge(diseaseEntityId,
                            diseaseClassEntityId, 'IS_INSTANCE', {});
71          }
72
73          // TODO: Disease class names
74
75          const { geneId, geneSymbol, DSI, DPI, score } = disease;
76
77                  const rootGeneEntityId = await bg.createEntityNode(
                        'Gene', geneSymbol.trim());
78                  const geneEntityId = await bg.createEntityNode('
                        Gene', `${geneSymbol.trim()}-9606`);
79                  await bg.createIdentifierNode(rootGeneEntityId, 'id
                        ', 'NCBI ID', geneId);
80                  await bg.createIdentifierNode(rootGeneEntityId, 'id
                        ', 'Gene Symbol', geneSymbol.trim());
81
82                  await bg.createIdentifierNode(geneEntityId, 'id', '
```

```
                    NCBI ID', geneId);
83                  await bg.createIdentifierNode(geneEntityId, 'name',
                        'Gene Symbol', geneSymbol.trim());
84
85                  await bg.createEntityEdge(geneEntityId,
                        rootGeneEntityId, 'IS_INSTANCE', {});
86                  await bg.createEntityEdge(geneEntityId,
                        organismEntityId, 'FROM', {});
87
88          const edgeData = {
89            dsi: DSI || null,
90            dpi: DPI || null,
91            score: score ? parseFloat(score) : null
92          }
93
94                  await bg.createEntityEdge(diseaseEntityId,
                        geneEntityId, 'RELATED_WITH', edgeData);
95        }
96      }
97
98
99          console.log(`DisGeNet: ${data.length}/${data.length}`);
100
101         // Finish and commit the import
102         await bg.finishImport();
103         console.log('DisGeNet import complete');
104     }
105 }
106
107 module.exports = DisGeNetImporter;
```

Listing A.3: DisGeNET data importer script

A.4 IEDB data importer

```
1  const Papa = require('papaparse');
2  const fs = require('fs');
3
4  class IEDBImporter {
5      constructor(bg) {
6          this.bg = bg;
7          this.importer = 'iedb';
8          this.importerVersion = '1.0';
9          this.dataSource = 'IEDB';
10         console.log('IEDB importer loaded');
11     }
12
13     parseCSV(path, skipRows = null) {
14         return new Promise((resolve, reject) => {
15             const loadedData = [];
16
17             console.log('Parsing input file: ', path);
18             const stream = fs.createReadStream(path, 'utf8');
```

```
19              const parseStream = Papa.parse(Papa.NODE_STREAM_INPUT,
                    { header: false, });
20
21              stream.pipe(parseStream);
22
23              parseStream.on('data', (data) => loadedData.push(data))
                    ;
24              parseStream.on('finish', () => resolve(skipRows != null
                     ? loadedData.slice(skipRows) : loadedData));
25              parseStream.on('error', err => reject(err))
26
27          })
28      }
29
30      async processEpitope(
31          bg,
32          epitopeIri,
33          objectType,
34          description,
35          epitopeModifiedResidues,
36          epitopeModifications,
37          startingPosition,
38          endingPosition,
39          nonPeptidicEpitopeIri,
40          epitopeSynonyms,
41          antigenName,
42          antigenIri,
43          parentProtein,
44          parentProteinIri,
45          parentOrganism,
46          parentOrganismIri,
47          epitopeComments,
48      ) {
49          const epitopeEntityId = await bg.createEntityNode('Epitope'
                , `${epitopeIri.split('/').slice(-1)[0]}`);
50          await bg.createIdentifierNode(epitopeEntityId, 'id', `IEDB
                ID`, `${epitopeIri.split('/').slice(-1)[0]}`);
51          await bg.createIdentifierNode(epitopeEntityId, 'url', `IEDB
                 URL`, `${epitopeIri}`);
52          await bg.createIdentifierNode(epitopeEntityId, 'name', `
                Epitope description`, description);
53          await bg.createIdentifierNode(epitopeEntityId, 'id', `IEDB
                IRI`, `${epitopeIri}`);
54
55          const epitopeData = {
56              ...(epitopeModifiedResidues.length > 0 && {
                    epitope_modified_residues: epitopeModifiedResidues
                    }),
57              ...(description.length > 0 && { epitope_description:
                    description }),
58              ...(startingPosition.length > 0 && { start_position:
                    startingPosition }),
59              ...(endingPosition.length > 0 && { end_position:
                    endingPosition }),
60              ...(objectType.length > 0 && { object_type: objectType
                    }),
```

```
61              ...(epitopeModifications.length > 0 && {
                    epitope_modifications: epitopeModifications }),
62              ...(epitopeComments.length > 0 && { epitope_comments:
                    epitopeComments })
63          };
64
65          if (Object.keys(epitopeData).length > 0) {
66              await bg.createDataNode(epitopeEntityId, 'iedb',
                    epitopeData, epitopeComments.length > 0 ? "
                    epitope_comments" : null);
67          }
68
69          if (nonPeptidicEpitopeIri.length > 0) {
70              const nonPeptidicMoleculeEntityId = await bg.
                    createEntityNode('Molecule', nonPeptidicEpitopeIri.
                    split('/').slice(-1)[0].split('_')[1]);
71              await bg.createIdentifierNode(
                    nonPeptidicMoleculeEntityId, 'sequence', `ChEBI ID`,
                     nonPeptidicEpitopeIri.split('/').slice(-1)[0].split
                    ('_')[1]);
72              await bg.createEntityEdge(nonPeptidicMoleculeEntityId,
                    epitopeEntityId, 'CONTAINS', {});
73
74              if (parentOrganismIri.length > 0) {
75                  const moleculeOrganismId = parentOrganismIri.split(
                        'NCBITaxon_')[1];
76                  const moleculeOrganismEntityId = await bg.
                        createEntityNode('Organism', `${
                        moleculeOrganismId}`);
77                  await bg.createEntityEdge(
                        nonPeptidicMoleculeEntityId,
                        moleculeOrganismEntityId, 'FROM');
78
79                  await bg.createIdentifierNode(
                        moleculeOrganismEntityId, 'id', `Taxon ID`, `${
                        parentOrganismIri.split('NCBITaxon_')[1]}`);
80                  await bg.createIdentifierNode(
                        moleculeOrganismEntityId, 'url', `Taxon URL`, `$
                        {parentOrganismIri}`);
81                  await bg.createIdentifierNode(
                        moleculeOrganismEntityId, 'name', `Taxon Name`,
                        `${parentOrganism}`);
82              }
83
84
85
86          }
87
88          if (epitopeSynonyms.length > 0) {
89              for (const synonym of epitopeSynonyms.split(', ')) {
90                  await bg.createIdentifierNode(epitopeEntityId, '
                        name', `Synonym`, synonym);
91              }
92          }
93
94          if (antigenName.length > 0) {
```

```
95              let antigenIriType = 'Antigen';
96
97              if (antigenIri.includes('uniprot')) {
98                  antigenIriType = 'UniProt';
99              }
100
101             if (antigenIri.includes('ncbi')) {
102                 antigenIriType = 'NCBI';
103             }
104
105             const antigenEntityId = await bg.createEntityNode('
                    Antigen', `${antigenIri.split('/').slice(-1)[0]}`);
106             await bg.createIdentifierNode(antigenEntityId, 'id', `$
                    {antigenIriType} ID`, `${antigenIri.split('/').slice
                    (-1)[0]}`);
107             await bg.createIdentifierNode(antigenEntityId, 'url', `
                    ${antigenIriType} URL`, `${antigenIri}`);
108             await bg.createEntityEdge(epitopeEntityId,
                    antigenEntityId, 'FROM', {});
109
110             const proteinEntityId = await bg.createEntityNode('
                    Protein', `${parentProteinIri.split('/').slice(-1)
                    [0]}`);
111             await bg.createIdentifierNode(proteinEntityId, 'id', `$
                    {antigenIriType} ID`, `${parentProteinIri.split('/')
                    .slice(-1)[0]}`);
112             await bg.createIdentifierNode(proteinEntityId, 'id', '
                    UniProt ID', `${parentProteinIri.split('/').slice
                    (-1)[0]}`);
113             await bg.createIdentifierNode(proteinEntityId, 'url', '
                    UniProt URL', `${parentProteinIri}`);
114             await bg.createIdentifierNode(proteinEntityId, 'name',
                    'Protein Name', `${parentProtein}`);
115             await bg.createEntityEdge(proteinEntityId,
                    antigenEntityId, 'HAS_ROLE', { relationDetails: '
                    PARENT_PROTEIN' });
116
117             const organismEntityId = await bg.createEntityNode('
                    Organism', `${parentOrganismIri.split('NCBITaxon_')
                    [1]}`);
118             await bg.createIdentifierNode(organismEntityId, 'id', `
                    Taxon ID`, `${parentOrganismIri.split('NCBITaxon_')
                    [1]}`);
119             await bg.createIdentifierNode(organismEntityId, 'url',
                    `Taxon URL`, `${parentOrganismIri}`);
120             await bg.createIdentifierNode(organismEntityId, 'url',
                    `Taxon Name`, `${parentOrganism}`);
121
122         }
123
124         return epitopeEntityId;
125     }
126
127
128     async run() {
129         const { bg } = this;
```

```
130
131         // Start new import
132         await bg.beginImport(this.importer, this.importerVersion,
                this.dataSource);
133
134         console.log('Importing data from IEDB...');
135
136
137
138         // Load data
139         const antigenCsvFile = fs.readFileSync('./importers/local-
                data/iedb-antigen.csv', 'utf8');
140         const { data: antigenData } = Papa.parse(antigenCsvFile.
                split('\n').slice(1).join('\n'), { header: true });
141
142         console.log('IEDB: Processing antigen data');
143
144         let i = 0;
145         for (const antigen of antigenData) {
146             i += 1;
147
148             const antigenName = antigen['Antigen Name'];
149             const rawId = antigen['Antigen ID'];
150             if (rawId == null || rawId.length == 0) {
151                 continue;
152             }
153
154             const antigenId = rawId.split('/').slice(-1)[0];
155             const organismName = antigen['Organism Name'];
156             const organismTaxonId = antigen['Organism ID'].split('
                    NCBITaxon_')[1];
157
158             let idType = 'Protein';
159             if (rawId.includes('uniprot')) {
160                 idType = 'UniProt';
161             }
162             if (rawId.includes('allergen')) {
163                 idType = 'Alergen.org';
164             }
165             if (rawId.includes('ontology.iedb')) {
166                 idType = 'IEDB ontology';
167             }
168
169             // console.log(organismNCBIId);
170             // break;
171
172             if (i % 1000 == 0 && i > 0) {
173
174                 console.log(`IEDB: ${i}/${antigenData.length} (
                        Antigen)`);
175             }
176
177           if (i % 10000 == 0 && i > 0) {
178                 await bg.closeBatch();
179           }
180
```

```
181             const antigenEntityId = await bg.createEntityNode('
                    Antigen', antigenId);
182             await bg.createIdentifierNode(antigenEntityId, 'url', `
                    ${idType} URL`, rawId);
183             await bg.createIdentifierNode(antigenEntityId, 'id', `${
                    idType} ID`, `${antigenId}`);
184             await bg.createIdentifierNode(antigenEntityId, 'name',
                    'Antigen Name', `${antigenName}`);
185
186             const proteinEntityId = await bg.createEntityNode('
                    Protein', antigenId);
187             await bg.createIdentifierNode(proteinEntityId, 'url',
                    `${idType} URL`, rawId);
188             await bg.createIdentifierNode(proteinEntityId, 'id', `${
                    idType} ID`, `${antigenId}`);
189             await bg.createIdentifierNode(proteinEntityId, 'name',
                    'Antigen Name', `${antigenName}`);
190
191             await bg.createEntityEdge(proteinEntityId,
                    antigenEntityId, 'HAS_ROLE', {});
192
193             if (organismTaxonId != null) {
194                 const organismEntityId = await bg.createEntityNode(
                        'Organism', organismTaxonId);
195
196                 await bg.createIdentifierNode(organismEntityId, 'id
                        ', 'Taxon ID', organismTaxonId);
197                 await bg.createIdentifierNode(organismEntityId, 'id
                        ', 'Taxon Name', organismName);
198                 await bg.createDataNode(organismEntityId, 'iedb', {
                         species: organismTaxonId });
199
200                 await bg.createEntityEdge(antigenEntityId,
                        organismEntityId, 'FROM', {});
201                 await bg.createEntityEdge(proteinEntityId,
                        organismEntityId, 'FROM', {});
202             }
203         }
204
205         const epitopeData = await this.parseCSV('./importers/local-
                data/iedb-epitope.csv', 2);
206
207         console.log(`IEDB: ${antigenData.length}/${antigenData.
                length}`);
208         await bg.closeBatch();
209
210         console.log('IEDB: Processing epitope data');
211         i = 0;
212         for (const epitope of epitopeData) {
213             const [
214                 epitopeIri,
215                 objectType,
216                 description,
217                 epitopeModifiedResidues,
218                 epitopeModifications,
219                 startingPosition,
```

```
                endingPosition,
                nonPeptidicEpitopeIri,
                epitopeSynonyms,
                antigenName,
                antigenIri,
                parentProtein,
                parentProteinIri,
                organismName,
                organismIri,
                parentOrganism,
                parentOrganismIri,
                epitopeComments,
                relatedObjectEpitopeRelationship,
                relatedObjectType,
                relatedObjectDescription,
                relatedObjectStartingPosition,
                relatedObjectEndingPosition,
                relatedObjectNonPeptidicEpitopeIri,
                relatedObjectSynonyms,
                relatedObjectAntigenName,
                relatedObjectAntigenIri,
                relatedObjectParentProtein,
                relatedObjectParentProteinIri,
                relatedObjectOrganismName,
                relatedObjectOrganismIri,
                relatedObjectParentOrganism,
                relatedObjectParentOrganismIri,

            ] = epitope;
            i += 1;
            // if (i < 40000) { continue; }


            if (i % 10000 == 0 && i > 0) {

                console.log(`IEDB: ${i}/${epitopeData.length} (
                    Epitope)`);
            }

            if (i % 10000 == 0 && i > 0) {
                await bg.closeBatch();
            }

            await this.processEpitope(
                bg,
                epitopeIri,
                objectType,
                description,
                epitopeModifiedResidues,
                epitopeModifications,
                startingPosition,
                endingPosition,
                nonPeptidicEpitopeIri,
                epitopeSynonyms,
                antigenName,
                antigenIri,
```

```
                parentProtein,
                parentProteinIri,
                parentOrganism,
                parentOrganismIri,
                epitopeComments,
            );
        }
        // Finish and commit the import
        await bg.finishImport();
        console.log('IEDB import complete');
    }
}

module.exports = IEDBImporter;
```

Listing A.4: IEDB data importer script

A.5 DisGeNET data importer

```
const fs = require('fs');
const tabletojson = require('tabletojson').Tabletojson;


class TantigenImporter {
    constructor(bg) {
        this.bg = bg;
        this.importer = 'tantigen';
        this.importerVersion = '1.0';
        this.dataSource = 'Tantigen';
        console.log('Tantigen importer loaded');
    }

    downloadTable(antigenId) {
        const url = 'http://projects.met-hilab.org/tadb/cgi/
            displayAntigen.pl';

    return new Promise((resolve, reject) => {
      tabletojson.convertUrl(
        `${url}?ACC=${antigenId}`,
        (tablesAsJson) => {
            if (tablesAsJson[1]) {
              fs.writeFileSync(`/tmp/tantigen/${antigenId}.json`,
                  JSON.stringify(tablesAsJson[1], null, 2), { flag :
                   'w'});
            }
            resolve(tablesAsJson[1]);
        })
      })
  }

    async run() {
        try {
            const { bg } = this;

```

```
33              // Start new import
34              await bg.beginImport(this.importer, this.
                    importerVersion, this.dataSource);
35
36              console.log('Importing data from Tantigen...');
37
38              // for (let i = 1; i <= 4507; i+= 1) {
39              //     try {
40              //         await this.downloadTable(`Ag${`${i}`.
                    padStart(6, '0')}`)
41              //     } catch (err) {
42              //         console.log(`Skipping Ag${`${i}`.padStart(6,
                     '0')}`);
43              //     }
44              // }
45
46              const files = fs.readdirSync('./importers/local-data/
                    tantigen');
47              const organismEntityId = await bg.createEntityNode('
                    Organism', `9606`);
48              await bg.createIdentifierNode(organismEntityId, 'id', '
                    NCBI ID', '9606');
49
50              let i = 0;
51              for (const filename of files) {
52                  // const num = `${i}`;
53                  // const antigenId = `Ag${num.padStart(6, '0')}`;
54                  const antigenId = filename.split('.')[0];
55                  const file = fs.readFileSync(`./importers/local-
                        data/tantigen/${filename}`, { encoding: 'utf8'})
                        ;
56                  const table = JSON.parse(file);
57
58                  i += 1;
59
60                  if (i % 100 == 0 && i > 0) {
61                      console.log(`Tantigen: ${i}/${files.length}`);
62                      await bg.closeBatch();
63                  }
64
65                  // if (i == 6) {
66                  //     break;
67                  // }
68
69                  const antigenEntityId = await bg.createEntityNode('
                        Antigen', antigenId);
70                  await bg.createIdentifierNode(antigenEntityId, 'id'
                        , 'Antigen ACC', antigenId);
71
72                  // objects.add(antigenEntityId);
73                  let geneEntityId = null;
74                  let rootGeneEntityId = null;
75
76                  // epitopes = null;
77                  // hlaLigands = null;
78
```

```
79                  const antigenData = {};
80
81                  for (const row of table) {
82                      const key = row[0];
83
84                      if (key == 'Antigen Name') {
85                          const antigenName = row[1].trim();
86
87                          rootGeneEntityId = await bg.
                               createEntityNode('Gene', antigenName);
88                          geneEntityId = await bg.createEntityNode('
                               Gene', `${antigenName}-9606`);
89                          await bg.createIdentifierNode(geneEntityId,
                                'name', 'Gene Name', antigenName);
90
91                          await bg.createEntityEdge(geneEntityId,
                               rootGeneEntityId, 'IS_INSTANCE', {});
92                          await bg.createEntityEdge(geneEntityId,
                               organismEntityId, 'FROM', {});
93                          await bg.createEntityEdge(geneEntityId,
                               antigenEntityId, 'HAS_ROLE');
94                      }
95
96                      if (key == 'Common Name') {
97                          const commonName = row[1].trim();
98                          await bg.createIdentifierNode(geneEntityId,
                                'name', 'Common Name', commonName);
99                      }
100
101                     if (key == 'Full name') {
102                         antigenData.fullName = row[1];
103                     }
104
105                     if (key == 'Comment') {
106                         antigenData.comment = row[1];
107                     }
108
109                     if (key == 'Synonym') {
110                         const synonyms = row[1].split('\n')[0].
                               split('another')[0].split(';').map(el =>
                                el.trim());
111
112                         for (const synonym of synonyms) {
113                             await bg.createIdentifierNode(
                                    antigenEntityId, 'name', 'Synonym
                                    Name', synonym);
114                         }
115                     }
116
117                     if (key == 'UniProt ID') {
118                         const uniprotId = row[1].trim();
119                         // proteins.add(uniprotId);
120                         const proteinEntityId = await bg.
                               createEntityNode('Protein', uniprotId);
121
122                         // objects.add(proteinEntityId);
```

```
123
124                         await bg.createIdentifierNode(
                               proteinEntityId, 'id', 'UniProt ID',
                               uniprotId);
125                         await bg.createIdentifierNode(
                               antigenEntityId, 'id', 'UniProt ID',
                               uniprotId);
126
127                         await bg.createEntityEdge(proteinEntityId,
                               antigenEntityId, 'HAS_ROLE');
128                         await bg.createEntityEdge(proteinEntityId,
                               organismEntityId, 'FROM', {});
129
130                         if (geneEntityId != null) {
131                             await bg.createEntityEdge(
                                   proteinEntityId, geneEntityId, 'FROM
                                   ', {});
132                         }
133                     }
134
135                     if (key == 'NCBI Gene ID') {
136                         const geneId = row[1].trim();
137                         if (geneEntityId != null) {
138                             await bg.createIdentifierNode(
                                   geneEntityId, 'id', 'NCBI ID',
                                   geneId);
139                             await bg.createIdentifierNode(
                                   rootGeneEntityId, 'id', 'NCBI ID',
                                   geneId);
140                         }
141                     }
142
143                     if (key == 'Annotation') {
144                         const annotation = row[1].trim();
145                         antigenData.annotation = annotation;
146                     }
147
148                     if (key == 'Isoforms') {
149                         const isoforms = row[1].split('Alignment')
                               [0].split('\n')[0].split(',').map(el =>
                               el.trim().split('Alignment')[0]);
150                         for (const isoform of isoforms) {
151                             const isoNodeId = await bg.
                                   createEntityNode('Antigen', isoform)
                                   ;
152
153                             if (antigenEntityId != isoNodeId) {
154                                 await bg.createEntityEdge(
                                       antigenEntityId, isoNodeId, '
                                       IS_VARIANT', { variationType: '
                                       ISOFORM' });
155                             }
156                         }
157                     }
158
159                     if (key == 'Mutation entries') {
```

```
160                         const mutationEntries = row[1].split('view')[0].split(',').map(el => el.trim().split('View')[0]);
161
162                         for (const entry of mutationEntries) {
163                             const mutationNodeId = await bg.createEntityNode('Antigen', entry);
164                             // objects.add(mutationNodeId);
165
166                             if (antigenEntityId != mutationNodeId) {
167                                 await bg.createEntityEdge(antigenEntityId, mutationNodeId, 'IS_VARIANT', { variationType: 'MUTATION'});
168                             }
169                         }
170                     }
171
172
173                     if (key == 'RNA/protein expression profile') {
174                         antigenData.rnaProteinExpressionProfile = row[1];
175                     }
176
177                     if (key == 'T cell epitope') {
178                         // First row / header
179                         if (row[1].trim() != "Epitope sequence") {
180                             const positions = row[2].split('  ');
181                             // const [from, to] = row[2].split('-');
182                             const epitope = {
183                                 sequence: row[1].trim(),
184                                 positions,
185                                 hlaAllele: row[3].trim(),
186                                 epitopeType: 'T-cell',
187                             };
188
189                             const epitopeReferences = `${row[4]}`.split('\n').map(el => el.trim());
190
191                             const epitopeEntityId = await bg.createEntityNode('Epitope', epitope.sequence);
192
193                             await bg.createIdentifierNode(epitopeEntityId, 'sequence', 'Epitope Sequence', epitope.sequence);
194                             await bg.createIdentifierNode(epitopeEntityId, 'id', 'HLA Allele', epitope.hlaAllele);
195
196                             for (const reference of epitopeReferences) {
197                                 await bg.createIdentifierNode(
```

```
                                    epitopeEntityId, 'id', '
                                    Reference ID', reference);
198                             }
199
200                             await bg.createDataNode(epitopeEntityId
                                    , 'Tantigen', epitope);
201                             await bg.createEntityEdge(
                                    antigenEntityId, epitopeEntityId, '
                                    CONTAINS', {});
202                         }
203                     }
204
205                     if (key == 'HLA ligand') {
206                         // if (hlaLigands == null) {
207                         //  hlaLigands = true;
208                         if (row[1] != 'Ligand Sequence' && row[1]
                                != 'Predicted HLA binders' && row[1] !=
                                'Antigen sequence') {
209                             const positions = row[2].split('  ');
210
211                             // console.log(row);
212                             const hlaLigand = {
213                                 sequence: row[1].trim(),
214                                 positions,
215                                 hlaAllele: row[3].trim(),
216                                 type: 'HLA ligand'
217                             }
218
219                             const ligandReferences = `${row[4]}`.
                                    split('\n').map(el => el.trim());
220
221                             const ligandEntityId = await bg.
                                    createEntityNode('HLA_Ligand',
                                    hlaLigand.sequence);
222
223                             await bg.createIdentifierNode(
                                    ligandEntityId, 'sequence', 'Ligand
                                    Sequence', hlaLigand.sequence);
224
225                             for (const reference of
                                    ligandReferences) {
226                                 await bg.createIdentifierNode(
                                        ligandEntityId, 'id', 'Reference
                                         ID', reference);
227                             }
228
229                             await bg.createDataNode(ligandEntityId,
                                     'Tantigen', hlaLigand);
230                             await bg.createEntityEdge(
                                    antigenEntityId, ligandEntityId, '
                                    CONTAINS', {});
231                         }
232                     }
233                 }
234             }
235
```

```
236             console.log(`Tantigen: ${files.length}/${files.length
                    }`)
237
238             // Finish and commit the import
239             await bg.finishImport();
240             console.log('Tantigen import complete');
241         } catch (err) {
242             console.log(err);
243             throw err;
244         }
245     }
246 }
247
248 module.exports = TantigenImporter;
```

Listing A.5: Tantigen data importer script

Биографија аутора

Александар Вељковић рођен је 23. септембра 1992. године у Пожаревцу. Завршио је основну школу „Бранко Радичевић" у Голупцу као носилац Вукове дипломе. Гимназију општег смера у Великом Градишту завршава као ученик генерације и носилац Вукове дипломе. Основне академске студије на Математичком факултету, Универзитета у Београду, смер информатика, уписао је 2011. године и завршио 2014. године са просечном оценом 9,73. Мастер академске студије на Математичком факултету, смер информатика, уписао је 2014. године и завршио 2016. са просечном оценом 9,69, одбрањеном мастер тезом под називом „Нова метода за асемблирање генома на основу PFG електрофорезе" под менторством проф. др Ненада Митића. Докторске студије на Математичком факултету уписао је 2016. године и положио све испите са просечном оценом 10,00. Од 2015. до 2017. године био је запослен као сарадник у настави на Математичком факултету, Универзитета у Београду, на катедри за рачунарство и информатику. Од 2017. изабран је у звање асистента на катедри за рачунарство и информатику Математичког факултета. Области интересовања су му истраживање података, биоинформатика и криптографија.

Прилог 1.

Изјава о ауторству

Потписани-а _____

број индекса _____

Изјављујем

да је докторска дисертација под насловом

_____

_____

- резултат сопственог истраживачког рада,
- да предложена дисертација у целини ни у деловима није била предложена за добијање било које дипломе према студијским програмима других високошколских установа,
- да су резултати коректно наведени и
- да нисам кршио/ла ауторска права и користио интелектуалну својину других лица.

Потпис докторанда

У Београду, _____

_____

Прилог 2.

Изјава о истоветности штампане и електронске верзије докторског рада

Име и презиме аутора _____

Број индекса _____

Студијски програм _____

Наслов рада _____

Ментор _____

Потписани/а _____

Изјављујем да је штампана верзија мог докторског рада истоветна електронској верзији коју сам предао/ла за објављивање на порталу **Дигиталног репозиторијума Универзитета у Београду.**

Дозвољавам да се објаве моји лични подаци везани за добијање академског звања доктора наука, као што су име и презиме, година и место рођења и датум одбране рада.

Ови лични подаци могу се објавити на мрежним страницама дигиталне библиотеке, у електронском каталогу и у публикацијама Универзитета у Београду.

Потпис докторанда

У Београду, _____

_____

Прилог 3.

Изјава о коришћењу

Овлашћујем Универзитетску библиотеку „Светозар Марковић“ да у Дигитални репозиторијум Универзитета у Београду унесе моју докторску дисертацију под насловом:

_____

_____

која је моје ауторско дело.

Дисертацију са свим прилозима предао/ла сам у електронском формату погодном за трајно архивирање.

Моју докторску дисертацију похрањену у Дигитални репозиторијум Универзитета у Београду могу да користе сви који поштују одредбе садржане у одабраном типу лиценце Креативне заједнице (Creative Commons) за коју сам се одлучио/ла.

1. Ауторство

2. Ауторство - некомерцијално

3. Ауторство – некомерцијално – без прераде

4. Ауторство – некомерцијално – делити под истим условима

5. Ауторство – без прераде

6. Ауторство – делити под истим условима

(Молимо да заокружите само једну од шест понуђених лиценци, кратак опис лиценци дат је на полеђини листа).

Потпис докторанда

У Београду, _____

_____

1. Ауторство - Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, и прераде, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце, чак и у комерцијалне сврхе. Ово је најслободнија од свих лиценци.

2. Ауторство – некомерцијално. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, и прераде, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце. Ова лиценца не дозвољава комерцијалну употребу дела.

3. Ауторство - некомерцијално – без прераде. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, без промена, преобликовања или употребе дела у свом делу, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце. Ова лиценца не дозвољава комерцијалну употребу дела. У односу на све остале лиценце, овом лиценцом се ограничава највећи обим права коришћења дела.

4. Ауторство - некомерцијално – делити под истим условима. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, и прераде, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце и ако се прерада дистрибуира под истом или сличном лиценцом. Ова лиценца не дозвољава комерцијалну употребу дела и прерада.

5. Ауторство – без прераде. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, без промена, преобликовања или употребе дела у свом делу, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце. Ова лиценца дозвољава комерцијалну употребу дела.

6. Ауторство - делити под истим условима. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, и прераде, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце и ако се прерада дистрибуира под истом или сличном лиценцом. Ова лиценца дозвољава комерцијалну употребу дела и прерада. Слична је софтверским лиценцама, односно лиценцама отвореног кода.