Solving Bioinformatics Scenarios with Apache Tinkerpop Gremlin

A. Messina

Rapporto Tecnico N.:
RT-ICAR-PA-18-04

Aprile 2018

Consiglio Nazionale delle Ricerche, Istituto di Calcolo e Reti ad Alte Prestazioni (ICAR)
– Sede di Cosenza, Via P. Bucci 41C, 87036 Rende, Italy, URL: *www.icar.cnr.it*
– Sede di Napoli, Via P. Castellino 111, 80131 Napoli, URL: *www.na.icar.cnr.it*
– Sede di Palermo, Viale delle Scienze, 90128 Palermo, URL: *www.pa.icar.cnr.it*

Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

Solving Bioinformatics Scenarios with Apache Tinkerpop Gremlin

A. Messina[1]

Rapporto Tecnico N.:
RT-ICAR-PA-18-04

Aprile 2018

---

[1] Istituto di Calcolo e Reti ad Alte Prestazioni, ICAR-CNR, Sede di Palermo, Via Ugo La Malfa n.153, 90146 Palermo.

I rapporti tecnici dell'ICAR-CNR sono pubblicati dall'Istituto di Calcolo e Reti ad Alte Prestazioni del Consiglio Nazionale delle Ricerche. Tali rapporti, approntati sotto l'esclusiva responsabilità scientifica degli autori, descrivono attività di ricerca del personale e dei collaboratori dell'ICAR, in alcuni casi in un formato preliminare prima della pubblicazione definitiva in altra sede.

Index

1 INTRODUCTION TO PROPERTY GRAPHS 5
1.1 Introduction 5
1.2 Elements of a property graph 5
1.3 Apache TinkerPop and Gremlin 5
2 BIOGRAPH AND BIOGRAPHDB 7
2.1 Introduction 7
2.2 Graph data model 7
2.2.1 Nodes labels and their properties 8
2.2.2 Edges labels and their properties 10
2.3 Templates and Scenarios in BioGraph 11
3 BASIC TRAVERSALS 12
3.1 Introduction 12
3.2 Finding vertices 12
3.3 Returning property values 13
3.4 Counting 14
3.5 Counting groups of things 14
3.6 Walking the graph 16
3.7 Testing values and ranges of values 17
3.8 Other useful Gremlin steps 18
3.8.1 dedup() 18
3.8.2 fold() 19
3.8.3 order() 19
3.8.4 group() and cap() 20
4 SOLVING SCENARIOS 21

4.1 Introduction 21

4.2 Genes associated to a GO annotation 21

4.3 Proteins in a pathway 21

4.4 Genes associated to a pathway 22

4.5 GO annotations for a gene 22

4.6 Pathways associated to a gene 23

4.7 GO annotations for a protein 23

4.8 Cancers associated to a miRNA 23

4.9 Validated interactions miRNA-Genes 24

4.10 miRNA functional analysis in cancer 25

4.11 miRNA-SNP functional analysis in cancer 25

4.12 Cancer involved miRNAs by pathway 26

5 REFERENCES 27

1 Introduction to Property Graphs

1.1 Introduction

A graph is a data structure composed of vertices (nodes, dots) and edges (arcs, lines). When modeling a graph in a computer and applying it to modern data sets and practices, the generic mathematically-oriented, binary graph is extended to support both labels and key/value properties. This structure is known as a *property graph*. More formally, it is a directed, binary, attributed multi-graph.

1.2 Elements of a property graph

A graph's structure is the topology formed by the explicit references between its vertices, edges, and properties:

- A vertex has incident edges
- A vertex is adjacent to another vertex if they share an incident edge
- A property is attached to an element and an element has a set of properties
- A property is a key/value pair, where the key is always a character *String*.

1.3 Apache TinkerPop and Gremlin

Apache TinkerPop [1] is a graph computing framework and top-level project hosted by the Apache Software Foundation. The project includes the following main components:

- **Gremlin**: a graph traversal query language;
- **Gremlin Console**: an interactive shell for working with local or remote graphs;
- **Gremlin Server**: allows hosting of graphs remotely via an HTTP/WebSocket connection.

Gremlin is composed of three interacting components: a graph, a traversal, and a set of traversers. The traversers move about the graph according to the instructions specified in the traversal, where the result of the computation is the ultimate locations of all halted traversers.

A Gremlin machine can be executed over any supporting graph computing system such as an OLTP graph database and/or an OLAP graph processor.

Gremlin, as a graph traversal language, is a functional language implemented in the user's native programming language and is used to define the traversal of a Gremlin machine.

2 BioGraph and BioGraphDB

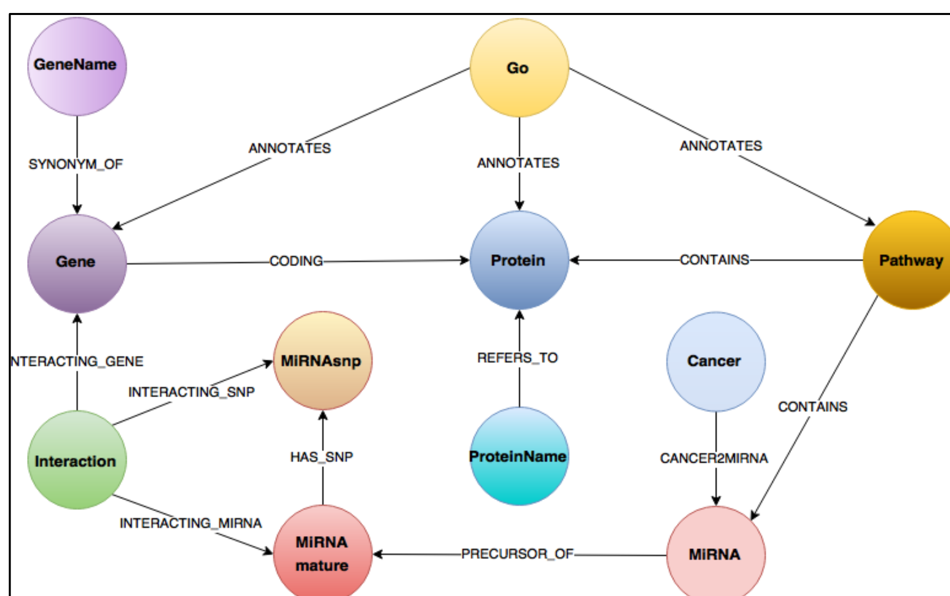
2.1 Introduction

BioGraph [2] is a web application that allows to query, visualize and analyze biological data belonging to several online available sources. BioGraph is built upon a previously developed graph database called BioGraphDB [3] [4] [5] [6] [7], that integrates and stores heterogeneous biological resources and make them available by means of a common structure and a unique query language. BioGraph implements state-of-the-art technologies and provides pre-compiled bioinformatics scenarios, as well as the possibility to perform custom queries and obtaining an interactive and dynamic visualization of results.

2.2 Graph data model

Graph data modeling is the process in which an arbitrary domain is described as a connected graph of nodes and relationships.

In our data sources, almost all entities and references are already well identified. Therefore, it is quite easy to give an abstract representation of BioGraphDB database, as shown below.



A simple general rule has been followed: any biological entity has been mapped into a node with attributes, and a relationship between two biological entities has been mapped into a relation. According to the nature of the entities, nodes and relations have been grouped into classes, each identified by a label. For example,

all the genes imported from Entrez Gene become nodes identified by the label Gene and all the proteins read from the Uniprot Knowledge Base become nodes identified by the label Protein. At this point, the relation CODING between genes and proteins can be created using the information on this relationship from HGNC.

The following table summarizes all associations between the biological information and the created graph entities.

Graph entity	*Label*	*Biological information*	*Source*
vertices	Gene	genes	NCBI Entrez Genes
	Go	functional annotations	Gene Ontology
	Protein	proteins	UniProtKB
	Pathway	pathways	Reactome
	MiRNA	miRNA precursors	miRBase
	MiRNAmature	miRNA matures	miRBase
	MiRNAsnp	miRNA SNPs	miRNASNP
	Cancer	cancers	mirCancer
	ProteinName	proteins accessions	UniProtKB
	GeneName	genes symbols	HGNC
	Interaction	miRNA-target interactions	mirTarBase, miRanda
edges	ANNOTATES	links to annotated entities	Gene Ontology
	CONTAINS	links to entities contained in pathways	Reactome
	PRECURSOR_OF	precursors-matures relations	miRBase
	HAS_SNP	miRNAs-mutations relations	miRNASNP
	SYNONYM_OF	symbols-genes relations	HGNC
	REFERS_TO	accessions-proteins relations	UniProtKB
	INTERACTING_GENE	genes-interactions relations	mirTarBase, miRanda, miRNASNP
	INTERACTING_MIRNA	miRNA-interactions relations	mirTarBase, miRanda
	INTERACTING_SNP	SNPs-interactions relations	miRNASNP

At present, BioGraphDB contains about 1,450,000 nodes and 2,820,000 relations.

2.2.1 Nodes labels and their properties

The following table summarizes the names and types of all properties of the nodes in BioGraphDB, grouped by the node label:

Nodes	Properties	Type
Gene	geneId	String
	locusTag	String
	chromosome	String
	mapLocation	String
	description	String
	type	String
	nomenclatureAuthoritySymbol	String
	nomenclatureAuthorityFullName	String
	nomenclatureStatus	String
	otherDesignations	String
GeneName	symbol	String

Go	goId	String
	name	String
	namespace	String
	definition	String
	obsolete	String
	comment	String
Protein	name	String
	fullName	String
	alternativeName	String
	gene	String
	sequence	String
	sequenceLenght	Int
	sequenceMass	Int
ProteinName	name	String
Pathway	pathwayId	String
	name	String
	summation	String
Cancer	name	String
MiRNA	accession	String
	name	String
	description	String
	comment	String
	sequence	String
MiRNAmature	accession	String
	name	String
	description	String
	comment	String
	location	String
	sequence	String
MiRNASNP	SNPid	String
	miRNA	String
	chr	String
	miRstart	Int
	miRend	Int
	lostNum	Int
	gainNum	Int
Interaction	transcriptId	String
	extTranscriptId	String
	mirAlignment	String
	alignment	String
	geneAlignment	String
	mirStart	Int
	mirEnd	Int
	geneStart	Int

	geneEnd	Int
	genomeCoordinates	String
	conservation	Double
	alignScore	Int
	seedCat	Int
	energy	Double
	mirSvrScore	String
	mirTarBaseId	String
	experiments	String
	supportType	String
	snpEnergy	Double
	basePair	String
	geneAve	Double
	mirnaAve	Double
	database	String

2.2.2 Edges labels and their properties

The following table summarizes the names and types of all properties of the edges in BioGraphDB, grouped by the edge label:

Edges	Properties	Type
ANNOTATES	evidence qualifier category	String String String
SYNONYM_OF	-	-
CODING	-	-
CONTAINS	-	-
REFERS_TO	-	-
CANCER2MIRNA	profile	String
PRECURSOR_OF	-	-
HAS_SNP	-	-
INTERACTING_GENE	-	-
INTERACTING_MIRNA	-	-
INTERACTING_SNP	-	-

2.3 Templates and Scenarios in BioGraph

The web user interface of BioGraph contains two useful tabs the user can select to query the graph database and to analyze the results:

- ***Templates*** proposes a set of simple predefined queries, grouped by the following categories: Functions, Genes, Proteins, and miRNAs. Each template accepts one or more parameters and the Execute button sends the related query to the Gremlin Workbench for execution.

- ***Scenarios*** contains, at present, four predefined complex queries, proposed as example of how BioGraph and Gremlin can help in the analysis of specific non-trivial problems. The available scenarios are shown in the following figure:

3 Basic Traversals

3.1 Introduction

A graph query is often referred to as a *traversal* [8] as that is what we are in fact doing. We are traversing the graph from a starting point to an ending point. Traversals consist of one or more *steps* (essentially methods) that are chained together.

As we start to look at some simple traversals here are a few steps that you will see used a lot. Firstly, you will notice that almost all traversals start with either a *g.V()* or a *g.E()*.

The V step returns vertices and the E step returns edges. You can also use a V step in the middle of a traversal as well as at the start but we will examine those uses a little later. The V and E steps can also take parameters indicating which set of vertices or edges we are interested in.

The other steps we need to introduce are the *has* and *hasLabel* steps. They can be used to test for a certain label or property having a certain value.

3.2 Finding vertices

Below are some simple queries against the *BioGraphDB* graph to get us started. The query below will return any vertices (nodes) that have the gene label.

```
// Find vertices that are genes
gremlin> g.V().hasLabel('Gene')
```

This query will return the vertex that represents the "thioredoxin 2 (TXN2)" gene.

```
// Find the TXN2 vertex
gremlin> g.V().has('nomenclatureAuthoritySymbol','TXN2')
```

The next two queries combine the previous two into a single query.

The first one just chains the queries together. The second shows a form of the *has* step that we have not looked at before that takes an additional label value as its first parameter.

```
// Combining those two previous queries (two ways that are equivalent)
gremlin> g.V().hasLabel('Gene').has('nomenclatureAuthoritySymbol','TXN2')

gremlin> g.V().has('Gene','nomenclatureAuthoritySymbol','TXN2')
```

3.3 Returning property values

There are several different ways of working with vertex properties. We can add, delete and query properties for any vertex or edge in the graph. Initially, let's look at a couple of simple ways that we can look up the property values of a given vertex.

```
// What property values are stored in the TXN2 vertex?
gremlin> g.V().has('Gene','nomenclatureAuthoritySymbol','TXN2').values()
```

Here is the output that the query returns. Note that we just get back the values of the properties when using the values step, we do not get back the associated keys.

```
==>thioredoxin 2
==>22
==>22q13.1
==>TXN2
==>25828
==>mitochondrial thioredoxin|thioredoxin-2
==>-
==>O
==>thioredoxin 2
==>protein-coding
```

The values step can take parameters that tell it to only return the values for the provided key names. The queries below return the values of some specific properties.

```
// Return just the description property
g.V().has('Gene','nomenclatureAuthoritySymbol','TXN2').values('description')

==>thioredoxin 2

// Return the 'chromosome' and 'type' property values.
g.V().has('Gene','nomenclatureAuthoritySymbol','TXN2').values('chromosome','type')

==>22
==>protein-coding
```

3.4 Counting

A common need when working with graphs is to be able to count how "many of something" there are in the graph. First of all, let's find out how many vertices in the graph represent genes.

```
// How many genes are there in the graph?
gremlin> g.V().hasLabel('Gene').count()

==>59839
```

Now, looking at edges that have a *CODING* label, let's find out how many proteins are encoded by genes in the graph. Note that the *outE* step looks at outgoing edges. In this case we could also have used the out step instead.

```
// How many encoded proteins are there?
gremlin> g.V().hasLabel('Gene').outE('CODING').count()

==>19089
```

You could shorten the above a little as follows but this would cause more edges to get looked as we do not first filter out all vertices that are not genes.

```
// How many encoded proteins are there?
gremlin> g.V().outE('CODING').count()

==>19089
```

You could also do it this way but generally starting by looking at all the edges in the graph is considered bad form as property graphs tend to have a lot more edges than vertices.

```
// How many encoded proteins are there?
gremlin> g.E().hasLabel('CODING').count()

==>19089
```

3.5 Counting groups of things

Sometimes it is useful to count how many of each type (or group) of things there are in the graph. This can be done using the *group* and *groupCount* steps.

While for a very large graph it is not recommended to run queries that look at all

of the vertices or all of the edges in a graph, for smaller graphs this can be quite useful. For the BioGraphDB graph, we could easily count the number of different vertex and edge types in the graph as follows.

```
// How many of each type of vertex are there?
gremlin> g.V().groupCount().by(label)
```

If we were to run the query we would get back a map where the keys are label names and the values are the counts for the occurrence of each label in the graph.

```
==>{MiRNAmature=38558, GeneName=115027, Gene=59839, Pathway=1920, Interaction=913285,
MiRNA=28645, Go=43969, ProteinName=219132, MiRNAsnp=236, Pubmed=76719, Protein=20193,
Cancer=107}
```

There are other ways we could write the query above that will yield the same result. One such example is shown below.

```
// How many of each type of vertex are there?
gremlin> g.V().label().groupCount()

==>{MiRNAmature=38558, GeneName=115027, Gene=59839, Pathway=1920, Interaction=913285,
MiRNA=28645, Go=43969, ProteinName=219132, MiRNAsnp=236, Pubmed=76719, Protein=20193,
Cancer=107}
```

We can also run a similar query to find out the distribution of edge labels in the graph. An example of the type of result we would get back is also shown.

```
// How many of each type of edge are there?
gremlin> g.E().groupCount().by(label)

==>{PRECURSOR_OF=38558,      POSITIVELY_REGULATES=2619,      INTERACTING_GENE=913285,
CODING=19089,     ANNOTATES=514528,     NEGATIVELY_REGULATES=2632,     PART_OF=7447,
MIRNA2PATHWAY=136, CANCER2MIRNA=2668, CONTAINS=99979, IS_A=72124, CITED_IN=631543,
INTERACTING_SNP=255381,   INTERACTING_MIRNA=657904,   HAS_SNP=236,   REGULATES=3053,
SYNONYM_OF=115027, REFERS_TO=219132}
```

As before we could rewrite the query as follows.

```
// How many of each type of edge are there?
gremlin> g.E().label().groupCount()

==>{PRECURSOR_OF=38558,      POSITIVELY_REGULATES=2619,      INTERACTING_GENE=913285,
CODING=19089,     ANNOTATES=514528,     NEGATIVELY_REGULATES=2632,     PART_OF=7447,
MIRNA2PATHWAY=136, CANCER2MIRNA=2668, CONTAINS=99979, IS_A=72124, CITED_IN=631543,
INTERACTING_SNP=255381,   INTERACTING_MIRNA=657904,   HAS_SNP=236,   REGULATES=3053,
SYNONYM_OF=115027, REFERS_TO=219132}
```

By way of a side note, the examples above are shorthand ways of writing something like this example which also counts vertices by label.

```
// As above but using group()
gremlin> g.V().group().by(label).by(count())

==>{MiRNAmature=38558, GeneName=115027, Gene=59839, Pathway=1920, Interaction=913285,
MiRNA=28645, Go=43969, ProteinName=219132, MiRNAsnp=236, Pubmed=76719, Protein=20193,
Cancer=107}
```

3.6 Walking the graph

We have mostly just explored queries that look at properties on a vertex or count how many things we can find of a certain type.

Where the power of a graph really comes into play is when we start to *walk* or *traverse* the graph by looking at the connections (edges) between vertices.

The term *walking the graph* is used to describe moving from one vertex to another vertex via an edge.

Typically, when using the phrase *walking a graph* the intent is to describe starting at a vertex traversing one or more vertices and edges and ending up at a different vertex or sometimes, back where you started in the case of a *circular walk*.

It is very easy to traverse a graph in this way using Gremlin. The journey we took while on our walk is often referred to as our *path*.

There are also cases when all you want to do is return edges or some combination of vertices and edges as the result of a query and Gremlin allows this as well.

The table below gives a brief summary of all the steps that can be used to *walk* or *traverse* a graph using Gremlin. You will find all of these steps used in various ways throughout the book.

Think of a graph traversal as moving through the graph from one place to one or more other places. These steps tell Gremlin which places to move to next as it traverses a graph for you.

In order to better understand these steps, it is worth defining some terminology. One vertex is considered to be *adjacent* to another vertex if there is an edge connecting them. A vertex and an edge are considered *incident* if they are connected to each other.

out	*	outgoing adjacent vertices
in	*	incoming adjacent vertices
both	*	both incoming and outgoing adjacent vertices
outE	*	outgoing incident edges
inE	*	incoming incident edges
bothE	*	both outgoing and incoming incident edges
outV		outgoing vertex
inV		incoming vertex
otherV		the vertex that was not the vertex we came from

Note that the steps labelled with an * can optionally take the name of one or more edge labels as a parameter. If omitted, all relevant edges will be traversed.

3.7 Testing values and ranges of values

Gremlin provides a number of different predicates that we can use to do range testing. The list below provides a summary of the available predicates.

eq	equal to
neq	not equal to
gt	greater than
gte	greater than or equal to
lt	less than
lte	less than or equal to
inside	inside a lower and upper bound (bounds excluded)
outside	outside a lower and upper bound (bounds excluded)
between	between two values (upper bound excluded)
within	must match at least one of the values provided
without	must not match any of the values provided

3.8 Other useful Gremlin steps

In previous sections, we have presented some essential Gremlin steps. However, those steps are not enough to productively query any non-trivial graph. Therefore, in the following subsections, other useful steps are presented, in order to fully understand the Template and Scenarios section of BioGraph.

3.8.1 dedup()

It is often desirable to remove duplicate values from query results. The *dedup* step allows us to do this. In the example below, the name of cancers associated to a given miRNA is queried. Note that in the returned results there are some duplicate values.

```
gremlin> g.V().hasLabel('MiRNA').has('name','hsa-mir-17').
    in('CANCER2MIRNA').values('name')

==>cholangiocarcinoma
==>breast carcinoma
==>breast cancer
==>b-cell lymphoma
==>colorectal carcinoma
==>colorectal cancer
==>colorectal cancer
==>cholangiocarcinoma
==>acute myeloid leukemia
==>malignant melanoma
==>malignant melanoma
==>lung cancer
==>lung cancer
==>nasopharyngeal cancer
==>osteosarcoma
==>mantle cell lymphoma
==>medulloblastoma
==>gastric cancer
==>gastric cancer
==>esophageal squamous cell carcinoma
==>gastric cancer
==>hepatocellular carcinoma
==>lung cancer
==>glioma
==>glioma
==>t-cell lymphoblastic lymphoma
```

If we only wanted a set of unique values in the result we could rewrite the query to include a *dedup* step. This time the query results only include one of each value.

```
// As above but using dedup()
gremlin> g.V().hasLabel('MiRNA').has('name','hsa-mir-17').
    in('CANCER2MIRNA').values('name').dedup()

==>cholangiocarcinoma
```

```
==>breast carcinoma
==>breast cancer
==>b-cell lymphoma
==>colorectal carcinoma
==>colorectal cancer
==>acute myeloid leukemia
==>malignant melanoma
==>lung cancer
==>nasopharyngeal cancer
==>osteosarcoma
==>mantle cell lymphoma
==>medulloblastoma
==>gastric cancer
==>esophageal squamous cell carcinoma
==>hepatocellular carcinoma
==>glioma
==>t-cell lymphoblastic lymphoma
```

3.8.2 fold()

There are situations when the traversal stream needs a "barrier" to aggregate all the objects and emit a computation that is a function of the aggregate. The *fold()* step is one particular instance of this. A parameter-less *fold()* will aggregate all the objects into a list and then emit the list. Applied to the previous example, we obtain the following results:

```
gremlin> g.V().hasLabel('MiRNA').has('name','hsa-mir-17').
    in('CANCER2MIRNA').values('name').dedup().fold()

==>[cholangiocarcinoma, breast carcinoma, breast cancer, b-cell lymphoma, colorectal
carcinoma, colorectal cancer, acute myeloid leukemia, malignant melanoma, lung cancer,
nasopharyngeal cancer, osteosarcoma, mantle cell lymphoma, medulloblastoma, gastric
cancer, esophageal squamous cell carcinoma, hepatocellular carcinoma, glioma, t-cell
lymphoblastic lymphoma]
```

fold() can also be provided two arguments: a seed value and a reduce bi-function. The seed is the value to provide as the first argument to the reduce function. The reduce function is the function to fold by where the first argument is the seed or the value returned from subsequent calls and the second argument is the value from the stream.

3.8.3 order()

You can use *order()* to sort things in either ascending (the default) or descending order. Note that the sort does not have to be the last step of a query. Also, it is perfectly ok to sort things in the middle of a query before moving on to a further step. The last sample query is rewritten as follow:

```
gremlin> g.V().hasLabel('MiRNA').has('name','hsa-mir-17').
    in('CANCER2MIRNA').values('name').dedup().order().fold()

==>[acute myeloid leukemia, b-cell lymphoma, breast cancer, breast carcinoma, cholangiocarcinoma, colorectal cancer, colorectal carcinoma, esophageal squamous cell carcinoma, gastric cancer, glioma, hepatocellular carcinoma, lung cancer, malignant melanoma, mantle cell lymphoma, medulloblastoma, nasopharyngeal cancer, osteosarcoma, t-cell lymphoblastic lymphoma]
```

By default, a sort performed using *order()* is performed in ascending order. If we wanted to sort in descending order instead we can specify *decr* as a parameter to *order()*. We can also specify *incr* if we want to be clear that we intend an ascending order sort.

```
gremlin> g.V().hasLabel('MiRNA').has('name','hsa-mir-17').
    in('CANCER2MIRNA').values('name').dedup().order().by(decr).fold()

==>[t-cell lymphoblastic lymphoma, osteosarcoma, nasopharyngeal cancer, medulloblastoma, mantle cell lymphoma, malignant melanoma, lung cancer, hepatocellular carcinoma, glioma, gastric cancer, esophageal squamous cell carcinoma, colorectal carcinoma, colorectal cancer, cholangiocarcinoma, breast carcinoma, breast cancer, b-cell lymphoma, acute myeloid leukemia]
```

You can also sort things into a random order using *shuffle*. Take a look at the example below and the output it produces.

```
gremlin> g.V().hasLabel('MiRNA').has('name','hsa-mir-17').
    in('CANCER2MIRNA').values('name').dedup().order().by(shuffle).fold()

==>[osteosarcoma, lung cancer, hepatocellular carcinoma, mantle cell lymphoma, breast cancer, malignant melanoma, breast carcinoma, t-cell lymphoblastic lymphoma, nasopharyngeal cancer, colorectal carcinoma, gastric cancer, glioma, colorectal cancer, cholangiocarcinoma, b-cell lymphoma, esophageal squamous cell carcinoma, acute myeloid leukemia, medulloblastoma]
```

3.8.4 group() and cap()

As traversers propagate across a graph as defined by a traversal, “side effect” computations are sometimes required. That is, the actual path taken or the current location of a traverser is not the ultimate output of the computation, but some other representation of the traversal. The *group()* step is one such side effect that organizes the objects according to some function of the object. Then, if required, that organization (a list) is reduced. Note that, when you’re doing *group()*, the map structure is not closed and not fully available for reading because we may accumulate more data into it later in the traversal. The *cap()* step iterates the traversal up to itself and emits the side effect referenced by the provided key.

4 Solving scenarios

4.1 Introduction

In this chapter, the queries from the *Templates* section and the *Scenarios* section of BioGraph are presented and explained. They are showed starting from the simplest and then progressing with those gradually more complex.

4.2 Genes associated to a GO annotation

As first example, let's see how to search for Genes that are associated with a particular Gene Ontology (GO) annotation, e.g., the *3'-5' DNA helicase activity*. Starting from the *GO* node, we simply have to traverse the *ANNOTATES* edges linked to all the outgoing adjacent *Gene* nodes.

```
g.V().hasLabel("Go").has("name","3'-5' DNA helicase activity").
      out("ANNOTATES").
      hasLabel("Gene").dedup().values("nomenclatureAuthoritySymbol").fold()

==>[FBXO18, GINS4, GINS2, GINS1, WRN, CDC45, ERCC3]
```

4.3 Proteins in a pathway

Now, let's look for all proteins contained in a given pathway, e.g., *the Gap-filling DNA repair synthesis and ligation in TC-NER*. Starting from the *Pathway* node, we simply traverse the *CONTAINS* edges. Proteins names are presented in alphabetic order.

```
g.V().hasLabel('Pathway').
      has('name','Gap-filling DNA repair synthesis and ligation in TC-NER').
      out('CONTAINS').
      values('name').order().fold()

==>[AQR_HUMAN, CCNH_HUMAN, CDK7_HUMAN, CUL4A_HUMAN, CUL4B_HUMAN, DDB1_HUMAN,
DNLI1_HUMAN, DNLI3_HUMAN, DPOD1_HUMAN, DPOD2_HUMAN, DPOD3_HUMAN, DPOD4_HUMAN,
DPOE1_HUMAN, DPOE2_HUMAN, EP300_HUMAN, ERCC2_HUMAN, ERCC3_HUMAN, ERCC6_HUMAN,
ERCC8_HUMAN, HMGN1_HUMAN, ISY1_HUMAN, MAT1_HUMAN, PCNA_HUMAN, POLK_HUMAN,
PPIE_HUMAN, PRP19_HUMAN, RBX1_HUMAN, RFA1_HUMAN, RFA2_HUMAN, RFA3_HUMAN, RFC1_HUMAN,
RFC2_HUMAN, RFC3_HUMAN, RFC4_HUMAN, RFC5_HUMAN, RL40_HUMAN, RPAB1_HUMAN,
RPAB2_HUMAN, RPAB3_HUMAN, RPAB4_HUMAN, RPAB5_HUMAN, RPB11_HUMAN, RPB1_HUMAN,
RPB2_HUMAN, RPB3_HUMAN, RPB4_HUMAN, RPB7_HUMAN, RPB9_HUMAN, RS27A_HUMAN, SYF1_HUMAN,
TCEA1_HUMAN, TF2H1_HUMAN, TF2H2_HUMAN, TF2H3_HUMAN, TF2H4_HUMAN, TF2H5_HUMAN,
UBB_HUMAN, UBC_HUMAN, UBP7_HUMAN, UVSSA_HUMAN, XRCC1_HUMAN, ZN830_HUMAN]
```

4.4 Genes associated to a pathway

Genes are linked to pathways indirectly, because we have to traverse proteins nodes first. Therefore, the Gremlin query will contain an extra step to traverse the *CODING* edges:

```
g.V().hasLabel('Pathway').
      has('name','Gap-filling DNA repair synthesis and ligation in TC-NER').
      out('CONTAINS').
      in('CODING').
      values('nomenclatureAuthoritySymbol').order().fold()

==>[AQR, CCNH, CDK7, CUL4A, CUL4B, DDB1, EP300, ERCC2, ERCC3, ERCC6, ERCC8, GTF2H1,
GTF2H2, GTF2H3, GTF2H4, GTF2H5, HMGN1, ISY1, LIG1, LIG3, MNAT1, PCNA, POLD1, POLD2,
POLD3, POLD4, POLE, POLE2, POLK, POLR2A, POLR2B, POLR2C, POLR2D, POLR2E, POLR2F,
POLR2G, POLR2H, POLR2I, POLR2J, POLR2K, POLR2L, PPIE, PRPF19, RBX1, RFC1, RFC2,
RFC3, RFC4, RFC5, RPA1, RPA2, RPA3, RPS27A, TCEA1, UBA52, UBB, UBC, USP7, UVSSA,
XAB2, XRCC1, ZNF830]
```

4.5 GO annotations for a gene

Let’s proceed in reverse order from Par. 4.2: given a gene, let’s look for its functional annotations.

```
g.V().hasLabel('Gene').
      has('nomenclatureAuthoritySymbol','PPARG').
      in('ANNOTATES').
      values('goId').order().fold()

==>[GO:0000122, GO:0000122, GO:0001012, GO:0001046, GO:0001228, GO:0001890,
GO:0002024, GO:0002674, GO:0003677, GO:0003677, GO:0003682, GO:0003700, GO:0003700,
GO:0003707, GO:0004879, GO:0004955, GO:0005515, GO:0005634, GO:0005654, GO:0005794,
GO:0005829, GO:0006367, GO:0006629, GO:0006919, GO:0007165, GO:0007186, GO:0007507,
GO:0007584, GO:0008144, GO:0008217, GO:0008270, GO:0009409, GO:0009612, GO:0010467,
GO:0010745, GO:0010745, GO:0010871, GO:0010887, GO:0010891, GO:0015909, GO:0019395,
GO:0019899, GO:0019903, GO:0030224, GO:0030308, GO:0030331, GO:0030374, GO:0030855,
GO:0031000, GO:0031100, GO:0032526, GO:0032869, GO:0032966, GO:0033189, GO:0033613,
GO:0033993, GO:0035357, GO:0035902, GO:0036270, GO:0042593, GO:0042594, GO:0042752,
GO:0042802, GO:0042953, GO:0043231, GO:0043401, GO:0043565, GO:0043627, GO:0044212,
GO:0044212, GO:0045087, GO:0045165, GO:0045600, GO:0045713, GO:0045892, GO:0045893,
GO:0045944, GO:0045944, GO:0046321, GO:0046965, GO:0048469, GO:0048471, GO:0048511,
GO:0048662, GO:0048714, GO:0050544, GO:0050872, GO:0050872, GO:0050873, GO:0051091,
GO:0051393, GO:0051974, GO:0055088, GO:0055098, GO:0060100, GO:0060336, GO:0060694,
GO:0060850, GO:0071285, GO:0071300, GO:0071306, GO:0071380, GO:0071455, GO:0090575,
GO:1901558, GO:2000230]
```

4.6 Pathways associated to a gene

Again, we proceed in reverse order from Par. 4.4: given a gene, we look for (indirect) associated pathways.

```
g.V().hasLabel('Gene').
      has('nomenclatureAuthoritySymbol','PPARG').
      out('CODING').
      in('CONTAINS').values('name').order()

==>Developmental Biology
==>Fatty acid, triacylglycerol, and ketone body metabolism
==>Gene Expression
==>Generic Transcription Pathway
==>Metabolism
==>Metabolism of lipids and lipoproteins
==>Nuclear Receptor transcription pathway
==>PPARA activates gene expression
==>Regulation of lipid metabolism by Peroxisome proliferator-activated receptor
alpha (PPARalpha)
==>Transcriptional regulation of white adipocyte differentiation
```

4.7 GO annotations for a protein

This is a modification of Par. 4.5: given a protein, we look for its functional annotations.

```
g.V().hasLabel('Protein').
      has('name','PPARG_HUMAN').
      in('ANNOTATES').values('goId').order().fold()

==>[GO:0000122, GO:0001046, GO:0001890, GO:0002674, GO:0003677, GO:0003682,
GO:0003700, GO:0003707, GO:0004879, GO:0004955, GO:0005634, GO:0005654, GO:0005794,
GO:0005829, GO:0006367, GO:0006629, GO:0006919, GO:0007165, GO:0007186, GO:0007507,
GO:0007584, GO:0008144, GO:0008217, GO:0008270, GO:0009409, GO:0009612, GO:0010467,
GO:0010745, GO:0010871, GO:0010887, GO:0010891, GO:0015909, GO:0019395, GO:0019899,
GO:0030224, GO:0030308, GO:0030374, GO:0030855, GO:0031000, GO:0031100, GO:0032526,
GO:0032869, GO:0032966, GO:0033189, GO:0033613, GO:0033993, GO:0035357, GO:0035902,
GO:0036270, GO:0042593, GO:0042594, GO:0042752, GO:0042802, GO:0042953, GO:0043231,
GO:0043627, GO:0044212, GO:0045087, GO:0045165, GO:0045600, GO:0045713, GO:0045892,
GO:0045893, GO:0045944, GO:0046321, GO:0046965, GO:0048469, GO:0048471, GO:0048511,
GO:0048662, GO:0048714, GO:0050544, GO:0050872, GO:0051091, GO:0051393, GO:0051974,
GO:0055088, GO:0055098, GO:0060100, GO:0060336, GO:0060694, GO:0060850, GO:0071300,
GO:0071306, GO:0071380, GO:0071455, GO:0090575, GO:1901558, GO:2000230]
```

4.8 Cancers associated to a miRNA

For a given miRNA, e.g., *has-mir-17*, the following query returns the associated cancers from miRCancer:

```
gremlin> g.V().hasLabel('MiRNA').
      has('name','hsa-mir-17').
      in('CANCER2MIRNA').dedup().values('name').order()

==>acute myeloid leukemia
==>b-cell lymphoma
==>breast cancer
==>breast carcinoma
==>cholangiocarcinoma
==>colorectal cancer
==>colorectal carcinoma
==>esophageal squamous cell carcinoma
==>gastric cancer
==>glioma
==>hepatocellular carcinoma
==>lung cancer
==>malignant melanoma
==>mantle cell lymphoma
==>medulloblastoma
==>nasopharyngeal cancer
==>osteosarcoma
==>t-cell lymphoblastic lymphoma
```

4.9 Validated interactions miRNA-Genes

For a given miRNA mature, e.g., has-miR-148a-3p, the following query walks on validated interaction from miRTarBase and returns the interacting genes:

```
gremlin> g.V().hasLabel('MiRNAmature').
      has('product','hsa-miR-148a-3p').
      in('INTERACTING_MIRNA').has('database','miRTarBase').
      out('INTERACTING_GENE').dedup().
      values('nomenclatureAuthoritySymbol').fold()

==>[ABLIM1, ACVR1, ADARB1, AGO2, AGO3, AKAP17A, AMELX, ANP32A, AP5B1, APC, APPBP2,
ARID3A, ARL8B, ARRDC3, ASB6, AURKB, B4GALT7, BAZ2B, BCL2, BCL2L11, BMP3, BTBD3,
CBX3, CCKBR, CCNA2, CCNI, CCT6A, CDC25B, CDK19, CDKN1B, CEBPG, CEP55, CHRFAM7A,
CNOT4, COLEC12, CYCS, DCUN1D3, DDX6, DENR, DICER1, DNAJB4, DNMT1, DNMT3B, DSTYK,
DTX4, DYNLL2, DYRK1A, EOGT, ERRFI1, ETV7, FAM104A, FAM212B, FOXP1, FZD5, GAS1,
GLRX5, GNB5, GOLIM4, GPATCH8, GPRC5A, HCCS, HLA-A, HLA-C, HLA-G, HMGB1, HOXC8,
HSP90AA1, HSP90B1, HSPA4, IGFBP5, IL23R, IRS1, ITGA5, ITGB8, JARID2, KANSL1,
KIAA0907, KIAA1549, KIF2C, KLF6, KPNA4, LBR, LNPEP, LYSMD1, MAFB, MAP3K4, MAP3K9,
MET, MLEC, MMP7, MPP5, MRPL45, MRPS27, MSL3, MTMR9, MYC, MYCBP2, MYO3A, NDRG1, NONO,
NPTX1, NR1I2, OR2C3, OTUD4, OVOL1, PAN3, PAPD4, PATL1, PBXIP1, PDIK1L, PHACTR2,
PLA2G12A, POC1A, POFUT1, PPP6R1, PRNP, PSMD9, PTPN23, PTPN4, RAB10, RAB12, RAB14,
RAB1B, RAB34, RASSF8, RBM23, RBM38, RCC2, RFT1, RNF219, ROCK1, RPS17, RPS6KA4,
RPS6KA5, RUNX3, S1PR1, S1PR2, SECISBP2L, SERPINE1, SESN3, SESTD1, SH3PXD2A, SIK1,
SLC12A7, SLC25A3, SLC2A3, SLC38A2, SMAD2, SNAPIN, SORD, SOS2, SPRY2, STARD13, STX16,
STX6, TGIF2, TMED7, TMEM14A, TMEM9B, TNRC6A, TNRC6B, TRIM59, TTLL1, TXNIP, UBE2D3,
UNKL, UQCRQ, USP38, USP4, VAV2, VGLL2, VPS37A, VPS37B, VPS41, WASL, WDTC1, WNT10B,
WNT2B, YPEL1, YWHAB, ZDHHC6, ZFYVE26, ZIC5, ZNF490, ZNF92]
```

4.10 miRNA functional analysis in cancer

The query investigates the functional role of miRNAs in cancer pathology.

Wild-type differentially expressed (DE) miRNAs in a specific cancer disease are investigated as regulative elements of gene targets through interaction analysis. At this point an energy filter is applied according to the free energy score of the binding site predicted by miRanda. This allows to highlight only miRNA-target interactions that are strongly bound.

The evidenced targets are then analyzed through GO enrichment, to show the functional annotations that link these molecules to the selected cancer disease.

```
gremlin> g.V().hasLabel('Cancer').
      has('name', 'colorectal cancer').
      out('CANCER2MIRNA').dedup().
      out('PRECURSOR_OF').
      in('INTERACTING_MIRNA').has('database','miRanda').has('energy',lt(-34)).
      out('INTERACTING_GENE').dedup().
      group('genes').
      by('nomenclatureAuthoritySymbol').
      by(__.in('ANNOTATES').dedup().values('goId').fold()).
      cap('genes')

==>{CSF1R=[GO:0001934, GO:1990682, GO:0019955, GO:0071345, GO:0090197, GO:0045217,
GO:0045087, GO:0048015, GO:0008284, GO:0008283, GO:0008285, GO:0005524, GO:0031529,
GO:0007519, GO:0006954, GO:0071902, GO:0036006, GO:0005011, GO:0045124, GO:0038145,
GO:0030316, GO:0061098, GO:0021879, GO:0009986, GO:0007411, GO:0045672, GO:0070374,
GO:0007275, GO:0030335, GO:0021772, GO:0046488, GO:0060603, GO:0046777, GO:0008360,
GO:2000249, GO:0005886, GO:0030224, GO:0005887, GO:0030225, GO:0007165, GO:0019903,
GO:0019221, GO:0042517, GO:0030097, GO:0007169, GO:0018108, GO:0043066, GO:0005622,
GO:0042803, GO:2000147], SUFU=[GO:0003714, GO:0003281, GO:0035904, GO:0043588,
GO:0060976, GO:0007275, GO:0008134, GO:0006355, GO:0001947, GO:0005654, GO:0006508,
GO:0005515, GO:0021776, GO:0021775, GO:0005737, GO:1901621, GO:2000059, GO:0008013,
GO:0019901, GO:0007165, GO:0042992, GO:0000122, GO:0042994, GO:0001843, GO:0005829,
GO:0043433, GO:0004871, GO:0001501, GO:0097542, GO:0045879, GO:0045668, GO:0072372,
GO:0097546, GO:0005634], TNFSF12=[GO:0048471, GO:0005576, GO:0001938, GO:0006915,
GO:0005125, GO:0006955, GO:0005615, GO:0001525, GO:0005164, GO:0005515, GO:0033209,
GO:2001238, GO:0005102, GO:0045732, GO:0045766, GO:0030154, GO:0097190, GO:0043542,
GO:0005886, GO:0097191, GO:0005887, GO:0007165], ATP13A3=[GO:0006874, GO:0043231,
GO:0005524, GO:0016020, GO:0098655, GO:0019829, GO:0005887, GO:0008152, GO:0046872],
VWA5B1=[GO:0005576]}
```

4.11 miRNA-SNP functional analysis in cancer

The query allows to evidence the functional significance of miRNA single nucleotide polymorphisms (SNPs) in cancer pathology.

Starting from a specific cancer type, miRNA SNPs linked to the cancer disease are selected and used in miRNA-target interactions DB (a free energy score is applied).

The result shows the functional annotations that link evidenced targets to the selected cancer disease.

```
gremlin> g.V().hasLabel('Cancer').
      has('name', 'colorectal cancer').
      out('CANCER2MIRNA').dedup().
      out('PRECURSOR_OF').
      out('HAS_SNP').
      in('INTERACTING_SNP').has('snpEnergy',lt(-34)).
      out('INTERACTING_GENE').dedup().
      group('genes').
      by('nomenclatureAuthoritySymbol').
      by(__.in('ANNOTATES').dedup().values('goId').fold()).
      cap('genes')

==>{PTP4A2=[GO:0035335, GO:0005769, GO:0005737, GO:0005886, GO:0004727, GO:0005634,
GO:0070062]}
```

4.12 Cancer involved miRNAs by pathway

Starting from a specific pathway, the following query finds the up-regulated miRNAs involved in a specific cancer scenario.

```
gremlin> g.V().hasLabel('Pathway').
      has('name', 'Cell Cycle').
      out('CONTAINS').
      in('CODING').
      in('INTERACTING_GENE').has('database','miRanda').has('energy',lt(-30)).
      out('INTERACTING_MIRNA').dedup().
      in('PRECURSOR_OF').
      group('mirnas').
      by('accession').
      by(__.inE('CANCER2MIRNA').has('profile','up').
            outV().dedup().values('name').fold()).
      cap('mirnas')

==>{MI0000268=[papillary thyroid carcinoma, squamous carcinoma, clear cell renal
cell cancer, pancreatic ductal adenocarcinoma, rectal cancer, gastric cancer, non-
small cell lung cancer], MI0000239=[lung cancer, follicular thyroid carcinoma,
breast cancer], MI0000095=[esophageal squamous cell carcinoma, colorectal cancer,
nasopharyngeal carcinoma, osteosarcoma, laryngeal squamous cell carcinoma, glioma,
head and neck squamous cell carcinoma, endometrial cancer, non-small cell lung
cancer]}
```

5 References

[1] The Apache Software Foundation, "Apache TinkerPop™," [Online]. Available: http://tinkerpop.apache.org/.

[2] A. Messina, A. Fiannaca, L. La Paglia, M. La Rosa and A. Urso, "Querying and analyzing biological data with BioGraph," 2017. [Online]. Available: https://doi.org/10.7287/peerj.preprints.3309v1.

[3] F. A. e. al., "Gremlin Language for Querying the BiographDB Integrated Biological Database," in *IWBBIO 2017: Bioinformatics and Biomedical Engineering*, Vols. Lecture Notes in Computer Science, vol 10208, Springer, Cham, 2017.

[4] A. Fiannaca, M. La Rosa, L. La Paglia, A. Messina and A. Urso, "BioGraphDB: A New graphDB Collecting Heterogeneous Data for Bioinformatics Analysis," in *BIOTECHNO 2016, The Eighth International Conference on Bioinformatics, Biocomputational Systems and Biotechnologies*, Lisbon, Portugal, 2016.

[5] A. Fiannaca, L. La Paglia, M. La Rosa, A. Messina, P. Storniolo and A. Urso, "Integrated DB for bioinformatics: a case study on analysis of functional effect of MiRNA SNPs in cancer," in *International Conference on Information Technology in Bio-and Medical Informatics*, 2016.

[6] A. Messina, "ETLs for importing UniProtKB, HGNC, and Reactome into a bioinformatics graph database," Palermo, 2016.

[7] A. Messina, "ETLs for importing NCBI Entrez Gene, miRBase, mirCancer and microRNA into a bioinformatics graph database," Palermo, 2015.

[8] K. R. Lawrence, "Practical Gremlin - An Apache TinkerPop Tutorial," 2018. [Online]. Available: http://kelvinlawrence.net/book/Gremlin-Graph-Guide.pdf.