Local DB support for SBGN maps in Newt

Personal Background

Personal Information

- **Name** : Pankaj Yadav
- **Email** : pankaj.yadav.ece17@itbhu.ac.in
- **Contact** : +91 - XXXXXXXXXX
- **Github** : github.com/y-pankaj
- **LinkedIn** : linkedin.com/in/y-pankaj
- **Institute** : Indian Institute of Technology (BHU), Varanasi, India
  Website - iitbhu.ac.in
- **Time Zone** : UTC +5:30 Hrs

Educational Background

I am a senior (final year student) pursuing Electronics Engineering from the Indian Institute of Technology (BHU), Varanasi, India. I was introduced to programming in the first semester of my college. Since then, I have delved deeper into the concept of Computer Science, such as Data Structures, Algorithms, Machine Learning, Operating systems, Databases, etc. I have a good grasp of Data Structures and Algorithms, Javascript, and Database concepts which I believe are crucial for successfully completing this project. I mostly code in Javascript and Python but have some experience using C++ and Java too.

Work Experience and Relevant Skills

My work experience includes working as a Backend Engineer at Kashiyatra. I modeled a database for storing a user's info and used it for Authentication and Authorization and keeping track of the user's activities. I did my summer internship last year at JP Morgan and Chase Co., where I had to translate input hand gesture videos in the Indian Sign Language into English sentences and vice versa. This project involved building robust machine learning models, a web interface to interact with the users, and hosting it on an AWS EC2 instance.

I actively solve data structures and algorithm (DSA) questions on websites such as codeforces which has helped me strengthen my grasp on DSA, and I feel comfortable writing complex algorithms. As evident from my Github profile, I have built extensions and websites using various javascript frameworks, which validates my ability to code in javascript.

This project requires the use of Javascript, HTML, CSS for the frontend, querying the Neo4j database using complex graph algorithms written using Cypher (the query language for the Neo4j), and hosting the database on a server. I believe my experience writing javascript code,

working with databases, deploying code on servers, and a firm grasp of data structures and algorithms are a perfect match for this project.

I hope to refine my skills, master new concepts such as graph databases, and contribute to an impactful and extensively used application this summer. I love exploring the application of computer science in various fields such as finance, sports, computational photography, etc. This project will give me insight into how computer science concepts can be applied in biomedical research.

Project Proposal

Project Info

Project Title: Local DB support for SBGN maps in Newt
Project Link: https://github.com/nrnb/GoogleSummerOfCode/issues/158
Potential Mentor: github.com/ugurdogrusoz
Contact with mentor: I have been in regular contact with the mentor and have asked for doubts and took suggestions regarding the project since the start of February.

Project Overview

The Systems Biology Graphical Notation (SBGN) is a standard graphical format to represent unambiguously biological networks and cellular processes. SBGN allows visualizing complex biological knowledge, including gene regulation, protein interaction, signaling pathways, and metabolic networks. SBGN is composed of three complementary languages: Activity Flow (AF), Process Description (PD), and Entity Relationship (ER), to cover different levels of detail of biological knowledge[1].

Newt is a web-based tool based on the Javascript library ChiSE to view, edit, and analyze biological maps in standard formats such as PD and AF languages of SBGN, SBML, and SIF. Currently, Newt does not have the functionality to store the SBGN maps drawn using it on an online database from Newt itself. These maps, when stored on the database, must be integrated with the existing pathways (like pieces of a puzzle) to construct a knowledge base that can act as a blueprint for simulations and other analysis methods[2]. Furthermore, Newt must also provide a functionality to query the database for maps of interest based on genes of interest using algorithms such as shortest path between entities, k-neighborhood, upstream/downstream of an entity, etc.

Project Implementation Details

The project can be divided into the following parts:

1. Selecting the type of database to use and hosting
2. Storing and then integrating the SBGN map in the database
3. Querying the database
4. UI changes

1. Selecting the type of database and hosting

SBGN maps represent biological entities and the interaction between them, therefore resemble the structure of a graph. Querying SBGN maps requires traversal over relationships and entities to a certain depth depending on the query. Graph databases are an ideal fit to store and query such kind of information. Therefore, this project will involve the use of Neo4j graph database.

The Neo4j database has been planned to be hosted along with some small services already running for the Newt Editor.

2. Storing and then integrating SBGN maps in the database

The algorithm to integrate the map can be divided as follows. First, the SBGN map as a whole is deposited in the database (as a separate graph). Then the integration algorithm is run to integrate the map with the entities already present in the database.

a. **Representation of SBGN map in the database:** I have used the insulin signaling SBGN PD map to demonstrate how the map will be stored in the Neo4j database. (The map is already present in the Newt Editor application as a sample). AF maps will be stored in a similar way.

   All the nodes i.e. Auxiliary units, Process nodes, Entity pool nodes, Container nodes, Reference Nodes, etc. will be stored as **nodes** in the database whereas the connecting arcs will be stored as **edges/relationships**. Each node and relationship (arc) has some properties associated with it such as id, class, label, styling information, etc which is stored as the property of the respective node and relationship in the database.

   Each relationship contains a property “weight” which is set to 1. And each node has a property “is_processed” which is 0 when the map is not integrated, after running the integration algorithm this property becomes 1. The “weight” property will be used when traversing the graph (see point 3. Querying the database) and the “is_processed” property will be used in the integration algorithm (see 2. f).

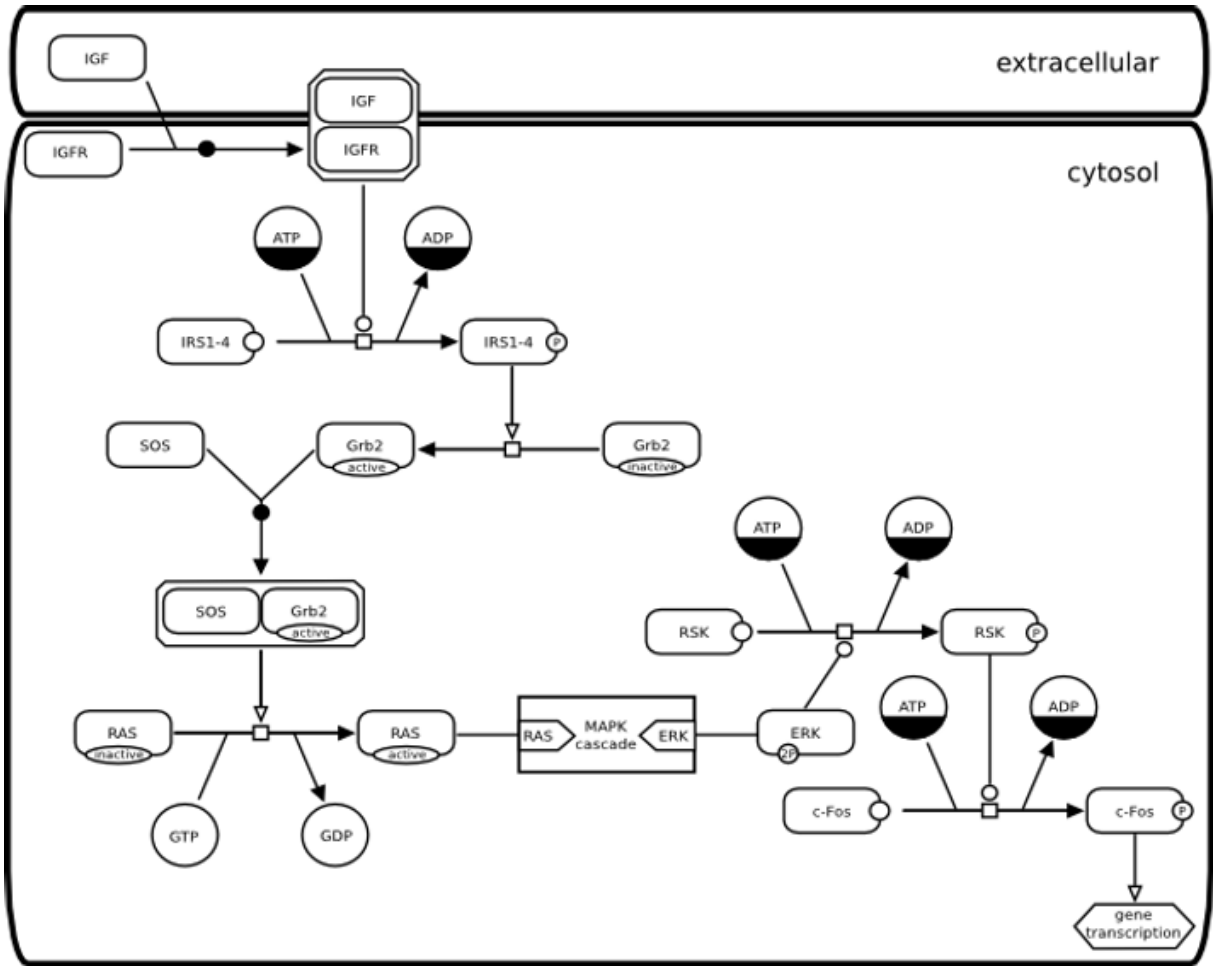


Figure 1: Insulin signaling PD map

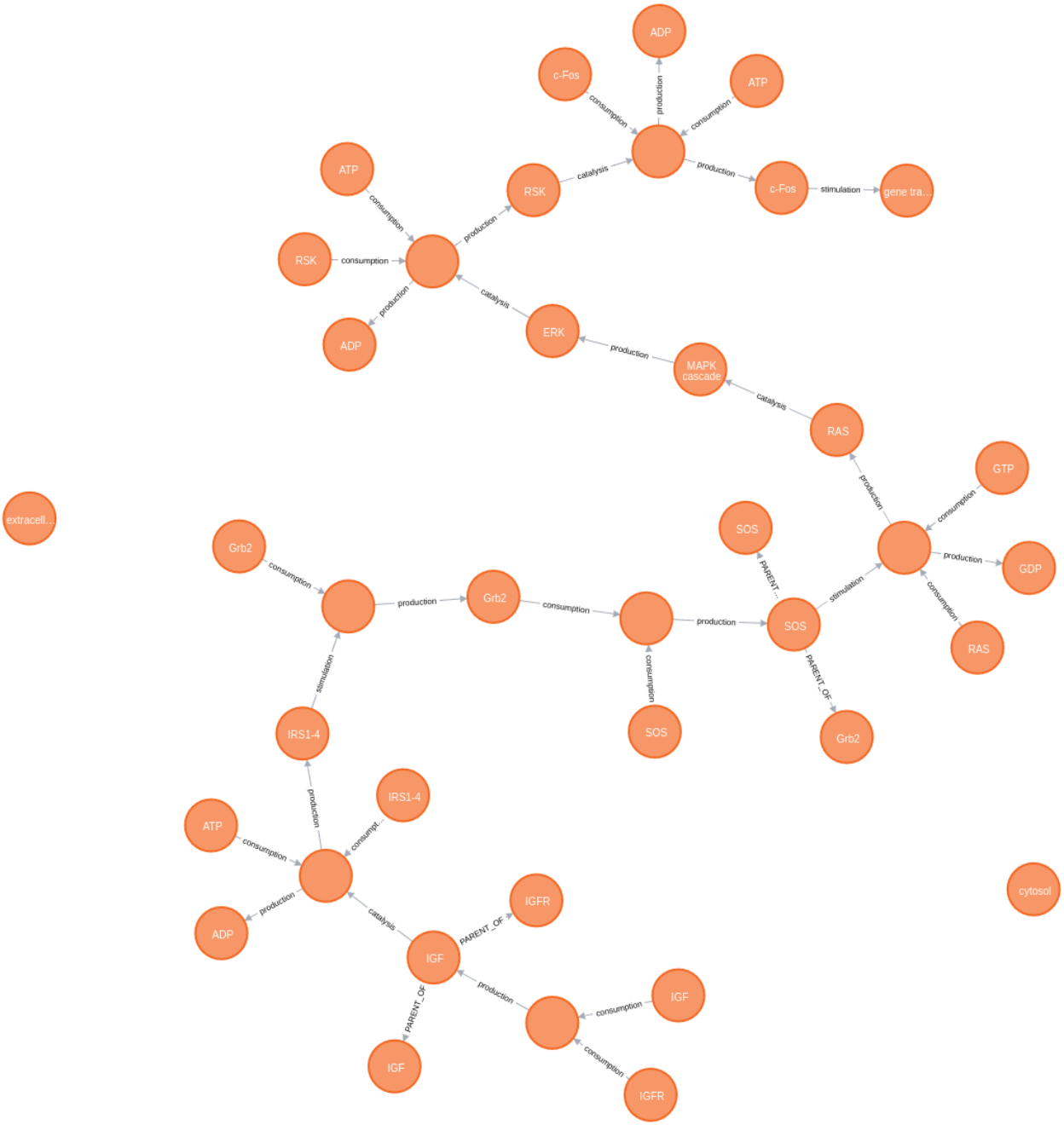


Figure 2: Insulin Signalling SBGN-PD map as represented in the database

b. **Dealing with complexes and container nodes:** Complexes can have multiple complexes, macromolecules, or any other entity inside them when represented as an SBGN map. Therefore for writing accurate and fast traversal algorithms we need to connect the parent complexes and their children. This is done by creating a PARENT_OF type of relationship from the complex to its children.
Container nodes are not connected to their children by the PARENT_OF relationship as all the entities inside such nodes are its children which is in the order of hundreds. Making such a relationship would make the traversal significantly slower. Moreover, traversal algorithms can be safely written even without such relationships.

PARENT_OF type of relationship has the property “weight” set to as 0. This will help us in writing querying algorithms in both cases when an **equivalence relation** between the compound node and its members exists and in the case when it does not. If an underlying "equivalence relation" exists between a compound node and its members, a traversal reaching a node in an equivalence set should also reach and visit other nodes in this set. For example, in the figure below if an equivalence relation is defined, when the traversal reaches the SOS complex node it should traverse Grb2 and SOS macromolecules as well.
The below images show how complexes are stored in the database.

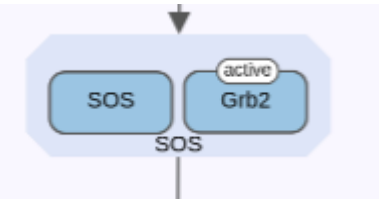


Figure 3: A complex in SBGN map

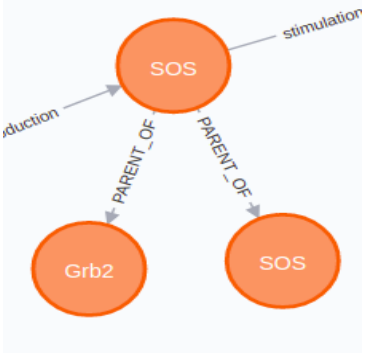


Figure 4: Complex stored in the database

c. **Converting SBGN map to a JSON object:** The SBGN map must be converted to a suitable format so that it can be processed and stored in the database. Since Newt is based on the ChiSE library we can use one of its inbuilt function createJsonFromSBGN() to convert an SBGN map to a JSON object. Below is a part of the JSON object generated from the insulin signaling map.


```
{
 "nodes":[
    {
       "data":{
          "id":"glyph1",
          "bbox":{
             "x":594.5807101845897,
             "y":477.92682101845907,
             "w":850.4114203691795,
             "h":608.9373569191597
          },
          "originalW":878.4114203691795,
          "originalH":636.9373569191597,
          "class":"compartment",
          "label":"cytosol",
          "statesandinfos":[
```

```
          ],
          "language":"PD",
          "border-width":3.25,
          "border-color":"#555555",
          "background-color":"#eff3ff",
          "background-opacity":"1",
          "background-image-opacity":1,
          "text-wrap":"wrap",
          "font-size":14,
          "font-family":"Helvetica",
          "font-style":"normal",
          "font-weight":"normal",
          "color":"#000",
          "background-fit":"cover",
          "background-position-x":"50%",
          "background-position-y":"50%",
          "background-image":"",
          "ports":[

          ]
       },
       "style":{

       }
    }, ...
. . . . .
 "edges":[
    {
       "data":{
          "id":"nwtE_205f3bda-b313-4d9f-9043-7a3a5f3c474e",
          "class":"consumption",
          "bendPointPositions":[

          ],
          "language":"PD",
          "line-color":"#555555",
          "width":1.25,
          "background-color":"#ffffff",
          "background-fit":"cover",
          "background-opacity":"1",
          "background-position-x":"50%",
```

```
          "background-position-y":"50%",
          "background-image":"",
          "cardinality":0,
          "source":"glyph2",
          "target":"nwtN_da50a90d-5185-4836-8f2e-2a29fabef5ac",
          "portsource":"glyph2",
      "porttarget":"nwtN_da50a90d-5185-4836-8f2e-2a29fabef5ac.1"
       },
       "style":{

       }
    },...
. . .
}
```

d. **Connecting to the database and sending requests:** For javascript applications, neo4j-driver can be installed using npm and requests can be made to the database in the following way.

```
const neo4j = require('neo4j-driver')

const driver = neo4j.driver(uri, neo4j.auth.basic(user, password))
const session = driver.session()
const personName = 'Alice'

try {
 const result = await session.run(
   'CREATE (a:Person {name: $name}) RETURN a',
   { name: personName }
 )

 const singleRecord = result.records[0]
 const node = singleRecord.get(0)

} finally {
 await session.close()
}
// on application exit:
await driver.close()
```

Here URI, user, and password are set when hosting the database and are known.

e. **Algorithm to store a map on the database:** The following code will be run for storing the map in the database. The algorithm first generates the nodes based on the JSON representation of the SBGN map, then it generates the relationships, lastly, it generates the PARENT_OF kind of relationship.

Creating nodes:

```
WITH json_object as value
UNWIND value.nodes as node  // getting the node data form the JSON object
CREATE (n:Node) // create empty nodes for each node in JSON object
// setting the properties for the node
SET n+= node.data, n.is_processed = 0;
```

Creating relationships:

```
WITH json_object as value
UNWIND value.edges AS edge
MATCH (n:Node), (m:Node)
// get the nodes corresponding to SOURCE and TARGET property of the edge
WHERE n.id=edge.data.source and m.id=edge.data.target
// create the relationship
CALL apoc.merge.relationship(n,edge.data.class,{},{},m) YIELD rel
SET rel+=edge.data, rel.weight = 1  // set the properties of the edges
```

Creating PARENT_OF kind of relationship:

```
// find all nodes who have a parent
MATCH (n:Node) WHERE EXISTS (n.parent)
WITH n as childNode
MATCH (parentNode:Node)
// get the parent node of a particular node
// node should not be of type compartment as explained in (2. b)
WHERE parentNode.id = childNode.parent AND NOT parentNode.class =
'compartment'
// create relationship of type PARENT_OF with weight 0
CREATE (parentNode)-[r:PARENT_OF]->(childNode)
SET r.weight = 0;
```

f. **Integrating the map:** Currently, the integration algorithm for AF language has been developed. In PD language, a biological entity may appear multiple times in different

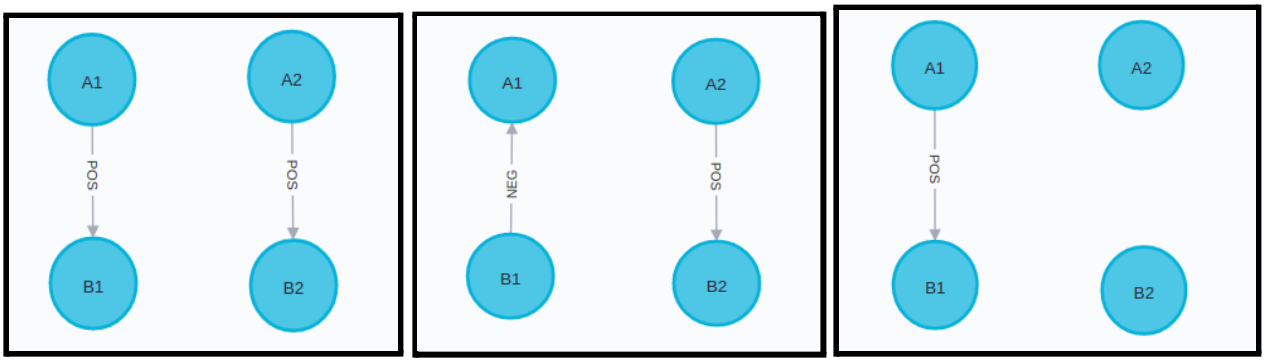
“states", might be expressed with a “slightly different” label, etc. Therefore, it would require further discussion with the mentor to properly develop the algorithm.

Let us call the SBGN AF map to be integrated into the database as **G1** and the integrated map/graph already present in the database as **G2**. Remember that G1 was stored in the database (but not integrated) as explained in part 2. (e). Let **R1** be a relationship present in G1. Let us define **A1** and **B1** as the source node and target node for the relationship R1. Also some nodes and relationships in G1 and G2 will be identical, therefore let **A2, B2, R2** correspond to A1, B1, and R1 respectively. Note that A2, B2, or R2 might not always exist.

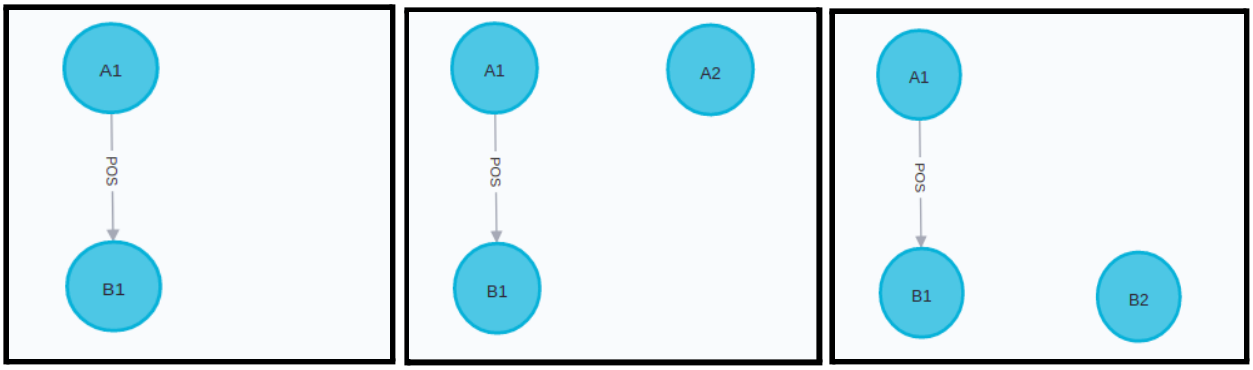
The idea behind the integration is based on the observation that there can be only **6 kinds of relationships in graph G1**.

i. Both nodes/entities connected by relationship and the relationship itself is already present in the database i.e. G2.
ii. Both nodes/entities connected by relationship are present in G2 but the type of relationship connecting A2 and B2 is different.
iii. Both nodes/entities connected by relationship are present in G2 but A2 and B2 are **not** connected by a relationship.
iv. Both nodes/entities connected by the relationship are not present in G2 i.e. A2 and B2 do not exist.
v. Only A2 exists in G2 (B2 does not exist).
vi. Only B2 exists in G2.

Pictorial representations of the cases are given below. The left side of each figure is G1 and the right side is G2.



Case (i) Case (ii) Case (iii)



Case (iv) Case (v) Case (vi)

Figure 5: Types of relationships

The integration algorithm generates some relationships at first depending on the conditions mentioned above and then deletes all the nodes from G1 which were **already present** in G2 and some relationships to get the integrated database. (As mentioned before, G1 is the SGBN AF graph and G2 is the integrated database).

The algorithm is as follows:

1. For each relationship in G1, we check which case from (i) to (vi) holds.
2. We do the following for the cases:
    i. For case (i) as the node and the relationship are already present in G2 we don't have to do anything.
    ii. For case (ii) as the nodes are already present in G2 but the relationship is different between A2 and B2 therefore we will have to create a new relationship that is the same as R1.
    iii. For case (iii) we generate a new relationship between A2 and B2 which is the same as R1.
    iv. For case (iv) we do nothing (as they will be automatically integrated at the end of the algorithm).
    v. For case (v) we make a relationship, same as R1 from A2 to B1.
    vi. For case (vi) we make a relationship, same as R1 from A1 to B2.
3. While generating the above relationships we set/update the "source" and "target" properties of the relationship.
4. Finally, we find nodes in G1 which are already present in G2 and delete these along with all the relationships coming in or going out from each one of these nodes.
5. We set the "is_processed" property to 1 for the remaining nodes as they have been integrated.

The final graph obtained is integrated.

Pictorial representation of point no. 2 is given below. The left side of each figure is G1 and the right side is G2.

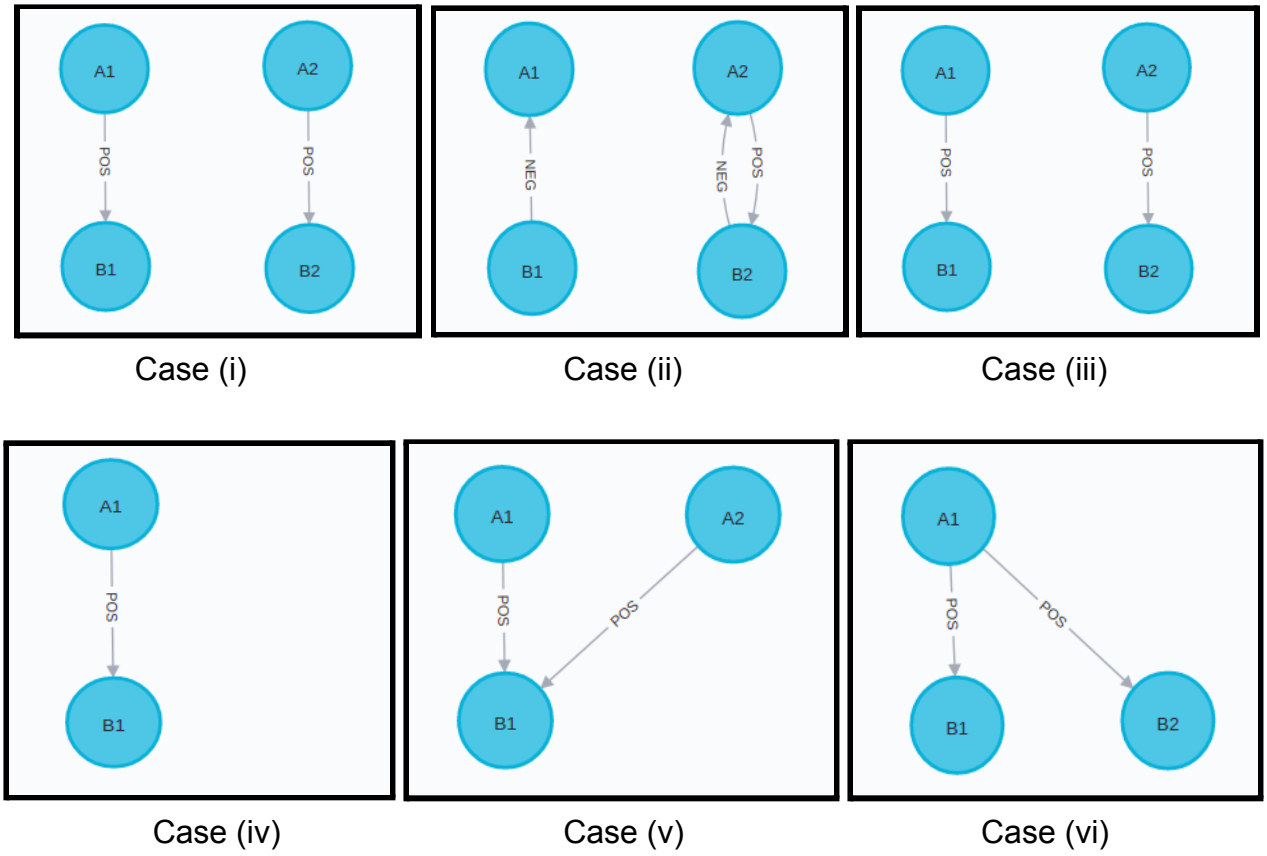


Figure 6: Generating the relationships

The working code along with comments is given below.

```
// get all the relationships, r for G1
MATCH (n:Node { is_processed:0}), (m:Node { is_processed:0})
WITH n,m
MATCH (n)-[r]->(m)
// get the nodes connected by the relationship r
WITH r
MATCH (sourceNodeG1:Node {id:r.source, is_processed:0}),
    (targetNodeG1:Node {id:r.target, is_processed:0})
WITH sourceNodeG1, targetNodeG1, r
// get the identical nodes from G2 which correspond to sourceNodeG1 and
targetNodeG1 respectively
// sourceNodeG2 and targetNodeG2 can be null as nodes present in G1 might
not always be present in G2
OPTIONAL MATCH (sourceNodeG2:Node {label:sourceNodeG1.label,
is_processed:1})
OPTIONAL MATCH (targetNodeG2:Node {label:targetNodeG1.label,
is_processed:1})
WITH sourceNodeG2, targetNodeG2, sourceNodeG1, targetNodeG1,
```

```
   r, type(r) as relType
// Depending of the case relationships need to be generated, total of six
cases
CALL apoc.do.case([
// case (i) and (ii) and (iii)
sourceNodeG2 is not null and targetNodeG2 is not null,
 'CALL apoc.cypher.run("MATCH (sourceNodeG2)-[rels:" + $relType +
"]->(targetNodeG2)
  RETURN count(rels) as cnt",
  {sourceNodeG2:sourceNodeG2, targetNodeG2:targetNodeG2, relType:relType})
  YIELD value
  WITH value.cnt as relCount, sourceNodeG2 as sourceNodeG2,
       targetNodeG2 as targetNodeG2, r as r, relType as relType
  CALL apoc.do.case([
      relCount = 0,
      "CALL apoc.create.relationship(sourceNodeG2, relType, r,
targetNodeG2)
      YIELD rel
      SET rel.source = sourceNodeG2.id, rel.target = targetNodeG2.id
      RETURN rel"],
  "", {relCount:relCount, sourceNodeG2:sourceNodeG2,
       targetNodeG2:targetNodeG2, r:r, relType:relType})
  YIELD value RETURN value',
// case (v)
sourceNodeG2 is not null and targetNodeG2 is null,
 'CALL apoc.create.relationship(sourceNodeG2, relType, r, targetNodeG1)
  YIELD rel
  SET rel.source = sourceNodeG2.id, rel.target = targetNodeG1.id
  RETURN rel',
// case (vi)
sourceNodeG2 is null and targetNodeG2 is not null,
 'CALL apoc.create.relationship(sourceNodeG1, relType, r, targetNodeG2)
  YIELD rel
  SET rel.source = sourceNodeG1.id, rel.target = targetNodeG2.id
  RETURN rel',
// case (iv)
sourceNodeG2 is null and targetNodeG2 is null,
 'RETURN r'
], '', {sourceNodeG2:sourceNodeG2, targetNodeG2:targetNodeG2,
sourceNodeG1:sourceNodeG1, targetNodeG1:targetNodeG1, r:r, relType:relType}
) yield value
```

```
WITH value
// delete nodes from G1 which are common with G2
// also deletes the relationship coming to or directed away from the node
MATCH (n:Node { is_processed:1}), (m:Node { is_processed:0})
WHERE n.label = m.label
DETACH DELETE (m)
WITH value
// set is_processed property to 0
MATCH (n:Node { is_processed: 0})
SET n.is_processed = 1;
```

Below is the result of the algorithm. The example covers all 6 cases of relationships.

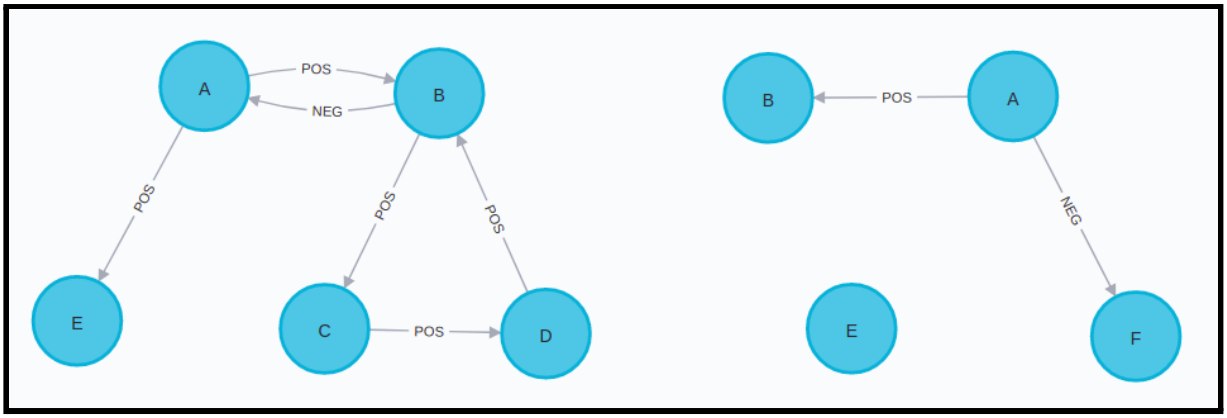


Figure 7: Left side: graph to be integrated G1, right side: G2, integrated graph

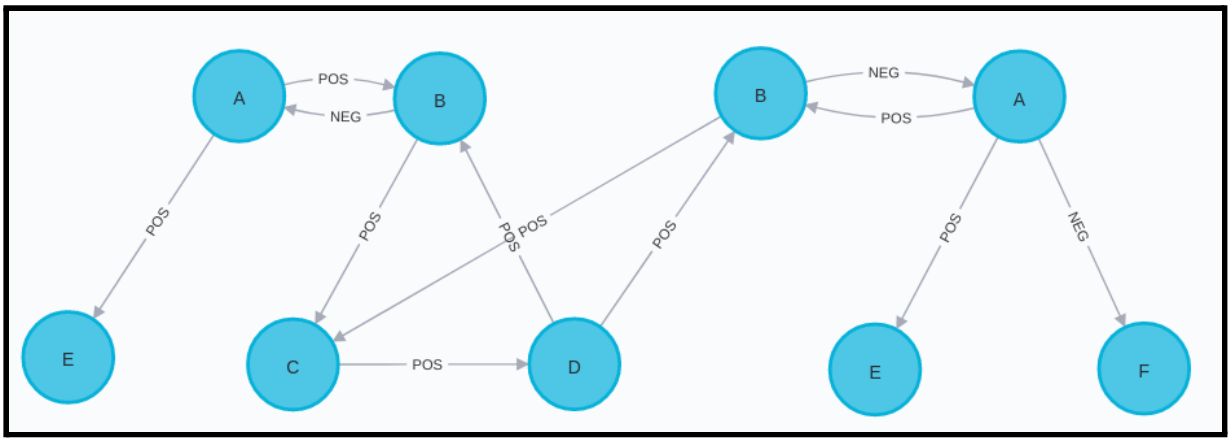


Figure 8: Running integration algorithms till step 3

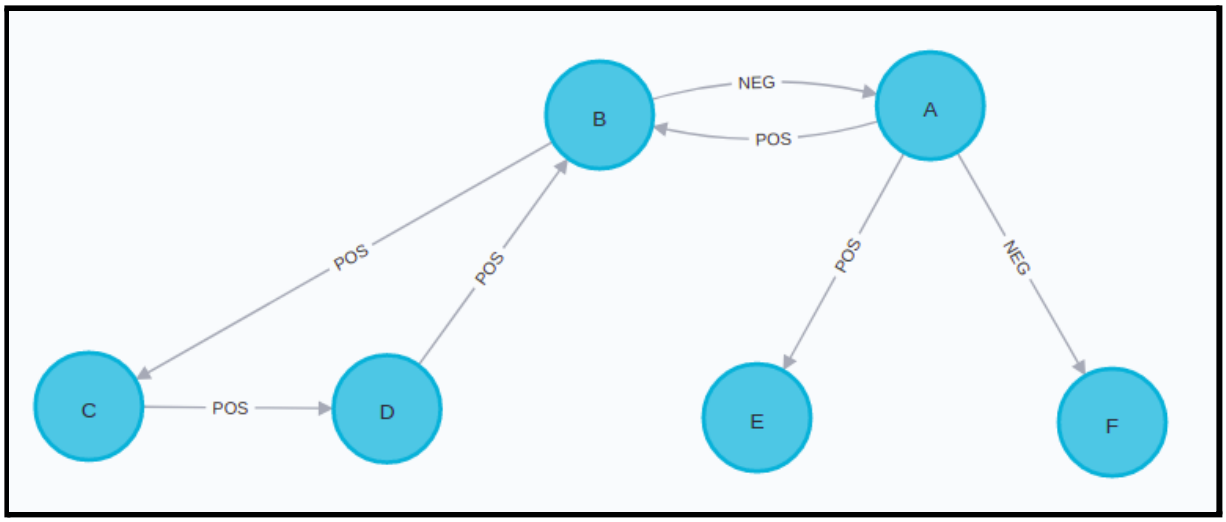


Figure 9: Final result of the integration algorithm

3. Querying the database

Some important querying algorithms along with their implementations are listed below. Over the course of the project, more algorithms will be researched and added.

a. k neighborhood when Equivalence Relation is not defined over the map

Since equivalence relation is not defined, we can't traverse between siblings or between a parent and a sibling if no direct node exists between them. This condition is imposed by disregarding all paths which contain the PARENT_OF relationship. For imposing k-neighborhood the sum of "weight" property over the edges of the path should not exceed k i.e. max distance of a node from the origin node can be at most k.

```
// get all the paths with no of relationships upto 15
MATCH p=(n:Node)-[*1..15]-(m:Node)
// node id for the origin node
WHERE n.id = 'glyph18'
WITH p, REDUCE(dis=0, r in relationships(p)| dis + r.weight) as pathlen
// keep only the paths where the total weight of path is less than k and
// disregard all the paths which contain node with 0 weight i.e. PARENT_OF
relationship
WHERE pathlen <= k and all(rel in relationships(p) where rel.weight = 1)
UNWIND nodes(p) as allnodes
// return the id of k-neighbourhood nodes
RETURN DISTINCT allnodes.id
```

b. k neighborhood when Equivalence Relation exists over the map

This is the same as above, the only difference is that we can traverse over the PARENT_OF kind of relationship.

```
// get all the paths with no of relationships up to 15
MATCH p=(n:Node)-[*1..15]-(m:Node)
// node id for the origin node
WHERE n.id = 'nwtN_59b63daf-15ae-4a4b-b048-f9dc3b8ed3b5'
WITH p, REDUCE(dis=0, r in relationships(p)| dis + r.weight) as pathlen
// keep only the paths where the total weight of path is less than k
WHERE pathlen <= k
UNWIND nodes(p) as allnodes
// return the id of k-neighbourhood nodes
RETURN DISTINCT allnodes.id
```

c. **Downstream of a node with path weight utmost k (Equivalence relation assumed):**

Here we make use of the fact that the nodes are directed in Neo4j. Therefore we only traverse the outgoing edges from each node. A similar algorithm can be written when the equivalence relation is not defined by not traversing over the PARENT_OF type of relationship.

```
// Direction of arrows in Relationship matters here
// -> means directed traversal and - is undirected traversal
MATCH p=(n:Node)-[*..15]->(m:Node)
// Only starting node is specified
WHERE n.id = 'glyph18'
WITH p, REDUCE(dis=0, r in relationships(p)| dis + r.weight) as pathlen
// keep only the paths where the total weight of path is less than equal to
k
WHERE pathlen <= k
WITH nodes(p) as allnodes
UNWIND allnodes as unwindednodes
RETURN unwindednodes.id;
```

d. **Upstream of a node with path weight utmost k (Equivalence relation assumed):**

This is the same as downstream but here we specify the node at which the path should end instead of specifying the start node.

```
// Direction of arrows in Relationship matter here
// -> means directed traversal and - is undirected traversal
MATCH p=(n:Node)-[*..15]->(m:Node)
// Only end node is specified
WHERE m.id = 'glyph18'
WITH p, REDUCE(dis=0, r in relationships(p)| dis + r.weight) as pathlen
// keep only the paths where the total weight of the path is less than k
WHERE pathlen <= k
WITH nodes(p) as allnodes
```

```
UNWIND allnodes as unwindednodes
RETURN unwindednodes.id;
```

e. Graph of interest where the path of length at most k: (equivalence relation assumed)

Here we just find all the paths between the given nodes n and m such that for any path the sum of the "weight" property of the edges lying on the path is at max k.

```
MATCH (n:Node), (m:Node)
// start and end nodes for GoI
WHERE n.id = 'GivenA' AND m.id = 'GivenB'
// paths with no of relationships up to 15
MATCH p = ((n)-[*..15]-(m))
// choosing the nodes with path weight up to k
WITH p, REDUCE(dis=0, r in relationships(p)| dis + r.weight) as pathlen
WHERE pathlen <= k
RETURN p;
```

f. Shortest path between entities:

This is the same as graph of interest, instead, we keep the shortest path out of all paths.

```
MATCH (n:Node), (m:Node)
// start and end nodes
WHERE n.id = 'GivenA' AND m.id = 'GivenB'
// paths with no of relationships upto 15
MATCH p = ((n)-[*..15]-(m))
// choosing the nodes with pathweight upto k
WITH p, REDUCE(dis=0, r in relationships(p)| dis + r.weight) as pathlen
WHERE pathlen <= k
RETURN p, pathlen
// keeping the shortest path
ORDER BY pathlen ASC
LIMIT 1
```

g. Feedback of an entity:

We just specify that the start and end node for the path should be the same as we are finding cycles here.

```
MATCH (n:Node)
WHERE n.id = 'GivenA'
// start and end nodes are the same as we are searching for cycles
// paths of length between 2 and 7 are chosen
MATCH p = ((n)-[*2..7]-(n))
```

```
RETURN p;
```

4. UI Changes

There are majorly two UI changes.

a. **Interface to make queries**
   This would require designing input fields based on the type of query.

b. **Coloring nodes and relationships based on the response from the database**
   This can easily be done by updating the active ChiSE instance. The result looks as follows. The green node is the origin node and the yellow node marks its neighbors.

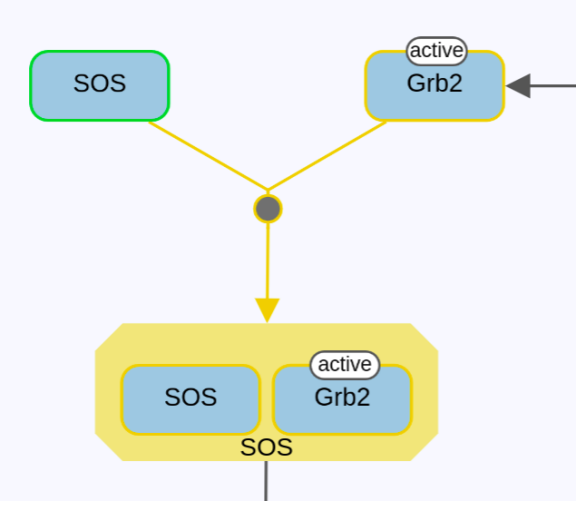


Figure 10: Colored nodes based on the response from the database

Project Timeline

Community bonding period	
May 17 - May 30	Understand Newt's codebase and related libraries. Develop a roadmap for the project along with the mentor.
May 31 - June 7	Study similar projects implemented in the past.
Phase 1	
June 7 - June 13	Host the database and connect it to Newt.
June 14 - June 20	Discuss and develop heuristics for integrating PD and AF maps.
June 21 - June 27	Implement the changes discussed in the algorithm

	for AF language. Test the robustness of the AF integration algorithm. Start implementing integration algorithm for PD language.
June 28 - July 4	Implement and test the integration algorithm for PD language.
July 5 - July 11	Discuss the types of queries to be implemented and research them.
Phase 2	
July 12 - July 18	Implement and test the query algorithms.
July 19 - July 25	Implement and test the query algorithms.
July 26 - August 1	Modify Newt's menu to include an option to query the database + other small changes.
August 2 - August 8	Modify frontend to display the result of the query.
August 9 - August 15	Clean up the codebase and improve the documentation

Availability

I have no prior work, college, or travel commitments from the date the community bonding period starts i.e. 17 May 2021 till 1 August 2021. I will be joining my job from 2nd August 2021. I believe I will be able to put in the required hours of work for GSOC even after joining my job as I'm willing to work in the after-work hours to complete my task at hand. My job is scheduled to be remote due to the ongoing COVID pandemic, therefore there are no plans to travel. Moreover, I plan to finish the project well before the deadline.

References

1. https://link.springer.com/referenceworkentry/10.1007%2F978-1-4419-9863-7_1096
2. https://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-10-376#Sec1
3. https://sbgn.github.io/learning#sbgn-pd-reference-card