



DIPARTIMENTO DI INFORMATICA

| **UniBa** |

DIPARTIMENTO DI  
INFORMATICA

Master degree in Computer Science

---

**F.O.A.I. Exam**

—

**Graph Brain: Ontology instances Restructurer  
and Graph Ranker**

**Students:**

*Antonio Silletti* — *Elio Musacchio*  
*765841* *763437*

# Summary

Introduction .....	1
Modules pipeline diagram .....	2
Raw data .....	3
Implementation .....	4
KBRestructurer .....	5
Missing 'arc_properties' predicate .....	6
Attributes to list/facts .....	7
SchemaToProlog .....	8
Directives .....	10
Entities .....	10
Relationships .....	11
Generic rules .....	13
XMLSchemaScanner and XMLSchemaElement .....	14
KB Translator .....	18
Graph ranking algorithms .....	21
Page Rank and Personalized Page Rank .....	21
Classical Page Rank results .....	24
Page Rank results - start vector and personalization vector .....	25
Spreading Activation .....	26
Spreading Activation results .....	27
Iterative Spreading Activation .....	28

# Introduction

This project focused on two objectives:

- *To code a program capable of restructuring the existing raw Prolog instances file exported from GraphBrain into a higher level Prolog representation*
- *To implement, in Prolog, algorithms to score portions of the instances graph from GraphBrain, so to select the most relevant subportions of it*

To this extent, both **Prolog** and **Java** programming languages were used. A pipeline of operations was established for the project which can be inspected in the diagram in the next page.

Given the pipeline, in the following we will describe:

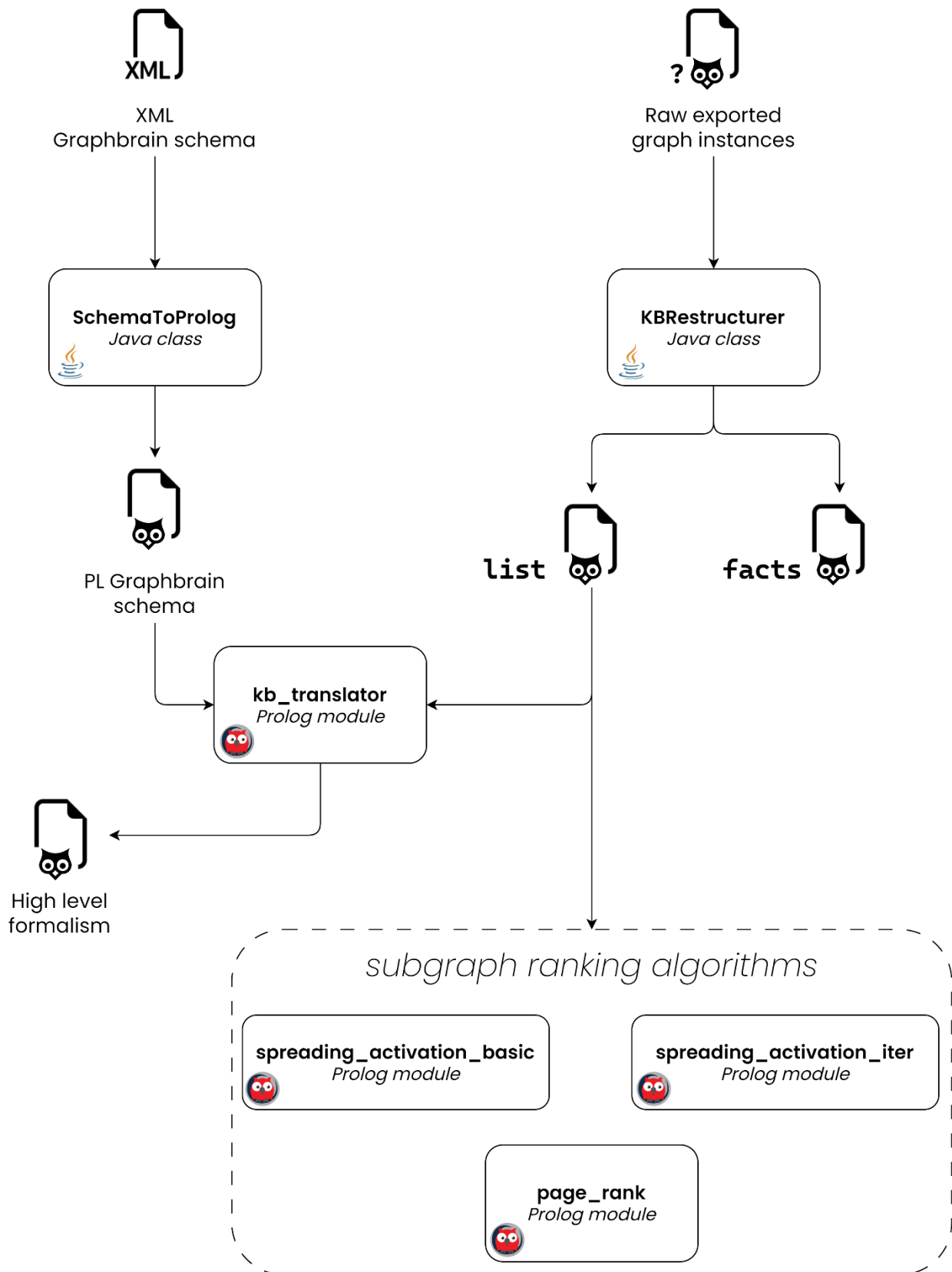
- The **KBRestructurer** and the **SchemaToProlog** Java classes, which perform *pre-processing* operations to transform the original files in a format that is more suitable for *Prolog* processing
- The **kb\_translator** Prolog module which effectively translates the pre-processed KB instances, on the basis of their formal definition in the associated ontology schema, into a higher level formalism
- The *graph ranking algorithms* Prolog modules (**page\_rank**, **spreading\_activation\_basic** and **spreading\_activation\_iter**), which given the restructured KB instances, will score each node based on their relevance

All *Prolog* modules were developed for the **YAP** interpreter, but an effort was made to make them fully compatible with **SWI**

- Only uncommenting **one** line of code for some of the modules is necessary. Check each module section or the [README.md](#) of the GitHub repository for details

As an addition, we will include the description of the developed **xmlschemaelement** Java package, which is used to intuitively navigate and process the XML *GraphBrain* schema files, independently of the task. It was developed for converting the XML schema file to the *Prolog* format.

# Modules pipeline diagram



# Raw data

The *exported raw instances graph file* contains three different types of predicates:

1. `node/2` where the first argument is the NodeID and the second argument is either the **top level class name** of NodeID or one of the **instance domains** of the instance

```
node(0, 'Person').  
node(0, 'retrocomputing').
```

The *top-level class* starts with a **capital letter**, while the *instance domains* start with a **lowercased letter**. The *top-level class* and the *instance domains* may be missing for certain NodeIDs and there can be more than one *instance domain* for the same NodeID

```
node(4, 'Place').  
node(4, 'retrocomputing').  
node(4, 'lam').
```

2. `node_properties/2` where the first argument is the NodeID and the second argument is a literal dictionary with all the attributes of the node

```
node_properties(0, '{name=Jack,gender=M,subClass=Person}').
```

3. `arc/4` where the first argument is the ArcID, then the ArcClass and finally the SubjectNodeID and the ObjectNodeID, which the specific ArcID links

```
arc(17440, 'produced', 2318, 337365).
```

There are some issues with the *raw data* that need to be addressed:

1. The attributes of a node, described in the second argument of the `node_properties/2` predicate, are **not in a 'Prolog friendly' formalism**. It would be better if there was a more suitable representation (e.g. list), so to avoid extracting information from a literal
2. There is **no description** for the possible attributes of an *arc*. The only one that is present is the name of the class for the arcs, but even in this case there is a consistency issue:

- For nodes, the *class name* is a value of the literal dictionary in the `node_properties/2` predicate (e.g. `subClass=Person`)
  - For arcs, the *class name* is the second argument of the `arc/4` predicate
3. Some instances do not have a consistent structure:
- Some entities do not have any instance domain associated to them
  - An instance in particular does not have any associated *top level class* and instead has a `null` value specified: `node(336551, null)`.

As described in this report, the code was designed to tackle these issues and to provide a more *refined* representation of the *GraphBrain* instances.

## Implementation

The description of the various modules of the project, will follow the order in which they should be used in the pipeline, namely:

- |                              |        |
|------------------------------|--------|
| 1. KRestructurer             |        |
| 2. SchemaToProlog            | Java   |
| <hr/>                        |        |
| • kb_translator              | Prolog |
| • page_rank                  |        |
| • spreading_activation_basic |        |
| • spreading_activation_iter  |        |

The modules do not have a strict order of execution, but may be used as standalone components (for the *Prolog* files, execution of the Java components is **necessary!**).

**NOTE:** all modules work assuming that the needed input files are present in the **inputs** folder located in the project root. These files are:


- The **raw graph instances file** (exported from *GraphBrain*)
- An **XML schema file** for the ontology to use as reference (exported from *GraphBrain*)

# KBRestructurer

To move to a *Prolog* based formalism, the **KBRestructurer** translates in a more *Prolog-friendly* format the raw instances file.

As described in the ['Raw Data' section](#), not only there is a **lack of coherence** between predicates of *nodes* and *arcs*, but attributes of a node are in a format that is **not suited** for *Prolog* processing.

To use the KBRestructurer Java class, simply call its `main()` function. The class expects the raw instances file to be placed into the **'inputs'** directory located in the project root:

Name	Date modified	Type	Size
 exportedGraph.pl	06/05/2023 10:23	Prolog Source File	80,729 KB

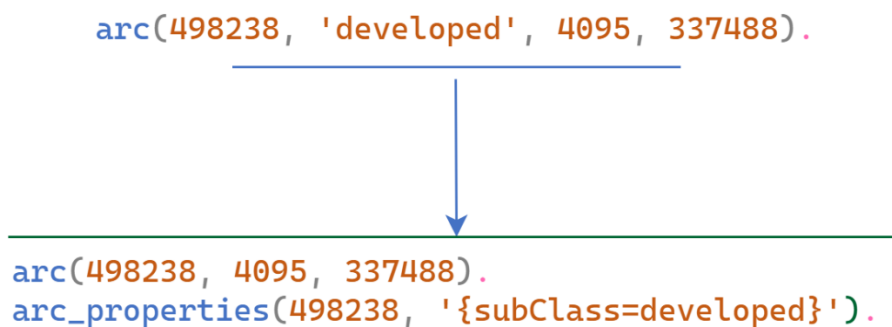
In case multiple `.pl` files are present in said directory, the program will prompt the user to select the appropriate one to convert.

## Missing 'arc\_properties' predicate

While for nodes there is the `node_properties/2` predicate which gathers all attributes of a node, such predicate is **missing** for arcs: this lack of coherence should be addressed by the tool that exports from *GraphBrain* the raw instances, but the `KBRestructurer` class performs an optional intermediate operation which tackles this issue before further restructuring the file

- In this way the `KBRestructurer` class is **future-proof**: when the tool which exports instances from *GraphBrain* will solve the coherence problem, including the `arc_properties/2` predicate, this intermediate step can be safely skipped

This optional step will split the `arc/4` predicate into two subpredicates `arc/3` and `arc_properties/2` where the class name, originally the second argument of the `arc/4` predicate, will be moved to the attributes stored in the new predicate `arc_properties/2`, in the same fashion as `node_properties/2`.



Note that the attributes of the `arc_properties/2` are purposely created as a dict literal to make the predicates homogeneous with `node_properties/2`

- They will both be restructured in the immediately next step

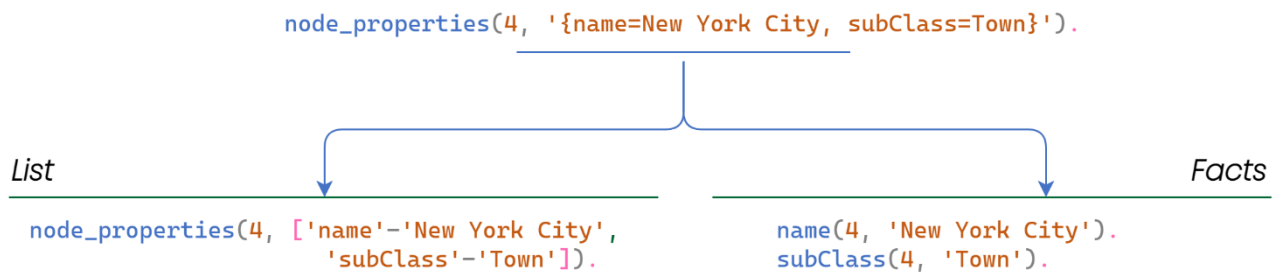


## Attributes to list/facts

The second and final task of the `KBRestructurer` class is to tackle the “not friendly” format of the attributes of `node_properties/2` and of the newly restructured `arc_properties/2`. Two different formalisms were developed, which the user could easily choose when prompted to do so after calling the `main()` of the `Java` class:

- **list**: attributes described as *key-value* pairs of literals
- **facts**: attributes described as *multiple facts*, one per each attribute



The following example shows how predicates are restructured, according to the chosen formalism:



It is suggested to use the *list-based formalism* since the rest of the project works assuming said option has been chosen, but we still provide to the user the option to obtain the *facts-based formalism*.

The output, containing the instances translated in the chosen formalism, will be saved in the **outputs** directory located in the project root with the format:

`{formalism}_{inputFilename}.pl`

Name	Date modified	Type	Size
 facts_exportedGraph.pl	31/05/2023 18:01	Prolog Source File	101,844 KB
 list_exportedGraph.pl	31/05/2023 20:50	Prolog Source File	106,109 KB

# SchemaToProlog

The conversion of the original instances of the *KB* to a *higher-level formalism*, carried out by the [kb\\_translator](#) *Prolog* module, **cannot be done** without having the schema of *nodes* and *arcs* available as reference.

Consider the two following `node_properties/2` predicates, each describing two different *nodes* of the same class, but with different attributes

- Note that the raw file instances have already been translated to the *list-based formalism*

```
node_properties(0,
    ['name'-'Jack',
     'gender'-'M',
     'subClass'-'Person']).
node_properties(1,
    ['name'-'Alex',
     'dateOfBirth'-'11/05/1972',
     'subClass'-'Person']).
```

Without a formal definition of the *Person* class, it is impossible to determine a consistent, complete and correct ordering for the attributes of each node of type *Person*.

That's why there's the need to use a schema as **reference** for the *high-level translation* of predicates, so that each of them adheres completely to the attributes and their order described in the formal description.

For *GraphBrain* specifically, an ontology schema could be used, but it is in XML format, while it would be ideal to have a native *Prolog* file. *GraphBrain* web UI provides a way to export the ontology schema as a `.p1` file, but the formalism in which predicates are exported is related to an inference engine which exploits the schema information:

```
fact(id_0, domain(retrocomputing), 1).
fact(id_1, entity(retrocomputing, artifact), 1).
fact(id_2, entity(retrocomputing, collection), 1).
fact(id_3, entity(retrocomputing, contentdescription), 1).
...
```

That's why we decided to translate the original XML schema into a more "standard" *Prolog* formalism, by implementing custom made *Java* classes which can easily navigate the schema and provide a **further layer of abstraction** for the *GraphBrain* developer, independently of the task.


To iterate over the *entities* and *relationships* in the XML schema file, we designed the **xmlschemaelement** package in which the **XMLSchemaScanner** class is the main responsible for iteration, and at each iteration, an **Entity/Relationship** object, which groups all of its information contained in the schema, is returned. *Entity* and *Relationship* classes are both children of the **XMLSchemaScanner** abstract class. These classes will be described more in depth in the [appropriate section](#).

The XML schema file provides the following information:

- **Attributes definition:** for both *entities* and *relationships*, **name** and **type** of attributes are defined, with an additional information specifying if the attribute is **mandatory** or not. For attributes with *type=select*, all possible values are listed
- **Taxonomy definition:** for both *entities* and *relationships*, their subclasses are defined
- **References definition:** for *relationships* only, **Subject-Object** entity names are defined
- **Inverse definition:** for *relationships* only, their inverse is defined with the inverse named attribute. If the relationship does not have an inverse, the named attribute value will be the same as the relationship name

It's important to note that, at the moment of writing, no *GraphBrain* ontology schema has subclasses for any relationship, but all the code developed is **future-proof**: it will work safely if in the future subclasses are added to relationships.

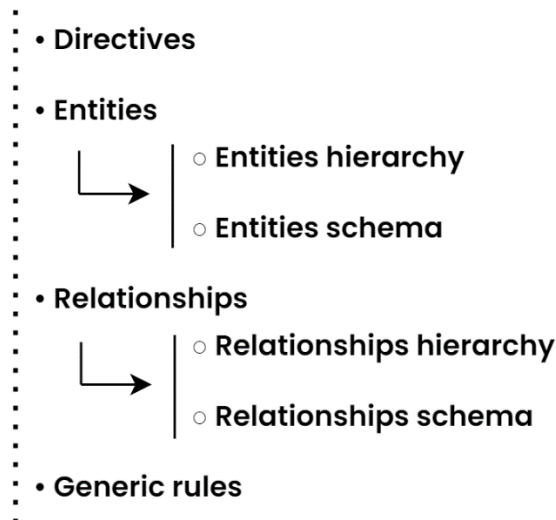
To use the SchemaToProlog Java class, simply call its `main()` function. The class expects the XML schema file to be placed into the '**inputs**' directory located in the project root:

Name	Date modified	Type	Size
 retrocomputing.xml	06/05/2023 12:08	XML Source File	51 KB

In case multiple XML files are present in said directory, the program will prompt the user to select the appropriate one to convert.

For each *entity* or *relationship*, all of its relevant information in the schema is extracted and the new predicates generated for it are then written to the converted *Prolog* schema file.

The final output file of the SchemaToProlog class can be described as if it was divided into the following sections:



## Directives

The directives written to the converted *Prolog* schema file are static, meaning that they are always written independently of the schema translated. They are:

```
:- use_module(library(lists)).
:- discontinuous inverse_of/2.
:- discontinuous subclass_of/2.
:- unknown(_, fail).
```

The only relevant aspect here, is that the `unknown` flag is set to `fail` when a predicate is missing, rather than `error`. This is done to guarantee compatibility between *SWI* and *YAP* interpreters.

## Entities

The **Entities hierarchy** section consists of multiple `subclass_of/2` predicates in the following format:

```
subclass_of(Subclass, Superclass).
```

The semantic of this predicate is to link a class to its **immediate** parent class. To find membership of a class to *any* of its parents in the hierarchy, the `is_subclass/2` predicate should be used, which will be described in the ['Generic rules' section](#).

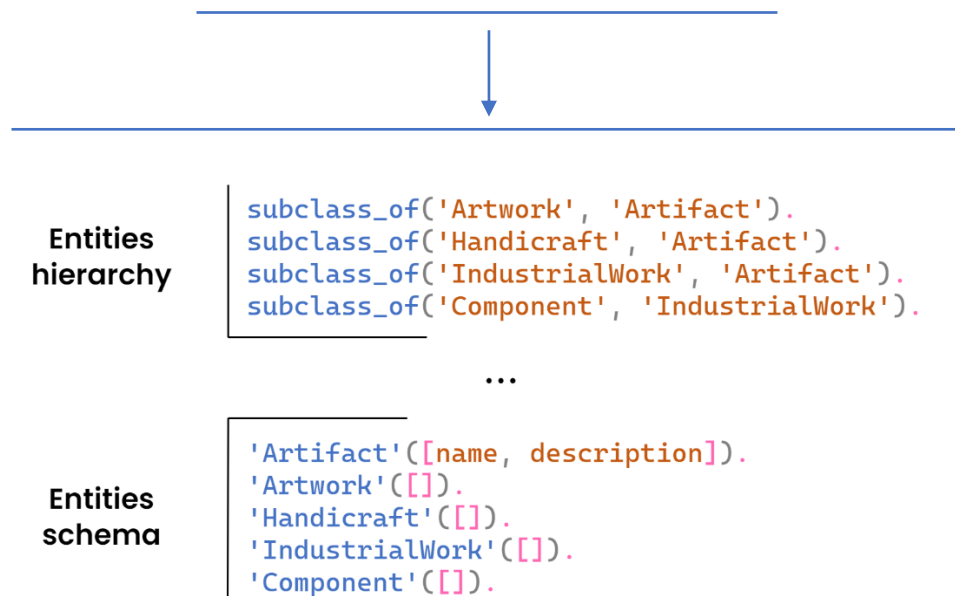
The **Entities schema** section consists of a different predicate for each Entity defined in the XML schema. Each predicate will be in the following format:

*ClassName*(*AttributesList*).

Where *ClassName* is the name of the entity, while *AttributesList* is the list of its attributes.

For example, for the **Artifact** entity defined as such in the XML schema:

```
<entity name="Artifact">
  <attributes>
    <attribute datatype="string" mandatory="true" name="name"/>
    <attribute datatype="string" mandatory="false" name="description"/>
  </attributes>
  <taxonomy>
    <value name="Artwork"/>
    <value name="Handicraft"/>
    <value name="IndustrialWork">
      <taxonomy>
        <value name="Component"/>
      </taxonomy>
    </value>
  </taxonomy>
</entity>
```



## Relationships

As for *entities*, the *Relationships* section can be further split into: **Relationships hierarchy** and **Relationships schema**.

The *Relationship hierarchy* section mirrors what was done for the *Entities hierarchy* section. Note that, as of now, there is no relationship in *GraphBrain* which inherits

from another one, but all the code developed is **future-proof**: it will work safely if in the future subclasses are added to relationships.

The **Relationships schema** section consists of one predicate for each Relationship defined in the XML schema. Each predicate will be in the following format:

*RelationshipClassName*(ReferencesList, AttributesList).

If the inverse of the relationship exists (i.e. the **inverse** named attribute of the XML tag has different value w.r.t. the **name** named attribute), an **inverse\_of/2** predicate will be added, in the following format:

**inverse\_of**(InverseRelationshipName, RelationshipName).

For example, for the **acquired** relationship defined as such in the XML schema:

```
<relationship inverse="acquiredBy" name="acquired">
  <references>
    <reference object="Item" subject="Person" />
  </references>
  <attributes>
    <attribute datatype="date" mandatory="false" name="date"/>
  </attributes>
  <taxonomy>
    <value inverse="boughtBy" name="bought">
      <attributes>
        <attribute datatype="integer" mandatory="true" name="price"/>
      </attributes>
    </value>
  </taxonomy>
</relationship>
```

Relationships  
hierarchy

**subclass\_of**('bought', 'acquired').

...

Relationships  
schema

'acquired'(['Person'-'Item'], [date]).  
**inverse\_of**('acquiredBy', 'acquired').  
 'bought'([], [price]).  
**inverse\_of**('boughtBy', 'bought').

## Generic rules

Finally, there is the **Generic rules** section, which provides useful *Prolog* rules for the converted schema. As the ['Directives' section](#) this is a static component of the final schema, meaning that it is always written independently of the schema translated.


The provided rules are:

- [is\\_subclass/2](#): checks if the given subclass inherits from the given superclass at any hierarchy level
- [invert\\_relationship/2](#): returns the inverted relationship clause given the relationship name
- [gather\\_attributes/2](#): returns the attributes list containing all attributes of the given *entity/relationship*, including those of their parents classes
- [gather\\_references/2](#): returns the references list containing all references of the given *relationship*, including those of their parents classes

It is possible to read an in-depth documentation associated to these rules once the final *Prolog* schema file is generated.

The output, containing the schema translated into a *Prolog* file, will be saved in the **outputs** directory located in the project root with the format:

schema\_{inputFilename}.pl

Name	Date modified	Type	Size
 schema_retrocomputing.pl	31/05/2023 18:58	Prolog Source File	27 KB

## XMLSchemaScanner and XMLSchemaElement

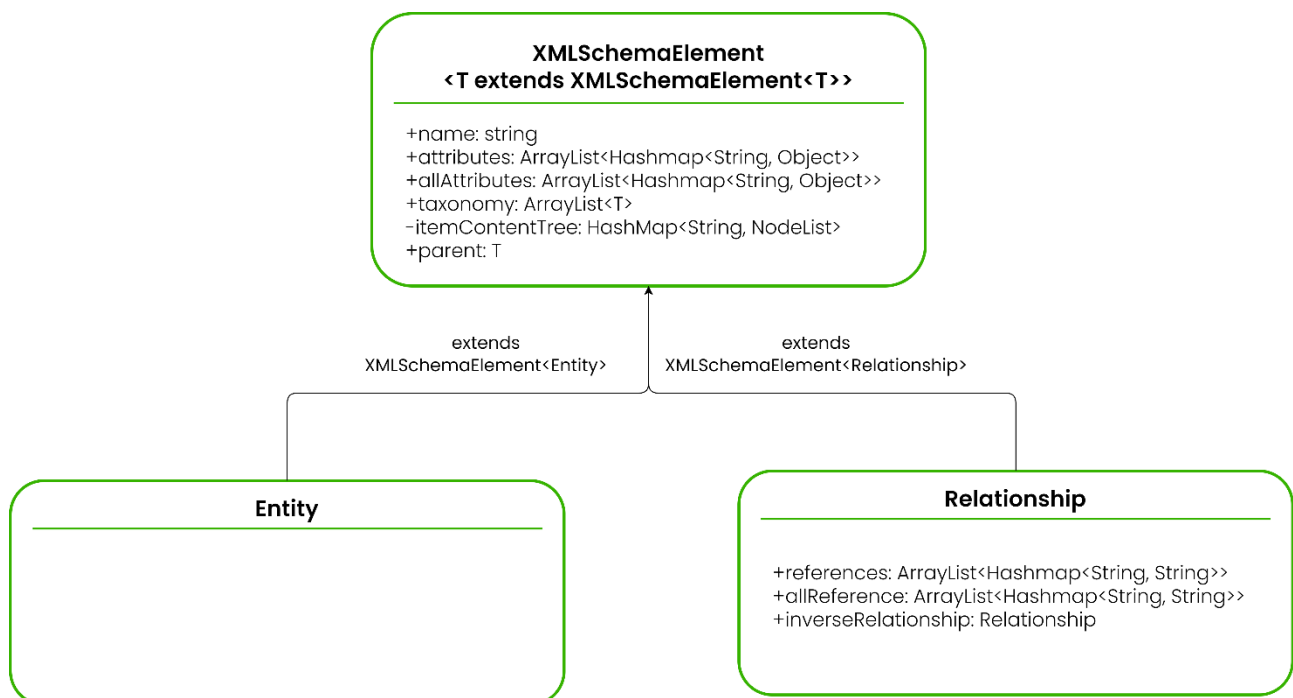
The `XMLSchemaScanner` class should be instantiated by simply setting the path for the XML schema file to navigate:

```
XMLSchemaScanner xmlFile = new XMLSchemaScanner("retrocomputing.xml");
```

The newly instantiated object allows iteration over *entities* and *relationships* contained in the original XML file with two different iterators:

- `xmlFile.iteratorRelationships()` for iterating over relationships, returns **Relationship** objects
- `xmlFile.iteratorEntities()` for iterating over entities, returns **Entity** objects

A simplified *UML* diagram, in which only attributes are present for the `XMLSchemaElement` class and its children *Entity* and *Relationship*, is the following:



To describe the meaning of all attributes, let's consider the following top level class with its attributes and taxonomy of the `retrocomputing.xml` schema:



```

<entity name="Collection">
  <attributes>
    <attribute datatype="string" mandatory="true" name="name"/>
    <attribute datatype="string" mandatory="false" name="acronym"/>
    <attribute datatype="select" mandatory="false" name="entityType">
      <values>
        <value name="Award"/>
        <value name="Organization"/>
        <value name="Component"/>
        <value name="Device"/>
        <value name="Document"/>
        <value name="Event"/>
        <value name="Person"/>
        <value name="Place"/>
        <value name="Software"/>
        <value name="System"/>
        <value name="Other"/>
      </values>
    </attribute>
    <attribute datatype="date" mandatory="false" name="startDate"/>
    <attribute datatype="date" mandatory="false" name="endDate"/>
  </attributes>
  <taxonomy>
    <value name="Family"/>
    <value name="Group"/>
    <value name="Series"/>
  </taxonomy>
</entity>

```

Considering the above case:

- **name** attribute is the *Entity/Relationship* name
 

```

> System.out.println(entity.name)
Collection

```
- **attributes** attribute is an ArrayList containing an HashMap for each attribute of the *Entity/Relationship*. Attributes with type=select have an additional field for all possible values

```

> System.out.println(entity.attributes)
[
  {datatype=string, name=name, mandatory=true},
  {datatype=string, name=acronym, mandatory=false},
  {datatype=select,
    values=[Award, Organization, ...],
    name=entityType,
    mandatory=false},
  ... ]

```

- **allAttributes** is the `ArrayList` containing attributes of the currently processed *Entity/Relationship* plus all the attributes of all its parent classes
  - In this specific case, `Collection` has no parent class so the output of `entity.allAttributes` would be the same as `entity.attribute`
- **taxonomy** is the `ArrayList` of *Entity/Relationship* objects that are part of the currently processed *Entity/Relationship* in its taxonomy. Note that the elements contained have their own name, attribute, taxonomy, etc.

```

> System.out.println(entity.taxonomy)
[Entity - Family, Entity - Group, Entity - Series]

```

- **itemContentTree** is a protected attribute pointing to the original portion of the XML file containing information about the currently processed *Entity/Relationship*
- **parent** is the *Entity/Relationship* object which is the immediate parent of the currently processed *Entity/Relationship*. It is **null** for top level classes (which don't have a parent node)

```

> System.out.println(entity.parent)
null

```

Relationships have additional attributes:

- **references**, which is an `ArrayList` of `HashMaps` with only fields `subject`, `object`

```

[
  {subject=Rel1, object=Rel2},
  {subject=Rel1, object=Rel3},
]

```

- **allReferences** which follows the intuition of the `allAttributes` attribute
- **inverseRelationship** which is the *Relationship* object representing the inverse of the currently processed *Relationship*. The inverse *Relationship* object will also have the subject-object `ArrayList` inverted. If a relationship does not have the inverse, this attribute is **null**.

Entities can be iterated with the following simple snippet of code:

```
Iterator<Entity> itEntities = xmlFile.iteratorEntities();

while (itEntities.hasNext()) {

    Entity entity = itEntities.next();
}
```

Same for relationships:

```
Iterator<Relationship> itRelationships = xmlFile.iteratorRelationships();

while (itRelationships.hasNext()) {

    Relationship rel = itRelationships.next();
}
```

This enables for an easy interface for processing information contained in the XML schema: for example, this is a simple code snippet which gathers all entity definitions which have at least 5 children in their taxonomy

```
XMLSchemaScanner xmlFile = new XMLSchemaScanner("retrocomputing.xml");

ArrayList<Entity> entitiesGathered = new ArrayList<>();

Iterator<Entity> itEntities = xmlFile.iteratorEntities();
while (itEntities.hasNext()) {

    Entity entity = itEntities.next();

    if (entity.taxonomy.size() >= 5){
        entitiesGathered.add(entity);
    }
}

System.out.println(entitiesGathered);
```



```
[Entity - Device,
Entity - Event,
Entity - IntellectualWork,
Entity - Item,
Entity - Organization,
Entity - ProcessComponent,
Entity - Software]
```

# KB Translator

The `kb_translator` module is responsible for performing the conversion of the *GraphBrain* instances to the *higher-level formalism*. It can be run both by *YAP Prolog* and *SWI Prolog* interpreters without any external requirements or operation.

This module must be used **only** after running the whole *Java* pipeline (`KBRestructurer` class and `SchemaToProlog` class), as it expects the raw instances file restructured to the *list-based formalism* and the XML schema file converted to *Prolog*.

When the module is *consulted*, two predicates will be publicly accessible: `translate/0` and `translate/2`. Both can be used to start the translation process.

- For `translate/0`, the reference schema converted to *Prolog* and the instances file restructured in the *list-based formalism* will be loaded automatically from the **outputs** directory
- For `translate/2`, the schema and instances file paths need to be specified directly

Once the process starts, the program will read each clause from the instances file and will convert it to the *higher-level formalism*. To do so, the program distinguishes between two main cases, that are clauses for entities and clauses for relationships.

In both situations, the ordering of the clauses in the file is important: the program assumes that before each `node_properties/2` predicate there may be `node/2` predicates defining the **top-level class** and the **instance domains** (in the latter case even *multiple ones*), and before each `arc_properties/2` predicate there is a single `arc/3` predicate defining the **Subject** and **Object** ids involved in the relationship.

In the case of entities, when a `node_properties/2` predicate is found, the program will first convert and write the clauses associated to the *top-level class* and the *instance domains*, using the following predicates:

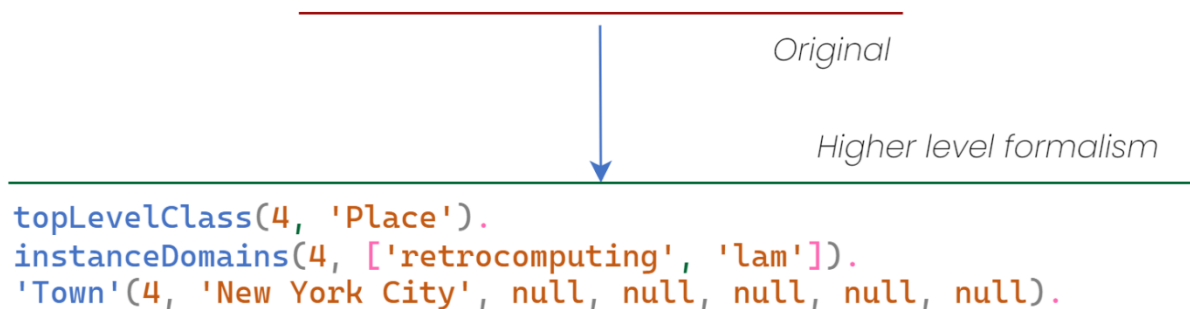
```
topLevelClass(EntityID, TopLevelClassName).  
instanceDomains(EntityID, InstanceDomainsList).
```

- If information regarding the top-level class is missing, the corresponding clause in the output file is skipped

- If information regarding the instance domains is missing, the corresponding clause in the output file is written with an empty list. If multiple instance domains are present, the list will contain all of them

Before writing the translated `node_properties/2` predicate, it will first complete and sort the attributes list of the entity instance based on its schema definition by filling it with `null` values for attributes present in the schema but missing from the instance attributes list.

```
node(4, 'Place').
node(4, 'retrocomputing').
node(4, 'lam').
node_properties(4, ['name'-'New York City', 'subClass'-'Town']).
```

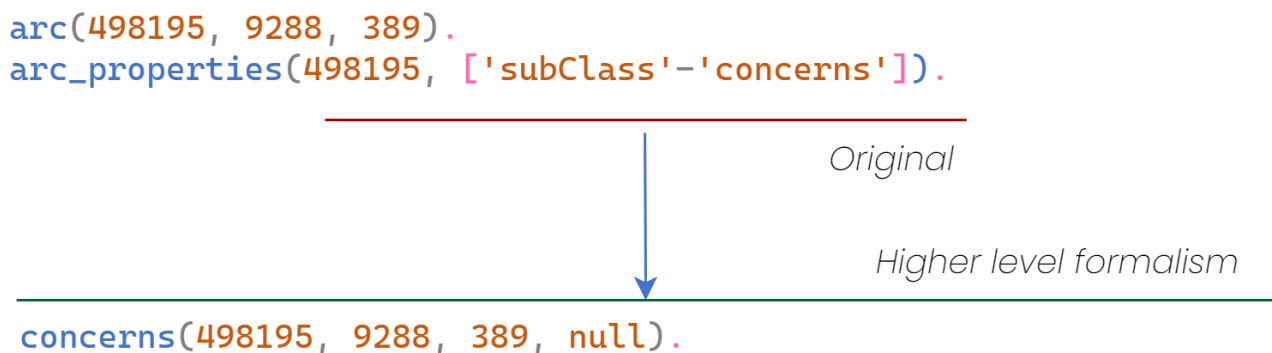


Where the `schema_retrocomputing.pl` (the converted XML schema file) contains the following definitions related to the 'Town' entity (remember that attributes in the *higher level formalism* are written starting from the **furthest** parent of the *entity*):

```
subclass_of('Administrative', 'Place').
subclass_of('Town', 'Administrative').
...
'Place'([name, language, latitude, longitude, description]).
'Administrative'([codeISO]).
'Town'([]).
```

For *arcs*, the same process will be followed, retaining information from *arc/3* predicates until an *arc\_properties/2* predicate is found and the final clause is written. Note that in this case, a single *arc/3* predicate is expected before *arc\_properties/2*.

Also for *arcs*, the program will complete the attribute list of the relationship instance, based on the schema definition of the relationship by filling it with **null** values for attributes present in the schema but missing from the instance attributes list. A translation example can be seen below:



Where the `schema_retrocomputing.pl` (the converted XML schema file) contains the following definitions related to the 'concerns' relationship:

```
'concerns'(['Category'-'Category', ...], [position]).
```

# Graph ranking algorithms

Since *GraphBrain* is populated by several instances, inspecting the actual graph that can be constructed from the data is a **complex task**. To analyze the graph in a meaningful way, it could be useful to consider only **subsets** of *relevant portions* of the graph. To find these relevant portions of the graph, it is necessary to **assign a score** to each *node* based on *how relevant* it is in the whole structure.

Because of this, we implemented *four* different algorithms which can assign these *relevance scores*: **Page Rank**, **Personalized Page Rank**, **Spreading Activation** and an iterative implementation of *Spreading Activation* based on the notion of **convergence**.

## Page Rank and Personalized Page Rank

The `page_rank` module is responsible for the computation of the *page rank scores* of the nodes in the graph.

It can be run both by *YAP Prolog* and *SWI Prolog interpreters* without any external requirements.

- If you use *SWI Prolog*, just make sure to uncomment **line 18** of the `page_rank.pl` file to import the statistics library, since *SWI* provides the `time/1` predicate in a library and it is not built-in like in *YAP*

By using *YAP* on a graph with **337239** nodes and **497933** arcs, the *PageRank* algorithm converges in **≈95 seconds**.

% 91.906 CPU in 95.221 seconds (96% CPU)

This module must be used **only** after running the `main()` of the `KBRestructurer` class, as it expects the raw instances file restructured to the *list-based formalism*.

When the module is *consulted*, the *Page Rank* algorithm could be started by using one of the following publicly accessible predicates: `page_rank/0`, `page_rank/3`, `page_rank/5`. There is no implementation difference between the various page rank predicates, for `page_rank/0` and `page_rank/3` we provide **default parameters** to *simplify* the algorithm usage.

The complete parameters definition for the *Page Rank* algorithm is the following:

- **DampingFactor**: parameter which decides the probability for the random surfer to follow an outgoing link or to teleport to a random node (with probability  $1 - \text{DampingFactor}$ )
- **Epsilon**: tolerance threshold, when the **L1 norm** of page rank values between two different iterations will be below this specified value, it means that the algorithm has converged and it can be stopped
- **MaxIter**: in case convergence cannot be reached, this parameter can be set to define the maximum number of iterations for the algorithm. Once this number has been reached, the algorithm will stop regardless of convergence
- **RankStartVector**: If initialized empty, the initial Rank Vector  $\mathbf{r}$  (containing the Page Rank values) will be automatically initialized with uniform distribution

$$\forall n \in \text{Graph}, r_n = 1/N \text{ where } N = \text{number of nodes.}$$

Otherwise, it expects a list of **key-value** pairs NodeID-InitialRank and for each node present in this list, the according InitialRank value will be used. All nodes missing from this list will have InitialRank **equal to 0**

- **PersonalizationVector**: in the classic Page Rank algorithm, when performing the random teleport, all nodes have the same probability of being picked. If initialized empty, the Personalization Vector  $\mathbf{p}$  (containing the Personalization values) will be automatically initialized with uniform distribution (no personalization):

$$\forall n \in \text{Graph}, p_n = 1/N \text{ where } N = \text{number of nodes.}$$

Otherwise, it expects a list of **key-value** pairs NodeID-PersonalizationValue and for each node present in this list, the according PersonalizationValue will be used. All nodes missing from this list will have PersonalizationValue **equal to 0**. Explicitly setting personalization values allows to perform a variation of the Page Rank algorithm known as **Personalized Page Rank**

The Page Rank algorithm could also take into account weighted arcs. In our case, we considered as weight 1 for all arcs (since *GraphBrain* arcs are not weighted).

If using [page\\_rank/5](#), all above parameters must be specified.

- Note that, as per *Page Rank* requirements, the sum of the values in RankStartVector and PersonalizationVector **must be equal to 1**. However, the program will automatically fulfil this requirement by normalizing the values of both lists



For `page_rank/3`, `RankStartVector` and `PersonalizationVector` are initialized empty: uniform distribution will be used for both.

For `page_rank/0`, `RankStartVector` and `PersonalizationVector` are initialized empty as in `page_rank/3`, but also the other parameters are set with the following values:

- *DampingFactor* = 0.85
- *Epsilon* =  $1e - 6$
- *MaxIter* = 100

Once the algorithm has completed, the Page Rank value associated to each node can be checked by using the only other publicly accessible predicate, `rank/2`:

```
rank(0, PageRankValue). --> PageRankValue = 0.3213
```

Internally, the *Page Rank* algorithm implementation is inspired by the one used by the **NetworkX** library, the *state-of-the-art* (for the **Python** programming language) for graph *manipulation and processing*.

However, instead of building *1d-arrays* and *2d-arrays*, we exploited **dynamic predicates** to keep track of the state of each node and have a **fast** direct access to them thanks to the indexing done by *Prolog* on clauses. Infact, an internal private predicate `node_pr_info/4` is used which, for each node, it saves the Page Rank value at the *previous iteration*, the *current one* and its *number of outgoing links*.

```
node_pr_info(NodeID, PageRankT0, PageRankT1, NOutlinks).
```

This implementation favours **time complexity** over **space complexity**. Infact, having a `node_pr_info/4` predicate for each node in the graph could be considered space heavy, but by monitoring the resources we determined that it was neglectable and by considering an alternative implementation based on lists, which favours **space** instead of **time**, we decided the *trade-off* appropriate.

- The performance was **≈60 times slower** (*4 minutes to perform a PR iteration instead of 4 seconds*)

The correctness of our implementation was tested against the already mentioned NetworkX library. There is only one slight difference: while NetworkX computes the stopping criterion to check convergence by multiplying the **L1 norm** of the *Page Rank* values to the **total number of nodes**:

$$N \cdot \sum_n |r_n^{new} - r_n^{old}|$$

where  $N = \text{number of nodes}$

Our implementation instead does not perform this product and simply refers to the computed *L1 norm*. By applying this small modification to the *NetworkX* Page Rank algorithm as well, the results that we obtained are the same ones, *independently* of the parameters that are set, when compared to our *Prolog* implementation.

The results obtained with the *NetworkX* implementation, can be reproduced by running the [colab IPython script](#).

## Classical Page Rank results

### *NetworkX*

```
>>> page_rank_dict = custom_pagerank_scipy(
...     graph,
...     alpha=0.85,
...     max_iter=100,
...     tol=0.000001
... )

page_rank_dict[0] = 8.919201525861478e-07
page_rank_dict[1] = 2.028072967268872e-06
...
page_rank_dict[457] = 1.1809186304811592e-06
...
```

### *Prolog*

```
?- page_rank(0.85, 0.000001, 100).

rank(0, PageRankValue). ---> PageRankValue = 8.91920152586148e-007
rank(1, PageRankValue). ---> PageRankValue = 2.02807296726887e-006
...
rank(457, PageRankValue). ---> PageRankValue = 1.18091863048116e-006
...
```

## Page Rank results - start vector and personalization vector

### *NetworkX*

```
>>> page_rank_dict = custom_pagerank_scipy(  
...     graph,  
...     alpha=0.85,  
...     max_iter=100,  
...     tol=0.000001,  
...     nstart={0: 5.0, 2: 3.2, 6: 2.4},  
...     personalization={15: 3.1, 67: 2.3, 457: 0.5}  
... )  
  
page_rank_dict[0] = 0.0  
page_rank_dict[1] = 0.0  
...  
page_rank_dict[457] = 0.024468298460878633  
...
```

### *Prolog*

```
?- page_rank(0.85,  
|           0.000001,  
|           100,  
|           [0-5.0, 2-3.2, 6-2.4],  
|           [15-3.1, 67-2.3, 457-0.5]).  
  
rank(0, PageRankValue). ---> PageRankValue = 0.0  
rank(1, PageRankValue). ---> PageRankValue = 0.0  
...  
rank(457, PageRankValue). ---> PageRankValue = 0.0244682984608786  
...
```

## Spreading Activation

The `spreading_activation_basic` module is responsible for the computation of the **activation scores** of nodes in the graph using the *classical implementation* of the algorithm, in which *nodes* can be fired **only once**. It can be run both by *YAP Prolog* and *SWI Prolog* interpreters without any external requirements.

- If you use *SWI Prolog*, make sure to uncomment **line 16** of the `spreading_activation_basic.pl` file to import the `statistics` library, since *SWI* provides the `time/1` predicate in a library and it is not built-in like in *YAP*

By using *YAP* on a graph with **337239** nodes and **497933** arcs, the Spreading Activation algorithm time is neglectable

- With three starting nodes, it converges in *less* than a second

This module must be used **only** after running the `main()` of the `KBRestructurer` class, as it expects the raw instances file restructured to the *list-based formalism*.

When the module is *consulted*, the *Spreading Activation* algorithm could be started by using the `spreading_activation/3` predicate.

The complete parameters definition for the spreading activation algorithm is the following:

- **StartNodes**: list of nodes identifiers from which the algorithm will start spreading (their *activation value* will be initialized to 1)
- **FiringThreshold**: the minimum activation value a node must have to be *fired* (Its activation value will be spread to its neighbors)
- **DecayRate**: factor by which the spread activation value will *decrease* when passed to another node (for example, if the current node has an *activation value* of 0.7 and the *decay rate* has a value of 0.9, the activation value for a neighboring node will be  $0.7 \cdot 0.9$ )

The spreading activation algorithm could also take into account weighted arcs. In our case, we considered as weight **1** for all arcs (since *GraphBrain* arcs are not weighted).

Once the algorithm has completed, the activation value associated to each node can be checked by using the only other publicly accessible predicate, `activation/2`:

```
activation(34, ActivationValue). ---> ActivationValue = 0.64
```

The Spreading Activation algorithm implementation has an internal predicate `node_sa_info/3` (similarly to the Page Rank algorithm) which is used to keep track of the node activation value at the *current iteration* and the *state of the node* (**fired** or **unfired**).

```
node_sa_info(NodeID, ActVal, FiredState).
```

While for the *Page Rank* algorithm we could test our implementation with respect to the one developed in a well-known and tested library, both the literature was scarce with very few available implementation. Therefore, to test the correctness of our implementation, we tested it by reproducing the [example](#) proposed in the **Wikipedia** page for the algorithm.

Note that in the formulation of the *Spreading Activation* algorithm it is also important to check the weight of arcs to compute the final activation value for the neighboring nodes. Since for *GraphBrain* we considered each arc to have weight equal to 1, just when reproducing the linked example, we hardcoded weight equal to 0.9, as it is the one being used.

By running the algorithm on the proposed example, assuming a firing threshold of 0.35 since it is not specified, we obtained the same results.

## Spreading Activation results

```
?- spreading_activation([1], 0.35, 0.85).
```

```
activation(1, ActivationValue). ---> ActivationValue = 1
activation(2, ActivationValue). ---> ActivationValue = 0.765
activation(3, ActivationValue). ---> ActivationValue = 0.585225
activation(4, ActivationValue). ---> ActivationValue = 0.447697125
...
activation(11, ActivationValue). ---> ActivationValue = 0.447697125
activation(12, ActivationValue). ---> ActivationValue = 0.342488300625
activation(13, ActivationValue). ---> ActivationValue = 0
```

# Iterative Spreading Activation

Furthermore, we also implemented a **variation** of the *Spreading Activation* algorithm where, nodes can be fired **multiple times** instead of only once. In this case, the algorithm does not stop when no nodes can be fired, but rather when **convergence** or the **maximum number of iteration** has been reached. This variation is taken from **step 8** of the [algorithm description in Wikipedia](#).

The `spreading_activation_iter` module is responsible for the computation of the activation scores of nodes in the graph using the *iterative implementation* of the algorithm. It can be run both by *YAP Prolog* and *SWI Prolog* interpreters without any external requirements.

- If you use *SWI Prolog*, just make sure to uncomment line 17 of the `spreading_activation_iter.pl` file to import the statistics library, since *SWI* provides the `time/1` predicate in a library and it is not built-in like in *YAP*

By using *YAP* on a graph with **337239** nodes and **497933** arcs, the iterative Spreading Activation algorithm converges in **≈21 seconds** specifying three different starting nodes

% 20.265 CPU in 21.247 seconds ( 95% CPU)

This module must be used **only** after running the `main()` of the `KBRestructurer` class, as it expects the raw instances file restructured to the *list-based formalism*.

When the module is *consulted*, the *Spreading Activation iterative* algorithm could be started by using one of the following publicly accessible predicates: `spreading_activation/3`, `spreading_activation/6`. There is no implementation difference between `spreading_activation/3` and `spreading_activation/6` we simply provide default parameters to simplify the algorithm usage.

The complete parameters definition for the spreading activation algorithm is the following:

- **StartNodes**: list of nodes identifiers from which the algorithm will start spreading (their activation value will be initialized to 1)
- **FiringThreshold**: the minimum activation value a node must have to be *fired* (Its activation value will be spread to its neighbors)
- **DecayRate**: factor by which the spread activation value will *decrease* when passed to another node (for example, if the current node has an *activation*

value of 0.7 and the *decay rate* has a value of 0.9, the activation value for a neighboring node will be  $0.7 \cdot 0.9$ )

- **GeometricDecayFactor**: since nodes are fired multiple times, the activation values for almost all nodes tend to rapidly increase and quickly reach the maximum value 1. To avoid this, a factor is multiplied to the decay rate at each iteration to progressively decrease the decay rate (which in turn decreases the spread activation value)
- **Epsilon**: tolerance threshold, when the **L1 norm** of activation values between two different iterations will be below this specified value, it means that the algorithm has converged and it can be stopped
- **NMaxIter**: in case convergence cannot be reached, this parameter can be set to define the maximum number of iterations for the algorithm, once this number has been reached the algorithm will stop regardless of convergence

In the case of `spreading_activation/6`, all parameters must be specified.

For `spreading_activation/3`, the following parameters are set:

- *GeometricDecayFactor* = 0.8
- *Epsilon* =  $1e - 4$
- *MaxIter* = 100

Once the algorithm has completed, the activation value associated to each node can be checked by using the only other publicly accessible predicate, `activation/2`:

```
activation(378, ActivationValue). --> ActivationValue = 0.25595174...
```

The iterative Spreading Activation algorithm implementation has an internal predicate `node_sa_info/3` (similarly to the *Page Rank* algorithm and the basic *Spreading Activation* algorithm) which is used to keep track of the node activation value at the previous iteration as well as the current one.

```
node_sa_info(NodeID, ActValT0, ActValT1).
```

As it was said in the `GeometricDecayFactor` parameter description, the algorithm tends to bring all the activation values for all the nodes to 1 (the maximum value). This is because our arc weights are set to 1, so the spreading tends to be significant. We do suggest to run the algorithm with a low `DecayRate` and a low `GeometricDecayFactor` (good results were obtained when setting both to 0.5).

Unfortunately, no extensive literature or examples were found related to this variation of the algorithm, so we couldn't compare our implementation and our

results with a state-of-the-art implementation. However, we did check that the obtained results were *meaningful* and *significant*.