UNIVERSITÀ DEGLI STUDI DI BARI ALDO MORO
DIPARTIMENTO DI INFORMATICA

Gr@ph BRAIN

Instances Restructurer - SubGraph Selector

[GitHub repository]

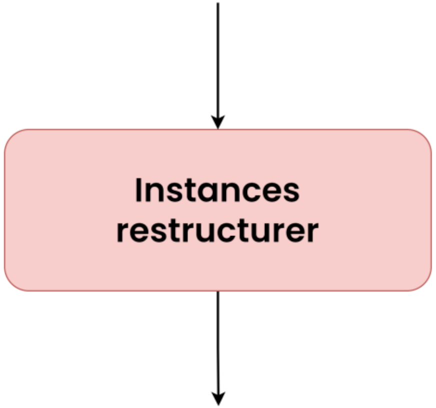
Antonio Silletti Elio Musacchio

Introduction

This project focused on *mainly* **two** objectives:

1. Restructuring the exported *GraphBrain* instances into a ***higher-level*** Prolog representation

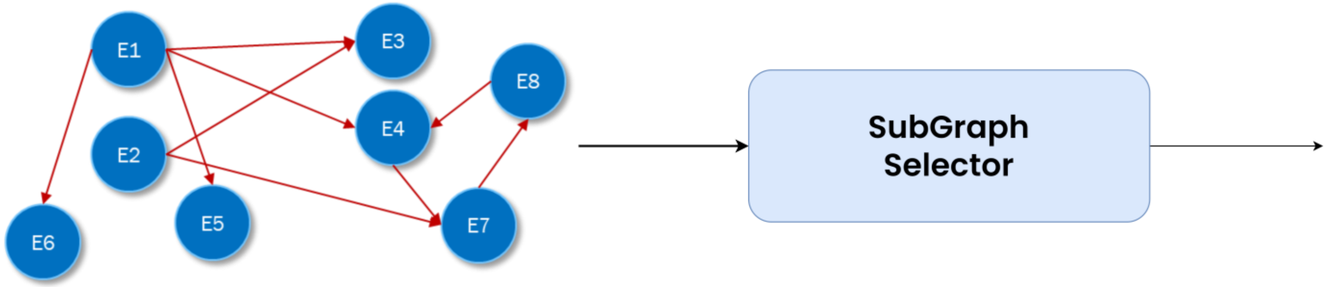
```
node_properties(0,'{name=Jack,gender=M,subClass=Person}').
```




```
'Person'(0, 'Jack', 'M').
```

Introduction

2. Assigning a **score** to instances of the graph, so to select the ***most relevant subportions*** of it



$$\begin{aligned} score(E1) &= 0.033 \\ score(E2) &= 0.033 \\ score(E3) &= 0.054 \\ &\dots \\ score(E8) &= 0.264 \end{aligned}$$

Pipeline

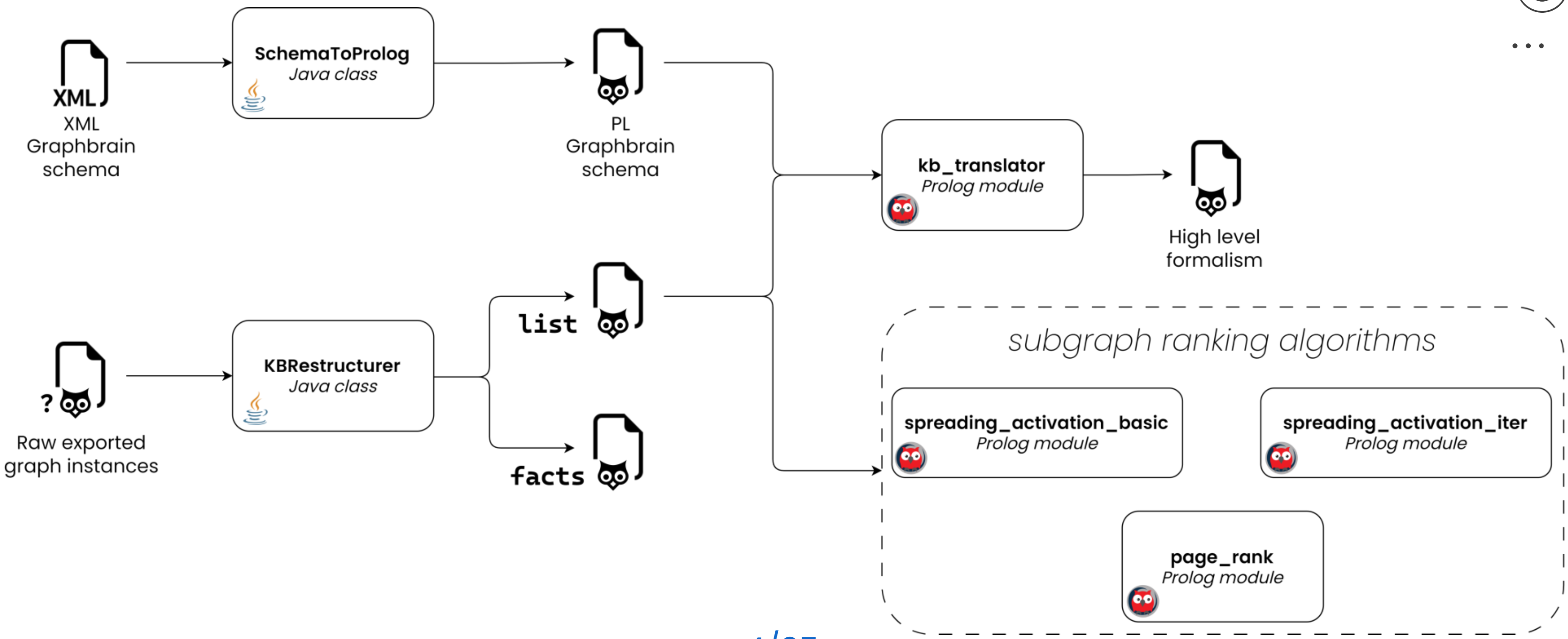

XML
XML Graphbrain schema
SchemaToProlog
Java class
PL Graphbrain schema
kb_translator
Prolog module
High level formalism
?
Raw exported graph instances
KBRestructurer
Java class
list
facts
subgraph ranking algorithms
spreading_activation_basic
Prolog module
spreading_activation_iter
Prolog module
page_rank
Prolog module

Table of Contents

Raw predicates | Preprocessing

- `node/2`: information about **instance domains** and **top-level class**

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

Top-level class

Instance domains

- `node_properties/2`: attributes of the node described as **dict literal**

```
node_properties(4, '{name=New York City, subClass=Town}').
```

- `arc/4`: **Class name** and **SubjectId**-**ObjectID** *pair*

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

KBRestructurer

The `KBRestructurer` *Java class* translates in a more *Prolog-friendly* format the raw instances file.

Missing `arc_properties/2`: lack of coherence!

- Intermediate *optional step* to split `arc/4` in `arc/3` and `arc_properties/2` to make the KB structure **homogeneous**

- Optional since if the *GraphBrain exporting tool* fixes this inconsistence, this step can be **skipped**

```
arc(498238, 'developed', 4095, 337488).
```

```
arc(498238, 4095, 337488).
arc_properties(498238, '{subClass=developed}').
```

KBRestructurer

Poor attributes representation: *difficult Prolog manipulation!*

Two different ways of restructuring attributes:

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

```
node_properties(4, '{name=New York City, subClass=Town}').
```

List

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

Facts

```
name(4, 'New York City').
subClass(4, 'Town').
```

SchemaToProlog

To convert original instances to a *higher-level formalism*, a **reference schema** is needed!

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

Wrong!

```
'Person'(0, 'Jack', 'M').
'Person'(1, 'Alex', '11/05/1972').
```

Impossible to determine a **consistent**, **complete** and **correct** ordering for the attributes of each node of type `'Person'`.

Existing Prolog schema

GraphBrain provides a way to export the ontology schema as a *Prolog* file in the following format:

```
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).
                    …
```

We decided to translate the original XML schema into a more *conventional Prolog format*

```
'Artifact'([name, description]).
subclass_of('Artwork', 'Artifact').
                    …
```

SchemaToProlog dynamic output

The converted *Prolog* schema can be divided into ***sections***:

- Directives
- Entities
  - Entities hierarchy
  - Entities schema
- Relationships
  - Relationships hierarchy
  - Relationships schema
- Generic rules

Entities hierarchy:

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

Entities schema:

```
ClassName(AttributesList).
```

Relationships hierarchy:

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

Relationships schema:

```
RelationshipClassName(ReferencesList, AttributesList).
inverse_of(InverseRelationshipName, RelationshipName).
```

XML Entities to Prolog

```
<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>
```

Entities hierarchy

```
subclass_of('Artwork', 'Artifact').
subclass_of('Handicraft', 'Artifact').
subclass_of('IndustrialWork', 'Artifact').
subclass_of('Component', 'IndustrialWork').
```

...

Entities schema

```
'Artifact'([name, description]).
'Artwork'([]).
'Handicraft'([]).
'IndustrialWork'([]).
'Component'([]).
```

XML Relationships to Prolog

```xml
<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

```prolog
subclass_of('bought', 'acquired').
```

...

Relationships schema

```prolog
'acquired'(['Person'-'Item'], [date]).
inverse_of('acquiredBy', 'acquired').
'bought'([], [price]).
inverse_of('boughtBy', 'bought').
```

SchemaToProlog static output

Directives and **Generic rules** sections are *static*:

- They do *not change* independently of the schema to translate

- Directives
- Entities
  - Entities hierarchy
  - Entities schema
- Relationships
  - Relationships hierarchy
  - Relationships schema
- Generic rules

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

```
is_subclass(?Subclass, ?Superclass).
invert_relationship(+RelName, -InvertedRelClause).
gather_attributes(+SubC, -Attributes).
gather_references(+SubC, -References).
```

Navigating the schema | XML Schema element

The `SchemaToProlog` class works by using a developed utils package which lets you iterate and handle in an easy and intuitive way the original XML schema

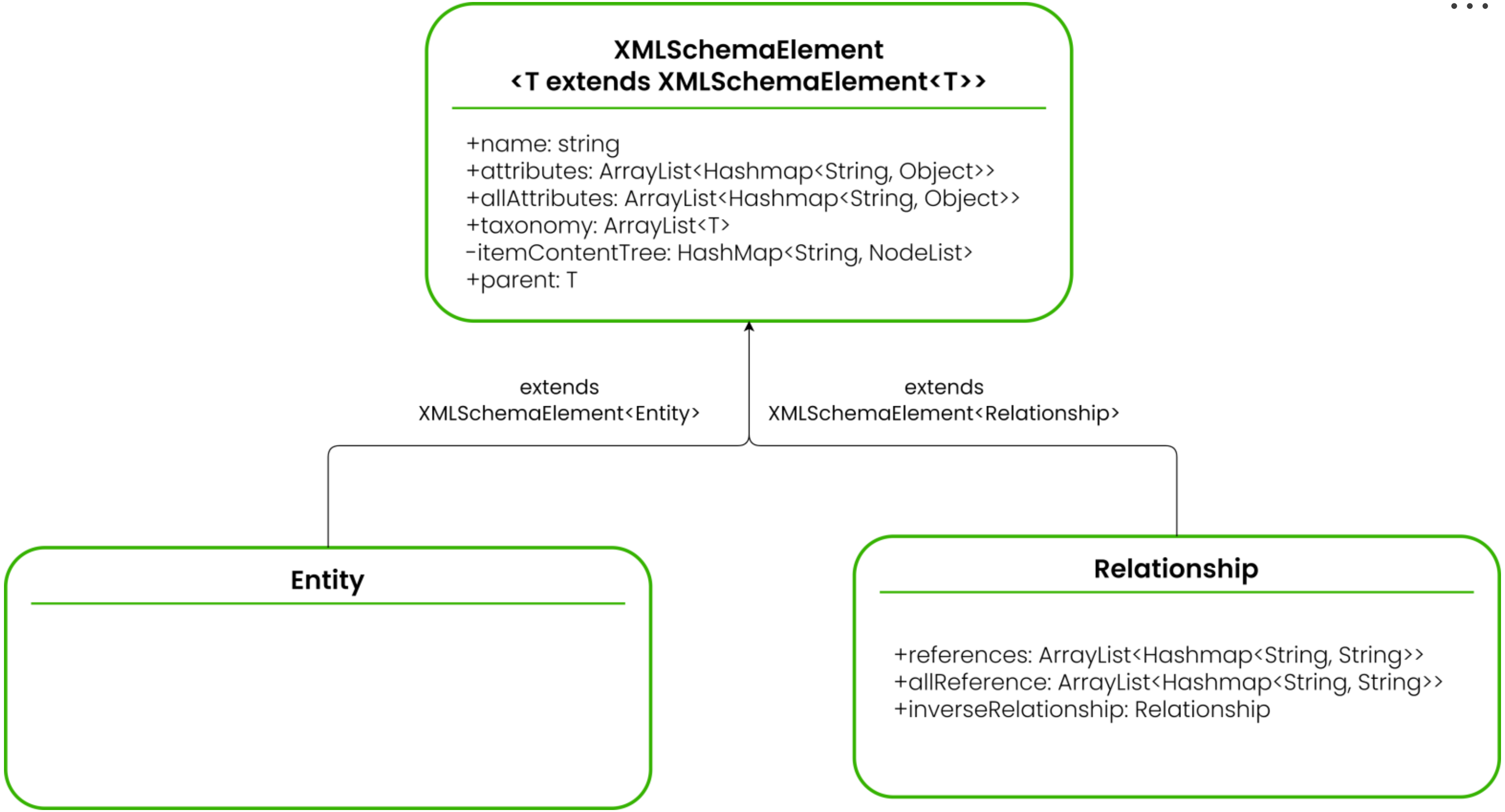
- **further layer of abstraction** for the *GraphBrain* developer!
- *`Entity`*/*`Relationship`* objects with `attributes`, `references`, `taxonomy` …

To start navigating the *GraphBrain* XML file, instantiate the `XMLSchemaScanner` class:

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

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

Simplified UML diagram



Gathering entities with more than five children

```
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 | Instances translation

The **`kb_translator`** *Prolog* module is responsible for performing the conversion of the *GraphBrain* instances to the **higher-level formalism**

This module must be used **only** after running the whole *Java* pipeline

- It expects the raw instances file restructured to the *list-based formalism* and the XML schema file converted to *Prolog*.

```prolog
:- module(kb_translator,
        [
            translate/0,
            translate/2
        ]).
```

`translate/0`: Restructured instances file and Prolog schema automatically loaded from **`outputs`**

`translate/2`: Restructured instances file and Prolog path must be specified

Translation strategy

- **`Entities`** related predicates will be converted in the following ones:

```
topLevelClass(EntityID, TopLevelClassName).
instanceDomains(EntityID, InstanceDomainsList).
ClassName(EntityID, Attribute1, Attribute2, …, AttributeN).
```

Where the *number of attributes* depends on the schema definition

- **`Relationships`** related predicates will be converted in the following ones:

```
ClassName(RelID, SubjectID, ObjectID, Attribute1, Attribute2, …, AttributeN).
```

Where the *number of attributes* depends on the schema definition

Entity translation

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

Original

Higher level formalism

```
topLevelClass(4, 'Place').
instanceDomains(4, ['retrocomputing', 'lam']).
'Town'(4, 'New York City', null, null, null, null, null).
```

High-level translation

Reference Schema

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

Relationship translation

```
arc(498195, 9288, 389).
arc_properties(498195, ['subClass'-'concerns']).
```

Original

Higher level formalism

```
concerns(498195, 9288, 389, null).
```

High-level translation

Reference Schema

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

Rank algorithms | Graph ranking

Since *GraphBrain* is populated by several instances, inspecting the actual graph is a **complex task**.

- To find 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

Four differents Graph ranking algorithms:

Page Rank
Personalized Page Rank

Spreading Activation
Iterative Spreading Activation

Page Rank

This module must be used **only** after running the `KBRestructurer` Java class

- It expects the raw instances file restructured to the *list-based formalism*

Once `consulted`, multiple `page_rank` predicates with different arieties, to **simplify** *usage,* will be available.

```
:- module(page_rank,
        [
            page_rank/0,
            page_rank/3,
            page_rank/5,
            rank/2
        ]).
```

- `DampingFactor`
- `Epsilon`
- `MaxIter`
- `RankStartVector`
- `PersonalizationVector`

Time vs Space

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
```

The `rank/2` predicate, is a simple projection of the `node_pr_info/4` predicate, which stores *relevant information* needed to compute the page rank for each node

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

This implementation favours **time** over **space**:

But **2x increase** in space to obtain a **60x boost** in time *(considering list implementation)*!

- *4 seconds to perform a PR iteration instead of 4 minutes*
- Final time: **≈95** seconds on a graph with 337239 *nodes* and 497933 *arcs*

Personalized Page Rank

In the classic *Page Rank* algorithm, when performing the *random teleport*, all nodes have the *same probability* of being picked:

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

Via the **`PersonalizationVector`** parameter, you could explicitly set custom *personalization values* in the form of **key-value** pairs `NodeID-PersonalizationValue`

- All nodes missing from this list will have `PersonalizationValue` **equal to 0**
- Each `PersonalizationValue` will be normalized by the total sum of the values (to obtain range $[0,1]$)

```
[1-0.4, 10-99.23, 76-4.12]
```

↓

```
[1-0.004, 2-0, …, 10-0.956, 11-0, … 76-0.04, …]
```

PageRank comparison

To test **correctness** and **robustness** of the *PageRank* algorithm implementation in *Prolog*, we compared results with a python implementation of the *PageRank* algorithm using **NetworkX** (state-of-the-art library)

We only modified how the stopping criterion is computed:

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

where $N = number\ of\ nodes$

`nx.pagerank` → `custom_pagerank_scipy`

The results reported are reproducible in the following [colab notebook]

Results example

```
>>> 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] = 0.02446829…
…
```

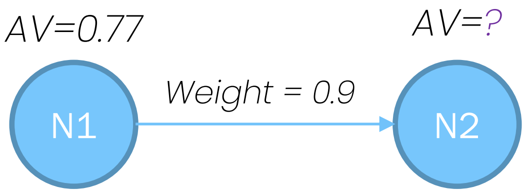
```
?- page_rank(0.85,
|            0.000001,
|            100,
|            [0-5.0, 2-3.2, 6-2.4],
|            [15-3.1, 67-2.3, 457-0.5]).
```

```
rank(0, PageRankValue).   -> PageRankValue = 0.0
rank(1, PageRankValue).   -> PageRankValue = 0.0
…
rank(457, PageRankValue).-> PageRankValue = 0.02446829…
…
```

Spreading Activation

The algorithm works by considering all **unfired** nodes and spreading their activation value to neighbors nodes if it is greater than a set **threshold**.

The algorithm starts by setting $Activation\,Value = 1$ to a set of **starting nodes** and stops when all nodes are marked as *fired* or no node has an activation value *greater than said threshold*.



$$FiringThreshold = 0.45$$

$$DecayRate = 0.7$$

$$IL(N2) = \{N1\}$$

$$AV(N2) = \sum_{n \in IL(N2)} AV(n) \cdot DecayRate \cdot ArcWeight = AV(N1) \cdot DecayRate \cdot ArcWeight = 0.4851$$

Spreading Activation

Activation values are bound to [0,1] range and, for *GraphBrain*, weight of arcs is considered to be 1

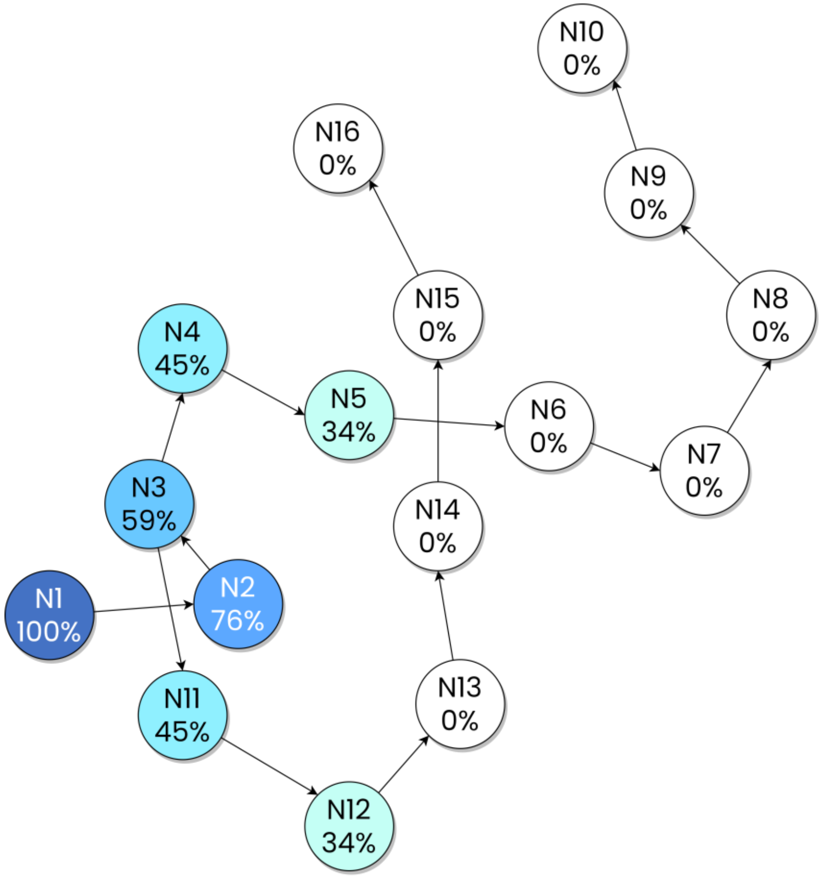
This module must be used **only** after running the `KBRestructurer` Java class

- It expects the raw instances file restructured to the *list-based formalism*

```
:- module(spreading_activation,
        [
            spreading_activation/3,
            activation/2
        ]).
```

- **StartNodes**
- **FiringThreshold**
- **DecayRate**

Comparison with Wikipedia example




```
?- 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

We also implemented a variation of the *Spreading Activation* algorithm where nodes can be fired **multiple times** instead of *only once*

- The algorithm will stop when either convergence (`L1 norm`) or maximum number of iterations has been reached

This module must be used **only** after running the `KBRestructurer` Java class

- It expects the raw instances file restructured to the *list-based formalism*

```
:- module(spreading_activation_iter,
        [
            spreading_activation/3,
            spreading_activation/6,
            activation/2
        ]).
```

- StartNodes
- FiringThreshold
- DecayRate
- GeometricDecayFactor
- Epsilon
- NMaxIter

Reducing activation spread

Since nodes are fired multiple times, the activation values for almost all nodes tend to quickly reach the maximum value 1.

To avoid this, the **`GeometricDecayFactor`** (in range $[0,1]$) is multiplied to the *decay rate* at each iteration to progressively decrease it

$$DecayRate(t+1) = DecayRate(t) \cdot GeometricDecayFactor$$

So, the more iterations you do, the less the activation value will be spread!

- An alternative would be to decrease the decay factor based on the *distance* from the starting nodes

Results example

No extensive literature or examples were found related to this variation of the algorithm, so we couldn't compare our results with a *state-of-the-art* implementation.

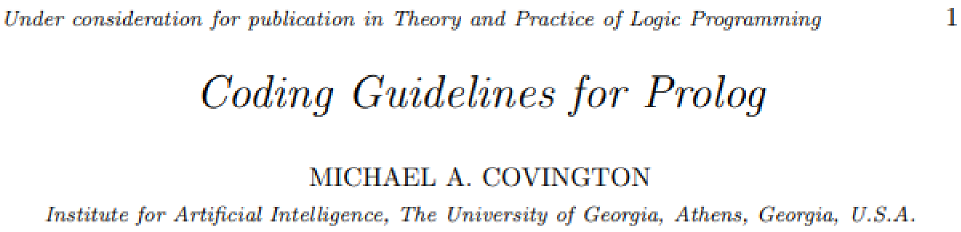
However, we did check that the obtained results were meaningful and significative.

```
?- spreading_activation([1], 0.35, 0.65,
|                        0.5, 0.0001, 100).

?- activation(1, ActivationValue).
ActivationValue = 1

?- activation(2, ActivationValue).
ActivationValue = 1

?- activation(3, ActivationValue).
ActivationValue = 1

?- activation(4, ActivationValue).
ActivationValue = 0.822586…

…

?- activation(11, ActivationValue).
ActivationValue = 0.822586…

?- activation(12, ActivationValue).
ActivationValue = 0.399695…

?- activation(13, ActivationValue).
ActivationValue = 0.030684…
```

Additional information

- All *Java* code has been thoroughly documented following [Javadoc]() standard

- All Prolog code has been thoroughly documented following ["Coding Guidelines for Prolog" paper]() by *Michael A. Covington et Al.*


Under consideration for publication in Theory and Practice of Logic Programming 1

Coding Guidelines for Prolog

MICHAEL A. COVINGTON

Institute for Artificial Intelligence, The University of Georgia, Athens, Georgia, U.S.A.

- All code is public and hosted on [GitHub]() with information regarding **Requirements, How To Use** and **Project Organization**

UNIVERSITÀ DEGLI STUDI DI BARI ALDO MORO

DIPARTIMENTO DI INFORMATICA

Thanks!

GitHub repository

Antonio Silletti

Elio Musacchio