VIDAL Adrien - 21206314 - M1 MABS parcours Bioinformatique - 02/05/2016

Projet bibliothèque python: GeneOntology

Les fichiers cités sont disponibles en annexe sous les chemins (UNIX) relatifs donnés.

Bibliothèque: `./Graph.py` (exécutable avec python 2.7).

1 Classes et méthodes créées au cours des TPs:

Les classes et méthodes implémentées dans le cadre des TPs seront utilisées pour l'implémentation de l'Ontology. Elles sont résumées ci-dessous avec leurs complexités respectives et un résumé de leurs fonctions. La **figure 1** donne le diagramme de classes de la bibliothèque.

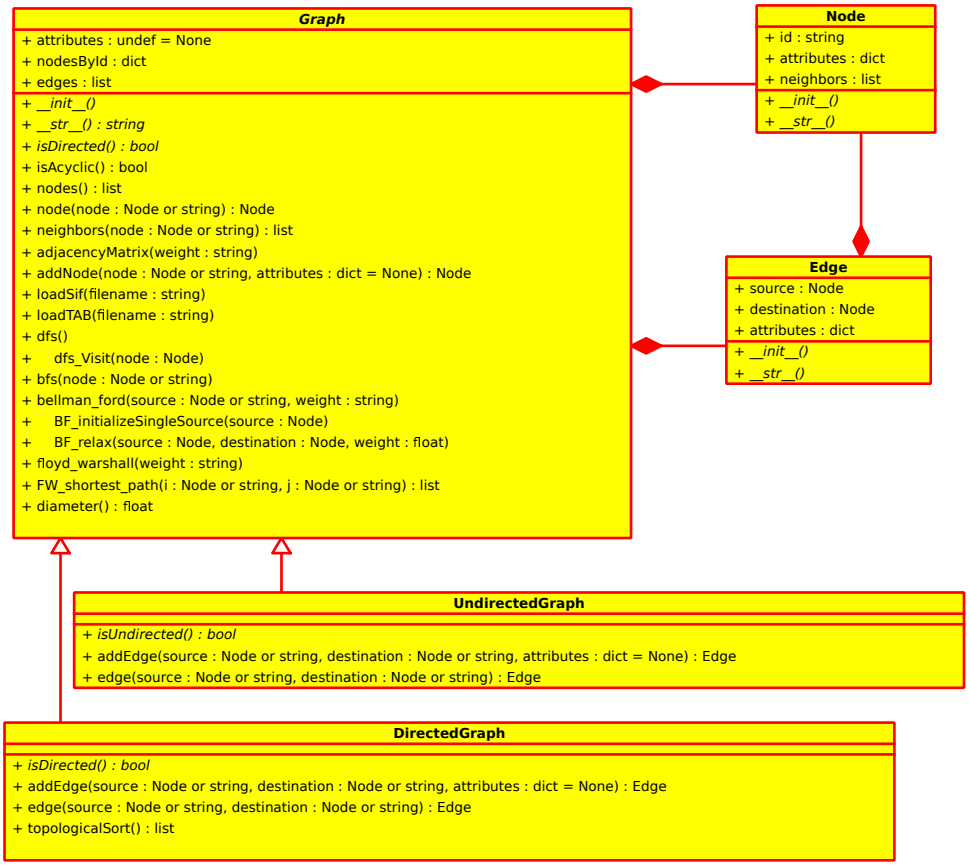

Figure 1. Diagramme de classe de la bibliothèque à l'issue des TP.

Classe: Graph:

Avec: V = Enssemble des nodes dans le graphe.
E = Enssemble des arrêtes dans le graphe.
$\boldsymbol{l}$ = Enssemble des lignes dans le fichier.

*: La complexité de la méthode augmente lorsque la taille du graphe augmente.

Méthode	Fonction	Complexité
`addNode(node)`	Ajoute une node au graphe si elle n'existe pas déjà.	$\boldsymbol{O}(\|V\|)$
`node(objet)`	Renvoie la node correspondante si elle existe.	$\boldsymbol{O}(\|V\|)$
`nodes()`	Renvoie une liste triée des nodes.	$\boldsymbol{O}(\|V\|)$
`neighbors(node)`	Renvoie une liste triée des voisins de la node.	$\boldsymbol{O}(\|V\|)$
`isAcyclic()`	Appelle `dfs()` afin de tester si le graphe est acyclique.	$\boldsymbol{O}(\|V\|+\|E\|)$
`adjacencyMatrix(weight)`	Calcule une matrice d'adjacence d'un graphe pondéré.	$\boldsymbol{O}(\|V\|^2)$
`dfs()` et `dfs_visit(node)`	Parcours en profondeur du graphe.	$\boldsymbol{O}(\|V\|+\|E\|)$
`bfs(source)`	Parcours en largeur du graphe	$\boldsymbol{O}(\|V\|+\|E\|)$
`bellman_ford(source, weight)` et `BF_initializeSingleSource(..)` et `BF_relax(..)`	Calcul des plus cours chemin depuis une source unique.	$\boldsymbol{O}(\|V\|\cdot\|E\|)$
`floyd_warshall(weight)`	Calcul des plus cours chemins entre chaque paire de nodes.	$\boldsymbol{O}(\|V\|^3)$
`FW_shortest_path(i,j)`	Récupération du plus cours chemin de i à j.	$\boldsymbol{O}(\|V\|)$
`diameter()`	Calcule le diamètre (plus long des plus courts chemins).	$\boldsymbol{O}(\|V\|^2)$
`loadSIF(filename)`	Charge un fichier .sif dans un graphe.	$\boldsymbol{O}\Big(\int_0^{\|\boldsymbol{l}\|}(2\|V\|\,d\|V\|)\Big)$*
`loadTab(filename)`	Charge un fichier .tab dans un graphe.	$\boldsymbol{O}\Big(\int_0^{\|\boldsymbol{l}\|}(2\|V\|\,d\|V\|)\Big)$*

Classe: DirectedGraph:

Avec: V = Enssemble des nodes dans le Graphe.
E = Enssemble des arrêtes dans le Graphe.

Méthode	Fonction	Complexité
`addEdge(source, destination)`	Ajoute une arrête entre deux nodes dans le graphe. Créée les nodes (`addNode()`) si elles n'existent pas.	$\boldsymbol{O}(2\|V\|)$
`isDirected()`	Confirme que ce graphe est un graphe dirigé.	$\boldsymbol{O}(0)$
`edge()`	Renvoie l'arrête correspondante si elle existe.	$\boldsymbol{O}(\|E\|)$
`topologicalSort()`	Construit un ordre topologique des nodes via `dfs()` si le graphe est acyclique.	$\boldsymbol{O}(\|V\|+\|E\|)$

2 Analyse et compréhension de la GeneOntology:

La GeneOntology est un système complexe de classification d'annotations fonctionnelles pouvant être utilisé pour catégoriser des banques de données complètes pour de nombreux organismes. Sa structure se prête particulièrement à une représentation en graphe et à une analyse par des outils de parcours de graphes.

2.1 La Gene Ontology:

La GeneOntology (GO) est une collection de définitions et classes qui décrivent des fonctions génétiques de manière abstaite ainsi que les relations entre ces concepts. Elle ne représente pas les fonctions génétiques d'un organisme en particulier mais sert plutôt de système de classification des fonctions en terme de concepts des plus généraux aux plus spécifiques. Tout terme de l'ontologie est en relation avec au moins un autre terme par une relation d'identité (*"is a"*).

La GO est en réalité composée de trois ontologies distinctes: Les fonctions moléculaires, les composants cellulaires et les procédés biologiques. Toutes les fonctions au sein d'une de ces ontologies sont reliées directement ou indirectement à toutes les autres par une relation d'identité de manière hiérarchique. De fait, tout terme d'une ontologie est lié par ces termes prédécesseurs à un unique terme racine qui précède tous les autres. Un terme *enfant* représente une fonction plus spécialisée que celle de son terme *parent*. Cette hiérarchie n'est cependant pas stricte car un terme peut être lié à plus d'un terme prédécesseur.

Aucune relation d'identité ne peut exister entre les trois ontologies mais d'autres types de relations peuvent exister entre les termes de différentes ontologies. Ces autres relations représentent également une hiérarchie. Aucun terme ne peut être en relation avec lui même.

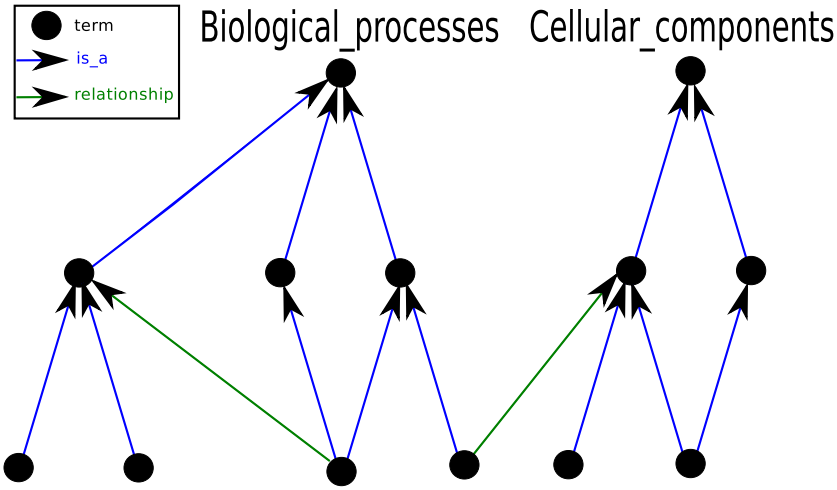

Figure 2. Exemple de structure de graphe pouvant représenter une GO. Ici, les arrêtes des relations ont pour source le terme *enfant* et pour destination le terme *parent*.

La GeneOntology peut être conceptualisée en un graphe mathématique dirigé dont les sommets représentent les termes et les arrêtes les relations entre les concepts de ces termes. Des propriétés de ce graphe se dégagent alors de la structure de l'ontologie:

→ Le graphe est acyclique du fait de la hiérarche des ontologies.

→ Il ne peut y avoir au maximum une arrête entre toute paire de sommet. Le graphe est donc un graphe dirigé simple.

→ Chaque sommet représentant un terme a pour ordre minimum 1.

→ Chaque ontologie représente une composante connexe du sous-graphe ne comprenant que les arrêtes qui représentent des relations d'identité. Autrement dit, ce sous graphe n'est composé ni plus ni moins de trois composantes connexes.

→ Le graphe n'est pas fortement connexe.

→ Il existe un chemin entre tout sommet d'une ontologie et le sommet racine de cette ontologie.

→ Tous les termes d'une ontologie à l'exception de sa racine possèdent au moins un terme prédecesseurs.

→ Un terme peut être lié à plus d'un terme prédécesseur. On ne peut donc pas considérer la GO comme étant un arbre.

L'orientation des arrêtes représentant les relations est arbitraire mais doit être le même pour toutes les relations d'identité ou autre. Chaque racine est donc un puit ou une source selon l'orientation choisie. Les termes n'ayant pas de successeurs seront donc des puits ou sources vice versa selon le choix fait. La **figure 2** représente l'un de ces choix de structure.

Les relations ont des propriétés de transitivité. Par exemple, si une relation *is a* existe entre des termes A et B et entre des termes B et C, alors une relation implicite *is_a* existe entre A et C: is_a∘is_a→is_a. Ces relations implicites ne seront pas à représenter sur un graphe.

Les autres relations qui peuvent exister entre les termes sont:

- *part of*: Une relation de parenté différente de *is a*. Cette relation représente une appartenance structurelle. Le terme *enfant* d'une telle relation entre toujours dans la composition du terme *parent*. Par exemple la mitochondrie fait toujours partie du cytoplasme. Cette parenté est transitive avec elle même et avec *is_a*.
- *has part*: Est le complément logique à *part of*. Le terme *enfant* d'une telle relation contient toujours dans sa composition du terme *parent*. Cette parenté est transitive avec elle même et avec *is_a*.
- *regulates* +/-: Ne représente pas des liens de parenté mais de régulation entre les termes.

Les fonctionnalités que l'on veut fournir dans cette bibliothèque consistent à parcourrir les liens de parenté entre les termes. On ne s'intéressera donc pas aux réseaux de régulation.

Il existe plusieurs versions de la GO fréquemment mises à jour: une version *core*, et une version *plus* (qui comprend le core et des extensions) ainsi qu'une version *slim* qui contient substentiellement moins de termes.

2.2 Les annotations:

Contrairement aux ontologies, les annotations représentent des fonctions génétiques physiques et spécifiques à un organisme mesurées ou déduites expérimentalement. Ces annotations proviennent généralement d'autres bases

de données. Des fichiers d'annotation pour la GO décrivent des relations entre ces fonctions réelles et les concepts et classes de la GO. Une annotation peut être liée à plus d'un terme de la GO parmi les trois ontologies. Il se peut également qu'une annotation soit associée à plusieurs termes partageant une relation de parenté.

Sur un graphe dirigé acyclique (DAG) représentant la GO, chaque annotation peut être modélisé par un sommet lié à au moins un sommet par un nouveau type de relation (*"product of"*). Le sens de ces arrêtes de relation doit être consistent avec celui choisi pour les relations de l'ontologie.

La **figure 3** représente l'ajout des annotations au graphe de la **figure 1**.

Selon le sens choisi, les sommets représentant des annotations seront tous soit des puits, soit des sources (au contraire de la racine de chaque ontologie).

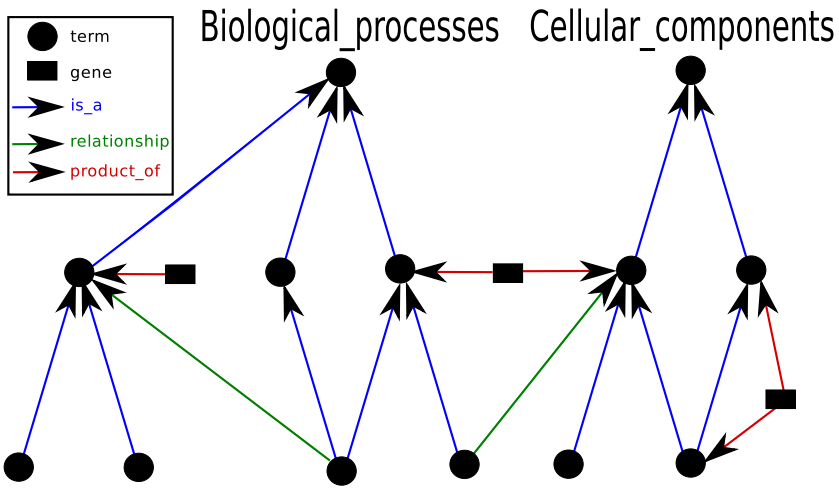

Figure 3. Exemple de structure de graphe pouvant représenter une GO et les associations possibles d'annotations à la GO.

Les annotations sont fournies et maintenues par des groupes et bases de données indépendantes. La pertinence de ces annotations par rapport aux versions disponibles de la GO peut dépendre de la fréquences à laquelle ces annotations sont mises à jour.

3 Représentation et implémentation dans la bibliothèque:

Comme vu précédement, il y a de multiples possibilités de représentation de la GO sous forme de graphe. Dans le cadre du développement d'une bibliothèque, il faut donc en choisir une pertinemment et la conceptualiser sous forme de classes et de méthodes pour pouvoir l'implémenter dans le language utilisé.

3.1 Conception objet:

Certaines des méthodes que nous désirons implémenter seront spécifiques au graphe représentant la GO. Il serait donc judicieux de créer une nouvelle classe pour de tels graphes. La GO peut être représentée par un DAG, la classe à créer héritera donc de la classe `DirectedGraph.` Cette nouvelle classe, `GeneOntology`, contiendra des nodes représentant les termes et d'autres représentant les associations ainsi que des arrêtes représentant les relations entre les nodes.

Les termes de la GeneOntology et les annotations possèdent différents attributs qu'il sera intéressant de conserver. Il est de même pour les relations possibles entre les concepts représentés par les termes et la relation d'association des annotations aux termes.

Ultimement, tout ces attributs pourraient être stockés dans le dictionnaire `attributes` des objets de classe `Node` et `Edge`. Cependant, il est important de bien distinguer les termes des annotations du fait que ces deux types de sommets ont des propriétés différentes au sein du graphe. On implémentera donc deux nouvelles classes d'objets héritant de `Node`, `GOTerm` et `GeneProduct` qui possèderont de nouveaux attributs pour stocker leurs informations. De

plus, celà permet de définir pour chacune, une nouvelle mise en forme d'impression.

Les arrêtes ne diffèrent que par le type de relation qu'elles représentent, il n'est pas forcément nécessaire de créer une nouvelle classe. Le type de relation sera donc stocké dans le dictionnaire `attributes` des objets de classe `Edge`.

Pour ce qui est de l'orientation des arrêtes, il correspondra à ce qu'indique la relation:

$$[\text{GO: 0000001}] \xrightarrow[\text{is\_a}]{} [\text{GO: 0048308}] \Leftrightarrow \texttt{"}\textit{GO:0000001 is a GO:0048308}\texttt{ "}.$$
$$[\textit{A9LNK9}] \xrightarrow[\text{product\_of}]{} [\text{GO: 0003677}] \Leftrightarrow \texttt{"}\textit{A9LNK9 is a product of GO:0003677}\texttt{ "}.$$

Il est donc à noter que dans le graphe, les termes *enfant* seront des prédecesseurs des termes *parent* et vice versa. Il faudra tenir compte de celà pour l'implémentation ou l'utilisation de méthodes de parcours de graphe.

La **figure 4** représente le nouveau diagramme de classe de la bibliothèque (méthodes présentées dans la section suivante).

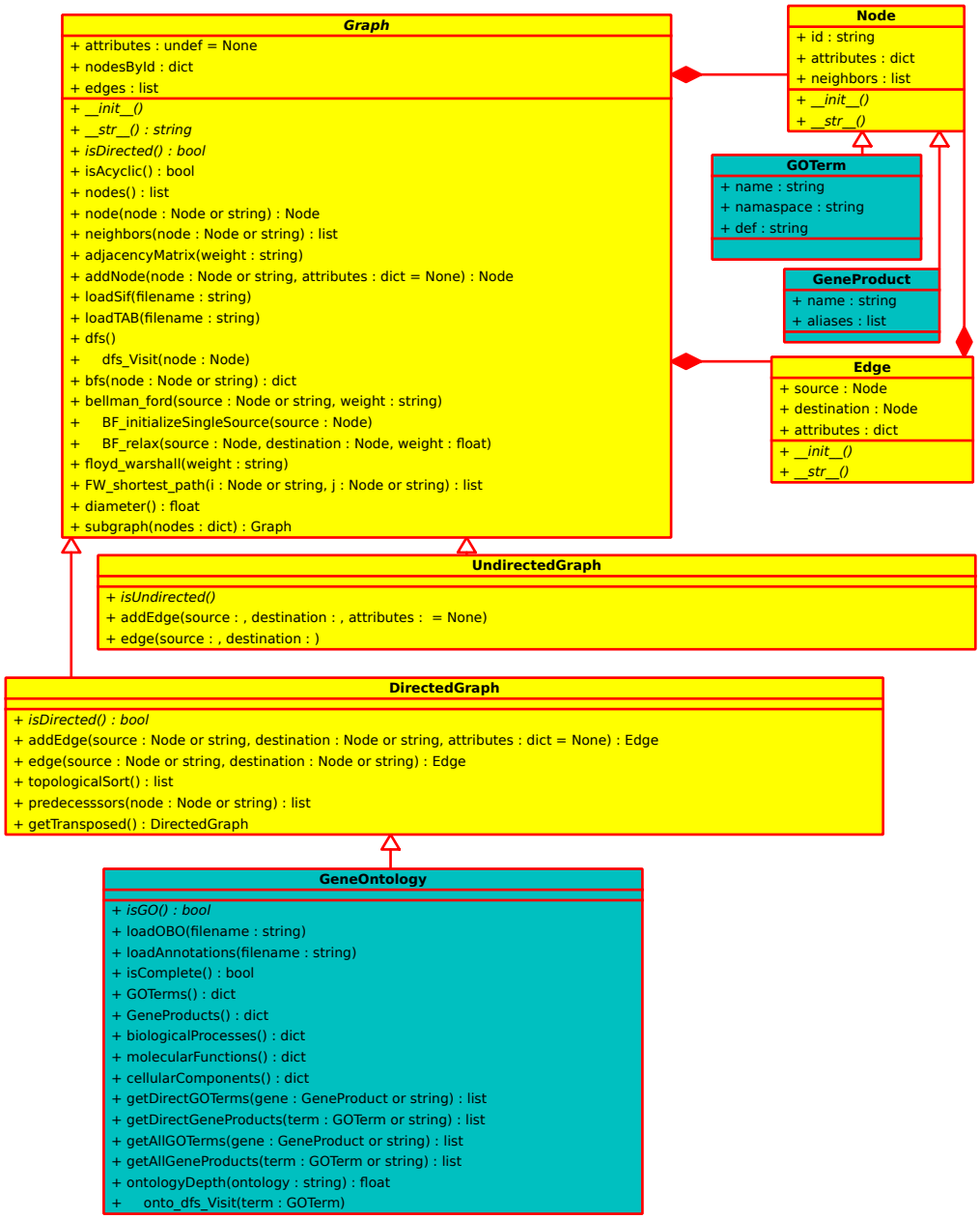

Figure 4. Diagramme de classe de la bibiothèque après implémentation de la GeneOntology et des méthodes.

3.2 Choix des fichiers et méthodes de chargement:

Je télécharge le fichier `go-basic.obo` depuis le site de GeneOntology ainsi que des fichiers d'annotations depuis le FTP. Le fichier `./gene_association.goa_arabidopsis` est complet et comporte un large nombre de termes sans pour autant être trop lourd. Afin de réaliser des tests plus rapidement, j'utilise des fichiers moins massifs. `./goslim_generic.obo` est une version très condensée de l'ontologie qui sera bien plus rapide à parser et analyser. Pour les annotations, je créée un fichier qui contient les ∼5000 premières annotations: `./gene_sample.goa_arabidopsis` Ces méthodes de chargement pouvant prendre un certain temps à s'exécuter avec de gros fichiers, j'implémente une méthode permettant d'imprimer une barre de progression simple: `parse_pbar(currentLine, numLines)` dans la classe `Graph`.

Avec:
- T = Enssemble des nodes `GOTerm` dans le graphe.
- P = Enssemble des nodes `GeneProduct` dans le graphe.
- V = T + P = Enssemble des nodes dans le graphe.
- E = Enssemble des arrêtes dans le graphe.
- $\boldsymbol{t}$ = Enssemble des identifiants `GOTerm` des tags `id`, `is_a` ou `part_of` dans le fichier.
- $\boldsymbol{l}$ = Enssemble des lignes dans la table d'annotations.

3.2.1 Chargement de l'Ontology: `loadOBO(filename):`

Le fichier `go-basic.obo` est lourd, il faut donc tenter de limiter au possible les calculs à effectuer par ligne. J'utilise les expressions régulières de la librairie python `re` afin de reconnaître les lignes correspondant aux tags à prendre en compte. Néanmoins ces tags apparaissent également pour des blocs ne correspondant pas à des termes et qu'il serait désirable d'ignorer. Utiliser une expression régulière correspondant à un bloc terme entier n'est pas viable car le nombre de tags contenu dans les blocs peut varier.

Je subdivise donc les calculs à effectuer entre une boucle externe qui à pour rôle de détecter les blocs terme et une boucle interne qui démare lorsque un terme est détecté pour reconnaître les tags. La boucle externe ne s'arrête qu'à la fin du document. La boucle interne n'est lancée que lorsque une ligne `[Term]\n` apparaît et compare chaque ligne du bloc aux expressions régulières correspondant aux tags à conserver. Celà permet d'ignorer automatiquement les lignes de commentaire ainsi que les blocs qui ne correspondent pas à des termes.

Les match aux expressions régulières sont effectués dans l'ordre d'apparition des tags dans le bloc et l'instance de la boucle est terminée par `continue` dès qu'une expression est matchée afin de ne pas faire de match inutiles. Les termes obsolètes sont éliminés dès que le mot `obsolete` apparaît dans le tag `name` (là où il apparaît le plus tôt dans le bloc) et on sort de la boucle interne par un `break`.

A chaque terme détecté, au moins une node est ajoutée (le terme lui même). D'autres termes sont ajoutés s'ils n'existent pas déjà pour chaque relation conservée (`is_a` et `part_of`) ainsi que les arrêtes re. La complexité des méthodes `addNode(node)` et `addEdge(source, destination)` étant linéairement dépendantes du nombre total de termes |V| présents dans le graphe. De fait, la complexité de chaque ajout de node ou d'arrête. L'ajout d'arrête étant plus lourd, on se base sur la complexité de `addEdge(source, destination)` pour calculer la complexité maximale de la méthode: $\boldsymbol{O}\Big(|l| + \int_0^{|\boldsymbol{t}|} (2|V|\, d|V|)\Big)$.

J'implémente une méthode `isComplete()` dans `GeneOntology` afin de vérifier que l'ontologie chargée est complète. Un fichier de GO incomplet fairait créer des nodes dans attributs via `addEdge()`. Cette méthode est de complexité $\boldsymbol{O}(|V|)$.

```
IS-COMPLETE(G)
  ▷Input: ontology graph (DAG): G = (V, E)
  ▶Output: boolean value
  T ← GOTERMS(G) # Voir: GOTErms(node): 2.1.2
  for u term ∈ T
    do if name(u) = ""
         then return False
    if namespace(u) = ""
      then return False
  return True
```

```
LOAD-OBO(G, filename)
  ▷Input: empty ontology graph G = (V, E)
  ▷Input: name of .obo file to parse: filename
  f ← open(filename, read )
  line ← readline(filename)
  while line ≠ ""
   do if line = "[Term]\n"
      then line ← readline(f) - "\n"
       while line ≠ ∅
        do tmp ← match("id: (GO:\d+)", line)
           if tmp = True
            then id ← match_group(1)
                 V[term(id)] ← term ← new GOTerm(id)
                 line ← readline(f) - "\n"
                 next while
           tmp ← match("name: (.+)", line)
           if tmp = True
            then if "obsolete" not in name
              then term(name) ← match_group(1)
                   line ← readline(f) - "\n"
                   next while
             else
              then delete(V[term(id)])
                   line ← readline(f) - "\n"
                   exit while
           tmp ← match("namespace: (.+)", line)
           if tmp = True
            then term(namespace) ← match_group(1)
                 line ← readline(f) - "\n"
                 next while
           tmp ← match("def: (.+)", line)
           if tmp = True
            then term(def) ← match_group(1)
                 line ← readline(f) - "\n"
                 next while
           tmp ← match("is_a: (GO:\d+)", line)
           if tmp = True
            then dest ← match_group(1)
              append(E, edge e(u = term, v = new GOTerm(dest)))
               e(type) ← "is_a"
                line ← readline(f) - "\n"
                next while
          tmp ← match("relationship: (.+) (GO:\d+)", line)
          if tmp = True
           then if match_group(1)="part_of" or "has_part"
             then dest ← match_group(2)
               append(E, edge e(u = term, v = new GOTerm(dest)))
               e(type) ← match_group(1)
              line ← readline(f) - "\n"
              next while
           line ← readline(f) - "\n"
    line ← readline(f)
  assert IS-COMPLETE(G)
```

```
.obo file (f):

format-version: 1.2
data-version: releases/2016-03-09
date: 08:03:2016 12:47
saved-by: tb
auto-generated-by: TermGenie 1.0
subsetdef: goantislim_grouping ...
subsetdef: gocheck_do_not_annotate ...
...

[...]

[Term]
id: GO:0051604
name: protein maturation
namespace: biological_process
def: "Any process leading ..." [GOC:ai]
subset: goslim_chembl
subset: goslim_generic
subset: goslim_pir
subset: goslim_pombe
subset: goslim_yeast
subset: gosubset_prok
is_a: GO:0008150 ! biological_process
relationship: part_of GO:0008150 ! ...

[Term]
...

[...]

[Typedef]
id: ends_during
name: ends_during
namespace: external
xref: RO:0002093

[...]

[Term]
...
```

La partie colorée du fichier correspond aux blocs [Term] qui seront lus par la boucle interne.
La syntaxe des expressions régulières (librairie `"re"`) est la même que celle de Perl.
Les expressions régulières sont colorées comme la partie de la ligne à laquelle elles correspondent et les groupes sont soulignés.

3.2.2 Chargement des annotations: `loadAnnotations(filename):`

Pour ce parseur, chaque ligne de la table doit être lue. Après avoir ignoré les lignes de commentaire (commençant par `!`) chaque ligne de la table est découpée dans une liste pour être accédés lorsqu'ils doivent être ajoutés au graphe. Les annotations associés à des nodes `GOTerm` qui n'ont pas étés ajoutés au graphe lors du chargement de la GeneOntology ne sont pas ajoutés au graphe. Celà permet d'éviter les conflits si l'annotation ne correspond pas exactement à la version utlisée de la GeneOntology. De fait chaque ajout demande la comparaison de l'annotation du terme à ajouter au reste du graphe.

Mais il n'est pas nécéssaire que la complexité de cette comparaison n'augmente avec la taille du graphe si on ne la fait que sur le sous graphe des nodes `GOTerm`. Afin de créer d'avoir accès à ce sous-graphe, j'impléménte une méthode `GOTerms()` qui renvoie un dictionnaire par `id` des nodes `GOTerm` du graphe (T). De même j'impléménte une

méthode `GeneProducts()` qui fait de même (|P|) avec les annotations pour une potentielle utilisation ultérieure dans la librairie ou par l'utilisateur. Ces deux méthodes sont de complexité $\boldsymbol{O}(|V|)$.

`GOTERMS(`G`)`
▷`Input:` ontology graph (DAG): $G=(V,E)$
▶`Output:` terms subset: $T \subset V$
`for each` node $u \in V$
  `do if` u `is a GOTerm`
    `then` $T[u(\text{id})] \leftarrow u$
`return` T

`GENEPRODUCTS(`G`)`
▷`Input:` ontology graph (DAG): $G=(V,E)$
▶`Output:` terms subset: $P \subset V$
`for each` node $u \in V$
  `do if` u `is a GeneProduct`
    `then` $P[u(\text{id})] \leftarrow u$
`return` P

Chaque ajout de product appelle `addEdge(source, destination)`. La complexité de cette fonction augmentera donc en fonction de 2|V| à l'ajout de chaque `GeneProduct`. Le chargement des annotations ajoute jusquà $|\boldsymbol{l}|$ nodes de type `GeneProduct` au graphe: $\boldsymbol{O}\Big(\int_0^{|\boldsymbol{l}|} (|T| \cdot 2|V|\, d|V|)\Big)$.

`LOAD-ANNOTATIONS(`G, filename`)`
▷`Input:` ontology graph (DAG): $G=(V,E)$
▷`Input:` name of annotation file to parse: filename
$T \leftarrow$ `GOTERMS(`G`)`
$f \leftarrow$ `open(`filename, read `)`
line $\leftarrow$ `readline(`f`) - "\n"`
`while` line $\neq \emptyset$
  `do if` line[0] $=$ `"!"`
    `then` line $\leftarrow$ `readline(`f`) - "\n"`
      `next while`
  $c[\,] =$ `split(`line, `"\t")`
  `if` $c[4] \in T$(id)
    `then` V[gene(id)] $\leftarrow$ gene $\leftarrow$ `new GeneProduct(`$c[1]$`)`
      gene(name) $\leftarrow c[2]$
      `append(`E, edge $e(u=$ gene, $v($id$) = c[4]$`))`
      e (typr) $\leftarrow$ `"product_of"`
      gene(evidence_code) $\leftarrow c[6]$
      gene(aliases) $\leftarrow$ `split(`$c[10]$, `"|")`
  line $\leftarrow$ `readline(`f`) - "\n"`

`annotation file (`f`):`

```
UniProtKB  A0A0A7EPL0  PIAL1  GO:0005634  GO_REF:0000039  IEA  UniProtKB-SubCell:SL-0191  C  E4  SUMO-protein  ligase  PIAL1
PIAL1_ARATH|PIAL1|EMB3001|At1g08910|F7G19.21  protein taxon:3702  20160312  UniProt
```

Les identifiants de colonne sont colorés comme la partie de la ligne à laquelle elles correspondent.

3.3 Méthodes de parcours de la GeneOntology:

Le graphe $G(V, E)$ obtenu est plutôt grand, il faut donc privilégier les méthodes de parcours de graphe présentant la plus petite complexité possible.

Avec:
T = Enssemble des nodes `GOTerm` dans le graphe.
P = Enssemble des nodes `GeneProduct` dans le graphe.
V = T + P = Enssemble des nodes dans le graphe.
E = Enssemble des arrêtes dans le graphe.

Parmi les méthodes à implémenter, nous savons à priori que certaines d'entre elles parcourront des chemins ascendants et d'autres des chemins descendants. Quel que soit l'orientation choisie pour les arrêtes du graphe, il faudra pouvoir parcourrir le graphe dans l'orientation inverse des arrêtes.

En premier lieu, j'avais pensé à implémenter une méthode similaire à `neighbors(node)` spécifique au `DirectedGraph` qui permet d'accéder aux prédécesseurs d'une node donnée. Cet accesseur, `predecessors(node)`, avait pour but d'être utilisé dans des méthodes de parcours afin de pouvoir explorer l'arbre en suivant l'oposé de leur orientation. Les succésseurs étant déjà implémentés dans les objets de type `Node` pour un graphe dirigé (attribut `neighbors`, la méthode `neighbors()` a pour complexité maximale $\boldsymbol{O}(|V|)$. Les succésseurs ne sont néanmoins pas implémentables directement sur les nodes car ce serait incompatible avec les graphes non-dirigés. La méthode doit donc visiter les listes d'adjacence des autres nodes du graphes afin d'y rechercher la destination donnée.

```
PREDECESSORS(G, v)
   ▷Input: directed graph: G = (V, E)
   ▷Input: Node: v
   ▶Output: sorted set of nodes: Π
   for each node u ∈ V
      do if u ∈ NEIGHBORS(G,v)
         then append(Π, u)
   Π ← sort(u_i ∈ Π, u_i(id) lexicographically)
   return Π
```

Néanmoins, l'utilisation de cette méthode pose deux problèmes majeurs. Premièrement, même si sa complexité maximale et égamement de $\boldsymbol{O}(|V|)$, chaque parcours des potentiels prédécesseurs s'effectue toujours en un temps |V|·linéaire. Elle est donc bien plus coûteuse que `neighbors(node)`. Deuxièmement, l'utilisation de cette méthode requiérait de devoir réimplémenter les algorithmes de parcours de graphe qui ont déjà étés implémentés dans les classes mères du graphe.

Une solution plus modulaire au problème de l'orientation des arrêtes est de modifier le graphe à parcourrir afin de pouvoir réutiliser les algorithmes précédement implémentés. J'ai néanmoins conservé l'implémentation de cet accésseur car il peut s'avérer utile pour l'utilisateur ou pour calculer le degré total d'une node dans un graphe dirigé.

Nous savons aussi que le calcul de la profondeur de chaque ontologie se fera sur une sous-partie de la GO. Les termes sont faciles à classer par ontologie de par leur attribut `namespace`. Trois méthodes simples de complexité $\boldsymbol{O}(|V|)$ permettent d'extraire une collection des nodes de chaque ontologie dans T:

```
BIOLOGICAL-PROCESES(G)
   ▷Input: ontology graph (DAG): G = (V, E)
   ▶Output: terms subset: B ⊂ T ⊂ V
   T ← GOTERMS(G)
   for each term u ∈ T
      do if namespace[u] = "biological_process"
         then B[u(id)] ← u
   return B
```

```
MOLECULAR-FUNCTIONS(G)
   ▷Input: ontology graph (DAG): G = (V, E)
   ▶Output: terms subset: F ⊂ T ⊂ V
   T ← GOTERMS(G)
   for each term u ∈ T
      do if namespace[u] = "molecular_function"
         then F[u(id)] ← u
   return F
```

```
CELLULAR-COMPONENTS(G)
   ▷Input: ontology graph (DAG) G = (V, E)
   ▶Output: terms subset C ⊂ T ⊂ V
   T ← GOTERMS(G)
   for each term u ∈ T
      do if namespace[u] = "cellular_component"
         then then C[u(id)] ← u
   return C
```

Pour le calcul de la profondeur, il faudra que les arrêtes et listes d'adjacence soient mise à jour dans un sous-graphe. On implémentera plutôt une méthode de création de sous-graphe dans la fonction mère `Graph` afin qu'elle soit utiliable en toute circonstance.

3.3.1 Méthodes de modification du graphe:

Les méthodes de modification du graphe requièrent en réalité de créer une nouvelle instance de graphe et de procéder à une *copie en profondeur* des attributs à modifier afin de ne pas les muter dans le graphe originel. Il est également désirable d'implémenter ces méthodes le plus en amont possible de la structure d'héritage afin de ne pas avoir à implémenter ces méthodes pour chaque sous-classe. La transposition est une opération spécifique aux graphes dirigés, on l'implémentera donc dans la classe `DirectedGraph`. L'extraction d'un sous-graphe peut, quant à elle, être réalisée sur tout types de graphes et sera donc implémentée dans `Graph`. Ces méthodes sont toutes deux de complexité $\boldsymbol{O}(|V|+|E|)$.

```
GET-TRANSPOSED(G)
    ▷Input: directed graph: G = (V, E)
    ▶Output: directed graph: G^T = (V, E^T)
    for each node u ∈ V(G)
        do V(G^T)[u(id)] ← u
    for each edge e(u, v) ∈ E
        do append(E^T, edge e'(v, u))
    return G^T
```

```
SUBGRAPH(G, U)
    ▷Input: graph: G = (V, E)
    ▷Input: nodes subset U ⊂ V
    ▶Output: graph: G[U] = (V', E')
    for each node u ∈ U
        do V'[u(id)] ← u
    for each edge e(u, v) where u ∈ U, v ∈ U
        do append(E', edge e'(v, u))
    return G[U]
```

Au niveau du code, il faut également s'assurer que le graphe obtenu soit de la même classe que le graphe depuis lequel on appelle ces méthodes pour toute sous-classe. De même, il faut que les nodes soient de la même classe que les nodes données et qu'elles contiennent tous leurs attributs. Pour celà, je fait appel à des champs et méthodes universels des classes `Python` qui existeront quelle que soit la sous-classe donnée.

3.3.2 Obtention des termes associés directs: `getDirectGOTerms(gene)` et `getDirectGeneProducts(term)`:

Dans le cas de `getDirectGOTerms(gene)`, du fait de l'orientation choisie pour les arrêtes des relations *product of*, il suffit d'obtenir la liste des nodes successeurs. Cela peut se faire avec une complexité maximale $\boldsymbol{O}(|V|)$ grâce à la méthode `neighbors(node)`. Les nodes `GeneProduct` n'étant lié au graphe que par des arrêtes *product of*, on sait que les successeurs de ces nodes ne peuvent être que des nodes `GOTerm`.

```
GET-DIRECT-GOTERMS(u)
    ▷Input: GeneProduct: u
    ▶Output: sorted terms subset T' ⊂ T
    for each node v ∈ NEIGHBORS(G, u)
        do T'[v(id)] ← v
    return T'
```

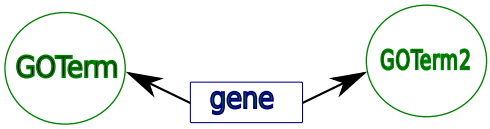

Figure 5. Visualisation de `getDirectGOTerms(GeneProduct)`

En revanche, pour `getDirectGeneProducts(term)`, l'orietation des arrêtes *product of* nous oblige à parcourrir les arrêtes dans leur orientation inverse. On pourrait donc travailler plutôt sur le graphe transposé de la GO tel que nous l'avons défini. Cependant, il est peut être excessif de transposer tout le graphe avec une complexité $\boldsymbol{O}(|V|+|E|)$ lorsque les nodes d'intérêt sont toutes à une arrête d'écart. On fera plutôt le choix d'utiliser `predecessors(node)` avec une complexité $\boldsymbol{O}(|V|)$. Les nodes `GOTerm` peuvent être en relation avec d'autres nodes du même type. Il faudra donc exclure ces nodes afin de ne récupérer que les prédecesseurs de type `GeneProduct`.

```
GET-DIRECT-GENEPRODUCTS(u)
    ▷Input: GOTerm: u
    ▶Output: sorted annotations subset P' ⊂ P
    for each node v ∈ PREDECESSORS(G, u)
        # else, ∉ PREDECESSORS(G, u)
        do if v is a GeneProduct # else, not a GeneProduct
            do P'[v(id)] ← v
    return P'
```

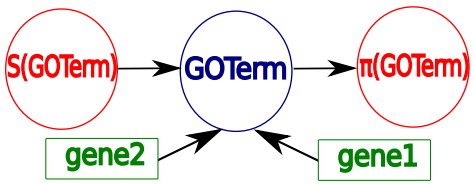

Figure 6. Visualisation de `getDirectGeneProducts(GOTerm)`

3.3.3 Obtention de tous les termes associés: `getAllGOTerms(gene)` et `getAllGeneProducts(term)`:

Ces méthodes-ci nécéscitent un véritable algorithme de parcours de graphe. On veut conserver les nodes du type cherché tout en parcourrant le chemin des nodes de type `GOTerm` dans le sens des successeurs ou des prédécesseurs selon l'implémentation de la GeneOntology. Comme on ne s'intéresse qu'au chemin suivant la source donnée, on peut utiliser un algorithme de parcours en largeur. Les méthodes `getAllGOTerms(gene)` et `getAllGeneProducts(term)` font donc appel à `bfs(source)`. Afin d'obtenir le sous-enssemble de nodes désiré, je modifie `bfs()` afin qu'elle renvoie l'ensemble des nodes traversée. Dans le cas de `getAllGeneProducts(term)`, le parcours des arrêtes des relations *product of* ainsi que des `GOTerm` descendants requiert, avec cette implémentation de la GO, de parcourrir les arrêtes dans leur orientation inverse. Comme on veut ici utiliser une méthode de parcours de complexité $\boldsymbol{O}(|V| + |E|)$ déjà implémentée, on parcourra le graphe transposé de la GO. La complexité maximale de chaque fonction est $\boldsymbol{O}(|V| + |E|)$. Pour `getAllGOTerms(gene)` seule le gène source est un `GeneProduct` qu'il suffit d'éliminer de la liste.

```
GET-ALL-GOTERMS(G, s)
    ▷Input:  GO graph (DAG): G = (V, E)
    ▷Input: GOTerm: s
    ▶Output: terms subset: T' ⊂ T
    T' ← bfs(G, s)
    delete T'[s(id)]
    return T'
```

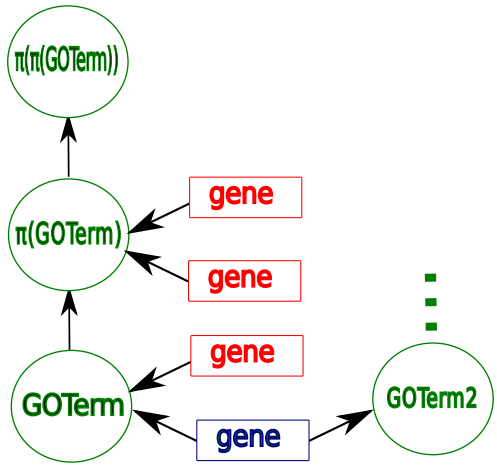

Figure 7. Visualisation de `getAllGOTerms(GeneProduct)`

```
GET-ALL-GENEPRODUCTS(G, s)
    ▷Input:  GO graph (DAG): G = (V, E)
    ▷Input: GeneProduct: s
    ▶Output: terms subset: P' ⊂ P
    G^T ← GET-TRANSPOSED(G)
    for each node u ∈ bfs(G^T, s)
       do if u is a GeneProduct
       # else, not a GeneProduct
          then P'[v(id)] ← v
    return P'
```

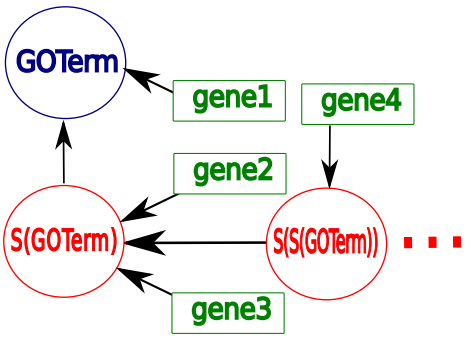

Figure 8. Visualisation de `getDirectGOTerms(GOTerm)`

3.3.4 Calcul de la profondeur des ontologies:

Calculer le plus long chemin d'un graphe G équivaut à calculer le plus court chemin d'un graphe -G (dont le poids de chaque arrête est le poids de l'arrête correspondante de G multipliée par -1). On a déjà implémenté l'algorithme de floyd-warshall qui permet de faire cela. Néanmoins, cet algorithme étant de complexité $\boldsymbol{O}(|V|^3)$, il serait inimaginable de l'utiliser sur des graphes de telle envergure. Le problème du plus long chemin est NP-difficile pour les graphes généraux.

On sait cependant que les ontologies sont des graphes dirigés acycliques. Puisque de tels graphes possèdent une structure optimale, le problème est résoluble en un temps $|V| + |E| \cdot$ linéaire en utilisant une méthode de tri topologique. On utilisera donc la méthode `topologicalSort()` qui fera appel au parcours en profondeur `dfs()`.

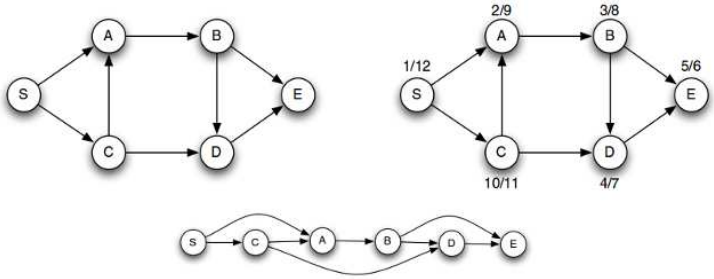

Figure 9. Linéarisation d'un DAG par tri topologique.Toutes les arrêtes sont orientées dans le même sens par rapport à la séquence de nodes.

Le calcul du plus long chemin se fait en parcourrant les termes dans l'ordre topologique afin de calculer la plus longue distance de chacun à la racine. Chaque distance est calculée à partir de la plus longue distance des prédécesseurs. Du fait de la linéarisation du graphe, tout les prédécesseurs d'un terme sont situés à des positions antérieures et leurs distances ont donc déjà étés calculées. Le calcul des plus longues distances est de complexité $\boldsymbol{O}(|P|)$, $\boldsymbol{O}(|F)$ ou $\boldsymbol{O}(|C|)$ selon l'ontologie parcourrue. Chaque ontologie est extraite en utilisant `subgraph(nodes)`. On parcourra le graphe transposé de ce sous-graphe afin de pouvoir parcourrir les arrêtes dans leur orientation.

```
ONTOLOGY-DEPTH(G, ontology):
   ▷Input: GO graph (DAG): G = (V, E)
   ▷Input: namespace: ontology ∈ {"biological_process", "molecular_function", "cellular_component"}
   ▶Output: longest path value: depth
   if ontology = "biological_process"
      then G[U] ←SUBGRAPH( G, BIOLOGICAL-PROCESSES(G) )
   else if ontology = "molecular_function"
      then G[U] ←SUBGRAPH( G, MOLECULAR-FUNCTIONS(G) )
   else if ontology = "cellular_component"
      then G[U] ←SUBGRAPH( G, CELLULAR-COMPONENTS(G) )
   G[U]^T ← GET-TRANSPOSED(G[U])
   topo ← TOPOLOGICAL-SORT(G[U]^T)
   for each term u in topo
      do dist[u] ← -∞
   dist[topo[0]] ← 0 #Ontology root
   for each term u in topo
      do if dist[u] ≠ -∞
         then for each term v ∈ NEIGHBORS(u)
            do if dist[v] < dist[u]+1
               then dist[v] = dist[u]+1
   return maxarg(dist[u_i])
```

4 Bilan technique et perspectives:

Classe Graph:

Méthode	Fonction	Complexité
`parse_pbar(currentLine, numLines)`	Simple barre de progression pour visualiser les parseurs.	-
`subgraph(nodes)`	Créée un sous-graphe à partir d'un enssemble de nodes.	$\boldsymbol{O}(\|V\|+\|E\|)$

Classe DirectedGraph:

Avec: V = Enssemble des nodes dans le Graphe.
E = Enssemble des arrêtes dans le Graphe.

Méthode	Fonction	Complexité
`predecessors(node)`	Renvoie une liste triée des prédécesseurs de la node.	$\boldsymbol{O}(\|V\|+\|E\|)$
`getTransposed()`	Créée le graphe transposé du graphe.	$\boldsymbol{O}(\|V\|+\|E\|)$

Classe GeneOntology:

Avec:
T = Enssemble des nodes `GOTerm` dans le graphe.
P = Enssemble des nodes `GeneProduct` dans le graphe.
V = T + P = Enssemble des nodes dans le graphe.
E = Enssemble des arrêtes dans le graphe.
$\boldsymbol{t}$ = Enssemble des identifiants `GOTerm` des tags `id`, `is_a` ou `part_of` dans le fichier.
$\boldsymbol{l}$ = Enssemble des lignes dans la table d'annotations.

Méthode	Fonction	Complexité
`loadOBO(filename)`	Charge un fichier d'ontologie .obo	$\boldsymbol{O}\Big(\int_0^{\lvert t\rvert}(2\lvert V\rvert\, d\lvert V\rvert)\Big)$
`loadAnnotations(filemame)`	Charge un fichier d'annotations pour l'ontologie.	$\boldsymbol{O}\Big(\int_0^{\lvert l\rvert}(\lvert T\rvert\cdot 2\lvert V\rvert\, d\lvert V\rvert)\Big)$
`isGO()`	Confirme que ce graphe est une ontologie.	$\boldsymbol{O}(0)$
`isComplete()`	Vérifie que l'ontologie est complète	$\boldsymbol{O}(\lvert V\rvert+\lvert E\rvert)$
`GOTerms()`	Extrait l'enssemble des `GOTerm` du graphe.	$\boldsymbol{O}(\lvert V\rvert)$
`GeneProducts()`	Extrait l'enssemble des `GeneProduct` du graphe.	$\boldsymbol{O}(\lvert V\rvert)$
`biologicalProcesses()`	Extrait l'enssemble des `GOTerm` de l'ontologie *biological process* du graphe.	$\boldsymbol{O}(\lvert V\rvert)$
`molecularFunctions()`	Extrait l'enssemble des `GOTerm` de l'ontologie *molecular function* du graphe.	$\boldsymbol{O}(\lvert V\rvert)$
`cellularComponents()`	Extrait l'enssemble des `GOTerm` de l'ontologie *cellular component* du graphe.	$\boldsymbol{O}(\lvert V\rvert)$
`getDirectGOTerms(gene)`	Renvoie les `GOTerm` dirrectement associés à un `GeneProduct`.	$\boldsymbol{O}(\lvert V\rvert)$
`getDirectGeneProducts(term)`	Renvoie les `GeneProduct` dirrectement associés à un `GOTerm`.	$\boldsymbol{O}(\lvert V\rvert)$
`getAllGOTerms(gene)`	Renvoie les `GOTerm` associés à un `GeneProduct` et leurs `GOTerm` ancêtres.	$\boldsymbol{O}(\lvert V\rvert+\lvert E\rvert)$
`getAllGeneProducts(term)`	Renvoie les `GeneProduct` associés à un `GOTerm` et ses descendants.	$\boldsymbol{O}(\lvert V\rvert+\lvert E\rvert)$
`ontologyDepth(ontology)`	Calcule la longeur du plus long chemin du graphe, i.e la profondeur.	$\boldsymbol{O}(\lvert P\rvert+\lvert V\rvert+\lvert E\rvert)$ ou $\boldsymbol{O}(\lvert F\rvert+\lvert V\rvert+\lvert E\rvert)$ ou $\boldsymbol{O}(\lvert C\rvert+\lvert V\rvert+\lvert E\rvert)$

Lors du chargement des anotations j'ai préféré m'assurer que des termes obsolètes ou absents ne soient pas artificièlement ajoutés dans l'ontologie dans le cas de traîtement d'annotations qui ne seraient pas à jour par rapport à l'ontologie utilisée. Cependant celà se fait au coût d'un plus long temps de traîtement des données du graphes; ce qui peut s'avérer problématique pour charger de très grandes collections d'annotations. L'utilisation d'une méthode moins scrupuleuse peut se faire mais il faut alors s'assurer de la validité et de la compatibilité des annotations avec l'ontologie utilisée au préalable.

Il pourrait être intéressant d'ajouter la possibiliter de lire/écrire les graphes dans un format spécifique ou non à la bibliothèque afin de pouvoir enregistrer des modifications faites au graphe au travers des méthodes de la bibliothèque. Il serait également intéressant d'implémenter des exports vers les formats lus (`.sif`, `.tab`) et l'import/export vers d'autres formats courrement utilisés pour la représentation et la visualisation de graphes comme `.xml`, `.gr`, `.dag`, etc.