Pseudo-codes des algorithmes

Table des matières

1 Introduction

Ce rapport présente les pseudo-codes détaillés des algorithmes utilisés dans le processus de génération de requêtes ontologiques-relationnelles. Chaque algorithme correspond à une étape spécifique du processus de transformation des demandes OntoRelQuery Languages (ORQLs) ou Modèles de Conception de Requêtess (MCRs) en requêtes Langage de requêtes structurées (SQL).

2 Organisation des algorithmes

Le tableau 1 présente la synthèse des algorithmes et leur organisation par famille fonctionnelle.

TABLE 1: Synthèse des algorithmes et pseudo-codes

Algorithme	**Famille**	**Pseudo-code (section)**
Algorithmes d'analyse d'une demande		
ASTBuilder	Analyse syntaxique	3.1
EntityValidator	Validation ontologique	3.2
TaskASTExtract	Décomposition	3.3
Algorithmes de traitement de tâches		
TaskProcessor	Coordination	4.1
BuildOntoGraphQuery	Construction	4.2
CombineOntoGraphQuery	Composition	4.3
ExploreOntoGraph	Exploration topologique	4.4
ExploreOntoRelGraph	Exploration relationnelle	4.5
FindPathsWithFilter	Parcours DFS	4.6
FindAllPaths	Critère ALL	4.7
FindMinPath	Critère MIN	4.8
FindInfPath	Critère INF	4.9
FindSupPath	Critère SUP	4.10
Algorithmes de génération SQL		
BuildSQLFromORGQ	Transformation SQL	5.1

3 Algorithmes d'analyse d'une demande

3.1 ASTBuilder : Construction de l'arbre syntaxique

L'algorithme ASTBuilder transforme une demande textuelle ORQL en un arbre syntaxique abstrait (Arbre Syntaxique Abstrait (AST)) structuré selon la grammaire ORQL. Il assure la tokenisation, la validation syntaxique et la construction hiérarchique de l'arbre.

Spécification.

- **Entrée** : AST $A = (V, E, r, \lambda, \mu)$ produit par ASTBuilder ; catalogue Catalogue sémantique stockant les métadonnées ontologiques (OntoRelCat) accessible
- **Sortie** :
  - si succès : AST $A_v = (V, E, r, \lambda, \mu_v)$ enrichi de métadonnées de validation
  - sinon : liste d'erreurs $E_{val} = \{e_1, e_2, \ldots, e_k\}$ avec contexte détaillé (ex. entité inexistante ou référence incorrecte)
- **Complexité** : $O(n)$ où n est le nombre de nœuds de l'AST.
- **Hypothèses** : H1, H2

Algorithme 1: EntityValidator : Validation des entités ontologiques

Pré-condition AST $A = (V, E, r, \lambda, \mu)$ produit par ASTBuilder, catalogue OntoRelCat
Post-condition AST enrichi A_v avec métadonnées de validation, ou liste d'erreurs E_{val}

$E_{val} \leftarrow \emptyset$ ▷ Liste des erreurs de validation
$\mu_v \leftarrow \mu$ ▷ Copie des métadonnées pour enrichissement
▷ **Étape 1 - Extraction des entités**
entitésÀValider $\leftarrow \emptyset$
pour tout $v \in V$ (parcours depuis r via E) ▷ Parcours de l'AST
  pour tout iri $\in$ ExtraireIRIs($\mu(v)$) ▷ Extraire les IRIs des métadonnées
    entité $\leftarrow \{\}$
    entité.iri $\leftarrow$ iri
    entité.type $\leftarrow$ DéterminerTypeEntité($v, \mu(v)$)
      ▷ CLASS, OBJECT_PROPERTY, DATA_PROPERTY, DATATYPE
      ▷ Basé sur le contexte syntaxique dans l'AST
    entité.nœudSource $\leftarrow v$
    entité.traçabilité $\leftarrow$ ObtenirTraçabilité(v) ▷ Ligne, colonne depuis $\mu(v)$
    entitésÀValider $\leftarrow$ entitésÀValider $\cup$ {entité}
  fin_pour
fin_pour
▷ **Étape 2 - Validation et annotation**
pour tout entité $\in$ entitésÀValider
  $v \leftarrow$ entité.nœudSource
  ▷ Sélection de la relation cible selon le type d'entité
  si entité.type = CLASS **alors**
    relationCible $\leftarrow$ Onto_Class
  sinon si entité.type = OBJECT_PROPERTY **alors**
    relationCible $\leftarrow$ Onto_ObjectProperty
  sinon si entité.type = DATA_PROPERTY **alors**
    relationCible $\leftarrow$ Onto_DataProperty
  sinon si entité.type = DATATYPE **alors**
    relationCible $\leftarrow$ Onto_DataType

sinon
    E_{val}.ajouter(TypeEntitéInconnu(entité.iri, entité.traçabilité))
    continue ▷ Passer à l'entité suivante
fin_si
▷ Recherche de l'entité dans OntoRelCat
résultat ← RechercherEntité($OntoRelCat$, relationCible, entité.iri)
si résultat $= \emptyset$ **alors** ▷ Entité inexistante
    E_{val}.ajouter(EntitéIntrouvable(entité.iri, entité.type, entité.traçabilité))
sinon ▷ Entité trouvée
▷ Vérification de la cohérence d'utilisation
    si ¬VérifierCohérenceUtilisation(entité.type, résultat.type_ontologique) **alors**
        E_{val}.ajouter(UtilisationIncorrecte(entité.iri, entité.type,
                résultat.type_ontologique, entité.traçabilité))
    sinon
▷ Enrichissement des métadonnées de validation
        $\mu_v(v) \leftarrow \mu_v(v) \cup \{$
            statut_validation : VALIDE,
            iri_validé : entité.iri,
            table_id : résultat.table_id,
            label : RechercherLabel($OntoRelCat$, entité.iri),
            definition : RechercherDéfinition($OntoRelCat$, entité.iri)
        $\}$
    fin_si
fin_si
fin_pour
▷ **Vérification finale**
si $E_{val} \neq \emptyset$ **alors**
    retourner E_{val} ▷ Liste d'erreurs avec contexte détaillé
sinon
    retourner $A_v = (V, E, r, \lambda, \mu_v)$ ▷ AST validé et enrichi
fin_si

3.2 EntityValidator : Validation des entités ontologiques

L'algorithme EntityValidator vérifie l'existence et la cohérence des entités ontologiques (classes, propriétés) référencées dans l'AST en consultant le catalogue OntoRelCat. Il enrichit les nœuds avec les métadonnées relationnelles nécessaires.

Spécification.

- **Entrée** : AST $A = (V, E, r, \lambda)$ produit par ASTBuilder ; catalogue OntoRelCat accessible
- **Sortie** :
  - si succès : AST A_v enrichi de métadonnées
  - sinon : liste d'erreurs $E_{val} = \{e_1, e_2, \ldots, e_k\}$ avec contexte détaillé (ex. entité inexistante ou référence incorrecte)
- **Complexité** : $O(n)$ où n est le nombre de nœuds de l'AST.
- **Hypothèses** : H1, H2

Algorithme 2: EntityValidator : Validation des entités ontologiques

Pré-condition AST $A = (V, E, r, \lambda)$ produit par ASTBuilder, catalogue OntoRelCat
Post-condition AST enrichi A_v avec métadonnées de validation, ou liste d'erreurs E_{val}

$E_{val} \leftarrow \emptyset$ ▷ Liste des erreurs de validation
▷ **Étape 1 - Extraction des entités**
entitésÀValider $\leftarrow \emptyset$
ParcoursAST(r_A, E_A) ▷ Parcours depuis la racine via les arêtes
pour tout $v \in V_A$
    pour tout iri $\in$ ExtraireEntitésRéférencées(v)
        entité $\leftarrow \{\}$
        entité.iri $\leftarrow$ iri
        entité.type $\leftarrow$ DéterminerTypeEntité(iri, $\mu(v)$) ▷ CLASS, OBJECT_PROPERTY, DATA_PROPERTY
        entité.nœudSource $\leftarrow v$
        entité.traçabilité $\leftarrow$ ObtenirTraçabilité(v) ▷ Ligne, colonne
        entitésÀValider $\leftarrow$ entitésÀValider $\cup$ {entité}
    fin_pour
fin_pour
▷ **Étape 2 - Validation et annotation**
pour tout entité $\in$ entitésÀValider
    $v \leftarrow$ entité.nœudSource
    relationCible $\leftarrow$ SélectionnerRelation(entité.type)
    si entité.type = CLASS **alors**
        relationCible $\leftarrow$ Onto_Class
    sinon si entité.type = OBJECT_PROPERTY **alors**
        relationCible $\leftarrow$ Onto_ObjectProperty
    sinon si entité.type = DATA_PROPERTY **alors**
        relationCible $\leftarrow$ Onto_DataProperty
    fin_si
    résultat $\leftarrow$ RechercherEntité($OntoRelCat$, relationCible, entité.iri)
    si résultat $= \emptyset$ **alors** ▷ Entité inexistante
        E_{val}.ajouter(EntitéIntrouvable(entité.iri, entité.traçabilité))
    sinon ▷ Entité validée : enrichir métadonnées
        AjouterValidation(v, entité.iri, VALIDE, résultat.table_id,
            ObtenirLabel(résultat), ObtenirDéfinition(résultat))
    fin_si
fin_pour
si $E_{val} \neq \emptyset$ **alors**
    retourner E_{val} ▷ Liste d'erreurs avec contexte détaillé
sinon
    retourner $A_v = (V_A, E_A, r_A, \lambda_A, \mu_v)$ ▷ AST validé et enrichi
fin_si

3.3 TaskASTExtract : Décomposition par tâche

L'algorithme TaskASTExtract décompose l'AST global en sous-arbres correspondant aux quatre types de tâches ORQL : CONSTRUCTION, EXPLORATION, EVALUATION et RENAME.

Spécification.

— **Entrée** : AST $A = (V, E, r, \lambda)$ produit par ASTBuilder
— **Sortie** : liste chaînées de AST annotés par tâche
— **Complexité** : $O(n)$ où n est le nombre de nœuds de l'AST.
— **Hypothèses** : H1, H2

Algorithme 3: TaskASTExtract : Extraction et annotation des tâches

Pré-condition AST $A = (V, E, r, \lambda)$ produit par ASTBuilder
Post-condition Liste chaînée de AST annotés : $\mathcal{L} = [A_1, A_2, \ldots, A_k]$

$\mathcal{L} \leftarrow \emptyset$ ▷ Liste chaînée des AST par tâche
tâchesIdentifiées $\leftarrow \emptyset$ ▷ Ensemble des tâches détectées
▷ **Étape 1 - Identification des tâches**
ParcoursAST(r, E) ▷ Parcours depuis la racine
pour tout $v \in V$
  si $\lambda(v) \in \{$CONSTRUCTION, EXPLORATION, EVALUATION, RENAME$\}$ **alors**
    tâche $\leftarrow \{\}$
    tâche.racine $\leftarrow v$ ▷ Nœud racine du sous-arbre
    tâche.type $\leftarrow \lambda(v)$ ▷ Type de tâche (mot-clé de haut niveau)
    tâche.nœuds $\leftarrow$ ExtraireNœudsTâche(v, E) ▷ Sous-arbre complet
    tâche.sousOpérations $\leftarrow$ IdentifierSousOpérations(v, E)
    tâche.nœudsSignificatifs $\leftarrow$ RepérerEntitésEtOpérateurs(v, E)
    tâchesIdentifiées $\leftarrow$ tâchesIdentifiées $\cup \{$tâche$\}$
  fin_si
fin_pour
▷ **Étape 2 - Annotation sémantique**
pour tout tâche $\in$ tâchesIdentifiées
  $V_{\text{tâche}} \leftarrow$ tâche.nœuds ▷ Nœuds du sous-arbre
  $E_{\text{tâche}} \leftarrow$ ExtraireArêtes$(V_{\text{tâche}}, E)$ ▷ Arêtes du sous-arbre
  $r_{\text{tâche}} \leftarrow$ tâche.racine ▷ Racine du sous-arbre
  $\lambda_{\text{tâche}} \leftarrow \lambda|_{V_{\text{tâche}}}$ ▷ Fonction d'étiquetage restreinte
  pour tout $v \in V_{\text{tâche}}$
    ▷ Type de tâche
    si tâche.type $=$ CONSTRUCTION **alors**
      AnnoterTypeTâche$(v,$ CONSTRUCTION_TREE$)$
    sinon si tâche.type $=$ EXPLORATION **alors**
      AnnoterTypeTâche$(v,$ EXPLORATION_TREE$)$
    sinon si tâche.type $=$ EVALUATION **alors**
      AnnoterTypeTâche$(v,$ EVALUATION_TREE$)$
    sinon si tâche.type $=$ RENAME **alors**
      AnnoterTypeTâche$(v,$ RENAME_TREE$)$
    fin_si
    ▷ Type des IRI (classes, propriétés)
    si v référence une Identificateur de ressource internationalisé (IRI) **alors**
      typeIRI $\leftarrow$ DéterminerTypeIRI(v)
      si typeIRI $=$ CLASS **alors**
        AnnoterTypeEntité$(v,$ CLASS$)$
      sinon si typeIRI $=$ OBJECT_PROPERTY **alors**
        AnnoterTypeEntité$(v,$ OBJECT_PROPERTY$)$
      sinon si typeIRI $=$ DATA_PROPERTY **alors**
        AnnoterTypeEntité$(v,$ DATA_PROPERTY$)$

fin_si

▷ Contexte d'utilisation (rôle de l'entité)

AnnoterContexte(v, DéterminerContexteUtilisation(v, tâche)) ▷ Classe source, classe cible, etc.

fin_si

▷ Dépendances inter-tâches

AnnoterDépendances(v, IdentifierDépendances(v, tâchesIdentifiées)) ▷ Références aux identifiants définis dans d'autres tâches

fin_pour

▷ Créer AST annoté pour cette tâche

$A_{\text{tâche}} \leftarrow (V_{\text{tâche}}, E_{\text{tâche}}, r_{\text{tâche}}, \lambda_{\text{tâche}})$ ▷ AST annoté

$\mathcal{L}$.ajouter($A_{\text{tâche}}$) ▷ Ajouter à la liste chaînée

fin_pour

retourner $\mathcal{L}$ ▷ Liste chaînée d'AST annotés par tâche

4 Algorithmes de traitement des tâches

4.1 TaskProcessor : Délégation de tâches

L'algorithme TaskProcessor coordonne l'exécution séquentielle des tâches en déléguant aux processeurs spécialisés selon le type de tâche détecté.

Spécification.

- **Entrée** : liste des AST des tâches.
- **Sortie** :
  - Si succès : une collection $\{OGQ_1, \ldots, OGQ_k\}$ de Représentation conceptuelle de la requête dans l'ontologie (OntoGraphQuery), une collection $\{ORGQ_1, \ldots, ORGQ_k\}$ de Représentation relationnelle de la requête (OntoRelGraphQuery) et une table des symboles TS associant chaque identifiant déclaré à son graphe ;
  - Sinon : une liste d'erreurs E_{cons} signalant les problèmes rencontrés.
- **Complexité** : $O(n)$ où n est le nombre de tâches dans la liste $\mathcal{L}$.
- **Hypothèses** : H1, H2, H3

Algorithme 4: TaskProcessor : Gestionnaire des tâches ORQL

Pré-condition Liste des AST des tâches $\mathcal{L} = [A_1, A_2, \ldots, A_k]$, Graphe orienté représentant les relations ontologiques (OntoGraph) G_o accessible

Post-condition Collections $\{OGQ_1, \ldots, OGQ_m\}$ de OntoGraphQuery, $\{ORGQ_1, \ldots, ORGQ_n\}$ de OntoRelGraphQuery, table des symboles TS, ou liste d'erreurs E_{cons}

$\{OGQ\} \leftarrow \emptyset$ ▷ Collection des OntoGraphQuery
$\{ORGQ\} \leftarrow \emptyset$ ▷ Collection des OntoRelGraphQuery
$TS \leftarrow \emptyset$ ▷ Table des symboles (identifiant, graphe)
$E_{cons} \leftarrow \emptyset$ ▷ Liste d'erreurs
▷ **Parcours séquentiel selon l'ordre des énoncés de l'AST**
pour tout $A_i \in \mathcal{L}$ dans l'ordre séquentiel
    typeTâche $\leftarrow$ ExtraireTypeTâche(r_{A_i}) ▷ Type depuis métadonnées de la racine
    si typeTâche $=$ CONSTRUCTION_TREE **alors**
        ▷ **Étape 1 - Traitement des tâches de construction**
        opération $\leftarrow$ ExtraireOpération(A_i)
        si opération $\in$ {EXTRACTION_CLASSE, EXTRACTION_CONNEXION} **alors**
            ▷ Délégation : extraction depuis OntoGraph
            $OGQ \leftarrow$ BuildOntoGraphQuery(G_o, A_i)
        sinon si opération $=$ COMPOSITION_ENSEMBLISTE **alors**
            ▷ Délégation : composition de graphes existants
            identifiants $\leftarrow$ ExtraireIdentifiants(A_i)
            opérateurEnsembliste $\leftarrow$ ExtraireOpérateur(A_i)
            $OGQ \leftarrow$ CombineOntoGraphQuery(identifiants, opérateurEnsembliste, TS)
        fin_si
        si $OGQ \neq$ null **alors**
            identifiant $\leftarrow$ ExtraireIdentifiant(A_i)
            TS[identifiant] $\leftarrow OGQ$ ▷ Enregistrer dans table des symboles
            $\{OGQ\} \leftarrow \{OGQ\} \cup \{OGQ\}$
        sinon
            E_{cons}.ajouter(ErreurConstruction(A_i))
        fin_si

sinon si typeTâche = EXPLORATION_TREE **alors**
▷ **Étape 2 - Traitement des tâches d'exploration**
cible ← ExtraireCibleExploration(A_i)
cheminSpec ← ExtraireSpécificationChemin(A_i)
typeDistance ← ExtraireTypeDistance(A_i)
critères ← ExtraireCritèresExploration(A_i)
▷ Résolution des dépendances
si cible est un identifiant **alors**
si cible $\in TS$ **alors**
$OGQ_{\text{source}} \leftarrow TS$[cible]
sinon
E_{cons}.ajouter(IdentifiantIntrouvable(cible, A_i))
continue
fin_si
fin_si
▷ Délégation selon l'opération
typeExploration ← DéterminerTypeExploration(A_i, μ)
si typeExploration = EXPLORATION_RÉFÉRENCE **alors**
$OGQ \leftarrow$ ExploreOntoGraph($OntoGraph$, cible, cheminSpec, critères) ▷ Traitement sur OntoGraph
sinon si typeExploration = EXPLORATION_SOUS_GRAPHE **alors**
$OGQ \leftarrow$ ExploreOntoGraph(OGQ_{source}, cible, cheminSpec, critères) ▷ Traitement sur OGQ construit
sinon si typeExploration = EXPLORATION_RELATIONNELLE **alors**
$ORGQ \leftarrow$ ExploreOntoRelGraph(OGQ_{source},OntoRelGraph, cheminSpec, critères) ▷ Traitement sur OntoRelGraph
$\{ORGQ\} \leftarrow \{ORGQ\} \cup \{ORGQ\}$
fin_si
si $OGQ \neq$ null **alors**
identifiant ← ExtraireIdentifiant(A_i, μ)
TS[identifiant] $\leftarrow OGQ$
$\{OGQ\} \leftarrow \{OGQ\} \cup \{OGQ\}$
fin_si
sinon si typeTâche = EVALUATION_TREE **alors**
▷ **Étape 3 - Traitement des tâches d'évaluation**
typeExpression ← DéterminerTypeExpression(A_i)
si typeExpression = SQL_DIRECT **alors**
$Q_{SQL} \leftarrow$ ExtraireRequêteSQL(A_i) ▷ Expression SQL directe respectant le standard
sinon si typeExpression = RÉFÉRENCE_IDENTIFIANT **alors**
identifiant ← ExtraireIdentifiant(A_i)
si identifiant $\in TS$ ET TS[identifiant] est un $ORGQ$ **alors**
$ORGQ_{\text{ref}} \leftarrow TS$[identifiant]
$Q_{SQL} \leftarrow$ BuildSQLFromORGQ($ORGQ_{\text{ref}}$) ▷ Génération SQL depuis ORGQ
sinon
E_{cons}.ajouter(IdentifiantORGQIntrouvable(identifiant, A_i))
fin_si
fin_si
sinon si typeTâche = RENAME_TREE **alors**
▷ **Étape 4 - Traitement des tâches de renommage**

▷ Extraction des spécifications
typeSource ← ExtraireTypeSource(A_i)
valeurSource ← ExtraireValeurSource(A_i)
cibleRenommage ← ExtraireCibleRenommage(A_i)
codeLinguistique ← ExtraireCodeLinguistique(A_i)
▷ Résolution et validation de la source
sourceValide ← false
si typeSource = CLASSE_IRI **alors**
    sourceValide ← RechercherDansNœuds($\bigcup_i V(OGQ_i)$, valeurSource)
sinon si typeSource = PROPRIÉTÉ_OBJET **alors**
    sourceValide ← RechercherDansArêtes($\bigcup_i E(OGQ_i) \cup E(ORGQ_i)$, valeurSource)
sinon si typeSource = PROPRIÉTÉ_DONNÉES **alors**
    sourceValide ← RechercherDansAttributs($\{ORGQ\}$, valeurSource)
sinon si typeSource = IDENTIFIANT **alors**
    sourceValide ← (valeurSource $\in TS$)
fin_si
▷ Validation
si ¬sourceValide **alors**
    E_{cons}.ajouter(SourceIntrouvable(valeurSource, A_i))
sinon si codeLinguistique $\neq$ null $\wedge$ ¬CodeLinguistiqueValide(codeLinguistique) **alors**
    E_{cons}.ajouter(CodeLinguistiqueInvalide(codeLinguistique, A_i))
sinon si DétecterConflitRenommage(valeurSource, cibleRenommage, TS) **alors**
    E_{cons}.ajouter(ConflitRenommage(valeurSource, A_i))
sinon
    EnregistrerRenommage(valeurSource, cibleRenommage, codeLinguistique, TS)
fin_si
fin_si
fin_pour
▷ **Retour des résultats**
si $E_{cons} \neq \emptyset$ **alors**
    retourner E_{cons} ▷ Liste d'erreurs avec traçabilité
sinon
    retourner $\{OGQ\}, \{ORGQ\}, TS$ ▷ Collections de graphes et table des symboles
fin_si

4.2 BuildOntoGraphQuery : Construction de sous-graphes ontologiques

L'algorithme BuildOntoGraphQuery extrait des sous-graphes depuis OntoGraph selon les opérations d'extraction de classe ou de connexion spécifiées.

Spécification.

- **Entrée** :
  - AST d'une tâche de construction $A_c = (V_c, E_c, r_c, \lambda_c, \mu)$ de type CONSTRUCTION_TREE
  - OntoGraph G_o
- **Sortie** :
  - si succès : un OntoGraphQuery OGQ sous-graphe de G_o
  - sinon : erreur e_{cons} avec contexte détaillé
- **Complexité** : $O(k \times \text{deg})$ où k est le nombre de segments de chemin et deg est le degré moyen

des nœuds. Pour l'extraction de classe, la complexité est $O(d)$ où d est le nombre de propriétés de données (généralement petit et constant). Pour l'extraction de connexion, chaque segment nécessite un filtrage des voisins en $O(\deg)$.

— **Hypothèses** : H1, H2, H3

Algorithme 5: BuildOntoGraphQuery : Construction du graphe ontologique de requête

Pré-condition AST $A_c = (V_c, E_c, r_c, \lambda_c, \mu)$ de type CONSTRUCTION_TREE, OntoGraph $G_o = \langle N_o, E_o \rangle$

Post-condition OntoGraphQuery $OGQ = \langle N_{ogq}, E_{ogq} \rangle$ ou erreur e_{cons}

$N_{ogq} \leftarrow \emptyset$ ▷ Ensemble des nœuds du sous-graphe
$E_{ogq} \leftarrow \emptyset$ ▷ Ensemble des arêtes du sous-graphe
▷ **Étape 1 - Identification de l'opération**
opération ← ExtraireTypeOpération(r_c) ▷ BASE CLASS ou BASE CONNECTION
si opération = BASE_CLASS **alors**
    ▷ **Étape 2 - Extraction de classe**
    classeIRI ← ExtraireClasseSource(r_c)
    $n_{\text{classe}} \leftarrow$ RechercherNœud(N_o, classeIRI)
    si n_{classe} = null **alors**
        retourner ErreurClasseIntrouvable(classeIRI)
    fin_si
    $N_{ogq} \leftarrow N_{ogq} \cup \{n_{\text{classe}}\}$
    ▷ Application de la projection de propriétés de données
    modeProjection ← ExtraireModeProjection(r_c) ▷ COMPLÈTE, INCLUSIVE, EXCLUSIVE
    arêtesDP $\leftarrow \{(n_{\text{classe}}, dp, n_d) \in E_o \mid n_d \in D \wedge \text{typeArête}(dp) = \text{'DP'}\}$
    si modeProjection = COMPLÈTE **alors**
        $E_{ogq} \leftarrow E_{ogq} \cup$ arêtesDP ▷ Toutes les propriétés de données
        $N_{ogq} \leftarrow N_{ogq} \cup \{n_d \mid (n_{\text{classe}}, dp, n_d) \in \text{arêtesDP}\}$
    sinon si modeProjection = INCLUSIVE **alors**
        propriétésIncluses ← ExtrairePropriétésIncluses(r_c)
        pour tout $(n_{\text{classe}}, dp, n_d) \in$ arêtesDP
            si étiquette(dp) $\in$ propriétésIncluses **alors**
                $E_{ogq} \leftarrow E_{ogq} \cup \{(n_{\text{classe}}, dp, n_d)\}$
                $N_{ogq} \leftarrow N_{ogq} \cup \{n_d\}$
            fin_si
        fin_pour
    sinon si modeProjection = EXCLUSIVE **alors**
        propriétésExclues ← ExtrairePropriétésExclues(r_c) ▷ ALL BUT
        pour tout $(n_{\text{classe}}, dp, n_d) \in$ arêtesDP
            si étiquette(dp) $\notin$ propriétésExclues **alors**
                $E_{ogq} \leftarrow E_{ogq} \cup \{(n_{\text{classe}}, dp, n_d)\}$
                $N_{ogq} \leftarrow N_{ogq} \cup \{n_d\}$
            fin_si
        fin_pour
    fin_si
sinon si opération = BASE_CONNECTION **alors**
    ▷ **Étape 3 - Extraction de connexion**
    classeSourceIRI ← ExtraireClasseSource(r_c)
    $n_{\text{source}} \leftarrow$ RechercherNœud(N_o, classeSourceIRI)
    si n_{source} = null **alors**

```
        retourner ErreurClasseIntrouvable(classeSourceIRI)
    fin_si
    N_ogq ← N_ogq ∪ {n_source}
    segments ← ExtraireSegmentsChemin(r_c)                    ▷ Liste des segments CONNECT
    nœudCourant ← n_source
    pour tout segment ∈ segments
        typeSegment ← segment.type                            ▷ AXI, ISA, ou ALL
        classeCibleIRI ← segment.classeCible
        propriétésObjet ← segment.propriétésObjet             ▷ Si spécifiées
        limites ← segment.limites                             ▷ Cardinalités (min, max)
                                                              ▷ Filtrage des voisins selon le type de navigation
        voisins ← EdgeTypeFilter(nœudCourant, typeSegment, G_o)
                                                              ▷ Application des contraintes de propriétés d'objet si spécifiées
        si propriétésObjet ≠ ∅ alors
            voisins ← FiltrerParPropriétés(voisins, propriétésObjet)
        fin_si
                                                              ▷ Application des contraintes de classe cible si spécifiée
        si classeCibleIRI ≠ null alors
            n_cible ← RechercherNœud(voisins, classeCibleIRI)
            si n_cible = null alors
                retourner ErreurClasseCibleInaccessible(classeCibleIRI)
            fin_si
            voisins ← {n_cible}
        fin_si
                                                              ▷ Ajout des nœuds et arêtes au sous-graphe
        pour tout n_v ∈ voisins
            N_ogq ← N_ogq ∪ {n_v}
            e ← RécupérerArête(nœudCourant, n_v, E_o)
                                                              ▷ Validation et ajustement des cardinalités
            si limites ≠ null alors
                cardinalité(e) ← limites
            sinon
                cardinalité(e) ← RécupérerCardinalitéOriginale(e, E_o)
            fin_si
            E_ogq ← E_ogq ∪ {e}
        fin_pour
                                                              ▷ Mise à jour pour le segment suivant
        si |voisins| = 1 alors
            nœudCourant ← voisins[0]
        sinon
            break                                             ▷ Arrêt si navigation ambiguë
        fin_si
    fin_pour
fin_si
retourner OGQ = ⟨N_ogq, E_ogq⟩                                ▷ Sous-graphe construit
```

4.2.1 Algorithme EdgeTypeFilter

Spécification.

- **Entrée** : Graphe OG (OntoGraph ou Graphe orienté représentant les relations ontologiques relationnelles (OntoRelGraph)), nœud courant n, type de navigation typeNav $\in \{\text{AXI}, \text{ISA}, \text{ALL}\}$, ensemble de nœuds terminaux
- **Sortie** : Ensemble de nœuds voisins filtrés voisinsFiltrés $\subseteq N$ accessibles depuis n selon le type de navigation
- **Complexité** : $O(\deg)$ où deg est le degré sortant du nœud courant. L'algorithme parcourt tous les voisins directs et teste le type de chaque arête en temps constant.
- **Hypothèses** : H1, H2, H3

Algorithme 6: EdgeTypeFilter : Filtrage par type navigation

Pré-condition Graphe OG, nœud courant n, type de navigation typeNav, nœuds terminaux
Post-condition Ensemble de nœuds voisins voisinsFiltrés

```
voisinsDirects ← obtenirVoisinsDirects(OG, nœudCourant)
voisinsFiltrés ← ∅
pour tout nœudVoisin ∈ voisinsDirects
    arêteConnexion ← obtenirArête(OG, nœudCourant, nœudVoisin)
    étiquetteArête ← extraireÉtiquetteArête(arêteConnexion)
    si typeNavigation = "AXI" alors
        si estPropriétéObjet(arêteConnexion) ∧ possèdeCardinalité(arêteConnexion) alors
            voisinsFiltrés ← voisinsFiltrés ∪ {nœudVoisin}
        sinon si estPropriétéDonnées(arêteConnexion) ∧ nœudVoisin ∈ nœudsTerminaux alors
            voisinsFiltrés ← voisinsFiltrés ∪ {nœudVoisin}
        fin_si
    sinon si typeNavigation = "ISA" alors
        si étiquetteArête = "isa" alors
            voisinsFiltrés ← voisinsFiltrés ∪ {nœudVoisin}
        fin_si
    sinon si typeNavigation = "ALL" alors
        voisinsFiltrés ← voisinsFiltrés ∪ {nœudVoisin}
    sinon
        retourner erreur(Type de navigation non supporté)
    fin_si
fin_pour
retourner voisinsFiltrés
```

4.3 CombineOntoGraphQuery : Composition ensembliste

L'algorithme CombineOntoGraphQuery applique les opérateurs ensemblistes (UNION, INTERSECTION, DIFFERENCE) sur des sous-graphes existants.

Spécification.

- **Entrée** :
  - Liste d'identifiants de graphes $\{\text{id}_1, \text{id}_2\}$ (minimum 2)
  - Opérateur ensembliste op $\in \{\text{UNION}, \text{INTERSECTION}, \text{DIFFERENCE}\}$
  - Table des symboles TS associant identifiants aux OntoGraphQuery

— **Sortie** :
  — si succès : OntoGraphQuery OGQ résultant de la composition
  — sinon : erreur e_{cons} avec contexte détaillé
— **Complexité** : $O(|N_1| + |N_2| + |E_1| + |E_2|)$ où N_i et E_i sont les ensembles de nœuds et d'arêtes des graphes sources. Les opérations ensemblistes (union, intersection, différence) sont linéaires avec des structures de type hash set.
— **Hypothèses** : H1, H2, H3

Algorithme 7: CombineOntoGraphQuery : Composition de graphes ontologiques

Pré-condition Liste d'identifiants $\{\text{id}_1, \text{id}_2\}$, opérateur op $\in \{\text{UNION}, \text{INTERSECTION}, \text{DIFFERENCE}\}$, table des symboles TS

Post-condition OntoGraphQuery $OGQ = \langle N_{ogq}, E_{ogq} \rangle$ ou erreur e_{cons}

$N_{ogq} \leftarrow \emptyset$ ▷ Ensemble des nœuds résultant
$E_{ogq} \leftarrow \emptyset$ ▷ Ensemble des arêtes résultant
▷ **Étape 1 - Résolution des graphes sources**
si $\text{id}_1 \notin TS$ **alors**
    retourner ErreurGrapheIntrouvable(id_1)
fin_si
si $\text{id}_2 \notin TS$ **alors**
    retourner ErreurGrapheIntrouvable(id_2)
fin_si
$OGQ_1 \leftarrow TS[\text{id}_1]$ ▷ Récupération depuis table des symboles
$OGQ_2 \leftarrow TS[\text{id}_2]$
$N_1 \leftarrow N(OGQ_1)$, $E_1 \leftarrow E(OGQ_1)$
$N_2 \leftarrow N(OGQ_2)$, $E_2 \leftarrow E(OGQ_2)$
▷ **Étape 2 - Application de l'opérateur ensembliste**
si op $=$ UNION **alors**
▷ Fusion des nœuds avec élimination de doublons basée sur IRI
    pour tout $n \in (N_1 \cup N_2)$
        $n_{\text{existant}} \leftarrow$ RechercherNœudParIRI(N_{ogq}, IRI(n))
        si $n_{\text{existant}} \neq$ null **alors**
▷ Consolidation des métadonnées pour nœuds identiques
            si $\mu(n)$.cardinalité $\neq$ null **alors**
                si $\mu(n_{\text{existant}})$.cardinalité $=$ null **alors**
                    $\mu(n_{\text{existant}})$.cardinalité $\leftarrow \mu(n)$.cardinalité
                sinon
                    $\mu(n_{\text{existant}})$.cardinalité.min $\leftarrow \max(\mu(n_{\text{existant}})$.cardinalité.min, $\mu(n)$.cardinalité.min)
                    $\mu(n_{\text{existant}})$.cardinalité.max $\leftarrow \min(\mu(n_{\text{existant}})$.cardinalité.max, $\mu(n)$.cardinalité.max)
                fin_si
            fin_si
            $\mu(n_{\text{existant}})$.labels $\leftarrow \mu(n_{\text{existant}})$.labels $\cup\ \mu(n)$.labels ▷ Fusion labels multilingues
            $\mu(n_{\text{existant}})$.propriétésDonnées $\leftarrow$ $\mu(n_{\text{existant}})$.propriétésDonnées $\cup$ $\mu(n)$.propriétésDonnées ▷ Agrégation propriétés
        sinon
            $N_{ogq} \leftarrow N_{ogq} \cup \{n\}$
        fin_si
    fin_pour
▷ Fusion des arêtes avec élimination de doublons basée sur triplet
    pour tout $e = (n_s, \text{label}, n_t) \in (E_1 \cup E_2)$

$e_{\text{existante}} \leftarrow$ RechercherArêteParTriplet$(E_{ogq}, \text{IRI}(n_s), \text{label}, \text{IRI}(n_t))$
si $e_{\text{existante}} \neq$ null **alors**
▷ Consolidation des cardinalités pour arêtes identiques
si cardinalité$(e) \neq$ null **alors**
si cardinalité$(e_{\text{existante}}) =$ null **alors**
cardinalité$(e_{\text{existante}}) \leftarrow$ cardinalité(e)
sinon
cardinalité$(e_{\text{existante}})$.min $\leftarrow \max($cardinalité$(e_{\text{existante}})$.min, cardinalité(e).min$)$
cardinalité$(e_{\text{existante}})$.max $\leftarrow \min($cardinalité$(e_{\text{existante}})$.max, cardinalité(e).max$)$
fin_si
fin_si
sinon
$E_{ogq} \leftarrow E_{ogq} \cup \{e\}$
fin_si
fin_pour
sinon si op $=$ INTERSECTION **alors**
$N_{ogq} \leftarrow N_1 \cap N_2$ ▷ Nœuds communs uniquement
$E_{ogq} \leftarrow E_1 \cap E_2$ ▷ Arêtes communes uniquement
▷ Filtrage des arêtes pour cohérence structurelle
pour tout $e = (n_s, \text{label}, n_t) \in E_{ogq}$
si $n_s \notin N_{ogq} \vee n_t \notin N_{ogq}$ **alors**
$E_{ogq} \leftarrow E_{ogq} \setminus \{e\}$ ▷ Supprimer arête orpheline
fin_si
fin_pour
sinon si op $=$ DIFFERENCE **alors**
$N_{ogq} \leftarrow N_1 \setminus N_2$ ▷ Nœuds dans G1 mais pas dans G2
$E_{ogq} \leftarrow E_1 \setminus E_2$ ▷ Arêtes dans G1 mais pas dans G2
▷ Filtrage des arêtes pour cohérence structurelle
pour tout $e = (n_s, \text{label}, n_t) \in E_{ogq}$
si $n_s \notin N_{ogq} \vee n_t \notin N_{ogq}$ **alors**
$E_{ogq} \leftarrow E_{ogq} \setminus \{e\}$ ▷ Supprimer arête orpheline
fin_si
fin_pour
fin_si
▷ **Validation de la cohérence structurelle**
arêtesValides $\leftarrow$ true
pour tout $e = (n_s, \text{label}, n_t) \in E_{ogq}$
si $n_s \notin N_{ogq} \vee n_t \notin N_{ogq}$ **alors**
arêtesValides $\leftarrow$ false
break
fin_si
fin_pour
si $\neg$arêtesValides **alors**
retourner ErreurIncohérenceStructurelle$(\text{id}_1, \text{id}_2, \text{op})$
fin_si
retourner $OGQ = \langle N_{ogq}, E_{ogq} \rangle$ ▷ Graphe composé valide

4.4 ExploreOntoGraph : Exploration topologique

L'algorithme ExploreOntoGraph explore OntoGraph ou un OntoGraphQuery selon des critères topologiques (ALL, MIN, INF, SUP) en déléguant aux algorithmes de parcours spécialisés.

Spécification.

- **Entrée** :
  - AST d'une tâche d'exploration $A_e = (V_e, E_e, r_e, \lambda_e, \mu)$ de type EXPLORATION_TREE
  - Graphe source OG : soit OntoGraph G_o (exploration de référence), soit OntoGraphQuery OGQ (exploration de sous-graphe)
- **Sortie** :
  - si succès : OntoGraphQuery $OGQ = \langle N_{ogq}, E_{ogq} \rangle$ sous-graphe de OG contenant les chemins découverts
  - sinon : erreur e_{expl} avec contexte détaillé
- **Complexité** : Variable selon le critère d'exploration. Pour **MIN**, **INF**, **SUP** : $O(|N| + |E|)$ où $|N|$ et $|E|$ sont les nombres de nœuds et d'arêtes du graphe source. Pour **ALL** : $O(|N|^k)$ où k est le nombre de chemins possibles entre source et cible (potentiellement exponentiel).
- **Hypothèses** : H1, H2, H3

Algorithme 8: ExploreOntoGraph

Pré-condition AST $A_e = (V_e, E_e, r_e, \lambda_e, \mu)$ de type EXPLORATION_TREE, graphe source OG (OntoGraph ou OntoGraphQuery)
Post-condition OntoGraphQuery $OGQ = \langle N_{ogq}, E_{ogq} \rangle$ ou erreur e_{expl}

```
nœudSource ← ExtraireNœudSource(A_e)
nœudCible ← ExtraireNœudCible(A_e)
typeNavigation ← ExtraireTypeNavigation(A_e)
critèreExploration ← ExtraireCritèreExploration(A_e)
poids ← ExtraireFonctionPoids(A_e)
seuilDistance ← ExtraireSeuilDistance(A_e)
nœudsTerminaux ← identifier tous les nœuds de type de données présents dans OntoGraph
si nœudSource = nœudCible alors
    retourner extraireSousGraphe(OG, {nœudSource})
fin_si
résultat ← ∅
si critèreExploration = ALL alors
    résultat ← FindAllPaths(OG, nœudSource, nœudCible,
                  typeNavigation, nœudsTerminaux)
sinon si critèreExploration = MIN alors
    résultat ← FindMinPath(OG, nœudSource, nœudCible,
                  typeNavigation, nœudsTerminaux, poids)
sinon si critèreExploration = INF alors
    résultat ← FindInfPath(OG, nœudSource, nœudCible,
                  typeNavigation, nœudsTerminaux, poids, seuilDistance)
sinon si critèreExploration = SUP alors
    résultat ← FindSupPath(OG, nœudSource, nœudCible,
                  typeNavigation, nœudsTerminaux, poids, seuilDistance)
sinon
    retourner erreur(Critère d'exploration non supporté dans ORQL)
fin_si
```

23: **si** nœudsDeRésultat(résultat) $= \emptyset$ **alors**
24: résultat $\leftarrow$ extraireSousGraphe(OG, {nœudSource})
25: **fin_si**
26: **retourner** résultat

4.4.1 Algorithme ObtenirPoids

Spécification.

- **Entrée** : Arête e du graphe, configuration de poids utilisateur configPoids (optionnelle)
- **Sortie** : Valeur numérique positive poids $\in \mathbb{R}^+$ représentant le poids de l'arête
- **Complexité** : $O(1)$ avec une structure de recherche indexée (hash map) pour configPoids. La recherche de l'arête et l'accès au poids sont en temps constant amorti.
- **Hypothèses** : H1, H2, H3

Algorithme 9: ObtenirPoids : Récupération du poids d'une arête

Pré-condition arête (arête de connexion), configPoids (configuration des poids utilisateur)
Post-condition poids (valeur numérique du poids)

1: poids $\leftarrow$ 1.0
2: **si** configPoids $\neq$ null **alors**
3: **si** configPoids.contient(arête) **alors**
4: poids $\leftarrow$ configPoids.obtenir(arête)
5: **sinon si** configPoids.défaut $\neq$ null **alors**
6: poids $\leftarrow$ configPoids.défaut
7: **fin_si**
8: **fin_si**
9: **retourner** poids

4.5 ExploreOntoRelGraph : Exploration relationnelle

L'algorithme ExploreOntoRelGraph navigue dans OntoRelGraph pour identifier les chemins de jointure entre relations SQL correspondant aux classes ontologiques.

Spécification.

- **Entrée** : AST validé de type EXPLORATION_TREE avec cible REL, OntoGraphQuery OGQ, OntoRelGraph G_{or}
- **Sortie** : OntoRelGraphQuery $ORGQ$ contenant les chemins de jointure entre relations
- **Complexité** : Identique à ExploreOntoGraph. Variable selon le critère : $O((|N_{or}|+|E_{or}|)\log|N_{or}|)$ pour **MIN**, **INF**, **SUP** ; $O(|N_{or}|^k)$ pour **ALL** où k est le nombre de chemins de jointure. Le graphe relationnel G_{or} peut être plus dense que G_o mais les complexités asymptotiques restent identiques.
- **Hypothèses** : H1, H2, H3

Algorithme 10: ExploreOntoRelGraph : Exploration du graphe relationnel

Pré-condition AST $A_e = (V_e, E_e, r_e, \lambda_e, \mu)$ de type EXPLORATION_TREE, OntoGraphQuery OGQ, OntoRelGraph $G_{or} = \langle N_{or}, E_{or} \rangle$
Post-condition OntoRelGraphQuery $ORGQ = \langle N_{orgq}, E_{orgq} \rangle$ ou erreur

1: ▷ **Étape 1 - Extraction et identification**

2: iriSource ← ExtraireClasseSource(r_e)
3: iriCible ← ExtraireClasseCible(r_e)
4: typeNavigation ← ExtraireTypeNavigation(r_e)
5: critèreExploration ← ExtraireCritèreExploration(r_e)
6: poids ← ExtraireFonctionPoids(r_e)
7: seuilDistance ← ExtraireSeuilDistance(r_e)
8: n_{rel}^{source} ← RechercherNœud(N_{or}, iriSource) ▷ Relation source
9: n_{rel}^{cible} ← RechercherNœud(N_{or}, iriCible) ▷ Relation cibe
10: **si** $n_{rel}^{source} = \text{null} \vee n_{rel}^{cible} = \text{null}$ **alors**
11: **retourner** ErreurNœudRelationnelIntrouvable(iriSource, iriCible)
12: **fin_si**
13: nœudsTerminaux ← $\{n \in N_{or} \mid n \in R_D\}$ ▷ Tables de types de données
14: **si** $n_{rel}^{source} = n_{rel}^{cible}$ **alors**
15: **retourner** extraireSousGraphe(G_{or}, $\{n_{rel}^{source}\}$)
16: **fin_si**
17: ▷ **Étape 2 - Délégation selon critère (sur graphe relationnel)**
18: $ORGQ \leftarrow \emptyset$
19: **si** critèreExploration = ALL **alors**
20: $ORGQ$ ← FindAllPaths(G_{or}, n_{rel}^{source}, n_{rel}^{cible}, nœudsTerminaux, typeNavigation)
21: **sinon si** critèreExploration = MIN **alors**
22: $ORGQ$ ← FindMinPath(G_{or}, n_{rel}^{source}, n_{rel}^{cible}, typeNavigation, nœudsTerminaux, poids)
23: **sinon si** critèreExploration = INF **alors**
24: $ORGQ$ ← FindInfPath(G_{or}, n_{rel}^{source}, n_{rel}^{cible}, typeNavigation, nœudsTerminaux, poids, seuilDistance)
25: **sinon si** critèreExploration = SUP **alors**
26: $ORGQ$ ← FindSupPath(G_{or}, n_{rel}^{source}, n_{rel}^{cible}, typeNavigation, nœudsTerminaux, poids, seuilDistance)
27: **sinon**
28: **retourner** ErreurCritèreInvalide(critèreExploration)
29: **fin_si**
30: ▷ **Étape 3 - Vérification et retour**
31: **si** nœudsDeRésultat($ORGQ$) = $\emptyset$ **alors**
32: $ORGQ$ ← extraireSousGraphe(G_{or}, $\{n_{rel}^{source}\}$)
33: **fin_si**
34: **retourner** $ORGQ$ ▷ Chemins de jointure avec FK

4.6 FindPathsWithFilter : Parcours en profondeur avec filtrage

L'algorithme FindPathsWithFilter implémente un DFS avec backtracking servant de base aux autres algorithmes de recherche de chemins.

Spécification.

- **Entrée** : Graphe OG, nœud source, nœud cible, chemin courant, chemins découverts, nœuds terminaux, type de navigation
- **Sortie** : OGQ
- **Complexité** : $O(|N|^k)$ où k est le nombre de chemins simples entre le nœud source et le nœud cible. L'algorithme explore tous les chemins possibles via un parcours en profondeur. Dans le pire cas (graphe complet), k peut être exponentiel, mais reste praticable pour des graphes peu denses.
- **Hypothèses** : H1, H2, H3

Algorithme 11: FindPathsWithFilter : parcours en profondeur

Pré-condition Graphe OG, nœud source, nœud cible, chemin courant, chemins découverts, nœuds terminaux, type de navigation
Post-condition Modification de cheminsDécouverts (procédure récursive)

nœudActif ← cheminCourant.dernierÉlément() ▷ Nœud en cours d'exploration
si nœudActif = nœudCible **alors**
    copieChemin ← créerCopieProfonde(cheminCourant) ▷ Chemin complet trouvé
    cheminsDécouverts ← cheminsDécouverts ∪ {copieChemin}
sinon
    voisinsCandidats ← EdgeTypeFilter(OG, nœudActif, typeNavigation) ▷ Filtrage sémantique
    voisinsValides ← voisinsCandidats \ (ensembleNœuds(cheminCourant) ∪ nœudsTerminaux) ▷ Éviter cycles et terminaux
    pour tout nœudVoisin ∈ voisinsValides
        cheminCourant.ajouterNœud(nœudVoisin)
        FindPathsWithFilter(OG, nœudSource, nœudCible, cheminCourant, cheminsDécouverts, nœudsTerminaux, typeNavigation)
        cheminCourant.retirerDernierNœud() ▷ Backtracking
    fin_pour
fin_si

4.7 FindAllPaths : Découverte tous les chemins (critère ALL)

L'algorithme FindAllPaths découvre tous les chemins possibles entre deux nœuds en utilisant FindPathsWithFilter.

Spécification.

— **Entrée** : Graphe OG, nœud source, nœud cible, nœuds terminaux, type de navigation
— **Sortie** : OntoGraphQuery OGQ contenant tous les chemins découverts entre source et cible
— **Complexité** : $O(|N|^k)$ où k est le nombre de chemins entre source et cible. La complexité est dominée par l'appel à FindPathsWithFilter qui explore tous les chemins. La collecte des nœuds et la construction du sous-graphe sont en $O(k \times |N| + |E|)$, négligeable comparé à l'exploration.
— **Hypothèses** : H1, H2, H3

Algorithme 12: FindAllPaths : chercher tous les chemins (ALL)

Pré-condition Graphe OG, nœud source, nœud cible, nœuds terminaux, type de navigation
Post-condition OntoGraphQuery OGQ contenant tous les chemins découverts entre source et cible

cheminsDécouverts ← ∅ ▷ Initialiser l'ensemble des chemins trouvés
▷ Recherche de tous les chemins via parcours en profondeur
FindPathsWithFilter(OG, nœudSource, nœudCible, [nœudSource], cheminsDécouverts, nœudsTerminaux, typeNavigation)
nœudsCollectés ← ∅ ▷ Ensemble des nœuds à inclure dans le résultat
si cheminsDécouverts = ∅ **alors**
    nœudsCollectés ← {nœudSource} ▷ Aucun chemin trouvé
sinon
    pour tout cheminValide ∈ cheminsDécouverts
        nœudsCollectés ← nœudsCollectés ∪ ensembleNœuds(cheminValide)
    fin_pour

nœudsCollectés ← collecterTerminauxAdjacents(OG, nœudsCollectés, nœudsTerminaux)
fin_si
retourner extraireSousGraphe(OG, nœudsCollectés)

4.8 FindMinPath : Plus court chemin (critère MIN)

L'algorithme FindMinPath implémente l'algorithme de Dijkstra adapté pour identifier le chemin de distance minimale.

Spécification.

- **Entrée** : Graphe OG (OntoGraph ou OntoRelGraph), nœud source, nœud cible, type de navigation, nœuds terminaux, fonction de poids
- **Sortie** :
  - si succès : OntoGraphQuery (ou OntoRelGraphQuery) OGQ contenant le chemin de distance minimale entre source et cible
  - sinon : sous-graphe minimal contenant uniquement le nœud source
- **Complexité** : $O((|N|+|E|)\log|N|)$ où $|N|$ et $|E|$ sont les nombres de nœuds et d'arêtes. L'algorithme implémente Dijkstra avec file de priorité binaire : chaque nœud est extrait en $O(\log|N|)$ et chaque arête est examinée une fois. Avec une file de priorité optimale (Fibonacci heap), la complexité devient $O(|N|\log|N|+|E|)$.
- **Hypothèses** : H1, H2, H3

Algorithme 13: FindMinPath : chercher le chemin minimal (MIN)

Pré-condition OG (OntoGraph), nœudSource, nœudCible, typeNavigation, nœudsTerminaux, poids
Post-condition OGQ_{expl} (OntoGraphQuery contenant le chemin optimal) ou $ORGQ$ (sous-graphe minimal)

▷ Initialisation des structures de données
tableauDistances : Nœud → $\mathbb{R}^+$ ← ∅ ▷ Distance min depuis source
tableauDistances[nœudSource] ← 0
tableauPrédécesseurs : Nœud → Nœud ← ∅ ▷ Pour reconstruction
filePriorité : FilePrioritéMin ← ∅ ▷ Organisée par distance
nœudsExplorés : Ensemble(Nœud) ← ∅ ▷ Nœuds déjà traités
filePriorité.insérer(nœudSource, 0)
tant_que ¬filePriorité.estVide()
    (nœudCourant, distanceCourante) ← filePriorité.extraireMin()
    si nœudCourant ∈ nœudsExplorés **alors**
        continue ▷ Éviter doublons
    fin_si
    nœudsExplorés ← nœudsExplorés ∪ {nœudCourant}
    si nœudCourant = nœudCible **alors**
▷ Cible atteinte : reconstruction du chemin optimal
        cheminOptimal : Liste(Nœud) ← reconstruireChemin(tableauPrédécesseurs, nœudSource, nœudCible)
        nœudsCollectés : Ensemble(Nœud) ← collecterNœudsDepuisChemins(cheminOptimal)
▷ Enrichissement : ajout des nœuds terminaux adjacents
        nœudsCollectés ← ajouterTerminauxAdjacents(OG, nœudsCollectés,

nœudsTerminaux)
retourner extraireSousGraphe(OG, nœudsCollectés) ▷ Retourne OGQ
fin_si
▷ Filtrage 1 : arêtes selon le type de navigation
voisinsCandidats : Ensemble(Nœud) ← EdgeTypeFilter(OG, nœudCourant, typeNavigation)
pour tout voisin ∈ voisinsCandidats
▷ Filtrage 2 : exclusion nœuds terminaux et déjà explorés
si voisin ∉ nœudsTerminaux ∧ voisin ∉ nœudsExplorés **alors**
arête : Arête ← obtenirArête(OG, nœudCourant, voisin)
coût : $\mathbb{R}^+$ ← obtenirPoids(arête, poids)
distanceTemp : $\mathbb{R}^+$ ← tableauDistances[nœudCourant] + coût
si distanceTemp < tableauDistances.obtenirOuDéfaut(voisin, ∞) **alors**
▷ Relâchement de l'arête
tableauDistances[voisin] ← distanceTemp
tableauPrédécesseurs[voisin] ← nœudCourant
filePriorité.insérer(voisin, distanceTemp)
fin_si
fin_si
fin_pour
fin_tant_que
▷ Aucun chemin trouvé : retourner sous-graphe minimal
nœudsMinimaux : Ensemble(Nœud) ← {nœudSource}
nœudsMinimaux ← ajouterTerminauxAdjacents(OG, nœudsMinimaux, nœudsTerminaux)
retourner extraireSousGraphe(OG, nœudsMinimaux) ▷ Retourne ORGQ

4.9 FindInfPath : Chemins sous seuil (critère INF)

L'algorithme FindInfPath utilise un BFS avec élagage pour découvrir les chemins de distance strictement inférieure au seuil.

Spécification.

- **Entrée** : Graphe OG, nœud source, nœud cible, type de navigation, nœuds terminaux, fonction de poids, seuil de distance
- **Sortie** : OntoGraphQuery OGQ contenant tous les chemins de distance $\leq$ seuil entre source et cible
- **Complexité** : $O(k \times |N| \times \log |N|)$ où k est le nombre de chemins de distance $\leq$ seuil. L'élagage par seuil réduit l'espace de recherche. Dans le meilleur cas (seuil petit), la complexité approche $O((|N|+|E|) \log |N|)$ similaire à Dijkstra. Dans le pire cas (seuil très grand), elle peut approcher $O(|N|^p)$.
- **Hypothèses** : H1, H2, H3

Algorithme 14: FindInfPath : exploration bornée par seuil (INF)

Pré-condition OG (OntoGraph), nœudSource, nœudCible, typeNavigation, nœudsTerminaux, poids, seuilDistance
Post-condition OGQ (OntoGraphQuery contenant les chemins de distance $\leq$ seuil)
cheminsValides ← ∅ ▷ Chemins satisfaisant le seuil

```
fileExploration ← créerFilePrioritéMin()                    ▷ Exploration par distance croissante
cheminsDécouverts ← ∅                                       ▷ Éviter les doublons de chemins
                                                            ▷ Initialisation avec nœud source
cheminInitial ← [nœudSource]
fileExploration.insérer((nœudSource, cheminInitial, 0), 0)  ▷ (nœud, chemin, distance), priorité
tant_que ¬fileExploration.estVide()
    (nœudCourant, cheminAccumulé, distanceAccumulée) ← fileExploration.extraireMin()
                                                            ▷ Vérification si le nœud cible est atteint
    si nœudCourant = nœudCible alors
        si distanceAccumulée ≤ seuilDistance alors
            si cheminAccumulé ∉ cheminsDécouverts alors      ▷ Éviter doublons
                cheminsValides ← cheminsValides ∪ {cheminAccumulé}
                cheminsDécouverts ← cheminsDécouverts ∪ {cheminAccumulé}
            fin_si
        fin_si
        continue                                            ▷ Ne pas explorer au-delà de la cible
    fin_si
                                                            ▷ Élagage : abandonner si seuil dépassé
    si distanceAccumulée ≥ seuilDistance alors
        continue                                            ▷ Branche élagué
    fin_si
                                                            ▷ Exploration des voisins
    voisinsAccessibles ← EdgeTypeFilter(OG, nœudCourant, typeNavigation)
    voisinsValides ← voisinsAccessibles \ (ensembleNœuds(cheminAccumulé) ∪ nœudsTerminaux)
    pour tout nœudVoisin ∈ voisinsValides
        arêteConnexion ← obtenirArête(OG, nœudCourant, nœudVoisin)
        poidsArête ← ObtenirPoids(arêteConnexion, poids)
        si poidsArête ≤ 0 alors
            retourner erreur(Poids non positifs non supportés)
        fin_si
        distanceProjetée ← distanceAccumulée + poidsArête
        si distanceProjetée ≤ seuilDistance alors
            cheminÉtendu ← cheminAccumulé + [nœudVoisin]    ▷ Ajouter voisin au chemin
            fileExploration.insérer((nœudVoisin, cheminÉtendu, distanceProjetée), distanceProjetée)
        fin_si
    fin_pour
fin_tant_que
                                                            ▷ Construction du résultat
si cheminsValides = ∅ alors
    retourner extraireSousGraphe(OG, {nœudSource})          ▷ Aucun chemin trouvé
fin_si
nœudsCollectés ← collecterNœudsDepuisChemins(cheminsValides)
nœudsCollectés ← collecterTerminauxAdjacents(OG, nœudsCollectés, nœudsTerminaux)
retourner extraireSousGraphe(OG, nœudsCollectés)
```

4.10 FindSupPath : Chemins au-dessus du seuil (critère SUP)

L'algorithme FindSupPath explore exhaustivement puis filtre les chemins de distance strictement supérieure au seuil.

Spécification.

- **Entrée** : Graphe OG, nœud source, nœud cible, type de navigation, nœuds terminaux, fonction de poids, seuil de distance
- **Sortie** : OntoGraphQuery OGQ contenant tous les chemins de distance $>$ seuil entre source et cible
- **Complexité** : $O(|N|^k)$ où k est le nombre de chemins entre source et cible. Identique à FindAllPaths car l'algorithme doit explorer tous les chemins avant de filtrer ceux dont la distance est $>$ seuil. Aucun élagage anticipé n'est possible contrairement à FindInfPath.
- **Hypothèses** : H1, H2, H3

Algorithme 15: FindSupPath : exploration avec seuil supérieur (SUP)

Pré-condition OG (OntoGraph), nœudSource, nœudCible, typeNavigation, nœudsTerminaux, poids, seuilDistance
Post-condition OGQ (OntoGraphQuery contenant les chemins de distance $>$ seuil)

▷ **Étape 1 - Découverte exhaustive des chemins**
cheminsDécouverts $\leftarrow \emptyset$ ▷ Tous les chemins source $\rightarrow$ cible
FindPathsWithFilter(OG, nœudSource, nœudCible, [nœudSource], cheminsDécouverts, nœudsTerminaux, typeNavigation)
▷ **Étape 2 - Filtrage par distance**
cheminsValides $\leftarrow \emptyset$ ▷ Chemins avec distance $>$ seuil
pour tout chemin $\in$ cheminsDécouverts
    distance $\leftarrow$ calculerDistanceChemin(OG, chemin, poids) ▷ Somme des poids des arêtes
    si distance $>$ seuilDistance **alors** ▷ Critère SUP : strictement supérieur
        cheminsValides $\leftarrow$ cheminsValides $\cup$ {chemin}
    fin_si
fin_pour
▷ **Construction du résultat**
si cheminsValides $= \emptyset$ **alors**
    retourner extraireSousGraphe(OG, {nœudSource}) ▷ Aucun chemin satisfaisant
fin_si
nœudsCollectés $\leftarrow$ collecterNœudsDepuisChemins(cheminsValides)
nœudsCollectés $\leftarrow$ collecterTerminauxAdjacents(OG, nœudsCollectés, nœudsTerminaux)
retourner extraireSousGraphe(OG, nœudsCollectés)

5 Algorithmes de génération SQL

5.1 BuildSQLFromORGQ : Transformation ORGQ vers SQL

L'algorithme BuildSQLFromORGQ transforme un OntoRelGraphQuery en requête SQL exécutable.

Spécification.

- **Entrée** : Modèle ontologique-relationnel (OntoRel) disponible ; collection de relations $R = \{r_1, r_2, \ldots, r_k\}$ extraites des nœuds d'OntoRelGraphQuery
- **Sortie** : Requête SQL Q_{SQL}
- **Complexité** : $O(n^2)$ où n est le nombre de relations extraites du OntoRelGraphQuery. Pour chaque relation à joindre, l'algorithme recherche une jointure valide parmi les relations déjà jointes via FindBestJoin. La somme des recherches donne $\sum_{i=1}^{n} O(i) = O(n^2)$.
- **Hypothèses** : H4

Algorithme 16: BuildJoinQuery : Construction de requête SQL

Pré-condition OntoRel, collection $R = \{r_1, r_2, \ldots, r_k\}$ de relations extraites des nœuds du OntoRelGraphQuery

Post-condition Requête SQL Q_{SQL}

```
1: premièreRelation ← R[0]
2: requête ← SELECT().FROM(premièreRelation)
3: relationsJointes ← {premièreRelation}
4: relationsÀJoindre ← R \ {premièreRelation}
5: tant_que relationsÀJoindre ≠ ∅
6:     joinEffectuée ← faux
7:     pour tout relation ∈ relationsÀJoindre
8:         infoJointure ← FindBestJoin(relation, relationsJointes)     ▷ Algorithme d'optimisation des jointures
9:         si infoJointure ≠ ∅ alors
10:            requête ← requête.JOIN(relation).ON(infoJointure.condition)
11:            relationsJointes ← relationsJointes ∪ {relation}
12:            relationsÀJoindre ← relationsÀJoindre \ {relation}
13:            joinEffectuée ← vrai
14:            break
15:        fin_si
16:    fin_pour
17:    si ¬joinEffectuée alors
18:        relationCroisée ← premier élément de relationsÀJoindre
19:        requête ← requête.CROSS_JOIN(relationCroisée)
20:        relationsJointes ← relationsJointes ∪ {relationCroisée}
21:        relationsÀJoindre ← relationsÀJoindre \ {relationCroisée}
22:    fin_si
23: fin_tant_que
24: champsUniques ← GetUniqueFields(R, OntoRel)
25: requête ← requête.SELECT(champsUniques)
26: retourner requête
```

5.1.1 Algorithme FindBestJoin

Spécification.

- **Entrée** : Relation candidate $r_{candidate}$; ensemble de relations jointes $R_{jointes}$
- **Sortie** : Information de jointure JoinInfo optionnelle contenant les relations et la condition
- **Complexité** : $O(m)$ où m est le nombre de relations déjà jointes. L'algorithme parcourt linéairement les relations jointes et appelle GetForeignKeyJoin en $O(k)$ pour chaque paire, où k est le nombre de clés étrangères (généralement petit et constant).
- **Hypothèses** : H4

Algorithme 17: FindBestJoin : Recherche de jointure optimale

Pré-condition Relation candidate $r_{candidate}$, ensemble $R_{jointes}$ de relations déjà jointes
Post-condition Information de jointure JoinInfo optionnelle

1: **pour tout** $r_{jointe} \in R_{jointes}$
2: infoJointure ← GetForeignKeyJoin$(r_{jointe}, r_{candidate})$
3: **si** infoJointure $\neq \emptyset$ **alors**
4: **retourner** infoJointure
5: **fin_si**
6: **fin_pour**

5.1.2 Algorithme GetForeignKeyJoin

Spécification.

- **Entrée** : Relation source r_{source} ; relation cible r_{cible}
- **Sortie** : Information de jointure JoinInfo optionnelle
- **Complexité** : $O(k)$ où k est le nombre de clés étrangères entre les deux relations. La recherche des clés étrangères dans les métadonnées de l'OntoRel est linéaire en k. La construction de la condition de jointure est en $O(1)$ pour une clé simple.
- **Hypothèses** : H4

Algorithme 18: GetForeignKeyJoin : Construction de jointure par clé étrangère

Pré-condition Relation source r_{source}, relation cible r_{cible}
Post-condition Information de jointure JoinInfo optionnelle

1: clésÉtrangères ← GetForeignKeys(r_{source}, r_{cible}) ▷ Depuis l'artefact OntoRel
2: **si** clésÉtrangères $\neq \emptyset$ **alors**
3: cléÉtrangère ← clésÉtrangères[0]
4: conditionJointure ← BuildJoinCondition(cléÉtrangère)
5: **retourner** JoinInfo$(r_{source}, r_{cible}$, conditionJointure$)$
6: **sinon**
7: **retourner** $\emptyset$
8: **fin_si**