Algorithme RAG Graph Complet

Implémentation détaillée étape par étape

Philippe Helluy

Pipeline Complet du RAG Graph

Notre système hybride combine deux mondes pour une précision maximale :

1. **Branche Vectorielle** : Recherche sémantique floue (Semantic search).
2. **Branche Graphe** : Recherche structurée et relationnelle.

Les deux flux sont **fusionnés**, **rerankés**, et **synthétisés** par le LLM.

Étape 1 : Calcul des Embeddings

Transformation du texte en vecteurs mathématiques.

Modèle : `paraphrase-multilingual-MiniLM-L12-v2`

Chaque chunk de 512 tokens devient un vecteur de dimension 384.

```
# Initialisation
embed_model = HuggingFaceEmbedding(
    model_name="sentence-transformers/..."
)

# Indexation
vector_index = VectorStoreIndex(
    documents, embed_model=embed_model
)
```

Étape 2 : Construction du Graphe

Extraction des **triplets** (Sujet, Prédicat, Objet) via un prompt LLM.

```
TRIPLET_PROMPT = """
Extraire les relations explicites et implicites :
(Alice, est_soeur_de, Bob)
(Bob, travaille_chez, Google)
"""


# Extraction
triplets = extract_triplets(text_chunk, llm=mistral)
# -> [('Jules', 'PORTE', 'chevalière'), ...]
```

Défi : Normaliser les entités (“Jules” vs “le jeune Jules”).

Étape 3 : Normalisation & Fusion

Pour éviter un graphe fragmenté, nous unifions les entités similaires.

Algorithme de Normalisation :

1. Comparaison **exacte** (case-insensitive).
2. Si échec, comparaison **vectorielle** des noms.
3. Si similarité > 0.9, on fusionne.

```
if sim("Jules", "Le jeune Jules") > 0.9:
    # Fusionner en un seul nœud "Jules"
    canonical_name = "Jules"
```

Étape 4 : Le Premier 'Top-K' (Retrieval)

Quand l'utilisateur pose une question, nous lançons deux recherches en parallèle :

1. **Vector Search** : Trouve les 5 chunks les plus similaires (ANN).
2. **Graph Search** : Trouve les entités nommées dans la question ("Jules") et explore le graphe.

```
# Vectoriel
nodes_vec = vector_index.as_retriever(k=5).retrieve(query)

# Graphe (NetworkX ou Neo4j)
nodes_graph = graph_retriever.retrieve(query)
```

Étape 5 : Extraction des Relations du Graphe

Comment transformer le graphe en texte pour le LLM ?

On traverse le graphe à partir des entités trouvées (profondeur 2 à 4).

Exemple : Question “Pourquoi se méfier de Jules ?”

1. Entité trouvée : `Jules`.
2. Voisins : `(Jules)-[PORTE]->(chevalière)`, `(Sophie)-[AIME]->(Jules)`.
3. Voisins des voisins : `(chevalière)-[SYMBOLE_DE]->(La Griffe)`.

Ces chemins sont convertis en texte : `"Jules PORTE chevalière; chevalière EST_SYMBOLE_DE La Griffe"`

Aparté : Similarité Cosinus vs Cross-Entropy

Bi-Encoder (Similarité Cosinus) :

- On calcule le vecteur de la Question (Q) une fois.
- On calcule le vecteur du Document (D) une fois.
- On compare l'angle entre les deux vecteurs ($\cos\theta$).
- **Rapide** mais **superficiel** : chaque texte est compressé séparément.

Cross-Encoder (Cross-Entropy) :

- On donne [Q + D] ensemble au modèle.
- Le modèle voit **toute** l'interaction mot-à-mot entre Q et D (Self-Attention).
- Il sort une probabilité (0-1) : "Est-ce que D répond à Q ?".

Pourquoi le Cross-Encoder est meilleur ?

Exemple subtil : Q: “Qui a mangé la pomme ?” D: “La pomme a mangé la banane.” (Phrase absurde mais mêmes mots clés)

Bi-Encoder (Cosinus) : Vecteurs très proches (mêmes Mots-clés “pomme”, “mangé”). -> **Faux Positif**.

Cross-Encoder : Comprend la grammaire et le sens de l’action grâce à l’attention croisée. -> Score faible.

Coût : Le Cross-Encoder est 100x plus lent (doit recalculer pour chaque paire Q-D), d’où son utilisation uniquement en **Reranking** sur une petite liste.

Étape 6 : Algorithme de Reranking

Nous utilisons donc un **Cross-Encoder** multilingue pour re-noter la pertinence avec précision.

Problème : Le texte structuré du graphe est mal noté par le modèle (qui attend des phrases).

Solution : `ForceGraphRerank`

```
class ForceGraphRerank(BaseNodePostprocessor):
    def postprocess(self, nodes):
        scores = cross_encoder.predict(nodes)

        # Toujours garder le graphe, même si score faible !
        if graph_node not in top_k:
           force_insert(graph_node)

        return top_k_nodes
```

Comparaison Finale : L'apport du Graphe

Question : “Pourquoi se méfier de Jules ?”

- **Réponse Vectorielle** : “Il a une chevalière bizarre et Sophie ne l’a jamais vue.” (Information locale).
- **Réponse Graphe** : “La chevalière est le symbole de **La Griffe**, une société de voleurs. Donc Jules est probablement un voleur.” (Information connectée).

→ Le graphe a permis de faire le saut : `Jules → Chevalière → La Griffe → Voleurs`

Conclusion

Le RAG Graph n'est pas magique, c'est une ingénierie précise :

1. **Extraction** de qualité (LLM).
2. **Normalisation** rigoureuse.
3. **Traversée** intelligente.
4. **Reranking** hybride (Bi-Encoder pour la vitesse, Cross-Encoder pour la précision).

Note Importante : Le LLM reste probabiliste. Même avec le meilleur contexte, la réponse générée peut varier légèrement d'une exécution à l'autre (stochasticité inhérente).