# Documentation des Fonctionnalités des API pour le développement d'un module de gestion de graphes

**Hatem Trigui**

Ce manuscrit a été compilé le 30 mai 2024

**Résumé**

Cette documentation détaille les fonctionnalités avancées de l'API pour la gestion des graphes, spécifiquement les opérations CRUD, la détection des cycles et le calcul du chemin le plus court entre deux nœuds.

**Keywords:** *API, Graphes, Détection de Cycles, Chemin le Plus Court*

## 1. Description du Module des Graphes

### 1.1. Introduction

Ce module de graphes est conçu pour fournir une application backend de gestion de graphes, utilisant les technologies Python et MongoDB. Le module est également containerisé avec Docker pour faciliter le déploiement et la maintenance

#### *1.1.1. Technologies utilisées*

- **FastAPI :** Framework web Python moderne et rapide pour créer des API RESTful.
- **MongoDB :** Base de données NoSQL orientée documents.
- **Docker :** Plateforme de conteneurisation pour déployer et gérer des applications.
- **Uvicorn :** Serveur ASGI rapide pour les applications web Python.
- **Docker Compose :** Outil pour définir et gérer des applications multi-conteneurs Docker.

### 1.2. Spécifications du Graphique

Chaque noeud dans notre graphe possède plusieurs attributs qui le définissent :

- **Titre** : Une chaîne de caractères qui identifie le noeud.
- **Description** : Un texte plus long expliquant ce que le noeud représente ou sa fonction dans le graphe.
- **Condition** : Une expression logique qui peut être utilisée pour filtrer les noeuds lors de requêtes complexes. Exemple : "$variable1 OR $variable2 AND $variable3".
- **Noeuds de Destination** : Un tableau de noeuds vers lesquels il existe un lien direct depuis le noeud actuel.

### 1.3. Fonctionnalités du Backend

Le backend offre une série d'API REST pour manipuler les noeuds du graphe :

- **Créer un Noeud** : Permet d'ajouter un nouveau noeud au graphe.
- **Modifier un Noeud** : Permet de mettre à jour les attributs d'un noeud existant.
- **Supprimer un Noeud** : Permet de retirer un noeud du graphe.
- **Récupérer un Noeud par ID** : Permet de consulter les détails d'un noeud spécifique.
- **Créer un Lien entre Deux Noeuds** : Ajoute une relation directe entre deux noeuds.
- **Supprimer un Lien entre Deux Noeuds** : Supprime une relation existante entre deux noeuds.

## 2. Fonctionnalités Avancées

En plus des opérations CRUD basiques, le module offre des fonctionnalités avancées :

### 2.1. Détection de Cycles

Une API pour identifier les boucles dans le graphe, ce qui est crucial pour la validation de la structure du graphe.

#### *2.1.1. Recherche en Profondeur (DFS)*

La recherche en profondeur, ou *Depth-First Search* (DFS), est un algorithme fondamental pour parcourir ou rechercher des éléments dans les graphes. Cet algorithme explore les branches du graphe aussi loin que possible avant de revenir sur ses pas pour explorer d'autres branches.

#### *2.1.2. Principe de Fonctionnement*

Le DFS commence par un nœud de départ et explore aussi profondément que possible le long de chaque branche avant de revenir en arrière.

#### *2.1.3. Détection de Cycles*

Le DFS peut être utilisé efficacement pour détecter les cycles dans un graphe, ce qui est crucial pour éviter les récursions infinies ou les dépendances circulaires dans certaines applications.

**Algorithme** L'algorithme pour la détection de cycles avec DFS fonctionne comme suit :

1. Initialiser un ensemble *visited* pour suivre les nœuds visités.
2. Utiliser une pile de récursion *rec_stack* pour suivre les nœuds en cours d'exploration.
3. Pour chaque nœud non visité, exécuter la fonction *dfs*.
4. Dans *dfs*, marquer le nœud comme visité et l'ajouter à *rec_stack*.
5. Pour chaque voisin du nœud, si le voisin n'est pas visité, appeler récursivement *dfs* sur ce voisin. Si le voisin est déjà dans *rec_stack*, un cycle est détecté.
6. Retirer le nœud de *rec_stack* après avoir exploré tous ses voisins.

**Pseudo-Code** Le pseudo-code pour la fonction *dfs* est donné par :

```
Fonction dfs(node):
    marquer node comme visité
    ajouter node à rec_stack

    pour chaque voisin de node:
        si voisin n'est pas visité:
            si dfs(voisin):
                retourner vrai
        sinon si voisin est dans rec_stack:
            retourner vrai
```

```
    retirer node de rec_stack
    retourner faux
```

### 2.2. Chemin le Plus Court

Une API pour déterminer le chemin le plus court entre deux noeuds, utilisant l'algorithme BFS pour garantir que le chemin trouvé est le plus direct.

#### *2.2.1. La recherche en largeur (BFS)*

La recherche en largeur, ou *Breadth-First Search* (BFS), est un autre algorithme essentiel pour explorer les graphes. Contrairement à DFS qui plonge aussi profondément que possible dans le graphe, BFS explore tous les voisins d'un nœud avant de passer aux nœuds au niveau suivant. Cet algorithme est particulièrement utile pour trouver le chemin le plus court dans un graphe non pondéré.

#### *2.2.2. Principe de Fonctionnement*

BFS commence à un nœud source et explore tous ses voisins directs, puis il explore leurs voisins et ainsi de suite, jusqu'à ce qu'il trouve le nœud cible ou explore tous les nœuds.

#### *2.2.3. Application à la Trouvaille du Chemin le Plus Court*

En utilisant BFS pour la recherche du chemin le plus court, chaque fois qu'un nœud est exploré pour la première fois, le chemin menant à ce nœud est le plus court possible. Ce principe est crucial pour garantir l'exactitude de l'algorithme dans la recherche du chemin le plus court.

**Algorithme** L'algorithme pour trouver le chemin le plus court avec BFS est le suivant :

1. Initialiser une queue avec le nœud de départ et marquer ce nœud comme visité.
2. Enregistrer le chemin parcouru pour atteindre chaque nœud.
3. Tant que la queue n'est pas vide :
   (a) Défiler un nœud de la queue.
   (b) Si c'est le nœud de destination, retourner le chemin enregistré.
   (c) Sinon, pour chaque voisin non visité, marquer comme visité, enregistrer le chemin et l'ajouter à la queue.

**Pseudo-Code** Le pseudo-code pour la fonction BFS est donné ci-dessous :

```
Fonction bfs(start_id, end_id):
    queue = Queue()
    queue.enqueue((start_id, [start_id]))
    visited = set(start_id)

    tant que queue n'est pas vide:
        current_node, path = queue.dequeue()

        si current_node == end_id:
            retourner path

        pour chaque voisin dans graph[current_node]:
            si voisin non visité:
                visited.add(voisin)
                queue.enqueue((voisin, path + [voisin]))

    lever une exception "Chemin non trouvé"
```

## 3. Contraintes et Tests

Les tests unitaires sont implémentés pour valider le bon fonctionnement des routes et des fonctionnalités de l'application. Les tests utilisent la bibliothèque pytest et httpx pour simuler des requêtes HTTP.

- Tests de création : Vérification de la création de nœuds avec des titres de différentes tailles.
- Tests de lecture : Vérification de la récupération des nœuds.
- Tests de mise à jour : Vérification de la mise à jour des nœuds existants.
- Tests de suppression : Vérification de la suppression des nœuds.
- Tests des algorithmes de graphe : Vérification de la détection des cycles et de la recherche du chemin le plus court.

## 4. Déploiement et Utilisation

Le module est containerisé avec Docker, permettant un déploiement facile et une scalabilité. Les données sont stockées dans MongoDB, une base de données NoSQL adaptée aux structures de données complexes comme les graphes.

## 5. Conclusion

En conclusion, ce document a présenté une série de fonctionnalités avancées pour la gestion de graphes à travers une API REST, en mettant l'accent sur les opérations CRUD, la détection de cycles, et le calcul de chemins les plus courts. Ces fonctionnalités sont essentielles pour les applications nécessitant une manipulation complexe et efficace des données structurées sous forme de graphes.

La mise en œuvre de ces fonctionnalités dans un contexte backend, utilisant Python, MongoDB, et Docker, assure non seulement une gestion efficace des données mais aussi une scalabilité et une maintenance facilitées grâce à la containerisation.