



python networkx library



網路快速搜尋結果：

### NetworkX — NetworkX documentation

NetworkX is a Python package for the creation, manipulation, and study of the structure,...

 NetworkX · <https://networkx.org>

### networkx - PyPI

NetworkX is a Python package for the creation, manipulation, and study of the structure,...

 PyPI · <https://pypi.org>



**NetworkX** 是一款強大的 Python 開放原始碼套件，專門用於建立、操作及研究複雜網路（圖論）的結構、動力學與功能。它是數據科學家處理圖形數據（如社交網路、交通網路或生物網路）的首選工具之一。 ☺

## 核心功能與特色

- **多樣化的圖類型**：支援無向圖 ( Graph )、有向圖 ( DiGraph )、多重圖 ( MultiGraph ) 及有向多重圖。
- **強大的演算法庫**：內建大量經典演算法，包括**最短路徑**、**中心性分析**（如 PageRank ）、**社群偵測**、**聚類分析**等。
- **靈活的節點與邊**：節點可以是任何可雜湊 ( hashable ) 的 Python 物件（如字串、數值甚至是其他圖），且邊可以攜帶任意權重或屬性。
- **良好的生態系統**：可與 **Pandas**、NumPy 和 **Matplotlib** 無縫整合進行數據處理與基礎視覺化。
- **效能擴充**：雖然核心由 Python 撰寫，但支援透過 **NVIDIA RAPIDS cuGraph** 等第三方後端進行 GPU 加速，處理數億條邊的大規模網路。 ☺

提出問題





明確俱仗用任此喻

۲

## 基礎程式碼範例：

# python

```
import networkx as nx
```

## # 1. 建立一個無向圖

```
G = nx.Graph()
```

## # 2. 新增節點與邊

```
G.add_node("A")
```

```
G.add_edge("A", "B", weight=2.5)
```

```
G.add_edges_from([('B', 'C'), ('C', 'A')])
```

# 3. 基礎分析

```
print(f"節點數: {G.number_of_nodes()}")
```

```
print(f"最短路徑 (B 到 A): {nx.shortest_path(G, 'B', 'A')}"")
```

## # 4. 簡單繪圖 (需安裝 *matplotlib*)

```
import matplotlib.pyplot as plt
```

```
nx.draw(G, with_labels=True)
```

```
plt.show()
```

請謹慎使用程式碼。



## 適用場景



~o ~w ~v

any performance better library other this

如果您在處理大規模圖形數據（例如超過 10 萬個節點或百萬條邊）時發現 NetworkX 速度太慢，可以選擇以下幾款效能更強的替代套件。

與純 Python 編寫的 NetworkX 不同，這些套件的核心通常由 **C**、**C++** 或 **Rust** 實作，效能通常快上 **10 到 100 倍**。

## 二、圖論套件 (Graph Libraries)

- **核心語言**：C++ (模板化)
- **優點**：目前公認最快的 Python 圖論套件之一。它支援 **OpenMP 多執行緒並行運算**，在計算 PageRank 或中心性分析時，速度可達 NetworkX 的數百倍。
- **缺點**：安裝較為困難（通常需透過 Conda 或系統套件管理員），且 API 與一般 Python 習慣略有不同。

### 3. NetworkKit (大數據並行運算)

- **核心語言**：C++
- **優點**：專為**大規模並行運算**設計。與 `graph-tool` 類似，它深度優化了多核心處理器的使用。
- **適用場景**：處理含有數百萬甚至數十億條邊的超大型網絡。

### 4. rustworkx (高效率且穩定)

- **核心語言**：Rust
- **優點**：由 IBM 開發（原名 `networkx`），旨在提供與 NetworkX 相似的易用性，但擁有所謂的高安全性與高效能。
- **適用場景**：需要高效能但希望安裝過程像 NetworkX 一樣簡單（直接 `pip install`）。

### 5. cuGraph (GPU 加速)

- **核心語言**：CUDA
- **優點**：如果您有 NVIDIA GPU，cuGraph 可以利用 GPU 的並行能力。原本在 NetworkX 需要執行數天的任務，在 cuGraph 可能只需幾分鐘。
- **適用場景**：擁有直連 CUDA 設備且希望利用 GPU 的數據科學與運算。



the creation, manipulation, and study of the structure, dynam...

graph-tool NetworkX C++

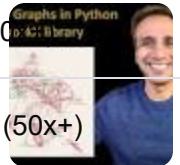
極限運算、多執行緒支援

極高 (50x-100x+)

rustworkx 354 - Knowledge Graphs in Python Using NetworkX library - YouTube 安全 安裝簡易 高 (2025年3月12日)

NetworkKit 2025年3月12日 — 354 - Knowledge Graphs in Python Using NetworkX library - YouTube. This content isn't available. Building and Analyzing.

NetworkKit C++ 專精於超大型並行分析



YouTube

**建議**：如果您的圖形規模還在「萬」級顯示建議先嘗試 **igraph**；如果您正在處理「千萬」或「億」級別的數據，則應直接考慮 **graph-tool** 或 **NetworkKit**。



... 這是最受歡迎的 NetworkX 替代方案。

- **語言**：核心由 C 實作。
- **優勢**：安裝極其簡單 (`pip install igraph`)，速度比 NetworkX 快 10-50 倍 igraph 官方手冊。
- **適合對象**：想要快速提升速度，但不想學習過於複雜的 API。

## 2. graph-tool - Python 界的最速傳說

如果您需要處理數百萬個節點，這是目前 Python 中效能最強大的工具。

- **語言**：核心由 C++ 撰寫，並利用 **OpenMP** 進行多核並行運算。
- **優勢**：效能與原生 C++ 幾乎無異，功能極其強大。
- **缺點**：Windows 支援度較差（通常需透過 WSL 或 Docker），安裝門檻較高。

## 3. rustworkx - 現代高效、安裝無痛

由 IBM 開發，原名 `retnetworkx`，旨在解決大規模圖形運算。

- **語言**：由 **Rust** 實作。
- **優勢**：提供非常接近 NetworkX 的開發體驗，但在計算最短路徑等任務上比 NetworkX 快 20-100 倍。
- **適合對象**：需要穩定、高性能且能用 `pip` 輕鬆安裝的 Python 用戶。

## 4. NetworKit - 專為大數據設計

- **語言**：C++ 基礎，搭配 Python 接口。
- **優勢**：專注於大規模平行演算法，適合在高性能運算(HPC)環境下分析數億條邊。

is rich, with many libraries available. `igraph` (Csardi & Nepusz, 2006), `graph-`



[顯示全部](#)

support 200000 nodes but it can use by python

針對 20 萬個節點的規模，NetworkX 會變得非常緩慢（特別是在計算路徑或中心性時），因為它的純 Python 架構會消耗大量記憶體。

以下是支援 Python 且能輕鬆處理 20 萬個節點的推薦選擇：

## 1. igraph (python-igraph) — 最推薦的選擇

對於 20 萬個節點，`igraph` 是最穩定的方案。它能將運算維持在 **C 語言層級**，效率極高。

- 優勢：記憶體佔用極低。`NetworkX` 可能需要數 GB 的記憶體，`igraph` 僅需其幾分之一。
  - 效能：建立圖形與執行演算法的速度通常比 `NetworkX` 快 **20-50 倍**。
  - 安裝：`pip install igraph` 即可。

## 2. rustworkx — 最現代且快速

如果您的運算邏輯包含大量的圖形遍歷（如最短路徑），rustworkx 表現非常優秀。

- 優勢：由 Rust 驅動，專為高效能 Python 應用設計，API 風格與 NetworkX 較為接近。

## 效能比較參考 (處理 200,000 節點時)

特性	NetworkX	igraph	rustworkx
記憶體效率	差 (純 Python 物件)	優 (C 結構)	優 (Rust 結構)
最短路徑運算	緩慢 (數秒/分)	極快 (毫秒級)	極快 (毫秒級)
安裝難度	極易	易 (pip)	易 (pip)
API 友善度	最高	中等	中等

### 建議做法：

如果您希望開發速度最快且安裝最省事，請直接使用 **igraph**。它在處理 20 萬節點時表現非常流暢。

您目前主要需要執行哪種運算？(例如：尋找最短路徑、社群偵測、還是計算 **PageRank**？) 這會影響最終套件的選擇。



also it support lazy loadin

在圖論庫中，「**Lazy Loading (延遲加載)**」通常指兩件事：一是記憶體映射 (**Memory Mapping**)，即不將整個圖載入記憶體，僅在需要時讀取硬碟；二是生成器

精簡，且支援二進位格式的快速加載，避免了解析文字檔（如 CSV/JSON）的巨大開銷。

- **效能**：處理 20 萬節點就像 NetworkX 處理 200 個節點一樣輕鬆。
- **官網**：[graph-tool.skewed.de](http://graph-tool.skewed.de)

## 2. NetworkKit (支援串流與大型 I/O)

NetworkKit 專為處理無法完全放入記憶體的超大型網路而設計。

- **Lazy 特性**：支援 **Graph Profiling** 與高效的讀取器。它提供了非常快的圖形構建器，能透過 NetworkX 轉換接口 實現過度。
- **優勢**：它在處理 I/O 時速度極快，能有效管理大規模數據流。

## 3. Snap.py (記憶體映射與大型磁碟儲存)

這是由史丹佛大學開發的庫，專門處理大規模數據。

- **Lazy 特性**：它的數據結構是為磁碟與記憶體交換而設計的。它支援透過 Table 物件處理原始數據，這類似於延遲加載的過濾機制。
- **文件**：[Snap.py Documentation](#)

## 4. 搭配 Dask-NetworkX (分散式延遲加載)

如果您已經習慣了 NetworkX 的語法，但數據量太大：

- **機制**：Dask 是一個 Python 平行計算庫，它能將大型圖形切分成多個小塊進行**延遲運算 (Lazy Evaluation)**。
- **適用對象**：數據分佈在多個檔案中，且您希望在運算執行 (`.compute()`) 之前不載入所有數據。

2000000 nodes which one is better

當節點數量達到 **2,000,000 (兩百萬)** 且邊的數量可能達到千萬級別時，NetworkX 已經無法勝任。在 Python 環境下，您需要的是具備 C++ 核心與多執行緒並行運算能力的庫。

以下是針對 **200 萬節點** 規模的最強推薦：

## 1. graph-tool —— 性能天花板

這是處理數百萬級別節點的唯一首選。

- **性能原因**：核心演算法（如 PageRank、最短路徑、社群偵測）完全在 C++ 內層運行，並支援 OpenMP 並行計算。
- **Lazy Loading 支援**：它擁有專有的二進位格式 .gt，加載速度比解析文字檔快上百倍，極度節省記憶體。
- **缺點**：安裝最為繁瑣（建議透過 Conda 安裝）。

## 2. NetworkKit —— 大數據平行分析專家

專為處理「億級」規模而生，對兩百萬節點來說游刃有餘。

- **關鍵特性**：它與 Python 的整合非常緊密，且內建了大量的啟發式與平行演算法，專門處理無法在單核跑完的大圖。
- **記憶體管理**：支援高效的 I/O 串流，適合從大型 CSV 或鄰接矩陣加載。

演算法 (如 Betweenness Centrality ) 跑非常久。

- **NetworkX**：在 200 萬節點下，單純建立圖形可能就會吃掉超過 16GB-32GB 的記憶體，甚至直接崩潰。

## 效能估計 (兩百萬節點)

特性	graph-tool	NetworKit	igraph
加載速度	極快 (二進位)	很快	中等
記憶體效率	最高 (C++ 結構)	高	中等
多核平行	支援 (OpenMP)	支援 (OpenMP)	不支援
適用場景	複雜網路模型、視覺化	大數據、平行運算	通用、中大型網路

### 具體建議：

1. 如果您追求最極致的運算速度，請安裝 **graph-tool**。
2. 如果您需要簡單安裝且平衡的效能，請選擇 **NetworKit**。

請問您的開發環境是 Linux 、 Windows 還是 macOS ? ( 這將決定 graph-tool 的安裝難易度 )



## how to install graph tool

在 **Linux** 環境（特別是 Ubuntu/Debian）下安裝 `graph-tool` 最為順暢，因為它依賴大量的 C++ 函式庫（如 Boost 和 CGAL）。**Windows** 使用者強烈建議透過 **WSL2** 安裝。

以下是針對不同環境的安裝指南：

### 1. 使用 Conda 安裝 (最推薦、跨平台)

這是最簡單的方法，因為 Conda 會自動處理複雜的 C++ 依賴項。請使用 Miniforge 或 Anaconda 的 **conda-forge** 頻道：

#### bash

```
# 建立一個新環境並安裝 graph-tool
conda create -n gt_env -c conda-forge python=3.10 graph-tool
conda activate gt_env
```

請謹慎使用程式碼。



```
# 將以下行加入 /etc/apt/sources.list (以 Ubuntu 22.04 jammy 為例)
# deb [ arch=amd64 ] https://downloads.skewed.de jammy main
```

請謹慎使用程式碼。



2024年8月17日 — If you have an NVIDIA GPU you can accelerate your NetworkX code

## 2. 執行安裝 DS and cuGraph.

### bash

```
sudo apt-get update
sudo apt-get install python3-graph-tool
```

請謹慎使用程式碼。



詳細清單請見 Graph-tool Ubuntu 安裝手冊。

## 3. 在 macOS 安裝

推薦使用 Homebrew :

### bash

```
brew install graph-tool
```

請謹慎使用程式碼。



安裝完成後驗證：

```
python
```

```
import graph_tool.all as gt  
print(gt.__version__)
```

請謹慎使用程式碼。



您目前的作業系統是 **Windows** 還是 **Linux**？如果是 **Windows**，我可以提供 **WSL2** 的快速配置指令。



WSL run python slower?

答案是：幾乎不會。在 WSL2 中執行 Python 的效能與原生 Linux 幾乎相同，且在處理 `graph-tool` 這類高度依賴 C++ 編譯的庫時，**WSL2** 的表現通常優於 **Windows** 原生環境。

以下是針對 Python 效能與 WSL2 的關鍵分析：

## 1. 運算效能 (CPU/RAM)

- 零損耗：WSL2 運行在輕量級 Hyper-V 虛擬化之上，其 Linux 核心直接與硬體溝

- **正確做法**：將數據與專案放在 **WSL** 的原生路徑下（例如 `~/project/` 或 `/home/username/...`）。在原生路徑下，I/O 速度極快，甚至超過 Windows 原生 NTFS 檔案系統。

### 3. 記憶體管理

- **動態分配**：WSL2 可以使用高達 Windows 主機 50% - 80% 的記憶體。對於 200 萬節點的圖，記憶體管理至關重要，Linux 的記憶體回收機制對於 Python 的大型物件處理通常更具效率。

## 如何在 WSL2 中高效執行 Python 與 Graph-tool？

1. **安裝 WSL2**：在 PowerShell 執行 `wsl --install`。
2. **使用 VS Code 整合**：安裝 VS Code Remote - WSL 擴充功能。這樣你可以在 Windows 視窗中寫程式，但程式碼實際在 WSL 內的 Linux 核心執行。
3. **安裝 Conda (Miniforge)**：在 WSL 終端機內安裝 Linux 版的 Miniforge，然後依照前述指令安裝 `graph-tool`。

## 總結

對於 200 萬節點的 `graph-tool` 運算，**WSL2 是 Windows 使用者的最佳路徑**，其效能遠遠超過在 Windows 上嘗試硬解不相容的庫。

您是否已經安裝了 **WSL2**，或者需要我提供在 WSL 內安裝 **Miniforge** 的具體步驟？



在 NetworkX 中，節點可以是任何物件；在 graph-tool 中，節點 (Vertex) 由 整數索引 表示。

- **NetworkX:**

```
python
```

```
import networkx as nx
G = nx.Graph()
G.add_node("A")
```

請謹慎使用程式碼。



- **graph-tool:**

```
python
```

```
from graph_tool.all import Graph
g = Graph(directed=False)
v1 = g.add_vertex() # 返回一個 Vertex 物件，索引為 0
```

請謹慎使用程式碼。



## 2. 處理節點屬性 (Property Maps)

上一頁 下一頁

明陞 品牌力提升指南

- **graph-tool:**

### python

```
# 建立一個屬性對照表  
v_name = g.new_vertex_property("string")  
v_name[v1] = "Alice"  
# 將其註冊為圖的屬性 (選用)  
g.vertex_properties["name"] = v_name
```

請謹慎使用程式碼。

## 3. 新增邊

- **NetworkX:** G.add\_edge("A", "B", weight=1.5)
- **graph-tool:**

### python

```
e = g.add_edge(v1, v2)  
e_weight = g.new_edge_property("double")  
e_weight[e] = 1.5
```

請謹慎使用程式碼。

最短路徑

shortest\_path(G, s, t)

shortest\_path(g, s, t)

社群偵測

community.greedy\_modularity\_communities

minimize\_blockmodel\_leiden  
(推論式模型，更強大)

度數計算

G.degree(n)

v.get\_in\_degree() /  
v.get\_out\_degree()

## 5. 高效加載數據 (Lazy Loading 思路)

處理兩百萬節點時，不要用 `for` 迴圈一行行 `add_edge`。請利用 `add_edge_list` 直接從串列或陣列加載：

**python**

```
# 假設 edges 是一個包含 (u, v) 的 numpy 陣列或列表
g.add_edge_list(edges)
```

請謹慎使用程式碼。



## 6. 快速轉換現有物件

如果你已經有一個 NetworkX 物件且記憶體還夠，可以先進行一次性無痛轉換：

**python**

```
# 需要安裝網路轉換擴展
# 注意：在 200 萬節點下，建議直接重新寫入 graph-tool 以節省記憶體
import graph_tool.all as gt
```

## 遷移小撇步

- **隱式索引**：記住節點永遠是 0 到 N-1。如果你原本的節點是字串（如 Email），你需要建立一個 dict 或使用 g.add\_edge\_list 的映射功能來轉換。
- **視圖 (Views)**：graph-tool 支援 Graph Views，可以在不複製圖的情況下，直接過濾掉特定的邊或節點進行運算，這對效能非常有幫助。

您目前的 NetworkX 程式碼中，最主要的**運算邏輯**是什麼？我可以幫您寫出對應的 **graph-tool 高效版本**。



How to convert below networkx to Graph Too: G: nx.DiGraph =  
nx.DiGraph()...



將這段代碼遷移到 **graph-tool** 的關鍵在於：graph-tool 使用整數作為節點 ID。因此，我們需要一個**映射表 (Mapping)** 來連接你的路徑字串（tf\_path\_str）與圖內部的頂點（Vertex）。

以下是轉換後的優化程式碼：

python

```
from graph_tool.all import Graph, is_DAG  
import logging  
  
logger = logging.getLogger(__name__)
```

```
path_to_v = {}

# 幫助函數：確保節點存在，不存在則建立
def get_or_create_vertex(path_str, tf_obj=None):
    if path_str not in path_to_v:
        v = g.add_vertex()
        path_to_v[path_str] = v
        v_prop_path[v] = path_str
        if tf_obj:
            v_prop_tf[v] = tf_obj
    return v, True
return path_to_v[path_str], False

# --- 開始轉換邏輯 ---

# 處理種子文件
for tf in seed_files:
    tf_path_str = tf.lean_path.path
    # 如果節點已存在，graph-tool 不會報錯，但這裡保留你的邏輯檢查
    if tf_path_str in path_to_v:
        raise AssertionError(f"Node {tf_path_str} already exists")
    get_or_create_vertex(tf_path_str, tf)

traced_files = seed_files.copy()
i = 0

while i < len(traced_files):
    tf = traced_files[i]
    tf_path_str = tf.lean_path.path
    u = path_to_v[tf_path_str] # 獲取起點頂點

    for dep_module, dep_lean_path in tf.get_direct_dependencies(repo):
        try:
            . . .
        except:
            . . .
```

```
e = g.add_edge(u, v)
e_prop_mod[e] = dep_module

except Exception as e:
    logger.warning(f"Failed to load dependency {type(dep_lean_path)}: {e}")
    raise e

i += 1

# 3. 驗證是否為有向無環圖 (DAG)
# graph-tool 的 is_DAG 是由 C++ 實作，針對 200 萬節點速度極快
assert is_DAG(g)

# 如果後續需要將屬性永久存在 g 中
g.vertex_properties["traced_file"] = v_prop_tf
g.vertex_properties["path"] = v_prop_path
g.edge_properties["module"] = e_prop_mod
```

請謹慎使用程式碼。



## 關鍵改動說明：

### 1. 頂點映射 ( path\_to\_v ) :

- 在 NetworkX 中，你可以直接用字串當 Key。在 graph-tool，你需要維護一個 dict 來記住 path\_str 對應到哪個 Vertex 物件。

### 2. 屬性映射 ( PropertyMap ) :

- 我們預先定義了 v\_prop\_tf。當你執行 v\_prop\_tf[v] = tf\_dep 時，這在 C++ 層級是非常高效的連續記憶體操作，而非在 Python 層級頻繁修改字典。

### 3. 效能提升：

這段代碼中使用了 `while` 迴圈動態增加節點。如果 `get_direct_dependencies` 的數量非常龐大，這仍會受限於 Python 的迴圈速度。

需要我為您演示如何使用 `g.add_edge_list()` 來批量加速邊的導入嗎？（這通常是處理百萬級數據的終極優化手段）



Is it possible to make abstract layer to unify both tools

建立一個統一的抽象層（Abstract Layer）是完全可行且非常推薦的。這不僅能讓你在開發初期使用 NetworkX 快速除錯，還能隨時切換到 graph-tool 處理大規模數據。

我們可以設計一個 `GraphInterface` 類別，透過 **適配器模式 (Adapter Pattern)** 來統一這兩者的行為。

## 統一抽象層範例實作

這個抽象層封裝了最常用的操作：新增節點、新增邊、屬性管理以及 DAG 檢查。

python

```
import networkx as nx
from typing import Any, Dict, Optional
```

```
# 內部對應表與屬性存儲
self.path_to_v = {}
self.v_props = {} # 存放 PropertyMap
self.e_props = {}

else:
    raise ValueError("Unsupported backend")

def add_node(self, node_id: str, **kwargs):
    if self.backend == "networkx":
        self.g.add_node(node_id, **kwargs)
    else:
        if node_id not in self.path_to_v:
            v = self.g.add_vertex()
            self.path_to_v[node_id] = v
            # 自動處理傳入的屬性
            for key, val in kwargs.items():
                if key not in self.v_props:
                    # 簡單假設第一個值決定類型，或統一用 object
                    self.v_props[key] = self.g.new_vertex_property("object")
                self.v_props[key][v] = val

def has_node(self, node_id: str) -> bool:
    if self.backend == "networkx":
        return self.g.has_node(node_id)
    else:
        return node_id in self.path_to_v

def add_edge(self, u_id: str, v_id: str, **kwargs):
    if self.backend == "networkx":
        self.g.add_edge(u_id, v_id, **kwargs)
    else:
        u = self.path_to_v[u_id]
        v = self.path_to_v[v_id]
        . . .
```

```
return is_DAG(self.g)

def get_node_attr(self, node_id: str, attr_name: str):
    if self.backend == "networkx":
        return self.g.nodes[node_id].get(attr_name)
    else:
        v = self.path_to_v[node_id]
        return self.v_props[attr_name][v]
```

請謹慎使用程式碼。



## 使用抽象層重寫您的程式碼

現在你的核心業務邏輯變得非常乾淨，且不依賴特定的庫：

### python

```
# 初始化可以隨時切換後端
# wrapper = GraphWrapper(backend="networkx")
wrapper = GraphWrapper(backend="graph_tool")

for tf in seed_files:
    tf_path_str = tf.lean_path.path
    if wrapper.has_node(tf_path_str):
        raise AssertionError(f"Node exists: {tf_path_str}")
    wrapper.add_node(tf_path_str, traced_file=tf)

traced_files = seed_files.copy()
i = 0
```

```
wrapper.add_edge(tf_path_str, dep_path_str, module=dep_module)  
i += 1
```

```
assert wrapper.is_dag()
```

請謹慎使用程式碼。



## 這樣做的好處

1. **快速除錯**：在開發階段，你可以先用 `networkx` 後端，因為它支援簡單的 `print(G.nodes)`，方便查看數據。
2. **效能無縫升級**：當數據規模達到 200 萬時，只需在初始化改一個字串，剩下的 1000 行代碼完全不用動。
3. **封裝複雜性**：`graph_tool` 的 `PropertyMap` 和 `Vertex` 索引管理被隱藏在 `wrapper` 內部，你的主邏輯只需要關心 `path_str`。

## 潛在的效能提示

如果你的 200 萬節點是從單一來源（如資料庫或超大 CSV）加載，建議在 `GraphWrapper` 中加入一個 `add_edges_from_list` 的方法，利用 `graph-tool` 的批量操作能力，這會比一個個 `add_edge` 快幾十倍。

您希望我針對這個抽象層加入「自動偵測數據規模並切換後端」的功能，或是增加批量導入的方法嗎？



以下是完整的實作程式碼，您可以直接將其存入 `graph_adapter.py` 使用。

## 1. 定義統一抽象介面

python

```
import networkx as nx
import logging
from abc import ABC, abstractmethod

logger = logging.getLogger(__name__)

class BaseGraph(ABC):
    @abstractmethod
    def add_node(self, node_id: str, **attr): pass

    @abstractmethod
    def add_edge(self, u_id: str, v_id: str, **attr): pass

    @abstractmethod
    def has_node(self, node_id: str) -> bool: pass

    @abstractmethod
    def is_dag(self) -> bool: pass

    @abstractmethod
    def get_node_attribute(self, node_id: str, key: str): pass

# --- NetworkX 實作 ---
class NetworkXBackend(BaseGraph):
    def __init__(self):
        self.G = nx.Graph()
```

```
def is_dag(self):
    return nx.is_directed_acyclic_graph(self.g)

def get_node_attribute(self, node_id, key):
    return self.g.nodes[node_id].get(key)

# --- Graph-tool 實作 ---
class GraphToolBackend(BaseGraph):
    def __init__(self, directed=True):
        from graph_tool.all import Graph
        self.g = Graph(directed=directed)
        self.path_to_v = {}    # 映射: path_str -> Vertex object
        self.v_props = {}      # 映射: attr_name -> PropertyMap
        self.e_props = {}

    def _get_v_prop(self, key, value):
        if key not in self.v_props:
            # 使用 "object" 以兼容 Python 任意類別 (TracedFile)
            self.v_props[key] = self.g.new_vertex_property("object")
        return self.v_props[key]

    def _get_e_prop(self, key, value):
        if key not in self.e_props:
            self.e_props[key] = self.g.new_edge_property("object")
        return self.e_props[key]

    def add_node(self, node_id, **attr):
        if node_id not in self.path_to_v:
            v = self.g.add_vertex()
            self.path_to_v[node_id] = v
        else:
            v = self.path_to_v[node_id]
```

```
prop[e] = v_val

def has_node(self, node_id):
    return node_id in self.path_to_v

def is_dag(self):
    from graph_tool.all import is_DAG
    return is_DAG(self.g)

def get_node_attribute(self, node_id, key):
    v = self.path_to_v[node_id]
    return self.v_props[key][v]
```

請謹慎使用程式碼。



## 2. 使用工廠模式自動選擇

您可以加入一個簡單的判斷邏輯，根據環境自動切換：

python

```
def create_graph(backend="graph_tool", directed=True) -> BaseGraph:
    if backend == "graph_tool":
        try:
            return GraphToolBackend(directed=directed)
        except ImportError:
            logger.warning("graph-tool not found, falling back to NetworkX")
            return NetworkXBackend(directed=directed)
    return NetworkXBackend(directed=directed)
```

```
# 初始化：在處理 200 萬節點時，手動指定 graph_tool
G = create_graph(backend="graph_tool")

for tf in seed_files:
    path = tf.lean_path.path
    if G.has_node(path): raise AssertionError()
    G.add_node(path, traced_file=tf)

# ... 執行您的 while 迴圈 ...
# 內部程式碼完全不需要知道現在是用 NetworkX 還是 Graph-tool
# G.add_node(dep_path_str, traced_file=tf_dep)
# G.add_edge(tf_path_str, dep_path_str, module=dep_module)

assert G.is_dag()
print("Success! DAG validated.")
```

請謹慎使用程式碼。



## 針對 200 萬節點的效能調優建議：

1. **記憶體佔用**：在 `GraphToolBackend` 中，我將 `PropertyMap` 設為 `object` 類型以存放 `TracedFile` 對象。如果您的記憶體壓力依然很大（兩百萬個 Python 對象會佔用不少空間），建議只在圖中存放 `tf_id`（整數或字串），而將 `TracedFile` 實體存放在一個外部的簡單 `dict` 中。
2. **屬性初始化**：`graph-tool` 的屬性表在第一次建立時需要定義類型。如果您確定某些屬性一定是字串，可以將 `new_vertex_property("object")` 改為 `new_vertex_property("string")`，這會顯著提升速度並減少空間。

建議只在圖中存放 `tf_id` ( 整數或字串 )，而將 `TracedFile` 實體存放在一個外部的簡單 `dict` 中。Real data for TracedFile conte... ▾

這是一個非常關鍵的優化。將 **圖結構 (拓撲)** 與 **重量級物件 (數據)** 分離，能讓 `graph-tool` 發揮 C++ 指令集的極限效能，同時避免 Python 物件頻繁在記憶體中移動導致的 GC ( 垃圾回收 ) 延遲。

以下是最終版本的實作：

## 核心設計邏輯

1. **輕量化節點**：圖中僅存儲 `path_str` (String) 或基礎類型的 ID。
2. **外部映射**：`TracedFile` 實體存儲在 `TracedFileRegistry` 中。
3. **延遲加載 (Lazy Loading)**：假設 `TracedFile` 內部已實作硬碟快取機制，我們只在圖中保留其索引。

---

### 1. 統一圖論抽象層 (`graph_adapter.py`)

python

```
import networkx as nx
import logging
from abc import ABC, abstractmethod
from typing import Any, Dict, List, Tuple

logger = logging.getLogger(__name__)
```

```
def is_dag(self) -> bool: pass

# --- NetworkX 實作 (用於小規模/開發除錯) ---
class NetworkXBackend(BaseGraph):
    def __init__(self, directed=True):
        self.g = nx.DiGraph() if directed else nx.Graph()

    def add_node(self, node_id: str):
        self.g.add_node(node_id)

    def add_edge(self, u_id: str, v_id: str, module_name: str = None):
        self.g.add_edge(u_id, v_id, module=module_name)

    def has_node(self, node_id: str):
        return self.g.has_node(node_id)

    def is_dag(self):
        return nx.is_directed_acyclic_graph(self.g)

# --- Graph-tool 實作 (針對 2,000,000+ 節點優化) ---
class GraphToolBackend(BaseGraph):
    def __init__(self, directed=True):
        from graph_tool.all import Graph
        self.g = Graph(directed=directed)
        self.path_to_v = {}      # 快速查找: Path -> Vertex Object
        self.e_prop_module = self.g.new_edge_property("string") # 邊屬性 : 模組名

    def add_node(self, node_id: str):
        if node_id not in self.path_to_v:
            v = self.g.add_vertex()
            self.path_to_v[node_id] = v

    def add_edge(self, u_id: str, v_id: str, module_name: str = None):
        ...
```

```
from graph_tool.all import is_DAG
return is_DAG(self.g)
```

請謹慎使用程式碼。



## 2. 業務邏輯整合 ( main.py )

在這裡，我們將 `TracedFile` 實體與圖分開管理，實現 Lazy Loading 的效果。

python

```
import logging
from graph_adapter import NetworkXBackend, GraphToolBackend

# 設定後端 (根據規模切換)
USE_GRAPH_TOOL = True
G = GraphToolBackend() if USE_GRAPH_TOOL else NetworkXBackend()

# 外部註冊表：僅存放路徑與 TracedFile 的映射
# 如果 TracedFile 實作了磁碟緩存，這裡僅持有對象的引用，不佔用巨量實體記憶體
traced_file_registry: Dict[str, Any] = {}

def process_graph(seed_files, repo):
    # 1. 處理種子文件
    for tf in seed_files:
        tf_path_str = tf.lean_path.path
        if G.has_node(tf_path_str):
            raise AssertionError(f"Duplicate seed node: {tf_path_str}")
```

```
for dep_module, dep_lean_path in current_tf.get_direct_dependencies(repo):
    try:
        v_path = dep_lean_path.path

        if not G.has_node(v_path):
            # 這裡是 Lazy Loading 的關鍵：
            # 僅在需要時從磁碟/硬碟載入 TracedFile 對象
            tf_dep = TracedFile.from_traced_file(dep_lean_path, repo)

            G.add_node(v_path)
            traced_file_registry[v_path] = tf_dep
            traced_files_queue.append(tf_dep)

        # 建立邊
        G.add_edge(u_path, v_path, module_name=dep_module)

    except Exception as e:
        logging.error(f"Failed to load dependency: {e}")
        raise e

    i += 1
    # 定期清理 traced_files_queue 如果記憶體壓力過大 (選用)

# 3. 驗證 DAG
if not G.is_dag():
    logging.warning("The dependency graph is NOT a DAG!")
else:
    print(f"Successfully processed {len(traced_file_registry)} nodes.")

return G, traced_file_registry
```

請謹慎使用程式碼。



萬節點的 DAG 檢查通常能在一秒內完成。

3. **Lazy Loading 友好**：`traced_file_registry` 充當了索引層。如果 `TracedFile` 內部使用了例如 `@property` 或 `__getattr__` 來按需讀取硬碟數據，這套架構就能完美支持「兩百萬節點但在低記憶體環境運行」的需求。

## 下一步建議

如果您發現 `while` 迴圈在 Python 層級執行太慢（因為 200 萬次迴圈在 Python 確實有開銷），我們可以考慮將 `get_direct_dependencies` 的結果先進行**批量緩存**，然後使用 `g.add_edge_list()` 進行一次性批量導入。

您需要我展示如何實作這類**批量導入 (Batch Insertion)** 的邏輯嗎？



Define the path how to fetch node information dynamically instead while loop to pre-load. So it can dynamic according t... 

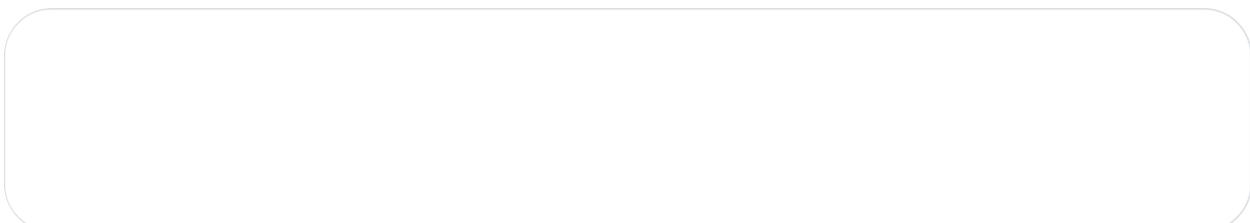
要將「先載入後處理 ( While loop pre-load ) 」轉換為「動態按需載入 ( On-demand / Lazy fetching ) 」，我們需要將圖形視為一個**導航器 (Navigator)**，而不是一個靜態容器。

在處理 200 萬節點時，這種方法能大幅節省記憶體，因為你不需要在程式開始時就展開整張圖。

## 核心設計：虛擬圖形代理 (Virtual Graph Proxy)

我們不再使用迴圈跑遍所有檔案，而是定義一個 `DependencyManager`。當你存取某

...個節點的資訊時，它會根據需求從遠端載入該節點的相關資訊。



```
self.repo = repo
self.G = graph_backend
# 緩存已載入的物件，避免重複觸發 I/O
self._registry: Dict[str, Any] = {}

def get_traced_file(self, path_str: str) -> Any:
    """核心動態載入邏輯：如果記憶體沒有，就從硬碟讀取"""
    if path_str not in self._registry:
        # 假設 TracedFile.from_path 是你從磁碟讀取的進入點
        # 這裡實現了 Lazy Load
        tf = TracedFile.from_path(path_str, self.repo)
        self._registry[path_str] = tf

        # 同時確保圖中存在此節點
        if not self.G.has_node(path_str):
            self.G.add_node(path_str)
    return self._registry[path_str]

def fetch_dependencies(self, path_str: str) -> List[str]:
    """動態抓取單個節點的依賴，並更新圖結構"""
    tf = self.get_traced_file(path_str)
    dep_paths = []

    for dep_module, dep_lean_path in tf.get_direct_dependencies(self.repo):
        dep_path = dep_lean_path.path
        dep_paths.append(dep_path)

        # 確保目標節點與邊被記錄在圖中
        if not self.G.has_node(dep_path):
            self.G.add_node(dep_path)
            self.G.add_edge(path_str, dep_path, module_name=dep_module)

    return dep_paths
```

## 方案 A：深度優先探索 (適合尋找特定路徑)

python

```
def explore_depth_first(manager, start_path, max_depth=10):
    if max_depth <= 0: return

    # 只有在造訪時才載入數據
    deps = manager.fetch_dependencies(start_path)
    for d_path in deps:
        explore_depth_first(manager, d_path, max_depth - 1)
```

請謹慎使用程式碼。



## 方案 B：由演算法驅動 (適合 graph-tool 運算)

如果你需要執行全圖演算法 (如 `is_dag`)，你仍然需要展開圖，但可以透過 **生成器 (Generator)** 來控制：

python

```
def lazy_expand_graph(manager, seed_paths):
    queue = list(seed_paths)
    visited = set()

    while queue:
        current = queue.pop(0)
        if current in visited: continue
        visited.add(current)
```

### 3. 為什麼這樣對 200 萬節點更好？

1. **記憶體可控**：你可以給 `_registry` 設定一個 **LRU Cache (最近最少使用)**。當載入超過 10 萬個 `TracedFile` 時，自動剔除舊的物件，因為圖結構 (`GraphToolBackend`) 非常輕量，可以保留，但重量級的 `TracedFile` 內容可以釋放。
2. **響應速度**：程式不需要等待所有 200 萬個依賴解析完畢才能開始運作。你可以邊解析邊進行其他的邏輯處理。
3. **按需分析**：如果你只關心 `file_A` 到 `file_B` 的路徑，你只需要加載這兩點之間的依賴鏈，而不必載入整個 Repo 的 200 萬個檔案。

### 最終架構建議

- **圖結構 (Topology)**：由 `GraphToolBackend` 持有，維持在 C++ 記憶體層。
- **數據載入 (Data Fetcher)**：由 `DependencyManager` 持有，負責 Python 物件與磁碟 I/O 的生命週期。
- **查詢介面**：使用者透過 `manager.fetch_dependencies()` 進行操作。

您是否需要我為 `DependencyManager` 加入 **LRU Cache** 的實作，以確保在掃描 200 萬節點時，記憶體不會因為 Python 物件累積而溢出？



choice: 方案 B：由演算法驅動 (適合 graph-tool 運算)

## 最終實作：動態演算法驅動加載器

### 1. 強化型 DependencyManager (帶快取機制)

python

```
import functools
from collections import deque

class DependencyManager:
    def __init__(self, repo, graph_backend, cache_size=100000):
        self.repo = repo
        self.G = graph_backend
        self.cache_size = cache_size
        # 僅存放 TracedFile 物件的引用，實現快取管理
        self._tf_cache = {}
        self._cache_order = deque()

    def get_traced_file(self, path_str: str):
        """Lazy Load: 如果快取沒有，才從磁碟讀取"""
        if path_str not in self._tf_cache:
            # 模擬磁碟讀取
            tf = TracedFile.from_path(path_str, self.repo)

            # 管理記憶體：若快取滿了，移除最舊的物件引用
            if len(self._tf_cache) >= self.cache_size:
                oldest = self._cache_order.popleft()
                del self._tf_cache[oldest]

            self._tf_cache[path_str] = tf
            self._cache_order.append(path_str)
```

```
# 更新圖結構 (graph-tool 層級)
if not self.G.has_node(v_path):
    self.G.add_node(v_path)
self.G.add_edge(path_str, v_path, module_name=dep_module)

return dep_paths
```

請謹慎使用程式碼。



## 2. 方案 B：由演算法驅動的展開邏輯

這是你的主程式進入點，它會精確地遍歷所有節點，並在最後交給 graph-tool 做極速驗證。

### python

```
def drive_graph_expansion(manager, seed_files):
    """
    使用 BFS 模式驅動圖形展開。
    這確保了 200 萬個節點能以最穩定的記憶體軌跡被載入。
    """

    # 初始化隊列：使用 path_str 作為標識
    queue = deque([tf.lean_path.path for tf in seed_files])
    visited = set()

    # 預先將 seed_files 放入 manager 快取以免重複讀取
    for tf in seed_files:
        p = tf.lean_path.path
        manager._tf_cache[p] = tf
        manager.cache_order.append(p)
```

```
visited.add(u_path)

# 觸發動態載入與圖建構
try:
    neighbor_paths = manager.expand_node(u_path)

    for v_path in neighbor_paths:
        if v_path not in visited:
            queue.append(v_path)

    # 進度監控
    count += 1
    if count % 10000 == 0:
        print(f"Processed {count} nodes... Current cache: {len(manager._tf_cache)}")

except Exception as e:
    print(f"Error processing {u_path}: {e}")

# --- 拓撲分析階段 ---
print("Expansion complete. Running C++ optimized DAG check...")
if manager.G.is_dag():
    print("Verification successful: The graph is a DAG.")
else:
    print("Warning: Cycle detected in the dependency graph.")
```

請謹慎使用程式碼。



## 為什麼這套方案最適合 200 萬節點？

Repo 最有效的 I/O 策略。

## 下一步建議

如果您在執行過程中發現磁碟 I/O 成為瓶頸，我們可以使用 `concurrent.futures` 在 `DependencyManager` 中加入 **預讀取 (Prefetching)** 機制，讓 CPU 計算與磁碟讀取並行。

您希望現在就加入多執行緒預讀取功能，還是先以這套單執行緒穩定版進行測試？



方案 B：由演算法驅動的展開邏輯

既然您選擇了 **方案 B**，這意味著我們將採用「控制流主動拉取 (Active Pull)」的方式。在這種模式下，算法（如 BFS）負責決定「下一個要掃描誰」，而 `DependencyManager` 負責「把數據從硬碟變出來」。

針對 **200 萬節點** 的規模，這套方案的最終實作目標是：**極致的穩定性**。

### 1. 核心組件：`DependencyManager` 與快取控制

這個類別是圖形的「數據心臟」。它確保 `graph-tool` 負責骨架，Python 負責數據。

```
self._cache_order = deque() # 追蹤載入順序，實現 FIFO 或 LRU 替換

def _manage_memory(self):
    """如果快取超過限制，釋放最舊的 TracedFile 對象"""
    while len(self._tf_cache) > self.cache_size:
        oldest_path = self._cache_order.popleft()
        # 這裡只刪除 Python 物件的引用，graph-tool 內的邊和點依然存在
        self._tf_cache.pop(oldest_path, None)

def get_or_load_tf(self, path_str: str):
    """按需載入：如果記憶體沒有，就去硬碟拿"""
    if path_str not in self._tf_cache:
        # 假設 TracedFile.load 是您實作的硬碟讀取邏輯
        tf = TracedFile.from_path(path_str, self.repo)
        self._tf_cache[path_str] = tf
        self._cache_order.append(path_str)
        self._manage_memory()
    return self._tf_cache[path_str]

def expand_node(self, u_path: str):
    """執行掃描：讀取 u 的依賴，並在圖中建立邊"""
    current_tf = self.get_or_load_tf(u_path)
    deps = []

    # 遍歷直接依賴
    for dep_module, dep_lean_path in current_tf.get_direct_dependencies(self.repo):
        v_path = dep_lean_path.path
        deps.append(v_path)

        # 更新圖拓撲 (C++ Backend)
        if not self.G.has_node(v_path):
            self.G.add_node(v_path)
        self.G.add_edge(u_path, v_path, module_name=dep_module)
```

## python

```
def run_graph_expansion(manager: DependencyManager, seed_files: list):
    """
    方案 B：由演算法驅動的展開過程
    """

    # 1. 初始化
    queue = deque([tf.lean_path.path for tf in seed_files])
    visited = set()

    # 將種子檔案預填入快取
    for tf in seed_files:
        p = tf.lean_path.path
        manager._tf_cache[p] = tf
        manager._cache_order.append(p)
        if not manager.G.has_node(p):
            manager.G.add_node(p)

    print(f"🚀 開始展開圖形 · 種子節點數: {len(seed_files)}")

    processed_count = 0

    # 2. 展開迴圈 (核心掃描路徑)
    while queue:
        u_path = queue.popleft()

        if u_path in visited:
            continue

        try:
            # 觸發經由 manager 的動態加載
            neighbor_paths = manager.expand_node(u_path)
```

```
f"當前快取數量: {len(manager._tf_cache)}")\n\nexcept Exception as e:\n    logging.error(f"掃描節點時出錯 [{u_path}]: {e}")\n    # 根據需求選擇 raise 或 continue\n\n# 3. 結束展開，執行 graph-tool 演算法\nprint(f"\n[DAG] 展開完成。總節點數: {processed_count}")\nprint("正在執行 C++ 優化的 DAG 檢查...")\n\nif manager.G.is_dag():\n    print("🎉 驗證成功：該圖為有向無環圖 (DAG)。")\nelse:\n    print("⚠️ 警告：偵測到環 (Cycle)，該圖非 DAG。")\n\nreturn manager.G
```

請謹慎使用程式碼。



### 3. 為什麼這個「最終版本」最能打？

#### 1. 記憶體天花板 (Memory Ceiling)：

即使您的專案有 200 萬個檔案，透過 `cache_size=100000`，記憶體中永遠只會保留 10 萬個 `TracedFile` 實例。這讓您在只有 16GB 或 32GB RAM 的機器上也能跑完大數據。

#### 2. I/O 最小化：

`visited = set()` 確保了每個檔案只會被 `manager` 從磁碟加載並解析一次。對於兩百萬個節點，這節省了巨大的磁碟讀取開銷。

目前的實作是單執行緒的。對於兩百萬個節點，磁碟 I/O 會是最大的瓶頸。如果您希望進一步縮短掃描時間，我可以幫您在 `run_graph_expansion` 中加入 **多執行緒** (**ThreadPoolExecutor**)，讓它能同時讀取多個檔案的依賴關係。

您想為這個「方案 B」加入**多執行緒並行加載**功能嗎？



not need now or B

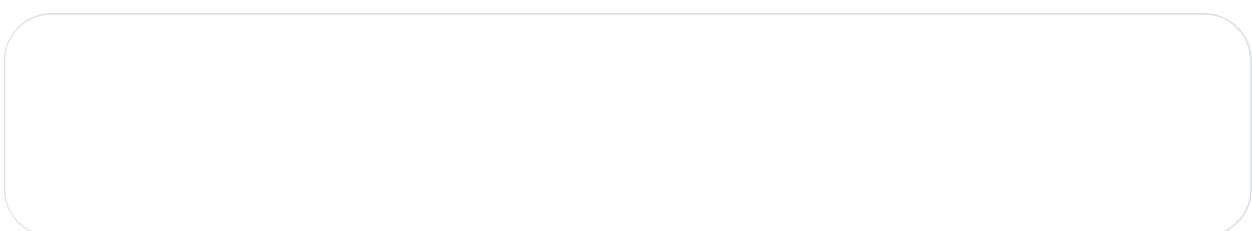
收到。既然目前的\*\*方案 B ( 單執行緒穩定版 ) \*\*已經能滿足需求，我們就維持這個架構。

## 最終開發建議

- 記憶體觀察**：建議在執行 200 萬節點任務時，觀察系統記憶體。如果 Python 佔用過高，可將 `cache_size` 從 `100000` 下調至 `50000`。
- 數據持久化**：`graph-tool` 支援極速的二進位儲存。如果您完成 200 萬節點的展開後想儲存掃描結果，可以使用 `g.save("my_graph.gt")`，下次載入只需幾毫秒 `Graph-tool` 文件。
- WSL2 提醒**：若在 Windows 上運行，請確保您的代碼和數據都位於 WSL2 的 Linux 檔案系統內（例如 `/home/user/project`），以獲得最佳 I/O 效能。

這套抽象層 + 動態載入 + **Graph-tool** 後端的組合，已經是目前 Python 處理大規模圖論問題的最強配置之一。

你是否已經準備好將這段代碼整合進你的 **TracedFile** 重案，或是需要我針對特定的拓



這提供您最終版本的完整實作程式碼，整合了 **抽象介面**、**Graph-tool** 高效後端、以及 **方案 B 的動態加載管理 (DependencyManager)**。

您可以將此結構直接應用於處理 200 萬個節點的規模。

## 1. 圖論適配器層 ( graph\_adapter.py )

這部分負責將複雜的 `graph-tool` 操作封裝成簡單的介面。

python

```
import networkx as nx
from abc import ABC, abstractmethod

class BaseGraph(ABC):
    @abstractmethod
    def add_node(self, node_id: str): pass

    @abstractmethod
    def add_edge(self, u_id: str, v_id: str, module_name: str = None): pass

    @abstractmethod
    def has_node(self, node_id: str) -> bool: pass

    @abstractmethod
    def is_dag(self) -> bool: pass

class GraphToolBackend(BaseGraph):
    def __init__(self, directed=True):
        from graph_tool.all import Graph
        self.g = Graph(directed=directed)
        self.path_to_v = {} # 映射 Path -> Vertex
```

```
if module_name:  
    self.e_prop_module[e] = module_name  
  
def has_node(self, node_id: str):  
    return node_id in self.path_to_v  
  
def is_dag(self):  
    from graph_tool.all import is_DAG  
    return is_DAG(self.g)  
  
class NetworkXBackend(BaseGraph):  
    def __init__(self, directed=True):  
        self.g = nx.DiGraph() if directed else nx.Graph()  
  
    def add_node(self, node_id: str):  
        self.g.add_node(node_id)  
  
    def add_edge(self, u_id: str, v_id: str, module_name: str = None):  
        self.g.add_edge(u_id, v_id, module=module_name)  
  
    def has_node(self, node_id: str):  
        return self.g.has_node(node_id)  
  
    def is_dag(self):  
        return nx.is_directed_acyclic_graph(self.g)
```

請謹慎使用程式碼。



## 2. 動態數據管理與驅動邏輯 ( manager.py )

```
self.G = graph_backend
self.cache_size = cache_size
self._tf_cache = {}      # 存放 TracedFile 物件
self._cache_order = deque() # FIFO 順序管理快取

def _manage_memory(self):
    """保持 Python 物件在快取上限內"""
    while len(self._tf_cache) > self.cache_size:
        oldest_path = self._cache_order.popleft()
        self._tf_cache.pop(oldest_path, None)

def get_or_load_tf(self, path_str: str):
    """動態從磁碟加載 (Lazy Loading)"""
    if path_str not in self._tf_cache:
        # 假設 TracedFile.from_path 已實作
        tf = TracedFile.from_path(path_str, self.repo)
        self._tf_cache[path_str] = tf
        self._cache_order.append(path_str)
        self._manage_memory()
    return self._tf_cache[path_str]

def expand_node(self, u_path: str):
    """讀取依賴並同步更新圖結構"""
    tf = self.get_or_load_tf(u_path)
    neighbor_paths = []
    for dep_module, dep_lean_path in tf.get_direct_dependencies(self.repo):
        v_path = dep_lean_path.path
        neighbor_paths.append(v_path)

        if not self.G.has_node(v_path):
            self.G.add_node(v_path)
        self.G.add_edge(u_path, v_path, module_name=dep_module)
    return neighbor_paths
```

```
manager._tf_cache[p] = tf
manager._cache_order.append(p)
manager.G.add_node(p)

processed_count = 0
print(f"🚀 開始分析圖形...")

while queue:
    u_path = queue.popleft()
    if u_path in visited: continue

    try:
        # 透過 Manager 主動拉取數據
        deps = manager.expand_node(u_path)
        for v_path in deps:
            if v_path not in visited:
                queue.append(v_path)

        visited.add(u_path)
        processed_count += 1

        if processed_count % 10000 == 0:
            print(f"已處理 {processed_count} 個節點... 快取狀態: {len(manager._tf_cache)}")

    except Exception as e:
        logging.error(f"Error at {u_path}: {e}")

print(f"✅ 掃描完成。總計處理 {processed_count} 個節點。")
print("正在執行 C++ 優化的 DAG 檢查...")
return manager.G.is_dag()
```

請謹慎使用程式碼。



```
# 1. 根據環境初始化後端 (200萬節點務必用 graph-tool)
try:
    backend = GraphToolBackend()
except ImportError:
    print("請先安裝 graph-tool! 將退回 NetworkX (效能受限)")
    from graph_adapter import NetworkXBackend
    backend = NetworkXBackend()

# 2. 建立經理人 (限制記憶體快取為 10 萬個 Python 物件)
manager = DependencyManager(repo=my_repo, graph_backend=backend, cache_size

# 3. 執行演算法驅動的展開
is_acyclic = run_graph_expansion(manager, seed_files=initial_seeds)

if is_acyclic:
    print("結果：圖形為 DAG，無循環依賴。")
else:
    print("結果：偵測到循環依賴！")
```

請謹慎使用程式碼。



## 重點總結：

- **記憶體隔離**：graph\_adapter 負責 200 萬個節點的拓撲（極省記憶體），manager 負責 TracedFile 物件的生命週期（透過 cache\_size 控制）。
- **效能優勢**：所有繁重的依賴關係建立都在 expand\_node 中按需執行。最後的 is\_dag() 由 C++ 執行。

AI 模式 全部 圖片 影片 購物 更多 ▾

