# 实验报告：Ullmann子图同构算法

## Overview

📌 程序源代码见文件或GitHub仓库：https://github.com/sylvain-wei/GraphData-2024Fall-PKU

语言：Python

Ullmann子图同构主要经过三个步骤

1. 首先是过滤、剪枝得到候选矩阵M。候选矩阵M表达的是从查询图到数据图的节点映射，每一个查询图节点可以对应若干个数据图节点。
2. 其次是枚举搜索，通过回溯搜索，得到映射矩阵M'。每一个映射矩阵M'代表一个从匹配图到数据图节点的单射匹配。
3. 最后是验证映射矩阵M'是否满足子图匹配条件（1）节点标签一致性（2）拓扑一致性，筛选出符合要求的子图匹配映射。

具体代码实现中：

Ullmann.py是Ullmann算法的主要框架。

主要步骤包括：

1. 调用read_graphs函数读取数据图、查询图。每一个图为node_labels、adj_matrix和edge_labels三个属性。
2. 调用filter_and_plan函数，使用LDF、NLF和剪枝操作，提前剪枝以减少后续搜索成本以及验证成本。该函数在filter_and_plan.py中实现。
3. 调用enumerate函数，根据剪枝后的搜索空间进行全局搜索，得到所有满足要求的单射匹配Ms=list of M'。该函数在enumerate.py中实现。
4. 调用verify函数，验证每一个单射匹配M'是否满足子图同构基本定义。该函数在verify.py中实现。

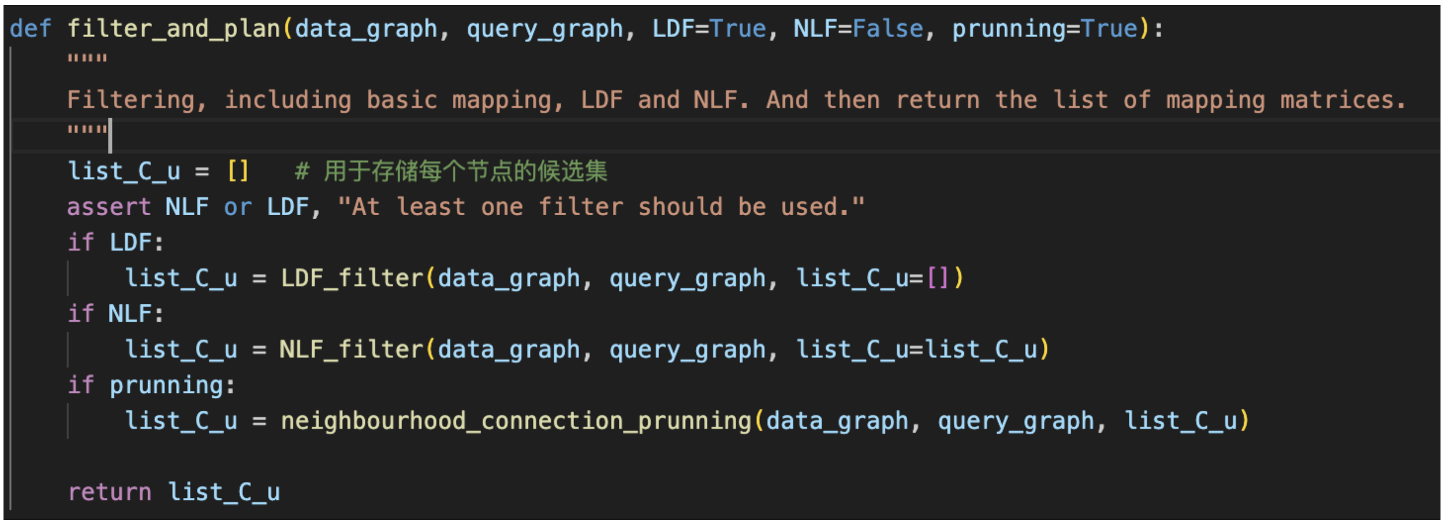
## 算法复杂度分析

此处以每一个查询图-数据图pair的匹配过程进行分析。

假设查询图节点数为n，数据图节点数为m，查询图平均度数为d，数据图平均度数为D。

### LDF、NLF过滤和剪枝

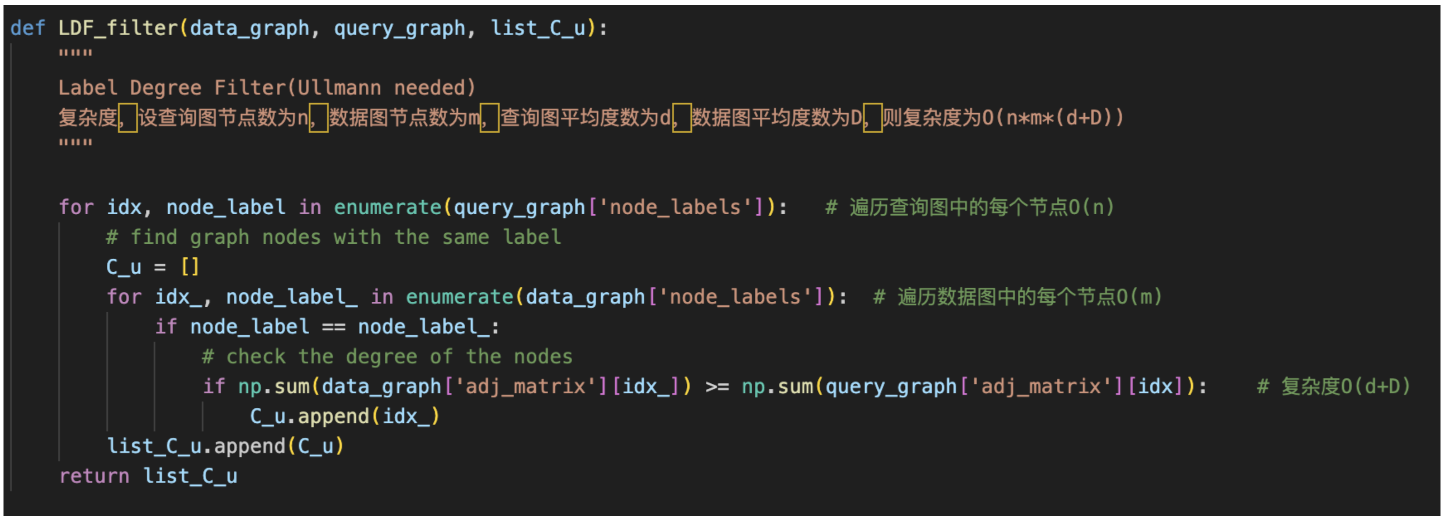
那么在LDF、NLF、剪枝操作中（filter_and_plan.py）



```python
def filter_and_plan(data_graph, query_graph, LDF=True, NLF=False, prunning=True):
    """
    Filtering, including basic mapping, LDF and NLF. And then return the list of mapping matrices.
    """
    list_C_u = []   # 用于存储每个节点的候选集
    assert NLF or LDF, "At least one filter should be used."
    if LDF:
        list_C_u = LDF_filter(data_graph, query_graph, list_C_u=[])
    if NLF:
        list_C_u = NLF_filter(data_graph, query_graph, list_C_u=list_C_u)
    if prunning:
        list_C_u = neighbourhood_connection_prunning(data_graph, query_graph, list_C_u)

    return list_C_u
```

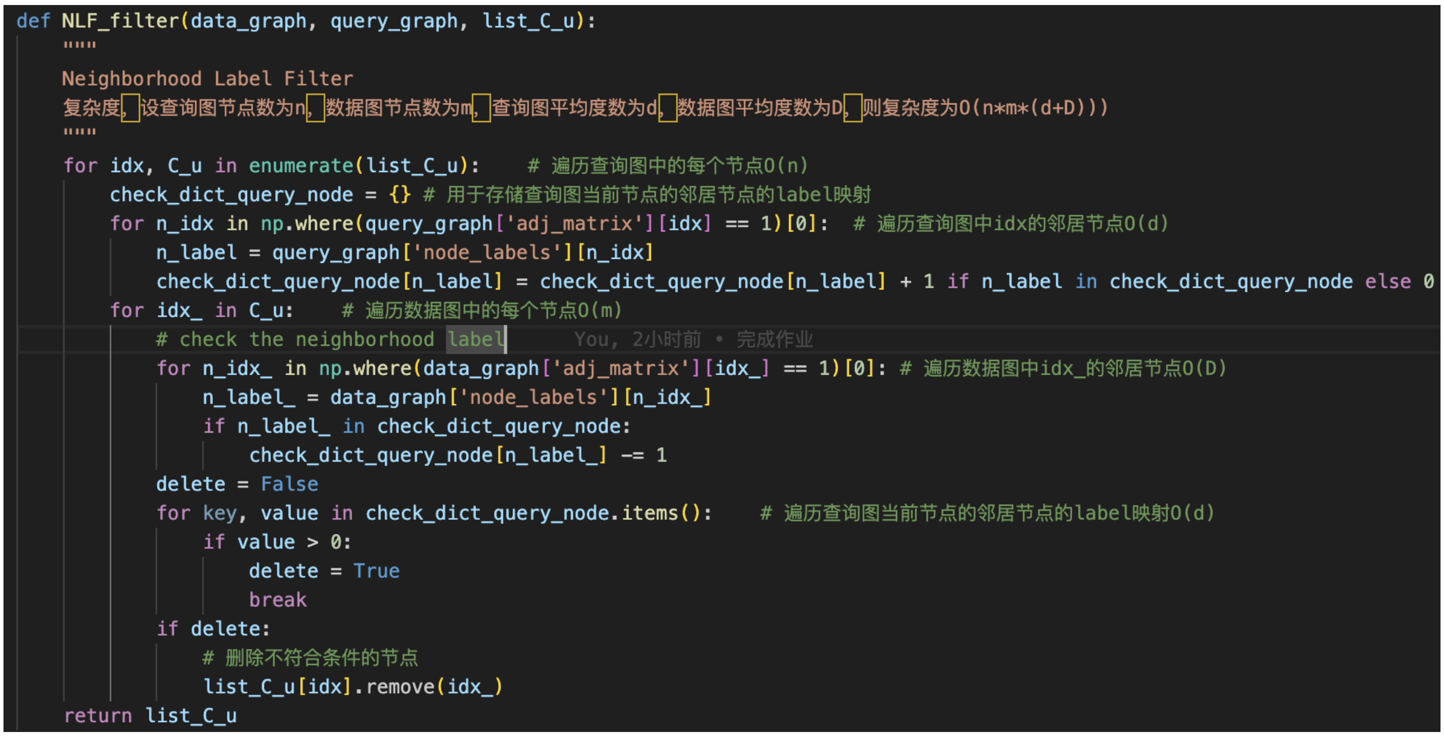
初始化，定义list_C_u用于存储每一个节点候选集。

首先LDF_filter函数中，分别遍历查询图、数据图每一个节点，对比时间复杂度为O(n*m)



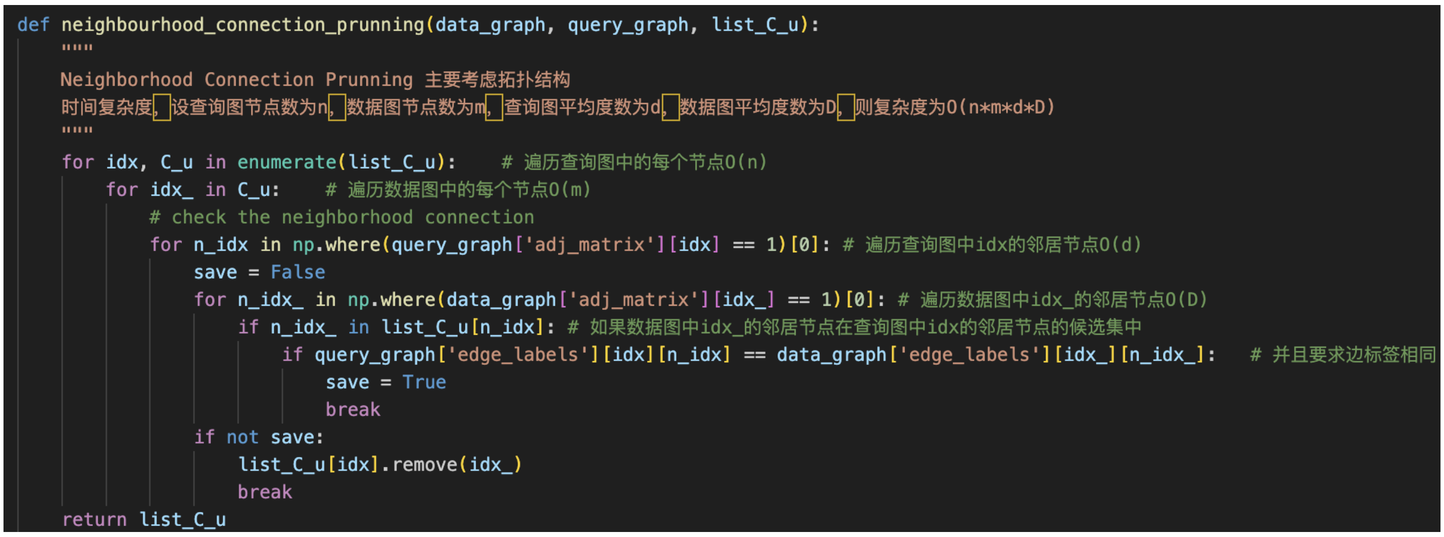
```python
def LDF_filter(data_graph, query_graph, list_C_u):
    """
    Label Degree Filter(Ullmann needed)
    复杂度，设查询图节点数为n，数据图节点数为m，查询图平均度数为d，数据图平均度数为D，则复杂度为O(n*m*(d+D))
    """

    for idx, node_label in enumerate(query_graph['node_labels']):   # 遍历查询图中的每个节点O(n)
        # find graph nodes with the same label
        C_u = []
        for idx_, node_label_ in enumerate(data_graph['node_labels']):  # 遍历数据图中的每个节点O(m)
            if node_label == node_label_:
                # check the degree of the nodes
                if np.sum(data_graph['adj_matrix'][idx_]) >= np.sum(query_graph['adj_matrix'][idx]):    # 复杂度O(d+D)
                    C_u.append(idx_)
        list_C_u.append(C_u)
    return list_C_u
```

其次，NLF_filter函数中，如下图所示，时间复杂度为O(n*m*(d+D))



```python
def NLF_filter(data_graph, query_graph, list_C_u):
    """
    Neighborhood Label Filter
    复杂度,□设查询图节点数为n,□数据图节点数为m,□查询图平均度数为d,□数据图平均度数为D,□则复杂度为O(n*m*(d+D)))
    """
    for idx, C_u in enumerate(list_C_u):     # 遍历查询图中的每个节点O(n)
        check_dict_query_node = {} # 用于存储查询图当前节点的邻居节点的label映射
        for n_idx in np.where(query_graph['adj_matrix'][idx] == 1)[0]:  # 遍历查询图中idx的邻居节点O(d)
            n_label = query_graph['node_labels'][n_idx]
            check_dict_query_node[n_label] = check_dict_query_node[n_label] + 1 if n_label in check_dict_query_node else 0
        for idx_ in C_u:     # 遍历数据图中的每个节点O(m)
            # check the neighborhood label        You, 2小时前 • 完成作业
            for n_idx_ in np.where(data_graph['adj_matrix'][idx_] == 1)[0]: # 遍历数据图中idx_的邻居节点O(D)
                n_label_ = data_graph['node_labels'][n_idx_]
                if n_label_ in check_dict_query_node:
                    check_dict_query_node[n_label_] -= 1
            delete = False
            for key, value in check_dict_query_node.items():     # 遍历查询图当前节点的邻居节点的label映射O(d)
                if value > 0:
                    delete = True
                    break
            if delete:
                # 删除不符合条件的节点
                list_C_u[idx].remove(idx_)
    return list_C_u
```

然后，邻居连接剪枝neighbourhood_connection_prunning函数中，时间复杂度为O(n*m*d*D)

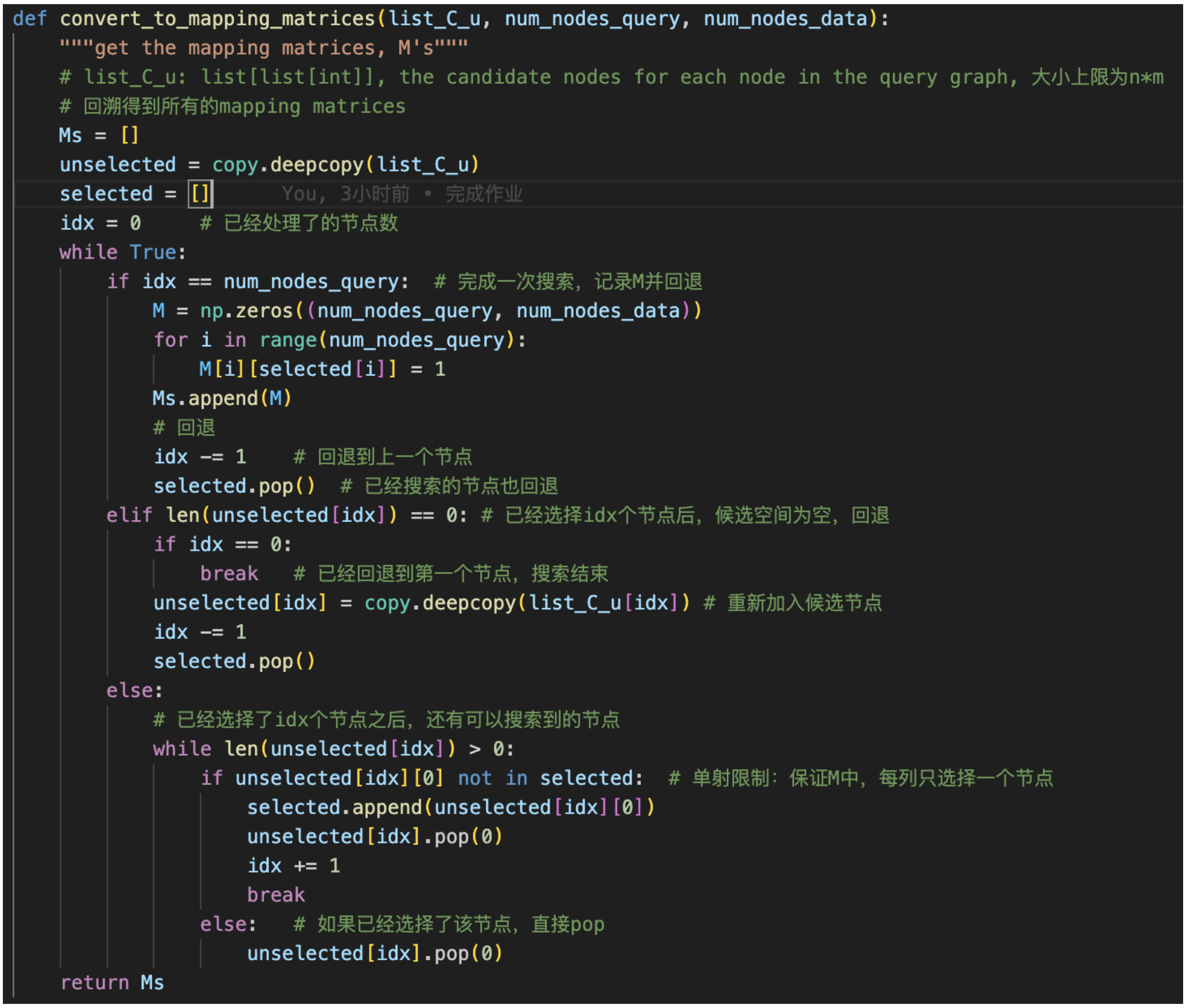


```python
def neighbourhood_connection_prunning(data_graph, query_graph, list_C_u):
    """
    Neighborhood Connection Pruning 主要考虑拓扑结构
    时间复杂度,□设查询图节点数为n,□数据图节点数为m,□查询图平均度数为d,□数据图平均度数为D,□则复杂度为O(n*m*d*D)
    """
    for idx, C_u in enumerate(list_C_u):     # 遍历查询图中的每个节点O(n)
        for idx_ in C_u:     # 遍历数据图中的每个节点O(m)
            # check the neighborhood connection
            for n_idx in np.where(query_graph['adj_matrix'][idx] == 1)[0]: # 遍历查询图中idx的邻居节点O(d)
                save = False
                for n_idx_ in np.where(data_graph['adj_matrix'][idx_] == 1)[0]: # 遍历数据图中idx_的邻居节点O(D)
                    if n_idx_ in list_C_u[n_idx]: # 如果数据图中idx_的邻居节点在查询图中idx的邻居节点的候选集中
                        if query_graph['edge_labels'][idx][n_idx] == data_graph['edge_labels'][idx_][n_idx_]:   # 并且要求边标签相同
                            save = True
                            break
                if not save:
                    list_C_u[idx].remove(idx_)
                    break
    return list_C_u
```

综上所述，过滤、剪枝步骤的时间复杂度为O(n*m*d*D)

## 枚举

枚举阶段相当于从过滤得到的候选节点列表list_C_u或者候选矩阵M中进行回溯搜索。可以抽象为一棵树，每一个棵树有n层、m个节点。所以搜索整棵树的时间复杂度为O(m^n)。

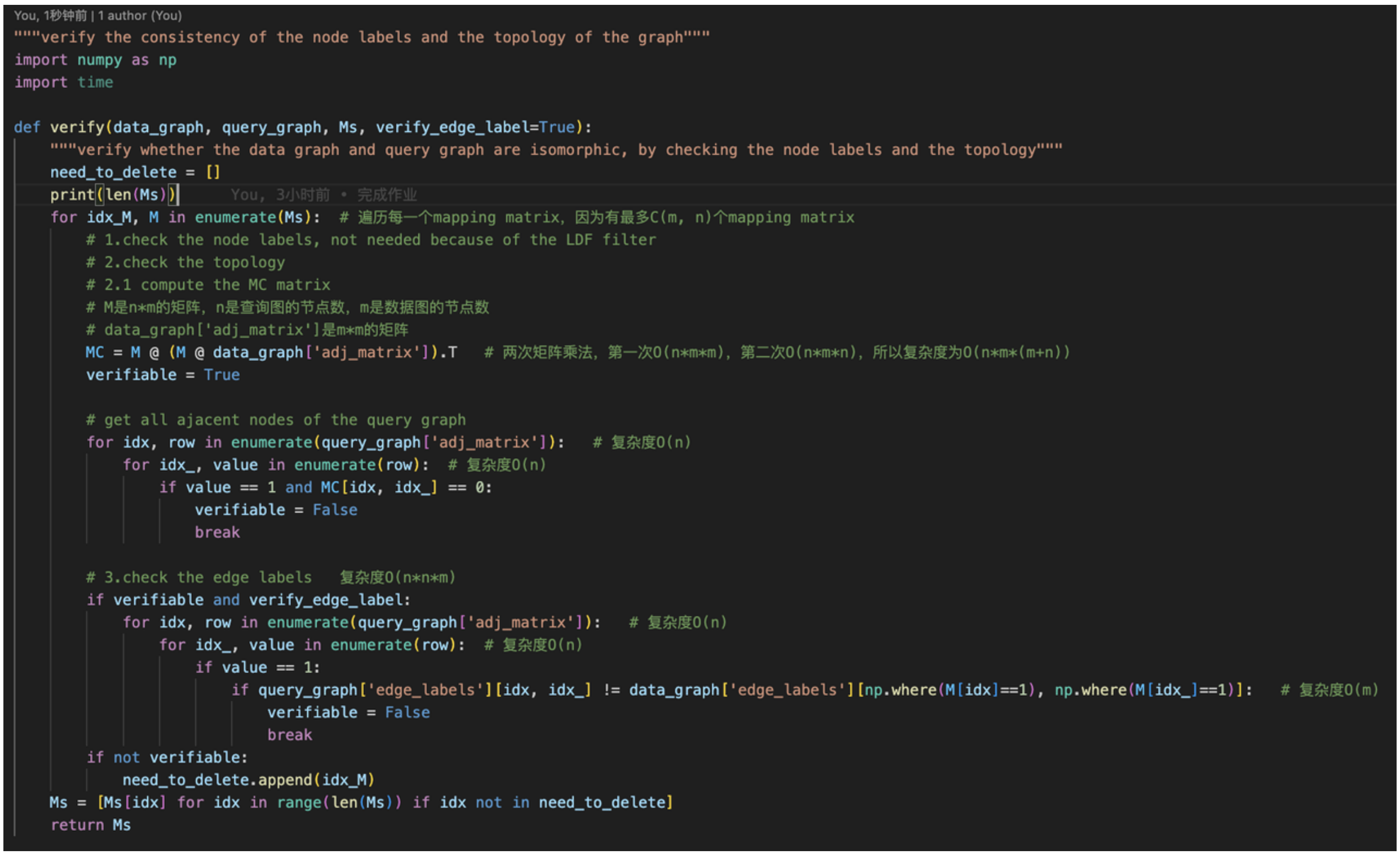


```python
def convert_to_mapping_matrices(list_C_u, num_nodes_query, num_nodes_data):
    """get the mapping matrices, M's"""
    # list_C_u: list[list[int]], the candidate nodes for each node in the query graph, 大小上限为n*m
    # 回溯得到所有的mapping matrices
    Ms = []
    unselected = copy.deepcopy(list_C_u)
    selected = []        You, 3小时前 • 完成作业
    idx = 0      # 已经处理了的节点数
    while True:
        if idx == num_nodes_query:  # 完成一次搜索，记录M并回退
            M = np.zeros((num_nodes_query, num_nodes_data))
            for i in range(num_nodes_query):
                M[i][selected[i]] = 1
            Ms.append(M)
            # 回退
            idx -= 1     # 回退到上一个节点
            selected.pop()  # 已经搜索的节点也回退
        elif len(unselected[idx]) == 0: # 已经选择idx个节点后，候选空间为空，回退
            if idx == 0:
                break    # 已经回退到第一个节点，搜索结束
            unselected[idx] = copy.deepcopy(list_C_u[idx]) # 重新加入候选节点
            idx -= 1
            selected.pop()
        else:
            # 已经选择了idx个节点之后，还有可以搜索到的节点
            while len(unselected[idx]) > 0:
                if unselected[idx][0] not in selected:  # 单射限制：保证M中，每列只选择一个节点
                    selected.append(unselected[idx][0])
                    unselected[idx].pop(0)
                    idx += 1
                    break
                else:    # 如果已经选择了该节点，直接pop
                    unselected[idx].pop(0)
    return Ms
```

## 验证

验证时，第一步是点标签一致性验证。由于点标签在LDF_filter中已经保证了，所以不需要再次验证。对于每一个M'矩阵进行验证时，第二步是验证拓扑一致性，时间复杂度主要在矩阵计算上面，因为M'是n*m的矩阵，n是查询图的节点数，m是数据图的节点数，data_graph['adj_matrix']是m*m的矩阵，两次矩阵乘法，第一次O(n*m*m)，第二次O(n*m*n)，所以复杂度为O(n*m*(m+n))；第三步是验证边标签一致性（但是按照课程Ullmann算法的情景，不一定需要验证边标签），复杂度为O(n*n*m)。综上，验证环节的复杂度为O(n*m*(m+n))。

由于M'矩阵一共有C(m, n)个（组合数），所以复杂度为O(C(m, n) * n * m * (m+n))



```
You, 1秒钟前 | 1 author (You)
"""verify the consistency of the node labels and the topology of the graph"""
import numpy as np
import time

def verify(data_graph, query_graph, Ms, verify_edge_label=True):
    """verify whether the data graph and query graph are isomorphic, by checking the node labels and the topology"""
    need_to_delete = []
    print(len(Ms))      You, 3小时前 • 完成作业
    for idx_M, M in enumerate(Ms):  # 遍历每一个mapping matrix, 因为有最多C(m, n)个mapping matrix
        # 1.check the node labels, not needed because of the LDF filter
        # 2.check the topology
        # 2.1 compute the MC matrix
        # M是n*m的矩阵, n是查询图的节点数, m是数据图的节点数
        # data_graph['adj_matrix']是m*m的矩阵
        MC = M @ (M @ data_graph['adj_matrix']).T   # 两次矩阵乘法, 第一次O(n*m*m), 第二次O(n*m*n), 所以复杂度为O(n*m*(m+n))
        verifiable = True

        # get all ajacent nodes of the query graph
        for idx, row in enumerate(query_graph['adj_matrix']):   # 复杂度O(n)
            for idx_, value in enumerate(row):  # 复杂度O(n)
                if value == 1 and MC[idx, idx_] == 0:
                    verifiable = False
                    break

        # 3.check the edge labels   复杂度O(n*n*m)
        if verifiable and verify_edge_label:
            for idx, row in enumerate(query_graph['adj_matrix']):   # 复杂度O(n)
                for idx_, value in enumerate(row):  # 复杂度O(n)
                    if value == 1:
                        if query_graph['edge_labels'][idx, idx_] != data_graph['edge_labels'][np.where(M[idx]==1), np.where(M[idx_]==1)]:   # 复杂度O(m)
                            verifiable = False
                            break
        if not verifiable:
            need_to_delete.append(idx_M)
    Ms = [Ms[idx] for idx in range(len(Ms)) if idx not in need_to_delete]
    return Ms
```

综合三个步骤，时间复杂度为：

$$O(nmdD) + O(m^n) + O(C(m,n) * nm(m+n)) = O(nm(dD + C(m,n) * (m+n)) + m^n)$$

## 空间复杂度分析

仍然假设查询图节点数为n，数据图节点数为m。

首先需要保存查询图、数据图的node_label、邻接矩阵、边标签：

- 查询图
    - node_label: O(n)
    - 邻接矩阵: O(n*n)
    - 边标签：O(n*n)
- 数据图
    - node_label: O(m)
    - 邻接矩阵: O(m*m)
    - 边标签：O(m*m)

在filter_and_plan.py中，创建了一个list_C_u，保存查询图每一个节点的候选映射节点，复杂度为O(n*m)。

由enumrate.py中创建的映射矩阵M'个数为O(m^n)，而每一个矩阵的复杂度为O(n*m)，所以全部M'的空间复杂度为 $O(n*m^{n+1})$ 。

在verify中，主要在矩阵乘法处创建新的矩阵，所以复杂度为O(n*n)。

综上所述，空间复杂度为 $O(n*m^{n+1}+n^2)$ ，但是因为一般n<m，所以空间复杂度为 $O(n*m^{n+1})$

## 实验结果

由于时间复杂度过高，所以选择data图中前3个图，以及每一个查询图文件的前2个图作为数据跑一遍。

### 对于大小为4的query图：

python Ullmann.py --data graphDB/3examples.data --query graphDB/Q4-2examples.my

结果保存在results/match_result_Q4-2examples.my_3examples.data.json

```
Query graphs: from graphDB/Q4-2examples.my, which has 2 graphs
Data graphs: from graphDB/3examples.data, which has 3 graphs
Query graph 0 against data graph 0
79200
Isomorphism result: [{0: 0, 1: 2, 2: 4, 3: 6, 4: 10}, {0: 1, 1: 0, 2: 2, 3: 4, 4: 6}, {0: 2, 1: 4, 2: 6, 3: 10, 4: 13}, {0: 6, 1: 4, 2: 2, 3: 0, 4: 1}, {0: 10, 1: 6, 2: 4, 3: 2, 4: 0}, {0: 11, 1: 15, 2: 12, 3: 16, 4: 18}, {0: 12, 1: 15, 2: 11, 3: 14, 4: 17}, {0: 13, 1: 10, 2: 6, 3: 4, 4: 2}, {0: 14, 1: 11, 2: 15, 3: 12, 4: 16}, {0: 16, 1: 12, 2: 15, 3: 11, 4: 14}, {0: 17, 1: 14, 2: 11, 3: 15, 4: 12}, {0: 18, 1: 16, 2: 12, 3: 15, 4: 11}]

Query graph 0 against data graph 1
0
Isomorphism result: []

Query graph 0 against data graph 2
6720
Isomorphism result: []

Query graph 1 against data graph 0
0
Isomorphism result: []

Query graph 1 against data graph 1
0
Isomorphism result: []

Query graph 1 against data graph 2
0
```

```
Isomorphism result: []

time_filter_plan: 0.0005183219909667969
time_enumerate: 0.04097954432169596
time_verify: 2.639620542526245
time_subgraph_match: 2.6811197996139526
```

## 对于大小为8的query图：

python Ullmann.py --data graphDB/3examples.data --query graphDB/Q8-2examples.my

结果保存在results/match_result_Q8-2examples.my_3examples.data.json

输出：

```
Query graphs: from graphDB/Q8-2examples.my, which has 2 graphs
Data graphs: from graphDB/3examples.data, which has 3 graphs
Query graph 0 against data graph 0
456960
Isomorphism result: []

Query graph 0 against data graph 1
0
Isomorphism result: []

Query graph 0 against data graph 2
60480
Isomorphism result: []

Query graph 1 against data graph 0
0
Isomorphism result: []

Query graph 1 against data graph 1
0
Isomorphism result: []

Query graph 1 against data graph 2
0
Isomorphism result: []

time_filter_plan: 0.0009919007619222004
time_enumerate: 0.3428366184234619
time_verify: 90.85457630952199
time_subgraph_match: 91.19840709368388
```

## 对于大小为16的query图：

python Ullmann.py --data graphDB/3examples.data --query graphDB/Q16-2examples.my

结果保存在results/match_result_Q16-2examples.my_3examples.data.json

输出：

```
Query graphs: from graphDB/Q16-2examples.my, which has 2 graphs
Data graphs: from graphDB/3examples.data, which has 3 graphs
Query graph 0 against data graph 0
0
Isomorphism result: []

Query graph 0 against data graph 1
0
Isomorphism result: []

Query graph 0 against data graph 2
0
Isomorphism result: []

Query graph 1 against data graph 0
0
Isomorphism result: []

Query graph 1 against data graph 1
0
Isomorphism result: []

Query graph 1 against data graph 2
0
Isomorphism result: []

time_filter_plan: 0.001646598180135091
time_enumerate: 5.904833475748698e-05
time_verify: 7.152557373046875e-06
time_subgraph_match: 0.001713871955871582
```

## 对于大小为20的query图：

python Ullmann.py --data graphDB/3examples.data --query graphDB/Q20-2examples.my

结果保存在：results/match_result_Q20-2examples.my_3examples.data.json

输出：

```
Query graphs: from graphDB/Q20-2examples.my, which has 2 graphs
Data graphs: from graphDB/3examples.data, which has 3 graphs
Query graph 0 against data graph 0
0
Isomorphism result: []

Query graph 0 against data graph 1
0
Isomorphism result: []

Query graph 0 against data graph 2
0
Isomorphism result: []

Query graph 1 against data graph 0
0
Isomorphism result: []

Query graph 1 against data graph 1
0
Isomorphism result: []

Query graph 1 against data graph 2
0
Isomorphism result: []

time_filter_plan: 0.0015791257222493489
time_enumerate: 0.0034161011377970376
time_verify: 5.602836608886719e-06
time_subgraph_match: 0.005001664161682129
```

## 对于大小为24的query图：

python Ullmann.py --data graphDB/3examples.data --query graphDB/Q24-2examples.my

结果保存在：results/match_result_Q24-2examples.my_3examples.data.json

输出：

```
Query graphs: from graphDB/Q24-2examples.my, which has 2 graphs
Data graphs: from graphDB/3examples.data, which has 3 graphs
Query graph 0 against data graph 0
0
Isomorphism result: []
```

```

Query graph 0 against data graph 1
0
Isomorphism result: []

Query graph 0 against data graph 2
0
Isomorphism result: []

Query graph 1 against data graph 0
0
Isomorphism result: []

Query graph 1 against data graph 1
0
Isomorphism result: []

Query graph 1 against data graph 2
0
Isomorphism result: []

time_filter_plan: 0.002213756243387858
time_enumerate: 0.008560379346211752
time_verify: 7.271766662597656e-06
time_subgraph_match: 0.010782559712727865
```