EE359-Multi-Graph-Matching

Python implementation of three algorithms in paper "Unifying Offline and Online Multi-graph Matching via Finding Shortest Paths on Supergraph". Homework of EE359, Prof. Junchi Yan.

EE359 HW Report

戴昊悦 李竞宇

517030910{288,318}

---

1. Overview of the HW

In this short project we're required to implement Python code of the three algorithms proposed in Jiang, Z., Wang, T., & Yan, J. (2020). Unifying Offline and Online Multi-graph Matching via Finding Shortest Paths on Supergraph. IEEE Transactions on Pattern Analysis and Machine Intelligence, 14(8), 1–1..

As follows we'll illustrate how we reproduce the algorithms, the result of pass test, and some of our observations.

2. Implementation

- **Affinity Score**

In the graph matching synthesis, affinity score is designed to measure two-graph matching, usually written as a quadratic assignment programming (QAP) problem which is also called Lawler's QAP:

$$J(\mathbf{X}) = \min_{\mathbf{X}\in\{0,1\}^{n_1\times n_2}} \mathrm{vec}(\mathbf{X})^{\mathsf{T}}\mathbf{K}\,\mathrm{vec}(\mathbf{X})$$

where $\mathbf{X}$ is a (partial) permutation matrix indicating the node correspondence, and $\mathbf{K} \in \mathbb{R}^{n1n2\times n1n2}$ is the affinity matrix whose diagonal (off-diagonal) encodes the node-to-node affinity (edge-to-edge affinity) between two graphs. The symbol $\mathrm{vec}(\cdot)$ here denotes the column-wise vectorization of the input matrix.

In our practice `X` is the matching result of multi graphs in shape `(num_graph, num_graph, num_node, num_node)`, where $\mathbf{X}$ in above formula is `X[i,j]`. Thus instead of calculating each pair of graphs, we can compute them in a bunch:

```python
def cal_affinity_score(X, K):
    """
    :param X: matching results, (num_graph, num_graph, num_node, num_node)
    :param K: affinity matrix, (num_graph, num_graph, num_node^2, num_node^2)
    :return: normalized affinity score, (num_graph, num_graph)
    """
    n, _, m, _ = X.shape
    vx = np.reshape(X.transpose((0, 1, 3, 2)), newshape=(n, n, -1, 1))
    vxT = vx.transpose((0, 1, 3, 2))
    affinity_score = np.matmul(np.matmul(vxT, K), vx)  # in shape (n, n, 1, 1)
    normalized_affinity_score = affinity_score.reshape(n, n) / np.max(affinity_score)
    return normalized_affinity_score
```

Note that affinity score is normalized to range `(0,1]` to be consistent with pairwise consistency.

- **Pairwise Consistency**

In the proposed unified approaches, given $\{G_k\}_{k-1}^N$ and matching configuration $\mathbb{X}$, for any pair G_i and G_j, the pairwise consistency is defined as:

$$C_p\left(\mathbf{X}_{ij}, \mathbb{X}\right) = 1 - \frac{\sum_{k=1}^N \left\|\mathbf{X}_{ij} - \mathbf{X}_{ik}\mathbf{X}_{kj}\right\|_F}{2nN} \in (0, 1]$$

Though it's defined in a `for any` way and `k` is traversed as $\sum_{k=1}^N$, we don't need to write the code with three `for` loop, since it's mutually independent to compute each pair G_i and G_j, as well as the summation of `k`.

Computation of $C_p\left(\mathbf{X}_{ij}\right)$ is related to $\mathbf{X}_{ik}\mathbf{X}_{kj}$ for k from 1 to N. This is similar to the form of matrix multiplication. However we want one step before: where $\mathbf{X}_{ik}\mathbf{X}_{kj}$ haven't been summarized so that $\left\|\mathbf{X}_{ij} - \mathbf{X}_{ik}\mathbf{X}_{kj}\right\|_F$ operation can be done. Thus we use `broadcasting` in `numpy` to align shapes with additional dimensions we add. Note that we need to swap the two axes with `transpose` to achieve it:

```python
def cal_pairwise_consistency(X):
    """
    :param X: matching results, (num_graph, num_graph, num_node, num_node)
    :return: pairwise_consistency: (num_graph, num_graph)
    """
    n, _, m, _ = X.shape
    X_t = X.transpose((1, 0, 2, 3)) # so that X_t[j,k] = X[k,j]
    pairwise_consistency = 1 - np.abs(X[:, :, None] - np.matmul(X[:, None], X_t[None,
        .sum((2, 3, 4)) / (2 * n * m)
    # X[:,None]*X_t[None,...] is X[i,k]*X[k,j] (matmul)
    return pairwise_consistency
```

We've had questions about whether to use pointwise or matrix multiplication here, which will be pointed out later.

- **MGM-Floyd**

MGM-Floyd is used for offline multiple graph matching. It's able to find the optimal composition path more efficiently with fewer comparisons and thus being more competitive. Pseudocode provided in the paper:

Algorithm 1: MGM-Floyd (Offline MGM)

Input: affinity matrix $\{\mathbf{K}_{ij}\}_{i,j=1}^{N}$, initial $\mathbb{X}^{(0)}$, λ.
1 Set consistency weight to 0 for affinity based boosting.
2 **for** *each graph* G_v **do**
3 **for** *each pair of graphs* G_x, G_y **do**
4 set $S_{org} = S(\mathbf{X}_{xy})$ by Eq. 5 (or using approximate $S_{pc}^{\mathbb{X}}$ by Eq. 7, $S_{uc}^{\mathbb{X}}$ by Eq. 8 for speedup);
5 set $S_{opt} = S(\mathbf{X}_{xv}\mathbf{X}_{vy})$ by Eq. 5 (or Eq. 7, Eq. 8);
6 **if** $S_{org} < S_{opt}$ **then**
7 $\mathbf{X}_{xy} \leftarrow \mathbf{X}_{xv}\mathbf{X}_{vy}$;
8 Set consistency weight to $\lambda > 0$ and run Line 2-7 again;
Output: optimized matching $\mathbb{X}$.

where $S\left(\mathbf{X}_{ij}, \mathbb{X}\right) = \overbrace{(1-\lambda)J\left(\mathbf{X}_{ij}\right)}^{\text{affinity score}} + \overbrace{\lambda C_p\left(\mathbf{X}_{ij}, \mathbb{X}\right)}^{\text{pairwise consistency}}$. In practice we use the pc approximated version $S_{pc}^{\mathbb{X}}\left(\mathbf{X}_{ij}, \mathbf{X}_{jk}\right) = (1-\lambda)J\left(\mathbf{X}_{ij}\mathbf{X}_{jk}\right) + \lambda\sqrt{C_p\left(\mathbf{X}_{ij}, \mathbb{X}\right) C_p\left(\mathbf{X}_{jk}, \mathbb{X}\right)}$. In this way we don't need to calculate pairwise consistency of the multiplied matrix again, but just multiply their original pairwise consistency value.

There are two rounds of updating $\mathbf{X}$, with each round traversing all graphs. In the first round λ is set to 0 for affinity based boosting. In the second round $\lambda = 0.3$. Similar as acceleration of above, each pair of graphs are computated parallelly.

```python
def mgm_floyd(X, K, num_graph, num_node):
    """
    :param X: matching results, (num_graph, num_graph, num_node, num_node)
    :param K: affinity matrix, (num_graph, num_graph, num_node^2, num_node^2)
    :param num_graph: number of graph, int
    :param num_node: number of node, int
    :return: matching results, (num_graph, num_graph, num_node, num_node)
    """
    for k in range(num_graph):
        Xopt = np.matmul(X[:, None, k], X[None, k, :])
        Sorg = cal_affinity_score(X, K)
        Sopt = cal_affinity_score(Xopt, K)
        update = (Sopt > Sorg)[:, :, None, None]
        X = update * Xopt + (1 - update) * X

    for k in range(num_graph):
        pairwise_consistency = cal_pairwise_consistency(X)
        Xopt = np.matmul(X[:, None, k], X[None, k, :])
        Sorg = (1 - LAMBDA) * cal_affinity_score(X, K) + LAMBDA * pairwise_consistency
        Sopt = (1 - LAMBDA) * cal_affinity_score(Xopt, K) + LAMBDA * np.sqrt(  # sqrt
            np.matmul(pairwise_consistency[:, k][:, None], pairwise_consistency[k, :][
        update = (Sopt > Sorg)[:, :, None, None]
        X = update * Xopt + (1 - update) * X

    return X
```

- **MGM-SPFA**

MGM-SPFA is based on SPFA, a single-source shortest path algorithm. It helps solve online multiple graph matching, which aims at matching the arriving graph G_N to $N-1$ previous graphs which have already been matched. Two constraints added: force termination when number of updated nodes reaches m^2, and pairwise consistency is updated every time two nodes get updated.

Algorithm 2: MGM-SPFA (Online MGM)

Input: Affinity matrix $\{\mathbf{K}_{ij}\}_{i,j=1}^{N}$, λ, initial matching $\mathbb{X}$.

for *each newly arriving graph* G_N **do**
  use two-graph solver to obtain $\mathbb{X}_{Ni}^{(0)}$ for G_N and others;
  initialize the graph queue $\mathcal{Q} = \{G_1, G_2 \cdots, G_{N-1}\}$;
  while $\mathcal{Q}$ *is not empty* **do**
    obtain G_x in $\mathcal{Q}$ and remove it;
    for *each graph* G_y **do**
      set $S_{org} = S(\mathbf{X}_{yN})$ by Eq. 5 (or Eq. 7, Eq. 8);
      set $S_{opt} = S(\mathbf{X}_{yx}\mathbf{X}_{xN})$ by Eq. 5 (or Eq. 7, Eq. 8);
      if $S_{org} < S_{opt}$ **then**
        $\mathbf{X}_{yN} \leftarrow \mathbf{X}_{yx}\mathbf{X}_{xN}$;
        add G_y into $\mathcal{Q}$;
  for *each pair of graphs* G_x, G_y *in* $\mathcal{H}\backslash\{G_N\}$ **do**
    set $S_{org} = S(\mathbf{X}_{xN})$ by Eq. 5 (or Eq. 7, Eq. 8);
    set $S_{opt} = S(\mathbf{X}_{xy}\mathbf{X}_{yN})$ by Eq. 5 (or Eq. 7, Eq. 8);
    if $S_{org} < S_{opt}$ **then**
      $\mathbf{X}_{xy} \leftarrow \mathbf{X}_{xN}\mathbf{X}_{Ny}$;

Output: optimized matching $\mathbb{X}$.

```python
def mgm_spfa(K, X, num_graph, num_node):
    """
    :param K: affinity matrix, (num_graph, num_graph, num_node^2, num_node^2)
    :param X: matching results, X[:-1, :-1] is the matching results obtained by last i
              X[num_graph,:] and X[:,num_graph] is obtained via two-graph matching sol
              graph is the new coming graph. (num_graph, num_graph, num_node, num_node
    :param num_graph: number of graph, int
    :param num_node: number of node, int
    :return: X, matching results, match graph_m to {graph_1, ... , graph_m-1)
    """
    q = queue.Queue()
    outnumber = 0
    [q.put(i) for i in range(num_graph - 1)]
    pairwise_consistency = cal_pairwise_consistency(X)
    while not q.empty():
        Gx = q.get()
        outnumber += 1

        Xopt = np.matmul(X[:, Gx][:, None], X[Gx, :][None, ...])  # X_opt[y,N]=X[y,x]·
        Sorg = (1 - LAMBDA) * cal_affinity_score(X, K) + LAMBDA * np.sqrt(pairwise_con
        Sopt = (1 - LAMBDA) * cal_affinity_score(Xopt, K) + LAMBDA * np.sqrt(
            np.matmul(pairwise_consistency[:, Gx][:, None], pairwise_consistency[Gx, :

        Sorg = Sorg[:, num_graph - 1, None, None]
        Sopt = Sopt[:, num_graph - 1, None, None] # only consider the new added one
        update = (Sopt > Sorg)
        update[Gx] = False # skip Gx the graph itself

        X[:, num_graph - 1] = update * Xopt[:, num_graph - 1] + (1 - update) * X[:, nu
        X[num_graph - 1, :] = update * Xopt[num_graph - 1, :].transpose((0, 2, 1)) + (
```

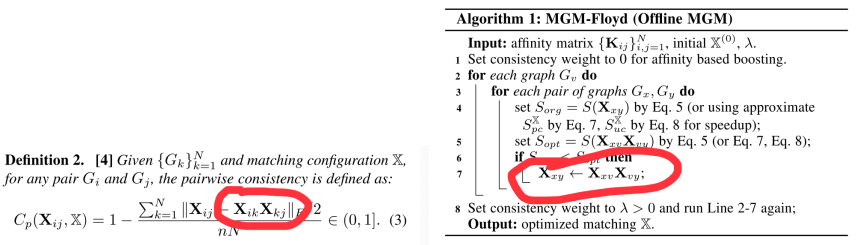
```
        [q.put(y) for y in range(num_graph) if update[y]]  # add Gy into Q

        if outnumber % 2 == 0:
            pairwise_consistency = cal_pairwise_consistency(X)
        if outnumber > num_graph ** 2:
            break

    Xopt = np.matmul(X[:, num_graph - 1][:, None], X[num_graph - 1, :][None, ...])  # 
    Sorg = (1 - LAMBDA) * cal_affinity_score(X, K) + LAMBDA * np.sqrt(pairwise_consist
    Sopt = (1 - LAMBDA) * cal_affinity_score(Xopt, K) + LAMBDA * np.sqrt(
        np.matmul(pairwise_consistency[:, num_graph - 1][:, None], pairwise_consistenc
    update = (Sopt > Sorg)[:, :, None, None]
    update[num_graph - 1] = False
    update[:, num_graph - 1] = False #Gx, Gy in H\GN
    X = update * Xopt + (1 - update) * X

    return X
```

Here some redundancy is made: all the pairwise `X_opt[i,j]` are calculated, while only `X_opt[i,N]` is needed. However this may not downshift time complexity a lot, since it's parallelized computation. What inpacts most is to add `Gy` back into queue `Q`. If this line is commented time and accuracy both pass the test. But with it, time test fails.

- **FAST-SPFA**

---

2. Question and Observation



Definition 2. [4] *Given $\{G_k\}_{k=1}^N$ and matching configuration $\mathbb{X}$, for any pair G_i and G_j, the pairwise consistency is defined as:*

$$C_p(\mathbf{X}_{ij}, \mathbb{X}) = 1 - \frac{\sum_{k=1}^{N} \|\mathbf{X}_{ij} - \mathbf{X}_{ik}\mathbf{X}_{kj}\|_F^2}{nN} \in (0, 1]. \quad (3)$$

Algorithm 1: MGM-Floyd (Offline MGM)

Input: affinity matrix $\{\mathbf{K}_{ij}\}_{i,j=1}^N$, initial $\mathbb{X}^{(0)}$, λ.
1 Set consistency weight to 0 for affinity based boosting.
2 **for** *each graph* G_v **do**
3 **for** *each pair of graphs* G_x, G_y **do**
4 set $S_{org} = S(\mathbf{X}_{xy})$ by Eq. 5 (or using approximate $S_{pc}^{\mathbb{X}}$ by Eq. 7, $S_{uc}^{\mathbb{X}}$ by Eq. 8 for speedup);
5 set $S_{opt} = S(\mathbf{X}_{xv}\mathbf{X}_{vy})$ by Eq. 5 (or Eq. 7, Eq. 8);
6 **if** $S_{org} < S_{opt}$ **then**
7 $\mathbf{X}_{xy} \leftarrow \mathbf{X}_{xv}\mathbf{X}_{vy}$;
8 Set consistency weight to $\lambda > 0$ and run Line 2-7 again;
Output: optimized matching $\mathbb{X}$.

We are curious about whether $X_{ik}X_{kj}$ should be pointwise or matrix multiplication, for each of the cases

Author Jiang Zetian has kindly answered our question:

> 矩阵乘法，举个例子吧
>
> Xiv =
> [[0, 1, 0]
> [1, 0, 0]
> [0, 0, 1]
>
> Xvj =
> [[0, 0, 1]
> [1, 0, 0]
> [0, 1, 0]
>
> 矩阵乘法乘起来表示 i 到 v 的匹配，再到 j 的匹配，结果还是一个置换矩阵
> XivXvj =
> [[1, 0, 0]
> [0, 0, 1]
> [0, 1, 0]]
>
> 但是 elementwise 的相乘的话，结果就不是置换矩阵了，这不符合匹配链组合的思路
> Xiv * Xvj =
> [[0, 0, 0]
> [1, 0, 0]
> [0, 0, 0]]
>
> 希望对你有帮助。

From the aspect of matching chain combination, it should be matmul here to maintain a permutation matrix. What we found interesting, however, is that if pointwise-mul is used in pairwise consistency computation rather than mat-mul, higher accuracy can be achieved (let alone speed). We would like to remain it here and research more thoroughly in the next big project.

---

3. Results Screenshots

Offline Floyd		Car	Motorbike	Face	Winebottle	Duck
Time Cost (s)	Required	4.384	4.227	4.220	4.339	4.209
	Ours	1.999	1.950	1.903	1.889	1.938
Accuracy (%)	Required	60.46	80.51	91.08	72.20	57.69
	Ours	82.56	92.27	95.34	91.24	76.93

Online SPFA		Car	Motorbike	Face	Winebottle	Duck
Time Cost (s)	Required	2.190	2.179	2.023	2.631	2.135
	Ours	0.828	0.815	0.857	0.821	0.850
Accuracy (%)	Required	63.32	83.43	91.41	75.23	59.05
	Ours	71.74	88.56	93.94	84.19	68.64

```
Test begin: test offline multi-graph matching on WILLOW-ObjectClass
**********************************************************************
Test offline multi-graph matching on class Car
iter: 0/5, init_acc: 0.4175, floyd_acc: 0.8509, init_time: 0.0004, floyd_time: 1.8562
iter: 1/5, init_acc: 0.4665, floyd_acc: 0.8179, init_time: 0.0003, floyd_time: 1.7210
iter: 2/5, init_acc: 0.4887, floyd_acc: 0.8116, init_time: 0.0003, floyd_time: 1.7932
iter: 3/5, init_acc: 0.5092, floyd_acc: 0.8597, init_time: 0.0003, floyd_time: 1.8903
iter: 4/5, init_acc: 0.4868, floyd_acc: 0.7880, init_time: 0.0003, floyd_time: 1.9977
Performance on class Car, accuracy: 0.8256, time: 1.8514
Test Car passed

**********************************************************************
Test offline multi-graph matching on class Motorbike
iter: 0/5, init_acc: 0.5986, floyd_acc: 0.9323, init_time: 0.0003, floyd_time: 2.0189
iter: 1/5, init_acc: 0.6135, floyd_acc: 0.9290, init_time: 0.0003, floyd_time: 1.9973
iter: 2/5, init_acc: 0.6373, floyd_acc: 0.8983, init_time: 0.0003, floyd_time: 2.0102
iter: 3/5, init_acc: 0.6483, floyd_acc: 0.9200, init_time: 0.0003, floyd_time: 2.0304
iter: 4/5, init_acc: 0.6778, floyd_acc: 0.9339, init_time: 0.0003, floyd_time: 1.8948
Performance on class Motorbike, accuracy: 0.9227, time: 1.9900
Test Motorbike passed

**********************************************************************
Test offline multi-graph matching on class Face
iter: 0/5, init_acc: 0.7328, floyd_acc: 0.9219, init_time: 0.0003, floyd_time: 1.8648
iter: 1/5, init_acc: 0.8497, floyd_acc: 0.9648, init_time: 0.0003, floyd_time: 1.9450
iter: 2/5, init_acc: 0.7840, floyd_acc: 0.9792, init_time: 0.0003, floyd_time: 2.0055
iter: 3/5, init_acc: 0.7845, floyd_acc: 0.9292, init_time: 0.0003, floyd_time: 1.9695
iter: 4/5, init_acc: 0.7832, floyd_acc: 0.9720, init_time: 0.0003, floyd_time: 2.0281
Performance on class Face, accuracy: 0.9534, time: 1.9623
Test Face passed
```

```
**********************************************************************
Test offline multi-graph matching on class Winebottle
iter: 0/5, init_acc: 0.5510, floyd_acc: 0.8682, init_time: 0.0003, floyd_time: 2.0348
iter: 1/5, init_acc: 0.5655, floyd_acc: 0.9139, init_time: 0.0003, floyd_time: 1.9724
iter: 2/5, init_acc: 0.4969, floyd_acc: 0.8969, init_time: 0.0003, floyd_time: 1.9542
iter: 3/5, init_acc: 0.5903, floyd_acc: 0.9444, init_time: 0.0003, floyd_time: 1.9734
iter: 4/5, init_acc: 0.5769, floyd_acc: 0.9387, init_time: 0.0003, floyd_time: 1.9785
Performance on class Winebottle, accuracy: 0.9124, time: 1.9824
Test Winebottle passed

**********************************************************************
Test offline multi-graph matching on class Duck
iter: 0/5, init_acc: 0.4632, floyd_acc: 0.8328, init_time: 0.0003, floyd_time: 1.9870
iter: 1/5, init_acc: 0.5264, floyd_acc: 0.8656, init_time: 0.0003, floyd_time: 1.8638
iter: 2/5, init_acc: 0.4319, floyd_acc: 0.6594, init_time: 0.0003, floyd_time: 2.0508
iter: 3/5, init_acc: 0.4743, floyd_acc: 0.7847, init_time: 0.0003, floyd_time: 2.0208
iter: 4/5, init_acc: 0.4156, floyd_acc: 0.7038, init_time: 0.0003, floyd_time: 2.0019
Performance on class Duck, accuracy: 0.7693, time: 1.9846
Test Duck passed
```

Offline test using MGM-floyd passed

```
Test begin: test offline multi-graph matching on WILLO
**********************************************************************
Test on class Car
iter: 0/5
number of graphs: 17, accuracy: 0.6883, time: 0.2724
number of graphs: 18, accuracy: 0.6446, time: 0.3308
number of graphs: 19, accuracy: 0.6636, time: 0.4302
number of graphs: 20, accuracy: 0.6493, time: 0.4631
number of graphs: 21, accuracy: 0.6520, time: 0.6011
number of graphs: 22, accuracy: 0.6753, time: 0.6460
number of graphs: 23, accuracy: 0.6810, time: 0.7859
number of graphs: 24, accuracy: 0.6747, time: 0.8439

iter: 1/5
number of graphs: 17, accuracy: 0.7461, time: 0.2533
number of graphs: 18, accuracy: 0.7208, time: 0.3020
number of graphs: 19, accuracy: 0.7315, time: 0.3731
number of graphs: 20, accuracy: 0.7380, time: 0.4697
number of graphs: 21, accuracy: 0.7428, time: 0.4957
number of graphs: 22, accuracy: 0.7444, time: 0.6272
number of graphs: 23, accuracy: 0.7417, time: 0.6451
number of graphs: 24, accuracy: 0.7174, time: 0.8191

iter: 2/5
number of graphs: 17, accuracy: 0.6887, time: 0.2489
number of graphs: 18, accuracy: 0.6938, time: 0.3143
number of graphs: 19, accuracy: 0.7037, time: 0.3909
number of graphs: 20, accuracy: 0.6961, time: 0.4042
number of graphs: 21, accuracy: 0.7067, time: 0.5114
number of graphs: 22, accuracy: 0.7063, time: 0.5989
number of graphs: 23, accuracy: 0.7174, time: 0.7307
number of graphs: 24, accuracy: 0.7136, time: 0.8197

iter: 3/5
number of graphs: 17, accuracy: 0.7711, time: 0.2515
number of graphs: 18, accuracy: 0.7647, time: 0.3001
number of graphs: 19, accuracy: 0.7577, time: 0.3408
number of graphs: 20, accuracy: 0.7568, time: 0.4355
number of graphs: 21, accuracy: 0.7400, time: 0.5071
number of graphs: 22, accuracy: 0.7302, time: 0.6153
number of graphs: 23, accuracy: 0.7372, time: 0.7248
number of graphs: 24, accuracy: 0.7510, time: 0.8382

iter: 4/5
number of graphs: 17, accuracy: 0.7977, time: 0.2690
number of graphs: 18, accuracy: 0.7976, time: 0.2930
number of graphs: 19, accuracy: 0.7951, time: 0.3445
number of graphs: 20, accuracy: 0.7388, time: 0.4755
number of graphs: 21, accuracy: 0.7300, time: 0.4922
number of graphs: 22, accuracy: 0.7252, time: 0.6218
number of graphs: 23, accuracy: 0.7312, time: 0.6956
number of graphs: 24, accuracy: 0.7301, time: 0.9451

Overall performance on class Car
number of graphs: 17, accuracy: 0.7384, time: 0.2590
number of graphs: 18, accuracy: 0.7243, time: 0.3080
number of graphs: 19, accuracy: 0.7303, time: 0.3759
number of graphs: 20, accuracy: 0.7158, time: 0.4496
number of graphs: 21, accuracy: 0.7143, time: 0.5215
number of graphs: 22, accuracy: 0.7163, time: 0.6218
number of graphs: 23, accuracy: 0.7217, time: 0.7164
number of graphs: 24, accuracy: 0.7174, time: 0.8532
Online Test, mode mgm-spfa, class Car passed
```

```
**********************************************************************
Test on class Motorbike
iter: 0/5
number of graphs: 17, accuracy: 0.8512, time: 0.2639
number of graphs: 18, accuracy: 0.8426, time: 0.3407
number of graphs: 19, accuracy: 0.8534, time: 0.3689
number of graphs: 20, accuracy: 0.8582, time: 0.4751
number of graphs: 21, accuracy: 0.8542, time: 0.5176
number of graphs: 22, accuracy: 0.8628, time: 0.6459
number of graphs: 23, accuracy: 0.8653, time: 0.7878
number of graphs: 24, accuracy: 0.8820, time: 0.9172

iter: 1/5
number of graphs: 17, accuracy: 0.8570, time: 0.2578
number of graphs: 18, accuracy: 0.8522, time: 0.3422
number of graphs: 19, accuracy: 0.8750, time: 0.3989
number of graphs: 20, accuracy: 0.8651, time: 0.4830
number of graphs: 21, accuracy: 0.8842, time: 0.5687
number of graphs: 22, accuracy: 0.8839, time: 0.6424
number of graphs: 23, accuracy: 0.8893, time: 0.7754
number of graphs: 24, accuracy: 0.9047, time: 0.9231

iter: 2/5
number of graphs: 17, accuracy: 0.8191, time: 0.2376
number of graphs: 18, accuracy: 0.8145, time: 0.2892
number of graphs: 19, accuracy: 0.8148, time: 0.3540
number of graphs: 20, accuracy: 0.8335, time: 0.4970
number of graphs: 21, accuracy: 0.8478, time: 0.5199
number of graphs: 22, accuracy: 0.8533, time: 0.6376
number of graphs: 23, accuracy: 0.8614, time: 0.6750
number of graphs: 24, accuracy: 0.8614, time: 0.8811

iter: 3/5
number of graphs: 17, accuracy: 0.8898, time: 0.2588
number of graphs: 18, accuracy: 0.8796, time: 0.2935
number of graphs: 19, accuracy: 0.8867, time: 0.3709
number of graphs: 20, accuracy: 0.8778, time: 0.4354
number of graphs: 21, accuracy: 0.8695, time: 0.5557
number of graphs: 22, accuracy: 0.8596, time: 0.6957
number of graphs: 23, accuracy: 0.8645, time: 0.8356
number of graphs: 24, accuracy: 0.8554, time: 1.0199

iter: 4/5
number of graphs: 17, accuracy: 0.9383, time: 0.2938
number of graphs: 18, accuracy: 0.9370, time: 0.3887
number of graphs: 19, accuracy: 0.9312, time: 0.3866
number of graphs: 20, accuracy: 0.9211, time: 0.4906
number of graphs: 21, accuracy: 0.9265, time: 0.5710
number of graphs: 22, accuracy: 0.9188, time: 0.6359
number of graphs: 23, accuracy: 0.9180, time: 0.7388
number of graphs: 24, accuracy: 0.9242, time: 0.9171

Overall performance on class Motorbike
number of graphs: 17, accuracy: 0.8711, time: 0.2624
number of graphs: 18, accuracy: 0.8652, time: 0.3309
number of graphs: 19, accuracy: 0.8722, time: 0.3759
number of graphs: 20, accuracy: 0.8711, time: 0.4762
number of graphs: 21, accuracy: 0.8764, time: 0.5466
number of graphs: 22, accuracy: 0.8757, time: 0.6515
number of graphs: 23, accuracy: 0.8797, time: 0.7625
number of graphs: 24, accuracy: 0.8856, time: 0.9317
Online Test, mode mgm-spfa, class Motorbike passed
```

```
**********************************************************************
Test on class Face
iter: 0/5
number of graphs: 17, accuracy: 0.9121, time: 0.2628
number of graphs: 18, accuracy: 0.9097, time: 0.3237
number of graphs: 19, accuracy: 0.9052, time: 0.3815
number of graphs: 20, accuracy: 0.9086, time: 0.4644
number of graphs: 21, accuracy: 0.9093, time: 0.5910
number of graphs: 22, accuracy: 0.9052, time: 0.6354
number of graphs: 23, accuracy: 0.9079, time: 0.7256
number of graphs: 24, accuracy: 0.9043, time: 0.9210

iter: 1/5
number of graphs: 17, accuracy: 0.9953, time: 0.2717
number of graphs: 18, accuracy: 0.9827, time: 0.3796
number of graphs: 19, accuracy: 0.9722, time: 0.4282
number of graphs: 20, accuracy: 0.9654, time: 0.4976
number of graphs: 21, accuracy: 0.9553, time: 0.5721
number of graphs: 22, accuracy: 0.9537, time: 0.7127
number of graphs: 23, accuracy: 0.9450, time: 0.7920
number of graphs: 24, accuracy: 0.9480, time: 0.9659

iter: 2/5
number of graphs: 17, accuracy: 0.9859, time: 0.2817
number of graphs: 18, accuracy: 0.9875, time: 0.3212
number of graphs: 19, accuracy: 0.9849, time: 0.3683
number of graphs: 20, accuracy: 0.9878, time: 0.4819
number of graphs: 21, accuracy: 0.9780, time: 0.5344
number of graphs: 22, accuracy: 0.9710, time: 0.6406
number of graphs: 23, accuracy: 0.9702, time: 0.7282
number of graphs: 24, accuracy: 0.9711, time: 0.8697

iter: 3/5
number of graphs: 17, accuracy: 0.9387, time: 0.2669
number of graphs: 18, accuracy: 0.9546, time: 0.2990
number of graphs: 19, accuracy: 0.9398, time: 0.3629
number of graphs: 20, accuracy: 0.9440, time: 0.4528
number of graphs: 21, accuracy: 0.9340, time: 0.4943
number of graphs: 22, accuracy: 0.9486, time: 0.6161
number of graphs: 23, accuracy: 0.9446, time: 0.6873
number of graphs: 24, accuracy: 0.9505, time: 0.8592

iter: 4/5
number of graphs: 17, accuracy: 0.9508, time: 0.2404
number of graphs: 18, accuracy: 0.9277, time: 0.3000
number of graphs: 19, accuracy: 0.9364, time: 0.3503
number of graphs: 20, accuracy: 0.9327, time: 0.4804
number of graphs: 21, accuracy: 0.9353, time: 0.5006
number of graphs: 22, accuracy: 0.9354, time: 0.5893
number of graphs: 23, accuracy: 0.9293, time: 0.6983
number of graphs: 24, accuracy: 0.9229, time: 0.8313

Overall performance on class Face
number of graphs: 17, accuracy: 0.9566, time: 0.2647
number of graphs: 18, accuracy: 0.9484, time: 0.3247
number of graphs: 19, accuracy: 0.9477, time: 0.3782
number of graphs: 20, accuracy: 0.9477, time: 0.4754
number of graphs: 21, accuracy: 0.9424, time: 0.5385
number of graphs: 22, accuracy: 0.9412, time: 0.6388
number of graphs: 23, accuracy: 0.9394, time: 0.7263
number of graphs: 24, accuracy: 0.9394, time: 0.8894
Online Test, mode mgm-spfa, class Face passed
```

```
**********************************************************************
Test on class Winebottle
iter: 0/5
number of graphs: 17, accuracy: 0.8238, time: 0.2545
number of graphs: 18, accuracy: 0.8270, time: 0.3247
number of graphs: 19, accuracy: 0.8509, time: 0.3861
number of graphs: 20, accuracy: 0.8618, time: 0.4804
number of graphs: 21, accuracy: 0.8518, time: 0.5652
number of graphs: 22, accuracy: 0.8404, time: 0.6358
number of graphs: 23, accuracy: 0.8295, time: 0.7776
number of graphs: 24, accuracy: 0.8163, time: 0.8838

iter: 1/5
number of graphs: 17, accuracy: 0.8605, time: 0.2990
number of graphs: 18, accuracy: 0.8633, time: 0.3389
number of graphs: 19, accuracy: 0.8738, time: 0.3852
number of graphs: 20, accuracy: 0.8762, time: 0.5166
number of graphs: 21, accuracy: 0.8720, time: 0.5359
number of graphs: 22, accuracy: 0.8748, time: 0.6889
number of graphs: 23, accuracy: 0.8847, time: 0.7509
number of graphs: 24, accuracy: 0.8834, time: 0.9178

iter: 2/5
number of graphs: 17, accuracy: 0.8039, time: 0.2668
number of graphs: 18, accuracy: 0.8021, time: 0.3394
number of graphs: 19, accuracy: 0.8228, time: 0.3808
number of graphs: 20, accuracy: 0.8122, time: 0.4864
number of graphs: 21, accuracy: 0.8037, time: 0.5485
number of graphs: 22, accuracy: 0.7941, time: 0.6483
number of graphs: 23, accuracy: 0.8112, time: 0.7231
number of graphs: 24, accuracy: 0.8195, time: 0.8789

iter: 3/5
number of graphs: 17, accuracy: 0.8355, time: 0.2445
number of graphs: 18, accuracy: 0.8453, time: 0.2980
number of graphs: 19, accuracy: 0.8398, time: 0.3778
number of graphs: 20, accuracy: 0.8471, time: 0.4362
number of graphs: 21, accuracy: 0.8635, time: 0.4941
number of graphs: 22, accuracy: 0.8023, time: 0.6241
number of graphs: 23, accuracy: 0.8143, time: 0.6792
number of graphs: 24, accuracy: 0.8293, time: 0.8728

iter: 4/5
number of graphs: 17, accuracy: 0.8246, time: 0.2330
number of graphs: 18, accuracy: 0.8277, time: 0.2964
number of graphs: 19, accuracy: 0.8327, time: 0.3483
number of graphs: 20, accuracy: 0.8407, time: 0.4789
number of graphs: 21, accuracy: 0.8343, time: 0.4961
number of graphs: 22, accuracy: 0.8558, time: 0.5834
number of graphs: 23, accuracy: 0.8560, time: 0.6990
number of graphs: 24, accuracy: 0.8612, time: 0.8469

Overall performance on class Winebottle
number of graphs: 17, accuracy: 0.8297, time: 0.2596
number of graphs: 18, accuracy: 0.8331, time: 0.3195
number of graphs: 19, accuracy: 0.8440, time: 0.3757
number of graphs: 20, accuracy: 0.8476, time: 0.4797
number of graphs: 21, accuracy: 0.8450, time: 0.5279
number of graphs: 22, accuracy: 0.8335, time: 0.6361
number of graphs: 23, accuracy: 0.8391, time: 0.7260
number of graphs: 24, accuracy: 0.8419, time: 0.8800
Online Test, mode mgm-spfa, class Winebottle passed
```

```
**********************************************************************
Test on class Duck
iter: 0/5
number of graphs: 17, accuracy: 0.7996, time: 0.2784
number of graphs: 18, accuracy: 0.7720, time: 0.3660
number of graphs: 19, accuracy: 0.7549, time: 0.4502
number of graphs: 20, accuracy: 0.7535, time: 0.5015
number of graphs: 21, accuracy: 0.6983, time: 0.6191
number of graphs: 22, accuracy: 0.7125, time: 0.6812
number of graphs: 23, accuracy: 0.6640, time: 0.8105
number of graphs: 24, accuracy: 0.6809, time: 1.0055

iter: 1/5
number of graphs: 17, accuracy: 0.7809, time: 0.2605
number of graphs: 18, accuracy: 0.7927, time: 0.3496
number of graphs: 19, accuracy: 0.7830, time: 0.4398
number of graphs: 20, accuracy: 0.7792, time: 0.4782
number of graphs: 21, accuracy: 0.7913, time: 0.5866
number of graphs: 22, accuracy: 0.7971, time: 0.6779
number of graphs: 23, accuracy: 0.8085, time: 0.7797
number of graphs: 24, accuracy: 0.7998, time: 0.9272

iter: 2/5
number of graphs: 17, accuracy: 0.6742, time: 0.2433
number of graphs: 18, accuracy: 0.6408, time: 0.3009
number of graphs: 19, accuracy: 0.5917, time: 0.3881
number of graphs: 20, accuracy: 0.6025, time: 0.4208
number of graphs: 21, accuracy: 0.6150, time: 0.5359
number of graphs: 22, accuracy: 0.6088, time: 0.5969
number of graphs: 23, accuracy: 0.6233, time: 0.7013
number of graphs: 24, accuracy: 0.6524, time: 0.8466

iter: 3/5
number of graphs: 17, accuracy: 0.7082, time: 0.2568
number of graphs: 18, accuracy: 0.6554, time: 0.3216
number of graphs: 19, accuracy: 0.6188, time: 0.3680
number of graphs: 20, accuracy: 0.6620, time: 0.5013
number of graphs: 21, accuracy: 0.6793, time: 0.5253
number of graphs: 22, accuracy: 0.6812, time: 0.6410
number of graphs: 23, accuracy: 0.6717, time: 0.7388
number of graphs: 24, accuracy: 0.6894, time: 0.8628

iter: 4/5
number of graphs: 17, accuracy: 0.7008, time: 0.2238
number of graphs: 18, accuracy: 0.6436, time: 0.3252
number of graphs: 19, accuracy: 0.6670, time: 0.3548
number of graphs: 20, accuracy: 0.6748, time: 0.3782
number of graphs: 21, accuracy: 0.6330, time: 0.5111
number of graphs: 22, accuracy: 0.6073, time: 0.5845
number of graphs: 23, accuracy: 0.6107, time: 0.6890
number of graphs: 24, accuracy: 0.6093, time: 0.8273

Overall performance on class Duck
number of graphs: 17, accuracy: 0.7327, time: 0.2525
number of graphs: 18, accuracy: 0.7009, time: 0.3326
number of graphs: 19, accuracy: 0.6831, time: 0.4002
number of graphs: 20, accuracy: 0.6944, time: 0.4560
number of graphs: 21, accuracy: 0.6834, time: 0.5556
number of graphs: 22, accuracy: 0.6814, time: 0.6363
number of graphs: 23, accuracy: 0.6757, time: 0.7438
number of graphs: 24, accuracy: 0.6864, time: 0.8939
Online Test, mode mgm-spfa, class Duck passed
```

Online test using MGM-SPFA passed