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:

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

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, 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) / X0)
    return normalized_affinity_score
```

Note that affinity score is normalized to range `(0,1]` to be consistent with pairwise consistency. We use the normalization factor `X0` to be the maximal affinity score of the raw input X, as proposed in CAO.

- **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:

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

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 $\|\mathbf{X}_{ij} - \mathbf{X}_{ik}\mathbf{X}_{kj}\|_F$ 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 $\mathbb{X}^{(0)}$, $\lambda$.
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]
        for i in range(num_graph):
          update[i, i] = False
        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 pc for approximate
            np.matmul(pairwise_consistency[:, k][:, None], \
            pairwise_consistency[k, :][None, ...]))
        update = (Sopt > Sorg)[:, :, None, None]
        update[np.diag_indices(num_graph)] = False
        X = update * Xopt + (1 - update) * X

    return X
```

We find that when using update in the matrix form would bring undesired update of self-assignment (i.e. X_{ii}), which should always be unit matrix. So we do not update these entries.

- **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 .

Algorithm 2: MGM-SPFA (Online MGM)

```
Input: Affinity matrix $\{\mathbf{K}_{ij}\}_{i,j=1}^{N}$, $\lambda$, initial matching $\mathbb{X}$.
1  for each newly arriving graph $G_N$ do
2      use two-graph solver to obtain $\mathbb{X}_{Ni}^{(0)}$ for $G_N$ and others;
3      initialize the graph queue $\mathcal{Q} = \{G_1, G_2 \cdots, G_{N-1}\}$;
4      while $\mathcal{Q}$ is not empty do
5          obtain $G_x$ in $\mathcal{Q}$ and remove it;
6          for each graph $G_y$ do
7              set $S_{org} = S(\mathbf{X}_{yN})$ by Eq. 5 (or Eq. 7, Eq. 8);
8              set $S_{opt} = S(\mathbf{X}_{yx}\mathbf{X}_{xN})$ by Eq. 5 (or Eq. 7,
                 Eq. 8);
9              if $S_{org} < S_{opt}$ then
10                 $\mathbf{X}_{yN} \leftarrow \mathbf{X}_{yx}\mathbf{X}_{xN}$;
11                 add $G_y$ into $\mathcal{Q}$;
12     for each pair of graphs $G_x, G_y$ in $\mathcal{H} \backslash \{G_N\}$ do
13         set $S_{org} = S(\mathbf{X}_{xN})$ by Eq. 5 (or Eq. 7, Eq. 8);
14         set $S_{opt} = S(\mathbf{X}_{xy}\mathbf{X}_{yN})$ by Eq. 5 (or Eq. 7, Eq. 8);
15         if $S_{org} < S_{opt}$ then
16             $\mathbf{X}_{xy} \leftarrow \mathbf{X}_{xN}\mathbf{X}_{Ny}$;
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 = [i for i in range(num_graph - 1)]
    outnumber = 0
    while len(q) > 0:
        Gx = q[0]
        del q[0]
        outnumber += 1
        Xopt = np.matmul(X[:, Gx, None], X[Gx, None, :])  # X_opt[y,N]=X[y,x]·X[x,N]
        for y in range(num_graph - 1):
            if y == Gx:
                continue
            Sorg = (1 - LAMBDA_SPFA) * cal_affinity_score_single(X[y, -1], K[y, -1]) +
                   LAMBDA_SPFA * cal_pairwise_consistency_single(X, y, -1)
            Sopt = (1 - LAMBDA_SPFA) * cal_affinity_score_single(Xopt[y, -1], K[y, -1]
                   LAMBDA_SPFA * cal_pairwise_consistency_single(Xopt, y, -1)
            if Sorg < Sopt:
                X[y, -1] = Xopt[y, -1]
                if y not in q:
                    q.append(y)
        if outnumber > num_graph ** 2:
            break

    pairwise_consistency = cal_pairwise_consistency(X)
    Xopt = np.matmul(X[:, num_graph - 1][:, None], X[num_graph - 1, :][None, ...])  # 
    Sorg = (1 - LAMBDA_SPFA) * cal_affinity_score(X, K) + LAMBDA_SPFA * pairwise_consi
    Sopt = (1 - LAMBDA_SPFA) * cal_affinity_score(Xopt, K) + LAMBDA_SPFA * 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
    update[np.diag_indices(num_graph)] = False
    X = update * Xopt + (1 - update) * X

    return X
```

Here we use iteration instead of matrix operation to update the matches, because matrix operation has some redundancies: all the pairwise `X_opt[i,j]` are calculated, while only `X_opt[i,N]` is needed.

Through experiments, we find that using iteration is more efficient than matrix operation in mgm-spfa.

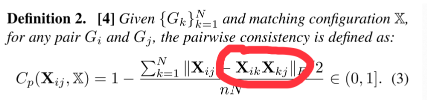
- **FAST-SPFA**

Fast-SPFA is based on MGM-SPFA. But instead of doing MGM-SPFA on all of the graphs, Fast-SPFA randomly partition the graphs into several clusters and doing MGM-SPFA-like update on each clusters instead. In this part, all clusters updates match information by the newly arrived graph. Then in the second part, all graph updates their match to each other (regardless of clusters) given the newly arrived graph.

```
Algorithm 3: FastSPFA (Online Fast MGM)
  Input: Affinity matrix $\{\mathbf{K}_{ij}\}_{i,j=1}^{N}$, $\lambda$, $C_{min}$, initial $\mathbb{X}$.
1 for each newly arriving graph $G_N$ do
2     calculate the cluster number $M = \max(1, N/C_{min})$.
3     partition $\mathcal{H}\backslash\{G_N\}$ into $M$ clusters randomly.
4     for each cluster $C_i$ do
5         generate the sub-supergraph $\mathcal{H}_i$ by $C_i \cup \{G_N\}$.
6         apply Line 1 to 16 in Alg. 2 to obtain the edges
            between $G_N$ and other vertices on $\mathcal{H}_i$: $\{\mathbb{X}_{Nh}\}_{h=1}^{N-1}$.
7     perform the same post-processing Line 12 to 16 in Alg. 2
        to optimize all pairwise matchings $\mathbf{X}_{uv}$ in $\mathcal{H}\backslash\{G_N\}$:
        i.e. use $\mathbf{X}_{uN}\mathbf{X}_{Nv}$ to update $\mathbf{X}_{uv}$ via $G_N$ for each pair.
  Output: optimized matching $\mathbb{X}$.
```

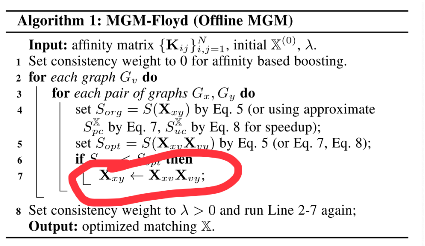
```python
def fast_spfa(K, X, num_graph, num_node):
    """
    :param K: affinity matrix, (num_graph, num_graph, num_node^2, num_node^2)
    :param X: matching results, X[:-1, :-1] is the matching results obtained by last i
              X[num_graph,:] and X[:,num_graph] is obtained via two-graph matching 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)
    """
    M = max(1, num_graph // CMIN)

    for k in range(num_graph):
        Xopt = np.matmul(X[:, k, None], X[k, None, :])
        Sorg = cal_affinity_score(X, K)
        Sopt = cal_affinity_score(Xopt, K)

        update = (Sopt > Sorg)[:, :, None, None]
        update[np.diag_indices(num_graph)] = False
        X = update * Xopt + (1 - update) * X

    for ci in range(M):
        if ci < M - 1:
            coord = [i for i in range(ci * CMIN, (ci + 1) * CMIN)]
            if num_graph - 1 not in coord:
                coord.append(num_graph - 1)
        else:
            coord = [i for i in range(ci * CMIN, num_graph)]
        Xc = X[coord, ...][:, coord]
        Kc = K[coord, ...][:, coord]

        q = [i for i in range(len(coord) - 1)]
        outnumber = 0
        while len(q) > 0:
            Gx = q[0]
            del q[0]
            outnumber += 1
            Xopt = np.matmul(Xc[:, Gx, None], Xc[Gx, None, :])  # X_opt[y,N]=X[y,x]·X[
            for y in range(len(coord) - 1):
                if y == Gx:
                    continue
                Sorg = (1 - LAMBDA_FAST) * cal_affinity_score_single(Xc[y, -1], Kc[y,
                    LAMBDA_FAST * cal_pairwise_consistency_single(Xc, y, -1)
                Sopt = (1 - LAMBDA_FAST) * cal_affinity_score_single(Xopt[y, -1], Kc[y
                    LAMBDA_FAST * cal_pairwise_consistency_single(Xopt, y, -1)
                if Sorg < Sopt:
                    Xc[y, -1] = Xopt[y, -1]
                    if y not in q:
                        q.append(y)
            if outnumber > CMIN ** 2:
                break

    pairwise_consistency = cal_pairwise_consistency(X)
    Xopt = np.matmul(X[:, num_graph - 1][:, None], X[num_graph - 1, :][None, ...])  # 
    Sorg = (1 - LAMBDA_FAST) * cal_affinity_score(X, K) + LAMBDA_FAST * pairwise_consi
    Sopt = (1 - LAMBDA_SPFA) * cal_affinity_score(Xopt, K) + LAMBDA_SPFA * 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
    update[np.diag_indices(num_graph)] = False
    X = update * Xopt + (1 - update) * X

    return X
```

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:*

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

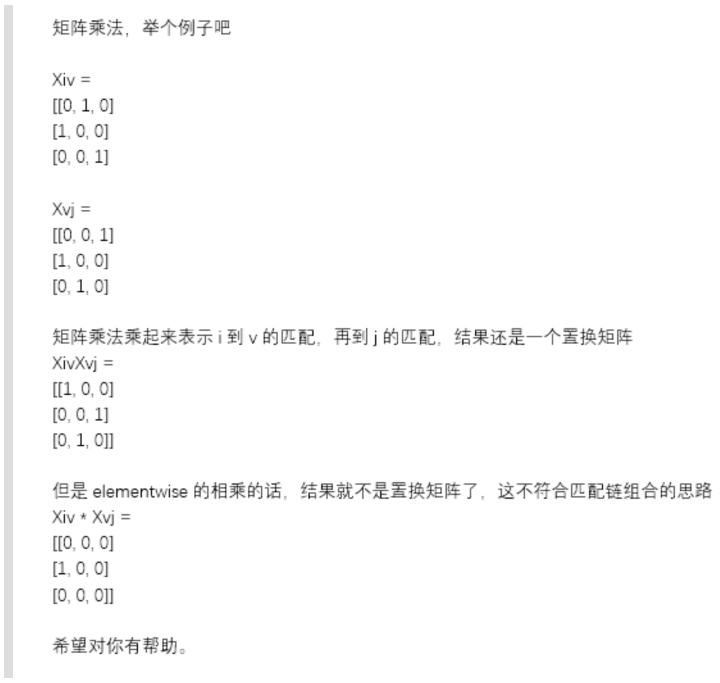


Algorithm 1: MGM-Floyd (Offline MGM)

```
  Input: affinity matrix $\{\mathbf{K}_{ij}\}_{i,j=1}^N$, initial $\mathbb{X}^{(0)}$, $\lambda$.
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).

Another observation is that, when adding affinity boost step before both MGM-SPFA and Fast-SPFA, the accuracy would be increased dramatically while using little extra time.

We would like to keep the above two here and research more thoroughly in the next big project.

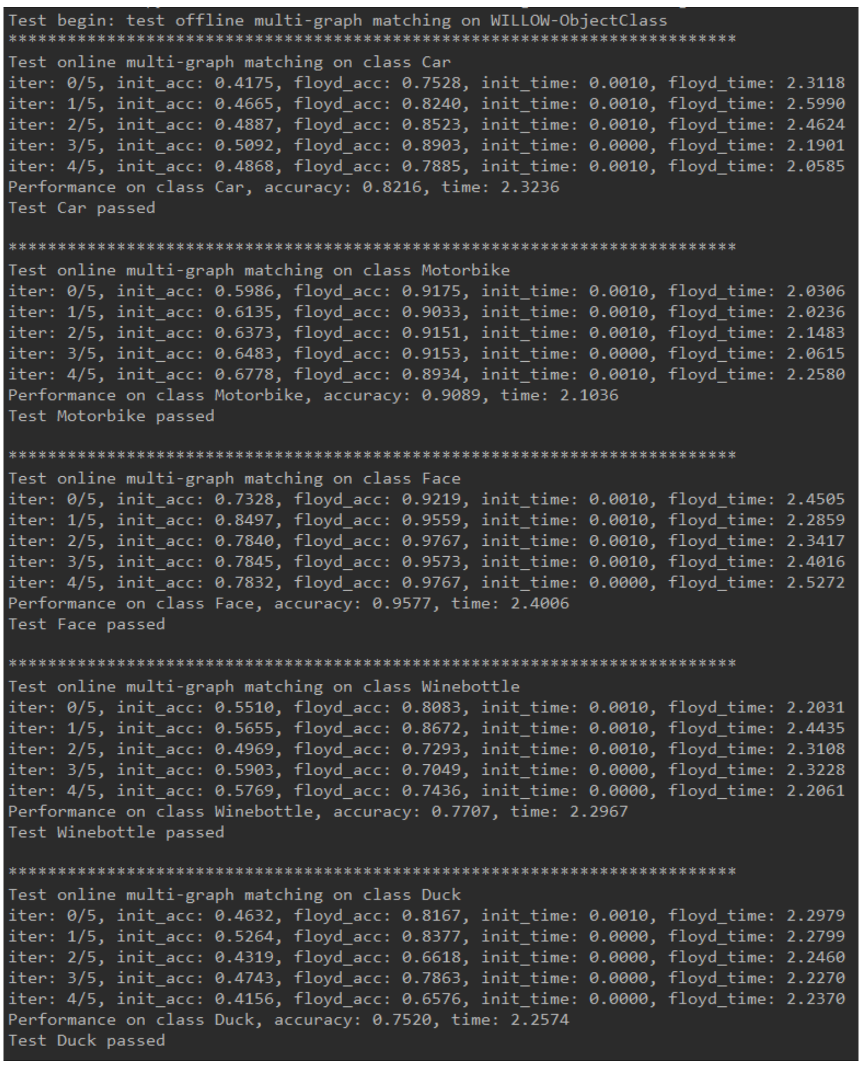
---

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	2.3236	2.1036	2.4006	2.2967	2.2574
Accuracy (%)	Required	60.46	80.51	91.08	72.20	57.69
	Ours	82.16	90.89	95.77	77.07	75.20

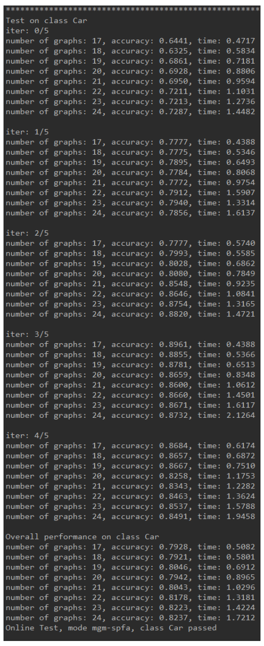
Online MGM-SPFA		**Car**	**Motorbike**	**Face**	**Winebottle**	**Duck**
Time Cost (s)	Required	2.190	2.179	2.023	2.631	2.135
	Ours	1.021	1.138	1.046	1.053	1.158
Accuracy (%)	Required	63.32	83.43	91.41	75.23	59.05
	Ours	80.65	91.78	96.46	78.48	77.89

Online Fast-SPFA		**Car**	**Motorbike**	**Face**	**Winebottle**	**Duck**
Time Cost (s)	Required	0.7323	0.7586	0.8077	0.7993	0.7591
	Ours	0.3661	0.3376	0.3507	0.3663	0.3285
Accuracy (%)	Required	61.87	82.90	91.45	73.57	57.82
	Ours	80.51	91.55	96.46	78.31	77.81

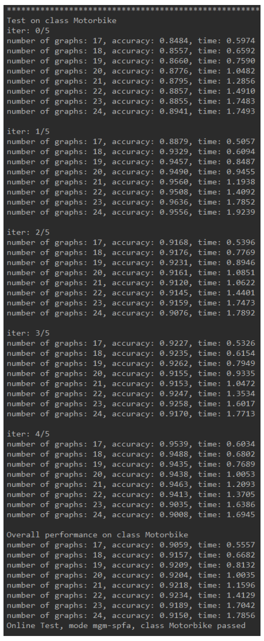


```
Test begin: test offline multi-graph matching on WILLOW-ObjectClass
**************************************************************************
Test online multi-graph matching on class Car
iter: 0/5, init_acc: 0.4175, floyd_acc: 0.7528, init_time: 0.0010, floyd_time: 2.3118
iter: 1/5, init_acc: 0.4665, floyd_acc: 0.8240, init_time: 0.0010, floyd_time: 2.5990
iter: 2/5, init_acc: 0.4887, floyd_acc: 0.8523, init_time: 0.0010, floyd_time: 2.4624
iter: 3/5, init_acc: 0.5092, floyd_acc: 0.8903, init_time: 0.0000, floyd_time: 2.1901
iter: 4/5, init_acc: 0.4868, floyd_acc: 0.7885, init_time: 0.0010, floyd_time: 2.0585
Performance on class Car, accuracy: 0.8216, time: 2.3236
Test Car passed

**************************************************************************
Test online multi-graph matching on class Motorbike
iter: 0/5, init_acc: 0.5986, floyd_acc: 0.9175, init_time: 0.0010, floyd_time: 2.0306
iter: 1/5, init_acc: 0.6135, floyd_acc: 0.9033, init_time: 0.0010, floyd_time: 2.0236
iter: 2/5, init_acc: 0.6373, floyd_acc: 0.9151, init_time: 0.0010, floyd_time: 2.1483
iter: 3/5, init_acc: 0.6483, floyd_acc: 0.9153, init_time: 0.0000, floyd_time: 2.0615
iter: 4/5, init_acc: 0.6778, floyd_acc: 0.8934, init_time: 0.0010, floyd_time: 2.2580
Performance on class Motorbike, accuracy: 0.9089, time: 2.1036
Test Motorbike passed

**************************************************************************
Test online multi-graph matching on class Face
iter: 0/5, init_acc: 0.7328, floyd_acc: 0.9219, init_time: 0.0010, floyd_time: 2.4505
iter: 1/5, init_acc: 0.8497, floyd_acc: 0.9559, init_time: 0.0010, floyd_time: 2.2859
iter: 2/5, init_acc: 0.7840, floyd_acc: 0.9767, init_time: 0.0010, floyd_time: 2.3417
iter: 3/5, init_acc: 0.7845, floyd_acc: 0.9573, init_time: 0.0010, floyd_time: 2.4016
iter: 4/5, init_acc: 0.7832, floyd_acc: 0.9767, init_time: 0.0000, floyd_time: 2.5272
Performance on class Face, accuracy: 0.9577, time: 2.4006
Test Face passed

**************************************************************************
Test online multi-graph matching on class Winebottle
iter: 0/5, init_acc: 0.5510, floyd_acc: 0.8083, init_time: 0.0010, floyd_time: 2.2031
iter: 1/5, init_acc: 0.5655, floyd_acc: 0.8672, init_time: 0.0010, floyd_time: 2.4435
iter: 2/5, init_acc: 0.4969, floyd_acc: 0.7293, init_time: 0.0010, floyd_time: 2.3108
iter: 3/5, init_acc: 0.5903, floyd_acc: 0.7049, init_time: 0.0000, floyd_time: 2.3228
iter: 4/5, init_acc: 0.5769, floyd_acc: 0.7436, init_time: 0.0000, floyd_time: 2.2061
Performance on class Winebottle, accuracy: 0.7707, time: 2.2967
Test Winebottle passed

**************************************************************************
Test online multi-graph matching on class Duck
iter: 0/5, init_acc: 0.4632, floyd_acc: 0.8167, init_time: 0.0010, floyd_time: 2.2979
iter: 1/5, init_acc: 0.5264, floyd_acc: 0.8377, init_time: 0.0000, floyd_time: 2.2799
iter: 2/5, init_acc: 0.4319, floyd_acc: 0.6618, init_time: 0.0000, floyd_time: 2.2460
iter: 3/5, init_acc: 0.4743, floyd_acc: 0.7863, init_time: 0.0000, floyd_time: 2.2270
iter: 4/5, init_acc: 0.4156, floyd_acc: 0.6576, init_time: 0.0000, floyd_time: 2.2370
Performance on class Duck, accuracy: 0.7520, time: 2.2574
Test Duck passed
```

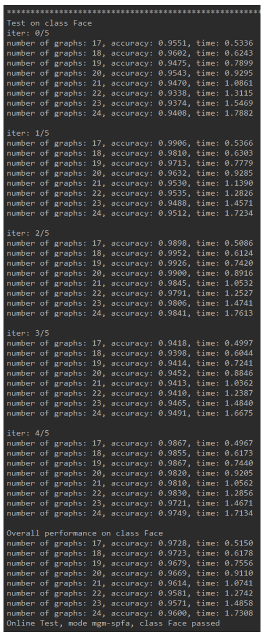
Offline test using MGM-floyd passed



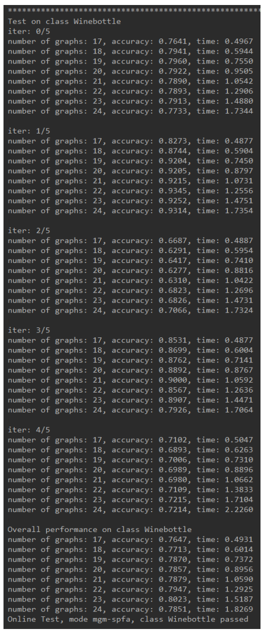
```
*****************************************************
Test on class Car
iter: 0/5
number of graphs: 17, accuracy: 0.6441, time: 0.4717
number of graphs: 18, accuracy: 0.6325, time: 0.5834
number of graphs: 19, accuracy: 0.6861, time: 0.7181
number of graphs: 20, accuracy: 0.6928, time: 0.8806
number of graphs: 21, accuracy: 0.6950, time: 0.9594
number of graphs: 22, accuracy: 0.7211, time: 1.1031
number of graphs: 23, accuracy: 0.7213, time: 1.2736
number of graphs: 24, accuracy: 0.7287, time: 1.4482

iter: 1/5
number of graphs: 17, accuracy: 0.7777, time: 0.4388
number of graphs: 18, accuracy: 0.7775, time: 0.5346
number of graphs: 19, accuracy: 0.7895, time: 0.6493
number of graphs: 20, accuracy: 0.7784, time: 0.8068
number of graphs: 21, accuracy: 0.7772, time: 0.9754
number of graphs: 22, accuracy: 0.7912, time: 1.5907
number of graphs: 23, accuracy: 0.7940, time: 1.3314
number of graphs: 24, accuracy: 0.7856, time: 1.6137

iter: 2/5
number of graphs: 17, accuracy: 0.7777, time: 0.5740
number of graphs: 18, accuracy: 0.7993, time: 0.5585
number of graphs: 19, accuracy: 0.8028, time: 0.6862
number of graphs: 20, accuracy: 0.8080, time: 0.7849
number of graphs: 21, accuracy: 0.8548, time: 0.9235
number of graphs: 22, accuracy: 0.8646, time: 1.0841
number of graphs: 23, accuracy: 0.8754, time: 1.3165
number of graphs: 24, accuracy: 0.8820, time: 1.4721

iter: 3/5
number of graphs: 17, accuracy: 0.8961, time: 0.4388
number of graphs: 18, accuracy: 0.8855, time: 0.5366
number of graphs: 19, accuracy: 0.8781, time: 0.6513
number of graphs: 20, accuracy: 0.8659, time: 0.8348
number of graphs: 21, accuracy: 0.8600, time: 1.0612
number of graphs: 22, accuracy: 0.8660, time: 1.4501
number of graphs: 23, accuracy: 0.8671, time: 1.6117
number of graphs: 24, accuracy: 0.8732, time: 2.1264

iter: 4/5
number of graphs: 17, accuracy: 0.8684, time: 0.6174
number of graphs: 18, accuracy: 0.8657, time: 0.6872
number of graphs: 19, accuracy: 0.8667, time: 0.7510
number of graphs: 20, accuracy: 0.8258, time: 1.1753
number of graphs: 21, accuracy: 0.8343, time: 1.2282
number of graphs: 22, accuracy: 0.8463, time: 1.3624
number of graphs: 23, accuracy: 0.8537, time: 1.5788
number of graphs: 24, accuracy: 0.8491, time: 1.9458

Overall performance on class Car
number of graphs: 17, accuracy: 0.7928, time: 0.5082
number of graphs: 18, accuracy: 0.7921, time: 0.5801
number of graphs: 19, accuracy: 0.8046, time: 0.6912
number of graphs: 20, accuracy: 0.7942, time: 0.8965
number of graphs: 21, accuracy: 0.8043, time: 1.0296
number of graphs: 22, accuracy: 0.8178, time: 1.3181
number of graphs: 23, accuracy: 0.8223, time: 1.4224
number of graphs: 24, accuracy: 0.8237, time: 1.7212
Online Test, mode mgm-spfa, class Car passed
```



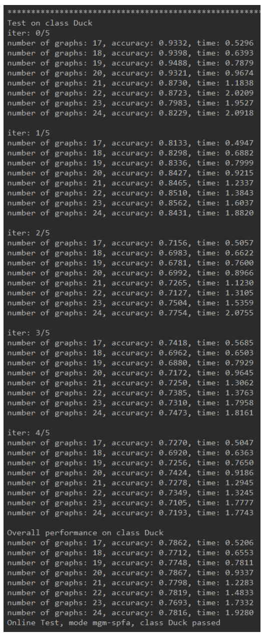
```
*****************************************************
Test on class Motorbike
iter: 0/5
number of graphs: 17, accuracy: 0.8484, time: 0.5974
number of graphs: 18, accuracy: 0.8557, time: 0.6592
number of graphs: 19, accuracy: 0.8660, time: 0.7590
number of graphs: 20, accuracy: 0.8776, time: 1.0482
number of graphs: 21, accuracy: 0.8795, time: 1.2856
number of graphs: 22, accuracy: 0.8857, time: 1.4910
number of graphs: 23, accuracy: 0.8855, time: 1.7483
number of graphs: 24, accuracy: 0.8941, time: 1.7493

iter: 1/5
number of graphs: 17, accuracy: 0.8879, time: 0.5057
number of graphs: 18, accuracy: 0.9329, time: 0.6094
number of graphs: 19, accuracy: 0.9457, time: 0.8487
number of graphs: 20, accuracy: 0.9490, time: 0.9455
number of graphs: 21, accuracy: 0.9560, time: 1.1938
number of graphs: 22, accuracy: 0.9508, time: 1.4092
number of graphs: 23, accuracy: 0.9636, time: 1.7852
number of graphs: 24, accuracy: 0.9556, time: 1.9239

iter: 2/5
number of graphs: 17, accuracy: 0.9168, time: 0.5396
number of graphs: 18, accuracy: 0.9176, time: 0.7769
number of graphs: 19, accuracy: 0.9231, time: 0.8946
number of graphs: 20, accuracy: 0.9161, time: 1.0851
number of graphs: 21, accuracy: 0.9120, time: 1.0622
number of graphs: 22, accuracy: 0.9145, time: 1.4401
number of graphs: 23, accuracy: 0.9159, time: 1.7473
number of graphs: 24, accuracy: 0.9076, time: 1.7892

iter: 3/5
number of graphs: 17, accuracy: 0.9227, time: 0.5326
number of graphs: 18, accuracy: 0.9235, time: 0.6154
number of graphs: 19, accuracy: 0.9262, time: 0.7949
number of graphs: 20, accuracy: 0.9155, time: 0.9335
number of graphs: 21, accuracy: 0.9153, time: 1.0472
number of graphs: 22, accuracy: 0.9247, time: 1.3534
number of graphs: 23, accuracy: 0.9258, time: 1.6017
number of graphs: 24, accuracy: 0.9170, time: 1.7713

iter: 4/5
number of graphs: 17, accuracy: 0.9539, time: 0.6034
number of graphs: 18, accuracy: 0.9488, time: 0.6802
number of graphs: 19, accuracy: 0.9435, time: 0.7689
number of graphs: 20, accuracy: 0.9438, time: 1.0053
number of graphs: 21, accuracy: 0.9463, time: 1.2093
number of graphs: 22, accuracy: 0.9413, time: 1.3705
number of graphs: 23, accuracy: 0.9035, time: 1.6386
number of graphs: 24, accuracy: 0.9008, time: 1.6945

Overall performance on class Motorbike
number of graphs: 17, accuracy: 0.9059, time: 0.5557
number of graphs: 18, accuracy: 0.9157, time: 0.6682
number of graphs: 19, accuracy: 0.9209, time: 0.8132
number of graphs: 20, accuracy: 0.9204, time: 1.0035
number of graphs: 21, accuracy: 0.9218, time: 1.1596
number of graphs: 22, accuracy: 0.9234, time: 1.4129
number of graphs: 23, accuracy: 0.9189, time: 1.7042
number of graphs: 24, accuracy: 0.9150, time: 1.7856
Online Test, mode mgm-spfa, class Motorbike passed
```



```
*****************************************************
Test on class Face
iter: 0/5
number of graphs: 17, accuracy: 0.9551, time: 0.5336
number of graphs: 18, accuracy: 0.9602, time: 0.6243
number of graphs: 19, accuracy: 0.9475, time: 0.7899
number of graphs: 20, accuracy: 0.9543, time: 0.9295
number of graphs: 21, accuracy: 0.9470, time: 1.0861
number of graphs: 22, accuracy: 0.9338, time: 1.3115
number of graphs: 23, accuracy: 0.9374, time: 1.5469
number of graphs: 24, accuracy: 0.9408, time: 1.7882

iter: 1/5
number of graphs: 17, accuracy: 0.9906, time: 0.5366
number of graphs: 18, accuracy: 0.9810, time: 0.6303
number of graphs: 19, accuracy: 0.9713, time: 0.7779
number of graphs: 20, accuracy: 0.9632, time: 0.9285
number of graphs: 21, accuracy: 0.9530, time: 1.1390
number of graphs: 22, accuracy: 0.9535, time: 1.2826
number of graphs: 23, accuracy: 0.9488, time: 1.4571
number of graphs: 24, accuracy: 0.9512, time: 1.7234

iter: 2/5
number of graphs: 17, accuracy: 0.9898, time: 0.5086
number of graphs: 18, accuracy: 0.9952, time: 0.6124
number of graphs: 19, accuracy: 0.9926, time: 0.7420
number of graphs: 20, accuracy: 0.9900, time: 0.8916
number of graphs: 21, accuracy: 0.9845, time: 1.0532
number of graphs: 22, accuracy: 0.9791, time: 1.2527
number of graphs: 23, accuracy: 0.9806, time: 1.4741
number of graphs: 24, accuracy: 0.9841, time: 1.7613

iter: 3/5
number of graphs: 17, accuracy: 0.9418, time: 0.4997
number of graphs: 18, accuracy: 0.9398, time: 0.6044
number of graphs: 19, accuracy: 0.9414, time: 0.7241
number of graphs: 20, accuracy: 0.9452, time: 0.8846
number of graphs: 21, accuracy: 0.9413, time: 1.0362
number of graphs: 22, accuracy: 0.9410, time: 1.2387
number of graphs: 23, accuracy: 0.9465, time: 1.4840
number of graphs: 24, accuracy: 0.9491, time: 1.6675

iter: 4/5
number of graphs: 17, accuracy: 0.9867, time: 0.4967
number of graphs: 18, accuracy: 0.9855, time: 0.6173
number of graphs: 19, accuracy: 0.9867, time: 0.7440
number of graphs: 20, accuracy: 0.9820, time: 0.9205
number of graphs: 21, accuracy: 0.9810, time: 1.0562
number of graphs: 22, accuracy: 0.9830, time: 1.2856
number of graphs: 23, accuracy: 0.9721, time: 1.4671
number of graphs: 24, accuracy: 0.9749, time: 1.7134

Overall performance on class Face
number of graphs: 17, accuracy: 0.9728, time: 0.5150
number of graphs: 18, accuracy: 0.9723, time: 0.6178
number of graphs: 19, accuracy: 0.9679, time: 0.7556
number of graphs: 20, accuracy: 0.9669, time: 0.9110
number of graphs: 21, accuracy: 0.9614, time: 1.0741
number of graphs: 22, accuracy: 0.9581, time: 1.2742
number of graphs: 23, accuracy: 0.9571, time: 1.4858
number of graphs: 24, accuracy: 0.9600, time: 1.7308
Online Test, mode mgm-spfa, class Face passed
```

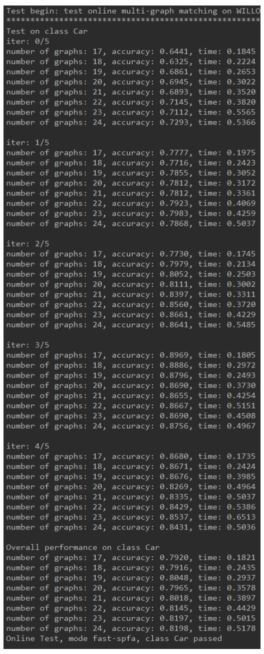


```
*****************************************************
Test on class Winebottle
iter: 0/5
number of graphs: 17, accuracy: 0.7641, time: 0.4967
number of graphs: 18, accuracy: 0.7941, time: 0.5944
number of graphs: 19, accuracy: 0.7960, time: 0.7550
number of graphs: 20, accuracy: 0.7922, time: 0.9505
number of graphs: 21, accuracy: 0.7890, time: 1.0542
number of graphs: 22, accuracy: 0.7893, time: 1.2906
number of graphs: 23, accuracy: 0.7913, time: 1.4880
number of graphs: 24, accuracy: 0.7733, time: 1.7344

iter: 1/5
number of graphs: 17, accuracy: 0.8273, time: 0.4877
number of graphs: 18, accuracy: 0.8744, time: 0.5904
number of graphs: 19, accuracy: 0.9204, time: 0.7450
number of graphs: 20, accuracy: 0.9205, time: 0.8797
number of graphs: 21, accuracy: 0.9215, time: 1.0731
number of graphs: 22, accuracy: 0.9345, time: 1.2556
number of graphs: 23, accuracy: 0.9252, time: 1.4751
number of graphs: 24, accuracy: 0.9314, time: 1.7354

iter: 2/5
number of graphs: 17, accuracy: 0.6687, time: 0.4887
number of graphs: 18, accuracy: 0.6291, time: 0.5954
number of graphs: 19, accuracy: 0.6417, time: 0.7410
number of graphs: 20, accuracy: 0.6277, time: 0.8816
number of graphs: 21, accuracy: 0.6310, time: 1.0422
number of graphs: 22, accuracy: 0.6823, time: 1.2696
number of graphs: 23, accuracy: 0.6826, time: 1.4731
number of graphs: 24, accuracy: 0.7066, time: 1.7324

iter: 3/5
number of graphs: 17, accuracy: 0.8531, time: 0.4877
number of graphs: 18, accuracy: 0.8699, time: 0.6004
number of graphs: 19, accuracy: 0.8762, time: 0.7141
number of graphs: 20, accuracy: 0.8892, time: 0.8767
number of graphs: 21, accuracy: 0.9000, time: 1.0592
number of graphs: 22, accuracy: 0.8567, time: 1.2636
number of graphs: 23, accuracy: 0.8907, time: 1.4471
number of graphs: 24, accuracy: 0.7926, time: 1.7064

iter: 4/5
number of graphs: 17, accuracy: 0.7102, time: 0.5047
number of graphs: 18, accuracy: 0.6893, time: 0.6263
number of graphs: 19, accuracy: 0.7006, time: 0.7310
number of graphs: 20, accuracy: 0.6989, time: 0.8896
number of graphs: 21, accuracy: 0.6980, time: 1.0662
number of graphs: 22, accuracy: 0.7109, time: 1.3833
number of graphs: 23, accuracy: 0.7215, time: 1.7104
number of graphs: 24, accuracy: 0.7214, time: 2.2260

Overall performance on class Winebottle
number of graphs: 17, accuracy: 0.7647, time: 0.4931
number of graphs: 18, accuracy: 0.7713, time: 0.6014
number of graphs: 19, accuracy: 0.7870, time: 0.7372
number of graphs: 20, accuracy: 0.7857, time: 0.8956
number of graphs: 21, accuracy: 0.7879, time: 1.0590
number of graphs: 22, accuracy: 0.7947, time: 1.2925
number of graphs: 23, accuracy: 0.8023, time: 1.5187
number of graphs: 24, accuracy: 0.7851, time: 1.8269
Online Test, mode mgm-spfa, class Winebottle passed
```

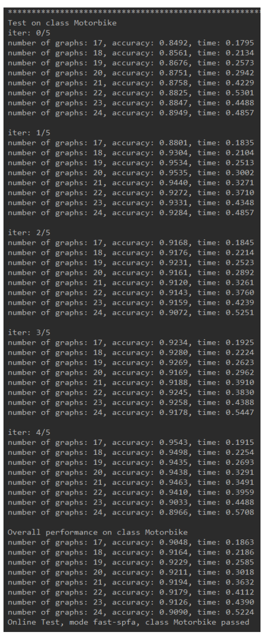


```
*****************************************************
Test on class Duck
iter: 0/5
number of graphs: 17, accuracy: 0.9332, time: 0.5296
number of graphs: 18, accuracy: 0.9398, time: 0.6393
number of graphs: 19, accuracy: 0.9488, time: 0.7879
number of graphs: 20, accuracy: 0.9321, time: 0.9674
number of graphs: 21, accuracy: 0.8730, time: 1.1838
number of graphs: 22, accuracy: 0.8723, time: 2.0209
number of graphs: 23, accuracy: 0.7983, time: 1.9527
number of graphs: 24, accuracy: 0.8229, time: 2.0918

iter: 1/5
number of graphs: 17, accuracy: 0.8133, time: 0.4947
number of graphs: 18, accuracy: 0.8298, time: 0.6882
number of graphs: 19, accuracy: 0.8336, time: 0.7999
number of graphs: 20, accuracy: 0.8427, time: 0.9215
number of graphs: 21, accuracy: 0.8465, time: 1.2337
number of graphs: 22, accuracy: 0.8510, time: 1.3843
number of graphs: 23, accuracy: 0.8562, time: 1.6037
number of graphs: 24, accuracy: 0.8431, time: 1.8820

iter: 2/5
number of graphs: 17, accuracy: 0.7156, time: 0.5057
number of graphs: 18, accuracy: 0.6983, time: 0.6622
number of graphs: 19, accuracy: 0.6781, time: 0.7600
number of graphs: 20, accuracy: 0.6992, time: 0.8966
number of graphs: 21, accuracy: 0.7265, time: 1.1230
number of graphs: 22, accuracy: 0.7127, time: 1.3105
number of graphs: 23, accuracy: 0.7504, time: 1.5359
number of graphs: 24, accuracy: 0.7754, time: 2.0755

iter: 3/5
number of graphs: 17, accuracy: 0.7418, time: 0.5685
number of graphs: 18, accuracy: 0.6962, time: 0.6503
number of graphs: 19, accuracy: 0.6880, time: 0.7929
number of graphs: 20, accuracy: 0.7172, time: 0.9645
number of graphs: 21, accuracy: 0.7250, time: 1.3062
number of graphs: 22, accuracy: 0.7385, time: 1.3763
number of graphs: 23, accuracy: 0.7310, time: 1.7958
number of graphs: 24, accuracy: 0.7473, time: 1.8161

iter: 4/5
number of graphs: 17, accuracy: 0.7270, time: 0.5047
number of graphs: 18, accuracy: 0.6920, time: 0.6363
number of graphs: 19, accuracy: 0.7256, time: 0.7650
number of graphs: 20, accuracy: 0.7424, time: 0.9186
number of graphs: 21, accuracy: 0.7278, time: 1.2945
number of graphs: 22, accuracy: 0.7349, time: 1.3245
number of graphs: 23, accuracy: 0.7105, time: 1.7777
number of graphs: 24, accuracy: 0.7193, time: 1.7743

Overall performance on class Duck
number of graphs: 17, accuracy: 0.7862, time: 0.5206
number of graphs: 18, accuracy: 0.7712, time: 0.6553
number of graphs: 19, accuracy: 0.7748, time: 0.7811
number of graphs: 20, accuracy: 0.7867, time: 0.9337
number of graphs: 21, accuracy: 0.7798, time: 1.2283
number of graphs: 22, accuracy: 0.7819, time: 1.4833
number of graphs: 23, accuracy: 0.7693, time: 1.7332
number of graphs: 24, accuracy: 0.7816, time: 1.9280
Online Test, mode mgm-spfa, class Duck passed
```

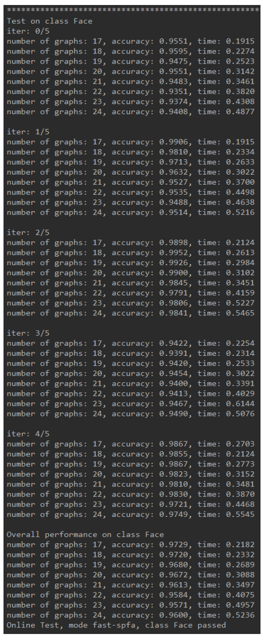
Online test using MGM-SPFA passed



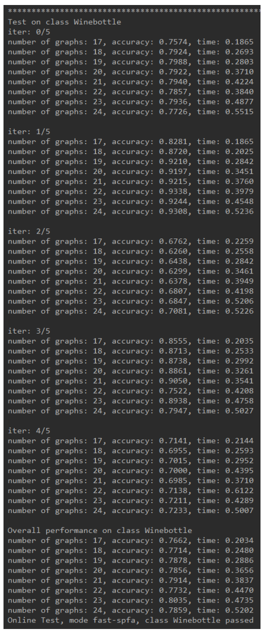
```
Test begin: test online multi-graph matching on WILLO
*****************************************************
Test on class Car
iter: 0/5
number of graphs: 17, accuracy: 0.6441, time: 0.1845
number of graphs: 18, accuracy: 0.6325, time: 0.2224
number of graphs: 19, accuracy: 0.6861, time: 0.2653
number of graphs: 20, accuracy: 0.6945, time: 0.3022
number of graphs: 21, accuracy: 0.6893, time: 0.3520
number of graphs: 22, accuracy: 0.7145, time: 0.3820
number of graphs: 23, accuracy: 0.7112, time: 0.5565
number of graphs: 24, accuracy: 0.7293, time: 0.5366

iter: 1/5
number of graphs: 17, accuracy: 0.7777, time: 0.1975
number of graphs: 18, accuracy: 0.7716, time: 0.2423
number of graphs: 19, accuracy: 0.7855, time: 0.3052
number of graphs: 20, accuracy: 0.7812, time: 0.3172
number of graphs: 21, accuracy: 0.7812, time: 0.3361
number of graphs: 22, accuracy: 0.7923, time: 0.4069
number of graphs: 23, accuracy: 0.7983, time: 0.4259
number of graphs: 24, accuracy: 0.7868, time: 0.5037

iter: 2/5
number of graphs: 17, accuracy: 0.7730, time: 0.1745
number of graphs: 18, accuracy: 0.7979, time: 0.2134
number of graphs: 19, accuracy: 0.8052, time: 0.2503
number of graphs: 20, accuracy: 0.8111, time: 0.3002
number of graphs: 21, accuracy: 0.8397, time: 0.3311
number of graphs: 22, accuracy: 0.8560, time: 0.3720
number of graphs: 23, accuracy: 0.8661, time: 0.4229
number of graphs: 24, accuracy: 0.8641, time: 0.5485

iter: 3/5
number of graphs: 17, accuracy: 0.8969, time: 0.1805
number of graphs: 18, accuracy: 0.8886, time: 0.2972
number of graphs: 19, accuracy: 0.8796, time: 0.2493
number of graphs: 20, accuracy: 0.8690, time: 0.3730
number of graphs: 21, accuracy: 0.8655, time: 0.4254
number of graphs: 22, accuracy: 0.8667, time: 0.5151
number of graphs: 23, accuracy: 0.8690, time: 0.4508
number of graphs: 24, accuracy: 0.8756, time: 0.4967

iter: 4/5
number of graphs: 17, accuracy: 0.8680, time: 0.1735
number of graphs: 18, accuracy: 0.8671, time: 0.2424
number of graphs: 19, accuracy: 0.8676, time: 0.3985
number of graphs: 20, accuracy: 0.8269, time: 0.4964
number of graphs: 21, accuracy: 0.8335, time: 0.5037
number of graphs: 22, accuracy: 0.8429, time: 0.5386
number of graphs: 23, accuracy: 0.8537, time: 0.6513
number of graphs: 24, accuracy: 0.8431, time: 0.5036

Overall performance on class Car
number of graphs: 17, accuracy: 0.7920, time: 0.1821
number of graphs: 18, accuracy: 0.7916, time: 0.2435
number of graphs: 19, accuracy: 0.8048, time: 0.2937
number of graphs: 20, accuracy: 0.7965, time: 0.3578
number of graphs: 21, accuracy: 0.8018, time: 0.3897
number of graphs: 22, accuracy: 0.8145, time: 0.4429
number of graphs: 23, accuracy: 0.8197, time: 0.5015
number of graphs: 24, accuracy: 0.8198, time: 0.5178
Online Test, mode fast-spfa, class Car passed
```



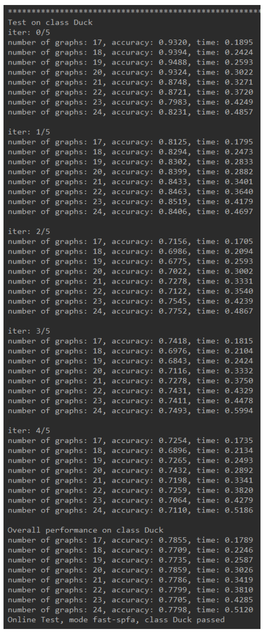
```
*****************************************************
Test on class Motorbike
iter: 0/5
number of graphs: 17, accuracy: 0.8492, time: 0.1795
number of graphs: 18, accuracy: 0.8561, time: 0.2134
number of graphs: 19, accuracy: 0.8676, time: 0.2573
number of graphs: 20, accuracy: 0.8751, time: 0.2942
number of graphs: 21, accuracy: 0.8758, time: 0.4229
number of graphs: 22, accuracy: 0.8825, time: 0.5301
number of graphs: 23, accuracy: 0.8847, time: 0.4488
number of graphs: 24, accuracy: 0.8949, time: 0.4857

iter: 1/5
number of graphs: 17, accuracy: 0.8801, time: 0.1835
number of graphs: 18, accuracy: 0.9304, time: 0.2104
number of graphs: 19, accuracy: 0.9534, time: 0.2513
number of graphs: 20, accuracy: 0.9535, time: 0.3002
number of graphs: 21, accuracy: 0.9440, time: 0.3271
number of graphs: 22, accuracy: 0.9272, time: 0.3710
number of graphs: 23, accuracy: 0.9331, time: 0.4348
number of graphs: 24, accuracy: 0.9284, time: 0.4857

iter: 2/5
number of graphs: 17, accuracy: 0.9168, time: 0.1845
number of graphs: 18, accuracy: 0.9176, time: 0.2214
number of graphs: 19, accuracy: 0.9231, time: 0.2523
number of graphs: 20, accuracy: 0.9161, time: 0.2892
number of graphs: 21, accuracy: 0.9120, time: 0.3261
number of graphs: 22, accuracy: 0.9143, time: 0.3760
number of graphs: 23, accuracy: 0.9159, time: 0.4239
number of graphs: 24, accuracy: 0.9072, time: 0.5251

iter: 3/5
number of graphs: 17, accuracy: 0.9234, time: 0.1925
number of graphs: 18, accuracy: 0.9280, time: 0.2224
number of graphs: 19, accuracy: 0.9269, time: 0.2623
number of graphs: 20, accuracy: 0.9169, time: 0.2962
number of graphs: 21, accuracy: 0.9188, time: 0.3910
number of graphs: 22, accuracy: 0.9245, time: 0.3830
number of graphs: 23, accuracy: 0.9258, time: 0.4388
number of graphs: 24, accuracy: 0.9178, time: 0.5447

iter: 4/5
number of graphs: 17, accuracy: 0.9543, time: 0.1915
number of graphs: 18, accuracy: 0.9498, time: 0.2254
number of graphs: 19, accuracy: 0.9435, time: 0.2693
number of graphs: 20, accuracy: 0.9438, time: 0.3291
number of graphs: 21, accuracy: 0.9463, time: 0.3491
number of graphs: 22, accuracy: 0.9410, time: 0.3959
number of graphs: 23, accuracy: 0.9033, time: 0.4488
number of graphs: 24, accuracy: 0.8966, time: 0.5708

Overall performance on class Motorbike
number of graphs: 17, accuracy: 0.9048, time: 0.1863
number of graphs: 18, accuracy: 0.9164, time: 0.2186
number of graphs: 19, accuracy: 0.9229, time: 0.2585
number of graphs: 20, accuracy: 0.9211, time: 0.3018
number of graphs: 21, accuracy: 0.9194, time: 0.3632
number of graphs: 22, accuracy: 0.9179, time: 0.4112
number of graphs: 23, accuracy: 0.9126, time: 0.4390
number of graphs: 24, accuracy: 0.9090, time: 0.5224
Online Test, mode fast-spfa, class Motorbike passed
```



```
*****************************************************
Test on class Face
iter: 0/5
number of graphs: 17, accuracy: 0.9551, time: 0.1915
number of graphs: 18, accuracy: 0.9595, time: 0.2274
number of graphs: 19, accuracy: 0.9475, time: 0.2523
number of graphs: 20, accuracy: 0.9551, time: 0.3142
number of graphs: 21, accuracy: 0.9483, time: 0.3461
number of graphs: 22, accuracy: 0.9351, time: 0.3820
number of graphs: 23, accuracy: 0.9374, time: 0.4308
number of graphs: 24, accuracy: 0.9408, time: 0.4877

iter: 1/5
number of graphs: 17, accuracy: 0.9906, time: 0.1915
number of graphs: 18, accuracy: 0.9810, time: 0.2334
number of graphs: 19, accuracy: 0.9713, time: 0.2633
number of graphs: 20, accuracy: 0.9632, time: 0.3022
number of graphs: 21, accuracy: 0.9527, time: 0.3700
number of graphs: 22, accuracy: 0.9535, time: 0.4498
number of graphs: 23, accuracy: 0.9488, time: 0.4638
number of graphs: 24, accuracy: 0.9514, time: 0.5216

iter: 2/5
number of graphs: 17, accuracy: 0.9898, time: 0.2124
number of graphs: 18, accuracy: 0.9952, time: 0.2613
number of graphs: 19, accuracy: 0.9926, time: 0.2984
number of graphs: 20, accuracy: 0.9900, time: 0.3102
number of graphs: 21, accuracy: 0.9845, time: 0.3451
number of graphs: 22, accuracy: 0.9791, time: 0.4159
number of graphs: 23, accuracy: 0.9806, time: 0.5227
number of graphs: 24, accuracy: 0.9841, time: 0.5465

iter: 3/5
number of graphs: 17, accuracy: 0.9422, time: 0.2254
number of graphs: 18, accuracy: 0.9391, time: 0.2314
number of graphs: 19, accuracy: 0.9420, time: 0.2533
number of graphs: 20, accuracy: 0.9454, time: 0.3022
number of graphs: 21, accuracy: 0.9400, time: 0.3391
number of graphs: 22, accuracy: 0.9413, time: 0.4029
number of graphs: 23, accuracy: 0.9467, time: 0.6144
number of graphs: 24, accuracy: 0.9490, time: 0.5076

iter: 4/5
number of graphs: 17, accuracy: 0.9867, time: 0.2703
number of graphs: 18, accuracy: 0.9855, time: 0.2124
number of graphs: 19, accuracy: 0.9867, time: 0.2773
number of graphs: 20, accuracy: 0.9823, time: 0.3152
number of graphs: 21, accuracy: 0.9810, time: 0.3481
number of graphs: 22, accuracy: 0.9830, time: 0.3870
number of graphs: 23, accuracy: 0.9721, time: 0.4468
number of graphs: 24, accuracy: 0.9749, time: 0.5545

Overall performance on class Face
number of graphs: 17, accuracy: 0.9729, time: 0.2182
number of graphs: 18, accuracy: 0.9720, time: 0.2332
number of graphs: 19, accuracy: 0.9680, time: 0.2689
number of graphs: 20, accuracy: 0.9672, time: 0.3088
number of graphs: 21, accuracy: 0.9613, time: 0.3497
number of graphs: 22, accuracy: 0.9584, time: 0.4075
number of graphs: 23, accuracy: 0.9571, time: 0.4957
number of graphs: 24, accuracy: 0.9600, time: 0.5236
Online Test, mode fast-spfa, class Face passed
```



```
*****************************************************
Test on class Winebottle
iter: 0/5
number of graphs: 17, accuracy: 0.7574, time: 0.1865
number of graphs: 18, accuracy: 0.7924, time: 0.2693
number of graphs: 19, accuracy: 0.7988, time: 0.2803
number of graphs: 20, accuracy: 0.7922, time: 0.3710
number of graphs: 21, accuracy: 0.7940, time: 0.4224
number of graphs: 22, accuracy: 0.7857, time: 0.3840
number of graphs: 23, accuracy: 0.7936, time: 0.4877
number of graphs: 24, accuracy: 0.7726, time: 0.5515

iter: 1/5
number of graphs: 17, accuracy: 0.8281, time: 0.1865
number of graphs: 18, accuracy: 0.8720, time: 0.2025
number of graphs: 19, accuracy: 0.9210, time: 0.2842
number of graphs: 20, accuracy: 0.9197, time: 0.3451
number of graphs: 21, accuracy: 0.9215, time: 0.3760
number of graphs: 22, accuracy: 0.9338, time: 0.3979
number of graphs: 23, accuracy: 0.9244, time: 0.4548
number of graphs: 24, accuracy: 0.9308, time: 0.5236

iter: 2/5
number of graphs: 17, accuracy: 0.6762, time: 0.2259
number of graphs: 18, accuracy: 0.6260, time: 0.2558
number of graphs: 19, accuracy: 0.6438, time: 0.2842
number of graphs: 20, accuracy: 0.6299, time: 0.3461
number of graphs: 21, accuracy: 0.6378, time: 0.3949
number of graphs: 22, accuracy: 0.6807, time: 0.4198
number of graphs: 23, accuracy: 0.6847, time: 0.5206
number of graphs: 24, accuracy: 0.7081, time: 0.5226

iter: 3/5
number of graphs: 17, accuracy: 0.8555, time: 0.2035
number of graphs: 18, accuracy: 0.8713, time: 0.2533
number of graphs: 19, accuracy: 0.8738, time: 0.2992
number of graphs: 20, accuracy: 0.8861, time: 0.3261
number of graphs: 21, accuracy: 0.9050, time: 0.3541
number of graphs: 22, accuracy: 0.7522, time: 0.4208
number of graphs: 23, accuracy: 0.8938, time: 0.4758
number of graphs: 24, accuracy: 0.7947, time: 0.5027

iter: 4/5
number of graphs: 17, accuracy: 0.7141, time: 0.2144
number of graphs: 18, accuracy: 0.6955, time: 0.2593
number of graphs: 19, accuracy: 0.7015, time: 0.2952
number of graphs: 20, accuracy: 0.7000, time: 0.4395
number of graphs: 21, accuracy: 0.6985, time: 0.3710
number of graphs: 22, accuracy: 0.7138, time: 0.6122
number of graphs: 23, accuracy: 0.7211, time: 0.4289
number of graphs: 24, accuracy: 0.7233, time: 0.5007

Overall performance on class Winebottle
number of graphs: 17, accuracy: 0.7662, time: 0.2034
number of graphs: 18, accuracy: 0.7714, time: 0.2480
number of graphs: 19, accuracy: 0.7878, time: 0.2886
number of graphs: 20, accuracy: 0.7856, time: 0.3656
number of graphs: 21, accuracy: 0.7914, time: 0.3837
number of graphs: 22, accuracy: 0.7732, time: 0.4470
number of graphs: 23, accuracy: 0.8035, time: 0.4735
number of graphs: 24, accuracy: 0.7859, time: 0.5202
Online Test, mode fast-spfa, class Winebottle passed
```



```
*****************************************************
Test on class Duck
iter: 0/5
number of graphs: 17, accuracy: 0.9320, time: 0.1895
number of graphs: 18, accuracy: 0.9394, time: 0.2424
number of graphs: 19, accuracy: 0.9488, time: 0.2593
number of graphs: 20, accuracy: 0.9324, time: 0.3022
number of graphs: 21, accuracy: 0.8748, time: 0.3271
number of graphs: 22, accuracy: 0.8721, time: 0.3720
number of graphs: 23, accuracy: 0.7983, time: 0.4249
number of graphs: 24, accuracy: 0.8231, time: 0.4857

iter: 1/5
number of graphs: 17, accuracy: 0.8125, time: 0.1795
number of graphs: 18, accuracy: 0.8294, time: 0.2473
number of graphs: 19, accuracy: 0.8302, time: 0.2833
number of graphs: 20, accuracy: 0.8399, time: 0.2882
number of graphs: 21, accuracy: 0.8433, time: 0.3401
number of graphs: 22, accuracy: 0.8463, time: 0.3640
number of graphs: 23, accuracy: 0.8519, time: 0.4179
number of graphs: 24, accuracy: 0.8406, time: 0.4697

iter: 2/5
number of graphs: 17, accuracy: 0.7156, time: 0.1705
number of graphs: 18, accuracy: 0.6986, time: 0.2094
number of graphs: 19, accuracy: 0.6775, time: 0.2593
number of graphs: 20, accuracy: 0.7022, time: 0.3002
number of graphs: 21, accuracy: 0.7278, time: 0.3331
number of graphs: 22, accuracy: 0.7122, time: 0.3540
number of graphs: 23, accuracy: 0.7545, time: 0.4239
number of graphs: 24, accuracy: 0.7752, time: 0.4867

iter: 3/5
number of graphs: 17, accuracy: 0.7418, time: 0.1815
number of graphs: 18, accuracy: 0.6976, time: 0.2104
number of graphs: 19, accuracy: 0.6843, time: 0.2424
number of graphs: 20, accuracy: 0.7116, time: 0.3332
number of graphs: 21, accuracy: 0.7278, time: 0.3750
number of graphs: 22, accuracy: 0.7431, time: 0.4329
number of graphs: 23, accuracy: 0.7411, time: 0.4478
number of graphs: 24, accuracy: 0.7493, time: 0.5994

iter: 4/5
number of graphs: 17, accuracy: 0.7254, time: 0.1735
number of graphs: 18, accuracy: 0.6896, time: 0.2134
number of graphs: 19, accuracy: 0.7265, time: 0.2493
number of graphs: 20, accuracy: 0.7432, time: 0.2892
number of graphs: 21, accuracy: 0.7198, time: 0.3341
number of graphs: 22, accuracy: 0.7259, time: 0.3820
number of graphs: 23, accuracy: 0.7064, time: 0.4279
number of graphs: 24, accuracy: 0.7110, time: 0.5186

Overall performance on class Duck
number of graphs: 17, accuracy: 0.7855, time: 0.1789
number of graphs: 18, accuracy: 0.7709, time: 0.2246
number of graphs: 19, accuracy: 0.7735, time: 0.2587
number of graphs: 20, accuracy: 0.7859, time: 0.3026
number of graphs: 21, accuracy: 0.7786, time: 0.3419
number of graphs: 22, accuracy: 0.7799, time: 0.3810
number of graphs: 23, accuracy: 0.7705, time: 0.4285
number of graphs: 24, accuracy: 0.7798, time: 0.5120
Online Test, mode fast-spfa, class Duck passed
```

Online test using Fast-SPFA passed

P.S. Our experiments is run on Intel® Core™ i7-8565U CPU @ 1.80GHz 1.99GHz, 16GB RAM.