

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:

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

where \mathbf{X} is a (partial) permutation matrix indicating the node correspondence, and $\mathbf{K} \in \mathbb{R}^{n_1 n_2 \times n_1 n_2}$ is the affinity matrix whose diagonal (off-diagonal) encodes the node-to-node affinity (edge-to-edge affinity) between two graphs. The symbol $\text{vec}(\cdot)$ 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:

```
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
    # UPDATE: this is column vectorize
    XT = X.transpose((0, 1, 3, 2))
    vx = np.reshape(XT, 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:

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

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 $C_p(\mathbf{X}_{ij})$ 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:

```
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 $S(\mathbf{X}_{ij}, \mathbb{X}) = \overbrace{(1 - \lambda)J(\mathbf{X}_{ij})}^{\text{affinity score}} + \overbrace{\lambda C_p(\mathbf{X}_{ij}, \mathbb{X})}^{\text{pairwise consistency}}$. In practice we use the pc approximated version $S_{pc}^{\mathbb{X}}(\mathbf{X}_{ij}, \mathbf{X}_{jk}) = (1 - \lambda)J(\mathbf{X}_{ij}\mathbf{X}_{jk}) + \lambda \sqrt{C_p(\mathbf{X}_{ij}, \mathbb{X}) C_p(\mathbf{X}_{jk}, \mathbb{X})}$. 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 computed parallelly.

```

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-SPEA (Online MGM)

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

```
1 for each newly arriving graph  $G_N$  do
2   use two-graph solver to obtain  $\mathbb{X}_{N_i}^{(0)}$  for  $G_N$  and others;
3   initialize the graph queue  $\mathcal{Q} = \{G_1, G_2 \dots, G_{N-1}\}$ ;
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,
9         Eq. 8);
10      if  $S_{org} < S_{opt}$  then
11         $\mathbf{X}_{yN} \leftarrow \mathbf{X}_{yx}\mathbf{X}_{xN}$ ;
12        add  $G_y$  into  $\mathcal{Q}$ ;
13
14  for each pair of graphs  $G_x, G_y$  in  $\mathcal{H} \setminus \{G_N\}$  do
15    set  $S_{org} = S(\mathbf{X}_{xN})$  by Eq. 5 (or Eq. 7, Eq. 8);
16    set  $S_{opt} = S(\mathbf{X}_{xy}\mathbf{X}_{yN})$  by Eq. 5 (or Eq. 7, Eq. 8);
17    if  $S_{org} < S_{opt}$  then
18       $\mathbf{X}_{xy} \leftarrow \mathbf{X}_{xN}\mathbf{X}_{Ny}$ ;
```

Output: optimized matching \mathbb{X} .

```

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 iteration,
               X[num_graph,:] and X[:,num_graph] is obtained via two-graph matching solver(RM),
               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, ...]) # X_opt
    Sorg = (1 - LAMBDA_SPFA) * cal_affinity_score(X, K) + LAMBDA_SPFA * pairwise_consistency
    Sopt = (1 - LAMBDA_SPFA) * cal_affinity_score(Xopt, K) + LAMBDA_SPFA * np.sqrt(
        np.matmul(pairwise_consistency[:, num_graph - 1][:, None], pairwise_consistency[num_
    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 $\{\mathbf{K}_{ij}\}_{i,j=1}^N$, λ , 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 $\mathcal{H} \setminus \{G_N\}$ 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 $\mathcal{H} \setminus \{G_N\}$:
 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} .

```
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 iteration
              X[num_graph,:] and X[:,num_graph] is obtained via two-graph matching solver(RM)
              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[x,N]
        for y in range(len(coord) - 1):
            if y == Gx:
                continue
            Sorg = (1 - LAMBDA_FAST) * cal_affinity_score_single(Xc[y, -1], Kc[y, -1])
                    LAMBDA_FAST * cal_pairwise_consistency_single(Xc, y, -1)
            Sopt = (1 - LAMBDA_FAST) * cal_affinity_score_single(Xopt[y, -1], Kc[y, -1])
                    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, ...]) # X_opt
    Sorg = (1 - LAMBDA_FAST) * cal_affinity_score(X, K) + LAMBDA_FAST * pairwise_consistency
    Sopt = (1 - LAMBDA_SPFA) * cal_affinity_score(Xopt, K) + LAMBDA_SPFA * np.sqrt(
        np.matmul(pairwise_consistency[:, num_graph - 1][:, None], pairwise_consistency[num_
    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:

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

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_u **do**
- 3 **for each pair of graphs** G_x, G_y **do**
- 4 set $S_{org} = S(\mathbb{X}_{xy})$ by Eq. 5 (or using approximate $S_{pc}^{\mathbb{X}}$ by Eq. 7, $S_{uc}^{\mathbb{X}}$ by Eq. 8 for speedup);
- 5 set $S_{opt} = S(\mathbb{X}_{xu} \mathbb{X}_{uy})$ by Eq. 5 (or Eq. 7, Eq. 8);
- 6 **if** $S_{opt} < S_{org}$ **then**
- 7 $\mathbb{X}_{xy} \leftarrow \mathbb{X}_{xu} \mathbb{X}_{uy}$;
- 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	Test on class Motorbike	Test on class Face	Test on class Minoturtle	Test on class Duck
<p>number of graphs: 17, accuracy: 0.6441, time: 0.4527</p> <p>number of graphs: 18, accuracy: 0.6325, time: 0.4834</p> <p>number of graphs: 19, accuracy: 0.6501, time: 0.7181</p> <p>number of graphs: 20, accuracy: 0.6902, time: 0.8806</p> <p>number of graphs: 21, accuracy: 0.6928, time: 0.9396</p> <p>number of graphs: 22, accuracy: 0.7211, time: 1.0303</p> <p>number of graphs: 23, accuracy: 0.7231, time: 1.1732</p> <p>number of graphs: 24, accuracy: 0.7287, time: 1.4476</p>	<p>number of graphs: 17, accuracy: 0.8484, time: 0.5974</p> <p>number of graphs: 18, accuracy: 0.8557, time: 0.6592</p> <p>number of graphs: 19, accuracy: 0.8660, time: 0.7594</p> <p>number of graphs: 20, accuracy: 0.8755, time: 0.8815</p> <p>number of graphs: 21, accuracy: 0.8795, time: 1.0256</p> <p>number of graphs: 22, accuracy: 0.8851, time: 1.1637</p> <p>number of graphs: 23, accuracy: 0.8855, time: 1.7463</p> <p>number of graphs: 24, accuracy: 0.8941, time: 1.7493</p>	<p>number of graphs: 17, accuracy: 0.9501, time: 0.5336</p> <p>number of graphs: 18, accuracy: 0.9502, time: 0.6243</p> <p>number of graphs: 19, accuracy: 0.9475, time: 0.7699</p> <p>number of graphs: 20, accuracy: 0.9543, time: 0.9295</p> <p>number of graphs: 21, accuracy: 0.9573, time: 1.0925</p> <p>number of graphs: 22, accuracy: 0.9338, time: 1.3115</p> <p>number of graphs: 23, accuracy: 0.9573, time: 1.5435</p> <p>number of graphs: 24, accuracy: 0.9489, time: 1.7782</p>	<p>number of graphs: 17, accuracy: 0.7641, time: 0.4967</p> <p>number of graphs: 18, accuracy: 0.7941, time: 0.5044</p> <p>number of graphs: 19, accuracy: 0.7960, time: 0.7558</p> <p>number of graphs: 20, accuracy: 0.9222, time: 0.9565</p> <p>number of graphs: 21, accuracy: 0.9255, time: 1.0925</p> <p>number of graphs: 22, accuracy: 0.7893, time: 1.2906</p> <p>number of graphs: 23, accuracy: 0.7913, time: 1.5435</p> <p>number of graphs: 24, accuracy: 0.7733, time: 1.7544</p>	<p>number of graphs: 17, accuracy: 0.9332, time: 0.5296</p> <p>number of graphs: 18, accuracy: 0.9304, time: 0.7043</p> <p>number of graphs: 19, accuracy: 0.9484, time: 0.7679</p> <p>number of graphs: 20, accuracy: 0.9321, time: 0.9784</p> <p>number of graphs: 21, accuracy: 0.9321, time: 1.0925</p> <p>number of graphs: 22, accuracy: 0.9732, time: 2.0009</p> <p>number of graphs: 23, accuracy: 0.9732, time: 2.0009</p> <p>number of graphs: 24, accuracy: 0.8229, time: 2.0876</p>
<p>iter: 1/5</p> <p>number of graphs: 17, accuracy: 0.7775, time: 0.4388</p> <p>number of graphs: 18, accuracy: 0.7775, time: 0.5346</p> <p>number of graphs: 19, accuracy: 0.7895, time: 0.6943</p> <p>number of graphs: 20, accuracy: 0.7775, time: 0.9454</p> <p>number of graphs: 21, accuracy: 0.7775, time: 0.9454</p> <p>number of graphs: 22, accuracy: 0.7912, time: 1.1507</p> <p>number of graphs: 23, accuracy: 0.7912, time: 1.3637</p> <p>number of graphs: 24, accuracy: 0.7856, time: 1.5369</p>	<p>iter: 1/5</p> <p>number of graphs: 17, accuracy: 0.9329, time: 0.5667</p> <p>number of graphs: 18, accuracy: 0.9329, time: 0.6696</p> <p>number of graphs: 19, accuracy: 0.9457, time: 0.8487</p> <p>number of graphs: 20, accuracy: 0.9500, time: 0.9696</p> <p>number of graphs: 21, accuracy: 0.9500, time: 1.1538</p> <p>number of graphs: 22, accuracy: 0.9508, time: 1.4092</p> <p>number of graphs: 23, accuracy: 0.9514, time: 1.7882</p> <p>number of graphs: 24, accuracy: 0.9556, time: 1.9239</p>	<p>iter: 1/5</p> <p>number of graphs: 17, accuracy: 0.9986, time: 0.5366</p> <p>number of graphs: 18, accuracy: 0.9910, time: 0.5893</p> <p>number of graphs: 19, accuracy: 0.9902, time: 0.7424</p> <p>number of graphs: 20, accuracy: 0.9832, time: 0.9597</p> <p>number of graphs: 21, accuracy: 0.9570, time: 1.1389</p> <p>number of graphs: 22, accuracy: 0.9826, time: 1.4731</p> <p>number of graphs: 23, accuracy: 0.9488, time: 1.4571</p> <p>number of graphs: 24, accuracy: 0.9512, time: 1.7234</p>	<p>iter: 1/5</p> <p>number of graphs: 17, accuracy: 0.8273, time: 0.4877</p> <p>number of graphs: 18, accuracy: 0.8744, time: 0.5798</p> <p>number of graphs: 19, accuracy: 0.9206, time: 0.7507</p> <p>number of graphs: 20, accuracy: 0.9206, time: 0.9597</p> <p>number of graphs: 21, accuracy: 0.9216, time: 1.0731</p> <p>number of graphs: 22, accuracy: 0.9216, time: 1.4731</p> <p>number of graphs: 23, accuracy: 0.9314, time: 1.7338</p> <p>number of graphs: 24, accuracy: 0.9314, time: 1.7338</p>	<p>iter: 1/5</p> <p>number of graphs: 17, accuracy: 0.8133, time: 0.4047</p> <p>number of graphs: 18, accuracy: 0.8296, time: 0.6825</p> <p>number of graphs: 19, accuracy: 0.8681, time: 0.7679</p> <p>number of graphs: 20, accuracy: 0.8782, time: 0.9321</p> <p>number of graphs: 21, accuracy: 0.8782, time: 1.0925</p> <p>number of graphs: 22, accuracy: 0.8510, time: 1.3643</p> <p>number of graphs: 23, accuracy: 0.8562, time: 1.6037</p> <p>number of graphs: 24, accuracy: 0.8341, time: 1.8020</p>
<p>iter: 2/5</p> <p>number of graphs: 17, accuracy: 0.7775, time: 0.5748</p> <p>number of graphs: 18, accuracy: 0.7993, time: 0.5858</p> <p>number of graphs: 19, accuracy: 0.8080, time: 0.6863</p> <p>number of graphs: 20, accuracy: 0.8828, time: 0.7849</p> <p>number of graphs: 21, accuracy: 0.8828, time: 0.9235</p> <p>number of graphs: 22, accuracy: 0.8854, time: 1.0841</p> <p>number of graphs: 23, accuracy: 0.8756, time: 1.1855</p> <p>number of graphs: 24, accuracy: 0.8820, time: 1.4721</p>	<p>iter: 2/5</p> <p>number of graphs: 17, accuracy: 0.9168, time: 0.5396</p> <p>number of graphs: 18, accuracy: 0.9176, time: 0.7769</p> <p>number of graphs: 19, accuracy: 0.9176, time: 0.8815</p> <p>number of graphs: 20, accuracy: 0.9101, time: 1.0851</p> <p>number of graphs: 21, accuracy: 0.9101, time: 1.2687</p> <p>number of graphs: 22, accuracy: 0.9194, time: 1.4481</p> <p>number of graphs: 23, accuracy: 0.9159, time: 1.7473</p> <p>number of graphs: 24, accuracy: 0.9076, time: 1.7893</p>	<p>iter: 2/5</p> <p>number of graphs: 17, accuracy: 0.9988, time: 0.5886</p> <p>number of graphs: 18, accuracy: 0.9952, time: 0.6424</p> <p>number of graphs: 19, accuracy: 0.9926, time: 0.7420</p> <p>number of graphs: 20, accuracy: 0.9926, time: 0.9235</p> <p>number of graphs: 21, accuracy: 0.9845, time: 1.0632</p> <p>number of graphs: 22, accuracy: 0.9719, time: 1.2527</p> <p>number of graphs: 23, accuracy: 0.9886, time: 1.4741</p> <p>number of graphs: 24, accuracy: 0.9841, time: 1.7613</p>	<p>iter: 2/5</p> <p>number of graphs: 17, accuracy: 0.8687, time: 0.4887</p> <p>number of graphs: 18, accuracy: 0.6291, time: 0.5954</p> <p>number of graphs: 19, accuracy: 0.6417, time: 0.7438</p> <p>number of graphs: 20, accuracy: 0.6417, time: 0.9235</p> <p>number of graphs: 21, accuracy: 0.6310, time: 1.0422</p> <p>number of graphs: 22, accuracy: 0.6823, time: 1.2656</p> <p>number of graphs: 23, accuracy: 0.6826, time: 1.4731</p> <p>number of graphs: 24, accuracy: 0.7866, time: 1.7327</p>	<p>iter: 2/5</p> <p>number of graphs: 17, accuracy: 0.7156, time: 0.5075</p> <p>number of graphs: 18, accuracy: 0.6983, time: 0.6622</p> <p>number of graphs: 19, accuracy: 0.6983, time: 0.7679</p> <p>number of graphs: 20</p>

Online test using MGM-SPFA passed

```

Test on class C*****
Iter: 0/5
number of graphs: 17, accuracy: 0.6441, time: 0.1845
number of graphs: 18, accuracy: 0.6326, time: 0.2224
number of graphs: 19, accuracy: 0.6861, time: 0.2653
number of graphs: 20, accuracy: 0.6904, time: 0.3022
number of graphs: 21, accuracy: 0.7015, time: 0.3367
number of graphs: 22, accuracy: 0.7145, time: 0.3720
number of graphs: 23, accuracy: 0.7291, time: 0.4068
number of graphs: 24, accuracy: 0.7493, time: 0.5160

Iter: 1/5
number of graphs: 17, accuracy: 0.7777, time: 0.1975
number of graphs: 18, accuracy: 0.7716, time: 0.2421
number of graphs: 19, accuracy: 0.8089, time: 0.2859
number of graphs: 20, accuracy: 0.7812, time: 0.3172
number of graphs: 21, accuracy: 0.8049, time: 0.3509
number of graphs: 22, accuracy: 0.7923, time: 0.4069
number of graphs: 23, accuracy: 0.7883, time: 0.4259
number of graphs: 24, accuracy: 0.7686, time: 0.5345

Iter: 2/5
number of graphs: 17, accuracy: 0.7768, time: 0.1943
number of graphs: 18, accuracy: 0.7979, time: 0.2384
number of graphs: 19, accuracy: 0.8111, time: 0.2809
number of graphs: 20, accuracy: 0.8171, time: 0.3003
number of graphs: 21, accuracy: 0.8397, time: 0.3311
number of graphs: 22, accuracy: 0.8212, time: 0.3750
number of graphs: 23, accuracy: 0.8661, time: 0.4220
number of graphs: 24, accuracy: 0.8661, time: 0.5485

Iter: 3/5
number of graphs: 17, accuracy: 0.8661, time: 0.1808
number of graphs: 18, accuracy: 0.8886, time: 0.2292
number of graphs: 19, accuracy: 0.8786, time: 0.2973
number of graphs: 20, accuracy: 0.8911, time: 0.2973
number of graphs: 21, accuracy: 0.8655, time: 0.3730
number of graphs: 22, accuracy: 0.8667, time: 0.4154
number of graphs: 23, accuracy: 0.8537, time: 0.4510
number of graphs: 24, accuracy: 0.8756, time: 0.4967

Iter: 4/5
number of graphs: 17, accuracy: 0.8886, time: 0.1795
number of graphs: 18, accuracy: 0.8671, time: 0.2424
number of graphs: 19, accuracy: 0.8670, time: 0.2925
number of graphs: 20, accuracy: 0.8629, time: 0.4964
number of graphs: 21, accuracy: 0.8670, time: 0.4971
number of graphs: 22, accuracy: 0.8429, time: 0.5380
number of graphs: 23, accuracy: 0.8537, time: 0.5513
number of graphs: 24, accuracy: 0.8631, time: 0.5910

Overall performance on class C
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: 0.8089, time: 0.2860
number of graphs: 20, accuracy: 0.7965, time: 0.3578
number of graphs: 21, accuracy: 0.8097, time: 0.3897
number of graphs: 22, accuracy: 0.7915, time: 0.4043
number of graphs: 23, accuracy: 0.8107, time: 0.5015
number of graphs: 24, accuracy: 0.8107, time: 0.5015

Online Test, mode Fast-sprf, class C 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: 0.8659, time: 0.2503
number of graphs: 20, accuracy: 0.8751, time: 0.2942
number of graphs: 21, accuracy: 0.8787, time: 0.4229
number of graphs: 22, accuracy: 0.8850, time: 0.4567
number of graphs: 23, accuracy: 0.8847, time: 0.4488
number of graphs: 24, accuracy: 0.8849, time: 0.4857

Iter: 1/5
number of graphs: 17, accuracy: 0.8881, time: 0.1835
number of graphs: 18, accuracy: 0.9304, time: 0.2184
number of graphs: 19, accuracy: 0.9304, time: 0.2513
number of graphs: 20, accuracy: 0.9304, time: 0.2859
number of graphs: 21, accuracy: 0.9440, time: 0.3271
number of graphs: 22, accuracy: 0.9304, time: 0.3578
number of graphs: 23, accuracy: 0.9311, time: 0.4348
number of graphs: 24, accuracy: 0.9284, time: 0.4857

Iter: 2/5
number of graphs: 17, accuracy: 0.9168, time: 0.1805
number of graphs: 18, accuracy: 0.9376, time: 0.2214
number of graphs: 19, accuracy: 0.9231, time: 0.2523
number of graphs: 20, accuracy: 0.9376, time: 0.2859
number of graphs: 21, accuracy: 0.9120, time: 0.3261
number of graphs: 22, accuracy: 0.9143, time: 0.3768
number of graphs: 23, accuracy: 0.9159, time: 0.4210
number of graphs: 24, accuracy: 0.9072, time: 0.5251

Iter: 3/5
number of graphs: 17, accuracy: 0.9204, time: 0.1925
number of graphs: 18, accuracy: 0.9308, time: 0.2244
number of graphs: 19, accuracy: 0.9209, time: 0.2623
number of graphs: 20, accuracy: 0.9109, time: 0.2962
number of graphs: 21, accuracy: 0.9120, time: 0.3012
number of graphs: 22, accuracy: 0.9245, time: 0.3838
number of graphs: 23, accuracy: 0.9120, time: 0.4210
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: 0.9480, time: 0.2254
number of graphs: 19, accuracy: 0.9480, time: 0.2584
number of graphs: 20, accuracy: 0.9438, time: 0.3291
number of graphs: 21, accuracy: 0.9438, time: 0.3592
number of graphs: 22, accuracy: 0.9410, time: 0.3959
number of graphs: 23, accuracy: 0.9835, time: 0.4488
number of graphs: 24, accuracy: 0.9566, time: 0.5706

Overall performance on class Motorbike
number of graphs: 17, accuracy: 0.9048, time: 0.1865
number of graphs: 18, accuracy: 0.9164, time: 0.2186
number of graphs: 19, accuracy: 0.9164, time: 0.2513
number of graphs: 20, accuracy: 0.9211, time: 0.3018
number of graphs: 21, accuracy: 0.9194, time: 0.3652
number of graphs: 22, accuracy: 0.9194, time: 0.3897
number of graphs: 23, accuracy: 0.9126, time: 0.4350
number of graphs: 24, accuracy: 0.9126, time: 0.4350

Online Test, mode Fast-sprf, 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: 0.9622, time: 0.2584
number of graphs: 20, accuracy: 0.9551, time: 0.3142
number of graphs: 21, accuracy: 0.9551, time: 0.3142
number of graphs: 22, accuracy: 0.9551, time: 0.3142
number of graphs: 23, accuracy: 0.9374, time: 0.4380
number of graphs: 24, accuracy: 0.9408, time: 0.4477

Iter: 1/5
number of graphs: 17, accuracy: 0.9986, time: 0.1915
number of graphs: 18, accuracy: 0.9810, time: 0.2354
number of graphs: 19, accuracy: 0.9810, time: 0.2685
number of graphs: 20, accuracy: 0.9622, time: 0.3022
number of graphs: 21, accuracy: 0.9527, time: 0.3780
number of graphs: 22, accuracy: 0.9527, time: 0.3780
number of graphs: 23, accuracy: 0.9488, time: 0.4638
number of graphs: 24, accuracy: 0.9314, time: 0.5216

Iter: 2/5
number of graphs: 17, accuracy: 0.9989, time: 0.1915
number of graphs: 18, accuracy: 0.9952, time: 0.2611
number of graphs: 19, accuracy: 0.9926, time: 0.2984
number of graphs: 20, accuracy: 0.9926, time: 0.3312
number of graphs: 21, accuracy: 0.9945, time: 0.4151
number of graphs: 22, accuracy: 0.9991, time: 0.4159
number of graphs: 23, accuracy: 0.9880, time: 0.5221
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: 0.9454, time: 0.3022
number of graphs: 22, accuracy: 0.9413, time: 0.4029
number of graphs: 23, accuracy: 0.9413, time: 0.4029
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: 0.9865, time: 0.2713
number of graphs: 19, accuracy: 0.9865, time: 0.2713
number of graphs: 20, accuracy: 0.9862, time: 0.3152
number of graphs: 21, accuracy: 0.9862, time: 0.3152
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: 0.5455

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: 0.9720, time: 0.2684
number of graphs:
```

Online test using Fast-SPFA passed

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