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)) + (
```

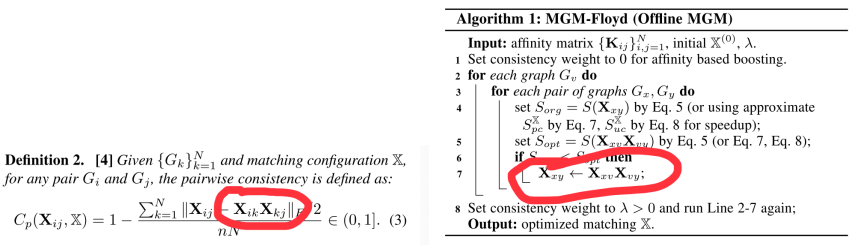
```
        [q.put(y) for y in range(num_graph) if update[y]]  # add Gy into Q

        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