pygmtools Documentation

Runzhong Wang

Sep 13, 2022

USER GUIDE

pygmtools provides graph matching solvers in Python and is easily accessible via:

```
pip install pygmtools
```

CHAPTER

ONE

BACKENDS

By default the solvers are executed on the `numpy` backend, and the required packages will be automatically downloaded.

For advanced and professional users, `pytorch/paddle/jittor` backends are also available if you have installed and configured the corresponding runtime. The `pytorch/paddle/jittor` backends exploit the underlying GPU-acceleration feature, and also support integrating graph matching modules into your deep learning pipeline.

CHAPTER

TWO

FEATURES

To highlight, **pygmtools** has the following features:

- *Support various backends*, including `numpy` which is universally accessible, and some state-of-the-art deep learning architectures with GPU-support: `pytorch/paddle/jittor`. The support of the following backends are also planned: `tensorflow`, `mindspore`;
- *Support various solvers*, including traditional combinatorial solvers and novel deep learning-based solvers;
- *Deep learning friendly*, the operations are designed to best preserve the gradient during computation and batched operations support for the best performance.

CHAPTER

THREE

BENCHMARKS

pygmtools is also featured with standard data interface of several graph matching benchmarks. We also maintain a repository containing non-trivial implementation of deep graph matching models, please check out ThinkMatch if you are interested!

CHAPTER

FOUR

DEVELOPERS AND MAINTAINERS

pygmtools is currently developed and maintained by members from ThinkLab at Shanghai Jiao Tong University.

4.1 What is Graph Matching

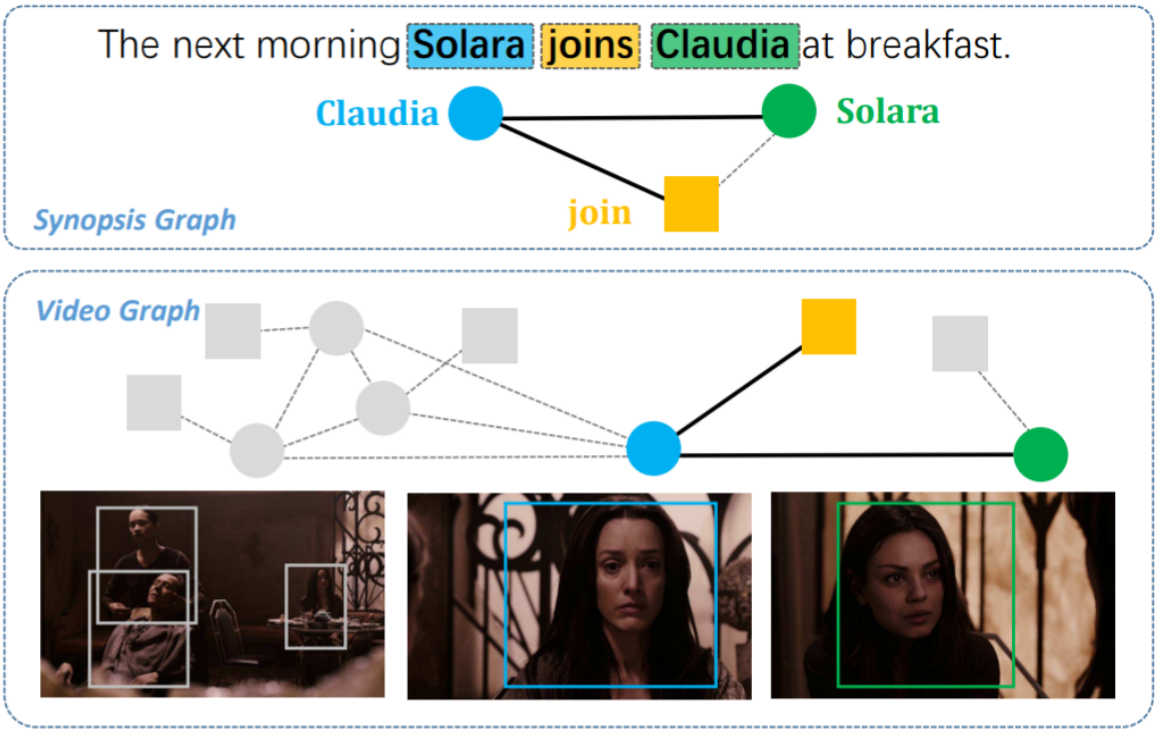
This page provides some background information for graph matching.

4.1.1 Introduction

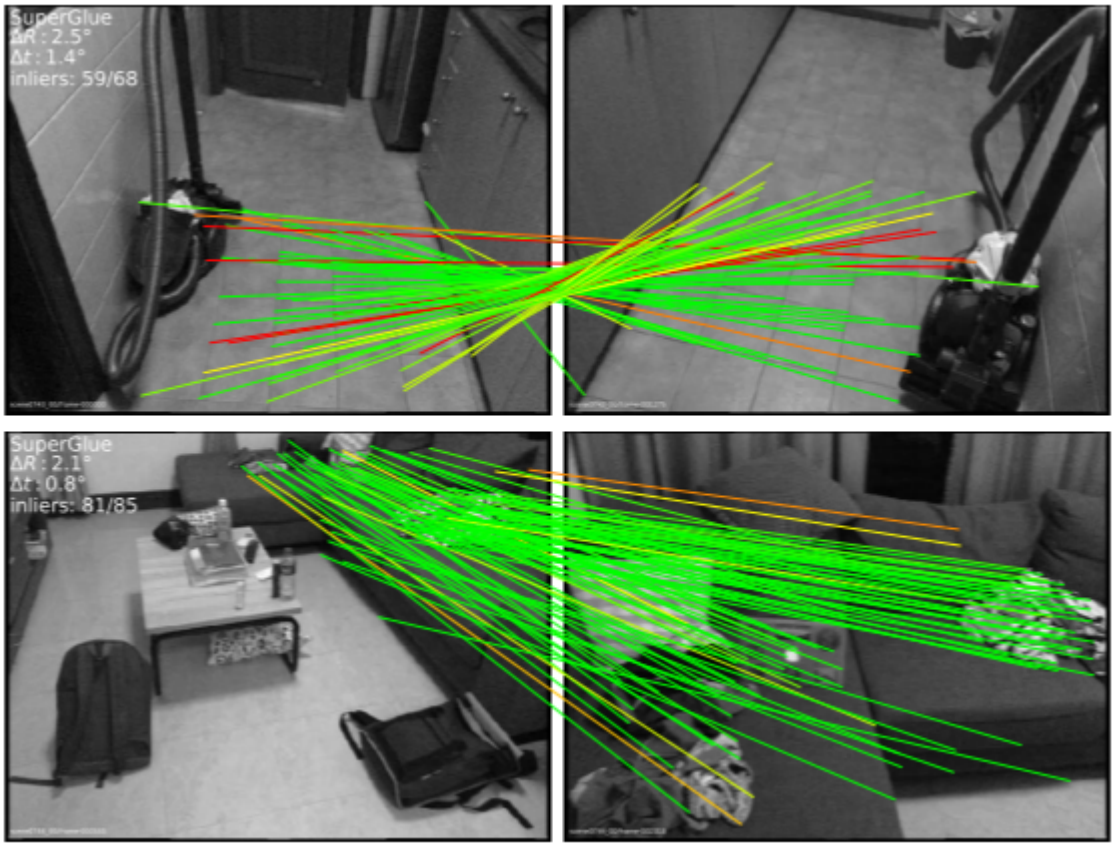
Graph Matching (GM) is a fundamental yet challenging problem in pattern recognition, data mining, and others. GM aims to find node-to-node correspondence among multiple graphs, by solving an NP-hard combinatorial problem. Recently, there is growing interest in developing deep learning based graph matching methods.

Graph matching techniques have been applied to the following applications:

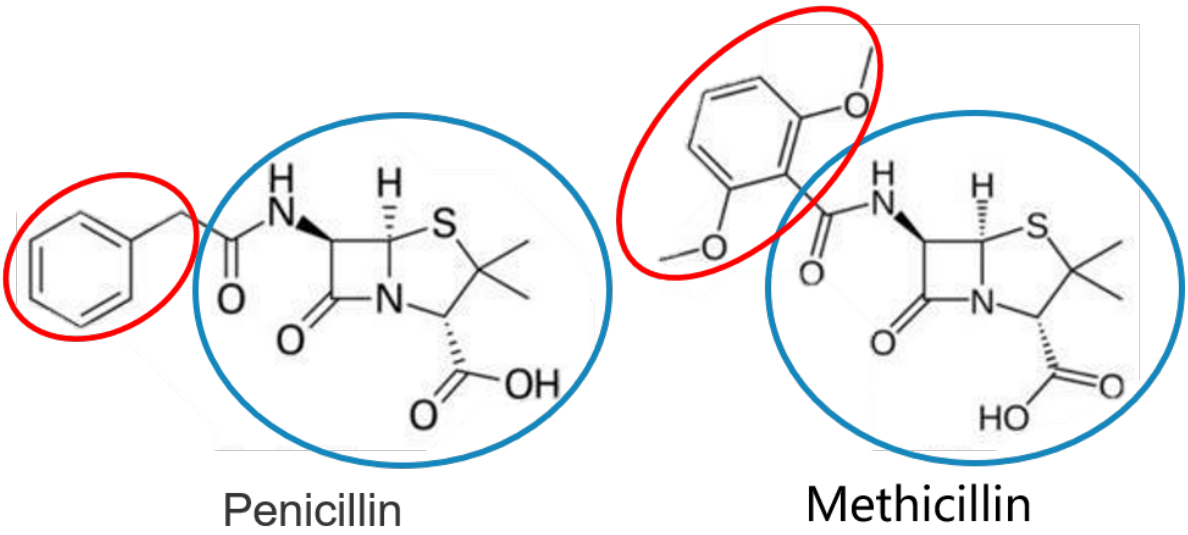
- Bridging movie and synopses



- Image correspondence



- Molecules matching



- and more…

4.1.2 Graph Matching Pipeline

Solving a real world graph matching problem may involve the following steps:

1. Extract node/edge features from the graphs you want to match.
2. Build affinity matrix from node/edge features.
3. Solve the graph matching problem by GM solvers.

And Step 1 maybe done by methods depending on your application, Step 2&3 can be handled by **pygmtools**.

4.1.3 The Math Form

Let's involve a little bit math to better understand the graph matching pipeline. In general, graph matching is of the following form, known as **Quadratic Assignment Problem (QAP)**:

$$\begin{aligned} &\max_{\mathbf{X}} \ \mathtt{vec}(\mathbf{X})^\top \mathbf{K} \mathtt{vec}(\mathbf{X}) \\ s.t. \quad &\mathbf{X} \in \{0,1\}^{n_1 \times n_2},\ \mathbf{X}\mathbf{1} = \mathbf{1},\ \mathbf{X}^\top \mathbf{1} \leq \mathbf{1} \end{aligned}$$

The notations are explained as follows:

- $\mathbf{X}$ is known as the **permutation matrix** which encodes the matching result. It is also the decision variable in graph matching problem. $\mathbf{X}_{i,a} = 1$ means node i in graph 1 is matched to node a in graph 2, and $\mathbf{X}_{i,a} = 0$ means non-matched. Without loss of generality, it is assumed that $n_1 \leq n_2$. $\mathbf{X}$ has the following constraints:
  - The sum of each row must be equal to 1: $\mathbf{X}\mathbf{1} = \mathbf{1}$;
  - The sum of each column must be equal to, or smaller than 1: $\mathbf{X}\mathbf{1} \leq \mathbf{1}$.
- $\mathtt{vec}(\mathbf{X})$ means the column-wise vectorization form of $\mathbf{X}$.
- $\mathbf{1}$ means a column vector whose elements are all 1s.
- $\mathbf{K}$ is known as the **affinity matrix** which encodes the information of the input graphs. Both node-wise and edge-wise affinities are encoded in $\mathbf{K}$:
  - The diagonal element $\mathbf{K}_{i+a\times n_1, i+a\times n_1}$ means the node-wise affinity of node i in graph 1 and node a in graph 2;
  - The off-diagonal element $\mathbf{K}_{i+a\times n_1, j+b\times n_1}$ means the edge-wise affinity of edge ij in graph 1 and edge ab in graph 2.

4.1.4 Other Materials

Readers are referred to the following surveys for more technical details about graph matching:

- Junchi Yan, Shuang Yang, Edwin Hancock. "Learning Graph Matching and Related Combinatorial Optimization Problems." *IJCAI 2020*.
- Junchi Yan, Xu-Cheng Yin, Weiyao Lin, Cheng Deng, Hongyuan Zha, Xiaokang Yang. "A Short Survey of Recent Advances in Graph Matching." *ICMR 2016*.

4.2 Get Started

4.2.1 Basic Install

Install `pygmtools` is easy:

```
pip install pygmtools
```

Now the pygmtools is available with the `numpy` backend. You may jump to *Example: Matching Isomorphic Graphs* if you do not need other backends.

The following packages are required, and shall be automatically downloaded by `pip install`:

- Python >= 3.5
- requests >= 2.25.1
- scipy >= 1.4.1
- Pillow >= 7.2.0
- numpy >= 1.18.5
- easydict >= 1.7

4.2.2 Install Other Backends

Currently, we also support deep learning architectures `pytorch/paddle/jittor` which are GPU-friendly and deep learning-friendly. The support of the following backends are also planned: `tensorflow`, `mindspore`.

Please follow the install instructions on your backend.

Once the backend is ready, you may switch to the backend globally by the following command:

```
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'pytorch'  # replace 'pytorch' by other backend names
```

4.2.3 Example: Matching Isomorphic Graphs

Here we provide a basic example of matching two isomorphic graphs (i.e. two graphs have the same nodes and edges, but the node permutations are unknown).

Step 0: Import packages and set backend

```
>>> import numpy as np
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'numpy'
>>> np.random.seed(1)
```

Step 1: Generate a batch of isomorphic graphs

```
>>> batch_size = 3
>>> X_gt = np.zeros((batch_size, 4, 4))
>>> X_gt[:, np.arange(0, 4, dtype=np.int64), np.random.permutation(4)] = 1
>>> A1 = np.random.rand(batch_size, 4, 4)
```

(continues on next page)

(continued from previous page)

```
>>> A2 = np.matmul(np.matmul(X_gt.transpose((0, 2, 1)), A1), X_gt)
>>> n1 = n2 = np.repeat([4], batch_size)
```

Step 2: Build affinity matrix and select an affinity function

```
>>> conn1, edge1, ne1 = pygm.utils.dense_to_sparse(A1)
>>> conn2, edge2, ne2 = pygm.utils.dense_to_sparse(A2)
>>> import functools
>>> gaussian_aff = functools.partial(pygm.utils.gaussian_aff_fn, sigma=1.) # set␣
↪affinity function
>>> K = pygm.utils.build_aff_mat(None, edge1, conn1, None, edge2, conn2, n1, ne1, n2,␣
↪ne2, edge_aff_fn=gaussian_aff)
```

Step 3: Solve graph matching by RRWM

```
>>> X = pygm.rrwm(K, n1, n2, beta=100)
>>> X = pygm.hungarian(X)
>>> X # X is the permutation matrix
[[[0. 0. 0. 1.]
  [0. 0. 1. 0.]
  [1. 0. 0. 0.]
  [0. 1. 0. 0.]]

 [[0. 0. 0. 1.]
  [0. 0. 1. 0.]
  [1. 0. 0. 0.]
  [0. 1. 0. 0.]]

 [[0. 0. 0. 1.]
  [0. 0. 1. 0.]
  [1. 0. 0. 0.]
  [0. 1. 0. 0.]]]
```

Final Step: Evaluate the accuracy

```
>>> (X * X_gt).sum() / X_gt.sum()
1.0
```

4.3 Graph Matching Benchmark

pygmtools also provides a protocol to fairly compare existing deep graph matching algorithms under different datasets & experiment settings. The `Benchmark` module provides a unified data interface and an evaluating platform for different datasets. Currently, pygmtools supports 5 datasets:

- PascalVOC
- Willow-Object
- SPair-71k
- CUB2011
- IMC-PT-SparseGM

4.3.1 Files

- `dataset.py`: The file includes 5 dataset classes, used to automatically download dataset and process the dataset into a json file, and also save train set and test set.
- `benchmark.py`: The file includes Benchmark class that can be used to fetch data from json file and evaluate prediction result.
- `dataset_config.py`: Fixed dataset settings, mostly dataset path and classes.

4.3.2 Notes

- Our evaluation metrics include **matching_precision (p)**, **matching_recall (r)** and **f1_score (f1)**. Also, to measure the reliability of the evaluation result, we define coverage (cvg) for each class in the dataset as *the number of evaluated pairs in the class / number of all possible pairs* in the class. Therefore, larger coverage refers to higher reliability.
- Dataset can be automatically downloaded and unzipped, but you can also download the dataset yourself, and make sure it in the right path. The expected dataset paths are listed as follows.

```
# Pascal VOC 2011 dataset with keypoint annotations
PascalVOC.ROOT_DIR = 'data/PascalVOC/TrainVal/VOCdevkit/VOC2011/'
PascalVOC.KPT_ANNO_DIR = 'data/PascalVOC/annotations/'

# Willow-Object Class dataset
WillowObject.ROOT_DIR = 'data/WillowObject/WILLOW-ObjectClass'

# CUB2011 dataset
CUB2011.ROOT_PATH = 'data/CUB_200_2011/CUB_200_2011'

# SWPair-71 Dataset
SPair.ROOT_DIR = "data/SPair-71k"

# IMC_PT_SparseGM dataset
IMC_PT_SparseGM.ROOT_DIR_NPZ = 'data/IMC-PT-SparseGM/annotations'
IMC_PT_SparseGM.ROOT_DIR_IMG = 'data/IMC-PT-SparseGM/images'
```

  Specifically, for PascalVOC, you should download the train/test split yourself, and make sure it looks like `data/PascalVOC/voc2011_pairs.npz`

4.3.3 Example

```
from pygmtools.benchmark import Benchmark

# Define Benchmark on PascalVOC.
bm = Benchmark(name='PascalVOC', sets='train',
               obj_resize=(256, 256), problem='2GM',
               filter='intersection')

# Random fetch data and ground truth.
data_list, gt_dict, _ = bm.rand_get_data(cls=None, num=2)
```