ThinkMatch Documentation

Runzhong Wang

Jun 28, 2022

USER GUIDE

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

```
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, the `pytorch` backend is also available if you have installed and configured a pytorch runtime. The `pytorch` backend exploits the underlying GPU-acceleration feature, and also supports 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 the state-of-the-art deep learning architecture `pytorch` with GPU-support. The support of the following backends are also planned: `tensorflow`, `mindspore`, `paddle`, `jittor`;
- *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

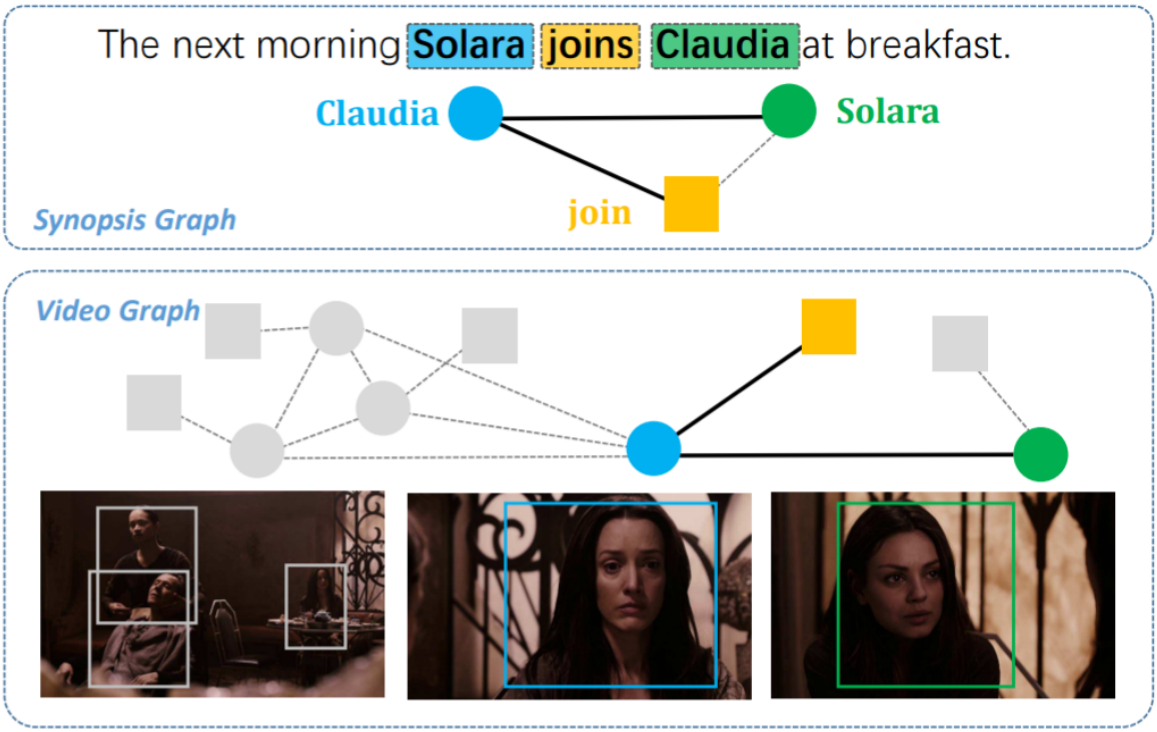
This page provides some background information for graph matching.

4.1.1 Introduction

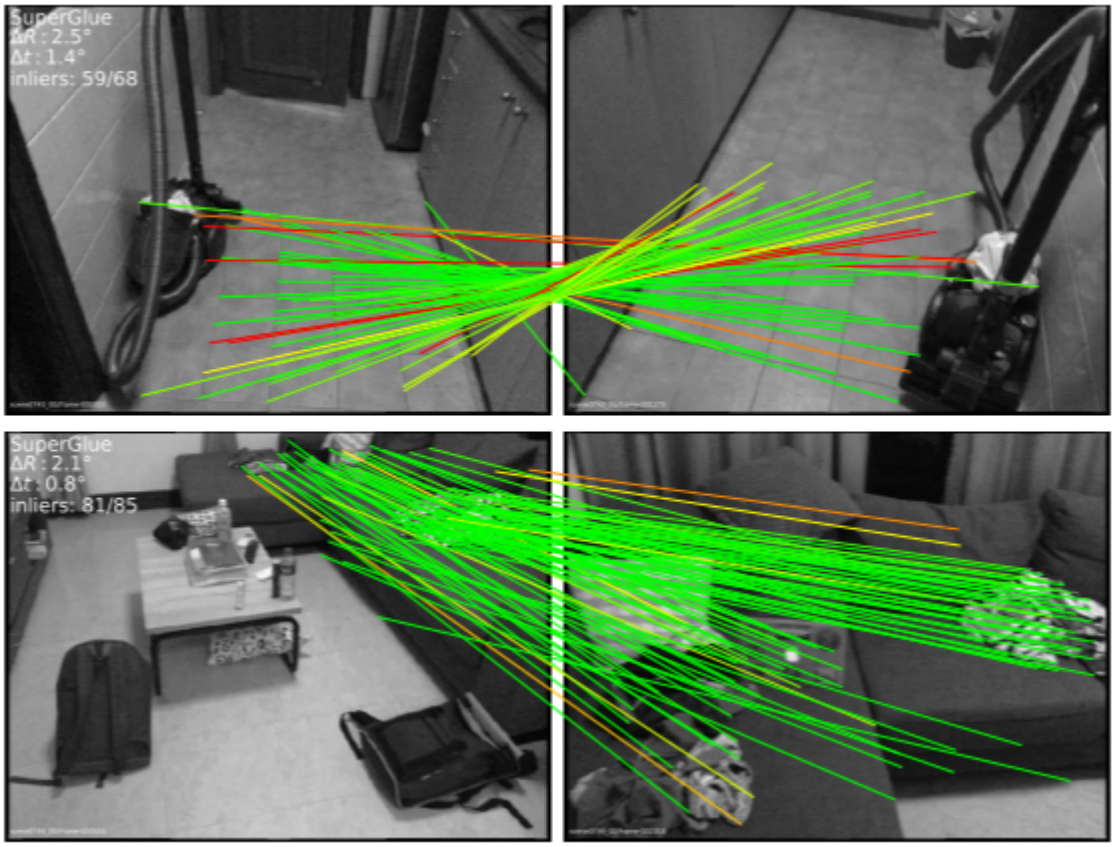
Graph Matching (GM) is a fundamental yet challenging problem in pattern recognition, data mining, and others. GM aims to find node-to-node correspondence among multiple graphs, by solving an NP-hard combinatorial problem. Recently, there is growing interest in developing deep learning based graph matching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

Graph matching techniques have been applied to the following applications:

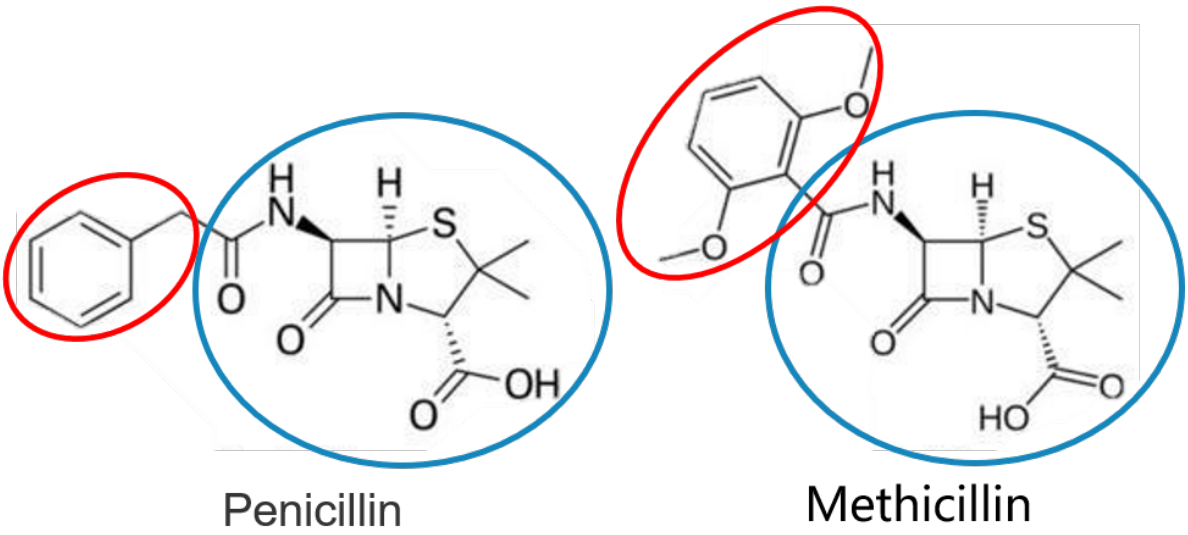
- Bridging movie and synopses



- 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

pygmtools can be installed by the `pip install` command:

```
pip install pygmtools
```

Now the pygmtools is available with the `numpy` backend. You may jump to *Example: Matching Isomorphism 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 the state-of-the-art architecture `pytorch` which is GPU-friendly and deep learning-friendly. The support of the following backends are also planned: `tensorflow`, `mindspore`, `paddle`, `jittor`.

Please follow the install instructions on your backend.

Set the backend globally by the following command:

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

4.2.3 Example: Matching Isomorphism Graphs

Here we provide a basic example of matching two isomorphism graphs (i.e. two graphs that are the same, 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)
```

4.4 pygmtools.benchmark

Classes

Benchmark	The *Benchmark* module provides a unified data interface and an evaluating platform for different datasets.

4.4.1 Benchmark

class pygmtools.benchmark.**Benchmark**(*name*, *sets*, *obj_resize=(256, 256)*, *problem='2GM'*, *filter='intersection'*, ***args*)

The *Benchmark* module provides a unified data interface and an evaluating platform for different datasets.

Parameters

- **name** – str, dataset name, currently support 'PascalVOC', 'WillowObject', 'IMC_PT_SparseGM', 'CUB2011', 'SPair71k'
- **sets** – str, problem set, 'train' for training set and 'test' for test set
- **obj_resize** – tuple, resized object size
- **problem** – str, problem type, '2GM' for 2-graph matching and 'MGM' for multi-graph matching
- **filter** – str, filter of nodes, 'intersection' refers to retaining only common nodes; 'inclusion' is only for 2GM and refers to filtering only one graph to make its nodes a subset of the other graph, and 'unfiltered' refers to retaining all nodes in all graphs
- **args** – specific settings for dataset

compute_img_num(*classes*)

Compute number of images in specified classes.

Parameters **classes** – list of dataset classes

Returns list of numbers of images in each class

compute_length(*cls=None*, *num=2*)

Compute the length of image combinations in specified class.

Parameters

- **cls** – int or str, class of expected data. None for all classes
- **num** – int, number of images in each image ID list; for example, 2 for 2GM

Returns length of combinations

eval(*prediction*, *classes*, *verbose=False*)

Evaluate test results and compute matching accuracy and coverage.

Parameters

- **prediction** – list, prediction result, like `[{'ids': (id1, id2), 'cls': cls, 'permmat': np.array or scipy.sparse},...]`
- **classes** – list of evaluated classes
- **verbose** – bool, whether to print the result

Returns evaluation result in each class and their averages, including p, r, f1 and their standard deviation and coverage

`eval_cls`(*prediction*, *cls*, *verbose=False*)

Evaluate test results and compute matching accuracy and coverage on one specified class.

Parameters

- **prediction** – list, prediction result on one class, like `[{'ids': (id1, id2), 'cls': cls, 'permmat': np.array or scipy.sparse},...]`
- **cls** – str, evaluated class
- **verbose** – bool, whether to print the result

Returns evaluation result on the specified class, including p, r, f1 and their standard deviation and coverage

`get_data`(*ids*, *test=False*, *shuffle=True*)

Fetch a data pair or pairs of data by image ID for training or test.

Parameters

- **ids** – list of image ID, usually in 'train.json' or 'test.json'
- **test** – bool, whether the fetched data is used for test; if true, this function will not return ground truth
- **shuffle** – bool, whether to shuffle the order of keypoints

Returns

data_list: list of data, like `[{'img': np.array, 'kpts': coordinates of kpts}, ...]`

perm_mat_dict: ground truth, like `{(0,1):scipy.sparse, (0,2):scipy.sparse, ...}`, *(0,1)* refers to data pair *(ids[0],ids[1])*

ids: list of image ID

`get_id_combination`(*cls=None*, *num=2*)

Get the combination of images and length of combinations in specified class.

Parameters

- **cls** – int or str, class of expected data. None for all classes
- **num** – int, number of images in each image ID list; for example, 2 for 2GM

Returns

id_combination_list: list of combinations of image ids

length: length of combinations

`rand_get_data`(*cls=None*, *num=2*, *test=False*, *shuffle=True*)

Randomly fetch data for training or test. Implemented by calling `get_data` function.

Parameters

- **cls** – int or str, class of expected data. None for random class
- **num** – int, number of images; for example, 2 for 2GM
- **test** – bool, whether the fetched data is used for test; if true, this function will not return ground truth
- **shuffle** – bool, whether to shuffle the order of keypoints

Returns

data_list: list of data, like `[{'img': np.array, 'kpts': coordinates of kpts}, ...]`

perm_mat_dict: ground truth, like `{(0,1):scipy.sparse, (0,2):scipy.sparse, ...}`, *(0,1)* refers to data pair *(ids[0],ids[1])*

ids: list of image ID

rm_gt_cache(*last_epoch=False*)
Remove ground truth cache.

Parameters **last_epoch** – Boolean variable, whether this epoch is last epoch; if true, the directory of cache will also be removed.

4.5 pygmtools.dataset

Classes

CUB2011	This class is defined to download and preprocess CUB2011 dataset.
IMC_PT_SparseGM	This class is defined to download and preprocess IMC_PT_SparseGM dataset.
PascalVOC	This class is defined to download and preprocess PascalVOC dataset.
SPair71k	This class is defined to download and preprocess SPair71k dataset.
WillowObject	This class is defined to download and preprocess WillowObject dataset.

4.5.1 CUB2011

class pygmtools.dataset.**CUB2011**(*sets*, *obj_resize*, *ds_dict=None*, ***args*)
This class is defined to download and preprocess CUB2011 dataset.

Parameters

- **sets** – str, problem set, 'train' for training set and 'test' for test set
- **obj_resize** – tuple, resized image size
- **ds_dict** – settings of dataset, containing at most 1 param(key) for CUB2011:

  CLS_SPLIT: str, 'ori' (original split), 'sup' (super class) or 'all' (all birds as one class)

download(*url=None*)
Automatically download CUB2011 dataset.

Parameters **url** – str, web url of CUB2011

process()
Process the dataset and generate 'data.json' for preprocessed dataset, 'train.json' for training set, and 'test.json' for test set.

4.5.2 IMC_PT_SparseGM

class pygmtools.dataset.**IMC_PT_SparseGM**(*sets*, *obj_resize*, *ds_dict=None*, ***args*)
This class is defined to download and preprocess IMC_PT_SparseGM dataset.

Parameters

- **sets** – str, problem set, 'train' for training set and 'test' for test set
- **obj_resize** – tuple, resized image size
- **ds_dict** – settings of dataset, containing at most 1 param(key) for IMC_PT_SparseGM:

  TOTAL_KPT_NUM: int, maximum kpt_num in an image

download(*url=None*)
Automatically download IMC_PT_SparseGM dataset.

Parameters **url** – str, web url of IMC_PT_SparseGM

process()
Process the dataset and generate 'data.json' for preprocessed dataset, 'train.json' for training set, and 'test.json' for test set.

4.5.3 PascalVOC

class pygmtools.dataset.**PascalVOC**(*sets*, *obj_resize*, ***args*)
This class is defined to download and preprocess PascalVOC dataset.

Parameters

- **sets** – str, problem set, 'train' for training set and 'test' for test set
- **obj_resize** – tuple, resized image size

download(*url=None*, *name=None*)
Automatically download PascalVOC dataset.

Parameters

- **url** – str, web url of PascalVOC and PascalVOC annotation
- **name** – str, "PascalVOC" to download PascalVOC and "PascalVOC_anno" to download PascalVOC annotation

process()
Process the dataset and generate 'data.json' for preprocessed dataset, 'train.json' for training set, and 'test.json' for test set.

4.5.4 SPair71k

class pygmtools.dataset.**SPair71k**(*sets*, *obj_resize*, *problem='2GM'*, *ds_dict=None*, ***args*)
This class is defined to download and preprocess SPair71k dataset.

Parameters

- **sets** – str, problem set, 'train' for training set and 'test' for test set
- **obj_resize** – tuple, resized image size
- **problem** – str, problem type, only '2GM' in SPair71k

- **ds_dict** – settings of dataset, containing at most 4 params(keys) for SPair71k:

  TRAIN_DIFF_PARAMS: list of images that should be dumped in train set

  EVAL_DIFF_PARAMS: list of images that should be dumped in test set

  COMB_CLS: bool, whether to combine images in different classes

  SIZE: str, 'large' for SPair71k-large and 'small' for SPair71k-small

download(*url=None*)
Automatically download SPair71k dataset.

Parameters **url** – str, web url of SPair71k

process()
Process the dataset and generate 'data.json' for preprocessed dataset, 'train.json' for training set, and 'test.json' for test set.

4.5.5 WillowObject

class pygmtools.dataset.**WillowObject**(*sets*, *obj_resize*, *ds_dict=None*, ***args*)
This class is defined to download and preprocess WillowObject dataset.

Parameters

- **sets** – str, problem set, 'train' for training set and 'test' for test set
- **obj_resize** – tuple, resized image size
- **ds_dict** – settings of dataset, containing at most 4 params(keys) for WillowObject:

  TRAIN_NUM: int, number of images for train in each class

  SPLIT_OFFSET: int, offset when split train and test set

  TRAIN_SAME_AS_TEST: bool, whether to use same images for training and test

  RAND_OUTLIER: int, number of added outliers in one image

download(*url=None*)
Automatically download WillowObject dataset.

Parameters **url** – str, web url of WillowObject

process()
Process the dataset and generate 'data.json' for preprocessed dataset, 'train.json' for training set, and 'test.json' for test set.

4.6 pygmtools.classic_solvers

Functions

hungarian	Solve optimal LAP permutation by hungarian algorithm.
ipfp	Integer Projected Fixed Point (IPFP) method for graph matching (QAP).
rrwm	Reweighted Random Walk Matching (RRWM) solver for graph matching (QAP).

continues on next page

Table 3 – continued from previous page

sinkhorn	Sinkhorn algorithm turns the input matrix into a doubly-stochastic matrix.
sm	Spectral Graph Matching solver for graph matching (QAP).

4.6.1 pygmtools.classic_solvers.hungarian

pygmtools.classic_solvers.**hungarian**(*s*, *n1=None*, *n2=None*, *nproc: int = 1*, *backend=None*)

Solve optimal LAP permutation by hungarian algorithm. The time cost is $O(n^3)$.

Parameters

- **s** – $(b \times n_1 \times n_2)$ input 3d tensor. b: batch size. Non-batched input is also supported if `s` is of size $(n_1 \times n_2)$
- **n1** – (b) (optional) number of objects in dim1
- **n2** – (b) (optional) number of objects in dim2
- **nproc** – (default: 1, i.e. no parallel) number of parallel processes
- **backend** – (default: `pygmtools.BACKEND` variable) the backend for computation.

Returns $(b \times n_1 \times n_2)$ optimal permutation matrix

---

Note: The parallelization is based on multi-processing workers that run on multiple CPU cores.

Note: For all backends, `scipy.optimize.linear_sum_assignment` is called to solve the LAP, therefore the computation is based on `numpy` and `scipy`. The `backend` argument of this function only affects the input-output data type.

Note: We support batched instances with different number of nodes, therefore `n1` and `n2` are required to specify the exact number of objects of each dimension in the batch. If not specified, we assume the batched matrices are not padded.

Example for numpy backend:

```
>>> import numpy as np
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'numpy'
>>> np.random.seed(0)

# 2-dimensional (non-batched) input
>>> s_2d = np.random.rand(5, 5)
>>> s_2d
array([[0.5488135 , 0.71518937, 0.60276338, 0.54488318, 0.4236548 ],
       [0.64589411, 0.43758721, 0.891773  , 0.96366276, 0.38344152],
       [0.79172504, 0.52889492, 0.56804456, 0.92559664, 0.07103606],
       [0.0871293 , 0.0202184 , 0.83261985, 0.77815675, 0.87001215],
       [0.97861834, 0.79915856, 0.46147936, 0.78052918, 0.11827443]])
```

(continues on next page)

(continued from previous page)

```
>>> x = pygm.hungarian(s_2d)
>>> x
array([[0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.],
       [1., 0., 0., 0., 0.]])

# 3-dimensional (batched) input
>>> s_3d = np.random.rand(3, 5, 5)
>>> n1 = n2 = np.array([3, 4, 5])
>>> x = pygm.hungarian(s_3d, n1, n2)
>>> x
array([[[0., 0., 1., 0., 0.],
        [0., 1., 0., 0., 0.],
        [1., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]],

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

       [[0., 0., 1., 0., 0.],
        [1., 0., 0., 0., 0.],
        [0., 0., 0., 0., 1.],
        [0., 1., 0., 0., 0.],
        [0., 0., 0., 1., 0.]]])
```

Example for Pytorch backend:

```
>>> import torch
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'pytorch'

# 2-dimensional (non-batched) input
>>> s_2d = torch.from_numpy(s_2d)
>>> s_2d
tensor([[0.5488, 0.7152, 0.6028, 0.5449, 0.4237],
        [0.6459, 0.4376, 0.8918, 0.9637, 0.3834],
        [0.7917, 0.5289, 0.5680, 0.9256, 0.0710],
        [0.0871, 0.0202, 0.8326, 0.7782, 0.8700],
        [0.9786, 0.7992, 0.4615, 0.7805, 0.1183]], dtype=torch.float64)
>>> x = pygm.hungarian(s_2d)
>>> x
tensor([[0., 1., 0., 0., 0.],
        [0., 0., 1., 0., 0.],
        [0., 0., 0., 1., 0.],
        [0., 0., 0., 0., 1.],
        [1., 0., 0., 0., 0.]], dtype=torch.float64)
```

(continues on next page)

(continued from previous page)

```
# 3-dimensional (batched) input
>>> s_3d = torch.from_numpy(s_3d)
>>> n1 = n2 = torch.tensor([3, 4, 5])
>>> x = pygm.hungarian(s_3d, n1, n2)
>>> x
tensor([[[0., 0., 1., 0., 0.],
         [0., 1., 0., 0., 0.],
         [1., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.]],

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

        [[0., 0., 1., 0., 0.],
         [1., 0., 0., 0., 0.],
         [0., 0., 0., 0., 1.],
         [0., 1., 0., 0., 0.],
         [0., 0., 0., 1., 0.]]], dtype=torch.float64)
```

Note: If you find this graph matching solver useful for your research, please cite:

```
@article{hungarian,
  title={Algorithms for the assignment and transportation problems},
  author={Munkres, James},
  journal={Journal of the society for industrial and applied mathematics},
  volume={5},
  number={1},
  pages={32--38},
  year={1957},
  publisher={SIAM}
}
```

4.6.2 pygmtools.classic_solvers.ipfp

pygmtools.classic_solvers.**ipfp**(*K*, *n1=None*, *n2=None*, *n1max=None*, *n2max=None*, *x0=None*, *max_iter: int = 50*, *backend=None*)

Integer Projected Fixed Point (IPFP) method for graph matching (QAP).

Parameters

- **K** – $(b \times n_1 n_2 \times n_1 n_2)$ the input affinity matrix, b: batch size.
- **n1** – (b) number of nodes in graph1 (optional if n1max is given, and all n1=n1max).
- **n2** – (b) number of nodes in graph2 (optional if n2max is given, and all n2=n2max).
- **n1max** – (b) max number of nodes in graph1 (optional if n1 is given, and n1max=max(n1)).

- **n2max** – (b) max number of nodes in graph2 (optional if n2 is given, and n2max=max(n2)).
- **x0** – $(b \times n_1 \times n_2)$ an initial matching solution for warm-start. If not given, x0 will filled with $\frac{1}{n_1 n_2}$).
- **max_iter** – (default: 50) max number of iterations in IPFP. More iterations will be lead to more accurate result, at the cost of increased inference time.
- **backend** – (default: `pygmtools.BACKEND` variable) the backend for computation.

Returns $(b \times n_1 \times n_2)$ the solved matching matrix

---

Note: We support batched instances with different number of nodes, therefore `n1` and `n2` are required to specify the exact number of objects of each dimension in the batch. If not specified, we assume the batched matrices are not padded.

Note: This solver is non-differentiable. The output is a discrete matching matrix (i.e. permutation matrix).

Example for numpy backend:

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

# Generate a batch of isomorphic graphs
>>> batch_size = 10
>>> 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)
>>> A2 = np.matmul(np.matmul(X_gt.transpose((0, 2, 1)), A1), X_gt)
>>> n1 = n2 = np.repeat([4], batch_size)

# Build affinity matrix
>>> 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, None,␣
↪n2, None, edge_aff_fn=gaussian_aff)

# Solve by IPFP
>>> X = pygm.ipfp(K, n1, n2)
>>> X[0]
array([[0., 0., 0., 1.],
       [0., 0., 1., 0.],
       [1., 0., 0., 0.],
       [0., 1., 0., 0.]])

# Accuracy
>>> (pygm.hungarian(X) * X_gt).sum() / X_gt.sum()
1.0
```

Example for Pytorch backend:

```
>>> import torch
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'pytorch'
>>> _ = torch.manual_seed(1)

# Generate a batch of isomorphic graphs
>>> batch_size = 10
>>> X_gt = torch.zeros(batch_size, 4, 4)
>>> X_gt[:, torch.arange(0, 4, dtype=torch.int64), torch.randperm(4)] = 1
>>> A1 = torch.rand(batch_size, 4, 4)
>>> A2 = torch.bmm(torch.bmm(X_gt.transpose(1, 2), A1), X_gt)
>>> n1 = torch.tensor([4] * batch_size)
>>> n2 = torch.tensor([4] * batch_size)

# Build affinity matrix
>>> 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, None,␣
↪n2, None, edge_aff_fn=gaussian_aff)

# Solve by IPFP
>>> X = pygm.ipfp(K, n1, n2)
>>> X[0]
tensor([[0., 1., 0., 0.],
        [0., 0., 0., 1.],
        [0., 0., 1., 0.],
        [1., 0., 0., 0.]])

# Accuracy
>>> (pygm.hungarian(X) * X_gt).sum() / X_gt.sum()
tensor(1.)
```

Note: If you find this graph matching solver useful in your research, please cite:

```
@article{ipfp,
  title={An integer projected fixed point method for graph matching and map␣
↪inference},
  author={Leordeanu, Marius and Hebert, Martial and Sukthankar, Rahul},
  journal={Advances in neural information processing systems},
  volume={22},
  year={2009}
}
```

4.6.3 pygmtools.classic_solvers.rrwm

pygmtools.classic_solvers.**rrwm**(*K*, *n1=None*, *n2=None*, *n1max=None*, *n2max=None*, *x0=None*, *max_iter: int = 50*, *sk_iter: int = 20*, *alpha: float = 0.2*, *beta: float = 30*, *backend=None*)

Reweighted Random Walk Matching (RRWM) solver for graph matching (QAP). This algorithm is implemented by power iteration with Sinkhorn reweighted jumps.

The official matlab implementation is available at https://cv.snu.ac.kr/research/~RRWM/

Parameters

- **K** – $(b \times n_1n_2 \times n_1n_2)$ the input affinity matrix, b: batch size.
- **n1** – (b) number of nodes in graph1 (optional if n1max is given, and all n1=n1max).
- **n2** – (b) number of nodes in graph2 (optional if n2max is given, and all n2=n2max).
- **n1max** – (b) max number of nodes in graph1 (optional if n1 is given, and n1max=max(n1)).
- **n2max** – (b) max number of nodes in graph2 (optional if n2 is given, and n2max=max(n2)).
- **x0** – $(b \times n_1 \times n_2)$ an initial matching solution for warm-start. If not given, x0 will filled with $\frac{1}{n_1n_2}$).
- **max_iter** – (default: 50) max number of iterations (i.e. number of random walk steps) in RRWM. More iterations will be lead to more accurate result, at the cost of increased inference time.
- **sk_iter** – (default: 20) max number of Sinkhorn iterations. More iterations will be lead to more accurate result, at the cost of increased inference time.
- **alpha** – (default: 0.2) the parameter controlling the importance of the reweighted jump. alpha should lie between 0 and 1. If `alpha=0`, it means no reweighted jump; if alpha=1, the reweighted jump provides all information.
- **beta** – (default: 30) the temperature parameter of exponential function before the Sinkhorn operator. `beta` should be larger than 0. A larger `beta` means more confidence in the jump. A larger `beta` will usually require a larger `sk_iter`.
- **backend** – (default: `pygmtools.BACKEND` variable) the backend for computation.

Returns $(b \times n_1 \times n_2)$ the solved matching matrix

---

Note: We support batched instances with different number of nodes, therefore `n1` and `n2` are required to specify the exact number of objects of each dimension in the batch. If not specified, we assume the batched matrices are not padded.

Note: This solver is differentiable and supports gradient back-propagation.

Warning: The solver's output is normalized with a sum of 1, which is in line with the original implementation. If a doubly- stochastic matrix is required, please call *sinkhorn()* after this. If a discrete permutation matrix is required, please call *hungarian()*. Note that the Hungarian algorithm will truncate the gradient.

Example for numpy backend:

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

# Generate a batch of isomorphic graphs
>>> batch_size = 10
>>> 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)
>>> A2 = np.matmul(np.matmul(X_gt.transpose((0, 2, 1)), A1), X_gt)
>>> n1 = n2 = np.repeat([4], batch_size)

# Build affinity matrix
>>> 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, None,␣
↪n2, None, edge_aff_fn=gaussian_aff)

# Solve by RRWM. Note that X is normalized with a sum of 1
>>> X = pygm.rrwm(K, n1, n2, beta=100)
>>> X.sum(axis=(1, 2))
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])

# Accuracy
>>> (pygm.hungarian(X) * X_gt).sum() / X_gt.sum()
1.0
```

Example for Pytorch backend:

```
>>> import torch
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'pytorch'
>>> _ = torch.manual_seed(1)

# Generate a batch of isomorphic graphs
>>> batch_size = 10
>>> X_gt = torch.zeros(batch_size, 4, 4)
>>> X_gt[:, torch.arange(0, 4, dtype=torch.int64), torch.randperm(4)] = 1
>>> A1 = torch.rand(batch_size, 4, 4)
>>> A2 = torch.bmm(torch.bmm(X_gt.transpose(1, 2), A1), X_gt)
>>> n1 = n2 = torch.tensor([4] * batch_size)

# Build affinity matrix
>>> 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, None,␣
↪n2, None, edge_aff_fn=gaussian_aff)
```

(continues on next page)

(continued from previous page)

```
# Solve by RRWM. Note that X is normalized with a sum of 1
>>> X = pygm.rrwm(K, n1, n2, beta=100)
>>> X.sum(dim=(1, 2))
tensor([1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
        1.0000])

# Accuracy
>>> (pygm.hungarian(X) * X_gt).sum() / X_gt.sum()
tensor(1.)

# This solver supports gradient back-propogation
>>> K = K.requires_grad_(True)
>>> pygm.rrwm(K, n1, n2, beta=100).sum().backward()
>>> len(torch.nonzero(K.grad))
272
```

Note: If you find this graph matching solver useful in your research, please cite:

```
@inproceedings{rrwm,
  title={Reweighted random walks for graph matching},
  author={Cho, Minsu and Lee, Jungmin and Lee, Kyoung Mu},
  booktitle={European conference on Computer vision},
  pages={492--505},
  year={2010},
  organization={Springer}
}
```

4.6.4 pygmtools.classic_solvers.sinkhorn

pygmtools.classic_solvers.**sinkhorn**(*s*, *n1=None*, *n2=None*, *dummy_row: bool = False*, *max_iter: int = 10*, *tau: float = 1.0*, *batched_operation: bool = False*, *backend=None*)

Sinkhorn algorithm turns the input matrix into a doubly-stochastic matrix.

Sinkhorn algorithm firstly applies an `exp` function with temperature τ:

$$\mathbf{S}_{i,j} = \exp\left(\frac{\mathbf{s}_{i,j}}{\tau}\right)$$

And then turns the matrix into doubly-stochastic matrix by iterative row- and column-wise normalization:

$$\mathbf{S} = \mathbf{S} \oslash (\mathbf{1}_{n_2}\mathbf{1}_{n_2}^\top \cdot \mathbf{S})$$
$$\mathbf{S} = \mathbf{S} \oslash (\mathbf{S} \cdot \mathbf{1}_{n_2}\mathbf{1}_{n_2}^\top)$$

where $\oslash$ means element-wise division, $\mathbf{1}_n$ means a column-vector with length n whose elements are all 1s.

Parameters

- **s** – $(b \times n_1 \times n_2)$ input 3d tensor. b: batch size. Non-batched input is also supported if `s` is of size $(n_1 \times n_2)$
- **n1** – (optional) (b) number of objects in dim1
- **n2** – (optional) (b) number of objects in dim2

- **dummy_row** – (default: False) whether to add dummy rows (rows whose elements are all 0) to pad the matrix to square matrix.
- **max_iter** – (default: 10) maximum iterations
- **tau** – (default: 1) the hyper parameter τ controlling the temperature
- **batched_operation** – (default: False) apply batched_operation for better efficiency (but may cause issues for back-propagation)
- **backend** – (default: `pygmtools.BACKEND` variable) the backend for computation.

Returns $(b \times n_1 \times n_2)$ the computed doubly-stochastic matrix

---

Note: `tau` is an important hyper parameter to be set for Sinkhorn algorithm. `tau` controls the distance between the predicted doubly-stochastic matrix, and the discrete permutation matrix computed by Hungarian algorithm (see *hungarian()*). Given a small `tau`, Sinkhorn performs more closely to Hungarian, at the cost of slower convergence speed and reduced numerical stability.

Note: Setting `batched_operation=True` may be preferred when you are doing inference with this module and do not need the gradient. It is assumed that `row number <= column number`. If not, the input matrix will be transposed.

Note: We support batched instances with different number of nodes, therefore `n1` and `n2` are required to specify the exact number of objects of each dimension in the batch. If not specified, we assume the batched matrices are not padded.

Note: The original Sinkhorn algorithm only works for square matrices. To handle cases where the graphs to be matched have different number of nodes, it is a common practice to add dummy rows to construct a square matrix. After the row and column normalizations, the padded rows are discarded.

Example for numpy backend:

```
>>> import numpy as np
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'numpy'
>>> np.random.seed(0)

# 2-dimensional (non-batched) input
>>> s_2d = np.random.rand(5, 5)
>>> s_2d
array([[0.5488135 , 0.71518937, 0.60276338, 0.54488318, 0.4236548 ],
       [0.64589411, 0.43758721, 0.891773  , 0.96366276, 0.38344152],
       [0.79172504, 0.52889492, 0.56804456, 0.92559664, 0.07103606],
       [0.0871293 , 0.0202184 , 0.83261985, 0.77815675, 0.87001215],
       [0.97861834, 0.79915856, 0.46147936, 0.78052918, 0.11827443]])
>>> x = pygm.sinkhorn(s_2d)
>>> x
array([[0.18880224, 0.24990915, 0.19202217, 0.16034278, 0.20892366],
       [0.18945066, 0.17240445, 0.23345011, 0.22194762, 0.18274716],
```

(continues on next page)

(continued from previous page)

```
       [0.23713583, 0.204348  , 0.18271243, 0.23114583, 0.1446579 ],
       [0.11731039, 0.1229692 , 0.23823909, 0.19961588, 0.32186549],
       [0.26730088, 0.2503692 , 0.15357619, 0.18694789, 0.1418058 ]])

# 3-dimensional (batched) input
>>> s_3d = np.random.rand(3, 5, 5)
>>> x = pygm.sinkhorn(s_3d)
>>> print('row_sum:', x.sum(2))
row_sum: [[1.         1.         1.         1.         1.        ]
 [0.99999998 1.00000002 0.99999999 1.00000003 0.99999999]
 [1.         1.         1.         1.         1.        ]]
>>> print('col_sum:', x.sum(1))
col_sum: [[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]

 # If the 3-d tensor are with different number of nodes
>>> n1 = np.array([3, 4, 5])
>>> n2 = np.array([3, 4, 5])
>>> x = pygm.sinkhorn(s_3d, n1, n2)
>>> x[0] # non-zero size: 3x3
array([[0.36665934, 0.21498158, 0.41835906, 0.        , 0.        ],
       [0.27603621, 0.44270207, 0.28126175, 0.        , 0.        ],
       [0.35730445, 0.34231636, 0.3003792 , 0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.        , 0.        ]])
>>> x[1] # non-zero size: 4x4
array([[0.28847831, 0.20583051, 0.34242091, 0.16327021, 0.        ],
       [0.22656752, 0.30153021, 0.19407969, 0.27782262, 0.        ],
       [0.25346378, 0.19649853, 0.32565049, 0.22438715, 0.        ],
       [0.23149039, 0.29614075, 0.13784891, 0.33452002, 0.        ],
       [0.        , 0.        , 0.        , 0.        , 0.        ]])
>>> x[2] # non-zero size: 5x5
array([[0.20147352, 0.19541986, 0.24942798, 0.17346397, 0.18021467],
       [0.21050732, 0.17620948, 0.18645469, 0.20384684, 0.22298167],
       [0.18319623, 0.18024007, 0.17619871, 0.1664133 , 0.29395169],
       [0.20754376, 0.2236443 , 0.19658101, 0.20570847, 0.16652246],
       [0.19727917, 0.22448629, 0.19133762, 0.25056742, 0.13632951]])

# non-squared input
>>> s_non_square = np.random.rand(4, 5)
>>> x = pygm.sinkhorn(s_non_square, dummy_row=True) # set dummy_row=True for non-
↪squared cases
>>> print('row_sum:', x.sum(1), 'col_sum:', x.sum(0))
row_sum: [1. 1. 1. 1.] col_sum: [0.78239609 0.80485526 0.80165627 0.80004254 0.
↪81104984]
```

Example for Pytorch backend:

```
>>> import torch
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'pytorch'
```

(continues on next page)

(continued from previous page)

```
# 2-dimensional (non-batched) input
>>> s_2d = torch.from_numpy(s_2d)
>>> s_2d
tensor([[0.5488, 0.7152, 0.6028, 0.5449, 0.4237],
        [0.6459, 0.4376, 0.8918, 0.9637, 0.3834],
        [0.7917, 0.5289, 0.5680, 0.9256, 0.0710],
        [0.0871, 0.0202, 0.8326, 0.7782, 0.8700],
        [0.9786, 0.7992, 0.4615, 0.7805, 0.1183]], dtype=torch.float64)
>>> x = pygm.sinkhorn(s_2d)
>>> x
tensor([[0.1888, 0.2499, 0.1920, 0.1603, 0.2089],
        [0.1895, 0.1724, 0.2335, 0.2219, 0.1827],
        [0.2371, 0.2043, 0.1827, 0.2311, 0.1447],
        [0.1173, 0.1230, 0.2382, 0.1996, 0.3219],
        [0.2673, 0.2504, 0.1536, 0.1869, 0.1418]], dtype=torch.float64)
>>> print('row_sum:', x.sum(1), 'col_sum:', x.sum(0))
row_sum: tensor([1.0000, 1.0000, 1.0000, 1.0000, 1.0000], dtype=torch.float64) col_
↪sum: tensor([1.0000, 1.0000, 1.0000, 1.0000, 1.0000], dtype=torch.float64)

# 3-dimensional (batched) input
>>> s_3d = torch.from_numpy(s_3d)
>>> x = pygm.sinkhorn(s_3d)
>>> print('row_sum:', x.sum(2))
row_sum: tensor([[1.0000, 1.0000, 1.0000, 1.0000, 1.0000],
        [1.0000, 1.0000, 1.0000, 1.0000, 1.0000],
        [1.0000, 1.0000, 1.0000, 1.0000, 1.0000]], dtype=torch.float64)
>>> print('col_sum:', x.sum(1))
col_sum: tensor([[1.0000, 1.0000, 1.0000, 1.0000, 1.0000],
        [1.0000, 1.0000, 1.0000, 1.0000, 1.0000],
        [1.0000, 1.0000, 1.0000, 1.0000, 1.0000]], dtype=torch.float64)

# If the 3-d tensor are with different number of nodes
>>> n1 = torch.tensor([3, 4, 5])
>>> n2 = torch.tensor([3, 4, 5])
>>> x = pygm.sinkhorn(s_3d, n1, n2)
>>> x[0] # non-zero size: 3x3
tensor([[0.3667, 0.2150, 0.4184, 0.0000, 0.0000],
        [0.2760, 0.4427, 0.2813, 0.0000, 0.0000],
        [0.3573, 0.3423, 0.3004, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.0000, 0.0000, 0.0000]], dtype=torch.float64)
>>> x[1] # non-zero size: 4x4
tensor([[0.2885, 0.2058, 0.3424, 0.1633, 0.0000],
        [0.2266, 0.3015, 0.1941, 0.2778, 0.0000],
        [0.2535, 0.1965, 0.3257, 0.2244, 0.0000],
        [0.2315, 0.2961, 0.1378, 0.3345, 0.0000],
        [0.0000, 0.0000, 0.0000, 0.0000, 0.0000]], dtype=torch.float64)
>>> x[2] # non-zero size: 5x5
tensor([[0.2015, 0.1954, 0.2494, 0.1735, 0.1802],
        [0.2105, 0.1762, 0.1865, 0.2038, 0.2230],
        [0.1832, 0.1802, 0.1762, 0.1664, 0.2940],
```

(continues on next page)

(continued from previous page)

```
        [0.2075, 0.2236, 0.1966, 0.2057, 0.1665],
        [0.1973, 0.2245, 0.1913, 0.2506, 0.1363]], dtype=torch.float64)

# non-squared input
>>> s_non_square = torch.from_numpy(s_non_square)
>>> x = pygm.sinkhorn(s_non_square, dummy_row=True) # set dummy_row=True for non-
↪squared cases
>>> print('row_sum:', x.sum(1), 'col_sum:', x.sum(0))
row_sum: tensor([1.0000, 1.0000, 1.0000, 1.0000], dtype=torch.float64) col_sum:␣
↪tensor([0.7824, 0.8049, 0.8017, 0.8000, 0.8110], dtype=torch.float64)
```

Note: If you find this graph matching solver useful for your research, please cite:

```
@article{sinkhorn,
  title={Concerning nonnegative matrices and doubly stochastic matrices},
  author={Sinkhorn, Richard and Knopp, Paul},
  journal={Pacific Journal of Mathematics},
  volume={21},
  number={2},
  pages={343--348},
  year={1967},
  publisher={Mathematical Sciences Publishers}
}
```

4.6.5 pygmtools.classic_solvers.sm

pygmtools.classic_solvers.**sm**(*K*, *n1=None*, *n2=None*, *n1max=None*, *n2max=None*, *x0=None*, *max_iter: int = 50*, *backend=None*)

Spectral Graph Matching solver for graph matching (QAP). This algorithm is also known as Power Iteration method, because it works by computing the leading eigenvector of the input affinity matrix by power iteration.

For each iteration,

$$\mathbf{v}_{k+1} = \mathbf{K}\mathbf{v}_k / ||\mathbf{K}\mathbf{v}_k||_2$$

Parameters

- **K** – $(b \times n_1 n_2 \times n_1 n_2)$ the input affinity matrix, b: batch size.
- **n1** – (b) number of nodes in graph1 (optional if n1max is given, and all n1=n1max).
- **n2** – (b) number of nodes in graph2 (optional if n2max is given, and all n2=n2max).
- **n1max** – (b) max number of nodes in graph1 (optional if n1 is given, and n1max=max(n1)).
- **n2max** – (b) max number of nodes in graph2 (optional if n2 is given, and n2max=max(n2)).
- **x0** – $(b \times n_1 \times n_2)$ an initial matching solution for warm-start. If not given, x0 will be randomly generated.
- **max_iter** – (default: 50) max number of iterations. More iterations will help the solver to converge better, at the cost of increased inference time.
- **backend** – (default: `pygmtools.BACKEND` variable) the backend for computation.

Returns $(b \times n_1 \times n_2)$ the solved doubly-stochastic matrix

Example for numpy backend:

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

# Generate a batch of isomorphic graphs
>>> batch_size = 10
>>> 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)
>>> A2 = np.matmul(np.matmul(X_gt.transpose((0, 2, 1)), A1), X_gt)
>>> n1 = n2 = np.repeat([4], batch_size)

# Build affinity matrix
>>> 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, None,␣
↪n2, None, edge_aff_fn=gaussian_aff)

# Solve by SM. Note that X is normalized with a squared sum of 1
>>> X = pygm.sm(K, n1, n2)
>>> (X ** 2).sum(axis=(1, 2))
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])

# Accuracy
>>> (pygm.hungarian(X) * X_gt).sum() / X_gt.sum()
1.0
```

Example for Pytorch backend:

```
>>> import torch
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'pytorch'
>>> _ = torch.manual_seed(1)

# Generate a batch of isomorphic graphs
>>> batch_size = 10
>>> X_gt = torch.zeros(batch_size, 4, 4)
>>> X_gt[:, torch.arange(0, 4, dtype=torch.int64), torch.randperm(4)] = 1
>>> A1 = torch.rand(batch_size, 4, 4)
>>> A2 = torch.bmm(torch.bmm(X_gt.transpose(1, 2), A1), X_gt)
>>> n1 = n2 = torch.tensor([4] * batch_size)

# Build affinity matrix
>>> conn1, edge1, ne1 = pygm.utils.dense_to_sparse(A1)
>>> conn2, edge2, ne2 = pygm.utils.dense_to_sparse(A2)
>>> import functools
```

(continues on next page)

(continued from previous page)

```
>>> 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, None,␣
↪n2, None, edge_aff_fn=gaussian_aff)

# Solve by SM. Note that X is normalized with a squared sum of 1
>>> X = pygm.sm(K, n1, n2)
>>> (X ** 2).sum(dim=(1, 2))
tensor([1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
        1.0000])

# Accuracy
>>> (pygm.hungarian(X) * X_gt).sum() / X_gt.sum()
tensor(1.)

# This solver supports gradient back-propogation
>>> K = K.requires_grad_(True)
>>> pygm.sm(K, n1, n2).sum().backward()
>>> len(torch.nonzero(K.grad))
2560
```

Note: If you find this graph matching solver useful for your research, please cite:

```
@inproceedings{sm,
  title={A spectral technique for correspondence problems using pairwise␣
↪constraints},
  author={Leordeanu, Marius and Hebert, Martial},
  year={2005},
  pages={1482-1489},
  booktitle={International Conference on Computer Vision},
  publisher={IEEE}
}
```

4.7 pygmtools.multi_graph_solvers

Functions

cao	Composition based Affinity Optimization (CAO) solver for multi-graph matching.
gamgm	Graduated Assignment-based multi-graph matching solver.
mgm_floyd	Multi-Graph Matching based on Floyd shortest path algorithm.

4.7.1 pygmtools.multi_graph_solvers.cao

pygmtools.multi_graph_solvers.**cao**(*K*, *x0=None*, *qap_solver=None*, *mode='accu'*, *max_iter=6*, *lambda_init=0.3*, *lambda_step=1.1*, *lambda_max=1.0*, *iter_boost=2*, *backend=None*)

Composition based Affinity Optimization (CAO) solver for multi-graph matching. This solver builds a super-graph for matching update to incorporate the two aspects by optimizing the affinity score, meanwhile gradually infusing the consistency.

Each update step is described as follows:

$$\arg\max_k (1-\lambda) J(\mathbf{X}_{ik}\mathbf{X}_{kj}) + \lambda C_p(\mathbf{X}_{ik}\mathbf{X}_{kj})$$

where $J(\mathbf{X}_{ik}\mathbf{X}_{kj})$ is the objective score, and $C_p(\mathbf{X}_{ik}\mathbf{X}_{kj})$ measures a consistency score compared to other matchings. These two terms are balanced by λ, and λ starts from a smaller number and gradually grows.

Parameters

- **K** – $(m \times m \times n^2 \times n^2)$ the input affinity matrix, where `K[i,j]` is the affinity matrix of graph `i` and graph `j` (m: number of nodes)
- **x0** – (optional) $(m \times m \times n \times n)$ the initial two-graph matching result, where `X[i,j]` is the matching matrix result of graph `i` and graph `j`. If this argument is not given, `qap_solver` will be used to compute the two-graph matching result.
- **qap_solver** – (default: pygm.rrwm) a function object that accepts a batched affinity matrix and returns the matching matrices. It is suggested to use `functools.partial` and the QAP solvers provided in the *classic_solvers* module (see examples below).
- **mode** – (default: `'accu'`) the operation mode of this algorithm. Options: `'accu'`, `'c'`, `'fast'`, `'pc'`, where `'accu'` is equivalent to `'c'` (accurate version) and `'fast'` is equivalent to `'pc'` (fast version).
- **max_iter** – (default: 6) max number of iterations
- **lambda_init** – (default: 0.3) initial value of λ, with $\lambda \in [0, 1]$
- **lambda_step** – (default: 1.1) the increase step size of λ, updated by `lambda = step * lambda`
- **lambda_max** – (default: 1.0) the max value of lambda
- **iter_boost** – (default: 2) to boost the convergence of the CAO algorithm, λ will be forced to update every `iter_boost` iterations.
- **backend** – (default: `pygmtools.BACKEND` variable) the backend for computation.

Returns $(m \times m \times n \times n)$ the multi-graph matching result

---

Note: The input graphs must have the same number of nodes for this algorithm to work correctly.

Note: Multi-graph matching methods process all graphs at once and do not support the additional batch dimension. Please note that this behavior is different from two-graph matching solvers in *classic_solvers*.

Example for Pytorch backend:

```
>>> import torch
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'pytorch'
>>> _ = torch.manual_seed(1)

# Generate 10 isomorphic graphs
>>> graph_num = 10
>>> As, X_gt = pygm.utils.generate_isomorphic_graphs(node_num=4, graph_num=10)
>>> As_1, As_2 = [], []
>>> for i in range(graph_num):
...     for j in range(graph_num):
...         As_1.append(As[i])
...         As_2.append(As[j])
>>> As_1 = torch.stack(As_1, dim=0)
>>> As_2 = torch.stack(As_2, dim=0)

# Build affinity matrix
>>> conn1, edge1, ne1 = pygm.utils.dense_to_sparse(As_1)
>>> conn2, edge2, ne2 = pygm.utils.dense_to_sparse(As_2)
>>> 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, None, None,
↪ None, None, edge_aff_fn=gaussian_aff)
>>> K = K.reshape(graph_num, graph_num, 4*4, 4*4)
>>> K.shape
torch.Size([10, 10, 16, 16])

# Solve the multi-matching problem
>>> X = pygm.cao(K)
>>> (X * X_gt).sum() / X_gt.sum()
tensor(1.)

# Use the IPFP solver for two-graph matching
>>> ipfp_func = functools.partial(pygmtools.ipfp, n1max=4, n2max=4)
>>> X = pygm.cao(K, qap_solver=ipfp_func)
>>> (X * X_gt).sum() / X_gt.sum()
tensor(1.)

# Run the faster version of CAO algorithm
>>> X = pygm.cao(K, mode='fast')
>>> (X * X_gt).sum() / X_gt.sum()
tensor(1.)
```

Note: If you find this graph matching solver useful in your research, please cite:

```
@article{cao,
  title={Multi-graph matching via affinity optimization with graduated consistency␣
↪regularization},
  author={Yan, Junchi and Cho, Minsu and Zha, Hongyuan and Yang, Xiaokang and Chu,␣
↪Stephen M},
  journal={IEEE transactions on pattern analysis and machine intelligence},
```

(continues on next page)

(continued from previous page)

```
  volume={38},
  number={6},
  pages={1228--1242},
  year={2015},
  publisher={IEEE}
}
```

4.7.2 pygmtools.multi_graph_solvers.gamgm

pygmtools.multi_graph_solvers.**gamgm**(*A*, *W*, *ns=None*, *n_univ=None*, *U0=None*, *sk_init_tau=0.5*, *sk_min_tau=0.1*, *sk_gamma=0.8*, *sk_iter=20*, *max_iter=100*, *param_lambda=1.0*, *verbose=False*, *backend=None*)

Graduated Assignment-based multi-graph matching solver. Graduated assignment is a classic approach for hard assignment problems like graph matching, based on graduated annealing of Sinkhorn's temperature τ to enforce the matching constraint.

The objective score is described as

$$\max_{\mathbf{X}_{i,j},i,j\in[m]} \sum_{i,j\in[m]} \left(\lambda \operatorname{tr}(\mathbf{X}_{ij}^\top \mathbf{A}_i \mathbf{X}_{ij} \mathbf{A}_j) + \operatorname{tr}(\mathbf{X}_{ij}^\top \mathbf{W}_{ij})\right)$$

Once the algorithm converges at a fixed τ value, τ shrinks as:

$$\tau = \tau \times \gamma$$

and the iteration continues. At last, Hungarian algorithm is applied to ensure the result is a permutation matrix.

Note: This algorithm is based on the Koopmans-Beckmann's QAP formulation and you should input the adjacency matrices `A` and node-wise similarity matrices `W` instead of the affinity matrices.

Parameters

- **A** – $(m \times n \times n)$ the adjacency matrix (m: number of nodes). The graphs may have different number of nodes (specified by the `ns` argument).
- **W** – $(m \times m \times n \times n)$ the node-wise similarity matrix, where `W[i,j]` is the similarity matrix
- **ns** – (optional) (m) the number of nodes. If not given, it will be inferred based on the size of `A`.
- **n_univ** – (optional) the size of the universe node set. If not given, it will be the largest number of nodes.
- **U0** – (optional) the initial multi-graph matching result. If not given, it will be randomly initialized.
- **sk_init_tau** – (default: 0.05) initial value of τ for Sinkhorn algorithm
- **sk_min_tau** – (default: 1.0e-3) minimal value of τ for Sinkhorn algorithm
- **sk_gamma** – (default: 0.8) the shrinking parameter of τ: $\tau = \tau \times \gamma$
- **sk_iter** – (default: 200) max number of iterations for Sinkhorn algorithm
- **max_iter** – (default: 1000) max number of iterations for graduated assignment

- **param_lambda** – (default: 1) the weight λ of the quadratic term
- **verbose** – (default: False) print verbose information for parameter tuning
- **backend** – (default: `pygmtools.BACKEND` variable) the backend for computation.

Returns the multi-graph matching result (a *MultiMatchingResult* object)

Note: Set `verbose=True` may help you tune the parameters.

Example for Pytorch backend:

```
>>> import torch
>>> import pygmtools as pygm
>>> import itertools
>>> import time
>>> pygm.BACKEND = 'pytorch'
>>> _ = torch.manual_seed(1)

# Generate 10 isomorphic graphs
>>> graph_num = 10
>>> As, X_gt, Fs = pygm.utils.generate_isomorphic_graphs(node_num=4, graph_num=10,␣
↪node_feat_dim=20)

# Compute node-wise similarity by inner-product and Sinkhorn
>>> W = torch.matmul(Fs.unsqueeze(1), Fs.transpose(1, 2).unsqueeze(0))
>>> W = pygm.sinkhorn(W.reshape(graph_num ** 2, 4, 4)).reshape(graph_num, graph_num,
↪ 4, 4)

# Solve the multi-matching problem
>>> X = pygm.gamgm(As, W)
>>> matched = 0
>>> for i, j in itertools.product(range(graph_num), repeat=2):
...     matched += (X[i,j] * X_gt[i,j]).sum()
>>> matched / X_gt.sum()
tensor(1.)

# This function supports graphs with different nodes (also known as partial␣
↪matching)
# In the following we ignore the last node from the last 5 graphs
>>> ns = torch.tensor([4, 4, 4, 4, 4, 3, 3, 3, 3, 3], dtype=torch.int)
>>> for i in range(graph_num):
...     As[i, ns[i]:, :] = 0
...     As[i, :, ns[i]:] = 0
>>> for i, j in itertools.product(range(graph_num), repeat=2):
...     X_gt[i, j, ns[i]:, :] = 0
...     X_gt[i, j, :, ns[j]:] = 0
...     W[i, j, ns[i]:, :] = 0
...     W[i, j, :, ns[j]:] = 0

# Partial matching is challenging and the following parameters are carefully tuned
>>> X = pygm.gamgm(As, W, ns, n_univ=4, sk_init_tau=.1, sk_min_tau=0.01, param_
↪lambda=0.3)
```

(continues on next page)

(continued from previous page)

```
# Check the partial matching result
>>> matched = 0
>>> for i, j in itertools.product(range(graph_num), repeat=2):
...     matched += (X[i,j] * X_gt[i, j, :ns[i], :ns[j]]).sum()
>>> matched / X_gt.sum()
tensor(1.)
```

Note: If you find this graph matching solver useful in your research, please cite:

```
@article{gamgm1,
  title={Graduated assignment algorithm for multiple graph matching based on a␣
↪common labeling},
  author={Sol{\'e}-Ribalta, Albert and Serratosa, Francesc},
  journal={International Journal of Pattern Recognition and Artificial Intelligence}
↪,
  volume={27},
  number={01},
  pages={1350001},
  year={2013},
  publisher={World Scientific}
}

@article{gamgm2,
  title={Graduated assignment for joint multi-graph matching and clustering with␣
↪application to unsupervised graph matching network learning},
  author={Wang, Runzhong and Yan, Junchi and Yang, Xiaokang},
  journal={Advances in Neural Information Processing Systems},
  volume={33},
  pages={19908--19919},
  year={2020}
}
```

This algorithm is originally proposed by paper `gamgm1`, and further improved by paper `gamgm2` to fit modern computing architectures like GPU.

4.7.3 pygmtools.multi_graph_solvers.mgm_floyd

pygmtools.multi_graph_solvers.**mgm_floyd**(*K*, *x0=None*, *qap_solver=None*, *mode='accu'*, *param_lambda=0.2*, *backend=None*)

Multi-Graph Matching based on Floyd shortest path algorithm. A supergraph is considered by regarding each input graph as a node, and the matching between graphs are regraded as edges in the supergraph. Floyd algorithm is used to discover a shortest path on this supergraph for matching update.

The length of edges on the supergraph is described as follows:

$$\arg\max_k (1-\lambda) J(\mathbf{X}_{ik}\mathbf{X}_{kj}) + \lambda C_p(\mathbf{X}_{ik}\mathbf{X}_{kj})$$

where $J(\mathbf{X}_{ik}\mathbf{X}_{kj})$ is the objective score, and $C_p(\mathbf{X}_{ik}\mathbf{X}_{kj})$ measures a consistency score compared to other matchings. These two terms are balanced by λ.

Parameters

- **K** – $(m \times m \times n^2 \times n^2)$ the input affinity matrix, where `K[i,j]` is the affinity matrix of graph `i` and graph `j` (m: number of nodes)
- **x0** – (optional) $(m \times m \times n \times n)$ the initial two-graph matching result, where `X[i,j]` is the matching matrix result of graph `i` and graph `j`. If this argument is not given, `qap_solver` will be used to compute the two-graph matching result.
- **qap_solver** – (default: pygm.rrwm) a function object that accepts a batched affinity matrix and returns the matching matrices. It is suggested to use `functools.partial` and the QAP solvers provided in the *classic_solvers* module (see examples below).
- **mode** – (default: `'accu'`) the operation mode of this algorithm. Options: `'accu'`, `'c'`, `'fast'`, `'pc'`, where `'accu'` is equivalent to `'c'` (accurate version) and `'fast'` is equivalent to `'pc'` (fast version).
- **param_lambda** – (default: 0.3) value of λ, with $\lambda \in [0, 1]$
- **backend** – (default: `pygmtools.BACKEND` variable) the backend for computation.

Returns $(m \times m \times n \times n)$ the multi-graph matching result

Example for Pytorch backend:

```
>>> import torch
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'pytorch'
>>> _ = torch.manual_seed(1)

# Generate 10 isomorphic graphs
>>> graph_num = 10
>>> As, X_gt = pygm.utils.generate_isomorphic_graphs(node_num=4, graph_num=10)
>>> As_1, As_2 = [], []
>>> for i in range(graph_num):
...     for j in range(graph_num):
...         As_1.append(As[i])
...         As_2.append(As[j])
>>> As_1 = torch.stack(As_1, dim=0)
>>> As_2 = torch.stack(As_2, dim=0)

# Build affinity matrix
>>> conn1, edge1, ne1 = pygm.utils.dense_to_sparse(As_1)
>>> conn2, edge2, ne2 = pygm.utils.dense_to_sparse(As_2)
>>> 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, None, None,
↪ None, None, edge_aff_fn=gaussian_aff)
>>> K = K.reshape(graph_num, graph_num, 4*4, 4*4)
>>> K.shape
torch.Size([10, 10, 16, 16])

# Solve the multi-matching problem
>>> X = pygm.mgm_floyd(K)
>>> (X * X_gt).sum() / X_gt.sum()
tensor(1.)

# Use the IPFP solver for two-graph matching
```

(continues on next page)

(continued from previous page)

```
>>> ipfp_func = functools.partial(pygmtools.ipfp, n1max=4, n2max=4)
>>> X = pygm.mgm_floyd(K, qap_solver=ipfp_func)
>>> (X * X_gt).sum() / X_gt.sum()
tensor(1.)

# Run the faster version of CAO algorithm
>>> X = pygm.mgm_floyd(K, mode='fast')
>>> (X * X_gt).sum() / X_gt.sum()
tensor(1.)
```

Note: If you find this graph matching solver useful in your research, please cite:

```
@article{mgm_floyd,
  title={Unifying offline and online multi-graph matching via finding shortest␣
↪paths on supergraph},
  author={Jiang, Zetian and Wang, Tianzhe and Yan, Junchi},
  journal={IEEE transactions on pattern analysis and machine intelligence},
  volume={43},
  number={10},
  pages={3648--3663},
  year={2020},
  publisher={IEEE}
}
```

4.8 pygmtools.utils

Functions

build_aff_mat	Build affinity matrix for graph matching from input node/edge features.
build_batch	Build a batched tensor from a list of tensors.
compute_affinity_score	Compute the affinity score of graph matching.
dense_to_sparse	Convert a dense connectivity/adjacency matrix to a sparse connectivity/adjacency matrix and an edge weight tensor.
from_numpy	Convert a numpy ndarray to a tensor.
gaussian_aff_fn	Gaussian kernel affinity function.
generate_isomorphic_graphs	Generate a set of isomorphic graphs, for testing purposes and examples.
inner_prod_aff_fn	Inner product affinity function.
to_numpy	Convert a tensor to a numpy ndarray.

4.8.1 pygmtools.utils.build_aff_mat

pygmtools.utils.**build_aff_mat**(*node_feat1*, *edge_feat1*, *connectivity1*, *node_feat2*, *edge_feat2*, *connectivity2*, *n1=None*, *ne1=None*, *n2=None*, *ne2=None*, *node_aff_fn=None*, *edge_aff_fn=None*, *backend=None*)

Build affinity matrix for graph matching from input node/edge features. The affinity matrix encodes both node-wise and edge-wise affinities and formulates the Quadratic Assignment Problem (QAP), which is the mathematical form of graph matching.

Parameters

- **node_feat1** – $(b \times n_1 \times f_{node})$ the node feature of graph1
- **edge_feat1** – $(b \times ne_1 \times f_{edge})$ the edge feature of graph1
- **connectivity1** – $(b \times ne_1 \times 2)$ sparse connectivity information of graph 1. `connectivity1[i, j, 0]` is the starting node index of edge `j` at batch `i`, and `connectivity1[i, j, 1]` is the ending node index of edge `j` at batch `i`
- **node_feat2** – $(b \times n_2 \times f_{node})$ the node feature of graph2
- **edge_feat2** – $(b \times ne_2 \times f_{edge})$ the edge feature of graph2
- **connectivity2** – $(b \times ne_2 \times 2)$ sparse connectivity information of graph 2. `connectivity2[i, j, 0]` is the starting node index of edge `j` at batch `i`, and `connectivity2[i, j, 1]` is the ending node index of edge `j` at batch `i`
- **n1** – (b) number of nodes in graph1. If not given, it will be inferred based on the shape of `node_feat1` or the values in `connectivity1`
- **ne1** – (b) number of edges in graph1. If not given, it will be inferred based on the shape of `edge_feat1`
- **n2** – (b) number of nodes in graph2. If not given, it will be inferred based on the shape of `node_feat2` or the values in `connectivity2`
- **ne2** – (b) number of edges in graph2. If not given, it will be inferred based on the shape of `edge_feat2`
- **node_aff_fn** – (default: inner_prod_aff_fn) the node affinity function with the characteristic `node_aff_fn(2D Tensor, 2D Tensor) -> 2D Tensor`, which accepts two node feature tensors and outputs the node-wise affinity tensor. See *inner_prod_aff_fn()* as an example.
- **edge_aff_fn** – (default: inner_prod_aff_fn) the edge affinity function with the characteristic `edge_aff_fn(2D Tensor, 2D Tensor) -> 2D Tensor`, which accepts two edge feature tensors and outputs the edge-wise affinity tensor. See *inner_prod_aff_fn()* as an example.
- **backend** – (default: `pygmtools.BACKEND` variable) the backend for computation.

Returns $(b \times n_1 n_2 \times n_1 n_2)$ the affinity matrix

Example for numpy backend:

```
>>> import numpy as np
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'numpy'

# Generate a batch of graphs
>>> batch_size = 10
```

(continues on next page)

(continued from previous page)

```
>>> A1 = np.random.rand(batch_size, 4, 4)
>>> A2 = np.random.rand(batch_size, 4, 4)
>>> n1 = n2 = np.repeat([4], batch_size)

# Build affinity matrix by the default inner-product function
>>> conn1, edge1, ne1 = pygm.utils.dense_to_sparse(A1)
>>> conn2, edge2, ne2 = pygm.utils.dense_to_sparse(A2)
>>> K = pygm.utils.build_aff_mat(None, edge1, conn1, None, edge2, conn2, n1, ne1,␣
↪n2, ne2)

# Build affinity matrix by gaussian kernel
>>> import functools
>>> gaussian_aff = functools.partial(pygm.utils.gaussian_aff_fn, sigma=1.)
>>> K2 = pygm.utils.build_aff_mat(None, edge1, conn1, None, edge2, conn2, n1, ne1,␣
↪n2, ne2, edge_aff_fn=gaussian_aff)

# Build affinity matrix based on node features
>>> F1 = np.random.rand(batch_size, 4, 10)
>>> F2 = np.random.rand(batch_size, 4, 10)
>>> K3 = pygm.utils.build_aff_mat(F1, edge1, conn1, F2, edge2, conn2, n1, ne1, n2,␣
↪ne2, edge_aff_fn=gaussian_aff)

# The affinity matrices K, K2, K3 can be further processed by GM solvers
```

Example for Pytorch backend:

```
>>> import torch
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'pytorch'

# Generate a batch of graphs
>>> batch_size = 10
>>> A1 = torch.rand(batch_size, 4, 4)
>>> A2 = torch.rand(batch_size, 4, 4)
>>> n1 = n2 = torch.tensor([4] * batch_size)

# Build affinity matrix by the default inner-product function
>>> conn1, edge1, ne1 = pygm.utils.dense_to_sparse(A1)
>>> conn2, edge2, ne2 = pygm.utils.dense_to_sparse(A2)
>>> K = pygm.utils.build_aff_mat(None, edge1, conn1, None, edge2, conn2, n1, ne1,␣
↪n2, ne2)

# Build affinity matrix by gaussian kernel
>>> import functools
>>> gaussian_aff = functools.partial(pygm.utils.gaussian_aff_fn, sigma=1.)
>>> K2 = pygm.utils.build_aff_mat(None, edge1, conn1, None, edge2, conn2, n1, ne1,␣
↪n2, ne2, edge_aff_fn=gaussian_aff)

# Build affinity matrix based on node features
>>> F1 = torch.rand(batch_size, 4, 10)
>>> F2 = torch.rand(batch_size, 4, 10)
>>> K3 = pygm.utils.build_aff_mat(F1, edge1, conn1, F2, edge2, conn2, n1, ne1, n2,␣
↪ne2, edge_aff_fn=gaussian_aff)
```

(continues on next page)

(continued from previous page)

```
# The affinity matrices K, K2, K3 can be further processed by GM solvers
```

4.8.2 pygmtools.utils.build_batch

pygmtools.utils.**build_batch**(*input*, *return_ori_dim=False*, *backend=None*)

Build a batched tensor from a list of tensors. If the list of tensors are with different sizes of dimensions, it will be padded to the largest dimension.

The batched tensor and the number of original dimensions will be returned.

Parameters

- **input** – list of input tensors
- **return_ori_dim** – (default: False) return the original dimension
- **backend** – (default: `pygmtools.BACKEND` variable) the backend for computation.

Returns batched tensor, (if `return_ori_dim=True`) a list of the original dimensions

Example for numpy backend:

```
>>> import numpy as np
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'numpy'

# batched adjacency matrices
>>> A1 = np.random.rand(4, 4)
>>> A2 = np.random.rand(5, 5)
>>> A3 = np.random.rand(3, 3)
>>> batched_A, ori_shape = pygm.utils.build_batch([A1, A2, A3], return_ori_dim=True)
>>> batched_A.shape
(3, 5, 5)
>>> ori_shape
([4, 5, 3], [4, 5, 3])

# batched node features (feature dimension=10)
>>> F1 = np.random.rand(4, 10)
>>> F2 = np.random.rand(5, 10)
>>> F3 = np.random.rand(3, 10)
>>> batched_F = pygm.utils.build_batch([F1, F2, F3])
>>> batched_F.shape
(3, 5, 10)
```

Example for Pytorch backend:

```
>>> import torch
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'pytorch'

# batched adjacency matrices
>>> A1 = torch.rand(4, 4)
>>> A2 = torch.rand(5, 5)
```

(continues on next page)

(continued from previous page)

```
>>> A3 = torch.rand(3, 3)
>>> batched_A, ori_shape = pygm.utils.build_batch([A1, A2, A3], return_ori_dim=True)
>>> batched_A.shape
torch.Size([3, 5, 5])
>>> ori_shape
(tensor([4, 5, 3]), tensor([4, 5, 3]))

# batched node features (feature dimension=10)
>>> F1 = torch.rand(4, 10)
>>> F2 = torch.rand(5, 10)
>>> F3 = torch.rand(3, 10)
>>> batched_F = pygm.utils.build_batch([F1, F2, F3])
>>> batched_F.shape
torch.Size([3, 5, 10])
```

4.8.3 pygmtools.utils.compute_affinity_score

pygmtools.utils.**compute_affinity_score**(*X*, *K*, *backend=None*)

Compute the affinity score of graph matching. It is the objective score of the corresponding Quadratic Assignment Problem.

$$\texttt{vec}(\mathbf{X})^\top \mathbf{K} \texttt{vec}(\mathbf{X})$$

here `vec` means column-wise vectorization.

Parameters

- **X** – $(b \times n_1 \times n_2)$ the permutation matrix that represents the matching result
- **K** – $(b \times n_1 n_2 \times n_1 n_2)$ the affinity matrix
- **backend** – (default: `pygmtools.BACKEND` variable) the backend for computation.

Returns (b) the objective score

Note: This function also supports non-batched input if the first dimension of `X, K` is removed.

4.8.4 pygmtools.utils.dense_to_sparse

pygmtools.utils.**dense_to_sparse**(*dense_adj*, *backend=None*)

Convert a dense connectivity/adjacency matrix to a sparse connectivity/adjacency matrix and an edge weight tensor.

Parameters

- **dense_adj** – $(b \times n \times n)$ the dense adjacency matrix. This function also supports non-batched input where the batch dimension `b` is ignored
- **backend** – (default: `pygmtools.BACKEND` variable) the backend for computation.

Returns $(b \times ne \times 2)$ sparse connectivity matrix, $(b \times ne \times 1)$ edge weight tensor, (b) number of edges

Example for numpy backend:

```
>>> import numpy as np
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'numpy'
>>> np.random.seed(0)

>>> batch_size = 10
>>> A = np.random.rand(batch_size, 4, 4)
>>> A[:, np.arange(4), np.arange(4)] = 0 # remove the diagonal elements
>>> A.shape
(10, 4, 4)

>>> conn, edge, ne = pygm.utils.dense_to_sparse(A)
>>> conn.shape # connectivity: (batch x num_edge x 2)
(10, 12, 2)

>>> edge.shape # edge feature (batch x num_edge x feature_dim)
(10, 12, 1)

>>> ne
[12, 12, 12, 12, 12, 12, 12, 12, 12, 12]
```

Example for Pytorch backend:

```
>>> import torch
>>> import pygmtools as pygm
>>> pygm.BACKEND = 'pytorch'
>>> _ = torch.manual_seed(0)

>>> batch_size = 10
>>> A = torch.rand(batch_size, 4, 4)
>>> torch.diagonal(A, dim1=1, dim2=2)[:] = 0 # remove the diagonal elements
>>> A.shape
torch.Size([10, 4, 4])

>>> conn, edge, ne = pygm.utils.dense_to_sparse(A)
>>> conn.shape # connectivity: (batch x num_edge x 2)
torch.Size([10, 12, 2])

>>> edge.shape # edge feature (batch x num_edge x feature_dim)
torch.Size([10, 12, 1])

>>> ne
tensor([12, 12, 12, 12, 12, 12, 12, 12, 12, 12])
```

4.8.5 pygmtools.utils.from_numpy

pygmtools.utils.**from_numpy**(*input*, *device=None*, *backend=None*)

Convert a numpy ndarray to a tensor. This is the helper function to convert tensors across different backends via numpy.

Parameters

- **input** – input ndarray/*MultiMatchingResult*
- **device** – (default: None) the target device
- **backend** – (default: pygmtools.BACKEND variable) the backend for computation.

Returns tensor for the backend

4.8.6 pygmtools.utils.gaussian_aff_fn

pygmtools.utils.**gaussian_aff_fn**(*feat1*, *feat2*, *sigma=1.0*, *backend=None*)

Gaussian kernel affinity function. The affinity is defined as

$$\exp(-\frac{(\mathbf{f}_1 - \mathbf{f}_2)^2}{\sigma})$$

Parameters

- **feat1** – $(b \times n_1 \times f)$ the feature vectors $\mathbf{f}_1$
- **feat2** – $(b \times n_2 \times f)$ the feature vectors $\mathbf{f}_2$
- **sigma** – (default: 1) the parameter σ in Gaussian kernel
- **backend** – (default: pygmtools.BACKEND variable) the backend for computation.

Returns $(b \times n_1 \times n_2)$ element-wise Gaussian affinity matrix

4.8.7 pygmtools.utils.generate_isomorphic_graphs

pygmtools.utils.**generate_isomorphic_graphs**(*node_num*, *graph_num=2*, *node_feat_dim=0*, *backend=None*)

Generate a set of isomorphic graphs, for testing purposes and examples.

Parameters

- **node_num** – number of nodes in each graph
- **graph_num** – (default: 2) number of graphs
- **node_feat_dim** – (default: 0) number of node feature dimensions
- **backend** – (default: pygmtools.BACKEND variable) the backend for computation.

Returns

if graph_num==2, this function returns $(m \times n \times n)$ the adjacency matrix, and $(n \times n)$ the permutation matrix;

else, this function returns $(m \times n \times n)$ the adjacency matrix, and $(m \times m \times n \times n)$ the multi-matching permutation matrix

4.8.8 pygmtools.utils.inner_prod_aff_fn

pygmtools.utils.**inner_prod_aff_fn**(*feat1*, *feat2*, *backend=None*)

Inner product affinity function. The affinity is defined as

$$\mathbf{f}_1^\top \cdot \mathbf{f}_2$$

Parameters

- **feat1** – $(b \times n_1 \times f)$ the feature vectors $\mathbf{f}_1$
- **feat2** – $(b \times n_2 \times f)$ the feature vectors $\mathbf{f}_2$
- **backend** – (default: `pygmtools.BACKEND` variable) the backend for computation.

Returns $(b \times n_1 \times n_2)$ element-wise inner product affinity matrix

4.8.9 pygmtools.utils.to_numpy

pygmtools.utils.**to_numpy**(*input*, *backend=None*)

Convert a tensor to a numpy ndarray. This is the helper function to convert tensors across different backends via numpy.

Parameters

- **input** – input tensor/*MultiMatchingResult*
- **backend** – (default: `pygmtools.BACKEND` variable) the backend for computation.

Returns numpy ndarray

Classes

MultiMatchingResult	A memory-efficient class for multi-graph matching results.

4.8.10 MultiMatchingResult

class pygmtools.utils.**MultiMatchingResult**(*cycle_consistent=False*, *backend=None*)

A memory-efficient class for multi-graph matching results. For non-cycle consistent results, the dense storage for m graphs with n nodes requires a size of $(m \times m \times n \times n)$, and this implementation requires $((m-1) \times m \times n \times n/2)$. For cycle consistent result, this implementation requires only $(m \times n \times n)$.

Numpy Example:

```
>>> import numpy as np
>>> import pygmtools as pygm
>>> np.random.seed(0)
```

```
>>> X = pygm.utils.MultiMatchingResult(backend='numpy')
>>> X[0, 1] = np.zeros((4, 4))
>>> X[0, 1][np.arange(0, 4, dtype=np.int64), np.random.permutation(4)] = 1
>>> X
MultiMatchingResult:
```

(continues on next page)

(continued from previous page)

```
{'0,1': array([[0., 0., 1., 0.],
       [0., 0., 0., 1.],
       [0., 1., 0., 0.],
       [1., 0., 0., 0.]])}
>>> X[1, 0]
array([[0., 0., 0., 1.],
       [0., 0., 1., 0.],
       [1., 0., 0., 0.],
       [0., 1., 0., 0.]])
```

static from_numpy(*data*, *new_backend*)

Convert a numpy-backend MultiMatchingResult data to another backend.

Parameters

- **data** – the numpy-backend data
- **new_backend** – the target backend

Returns a new MultiMatchingResult instance for `new_backend`

from_numpy_(*new_backend*)

In-place operation for *from_numpy()*.

static to_numpy(*data*)

Convert an any-type MultiMatchingResult to numpy backend.

Parameters **data** – the any-type data

Returns a new MultiMatchingResult instance for numpy

to_numpy_()

In-place operation for *to_numpy()*.

PYTHON MODULE INDEX

p

INDEX

R

S

T

W