Note

Click here to download the full example code

How Does DGL Represent A Graph?

By the end of this tutorial you will be able to:

- Construct a graph in DGL from scratch.
- Assign node and edge features to a graph.
- Query properties of a DGL graph such as node degrees and connectivity.
- Transform a DGL graph into another graph.
- Load and save DGL graphs.

(Time estimate: 16 minutes)

DGL Graph Construction

DGL represents a directed graph as a `DGLGraph` object. You can construct a graph by specifying the number of nodes in the graph as well as the list of source and destination nodes. Nodes in the graph have consecutive IDs starting from 0.

For instance, the following code constructs a directed star graph with 5 leaves. The center node's ID is 0. The edges go from the center node to the leaves.

```python
import os

os.environ["DGLBACKEND"] = "pytorch"
import dgl
import numpy as np
import torch

g = dgl.graph(([0, 0, 0, 0, 0], [1, 2, 3, 4, 5]), num_nodes=6)
# Equivalently, PyTorch LongTensors also work.
g = dgl.graph(
    (torch.LongTensor([0, 0, 0, 0, 0]), torch.LongTensor([1, 2, 3, 4, 5])),
    num_nodes=6,
)

# You can omit the number of nodes argument if you can tell the number of nodes from the edge
list alone.
g = dgl.graph(([0, 0, 0, 0, 0], [1, 2, 3, 4, 5]))
```

Edges in the graph have consecutive IDs starting from 0, and are in the same order as the list of source and destination nodes during creation.

```
# Print the source and destination nodes of every edge.
print(g.edges())
```

Out:

```
(tensor([0, 0, 0, 0, 0]), tensor([1, 2, 3, 4, 5]))
```

Note

`DGLGraph`'s are always directed to best fit the computation pattern of graph neural networks, where the messages sent from one node to the other are often different between both directions. If you want to handle undirected graphs, you may consider treating it as a bidirectional graph. See Graph Transformations for an example of making a bidirectional graph.

Assigning Node and Edge Features to Graph

Many graph data contain attributes on nodes and edges. Although the types of node and edge attributes can be arbitrary in real world, `DGLGraph` only accepts attributes stored in tensors (with numerical contents). Consequently, an attribute of all the nodes or edges must have the same shape. In the context of deep learning, those attributes are often called *features*.

You can assign and retrieve node and edge features via `ndata` and `edata` interface.

```
# Assign a 3-dimensional node feature vector for each node.
g.ndata["x"] = torch.randn(6, 3)
# Assign a 4-dimensional edge feature vector for each edge.
g.edata["a"] = torch.randn(5, 4)
# Assign a 5x4 node feature matrix for each node.  Node and edge features in DGL can be multi-
dimensional.
g.ndata["y"] = torch.randn(6, 5, 4)

print(g.edata["a"])
```

Out:

```
tensor([[ 0.4923,  1.1030,  0.0339, -0.3949],
        [ 0.3530,  0.0209,  1.3001,  1.8026],
        [-0.4374,  0.8340,  0.5542,  0.4071],
        [-0.9745,  0.6320, -0.4542,  1.1930],
        [-0.3848, -0.7547, -0.5155, -1.5347]])
```

Note

The vast development of deep learning has provided us many ways to encode various types of attributes into numerical features. Here are some general suggestions:

- For categorical attributes (e.g. gender, occupation), consider converting them to integers or one-hot encoding.
- For variable length string contents (e.g. news article, quote), consider applying a language model.
- For images, consider applying a vision model such as CNNs.

You can find plenty of materials on how to encode such attributes into a tensor in the PyTorch Deep Learning Tutorials.

Querying Graph Structures

`DGLGraph` object provides various methods to query a graph structure.

```
print(g.num_nodes())
print(g.num_edges())
# Out degrees of the center node
print(g.out_degrees(0))
# In degrees of the center node - note that the graph is directed so the in degree should be 0.
print(g.in_degrees(0))
```

Out:

```
6
5
5
0
```

Graph Transformations

DGL provides many APIs to transform a graph to another such as extracting a subgraph:

```
# Induce a subgraph from node 0, node 1 and node 3 from the original graph.
sg1 = g.subgraph([0, 1, 3])
# Induce a subgraph from edge 0, edge 1 and edge 3 from the original graph.
sg2 = g.edge_subgraph([0, 1, 3])
```

You can obtain the node/edge mapping from the subgraph to the original graph by looking into the node feature `dgl.NID` or edge feature `dgl.EID` in the new graph.

```
# The original IDs of each node in sg1
print(sg1.ndata[dgl.NID])
# The original IDs of each edge in sg1
print(sg1.edata[dgl.EID])
# The original IDs of each node in sg2
print(sg2.ndata[dgl.NID])
# The original IDs of each edge in sg2
print(sg2.edata[dgl.EID])
```

Out:

```
tensor([0, 1, 3])
tensor([0, 2])
tensor([0, 1, 2, 4])
tensor([0, 1, 3])
```

`subgraph` and `edge_subgraph` also copies the original features to the subgraph:

```
# The original node feature of each node in sg1
print(sg1.ndata["x"])
# The original edge feature of each node in sg1
print(sg1.edata["a"])
# The original node feature of each node in sg2
print(sg2.ndata["x"])
# The original edge feature of each node in sg2
print(sg2.edata["a"])
```

Out:

```
tensor([[ 0.9192,  0.4320, -1.1305],
        [-0.0056, -0.5874,  0.1528],
        [ 0.2726, -1.6391, -0.3148]])
tensor([[ 0.4923,  1.1030,  0.0339, -0.3949],
        [-0.4374,  0.8340,  0.5542,  0.4071]])
tensor([[ 0.9192,  0.4320, -1.1305],
        [-0.0056, -0.5874,  0.1528],
        [-1.3349, -0.1582, -0.7141],
        [-1.7423,  0.4454,  1.3130]])
tensor([[ 0.4923,  1.1030,  0.0339, -0.3949],
        [ 0.3530,  0.0209,  1.3001,  1.8026],
        [-0.9745,  0.6320, -0.4542,  1.1930]])
```

Another common transformation is to add a reverse edge for each edge in the original graph with `dgl.add_reverse_edges`.

Note

If you have an undirected graph, it is better to convert it into a bidirectional graph first via adding reverse edges.

```
newg = dgl.add_reverse_edges(g)
print(newg.edges())
```

Out:

```
(tensor([0, 0, 0, 0, 0, 1, 2, 3, 4, 5]), tensor([1, 2, 3, 4, 5, 0, 0, 0, 0, 0]))
```

Loading and Saving Graphs

You can save a graph or a list of graphs via `dgl.save_graphs` and load them back with `dgl.load_graphs`.

```
# Save graphs
dgl.save_graphs("graph.dgl", g)
dgl.save_graphs("graphs.dgl", [g, sg1, sg2])

# Load graphs
(g,), _ = dgl.load_graphs("graph.dgl")
print(g)
(g, sg1, sg2), _ = dgl.load_graphs("graphs.dgl")
print(g)
print(sg1)
print(sg2)
```

Out:

```
Graph(num_nodes=6, num_edges=5,
      ndata_schemes={'y': Scheme(shape=(5, 4), dtype=torch.float32), 'x': Scheme(shape=(3,),
dtype=torch.float32)}
      edata_schemes={'a': Scheme(shape=(4,), dtype=torch.float32)})
Graph(num_nodes=6, num_edges=5,
      ndata_schemes={'y': Scheme(shape=(5, 4), dtype=torch.float32), 'x': Scheme(shape=(3,),
dtype=torch.float32)}
      edata_schemes={'a': Scheme(shape=(4,), dtype=torch.float32)})
Graph(num_nodes=3, num_edges=2,
      ndata_schemes={'_ID': Scheme(shape=(), dtype=torch.int64), 'x': Scheme(shape=(3,),
dtype=torch.float32), 'y': Scheme(shape=(5, 4), dtype=torch.float32)}
      edata_schemes={'_ID': Scheme(shape=(), dtype=torch.int64), 'a': Scheme(shape=(4,),
dtype=torch.float32)})
Graph(num_nodes=4, num_edges=3,
      ndata_schemes={'_ID': Scheme(shape=(), dtype=torch.int64), 'x': Scheme(shape=(3,),
dtype=torch.float32), 'y': Scheme(shape=(5, 4), dtype=torch.float32)}
      edata_schemes={'_ID': Scheme(shape=(), dtype=torch.int64), 'a': Scheme(shape=(4,),
dtype=torch.float32)})
```

What's next?

- See here for a list of graph structure query APIs.
- See here for a list of subgraph extraction routines.
- See here for a list of graph transformation routines.

- API reference of `dgl.save_graphs()` and `dgl.load_graphs()`

```
# Thumbnail credits: Wikipedia
# sphinx_gallery_thumbnail_path = '_static/blitz_2_dglgraph.png'
```

Total running time of the script: (0 minutes 0.023 seconds)

Download Python source code: 2_dglgraph.py

Download Jupyter notebook: 2_dglgraph.ipynb

Gallery generated by Sphinx-Gallery