1.2 Graphs, Nodes, and Edges

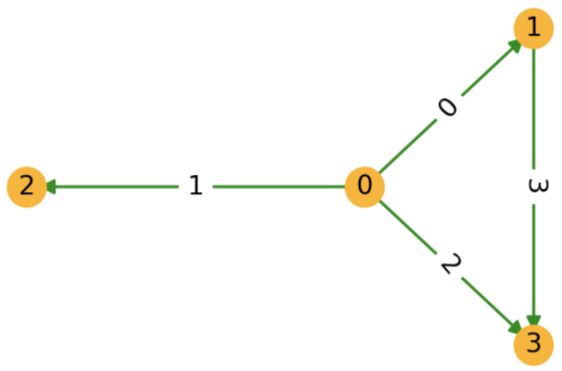
(中文版)

DGL represents each node by a unique integer, called its node ID, and each edge by a pair of integers corresponding to the IDs of its end nodes. DGL assigns to each edge a unique integer, called its **edge ID**, based on the order in which it was added to the graph. The numbering of node and edge IDs starts from 0. In DGL, all the edges are directed, and an edge (u, v) indicates that the direction goes from node u to node v.

To specify multiple nodes, DGL uses a 1-D integer tensor (i.e., PyTorch's tensor, TensorFlow's Tensor, or MXNet's ndarray) of node IDs. DGL calls this format "node-tensors". To specify multiple edges, it uses a tuple of node-tensors (U, V). $(U[i], V[i])$ decides an edge from $U[i]$ to $V[i]$.

One way to create a `DGLGraph` is to use the `dgl.graph()` method, which takes as input a set of edges. DGL also supports creating graphs from other data sources, see 1.4 Creating Graphs from External Sources.

The following code snippet uses the `dgl.graph()` method to create a `DGLGraph` corresponding to the four-node graph shown below and illustrates some of its APIs for querying the graph's structure.



```
>>> import dgl
>>> import torch as th

>>> # edges 0->1, 0->2, 0->3, 1->3
>>> u, v = th.tensor([0, 0, 0, 1]), th.tensor([1, 2, 3, 3])
>>> g = dgl.graph((u, v))
>>> print(g) # number of nodes are inferred from the max node IDs in the given edges
Graph(num_nodes=4, num_edges=4,
      ndata_schemes={}
      edata_schemes={})

>>> # Node IDs
>>> print(g.nodes())
tensor([0, 1, 2, 3])
>>> # Edge end nodes
>>> print(g.edges())
(tensor([0, 0, 0, 1]), tensor([1, 2, 3, 3]))
>>> # Edge end nodes and edge IDs
>>> print(g.edges(form='all'))
(tensor([0, 0, 0, 1]), tensor([1, 2, 3, 3]), tensor([0, 1, 2, 3]))

>>> # If the node with the largest ID is isolated (meaning no edges),
>>> # then one needs to explicitly set the number of nodes
>>> g = dgl.graph((u, v), num_nodes=8)
```

For an undirected graph, one needs to create edges for both directions. `dgl.to_bidirected()` can be helpful in this case, which converts a graph into a new one with edges for both directions.

```
>>> bg = dgl.to_bidirected(g)
>>> bg.edges()
(tensor([0, 0, 0, 1, 1, 2, 3, 3]), tensor([1, 2, 3, 0, 3, 0, 0, 1]))
```

Note

Tensor types are generally preferred throughout DGL APIs due to their efficient internal storage in C and explicit data type and device context information. However, most DGL APIs do support python iterable (e.g., list) or numpy.ndarray as arguments for quick prototyping.

DGL can use either 32- or 64-bit integers to store the node and edge IDs. The data types for the node and edge IDs should be the same. By using 64 bits, DGL can handle graphs with up to $2^{63} - 1$ nodes or edges. However, if a graph contains less than $2^{31} - 1$ nodes or edges, one should use 32-bit integers as it leads to better speed and requires less memory. DGL provides methods for making such conversions. See below for an example.

```
>>> edges = th.tensor([2, 5, 3]), th.tensor([3, 5, 0])  # edges 2->3, 5->5, 3->0
>>> g64 = dgl.graph(edges)  # DGL uses int64 by default
>>> print(g64.idtype)
torch.int64
>>> g32 = dgl.graph(edges, idtype=th.int32)  # create a int32 graph
>>> g32.idtype
torch.int32
>>> g64_2 = g32.long()  # convert to int64
>>> g64_2.idtype
torch.int64
>>> g32_2 = g64.int()  # convert to int32
>>> g32_2.idtype
torch.int32
```

See APIs: `dgl.graph()`, `dgl.DGLGraph.nodes()`, `dgl.DGLGraph.edges()`, `dgl.to_bidirected()`, `dgl.DGLGraph.int()`, `dgl.DGLGraph.long()`, and `dgl.DGLGraph.idtype`.