Mini-Project Specification - Graph Partition

Background

To store and process a large graph in the distributed machines, the graph partition will be used. A **Graph Partition** is the reduction of a graph to a smaller graph by partitioning its set of nodes into mutually exclusive groups. Edges of the original graph that cross between the groups will produce edges in the partitioned graph.

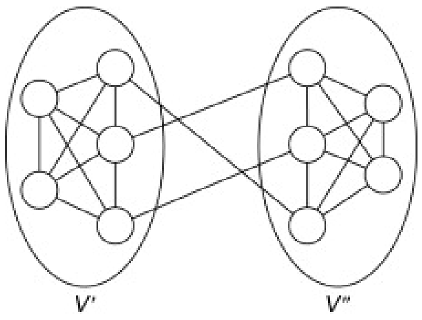


Figure 1. An example of the graph partition

Existing graph partitioning algorithms are mainly designed to reduce inter-partition edges and balance the workload. They have been widely adopted in distributed graph processing systems to reduce inter-machine communication. However, sample-based GNN training focuses on the ***K***-hop neighborhood of only the vertices in the training, validation and test sets (instead of all vertices).

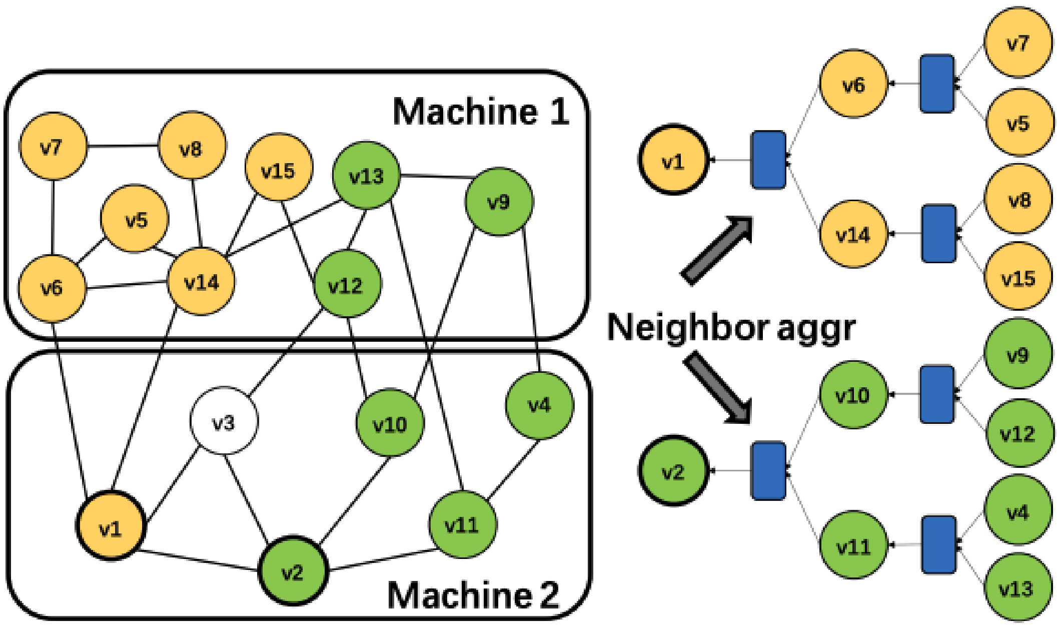
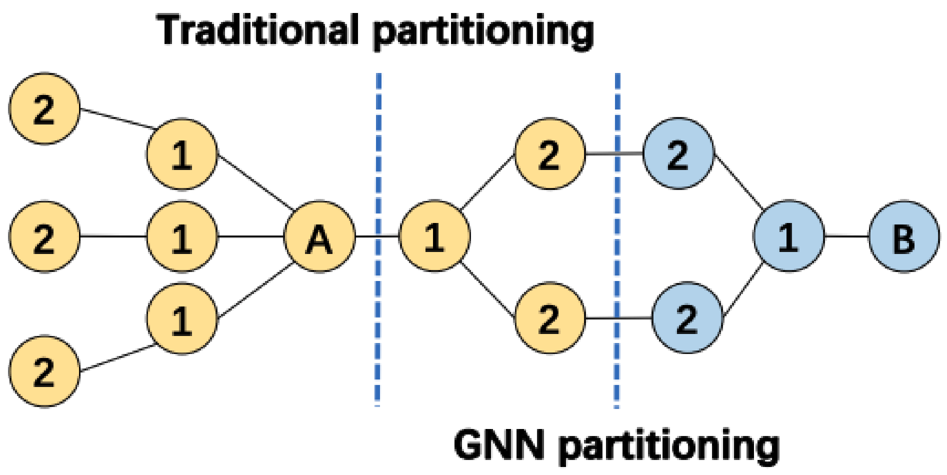


Figure 2. 2-hop mini-batch graph sampling

Figure 2 illustrates how mini-batch graph sampling is applied in the training of a 2-layered GNN model. We show the sampled 2-hop neighborhood subgroups of two seed vertices, *v*1 and *v*2, where we set the sampling configuration *D*1 = 2 and *D*2 = 2, meaning that a vertex *v* first samples at most *D*1 of its 1-hop neighbors, and then each *u* of *v*'s sampled 1-hop neighbors further samples *D*2 of *u*'s neighbors.

In the following figure, traditional partitioning strategies cut the graph into two parts by the left dotted line since it not only balances the vertices but also has the least cut edge. But for a 2-layer GNN training, since vertex ***A*** and vertex ***B*** are the labeled vertices, partitioning by the right dotted line is actually a better choice. Even if this results in two cut edges, it would not cause any data movement in the training process as only the 2-hop neighbors of the labeled vertices are required.



Algorithm Explanation

We propose a heuristic two-step graph partitioning strategy tailored for GNN sampling workloads. The main idea is to group vertices into multi-hop neighborhood-based blocks and then assign these blocks to partitions by balancing the numbers of training, validation, and test vertices in the partitions.

Step 1. Neighborhood block construction.

In order to keep the locality of data access, our partitioning algorithm first constructs a neighborhood block for each vertex in the training, validation and test sets. For each vertex *v* in these sets, *v* obtains a unique ID and then broadcasts the ID to its *K*-hop neighbors being visited by BFS. Note that each vertex only keeps the first block ID it receives. All the vertices with the same block ID will then form a neighborhood block. **We sort the blocks in descending order of their sizes and then start the assignment from the largest block.**

Step 2. Block assignment.

As our second objective is to balance the number of labeled vertices in each partition, we define two metrics to decide which partition each block belongs to. Each block is assigned to the partition with the highest product of these two metrics.

Algorithm 2 shows how to assign the blocks.

First, *Cross_Edge(i, j)* counts the number of cross-edges between *Block_i* and *Partition_i*, where these edges will be eliminated if *Block_i* is assigned to *Partition_i*. Thus, the larger *Cross_Edge(i, j)* is, the more likely *Block_i* is assigned to *Partition_i*. We also normalize *Cross_Edge(i, j)* by | *Partition_i* |, i.e., the number of total vertices in partition *j*.

The second metric is the balancing score that controls the number of training/validation/test vertices in partition *j* to be close to the average value. Considering every type of vertices, we also put weights on different types of vertices for better overall performance. The assignment phase will iterate every block and each block will be assigned to the partition with the highest product of the cross edge score and the balancing score.

```
Algorithm 2: Block Assignment
Input: List of Blocks $\mathbf{B} = B_1, B_2, \ldots.., B_n$
Output: Graph partitions $P_1, P_2, P_3, \ldots..., P_k$
for each block $B_i$ in $\mathbf{B}$ do
    for $j \leftarrow 1$ to $k$ do
        CE[j] = $|\text{Cross\_Edge}(P_j, B_i)| / |P_j|$
        BS[j] = $(1-\alpha * \frac{|P_j(train)|}{C(train)} - \beta * \frac{|P_j(val)|}{C(val)} - \gamma * \frac{|P_j(test)|}{C(test)})$
    end
    $x = \mathop{\mathbf{argmax}}_{1 \le t \le k} \{CE[t] * BS[t]\}$
    $P_x = P_x \cup B_i$
end
return $P_1, P_2, P_3, \ldots..., P_k$
```

Requirements

The implementation should include the following requirement to demonstrate your ability to accomplish efficient engineering works.

1. Customized class

You should define your own classes with appropriate constructor, methods, and variables. The organization of classes should be efficient and modularized.

```
# including but not limited to

class Graph {

};

class Block: public Graph {

};

class Partition: public Graph {

};

```

2. Multi-threading programming

In the first stage, we can obtain the blocks by a multi-source distributed BFS while each block can get scores with different partitions in parallel for the assignment. Try to use multi-threading programming to solve this procedure to accelerate the program.

3. Manage the pointer with the smartpointer.

A C++ smart pointer is a wrapper class. While the objects of this class look like regular pointers, they have additional functionalities – e.g., reference counting, automatic object destruction, and C++ memory management. Try to use smart pointer to manage the objects in the program, especially when there are multiple threads.

4. Readability

Make sure the code has good readability. You should present proper **comments**, meaningful **naming**, and consistent **code format**. We suggest using Google C++ code style.

Input and Output

Input

The original full graph data includes 5 files under the folder.

```
data_root_dir/
  |-- node_table
  |-- edge_table
  |-- train_table
  |-- val_table
  |-- test_table
```

1. Vertex file

The format of the vertex file. The first row is the column name, which represents the mandatory or extended information, separated by **tab**, and each element is "column name:data type". The rest of the data in each row represents the information of a vertex, corresponding to the information name of the first column, separated by **tab**.

To simplify, you can think there are only two data types in the node table data.

```
# file://node_table
id:int64 feature:string
0 shanghai:0:s2:10:0.1:0.5
1 beijing:1:s2:11:0.1:0.5
2 hangzhou:2:s2:12:0.1:0.5
3 shanghai:3:s2:13:0.1:0.5
```

2. Edge file

The first row is the column name, which indicates the mandatory or extended information, separated by **tab**, and each element is "column name:data type". The rest of the data in each row represents the information of an edge, corresponding to the information name of the first column, separated by **tab**.

To simplify, you can think there are only three data types in the edge table data.

```
# file://edge_table
src_id:int64 dst_id:int64 weight:float feature:string
0 5 0.215340 red:0:s2:10:0.1:0.5
0 7 0.933091 grey:0:s2:10:0.1:0.5
0 1 0.362519 blue:0:s2:10:0.1:0.5
0 9 0.097545 yellow:0:s2:10:0.1:0.5
```

3. Train vertex file

```
# file://train_table
id:int64
0
2
3
```

4. Validate vertex file

```
# file://val_table
id:int64
1
6
```

5. Test vertex file

```
# file://test_table
id:int64
7
8
9
10
```

Output

In the output, there should be one metadata to indicate which part each node belongs to.

For each part, partitioned data is stored into multiple files organized as follows:

```
data_root_dir/
  |-- metadata            # partition id of each node
  |-- part0/              # data for partition 0
      |-- node_table      # node table in this partition
      |-- edge_table      # all edges whose source node is in the partition
      |-- train_table     # train nodes in this partition
      |-- val_table       # val nodes in this partition
      |-- test_table      # test nodes in this partition
  |-- part1/             # data for partition 1
      |-- node_feats
      |-- edge_feats
      |-- train_table
      |-- val_table
      |-- test_table
  ....
```

Run Example

```
./partition input_folder output_folder partition_num alpha beta gamma
```

Reference

ByteGNN