Optimizing Graph Transformer Networks with Graph-based Techniques

Loc Hoang [*]
The University of Texas at Austin
loc@cs.utexas.edu

Udit Agarwal, **Gurbinder Gill, Roshan Dathathri**
KatanaGraph
{udit,gill,roshan}@katanagraph.com

Abhik Seal, Brian Martin
AbbVie Inc.
{abhik.seal,brian.martin}@abbvie.com

Keshav Pingali
The University of Texas at Austin
pingali@cs.utexas.edu

REVIEW ; ii tae jeong 22.07.21

Novelty

1. We present a new algorithm for the graph transformer network that formulates the problem as a series of graph operations rather than as matrix operations.

2. We present a random walk based approach that uses this graph-based formulation **to sample important metapaths to further reduce memory usage and computation cost**.

3. We implement this algorithm and show that it outperforms the original implementation by 6.5× on average. We also show that random-walk sampling improves performance by 155× over the original implementation without compromising accuracy of node classification.

4. We show experimentally that the sampling approach can run and scale on large graphs with up to 1.5 billion edges.

→ GTN(heterogeneous)’s metapath → graph pathfinding algorithm

Preliminary

METAPATH

Heterogeneous graph 의 node-node 관계를 sequence화 하여 meta 정보만을 담아 놓은 schema

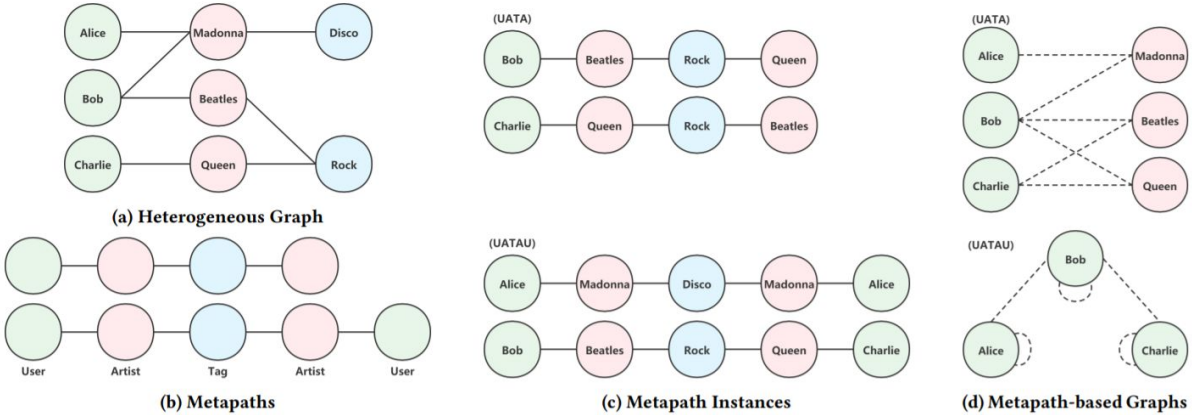


Figure 1: An illustration of the terms defined in Section 2. (a) An example heterogeneous graph with three types of nodes (i.e., users, artists, and tags). (b) The User-Artist-Tag-Artist (UATA) metapath and the User-Artist-Tag-Artist-User (UATAU) metapath. (c) Example metapath instances of the UATA and UATAU metapaths, respectively. (d) The metapath-based graphs for the UATA and UATAU metapaths, respectively.

Definition 2.1. Heterogeneous Graph. A heterogeneous graph is defined as a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ associated with a node type mapping function $\phi : \mathcal{V} \rightarrow \mathcal{A}$ and an edge type mapping function $\psi : \mathcal{E} \rightarrow \mathcal{R}$. $\mathcal{A}$ and $\mathcal{R}$ denote the predefined sets of node types and edge types, respectively, with $|\mathcal{A}| + |\mathcal{R}| > 2$.

Definition 2.2. Metapath. A metapath P is defined as a path in the form of $A_1 \xrightarrow{R_1} A_2 \xrightarrow{R_2} \cdots \xrightarrow{R_l} A_{l+1}$ (abbreviated as $A_1A_2 \cdots A_{l+1}$), which describes a composite relation $R = R_1 \circ R_2 \circ \cdots \circ R_l$ between node types A_1 and A_{l+1}, where $\circ$ denotes the composition operator on relations.

Definition 2.3. Metapath Instance. Given a metapath P of a heterogeneous graph, a metapath instance p of P is defined as a node sequence in the graph following the schema defined by P.

Definition 2.4. Metapath-based Neighbor. Given a metapath P of a heterogeneous graph, the metapath-based neighbors $\mathcal{N}_v^P$ of a node v is defined as the set of nodes that connect with node v via metapath instances of P. A neighbor connected by two different metapath instances is regarded as two different nodes in $\mathcal{N}_v^P$. Note that $\mathcal{N}_v^P$ includes v itself if P is symmetric.

source

preliminary

Graph Transformer

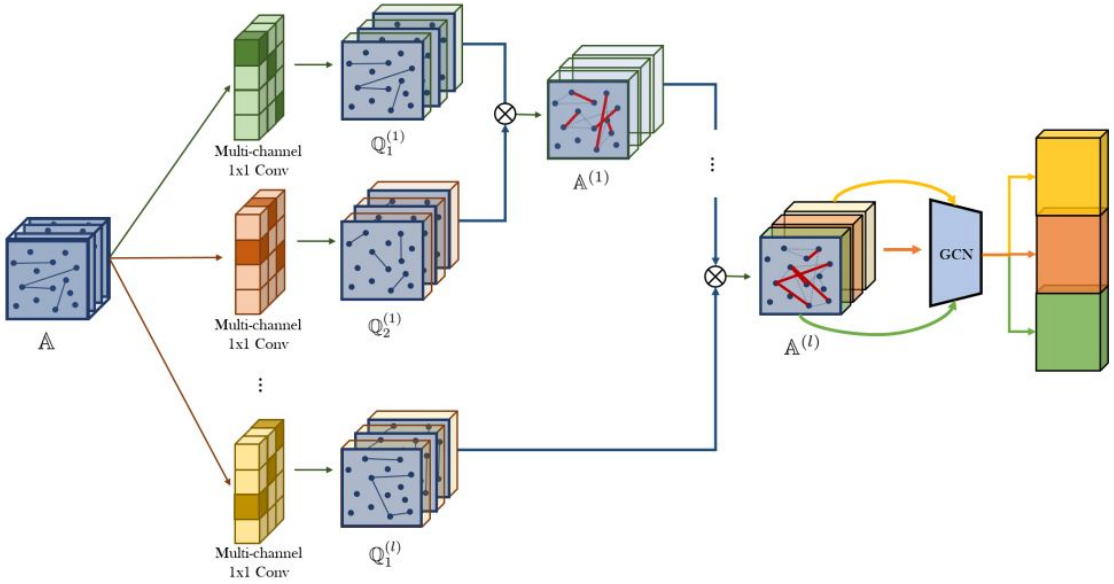


Figure 2: Graph Transformer Networks (GTNs) learn to generate a set of new meta-path adjacency matrices $\mathbb{A}^{(l)}$ using GT layers and perform graph convolution as in GCNs on the new graph structures. Multiple node representations from the same GCNs on multiple meta-path graphs are integrated by concatenation and improve the performance of node classification. $\mathbb{Q}_1^{(l)}$ and $\mathbb{Q}_2^{(l)} \in \mathbf{R}^{N \times N \times C}$ are intermediate adjacency tensors to compute meta-paths at the lth layer.

1. GT layer를 활용하여 metapath 를 생성함.
2. 생성된 metapath 들과 기존 graph adj(structure, topology)를 concat하여 new Adjacency matrix 생성
3. 앞서 생성된 matrix 를 GCN 레이어를 통과시켜 벡터화.

source code

preliminary

Graph Transformer

```python
class GTConv(nn.Module):

    def __init__(self, in_channels, out_channels):
        super(GTConv, self).__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.weight = nn.Parameter(torch.Tensor(out_channels,in_channels,1,1))
        self.bias = None
        self.scale = nn.Parameter(torch.Tensor([0.1]), requires_grad=False)
        self.reset_parameters()
    def reset_parameters(self):
        n = self.in_channels
        nn.init.constant_(self.weight, 0.1)
        if self.bias is not None:
            fan_in, _ = nn.init._calculate_fan_in_and_fan_out(self.weight)
            bound = 1 / math.sqrt(fan_in)
            nn.init.uniform_(self.bias, -bound, bound)

    def forward(self, A):
        A = torch.sum(A*F.softmax(self.weight, dim=1), dim=1)
        return A
```

```python
class GTLayer(nn.Module):

    def __init__(self, in_channels, out_channels, first=True):
        super(GTLayer, self).__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.first = first
        if self.first == True:
            self.conv1 = GTConv(in_channels, out_channels)
            self.conv2 = GTConv(in_channels, out_channels)
        else:
            self.conv1 = GTConv(in_channels, out_channels)

    def forward(self, A, H_=None):
        if self.first == True:
            a = self.conv1(A)
            b = self.conv2(A)
            H = torch.bmm(a,b)
            W = [(F.softmax(self.conv1.weight, dim=1)).detach(),(F.softmax(self.conv2.weight, dim=1)).detach()]
        else:
            a = self.conv1(A)
            H = torch.bmm(H_,a)
            W = [(F.softmax(self.conv1.weight, dim=1)).detach()]
        return H,W
```

First, GT layer softly selects two graph structures Q1 and Q2 from candidate adjacency matrices A.
Second, it learns a new graph structure by the composition of two relations (i.e., matrix multiplication of two adjacency matrices, Q1Q2)

preliminary

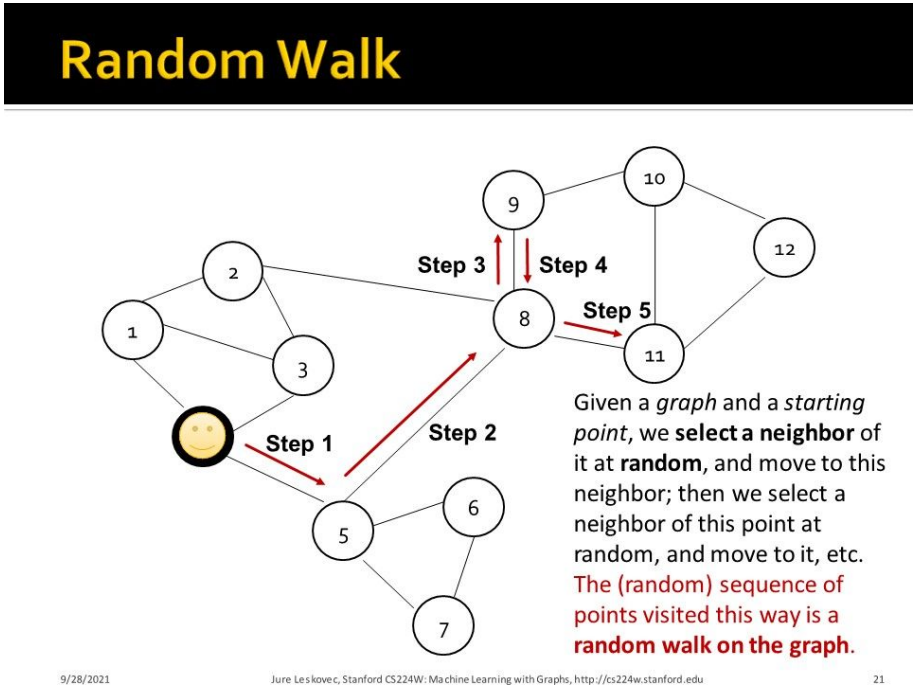
RANDOMWALK

Algorithm 4 Random Walk Algorithm

Input: Graph G; vertex v; Number of walks num_walks; Walk-length $walk\_length$
Output: Set of paths $\mathcal{P}$

$\mathcal{P} \leftarrow \phi$
for $i \leftarrow 1$ to num_walks **do**
  $p \leftarrow \{v\}$
  for $j \leftarrow 1$ to $walk\_length$ **do**
    $last \leftarrow p[j-1]$
    Randomly sample u from $\mathcal{N} \cup \{last\}$ using acceptance-rejection sampling
    $p \leftarrow p \cdot (last, u)$
  $\mathcal{P} \leftarrow \mathcal{P} \cup p$
  return $\mathcal{P}$

which takes into account **the weights of outgoing edges**. **Edges with higher weights are more likely to get picked, meaning more important metapaths** (at that point in training/inferencing) are more likely to be sampled.



novelty , keypoint

	ACM		IMDB		DBLP	
	Time (s)	Accuracy	Time (s)	Accuracy	Time (s)	Accuracy
GCN	0.05	0.92	0.04	0.57	0.03	0.89
P-GTN	12.20	0.90	32.25	0.57	73.03	0.94
G-GTN	6.88	0.90	0.26	0.57	56.68	0.95
W-GTN-10	0.06	0.90	0.10	0.54	0.11	0.90
W-GTN-50	0.14	0.92	0.17	0.59	0.32	0.94
W-GTN-100	0.20	0.92	0.24	0.60	0.57	0.94

Table 2: Average epoch time and peak accuracy across 300 epochs with GTNs with 3 graph transformer layers (i.e., metapath with up to 4 edges).

P-GTN (Pytorch implemented GTN)
G-GTN (Graph-GTN)
W-GTN-X (Walk-GTN)

X 는 random walk 를 활용해 생성된 metapath counts 를 의미함.

ex) W-GTN-100 , 100개의 metapaths generated GTN

the key computation is a dense matrix multiplication
→ find graph paths(metapath)→ dynamic programming

scoring func

traditional methodology

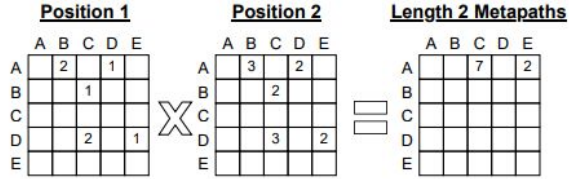


Figure 2: Example of matrix computation that occurs to find the metapaths in Fig. 1. The graph's adjacency matrix is duplicated, and the scores corresponding to each edge type for a particular on is filled accordingly. A matrix multiply then finds metapaths edges with the correct score: $(A, C) = 7$ and $(A, E) = 2$.

our approach

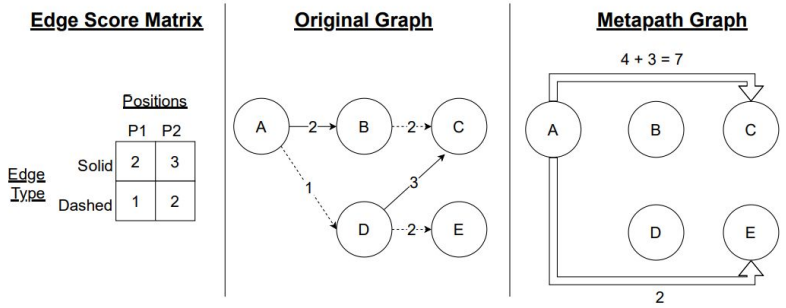


Figure 1: Metapath graph construction. The matrix denotes the importance of an edge type for a position in the metapath. The heterogeneous graph's edges are scored based on this matrix. The metapath graph has edges (A, C) and (A, E): the former is composed of (A, B, C) and (A, D, C) with scores $2 * 2 = 4$ and $1 * 3 = 3$, and the latter is composed of (A, D, E) with score $1 * 2 = 2$.

→ (O^2) cost (dense matrix) → algorithm(random-walk) based compression

architecture

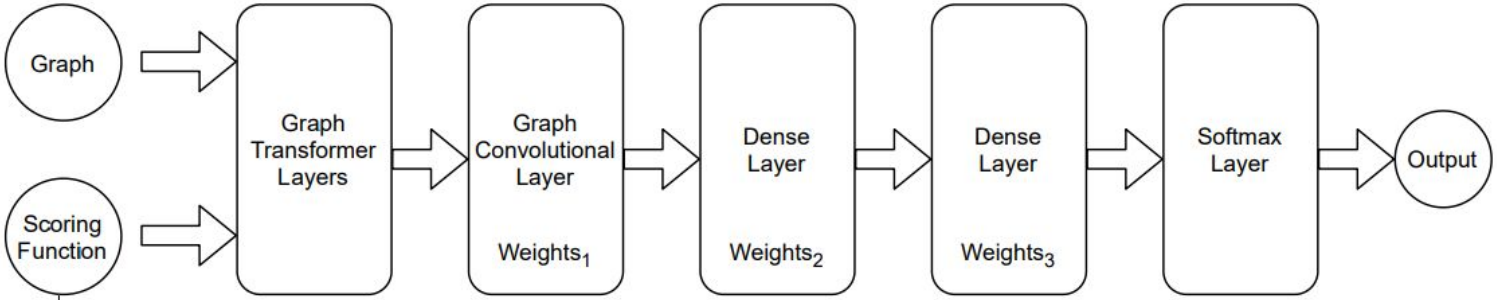


Figure 3: Illustration of the Graph Transformer Network architecture. The Graph Transformer Layers (one layer for each metapath edge) output a metapath graph for use by the GCN and dense layers.

$$A_P = \left(\sum_{t_1 \in \mathcal{T}^e} \alpha_{t_1}^{(1)} A_{t_1}\right)\left(\sum_{t_2 \in \mathcal{T}^e} \alpha_{t_2}^{(2)} A_{t_2}\right)\cdots\left(\sum_{t_l \in \mathcal{T}^e} \alpha_{t_l}^{(l)} A_{t_l}\right)$$

metapath-generation

Algorithm 1 Vanilla Metapath Graph Generation

Input: Graph G; Edge Score Function ES; Edge Type Function ET
Output: Metapath Graph $MG = (V, E, W)$

1: **for** all vertices v in G **do**
2: Enumerate length l paths P from v
3: **for** path $p = (v, x_1, ..., x_l)$ in P **do**
4: $score = ES(ET(v, x_1), 1) \cdot \prod_{i=2}^{l} ES(ET(x_{i-1}, x_i), i)$
5: add edge (v, x_l) to $MG.E$, if it already doesn't exist
6: $MG.W(v, x_l) += score$

This approach allows us to significantly reduce the total amount of computation and memory usage while also allowing us to build a metapath graph that gives comparable accuracy to original GTN formulation.
Because **we only find a constant number of paths per node, we can explicitly store them as well so recomputation of paths is not needed during the backward step.**

Algorithm 2 Metapath Graph Generation using Random Walks

Input: Graph G; Edge Score Function ES; Edge Type Function ET; Number of walks num_walks
Output: Metapath Graph $MG = (V, E, W)$

1: **for** all vertices v in G **do**
2: Sample length l num_walks paths P from v
3: **for** path $p = (v, x_1, ..., x_l)$ in P **do**
4: $score = ES(ET(v, x_1), 1) \cdot \prod_{i=2}^{l} ES(ET(x_{i-1}, x_i), i)$
5: add edge (v, x_l) to $MG.E$, if it already doesn't exist
6: $MG.W(v, x_l) += score$

metapath-generation

Algorithm 3 Metapath Graph Generation with $l/2$ Paths

Input: Graph G; Edge Score Function ES; Edge Type Function ET;
Output: Metapath Graph for first-half $l/2$ paths MG_1; Metapath Graph for second-half $l/2$ paths MG_2; Metapath Graph for full l paths MG

for all vertices v in G **do**
  Enumerate length $l/2$ paths P from v
  for path $p = (v, x_1, ..., x_{l/2})$ in P **do**
    $score_1 = ES(ET(v, x_1), 1) \cdot \prod_{i=2}^{l/2} ES(ET(x_{i-1}, x_i), i)$
    $score_2 = ES(ET(v, x_1), l/2+1) \cdot \prod_{i=2}^{l/2} ES(ET(x_{i-1}, x_i), l/2 + i)$
    add edge $(v, x_{l/2})$ to $MG_1.E$ and $MG_2.E$, if it already doesn't exist
    $MG_1.W(v, x_{l/2}) += score_1$
    $MG_2.W(v, x_{l/2}) += score_2$
for metapath edge $e_1 = (a, b)$ in MG_1 **do**
  for metapath edge $e_2 = (b, c)$ in MG_2 **do**
    add edge (a, c) to MG, if it already doesn't exist
    $MG.W(a, c) += MG_1.W(e_1) * MG_2.W(e_2)$

experiment_comparsion with matrix-based GTN

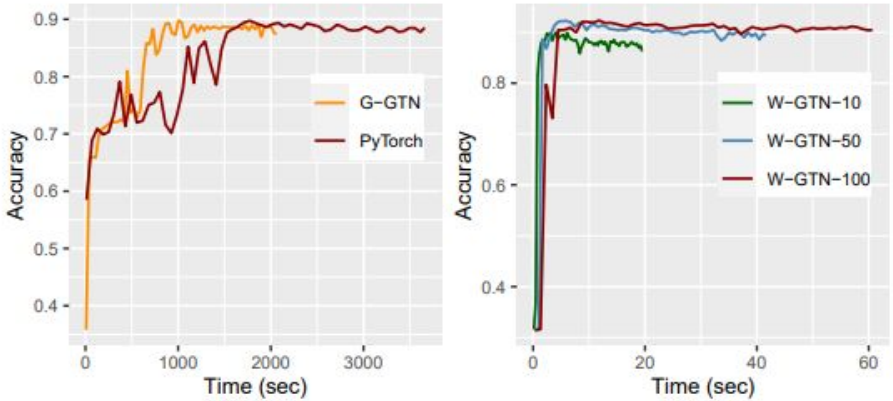


Figure 4: Time to accuracy for GTN based systems for runs with 3 metapath layers.

Fig. 4 shows the time to accuracy for the ACM graph for the GTN-based systems. G-GTN reaches high accuracy significantly faster than P-GTN because each epoch takes less time. Similarly, the variations of W-GTN reach good accuracy faster than G-GTN because of faster epoch time.

PyTorch ; The PyTorch implementation **uses dense matrices for each intermediate metapath graph**, so the memory overhead is impractical for large graphs.

G-GTN ; Our graph formulation **precomputes the space required for the metapath graph and the intermediate metapath graphs**.

ad

next . GCN from scratch in Numpy

next . dynamic graph embedding in e-commerce sector (PYG)

next . heterogeneous graph embedding in e-commerce sector (DGL)

next . study group for GNN application at pseudo lab

give me the your email address then i will alert next study time :)