TIGER: Training Inductive Graph Neural Network for Large-scale Knowledge Graph Reasoning

Kai Wang
kai_wang@ntu.edu.sg
Nanyang Technological University
Singapore

Yuwei Xu
S220060@e.ntu.edu.sg
Nanyang Technological University
Singapore

Siqiang Luo*
siqiang.luo@ntu.edu.sg
Nanyang Technological University
Singapore

ABSTRACT

Knowledge Graph (KG) Reasoning plays a vital role in various applications by predicting missing facts from existing knowledge. Inductive KG reasoning approaches based on Graph Neural Networks (GNNs) have shown impressive performance, particularly when reasoning with unseen entities and dynamic KGs. However, such state-of-the-art KG reasoning approaches encounter efficiency and scalability challenges on large-scale KGs due to the high computational costs associated with subgraph extraction – a key component in inductive KG reasoning. To address the computational challenge, we introduce TIGER, an inductive GNN training framework tailored for large-scale KG reasoning. TIGER employs a novel, efficient streaming procedure that facilitates rapid subgraph slicing and dynamic subgraph caching to minimize the cost of subgraph extraction. The fundamental challenge in TIGER lies in the optimal subgraph slicing problem, which we prove to be NP-hard. We propose a novel two-stage algorithm SiGMa to solve the problem practically. By decoupling the complicated problem into two classical ones, SiGMa achieves low computational complexity and high slice reuse. We also propose four new benchmarks for robust evaluation of large-scale inductive KG reasoning, the biggest of which performs on the Freebase KG (encompassing 86M entities, 285M edges). Through comprehensive experiments on state-of-the-art GNN-based KG reasoning models, we demonstrate that TIGER significantly reduces the running time of subgraph extraction, achieving an average 3.7× speedup relative to the basic training procedure.

PVLDB Reference Format:
Kai Wang, Yuwei Xu, and Siqiang Luo. TIGER: Training Inductive Graph Neural Network for Large-scale Knowledge Graph Reasoning. PVLDB, 17(10): 2459 - 2472, 2024.
doi:10.14778/3675034.3675039

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/KyneWang/TIGER.

1 INTRODUCTION

Knowledge Graph (KG) Reasoning aims to predict missing facts by reasoning over domain knowledge and human commonsense in the form of *head-relation-tail* triples [20, 47], which has been extensively studied and contributed to various applications such as information retrieval, recommendation systems, drug discovery, and financial trend prediction [3, 8, 10, 36, 57]. Given a query, i.e., two items of a triple (head entity, relation), the central KG reasoning task is to predict the missing entity from the entity set of the KG. For instance in Figure 1(a), the answer to the query ("Avatar 2", "directed_by") would be "J. Cameron". Conventional KG reasoning relies on Knowledge Graph Embedding (KGE) [4, 42, 46], which represents entities as trainable embedding vectors, but struggles with the *Inductive Reasoning* task, i.e., handling updated triples or new entities unseen during the training phase [43]. Recent inductive KG reasoning models based on Graph Neural Networks (GNNs) [17, 22, 26, 63], such as REDGNN [58] and NBFNet [66], have surpassed the limitations of KGE methods. They achieve inductive KG reasoning by utilizing local path evidence from KG subgraphs [51, 59, 65].

The rapid advancement in Artificial Intelligence (AI), with new applications like AI assistants, is escalating the need for comprehensive knowledge understanding to address 'AI Hallucinations' [12, 21]. This development has significantly heightened the demand for inductive reasoning on large-scale Knowledge Graphs (KGs) such as Freebase [2] and WikiData [45]. These KGs, often comprising millions or even billions of triples, can expand to data sizes of several terabytes [16]. Such a dramatic increase in size poses not only a challenge to storage capabilities but also leads to GPU memory overload and substantial computational costs, making the training of GNN-based inductive reasoning models on large-scale KGs economically and practically unfeasible [65]. To address this scalability issue, an easily adopted approach is to manage data size by extracting L-hop neighborhood KG subgraphs that correspond to the sampled queries for each mini-batch training session. For instance, as depicted in Figure 1(a), a 2-hop subgraph centered on "Avatar" can be selectively fed into the GNN-based model, while the complete KG data remains stored on a solid-state disk (SSD). However, while GNN training using KG subgraphs enhances scalability, the repeated gathering of L-hop subgraphs introduces significant efficiency challenges. As shown in Figures 1(b)(c), a 3-hop subgraph in the FB5M dataset can encompass up to one million triples, with subgraph extraction consuming up to 90% of training time across three large-scale KG datasets. Consequently, optimizing the subgraph extraction process becomes essential for efficient GNN-based inductive reasoning on large-scale KGs.

To the best of our knowledge, no existing GNN training system specifically addresses the efficiency challenges of subgraph extraction in large-scale KGs. Existing large-scale KG system studies for traditional KGE models mainly focus on negative sampling and

[*] Siqiang Luo is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 17, No. 10 ISSN 2150-8097.
doi:10.14778/3675034.3675039

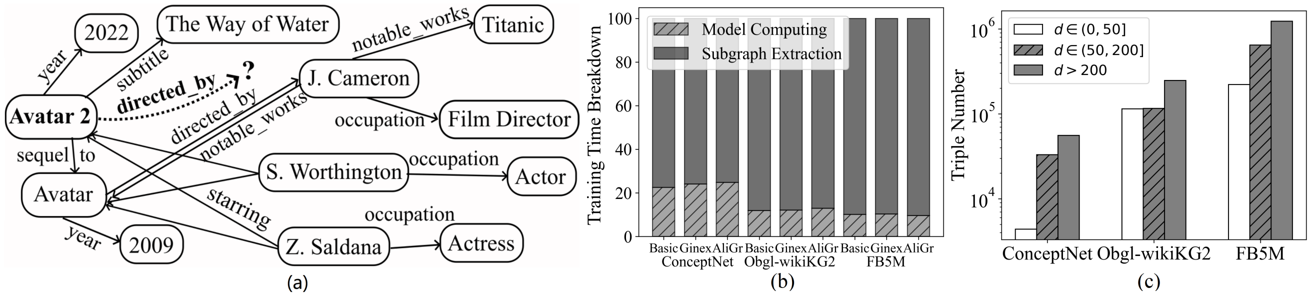


Figure 1: (a) Example of knowledge graph reasoning, (b) Training time analysis for large-scale KGs under the basic pipeline and techniques from AliGraph [62] and Ginex [35], and (c) 3-hop subgraph size (in triples) for central entities of varying degrees.

embedding storage, which are usually not applicable to GNN-based inductive reasoning [23, 37, 61]. We contend that accelerating subgraph extraction in the large-scale inductive KG reasoning system poses three significant challenges, rendering other system-level acceleration techniques nearly ineffective. (1) **Subgraph Data Completeness**: The data integrity of the input subgraph is crucial, as it determines the candidate targets and reasoning paths for a given query. Any loss of information in the subgraph directly impacts the model's performance. Common strategies like node/edge sampling [18] or graph partitioning [13] are unsuitable, as they would break subgraph completeness, requiring additional data communication to compensate for lost information. (2) **Heterogeneous Graph Structure**: The data characteristics of KG subgraphs complicate their calculations and storage. While recent GNN training systems, like AliGraph [62] and Ginex [35], cache neighbor information for central nodes. As verified in Figure 1(b), this strategy does not alleviate the efficiency pressure for frequent subgraph extraction. It also struggles with handling the various relation types in the KG and the overlapping of triples among KG subgraphs. (3) **SSD Storage and Access**: Subgraph extraction involves extensive random SSD access to load KG triple data, creating a significant efficiency bottleneck. Although pre-storing all subgraphs for sequential access might increase speed, it would also lead to substantial storage demands. Furthermore, the random nature of mini-batch loading hinders the reusability of subgraph storage.

To address these challenges, we present a newly designed subgraph extraction framework, TIGER, which supports the efficient training of state-of-the-art GNN-based reasoning models on a single machine, particularly for large-scale inductive KG datasets. Since GNN training optimization is a multi-level, modular system engineering process, TIGER focuses on the SSD storage and caching parts to accelerate subgraph extraction under the constraints of single-machine resources. As a preview, compared to the typical subgraph-based training procedure, our TIGER achieves an average 3.7× speedup in training time without performance degradation and easily scales to large-scale KGs. Specifically, TIGER has four novel contributions:

First, to accelerate subgraph extraction while maintaining subgraph completeness in each mini-batch training, TIGER employs a novel 'Slice&Cache' procedure to precompute subgraphs in a streaming way. Given a batch of queries, each subgraph is sliced into reusable data slices which are stored on the SSDs for fast sequential reading. To further reduce SSD access, high-priority slices are cached in the main memory utilizing a dynamic cache mechanism. (Section 3)

Second, considering the heterogeneous graph structure of KG subgraphs, we propose a specific atom-based subgraph slicing problem, in which each 1-hop subgraph is treated as an atom. By representing an L-hop subgraph as a combination of multiple atoms, triple data stored on the SSDs can be excluded from the slicing algorithm thereby reducing data communication and algorithmic complexity. Regarding slicing quality and reusability, we prove that the atom-based slicing problem is NP-hard. (Section 4.1)

Third, we design a novel two-stage subgraph slicing algorithm, SiGMa, which first matches the atoms of a subgraph with existing slices and then generates new reusable slices for the unmatched atoms. Specifically, we map the SiGMa's two stages, slice matching and slice generating, to two classic problems of set covering and bin packing. Leveraging existing heuristic solutions, we devise efficient algorithmic approaches with approximate guarantees. SiGMa optimizes slice redundancy and utilization while costing around 40% time and 75% storage space of competitors. (Section 4.2)

Finally, we propose new fundamental benchmarks for large-scale inductive KG reasoning. We construct four new inductive KG datasets from real-world KG data, including the most influential Freebase containing more than 300 million triples. Each benchmark KG contains more than two million entities, whose data scale is around 100× bigger than the previous ones. We conduct extensive experiments on six state-of-the-art GNN-based reasoning models and lay the groundwork for this new research problem. (Section 5)

2 BACKGROUND

2.1 GNN-based Inductive KG Reasoning

A Knowledge Graph $\mathcal{G}$ can be represented as a set of factual triples $\{(e_h, r, e_t) | e_h, e_t \in \mathcal{E}, r \in \mathcal{R}\}$, where the relation r denotes the edge type between the head entity e_h and the tail entity e_t, and $\mathcal{E}$, $\mathcal{R}$ are the sets of entities and relations, respectively. Given a query (q, r_q) containing a query entity $q \in \mathcal{E}$ and a query relation $r_q \in \mathcal{R}$, the KG reasoning task aims to find the target entity $e_a \in \mathcal{E}$ satisfying that (q, r_q, e_a) belongs to the knowledge graph $\mathcal{G}$. The inductive reasoning query for model evaluation is extracted from a new KG $\mathcal{G}'$ whose entity and relation sets satisfy $\mathcal{R}' \subseteq \mathcal{R}$ and $\mathcal{E}' \cap \mathcal{E} = \emptyset$. It means the query entity and target entity are unseen during the training phase. To achieve inductive KG reasoning, recent GNN-based methods, such as NBFNet [66] and REDGNN [58], encode each candidate entity by extracting local paths starting from the query entity q within the L-hop neighborhood subgraph $\mathcal{G}^L_q$ centred on q, where $\mathcal{G}^L_q$ is called the query subgraph in this paper. Specifically, given the initialized message vector as $\mathbf{e}^0_{q|q}$, the basic GNN formula in the ℓ-th iteration is:

$$\mathbf{e}^{\ell}_{t|q} = UPDATE\Big(\mathbf{W}^{\ell} AGG(\mathbf{e}^{\ell-1}_{i|q} \otimes \mathbf{r}^{\ell}|_{(e_i,r,e_t)\in\mathcal{G}^L_q}), \mathbf{e}^{\ell-1}_{t|q}\Big), \tag{1}$$

in which $\otimes$ is the relation-specific transformation operation, $AGG(\cdot)$ and $UPDATE(\cdot)$ refer to the aggregation function and update function in GNNs, and $\mathbf{W}^{\ell}$ is a weighting matrix in the ℓ-th layer. After the L-layer GNN message passing, the entity embedding vector $\mathbf{e}^L_{e|q}$ is decoded to output the plausibility score of this candidate entity e. A higher score means the triple (q, r_q, e) is more likely to be true. To facilitate subsequent discussions, we give the definition of query subgraph and three basic properties of subgraph atom as follows:

Definition 1 (Query Subgraph). *A query subgraph $\mathcal{G}^L_q \subseteq \mathcal{G}$ is the set of triples whose head entity can be reached from the query entity q through at most $L-1$ edge-hops. Denote $\mathcal{N}^0_q = \{q\}$ and $\mathcal{N}^{L-1}_q$ as the neighbor entities whose shortest distance from q is not more than $L-1$, the following holds:*

$$\forall (e_h, r, e_t) \in \mathcal{G}, e_h \in \mathcal{N}^{L-1}_q \Rightarrow (e_h, r, e_t) \in \mathcal{G}^L_q. \tag{2}$$

The 1-hop subgraph $\mathcal{G}^1_q$ is referred to as a **Subgraph Atom**, *which encompasses all triples in which the head entity is q.*

Example 1. *Consider the sub-KG in Figure 1(a) as the 2-hop query subgraph of "Avatar 2", a subgraph atom centered on "Z.Saldana" includes three triples: ("Z.Saldana", "starring", "Avatar"), ("Z.Saldana", "starring", "Avatar 2"), ("Z.Saldana", "occupation", "Actress").*

Property 1. *Any two atoms are edge-disjoint and no proper subset of an atom solely appears within a query subgraph.*

Property 2. *Any query subgraph $\mathcal{G}^L_q$ can be partitioned into multiple atoms with the total quantity $|\mathcal{N}^{L-1}_q|$.*

Property 3. *Assume two atoms of e_1, e_2 having the shortest distance k, the count of their co-existing query subgraphs is at least $|\mathcal{N}^{L-k}_{e_1} \cup \mathcal{N}^{L-k}_{e_2}|$.*

Notably, the subgraph atom of an entity q only contains its outgoing edges to ensure Property 1. If one atom contained bidirectional triples, the same triple could be present in two atoms.

2.2 Subgraph Extraction for GNN Training

For large-scale KG datasets, the standard training pipeline of inductive GNN models is illustrated in Figure 2. Given the queries generated from KG triples in the training dataset, each mini-batch loop samples a batch of training queries, extracts their query subgraphs, and conducts the model forward and backward calculations [58, 65]. The efficiency bottleneck lies in extracting L-hop subgraphs repeatedly in the mini-batch loop. To explain, given the number of training queries n_{tr}, the cumulative count of KG triple accesses required for subgraph extraction across all training queries during an n_{ep}-epoch training process approximates $O(n_{ep} \cdot n_{tr} \cdot d^L)$, where d represents the average entity degree. Take our FB5M dataset in Table 2 as an example, training ten epochs requires more than 100 billion KG triple accesses.

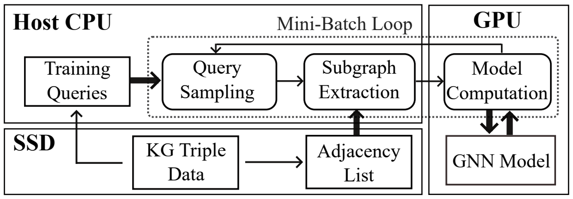


Figure 2: Basic GNN training framework.

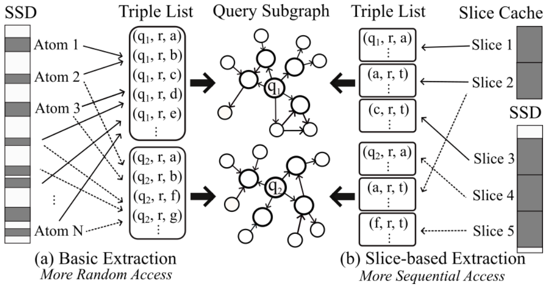


Figure 3: Illustrations of two subgraph extraction ways

Table 1: Time speedup of slice-based subgraph extraction.

Datasets	ConceptNet	Obgl_wikiKG2	FB5M
Avg. Atom Number	889.34	13587.89	9294.63
Avg. Total Time(s)	0.0408	0.3611	0.4048
Avg. Slice Number	15.27	113.73	158.2
Avg. Total Time(s)	0.0002	0.0013	0.0018
Time Speedup	177.37	267.85	227.08

Meanwhile, each subgraph extraction operation requires numerous SSD access. The original triple data of large-scale KGs is typically stored on SSDs as a $|\mathcal{G}| \times 3$ matrix, where each row consisting of three integers representing the individual IDs of the head entity, relation, and tail entity of a triple, respectively. To accelerate graph loading, an adjacency matrix is stored in the compressed sparse row (CSR) format as it allows fast access to in-neighbors of each entity. Besides, an adjacency list for each entity is constructed to fast collect the triples of each subgraph atom via direct addressing. Nevertheless, extracting one query subgraph still requires $O(\mathcal{N}^{L-1}_q)$ disk accesses when assuming each adjacency list is contained within a disk page. Due to the power-law distribution of KG datasets, most atoms contain only a few dozen triples. Consequently, the basic process of subgraph extraction illustrated in Figure 3(a) necessitates massive SSD random access, which is substantially slower than sequential access [28].

Our Motivations: The Slice&Cache Procedure. To enhance the efficiency of GNN model training, we focus on the challenges in subgraph extraction stemming from repetitive calculations and slow SSD access. A straightforward solution might be pre-storing all possible subgraphs on SSDs sequentially, but this would lead to inefficient storage and retrieval due to the immense data volume of subgraphs. To avoid this inefficiency, we introduce a novel procedure, Slice&Cache, which segments the subgraph triples into multiple reusable slices and partially caches them in the main memory. This procedure, illustrated in Figure 3(b), retrieves a small number of slices from both the SSDs and the slice cache when assembling a

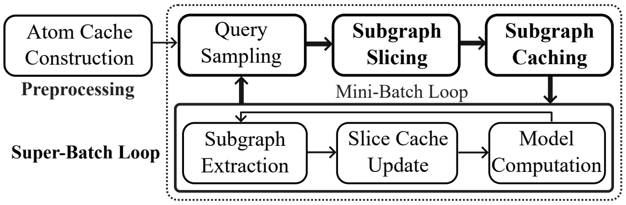


Figure 4: TIGER training pipeline overview.

batch of query subgraphs. The rationale behind Slice&Cache draws from the observation that triples from closely connected atoms in KG subgraphs are often loaded simultaneously. Preliminary results in Table 1 demonstrate that loading subgraphs from SSDs in sliced form—each slice containing no more than 2,048 triples—is over 177 times faster than retrieving subgraph atoms from adjacency lists. This significant speedup can be attributed to the higher bandwidth of sequential access compared to random access across both SSDs and main memory, as noted in [38]. Consequently, organizing subgraph triples into sequentially stored slices not only streamlines SSD reads but also enhances cache utilization, leading to accelerated training processes for GNN models.

3 THE PROPOSED FRAMEWORK: TIGER

To implement the Slice&Cache procedure for highly efficient subgraph extraction, we design a novel training framework, TIGER, tailored for large-scale inductive KG reasoning. We overview the TIGER training pipeline in Section 3.1. The two core components of TIGER, subgraph slicing and subgraph caching, are described in Sections 3.2 and 3.3.

3.1 TIGER Training Pipeline Overview

Figure 4 illustrates the training pipeline of TIGER. Given the KG triple data stored on the SSDs, TIGER initially undertakes *Atom Cache Construction*, loading a portion of atom triples (1-hop subgraphs) into the main memory to accelerate the subsequent subgraph extraction. Following this, TIGER starts model training by iterating the *Super-batch Loop*, which is specifically designed to precompute input data of multiple batches before mini-batch training, thus reducing repetitive calculations and facilitating the upcoming cache mechanism. A super-batch loop starts with three precomputation stages: *Query Sampling*, *Subgraph Slicing*, and *Subgraph Caching*, where all required subgraphs are reconstructed into uniformly sized slices and stored in either the Slice Cache or SSDs. Subsequently, the super-batch loop trains multiple batches of sampled queries through continuous *Mini-batch Loops*, leveraging the precomputed slices to minimize subgraph extraction costs. As the super-batch loops progress, the precomputation time decreases due to the dwindling number of unsliced subgraphs. Notably, such subgraph precomputation ensures efficient training without altering model calculations, thus there is no sacrifice for effectiveness.

Atom Cache Construction. The Atom Cache, established at the start and unchanging in subsequent super-batch loops, aims to reduce the cost of reading atoms (e.g. 1-hop query subgraphs) from SSDs. It contains two cache structures within a predefined size: one as a 1-D array for storing neighbor entity IDs and another for atom triples of high-degree entities. This cache employs direct addressing, recording each entity's array index and data length to achieve swift $O(1)$ cache lookups.

Query Sampling. Given the hyperparameter *superbatch_size* determining the number of mini-batches, this stage samples all batches of training queries together. Unlike the basic pipeline samples within each mini-batch loop, TIGER preemptively identifies the specific subgraphs each batch will access prior to mini-batch training. This foresight enables the implementation of an efficient caching mechanism for subgraph extraction. Without altering the data sampling strategy, such a centralized sampling procedure does not affect model performance.

Subgraph Slicing. This stage is crucial for optimizing subgraph extraction. It extracts unsliced subgraphs accelerated by the Atom Cache, and segments each of them into uniformly sized slices. The slicing results are recorded in a mapping dictionary which links query entity IDs to their corresponding slice IDs. This arrangement allows for the direct retrieval of sliced subgraphs from sequential slice data, substantially reducing random SSD access. To enhance the efficiency of subgraph slicing, we propose an innovative paradigm called atom-level subgraph slicing. It significantly cuts down time complexity and memory usage by converting subgraphs into disjoint atoms for more efficient slicing. The comprehensive design of this module is detailed in Section 3.2.

Subgraph Caching. Upon obtaining the slicing results, this stage entails storing the newly generated slices on SSDs. An atom-based slice structure is proposed effectively reducing SSD storage requirements by a factor of three. To decrease the frequency of SSD access, a portion of slices are cached in the main memory, termed the Slice Cache, and is systematically arranged in a 2-D array utilizing direct addressing. To enhance the efficiency of slice data loading, a powerful caching mechanism is implemented by precomputing the next-access slices of each mini-batch to support the dynamic cache update in the mini-batch training. The specifics of this process are elaborated in Section 3.3.

Mini-batch Loop. In this stage, given a batch of queries, the associated slice IDs are retrieved from the slice mapping dictionary. The corresponding slice data is first extracted from the Slice Cache and then from the SSDs, which is used to construct the batch subgraph. After that, the Slice Cache is updated with the precomputed changeset for this batch to enhance the cache hit ratio in subsequent mini-batch loops. Finally, TIGER transfers batch queries and complete subgraphs into the GPU, where it performs the forward and backward computations of the GNN-based model.

3.2 Subgraph Slicing Module

Subgraph slicing is designed to circumvent the need for repeated subgraph extraction by segmenting query subgraphs into efficiently loadable slices. Specifically, for a given query subgraph $\mathcal{G}_q^L$, subgraph slicing is to find a series of disjoint triple sets, termed slices ($\mathbb{S}_q = S_1, S_2, \cdots$), whose union is equivalent to $\mathcal{G}_q^L$. Given a predefined slice size h, each slice $S \in \mathbb{S}_q$ is limited to containing at most h triples ($|S| \leqslant h$), with any remaining space in the slice being zero-filled when storing. A global dictionary maintains the association between slices and query entities, mapping each query entity ID q to its corresponding slice IDs within $\mathbb{S}_q$. Consequently, for queries previously processed through slicing, their subgraphs are denoted as lists of slice IDs. This allows for direct loading of subgraph triples from these slices, eliminating the need for traditional subgraph extraction methods.

Atom-level Subgraph Slicing. When processing an unsliced subgraph, a straightforward approach is retrieving all its triples from SSDs and distributing them into slices whose complexity is contingent upon the number of triples. We refine this process by slicing at the atom level. According to Properties 1 and 2, a query subgraph $\mathcal{G}_q^L$ can be decomposed into a set of disjoint atoms $\{\mathcal{G}_e^1 | e \in \mathcal{N}_q^{L-1}\}$. The triples within each atom are regarded as a singular, inseparable unit for efficient slicing and an h-length slice would encompass multiple complete atoms. For an atom exceeding the size h, its triples are jointly stored across multiple slices, thereby being excluded from the slicing process.

Example 2. *Consider the 3-hop subgraph centred on q in Figure 5(a), its atom-based subgraph is a sequence of atoms in the form of (entity ID, atom weight). The 1-hop subgraph atom A_q contains five triples whose head entity is q, thus the atom weight of A_q is the triple number 5.*

Slicing on atom-based subgraphs offers a significant reduction in time complexity and memory usage. The quantity of atoms is considerably less than that of subgraph triples, leading to a markedly reduced time complexity for slicing operations. Furthermore, atom-level slicing necessitates only atom information, e.g. L-1 hops of neighbors and atom weights, avoiding the need to load all subgraph triples into the main memory. Additionally, the memory footprint of both atom-based subgraph and slice data during calculations is substantially smaller.

Two-stage Slicing Algorithm. A straightforward slicing method is to partition triples or atoms into fixed-length slices without considering their reuse, leading to each slice being uniquely associated with a single subgraph. It would result in increased SSD storage pressure and inefficient access. TIGER tackles this issue with a two-stage slicing algorithm SiGMa, which prioritizes the reuse of existing slices before allocating new slices for unmatched atoms. Specifically, SiGMa begins with a list of query entities and existing slices. For each entity's atom-based subgraph, SiGMa first undergoes a *Slice Matching* stage, identifying existing slices that are fully constituted by certain atoms from the subgraph. Subsequently, the *Slice Generating* stage generates new slices for remaining unmatched atoms. The outcome is a list of slice IDs corresponding to this subgraph, which is then recorded in the global mapping dictionary. And new slices built from the generating results will be stored in SSDs.

Example 3. *Regarding the atom-based subgraph of q in Figure 5(b), two slices S_1 and S_2 in existing slices are matched first. Despite three other slices also being composed of certain atoms, they overlap with S_1 or S_2 without higher slice capacity (e.g. $|S_4|$=9<14=$|S_1|$). Subsequently, the remaining atoms are allocated to new slices S_6 and S_7, resulting in the subgraph being represented by these four slices in total.*

The primary objective of subgraph slicing, given a set of subgraphs, is to efficiently load all subgraph triples while minimizing the number of slices accessed, thus reducing data communication with SSDs. In Section 4, we will not only formally define the atom-level subgraph slicing problem but also demonstrate its NP-hard nature. This revelation underscores the significance of crafting effective algorithms for both subgraph matching and generating. We will delve into the theoretical analysis of the proposed SiGMa algorithm in Section 4.1 and explore its algorithmic design in Section 4.2, thereby providing a comprehensive understanding of its functionality and efficiency.

3.3 Subgraph Caching Module

The Subgraph Caching Module is dedicated to both storing slices on SSDs and managing the Slice Cache. Once subgraphs are sliced in a super-batch, new slices comprising specific triple data are encoded with the support of the Atom Cache and then stored on SSDs for future use. To minimize SSD access and hasten subgraph extraction, TIGER employs main memory caching for a proportion of slices, called Slice Cache. The Slice Cache efficiently organizes the equal-length slices in a straightforward 2-D array, utilizing direct addressing similar to the Atom Cache. During mini-batch loops, a slice is swiftly retrieved using its specific slice ID and subsequently decoded back into the original triples.

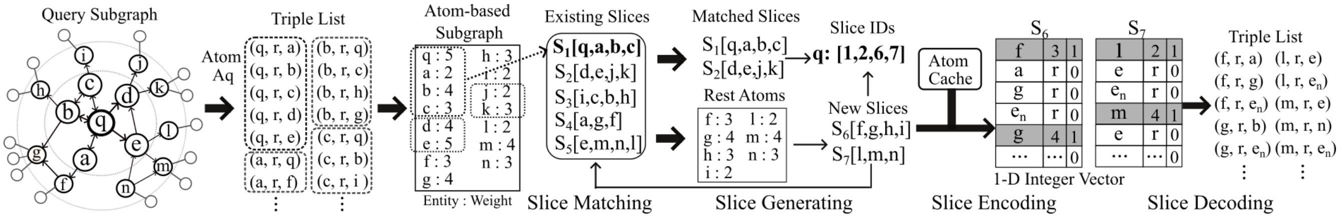
Atom-based Slice Storage. To further compress storage space and accelerate slice loading, we propose an atom-based slice storage structure. In one slice, the stored triples are organized into multiple atoms where all triples of one atom share the same head entity. Therefore, we compress the redundant triple data by reconstructing atom triples into a 1-D *uint*64 array. We concatenate the common head entity ID and the triple number of the atom to form the first *uint*64 integer, and then concatenate the two IDs in each (r, e_t) pair. Moreover, all concatenated *uint*64 integers are shifted one place to the left. The vacated space serves as a signal spot: it is set to 1 for the head ID line and to 0 otherwise.

Example 4. *Consider storing the new slice S_6 in Figure 5(c). Focusing on the atom associated with the entity ID f, its first item is concatenated by three parts $(f, 3, 1)$, indicating the presence of three triples headed by f. The next three rows represent these triples, each records one relation ID and one tail entity ID, marked by a 0 signal spot. Consequently, the atom f is encapsulated within the 1-D uint64 array, comprised of four uint64 integers.*

This innovative slice storage, replacing three *int*64 integers with a single *uint*64 integer, effectively reduces the storage requirement by a factor of three. For instance, consider a scenario with 10,000 slices and the slice size h = 2048. In this case, the total storage needed for the basic structure would be approximately 4.58 GB. In contrast, the advanced atom-based slice structure would require only about 1.53 GB.

Dynamic Caching Mechanism. To minimize SSD access, TIGER's Slice Cache is dynamically updated with each mini-batch loop, maintaining a consistently high hit ratio. The dynamic caching mechanism is inspired by the recent SSD-based GNN training system, Ginex [35]. Ginex adopts Belady's cache replacement algorithm [1], which is provably optimal by evicting data with the highest reuse distance at every timestep. Following Belady's cache mechanism, the Slice Cache prioritizes slices that would be accessed in the early mini-batch loops of the current super-batch, and evicts those with the longest wait time before needed.

The dynamic caching process in one super-batch is shown in Algorithm 1. Specifically, in the Subgraph Caching stage, the slices


Query Subgraph
Atom Aq
Triple List
(q, r, a) (q, r, b) (q, r, c) (q, r, d) (q, r, e)
(a, r, q) (a, r, f)
(b, r, q) (b, r, c) (b, r, h) (b, r, g)
(c, r, q) (c, r, b) (c, r, i)
Atom-based Subgraph
q : 5 a : 2 b : 4 c : 3 d : 4 e : 5 f : 3 g : 4
h : 3 i : 2 j : 2 k : 3 l : 2 m : 4 n : 3
Entity : Weight
Existing Slices
S1[q,a,b,c] S2[d,e,j,k] S3[i,c,b,h] S4[a,g,f] S5[e,m,n,l]
Matched Slices
S1[q,a,b,c] S2[d,e,j,k]
Slice IDs
q: [1,2,6,7]
Rest Atoms
f : 3 l : 2 g : 4 m : 4 h : 3 n : 3 i : 2
New Slices
S6[f,g,h,i] S7[l,m,n]
Slice Matching
Slice Generating
Atom Cache
S6
S7
1-D Integer Vector
Triple List
(f, r, a) (l, r, e) (f, r, g) (l, r, en) (f, r, en) (m, r, e) (g, r, b) (m, r, n) (g, r, en) (m, r, en)
Slice Encoding
Slice Decoding

(a) Atom-based Subgraph Data **(b) Two-stage Slicing Algorithm** **(c) Atom-based Slice Storage**

Figure 5: Graphical illustration of subgraph processing.

```
Algorithm 1: Dynamic Caching
  Input  : Query entities $Q$ of one super-batch, slice mapping dictionary $M$.
1 Get the list of slice IDs from $M$ for each query entity $q$ in $Q$;
2 Precompute the initializing slice IDs $S_{init}$;
3 Gather slices $\mathcal{S}_{init}$ from SSDs corresponding to $S_{init}$;
4 Initialize Slice Cache: $\mathcal{S}_{ca} \leftarrow \mathcal{S}_{init}$;
5 Precompute the changeset $C_b$ of each mini-batch $b$;
6 foreach mini-batch b of the super-batch do
7     Gather slices $\mathcal{S}_b$ for the mini-batch query entities $Q_b$;
8     Train the GNN model with $Q_b$ and $\mathcal{S}_b$;
9     Evict slices from Slice Cache: $\mathcal{S}'_{ca} \leftarrow$ Evict($\mathcal{S}_{ca}, C_b$);
10    Gather inserted slices from $\mathcal{S}_b$: $\mathcal{S}_{up} \leftarrow$ Gather($\mathcal{S}_b, C_b$);
11    Update Slice Cache: $\mathcal{S}_{ca} \leftarrow$ Update($\mathcal{S}_{up}, \mathcal{S}'_{ca}, C_b$);
12 Empty the Slice Cache;
```

utilized in the first few batches are prefetched as the initialization of Slice Cache (in Lines 2-4). Then, given the slice IDs required by each mini-batch, TIGER follows Ginex to precompute which slices to insert into and evict from the Slice Cache (in Line 5). The precomputation results, called changesets, will be utilized in the Slice Cache Update stage at each mini-batch loop. When updating the cache guided by the changeset (in Lines 9-11), slices in the cache that are marked for eviction are replaced with slices from the current batch that have been loaded into the main memory.

Unlike Ginex [35], which focuses on caching nodes' feature vectors, integrating the dynamic caching mechanism into TIGER introduces three principal challenges for the subgraph slicing algorithm: (1) Subgraph slicing, being more time-consuming than processing feature vectors, requires further acceleration techniques to sustain training efficiency. (2) The caching mechanism's effect hinges on the quality of subgraph slicing, as slices with greater reusability would improve cache hit ratios. (3) This mechanism necessitates that cached slices have a fixed length, similar to feature vectors, demanding a balance between slice redundancy and utilization.

4 ATOM-LEVEL SUBGRAPH SLICING

4.1 Problem Formulation

To reduce the time complexity and memory footprint of the subgraph slicing process, we first convert the triple-level subgraph slicing problem to the atom level. Ignoring the specific triple data, we can represent an atom-based subgraph as $G_q = \{(e_a, w_a)|e_a \in \mathcal{N}_q^{L-1}, w_a = |\mathcal{G}^1_{e_a}|\}$. The subgraph size $|G_q| = \sum_{a \in G_q} w_a$ and w_a refers to the atom weight (the number of atom triples). Formally, the atom-level subgraph slicing problem is defined as follows:

Definition 2 (Atom-level Subgraph Slicing). *Given a predetermined slice size h and a collection of query entities Q, the corresponding atom-based subgraphs are* $\mathbf{G} = \{G_q|q \in Q\}$ *and the collection of atoms appearing in* $\mathbf{G}$ *are denoted as* $\mathbf{A} = \{a|w_a < h\}$. *The atom-level subgraph slicing problem is to construct a collection of slices* S *satisfying that: each slice* $S_i \in \mathrm{S}$ *consists of multiple distinct atoms in* $\mathbf{A}$ *and the total weight of atoms* $|S_i| = \sum_{a \in S_i} w_a \leqslant h$; *any subgraph* $G_q \in \mathbf{G}$ *can be composed by a set of non-overlapping slices* $\mathrm{S}_q \subseteq \mathrm{S}$, *i.e.,* $G_q = \bigcup \mathrm{S}_q$.

Optimization Target. Unlike graph partitioning algorithms that divide the entire graph into multiple disjoint parts, the stored slices are allowed to overlap with the other, enabling the possibility of being reused by various KG subgraphs. Given a collection of subgraphs $\mathbf{G}$, the optimization of atom-level subgraph slicing aims to load all subgraph triples by visiting as few slices as possible thereby reducing SSD data communication. When the sizes of subgraphs and slices are given, the number of required slices is determined by two aspects of slice quality: *Slice Redundancy* and *Slice Utilization*. The lower slice redundancy is better, indicating less zero-filling/redundant space in each slice. Meanwhile, reusing more slices in multiple overlapping subgraphs can also decrease the total required slices. Therefore, we can define the optimization target, named *Slicing Score*, as follows:

$$f(\mathbf{G}, \mathrm{S}, h) = \frac{|\mathrm{S}|}{\sum_{G_q \in \mathbf{G}} \lceil |G_q|/h \rceil} = \underbrace{\frac{\sum_{G_q \in \mathbf{G}} |\mathrm{S}_q|}{\sum_{G_q \in \mathbf{G}} \lceil |G_q|/h \rceil}}_{\text{Slice Redundancy Rate } \delta_R} \times \underbrace{\frac{|\mathrm{S}|}{\sum_{G_q \in \mathbf{G}} |\mathrm{S}_q|}}_{\text{Slice Utilization Rate } \delta_U}, \quad (3)$$

where $|\mathcal{G}_q|$ and $|\mathrm{S}_q|$ denote the quantities of subgraph triples and slices, respectively. The slicing score is the ratio between the number of distinct slices in S and the total number of the fewest slices needed for each subgraph, which is derived from the original goal of atom-level subgraph slicing. Interestingly, according to Equation 3, the slicing score can be decoupled into two helpful metrics: Slice Redundancy Rate (δ_R) and Slice Utilization Rate (δ_U). The former δ_R compares the actual number of loaded slices to the minimum required, where the lower δ_R means less redundant space in these slices. While δ_U measures the ratio of distinct slices to the total required, reflecting slice reuse frequency in these subgraphs. Therefore, according to the definitions, a subgraph slicing is optimal if the slice set S achieves the lowest slicing score, i.e., $\delta_R \times \delta_U$.

Theorem 1. *Minimizing* $\delta_R \times \delta_U$ *to obtain an optimal atom-based subgraph slicing is NP-hard.*

Proof. We establish the NP-hardness of our target problem, atom-based subgraph slicing, by reducing the classical Bin Packing problem to it. Given multiple items/atoms with different weights, Bin Packing is to assign each atom to a bin of size h such that the total number of bins used is minimized. This Bin Packing problem is equivalent to the special case of our target problem where we are slicing only one query subgraph G_q. In this case, the slice utilization rate δ_U is fixed at 1 and the optimization target transforms to minimizing the slice redundancy rate δ_R. If a polynomial-time algorithm solves our problem, it implies one exists for the NP-hard Bin Packing problem. Consequently, atom-based subgraph slicing must also be NP-hard. □

Theorem 2. *Given the subgraph set* $\mathbf{G}$ *and atom set* $\mathbf{A}$*, the metric* δ_R *of a slicing result* $\mathbf{S}$ *has a lower bound:*

$$\delta_R \geqslant \left\lceil (\textstyle\sum_{a\in\mathbf{A}} w_a)/h \right\rceil / (\delta_U \cdot \textstyle\sum_{G_q\in\mathbf{G}} \lceil |G_q|/h \rceil) \tag{4}$$

Proof. According to the definition of the slicing score, $\delta_R \times \delta_U = f(\mathbf{G}, \mathbf{S}, h) = |\mathbf{S}|/(\sum_{G_q\in\mathbf{G}} \lceil |G_q|/h \rceil)$. Meanwhile, the total slice number $|\mathbf{S}| \geqslant \lceil (\sum_{a\in\mathbf{A}} w_a)/h \rceil$. Because $\delta_U \leqslant 1$ and the other terms are constants, the lower bound holds. □

Challenges. According to the preceding analysis, the atom-based subgraph slicing is presented as an NP-hard problem, incorporating two optimization objectives, δ_R and δ_U. According to Theorem 2, we know that the lower bound of the metric δ_R is increased by the decline of δ_U. It indicates minimizing just one metric would not be a valid solution. Take two extreme solutions as instances, treating every h triples as a slice would obtain the lowest redundancy rate δ_R but a relatively high utilization rate δ_U; while treating each atom as a distinct slice would get a low δ_U, but the δ_R score would be extremely high. Therefore, when designing the slicing algorithm, we balance the two metrics and target the lowest slicing score.

4.2 Subgraph Slicing Algorithm: SiGMa

Considering both slice redundancy and slice utilization, we design a novel two-stage algorithm SiGMa for atom-based subgraph slicing. Within two stages, SiGMa first matches existing slices for slice reuse and then assigns the unmatched atoms into multiple slices. Moreover, we observe the complicated connections between the slice quality and the functions for slice generating and matching. The slice-generating process focuses on the construction of each slice controlling the redundancy rate δ_R directly, while the slice-matching process determines the reuse of existing slices (δ_U). Meanwhile, the generated slice indirectly impacts how frequently it can be reused and the slice matching can impact the final δ_R metric by skipping the high-redundancy slices. Therefore, considering both δ_R and δ_U metrics while maintaining efficiency, we design two specific algorithms for slice generating and matching, respectively.

Slice Generating Algorithm. Slice generating aims to generate a few h-length slices $\mathbf{S}_{gen}$ whose union composes the input atom set $G'_q = \{a_1, a_2, \cdots, a_m\} (m = |G'_q|)$. These atoms come from the same atom-based subgraph G_q and the weight w_a of each atom is not more than h (as described in Section 3.2). According to Theorem 2, to prioritize minimizing δ_R while reducing δ_U, the payload of one slice should be as close as possible to h thereby minimizing the slice amount $|\mathbf{S}_{gen}|$. Minimizing $|\mathbf{S}_{gen}|$ solely is equal to the NP-Hard *Bin Packing* problem [31], in which items of different sizes must be packed into a minimum number of bins with a fixed capacity. Here, we first discuss some feasible solutions to this classical problem:

- *Naive Solution (Next-Fit Algorithm, NF):* Filling each slice with the atoms from G'_q sequentially until reaching the maximum capacity h. The time complexity is $O(m)$.
- *Greedy Solution (First-Fit Decreasing Algorithm, FFD):* Iterating over G'_q in the decreasing order, and placing each atom in the first slice that can accommodate it; if no such slice exists, creating a new slice. The time complexity is $O(m \cdot \log m)$.
- *Optimal Solution:* More sophisticated methods like integer programming can be used, but the time costs would be unaffordable.

Although NF and FFD algorithms can produce good solutions in a reasonable timeframe, they do not consider the optimization of the slice utilization rate δ_U. According to Properties 3, a clique in which atoms are connected densely with others usually appears in more subgraphs than a random combination of atoms. To this end, we design a slice-generating algorithm that indirectly controls the slice utilization by adjusting the atom processing order, as shown in Algorithm 2. Specifically, different from the FFD algorithm uses the decreasing order of atom weights, we visit the atom list with the Postorder Depth-First Search thereby decreasing the distances among atoms in a local window. Because the input atom list G'_q is usually not a complete subgraph, starting from only one atom may not cover the whole list. Therefore, we traverse the k-hop neighbors of q in G'_q as multiple root nodes (Line 2). For each neighbor atom u, the algorithm checks if u is in G'_q, pushing u onto the atom stack L_{stack} (Lines 3-5). While L_{stack} is not empty, it retrieves the top atom a. If a is unvisited, mark it as visited, gather its unvisited neighbors, sort by atom weights, and finally push them onto the stack (Lines 6-11). If a is visited, it is popped from L_{stack} and inserted into one slice in $\mathbf{S}_{new}$ via the FFD algorithm (Lines 12-14). On Lines 15-16, after searching the whole branch of one neighbor, the generated slices are filtered by the capacity threshold $alpha$ to ensure a low δ_R. The atoms in low-capacity slices would be re-packed in the next iteration. Finally, on Line 17, the rest atoms would be packed via the FFD algorithm.

Complexity Analysis: Based on the FFD algorithm, the proposed slice-generating algorithm optimizes both slice capacity (for lower δ_R) and atom distances (for lower δ_U). Except for the Postorder Depth-First Search having $O(m)$ complexity, the time complexity of the rest algorithm is similar to that of FFD, i.e., $O(m \cdot \log m)$. In the implementation, we can further reduce the time complexity and neighbor extraction I/Os by pre-ordering the atoms when loading each atom-based subgraph.

Slice Matching Algorithm. Slice matching aims to match an atom-based subgraph G_q with existing slices from the slice data $\mathbf{S}$. Specifically, the problem is to select a collection of disjoint slices $\mathbf{S}_{mat} \subseteq \mathbf{S}$, each of which is a proper subset of G_q, and maximize the union of these slices. The above problem is a variant of the classical NP-hard *Set Cover* problem [14], where one aims to select the smallest number of subsets from a collection so that their union equals the entire set. Differently, we require each selected slice to be exactly matched and non-overlapping with others to minimize the Slice Redundancy Rate (δ_R). While fuzzy matching can enhance

```
Algorithm 2: Slice Generating
Input  : Atom list $G'_q = \{(e_a, w_a)\}$, slice capacity $h$, hop
         number $k$, capacity threshold $\alpha$.
Output : Generated slices $\mathrm{S}_{gen}$.
1  Initialize the slice set $\mathrm{S}_{dfs}$;
2  foreach $k$-hop neighbor atom $u$ of $q$ do
3      if $u \notin G'_q$ then continue;
4      Initialize a collection $\mathrm{S}_{new}$ and an atom stack $L_{stack}$;
5      Push the atom $u$ to $L_{stack}$;
6      while $L_{stack}$ is not empty do
7          $a \leftarrow$ top atom of $L_{stack}$;
8          if $a$ is not visited then
9              Mark $a$ as visited;
10             Gather unvisited neighbor atoms $w$ of $a$ in $G'_q$;
11             Push all $w$ to $L_{stack}$ sorted by weight;
12         else
13             $a \leftarrow$ pop $L_{stack}$;
14             Bin packing $\mathrm{S}_{new} \leftarrow FFD(\mathrm{S}_{new}, \{a\})$;
15     $\mathrm{S}_{fill} \leftarrow \{S_i \in \mathrm{S}_{new} | size(S_i) \geqslant \alpha\}$;
16     $\mathrm{S}_{dfs} \leftarrow \mathrm{S}_{dfs} \cup \mathrm{S}_{fill}$, $G'_q \leftarrow G'_q - \bigcup \mathrm{S}_{fill}$;
17 Bin packing the rest atoms: $\mathrm{S}_{gen} \leftarrow \mathrm{S}_{dfs} \cup FFD(\varnothing, G'_q)$;
```

slice utilization, it also demands increased computation of atom weights to adjust similarity and introduces additional redundant triples. Here, we first discuss some feasible solutions.

- *Naive Solution (Next-Fit Algorithm, NF):* Matching slices from S in sequential order; for each slice S, adding S to S_{mat} as long as $S \subseteq G_q$, then updating G_q as $G_q - S$. The time complexity is $O(mn)$ ($m = |G_q|, n = |\mathrm{S}|$).
- *Greedy Solution (Greedy Algorithm, GRD)* Iteratively selecting the slice S that contributes the maximum number of new atoms, ensuring the slices S_{mat} chosen are mutually disjoint, and repeats this process until no further slices can be chosen. The time complexity is $O(mn^2)$.
- *Optimal Solution:* This problem can be solved by integer programming, but the huge slice amount results in prohibitive complexity.

The complexity of both NF and GRD algorithms is determined by the slice amount $n (n \gg m)$. For large-scale KGs with an exceedingly high number of slices, we improve the slice-matching algorithm by minimizing the number of slices required for matching. Our slice matching algorithm contains two stages as shown in Algorithm 3. Specifically, in the first stage on Lines 2-7, we gather the k-hop neighbors N_q^k of query q (Line 2) and collect the slices $\mathcal{S}_N$ whose query entity is in N_q^k (Line 3). These slices $\mathcal{S}_N$ are then sorted based on their historical utilization times (Line 4) and matched preferentially on Lines 5-8. Because these 'nearby' slices are more likely to be matched, the size of atom set G'_q would be significantly decreased. Secondly, we match slices that contain at least one atom in the rest G'_q on Lines 9-16. Such that a large number of slices that do not intersect with G'_q are excluded. Following the GRD algorithm, we sort the candidate slices by slice capacity on Line 11 thereby matching the slice having more atoms first on Lines 12-16. In addition, to ensure the low redundancy rate δ_R in the S_{mat}, we remove slices with a lower payload than the capacity threshold α on Lines 6 and 13. The ratio of slice matching $|G'_q|/|G_q|$ is determined by multiple factors including both atom list and slice data, but the capacity of all matched slices can be guaranteed ($\geqslant \alpha$).

Complexity Analysis: Different from the NF algorithm with $O(mn)$ complexity, the complexity of Algorithm 3 is around $O(mn')$, where $n' \ll n$ denotes the number of slices calculated in two stages. The complexity of the additional sorting operation is independent of m and n thus the cost is negligible. Then, gathering S_a on Line 11 is a $O(1)$ operation because we construct a mapping dictionary to record the slice IDs corresponding to each query entity. Removing low payload slices is $O(1)$ because we can record the capacity of each slice in the slice data. In summary, the total complexity of the proposed algorithm is much lower than that of FF and GRD.

4.3 SiGMa Approximation Guarantee.

According to the complexity analysis of two sub-algorithms, the total time complexity of SiGMa for one query subgraph is around $O(m \cdot max(n', \log m))$. Since each entity is only sliced once, the time cost of SiGMa is affordable. We further discuss the approximation guarantee of SiGMa. The SiGMa algorithm is designed to optimize both slice redundancy (δ_R) and slice utilization (δ_U). Because δ_U is difficult to measure, we focus on the approximation guarantee of minimizing the δ_R metric.

```
Algorithm 3: Slice Matching
Input  : Atom list $G_q = \{(e_a, w_a)\}$, Slice data $\mathrm{S}$, hop
         number $k$, capacity threshold $\alpha$.
Output : Matched slices $\mathrm{S}_{mat}$, rest atom set $G'_q$
1  Initialize the empty set $\mathrm{S}_{mat}$, $G'_q \leftarrow G_q$;
2  Gather the $k$-hop neighbors $N_q^k$ of $q$;
3  Gather the slices $\mathcal{S}_N \subset \mathcal{S}$ whose query entity is in $N_q^k$;
4  Sort $\mathcal{S}_N$ by the historical utilization times;
5  foreach slice $S$ in $\mathcal{S}_N$ do
6      if $size(S) \geqslant \alpha$ and $S \subseteq G'_q$ then
7          $G'_q \leftarrow G'_q - S$;
8          $\mathrm{S}_{mat} \leftarrow \mathrm{S}_{mat} \cup \{S\}$;
9  foreach atom $a$ in $G'_q$ do
10     Gather the slices $\mathrm{S}_a \subset \mathrm{S}$ that containing $a$;
11     Sort $\mathrm{S}_a$ by the capacity in decreasing order;
12     for each slice $S$ in $\mathrm{S}_a$ do
13         if $size(S) \geqslant \alpha$ and $S \subseteq G'_q$ then
14             $G'_q \leftarrow G'_q - S$;
15             $\mathrm{S}_{mat} \leftarrow \mathrm{S}_{mat} \cup \{S\}$;
16             break;
```

Theorem 3. *The slice capacity threshold $\alpha \geqslant 0.9$ guarantees SiGMa outperforming the FFD algorithm when the slice amount $|\mathrm{S}_{mat}| + |\mathrm{S}_{dfs}| > 3$.*

Proof. Given a subgraph G_q, the output slice set of SiGMa algorithm consists of three parts, S_{mat} by matching, S_{dfs} by generating with DFS order, and the rest slices S_{ffd} generated via the FFD algorithm on Line 17 of Algorithm 2.

We first deduce the upper bound of δ_R and discuss the slice amount in each part. The slice redundancy in the first two parts is constrained by the minimum capacity threshold α. While the redundancy rate of the third part is determined by FFD, whose tight approximation ratio is known as $(\frac{11}{9}\mathrm{OPT} + \frac{6}{9})$ from [11], where OPT refers to the optimal value.

Assume the atom set G_1 is the atoms stored in S_{mat} and S_{dfs}, the other atoms $G_2 = G_q - G_1$ are inputted into the final FFD process. For one subgraph G_q, we prove the upper bound of the δ_R metric as follows:

$$\begin{aligned}\delta_R(G_q) &= \frac{|\mathrm{S}_q|}{\lceil |G_q|/h \rceil} = \frac{|\mathrm{S}_{mat}| + |\mathrm{S}_{dfs}| + |\mathrm{S}_{ffd}|}{\lceil |G_q|/h \rceil} \\ &\leqslant \frac{\lceil |G_1|/(\alpha h) \rceil + |\mathrm{S}_{ffd}|}{\lceil |G_q|/h \rceil} \leqslant \frac{\lceil |G_1|/(\alpha h) \rceil + \lceil \frac{11}{9} \lceil |G_2|/h \rceil + \frac{6}{9} \rceil}{\lceil |G_q|/h \rceil} \end{aligned} \tag{5}$$

Then, we discuss the value range of α that guarantees SiGMa outperforming FFD. The statement is true if and only if the following inequalities hold:

$$\left\lceil \frac{|G_1|}{\alpha h} \right\rceil + \left\lceil \frac{11}{9} \left\lceil \frac{|G_2|}{h} \right\rceil + \frac{6}{9} \right\rceil < \left\lceil \frac{11}{9} \left\lceil \frac{|G_q|}{h} \right\rceil + \frac{6}{9} \right\rceil \tag{6}$$

Given $\beta = \lceil |G_1|/h \rceil$ and $\lceil a \rceil + \lceil b \rceil \leqslant \lceil a + b \rceil + 2$, we have:

$$\frac{\beta}{\alpha} < \left\lceil \frac{11}{9}\beta + \frac{6}{9} \right\rceil - 2 \Rightarrow \alpha > \frac{9\beta}{11\beta - 3} \tag{7}$$

Finally, according to numerical calculation, we prove that when the slice number $|\mathrm{S}_{mat}| + |\mathrm{S}_{dfs}|$ is more than three ($\beta > 3$), the δ_R metric of SiGMa is lower than that of FFD as long as $\alpha \geqslant 0.9$. □

In large-scale KGs, the slice amount for one query subgraph is usually more than ten, as shown in Table 1. According to Theorem 3, a high capacity threshold $\alpha \geqslant 0.9$ can guarantee the performance of SiGMa for minimizing slice redundancy.

5 EXPERIMENTS

5.1 Experimental Setup

Inductive KG Datasets. Current benchmarks for inductive KG reasoning, typically sourced from small-scale KG datasets, consist of two disjoint subgraphs, *train graph* and *test graph*, with only thousands of entities and triples [43]. To assess our framework's efficiency and scalability, we create four new inductive benchmarks from larger, real-world KG datasets: Ogbl-wikiKG2 [19], FB5M [5], ConceptNet [41], and Freebase [2]. The Ogbl-wikiKG2 is derived from Wikidata, while FB5M is a Freebase subset. Following the small-scale benchmark work [43], we construct the *test graph* by sampling a set of entities as central nodes and taking the union of the 2-hop query subgraphs of these entities. To ensure the *test graph* has enough relation types, we sample at most N_s relation-specific triples per relation type and take their head entities as the central nodes. For the above four KGs, the N_s values are [100, 100, 20, 5]. To address sparsity issues, we only maintain relation types with at least ten triples in the original KG. To prevent exponential growth, we impose a maximum degree limit of 10,000 for all entities. The *train graph* comprises the remaining triples whose entities are not in *test-graph* and relation exists in *test graph*. The statistics of these inductive benchmarks are given in Table 2.

Table 2: Statistics of inductive KG datasets. OgblKG2 denotes the Ogbl-wikiKG2 dataset, validation/test sets are comprised of valid/test queries using triples in Test Graph as facts.

KG Dataset		OgblKG2	FB5M	ConceptNet	Freebase
Origin	relations	535	7,523	50	14,851
	entities	2,500,604	3,988,105	28,370,083	86,054,501
	triples	17,137,181	17,872,174	34,074,917	338,669,200
Train Graph	relations	457	3,935	50	8,665
	entities	2,430,605	2,294,045	28,251,829	85,076,400
	triples	10,095,634	5,436,199	32,392,485	281,933,370
	epoch queries	30,891	44,211	9,665	70,496
Test Graph	relations	457	3,935	50	8,665
	entities	120,882	785,700	166,479	1,850,074
	triples	391,351	1,482,923	217,555	2,975,287
	valid queries	29,934	112,362	31,590	178,174
	test queries	49,188	175,704	39,862	292,110

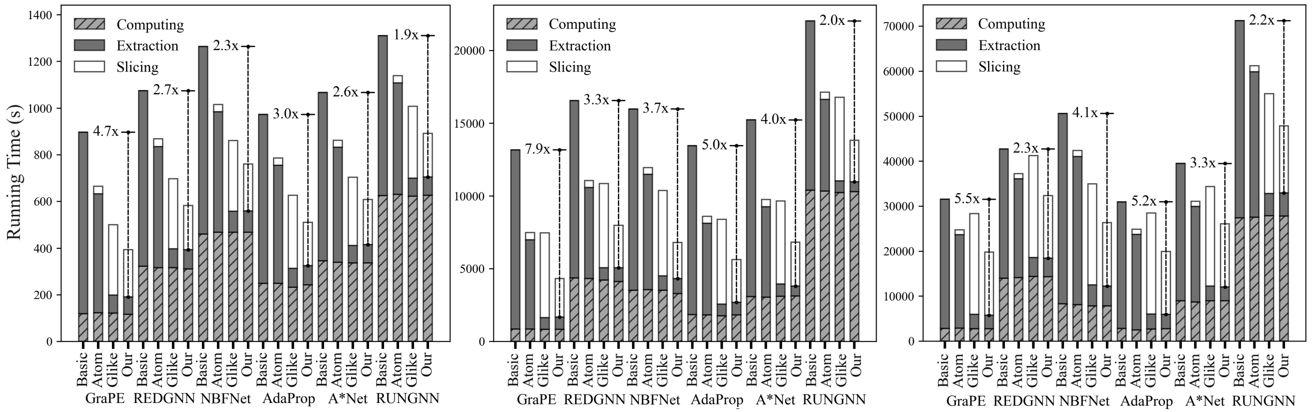
Models and Evaluation Metrics. We employ six recent GNN-based models for inductive KG reasoning, including REDGNN [58], NBFNet [66], AdaProp [59], A*Net [65], GraPE [51] and RUNGNN [54]. By default, we set the hidden dimension to 32, the layer number L to 3, and the batch size to 16. We train the six models by minimizing the multi-class cross-entropy loss. Following the default settings of these models [58], we augment the triples in $\mathcal{G}$ with reverse and identity relations. To evaluate the performance of subgraph slicing algorithms, we utilize the slicing score and two metrics, Slice Redundancy Rate (δ_R) and Slice Utilization Rate (δ_U), defined in Equation 3. Besides, for the efficiency of the training framework, we also measure the cost time of different stages, the cache hit ratio, the total slice amount, and slice storage memory.

Comparison Baselines. We compare TIGER with three baselines: *Basic*, *Atom*, and *Glike*. *Basic* refers to the basic training pipeline of inductive GNN models as depicted in Figure 2. *Atom* records atom-based subgraphs in memory and loads subgraph atoms accelerated by Atom Cache. This baseline represents the general solution of caching neighbor information in previous GNN training systems [37, 62], and its bottleneck is the massive SSD random access. Following the 'Slice&Cache' procedure of TIGER, the *Glike* baseline utilizes the dynamic caching mechanism in Ginex [35] for slice caching, but it divides subgraph triples into equal-length slices sequentially without slice reuse, thereby the large volume of slices would be its major bottleneck. As we focus on optimizing the subgraph extraction process, studies contributing to the other aspects of GNN training are excluded in our experiments.

Implementation Details. For the default TIGER training system, we set the super-batch size to 800, the slice size h to 2048, the minimum capacity threshold α to 0.9, and the hop number k in SiGMa to 1. We will discuss the sensitivity of hyperparameters in Section 5.6. For a fair efficiency comparison, we use a sampled subset of factual triples from the *train-graph* as standard training queries for all epochs, referred to as "epoch queries" in Table 2. All experiments are performed on Intel Xeon Gold 6238R 26-core CPU and NVIDIA RTX A5000 GPU. We utilize a 2TB PCIe SSD and the total main memory is 64GB. The framework is implemented in Python using the PyTorch framework.

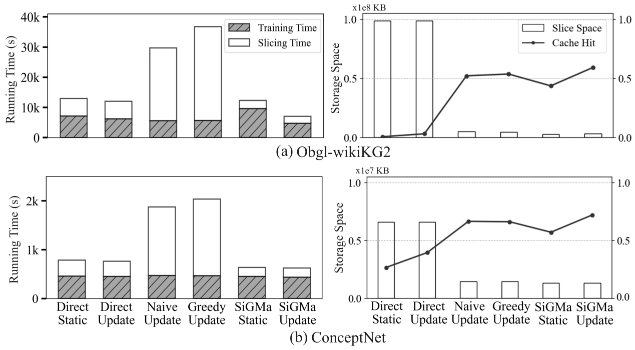
5.2 Efficiency w.r.t. the three Baselines

We present the end-to-end running time for a single training epoch of six GNN-based models in Figure 6. These models are trained using both TIGER (*Our*) and three baselines. The total running time



(a) ConceptNet (b) Obgl-wikiKG2 (c) FB5M

Figure 6: Comparison of running time (s) for one training epoch on three large-scale KG datasets.



(a) Obgl-wikiKG2

(b) ConceptNet

Figure 7: Ablation studies of the TIGER and five variants. Training time (s) refers to the total time of subgraph extraction and model computing.

is subdivided into three components: the time spent on subgraph slicing, subgraph extraction, and model computing. The slicing time includes slice storage time and is only substantial during the initial training epoch. In subsequent epochs training the same queries, the model can directly extract subgraph slices. Thereby we observe that TIGER without subgraph slicing significantly outshines the *Basic* pipeline, registering a speedup ranging from 1.9× to 7.9×. As subgraph slicing eliminates redundant extraction operations, the time reduction would become more pronounced with an increase in the number of training epochs.

The *Atom* baseline outpaces *Basic* due to the preloaded atom-based subgraphs, but extracting substantial atoms via SSD random access still consumes considerable time. The lower extraction time of *Glike* and *Our* underlines the effectiveness of slice-based subgraph extraction due to more SSD sequential access. The subgraph slicing time of TIGER is at least 1.5 times faster than that of the second-best baseline, *Glike*. Moreover, due to the sequential slicing in *Glike*, it incurs a significantly higher storage space requirement for slicing data on SSDs, which will be discussed in Section 5.3.

In the comparison of the six GNN models, GraPE stands out for its notable speedup, attributed to its efficient model computation achieved through path pruning. Generally, the extent of speedup is linked to the duration of GNN model computation. Utilizing path pruning techniques, AdaProp and A*Net exhibit higher speedups in TIGER than REDGNN and NBFNet, while RUNGNN, an enhanced version of REDGNN, emerges as the slowest one.

5.3 Efficiency due to Individual Parts

To evaluate the system design of TIGER and two core processes: SiGMa slicing algorithm and subgraph caching mechanism, we have implemented five types of variants and conducted a series of ablation studies training the REDGNN model. Specifically, regarding the slicing process, we implemented three variants of SiGMa. *Direct* refers to dividing subgraph triples into equal-length slices sequentially without slice reuse. In contrast, *Naive* and *Greedy* apply the classical algorithms mentioned in Section 4 for slice generation and matching. For the caching process, we implemented a *Static* slice cache, which is initialized like the Slice Cache in TIGER (*Update*) but does not update during mini-batch loops. The related experimental results are reported in Figure 7.

Our solution in TIGER, *SiGMa+Update*, outperforms other baselines in all four aspects: training time, slicing time, slice storage space, and the cache hit ratio. We further discuss three key components in TIGER by comparing these baselines: (1) **Slice utilization:** The two variants using *Direct* train faster than two classical slicing algorithms, but the simplistic slicing strategy without slice reuse generates a large number of independent slices, leading to considerable slice storage and low cache hits. (2) **Algorithm performance:** The *Naive* and *Greedy* variants utilize classical algorithms for subgraph slicing and achieve a higher cache hit ratio than *Direct*. However, their slicing time is longer than SiGMa's due to higher complexity, and they also have a lower cache hit ratio compared to SiGMa. (3) **Caching updates:** The cache hit ratio of *Static* caching is lower than that of *Update*. Hence, the two variants using static slice cache take a longer training time, which indicates the necessity of dynamic cache replacement.

5.4 Performance of Slicing Algorithms

We compare the subgraph slicing performance of different matching and generating algorithms on three datasets in Table 3, where *GRD+NF* denotes using the Greedy slice-matching algorithm and the Next-Fit slice-generating algorithm, and *SiGMa +SiGMa* denotes

Table 3: Performance comparison of slicing algorithms.

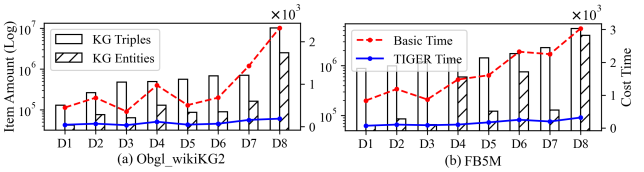
ConceptNet	Cost Time	Slice Amount	$\delta_R\downarrow$	$\delta_U\downarrow$	Score↓
NF+NF	133s (×2.50)	46692 (×1.24)	1.19	0.86	1.02
NF+FFD	186s (×3.50)	43739 (×1.17)	1.11	0.86	0.95
NF+SiGMa	128s (×2.41)	**36715 (×0.98)**	2.12	**0.55**	1.18
GRD+NF	133s (×2.50)	46631 (×1.24)	1.18	0.86	1.02
GRD+FFD	188s (×3.53)	43402 (×1.16)	1.11	0.85	0.94
GRD+SiGMa	123s (×2.30)	36816 (×0.98)	1.30	0.65	0.84
SiGMa+NF	54s (×1.02)	47103 (×1.25)	1.12	0.90	1.01
SiGMa+FFD	70s (×1.31)	43675 (×1.16)	**1.09**	0.86	0.94
SiGMa+SiGMa	**53s (×1.00)**	37542 (×1.00)	1.17	0.71	**0.83**
ObglKG2	**Cost Time**	**Slice Amount**	$\delta_R\downarrow$	$\delta_U\downarrow$	**Score↓**
NF+NF	66s (×2.14)	11076 (×1.80)	1.18	0.28	0.33
NF+FFD	83s (×2.70)	10408 (×1.69)	1.16	0.27	0.31
NF+SiGMa	46s (×1.51)	**5899 (×0.96)**	1.64	**0.13**	0.22
GRD+NF	95s (×3.09)	11427 (×1.86)	1.18	0.29	0.34
GRD+FFD	121s (×3.94)	10263 (×1.67)	1.16	0.26	0.30
GRD+SiGMa	93s (×3.04)	6061 (×0.99)	1.24	0.15	0.19
SiGMa+NF	40s (×1.32)	11554 (×1.88)	1.13	0.30	0.34
SiGMa+FFD	47s (×1.53)	11023 (×1.79)	**1.12**	0.29	0.33
SiGMa+SiGMa	**31s (×1.00)**	6146 (×1.00)	1.16	0.16	**0.18**
FB5M	**Cost Time**	**Slice Amount**	$\delta_R\downarrow$	$\delta_U\downarrow$	**Score↓**
NF+NF	148s (×4.84)	16675 (×1.39)	1.18	0.52	0.62
NF+FFD	59s (×1.92)	14827 (×1.24)	1.14	0.49	0.55
NF+SiGMa	44s (×1.44)	11887 (×0.99)	2.14	**0.35**	0.75
GRD+NF	40s (×1.30)	16606 (×1.38)	1.18	0.52	0.62
GRD+FFD	62s (×2.03)	14579 (×1.22)	1.14	0.48	0.54
GRD+SiGMa	49s (×1.61)	**11855 (×0.99)**	1.31	0.37	0.49
SiGMa+NF	**26s (×0.84)**	17657 (×1.47)	1.14	0.56	0.64
SiGMa+FFD	186s (×6.05)	14931 (×1.24)	**1.11**	0.49	0.54
SiGMa+SiGMa	31s (×1.00)	11993 (×1.00)	1.17	0.39	**0.45**

our solution with two proposed algorithms. After conducting a super-batch loop with 3,200 queries, we collect the performance metrics including the cost time of the slicing process, the total slice number, two slice quality metrics, and the slicing score $\delta_R \times \delta_U$. From experimental results, we observe that *SiGMa +SiGMa* achieves the best quality score with around 40% of the cost time and 75% of the slice amount on average. *NF+SiGMa* obtains the lowest δ_U and a lower slice amount but gets the highest δ_R metric because the Next-Fit algorithm cannot avoid high-redundancy slices. *SiGMa +FFD* gets the lowest δ_R but the other three metrics are weak, which indicates minimizing a single metric is not enough. Comparing three slice-generating algorithms solely, SiGMa gets the best δ_U but the highest δ_R. After combining with the SiGMa slice-matching algorithm, the two metrics are balanced thereby achieving the best quality score.

5.5 Scalability Studies

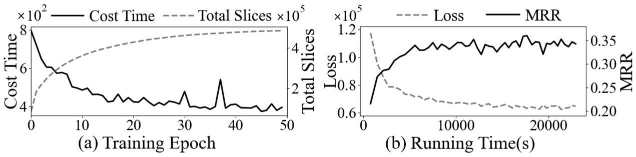
In Figure 8, we present a scalability study with varying-sized datasets generated from large-scale KGs. Additional experimental results can be found in our Technical Report [52]. For each KG, we generate eight datasets with different amounts of entities and queries, and compare the training time of REDGNN with the *Basic* baseline and TIGER for 1,000 queries. We observe a significant increase in the time required for a single training epoch of the Basic framework when applied to larger-scale datasets. Conversely, the stable performance exhibited by TIGER underscores its superior scalability.

Besides, in Figure 9, we report the training curves of GraPE [51] on the largest Freebase dataset over 50 training epochs with 4000 mini-batches per epoch. Figure 9(a) presents the changes in the cost time and the total slice amount per epoch. The results indicate a rapid increase in slice count initially and a stabilizing reduction in epoch time cost due to faster subgraph loading from accumulated slices, showcasing TIGER's training efficiency through the Slice&Cache procedure. Figure 9(b) presents the training loss curve and the MRR metric, demonstrating the effectiveness of TIGER for training inductive KG reasoning models.



(a) Obgl_wikiKG2 (b) FB5M

Figure 8: Scalability studies on various-scale KG datasets.



(a) Training Epoch (b) Running Time(s)

Figure 9: Training curves on the Freebase dataset.

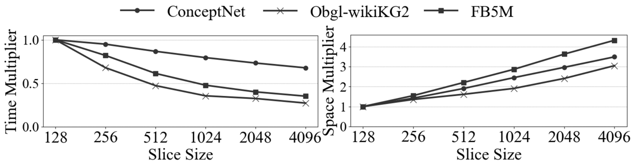


Figure 10: Slicing time and storage space w.r.t. slice sizes

5.6 Sensitivity Studies

We conduct sensitivity studies using the REDGNN model with varying three hyperparameters of TIGER that may affect performance. **Slice Size:** In Figure 10, we assess the impact of varying slice sizes (h) on the performance of TIGER. As the slice size increases, the subgraph slicing time reduces. This is because a larger slice can accommodate more triples, thereby reducing the slice operations in TIGER. On the contrary, larger slice sizes do result in expanded slice storage space, as they lead to more redundant space within slices and lower levels of slice reuse. Therefore, the selection of the slice size is a balancing act that should take into account both the desire to accelerate slicing time and the constraints imposed by storage space limitations.

Minimum Capacity: The minimum capacity threshold $\alpha \in (0, 1]$ directly determines the slice redundancy of SiGMa. As shown in Figure 11, a higher minimum capacity leads to a smaller slice amount and better slicing score, which is fitting with Equation 7. The inflection point of the curve occurs at around 0.9 because the slice utilization would be hindered when the minimum capacity is too high. Therefore, to balance the slice utilization and slice redundancy, we set the minimum capacity α to 0.9 by default.

Slice Cache Size: The size of the slice cache crucially affects the number of slices that can be stored in the main memory for subgraph loading, influencing the efficiency of the input gather stage during mini-batch processing. Our analysis, depicted in Figure 12, demonstrates that on three KG datasets, both the runtime for input gathering decreases and cache hit rates increase with larger slice cache sizes. Additionally, since input gathering time encompasses both slice loading and subgraph construction, the impact of cache hits on reducing loading time diminishes beyond a 60% hit rate, with subgraph construction times prevailing.

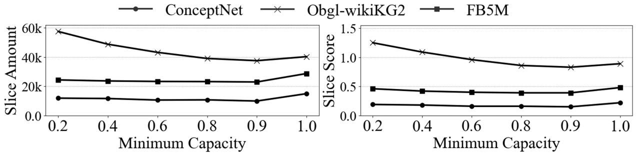


Figure 11: Slice amount and slice score w.r.t. the threshold α.

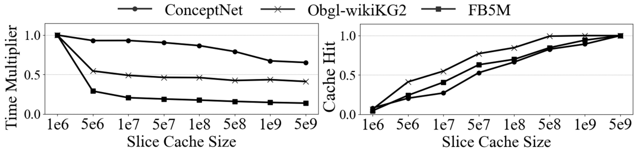


Figure 12: Slice loading time and hit ratio w.r.t. cache sizes.

6 RELATED WORK

6.1 GNN-based Knowledge Graph Reasoning

Traditional embedding-based KG reasoning methods [48, 49, 53] learn continuous embedding vectors for each entity and relation, such as TransE [4], DistMult [55], RotatE [42]. Earlier GNN-based methods, R-GCN [40] and CompGCN [44], encode entity-specific embeddings by combining the aggregated message from neighbors with relation-specific parameters [50]. To handle unseen entities for inductive KG reasoning, NeuralLP [56] and DRUM [39] encode entities by aggregating the features along all the paths that reach the candidate entity from the query entity. Recent studies concentrate on subgraph-based inductive KG reasoning. MorsE [7] learns transferable meta-knowledge in entity-independent modules to produce embeddings for unseen entities. INDIGO [27] maps the existing KG and candidate triples to a node-annotated graph for pair-wise encoding. GraIL [43] extracts an enclosing subgraph for each candidate triple and train relational GNNs independent of any specific entities. However, encoding triple-dependent subgraphs still suffer from high computational complexity. Current state-of-the-art GNN-based inductive methods, represented by REDGNN [58] and NBFNet [66], follow a progressive GNN message-passing paradigm from the source node to multi-hop neighbor entities layer by layer. RUNGNN [54] utilizes a fusion gate unit tailored to query-related contexts for modeling the sequential relation composition. AdaProp [59] and A*Net [65] utilize attention mechanisms to select top K edges (or nodes) in each GNN iteration. GraPE [51] introduces an entropy-guided graph percolation procedure to maintain the shortest paths while eliminating redundant paths. Although those recent models are more efficient, they rely on the initial extraction of a complete subgraph from large-scale KGs. TIGER distinguishes itself from prior research by presenting a general framework aimed at efficient subgraph extraction, rather than just another standalone reasoning model. It extends support to various leading GNN-based reasoning models, enabling their efficient training on large-scale KGs while maintaining their reasoning performance.

6.2 Large-scale KGE/GNN Training System

To address the efficiency and scalability challenges with large graphs, there are some well-engineered systems for accelerating KGE training, such as DGL-KE [61], HET-KG [9], and SMORE [37]. However, these graph embedding systems mainly focus on distributed parallelism and embedding storage, which cannot be directly used for large-scale inductive KG reasoning due to the more complex nature of the GNN models [13, 24, 29, 34, 64]. Notably, many scalable frameworks work on large-scale GNN training, but they do not specifically address the issue of subgraph extraction [6, 15, 25, 32, 33]. NeuGraph [30] harmoniously combines graph computation optimizations with aspects like data partitioning, scheduling, and parallelism, within dataflow-oriented deep learning frameworks. AliGraph [62] refines sampling operators for distributed GNN training, and minimizes network communication by caching nodes on local systems. DistDGL [60], a distributed GNN training framework, distributes the graph and its related vector data among machines. Despite their efficiencies, the graph sampling and partitioning algorithms tend to obstruct subgraph completeness. Therefore, additional data communication becomes unavoidable to compile the entire subgraph information. To address the subgraph extraction issue, some recent GNN training systems, like AliGraph [62] and Ginex [35], cache the neighbor information for a few central nodes, but cannot work well when dealing with frequent extraction of large subgraphs. In contrast, our work designs subgraph slicing algorithms and utilizes slice-level caching to optimize this process, for demands of loading entire subgraph data.

7 CONCLUSION

We proposed TIGER, a novel inductive KG reasoning framework with improved efficiency and scalability. To accelerate subgraph extraction in each mini-batch loop, TIGER employs a newly designed Slice&Cache procedure in which subgraph triples are sliced to enhance SSD sequential access and cached in memory via a dynamic caching mechanism. An atom-based subgraph slicing algorithm SiGMa with approximate guarantee is proposed to balance slice redundancy and utilization while achieving low computational complexity. Our experimental results on four large-scale inductive datasets demonstrated that TIGER achieves average 3.7× faster training time than the basic training procedure on six state-of-the-art GNN-based reasoning models.

8 LIMITATION

There are further optimization opportunities of the TIGER framework: (1) Optimizing Scope: TIGER focuses exclusively on optimizing the subgraph extraction phase, without further optimizing other phases of the training pipeline, such as the acceleration of GNN calculations. (2) Application Scope: TIGER is designed for a series of the latest GNN-based KG inductive reasoning models, rather than all GNN-based models or previous inductive models. Although the slicing techniques in TIGER would be beneficial, loading graph data for these models may face unique challenges not covered by TIGER. The future work includes using multiple GPUs for distributed parallelism and extending to the other graph tasks.

ACKNOWLEDGMENTS

This research is supported by the Ministry of Education, Singapore, under its AcRF Tier-2 Grant (T2EP20122-0003). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the Ministry of Education, Singapore.

REFERENCES

[1] L. A. Belady. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal* 5, 2 (1966), 78–101.

[2] Kurt D. Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. 2008. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. 1247–1250.

[3] Stephen Bonner, Ian P Barrett, Cheng Ye, Rowan Swiers, Ola Engkvist, Charles Tapley Hoyt, and William L Hamilton. 2022. Understanding the performance of knowledge graph embeddings in drug discovery. *Artificial Intelligence in the Life Sciences* 2 (2022), 100036.

[4] Antoine Bordes, Alberto García-Durán, Jason Weston, and Oksana Yakhnenko. 2013. Translating Embeddings for Modeling Multi-relational Data. In *Proceedings of the 27th Annual Conference on Neural Information Processing Systems, December 5-8, 2013, Lake Tahoe, Nevada, United States*. 2787–2795.

[5] Antoine Bordes, Nicolas Usunier, Sumit Chopra, and Jason Weston. 2015. Large-scale Simple Question Answering with Memory Networks. *CoRR* abs/1506.02075 (2015).

[6] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. 2021. DGCL: an efficient communication library for distributed GNN training. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*. ACM, 130–144.

[7] Mingyang Chen, Wen Zhang, Yushan Zhu, Hongting Zhou, Zonggang Yuan, Changliang Xu, and Huajun Chen. 2022. Meta-Knowledge Transfer for Inductive Knowledge Graph Embedding. In *SIGIR '22: The 45th International ACM SIGIR Conference on Research and Development in Information Retrieval, Madrid, Spain, July 11 - 15, 2022*. ACM, 927–937. https://doi.org/10.1145/3477495.3531757

[8] Shumin Deng, Ningyu Zhang, Wen Zhang, Jiaoyan Chen, Jeff Z. Pan, and Huajun Chen. 2019. Knowledge-Driven Stock Trend Prediction and Explanation via Temporal Convolutional Network. In *Companion of The 2019 World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*. ACM, 678–685.

[9] Sicong Dong, Xupeng Miao, Pengkai Liu, Xin Wang, Bin Cui, and Jianxin Li. 2022. HET-KG: Communication-Efficient Knowledge Graph Embedding Training via Hotness-Aware Cache. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 1754–1766.

[10] Xin Luna Dong. 2023. Generations of Knowledge Graphs: The Crazy Ideas and the Business Impact. *Proc. VLDB Endow.* 16, 12 (2023), 4130–4137.

[11] György Dósa. 2007. The tight bound of first fit decreasing bin-packing algorithm is FFD (I)<= 11/9 OPT (I)+ 6/9. In *International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*. Springer, 1–11.

[12] Nouha Dziri, Sivan Milton, Mo Yu, Osmar R. Zaïane, and Siva Reddy. 2022. On the Origin of Hallucinations in Conversational Models: Is it the Datasets or the Models?. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL 2022, Seattle, WA, United States, July 10-15, 2022*. Association for Computational Linguistics, 5271–5285.

[13] Peng Fang, Arijit Khan, Siqiang Luo, Fang Wang, Dan Feng, Zhenli Li, Wei Yin, and Yuchao Cao. 2023. Distributed Graph Embedding with Information-Oriented Random Walks. *Proc. VLDB Endow.* 16, 7 (2023), 1643–1656.

[14] Uriel Feige. 1998. A Threshold of ln *n* for Approximating Set Cover. *J. ACM* 45, 4 (1998), 634–652.

[15] Swapnil Gandhi and Anand Padmanabha Iyer. 2021. P3: Distributed Deep Graph Learning at Scale. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*. USENIX Association, 551–568.

[16] Congcong Ge, Xiaoze Liu, Lu Chen, Baihua Zheng, and Yunjun Gao. 2021. LargeEA: Aligning Entities for Large-scale Knowledge Graphs. *Proc. VLDB Endow.* 15, 2 (2021), 237–245.

[17] Chenghua Gong, Yao Cheng, Xiang Li, Caihua Shan, and Siqiang Luo. 2024. Learning from Graphs with Heterophily: Progress and Future. arXiv:2401.09769 [cs.SI]

[18] William L. Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. 1024–1034.

[19] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

[20] Shaoxiong Ji, Shirui Pan, Erik Cambria, Pekka Marttinen, and Philip S. Yu. 2022. A Survey on Knowledge Graphs: Representation, Acquisition, and Applications. *IEEE Trans. Neural Networks Learn. Syst.* 33, 2 (2022), 494–514.

[21] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Yejin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of Hallucination in Natural Language Generation. *ACM Comput. Surv.* 55, 12 (2023), 248:1–248:38.

[22] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net.

[23] Adrian Kochsiek and Rainer Gemulla. 2021. Parallel Training of Knowledge Graph Embedding Models: A Comparison of Techniques. *Proc. VLDB Endow.* 15, 3 (2021), 633–645.

[24] Adam Lerer, Ledell Wu, Jiajun Shen, Timothée Lacroix, Luca Wehrstedt, Abhijit Bose, and Alex Peysakhovich. 2019. Pytorch-BigGraph: A Large Scale Graph Embedding System. In *Proceedings of Machine Learning and Systems 2019, MLSys 2019, Stanford, CA, USA, March 31 - April 2, 2019*. mlsys.org.

[25] Ningyi Liao, Dingheng Mo, Siqiang Luo, Xiang Li, and Pengcheng Yin. 2022. SCARA: Scalable Graph Neural Networks with Feature-Oriented Optimization. *Proc. VLDB Endow.* 15, 11 (2022), 3240–3248.

[26] Ningyi Liao, Dingheng Mo, Siqiang Luo, Xiang Li, and Pengcheng Yin. 2023. Scalable decoupling graph neural network with feature-oriented optimization. *The VLDB Journal* (2023), 1–17.

[27] Shuwen Liu, Bernardo Cuenca Grau, Ian Horrocks, and Egor V. Kostylev. 2021. INDIGO: GNN-Based Inductive Knowledge Graph Completion Using Pair-Wise Encoding. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan (Eds.). 2034–2045.

[28] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. WiscKey: Separating Keys from Values in SSD-Conscious Storage. *ACM Trans. Storage* 13, 1 (2017), 5:1–5:28.

[29] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. 2020. Rosetta: A Robust Space-Time Optimized Range Filter for Key-Value Stores. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 2071–2086. https://doi.org/10.1145/3318464.3389731

[30] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. NeuGraph: Parallel Deep Neural Network Computation on Large Graphs. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*. USENIX Association, 443–458.

[31] Silvano Martello and Paolo Toth. 1990. Lower bounds and reduction procedures for the bin packing problem. *Discret. Appl. Math.* 28, 1 (1990), 59–70.

[32] Seungwon Min, Kun Wu, Sitao Huang, Mert Hidayetoglu, Jinjun Xiong, Eiman Ebrahimi, Deming Chen, and Wen-mei W. Hwu. 2021. Large Graph Convolutional Network Training with GPU-Oriented Data Communication Architecture. *Proc. VLDB Endow.* 14, 11 (2021), 2087–2100.

[33] Dingheng Mo, Fanchao Chen, Siqiang Luo, and Caihua Shan. 2023. Learning to Optimize LSM-trees: Towards A Reinforcement Learning based Key-Value Store for Dynamic Workloads. *Proc. ACM Manag. Data* 1, 3 (2023), 213:1–213:25. https://doi.org/10.1145/3617333

[34] Jason Mohoney, Roger Waleffe, Henry Xu, Theodoros Rekatsinas, and Shivaram Venkataraman. 2021. Marius: Learning Massive Graph Embeddings on a Single Machine. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*. 533–549.

[35] Yeonhong Park, Sunhong Min, and Jae W. Lee. 2022. Ginex: SSD-enabled Billion-scale Graph Neural Network Training on a Single Machine via Provably Optimal In-memory Caching. *Proc. VLDB Endow.* 15, 11 (2022), 2626–2639.

[36] Kashif Rabbani, Matteo Lissandrini, and Katja Hose. 2023. Extraction of Validating Shapes from very large Knowledge Graphs. *Proc. VLDB Endow.* 16, 5 (2023), 1023–1032.

[37] Hongyu Ren, Hanjun Dai, Bo Dai, Xinyun Chen, Denny Zhou, Jure Leskovec, and Dale Schuurmans. 2022. SMORE: Knowledge Graph Completion and Multi-hop Reasoning in Massive Knowledge Graphs. In *KDD '22: The 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 14 - 18, 2022*. ACM, 1472–1482.

[38] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: edge-centric graph processing using streaming partitions. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*. ACM, 472–488.

[39] Ali Sadeghian, Mohammadreza Armandpour, Patrick Ding, and Daisy Zhe Wang. 2019. DRUM: End-To-End Differentiable Rule Mining On Knowledge Graphs. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. 15321–15331.

[40] Michael Sejr Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. 2018. Modeling Relational Data with Graph Convolutional Networks. In *The Semantic Web - 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, June 3-7, 2018, Proceedings (Lecture Notes in Computer Science)*, Vol. 10843. Springer, 593–607.

[41] Robyn Speer, Joshua Chin, and Catherine Havasi. 2017. ConceptNet 5.5: An Open Multilingual Graph of General Knowledge. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*. 4444–4451.

[42] Zhiqing Sun, Zhi-Hong Deng, Jian-Yun Nie, and Jian Tang. 2019. RotatE: Knowledge Graph Embedding by Relational Rotation in Complex Space. In *Proceedings of the 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*.

[43] Komal K. Teru, Etienne G. Denis, and William L. Hamilton. 2020. Inductive Relation Prediction by Subgraph Reasoning. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event (Proceedings of Machine Learning Research)*, Vol. 119. PMLR, 9448–9457.

[44] Shikhar Vashishth, Soumya Sanyal, Vikram Nitin, and Partha P. Talukdar. 2020. Composition-based Multi-Relational Graph Convolutional Networks. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

[45] Denny Vrandečić and Markus Krötzsch. 2014. Wikidata: a free collaborative knowledgebase. *Commun. ACM* 57, 10 (2014), 78–85.

[46] Kai Wang, Yu Liu, Dan Lin, and Michael Sheng. 2021. Hyperbolic Geometry is Not Necessary: Lightweight Euclidean-Based Models for Low-Dimensional Knowledge Graph Embeddings. In *Findings of the Association for Computational Linguistics: EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 16-20 November, 2021*. 464–474.

[47] Kai Wang, Yu Liu, Qian Ma, and Quan Z. Sheng. 2021. MulDE: Multi-teacher Knowledge Distillation for Low-dimensional Knowledge Graph Embeddings. In *Proceedings of the WWW '21: The Web Conference 2021, Virtual Event / Ljubljana, Slovenia, April 19-23, 2021*. 1716–1726.

[48] Kai Wang, Yu Liu, and Quan Z. Sheng. 2021. Neighborhood Intervention Consistency: Measuring Confidence for Knowledge Graph Link Prediction. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*. 2090–2096.

[49] Kai Wang, Yu Liu, and Quan Z. Sheng. 2022. Swift and Sure: Hardness-aware Contrastive Learning for Low-dimensional Knowledge Graph Embeddings. In *WWW '22: The ACM Web Conference 2022, Virtual Event, Lyon, France, April 25 - 29, 2022*. ACM, 838–849.

[50] Kai Wang, Yu Liu, Xiujuan Xu, and Quan Z. Sheng. 2020. Enhancing knowledge graph embedding by composite neighbors for link prediction. *Computing* 102, 12 (2020), 2587–2606.

[51] Kai Wang, Siqiang Luo, and Dan Lin. 2023. River of No Return: Graph Percolation Embeddings for Efficient Knowledge Graph Reasoning. *arXiv preprint arXiv:2305.09974* (2023).

[52] Kai Wang, Yuwei Xu, and Siqiang Luo. 2024. Technical Report. https://github.com/KyneWang/TIGER/blob/main/Report.pdf. Last Accessed Date: 2024-06-08.

[53] Quan Wang, Zhendong Mao, Bin Wang, and Li Guo. 2017. Knowledge Graph Embedding: A Survey of Approaches and Applications. *IEEE Transactions on Knowledge and Data Engineering* 29, 12 (2017), 2724–2743.

[54] Shuhan Wu, Huaiyu Wan, Wei Chen, Yuting Wu, Junfeng Shen, and Youfang Lin. 2023. Towards Enhancing Relational Rules for Knowledge Graph Link Prediction. In *Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6-10, 2023*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, 10082–10097. https://doi.org/10.18653/V1/2023.FINDINGS-EMNLP.676

[55] Bishan Yang, Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. 2015. Embedding Entities and Relations for Learning and Inference in Knowledge Bases. In *Proceedings of the 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015*.

[56] Fan Yang, Zhilin Yang, and William W. Cohen. 2017. Differentiable Learning of Logical Rules for Knowledge Base Reasoning. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. 2319–2328.

[57] Xiangxiang Zeng, Xiang Song, Tengfei Ma, Xiaoqin Pan, Yadi Zhou, Yuan Hou, Zheng Zhang, Kenli Li, George Karypis, and Feixiong Cheng. 2020. Repurpose open data to discover therapeutics for COVID-19 using deep learning. *Journal of proteome research* 19, 11 (2020), 4624–4636.

[58] Yongqi Zhang and Quanming Yao. 2022. Knowledge Graph Reasoning with Relational Digraph. In *WWW '22: The ACM Web Conference 2022, Virtual Event, Lyon, France, April 25 - 29, 2022*. ACM, 912–924.

[59] Yongqi Zhang, Zhanke Zhou, Quanming Yao, Xiaowen Chu, and Bo Han. 2023. AdaProp: Learning Adaptive Propagation for Graph Neural Network based Knowledge Graph Reasoning. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2023, Long Beach, CA, USA, August 6-10, 2023*, Ambuj K. Singh, Yizhou Sun, Leman Akoglu, Dimitrios Gunopulos, Xifeng Yan, Ravi Kumar, Fatma Ozcan, and Jieping Ye (Eds.). ACM, 3446–3457. https://doi.org/10.1145/3580305.3599404

[60] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. 2020. DistDGL: Distributed Graph Neural Network Training for Billion-Scale Graphs. In *10th IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms, IA3 2020, Atlanta, GA, USA, November 11, 2020*. IEEE, 36–44.

[61] Da Zheng, Xiang Song, Chao Ma, Zeyuan Tan, Zihao Ye, Jin Dong, Hao Xiong, Zheng Zhang, and George Karypis. 2020. DGL-KE: Training Knowledge Graph Embeddings at Scale. In *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval, SIGIR 2020, Virtual Event, China, July 25-30, 2020*. 739–748.

[62] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. AliGraph: A Comprehensive Graph Neural Network Platform. *Proc. VLDB Endow.* 12, 12 (2019), 2094–2105.

[63] Zulun Zhu, Jiaying Peng, Jintang Li, Liang Chen, Qi Yu, and Siqiang Luo. 2022. Spiking Graph Convolutional Networks. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*, Luc De Raedt (Ed.). ijcai.org, 2434–2440.

[64] Zhaocheng Zhu, Shizhen Xu, Jian Tang, and Meng Qu. 2019. GraphVite: A High-Performance CPU-GPU Hybrid System for Node Embedding. In *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*. ACM, 2494–2504.

[65] Zhaocheng Zhu, Xinyu Yuan, Michael Galkin, Louis-Pascal A. C. Xhonneux, Ming Zhang, Maxime Gazeau, and Jian Tang. 2023. A*Net: A Scalable Path-based Reasoning Approach for Knowledge Graphs. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (Eds.). http://papers.nips.cc/paper_files/paper/2023/hash/b9e98316cb72fee82cc1160da5810abc-Abstract-Conference.html

[66] Zhaocheng Zhu, Zuobai Zhang, Louis-Pascal A. C. Xhonneux, and Jian Tang. 2021. Neural Bellman-Ford Networks: A General Graph Neural Network Framework for Link Prediction. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*. 29476–29490.