Enabling multi-agent collaboration in knowledge graph environments

Iñaki Arango[*]
Department of Biomedical Informatics
Harvard Medical School
Boston, MA 02115
inakiarango@college.harvard.edu

Ayush Noori[*]
Department of Biomedical Informatics
Harvard Medical School
Boston, MA 02115
anoori@college.harvard.edu

Lucas Vittor[*]
Department of Biomedical Informatics
Harvard Medical School
Boston, MA 02115
lvvittor@gmail.com

Joaquin Polonuer[*]
Department of Biomedical Informatics
Harvard Medical School
Boston, MA 02115
jtpolonuer@gmail.com

Marinka Zitnik[†]
Department of Biomedical Informatics
Harvard Medical School
Boston, MA 02115
marinka@hms.harvard.edu

Abstract

Knowledge graphs (KGs) are critical for grounding large language models and providing them with persistent memory. However, the development of agents capable of collaboratively building and maintaining these KGs is hindered by a lack of suitable environments. Current tools often obscure the construction process, lack standardized editing APIs, and offer poor support for concurrent, multi-agent collaboration. To address this, we introduce GRAPHWORLD, an environment for agents to build and edit graphs. GRAPHWORLD provides agents with tools to create, update, or delete nodes, edges, or properties of KGs, with support for Python, TypeScript, and Rust. GRAPHWORLD relies on DIAMOND, a compact, property-preserving storage format that permits scaling to multi-million-edge KGs. All changes are version-controlled by integrating graph-aware diffing and merging directly into Git, enabling multi-agent and human-agent collaboration through standard branching workflows.

1 Introduction

Knowledge graphs (KGs) are relational databases that use a graph-based data model to encode knowledge-informed interactions between different objects [1, 2]. Formally, a KG is defined by a set of nodes as well as a set of edges that describe relationships between the nodes. In modern heterogeneous KGs, nodes and edges have different types, and may also contain extra properties or information [3]. KGs are rapidly becoming a core component of modern large language model

[*]Equal contribution
[†]Correspondence: marinka@hms.harvard.edu

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: Scaling Environments for Agents (SEA).

(LLM)-based systems, enabling retrieval [4], grounded reasoning, and persistent memory. Integrating KGs with LLMs can equip LLMs with rich, structured factual knowledge and traceable information provenance, addressing common challenges faced by LLMs, including hallucination, indecision, poor interpretability, and lack of domain-specific knowledge [5]. Further, KGs offer a mechanism for agents to build and maintain a persistent memory of their interactions and learned knowledge [6]. LLMs and agents excel at generating datasets from structured or unstructured sources, including textual datasets (*e.g.*, LangChain's Tuna [7] and Alibaba's DataJuicer [8, 9]) and graph datasets (*e.g.*, Graphiti's Zep [10]); however, to date, most KGs are created and maintained through manual or semi-automated, non-agentic data pipelines. Tools to build and maintain KG environments both for and with agents remain underdeveloped. Current methods for KG editing often do not expose intermediate edits, lack a standard, multi-language edit API, and provide weak versioning and merge semantics for concurrent work. These obstacles make agent-driven KG editing challenging.

To address these challenges, we introduce GRAPHWORLD, an environment for agents to build and edit knowledge graphs. GRAPHWORLD provides a set of atomic edit tools that allow agents to create, update, or delete nodes, edges, and knowledge properties. These tools are accessible from Python, TypeScript, and Rust, ensuring broad interoperability across development ecosystems. To support distributed and concurrent multi-agent or human–agent collaboration, GRAPHWORLD integrates graph-aware diffs and three-way merges directly with Git. For scaling to multi-million-edge graphs, we further introduce DIAMOND, a compact, property-preserving storage format optimized for labeled property graphs. In experiments, DIAMOND achieves up to $34.1\times$ compression over widely used LPG formats, reducing storage demands while preserving all graph structure and metadata. Benchmarking on synthetic and real-world KGs demonstrates that DIAMOND consistently outperforms JSON, JSON Lines, and PG-JSON, especially as graphs grow in property density and size. The Git integration in GRAPHWORLD produces semantic diffs that reveal node- and edge-level changes rather than opaque binary differences, enabling reproducible and auditable histories of KGs. Applications of GRAPHWORLD include automated validation of human edits, natural language editing tools for domain experts, and ontology alignment workflows, all of which highlight GRAPHWORLD as a human-AI environment for KG development. In biomedical settings, for example, GRAPHWORLD compresses PrimeKG [11] to less than 9% of its original size while supporting transparent version control, facilitating large-scale analysis and sharing. More broadly, GRAPHWORLD supports multi-agent pipelines that continuously propose, review, and merge edits in knowledge graphs.

2 Related work

2.1 Tools or environments for agent-based graph interaction

Prior systems address specific features of the KG construction and agent integration workflow, converting text into graph structures using LLMs [12], managing agent memory in real time [10, 13], enabling retrieval-augmented generation (RAG) over graphs [14], or supporting versioning and provenance [15–17]. By contrast, GRAPHWORLD treats KG construction itself – observable, version-controlled, multi-language editing of property-rich labeled property graphs (LPGs) – as the first-class object.

Single-agent KG construction. Several methods exist to convert unstructured text into graph representations. LangChain's `LLMGraphTransformer`, KGGen [12], and related approaches build graphs via prompting heuristics or extraction pipelines. Methods also exist to allow single agents to build a KG; for example, Graphiti [10] provides an interface for an agent to build and query graphs. While effective for initially populating a KG, these approaches do not expose per-edit observability, structured diffs, or merge semantics, and are often bound to a single runtime or framework. By contrast, GRAPHWORLD provides a standard set of atomic graph editing operations across languages and under version control, allowing multiple agents, rather than a single LLM, to collaboratively edit a graph. Moreover, single-agent KG construction systems can adopt the observable edit tools, branching or merging semantics, and graph storage of GRAPHWORLD as a backend.

Graph retrieval for LLM reasoning. Graph-based RAG systems such as Neo4j [18], GraphRAG [19], GNN-RAG [20], KG^{2}RAG [21], KET-RAG [22], KG-RAG [23], and LlamaIndex KG indices are consumers of graphs, improving grounding and multi-hop reasoning. GRAPHWORLD sits upstream, allowing agents to build and maintain the graphs that retrieval systems depend on.

2.2 Graph data models

Various data models exist to represent graphs. GRAPHWORLD includes a version control system that operates on top of a storage format, DIAMOND, and represents graphs using a specific data model. Therefore, to explain the design choices of GRAPHWORLD, it is necessary to discuss the significant body of existing work on graph data models, storage structures, and versioning approaches. We highlight components that were adopted in GRAPHWORLD as well as features of GRAPHWORLD that differ from previous work.

Machine-readable representations of KGs often use the Resource Description Framework (RDF) data model [24], introduced at the World Wide Web Consortium (W3C) in 1999. Under RDF, a graph is represented by a collection of triples. These triples follow the structure of subject-predicate-object (*e.g.*, `(Horacio,likes,cars)`). A resource is any subject, predicate, or object, and is identified through a Unique Resource Identifier, or URI. Subjects in a triple can be a URI or a blank node, while predicates must be a URI, and objects can be a URI, a blank node, or a literal (*e.g.*, an integer, a float, a string). A graph $\mathcal{R}$ is then represented as a set of RDF triples (also called semantic triples):

$$\mathcal{R} \subseteq (\texttt{URI} \cup \texttt{blank}) \times \texttt{URI} \times (\texttt{URI} \cup \texttt{blank} \cup \texttt{literal})$$

where `URI` is the set of all URIs, and `blank` and `literal` are the sets of blank nodes and literals, respectively.

Often KGs require the attachment of non-structural information to a node or edge in the graph. RDF does not natively support this, and instead relies on reification. Reification achieves this through the use of metadata predicate types that enable writing triples about other triples, but comes at the cost of larger graph sizes, and has been a source of criticism for the format [25].

Labeled Property Graphs (LPGs) have recently emerged as a more flexible and efficient alternative to RDF [3]. They combine edge-labeled graphs, which consist of graphs where nodes are connected by directed edges that contain a label, or type, with property graphs, which allow arbitrary property information to be added to nodes and edges. LPGs have been adopted as the official storage format of the leading graph databases, including Neo4j and ArangoDB, which has driven to further adoption in production settings [26]. It is worth nothing that not all databases adopted the same exact defition of LPGs, and each provider has slightly different variants. For example, Neo4j allows any number of labels on nodes but only one label per edge, while ArangoDB supports only one label per node and one label per edge [26].

GRAPHWORLD adopts an multi-label definition of LPGs with flat properties that allow agents trained on it to interact with modern KG tooling and databases.

2.3 Storage formats and compression

While these databases offer avenues to store graphs, they do not do so in a stateless manner. To load, process, and serve the data, they database server must be on. A file format is needed to serialize KGs and enable the exchange of information in a standardized format which can be loaded by the database system or program of the user's choice. This is akin to relational database management systems (RDBMs) and the "comma-separated values" (CSV) file format that stores columnar data in a text-based encoding. To address this need, Chiba *et al.* [27] have proposed the "Property Graph Exchange Format" (PG) as a standard for representing LPGs. PG is a text-based format that encodes the graph structure as well as the node and edge properties in a human-readable format. The format specification also contains the definition of PG-JSON, a JSON-based serialization of PG that is easy to parse, and PG-JSONL, a newline-delimited JSON format. The format is designed to be easy to read and write, making it suitable for both humans and machines.

While versatile, the format can make graph files grow quickly in size, making it impractical to store on git-backed server, which oftentimes have file size restrictions. Thus, GRAPHWORLD supports the PG-JSON family of formats and also introduces DIAMOND, an LPG format that heavily compresses graphs into a fraction of the size of their PG-JSON equivalents.

There is a significant body of literature on algorithms for lossless RDF graph compression, which has been reviewed by Besta & Hoefler [28]. For example, some methods apply text compression methods to text-based graph representations [29, 30]. The seminal WebGraph framework uses lexicographic locality and reference encoding to greatly reduce storage per edge [31]. Brisaboa *et al.* [32] proposed

k^2 trees, a succinct data structure for graph adjacency that models the graph as a tree, then recursively partitions and stores the adjacency matrix to capitalize on large empty regions in sparse graphs [33, 34]. However, these methods only target topology, or the adjacency structure or derived indices. Edge or node attributes are usually not considered or stored in separate, uncompressed arrays. We could identify no compression tools compatible with storing node and edge properties, which is necessary for our use case. That is, no binary encoding format or other efficient representation exists for LPGs, motivating the need for DIAMOND. DIAMOND is described in more detail in Appendix A.2.

2.4 Version control systems

Unlike text or code, where line-based version control (à la Git) works well, KGs contain complex structured data that requires specialized versioning strategies. In particular, KG versioning approaches encounter the following challenges:

- **Conflict management and resolution.** While many existing solutions can record linear KG edit histories, they lack support for branching and merging graph versions, and thus cannot support parallel development or KG contextualization [17]. Reconciling KG merge conflicts entails more than a simple union of all changes: for example, one branch may delete an edge that another branch modifies.
- **Static and queryable storage sizes.** Naïvely storing multiple versions of large KGs can impose significant storage demands. This is especially true of modern KGs that consist of millions, or billions, of nodes and edges. For example, GenomicKG contains 347 million nodes, 1.36 billion edges, and 3.9 billion node and edge properties [35]. GenomicKB was constructed from over 30 genomic datasets and annotations which are regularly updated, necessitating versioning of this large KG. Instead of representing each version as a standalone file, which does not scale with the number of edits, binary encoding and compression techniques can be used to both encode graph structure and eliminate redundancy by only storing version changes. However, graph look-up, membership, or other queries often cannot be executed directly over a compressed KG, leading to undesirable query performance degradation.
- **Lack of standards and interoperability.** Currently, there does not exist a unified "Git for KGs" standard; existing systems often sacrifice one aspect (*e.g.*, merge support or query performance) to gain another (*e.g.*, storage efficiency or simplicity).

Several approaches have been proposed to address these challenges. Existing methods can be categorized as follows [15–17]:

- **Independent copies.** Each graph version is managed and stored as a separate dataset. This method is straightforward to implement but highly redundant and inefficient, both in terms of query time and storage.
- **Change-based versioning.** Only differences, or deltas, between versions are stored. Broadly, change-based methods reduce storage requirements but incur query performance overhead when reconstructing past versions. Many existing change-based approaches like Frommhold *et al.* [36], TailR [37], and RDF-adapted Darcs [38] lack support for branching [17]. Those that do support branching, like R43ples [39], lack streamlined merge support. Thus, current methods are not suited for parallel development.
- **Timestamp-based versioning.** Each edge is annotated with temporal validity information in a single, unified KG. Query performance of timestamp-based methods like R&Wbase [40], ConVer-G [41], and Aion [42] suffers as queries need to filter by version and time, and storage requirements also increase due to the need to preserve temporal metadata. For example, ConVer-G, which uses PostgreSQL as the storage, introduces in-graph provenance information in the graph to facilitate querying, increasing the storage demands.

Finally, inspired by Git, some systems have implemented distributed version control mechanisms for KGs. Most notably, QuitStore [36] and Git4Voc [43] adapt Git-like operations such as branching, merging, and commit tracking to RDF data. However, QuitStore [17] relies on NQuads [44, 45], an inefficient text-based storage format, to store the change information in a manner that Git tools can easily interpret and display. In fact, in the Discussion section of QuitStore, Arndt *et al.* [17] write, "The evaluation of more advanced compression systems for the stored RDF dataset or the employment of a binary RDF serialization format, such as HDT [46] is still subject to future work." Thus, to

date, no single system for versioning large-scale KGs has successfully combined efficient branching and merging with scalable storage and high-performance queries. We sought to achieve this goal in GRAPHWORLD by developing a novel versioning system for graphs, which we describe in Section 3.2.2. This version control system underlies the ability of GRAPHWORLD to support multi-agent concurrent KG editing.

3 The GRAPHWORLD environment

3.1 Theoretical framework

We model GRAPHWORLD as an environment whose state is a fully observable LPG together with its version history. At each time step, the agent proposes an atomic edit to the graph; the environment applies this edit deterministically, producing a new LPG and a new commit in the version history. Formally, GRAPHWORLD is defined by the tuple $\mathcal{M} = (S, A, \delta, R, O)$ where S is the space of states (G, H) with G a labeled property graph and H its commit history, A is the set of atomic LPG edits, δ is the deterministic transition function, R is an externally specified reward function, and $O = S$ since the agent can access the entire LPG structure together with the version history.

State and observation. At time t, the environment is in state $s_t = (G_t, H_t)$, where $G_t = (V, E, L, l_V, l_E, K, W, p_V, p_E)_t$ is an LPG as defined in Appendix A.1, and H_t is the sequence of commits leading to G_t. The observation o_t is identical to the state s_t for every t.

Actions. Each action $a_t \in A$ corresponds to a single mutation of the 9-tuple structure, for example adding nodes `add_node`$(v, l_V(v), p_V(v))$, edges `add_edge`$(e, l_E(e), p_E(e))$, updating properties `update_property`(x, k, w) and deleting nodes `delete_node`(v) and edge `delete_edge`(e). All actions are recorded in the commit history H_t, preserving an auditable trail of modifications.

Transition dynamics. The transition function is deterministic $s_{t+1} = \delta(s_t, a_t)$, where δ is the graph edit operator. A valid edit updates the LPG tuple G_t and appends a commit to H_t. Invalid edits (*e.g.*, deleting a non-existent node) produce a no-op or transition to an error state, depending on the configuration of the task.

Rewards. GRAPHWORLD is task-agnostic. The reward function $R(s_t, a_t, s_{t+1})$ is external to the environment and can be specified according to downstream objectives. For example, rewards may encourage edits that improve graph completeness, penalize violations of ontology constraints, or favor the addition of provenance information. This separation allows evaluation along two complementary axes: construction quality metrics (task-dependent) and process metrics (environment-dependent, *e.g.*, edit efficiency or branching performance).

Episodes. An episode begins with an initial state (G_0, H_0) and proceeds until a stopping condition is met, such as reaching a target benchmark graph, exhausting an edit budget, or receiving an explicit termination signal. Because the environment incorporates Git-style versioning, episodes may include branching and merging trajectories, enabling the study of collaborative and multi-agent editing strategies.

3.2 System overview

GRAPHWORLD is designed to provide agents with a practical and extensible environment for building and maintaining knowledge graphs. Its design follows three guiding principles: standarization, production readiness, and observability.

Standardization. GRAPHWORLD offers a uniform interface for graph editing, allowing agents to interact with knowledge graphs through a consistent set of tools.

Production readiness. By relying on proven libraries and widely used infrastructure, any agent developed for GRAPHWORLD can be moved into production with minimal overhead.

Observability. The framework records each agent action as an auditable step, enabling fine-grained evaluation of graph construction processes (*e.g.*, scoring agents based on the efficiency of their editing strategies).

At the system level, GRAPHWORLD consists of modular components that can be used independently or in combination. The two core components are a language-agnostic software development kit (SDK) for graph compression and manipulation, and a command-line interface (CLI) for versioning and collaboration.

3.2.1 SDK for KG compression with DIAMOND

A core component of GRAPHWORLD is DIAMOND, a compact, property-preserving graph storage format. DIAMOND provides high-performance graph compression and decompression routines, implemented in Rust for efficiency, with bindings for Python and Node.js via `PyO3` and `napi-rs`. This multi-language design enables developers to embed the same compression methods into agents written in different ecosystems, while maintaining consistency across them. Example programs demonstrating usage in each language are provided in Appendix A.5.

The SDK supports two primary capabilities. First, it enables compression and decompression of property graphs from common formats such as JSON Lines and serialized into the `.diamond` binary encoding described in Appendix A.2. Second, it provides in-memory graph representations via the `PropertyGraph` class, which allows developers to manipulate graphs directly and convert them across formats without loss of information.

3.2.2 CLI for KG versioning

While the SDK addresses scalability, effective multi-agent editing requires version control. To this end, GRAPHWORLD extends Git to support property graphs by defining custom drivers for filtering, diffing, and merging. This integration leverages Git's mature branching, archiving, and collaboration workflows while adapting them to the semantics of graph data.

The interaction between GRAPHWORLD and Git is coordinated through the `.gitattributes` file, which specifies how different file types should be handled during serialization, comparison, and merging. When a repository is configured for use with GRAPHWORLD, graph files matching user-defined patterns (*e.g.*, `.jsonl` or `.diamond`) are automatically associated with these drivers. This allows contributors to work with a familiar Git interface, while GRAPHWORLD manages the underlying representation of graphs.

Filter drivers handle the translation between formats stored in the repository and those exposed in the working tree. For example, a graph may be archived in the compressed DIAMOND format to save space and ensure consistency, while being presented in a human-readable format when checked out. This dual representation enables contributors to inspect or edit graphs locally without sacrificing the efficiency of binary storage.

Diff drivers define how Git compares graph files across commits. Since graphs are often stored in compressed binary form, traditional line-based diffs are meaningless. The custom diff driver decompresses and interprets the files, producing a semantic comparison in terms of nodes, edges, and properties. This allows reviewers to see meaningful changes, such as a node label update or an edge addition, rather than an opaque binary. Figure 1 provides an example diff between two graphs.

Merge drivers extend this functionality to conflict resolution. Git's default merge algorithm assumes text-based files, where order matters. For graphs, order is irrelevant, and conflicts should only arise when the same structural element is edited differently across branches. The custom merge driver reconciles graphs by structure, reducing spurious conflicts and enabling order-independent merging (Figure 2). This makes it possible for multiple agents or human collaborators to work concurrently without frequent manual intervention.

These drivers integrate with Git's existing workflows. Contributors can branch, commit, and merge as usual, while GRAPHWORLD ensures that the version control semantics respect the underlying graph structure.

```
~ Node (node_101)
    labels: ["person"]
  ~ country: ["United States", "Japan"]
  ~ name: ["Alice", "Carol", "Juan"]
  + nicknames: ["Jan"]

~ Edge (edge_101_103_directed)
    from: 101
    to: 103
    directed: true
    labels: ["likes"]
    since: ["2015"]
   engaged: [false, true]
```

Figure 1: A human-readable Git diff of an LPG used in GRAPHWORLD. Red elements indicate deletions, yellow elements indicate modifications, and green elements indicate additions.

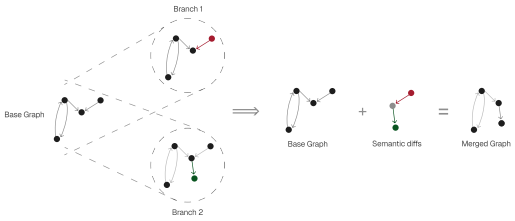

Figure 2: Three-way merge of an LPG in GRAPHWORLD. Starting from a base graph, two branches introduce independent edits. GRAPHWORLD computes semantic diffs and reconciles them through a three-way merge to produce the merged graph.

4 Applications of GRAPHWORLD environment

4.1 Multi-agent collaboration in GRAPHWORLD environment

The GRAPHWORLD environment is designed to model agent-agent and agent-human collaboration. It abstracts away the other participants in the collaboration, such that each contributor must make no assumptions about the features or functionalities of other collaborators. The agents that operate within GRAPHWORLD can vary widely in their goals, design, and implementation. We provide two examples of agents that could exist within the GRAPHWORLD environment.

Change-proposing agent. An agent might specialize in proposing improvements to a graph, both in the form of new nodes or edges, or in an edit proposal for an existing entity within the graph. There are many ways in which these suggestions can be made. The agent could train a graph neural network (GNN) on the graph with a link prediction objective, predict the edges that are more likely to exist, and pick the top $k \in \mathbb{N}$ as new edges to add. Alternatively, it could read unstructured documents to identify known graph entities and look for relationships referenced in the text that are not present in the current version of the graph. It could also take as input a particular node of interest in a sparse region of the graph and look for information online that suggests the existence of currently missing edges. Regardless of how these proposals arise, the agent can create a new branch, perform the changes, and open a pull request (PR) to merge the new branch into the `main` one. Figure 3a depicts the structure of such an agent.

Evidence-gathering agent. An evidence-gathering agent might review open PRs in the repository that contain edge proposals and look for information on the Internet, in the scientific literature, in proprietary databases, or perform independent tests to support or oppose the creation of the new edges (Figure 4a). This agent plays the same role as peer review in the academic world, but it is automated and scalable. Scientific research agents show strong performance in synthesizing existing scientific knowledge from multiple sources [47, 48] and, therefore, may be adept at this role; however, they do not produce consistent, structured outputs unless appropriately prompted; graphs provide a universal schema to record their outputs in the context of existing scientific knowledge.

4.2 Human-AI co-creation of KGs in GRAPHWORLD

Beyond the design of individual agents, GRAPHWORLD supports complex workflows that integrate human input with automated reasoning. These workflows combine the strengths of human domain expertise with the scalability of agents. We describe three representative examples.

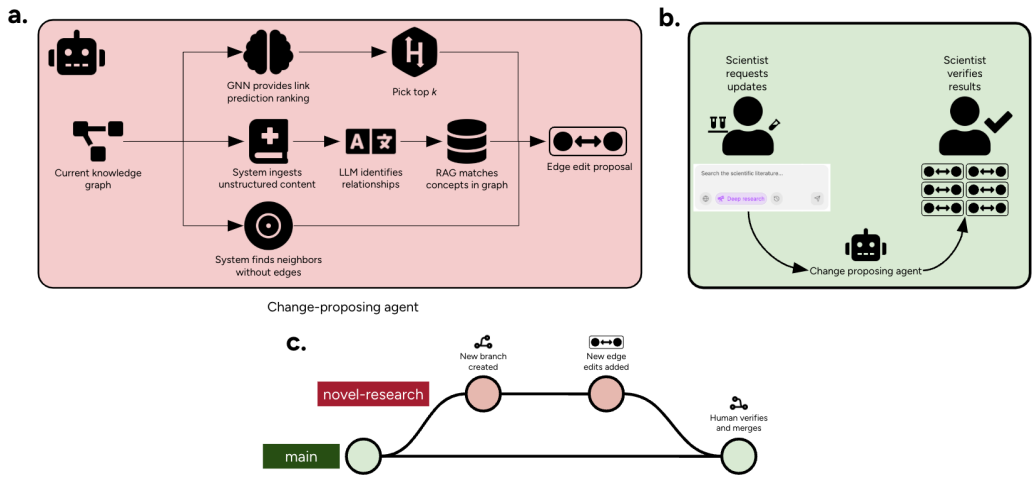

Figure 3: Overview of a change-proposing agent. **(a)** The agent has access to a toolbox of methods, including GNN-based link prediction on the current KG, relation extraction from unstructured text, and RAG over the literature to score candidates and then synthesize the top-k into edge-edit proposals. **(b)** An expert scientist states an intent in natural language. The agent materializes it as concrete graph edits on an isolated branch, opens a PR, and returns a reviewable checklist the scientist can accept, modify, or reject. **(c)** The new agent-created branch does not interfere with the main KG and provides an auditable, reproducible history of KG evolution.

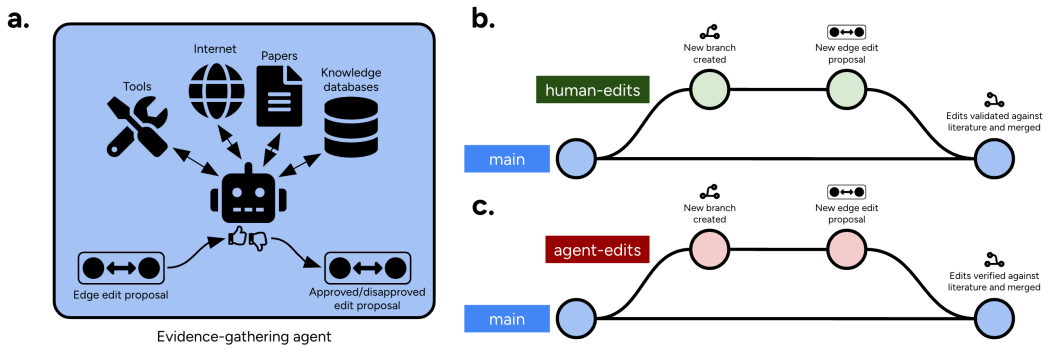

Figure 4: Overview of an evidence-gathering agent. **(a)** The agent consults multiple external sources, including online tools, scientific papers, and structured knowledge databases, to validate edit proposals. Each candidate edit is either approved or disapproved based on supporting evidence. **(b)** Human-generated edits proceed through the same process; they are insolated in a new branch, validated agains the literature, and then merged into the mainline. **(c)** Similarly, agent-generated edits, such as those from the change-proposing agent in Figure 3, are proposed on a separate branch and merged once validated.

Automated validation of human edits. Human maintainers frequently seek to extend or refine a KG. However, detecting semantic inconsistencies or unsupported claims is challenging without computational assistance. In GRAPHWORLD, a change-proposing agent can continuously monitor open pull requests (PRs) submitted by human editors. The agent verifies proposed edits against the available literature or other structured resources and can take several actions: independently approving and merging the PR, providing detailed feedback, or suggesting alternative edits (Figure 4b). This workflow establishes a cycle in which human edits are systematically verified by machine reasoning. The result is improved accuracy of human-generated graphs and reduced reliance on costly manual validation. For graphs deployed in safety-critical applications, such as healthcare or logistics [49–51], the verification workload can be dynamically scaled according to the estimated importance of the affected nodes and edges.

Natural language editing for human domain experts. Domain experts often wish to contribute to KG construction but may lack the technical expertise to interface directly with graph databases or APIs. In GRAPHWORLD, agents serve as interpreters and actuators that translate high-level human intentions into executable graph edits. A user specifies a desired change in natural language. The agent supplements this instruction with information retrieved from structured or unstructured data sources and proposes a set of candidate edits. These edits are applied in an isolated branch, committed, and returned to the user for inspection (Figure 3b, Figure 3c). This workflow allows domain experts to contribute without using graph query languages. As agent capabilities in GRAPHWORLD expand, these systems can move beyond execution of text instructions toward co-pilots that engage in multi-turn discussions with experts about the validity and scope of proposed knowledge changes [52].

Ontology alignment and entity deduplication. Large-scale KG construction often introduces redundant nodes when integrating diverse sources or ontologies [11, 53]. These duplications can fragment the graph: metadata and updates may apply to one duplicate node but not to others, and the topology can become misleading if neighbors are split across copies [54]. In GRAPHWORLD, an agent can be configured to inspect graph branches (such as `main`) and detect such redundancies. Detection leverages semantic signals, including lexical similarity of node labels, synonym resolution, and consistency of attached properties [55]. Upon identifying a likely duplication, the agent can either (i) perform a merge of the redundant nodes, preserving their combined metadata and edges, or (ii) raise an alert to human maintainers for confirmation. This workflow ensures that entity resolution is continuous and systematic, improving graph integrity as the KG evolves. Multiple agents can monitor graph development in parallel [56, 57], providing early detection of conflicts and contributing to a high-quality and reliable final knowledge base.

4.3 Limitations and future work

Inneficient memory use of library bindings. Internally, the Rust core stores the decompressed graphs in memory as Polars tables. This is extremely memory efficient, but it is difficult to transfer across the bridge to the host binding language. Currently, the graph is converted into a simpler PG-JSON-like representation before it is sent, but switching to this representation forces a copy of memory, uses a space-inefficient layout, and prevents vectorized operations. To address this, we will expose table-level APIs in all bindings and develop zero-copy exchange PyPolars and NodePolars.

Unnecessary deserialization cost. Every file read is currently canonicalized into an in-memory PG-JSON representation before compression, duplicating the memory requirements to read the graph and introducing an unnecessary extra pass. We will add streaming front-ends for every supported format that will perform early type inference to amortize the ingestion overhead and will reduce the need to maintain the entire graph at once in a memory-inefficient representation.

Property type constraints. DIAMOND, following the PG-JSON standard, currently assumes flat optional properties whose values are homogeneous lists of primitive types (Boolean, String, Number). Mixed-type lists, nested objects, and unions are not supported. We will extend schema inference to nested types and perform an intermediate flattening step that enables support for nested types while preserving compact encodings and round-trip fidelity.

These limitations largely reflect the unoptimized nature of the first version of our compression library. We aim to reduce memory consumption through a mix of streaming, pipelining, and the elimination of unnecessary steps, all while improving developer experience.

5 Conclusion

We presented GRAPHWORLD, an environment that makes the construction of KGs observable, collaborative, and reproducible. GRAPHWORLD combines a language-agnostic manipulation library for labeled property graphs, semantic graph diffs and merges built on top of Git, and DIAMOND, a compact, property-preserving columnar graph encoding that scales to multi-million-edge KGs. Together, these components let agents and humans propose, review, and merge edits with auditable histories. We hope GRAPHWORLD will serve as the interoperability layer upon which multi-agent KG systems are built, compared, and improved.

References

1. Ehrlinger, L. & Wöß, W. Towards a definition of knowledge graphs. *SEMANTiCS (Posters, Demos, SuCCESS)* **48,** 2 (2016).
2. Hogan, A. *et al.* Knowledge Graphs. *ACM Comput. Surv.* **54,** 71:1–71:37. doi:10.1145/3447772 (2021).
3. Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J. & Vrgoč, D. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* **50,** 68:1–68:40. doi:10.1145/3104031 (2017).
4. Wu, S. *et al. STaRK: Benchmarking LLM Retrieval on Textual and Relational Knowledge Bases* 2024. doi:10.48550/arXiv.2404.13207.
5. Pan, S., Luo, L., Wang, Y., Chen, C., Wang, J. & Wu, X. Unifying Large Language Models and Knowledge Graphs: A Roadmap. *IEEE Transactions on Knowledge and Data Engineering* **36,** 3580–3599. doi:10.1109/TKDE.2024.3352100 (2024).
6. Park, J. S., O'Brien, J., Cai, C. J., Morris, M. R., Liang, P. & Bernstein, M. S. *Generative Agents: Interactive Simulacra of Human Behavior* in *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology* (Association for Computing Machinery, New York, NY, USA, 2023), 1–22. doi:10.1145/3586183.3606763.
7. Gao, A. K. *Introducing Tuna - A Tool for Rapidly Generating Synthetic Fine-Tuning Datasets* 2023.
8. Chen, D. *et al. Data-Juicer: A One-Stop Data Processing System for Large Language Models* 2023. doi:10.48550/arXiv.2309.02033.
9. Chen, D. *et al. Data-Juicer 2.0: Cloud-Scale Adaptive Data Processing for and with Foundation Models* 2025. doi:10.48550/arXiv.2501.14755.
10. Rasmussen, P., Paliychuk, P., Beauvais, T., Ryan, J. & Chalef, D. *Zep: A Temporal Knowledge Graph Architecture for Agent Memory* 2025. doi:10.48550/arXiv.2501.13956.
11. Chandak, P., Huang, K. & Zitnik, M. Building a knowledge graph to enable precision medicine. *Scientific Data* **10,** 67. doi:10.1038/s41597-023-01960-3 (2023).
12. Mo, B. *et al. KGGen: Extracting Knowledge Graphs from Plain Text with Language Models* 2025. doi:10.48550/arXiv.2502.09956.
13. Zhao, X. *et al. AGENTiGraph: An Interactive Knowledge Graph Platform for LLM-based Chatbots Utilizing Private Data* 2024. doi:10.48550/arXiv.2410.11531.
14. Larson, J. & Truitt, S. *GraphRAG: A new approach for discovery using complex information* 2024.
15. Tzitzikas, Y., Theoharis, Y. & Andreou, D. *On Storage Policies for Semantic Web Repositories That Support Versioning* in *The Semantic Web: Research and Applications* (eds Bechhofer, S., Hauswirth, M., Hoffmann, J. & Koubarakis, M.) (Springer, Berlin, Heidelberg, 2008), 705–719. doi:10.1007/978-3-540-68234-9_51.
16. Fernández, J. D., Polleres, A. & Umbrich, J. Towards Efficient Archiving of Dynamic Linked Open Data. *DIACRON@ ESWC* **1377,** 34–49 (2015).
17. Arndt, N., Naumann, P., Radtke, N., Martin, M. & Marx, E. Decentralized Collaborative Knowledge Management using Git. *Journal of Web Semantics* **54,** 29–47. doi:10.1016/j.websem.2018.08.002 (2019).
18. Miller, J. J. *Graph database applications and concepts with Neo4j* in *Proceedings of the southern association for information systems conference, Atlanta, GA, USA* **2324** (2013), 141–147.
19. Edge, D. *et al.* From local to global: A graph rag approach to query-focused summarization. *arXiv:2404.16130* (2024).
20. Mavromatis, C. & Karypis, G. GNN-RAG: Graph neural retrieval for large language model reasoning. *arXiv:2405.20139* (2024).
21. Zhu, X., Xie, Y., Liu, Y., Li, Y. & Hu, W. Knowledge graph-guided retrieval augmented generation. *NAACL* (2025).
22. Huang, Y., Zhang, S. & Xiao, X. *KET-RAG: A cost-efficient multi-granular indexing framework for graph-rag* in *KDD* (2025), 1003–1012.
23. Sanmartin, D. KG-RAG: Bridging the gap between knowledge and creativity. *arXiv:2405.12035* (2024).

24. Wood, D., Lanthaler, M. & Cyganiak, R. RDF 1.1 concepts and abstract syntax. *W3C Recommendation, W3C* (2014).
25. Zaho, Z., Han, S. K. & Kim, J. R. LPG Representation of the Reification of RDF. *International Journal of Engineering and Technology* **7,** 562–566. doi:10.14419/ijet.v7i3.34.19382 (2018).
26. Besta, M. *et al.* Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries. *ACM Comput. Surv.* **56,** 31:1–31:40. doi:10.1145/3604932 (2023).
27. Chiba, H., Yamanaka, R. & Matsumoto, S. *Property Graph Exchange Format* 2019. doi:10.48550/arXiv.1907.03936. arXiv: 1907.03936 [cs].
28. Besta, M. & Hoefler, T. *Survey and Taxonomy of Lossless Graph Compression and Space-Efficient Graph Representations* 2019. doi:10.48550/arXiv.1806.01799.
29. Navarro, G. Compressing web graphs like texts. *Dept. Comput. Sci., Univ. Chile, Santiago, Chile, Tech. Rep. TR/DCC-2007-2* (2007).
30. Claude, F. & Navarro, G. Fast and Compact Web Graph Representations. *ACM Trans. Web* **4,** 16:1–16:31. doi:10.1145/1841909.1841913 (2010).
31. Boldi, P. & Vigna, S. *The webgraph framework I: compression techniques* in *Proceedings of the 13th international conference on World Wide Web* (Association for Computing Machinery, New York, NY, USA, 2004), 595–602. doi:10.1145/988672.988752.
32. Brisaboa, N. R., Ladra, S. & Navarro, G. *k2-Trees for Compact Web Graph Representation* in *String Processing and Information Retrieval* (eds Karlgren, J., Tarhio, J. & Hyyrö, H.) (Springer, Berlin, Heidelberg, 2009), 18–30. doi:10.1007/978-3-642-03784-9_3.
33. Claude, F. & Ladra, S. *Practical representations for web and social graphs* in *Proceedings of the 20th ACM international conference on Information and knowledge management* (Association for Computing Machinery, New York, NY, USA, 2011), 1185–1190. doi:10.1145/2063576.2063747.
34. Brisaboa, N. R., Ladra, S. & Navarro, G. Compact representation of Web graphs with extended functionality. *Information Systems* **39,** 152–174. doi:10.1016/j.is.2013.08.003 (2014).
35. Feng, F. *et al.* GenomicKB: a knowledge graph for the human genome. *Nucleic Acids Research* **51,** D950–D956. doi:10.1093/nar/gkac957 (2023).
36. Frommhold, M., Piris, R. N., Arndt, N., Tramp, S., Petersen, N. & Martin, M. *Towards Versioning of Arbitrary RDF Data* in *Proceedings of the 12th International Conference on Semantic Systems* (Association for Computing Machinery, New York, NY, USA, 2016), 33–40. doi:10.1145/2993318.2993327.
37. Meinhardt, P., Knuth, M. & Sack, H. *TailR: a platform for preserving history on the web of data* in *Proceedings of the 11th International Conference on Semantic Systems* (Association for Computing Machinery, New York, NY, USA, 2015), 57–64. doi:10.1145/2814864.2814875.
38. Cassidy, S. & Ballantine, J. Version Control for RDF Triple Stores. *ICSOFT (ISDM/EHST/DC)* **7,** 5–12 (2007).
39. Graube, M., Hensel, S. & Urbas, L. *R43ples: Revisions for triples* in *Proceedings of the 1st Workshop on Linked Data Quality co-located with 10th International Conference on Semantic Systems (SEMANTiCS 2014)* (2014).
40. Vander Sande, M., Colpaert, P., Verborgh, R., Coppens, S., Mannens, E. & Van de Walle, R. R&Wbase: Git for triples. *LDOW* **996** (2013).
41. Gil, J. P., Coquery, E., Samuel, J. & Gesquiere, G. *ConVer-G: Concurrent versioning of knowledge graphs* 2024. doi:10.48550/arXiv.2409.04499.
42. Theodorakis, G., Clarkson, J. & Webber, J. *Aion: Efficient Temporal Graph Data Management* in (Paestum, Italy, 2024). doi:10.48786/EDBT.2024.43.
43. Halilaj, L., Grangel-González, I., Coskun, G. & Auer, S. *Git4Voc: Git-based Versioning for Collaborative Vocabulary Development* 2016. doi:10.48550/arXiv.1601.02433.
44. *RDF 1.1 N-Quads* 2014.
45. *RDF Binary using Apache Thrift* 2025.
46. Fernandez, J. D., Martínez-Prieto, M. A., Gutiérrez, C., Polleres, A. & Arias, M. *Binary RDF Representation for Publication and Exchange (HDT)* SSRN Scholarly Paper. Rochester, NY, 2013. doi:10.2139/ssrn.3198999.

47. Skarlinski, M. D. *et al. Language agents achieve superhuman synthesis of scientific knowledge* 2024. doi:10.48550/arXiv.2409.13740.
48. Ai2. *Introducing Ai2 Paper Finder* 2025.
49. Alber, D. A. *et al.* Medical large language models are vulnerable to data-poisoning attacks. *Nature Medicine* **31,** 618–626 (2025).
50. Alsentzer, E. *et al.* Few shot learning for phenotype-driven diagnosis of patients with rare genetic diseases. *npj Digital Medicine* **8,** 380 (2025).
51. Yang, J. *et al.* Poisoning medical knowledge using large language models. *Nature Machine Intelligence* **6,** 1156–1168 (2024).
52. Gao, S. *et al. Empowering Biomedical Discovery with AI Agents* 2024. doi:10.48550/arXiv.2404.02831.
53. Lobentanzer, S. *et al.* Democratizing Knowledge Representation with BioCypher. *Nature Biotechnology* **41,** 1056–1059. doi:10.1038/s41587-023-01848-y (2023).
54. Callahan, T. J. *et al.* An open source knowledge graph ecosystem for the life sciences. *Scientific Data* **11,** 363 (2024).
55. Johnson, R. *et al.* ClinVec: Unified Embeddings of Clinical Codes Enable Knowledge-Grounded AI in Medicine. *medRxiv,* 2024–12 (2024).
56. Lu, Y. & Wang, J. KARMA: Leveraging Multi-Agent LLMs for Automated Knowledge Graph Enrichment. *arXiv:2502.06472* (2025).
57. Liu, B., Zhang, J., Lin, F., Yang, C., Peng, M. & Yin, W. *SymAgent: A neural-symbolic self-learning agent framework for complex reasoning over knowledge graphs* in *Proceedings of the ACM on Web Conference* (2025), 98–108.
58. Chiba, H. & Voß, J. Property Graph Exchange Format (PG). doi:10.5281/zenodo.13859531 (2024).
59. *pola-rs/polars* 2025.
60. *apache/arrow-rs* 2025.
61. Vohra, D. in *Practical Hadoop Ecosystem: A Definitive Guide to Hadoop-Related Frameworks and Tools* (ed Vohra, D.) 325–335 (Apress, Berkeley, CA, 2016). doi:10.1007/978-1-4842-2199-0_8.
62. *apache/parquet-java* 2025.
63. Collet, Y. & Kucherawy, M. *Zstandard Compression and the application/zstd Media Type* Request for Comments RFC 8478 (Internet Engineering Task Force, 2018). doi:10.17487/RFC8478.
64. Chandak, P., Huang, K. & Zitnik, M. Building a Knowledge Graph to Enable Precision Medicine. *Scientific Data* **10,** 67. doi:10.1038/s41597-023-01960-3 (2023).

A Technical Appendices and Supplementary Material

A.1 Formal definition of labeled property graphs

Formally, LPGs can be represented as a 9-tuple $G = (V, E, L, l_V, l_E, K, W, p_V, p_E)$ [26], where:

- V is the set of nodes,
- E is the set of edges,
- L is the set of labels,
- $l_V : V \to \mathcal{P}(L)$ is the function that maps nodes to their labels,
- $l_E : E \to \mathcal{P}(L)$ is the function that maps edges to their labels,
- K is the set of all possible property keys,
- W is the set of all possible property values,
- $p_V : V \to K \times W$ is the function that maps nodes to their property key-value pairs, and
- $p_E : E \to K \times W$ is the function that maps edges to their property key-value pairs.

Note that under this formulation l_V and l_E map to the power set of L, and not to L itself. Therefore, every node and edge can have one label, more than one label, or no labels associated.

A.2 Graph compression algorithm

We designed binary encoding and compression techniques to efficiently store and represent labeled property (LP) graphs. This enabled us to encode graph structure and metadata in a single, reduced-size file, and simplified storage in teh git server. Initially, we attempted to identify and use a pre-existing binary encoding format for LPG; however, as described in Section 2.3, given the recency of the LPG format, we could not identify any. Therefore, we developed our own binary encoding format for compressing graphs, which we refer to as DIAMOND. The compression steps are as follows (Figure 5).

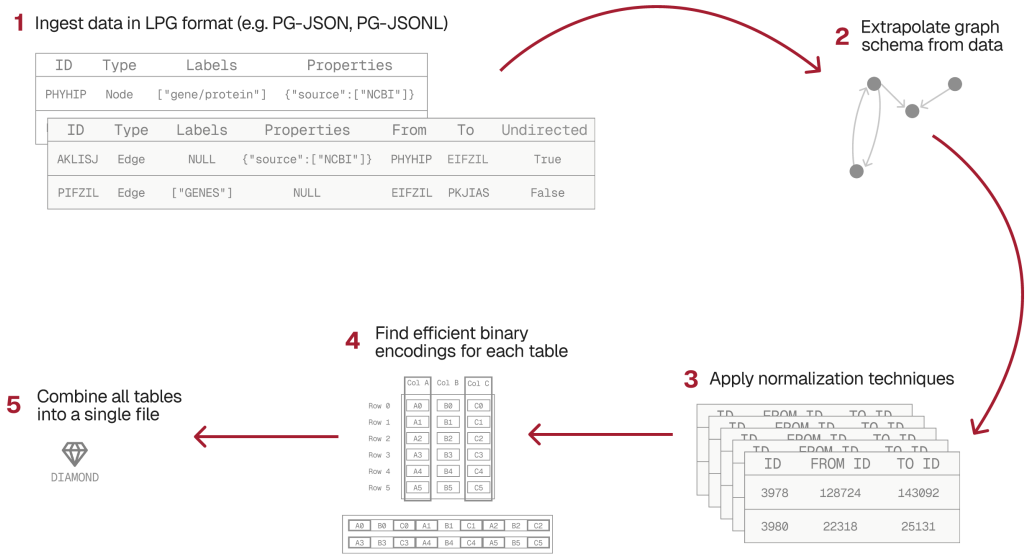

Figure 5: The DIAMOND graph compression algorithm. Each step of the compression algorithm is depicted.

Ingestion. First, we ingest the data in a standard format used to represent LPG graphs, for example, in Graphviz DOT, JSON lines, or PG file formats. If the input format is not PG-JSON, the original file is first converted to PG-JSON. This avoids having to define a separate compression routine for every standard.

Schema extrapolation. Next, we extrapolate a strongly typed schema for the graph by sifting through all the node and edge records to understand their structure. The first step in the schema extrapolation is to split all the nodes and edges into groups. Groups are understood to potentially have distinct properties. Separation into groups allows us to find an efficient representation for each group that avoids having many `null`s in our final binary file.

We create these groups by separating graph elements by label combinations, with all nodes with a specific order-agnostic label combination grouped together. The same is done for edges. In practice, we achieve order-agnostic grouping by lexicographically sorting the labels, but the compression mechanism is method-agnostic. Each group within each element type is assigned an identifier in the form of a natural number that will be used to reference the group in other parts of the algorithm. The full pseudocode for GROUP_BY_LABELS can be found in the appendix under Algorithm 1.

Once all elements have been partitioned into their respective groups, the types of their properties are inferred (see Algorithm 2). Following the PG-JSON standard [58], we assume that all properties are optional. That is, a node with labels $\{\text{animal}, \text{dog}\}$ might have property `nicknames: ["puppy", "pup"]` declared but another node with the same labels might not. According to the PG-JSON standard [58], all properties must have as their value a list of type `Boolean`, `String` or `Number` (floating point). For each element in a group we iterate collecting all the properties present and storing their corresponding types. There are two potential sources of conflict in this scenario.

1. **Mismatched type within a property value list.** The value for a property list in the PG-JSON format can hold elements of different primitive types (*e.g.*, `["puppy", 1]`). This is an unsupported feature in DIAMOND, as much of the space gains come from leveraging the assumption that properties will always have the same type.
2. **Mismatched type for a property across elements.** Two elements belonging to the same group could have lists composed of different values. Once again, DIAMOND relies on all instances of a property within a group being of the same type, so an error is found if a mismatch occurs.

If no conflicts are found the result after this step is finished is a mapping from a group identifier to a map from property name to value list inner type. The full implementation of the type inference for a specific group can be found in the appendix under Algorithm 3.

Table creation. We create two tables, one for nodes and one edges, that contain metadata about the group. They each have two columns, `id` and `type`. The `type` column contains in each cell a value of type `String[]` that represents a unique label combination and the `id` column contains the numerical identifier assigned to that combination in the group partitioning step. For the detailed algorithm see Algorithm 4.

Following the creation of the group metadata tables, we proceed to create one table per element group to hold the information about its elements. For node groups each table consists of an `id` column that holds the node identifier and one column per possible property for that node group (see Algorithm 5). For edge groups each table consists of columns `id`, `from`, `to`, `undirected` plus one column per possible property (see Algorithm 6).

Transform tables to efficient encoding. We encode all the group tables efficiently in memory by using Polars [59] data frames. Polars data frames are tables consisting of multiple Apache Arrow columns [60]. Arrow is a software framework for dealing with columnar data. It provides efficient in-memory storage for various data types as well as utility functions for operating on that data. For each node and edge group we iterate over their elements and convert each their static (*e.g.*, `id`, `from`) and dynamic (*e.g.*, `nickname`) properties to Arrow series (see Algorithm 7) by finding the appropriate Arrow data type that efficiently represents the underlying values. After all columns for a group are created we group them in a Polars data frame, attaching the appropriate column name to the data frame header.

Serializing the graph. We obtain a binary file by saving all the data frames created so far to disk using the Apache Parquet format [61, 62] and grouping the files together using `tar`. We save each data frame as a column-oriented Parquet file, compressing every column with Zstandard [63] at level 3 – an intermediate setting that offers a favorable trade-off between compression ratio and processing speed – and partition the output into row groups of 1024×1024 rows (roughly one million rows) so

that the files remain highly compressible while still allowing efficient parallel reads and selective access to individual subsets of the data. Finally, all Parquet tables are combined together into a single on-disk file, which is further compressed to minimize the on-disk file size. The final `tar`-compressed file is saved with a `.diamond` extension.

By applying these steps – data ingestion, schema inference, normalization, binary encoding, and bundling – the DIAMOND algorithm losslessly shrinks the size of the original graph data.

A.2.1 Algorithms

Algorithm 1 GROUP_BY_LABELS

Input: `elements` ▷ a list of items, each with a `labels()` method
Output: `groups` ▷ a map from a set of labels to `(type_id, list of items)`

`groups` ← empty map
`next_type_id` ← 0
for all `item` **in** `elements` **do**
    `labels` ← `item.labels()`
    if `labels` is not empty **then**
        `key` ← `sort(labels)` ▷ sort alphabetically for a stable key
        if `key` **not in** `groups` **then**
            `groups[key]` ← (`next_type_id`, empty list)
            `next_type_id` ← `next_type_id` +1
        end if
        append `item` to the list inside `groups[key]`
    end if
end for
return `groups`

Algorithm 2 INFER_PROPERTY_TYPES_FOR_GROUPS

Input: `groups` ▷ a map from a set of labels to `(type_id, list of items)`
Output: `prop_types_map` ▷ a map from the same label set to `(property name → data type)`

`prop_types_map` ← empty map
for all `labels` **in** `groups` **do**
    `items` ← `groups[labels].items`
    `properties_list` ← list of `item.properties()` for every `item` in `items`
    `status, inferred` ← INFERTYPES(`properties_list`)
    if `status` is `Success` **then**
        `prop_types_map[labels]` ← `inferred`
    else if `status` is `MismatchedTypesWithinItem` **then**
        raise `TypeInferenceFailed(labels, status.property)`
    else ▷ mismatched types across items
        raise `MismatchedListInnerType(labels, status.property, status.expected)`
    end if
end for
return `prop_types_map`

Algorithm 3 INFER_TYPES_FOR_PROPERTIES_VEC

Input: `properties_vec` ▷ list of property dictionaries
Output: `types_map` ▷ map (`property name` → `data type`)

```
types_map ← empty map
for all properties in properties_vec do
    for all (key, value_list) in properties do
        status, data_type ← INFERTYPE(value_list)
                                ▷ Iterates over list checking all values are of the same type
        if status is ErrorWithinList then
            raise MismatchedTypesWithinItem(key, status.data_types)
        end if
        if key in types_map then
            prev_type ← types_map[key]
            if prev_type ≠ data_type then
                raise MismatchedTypesAcrossItems(key, prev_type, data_type)
            end if
        end if
        types_map[key] ← data_type
    end for
end for
return types_map
```

Algorithm 4 BUILD_TYPES_DATA_FRAME

Input: `type_map` ▷ map from labels to `(type_id, list of items)`
Output: `df` ▷ table with two columns: `id` (integer) and `type` (list of labels)

```
type_ids ← empty list
type_labels ← empty list
for all (labels, (tid, _)) in type_map do
    append tid to type_ids
    append labels to type_labels
end for
col_id ← CREATECOLUMN("id", type_ids)
col_type ← CREATELISTCOLUMN("type", type_labels)
df ← CREATEDATAFRAME(col_id, col_type)
return df
```

Algorithm 5 BUILD_DATA_FRAME_FOR_NODE_GROUP

Input: `nodes` ▷ list of node objects
Input: `property_types` ▷ map (`property name` → `data type`)
Output: `df` ▷ table with an `id` column plus one column per property

```
columns ← empty list
append COLUMN("id", list of node.id for each node in nodes) to columns
prop_cols ← GETPROPERTYCOLUMNS(nodes, property_types)
append every element of prop_cols to columns
df ← CREATEDATAFRAME(columns)
return df
```

Algorithm 6 BUILD_DATA_FRAME_FOR_EDGE_GROUP

Input: `edges` ▷ list of edge objects
Input: `property_types` ▷ map (`property name` → `data type`)
Output: `df` ▷ table with columns `id`, `from`, `to`, `undirected`, plus one column per property

```
columns ← empty list
append COLUMN("id", list of edge.id for each edge in edges) to columns
append COLUMN("from", list of edge.from for each edge in edges) to columns
append COLUMN("to", list of edge.to for each edge in edges) to columns
append COLUMN("undirected", list of edge.undirected for each edge in edges) to columns
prop_cols ← GETPROPERTYCOLUMNS(edges, property_types)
append every element of prop_cols to columns
df ← CREATEDATAFRAME(columns)
return df
```

Algorithm 7 GET_DATA_FRAME_PROPERTY_COLUMNS

Input: `elements` ▷ list of items, each with a `properties()` map
Input: `property_types` ▷ map (`property name` → `data type`)
Output: `columns` ▷ list of table columns, one per property

```
columns ← empty list
for all (prop_name, dtype) in property_types do
    builder ← CREATELISTBUILDER(dtype)
    for all item in elements do
        values ← item.properties()[prop_name]                ▷ may be missing
        if values exists then
            series ← SERIESFROMVALUES(values, dtype)
            builder.add(series)
        else
            builder.add_null()
        end if
    end for
    col ← COLUMN(prop_name, builder.finish())
    append col to columns
end for
return columns
```

A.3 Benchmarking DIAMOND

A.3.1 Benchmarking on diverse synthetic graphs

To evaluate the performance of the DIAMOND library under various graph characteristics and scales, we conducted a benchmarking study using synthetically generated graphs. This approach allowed us to control specific graph properties and observe their impact on compression efficiency and memory consumption.

First, we designed a synthetic graph generator parameterized by three variables: $\mu \in \mathbb{R}$, $\sigma \in \mathbb{R}$, and $p \in [0, 1]$. The parameters μ and σ governed the number of properties associated with a group of nodes or edges, which followed a Normal distribution with mean μ and standard deviation σ. The parameter p represented the probability that a given node or edge would have a non-null value for a specific property. By adjusting μ, σ, and p, we could simulate graphs with varying levels of property density.

Using this generator, we created LPGBENCH, a dataset of diverse graphs with varying numbers of nodes, edges, properties, and labels per element. The configurations in LPGBENCH included:

- **Micro:** 10 nodes, 100 edges, maximum 1 label per element, $\mu = 2.0$, $\sigma = 1.0$, $p = 0.3$, maximum 1 value per selected property.

- **Small:** 1,000 nodes, 10,000 edges, maximum 1 label per element, $\mu = 3.0$, $\sigma = 1.0$, $p = 0.5$, maximum 2 values per selected property.
- **Medium:** 100,000 nodes, 1,000,000 edges, maximum 1 label per element, $\mu = 4.0$, $\sigma = 1.0$, $p = 0.7$, maximum 2 values per selected property.
- **Large:** 1,000,000 nodes, 10,000,000 edges, maximum 2 labels per element, $\mu = 5.0$, $\sigma = 1.0$, $p = 0.9$, maximum 3 values per selected property.

We evaluated DIAMOND on LPGBENCH to understand the performance characteristics of the DIAMOND library and compare the `.diamond` format against other popular graph representations. Specifically, our benchmarking analyses were as follows.

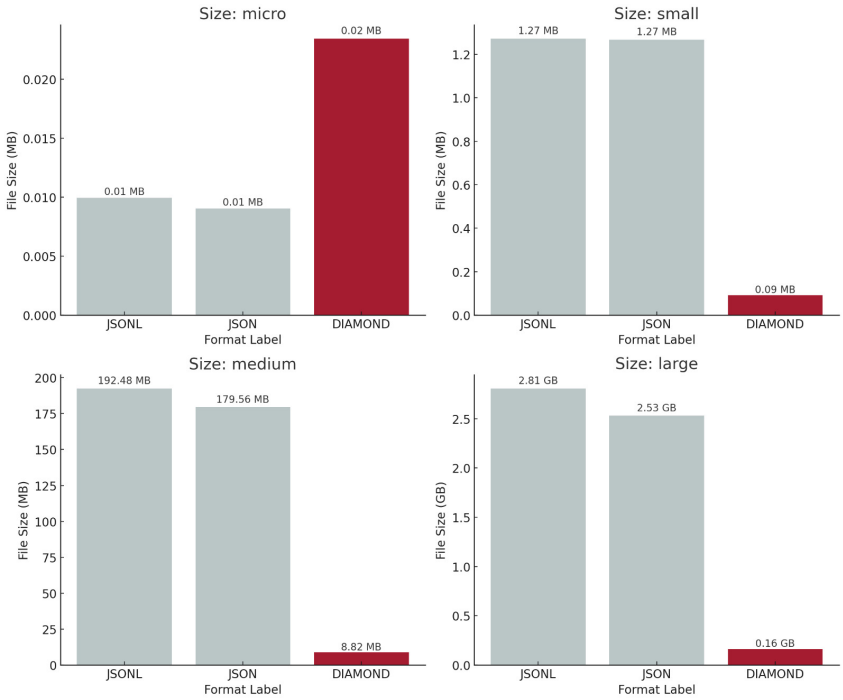

Figure 6: File size vs. graph format across graph sizes. The `.diamond` file size is shown in red, as compared to JSON and JSON Lines.

File size versus graph format (Figure 6). This analysis aimed to establish a baseline comparison of the on-disk storage requirements for different graph formats across representative graph sizes (micro, small, medium, and large). By fixing the graph structure and size, we directly compared the inherent storage overhead and compression effectiveness of each format without the influence of scaling property densities. As shown in Figure 6, at the smallest graph sizes with only 10 nodes and 100 edges, DIAMOND was outperformed by JSON and JSON lines. However, at even small graph sizes with 1,000 nodes and 10,000 edges, the `.diamond` file is significantly smaller than its JSON and JSON lines counterparts. Once the graph size scales to 10 million edges, the DIAMOND-compressed file is only 5.69% the size of JSON Lines and 6.32% the size of JSON.

File size across graph sizes and property densities (Figure 7). This analysis investigated how the file size of each graph format scales as the total number of graph elements increases, while maintaining consistent property distributions defined by μ, σ and p. We sought to understand the scalability of each format and evaluate how efficiently they handle increasing graph size under different scenarios of property density and sparsity. As depicted in Figure 7, the compression ratio achieved by DIAMOND, relative to JSON or JSON Lines, improves as properties become more dense (*e.g.*, comparing results for $p = 0.3$ versus $p = 0.9$, note that the y-axis is shared across the panels). This observation aligns with our hypothesis that DIAMOND achieves greater compression efficiency on graphs with higher property density.

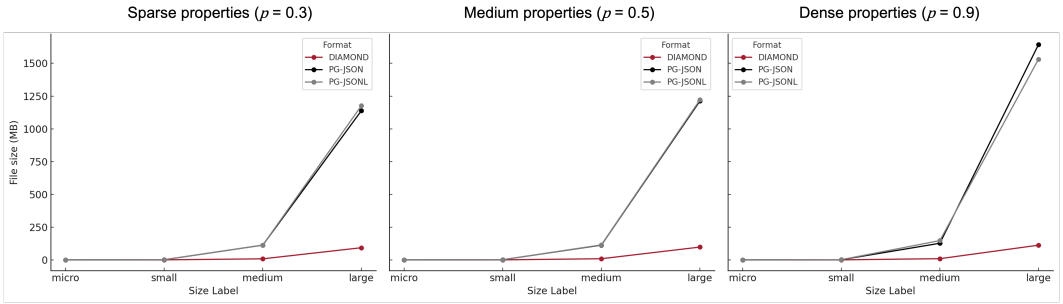

Figure 7: File size across graph sizes and property densities. The `.diamond` file size is shown in red, as compared to JSON and JSON Lines.

A.3.2 Benchmarking on real-world KGs

Beyond our synthetic benchmarks, we also sought to test the performance of DIAMOND on a real-world KG. When applied on PrimeKG [64], a popular biomedical KG with over 55,000 downloads on Harvard Dataverse at the time of writing, DIAMOND achieves up to $34.1\times$ compression as compared to other prevalent LPG graph formats, including CSV header, PG, YARS-PG, DOT, Cypher, and JSONL (Figure 8). It consumes only 8.9% the size of the next smallest format and uses 2.9% of the space required by the JSONL representation of the KG. Therefore, we successfully designed a compressed format for LPG graphs that outperforms state-of-the-art graph representations.

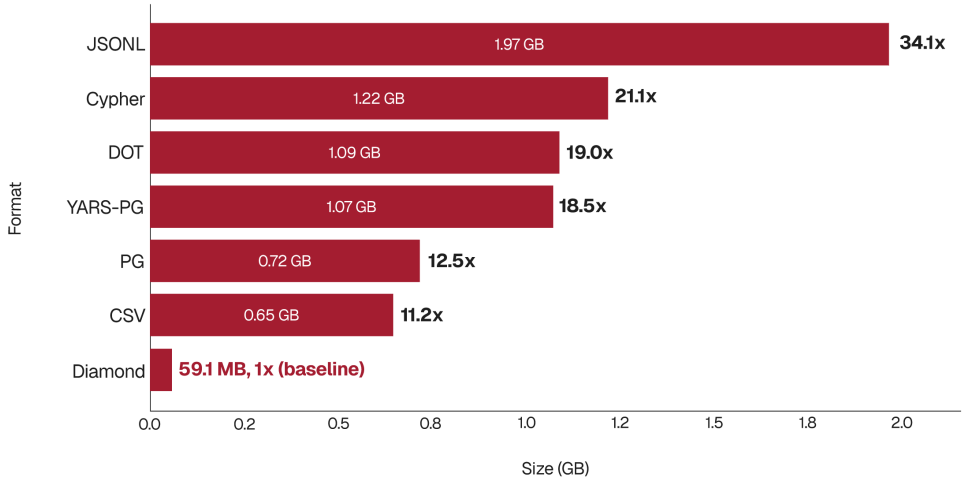

Figure 8: DIAMOND performance on a real-world KG. As compared to other popular graph-encoding formats, DIAMOND achieves up to a $34.1\times$ compression ratio when used to compress PrimeKG [11].

A.4 Production-readyness of DIAMOND

The DIAMOND library employs a continuous integration and continuous deployment (CI/CD) pipeline based on GitHub actions to automate testing and streamline code release. CI/CD generates releases based on the code modified in a pull request. The CI/CD pipeline is as follows:

Python continuous integration. If Python binding code is altered, a workflow is triggered that performs linting, uses Bandit for security analysis, uses `interrogate` to evaluate docstring coverage, performs static analysis and type checking with `mypy`, and performs unit testing with `pytest`. All these checks are orchestrated via a Makefile.

Rust continuous integration. Similarly, modifications to the Rust binding code trigger a separate workflow to lint and format the Rust code. A scheduled workflow is also in place to clear CI/CD caches.

Continuous deployment. Upon pushing Python binding code to the trunk, a release workflow is executed. This comprehensive workflow generates Python wheels for a range of platforms and architectures, including various Linux targets (x86_64, x86, aarch64, armv7, s390x, ppc64le), Windows (x64, x86), and macOS (x86_64, aarch64), creates a release following semantic versioning; and publishes the package to PyPI.

A.5 Multi-language support in DIAMOND

The following code snippets illustrate the core graph loading and saving operations using the `PropertyGraph` class across different language bindings for the DIAMOND library. First, the user can instantiate a `PropertyGraph` object by reading data from a source file in the JSON Lines format using methods like `read_pg_jsonl` or language-specific equivalents. This process parses the input data and constructs the in-memory graph representation. Subsequently, the `write_diamond` (or equivalent) method allows the user to serialize the `PropertyGraph` object into the `.diamond` binary format. This can be accomplished as follows in the Rust core:

```rust
use diamond_core::PropertyGraph;

fn main() -> Result<(), Box<dyn std::error::Error>> {
        let pg = PropertyGraph::read_pg_jsonl("./data/my_graph.jsonl")
        .expect("Error reading graph");
        println!("Successfully read graph");

        pg.write_diamond("./data/my_graph.diamond")
        .expect("Error writing diamond file");
        println!("Successfully wrote graph");
}
```

The TypeScript binding allows the same operations to be performed in TypeScript:

```typescript
import { PropertyGraph } from "diamond-graph";

const pg = PropertyGraph.readPgJsonl(inputJsonlPath);
console.log(`Successfully read graph from ${inputJsonlPath}`);
pg.writeDiamond(outputDiamondPath);
```

Finally, the Python binding allows the same operations to be performed in Python:

```python
from diamond_graph import PropertyGraph

pg = PropertyGraph.read_pg_jsonl("./data/my_graph.jsonl")
pg.write_diamond("./data/my_graph.diamond")
```

A.6 Identifying graph differences through recursion

To compute the differences between two graphs, we recursively apply two generic primitives, `computeSetChanges` and `computeListChanges`. These graph deltas are then useful in the merging process.

A.6.1 Computing set changes

As described in Algorithm 15, we first define a primitive to determine the changes necessary to transform an initial "source" set of elements, denoted as S_{source}, into a "target" set, S_{target}, by identifying elements that must be added, removed, or modified. We define a comparison function, $f_{\text{compare}}(e_1, e_2)$, which evaluates any two elements e_1 and e_2 (one from the source and one from the target) and determines if they are identical (`Equal`), if the target element is a modified version of the source element (`Modified`), or if they are otherwise distinct.

Next, we attempt to establish correspondences between elements in S_{source} and S_{target}. The algorithm iterates through each element $s_i \in S_{\text{source}}$. For each s_i that has not yet been matched, it then scans through elements $t_j \in S_{\text{target}}$ that also remain unmatched. Upon comparing s_i and t_j using f_{compare}, if an `Equal` relationship is found, s_i is recorded as unchanged, and both s_i and t_j are marked as accounted for, preventing their re-evaluation. The search for a match for s_i then concludes. Similarly, if f_{compare} indicates a `Modified` relationship, the pair (s_i, t_j) is stored to signify that s_i transforms into t_j, both elements are marked as accounted for, and the algorithm moves to the next source element. This systematic pairing ensures that each element from either set is part of at most one such `Equal` or `Modified` relationship.

Following this matching phase, the algorithm identifies elements for removal by examining the source set. Any element $s_i \in S_{\text{source}}$ that was not marked as matched during the previous step is considered to be absent from the target set (either directly or as a modified version) and is thus designated for removal. Conversely, elements for addition are identified by examining the target set. Any element $t_j \in S_{\text{target}}$ that remains unmarked is interpreted as a new element not present in the source set and is designated for addition.

Ultimately, the algorithm outputs four collections: a list of elements to be added (L_{add}), a list of elements to be removed (L_{remove}), a list of pairs representing modifications (L_{modify}), and a list of elements that were found in both sets and remained unchanged (L_{no_change}). These collections collectively define the delta transforming S_{source} into S_{target}. The primary computational load arises from the nested iterative search for matches, leading to a worst-case time complexity of $O(|S_{\text{source}}| \cdot |S_{\text{target}}|)$ comparisons.

A.6.2 Computing list changes

Next, as shown in Algorithm 14, we define a primitive to determine the most efficient sequence of changes necessary to transform an initial "source" list of elements into a "target" list using dynamic programming. Note that, unlike Algorithm 15, this primitive deals with ordered lists rather than unordered sets.

The first step involves constructing the cost matrix, where each cell represents the minimum number of operations required to convert a prefix of the source sequence into a prefix of the target sequence. The matrix edges are initialized, which corresponds to transforming a sequence into an empty one (requiring deletions) or an empty sequence into a target sequence (requiring insertions). Then, the rest of the matrix is iteratively computed. For any given pair of prefixes, the last elements of these prefixes are considered to calculate the transformation cost. If these elements are deemed identical by a provided comparison function $f_{\text{compare}}(e_1, e_2)$, no new cost is incurred, and the value is carried over from a previous state. If they differ, the algorithm explores the costs of three potential operations: deleting the element from the source, inserting the element into the target, or modifying the source element to match the target element. The comparison function can assign different costs based on whether differing elements are considered "modified" versions or entirely distinct. The algorithm always chooses the operation that produces the minimum cumulative cost for that particular cell.

Once this cost matrix is computed, the value in the cell corresponding to the full source and target sequences represents the total minimum cost for the entire transformation. Starting from this final cell, the algorithm traces a path back to the beginning of the matrix. The path taken is determined by reversing the equality, modification, addition, or removal decisions made during the matrix construction. This path directly translates into the sequence of operations that optimally transform the source sequence into the target sequence. The final output is this ordered list of changes. This algorithm is quadratic in terms of the lengths of the two sequences, both in time and memory.

A.6.3 Leveraging change primitives to compute graph differences

We recursively utilize the following primitives to compare two graphs, starting at the highest level and traversing all the way down to single property lists. This level of detail enables us to write very flexible logic for the graph merging strategies.

Algorithm 8 COMPARE_GRAPHS

Input: `source`, `target` ▷ full graph objects
Output: `graphChanges` ▷ {`nodeChanges`, `edgeChanges`}

```
nodeChanges ← CompareNodes(source.nodes, target.nodes)
edgeChanges ← CompareEdges(source.edges, target.edges)
return { nodeChanges, edgeChanges }
```

Algorithm 9 COMPARE_NODES

Input: `source`, `target` ▷ lists of nodes
Output: `nodeChanges` ▷ add / remove / modify / unchanged summary

```
procedure cmpNode(n_a, n_b)
    if NodeKey(n_a) ≠ NodeKey(n_b) then
        return Different
    end if
    labelDiffs ← CompareLabels(n_a.labels, n_b.labels)
    propertyDiffs ← CompareProperties(n_a.properties, n_b.properties)
    labelsEqual ← every change in labelDiffs is Equal
    propsEqual ← propertyDiffs.add = remove = modify = 0
    return if labelsEqual ∧ propsEqual then Equal else Modified
end procedure
nodeChanges ← ComputeSetChanges(Set(source), Set(target), cmpNode)
return nodeChanges
```

Algorithm 10 COMPARE_EDGES

Input: `source` ▷ list of edges in graph A
Input: `target` ▷ list of edges in graph B
Output: `edgeChanges` ▷ add / remove / modify / unchanged summary

```
procedure cmpEdge(e_a, e_b)                    ▷ returns Equal, Modified, Different
    if EdgeKey(e_a) ≠ EdgeKey(e_b) then
        return Different                       ▷ different IDs → cannot match
    end if
    labelDiffs ← CompareLabels(e_a.labels, e_b.labels)
    propertyDiffs ← CompareProperties(e_a.properties, e_b.properties)
    sameEnds ← (e_a.from = e_b.from) ∧ (e_a.to = e_b.to)
    sameDir ← (e_a.undirected = e_b.undirected)
    labelsEqual ← every change in labelDiffs is Equal
    propsEqual ← propertyDiffs.add = remove = modify = 0
    return if sameEnds ∧ sameDir ∧ labelsEqual ∧ propsEqual then Equal else
Modified
end procedure
edgeChanges ← ComputeSetChanges(Set(source), Set(target), cmpEdge)
return edgeChanges
```

Algorithm 11 COMPARE_LABELS

Input: `srcLabels`, `tgtLabels` ▷ ordered lists
Output: `labelChanges` ▷ add / remove / modify / unchanged summary

```
procedure CMPLABEL(a, b)
    return if a = b then Equal else Different
end procedure
return COMPUTELISTCHANGES(srcLabels, tgtLabels, cmpLabel)
```

Algorithm 12 COMPARE_PROPERTIES

Input: `srcProps`, `tgtProps` ▷ maps key → list
Output: `propChanges` ▷ set-style diff for (key, list) entries

```
procedure CMPENTRY((k_a, v_a), (k_b, v_b))
    if k_a ≠ k_b then
        return Different
    end if
    listDiffs ← COMPAREPROPERTYLIST(v_a, v_b)
    return if every change in listDiffs is Equal then Equal else Modified
end procedure
propChanges ← COMPUTESETCHANGES( Set(Entries(srcProps)), Set(Entries(tgtProps)), cmpEntry)
return propChanges
```

Algorithm 13 COMPARE_PROPERTY_LIST

Input: `srcList`, `tgtList` ▷ ordered value lists
Output: `valueChanges` ▷ list-style diff result

```
procedure CMPVALUE(a, b)
    return if a = b then Equal else Different
end procedure
return COMPUTELISTCHANGES(srcList, tgtList, cmpValue)
```

A.7 Using graph differences to solve the three-way merge

When merging LPGs, a crucial first step is to reliably identify corresponding nodes and edges across different versions of the graph: `base`, `ours`, and `theirs`. We achieve this using unique keys generated by the `nodeKey` and `edgeKey` functions. This keying strategy allows us to distinguish between structural changes (deletion or addition) and modifications. If a change causes an element's key to differ from its base version, then that change is regarded as structural (*i.e.*, the old element was deleted and a new element was added). A modification occurs when an element retains the *same* key but its labels or properties are altered. Similarly to our diffing computation approach, we use primitives based on sets and lists to perform all the necessary computations to merge graphs across two branches.

The `mergeSets` algorithm implements a three-way merge for sets, designed to reconcile differences between a local version (S_{ours}) and a remote version (S_{theirs}), both derived from a common ancestor (S_{base}). The algorithm is generic, operating on elements of type `T`. It leverages a caller-provided comparison function, $f_{\text{compare}} : (T, T) \to$ `ComparisonResult`, to determine if two elements are identical, modified, or distinct. A diffing function, f_{diff} (which defaults to `computeSetChanges`), is used to calculate the changes (`SetChanges<T>`, comprising additions, removals, and modifications) between S_{base} and S_{ours}, and between S_{base} and S_{theirs}. An important aspect of this algorithm is its pluggable conflict resolution mechanism, defined by a `MergeStrategy<T>` (denoted as Σ), which dictates how disagreements are handled. The function returns a `MergeResult<T>` containing the `merged` set (S_{merged}) and an array of any `conflicts` encountered.

The process begins by computing the deltas: $\Delta_{\text{ours}} = f_{\text{diff}}(S_{\text{base}}, S_{\text{ours}}, f_{\text{compare}})$ and $\Delta_{\text{theirs}} = f_{\text{diff}}(S_{\text{base}}, S_{\text{theirs}}, f_{\text{compare}})$. These deltas itemize elements added to, removed from, or modified

in S_{ours} and S_{theirs} relative to S_{base}. For efficient lookup, modifications are stored in maps (M_{ours}, M_{theirs}), mapping base elements to their modified versions, and deletions are stored in sets (D_{ours}, D_{theirs}). The algorithm then iterates through each element $e_{\text{base}} \in S_{\text{base}}$ to determine its fate in S_{merged}. The following cases arise. Let e'_{ours} be the modified version of e_{base}.

First, cases 1-4 consider when the element was modified in S_{ours} (Figure 9, left panel).

Case 1: modified identically in both branches. If e_{base} was also modified in S_{theirs} to e'_{theirs} (a "modify-modify" scenario), and if $f_{\text{compare}}(e'_{\text{ours}}, e'_{\text{theirs}})$ yields `Equal`, e'_{ours} (or e'_{theirs}) is added to S_{merged}.

Case 2: modified differently in both branches. Otherwise, a `MergeConflict<T>` of type "modify-modify" is created with e_{base}, e'_{ours}, and e'_{theirs}. The strategy Σ is invoked. If it returns a resolved value, that value is added to S_{merged}; otherwise, the conflict is recorded.

Case 3: modified in ours, deleted in theirs. If e_{base} was deleted in S_{theirs} (a "modify-delete" scenario, from ours/theirs perspective): A conflict of type "modify-delete" is created (with e_{base} and e'_{ours}). The strategy Σ is invoked. If resolved, the result is added to S_{merged}; otherwise, the conflict is recorded.

Case 4: modified in ours, unchanged in theirs. If e_{base} was unchanged in S_{theirs} (not modified or deleted): e'_{ours} is added to S_{merged}.

Next, cases 5-7 consider when the element was deleted in S_{ours}, not modified (Figure 9, left panel).

Case 5: deleted in ours, modified in theirs. If e_{base} was modified in S_{theirs} to e'_{theirs}, a conflict of type "delete-modify" is created (with e_{base} and e'_{theirs}). The strategy Σ is invoked. If resolved, the result is added to S_{merged}; otherwise, the conflict is recorded.

Case 6: deleted in ours, unchanged in theirs. If e_{base} was unchanged in S_{theirs}, e_{base} is considered deleted and is not added to S_{merged}.

Case 7: deleted in both. If e_{base} was also deleted in S_{theirs}, e_{base} is considered deleted and is not added to S_{merged}.

Next, cases 8-10 consider when the element was unchanged S_{ours}, not modified or deleted (Figure 9, left panel).

Case 8: unchanged in ours, modified in theirs. If e_{base} was modified in S_{theirs} to e'_{theirs}, e'_{theirs} is added to S_{merged}.

Case 9: unchanged in ours, deleted in theirs. If e_{base} was deleted in S_{theirs}, e_{base} is not added to S_{merged}.

Case 10: unchanged in both. If e_{base} was also unchanged in S_{theirs}, e_{base} is added to S_{merged}.

After processing all elements from S_{base}, the algorithm handles additions. Let A_{ours} be the set of elements added in S_{ours} and A_{theirs} be those added in S_{theirs}. The algorithm iterates through each $a_{\text{ours}} \in A_{\text{ours}}$. It attempts to find a corresponding $a_{\text{theirs}} \in A_{\text{theirs}}$ such that $f_{\text{compare}}(a_{\text{ours}}, a_{\text{theirs}})$ is either `Equal` or `Modified`. A set $S_{\text{matchedTheirs}}$ tracks elements from A_{theirs} that have already been matched. If such a match $a_{\text{match}} \in A_{\text{theirs}}$ is found, then a_{match} is added to $S_{\text{matchedTheirs}}$. The following cases then arise (Figure **??**, right panel).

Case 11: added identically to both. If $f_{\text{compare}}(a_{\text{ours}}, a_{\text{match}})$ was `Equal`, so a_{ours} is added to S_{merged} (representing a common addition).

Case 12: added differently to both. If $f_{\text{compare}}(a_{\text{ours}}, a_{\text{match}})$ was `Modified` (an "add-add-different" scenario), so a conflict of type "add-add-different" is created with a_{ours} and a_{match}. The strategy Σ is invoked. If resolved, the result is added to S_{merged}; otherwise, the conflict is recorded.

Case 13: added in ours, not added in theirs. If no such match is found for a_{ours}: a_{ours} is considered a unique addition by "ours" and is added to S_{merged}.

Case 14: not added in ours, added in theirs. Finally, any elements $a_{\text{theirs}} \in A_{\text{theirs}}$ that were not in $S_{\text{matchedTheirs}}$ are considered unique additions by "theirs" and are added to S_{merged}.

Present in base

Ours \ Theirs	+	−	∼	/
+	×	×	×	×
−	×	7	5	6
∼	×	3	1,2	4
/	×	9	8	10

Not present in base

Ours \ Theirs	+	−	∼	/
+	11,12	×	×	13
−	×	×	×	×
∼	×	×	×	×
/	14	×	×	×

No conflict Potential conflict Impossible case

Figure 9: Cases for merging set primitives. Case 2 results in a "modify-modify-different" conflict, Case 3 in a "modify-delete" conflict, Case 5 in a "delete-modify" conflict, and Case 12 in a "add-add-different" conflict. All green cases results in no conflict, and cases 1 and 11 specifically result in no conflict because the added or modified elements in both branches are equal modulo f_{compare}. All red cases cannot occur (*e.g.*, a branch cannot add an element that already existed in base).

The `MergeStrategy<T>` Σ is a function type (`MergeConflict<T>` $\times \mathcal{F}_{\text{compare}} \to$ `Resolution<T>`). A `Resolution<T>` can either indicate a successful resolution with a resulting value, or that the conflict remains unresolved. The default strategy, `throwAllConflictsStrategy`, leaves all conflicts unresolved. An example strategy like `timestampWins` might resolve "modify-modify" conflicts by selecting the version with a more recent `updatedAt` timestamp.

The algorithm concludes by returning S_{merged} and a list of all `MergeConflict<T>` objects for conflicts that were not resolved by the chosen strategy Σ.

A.7.1 Merging list primitives

The `mergeLists` algorithm performs a three-way merge for ordered lists (arrays), reconciling a local version (L_{ours}) and a remote version (L_{theirs}) against a common ancestor (L_{base}). This function is generic for elements of type `T`. It relies on a comparison function, $f_{\text{compare}} : T \times T \to$ `ComparisonResult`, to ascertain equality or difference between elements. A diffing function, f_{diff} (defaulting to `computeListChanges`), is employed to generate sequences of changes, Δ_{ours} and Δ_{theirs}, representing the transformations from L_{base} to L_{ours} and L_{base} to L_{theirs}, respectively. Each change object in these sequences specifies an operation (such as `Add`, `Remove`, `Modified`, or `Equal`), associated elements, and relevant indices from L_{base}. A pluggable `ListMergeStrategy<T>`, denoted as Σ_{list}, is used for resolving conflicts. The function outputs a `ListMergeResult<T>`, which includes the merged list, L_{merged}, and an array of any unresolved `ListMergeConflict<T>` objects, C_{list}.

Initially, the algorithm computes the delta sequences: $\Delta_{\text{ours}} = f_{\text{diff}}(L_{\text{base}}, L_{\text{ours}}, f_{\text{compare}})$ and $\Delta_{\text{theirs}} = f_{\text{diff}}(L_{\text{base}}, L_{\text{theirs}}, f_{\text{compare}})$. Pointers, p_{ours} and p_{theirs}, are initialized to traverse Δ_{ours} and Δ_{theirs}, respectively. The core of the algorithm is a loop that continues as long as there are changes to process in either delta sequence. Inside the loop, let c_{ours} be the current change from Δ_{ours} and c_{theirs} be from Δ_{theirs}.

First, we consider if one delta sequence is exhausted. If c_{theirs} is undefined (all changes in Δ_{theirs} processed), the remaining changes $c_{\text{ours}} \in \Delta_{\text{ours}}$ are processed. If c_{ours} indicates an `Add` or `Modified` operation, its target element is added to L_{merged}. If it's an `Equal` operation, its source element is

added. `Remove` operations are implicitly handled by not adding the element. p_{ours} is incremented. A symmetric process occurs if c_{ours} is undefined.

Next, we consider if both delta sequences have changes. Let $\texttt{op}_{\text{ours}}$ and $\texttt{op}_{\text{theirs}}$ be the operations for c_{ours} and c_{theirs}. First, we consider addition operations.

Concurrent additions. If $\texttt{op}_{\text{ours}} = \texttt{Add}$ and $\texttt{op}_{\text{theirs}} = \texttt{Add}$, if $f_{\text{compare}}(c_{\text{ours}}.\text{targetElement}, c_{\text{theirs}}.\text{targetElement})$ is `Equal`, c_{ours}.targetElement is added to L_{merged}. Otherwise, an "add-add-concurrent" conflict is created with c_{ours}.targetElement and c_{theirs}.targetElement. This conflict is passed to the `handleListConflict` helper, which uses Σ_{list} for resolution. In either case, both p_{ours} and p_{theirs} are incremented.

Unilateral additions on ours or theirs. If $\texttt{op}_{\text{ours}} = \texttt{Add}$ (and $\texttt{op}_{\text{theirs}}$ is not `Add`), c_{ours}.targetElement is added to L_{merged}. p_{ours} is incremented. Similarly, if $\texttt{op}_{\text{theirs}} = \texttt{Add}$ (and $\texttt{op}_{\text{ours}}$ is not `Add`), c_{theirs}.targetElement is added to L_{merged}. p_{theirs} is incremented.

Next, we consider non-add operations (referencing L_{base} elements). Both $\texttt{op}_{\text{ours}}$ and $\texttt{op}_{\text{theirs}}$ are either `Modified`, `Remove`, or `Equal`. Let $\texttt{idx}_{\text{ours}} = c_{\text{ours}}.\text{sourceIndex}$ and $\texttt{idx}_{\text{theirs}} = c_{\text{theirs}}.\text{sourceIndex}$.

Non-add operation on the same element. If $\texttt{idx}_{\text{ours}} = \texttt{idx}_{\text{theirs}}$ (both changes refer to the same element in L_{base}), let $e_{\text{base}} = L_{\text{base}}[\texttt{idx}_{\text{ours}}]$. We have the following cases:

- If $\texttt{op}_{\text{ours}} = \texttt{Equal}$ and $\texttt{op}_{\text{theirs}} = \texttt{Equal}$, c_{ours}.sourceElement is added to L_{merged}.
- If $\texttt{op}_{\text{ours}} = \texttt{Modified}$ and $\texttt{op}_{\text{theirs}} = \texttt{Modified}$: If $f_{\text{compare}}(c_{\text{ours}}.\text{targetElement}, c_{\text{theirs}}.\text{targetElement})$ is `Equal`, c_{ours}.targetElement is added. Otherwise, a "modify-modify" conflict (with e_{base}, $\texttt{idx}_{\text{ours}}$, c_{ours}.targetElement, c_{theirs}.targetElement) is handled via Σ_{list}.
- If $\texttt{op}_{\text{ours}} = \texttt{Modified}$ and $\texttt{op}_{\text{theirs}} = \texttt{Remove}$, a "modify-delete" conflict (with e_{base}, $\texttt{idx}_{\text{ours}}$, c_{ours}.targetElement) is handled.
- If $\texttt{op}_{\text{ours}} = \texttt{Remove}$ and $\texttt{op}_{\text{theirs}} = \texttt{Modified}$, a "delete-modify" conflict (with e_{base}, $\texttt{idx}_{\text{ours}}$, c_{theirs}.targetElement) is handled.
- If $\texttt{op}_{\text{ours}} = \texttt{Remove}$ and $\texttt{op}_{\text{theirs}} = \texttt{Remove}$, the element is implicitly deleted.

Both p_{ours} and p_{theirs} are incremented.

Non-add operation on different elements. If $\texttt{idx}_{\text{ours}} < \texttt{idx}_{\text{theirs}}$, c_{ours} is processed. If $\texttt{op}_{\text{ours}}$ is `Equal` or `Modified`, the respective element (c_{ours}.sourceElement or c_{ours}.targetElement) is added to L_{merged}. p_{ours} is incremented. If $\texttt{idx}_{\text{theirs}} < \texttt{idx}_{\text{ours}}$, c_{theirs} is processed similarly. p_{theirs} is incremented.

The `handleListConflict` helper function takes a `ListMergeConflict<T>`, the strategy Σ_{list}, L_{merged}, C_{list}, and f_{compare}. It calls $\Sigma_{\text{list}}(\texttt{conflict}, f_{\text{compare}})$. If the returned `Resolution<T>` indicates `resolved` as `true` and provides a `value` (not undefined), this value is pushed to L_{merged}. Otherwise, the conflict is added to C_{list}. The default strategy marks all conflicts as unresolved. The algorithm concludes by returning an object containing L_{merged} and the list C_{list} of any unresolved conflicts.

A.7.2 Merging graphs

A similar approach can be taken to merging graphs through primitives than was taken before in Appendix A.6.3. The only caveat is that recursive calls of primitives to lower level aspects of the graph cannot be treated using a one-size-fits-all solution like for diffing. The merging strategy is in charge of make the sub-call to the appropriate primitive to resolve a merge conflict as it sees fit. For example, suppose an "add-add-different" conflict arises. One strategy might always pick the element with the latest timestamp, requiring no sub-calls, while another, that might attempt to union the properties of both version will have to use call a merge primitive on the two elements.

A.7.3 Sub-algorithms for git driver diffing

Algorithm 14 COMPUTE_LIST_CHANGES

Input: `source` ▷ original list
Input: `target` ▷ desired list
Input: `compare` ▷ function returning `Equal`, `Modified` or `Different`
Output: `changes` ▷ ordered list of edit operations

```
m ← length(source), n ← length(target)
matrix ← (m+1) × (n+1) table filled with zeros

for i = 0 to m do                                   ▷ initialize first row/column
    matrix[i][0] ← i
end for
for j = 0 to n do
    matrix[0][j] ← j
end for

for i = 1 to m do                                   ▷ dynamic-programming fill
    for j = 1 to n do
        cmp ← compare(source[i-1], target[j-1])
        if cmp = Equal then
            matrix[i][j] ← matrix[i-1][j-1]
        else
            removeCost ← matrix[i-1][j] +1
            addCost ← matrix[i][j-1] +1
            replaceCost ← matrix[i-1][j-1] + if cmp = Modified then 1 else 2
            matrix[i][j] ← min(removeCost, addCost, replaceCost)
        end if
    end for
end for

changes ← empty list                                ▷ back-trace to build edit script
i ← m, j ← n
while i > 0 or j > 0 do
    if i > 0 ∧ j > 0 ∧ compare(source[i-1],target[j-1]) = Equal then
        prepend Equal(i-1,j-1) to changes
        i-, j-
    else if i > 0 ∧ j > 0 ∧ compare(source[i-1],target[j-1]) = Modified ∧
matrix[i][j] = matrix[i-1][j-1] + 1 then
        prepend Modified(i-1,j-1) to changes
        i-, j-
    else if j > 0 ∧ (i = 0 ∨ matrix[i][j] = matrix[i][j-1] + 1) then
        prepend Add(j-1) to changes
        j-
    else
        prepend Remove(i-1) to changes
        i-
    end if
end while
return changes
```

A.7.4 Sub-algorithms for graph merging

Algorithm 15 COMPUTE_SET_CHANGES

Input: `sourceSet`, `targetSet` ▷ sets of elements
Input: `compare` ▷ returns `Equal`, `Modified` or `Different`
Output: `changes` ▷ object containing `add`, `remove`, `modify`, `unchanged`

```
source ← list of elements in sourceSet
target ← list of elements in targetSet
toAdd ← empty list
toRemove ← empty list
toModify ← empty list of pairs (old,new)
toDoNothing← empty list
sourceMatched ← list of false of length source
targetMatched ← list of false of length target

for i = 0 to length(source)−1 do                    ▷ match equal or modified pairs
    if sourceMatched[i] then continue
    end if
    for j = 0 to length(target)−1 do
        if targetMatched[j] then continue
        end if
        cmp ← compare(source[i], target[j])
        if cmp = Equal then
            mark sourceMatched[i] and targetMatched[j] as true
            prepend source[i] to toDoNothing
            break inner loop
        else if cmp = Modified then
            prepend (oldValue: source[i], newValue: target[j]) to toModify
            mark sourceMatched[i] and targetMatched[j] as true
            break inner loop
        end if
    end for
end for

for i = 0 to length(source)−1 do                    ▷ collect unmatched elements
    if not sourceMatched[i] then prepend source[i] to toRemove
    end if
end for
for j = 0 to length(target)−1 do
    if not targetMatched[j] then prepend target[j] to toAdd
    end if
end for
return SetChanges(toAdd, toRemove, toModify, toDoNothing)
```

Algorithm 16 MERGE_SETS

```
Input: baseSet, oursSet, theirsSet                          ▷ three versions of the same set
Input: compare                                              ▷ equality / modified / different
Input: diffFn                                               ▷ produces SetChanges
Input: strategy                                             ▷ conflict-resolution policy
Output: merged                                              ▷ final reconciled set
Output: conflicts                                           ▷ unresolved merge conflicts
 1: merged ← empty set                                      ▷ initialise
 2: conflicts ← empty list
 3:
 4: ours ← diffFn(baseSet, oursSet, compare)                ▷ compute deltas
 5: theirs ← diffFn(baseSet, theirsSet, compare)
 6:
 7: ourMods ← map base→new from ours.modify                 ▷ index modifications / deletions
 8: theirMods ← map base→new from theirs.modify
 9: ourDel ← set of ours.remove
10: theirDel ← set of theirs.remove
11:
12: for all item in baseSet do                              ▷ iterate through base elements
13:     oursChanged ← item ∈ ourMods
14:     theirsChanged ← item ∈ theirMods
15:     oursRemoved ← item ∈ ourDel
16:     theirsRemoved ← item ∈ theirDel
17:     if oursChanged then
18:         ourV ← ourMods[item]
19:         if theirsChanged then
20:             theirV ← theirMods[item]
21:             if compare(ourV,theirV)=Equal then
22:                 add ourV to merged
23:             else
24:                 create conflict (modify-modify, item, ourV, theirV)
25:                 res ← strategy(conflict, compare)
26:                 if res.resolved then add res.value to merged
27:                 elsepush conflict
28:                 end if
29:             end if
30:         else if theirsRemoved then
31:             create conflict (modify-delete, item, ourV)
32:             handle with strategy as above
33:         else
34:             add ourV to merged                          ▷ ours only
35:         end if
36:         continue
37:     end if
```

```
    if oursRemoved then
        if theirsChanged then
            theirV ← theirMods[item]
            create conflict (delete-modify, item, theirV)
            handle with strategy
        end if
        continue                                              ▷ deleted in ours
    end if
    if theirsChanged then
        add theirMods[item] to merged
    else if not theirsRemoved then
        add item to merged
    end if
end for

ourAdds ← set of ours.add                                 ▷ stage 4 : handle additions
theirAdds ← set of theirs.add
matchedT ← empty set                                      ▷ their additions already paired
for all a in ourAdds do
    match ← nil; cmpRslt ← nil
    for all b in theirAdds do
        if b ∈ matchedT then continue
        end if
        c ← compare(a,b)
        if c = Equal ∨ c = Modified then
            match ← b; cmpRslt ← c; break
        end if
    end for
    if match then
        add match to matchedT
        if cmpRslt = Equal then
            add a to merged                                   ▷ identical add
        else
            create conflict (add-add-modified, a, match)
            handle with strategy
        end if
    else
        add a to merged                                       ▷ unique add
    end if
end for
for all b in theirAdds do
    if b ∉ matchedT then add b to merged
    end if
end for
return {merged, conflicts}
```

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: We have, to the best of our ability, considered each claim and we believe each to be accurate.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: They are described in Section 4.3.

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (*e.g.*, independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, *e.g.*, if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [NA]

Justification: [NA]

Guidelines:

- The answer NA means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: To the best of our ability, we have made the details necessary to reproduce the experimental results of the paper available in our methodological description.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (*e.g.*, in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (*e.g.*, a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (*e.g.*, with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (*e.g.*, to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [No]

Justification: The code for the environment will be publicly released via an open-source GitHub repository at a future date.

Guidelines:

- The answer NA means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (*e.g.*, for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (*e.g.*, data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [NA]

   Justification: [NA]

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [No]

   Justification: The DIAMOND compression experiments reported in the paper are deterministic given our choice of KG and hardware, both of which are reported in the paper. Therefore, error bars or confidence intervals are not provided, as there is no run-to-run variability to quantify.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).

- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (*e.g.*, Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (*e.g.*negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Yes, this information is provided for every experiment.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (*e.g.*, preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: We believe that we are fully compliant with the NeurIPS Code of Ethics.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (*e.g.*, if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [NA]

    Justification: Our work introduces developer tooling for graph construction and generation. We have not identified direct harms associated with the research process, and negative societal impacts from the intended use of this tooling are unlikely. While automatically generated graphs could contain factual inaccuracies, this risk is not unique to our method and already arises from both human error and existing automation pipelines. Moreover, such inaccuracies are generally low-stakes. Consequently, we assess that the broader societal impact of this work is limited.

    Guidelines:

- The answer NA means that there is no societal impact of the work performed.
- If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (*e.g.*, disinformation, generating fake profiles, surveillance), fairness considerations (*e.g.*, deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (*e.g.*, gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (*e.g.*, pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: [NA]

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (*e.g.*, code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [NA]

    Justification: [NA]

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (*e.g.*, CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (*e.g.*, website), the copyright and terms of service of that source should be provided.

- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [NA]

    Justification: [NA]

    Guidelines:

    - The answer NA means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [NA]

    Justification: [NA]

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: [NA]

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.

- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

    Answer: [NA]

    Justification: [NA]

    Guidelines:

    - The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy (`https://neurips.cc/Conferences/2025/LLM`) for what should or should not be described.