Can Large Language Models Reason on Dynamic Graphs?

Anonymous Author(s)

Abstract

Graphs are essential tools for modeling complex relationships, and recent work has shown that large language models (LLMs) have grown increasingly powerful at reasoning over graph-structured data. This existing work has focused primarily on *static graphs* that do not change over time. In any many applications, however, the underlying graph is *dynamic*, in that it changes over time. In this work, we address the capabilities of LLMs for reasoning on dynamic graphs, focusing on a number of challenging aspects of the problem: the *fully dynamic* case in which both nodes and edges can be added or deleted, and in multiple settings where the graph may be implicitly described in natural-language text or may be represented as structured data. To explore these dimensions of the problem, we introduce **DyGraphQA**, a new benchmark dataset for dynamic graph reasoning by LLMs. The benchmark contains prompts specifying graphs both in natural language (**DyGraphQA-Real**) and as structured data (**DyGraphQA-Synth**). We find that current LLMs struggle with dynamic graph data in both these forms, and analyze how graph structure, size, edge density, and prompting strategies impact performance, finding that each factor significantly shapes model accuracy and reasoning behavior. Our findings highlight a critical gap in current LLM capabilities regarding dynamic graph reasoning tasks and underscore the potential of techniques like MaP to mitigate these challenges.

CCS Concepts

• **Computing methodologies → Natural language generation**.

Keywords

Dyanamic Graph Reasoning, Benchmarks, Large Language Models

ACM Reference Format:

Anonymous Author(s). 2025. Can Large Language Models Reason on Dynamic Graphs?. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 19 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Large Language Models (LLMs) have revolutionized not just the area of natural language processing, but also a wide range of domains that benefit from their abilities in reasoning and problem-solving. In particular, recent work has shown that LLMs display strong reasoning capabilities in domains such as math [1, 5, 13, 27, 34, 42], coding [18, 28, 31, 36], and common-sense reasoning [3, 4, 6, 8, 21, 38].

As a result, an important ongoing project for the field is to understand LLMs' relative strengths and weaknesses in reasoning over different kinds of data. Exploring this requires careful evaluation on a range of fundamental types of data, ranging from unstructured data to different forms of structured data. While much research has explored LLMs' ability to reason over unstructured data like text, relatively little is known about their reasoning on structured data.

Graphs are a central type of structured data that appears frequently in real-world problems, whenever the goal is to represent relationships and connections between entities. Given that graphs are a common and fundamental structured representation, recent work has begun to investigate LLMs' ability to reason over them. For example, [30] introduced a benchmark with textual graph descriptions followed by queries like shortest path, evaluating LLM performance. Similarly, [10] expanded this work by posing more fundamental questions and varying graph representations in text from natural language to data structures. Overall, these studies show that LLM performance varies across graph-based reasoning tasks, with some proving significantly more challenging than others.

Prior research has investigated the graph reasoning abilities of LLMs, specifically their ability to extract properties of input graphs represented either as *structured data* (such as adjacency matrices or edge lists) [30] or in *natural language* [10]. These studies primarily focus on *static* graphs, where the structure is fixed and presented as a single snapshot in time. However, in many real-world applications, graphs are rarely static; they evolve over time with the addition or removal of nodes and edges. Such *dynamic graphs* appear frequently in domains such as social network analysis, where relationships and interactions constantly change ([17]), and in evolving knowledge bases that must adapt to new information ([25, 29]).

Despite the prevalence of dynamic graphs, far less research has explored how well LLMs handle them. To fully assess the graph reasoning capabilities of modern LLMs and ensure applicability to real-world situations, it is essential to evaluate their ability to understand and manipulate graphs that change over time. In this paper, we address the capabilities of LLMs for reasoning on dynamic graphs, exploring multiple dimensions of the problem. In particular, we study the *fully* dynamic setting, in which both nodes and edges can be added or deleted; and we study dynamic graphs presented both through natural language descriptions and through structured data. Both of these modalities are important for understanding the problem: while evaluating on structured data is important for isolating graph reasoning from language understanding biases, dynamic graphs frequently appear in natural language as well. Many NLP tasks inherently involve dynamic graph reasoning, as implicit dynamic graphs emerge in text. For instance, tracking whether two characters in a story remain connected by its end requires identifying their initial relationships, following changes over time, and verifying a persistent link.

Conference acronym 'XX, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

We can therefore summarize our central research question as follows:

*Can LLMs reason on **fully dynamic graphs** presented either in **natural language** or as **structured data**?*

To address this question, we introduce **DyGraphQA**, a novel and challenging benchmark dataset containing over 160,000 question-answer pairs pertaining to fully dynamic graph reasoning. DyGraphQA consists of two component datasets: **DyGraphQA-Real**, featuring real-world prompts in natural language, and **DyGraphQA-Synth**, containing synthetic graph prompts as structured data. By evaluating both representations, we provide a comprehensive assessment of LLMs' ability to reason over fully dynamic graphs.

The work closest to ours is a recent study that explored LLMs' spatial-temporal understanding of dynamic graphs presented as timestamped edge lists [41]. A key distinction between our work and theirs is that the core problems are formulated very differently, enabling us to explore a number of dimensions that are not present in their work. Specifically, their work only considers partially dynamic graphs in which the node set remains constant, while edges can only be added. Also, their work used only structured data for representing graphs, whereas we consider both structured data and natural language. Finally, we ask about a much wider range of graph properties, whereas [41] focuses on basic retrieval questions such as when two nodes become connected or retrieving the list of neighbors of a node. As discussed above, many of the key applications of graph reasoning by LLMs involve fully dynamic graphs represented in natural language (for example, any textual data that describes changing relationships among entities over time), and evaluating this setting is a distinctive feature of our approach.

In this paper, we first use **DyGraphQA-Real** to evaluate LLMs' reasoning over fully dynamic graphs presented in natural language. Since many real-world dynamic graph tasks naturally appear in text, we constructed prompts based on real-world fully dynamic graph data. However, publicly available datasets are limited, as most dynamic graph datasets are only partially dynamic, tracking only edge additions while keeping the node set fixed. To address this gap, we created **coauth-DBLP-fully**, a dataset of 750 real-world fully dynamic ego-networks extracted from the coauth-DBLP coauthorship dataset. Each ego-network's first five years of collaborations is presented as a chronologically ordered edge list. We then iterate over the network's remaining history, adding and removing both nodes and edges over time. At the end of the sequence, we pose a follow-up question to form the prompt. Our results show that SOTA LLMs vary significantly in their ability to answer these questions, with performance generally declining as the number of graph modifications increases. Advanced reasoning models like o1-mini and o3-mini outperformed others but still exhibited performance gaps on more challenging questions.

To assess LLMs' reasoning on fully dynamic graphs from structured data, we use **DyGraphQA-Synth**, which includes 250 generated Erdős-Rényi (ER) graphs. Each prompt included (1) an adjacency matrix representation of the graph, (2) a sequence of modifications to be performed, and (3) a final query. Our results show that state-of-the-art (SOTA) LLMs struggle with dynamic graph reasoning over adjacency matrices, even after a minimal number of modifications. To address this, we explore techniques to improve LLM performance on fully dynamic graphs. We find that Chain-of-Thought (CoT) prompting enhances performance for Claude 3.5 Sonnet and Llama 3.1 405B, though additional CoT examples provide diminishing returns. More notably, prompting models to output intermediate graphs leads to significant improvements. We introduce *Modify-and-Print (MaP)* prompting, a simple yet effective technique where models explicitly print the adjacency matrix after each modification step. We additionally investigate how graph structure, size, and edge density impact performance.

Overall, LLMs are still not proficient in reasoning on fully dynamic graphs presented in either natural language or as structured data. The observed difficulties with graph modifications and the adjacency matrix underscore the need for improved models or training strategies that can handle dynamic, structured data more effectively. These results call for a shift in benchmarking practices toward tasks that require reasoning over evolving graph data, thereby better aligning evaluations with real-world applications in dynamic networks and systems.

In summary, this work makes the following contributions:

- **Introduction of DyGraphQA:** We present a novel benchmark dataset that provides a rigorous testbed for dynamic graph reasoning across graphs presented in either natural language or as structured data.
- **Analysis of LLM Performance on Graph Modifications:** We reveal that SOTA LLMs experience significant performance degradation on graph modification tasks, especially with increasing numbers of modifications.
- **Exploration of factors affecting LLM performance on dynamic graphs:** We analyze how graph structure, size, edge density, and prompting strategies—including in-context learning and our *MaP* method—impact performance, finding that each factor significantly shapes model accuracy and reasoning behavior.

By addressing the challenges identified in this study, we aim to advance the development of LLMs capable of sophisticated reasoning over dynamic and structured data, thereby expanding their applicability in complex, real-world scenarios. Our code and datasets are openly accessible.[1]

2 Related Works

[30] explores the capability of LLMs to tackle various graph-based tasks. This study evaluates tasks such as topological sort, maximum flow, and bipartite graph matching. [10] delves into different methods for encoding graphs as text, with a particular focus on evaluating different encodings of graphs as text. This work builds upon [30] by introducing more interpretable, straightforward, and fundamental tasks, focusing on fundamental graph properties. Following [10] and [30], we freeze the parameters of the LLM, and the model operates in a black box setup, consuming and producing text. Both sets of tasks found in [30] and [10] are applied only for static graphs, whereas we focus on the fully dynamic case to assess multi-step dynamic graph reasoning [35, 37].

Outside of [10] and [30], there exists an emerging body of work at the intersection of LLMs and graph reasoning [7, 9, 11, 12, 22, 23]. [24] directly follows up on [10] by utilizing and finetuning soft-token prompts to better encode graphs for LLMs. [41] addresses

[1]https://anonymous.4open.science/r/DyGraphQA-7394/

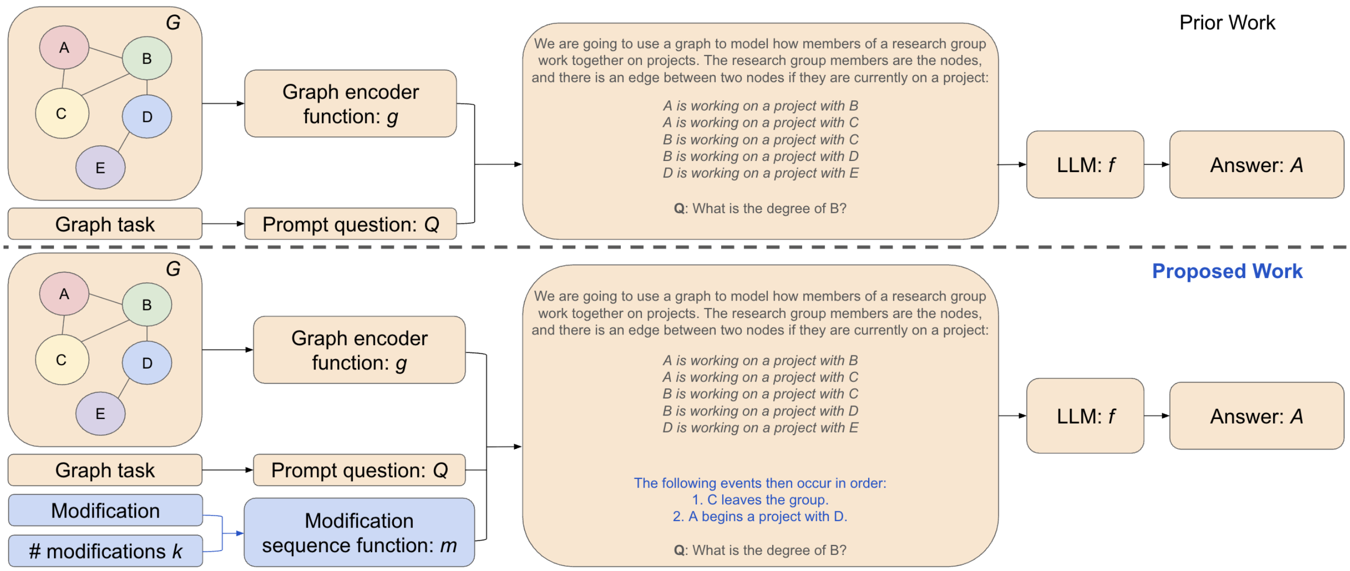


Figure 1: Previous work ([10]) focus their effort on evaluating LLMs on static graphs (top), whereas this work focuses on fully dynamic graph tasks (bottom).

the challenges of solving spatial-temporal problems on partially dynamic graphs using LLMs, evaluating various LLMs' abilities to solve various spatio-temporal graph property tasks. [16] utilizes retrieval-augmented generation techniques to improve LLM performance on graph understanding. [2, 14, 19, 20, 26, 33, 40] provide broad empirical evaluation of LLMs' understanding of graph-structured data. [39] focuses on the generation of graphs from scratch by LLMs, exploring the potential of LLMs to create coherent graph structures. [32] examines how well LLMs can recall graph structures from text.

3 DyGraphQA

In this section, we introduce **DyGraphQA**, a dataset for evaluating the fully dynamic graph reasoning capabilities of LLMs. DyGraphQA consists of two subsets: **DyGraphQA-Real** (D_{Real}) and **DyGraphQA-Synth** (D_{Synth}), each composed of 4-tuples:

$$(g(G), m(m_1, m_2, \dots, m_k), q(Q), S)$$

Here, G is the graph, $g(G)$ is the graph encoding function (e.g., adjacency matrix or descriptive text), $q(Q)$ is the question rephrasing function (e.g., zero-shot or chain-of-thought prompting [34]), and S is the ground-truth answer.

We define $m(m_1, m_2, \dots, m_k)$ as the **modification sequence function**, which lists in text a sequence of k modification events $m_1, m_2, \dots, m_k$ to be applied to G, resulting in a final graph G_k. Each modification event m_i is drawn from a set of possible modification types M, where:

$$m_i \in M = \{\text{AddEdge}, \text{RemoveEdge}, \text{AddNode}, \text{RemoveNode}\}, \forall i$$

Here, **AddEdge(u, v)** adds an edge between two unconnected nodes **u** and **v**, **RemoveEdge(u, v)** removes an existing edge, **AddNode(u)** introduces a new node **u** into the graph, and **RemoveNode(u)** deletes an existing node and all its associated edges.

The LLM f's output $A = f(g(G), q(Q))$ is evaluated against S, with performance measured by the proportion of matches across all tuples in both D_{Real} and D_{Synth}.

The following sections detail the construction of both datasets.

3.1 DyGraphQA-Real

The graphs in **DyGraphQA-Real** are derived from **coauth-DBLP**, a coauthorship dataset with over 1.4 million nodes and 10.5 million edges. This dataset is structured as a timestamped edge list, where each edge is represented as a (Author A, Author B, year) triple, corresponding to a paper written between Author A and Author B at a particular year. To construct DyGraphQA-Real, we sampled 750 nodes and built their **ego-networks**, the subgraph of all edges including the ego (the sampled node) and all edges between the ego's neighbors. Each ego-network was formed by chronologically sorting its corresponding edge list. We preprocess the dataset so that any two authors can only share at most one paper together.

3.1.1 Ego-Network Construction as a Sequence of Modification Events. Our goal is to use coauth-DBLP as the basis for a dataset that is fully dynamic in the sense that edges and nodes can be both added and deleted. Edge addition corresponds naturally to the authoring of a new paper. Similarly, nodes would be added when an author writes their first paper in the ego-network. However, since coauthorship ego-networks are inherently cumulative, defining edge and node deletion events in this context is non-trivial. In practice, many coauthorship datasets include edge and node deletions–for example, during data cleaning or when users remove their papers or delete their accounts on platforms like arXiv or Google Scholar. Modeling such deletions in coauth-DBLP is therefore both a practical necessity and a challenge that tests how well LLMs handle fully dynamic graphs. For edge removal, we add a further dimension for purposes of constructing the dataset: for a parameter t, we declare that each edge corresponding to a paper is deleted again from the graph t years after the paper is written. Here we will use $t = 5$. Thus, in the most straightforward formulation of the dataset, we add each edge when its corresponding paper is published and remove it $t = 5$ years later. Additionally, we remove nodes from an ego-network when their last remaining paper in the ego-network is removed. In this way, every edge and node would have both an addition and a removal event.

However, this approach would result in each ego-network starting and ending with an empty graph, whereas we prefer to start with an initial graph and end with some non-trivial final graph. As a result, we introduce an **initial graph** G, which includes all papers published in the first five years of the ego-network. Any subsequent paper adds an edge via an **Add Edge** event. Edges in both the initial graph and those that are later added are removed from the ego-network **five years after they are added**, except those introduced in the last five years of the ego-network, which remain in the final graph, allowing for a partially full final graph.

Authors are included in the initial graph if they wrote their first-ever paper—whether inside the ego-network with another member of the ego-network or outside it—within or before the initial five-year window. Otherwise, they are added via an **Add Node** event when they publish their first paper (again, even if this paper was published inside our outside the ego-network). A **Remove Node** event is triggered when an author writes their final-ever paper before the end of the ego-network, regardless of whether that paper is inside or outside the ego-network. Authors who continue publishing beyond the ego-network's duration remain in the final graph. In this way, nodes can be added or removed from the graph. This ensures that the ego-network starts with an informative subgraph and maintains a non-trivial structure at the end. Finally, in order to construct the full sequence of modification events, we first note the timestamp where each modification event occurred, and then chronologically sort the set of all timestamped modification events.

3.1.2 *Graph Encoding Function $g(G)$.*

The next challenge is determining how to present the initial graph G in natural language within the prompt. One approach is to explicitly describe it as an ego-network: *"The following graph is a coauthorship ego-network"*, encoding each edge $(\text{Author A}, \text{Author B}, \text{year}) \in G$ as a timestamped paper written between two authors: *"(Author A) wrote a paper with (Author B) in year (year)."*

However, to ensure a more natural presentation of the prompt, we instead frame the graph as a **research group**, where each edge represents a collaboration between two researchers. This framing better aligns with the dynamics of a fully dynamic graph, where both nodes and edges can be added or removed. It is more intuitive to describe researchers joining or leaving a group and collaborations forming or dissolving than to describe papers being added or removed from a coauthorship network.

Thus, we begin each prompt with the textual introduction of the initial graph G found in Figure 1, which begins as follows:

"We are going to use a graph to..."

Additionally, each edge in G is encoded as:

"Author A is working on a project with Author B."

We omit collaboration years, as the chronological order of edges already reflects the graph's temporal evolution.

3.1.3 *Modification Sequence Function $m(\cdot)$.*

After generating the modification sequence $(m_1, m_2, \ldots, m_k)$ as described in Section 3.1.1, we introduce it in the prompt with:

"The following events then occur in order:"

Each modification m_i is expressed in natural language as:

- m_i = Add Edge(u, v): *"i) u starts a project with v"*
- m_i = Remove Edge(u, v): *"i) The project between u and v comes to an end"*
- m_i = Add Node(u): *"i) u joins the group"*
- m_i = Remove Node(u): *"i) u leaves the group"*

Applying this sequence to G returns a final modified graph G_k. Note that, in this process, both the total number of modifications and the type of each modification event are directly derived from the dataset. We sample ego-networks across a range of k values to ensure coverage of both short and long modification sequences.

3.1.4 *Final Question Q and Question Rephrasing $q(Q)$.*

We finally append a Final Question Q to be asked to the model. We include the following graph property questions from [10]: **Node Count** (calculating the total number of nodes in the modified graph G_k), **Edge Count** (calculating the total number of edges present in the G_k), **Node Degree** (calculating the degree of a uniformly sampled node in G_k), and **Connected Nodes** (returning the set of all nodes that are directly connected to a uniformly sampled node in G_k).

In addition to these property tasks, we introduce five new final questions: **Print Graph** (returning the entire G_k in the same format as the graph encoding function $g(G)$), **Isolated Nodes** (returning the set of isolated nodes who have no neighbors in G_k), **Overlapping Nodes** (determining if two sampled nodes were part of the graph at the same), **Overlapping Edges** (determining if two sampled edges were part of the graph at the same), and **Triangle Count** (counting the total number of triangles that formed throughout the entire history of the graph as modifications were applied).

For the **question rephrasing function** $q(Q)$, DyGraphQA-Real uses only **zero-shot prompting** due to the large number of modification steps, which often exhausts the context window, allowing us to evaluate models' innate reasoning abilities.

3.1.5 *Additional Details.*

coauth-DBLP-fully consists of 750 ego-networks, each paired with a sequence of modification events. These are used to construct natural language prompts in **DyGraphQA-Real**, ending with a final question Q. Further dataset generation details, along with a summary table of key dataset statistics, are provided in the accompanying code repository.[2]

We define the *size* of each ego-network as the sum of its initial edge count (which we ensure is at least 10) and the number of applied modifications. Ego-networks are grouped into **small** (10–24), **medium** (25–49), and **large** (50–75) sizes, with 250 samples per group. To ensure diversity, each ego-network contains 7-20 nodes and includes at least one **Add Node** event and one **Remove Node** event. Each ego-network is paired with one of 9 final questions, resulting in $750 \times 9 =$ **6,750 evaluation prompts**.

[2] https://anonymous.4open.science/r/DyGraphQA-7394/

3.2 DyGraphQA-Synth

DyGraphQA-Synth consists of synthetic prompts based on fully dynamic graphs presented as structured data.

3.2.1 Generation of G. Following [30] and [10], we generate 250 undirected Erdős-Rényi (ER) graphs. Each graph has n nodes, with n sampled from a uniform distribution on a finite interval, and edges are added independently with $p \sim U(0, 1)$. This variation in size and density ensures broad coverage of graph structures to evaluate LLM reasoning under diverse structural conditions.

3.2.2 Graph Encoding Function $g(G)$. We introduce and use the **Adjacency Matrix** encoding for all DyGraphQA-Synth prompts due to its difficulty for LLMs. Each entry A_{ij} indicates the presence (1) or absence (0) of an edge between nodes i and j. This encoding has yet to be explored as a graph encoding function in previous studies. Effectively manipulating adjacency matrices is important for LLMs because they are fundamental to many modern graph algorithms and applications.

This encoding challenges LLMs due to its lack of natural language cues, the need to reason about both edge presence and absence, and notably its reliance on an implicit numbering scheme for nodes, where node identifiers correspond directly to the indices of the matrix. When nodes are removed, nodes in the resulting graph must be renumbered to maintain a contiguous matrix structure. For example, if an adjacency matrix represents nodes 0 to 4 and node 2 is removed, the third row and column are eliminated, and subsequent nodes are renumbered—node 3 becomes node 2, node 4 becomes node 3. This renumbering adds an extra layer of complexity for the LLM to manage during reasoning and updates.

3.2.3 Modification Sequence Function $m(\cdot)$. In **DyGraphQA-Synth**, each modification sequence contains a single modification type, allowing us to isolate the effects of each of the four core operations on performance. We also include a fifth type, **Mix**, which samples uniformly from the four base modifications at each step k, introducing modification diversity similar to **DyGraphQA-Real**. We set k to 1-5 as adjacency matrices scale quadratically with graph size, keeping evaluation tractable while still capturing the impact of dynamic modifications.

3.2.4 Final Question Q and Question Rephrasing $q(Q)$. In DyGraph-QA-Synth, we include 5 final questions: **Node Count**, **Edge Count**, **Node Degree**, **Connected Nodes**, and **Print Graph**. We omit **Isolated Nodes**, **Overlapping Nodes**, and **Overlapping Edges** as values of the solutions of these questions changed minimally over few modifications, and omit **Triangle Count** as the high density of larger ER graphs tend to produce excessively high triangle counts.

We use three question rephrasing functions: **zero-shot**, **Chain-of-Thought (CoT)** with 1–3 reasoning examples, and our proposed **Modify-and-Print (MaP)** technique (Section 4.2.3), which encourages state tracking through explicit intermediate outputs.

3.2.5 Additional Details. To illustrate the dataset construction process, we include Algorithm 6 in Section A, which describes how DyGraphQA-Synth entries are generated in order to create **156250 unique example** prompts.

4 Experiments

In this section, we summarize the results of our experiments on both DyGraphQA-Real and DyGraphQA-Synth. We evaluated four SOTA LLMs on both datasets: o1-mini, GPT 4o-mini, Claude 3.5 Sonnet, Llama 3.1 405B, using the OpenAI, Anthropic, and Fireworks AI APIs. We set the decoding temperature of all models to zero. We used the NetworkX library [15] to generate all graphs, as well as the solutions to each final question.

4.1 DyGraphQA-Real

We first evaluate LLM performance on **DyGraphQA-Real**, assessing their ability to reason over fully dynamic ego-networks presented in natural language.

4.1.1 Experimental Setup. For DyGraphQA-Real, we assess recently released models, namely **o3-mini** and **Claude 3.7 Sonnet**, only on **Triangle Count**, as it is the most challenging task.

4.1.2 Findings. Table 1 presents the results. We observe significant variation in LLMs' ability to reason over fully dynamic graphs in natural language, with performance generally decreasing as the number of modifications to the graph increases. Most models struggle with counting tasks such as **Node Count** and **Edge Count**. The exception is **o1-mini**, which only struggles on **Edge Count** for large graphs. This difficulty suggests that LLMs face challenges in tracking simple dynamic properties of the graph over time. Interestingly, models perform better on the **Print Graph** task than on these counting tasks, highlighting a discrepancy between their ability to reconstruct the final graph and their ability to reason over the final graph and derive properties from it.

Performance is notably strong on overlap-based tasks, involving determining whether two authors (**Overlapping Nodes**) or collaborations (**Overlapping Edges**) exist at the same point in time. Models tend to perform better on node-level tasks, such as **Node Degree** and **Connected Nodes**, compared to tasks requiring a more global understanding of the graph.

Larger and more advanced models, including **Llama 3.1 405B** and **Claude 3.5 Sonnet**, consistently outperform the smaller **GPT-4o mini**, which performs poorly across the majority of tasks. Reasoning models, such as **o1-mini**, show notable performance gains over non-reasoning models, particularly on challenging tasks such as **Triangle Count**, suggesting the importance of internal reasoning capabilities for stronger performance on more complex dynamic graph reasoning tasks. However, despite these improvements, the Triangle Count task remains difficult for all models, demonstrating substantial limitations in complex fully dynamic graph reasoning in natural language. While **o3-mini** significantly outperforms other models on this task for small and medium graphs, its performance drops sharply on large graphs. Overall, these findings suggest that while SOTA LLMs demonstrate an preliminary ability to reason over fully dynamic graphs presented in natural language, they still face significant challenges, particularly in accurately tracking and counting both simple and complex graph properties as they evolve.

4.2 DyGraphQA-Synth

With DyGraphQA-Synth, we evaluated the ability of SOTA LLMs to reason over fully dynamic graphs presented as structured data. Due

	Node Count			Edge Count			Node Degree			Connected Nodes			Print Graph			Isolated Nodes			Overlapping Nodes			Overlapping Edges			Triangle Count		
Model	S	M	L	S	M	L	S	M	L	S	M	L	S	M	L	S	M	L	S	M	L	S	M	L	S	M	L
o1-mini	**99.2**	**98.0**	**95.6**	**98.8**	**93.2**	**68.8**	**99.6**	97.6	92.4	88.8	91.6	87.2	**91.6**	74.8	56.8	**92.0**	87.2	87.6	**99.2**	**100.0**	**98.8**	**99.6**	**100.0**	**99.2**	81.2	40.4	7.6
Llama3.1	87.6	78.8	67.6	67.8	45.6	24.4	97.6	90.8	88.0	92.4	95.2	90.0	90.8	71.6	51.3	54.2	62.5	66.7	98.8	99.6	95.6	94.8	92.4	94.8	30.0	20.9	3.2
Claude 3.5	94.0	90.0	79.2	81.2	45.2	32.8	97.6	**98.4**	**96.8**	**97.6**	**99.6**	**96.4**	87.2	**75.6**	**62.8**	86.4	**91.6**	**88.0**	**99.2**	**100.0**	98.4	98.0	99.6	**99.2**	28.0	6.4	0.4
GPT-4o-mini	55.6	55.6	53.6	66.4	31.6	7.2	88.4	70.0	58.0	82.8	68.4	44.4	70.0	38.8	12.0	60.4	56.8	39.6	85.2	84.8	82.8	74.0	79.2	75.2	13.2	2.8	0.4
Claude 3.7	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	44.8	16.4	5.2
o3-mini	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	**98.0**	**79.6**	**42.0**

Table 1: Mean accuracy %s of LLMs on DyGraphQA-Real, with the best result in each column shown in bold.

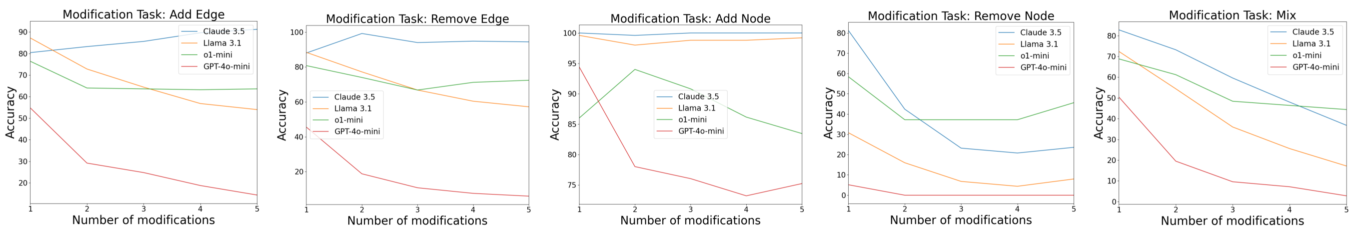


Figure 2: Performance (mean accuracy %s) of LLMs on DyGraphQA-Synth on the Print Graph task.

to monetary constraints, we limited evaluation to o1-mini, GPT-4o-mini, Claude 3.5 Sonnet, and LLaMA 3.1 405B, and encourage future work to extend this benchmarking. DyGraphQA-Synth offers fine-grained control over graph and prompt generation, enabling us to isolate the impact of various components and their interaction with the four core modification types. We therefore perform further ablations, which can be found from Sections 4.2.3 to 4.2.5. For comparison with static prompts (as in [10]), we report results in Tables 2 and 3 in the Appendix.

4.2.1 Experimental Setup. DyGraphQA-Synth contains 250 initial graphs where the size of each graph n is drawn from $U(7, 20)$, and for each pair of nodes (i, j), the probability p that an edge exists between them is also sampled from a uniform distribution $U(0, 1)$. We encoded each of these graphs as adjacency matrices, and applied 1 to 5 modifications for each of the five modification types—**Add Edge**, **Remove Edge**, **Add Node**, **Remove Node**, and **Mix**—resulting in multiple sets of modified graphs. After applying the specified modifications to each initial graph, we posed the **Print Graph** final question to the LLMs, instructing them to output the resulting modified graph in the form of an adjacency matrix. This comprehensive approach allows us to systematically evaluate the models' capabilities in maintaining and updating internal representations of structured data across varying levels of complexity. Results for the remaining final questions can be found in Section B in the Appendix. Additionally, we selected **Print Graph** as it provides a clear means to track modification errors, which we analyze further in Section C.

4.2.2 Findings. Figure 2 illustrates LLM performance on **Print Graph**. Our results indicate that across all modification types, models generally perform worse as the number of modifications increases, which suggests challenges in maintaining and updating an internal graph representation over a small number of modification steps. Notably, the models perform the worst on the **Remove Node** and **Mix** modifications. **Remove Node** is challenging due to required row/column deletion and index renumbering in the adjacency matrix. With **Mix** modifications, the models face the compounded challenge of handling a variety of modification types within a sequence. The necessity to adapt to different operations—such as adding an edge in one step and removing a node in the next—requires flexible reasoning and robust state tracking, which LLMs struggle to perform with the adjacency matrix. Overall, while Claude 3.5 Sonnet outperforms other models across all modification types, o1-mini demonstrates superior performance on the two most challenging modifications, **Remove Node** and **Mix**, at higher modification steps. This suggests that o1-mini's internal reasoning capabilities becomes effective as the complexity of the modification sequence grows.

Overall, our findings indicate that fully dynamic graph reasoning on structured data, specifically adjacency matrices, remains a significant challenge. These results highlight the need for improved models and prompting techniques to enhance LLMs' graph reasoning capabilities in real-world dynamic networks represented as structured data.

4.2.3 Ablation: In-context Learning and MaP Prompting. We explore potential methods for increasing the performance of LLMs on **DyGraphQA-Synth**. We track the performance of various in-context learning methods across 1 to 5 modification steps for the same 250 graphs, and compare the performance of these methods to the previously reported zero-shot performance. Results are shown in Figure 3.

Chain-of-thought (CoT) prompting ([34]) guides the model to generate intermediate reasoning steps by including examples of detailed reasoning in the prompt. We evaluated the effect of including 1, 2, and 3 CoT examples on performance. Models varied significantly in their response to CoT prompting. For Claude 3.5 Sonnet (Figure 3b) and Llama 3.1 405B (Figure 3c), CoT consistently improved performance across all five modification types. In contrast, GPT-4o-mini showed either no improvement or a decline in

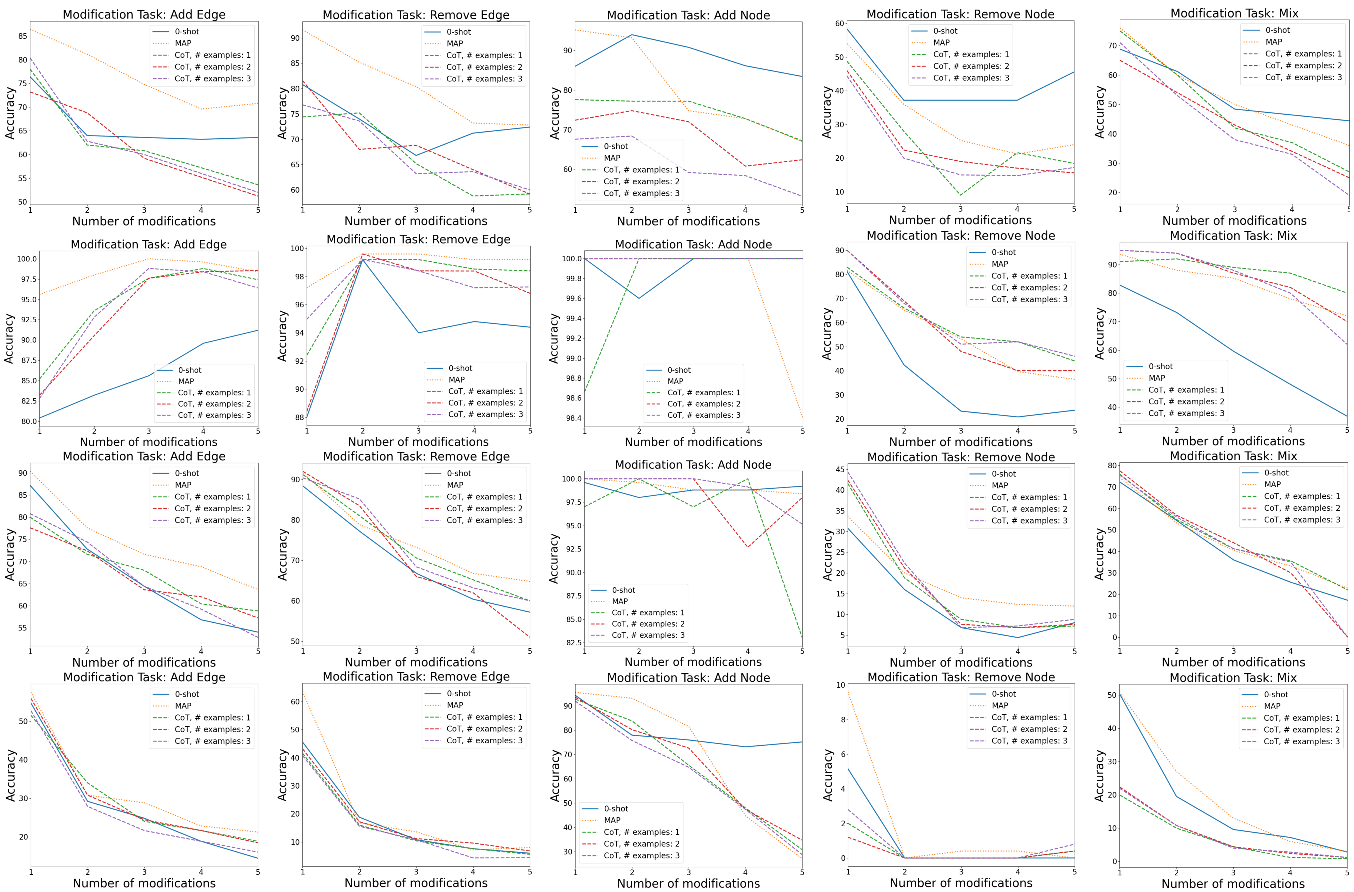
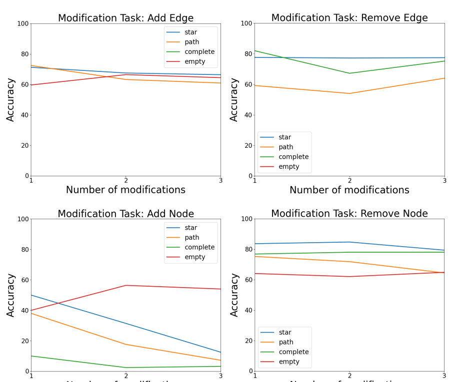
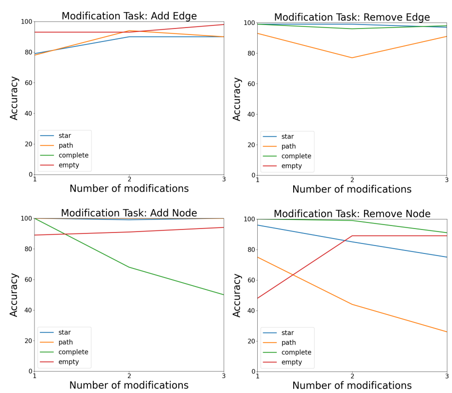


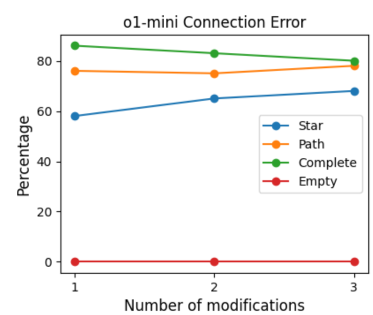
Figure 3: In-context learning results with a) o1-mini, b) Claude 3.5 Sonnet, c) Llama 3.1 405B, and d) GPT-4o-mini.



(a) o1-mini.



(b) Claude 3.5.



(c) Percentage of total errors by o1-mini involving connecting to newly-added node.

Figure 4: Performance across different graph types.

performance with CoT prompting (Figure 3d). o1-mini exhibited a significant drop in performance with CoT prompting (Figure 3a), performing worse than in the zero-shot setting across all modification types. This decline in performance is likely due to the fact that o1-mini reasons internally, and external CoT prompting does not complement its internal reasoning processes. Across all models, performance remained stable regardless of the number of examples, suggesting limited marginal benefit from additional reasoning steps.

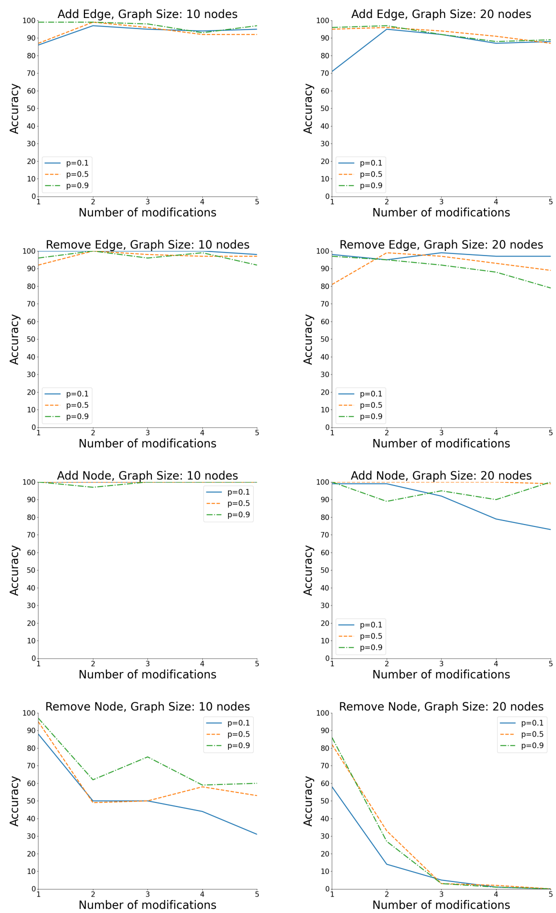


Figure 5: Performance of Claude 3.5 Sonnet on edge density and graph size ablation.

We introduce **Modify-and-Print (MaP) prompting**, a simple yet effective strategy for improving performance on fully dynamic graph reasoning tasks on structured data. In MaP prompting, the model is instructed to output the intermediate graph after each modification. This encourages explicit state tracking and helps the model maintain a coherent internal representation of the evolving graph.

MaP is especially effective for edge-related modifications (**Add Edge**, **Remove Edge**), consistently outperforming both zero-shot and CoT prompting. The gains are particularly pronounced for o1-mini, highlighting MaP's strength in reinforcing state tracking through explicit intermediate graph outputs. For other operations (**Add Node** and **Remove Node**), MaP remains competitive with CoT. Interestingly, MaP also improves performance on the first modification step, where it should behave similarly to zero-shot prompting. This suggests that the mere instruction to output intermediate graphs boosts model performance, even when the state tracking demand is minimal.

Overall, MaP prompting enhances dynamic graph reasoning by aligning the prompting strategy with the task's structural requirements, especially where fine-grained state tracking is critical.

4.2.4 Ablation: Graph Types & The Preservation of Graph Structure. We further evaluate Claude 3.5 Sonnet and o1-mini, the two strongest models, on **Print Graph** across 3 modification steps of different graph types, including: **1) star graphs**, **2) path graphs**, **3) complete graphs**, and **4) empty graphs**. Each model is tested on 250 instances per graph type.

Figures 4a and 4b highlight the models' varying strengths across these structured graphs. Performance on edge-related tasks is largely consistent with previous results on ER graphs. Notably, both models improve on **Remove Node** for structured graphs compared to ER, suggesting that clearer initial structure aids node removal reasoning. However, o1-mini performs poorly on **Add Node** across all types, with sharp performance drops across modification steps.

To investigate, we analyze o1-mini's errors by tracking how often it connects the newly added node to specific targets: the central node (star), the final node in the path at the bottom row of the adjacency matrix (path), all nodes (complete), or any node (empty). Results in Figure 4c reveal a strong bias: o1-mini frequently connects the new node in a way that **preserves the original graph's structure**, suggesting an **intrinsic tendency of reasoning models to maintain topological patterns, even when incorrect**.

4.2.5 Ablation: Edge Density and Graph Size. We analyze how graph size and edge density affect model performance in **DyGraphQA-Synth** by evaluating graphs with $n \in \{10, 20\}$ nodes and edge densities $p \in \{0.1, 0.5, 0.9\}$. For each configuration, we generate 100 graphs, focusing on the **Print Graph** task, and evaluate with Claude 3.5 Sonnet. Results are show in Figure 5.

For **Add Edge**, the model performs well overall but struggles on low-density graphs when only a single edge is added—likely due to difficulty identifying the correct 0 entry to update. In contrast, **Remove Edge** reveals the opposite pattern: accuracy drops with density, suggesting difficulty identifying the correct 1 entry in dense matrices. For **Add Node**, performance declines with graph size and sparsity, contrasting with Claude 3.5 Sonnet's strong base performance from Section 4.2.2 and indicating new errors—such as dimension mismatches or adding too few/many nodes—discussed in Section C. Finally, **Remove Node** is most sensitive to both size and sparsity, with the worst performance on large and sparse graphs, reflecting the general difficulty of the modification.

5 Conclusion

In this paper, we introduce **DyGraphQA**, a challenging benchmark dataset designed to evaluate LLMs' ability to reason over fully dynamic graphs. DyGraphQA consists of two datasets: **DyGraphQA-Real**, featuring real-world dynamic graphs in natural language, and **DyGraphQA-Synth**, containing synthetic graphs as structured data. Capturing both representations enables a comprehensive assessment of LLMs' reasoning capabilities across modalities. Our results show that SOTA LLMs struggle significantly with fully dynamic graph reasoning, particularly as the number of modifications increases. Extensive ablations show how graph structure, size, edge density, and prompting strategies impact performance. Future work will explore **fine-tuning models on dynamic graph reasoning tasks** and **developing more robust prompting strategies** to further enhance LLMs' ability to process evolving graph structures.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[2] Palaash Agrawal, Shavak Vasania, and Cheston Tan. 2025. Can LLMs Perform Structured Graph Reasoning Tasks?. In *International Conference on Pattern Recognition*. Springer, 287–308.

[3] Rohan Anil, Andrew M Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, et al. 2023. Palm 2 technical report. *arXiv preprint arXiv:2305.10403* (2023).

[4] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609* (2023).

[5] Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, et al. 2024. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 17682–17690.

[6] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL]

[7] Serina Chang, Alicja Chaszczewicz, Emma Wang, Maya Josifovska, Emma Pierson, and Jure Leskovec. 2024. LLMs generate structurally realistic social networks but overestimate political homophily. *arXiv preprint arXiv:2408.16629* (2024).

[8] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2023. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research* 24, 240 (2023), 1–113.

[9] Xinnan Dai, Haohao Qu, Yifen Shen, Bohang Zhang, Qihao Wen, Wenqi Fan, Dongsheng Li, Jiliang Tang, and Caihua Shan. 2024. How Do Large Language Models Understand Graph Patterns? A Benchmark for Graph Pattern Comprehension. *arXiv preprint arXiv:2410.05298* (2024).

[10] Bahare Fatemi, Jonathan Halcrow, and Bryan Perozzi. 2023. Talk like a graph: Encoding graphs for large language models. *arXiv preprint arXiv:2310.04560* (2023).

[11] Yifan Feng, Chengwu Yang, Xingliang Hou, Shaoyi Du, Shihui Ying, Zongze Wu, and Yue Gao. 2024. Beyond Graphs: Can Large Language Models Comprehend Hypergraphs? *arXiv preprint arXiv:2410.10083* (2024).

[12] Hamed Firooz, Maziar Sanjabi, Wenlong Jiang, and Xiaoling Zhai. 2024. Lost-in-distance: Impact of contextual proximity on llm performance in graph tasks. *arXiv preprint arXiv:2410.01985* (2024).

[13] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[14] Jiayan Guo, Lun Du, and Hengyu Liu. 2023. Gpt4graph: Can large language models understand graph structured data? an empirical evaluation and benchmarking. *arXiv preprint arXiv:2305.15066* (2023).

[15] Aric Hagberg, Pieter J Swart, and Daniel A Schult. 2008. *Exploring network structure, dynamics, and function using NetworkX*. Technical Report. Los Alamos National Laboratory (LANL), Los Alamos, NM (United States).

[16] Xiaoxin He, Yijun Tian, Yifei Sun, Nitesh V Chawla, Thomas Laurent, Yann LeCun, Xavier Bresson, and Bryan Hooi. 2024. G-Retriever: Retrieval-Augmented Generation for Textual Graph Understanding and Question Answering. *arXiv preprint arXiv:2402.07630* (2024).

[17] Seyed Mehran Kazemi, Rishab Goel, Kshitij Jain, Ivan Kobyzev, Akshay Sethi, Peter Forsyth, and Pascal Poupart. 2020. Representation learning for dynamic graphs: A survey. *Journal of Machine Learning Research* 21, 70 (2020), 1–73.

[18] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems* 35 (2022), 22199–22213.

[19] Yuhan Li, Peisong Wang, Xiao Zhu, Aochuan Chen, Haiyun Jiang, Deng Cai, Victor W Chan, and Jia Li. 2024. Glbench: A comprehensive benchmark for graph with large language models. *Advances in Neural Information Processing Systems* 37 (2024), 42349–42368.

[20] Elan Markowitz, Krupa Galiya, Greg Ver Steeg, and Aram Galstyan. 2025. KG-LLM-Bench: A Scalable Benchmark for Evaluating LLM Reasoning on Textualized Knowledge Graphs. *arXiv preprint arXiv:2504.07087* (2025).

[21] Team OLMo, Pete Walsh, Luca Soldaini, Dirk Groeneveld, Kyle Lo, Shane Arora, Akshita Bhagia, Yuling Gu, Shengyi Huang, Matt Jordan, et al. 2024. 2 OLMo 2 Furious. *arXiv preprint arXiv:2501.00656* (2024).

[22] Sheng Ouyang, Yulan Hu, Ge Chen, and Yong Liu. 2024. GUNDAM: Aligning Large Language Models with Graph Understanding. *arXiv preprint arXiv:2409.20053* (2024).

[23] Marios Papachristou and Yuan Yuan. 2024. Network formation and dynamics among multi-llms. *arXiv preprint arXiv:2402.10659* (2024).

[24] Bryan Perozzi, Bahare Fatemi, Dustin Zelle, Anton Tsitsulin, Mehran Kazemi, Rami Al-Rfou, and Jonathan Halcrow. 2024. Let Your Graph Do the Talking: Encoding Structured Data for LLMs. *arXiv preprint arXiv:2402.05862* (2024).

[25] Phillip Schneider, Tim Schopf, Juraj Vladika, Mikhail Galkin, Elena Simperl, and Florian Matthes. 2022. A Decade of Knowledge Graphs in Natural Language Processing: A Survey. arXiv:2210.00105 [cs.CL]

[26] Jianheng Tang, Qifan Zhang, Yuhan Li, Nuo Chen, and Jia Li. 2025. Grapharena: Evaluating and exploring large language models on graph computation. In *The Thirteenth International Conference on Learning Representations*.

[27] Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805* (2023).

[28] Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, et al. 2024. Gemma 2: Improving open language models at a practical size. *arXiv preprint arXiv:2408.00118* (2024).

[29] Rakshit Trivedi, Hanjun Dai, Yichen Wang, and Le Song. 2017. Know-evolve: Deep temporal reasoning for dynamic knowledge graphs. In *international conference on machine learning*. PMLR, 3462–3471.

[30] Heng Wang, Shangbin Feng, Tianxing He, Zhaoxuan Tan, Xiaochuang Han, and Yulia Tsvetkov. 2024. Can language models solve graph problems in natural language? *Advances in Neural Information Processing Systems* 36 (2024).

[31] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171* (2022).

[32] Yanbang Wang, Hejie Cui, and Jon Kleinberg. 2024. Microstructures and Accuracy of Graph Recall by Large Language Models. *arXiv preprint arXiv:2402.11821* (2024).

[33] Yuxiang Wang, Xinnan Dai, Wenqi Fan, and Yao Ma. 2025. Exploring Graph Tasks with Pure LLMs: A Comprehensive Benchmark and Investigation. *arXiv preprint arXiv:2502.18771* (2025).

[34] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[35] Jason Weston, Antoine Bordes, Sumit Chopra, Alexander M Rush, Bart Van Merriënboer, Armand Joulin, and Tomas Mikolov. 2015. Towards ai-complete question answering: A set of prerequisite toy tasks. *arXiv preprint arXiv:1502.05698* (2015).

[36] Colin White, Samuel Dooley, Manley Roberts, Arka Pal, Ben Feuer, Siddhartha Jain, Ravid Shwartz-Ziv, Neel Jain, Khalid Saifullah, Siddartha Naidu, Chinmay Hegde, Yann LeCun, Tom Goldstein, Willie Neiswanger, and Micah Goldblum. 2024. LiveBench: A Challenging, Contamination-Free LLM Benchmark. arXiv:2406.19314 [cs.CL] https://arxiv.org/abs/2406.19314

[37] Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W Cohen, Ruslan Salakhutdinov, and Christopher D Manning. 2018. HotpotQA: A dataset for diverse, explainable multi-hop question answering. *arXiv preprint arXiv:1809.09600* (2018).

[38] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems* 36 (2023), 11809–11822.

[39] Yang Yao, Xin Wang, Zeyang Zhang, Yijian Qin, Ziwei Zhang, Xu Chu, Yuekui Yang, Wenwu Zhu, and Hong Mei. 2024. Exploring the Potential of Large Language Models in Graph Generation. *arXiv preprint arXiv:2403.14358* (2024).

[40] Yizhuo Zhang, Heng Wang, Shangbin Feng, Zhaoxuan Tan, Xiaochuang Han, Tianxing He, and Yulia Tsvetkov. 2024. Can LLM Graph Reasoning Generalize beyond Pattern Memorization? *arXiv preprint arXiv:2406.15992* (2024).

[41] Zeyang Zhang, Xin Wang, Ziwei Zhang, Haoyang Li, Yijian Qin, Simin Wu, and Wenwu Zhu. 2023. LLM4DyG: Can Large Language Models Solve Problems on Dynamic Graphs? *arXiv preprint arXiv:2310.17110* (2023).

[42] Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, et al. 2022. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625* (2022).

A Dataset Generation Algorithm

In this section, we provide the pseudocode for the algorithms necessary for generating the DyGraphQA-Synth dataset. The algorithm assumes fixed graph encoding and question rephrasing functions. For each of the 250 initial graphs, the algorithm performs five modification rounds. In each round, five different modification types are applied, generating five modified graphs. Each of these is paired with five final questions, resulting in $5 \times 5 \times 5 = 125$ examples per graph. Accounting for five question rephrasings, the total dataset size is: 250 graphs $\times$ 125 examples per graph $\times$ 5 rephrasings = **156,250** examples. More details can be found at our code repository.[3]

During evaluation, we found that all LLMs performed poorly on the **Remove Node** modification until the instruction *"and renumber the nodes accordingly"* was added to *"Remove node v from the graph"* in Algorithm 4. This highlights the importance of providing explicit instructions when tasks depend on implicit node indexing and renumbering.

[3]https://anonymous.4open.science/r/DyGraphQA-7394/

Algorithm 1 AddEdge

Require: Graph G
Ensure: Modified Graph G'

$G' \leftarrow G$
$(i, j) \sim \mathcal{U}(V_{G'} \times V_{G'} \setminus E_{G'})$
$E_{G'} \leftarrow E_{G'} \cup \{(i, j)\}$
return G', *"Add an edge between nodes i and j."*

Algorithm 2 RemoveEdge

Require: Graph G
Ensure: Modified Graph G'

$G' \leftarrow G$
$(i, j) \sim \mathcal{U}(E_{G'})$
$E_{G'} \leftarrow E_{G'} \setminus \{(i, j)\}$
return G', *"Remove the edge between nodes i and j."*

Algorithm 3 AddNode

Require: Graph G
Ensure: Modified Graph G'

$G' \leftarrow G$
$V_{G'} \leftarrow V_{G'} \cup \{v\}, E_{G'} \leftarrow E_{G'}$
return G', *"Add a node v to the graph."*

Algorithm 4 RemoveNode

Require: Graph G
Ensure: Modified Graph G'

$G' \leftarrow G$
$v \sim \mathcal{U}(V_{G'})$
$V_{G'} \leftarrow V_{G'} \setminus \{v\}, E_{G'} \leftarrow E_{G'} \setminus \{(v, u) \mid u \in V_{G'}\}$
return G', *"Remove node v from the graph."*

Algorithm 5 Mix

Require: Graph G
Ensure: Modified Graph G'

$G' \leftarrow G$
$\mathrm{M} \sim \mathcal{U}(\{\textsc{AddEdge}, \textsc{RemoveEdge}, \textsc{AddNode}, \textsc{RemoveNode}\})$
return $\mathrm{M}(G')$

Algorithm 6 ConstructDyGraphQA-Synth

Require: Number of graphs to generate N
Ensure: Dataset D containing multi-step tasks for all final queries and k values

Initialize an empty dataset D
Define the set of possible final questions $\mathcal{Q} = \{\text{Node Count}, \text{Edge Count}, \text{Node Degree}, \text{Connected Nodes}, \text{Print Graph}\}$
Define the maximum number of modifications $k_{max} = 5$
Define V_G as the set of nodes in any graph G, and E_G as the set of edges in any graph G
for $i = 1$ to N **do**
  Sample $n \sim \mathcal{U}(7, 20)$
  Generate an undirected Erdős-Rényi graph $G = (V, E)$ with $|V| = n$ and sample edge probability $p \sim \mathcal{U}(0, 1)$
  Initialize graphs $G_{AE}, G_{RE}, G_{AN}, G_{RN}, G_{MX} \leftarrow G$
  Initialize $M_{AE}, M_{RE}, M_{AN}, M_{RN}, M_{MX} \leftarrow []$
  for $k = 1$ to k_{max} **do**
    $G_{AE}, m_{AE} \leftarrow \textsc{AddEdge}(G_{AE})$ 1
    $M_{AE} \leftarrow M_{AE} \parallel m_{AE}$
    $G_{RE}, m_{RE} \leftarrow \textsc{RemoveEdge}(G_{RE})$ 2
    $M_{RE} \leftarrow M_{RE} \parallel m_{RE}$
    $G_{AN}, m_{AN} \leftarrow \textsc{AddNode}(G_{AN})$ 3
    $M_{AN} \leftarrow M_{AN} \parallel m_{AN}$
    $G_{RN}, m_{RN} \leftarrow \textsc{RemoveNode}(G_{RN})$ 4
    $M_{RN} \leftarrow M_{RN} \parallel m_{RN}$
    $G_{MX}, m_{MX} \leftarrow \textsc{Mix}(G_{MX})$ 5
    $M_{MX} \leftarrow M_{MX} \parallel m_{MX}$
    $Mods = \{(G_{AE}, M_{AE}), (G_{RE}, M_{RE}), (G_{AN}, M_{AN}), (G_{RN}, M_{RN}), (G_{MX}, M_{MX})\}$
    for $Q \in \mathcal{Q}$ **do**
      for $(G_{Mod}, M_{Mod}) \in Mods$ **do**
        if $Q =$ Node Count **then**
          $S \leftarrow |V_{G_{Mod}}|$
        else if $Q =$ Edge Count **then**
          $S \leftarrow |E_{G_{Mod}}|$
        else if $Q =$ Node Degree **then**
          $v \sim \mathcal{U}(V_{G_{Mod}})$
          $S \leftarrow |\{u \in V_{G_{Mod}} \mid (v, u) \in E_{G_{Mod}}\}|$
        else if $Q =$ Connected Nodes **then**
          $v \sim \mathcal{U}(V_{G_{Mod}})$
          $S \leftarrow \{u \in V_{G_{Mod}} \mid (v, u) \in E_{G_{Mod}}\}$
        else if $Q =$ Print Graph **then**
          $S \leftarrow G_{Mod}$
        end if
        $D \leftarrow D \cup (G, M_{Mod}, Q, S)$
      end for
    end for
  end for
end for
return D

B Results on Varying Final Questions and Graph Encoders

Within DyGraphQA-Synth, in addition to the **Print Graph** question, we evaluated model performance on other final questions, including **Node Count**, **Edge Count**, **Node Degree**, and **Connected Nodes**. Detailed results for these tasks are provided in Figure 6 respectively. Our analysis reveals that models consistently perform poorly on the **Print Graph** task when compared to other graph property tasks. This finding is significant because it illustrates the challenges with maintaining the modified structure, as outputting the entire adjacency matrix requires carefully managing the structured data it contains.

B.1 Node Count

o1-mini demonstrates slight drops in performance on all modification types compared to Claude 3.5 Sonnet and Llama 3.1 405B. This observation follows from Table 3, which also indicates that even in the static case, o1-mini lags slightly behind both Claude 3.5 Sonnet and Llama 3.1 405B on counting the number of nodes in an adjacency matrix.

B.2 Edge Count

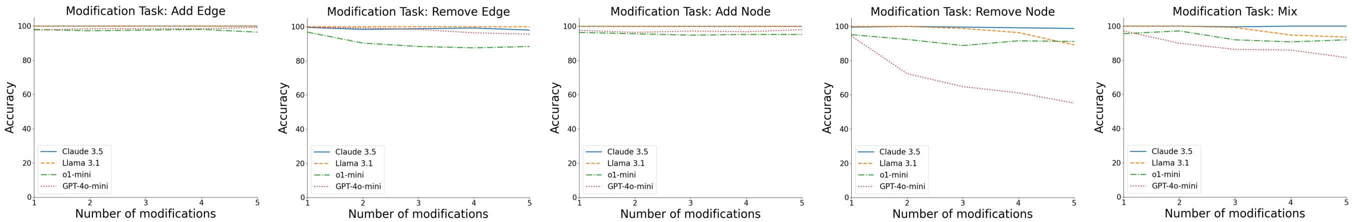
o1-mini consistently outperforms all other models, aligning with the trends observed in Table 2.

B.3 Node Degree

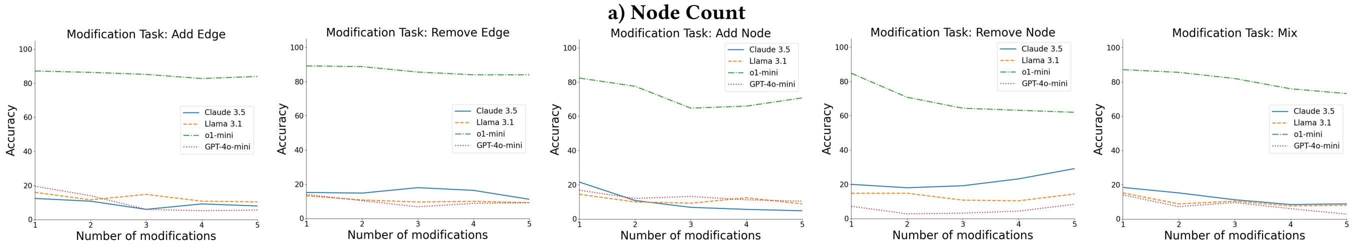
o1-mini again outperforms others for all modification types except Add Node. Models show significant drops in performance on the Remove Node operation, an inherently more error-prone operation due to the renumbering and recalibration of indices. Interestingly, Claude 3.5 Sonnet's performance increases slightly on the Add Node modification as the number of modifications increase.

B.4 Connected Nodes

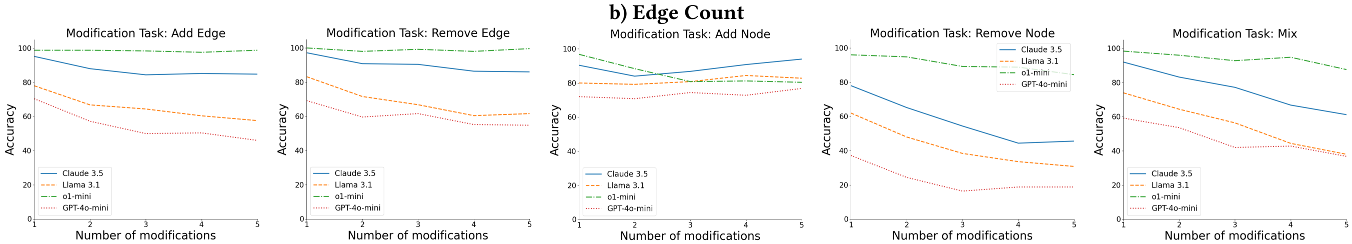
The Connected Nodes task mirrors the patterns found in Node Degree. o1-mini outperforms all other models. As with Node Degree, the Remove Node modification introduces the most notable performance drop for all models. Llama 3.1 405B shows slight improvement in accuracy for Add Node modifications as the number of modifications increases.



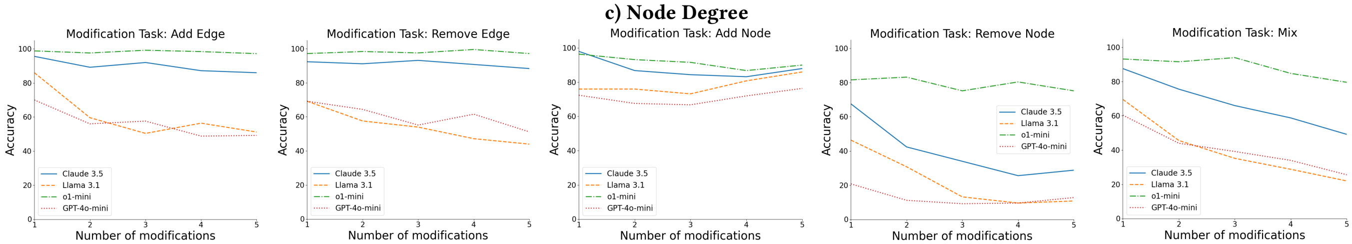
a) Node Count



b) Edge Count



c) Node Degree



d) Connected Nodes

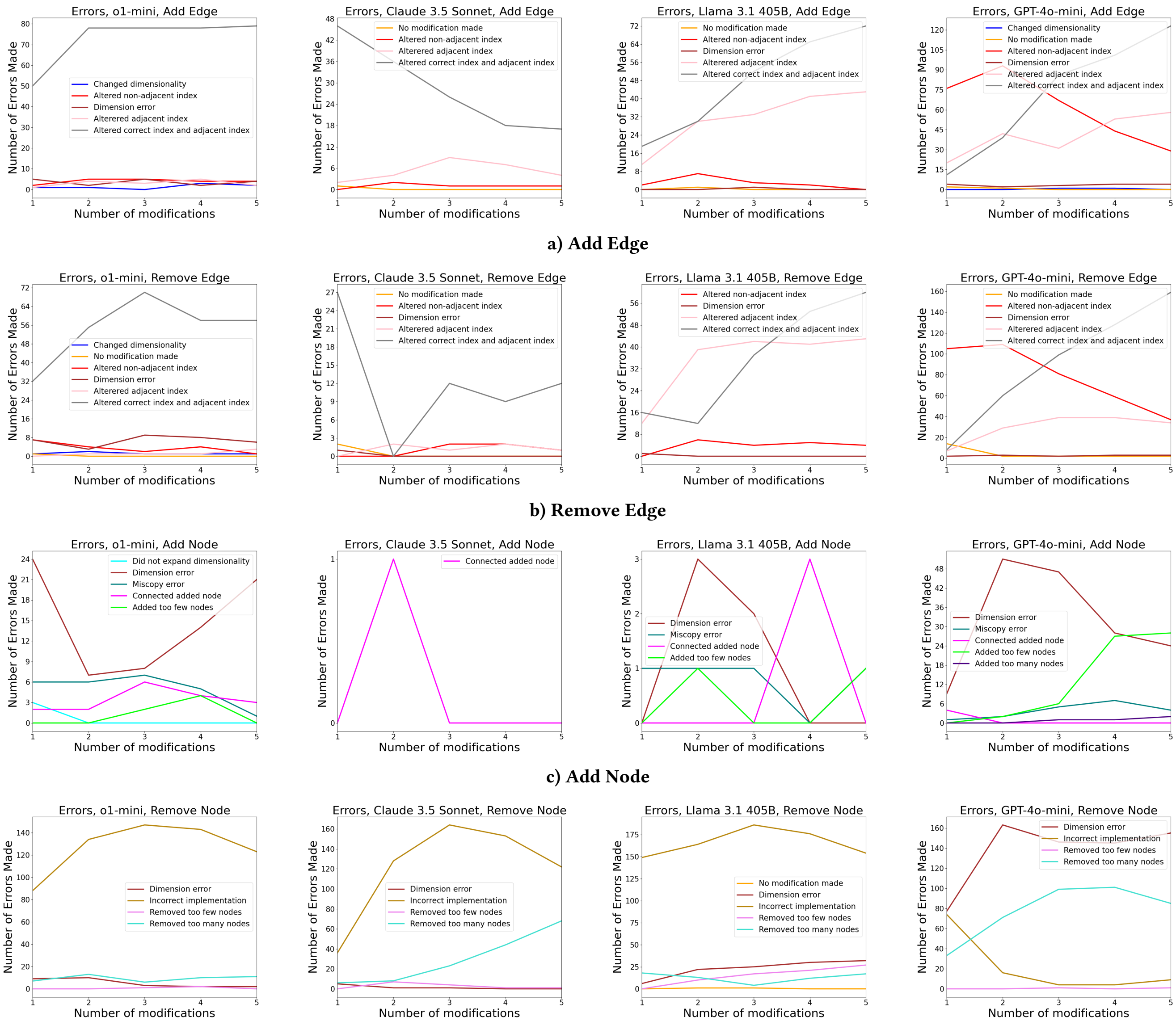
Figure 6: Performance of models on the following Final Questions: a) Node Count, b) Edge Count, Node Degree, and Connected Nodes.

C Error Analysis

We analyze error types and frequencies for all baseline models on the **Print Graph** task, illuminating where LLMs falter in dynamic graph reasoning on structured data.

For **Add Edge**, Figure 7a shows the different types of errors models make. We observe the following error types:

- **Altered correct index and adjacent index**: The model correctly modifies the target index but also adds an edge to an adjacent one. This is the most frequent error for all models. Both Llama 3.1 405B and GPT-4o mini exhibit an increase in this error type as the number of modifications grows, indicating potential scaling issues. For both o1-mini and Claude 3.5 Sonnet, this error overwhelmingly dominates their performance, as they both make few other types of errors. Claude 3.5 Sonnet reduces this error with more modifications, aligning with improved performance in Figure 2.
- **Altered adjacent index**: The model modifies only an adjacent (incorrect) index. This error remains low and stable for o1-mini and Claude 3.5 Sonnet, whereas it becomes more common for Llama 3.1 405B and GPT-4o mini with an increasing number of modifications.
- **Altered non-adjacent index**: A rare error for most models, where a non-relevant index is modified. The error is more prominent in GPT-4o mini, suggesting that this error decreases with larger model sizes and improved reasoning capabilities. Interestingly, GPT-4o mini makes this error less often as the number of modifications increases. As shown by Figure 2, GPT-4o mini's performance on the **Add Edge** modification still decreases across the number of modifications, suggesting that the model's edits become increasingly closer to the correct indices as the complexity of the problem increases.
- **No modification made**: The model outputs the original matrix unchanged. Rare overall and absent in o1-mini.



a) Add Edge

b) Remove Edge

c) Add Node

d) Remove Node

Figure 7: Errors made by models on the Print Graph task, with the following modifications: a) Add Edge, b) Remove Edge, c) Add Node, d) Remove Node.

- **Dimension error**: The output is not a valid matrix—typically due to inconsistent row lengths, never occuring for Claude 3.5 Sonnet.
- **Changed dimensionality**: The model outputs a matrix of incorrect dimensions. Seen occasionally in o1-mini and GPT-4o-mini, and never in Claude 3.5 Sonnet and Llama 3.1 405B.

For **Remove Edge**, Figure 7b shows that the error distribution closely mirrors that of **Add Edge**:

- **Altered correct and adjacent index**: This remains the most frequent error across models, dominating overall error rates. Both Llama 3.1 405B and GPT-4o mini exhibit an increase in this error as the number of modifications grows, reflecting a recurring challenge with hallucinating adjacent edges.
- **Previously defined errors**: As in **Add Edge**, the following errors follow similar trends—**Altered adjacent index**, **Altered non-adjacent index**, **No modification**, **Dimension error**, and **Changed dimensionality** (only seen occasionally in o1-mini).

For **Add Node**, Figure 7c highlights the strong performance of Claude 3.5 Sonnet and Llama 3.1 405B:

- **Connected added node**: The newly added node is incorrectly connected to existing nodes. Rare overall, with Claude 3.5 Sonnet making this error once; o1-mini shows a higher rate, consistent with its bias toward preserving graph structure (Section 4.2.4).

- **Dimension error**: Occurs frequently for o1-mini and GPT-4o-mini. Interestingly, dimension errors are less common at extreme values of k for o1-mini, with GPT-4o-mini showing the opposite trend.
- **Other rare errors**: **Miscopy errors** (modifying existing edges), **Added too few nodes**, and **Added too many nodes** (occurring only for GPT-4o-mini).

For **Remove Node**, Figure 7d shows error types for the most challenging modification in DyGraphQA-Synth:

- **Removed too many nodes**: This error arises when the model removes more than the required k nodes. It is less frequent in o1-mini and Llama 3.1 405B but occurs at a high frequency in GPT-4o mini and Claude 3.5 Sonnet, with Claude 3.5 Sonnet exhibiting an increase in this error as k grows.
- **Removed too few nodes**: This error occurs when the model removes fewer than k nodes. It is generally infrequent, though Llama 3.1 405B makes this error slightly more often than the other models.
- **No modification made**: Only Llama 3.1 405B produces this error, and produces it very rarely.
- **Dimension error**: This error is made by Claude 3.5 Sonnet and o1-mini, while Llama 3.1 405B produces it slightly more often. However, this is the most frequent error for GPT-4o mini, indicating that it struggles significantly with maintaining a valid matrix structure and returning a mathematically well-defined object.
- **Incorrect implementation**: This error occurs when the model removes the correct number of rows and columns, but the internal connections of the resulting matrix deviate substantially from the solution matrix. This is the most common error type across all models except GPT-4o mini, which struggles to produce valid outputs at all due to frequent dimension errors. o1-mini makes this error less frequently than Claude 3.5 Sonnet and Llama 3.1 405B, contributing to its superior overall performance on the Remove Node modification as seen in Figure 2.

D Illustrating Prompts and Model Outputs

In this section, we show example input prompts and model outputs from the three in-context methods we evaluate on DyGraphQA-Synth. We evaluated Claude 3.5 Sonnet on a single example, involving three **Add Edge** modifications followed by a **Print Graph** task. Figure 8 show the input prompt and model output using zero-shot prompting, Figure 9 show the input prompt and model output using CoT prompting with one example, and Figure 10 show the input prompt and model output using MaP prompting

Zero-shot Prompt

```
The following matrix represents the adjacency
matrix of an undirected graph, where the first
row corresponds to node 0, the second row
corresponds to node 1, and so on:

[[0 1 0 0 0 0 0 0 0 1 0 0 0 0 0]
 [1 0 1 1 0 0 1 0 1 0 1 0 0 0 0]
 [0 1 0 0 0 0 0 0 0 0 1 0 1 0 0]
 [0 1 0 0 0 0 0 0 0 0 0 0 1 0 0]
 [0 0 0 0 0 1 0 0 0 0 0 0 0 1 0]
 [0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 1]
 [0 1 0 0 0 0 0 0 1 0 0 1 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 1 1 1]
 [0 1 0 0 0 0 1 0 0 1 0 0 0 0 0]
 [1 0 0 0 0 1 0 0 1 0 0 0 0 0 1]
 [0 1 1 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 1 0 0 0 0 0 0 0 1]
 [0 0 1 1 0 0 0 1 0 0 0 0 0 0 0]
 [0 0 0 0 1 0 0 1 0 0 0 0 0 0 0]
 [0 0 0 0 0 1 0 1 0 1 0 1 0 0 0]]

Perform the following operations on the graph:
1: Add an edge between node 14 and node 8.
2: Add an edge between node 8 and node 3 in
the resulting graph of operation 1.
3: Add an edge between node 2 and node 8 in
the resulting graph of operation 2.
Q: What is the final resulting adjacency
matrix?
A:
```

LLM Output on Zero-shot Prompt

```
To answer this question, I'll start with the
given adjacency matrix and perform the
requested operations one by one. After each
operation, I'll update the matrix.
Here's the process:

1. Add an edge between node 14 and node 8:
   This adds a 1 in position (14,8) and (8,14)
   of the matrix.

2. Add an edge between node 8 and node 3:
   This adds a 1 in position (8,3) and (3,8)
   of the matrix.

3. Add an edge between node 2 and node 8:
   This adds a 1 in position (2,8) and (8,2)
   of the matrix.

After performing these operations, the final
resulting adjacency matrix is:

[[0 1 0 0 0 0 0 0 0 1 0 0 0 0 0]
 [1 0 1 1 0 0 1 0 1 0 1 0 0 0 0]
 [0 1 0 0 0 0 0 1 0 0 1 0 1 0 0]
 [0 1 0 0 0 0 0 1 0 0 0 0 1 0 0]
 [0 0 0 0 0 1 0 0 0 0 0 0 0 1 0]
 [0 0 0 0 1 0 0 0 0 1 0 0 0 0 1]
 [0 1 0 0 0 0 0 0 1 0 0 1 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 1 1 1]
 [0 1 1 1 0 0 1 0 0 1 0 0 0 0 1]
 [1 0 0 0 0 1 0 0 1 0 0 0 0 0 1]
 [0 1 1 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 1 0 0 0 0 0 0 0 1]
 [0 0 1 1 0 0 0 1 0 0 0 0 0 0 0]
 [0 0 0 0 1 0 0 1 0 0 0 0 0 0 0]
 [0 0 0 0 0 1 0 1 1 1 0 1 0 0 0]]

This matrix represents the final state of the
graph after all the requested operations
have been performed.
Incorrect!
```

Figure 8: (a) An example zero-shot prompt. The prompt consists of three Add Edge modifications, followed by the Print Graph task. (b) The output of Claude 3.5 Sonnet on the example prompt. Here, the model answered the question incorrectly, as it modified positions (2,7) and (3,7) instead of (2,8) and (3,8), despite modifying positions (8,2) and (8,3) correctly. Correctly modified indices are highlighted in green, while incorrectly modified indices are highlighted in red.

CoT Prompt with one example

```
The following matrix represents the adjacency
matrix of an undirected graph, where the first
row corresponds to node 0, the second row
corresponds to node 1, and so on:

[[0 1 1 0 0 0 1 0 1 0 1 1 0 0 0 0]
 [1 0 1 0 0 1 0 0 1 0 0 1 0 1 0 0]
 ...
 [0 0 0 0 0 1 0 1 0 0 1 0 1 0 0 0]
 [0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0]]

Perform the following operations on the graph:
1: Add an edge between node 15 and node 13.
2: Add an edge between node 15 and node 3 in
the resulting graph of operation 1.
3: Add an edge between node 10 and node 7 in
the resulting graph of operation 2.
Q: What is the final resulting adjacency
matrix?
A: After changing both entries [15, 13] and
[13, 15] from 0 to 1, both entries [15, 3] and
[3, 15] from 0 to 1, and both entries [10, 7]
and [7, 10] from 0 to 1, the resulting
adjacency matrix is as defined below:

[[0 1 1 0 0 0 1 0 1 0 1 1 0 0 0 0]
 [1 0 1 0 0 1 0 0 1 0 0 1 0 1 0 0]
 ...
 [0 0 0 0 0 1 0 1 0 0 1 0 1 0 0 0]
 [0 0 1 1 1 0 1 0 0 0 0 0 0 1 0 0]]

The following matrix represents the adjacency
matrix of an undirected graph, where the first
row corresponds to node 0, the second row
corresponds to node 1, and so on:
...
Q: What is the final resulting adjacency
matrix?
A:
```

LLM Output on CoT Prompt

```
To solve this problem, we'll perform the
operations one by one and update the adjacency
matrix accordingly. Remember that for an
undirected graph, when we add an edge between
two nodes, we need to update both corresponding
entries in the matrix.

Let's perform the operations:

1. Add an edge between node 14 and node 8:
   Update [14][8] and [8][14] from 0 to 1

2. Add an edge between node 8 and node 3:
   Update [8][3] and [3][8] from 0 to 1

3. Add an edge between node 2 and node 8:
   Update [2][8] and [8][2] from 0 to 1

After applying these changes, the final
resulting adjacency matrix is:

[[0 1 0 0 0 0 0 0 0 1 0 0 0 0 0]
 [1 0 1 1 0 0 1 0 1 0 1 0 0 0 0]
 [0 1 0 0 0 0 0 0 1 0 1 0 1 0 0]
 [0 1 0 0 0 0 0 0 1 0 0 0 1 0 0]
 [0 0 0 0 0 1 0 0 0 0 0 0 0 1 0]
 [0 0 0 0 1 0 0 0 0 1 0 0 0 0 1]
 [0 1 0 0 0 0 0 0 1 0 0 1 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 1 1 1]
 [0 1 1 1 0 0 1 0 0 1 0 0 0 0 1]
 [1 0 0 0 0 1 0 0 1 0 0 0 0 0 1]
 [0 1 1 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 1 0 0 0 0 0 0 0 1]
 [0 0 1 1 0 0 0 1 0 0 0 0 0 0 0]
 [0 0 0 0 1 0 0 1 0 0 0 0 0 0 0]
 [0 0 0 0 0 1 0 1 1 1 0 1 0 0 0]]

This is the final resulting adjacency matrix
after performing all the specified operations.
Correct!
```

Figure 9: (a) An example CoT prompt with one example, created by adding the example before the prompt from Figure 8(a) We truncate all matrices in the prompt for space reasons. (b) The output of Claude 3.5 Sonnet on the CoT prompt from Figure 9(a). Here, the model answered the correctly, and the correctly modified indicies are highlighted in green in the final matrix.

MaP Prompt

```
The following matrix represents the adjacency
matrix of an undirected graph, where the first
row corresponds to node 0, the second row
corresponds to node 1, and so on:

[[0 1 0 0 0 0 0 0 0 1 0 0 0 0 0]
 [1 0 1 1 0 0 1 0 1 0 1 0 0 0 0]
 [0 1 0 0 0 0 0 0 0 0 1 0 1 0 0]
 [0 1 0 0 0 0 0 0 0 0 0 0 1 0 0]
 [0 0 0 0 0 1 0 0 0 0 0 0 0 1 0]
 [0 0 0 0 1 0 0 0 0 1 0 0 0 0 1]
 [0 1 0 0 0 0 0 0 1 0 0 1 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 1 1 1]
 [0 1 0 0 0 0 1 0 0 1 0 0 0 0 0]
 [1 0 0 0 0 1 0 0 1 0 0 0 0 0 1]
 [0 1 1 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 1 0 0 0 0 0 0 0 1]
 [0 0 1 1 0 0 0 1 0 0 0 0 0 0 0]
 [0 0 0 0 1 0 0 1 0 0 0 0 0 0 0]
 [0 0 0 0 0 1 0 1 0 1 0 1 0 0 0]]

Perform the following operations on the graph:
1: Add an edge between node 14 and node 8.
2: Add an edge between node 8 and node 3 in
the resulting graph of operation 1.
3: Add an edge between node 2 and node 8 in
the resulting graph of operation 2.
Q: What is the final resulting adjacency
matrix? For each operation, write out the
entire resulting adjacency matrix.
A:
```

LLM Output on MaP Prompt

```
I'll perform the operations step by step and
provide the resulting adjacency matrix
after each operation.

Operation 1: Add an edge between node 14 and
node 8

[[0 1 0 0 0 0 0 0 0 1 0 0 0 0 0]
[1 0 1 1 0 0 1 0 1 0 1 0 0 0 0]
...
[0 0 0 0 1 0 0 1 0 0 0 0 0 0 0]
[0 0 0 0 0 1 0 1 1 1 0 1 0 0 0]]

Operation 2: Add an edge between node 8 and
node 3

[[0 1 0 0 0 0 0 0 0 1 0 0 0 0 0]
 [1 0 1 1 0 0 1 0 1 0 1 0 0 0 0]
 [0 1 0 0 0 0 0 0 0 0 1 0 1 0 0]
 [0 1 0 0 0 0 0 0 1 0 0 0 1 0 0]
 ...
 [0 0 1 1 0 0 0 1 0 0 0 0 0 0 0]
 [0 0 0 0 1 0 0 1 0 0 0 0 0 0 0]
 [0 0 0 0 0 1 0 1 1 1 0 1 0 0 0]]

Operation 3: Add an edge between node 2 and
node 8

Here's the final resulting adjacency matrix:

[[0 1 0 0 0 0 0 0 0 1 0 0 0 0 0]
 [1 0 1 1 0 0 1 0 1 0 1 0 0 0 0]
 [0 1 0 0 0 0 0 0 1 0 1 0 1 0 0]
 [0 1 0 0 0 0 0 0 1 0 0 0 1 0 0]
 [0 0 0 0 0 1 0 0 0 0 0 0 0 1 0]
 [0 0 0 0 1 0 0 0 0 1 0 0 0 0 1]
 [0 1 0 0 0 0 0 0 1 0 0 1 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 1 1 1]
 [0 1 1 1 0 0 1 0 0 1 0 0 0 0 1]
 [1 0 0 0 0 1 0 0 1 0 0 0 0 0 1]
 [0 1 1 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 1 0 0 0 0 0 0 0 1]
 [0 0 1 1 0 0 0 1 0 0 0 0 0 0 0]
 [0 0 0 0 1 0 0 1 0 0 0 0 0 0 0]
 [0 0 0 0 0 1 0 1 1 1 0 1 0 0 0]]

Correct!
```

Figure 10: (a) An example MaP prompt, created by appending the phrase *For each operation, write out the entire resulting adjacency matrix.* to the prompt from Figure 8(a). (b) The output of Claude 3.5 Sonnet on the example MaP prompt from Figure 10(a). The first and second intermediate matrices are truncated in order to fit the entire output on the page. Here, the model answered the question correctly, and the correctly modified indicies are highlighted in green in the final matrix.

Table 2: Accuracy percentages (mean ± s.d) on graph property questions from [10].

Graph Encoder	Final Task	GPT-4o mini	Llama 3.1	Claude 3.5	o1-mini
Adjacency List	Node Count	86.6	89.8	95.4	99.8
	Edge Count	30.4	48.8	54.8	93.6
	Node Degree	95.2	100.0	100.0	98.4
	Edge Existence	71.8	70.6	86.8	66.0
	Connected Nodes	97.8	100.0	100.0	98.2
	Cycle	90.4	91.0	95.0	99.0
	Average	**78.7 ± 23.2**	**83.4 ± 18.3**	**88.7 ± 15.8**	**92.5 ± 12.02**
Incident	Node Count	100.0	99.8	100.0	100.0
	Edge Count	30.0	60.4	76.2	99.0
	Node Degree	99.2	99.2	100.0	99.6
	Edge Existence	95.2	91.0	99.8	66.6
	Connected Nodes	99.8	100.0	100.0	100.0
	Cycle	86.2	87.4	88.4	98.8
	Average	**85.1 ± 25.1**	**91.3 ± 14.2**	**94.1 ± 9.0**	**94.0 ± 12.3**
Friendship	Node Count	99.6	98.8	100.0	100.0
	Edge Count	27.6	49.2	57.0	86.8
	Node Degree	91.6	98.2	100.0	98.0
	Edge Existence	73.0	76.0	77.4	66.0
	Connected Nodes	87.8	93.4	95.2	92.6
	Cycle	91.6	91.8	95.6	99.8
	Average	**78.5 ± 24.1**	**84.6 ± 17.5**	**87.5 ± 15.7**	**90.5 ± 11.9**
Coauthorship	Node Count	99.0	99.0	95.6	100.0
	Edge Count	27.4	42.8	54.2	78.2
	Node Degree	88.0	94.0	99.6	96.4
	Edge Existence	85.6	84.2	88.6	65.0
	Connected Nodes	75.2	91.6	98.2	93.4
	Cycle	92.4	95.6	100.0	99.4
	Average	**77.9±23.7**	**84.5±19.2**	**89.4±16.2**	**88.7 ± 12.9**
Expert	Node Count	87.4	82.8	79.2	99.4
	Edge Count	35.2	52.2	62.8	95.0
	Node Degree	95.8	99.8	100.0	99.4
	Edge Existence	67.0	66.8	100.0	65.0
	Connected Nodes	97.4	97.4	95.2	89.4
	Cycle	86.2	85.8	96.0	98.0
	Average	**78.2±21.6**	**80.8±16.7**	**88.9±13.6**	**91.0 ± 12.1**
Social Network	Node Count	99.6	99.4	100.0	100.0
	Edge Count	26.4	48.0	57.8	81.8
	Node Degree	94.0	97.4	100.0	97.2
	Edge Existence	86.6	85.2	100.0	64.2
	Connected Nodes	85.4	92.8	94.8	93.4
	Cycle	91.8	90.4	93.6	98.6
	Average	**80.6±24.7**	**85.5±17.4**	**91.0±15.1**	**89.2 ± 12.7**
Politician	Node Count	99.4	100	99.6	100.0
	Edge Count	25.2	48.2	55.4	85.8
	Node Degree	94.0	97.0	99.8	98.6
	Edge Existence	88.8	81.6	71.0	66.0
	Connected Nodes	79.6	79.4	100.0	97.2
	Cycle	91.4	89.0	95.8	99.4
	Average	**79.7±25.1**	**82.5±17.1**	**86.9±17.4**	**91.2 ± 12.2**
GoT	Node Count	100.0	100.0	99.0	100.0
	Edge Count	26.8	46.0	57.4	84.8
	Node Degree	93.2	95.2	100.0	96.8
	Edge Existence	83.4	80.4	87.4	65.2
	Connected Nodes	68.4	95.8	100.0	94.6
	Cycle	91.4	95.6	94.8	100.0
	Average	**77.2±24.6**	**85.5±18.7**	**89.8±15.1**	**90.2 ± 12.3**
SP	Node Count	99.4	99.8	99.2	100.0
	Edge Count	26.0	44.4	59.2	86.0
	Node Degree	94.4	96.4	100.0	98.2
	Edge Existence	85.2	87.0	82.2	65.2
	Connected Nodes	74.2	98.6	100.0	98.0
	Cycle	91.4	93.0	95.0	99.6
	Average	**78.4±24.8**	**86.5±19.3**	**89.3±14.8**	**91.2 ± 12.6**

Table 3: Accuracy percentages (mean ± s.d) on graph property questions from [10] for the adjacency matrix encoder. As this work was being conducted, the PaLM API was deprecated, and fortunately we were able to evaluate PaLM 2 L on the adjacency matrix encoder before this.

Graph Encoder	Final Task	PaLM 2 L	GPT-4o mini	Llama 3.1	Claude 3.5	o1-mini
Adjacency Matrix	Node Count	55.4	98.4	100.0	100.0	98.4
	Edge Count	6.4	28.0	44.8	38.6	91.2
	Node Degree	28.6	73.4	88.6	98.6	99.2
	Edge Existence	70.3	85.0	93.8	99.2	68.2
	Connected Nodes	8.4	84.8	98.2	99.0	98.8
	Cycle	49.6	87.8	87.6	92.8	100.0
	Average	**36.5 ± 23.9**	**76.2 ± 22.8**	**85.5 ± 18.8**	**88.0 ± 22.2**	**92.6 ± 11.3**