Tool Graph Retriever: Exploring Dependency Graph-based Tool Retrieval for Large Language Models

Anonymous ACL submission

Abstract

With the remarkable advancement of AI agents, the number of their equipped tools is increasing rapidly. However, integrating all tool information into the limited model context becomes impractical, highlighting the need for efficient tool retrieval methods. In this regard, dominant methods primarily rely on semantic similarities between tool descriptions and user queries to retrieve relevant tools. However, they often consider each tool independently, overlooking dependencies between tools, which may lead to the omission of prerequisite tools for successful task execution. To deal with this defect, in this paper, we propose **T**ool **G**raph **R**etriever (TGR), which exploits the dependencies among tools to learn better tool representations for retrieval. First, we construct a dataset termed TDI300K to train a discriminator for identifying tool dependencies. Then, we represent all candidate tools as a tool dependency graph and use graph convolution to integrate the dependencies into their representations. Finally, these updated tool representations are employed for online retrieval. Experimental results on several commonly used datasets show that our TGR can bring a performance improvement to existing dominant methods, achieving SOTA performance. Moreover, in-depth analyses also verify the importance of tool dependencies and the effectiveness of our TGR. [1]

1 Introduction

As an important step towards artificial general intelligence (AGI), tool learning expands the ability of LLM-based AI agents and enables them to interact with the external environment. (Goertzel, 2014; Dou et al., 2023; McLean et al., 2023). However, as the number of equipped tools increases rapidly, it has become challenging for LLMs to process all the tool information, primarily due to

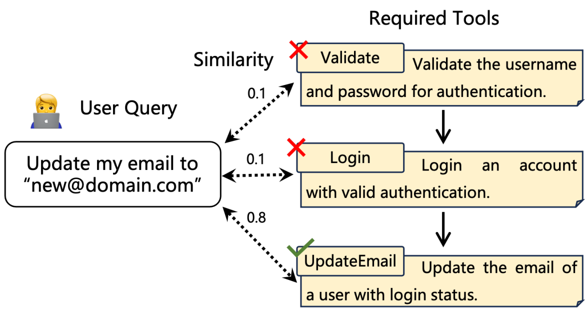

Figure 1: An example of dominant tool retrieval process, where some necessary prerequisite tools are omitted due to low semantic similarities. The down arrows ↓ denote the calling order of the tools.

the context length limitations. Therefore, a typical framework of AI agents employs a retriever to retrieve the candidate tools before the practical task, which involves the following four steps. First of all, relevant tools are retrieved from the equipped tool set according to the task description provided by user. Secondly, the LLM, guided by a delicately-designed prompt and the tool retrieval results, creates a tool-invoking plan as the solution path for the task. Thirdly, it takes actions to invoke tools based on the plan and receives feedback from the tool execution result. Finally, if the task is considered complete, it will generate the final response to the user.

As the first step in the above process, tool retrieval plays a critical role in constructing a high-performing tool-augmented agent. This is because the context length of the model restricts us to using only a limited number of tools. If necessary tools cannot be accurately retrieved, it will result in an execution error. To achieve accurate tool retrieval, prevalent tool retrieval methods primarily focus on the semantic similarities between the tool descriptions and the user queries (Patil et al., 2023; Li et al., 2023; Qin et al., 2023). They consider each tool independently, which, however, results in

[1]We will release our code and dataset upon the acceptance of our paper.

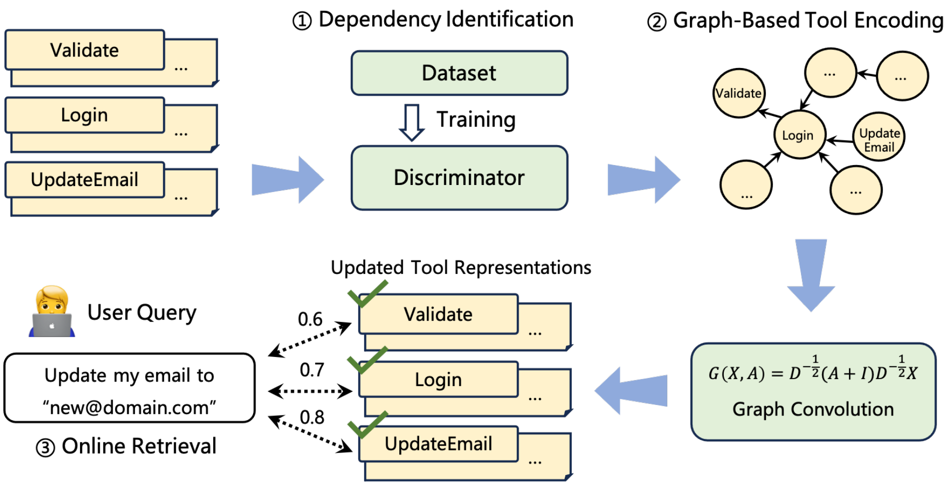

Figure 2: Our proposed TGR involves three steps: (1) Dependency Identification, where we build a dataset for tool dependency identification and train a discriminator; (2) Graph-Based Tool Encoding, where we represent the tools with dependencies as a graph and integrate the dependencies into tool representations with graph convolution; (3) Online Retrieval, where we utilize the updated tool embeddings to compute query-tool similarities as the final retrieval scores.

the omission of some necessary prerequisite tools during retrieval. For instance, in the example of Figure 1, the solution path for the query "*Update my email to 'new@domain.com'*." involves three tools that should be invoked in sequence: "*Validate*", "*Login*", and "*UpdateInfo*". However, the descriptions of tools "*Validate*" and "*Login*", which are about "*Validate credential*" and "*Login account*", are semantically irrelevant to the query. As a result, although the invocation of "*UpdateInfo*" depends on the results of "*Validate*" and "*Login*", only the tool "*UpdateInfo*" can be successfully retrieved.

In this paper, we propose **T**ool **G**raph **R**etriever (TGR), which exploits the dependencies between tools to refine the tool retrieval process. As shown in Figure 2, it involves three steps: (1) Dependency Identification. In this step, we construct a dataset, termed as TDI300K, and train a discriminator to identify the tool dependencies; (2) Graph-Based Tool Encoding. To model the dependencies, we construct a graph with tools as nodes and their dependencies as edges. Then we use graph convolution to integrate the dependencies for a better learning of the tool representations; (3) Online Retrieval. We conduct online retrieval by calculating the query-tool similarity with the updated tool representations. Compared with previous studies (Li et al., 2023; Qin et al., 2023; Patil et al., 2023), TGR leverages the tool dependencies as additional information to refine the retrieval process, thus leading to better results.

Overall, our contributions can be summarized as follows:

- We propose **T**ool **G**raph **R**etriever (TGR), leveraging tool dependencies as additional information to improve the performance of tool retrieval.
- We construct a tool dependency identification dataset termed TDI300K and subsequently train a discriminator, facilitating the subsequent studies in this area.
- Experimental results and in-depth analyses on several commonly-used datasets demonstrate that TGR brings the improvement of Recall, NDCG and Pass Rate to existing dominant methods, achieving state-of-the-art performance on several commonly-used datasets.

2 Related Work

Recently, LLMs have demonstrated outstanding abilities in many tasks. Meanwhile, it becomes dominant to equip LLMs with external tools, deriving many tool-augmented LLMs such as Toolformer (Schick et al., 2023), ART (Paranjape et al., 2023) and ToolkenGPT (Hao et al., 2023). However, as the number of tools grows rapidly, how to efficiently conduct tool retrieval becomes more important.

In this regard, Qin et al. (2023) employ Sentence-BERT (Reimers and Gurevych, 2019) to train a dense retriever based on a pretrained BERT-base (Devlin et al., 2019). The retriever encodes the queries and tool descriptions into embeddings respectively and selects top-k tools with the highest query-tool similarities. Similarly, Li et al. (2023) and Patil et al. (2023) employ paraphrase-MiniLM-L3-v2 (Reimers and Gurevych, 2019) and text-embedding-ada-002 [2] as the tool retrievers respectively. Besides, Hao et al. (2023) represent tools as additional tokens and finetune the original LLM to autonomously select the tool to be invoked. Unlike the studies mentioned above, Liang et al. (2023) divide tools into different categories to quickly locate relevant ones. They also employ Reinforcement Learning from Human Feedback for the entire task execution, so as to enhance the ability of the tool retriever.

Different from the above studies, TGR improves the effectiveness of tool retrieval with tool dependencies as additional information. We first identify the dependencies between tools and model them as a graph. Then, we use graph convolution to integrate the dependencies into tool representations, which are used for final online retrieval. To the best of our knowledge, our work is the first attempt to leverage tool dependencies to refine the retrieval process.

3 Tool Graph Retriever

As shown in Figure 2, the construction and utilization of our retriever involve three steps: 1) Dependency Identification; 2) Graph-Based Tool Encoding; 3) Online Retrieval. he following sections provide detailed descriptions of these steps.

3.1 Dependency Identification

In this work, we consider the tool t_a depends on the tool t_b if they satisfy one of the following conditions:

- The tool t_a requires the result from the tool t_b as the input. For example, if we want to update the email of a user with the tool "*UpdateEmail*", we should first get the permission from the user with the tool "*Login*". Therefore "*UpdateEmail*" depends on "*Login*" for permission acquisition.

- The tool t_a requires the tool t_b for prior verification. For example, the tool "*Login*" depends on the tool "*Validate*" to ensure a valid username combined with the correct password.

[2] https://platform.openai.com/docs/guides/embeddings

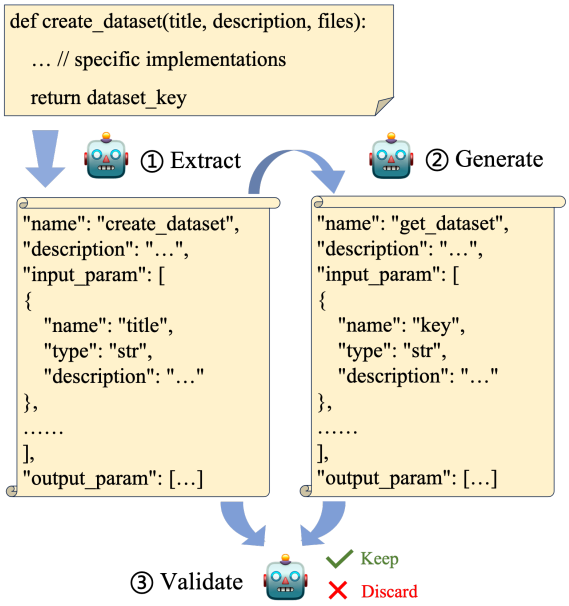

Figure 3: The pipeline used to construct the pretraining dataset, which involves three steps: 1) Extract tool document; 2) Generate dependent tool document; 3) Validate and filter the dependency.

Based on the definition above, we build a dataset termed TDI300K for tool dependency identification with the format $\{\langle t_a, t_b \rangle, y\}$, where $\langle t_a, t_b \rangle$ denotes a pair of tools and y denotes their dependencies with three categories: (1) t_a depends on t_b, (2) no dependency exists between t_a and t_b and (3) t_b depends on t_a. It is worth noting that the dependencies between tools are sparse in the tool set, which, however, poses challenges for training the discriminator on a dataset with an imbalanced proportion of different categories. To solve this problem, we adopt a two-stage strategy to train a 3-class discriminator: the pretraining stage enables the discriminator to understand tool functions, and the finetuning stage enhances its ability to identify tool dependencies.

Pretraining Due to the lack of open-source tool dependency identification dataset, we design a three-step pipeline to construct the pretraining dataset derived from CodeSearchNet (Husain et al., 2019), which contains 1.78 million real function implementations across various programming languages. As shown in Figure 3, we employ three agents based on gpt-3.5-turbo to extract tool documents, generate dependent tool documents, and validate the dependency. The LLM-specific prompts are shown in the Appendix B. Firstly, given the specific implementation of a tool function, which is

Category	Pretraining	Finetuning
$t_a \rightarrow t_b$	92,000	1,029
$t_a \times t_b$	92,000	33,365
$t_a \leftarrow t_b$	92,000	1,056

Table 1: The statistics of our constructed dataset TDI300K for tool dependency identification. The arrow $\rightarrow$ indicates the direction of the dependency and $\times$ means no dependency.

the source of t_a, we extract the document in JSON format, containing descriptions of tool functions, input parameters, and output results. Subsequently, the document of another tool t_b is generated which is required to depend on t_a. Finally, we evaluate whether the dependency between t_a and t_b fulfills the predefined criteria, discarding tool pairs that do not satisfy the conditions.

Once we obtain an instance where t_b depends on t_a, their positions can be swapped to obtain the opposing dependency category. Finally, we construct the instances without tool dependencies by breaking up and shuffling the tool pairs to make t_a and t_b independent. The statistics of the pretraining part of TDI300K are shown in Table 1. Notice that here we keep three categories balanced to ensure a comprehensive learning of our discriminator on all three categories.

The pretraining process is a 3-class classification task, where we concatenate the documents of t_a and t_b and separate them with a special token `[SEP]`, following Devlin et al. (2019). Besides, we add a special classification token `[CLS]` before the input sequence, whose final hidden state is used for the classification task. With $\hat{y}$ denoting the model prediction, we define the following cross-entropy training objective:

$$L(y, \hat{y}) = -\sum_{k=1}^{3} y_k \log(\hat{y}_k). \quad (1)$$

Finetuning To enhance the ability of the discriminator, we further finetune it on a manually constructed dataset with imbalanced category proportions which is more consistent with the real application scenario. First, we collect real function tools from open-source datasets, projects, and libraries[3]. Then, we write documents for these function tools with the same format as those in the pretraining dataset. Subsequently, these tools are organized as several tool sets to facilitate dependency annotation. Based on the definition of tool dependency mentioned above, we manually annotate the dependency categories given a pair of tools within a tool set. The statistics of the result datasets are also shown in Table 1.

From Table 1, we can clearly find that the imbalance category proportions propose a challenge for the discriminator. To deal with this problem and avoid overfitting, we define the following category-specific average training loss:

$$L(y, \hat{y}) = -\sum_{k=1}^{3} \frac{\sum_{i=1}^{N_k} y_{i,k} \log(\hat{y}_{i,k})}{N_k} \quad (2)$$

where N_k denotes number of instances with the k-th dependency category.

Notably, during the practical finetuning process, we split 20 percent of the whole dataset as the validation dataset, which is used to keep the checkpoint with the best performance. We also manually construct the testing dataset, which is derived from the existing tools in API-Bank, and the dependency categories are manually annotated. It contains 60, 500, and 60 samples for each category respectively. Here we choose API-Bank as the source of the test dataset since the tools are massive and the dependencies are hard to annotate in ToolBench. The performances of the discriminator on the validation and testing dataset will be presented in Section 4.2.

3.2 Graph-Based Tool Encoding

With the above tool dependency discriminator, we use it to identify the dependencies among the tool set and then construct a tool dependency graph. Formally, our graph is directed and can be formalized as $G = (V, E)$. In the node set V, each node represents a candidate tool. As for the edge set E, if the tool t_a depends on the tool t_b, the node of t_a will be linked to that of t_b, forming an edge. Let us revisit the graph in Figure 2. In this graph, we include the tools "*Validate*", "*Login*", and "*UpdateEmail*" as separate nodes, and construct two edges linking the tool nodes: "*Login*" to "*Validate*", "*UpdateEmail*" to "*Login*", respectively.

Then, based on the tool dependency graph, we adopt graph convolution (Kipf and Welling, 2017) to learn tool representations, where the tool dependency information is fully incorporated. Formally, we follow Kipf and Welling (2017) to con-

[3]The sources include datasets like the training dataset of ToolBench, projects like online shopping, and libraries like OpenGL.

duct graph-based tool encoding in the following way:

$$G(X, A) = D^{-\frac{1}{2}}(A + I)D^{-\frac{1}{2}}X. \quad (3)$$

Here X stands for the tool embedding matrix. A and D denote the adjacency matrix and degree matrix of the graph respectively. There are several ways to initialize the tool embeddings here. For latter experiment, we follow Qin et al. (2023) to use the retriever to encode the tool documents with a specific format for ToolBench to embeddings. While for API-Bank, we only encode the tool descriptions to mitigate the difference between the query domain and tool document domain. It is also worth noting that Equation 3 removes the trainable parameters of GCN (Kipf and Welling, 2017) to accelerate the retrieval process.

3.3 Online Retrieval

The final process of TGR is to retrieve tools with the updated tool representations, which have incorporated the dependency information. Specifically, given a user query, we encode the query to an embedding vector with the same dimension as the updated tool representations. Following Qin et al. (2023), we compute the similarities between the embeddings of queries and tools as the retrieval score. Subsequently, we rank all the candidate tools in descending order and return top-k tools with the highest scores.

4 Experiment

In this section, we conduct comprehensive experiments and in-depth analyses to evaluate the effectiveness of TGR.

4.1 Setup

Datasets We carry out experiments on two commonly-used datasets:

- **API-Bank** (Li et al., 2023). The test dataset of API-Bank involves 3 levels, including a total of 311 test samples which are composed of the user query, the corresponding tools, and the final execution results. During the evaluation, we extract user queries and the corresponding tool retrieval results to quantify the tool retrieval performance.

- **ToolBench** (Qin et al., 2023). Considering the massive number of APIs and time complexity, we conduct experiments with the ToolBench instances at the I1 level[4]. Given the category information about the APIs in ToolBench, we first group these APIs based on their categories. Subsequently, we identify the dependencies between APIs within each group and build a graph. Finally, on the basis of the graph, all API representations are updated for retrieval.

Baselines We compare TGR with several commonly used retrieval baselines, which can be mainly divided into the following two categories:

- **Word frequency-based retrieval methods**. Typically, these methods compute the similarities between the queries and tool descriptions according to the word frequency. In this category, the commonly used methods include **BM25** (Robertson et al., 2009) and **TF-IDF** (Ramos et al., 2003).

- **Text embedding-based retrieval methods**. The methods we consider in this category involve different text embedding models: **Paraphrase MiniLM-L3-v2** (Reimers and Gurevych, 2019) and **ToolBench-IR** (Qin et al., 2023), which have been used in API-Bank and ToolBench as tool retrievers respectively.

Implementation Details We use BERT-base-uncased (Devlin et al., 2019) as the base model of the discriminator. As described in Section 3.1, we first pretrain the discriminator on the category-balanced pretraining dataset, and then finetune it on the category-imbalanced finetuning dataset. During this process, we keep the checkpoint with the best performance on the validation dataset and evaluate its performance on the test dataset. Finally, we use Precision, Recall, and F1 score as the evaluation metrics for the discriminator.

As for the tool retrieval experiment on API-Bank, we simply use the description of tools for retrieval. For ToolBench, we follow Qin et al. (2023) to use a structured document format of tools containing names, descriptions, and parameters for retrieval. Lastly, following Qu et al. (2024), we consider three metrics: Recall, NDCG, and Pass Rate at the settings of top-5 and top-10 for both API-Bank and

[4]At I2 and I3 levels, each query involves APIs across different categories, which proposes challenges of high time complexity for constructing graphs. Thus, we leave extending our retrieval to other levels as future work.

Dataset	Method	Recall		NDCG		Pass Rate	
		@5	@10	@5	@10	@5	@10
API-Bank	BM25 (Robertson et al., 2009)	0.391	0.493	0.353	0.394	0.228	0.302
	TF-IDF (Ramos et al., 2003)	0.566	0.746	0.501	0.573	0.383	0.605
	PMLM-L3-v2 (Reimers and Gurevych, 2019)	0.659	0.763	0.569	0.609	0.479	0.592
	PMLM-L3-v2+TGR	**0.736**	**0.834**	**0.622**	**0.659**	**0.576**	**0.698**
	ToolBench-IR (Qin et al., 2023)	0.714	0.790	0.639	0.670	0.531	0.624
	ToolBench-IR+TGR	**0.761**	**0.878**	**0.664**	**0.712**	**0.595**	**0.788**
ToolBench-I1	BM25 (Robertson et al., 2009)	0.175	0.218	0.224	0.221	0.030	0.090
	TF-IDF (Ramos et al., 2003)	0.406	0.525	0.442	0.473	0.210	0.330
	PMLM-L3-v2 (Reimers and Gurevych, 2019)	0.365	0.468	0.399	0.421	0.140	0.250
	PMLM-L3-v2+TGR	**0.429**	**0.556**	**0.451**	**0.483**	**0.240**	**0.450**
	ToolBench-IR (Qin et al., 2023)	0.709	0.841	0.791	0.807	0.460	0.690
	ToolBench-IR+TGR	**0.742**	**0.868**	**0.811**	**0.829**	**0.510**	**0.730**

Table 2: Evaluation results on API-Bank and ToolBench-I1.

	Valid	**Test**
Precison	0.775	0.893
Recall	0.814	0.760
F1	0.792	0.817

Table 3: Performance of the tool dependency discriminator. We evaluate the Precision, Recall, and F1 score on the train, valid, and test datasets.

	API-Bank	**ToolBench**
#Total	119	10,439
#Connected	50	8,600
Proportion	0.420	0.824

Table 4: The proportion of connected graph nodes in API-Bank and ToolBench.

ToolBench. Here we define the Pass Rate as the proportion of test samples whose required tools are totally retrieved successfully, which can be formalized as follows:

$$pass@k = \frac{1}{|Q|}\sum_{q}^{Q} \mathbb{I}(\Phi(q) \subseteq \Psi^k(q)) \quad (4)$$

where $\Phi(q)$ denotes the set of ground-truth tools for query q, $\Psi^k(q)$ represents the top-k tools retrieved for query q, and $\mathbb{I}(\cdot)$ is an indicator function that returns 1 if the retrieval results include all ground-truth tools within the top-k results for query q, and 0 otherwise.

A higher Recall demonstrates that more required tools are successfully retrieved, a higher NDCG score indicates that the target tools achieve higher ranks, and a higher Rass Rate signifies that more queries are completed with all the required tools retrieved.

4.2 Discriminator Performance

In this group of experiments, we first focus on the quality of the constructed tool dependency graph, which, intuitively, greatly depends on the discriminator and is crucial for the performance of TGR.

To this end, we present the Precision, Recall, and F1 score of our discriminator across the validation and testing datasets in Table 3. Overall, our discriminator can achieve decent performance on two datasets. Additionally, the resulting graphs are visualized in Appendix C.

Furthermore, we calculate the proportions of connected nodes in the tool dependency graphs, as shown in Table 4. We note that the proportions of connected graph nodes differ between the two datasets, which is influenced by the granularity of tool functions because fully-featured tools are less likely to depend on others while specialized tools designed with fine-grained functions usually have more intensive dependencies.

Method		Recall		NDCG		Pass Rate	
		@5	@10	@5	@10	@5	@10
PMLM-L3-v2 (Reimers and Gurevych, 2019)	+TGR-d	0.736	0.834	0.622	0.659	0.576	0.698
	+TGR-m	**0.745**	**0.846**	**0.634**	**0.672**	**0.592**	**0.711**
ToolBench-IR (Qin et al., 2023)	+TGR-d	0.761	0.878	0.664	0.712	0.595	0.788
	+TGR-m	**0.788**	**0.893**	**0.698**	**0.741**	**0.646**	**0.817**

Table 5: Performance comparison between different TGRs, of which tool dependency graphs are constructed by our discriminator (represented as +TGR-d) and manual annotations (represented as +TGR-m). These group of experiments are conducted on the API-Bank (Li et al., 2023).

4.3 Main Results

The results of tool retrieval are presented in Table 2, showing that on all three metrics, TGR significantly improves the performance of base text embedding models and outperforms word frequency-based retrieval methods to a large extent. This indicates that incorporating tool dependency as additional information greatly enhances the effectiveness of tool retrieval. Furthermore, we arrive at the following interesting conclusions.

Firstly, when applying TGR to ToolBench-IR, which is finetuned specifically for the tool retrieval task, it can achieve the SOTA performance on both datasets. Therefore, we believe that finetuning and TGR are two methods that are compatible with each other and thus can be used to improve the performance of tool retrieval simultaneously.

It can also be seen that the methods based on ToolBench-IR greatly surpass others on ToolBench. This is because the tool documents in ToolBench have a specific format that only ToolBench-IR can fit well since it is finetuned on the training set of ToolBench.

4.4 Effect of Different Dependency Graph

In this subsection, we study the effect of the graph construction quality for TGR. Due to the extensive number of tools in ToolBench, which makes manual annotation of the entire graph impractical, we choose API-Bank and the same two text embedding models: Paraphrase MiniLM-L3-v2 (Reimers and Gurevych, 2019) and ToolBench-IR (Qin et al., 2023) for this experiment. We also use the same metric as the main experiments in Section 4.3.

Table 5 lists the experimental results. To avoid confusion, we term the TGR based on the discriminator as +TGR-d and on manual annotation as +TGR-m. From this table, we can clearly observe that +TGR-m performs better than +TGR-d, no matter which embedding model is used. In our opinion, this result is reasonable because the quality of the manually-constructed tool dependency graph is higher than that of the discriminator-constructed graph. Thus, we believe that how to improve the performance of our discriminator is very important for the further improvement of TGR.

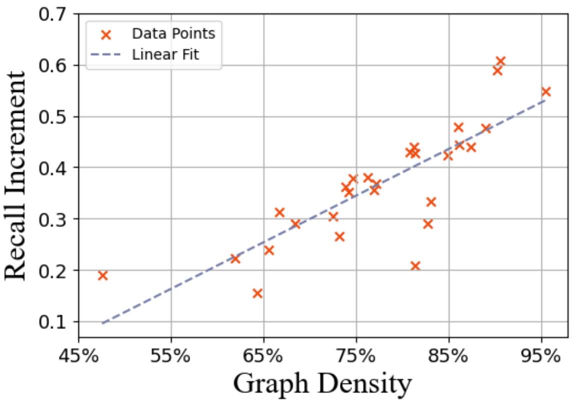

Figure 4: The relationship between the density of the tool dependency graph and the recall increment.

4.5 Effect of Graph Density

In this subsection, we evaluate the effect of the density of the tool dependency graph on tool retrieval. Specifically, we collate all the tools in ToolBench by their categories and rank the categories according to their graph density, which is measured by the proportion of connected tool nodes. Due to the limited size of the test set, we extract 100 queries for each category from the train set for evaluation, which are completely unused during the procedure of discriminator dataset construction. For the evaluation metric, we measure the recall increment of the

Query	Can you please help me delete my account? My username is foo and my password is bar.
Ground Truth	GetUserToken DeleteAccount
Dependency	DeleteAccount → GetUserToken
ToolBench-IR	1. DeleteAccount 2. AccountInfo 3. DeleteReminder 4. DeleteBankAccount 5. DeleteScene
Toolbench-IR+TGR	1. GetUserToken 2. Transfer 3. OpenBankAccount 4. RegisterUser 5. DeleteAccount

Table 6: Case study of tool retrieval on API-Bank. Correct APIs are highlighted in blue.

Query	Which football leagues' predictions are available for today? I want to explore the predictions for the Premier League and La Liga.
Ground Truth	Get Today's Predictions Get Next Predictions
Dependency	Get Next Predictions → Get Today's Predictions
ToolBench-IR	1. Daily Predictions 2. Football predictions by day 3. Get Next Predictions 4. VIP Scores 5. Prediction DetPredictionails
ToolBench-IR+TGR	1. Football predictions by day 2. Basketball predictions by day 3. Get Today's Predictions 4. Get Next Predictions 5. Sample predictions

Table 7: Case study of tool retrieval on ToolBench. Correct APIs are highlighted in blue.

TGR-enhanced text embedding model over the base text embedding model at the top-5 setting. Here we use the ToolBench-IR as the text embedding model considering its excellent retrieval performance.

The result is shown in Figure 4. We can see that as the density of the graph increases, the recall increment also exhibits an upward trend, which validates that dependencies between tools indeed help to improve the performance of tool retrieval. It also demonstrates that TGR is highly robust and more effective for dependency-intensive tool retrieval.

4.6 Case Study

Finally, we provide two examples to further illustrate how TGR improves the performance of tool retrieval. We conduct case studies on both API-Bank and ToolBench with ToolBench-IR as the base text embedding model, since it achieves the best performance in our main experiments.

Table 6 presents the first example in API-Bank, where the tool "*DeleteAccount*" requires the result (the user token) from the tool "*GetUserToken*" as an input parameter. We display the retrieval results by their ranking orders. The retrieval results of ToolBench-IR contain only one correct API "*DeleteAccount*" with the top rank due to its high semantic similarity with the query. With the enhancement of TGR, "*GetUserToken*", which "*DeleteAccount*" depends on, incorporates the information from "*DeleteAccount*" and is also retrieved with a high rank.

Table 7 presents the second example in ToolBench. It is obvious that the base ToolBench-IR misses the required API "*Get Today's Prediction*". Given the relationship that "*Get Next Prediction*" depends on "*Get Today's Prediction*", the TGR-enhanced ToolBench-IR succeeds in retrieving the missing tool.

5 Conclusion

In this paper, we introduce Tool Graph Retriever (TGR), leveraging tool dependencies to enhance the tool retrieval process for LLMs. We first define the criteria for tool dependency and establish a dataset to train a discriminator for identifying tool dependencies. Then, we use this discriminator to handle candidate tools, forming a tool dependency graph. Subsequently, via graph convolution, we perform tool encoding based on this graph, where the updated tool representations can be used for the final tool retrieval. Experimental results and in-depth analyses strongly demonstrate the effectiveness of TGR across multiple datasets.

In the future, we will explore more features to improve our discriminator, which has a significant impact on the performance of our TGR. Besides, we will try some efficient graph networks to obtain better tool representations. Finally, how to further enhance the generalization of our TGR is also one of our future research focuses.

Limitations

In our opinion, due to the absence of a tool dependency identification dataset, the accuracy of the discriminator is somewhat limited. The time complexity of graph construction is $O(N^2)$, which could be optimized by developing prior rules to filter out tools with no apparent dependency.

References

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *NAACL*.

Fei Dou, Jin Ye, Geng Yuan, Qin Lu, Wei Niu, Haijian Sun, Le Guan, Guoyu Lu, Gengchen Mai, Ninghao Liu, et al. 2023. Towards artificial general intelligence (agi) in the internet of things (iot): Opportunities and challenges. *arXiv*.

Ben Goertzel. 2014. Artificial general intelligence: concept, state of the art, and future prospects. *JAGI*.

Shibo Hao, Tianyang Liu, Zhen Wang, and Zhiting Hu. 2023. Toolkengpt: Augmenting frozen language models with massive tools via tool embeddings. *arXiv*.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv*.

Thomas N Kipf and Max Welling. 2017. Semi-supervised classification with graph convolutional networks. In *ICLR*.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023. Api-bank: A comprehensive benchmark for tool-augmented llms. In *EMNLP*.

Yaobo Liang, Chenfei Wu, Ting Song, Wenshan Wu, Yan Xia, Yu Liu, Yang Ou, Shuai Lu, Lei Ji, Shaoguang Mao, et al. 2023. Taskmatrix. ai: Completing tasks by connecting foundation models with millions of apis. *arXiv*.

Scott McLean, Gemma JM Read, Jason Thompson, Chris Baber, Neville A Stanton, and Paul M Salmon. 2023. The risks associated with artificial general intelligence: A systematic review. *JETAI*.

Bhargavi Paranjape, Scott Lundberg, Sameer Singh, Hannaneh Hajishirzi, Luke Zettlemoyer, and Marco Tulio Ribeiro. 2023. Art: Automatic multi-step reasoning and tool-use for large language models. *arXiv*.

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. 2023. Gorilla: Large language model connected with massive apis. *arXiv*.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv*.

Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. 2024. Colt: Towards completeness-oriented tool retrieval for large language models. *arXiv*.

Juan Ramos et al. 2003. Using tf-idf to determine word relevance in document queries. In *ICML*.

Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. In *ACL*.

Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends® in Information Retrieval*, 3(4):333–389.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *arXiv*.

A Evaluation on Different Similarity Computing Methods

We evaluate the effect of different similarity computing methods, including cosine similarity, dot product similarity, L1 distance, and L2 distance. The experiment is conducted on both API-Bank and ToolBench using Paraphrase-MiniLM-L3-v2 and ToolBench-IR as the text-embedding models. From Figure 5, we can see that cosine similarity performs the best, while L1 and L2 distance have relatively lower performance. In our opinion, it is because the L1 and L2 distance ignore the angles between vectors, thus losing some features.

B Prompts for LLMs during Dataset Construction

Here we provide the prompts we used for each LLM in the dataset construction pipeline we mentioned in Section 3.1. The prompts are shown in Figure 6, 7, and 8 respectively.

C Visualization of the Graph

We also visualize the tool dependency graph. Considering aesthetics and simplicity, we display part of the graph in API-Bank and ToolBench constructed by the discriminator. The graph is shown in Figure 9 and 10.

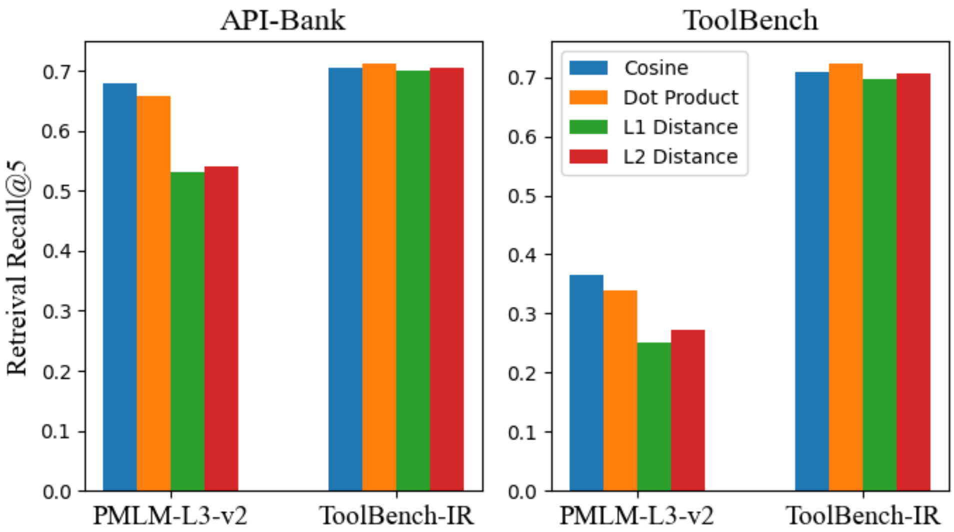

Figure 5: Evaluation on different similarity computing methods.

~~~
The following is a code for a Python function. Please extract the corresponding API
document and return it in JSON format:
```{language}
{code}
```

Please strictly follow the JSON format below to return the API document:```json
{
  "name": "function name",
  "description": " the function description can be directly filled with `doc_string` ",
  "input_param": [
    {
      "name": "name of input parameter ",
      "type": "type of input parameter",
      "description": "description of input parameter"
    }
  ],
  "output_param": [
    {
      "name": "name of output parameter",
      "type": "type of output parameter",
      "description": "description of output parameter"
    }
  ]
}
```
Among them, the function name field name and function description field are strings,
and the input parameter `input_parm` field and output parameter `ouput_parm` field
are lists. Each parameter contains three fields: parameter name, parameter type, and
parameter description, all of which are string types. If no function or parameter
description is provided in the code, please analyze the function based on its
functionality and fill it in English completely.

Please provide the API documentation for the above function:
~~~

Figure 6: The prompt for the LLM to extract API documentation.

```
Here is a API document. To analyze the input and output parameters of this document, you
need to generate another API document that meets one of the following conditions:
1. The input parameters of the generated API are obtained from the output parameters of the
given API
2. The output parameters of the generated API can serve as input parameters for the given API

Below is the given API documentation:
```json
{api}
```
Please return your results in the following json format:
```json
{
  "satisfied": " The serial number that satisfies the above conditions, with a value of 1 or 2 ",
  “name”: ”function name",
  "description": " the function description can be directly filled with `doc_string` ",
  "input_param": [
    {
      "name": " name of input parameter ",
      "type": " type of input parameter ",
      "description": " description of input parameter "
    }
  ],
  "output_param": [
    {
      "name": " name of output parameter ",
      "type": " type of output parameter ",
      "description": " description of output parameter "
    }
  ]
}
```
Among them, satisfied indicates that the generated API satisfies which condition, numerical
type, the value is 1 or 2. function name field name and function description field are strings,
input parameter `input_parm` field and output parameter `ouput_param` are lists, each
parameter contains parameter name, parameter type type, parameter description description
three fields, all of which are of string type. Please return all the above fields in English.
Please give the documentation of the generated API:
```

Figure 7: The prompt for the LLM to generate API documentation of a dependent tool function.

````
Here are two API documents given in json format, you need to determine
if there is a parameter dependency between the two APIs, i.e. whether the
input of one API should be taken from the output of the other.
The first API:
```json
{api1}
```
The second API:
```json
{api2}
```
Determine whether the above two APIs have dependencies on parameters,
and return them in the following json format:
```json
{
  "result": " 1 means there is a dependency on the parameter, 0 means
there is no dependency on the parameter "
}
```
````

Figure 8: The prompt for the LLM to verify the dependency between two tools with the format of API documentation.

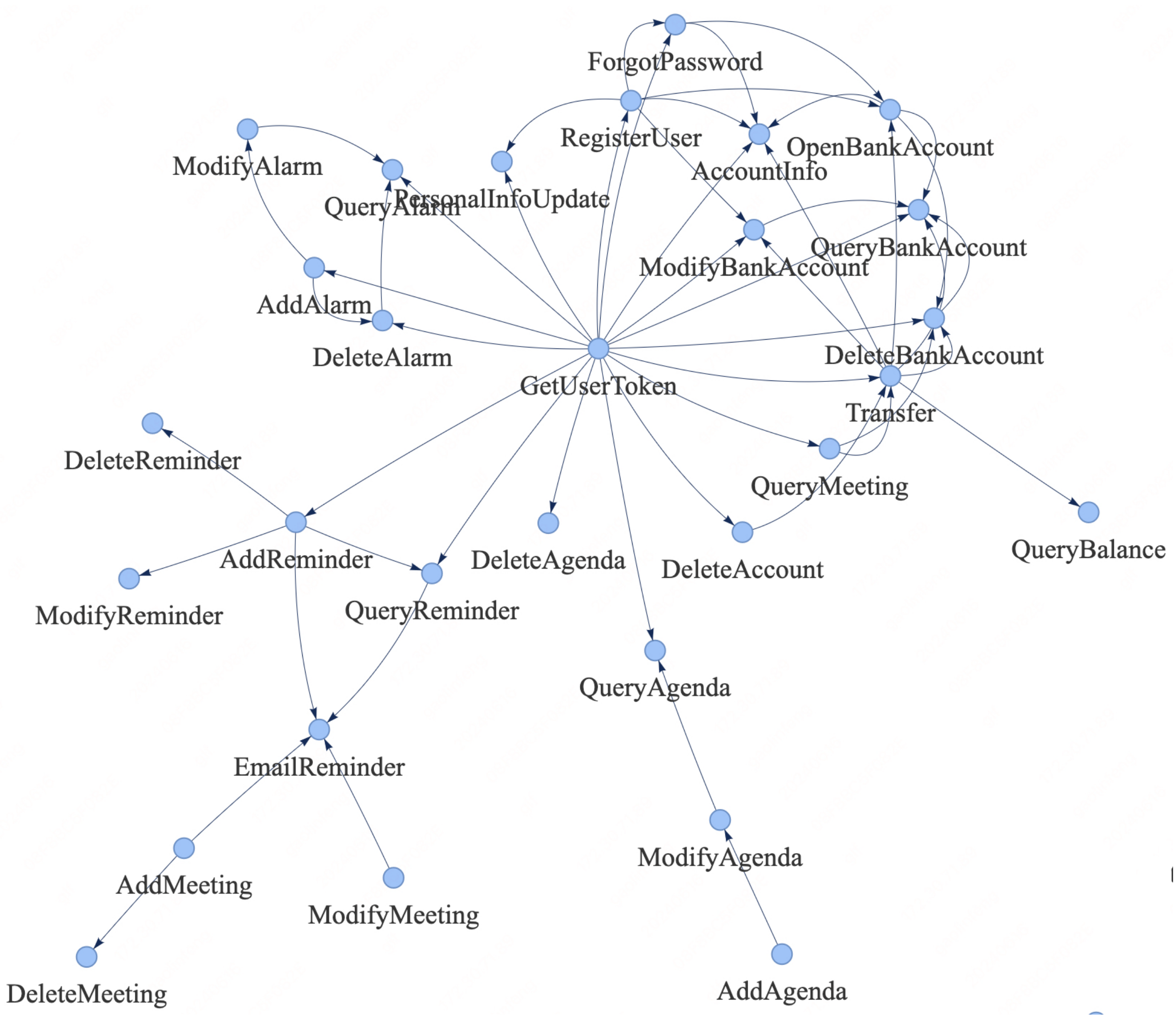


Figure 9: Visualization of the part of our constructed tool dependency graph in API-Bank (Li et al., 2023). The directed edge from $t_a$ to $t_b$ means $t_a$ is the prerequisite of $t_b$, i.e. the calling of $t_b$ depends on $t_a$.

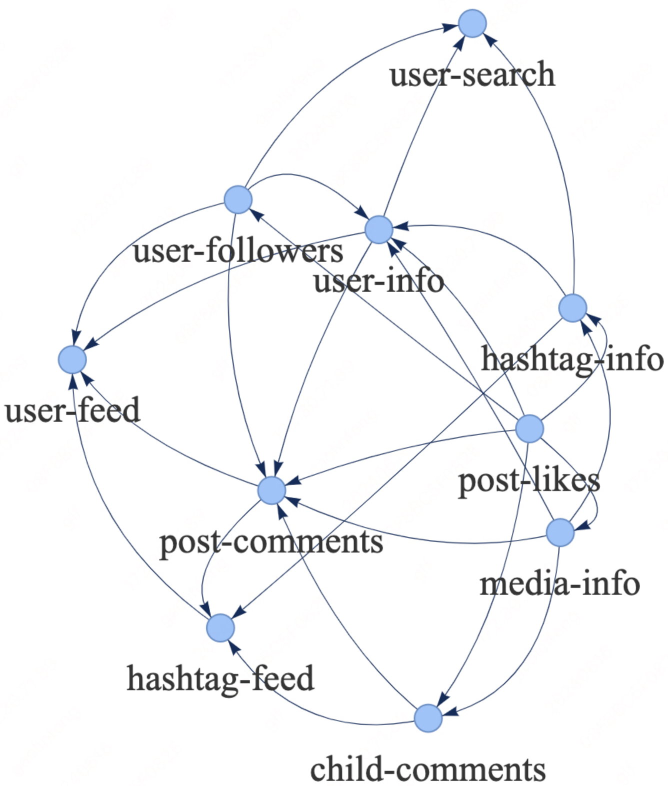


Figure 10: Visualization of the part of our constructed tool dependency graph in Toolbench (Qin et al., 2023).