

# Feasible is Not Enough: Cost-Aware Optimal Tool-Chain Planning on Multi-Solution Tool Graphs

Liangliang Liu<sup>1\*</sup>, Yanming Li<sup>1\*</sup>, Yigang Liu<sup>1</sup>, Jialong Han<sup>1</sup>, Rujia Shen<sup>1</sup>,  
Yi Guan<sup>1†</sup>, Yi Lin<sup>2†</sup>, Jingchi Jiang<sup>1†</sup>

<sup>1</sup>Harbin Institute of Technology, <sup>2</sup>Harbin Medical University  
{liull, liyanming, 25s103345, 25s003074, shenrujia}@stu.hit.edu.cn,  
linyi@hrbmu.edu.cn, {guanyi, jiangjingchi}@hit.edu.cn

## Abstract

Tool graphs (TG) model dependencies among tools and resources, enabling more structured organization and management of large toolsets. However, existing methods and benchmarks often formulate tool learning (TL) as a single-solution setting, overlooking the fact that many tasks admit multiple valid tool combinations and therefore require optimal solution selection. Moreover, exploring large-scale TG is computationally expensive, especially under constrained context budgets. To address these challenges, we construct MultiSoTLBench, Multi-Solution TL Benchmark in which each task admits multiple valid solutions. To solve this problem efficiently on large TG, we further propose TOPT, an efficient framework for learning optimal TL policies. Specifically, to improve search efficiency in large action spaces, TOPT adopts a progressive graph expansion strategy: we train a reinforcement learning (RL) agent to acquire transferable expansion skills and construct, on demand, a compact solvable subgraph that preserves only task-relevant chains. This reduces the size of the candidate space and the context usage from the outset. To enable optimal selection, we further propose a progressive graph reasoning framework. It performs cost-aware planning over the expanded subgraph to generate an optimal tool-chain that balances path length and tool cost. Comprehensive experiments demonstrate that TOPT generalizes effectively, improving task success and solution optimality by 46.21% and 66.34%, respectively.

## 1 Introduction

Large language models (LLMs), such as GPT (Achiam et al., 2023), Claude, Gemini (Comanici et al., 2025), and DeepSeek (Guo et al., 2025), are driving rapid technological advances. More recently, tool learning (TL) has emerged as a promising approach to equip LLM-based agents with ex-

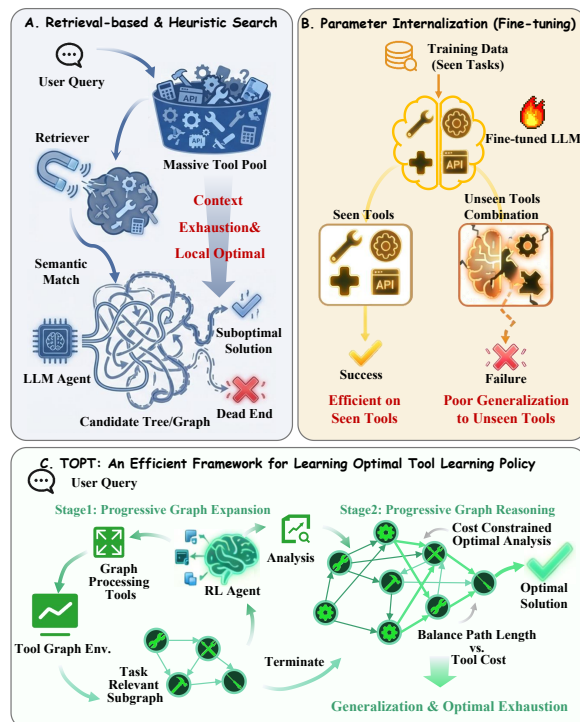


Figure 1: Comparison of TL paradigms and the proposed TOPT.

ternal tools for solving complex tasks (Schick et al., 2023; Qin et al., 2023; Tang et al., 2023; Wang et al., 2024; Jin et al., 2025). In TL, given a problem, the agent must construct a tool invocation chain, an ordered sequence of tool calls whose execution yields a solution (Qu et al., 2025). However, the resulting candidate action space can be enormous: each tool may have multiple name variants and parameter configurations. This combinatorial growth not only strains the LLM’s context window but also makes searching for a globally optimal solution extremely challenging.

Recent work on LLM TL under limited context windows broadly follows two paradigms (Figure 1A–B). The first mitigates context constraints via retrieval and heuristic search: it either uses external retrievers (e.g., term matching or semantic similarity) to surface task-relevant tools from large reposi-

\*Equal contribution. † Corresponding authors.

ories (Gao et al., 2024; Xu et al., 2024; Zheng et al., 2024; Chen et al., 2024), or organizes tools into compact structures (e.g., trees/graphs) and applies efficient search to generate candidate tool invocation chains, which are then injected into prompts to guide execution (e.g., CoT/ToT (Wei et al., 2022; Yao et al., 2023) or MCTS(Hao et al., 2024) and ToolChain\* (Zhuang et al., 2023)). However, such candidates are typically driven by local relevance heuristics and may miss compositional dependencies among tools. Motivated by this, the second paradigm internalizes tool-use knowledge by constructing TL datasets and fine-tuning LLMs to learn selection and invocation patterns (e.g., Tool-LaMA(Qin et al., 2023), TRICE(Qiao et al., 2024), ToolVerifier (Mekala et al., 2024)), with recent extensions that combine graph reasoning and RL (e.g., GraphChain (Wei et al., 2025)).

Despite these advances, existing methods and benchmarks often treat TL as a single-solution setting, implicitly assuming that any valid tool chain suffices once a task is solvable. In practice, however, many tasks admit multiple valid tool combinations that vary significantly in efficiency, reliability, and cost. Ignoring this multiplicity prevents the agent from reasoning about trade-offs among alternative tool chains and from selecting an optimal solution instead of an arbitrary feasible one. Meanwhile, exploring large-scale tool graphs (TG) to uncover such alternatives remains computationally intensive, particularly under constrained context budgets. These challenges call for frameworks that can reason over multiple valid solutions while maintaining scalability and efficiency.

Inspired by humans’ exploratory problem solving, which incrementally focuses on task-relevant options, we revisit tool learning over large tool graphs from a multi-solution perspective. To systematically study LLMs’ ability to select optimal solutions among multiple valid tool paths, we first propose MultiSoTLBench, a high-quality benchmark based on TMDB and Spotify, constructed automatically and verified through rigorous human annotation. Building on this setting, we propose TOPT, an efficient framework for learning optimal TL policies over large TG (Figure 1C). Specifically, to improve search efficiency in large tool combination spaces, TOPT first adopts a progressive graph expansion strategy: we train an RL agent to acquire transferable expansion skills and, on demand, construct a compact solvable subgraph that preserves only task-relevant chains. This substantially re-

duces both the candidate space and context usage from the outset. Based on the expanded subgraph, TOPT then performs cost-aware progressive graph reasoning to generate an executable tool chain that balances path length and tool cost. Comprehensive experiments show that TOPT improves task success and solution optimality by 46.21% and 66.34% over state-of-the-art baselines, respectively, and generalizes well to unseen tasks and tool compositions. The main contributions are:

- We construct MultiSoTLBench, a high-quality, large-scale multi-solution TL benchmark for systematically evaluating optimality-aware tool learning, where each task admits multiple valid solutions with different costs and efficiencies.
- We propose TOPT, an efficient framework for learning optimal TL policies over large TG, explicitly addressing the challenge of optimal solution selection under multiple valid tool chains.
- We propose a two-stage progressive graph expansion and reasoning strategy, which improves generalization to unseen tools, by learning a query-conditioned RL policy to construct a compact solvable subgraph and perform cost-aware planning for optimal tool-chain selection.

## 2 Related Work

TOPT is inspired by two research threads: improving TL under large TG, and learning optimal planning in large action spaces.

### 2.1 Improving Tool Learning under Large Tool Graphs

Due to the limited context capacity of LLMs, prior work has explored several directions. Some methods introduce external retrievers to bridge the gap between LLM input constraints and the scale of tool libraries, using information retrieval signals to select task-relevant tools from large collections. Such as sparse retrieval via term matching (Patil et al., 2024) and dense retrieval via semantic representations (Qin et al., 2023; Gao et al., 2024; Yuan et al.; Zheng et al., 2024). Others organize tools into compact structured representations (e.g., trees/graphs), apply structured retrieval or graph-based analysis to generate candidate tool chains, and then rely on policy prompting for decision making and execution, e.g., CoT (Wei et al., 2022), ReAct (Yao et al., 2022), DFSDT (Qin et al., 2023), ToolNet (Liu et al., 2024), and ToolChain\* (Zhuang et al., 2023). Another line constructs TL datasets

and fine-tunes LLMs to internalize tool-use knowledge into model parameters, e.g., ToolLLaMA (Qin et al., 2023), GPT4Tools (Yang et al., 2023), and ToolRL (Qian et al., 2025).

Motivated by these ideas, TOPT also models the tool space with a compact structure. The key distinction is that TOPT does not rely on static retrieval or matching over a pre-built TG. Instead, it learns a query-conditioned graph expansion strategy that constructs, on demand, a solvable subgraph containing task-relevant chains. This design reduces context usage and candidate-space size at the source, while improving generalization to unseen tools and novel compositions.

## 2.2 Optimal planning in large action spaces

Obtaining optimal tool chains in a large tool action space is a central challenge for LLM-based agents. Prior work mainly follows two lines: structured reasoning over intermediate traces and search-based planning. Chain-of-Thought (CoT) makes intermediate steps explicit, Self-Consistency (Wang et al., 2022) aggregates multiple reasoning traces, and Tree-of-Thought (ToT) structures reasoning as a tree with heuristic search over branches. In parallel, planning-based methods integrate search into tool use; for example, RAP (Hao et al., 2023) combines a world model with search-based decision making and reward signals, while ToolChain\* uses efficient A\* search to reduce the cost of MCTS-style exploration. Recent work also couples graph-structured reasoning with RL, for example, GraphChain injects RL interactive exploration into graph reasoning to distill task-relevant structured knowledge.

Inspired by these ideas, TOPT also integrates RL with tool-graph planning, but with a different focus. Our contribution is not a new generic graph-search primitive, but a multi-solution, cost-aware formulation of tool learning, a benchmark tailored to this setting, and a task-conditioned framework for efficient tool-chain planning. Within this framework, TOPT trains an RL agent to decide when and where to expand the graph or invoke graph operations under budget constraints, while classical graph algorithms remain responsible for deterministic path computation.

## 3 Methods

### 3.1 Problem Definition

Given a problem description, TL aims to construct a tool invocation chain and execute it sequentially

to obtain a valid answer. However, existing methods often rely on static retrieval or single-step selection, treating TL as a single-solution problem and overlooking the fact that many tasks admit multiple valid tool chains. They also struggle with the scalability of tool reasoning, as searching over large TG is computationally expensive, especially under limited context budgets. Inspired by RL, we reformulate TL as a sequential graph-expansion reasoning process and formalize it as a Markov Decision Process (MDP)  $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R, \gamma)$ .

At time step  $t$ , the state  $s_t \in \mathcal{S}$  is the natural-language context provided by the TG environment, consisting of the query  $\mathcal{Q}$ , an abstract description of the current subgraph  $d^{\text{Info}}$ , and the interaction history  $\{a_i, d_i^{\text{Tool}}\}_{i=0}^{t-1}$ . Here  $d_i^{\text{Tool}}$  is a summary of the  $i$ -th tool execution result, automatically populated by predefined templates. An action  $a_t \in \mathcal{A}$  is either  $a_t = (T, \theta_T)$ , which selects a graph processing tool  $T \in \mathcal{T}$  with parameters  $\theta_T$ , or the terminal action Terminate. After executing  $a_t$ , the transition  $P$  updates the current subgraph and its textual description based on the selected tool and its output, yielding the next state  $s_{t+1}$ . The reward  $r(s_t, a_t, s_{t+1})$  measures reasoning progress and task completion, encouraging effective expansions and eventual success, while the discount factor  $\gamma \in [0, 1]$  trades off immediate and long-term returns. The agent follows a policy  $\pi_\theta(a_t|s_t)$  parameterized by  $\theta$ , inducing a trajectory  $\tau = \{s_0, a_0, r_0, \dots, s_T, a_T, r_T\}$  that captures the sequential interaction with the environment.

### 3.2 MultiSoTLBench

To systematically study LLMs’ ability to select optimal solutions among multiple valid tool paths, we propose MultiSoTLBench<sup>1</sup>, a high-quality, large-scale multi-solution TL benchmark covering TMDb<sup>2</sup> and Spotify<sup>3</sup>. The dataset is automatically constructed and rigorously human-verified. While prior TG benchmarks (Guo et al., 2024; Shen et al., 2024; Ma et al., 2024; Lu et al., 2025) for tool augmented LLMs often focus on a single tool or short, fixed chains for realistic tasks, MultiSoTLBench emphasizes cost-aware decision-making across multiple feasible solutions, encouraging models to generate tool paths with lower overall cost.

<sup>1</sup><https://github.com/LLLYM12138/TOPT>

<sup>2</sup><https://themoviedb.org>

<sup>3</sup><https://developer.spotify.com>

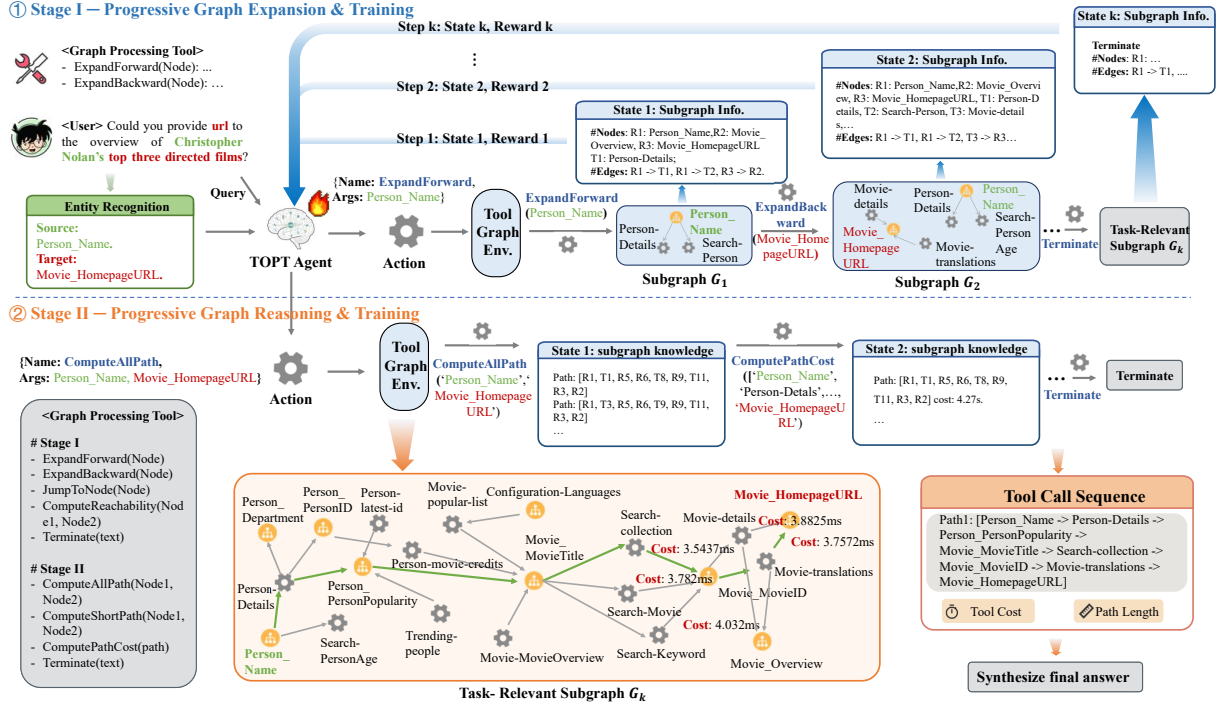


Figure 2: Overview of the TOPT framework. TOPT decomposes tool learning into two stages: progressive graph expansion and reasoning. In Stage I, the agent uses graph processing tools to incrementally construct a task-relevant subgraph  $G_k$ , terminating with a Terminate action. In Stage II, the agent performs cost-aware planning over the expanded subgraph to derive an optimal tool invocation chain by jointly considering tool cost and path length. The final tool sequence is executed, and outputs are aggregated to produce the answer.

Table 1: Dataset statistics. Task difficulty is defined by the number of required tool calls (EASY: 1–2, MEDIUM: 3–5, HARD: > 5). The test set is obtained via real tool-environment interactions, so the number of feasible paths (#Paths) is not pre-recorded.

Dataset	#Tools	EASY	MEDIUM	HARD	#Task	#Paths
TMDB-Train	112	483	250	216	949	10k
Spotify-Train	61	497	245	210	952	10k
TMDB-Test	112	50	30	20	100	–
Spotify-Test	61	50	30	20	100	–

### 3.2.1 Graph Construction

TG instruction involves a large collection of mutually independent tools. When tools are composed, their dependencies can take different forms (e.g., resource, temporal, or environment); here, we focus on resource dependencies. We model both tools and the resources in their inputs/outputs as nodes: tool nodes  $\mathcal{N}_T$  and resource nodes  $\mathcal{N}_R$ . Dependencies are encoded as a directed graph: if  $N_t$  produces  $N_r$ , we add  $N_t \rightarrow N_r$ ; if  $N_{t'}$  consumes  $N_{r'}$ , we add  $N_{r'} \rightarrow N_{t'}$ . Under this construction, tool composability is induced by resource sharing and propagation, without hand-crafted type-matching rules. We construct MultiSoTLBench in two domains:

-**TMDB**. TMDB provides an official API for

movies, TV shows, actors, and images. The resulting graph contains 112 tool nodes and 155 resource nodes, with 1,445 directed edges.

-**Spotify**. Spotify provides APIs for content metadata, recommendations, playlist management, and playback control. The resulting graph contains 61 tool nodes and 266 resource nodes, with 1,949 directed edges.

### 3.2.2 Data Construction

Given the TG, we sample subgraphs and use LLMs to synthesize user queries. Specifically, for each sampled subgraph  $G_s$ , we identify source-like resource nodes with no incoming edges and target-like resource nodes with no outgoing edges. We then run Dijkstra’s algorithm to find a shortest path  $P$  between a source resource and a target resource, and prompt an LLM to generate a corresponding query  $Q$ :

$$Q = \text{LLM}(\text{prompt}, P). \quad (1)$$

If the LLM deems  $P$  unnatural for a user query, we discard the path, which helps maintain both task complexity and data quality. Since  $G_s$  provides only structural tool invocation information, we further prompt the LLM to instantiate missing execution parameters and produce the final

query. Finally, three human experts review the tasks and instructions for complexity, logical consistency, and alignment with the intended goals. Overall, MultiSoTLBench contains approximately 20K samples. To further assess model performance, we stratify task difficulty by the required number of tool calls and report results within each bucket (Table 1).

### 3.3 The Overview of TOPT

As shown in Figure 2, TOPT employs a two-stage framework for tool learning. Stage I performs progressive graph expansion to construct a compact, task-relevant subgraph on demand. Stage II performs cost-aware progressive graph reasoning over the expanded subgraph to derive an executable tool chain and produce the final answer. In this section, we describe the overall design of TOPT and the details of its two stages.

#### 3.3.1 Stage I: Progressive Graph Expansion

In large TG, heuristic or rule-based expansion often suffers from over-expansion: to avoid missing key nodes, it conservatively branches out or repeatedly enlarges local neighborhoods, rapidly inflating the subgraph. This growth blows up the candidate path set and action space, raising retrieval and path-evaluation costs and further straining the limited context budget, so downstream reasoning must operate on noisier and more redundant evidence.

To address this issue, we formulate Stage I as a constrained subgraph construction problem with optimal stopping. Each graph processing tool call is treated as one decision step: at step  $t$ , the agent selects a tool and the node it applies to (or chooses Terminate); the environment executes the operation, updates the subgraph, and returns feedback. In this view, Terminate is a substantive decision that governs the solvability, which compactness trade-off under a strict budget. Let  $G_t$  denote the subgraph after step  $t$  and  $K$  the termination step at which the agent outputs  $G_K$ . We cast Stage I as a unified utility maximization that favors a solvable terminal subgraph, rewards task-relevant progress, and regularizes invalid calls and unnecessary growth. Formally, we optimize:

$$\mathcal{J}^E = \max_{\pi^E} \mathbb{E}_{\tau^E \sim \pi^E} \left[ \Phi_{\text{conn}}(G_K) + \sum_{t=0}^K \gamma^t \Phi_{\text{rel}}(s_t, a_t) - \lambda_f \sum_{t=0}^K \gamma^t C_t^f - \lambda_s \sum_{t=0}^K \gamma^t C_t^s \right], \quad (2)$$

where  $\tau^E = \{(s_t, a_t, r_t)\}_{t=0}^K$  denotes the expansion trajectory.

Our TG environment maintains global TG metadata (e.g., node/edge types, and tool attributes such as latency/cost) together with the evolving subgraph state, so all utility and cost items are dynamically calculated. Additional details of the environment and tool implementations are provided in Appendix E and Appendix F. Let  $\mathcal{P}_{G_K}(N_{\text{src}}, N_{\text{tgt}})$  denote the set of candidate source-target paths in the terminal subgraph  $G_K$ . The terminal utility  $\Phi_{\text{conn}}(G_K)$  measures solvability via path existence:

$$\Phi_{\text{conn}}(G_K) = C + \lambda_t \cdot \mathbb{I}[|\mathcal{P}_{G_K}(N_{\text{src}}, N_{\text{tgt}})| \geq 1], \quad (3)$$

where  $C$  and  $\lambda_t$  are hyperparameters used to balance the terminal utility.

To provide dense progress signals during expansion, the process utility  $\Phi_{\text{rel}}(s_t, a_t)$  indicates whether the executed operation yields task-relevant evidence (e.g., reaching valid nodes/chain under the query-conditioned mapping):

$$\Phi_{\text{rel}}(s_t, a_t) = \mathbb{I}[\text{Relevant}(s_t, a_t) = 1], \quad (4)$$

To ensure that the learned policy operates within the tool interface, we introduce an executability penalty  $C_t^f$  that is activated when the chosen tool call is ill-formed (e.g., schema mismatch or invalid arguments):

$$C_t^f = \mathbb{I}[\text{InvalidToolOrArgs}(a_t) = 1], \quad (5)$$

In addition, to control subgraph growth under a finite interaction budget, we impose a step cost  $C_t^s$  that increases once the expansion length exceeds a task-dependent threshold  $t_{\text{GT}}$ :

$$C_t^s = 0.1 (t - t_{\text{GT}}) \cdot \mathbb{I}[t > t_{\text{GT}}]. \quad (6)$$

Finally,  $\lambda_f, \lambda_s \geq 0$  control the trade-off between expansion quality and efficiency. Overall, these definitions integrate terminal solvability, step-wise relevance progress, and constraint regularization into a unified optimization criterion, enabling the learned policy to retain task-critical chain under budget constraints and provide a compact, information-concentrated search space for Stage II.

#### 3.3.2 Stage II: Progressive Graph Reasoning

Even after Stage I constructs a task-relevant subgraph  $G_k$ , the source and target nodes are often linked by multiple feasible paths, whose corresponding tool chains can vary widely in both execution cost and length. Therefore, returning an arbitrary feasible path is insufficient, while exhaustively enumerating and scoring all candidates becomes prohibitive when  $G_k$  admits many alternatives. Stage II resolves this by formulating tool-

chain selection as cost-aware planning over a fixed subgraph: The agent incrementally collects information about feasible paths, tool cost, and path length, which includes feasibility, tool cost, and path length, and then decides when to terminate and commit to a final tool chain. Importantly, Stage II is not intended to replace classical graph algorithms for deterministic path computation. Instead, such algorithms are used as graph-operation primitives (e.g., path enumeration, shortest-path computation, and path-cost estimation), while the learned policy decides which primitive to invoke, how much evidence to gather, and when to terminate under bounded interaction and context budgets.

Formally, Stage II is a constrained multi-path planning problem on  $G_k$ , aiming to identify the best tool chain under the cost-length objective among feasible source-target paths. Each graph reasoning tool invocation corresponds to a decision step. In particular, the reasoning tools themselves implement standard graph operations, whereas the policy learning component is responsible for adaptively orchestrating these tools for optimal selection. At step  $t$ , the agent selects a reasoning tool, and the environment applies it to  $G_k$ , updates the agent’s internal state over paths (e.g., newly discovered paths and updated cost estimates), and returns feedback. The agent may invoke `Terminate` at step  $M$  to stop further exploration and output a selected path  $p$ . Let  $\mathcal{P}_{G_k}(N_{\text{src}}, N_{\text{tgt}})$  denote the set of feasible source-target paths in  $G_k$ . For any candidate path  $p \in \mathcal{P}_{G_k}(N_{\text{src}}, N_{\text{tgt}})$ , we define a cost-length trade-off objective and the corresponding compromise-optimal path  $p^*$  as follows:

$$p^* = \arg \min_{p \in \mathcal{P}_{G_k}(N_{\text{src}}, N_{\text{tgt}})} J(p), \quad (7)$$

where  $J(p) = \alpha \cdot \text{Cost}(p) + \beta \cdot |p|$  is jointly determined by the tool cost and the path length. Here, optimality is defined with respect to the benchmark objective  $J(p)$ , which serves as a practical engineering criterion balancing tool cost and path length, rather than a universal notion of optimality ( $\alpha = 0.75, \beta = 0.25$ ); in practical deployment,  $\alpha$  and  $\beta$  can be adjusted to reflect different budget and latency preferences. Accordingly, we specify the optimization objective of Stage II as an episodic objective that evaluates the quality of the final selected path at termination and penalizes ill-formed tool calls during the search:

$$\mathcal{J}^R = \max_{\pi^R} \mathbb{E}_{\tau^R \sim \pi^R} \left[ \Phi_{\text{opt}}(p; G_k) - \sum_{t=0}^M \gamma^t C_t^f \right], \quad (8)$$

where  $\tau^R = \{(s_t, a_t, r_t)\}_{t=0}^M$  denotes the planning trajectory, and  $p$  is the path returned when the agent invokes `Terminate`.  $C_t^f$  is a validity penalty computed by Eq. 5.  $\Phi_{\text{opt}}(p; G_k)$  denotes the terminal utility, encouraging the agent to output the best path under the objective  $J(p)$  whenever the instance is solvable:

$$\begin{aligned} \Phi_{\text{opt}}(p; G_k) = & -\lambda_1 \cdot \mathbb{I}[\mathcal{P}_{G_k}(N_{\text{src}}, N_{\text{tgt}}) = 0] \\ & + \mathbb{I}[\mathcal{P}_{G_k}(N_{\text{src}}, N_{\text{tgt}}) \geq 1] \\ & + \lambda_2 \cdot \mathbb{I}[p = p^*]. \end{aligned} \quad (9)$$

Overall, this objective formulation supports efficient path selection on  $G_k$ . It penalizes invalid tool invocations during exploration and encourages the agent to finally commit to a feasible path that is best under the tool cost and path length objective  $J(p)$ .

### 3.3.3 RL for Training TOPT Framework

Due to the two-stage design, we implement a dedicated TG environment for training TOPT. At each step, the environment parses the agent action  $a_t$ , executes the corresponding graph operation, updates the current graph state, and constructs the next state  $s_{t+1}$ . In each episode, the agent first runs Stage I until termination to obtain a compact task-relevant subgraph  $G_k$ , and then runs Stage II on the fixed subgraph  $G_k$  to produce the final tool path.

We jointly optimize the two stages under a unified PPO (Schulman et al., 2017) objective by using the composite objective  $\mathcal{J} = \mathcal{J}^E + \mathcal{J}^R$ . Formally, to optimize the LLM agent policy  $\pi_\theta$ , we use proximal policy optimization with clipping (PPO-clip). Given trajectories  $\tau$  sampled by the behavior policy  $\pi_{\theta_{\text{old}}}$ , we maximize the clipped surrogate objective:

$$\mathcal{L}^{\text{CLIP}}(\theta) = \mathbb{E}_t \left[ \min(w_t(\theta) \hat{A}_t, \text{clip}(w_t(\theta), 1 - \epsilon, 1 + \epsilon) \hat{A}_t) \right], \quad (10)$$

where  $w_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$  and  $\hat{A}_t$  is the advantage estimate computed with respect to the composite objective  $\mathcal{J}$ . The hyperparameter  $\epsilon$  constrains the policy update ratio.

## 4 Experiments

In this section, we conduct comprehensive evaluations of TOPT by addressing the following research questions:

- RQ1:** How does TOPT compare to state-of-the-art baselines?
- RQ2:** Does graph expansion reasoning improve generalization to unseen tools and novel compositions, as well as reasoning optimality?
- RQ3:** Can TOPT work in a tuning-free setting?

Table 2: Performance comparison of TOPT in two scenarios, reporting Correct Path rate (CP %), Optimal path rate (OP %), time cost (s), and token usage (k).

Methods	Difficulty	Param.	TMDB				Spotify				Average			
			CP%↑	OP%↑	Time Cost↓	Token↓	CP%↑	OP%↑	Time Cost↓	Token↓	CP%↑	OP%↑	Time Cost↓	Token↓
Tuning-free	GPT-4.1	EASY	84	68	2.13s	8.29	86	86	1.71s	8.30	85	77	1.92s	8.29
		MEDIUM	33.33	20	2.15s	10.29	20	16.67	1.76s	8.26	26.67	18.34	1.96s	9.28
		HARD	5	5	2.74s	13.29	0	0	1.64s	9.29	2.5	2.5	2.19s	11.29
		Total	53	41	2.34s	9.89	49	48	1.71s	8.49	51	44.5	2.03s	9.19
	CoT (GPT-4.1)	EASY	84	68	3.59s	8.31	88	86	1.75s	8.30	86	77	2.67s	8.31
		MEDIUM	40	26.67	3.26s	16.60	20	16.67	1.72s	13.41	30	21.67	2.49s	15.01
		HARD	5	5	4.82s	24.89	0	0	1.00s	24.91	2.5	2.5	2.91s	24.90
		Total	55	43	3.74s	14.12	50	48	1.74s	13.16	52.5	45.5	2.74s	13.64
	ReAct (GPT-4.1)	EASY	90	82	15.15s	11.41	92	82	6.23s	10.76	91	82	10.69s	11.08
		MEDIUM	56.67	43.33	15.45s	24.83	23.33	20	7.73s	19.73	40	31.67	11.59s	22.28
		HARD	0	0	14.03s	38.08	20	15	8.37s	34.41	10	7.5	11.20s	36.24
		Total	62	54	15.02s	20.77	57	50	7.11s	18.18	59.5	52	11.07s	19.47
RestGPT (GPT-4.1)	EASY	46	30	29.57s	24.10	66	44	29.68s	10.61	56	37	29.63s	17.36	
	MEDIUM	33.33	16.67	37.56s	30.43	20	10	38.13s	12.45	26.67	13.34	37.85s	21.44	
	HARD	30	15	26.29s	25.46	15	15	35.52s	14.22	22.5	15	30.91s	19.84	
	Total	39	23	31.31s	26.27	42	28	33.38s	11.88	40.5	25.5	32.35s	19.08	
Tuning-based	ToolLLaMA	EASY	38	38	15.71s	6.72	36	36	15.86s	5.87	37	37	15.79s	6.30
		MEDIUM	33.33	33.33	15.84s	10.53	36.67	36.67	16.01s	11.37	35	35	15.93s	10.95
		HARD	10	10	16.41s	17.46	15	15	16.21s	16.45	12.5	12.5	16.31s	16.96
		Total	27.11	27.11	15.99s	11.57	29.22	29.22	16.03s	11.23	28.17	28.17	16.01s	11.40
	TOPT (Ours)	EASY	98	96	22.49s	15.06	98	98	35.12s	13.07	98	97	28.81s	14.07
		MEDIUM	60	60	29.68s	10.78	100	100	29.09s	16.04	80	80	29.39s	13.41
		HARD	40	40	32.54s	12.35	100	100	25.89s	12.84	70	70	29.22s	12.59
		Total	75 (20.9% ↑)	74 (37% ↑)	28.24s	12.73	99 (73.7% ↑)	99 (98% ↑)	31.46s	13.92	87 (46.21% ↑)	86.5 (66.34% ↑)	29.85s	13.32

-**RQ4:** How does Stage I compare with heuristic expansion strategies?

-**RQ5:** Does TOPT remain effective on smaller backbone models?

#### 4.1 Experimental Setup

We implement TOPT in PyTorch and use Qwen3-8B as the base model. All training and evaluation are conducted on two NVIDIA H800 GPUs. We first apply supervised fine-tuning (SFT) to help the model quickly acquire the desired output format, and then further optimize it with PPO. The definitions and configurations of all hyperparameters are detailed in Appendix B.

#### 4.2 Evaluation Metrics

Following prior work (Song et al., 2023; Zhuang et al., 2023), we evaluate performance using the correct path rate (CP%), a standard metric in TL. Since our setting involves optimization-oriented reasoning, we further report the optimal path rate (OP%), time cost (s), and the number of tokens (k) to provide a comprehensive assessment.

#### 4.3 Baselines.

We evaluate TOPT against a set of strong, widely adopted TL baselines, covering both tuning-free and tuning-based paradigms. For tuning-free methods, we test two-shot prompting with CoT (Wei et al., 2022), ReAct (Yao et al., 2022), and the RestGPT (Song et al., 2023). For tuning-based methods, we reproduce ToolLLaMA (Qin et al., 2023). Additional implementation details and settings are provided in the Appendix C.

#### 4.4 RQ1. Overall Performance

Table 2 compares TOPT with state-of-the-art baselines. Improvements are statistically significant under a two-sample  $t$ -test ( $p < 0.05$ ). The key insights include:

-**Superior performance.** TOPT achieves 87% CP and 86.5% OP, outperforming the strongest baseline ReAct (CP: 52%) by 67.3% relative.

-**Better tool-chain optimization.** Baselines often reach feasible solutions but struggle to optimize chain length and cost, yielding OP well below CP. TOPT first narrows the search space via task-relevant subgraph construction and then applies cost-aware planning to select a near-optimal tool chain, improving optimality.

-**Parameter efficiency.** TOPT outperforms GPT-4.1-based approaches ( $\sim 200B$ ) with only 8B parameters, demonstrating strong parameter efficiency.

#### 4.5 RQ2. Cross-Domain Generalization

To evaluate TOPT’s cross-domain generalization, we conduct a transfer experiment by training on TMDB and testing directly on Spotify. As shown in Table 4, TOPT transfers well to the unseen toolset, achieving CP/OP = 80/80 on Spotify (vs. 75/75 on TMDB) with consistently strong results across difficulty levels (EASY: 82/82, MEDIUM: 73.33/73.33, HARD: 85/85).

We also note that the Spotify TG is structurally simpler than TMDB. Many Spotify tools generate outputs mainly for user presentation and cannot be used as inputs to other tools. This reduces inter-tool dependencies, shortens feasible tool chains, and shrinks the branching search space, which in turn makes it easier for TOPT to identify solvable

Table 3: Performance comparison of TOPT under a tuning-free framework, with reporting success rate%, correct path rate(CP%), average path length (PL), optimal path rate(OP%), and time Cost.

		TMDB					Spotify					Average				
		Success↑	CP%↑	PL↓	OP%↑	Time Cost↓	Success↑	CP%↑	PL↓	OP%↑	Time Cost↓	Success↑	CP%↑	PL↓	OP%↑	Time Cost↓
GPT-4.1	EASY	99.12	92	7.78	92	8.21s	99.03	100	7.28	98	7.65s	99.07	96	7.53	95	7.93s
	MEDIUM	98.75	43.33	9.07	43.33	5.92s	100	100	8.7	100	8.78s	99.37	71.66	8.88	71.66	7.55s
	HARD	100	20	15.5	20	5.14s	95	95	8.3	95	7.93s	97.5	57.5	11.9	57.5	6.53s
	Total	99.18	63	10.78	63	6.91s	98	99↑	7.91	99↑	8.05s	98.59	81	9.345	81	7.48s
Qwen3-14B	EASY	97.84	86	6.62	86	106.82s	93.27	68	6.66	68	150.65s	95.55	77	6.64	77	128.73s
	MEDIUM	98.89	46.67	5	46.67	92.61s	98.68	56.67	5.93	56.67	128.64s	98.78	51.67	5.465	51.67	110.62s
	HARD	93.99	25	6.65	25	180.2s	97.14	85	6.55	85	91.53s	95.56	55	6.6	55	135.86s
	Total	97.38	62	6.14	62	117.23s	95.67	68	6.42	68	132.23s	96.52	65	6.28	65	124.73s
Qwen3-8B	EASY	97.07	78	6.32	78	119.49s	88	90	10.68	90	81.69s	92.53	84	8.5	84	100.59s
	MEDIUM	98.89	46.67	5	46.67	92.61s	98.68	56.67	5.93	56.67	128.64s	98.78	51.67	5.465	51.67	110.62s
	HARD	96.04	15	5.3	15	212.74s	82.39	85	10.15	85	83.22s	89.21	50	7.725	50	147.98s
	Total	97.18	52	5.68	52	148.96s	82.11	84	10.64	84	83.21s	89.64	68	8.16	68	116.08s
Qwen3-4B	EASY	45.04	36	27.98	36	144.72s	96	82	6.58	82	49.78s	70.52	59	17.28	59	97.25s
	MEDIUM	57.58	23.33	18.33	23.33	155.58s	88.21	73.33	10.9	73.33	85.74s	72.89	21.66	12.05	21.66	120.66s
	HARD	45.64	0	14.25	0	144.85s	85.95	35	6.55	35	62.95s	65.79	17.5	10.4	17.5	103.9s
	Total	48.92	25	22.34	25	144.74s	91.66	54	6.33	54	55.78s	70.29	39.5	14.33	39.5	100.26s
TOPT	EASY	98.43	98	9.4	96	22.49s	97.85	98	9.96	98	35.12s	98.14	98	9.68	97	28.80s
	MEDIUM	98.39	60	8.07	60	29.68s	94.9	100	10.9	100	29.09s	96.645	80	9.48	80	29.38s
	HARD	83.8	40	13.2	40	32.54s	97.26	100	10	100	25.89s	90.53	70	11.6	70	29.21s
	Total	95.49	75(19% ↑)	9.76	74(17.4% ↑)	28.24s	96.85	99↑	10.25	99↑	31.46s	96.17	87(7.4% ↑)	10.005	86.5(6.7% ↑)	29.85s

Table 4: Cross-domain generalization of TOPT by training on TMDB and testing on the unseen Spotify tool graph, reporting CP and OP across difficulty levels.

Difficulty	TMDB		Spotify	
	CP%↑	OP%↑	CP%↑	OP%↑
EASY	98	96	82	82
MEDIUM	60	60	73.33	73.33
HARD	40	40	85	85
Total	75	75	80	80

Table 5: Comparison of Stage I with heuristic expansion strategies.

Method	Node	Edge	Useful	Total
Uni. AND-OR	199.30	198.30	9.22	199.30
Bi. AND-OR	27.33	29.63	5.41	27.33
TOPT	92.94	209.75	3.42	3.70

paths and select near-optimal tool chains under a constrained context budget.

#### 4.6 RQ3. Tuning-Free Performance

To assess TOPT’s adaptability, we adapt the TG environment and plug GPT-4.1 and the Qwen3 series into TOPT. Based on Table 3, we obtain three insights.

**-Framework-level gains without tuning.** In tuning-free mode, integrating GPT-4.1 into TOPT markedly improves over the strongest baseline in Table 2 (ReAct: +36.1% CP, +55.7% OP), indicating that the gains mainly stem from the framework rather than task-specific tuning.

**-Capacity helps, but TOPT benefits are consistent.** Larger backbones perform better (Avg CP/OP: GPT-4.1 81/81, Qwen3-14B 65/65, Qwen3-8B 68/68, Qwen3-4B 39.5/39.5), yet TOPT consistently lifts different backbones beyond the baseline level in Table 2.

#### -Strong optimality with parameter efficiency.

TOPT (ours) reaches 87/86.5 (CP/OP), surpassing all tuning-free variants and achieving near-optimal tool-chain selection with only 8B parameters.

#### 4.7 RQ4. Isolating the contribution of Stage I

To better isolate the contribution of Stage I, we first evaluate it independently under budget constraints by examining whether it can construct a task-relevant subgraph that covers the key tools and resource fields required for downstream planning. The results show that Stage I achieves success rates of 100% (50/50), 73.33% (22/30), and 50% (10/20) on EASY, MEDIUM, and HARD tasks, respectively, with an overall success rate of 82% (82/100). This suggests that Stage I alone can provide feasible candidate coverage in most cases, while also indicating that coverage becomes the main bottleneck as task difficulty increases.

We further compare TOPT with heuristic expansion baselines, including unidirectional AND-OR search and bidirectional AND-OR search. Table 5 reports their exploration cost and effectiveness, where Node/Edge Scale denotes the size of the resulting task-relevant subgraph, Total actions the number of expansion actions, and Useful actions the subset that truly advances executable tool-chain construction.

The results show that heuristic methods require substantially more exploration than TOPT. Unidirectional AND-OR search takes 199.3 actions on average, and bidirectional AND-OR search reduces this to 27.33. In contrast, TOPT (Stage I) completes effective graph expansion with only 3.7 actions on average, 3.42 of which are useful. This suggests that TOPT replaces large amounts of low-value exploration with a few highly targeted expansions.

The gain mainly comes from learning a query-

Table 6: Results on smaller backbones under three settings: Prompt, TOPT (tuning-free), and TOPT (tuning-based).

		TMDB			Spotify			Average		
		CP% $\uparrow$	OP% $\uparrow$	Time Cost $\downarrow$	CP% $\uparrow$	OP% $\uparrow$	Time Cost $\downarrow$	CP% $\uparrow$	OP% $\uparrow$	Time Cost $\downarrow$
Qwen3-1.7B	Prompt	20	14	15.11s	42	42	13.84s	31	28	14.48s
	TOPT (tuning-free)	22	22	113.99s	46	46	77.25s	34	34	95.62s
	TOPT (tuning-based)	60	60	22.59s	64	64	26.00s	61	61	24.30s
Qwen3-0.6B	Prompt	18	16	15.82s	28	28	14.69s	23	22	15.26s
	TOPT (tuning-free)	18	18	112.05s	34	34	54.38s	26	26	83.22s
	TOPT (tuning-based)	40	34	23.20s	46	40	19.61s	43	37	21.41s

conditioned expansion policy, which allocates budget to the most promising frontier and can flexibly switch between forward and backward expansion. Although RL introduces a one-time offline training cost, it substantially reduces online expansion and evaluation calls at inference time. Since the current benchmark scale is still relatively limited, we expect this efficiency advantage to become even more pronounced on larger and more redundant tool graphs.

#### 4.8 RQ5. Results on smaller backbones

To examine whether TOPT also benefits low-resource LLMs for edge deployment, we further evaluate two smaller backbones, Qwen3-1.7B and Qwen3-0.6B, under three settings: Prompt, TOPT (tuning-free), and TOPT (tuning-based). Table 6 reports the total results on TMDB and Spotify.

The results show that TOPT remains effective on smaller backbones. Compared with pure prompting, tuning-free TOPT still yields consistent CP/OP gains on Spotify and modest gains on TMDB for Qwen3-1.7B, while the gains are smaller but still visible for Qwen3-0.6B. After tuning, the improvements become substantially larger: on Spotify, CP improves from 42% to 46% to 64% for Qwen3-1.7B, and from 28% to 34% to 46% for Qwen3-0.6B; on TMDB, CP improves from 20% to 22% to 60% for Qwen3-1.7B, and from 18% to 18% to 40% for Qwen3-0.6B. Tuning also greatly reduces inference latency by suppressing ineffective expansion and path evaluation. For example, for Qwen3-1.7B, the Time Cost drops from 113.99s to 22.59s on TMDB and from 77.25s to 26.00s on Spotify after tuning. These results suggest that the benefit of TOPT is not limited to larger backbones and remains meaningful in low-resource deployment settings.

## 5 Conclusion

In this study, we construct MultiSoTLBench, a high-quality, large-scale benchmark for multi-solution TL, enabling systematic evaluation of

optimality-aware TL. Building on this setting, we further propose TOPT, an efficient framework for learning optimal TL policies over a large TG. Inspired by humans’ exploratory cognition, TOPT first adopts a progressive graph expansion strategy, which trains an RL agent to acquire transferable expansion skills and, on demand, construct a compact solvable subgraph that retains only task-relevant chains, thereby reducing both the candidate search space and context usage from the outset. It then performs cost-aware planning over the expanded subgraph to produce an optimal tool chain that balances path length and tool execution cost. Extensive experiments on MultiSoTLBench show that TOPT generalizes effectively, improving task success and solution optimality by 46.21% and 66.34%, respectively.

## Limitations

Despite these gains, several limitations remain. MultiSoTLBench is grounded in real APIs and explicit dependency graphs, providing a realistic testbed for studying cost-aware tool learning. At the same time, the current benchmark mainly models resource dependencies and does not yet cover some richer interaction types, such as temporal constraints or environment-dependent conditions. In addition, the current tool graphs remain moderate in scale and domain coverage relative to large real-world tool ecosystems. Future work will further extend the benchmark with more domains, larger tool collections, and richer dependency structures.

## Acknowledgment

This work was supported in part by a grant from the National Key Research and Development Program of China [2025YFE0209200], the Key Research and Development Program of Heilongjiang Province (China) [JD2023GJ01, 2024ZX01A07], and the National Natural Science Foundation of China [72293584].

## References

- Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, and 1 others. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.
- Jianlyu Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. 2024. M3-embedding: Multi-linguality, multi-functionality, multi-granularity text embeddings through self-knowledge distillation. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 2318–2335.
- Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, and 1 others. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*.
- Shen Gao, Zhengliang Shi, Minghang Zhu, Bowen Fang, Xin Xin, Pengjie Ren, Zhumin Chen, Jun Ma, and Zhaochun Ren. 2024. Confucius: Iterative tool learning from introspection feedback by easy-to-difficult curriculum. In *Proceedings of the AAAI conference on artificial intelligence*, volume 38, pages 18030–18038.
- Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.
- Zhicheng Guo, Sijie Cheng, Hao Wang, Shihao Liang, Yujia Qin, Peng Li, Zhiyuan Liu, Maosong Sun, and Yang Liu. 2024. Stabletoolbench: Towards stable large-scale benchmarking on tool learning of large language models. *arXiv preprint arXiv:2403.07714*.
- Shibo Hao, Yi Gu, Haotian Luo, Tianyang Liu, Xiyan Shao, Xinyuan Wang, Shuhua Xie, Haodi Ma, Adithya Samavedhi, Qiyue Gao, and 1 others. 2024. Llm reasoners: New evaluation, library, and analysis of step-by-step reasoning with large language models. In *ICLR 2024 Workshop on Large Language Model (LLM) Agents*.
- Shibo Hao, Tianyang Liu, Zhen Wang, and Zhiting Hu. 2023. Toolkengpt: Augmenting frozen language models with massive tools via tool embeddings. *Advances in neural information processing systems*, 36:45870–45894.
- Qiao Jin, Zhizheng Wang, Yifan Yang, Qingqing Zhu, Donald Wright, Thomas Huang, Nikhil Khandekar, Nicholas Wan, Xuguang Ai, W John Wilbur, and 1 others. 2025. Agentmd: Empowering language agents for risk prediction with large-scale clinical tool learning. *Nature Communications*, 16(1):9377.
- Xukun Liu, Zhiyuan Peng, Xiaoyuan Yi, Xing Xie, Lirong Xiang, Yuchen Liu, and Dongkuan Xu. 2024. Toolnet: Connecting large language models with massive tools via tool graph. *arXiv preprint arXiv:2403.00839*.
- Jiarui Lu, Thomas Holleis, Yizhe Zhang, Bernhard Aumayer, Feng Nan, Haoping Bai, Shuang Ma, Shen Ma, Mengyu Li, Guoli Yin, and 1 others. 2025. Tool-sandbox: A stateful, conversational, interactive evaluation benchmark for llm tool use capabilities. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 1160–1183.
- Zixian Ma, Weikai Huang, Jieyu Zhang, Tanmay Gupta, and Ranjay Krishna. 2024. m & m’s: A benchmark to evaluate tool-use for multi-step multi-modal tasks. In *European Conference on Computer Vision*, pages 18–34. Springer.
- Dheeraj Mekala, Jason E Weston, Jack Lanchantin, Roberta Raileanu, Maria Lomeli, Jingbo Shang, and Jane Dwivedi-Yu. 2024. Toolverifier: Generalization to new tools via self-verification. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 5026–5041.
- Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. 2024. Gorilla: Large language model connected with massive apis. *Advances in Neural Information Processing Systems*, 37:126544–126565.
- Cheng Qian, Emre Can Acikgoz, Qi He, Hongru Wang, Xiushi Chen, Dilek Hakkani-Tür, Gokhan Tur, and Heng Ji. 2025. Toolrl: Reward is all tool learning needs. *arXiv preprint arXiv:2504.13958*.
- Shuofei Qiao, Honghao Gui, Chengfei Lv, Qianghuai Jia, Huajun Chen, and Ningyu Zhang. 2024. Making language models better tool learners with execution feedback. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 3550–3568.
- Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, and 1 others. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789*.
- Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. 2025. Tool learning with large language models: A survey. *Frontiers of Computer Science*, 19(8):198343.
- Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551.

- John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.
- Yongliang Shen, Kaitao Song, Xu Tan, Wenqi Zhang, Kan Ren, Siyu Yuan, Weiming Lu, Dongsheng Li, and Yueting Zhuang. 2024. Taskbench: Benchmarking large language models for task automation. *Advances in Neural Information Processing Systems*, 37:4540–4574.
- Yifan Song, Weimin Xiong, Dawei Zhu, Wenhao Wu, Han Qian, Mingbo Song, Hailiang Huang, Cheng Li, Ke Wang, Rong Yao, and 1 others. 2023. Restgpt: Connecting large language models with real-world restful apis. *arXiv preprint arXiv:2306.06624*.
- Qiaoyu Tang, Ziliang Deng, Hongyu Lin, Xianpei Han, Qiao Liang, Boxi Cao, and Le Sun. 2023. Toolalpaca: Generalized tool learning for language models with 3000 simulated cases. *arXiv preprint arXiv:2306.05301*.
- Hongru Wang, Yujia Qin, Yankai Lin, Jeff Z Pan, and Kam-Fai Wong. 2024. Empowering large language models: Tool learning for real-world interaction. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 2983–2986.
- 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*.
- Chunyu Wei, Wenji Hu, Xingjia Hao, Xin Wang, Yifan Yang, Yunhai Wang, Yang Tian, and Yueguo Chen. 2025. Graphchain: Large language models for large-scale graph analysis via tool chaining. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.
- Qiancheng Xu, Yongqi Li, Heming Xia, and Wenjie Li. 2024. Enhancing tool retrieval with iterative feedback from large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 9609–9619.
- Rui Yang, Lin Song, Yanwei Li, Sijie Zhao, Yixiao Ge, Xiu Li, and Ying Shan. 2023. Gpt4tools: Teaching large language model to use tools via self-instruction. *Advances in Neural Information Processing Systems*, 36:71995–72007.
- 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*, 36:11809–11822.
- Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*.
- Lifan Yuan, Yangyi Chen, Xingyao Wang, Yi Fung, Hao Peng, and Heng Ji. Craft: Customizing llms by creating and retrieving from specialized toolsets. In *The Twelfth International Conference on Learning Representations*.
- Yuanhang Zheng, Peng Li, Wei Liu, Yang Liu, Jian Luan, and Bin Wang. 2024. Toolrerank: Adaptive and hierarchy-aware reranking for tool retrieval. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 16263–16273.
- Yuchen Zhuang, Xiang Chen, Tong Yu, Saayan Mitra, Victor Bursztyn, Ryan A Rossi, Somdeb Sarkhel, and Chao Zhang. 2023. Toolchain\*: Efficient action space navigation in large language models with a\* search. *arXiv preprint arXiv:2310.13227*.

## A Case Study

We present a case study in Figure 4-5 involving a TMDB network for the MultiSoTLBench, comprising 112 tool nodes, 155 resource nodes and 1,445 edges—exceeding the tool learning capabilities of current LLMs. TOPT demonstrates robust capability to systematically handle this task through a series of graph processing tool invocations. We propose a progressive graph expansion and reasoning strategy: a query-conditioned RL policy incrementally constructs a compact solvable subgraph and performs cost-aware planning over it to derive optimal tool chains.

## B Details of Experimental Setup

We provide implementation details to ensure reproducibility. All experiments were conducted on two NVIDIA H800 (80GB) GPUs. We fine-tuned the Qwen3-8B backbone using LoRA with rank  $r=16$  and  $\alpha=32$ .

### B.1 Training Configuration

Our training pipeline consists of two stages:

- **Supervised Fine-Tuning (SFT).** We use a learning rate of  $5 \times 10^{-5}$  with a 4% warmup ratio and a cosine learning-rate schedule for 5 epochs. This stage teaches the model to follow graph-reasoning instructions.
- **Reinforcement Learning (RL).** We further optimize the model with Proximal Policy Optimization (PPO) using step-level rewards, rather than applying rewards only at the final step as in many RLHF settings. Unless otherwise specified, we use:
  - Learning rate:  $5 \times 10^{-5}$
  - Batch size: 8
  - Initial KL coefficient: 0.3
  - Loss coefficient ( $\beta$ ): 0.15
  - Discount factor ( $\gamma$ ): 0.99

## C Baseline Implementation

To comprehensively evaluate TOPT, we implement state-of-the-art tool-learning baselines spanning both tuning-free and tuning-based paradigms.

### C.1 Tuning-Free Methods

For tuning-free methods (e.g., CoT, ReAct, and RestGPT), we use the GPT-4.1 API as the underlying base model.

### C.2 Tuning-Based Methods

For tuning-based methods (e.g., ToolLLaMA), we reproduce the model using the official open-source implementation. We train with a learning rate of  $2 \times 10^{-5}$  for 3 epochs and set the maximum sequence length to 2048. During inference, we set the temperature to 0 and cap the maximum generation length at 1024 tokens to ensure stable outputs.

All open-source baselines are trained and evaluated on the same hardware as TOPT (two NVIDIA H800 GPUs). For closed-source baselines, we use the official GPT-4.1 API.

## D Tool Chain Analysis

To further characterize how TOPT adapts across domains, we analyze the distribution of tool invocations by difficulty on TMDB and Spotify (Figure 3). Overall, TOPT follows an information-gain-driven search over the directed tool graph: Stage I tools (ExpandForward, ExpandBackward, ComputeReachability, Terminate) dominate the trajectories, while Stage II tools (ComputeAllPath, ComputeShortPath, ComputePathCost) are invoked selectively when multiple feasible routes remain and optimal selection is necessary.

**TMDB: tool-dense graphs induce deeper exploration and verification.** TMDB is tool-dense (112 tools, 155 resources, 1,445 edges; tool/resource  $\approx 0.72$ ), which leads TOPT to expand more aggressively and verify more often. Concretely, ExpandForward is the most frequent operation (232 calls) with a sizable MEDIUM/HARD share (30%/24%), reflecting the need to progressively grow the frontier under a large action space. In harder instances, TOPT relies more on backward dependency completion and global path enumeration: ExpandBackward and ComputeAll exhibit substantial HARD proportions (39% and 43%, respectively), suggesting that difficult queries in TMDB often require recovering missing prerequisites and comparing many alternative routes. Correspondingly, ComputePathCost and ComputeShortPath are also used more in hard cases (35% and 30%), consistent with cost-aware selection among competing tool chains.

**Spotify: resource-connected graphs enable faster convergence and more shortcuts.** Spotify is more resource-connected (61 tools, 266 resources, 1,949 edges; tool/resource  $\approx 0.23$ ), which typically reduces long cross-tool depen-

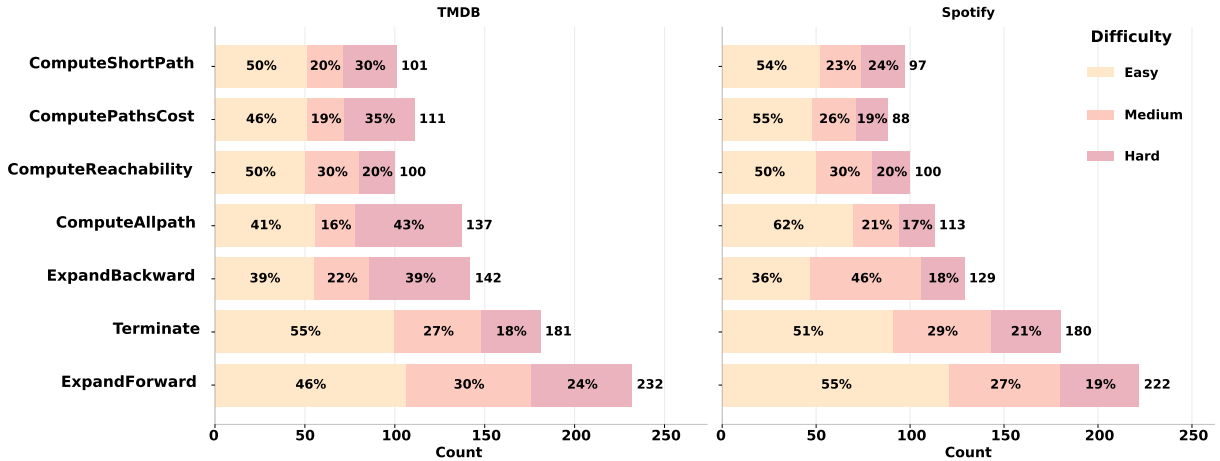


Figure 3: Distribution of tools used by TOPT across different graph domains.

dependency chains. This is reflected by fewer Hard-heavy operations and more early termination: ExpandForward remains the most frequent tool (222 calls) but is skewed toward EASY (55%), and Terminate is similarly dominated by EASY/MEDIUM (51%/29%), indicating earlier convergence. Notably, ExpandBackward peaks in MEDIUM (46%), suggesting that in Spotify, anchoring from target-side requirements can quickly narrow candidates without extensive forward exploration. Moreover, ComputeAllPath is predominantly used in EASY cases (62%), implying that path enumeration often acts as a shortcut when the reachable subgraph already contains a small number of viable routes.

**Takeaway.** Together, these patterns show that TOPT adapts its tool-chain strategy to graph structure: it favors multi-step expand-and-verify behavior on tool-dense graphs (TMDB), and shifts toward shorter trajectories with more direct convergence (and occasional shortcut reasoning) on resource-connected graphs (Spotify). This aligns with our design goal of adjusting exploration and selection effort to the effective complexity of the underlying tool graph.

## E Tool-Graph Environment

We implement tool learning as a text-based, episodic decision process over a directed tool graph. The environment maintains a global tool graph  $G = (V, E)$  and a dynamically constructed subgraph  $G_k \subseteq G$  that the agent incrementally expands and then reasons over. Nodes are typed as *resource* or *tool*; directed edges represent input-output dependencies (resource  $\rightarrow$  tool for argu-

ments, tool  $\rightarrow$  resource for produced outputs). We implement the environment with NetworkX for dependency bookkeeping and path computations.

### E.1 Episode Initialization

Each episode corresponds to a benchmark task specified by: (i) a natural-language task description, (ii) a set of known nodes  $S$  (resources initially available to the agent), and (iii) a target node  $t$  (the required resource to produce). The environment initializes Stage I (extend) with a minimal subgraph containing  $S \cup \{t\}$ , empty edges, and sets the focus to the target node. To make tool calls compact and robust to long node names, the environment assigns a deterministic identifier to every node in the global graph using a bijection: resources are mapped to  $R_0, R_1, \dots$  and tools to  $T_0, T_1, \dots$ . The agent interacts using these identifiers, and the environment maps them back to the original node names internally.

### E.2 Observations and Prompt Construction

At each step, the environment returns a textual observation (prompt) that concatenates: (1) the task description, current stage, and current focus node; (2) the current subgraph  $G_k$  rendered as a node list (#Nodes) and a compact edge list (#Edges), where tool nodes are shown in a functional form  $T_j(\text{arg\_ids}) \rightarrow \text{out\_ids}$ ; (3) optional graph-derived feedback stored in `nodes_information`, such as reachability signals and path candidates/costs (when available); and (4) the full interaction history (previous tool calls and execution failures). This design exposes the evolving frontier and intermediate reasoning evidence while keeping the action interface purely

**Query:**

Could you provide Christopher Nolan's top three directed films?

**Step 1**

<SubGraph Info.>

#Nodes: R1: Person\_Name, R2: Movie\_MovieTitle

#Edges: None

<Think>

Task at hand: "Could you provide Christopher Nolan's top three directed films?". We will first resolve "Christopher Nolan" to Person\_Name. Directly expanding Person\_Name will make the subgraph very large. I can try expanding backward on the target node R2: Movie\_MovieTitle first.

<Action>

{'tool': 'ExpandBackward', 'args': {'resource': 'Movie\_MovieTitle'}}

**Step 2**

<SubGraph Info.>

#Nodes: R1: Person\_Name, R2: Movie\_MovieTitle, T1: Movie-popular-list, T2: Movie-MovieOverview

#Edges: T1() -> R2, T2() -> R2

<Think>

I now know there are two tool nodes pointing to the target resource, but I haven't retrieved the input nodes for these two tools yet. Perhaps I could try expanding R1: Person\_Name.

<Action>

{'tool': 'ExpandForward', 'args': {'resource': 'Person\_Name'}}

**Step 3**

<SubGraph Info.>

#Nodes: R1: Person\_Name, R2: Movie\_MovieTitle, T1: Movie-popular-list, T2: Movie-MovieOverview, T3: Person-Details, T4: Search-PersonAge, R3: Person\_Department, R4: Person\_PersonID, R5: Person\_PersonPopularity,...

#Edges: T1() -> R2, T2() -> R2, T3(R1)->R3, R4, R5, T4(R1)->R7,R5->R9...

<Think>

I have discovered that R5: Person\_PersonPopularity in the current subgraph may be the key node for completing the task, and I should immediately perform Expandforward(Person\_PersonPopularity).

<Action>

{'tool': 'ExpandForward', 'args': {'resource': 'Person\_PersonPopularity'}}

**Step 4**

<SubGraph Info.>

#Nodes: R1: Person\_Name, R2: Movie\_MovieTitle, T1: Movie-popular-list, T2: Movie-MovieOverview, T3: Person-Details, T4: Search-PersonAge, R3: Person\_Department, R4: Person\_PersonID, R5: Person\_PersonPopularity, R9: Movie\_MovieTitle, T3: Movie-popular-list, T4: Movie-MovieOverview, T5: Search-Movie, T6: Search-Keyword, T7: Search-collection...

#Edges: T1(R1)->R3, R4, R5, T2(R1)->R7,R5->R9, T3(R9)->R14,R15,R16...

<Think>

I should make sure that R1 and R2 in the current subgraph are truly connected, so that I don't forget some required nodes.

<Action>

{'tool': 'ComputeReachability', 'args': {'node': 'Person\_Name', 'node': 'Movie\_MovieTitle'}}

**Step 5**

<SubGraph Info.>

#Nodes: R1: Person\_Name, R2: Movie\_MovieTitle, T1: Movie-popular-list, T2: Movie-MovieOverview, T3: Person-Details, T4: Search-PersonAge, R3: Person\_Department, R4: Person\_PersonID, R5: Person\_PersonPopularity, R9: Movie\_MovieTitle, T3: Movie-popular-list, T4: Movie-MovieOverview, T5: Search-Movie, T6: Search-Keyword, T7: Search-collection...

#Edges: T1(R1)->R3, R4, R5, T2(R1)->R7,R5->R9, T3(R9)->R14,R15,R16...

<Think>

I discovered that my subgraph already has a complete path from Person\_Name to Movie-MovieTitle, and I should terminate it immediately.

<Action>

{'tool': 'Terminate', 'args': {'text': 'Finished expand.'}}

Figure 4: A typical case of TOPT on Stage I: Progressive Graph Expansion.

**Query:**

Could you provide Christopher Nolan's top three directed films?

----- **Step 6** -----

<SubGraph Info.>

#Nodes: R1: Person\_Name, R2: Movie\_MovieTitle, T1: Movie-popular-list, T2: Movie-MovieOverview, T3: Person-Details, T4: Search-PersonAge, R3: Person\_Department, R4: Person\_PersonID, R5: Person\_PersonPopularity, R9: Movie\_MovieTitle, T3: Movie-popular-list, T4: Movie-MovieOverview, T5: Search-Movie, T6: Search-Keyword, T7: Search-collection...

#Edges: T1(R1)->R3, R4, R5, T2(R1)->R7,R5->R9, T3(R9)->R14,R15,R16...

<Think>

I'm now moving on to the second phase. I should first find all the solution paths in the subgraph.

<Action>

{'tool':'ComputeAllPath', 'args': {'Node': 'Person\_Name', 'Node': 'Movie\_MovieTitle'}}

----- **Step 7** -----

<SubGraph Info.>

#Nodes: R1: Person\_Name, R2: Movie\_MovieTitle, T1: Movie-popular-list, T2: Movie-MovieOverview, T3: Person-Details, T4: Search-PersonAge, R3: Person\_Department, R4: Person\_PersonID, R5: Person\_PersonPopularity, R9: Movie\_MovieTitle, T3: Movie-popular-list, T4: Movie-MovieOverview, T5: Search-Movie, T6: Search-Keyword, T7: Search-collection, ...

#Edges: T1(R1)->R3, R4, R5, T2(R1)->R7,R5->R9, T3(R9)->R14,R15,R16...

#Path1: [R1, T1, R5,R2], Path2: [R1, T3, R5, R2],...

<Think>

I found that there are two paths of the same length in the current subgraph. I should use ComputePathCost to calculate which path has a lower cost.

<Action>

{'tool':'ComputePathCost', 'args': {'Path': '[R1,T1,R5,R2]', 'Path': '[R1,T3,R5,R2]'}}

----- **Step 8** -----

<SubGraph Info.>

#Nodes: R1: Person\_Name, R2: Movie\_MovieTitle, T1: Movie-popular-list, T2: Movie-MovieOverview, T3: Person-Details, T4: Search-PersonAge, R3: Person\_Department, R4: Person\_PersonID, R5: Person\_PersonPopularity, R9: Movie\_MovieTitle, T3: Movie-popular-list, T4: Movie-MovieOverview, T5: Search-Movie, T6: Search-Keyword, T7: Search-collection, ...

#Edges: T1(R1)->R3, R4, R5, T2(R1)->R7,R5->R9, T3(R9)->R14,R15,R16...

#Path1: [R1, T1, R5,R2], Path2: [R1, T3, R5, R2],...

#Cost: [R1, T1, R5,R2]: 5.93s, [R1, T3, R5, R2]: 4.53s

<Think>

The path [R1,T3,R5,R2] takes the shortest time, so I should return to that path. I need to first map the corresponding node back and then call Terminate to end the process.

<Action>

{'tool':'Terminate', 'args': {'Path': '[Person\_Name, Movie-popular-list, Person\_PersonPopularity, Movie\_MovieTitle]'}}

Figure 5: A typical case of TOPT on Stage II: Progressive Graph Reasoning.

tool-driven.

### E.3 Action Space and Tool-Call Interface

An action is a JSON tool call  $\{\text{tool} : \cdot, \text{args} : \cdot\}$ . The environment parses the JSON, validates that the tool name is supported and the argument keys match the tool signature, and converts node identifiers (e.g., R7) back to internal node names. Invalid formatting or signature mismatches are treated as failed actions.

## F Graph Processing Toolkit

To support the proposed progressive graph expansion (Stage I) and progressive graph reasoning (Stage II), we design a graph processing toolkit consisting of nine tools (including a stage specific Terminate in each stage)(Table 7). All tools are implemented with NetworkX and operate on the currently constructed subgraph, enabling incremental expansion, reachability checking, and cost-aware planning. Intuitively, Stage I focuses on constructing a compact solvable subgraph that preserves only task relevant dependencies from a large tool graph, while Stage II performs optimization oriented reasoning over this fixed subgraph to select a optimal tool chain. Below we describe each tool in detail.

### F.1 Stage I: Progressive Graph Expansion

- **ExpandForward(Node)**. This tool performs forward expansion starting from a resource node. It searches for tool nodes that can consume the given resource as input and then adds these tools and their output resources into the current subgraph. In other words, it answers: “Given an available resource, which tools can use it next, and what new resources can be produced?” This operation expands the frontier of reachable candidates under a large action space.
- **ExpandBackward(Node)**. This tool traces predecessors that can produce a given node. Importantly, ExpandBackward accepts either a resource node or a tool node. If the input is a resource node, it retrieves all upstream tools that generate this resource and introduces their required input resources. If the input is a tool node, it traces the tool’s prerequisite input resources and, recursively, their potential producers. This backward expansion complements ExpandForward by recovering missing dependencies needed to reach a target.

- **JumpToNode(Node)**. This tool switches the exploration focus to another node in the current subgraph (e.g., a target side node or a promising intermediate node) when the search is stuck locally or when a different expansion direction is needed. It improves efficiency by avoiding excessive expansion along unproductive branches.
- **ComputeReachability(Node1, Node2)**. This tool checks reachability between two nodes within the current subgraph, i.e., whether a directed path exists from Node1 to Node2. Optionally, it can return feasible paths as evidence. In Stage I, it serves as the primary solvability criterion: once the source and target become connected, the constructed subgraph already contains at least one feasible tool chain.
- **Terminate(text)**. The Stage I termination tool ends the expansion process and outputs a textual summary, such as the constructed task-relevant solvable subgraph or a failure message when the budget is exhausted, and the target remains unreachable.

### F.2 Stage II: Progressive Graph Reasoning

- **ComputeAllPath(Node1, Node2)**. This tool enumerates all feasible paths from Node1 to Node2 on the fixed subgraph (typically restricting to simple paths to avoid cycles), producing a candidate set of tool chains for subsequent selection.
- **ComputeShortPath(Node1, Node2)**. This tool computes shortest paths between Node1 and Node2 (e.g., by hop count or a user-defined length metric), providing length-efficient tool chains and serving as a strong heuristic baseline for reasoning.
- **ComputePathCost(path)**. This tool evaluates the cost of a given path and returns a scalar score. The cost function can incorporate multiple factors such as tool latency, execution cost, and path-length penalties, enabling cost-aware comparison across multiple feasible tool chains.
- **Terminate(text)**. The Stage II termination tool outputs the final selected tool chain (and optionally its cost/justification) and ends the reasoning process.

Table 7: Summary of the graph-processing tools used in Stage I (progressive graph expansion) and Stage II (progressive graph reasoning). All tools are implemented with NetworkX and operate on the current subgraph.

Tool	Stage	Input	Function / Output
ExpandForward	I	Resource node $r$	Finds tools that <i>consume</i> $r$ as input and adds these tools and their output resources to the current subgraph (frontier expansion).
ExpandBackward	I	Resource node $r$ or tool node $\tau$	Retrieves upstream producers: if input is $r$ , finds tools that <i>produce</i> $r$ and adds their required inputs; if input is $\tau$ , traces its prerequisite input resources and their potential producers (dependency completion).
JumpToNode	I	Any node $v$	Switches the exploration focus to $v$ as a new expansion anchor to escape local dead ends or change expansion direction.
ComputeReachability	I	Two nodes $(v_s, v_t)$	Checks whether a directed path exists from $v_s$ to $v_t$ in the current subgraph; optionally returns feasible path(s) as evidence (solvability check).
Terminate	I	Text	Stops Stage I and outputs a summary (e.g., constructed solvable subgraph or failure due to budget/unreachability).
ComputeAllPath	II	Two nodes $(v_s, v_t)$	Enumerates all feasible paths from $v_s$ to $v_t$ on the fixed subgraph (typically simple paths), producing candidate tool chains.
ComputeShortPath	II	Two nodes $(v_s, v_t)$	Computes shortest path(s) from $v_s$ to $v_t$ (e.g., by hop count or a chosen length metric), yielding length-efficient tool chains.
ComputePathCost	II	A path $p$	Evaluates the cost of $p$ and returns a scalar score (e.g., combining tool latency/cost and length penalties) for cost-aware selection.
Terminate	II	Text	Stops Stage II and outputs the selected final tool chain (optionally with cost/justification).

## G Broader Impact

TOPT’s ability to learn cost-aware tool-use policies over large tool graphs can enhance the reliability and efficiency of tool-augmented AI systems in practical settings such as enterprise workflow automation, customer support, and data-intensive analytics. By explicitly selecting near-optimal tool chains under context and compute constraints, TOPT can reduce unnecessary tool calls and latency, helping systems retrieve relevant information more efficiently, execute multi-step tasks with fewer failures, and provide more consistent responses when multiple valid tool combinations exist. Its capability to generalize to unseen tools and novel compositions is particularly valuable in real-world tool ecosystems that evolve over time, where new APIs, resources, and integrations are continuously introduced and manual retuning is costly. Moreover, compared to approaches that rely on scaling up model size or retraining for each toolset, TOPT’s framework-level efficiency can lower deployment and maintenance overhead, which may translate into more environmentally sustainable deployments by reducing repeated training and ex-

cessive inference-time exploration. Overall, these advances contribute to more accessible, efficient, and dependable tool-learning systems that can support a wide range of real-world applications.