Large Language Models Domain Adaptation Methods for solving complex problems

Beliakin Evgeniy
beliakin.eo@phystech.edu

Center for Cognitive Modeling, MIPT

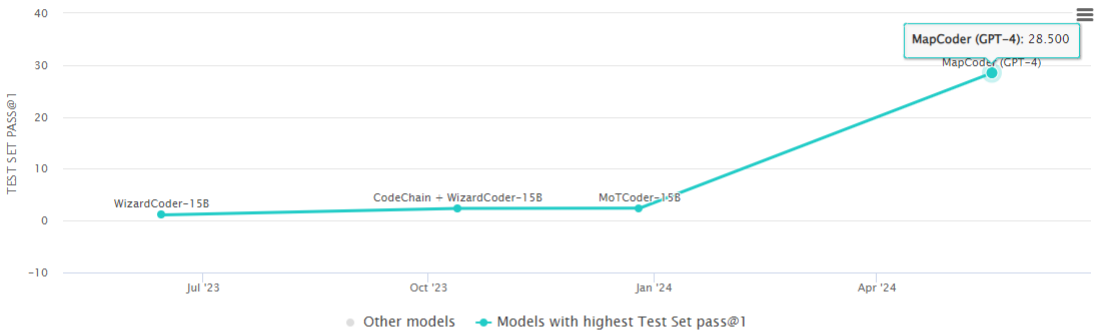
2025, Moscow

Motivation

Code Generation on CodeContests

Leaderboard | Dataset

View: Test Set pass@1 by Date



Filter: untagged

Edit Leaderboard

Rank	Model	Test Set pass@1	Test Set pass@5	Val Set pass@1	Val Set pass@5	Paper	Code	Result	Year	Tags
1	**MapCoder** (GPT-4)	28.5	35.2	28.5		MapCoder: Multi-Agent Code Generation for Competitive Problem Solving			2024	
2	**MoTCoder-15B**	2.39	7.69	6.18	12.73	MoTCoder: Elevating Large Language Models with Modular of Thought for Challenging Programming Tasks			2023	
3	**CodeChain + WizardCoder-15B**	2.35	3.29	2.48	3.30	CodeChain: Towards Modular Code Generation Through Chain of Self-revisions with Representative Sub-modules			2023	
4	**WizardCoder-15B**	1.11	3.18	1.98	3.27	WizardCoder: Empowering Code Large Language Models with Evol-Instruct			2023	

Figure: CodeContests Benchmark[1]

[1] https://paperswithcode.com/sota/code-generation-on-codecontests

Goal

LLM domain adaptation for complex problem-solving.

The main research objective is to develop domain adaptation method for Large Language Models (LLMs) based on model memory augmentation.

Tasks:

- Review and analyze state-of-the-art domain adaptation techniques.
- Prepare a competitive programming data set and relevant data sources.
- Conduct a comparative analysis of domain adaptation methods based on performance metrics in competitive programming tasks.
- Develop memory-based approach for solving complex tasks.

Problem statement

Adapt large language models (LLMs) to effectively solve competitive programming tasks.

Definitions:

- $\mathcal{M}$ — large language model parameterized by weights θ.
- $\mathcal{T}$ — target domain, specifically competitive programming tasks.
- $\mathcal{D}$ — domain-specific dataset containing problems P_i with correct solutions S_i.
- $\mathcal{A}$ — domain adaptation method applied to the model $\mathcal{M}$, including multi-agent architectures, fine-tuning, and prompt-based approaches.

Objective function:

$$\max_{\theta,\mathcal{A}} Q(\mathcal{M}, \mathcal{A}, \mathcal{D}) = \mathbb{E}_{(P_i,S_i)\sim\mathcal{D}}[\text{solve}(\mathcal{A}(\mathcal{M}, P_i), S_i)] \tag{1}$$

Where:

- $\text{solve}(X, S_i)$ — function indicating correctness of solution X for problem P_i compared to solution S_i.

Metrics:

- Accuracy (percentage of correctly solved tasks).
- Execution correctness on CodeContests benchmark.

Datasets

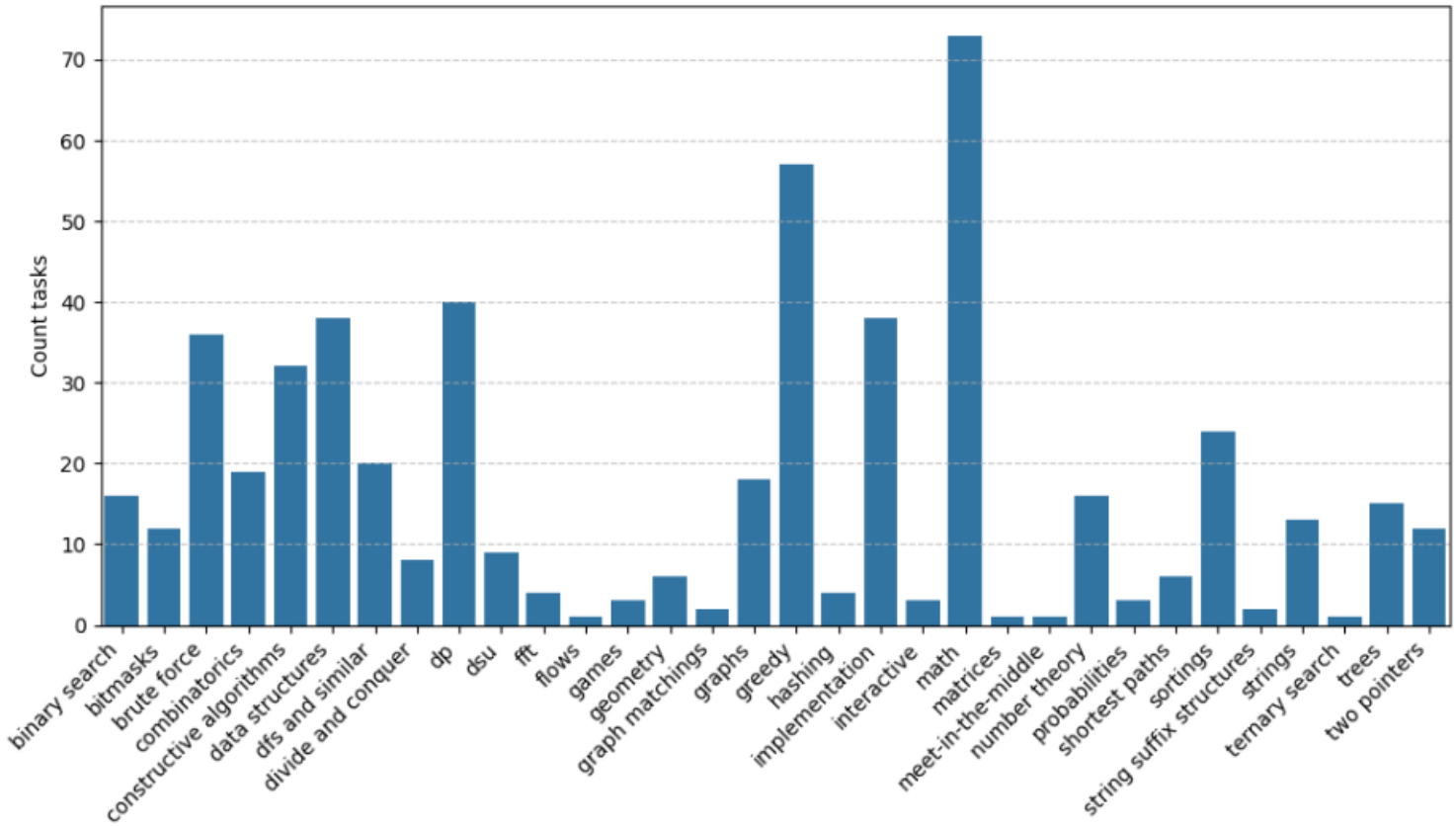


Figure: Algorithm tags distribution for test part of codecontest dataset.

- Code Contests - tasks from Codeforces[2]

[2] https://codeforces.com

Problem Example

Backspace

You are given two strings s and t, both consisting of lowercase English letters. You are going to type the string s character by character, from the first character to the last one.

When typing a character, instead of pressing the button corresponding to it, you can press the "Backspace" button. It deletes the last character you have typed among those that aren't deleted yet (or does nothing if there are no characters in the current string). For example, if s is "abcbd" and you press Backspace instead of typing the first and the fourth characters, you will get the string "bd" (the first press of Backspace deletes no character, and the second press deletes the character 'c'). Another example, if s is "abcaa" and you press Backspace instead of the last two letters, then the resulting text is "a".

Your task is to determine whether you can obtain the string t, if you type the string s and press "Backspace" instead of typing several (maybe zero) characters of s.

Input

The first line contains a single integer q ($1 \le q \le 10^5$) the number of test cases.
The first line of each test case contains the string s ($1 \le |s| \le 10^5$). Each character of s is a lowercase English letter.
The second line of each test case contains the string t ($1 \le |t| \le 10^5$). Each character of t is a lowercase English letter.
It is guaranteed that the total number of characters in the strings over all test cases does not exceed $2 \cdot 10^5$.

Output

For each test case, print "YES" if you can obtain the string t by typing the string s and replacing some characters with presses of "Backspace" button, or "NO" if you cannot.
You may print each letter in any case (YES, yes, Yes will all be recognized as positive answer, NO, no and nO will all be recognized as negative answer).

Example Input

```
4
ababa
ba
ababa
bb
aaa
aaaa
aababa
ababa
```

Example Output

```
YES
NO
NO
YES
```

Explanation

In order to obtain "ba" from "ababa", you may press Backspace instead of typing the first and the fourth characters.

There's no way to obtain "bb" while typing "ababa".

There's no way to obtain "aaaa" while typing "aaa".

In order to obtain "ababa" while typing "aababa", you have to press Backspace instead of typing the first character, then type all the remaining characters.

Figure: Competitive programming problem statement[3]

[3] Li et al., "Competition-level code generation with AlphaCode".

Approaches

Domain adaptation methods for LLMs:

Without weights changing:

- Prompt engineering
- Retrieval-augmented generation (RAG)
- Agent-based architectures

With weights changing:

- Supervised fine-tuning (SFT)
  - Parameter-efficient fine-tuning (PEFT)
  - Adapter modules
- Reinforcement learning (GRPO)

Key criteria for method comparison:

- Task performance improvement

Textbooks are all you need[4]

```
To begin, let us define singular and nonsingular matrices. A matrix is said to be singular if its
determinant is zero. On the other hand, a matrix is said to be nonsingular if its determinant is not
 zero. Now, let's explore these concepts through examples.

Example 1: Consider the matrix A = np.array([[1, 2], [2, 4]]). We can check if this matrix is
singular or nonsingular using the determinant function. We can define a Python function, `
is_singular(A)`, which  returns true if the determinant of A is zero, and false otherwise.

import numpy as np
def is_singular(A):
    det = np.linalg.det(A)
    if det == 0:
        return True
    else:
        return False

A = np.array([[1, 2], [2, 4]])
print(is_singular(A)) # True
```

Figure: The synthetic textbook dataset example

[4] Gunasekar et al., *Textbooks Are All You Need*.

Competitive Programming with Large Reasoning Models

OpenAI research highlights (IOI 2024):

- Reinforcement learning (RL) significantly enhances LLM performance in competitive programming.
- Three model variants evaluated:
  - **o1**: General-purpose RL-trained model.
  - **o1-ioi**: Domain-specific fine-tuned model with handcrafted inference strategies.
  - **o3**: Advanced general-purpose RL model without handcrafted heuristics.
- Results on IOI 2024:
  - **o1-ioi**: Gold medal performance under relaxed constraints (49th percentile live).
  - **o3**: Achieved gold medal without handcrafted strategies under strict constraints (99.8th percentile on Codeforces).

Main conclusions

- General-purpose RL models outperform specialized domain-specific systems.
- Scaled RL enables models to autonomously discover efficient inference strategies.

MapCoder[5,6]

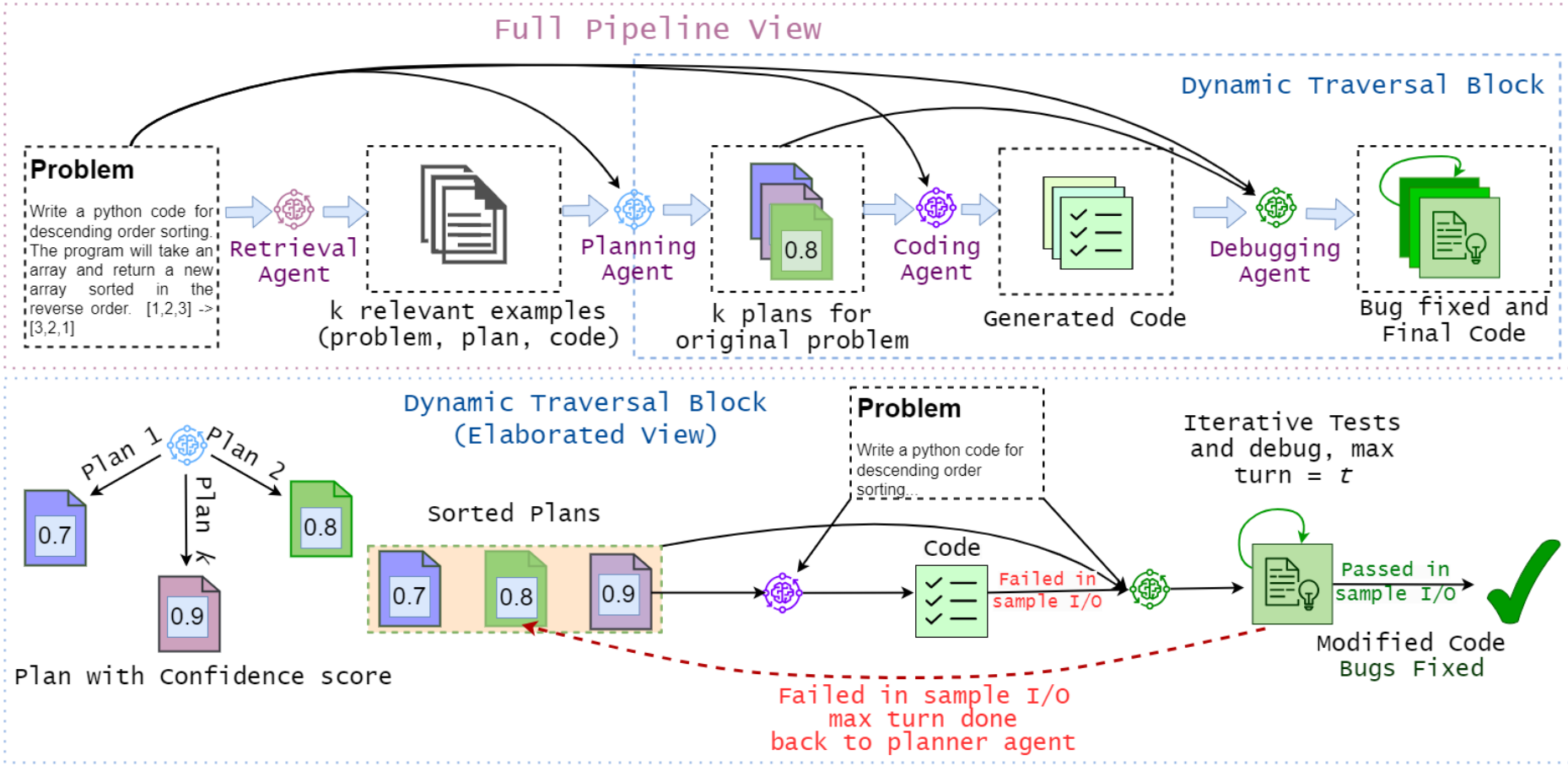


Figure: Overview of MapCoder

[5] Islam, Ali, and Parvez, *MapCoder: Multi-Agent Code Generation for Competitive Problem Solving*.

[6] SOTA on CodeContests Benchmark

Uncertainty Guided Global Memory Improves Multi-Hop Question Answering

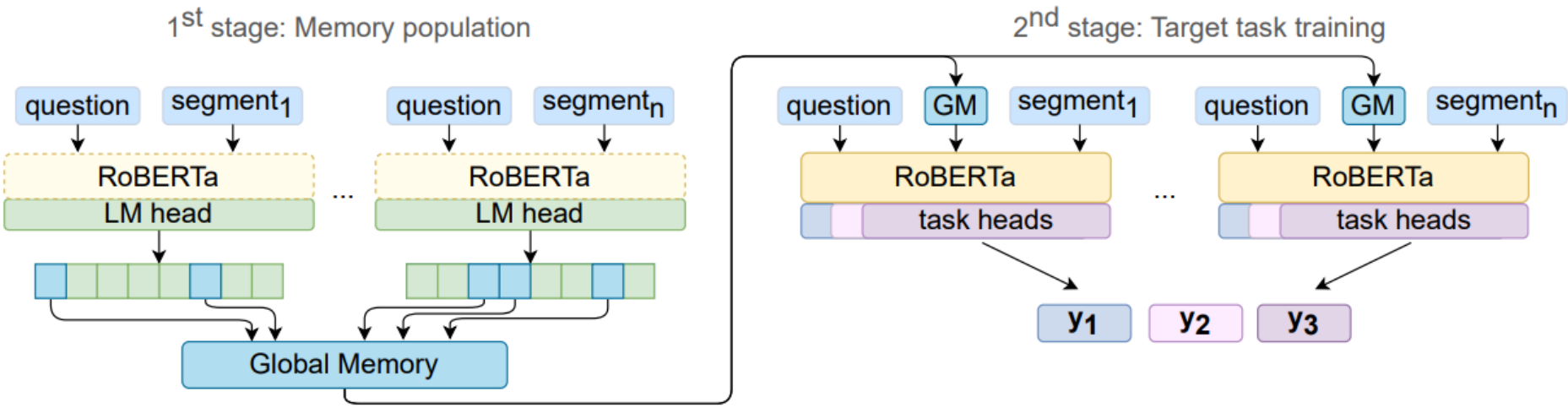


Figure: The input segments are processed by RoBERTa+LM head to generate prediction distributions for uncertainty estimation. Global memory is populated based on the given entropy condition.

Experiments. Dataset and Domain Separation

Original Dataset: CodeForces problems with diverse algorithmic tags.

Grouped into domains for specialized LLM adaptation:

- **Graphs**
- **Dynamic Programming**
- **Data Structures**
- **Search & Optimization**
- **Greedy Algorithms**
- **Constructive Algorithms**
- **Graphs**
- **Strings**
- **Math**

Experiments: Metrics

Method	**pass@10**	
	test (Python)	test (C++17)
phi-4 baseline	4.5% (2/44)	22.27% (10/44)
phi-4 + PE (Specialization)	9% (4/44)	22.27% (11/44)
phi-4 SFT (LoRA)	13.63% (6/44)	-
Olympic-coder-7b	-	43.18% (19/44)

Domain: Dynamic Programming.

Dataset: Deepmind/code-contests.

Base model: Microsoft/Phi-4 (14b).

Experiments: Metrics

Method	**pass@10**	
	test (Python)	test (C++17)
phi-4 baseline	9.67% (3/31)	25.8% (8/31)
phi-4 + PE (Specialization)	9.67% (3/31)	25.8% (8/31)
phi-4 SFT (LoRA)	12.9% (4/31)	-
Olympic-coder-7b	-	41.93% (13/31)

Domain: Graphs.

Dataset: Deepmind/code-contests.

Base model: Microsoft/Phi-4 (14b).

Conclusion and Future Work

Conclusions:

- Reasoning-enhanced methods significantly improve model performance, surpassing baseline approaches in solving competitive programming tasks.
- Domain-specific fine-tuning methods (LoRA, PEFT) do not achieve improvements and are insufficient for highly complex reasoning tasks without additional strategies.
- Prompt Engeneering specialization approach has a perfomance improvment in comparison baseline without specialization.

Future Work:

- Further experiments to optimize **RAG** techniques, identifying optimal data sources and retrieval strategies.
- Explore advanced specialized architectures, including multi-agent systems and cooperative reasoning frameworks.

Thank you for your attention!

beliakin.eo@phystech.edu