Министерство науки и высшего образования Российской Федерации
Московский физико-технический институт
(национальный исследовательский университет)
Физтех-школа прикладной математики и информатики
Магистерская программа
«Методы и технологии искусственного интеллекта»

Студент Белякин Евгений Олегович

Методы доменной адаптации больших языковых моделей для решения комплексных задач

03.04.01 — Прикладные математика и физика

Магистерская диссертация

Научный руководитель:
Сагирова Алсу Рафаэлевна
к.ф.-м.н., научный сотрудник

Москва
2025 г.

Содержание

1 Введение **5**

2 Обзорно-постановочный раздел работы **6**

2.1 Основные понятия и определения 6

2.2 Методы адаптации без дообучения модели 6

2.2.1 Prompt Engineering 6

2.2.2 Генерация текста с использованием внешних источников (Retrieval-Augmented Generation) 8

2.2.3 Агентные системы и многошаговые подходы к решению задач . 9

2.3 Методы с изменением весов модели 12

2.3.1 Supervised Fine-tuning 12

2.3.2 Параметро-эффективные методы дообучения (адаптеры, LoRA) 14

2.3.3 Обучение с подкреплением (RL, RLHF) 16

2.3.4 Генерация и использование синтетических данных 18

2.4 Выводы 20

3 Основной раздел работы **23**

3.1 Формальная постановка задачи 23

3.2 Доменная таксономия и исходный корпус задач 25

3.3 Исследование методов доменной адаптации 27

3.3.1 Prompt Engineering 27

3.3.2 Supervised Fine-Tuning (SFT) 28

3.3.3 Retrieval-Augmented Generation (RAG) 29

4 Вычислительные эксперименты **32**

4.1 Supervised Fine-Tuning 32

4.2 RAG 35

4.3 Результаты экспериментов 36

5 Заключение **42**

A Инструкции для доменных экспертов **47**

B Пример олимпиадной задачи **50**

C Пример решения из синтетического датасета **51**

Аннотация

Методы доменной адаптации больших языковых моделей (LLM) позволяют существенно повысить их эффективность в узкоспециализированных задачах. Одной из таких задач является олимпиадное программирование, требующее глубокого понимания алгоритмических парадигм и способности к генерации точных решений.

В работе исследуются методы адаптации LLM к задачам олимпиадного программирования на примере классов: динамическое программирование, жадные алгоритмы, структуры данных, графы и другие. Реализованы подходы, включающие генерацию синтетических решений с пояснениями, дообучение моделей с использованием LoRA, а также интеграцию внешних источников знаний в рамках Retrieval-Augmented Generation (RAG).

Эксперименты на датасете CodeContests показали стабильный прирост метрики pass@1 при использовании RAG во всех предметных областях. Одновременно наблюдается снижение вариативности по pass@10, что указывает на сужение пространства решений. Анализ распределения сложности решаемых задач подтверждает, что адаптация смещает модель из режима «широкого, но поверхностного» поиска к режиму «глубокой, но узкоспециализированной» экспертизы.

Полученные результаты демонстрируют потенциал построения адаптивных мультидоменных агентных систем на основе LLM с динамическим выбором стратегии генерации решений.

1 Введение

Большие языковые модели (Large Language Models, LLM), основанные на архитектуре трансформер, показали высокую эффективность во многих задачах обработки естественного языка, включая генерацию текстов, машинный перевод, вопросно-ответные системы и т.д. [5]. Тем не менее, при переходе к узкоспециализированным областям знания, требующим сложных рассуждений, LLM без соответствующей адаптации могут проявлять ограниченную точность и устойчивость. Одним из примеров таких областей являются задачи олимпиадного программирования, где требуется не только понимать формулировку задачи, но и уметь формировать корректный и эффективный алгоритмический код, удовлетворяя ограничениям по времени и памяти.

Наиболее перспективным направлением преодоления этих ограничений является доменная адаптация больших языковых моделей, то есть дообучение или настройка предварительно обученного трансформера на данных, характерных для рассматриваемой предметной области [8, 18, 3, 12]. Благодаря этому подходу модели обретают более глубокое понимание сложных формулировок и умений, требуемых для алгоритмических задач. Это особенно важно при разработке интеллектуальных систем, способных помогать в написании кода, решать тестовые примеры и участвовать в соревнованиях на уровне экспертов.

Актуальность темы данной работы связана с возрастающей потребностью в разработке эффективных методов автоматической генерации решений для сложных задач, требующих многошагового логического вывода. В частности, успешная адаптация LLM к олимпиадным задачам открывает новые горизонты в области гибридных систем человек+машина, обеспечивая разработчикам и исследователям мощные инструменты для упрощения и ускорения процесса программирования. Целью настоящей работы является исследование и сравнение различных методов доменной адаптации LLM на примере задач олимпиадного программирования, а также формальная постановка проблемы и анализ факторов, влияющих на качество решения.

2 Обзорно-постановочный раздел работы

В данном разделе будут рассмотрены актуальные методы адаптации LLM в контексте задачи генерации кода.

2.1 Основные понятия и определения

LLM - Large Language Model

CPT - Continual Pre-training

SFT - Supervised Fine-tuning

PEFT - Parameter-efficient Fine-tuning

RAG - Retrieval-Augmented Generation

PT - Prompt tuning

GRPO - Group Relative Policy Optimization

2.2 Методы адаптации без дообучения модели

В первую группу входят методы, которые позволяют использовать возможности уже имеющейся LLM для решения задач целевого домена, не производя затратного обновления всех параметров модели. Такие методы особенно актуальны, когда базовая модель очень велика (сотни миллиардов параметров) и дообучение на новых данных затруднительно или нежелательно. Ниже рассмотрены основные подходы этого типа: от простого подбора и форматирования подсказок до сложных многошаговых агентных схем с подключением внешних инструментов.

2.2.1 Prompt Engineering

Одним из наиболее прямых способов адаптировать LLM к новой задаче является инженерия подсказок (prompt engineering). Модель, обученная на больших объёмах текста и кода, зачастую способна решить новую задачу, если её правильно сформулировать во входной инструкции. Для олимпиадного программирования это означает, что можно детально описать условие задачи, указать формат ввода-вывода и явным образом запросить решение (например, с указанием языка программирования). Известно, что большие модели способны к обучению по контексту (in-context learning), когда предоставление нескольких примеров задач с решениями в инструкции значительно улучшает результат на новой аналогичной задаче [5]. В контексте генерации

кода этот приём выражается в передаче модели одного или нескольких примеров пар «описание задачи – корректный код»; продолжая такой контекст, LLM порождает код для новой, ещё не виденной задачи. Авторы [13] отмечали, что даже модели вроде GPT-Neo, не специализированные под соревнования, могут решить до 20% простых задач, если в подсказке дать хорошую формулировку проблемы и примеры решений аналогичного типа. Ещё более эффективным оказалось применение цепочек рассуждений (chain-of-thought prompting). В исследовании [16] показано, что если явно попросить модель рассуждать пошагово, перед тем как дать окончательный ответ, то на сложных логических и математических задачах качество решений значительно возрастает. Для программирования подобный подход тоже полезен. В частности, отчёт OpenAI по GPT-4 [30] демонстрирует, что использование многошагового рассуждения и разбивки задачи на подзадачи повышает процент решённых моделью задач по сравнению с прямым одношаговым выводом. Модель можно попросить сначала описать план решения (например: «анализируем условие; определяем подходящий алгоритм; реализуем решение на Python»), а затем на основе этого плана сгенерировать финальный код. Такой приём помогает модели явным образом структурировать алгоритм перед написанием кода. Согласно отчёту, добавление явной цепочки мыслей принесло порядка +10% улучшения показателя pass@1 на задачах, требующих сложного мног шагового рассуждения, включая некоторые задачи кодирования [30]. Кроме того, псевдокодирование (генерация комментариев или псевдокода перед собственно кодом) служит аналогом chain-of-thought для кода, облегчая модели процесс разработки решения. Отдельным направлением является soft prompt tuning – адаптация входной инструкции без изменения параметров модели. Авторы статей [10, 11] предложили методы, позволяющие обучать небольшое векторное префиксное представление, которое добавляется ко входной последовательности, в то время как сама LLM (например, GPT-3) остаётся неизменной. Такой «встраиваемый префикс» способен нести информацию о целевом задании или домене. В задачах генерации кода этот подход потенциально позволяет адаптировать модель к стилю решений или специфике вывода, не затрагивая сотни миллиардов базовых параметров. Хотя большинство работ по prompt tuning демонстрировали эффективность на задачах NLU (вопрос-ответ, классификация), имеются попытки применить его и для кода. Например, в исследовании [34] экспериментировали с prompt tuning для задач кодогенерации и показали, что даже без обновления весов модели можно добиться улучшения метрик на специализированных датасетах, если выучить оптимальный префикс для этих

задач. Тем не менее, применительно к сложным задачам, требующим изобретения нового алгоритма, подходы, основанные только на подсказках, ограничены знаниями и навыками, уже присутствующими в модели. Если в тренировочных данных LLM не встречалось аналогичного алгоритмического паттерна, никакая инженерия подсказок не заставит модель изобрести его с нуля. Таким образом, методы данного раздела, хотя и просты в применении, служат скорее первым приближением: они могут активировать «скрытые» знания модели о программировании, но не добавляют новых умений.

2.2.2 Генерация текста с использованием внешних источников (Retrieval-Augmented Generation)

Для решения многих задач программирования модели может не хватать конкретных знаний: например, сведений о тонкостях определённой библиотеки, о редкой структуре данных или просто примера реализации похожего алгоритма. Retrieval-Augmented Generation (RAG) представляет собой подход, при котором перед генерацией ответа модель осуществляет поиск по внешней базе знаний, чтобы подкрепить входную последовательность с описанием задачи релевантной информацией. В контексте кода таким внешним источником могут быть репозитории решений, документация API, учебные материалы или архивы соревнований. В работе [14] предложена система REDCoder, которая при генерации кода ищет в базе данных схожие фрагменты и включает их в подсказку для модели. Авторы показали, что такая система извлечения информации улучшает как качество генерируемых программ, так и качество сопутствующей документации (комментариев) при суммаризации кода. В экспериментах на задачах программирования модель с доступом к похожим решениям превзошла базовый трансформер без поиска: метрика точного совпадения решений выросла на несколько пунктов [14]. К аналогичным выводам пришли и другие исследователи. Так, авторы статьи [9] отметили, что предоставление модели примеров кода из ретривера делает сгенерированный код более информативным и полным, особенно когда задача подразумевает стандартные шаблоны (например, типичные способы чтения ввода или форматирования вывода). Кроме поиска похожих решений, возможно привлечение и других источников. D работе [27] был предложен метод DocPrompting, где в роли внешней базы знаний выступает библиотека технической документации (описания API, официальные технические руководства). Для

каждой задачи определяются потенциально полезные API или функции, после чего из документации извлекаются соответствующие разделы и добавляются в инструкцию. По сути, модель снабжается справочными материалами, уменьшая вероятность ошибок при использовании определённых функций. Этот метод продемонстрировал улучшение качества на задачах генерации кода, связанных с использованием сложных библиотечных вызовов: по сравнению с моделью без доступа к документации DocPrompting повысил показатель функциональной корректности (долю решений, успешно прошедших все тесты) благодаря наличию актуальной информации об используемых API [27]. Эффективность RAG для задач кода подтверждена недавним исследованием [37]. В нём систематически протестированы различные варианты интеграции ретривера и генератора на трёх моделях (CodeGen [26], UniXcoder [23], CodeT5 [7]) и трёх датасетах программных задач. По результатам все модели в среднем выигрывали от retrieval-augmented подхода: на некоторых наборах точность (Exact Match) выросла более чем на 40%, а метрики BLEU/CodeBLEU [4] улучшились на 8–10% по сравнению с генерацией без поиска [37]. Эти результаты свидетельствуют, что даже относительно простые приёмы (например, поиск похожих фрагментов с помощью BM25 [1] и конкатенация их с описанием задачи) дают существенный выигрыш. Однако у данного подхода есть и ограничения. Во-первых, если механизм поиска возвращает не релевантную задаче информацию, модель может быть сбита с толку и сгенерировать неверное решение [14]. Качество ретривера и стратегия отбора фрагментов являются критически важными компонентами. Во-вторых, применение RAG требует наличия подходящей базы знаний.

В данной работе рассматривается комплексный подход, основанный на формировании базы релевантных документов для различных направлений теории алгоритмов, используя разметку от большой языковой модели для определения принадлежности документа к конкретному домену или набору доменов. Более подробно рассмотренный алгоритм работы RAG отражен в секции 3.3.3.

2.2.3 Агентные системы и многошаговые подходы к решению задач

Даже при хорошем формулировании подсказки и обилии вспомогательной информации, модель может не решить сложную задачу с первой попытки. Люди в таких ситуациях задействуют итеративный процесс: написать черновое решение, протестировать его, отладить, при необходимости перепроектировать алгоритм. Возникает

идея имитировать такой подход с помощью LLM, превратив модель в своеобразного агента, способного взаимодействовать со средой (например, компилировать и запускать код, читать сообщения об ошибках, выполнять веб-поиск) и постепенно улучшать решение. В 2022 году был предложен архитектурный паттерн ReAct (Reasoning and Acting) [20], сочетающий цепочку мыслей с вызовом инструментов. В рамках ReAct модель генерирует не только текст рассуждения, но и специальные токены-действия: например, команду поиска информации или выполнения сгенерированного кода. Такой подход позволяет LLM поэтапно исследовать проблему, получать обратную связь и корректировать дальнейшие шаги. Хотя ReAct изначально демонстрировался на примерах вопросов из базы знаний, его идея легла в основу многих систем для программирования, где модель окружена инструментами (компилятором, интерпретатором, средой тестирования). Одним из ярких примеров является система AlphaCode от DeepMind [19]. Помимо самой модели генерации кода, AlphaCode включает этапы фильтрации и выбора решений. Модель сэмплирует тысячи вариантов программ, после чего автоматически проверяет их на наборах тестов (включая примеры из условия задачи) и отсеивает неправильные. Далее схожие решения кластеризуются, чтобы выбрать несколько существенно различных подходов, которые и отправляются в качестве окончательного ответа. Такая многоходовая схема (генерация множества решений + исполнение и отбор) позволила AlphaCode достичь уровня среднего участника соревнований – модель вошла в топ-54.3% на нескольких недавних контестах Codeforces [19]. В более позднем улучшении (неофициально названном AlphaCode2) разработчики практически удвоили число решаемых задач и довели виртуальный рейтинг модели до 85-го перцентиля [29]. Примечательно, что эти успехи были достигнуты не за счёт принципиально нового обучения модели, а благодаря инженерии процесса решения: широкому перебору программ и эффективному автоматическому тестированию/отбору. Другой подход сфокусирован на активном самоотладке модели. Авторы статьи [42] представили архитектуру, имитирующую отладку кода: после первичной генерации решения программа компилируется и запускается на тестах, затем модель получает обратную связь – информацию о том, какие тесты не пройдены, или даже состояния ключевых переменных во время выполнения. На основе этой информации LLM генерирует исправленный код. Цикл может повторяться, пока решение не станет корректным. Авторы сообщают, что их система существенно повысила долю полностью правильных решений на наборах MBPP и HumanEval, превзойдя даже некоторые специализированные модели [42].

Схожие результаты получены и в других исследованиях, где модели разрешают итеративно улучшать свой ответ. Например, проект Self-Refine [32] показал, что GPT-4, способный самостоятельно проанализировать сгенерированный им же код и выявить недостатки, при повторной попытке выдаёт более корректные решения. Аналогично, в статье [40] был предложен метод LEDEX, в котором модель сначала пишет решение, затем генерирует объяснение, почему код может быть неправильным, и на основе этого объяснения пытается его исправить. Такая комбинация самопроверки и рефакторинга улучшила показатель pass@1 на сложных задачах более чем на 15% после обучения модели этим навыкам [40]. Важно отметить, что агентные многошаговые подходы демонстрируют высокую эффективность, но предъявляют повышенные требования к вычислительным ресурсам и усложняют инфраструктуру. Запуск LLM в цикле с компиляцией и выполнением кода может быть медленным; генерация сотен и тысяч пробных решений, как в AlphaCode, требует значительного времени и параллельных вычислений. Кроме того, при итеративном подходе модель может «застрять» в цикле неудачных попыток исправления, если не получает достаточно информативного фидбека. В ряде случаев требуется вмешательство человека (например, подсказка, куда направить внимание в отладке) или специально обученные вспомогательные модели (критики, проверяющие решения). Тем не менее, агентный подход приближает процесс решения к человеческому и уже позволил добиться впечатляющих результатов. Совсем недавно команда OpenAI сообщила, что их экспериментальная связка моделей, усиленная стратегиями разбиения задач и самоанализа, смогла самостоятельно завоевать золотую медаль Международной олимпиады по информатике (IOI 2024) в неофициальном зачёте, решив задач больше, чем половина участников-людей [43]. Это стало возможным благодаря сочетанию общей мощности модели (предположительно, GPT-4) и специального агентного режима работы: планирования решений, запуска кода на собственно сформулированных тестах, многократной корректировки решений и т.д. Данный пример подтверждает, что без изменения весов модели, а лишь за счёт продуманного управления её поведением, можно достичь уровня, сопоставимого с лучшими экспертами в домене соревновательного программирования.

2.3 Методы с изменением весов модели

Вторая категория подходов основывается на дообучении или модификации параметров LLM для повышения производительности в целевом домене. Такие методы требуют доступа к самой модели и вычислительных ресурсов для обучения, но зачастую дают более существенный выигрыш, чем одни лишь трюки с подсказками. В этом разделе рассматриваются классическое супервизированное дообучение модели на задачах программирования, современные техники параметро-эффективной настройки (например, LoRA), методы обучения с подкреплением, а также использование синтетически сгенерированных данных для расширения тренировочной выборки в домене программирования.

2.3.1 Supervised Fine-tuning

Самый прямолинейный способ адаптировать языковую модель к новому домену – дообучить её на размеченных данных из этого домена. В контексте программирования такими данными могут служить пары *описание задачи – корректный исходный код*. Идея состоит в том, что модель, даже имея общее представление о языке программирования, значительно лучше научится решать сложные задачи, если «увидит» множество примеров реальных решений во время обучения. Одной из первых работ, подтвердивших эффективность fine-tuning на кодовых задачах, была уже упомянутая работа OpenAI над Codex [8]. Там сравнивалась исходная 12-миллиардная GPT-3 (не обученная на коде) и её версия, дообученная на огромном корпусе публичных репозиториев GitHub. Результат оказался впечатляющим: дообученная модель решила почти треть тестовых задач (HumanEval) без подсказок, тогда как базовая не решила ни одной [8]. Более того, при увеличении количества параметров качество росло (почти логарифмически), и авторам удалось достичь 70% решённых задач при генерации 100 вариантов решений и выборе корректных. Это показало, что специализированное обучение на коде существенно расширяет возможности LLM: помимо синтаксической грамотности, модель начинает понимать семантику задач и даже некоторые алгоритмические идеи, присутствовавшие в обучающей выборке. Вслед за Codex появились и другие модели, обученные или дообученные на коде. Например, исследователи представили PolyCoder [15] – открытую модель на базе GPT-2 (2.7 миллиарда параметров), обученную на 249ГБ исходного кода на 12 языках программирования. Хотя PolyCoder уступал Codex в целом, на языке Си он превзо-

шёл по точности все доступные на тот момент модели, включая Codex [15]. Компания Salesforce предложила модель CodeT5 [7] – адаптацию текстовой модели T5 для кода, обученную как на заданиях понимания кода, так и на задачах генерации. CodeT5 достигла лучших на тот момент результатов на ряде бенчмарков (например, CodeXGLUE). Позднее, в 2023 году, были выложены более крупные специализированные модели: StarCoder (15B параметров) [24] от сообщества BigCode и Code Llama (7B–34B) от Meta AI [25], предобученные на обширных корпусах исходного кода. Эти модели, обученные «с нуля» на коде, уже можно рассматривать как частично адаптированные к домену программирования. Однако для максимальной эффективности их также дообучают под конкретную задачу: например, чтобы модель принимала на вход именно текстовое описание задачи и выдавала решение, её дополнительно тренируют в режиме инструкций на наборе пар «описание -> код». Такая настройка была проделана, например, для модели StarCoder, в результате чего получился вариант WizardCoder [35], и для модели CodeT5 (вариант InstructCodeT5+ [28]), что значительно повысило их способность следовать описанию задачи и генерировать корректный код. Специализированные датасеты сыграли ключевую роль в успехе supervised fine-tuning для олимпиадного программирования. Авторы статьи [13] создали набор APPS (Automated Programming Progress Standard) из 10,000 задач разного уровня сложности, собранных из открытых источников (включая Codeforces, Kattis, e-olymp и др.). Для объективной оценки решений они предложили использовать систему тестов (как на соревнованиях): программа модели считается успешной, если она проходит все секретные тесты для задачи. В первых экспериментах на APPS даже большие модели, обученные на коде, решали лишь около 5% задач полностью. Однако дообучение модели GPT-2 на смеси GitHub-кода и задач APPS повысило этот показатель до 8%, что стало заметным шагом вперёд [13]. Система AlphaCode [19] формировала для обучения ещё более сфокусированный набор CodeContests, включающий 13,500 задач с платформем Codeforces, AtCoder, TopCoder и т.д. Обученная на нём трансформер-модель (41B параметров) смогла в итоге решить порядка 34% предложенных задач (при ограничении 10 попыток на задачу), что сопоставимо с уровнем сильного любителя [19]. Эти цифры значительно превышали результаты моделей без дообучения, подчёркивая ценность профильного тренировочного материала. Метрики, используемые для оценки качества решений после fine-tuning, обычно ориентированы на функциональную корректность. На стандартных бенчмарках (HumanEval, MBPP) применяются метрики pass@k: например, pass@1 – доля задач, решённых с

первой попытки, pass@5 – доля задач, для которых хотя бы одно из пяти сгенерированных решений проходит все тесты [8]. На крупных наборках типа APPS часто приводят долю полностью решённых задач (аналогично pass@∞, хотя на практике обычно ограничиваются 100 сгенерированными решениями). Также используются вспомогательные метрики вроде BLEU и специализированного CodeBLEU [4], измеряющие схожесть с эталонным решением. Однако они плохо коррелируют с реальной корректностью и применяются скорее для абляций (например, чтобы сравнить варианты модели между собой на простых задачах).

Ограничения подхода supervised fine-tuning связаны прежде всего с качеством и объёмом данных. Собрать большой корпус решений олимпиадных задач с подробными условиями – нетривиальная задача (APPS и CodeContests в сумме содержат около 23k задач, что несопоставимо с миллионами образцов кода на GitHub). Модель, обученная на таком ограниченном корпусе, рискует переобучиться на типичные шаблоны решений и не обобщать на новые, нестандартные задачи. Кроме того, полный fine-tuning очень крупных моделей требует огромных вычислительных ресурсов (GPU памяти, времени обучения) и может занимать много времени. Обновление всех 175 млрд параметров (как у Codex) не всегда осуществимо вне крупных промышленных лабораторий. Наконец, существует риск, что при дообучении на узком домене модель частично потеряет ранее усвоенные общие знания (явление catastrophic forgetting [41]). Чтобы этого избежать, часто применяют миксы данных (доменные + небольшое подмножество общего корпуса) или очень небольшие коэффициенты обучения.

2.3.2 Параметро-эффективные методы дообучения (адаптеры, LoRA)

В 2021 году был предложен подход, позволяющий обойтись без полного обновления всех параметров LLM при адаптации к новой задаче. Речь идёт о методе Low-Rank Adaptation (LoRA) [12]. Суть LoRA: веса исходной модели замораживаются, а в каждое её матричное преобразование добавляются небольшие обучаемые матрицы низкого ранга. Во время fine-tuning обучаются только эти дополнительные параметры (плюс, возможно, несколько слоёв наподобие LayerNorm), что радикально снижает число изменяемых величин. На примере GPT-3 (175B параметров) авторы показали сокращение числа обучаемых параметров в 10,000 раз, при этом качество на NLP-задачах практически не ухудшилось по сравнению с полным fine-tuning [12].

Огромным плюсом LoRA является то, что адаптация не увеличивает время вывода: после обучения дополнительные матрицы можно «внедрить» в исходные веса, не удваивая количество вычислений. Хотя в оригинальной статье [12] LoRA проверялся на текстовых задачах (GLUE, вопросы на понимание), этот метод очень быстро был подхвачен сообществом, работающим с кодовыми моделями. Причина проста: параметры современных LLM для кода (например, Code Llama-34B или GPT-4 с неизвестным числом параметров) настолько многочисленны, что классический fine-tuning без специальных приёмов невозможен вне компаний уровня OpenAI или Meta. LoRA позволил независимым разработчикам и академическим группам обучать адаптации для больших моделей на сравнительно доступном оборудовании. Например, проект Alpaca (Stanford) воспользовался LoRA для настройки LLaMA-7B на синтетических инструкциях, получив недорогой аналог ChatGPT. По аналогии, для кода появились модели вроде WizardCoder [35] и CodeAlpaca [33], где тоже были применены LoRA-адаптации. Помимо LoRA, существуют и другие параметро-эффективные подходы: обучение адаптеров [3] (небольших дополнительных слоёв между трансформерными блоками), метод BitFit [6] (обучение только параметров сдвига в нейронах), комбинирование нескольких тонко настроенных моделей модульным образом и т.п. Они различаются технически, но общий принцип тот же: минимизировать изменения оригинальной модели, добавляя лишь немного новых параметров. В применении к коду LoRA и адаптеры обычно достигают $\sim$95% качества полного fine-tuning при обучении менее 1% параметров модели [12]. Для олимпиадного программирования параметро-эффективные методы особенно ценны, когда нужно адаптировать уже очень мощную модель под специфичные требования. Допустим, имеется большая модель, обладающая базовыми навыками кодирования (например, Code Llama), но её хотят «донаучить» решать именно алгоритмические пазлы. Можно взять такую модель и обучить LoRA-слои на относительно небольшом наборе задач (например, из архива IOI или Codeforces). Практика показывает, что даже несколько тысяч задач для дообучения могут заметно сдвинуть качество: модель начинает лучше справляться с известными паттернами, чаще писать оптимальный по сложности код, реже делать грубые ошибки. При этом базовые знания (синтаксис языков, простые алгоритмы) никуда не пропадают, так как основные веса не меняются, а лишь слегка корректируются дополнительными матрицами. Ограничения данного подхода связаны с тем, что модель всё равно ограничена тем, чему она научилась на предобучении. LoRA не добавляет новых знаний, если их изначально не было; она лишь перенастраивает

уже существующие представления. Поэтому если базовая LLM плохо понимала, скажем, динамическое программирование, то без примеров таких задач при обучении не обойтись – придётся включать их в новый датасет (что по сути сводится к SFT, только менее ресурсоёмкому). Кроме того, иногда возникает вопрос: может, достаточно просто хорошей инженерии инструкции? Практика показывает, что обучение LoRA-слоёв гораздо эффективнее soft-prompt подхода, особенно на сложных генеративных задачах. Например, авторы статьи LoRA [12] в своём исследовании отметили, что prompt tuning провалился на ряде задач, тогда как предложенный метод успешно адаптировал модель. Таким образом, параметро-эффективный tuning выступает «золотой серединой»: он существенно дешевле полного fine-tuning, но позволяет явно внедрить в модель новые знания и навыки.

2.3.3 Обучение с подкреплением (RL, RLHF)

Классическое обучение языковых моделей использует следующего-токена прогнозирование или условную максимизацию правдоподобия, что не всегда оптимизирует то, чего мы хотим от модели. В случае программирования наша цель – чтобы модель писала *правильные и эффективные программы*, а не просто выдавалась статистически правдоподобную последовательность кода. Обучение с подкреплением (Reinforcement Learning, RL) предоставляет возможность оптимизировать модель непосредственно по некоторой целевой метрике, рассматривая процесс генерации кода как последовательность действий, за которые выдаётся награда (reward). Если суметь сформулировать награду, отражающую качество решения задачи, можно дополнительно обучить модель решению задач методом проб и поощрений. Одной из первых работ, применивших RL для генерации кода, стала CodeRL [17]. Авторы отметили, что стандартное SFT игнорирует важную информацию, доступную в тексте задачи, – например, примеры входов-выходов и описанные тестовые сценарии. В CodeRL предложено обучать связку актор–критик: LLM выступает как *актор*, генерирующий код, а дополнительная нейросеть-критик обучается предсказывать функциональную корректность сгенерированных программ и выдавать актору плотный сигнал оценки [17]. Критик тренируется на основе результатов запуска программ (прохождение или непрохождение тестов) и учит актор «пошагово» улучшать решения. В процессе вывода, помимо этого, используется стратегия критического семплинга: модель генерирует несколько вариантов решения и автоматически перере-

нерирует части кода, если критик дал им низкие оценки. Предложенный метод не только достиг новых SOTA-результатов на сложном бенчмарке APPS, но и показал сильный трансфер без дообучения – например, модель CodeRL установила SOTA и на более простом наборе MBPP, хотя обучалась на APPS [17]. Это указывает, что RL-оптимизация по функциональному сигналу (прохождение тестов) приводит к более общим улучшениям способностей модели решать задачи. Сходные идеи стали развиваться и далее. Появились работы, обучающие модель пошагово планировать и решать задачи с помощью RL. Например, авторы статьи [31] предложили метод Self-Planning Code Generation, где модель сначала генерирует явный план решения (набор промежуточных шагов на естественном языке), а затем код по этому плану. Если тесты не пройдены, модель получает отрицательный вознаграждающий сигнал и с помощью RL корректирует стратегию планирования. Такой подход научил модель более эффективно декомпозировать задачи и решать их итеративно, превзойдя качество прямого генеративного метода. Другой пример – работа OpenAI, упомянутая выше, где модель (*o1*) с помощью RL научилась вырабатывать собственные общие стратегии решения задач IOI [43]. По отчёту, RL существенно повышает pass@1 на сложных задачах (например, прирост почти 10 процентных пунктов по сравнению с версией модели без RL) и позволяет достичь результатов уровня медалистов IOI без жёстко запрограммированных эвристик [43]. Особо стоит упомянуть RLHF (Reinforcement Learning from Human Feedback) в контексте генерации кода. Изначально RLHF применялся для того, чтобы наделить модели навыком следования человеческим инструкциям и запретам (как в InstructGPT [22]). В случае кода «человеческий фидбек» может включать, например, предпочтения программистов относительно стиля, читаемости или детальности комментариев, а также общее чувство правильности решения. На практике крупные модели типа ChatGPT действительно обучались с участием экспертов-программистов, оценивающих ответы: это привело к тому, что такие модели предпочитают генерировать решения с пояснениями, избегать небезопасного кода, предлагать оптимальные варианты там, где возможно. Тем не менее, RLHF для непосредственно повышения *корректности* кода пока не получил широкого распространения – человеку тяжело в интерактивном режиме указывать модели на логические ошибки в нетривиальном коде. Гораздо естественнее автоматизировать проверку с помощью самих же компьютеров. Именно это и реализуется в методах типа CodeRL, описанных выше, где роль «критика» выполняет программа запуска тестов или дополнительная модель. Недавно некоторые работы предложи-

ли гибридные подходы: объединить автоматическую проверку с обобщённым человеческим знанием. Главное ограничение RL-подходов – их сложность и стоимость. Обучение модели с подкреплением, особенно если модель очень большая, – процесс дорогостоящий и чувствительный к настройкам. Нужно аккуратно прорабатывать сигнал награды, чтобы избежать «нечестного» обучения (например, ситуации, когда модель научится проходить тесты нечестным путём). Известен курьёзный пример: модель, оптимизируя метрику прохождения тестов, начала выводить пустой код для задач, где пустой код формально проходил тесты (из-за недоработки в тестовом наборе). Чтобы такие случаи исключить, необходимо усложнять тесты и вводить негативные награды за тривиальные или неполные решения. Другой аспект – RL может привести к дрейфу модели от исходного поведения, иногда в ущерб другим задачам. Поэтому на практике нередко сочетают RL с фазой SFT (как в InstructGPT [22]) или применяют относительно «осторожные» алгоритмы вроде PPO, которые ограничивают степень отклонения модели. Несмотря на эти сложности, применение RL уже принесло ощутимые улучшения. Модель CodeRL улучшила state-of-the-art на конкурсных задачах, а масштабные LLM с RLHF демонстрируют беспрецедентные результаты, решая ранее недоступные машине проблемы. Ожидается, что дальнейшее развитие этого направления (например, обучение модели самостоятельно исполнять и проверять свой код в симулированной среде) ещё больше повысит её возможности.

2.3.4 Генерация и использование синтетических данных

Как отмечалось ранее, одним из узких мест адаптации LLM к олимпиадному программированию является ограниченное количество доступных обучающих данных. Аннотирование новых задач и решений вручную – крайне трудоёмко, а использовать все имеющиеся открытые задачи для обучения – значит рисковать переобучением модели на них. Поэтому появился подход: привлечь к созданию обучающих данных саму модель. Если мощная LLM способна сгенерировать новые задачи программирования вместе с решениями, эти синтетические примеры можно добавить в тренировочный набор и дообучить модель, расширив тем самым её знания и навыки в нужном домене. В 2022–2023 годах этот подход получил развитие как метод self-instruct [21]: сначала большая модель (например, GPT-3 или GPT-4) генерирует разнообразные задания и соответствующие ответы на основе нескольких вручную заданных примеров, затем полученная коллекция используется для обуче-

ния целевой модели поменьше. В области кода подобную идею реализовали авторы WizardCoder [35]. Они взяли открытую модель StarCoder-15B (обученную на коде) и провели её инструкционное дообучение на *синтетически созданном* наборе сложных задач. Для генерации данных применялся метод Evol-Instruct: сначала подготовили небольшое ядро простых задач с решениями, затем использовали модель GPT-4, чтобы итеративно усложнять эти задачи (добавлять новые требования, увеличивать размер входных данных, накладывать дополнительные ограничения) и генерировать решения для новых, более сложных вариаций. Итоговый синтетический датасет содержал десятки тысяч задач значительно более высокого уровня сложности, чем исходные, – фактически приближенных к уровню конкурсов. Обучив WizardCoder на этом материале, авторы добились впечатляющих результатов: модель превзошла все предыдущие открытые модели на четырёх популярных бенчмарках (HumanEval, HumanEval+, MBPP, DS-1000), а на HumanEval обошла даже некоторые закрытые коммерческие LLM (например, Anthropic Claude и Google Bard) [35]. Это демонстрирует силу синтетического обучающего набора, сгенерированного более мощной моделью: фактически, младшая модель обучается решать задачи, которым её «научил» старший собрат. Другой пример – CodeAlpaca [33]. Авторы этой инициативы сгенерировали около 20 тысяч пар «описание задачи – решение» с помощью API модели text-davinci-003 (родственной GPT-3.5), ориентируясь на формат задач из учебников и сайтов вроде LeetCode. Затем они дообучили базовую LLaMA-7B на этих парах. Получившаяся модель CodeAlpaca заметно улучшила способность LLaMA следовать инструкциям по кодированию, приблизившись по качеству к закрытому Codex (хотя и не достигнув уровня GPT-3.5). Данный проект продемонстрировал, что даже без доступа к реальным данным соревнований можно повысить навык модели решать кодовые задачи, если привлечь «эксперта» (в данном случае, мощный проприетарный LLM) для генерации учебных примеров. Интересное развитие идеи самообучения представлено в работе InverseCoder [39]. Авторы попытались заставить модель улучшаться итеративно за счёт собственных решений. Сначала базовая модель генерирует решения для множества задач, затем эти решения автоматически классифицируются (правильные/неправильные) с помощью существующих проверочных инструментов. После этого формируется новый обучающий набор, состоящий из тройки: условие задачи, неправильное решение модели, исправленное решение (сгенерированное либо человеком, либо самой моделью при повторном запуске). Обучаясь на таком корпусе (посредством SFT или LoRA), модель приобретает навык исправления собственных

ошибок: при дальнейшем использовании она уже способна распознавать типичные промахи и избегать их. Этот метод, по сути, моделирует подход, когда модель учится на своём опыте («ошибках»), подобно тому как человек, решая задачи, запоминает, чего делать не следует. Главное преимущество генерации синтетики – возможность практически неограниченно масштабировать тренировочный набор. Модель можно заставить генерировать всё новые и новые задачи, варьируя условия, увеличивая сложность, комбинируя разные темы. Например, WizardCoder воспользовался мощью GPT-4 для эволюции задач, получив значительно более комплексный датасет, чем исходные реальные задачи. Однако качество синтетических данных становится критичным фактором. Если в автоматически сгенерированных примерах будут ошибки или они окажутся слишком однообразными, то модель обучится на них неправильно. При подготовке WizardCoder это решалось ручным контролем: авторы убедились, что GPT-4 генерирует в основном корректные решения, и отфильтровывали заведомо неверный синтез. Но такая роскошь, как GPT-4, есть не у всех, а слабая модель может генерировать и некорректный код, и тривиальные задачи. Поэтому есть риск, что синтетический набор не будет покрывать реальное разнообразие олимпиадных задач. Например, модель может сгенерировать много задач на вычисление факториалов и сортировки (которые она знает), но не сгенерирует задачи на сложные структуры данных, о которых не имеет понятия. Тем не менее, описанные проекты показали, что синтетические данные – мощный инструмент. Они снижают зависимость от дорогой ручной разметки и позволяют быстро адаптировать модель под новые форматы задач. Можно ожидать, что в будущем появятся схемы кооперативного обучения моделей, когда одна модель генерирует задачи, другая их решает, и обе улучшаются через обмен информацией. Подобный самоиграющий подход уже принес успехи в других областях (Go, программирование игр), а в программировании он только начинает применяться. В перспективе, сочетая генерацию данных, RL и агентные методы, возможно создание системы, способной самостоятельно осваивать всё новые и новые классы задач, постепенно приближаясь к универсальному программисту-ассистенту.

2.4 Выводы

Адаптация больших языковых моделей к домену олимпиадного программирования – динамично развивающееся направление, стоящее на стыке NLP и автоматиза-

ции программирования. Мы рассмотрели широкий спектр методов: от сравнительно простых (инженерия подсказок, дополнение внешней информацией) до комплексных (многоступенчатые агенты с тестированием, обучение с подкреплением, генерация данных). Эволюция подходов показывает, как по мере роста возможностей LLM изменялись стратегии их использования: сначала пытались максимально использовать встроенные знания модели через удачную формулировку запросов, затем – досуществлять обучение, снабжая модель новыми данными и сигналами, а теперь – всё чаще наделяют модель механизмами активного поиска решений, интегрируют её в более сложные системы. Для задач генерации кода ключевым фактором успеха стало появление обширных датасетов программ (как в pre-training, так и для fine-tuning). Это позволило моделям вроде Codex и AlphaCode впервые приблизиться к уровню человеческих решений. Методы без обучения весов, в частности продвинутые схемы prompt engineering и агентные системы, позволили выжать максимум из уже обученных моделей, продемонстрировав, что иногда правильная постановка задачи для модели не менее важна, чем её размеры. С другой стороны, методы с дообучением явно наделили модели новыми навыками: SFT обучило их шаблонам решений, RL направило на достижение конкретной метрики успеха (прохождения тестов), а синтетические данные существенно расширили кругозор моделей без привлечения ручной разметки. Несмотря на впечатляющие успехи – модели уже в состоянии решать значительную часть стандартных соревновательных задач – остаются заметные вызовы. Во-первых, многие модели всё ещё уязвимы для ошибок на действительно нетривиальных задачах, требующих изобретательности или глубокого понимания. Они могут генерировать синтаксически верный, но неэффективный код, или упускать редкие краевые случаи. Методы самопроверки и RL отчасти решают эти проблемы, но не полностью. Во-вторых, эффективность вычислений остаётся проблемой: самые сильные модели требуют больших ресурсов, а многоходовые подходы замедляют получение ответа. Это затрудняет применение таких систем на практике (например, в ограниченных средах выполнения или интерактивных задачах с жёсткими таймаутами). Идут работы над оптимизацией – от сжатия моделей до разработки гибридных систем, где тяжёлая модель вызывается только при необходимости, в сочетании с более простыми эвристиками. Наконец, возникает вопрос доверия и оценки таких моделей. В соревнованиях по программированию важна не только корректность, но и объяснимость решений. Современные LLM зачастую предоставляют только код, без обоснования, или генерируют длинные комментарии, не всегда соответствующие

коду. Будущие исследования вероятно сфокусируются на том, чтобы модели могли аргументировать свой выбор алгоритма, доказывать корректность (возможно, с помощью встроенных проверок или даже средств формальной верификации).

3 Основной раздел работы

В данном разделе приведены предложенные методы для генерации и сбора данных, требуемых для обучения моделей и обогащения их релевантным контекстом, а также результаты вычислительных экспериментов обучения и измерения метрик качества на наборе данных олимпиадного программирования CodeContests.

3.1 Формальная постановка задачи

Основной задачей данной работы является: адаптация больших языковых моделей (LLM) для эффективного решения задач олимпиадного программирования.

Требуется адаптировать предобученную большую языковую модель $\mathcal{M}$, параметризованную весами θ, к целевой предметной области $\mathcal{T}$, представляющей собой задачи олимпиадного программирования. Для этого предлагается набор методов доменной адаптации $\mathcal{A}$, а также специализированный набор данных $\mathcal{D}$, состоящий из задач P_i с соответствующими корректными решениями S_i.

Целевая функция обучения:

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

где $\text{solve}(X, S_i)$ — функция оценки корректности решения X, полученного моделью, относительно эталонного решения S_i задачи P_i.

Для оценки функциональной корректности генерируемого кода используется метрика pass@k, предложенная для более адекватной оценки моделей генерации программ по сравнению с традиционными текстовыми метриками. Метрика pass@k характеризует вероятность того, что хотя бы один из k лучших сгенерированных решений задачи проходит все тесты, что соответствует реальным практикам программистов, оценивающих корректность кода по прохождению тестовых случаев.

Формально, для задачи, для которой модель сгенерировала n решений, из которых c решений являются корректными, метрика pass@k определяется следующим образом:

$$\text{pass@}k := \mathbb{E}_{\text{problems}}\left[1 - \frac{C(n-c,\, k)}{C(n,\, k)}\right],$$

где $C(n,\, k)$ — число сочетаний из n по k.

Данная формула вычисляет вероятность того, что среди случайно выбранных k решений хотя бы одно будет корректным. В частности, $C(n-c,k)/C(n,k)$ — это

вероятность того, что все k выбранных решений окажутся некорректными, а $1-$ данная величина — вероятность хотя бы одного корректного решения среди k попыток. Математическое ожидание берётся по всем задачам. Такой подход к метрике оценки позволяет более реалистично отражать способность модели к функционально верной генерации кода и используется как основной критерий качества моделей в данной работе.

3.2 Доменная таксономия и исходный корпус задач

В качестве целевого предметного пространства выбраны *задачи олимпиадного программирования*. Они обладают большим методическим разнообразием и определенным критерием оценки (код либо проходит все тесты, либо нет). В качестве источника данных для олимпиадных задач был выбран набор данных CodeContests [19]. На рисунке 1 изображен пример задачи из рассматриваемого набора данных. Тэги были сгруппированы в домены на основе их тематической близости и схожести применяемых алгоритмических методов. Такая агрегация отражает ключевые направления олимпиадных задач и облегчает дальнейший анализ результатов по отдельным областям.

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.

Рис. 1: Пример задачи олимпиадного программирования из набора данных CodeContests. В описании задачи присутствуют описание, примеры входных и выходных данных, список публичных и закрытых тестов, которые должны быть выполнены для успешного выполнения задачи.

Каждая задача в датасете сопровождается набором тэгов, отражающих её при-

надлежность к определённой алгоритмической категории. Эти тэги послужили основой для формирования подхода к доменной адаптации, заключающегося в обучении специализированных языковых моделей, обладающих экспертностью в конкретных областях алгоритмов. Под доменом в рамках данной работы понимается узкая предметная область, соответствующая одному из разделов теории алгоритмов, таких как динамическое программирование, графовые алгоритмы, жадные стратегии и др. Такой подход позволяет учитывать структурную специфику задач и адаптировать поведение LLM к различным алгоритмическим парадигмам. Стоит отметить факт, что задача может несколько тэгов - это может говорить о том, что либо эту задачу можно решить несколькими способами, либо что необходима комбинация нескольких подходов, но также нужно учитывать погрешность оценки при оценке принадлежности задаче к конкретно области эксператами. Предлагаемая таксономия выделенных доменов представлена в таблице 1.

Домен	Тэги
`adhoc`	implementation, constructive algorithms, interactive, schedules, expression parsing, *special
`greedy`	greedy, sortings, two pointers
`search`	binary search, ternary search, meet-in-the-middle, divide and conquer, brute force
`dp`	dp
`structures`	data structures, trees, bitmasks, dsu
`graphs`	graphs, dfs and similar, shortest paths, flows, graph matchings, 2-sat
`strings`	strings, string suffix structures, hashing
`geometry`	geometry
`math`	math, number theory, combinatorics, probabilities, matrices, chinese remainder theorem, fft, games

Таблица 1: Классификация алгоритмических доменов и соответствующих тэгов

3.3 Исследование методов доменной адаптации

В данном разделе будут рассматриваться исследованные методы доменной адаптации, основанные на таксономии в пункте 3.2. В качестве базового метода относительно которого будет происходить сравнение является стандартный режим генерации LLM с единой инструкцией, в которой есть информация только о направлении решения - задаче олимпиадного программирования, без указания конкретной доменной специализации.

3.3.1 Prompt Engineering

Метод Prompt Engineering трактуется как *«нулевая»* (безградиентная) адаптация модели за счёт подбора специальной инструкции, которая подчёркивает алгоритмические особенности целевого домена. Гипотеза метода формулируется так: **корректно спроектированная инструкция способена индуцировать у LLM локальную экспертизу без каких-либо изменений её весов**.

Формальная схема. Для каждого домена $d \in \mathcal{D}$ создаётся шаблон инструкции τ_d (см. Приложение А), отражающий эвристику решения задач данного класса. Пусть P_i — формулировка задачи, d_i — её домен. Тогда итоговая инструкция строится как

$$\text{prompt}(P_i) = \underbrace{\texttt{<SYS>} + \tau_{d_i}}_{\text{доменная инструкция}} \;\|\; P_i.$$

Ответ модели вычисляется по правилу $X_i = \mathcal{M}(\text{prompt}(P_i), \theta)$.

Оптимизационная задача. Под *оптимизацией* в Prompt Engineering понимается отбор такого семейства $\{\tau_d\}$, что

$$\max_{\{\tau_d\}} \mathbb{E}_{(P_i,S_i)}\big[\text{solve}\big(\mathcal{M}(\tau_{d_i} \| P_i, \theta), S_i\big)\big]$$

достигает максимума. На практике поиск ведётся вручную (итеративный A/B-тест), иногда дополняется небольшим автоматическим grid-search по вариантам формулировок и порядку секций внутри инструкции.

Особенности метода

- Безградиентная адаптация. Не требует дообучения модели и потребления GPU-ресурса.

- Мгновенное переключение доменов. Экспертиза диктуется только выбранной инструкцией τ_d.
- Ограничения длиной контекста. Инструкция конкурирует за токены с текстом задачи; при длинных задачах полезная часть инструкции может усекаться.
- Чувствительность к формулировке. Малейшая стилистическая правка способна существенно изменить качество решения, что затрудняет воспроизводимость.

3.3.2 Supervised Fine-Tuning (SFT)

В отличие от Prompt Engineering, метод SFT осуществляет *эксплицитную* адаптацию весов модели на доменно-специфической разметке. Базовая идея: **обучить модель генерировать пошаговое объяснение и код, наблюдая примеры «задача → решение»**.

Построение датасета.

1. Генерация описаний решений. Для каждой пары (P_i, S_i) из набора данных CodeContests крупная «teacher»-модель (32–72B) создаёт текст объяснения E_i по инструкции вида *«Проанализируй задачу и данное корректное решение, опиши алгоритм и докажи правильность»*.
2. Формирование обучающих примеров. Итоговый source-контекст: $\big[$<SYS> + инструкция$\big] \| P_i$. Target — последовательность $E_i \| S_i$.
3. Разделение по доменам. Каждому домену d соответствует собственный поднабор $\mathcal{D}_d^{\mathrm{SFT}}$, что позволяет тренировать либо (а) отдельные LoRA-адаптеры, либо (б) единую «мультидоменную» модель.

Оптимизационная задача. Используется минимизация потокенной кросс-энтропии:

$$\min_{\varphi} \; \mathbb{E}_{(x,y)\sim\mathcal{D}^{\mathrm{SFT}}}\Big[-\sum_{t=1}^{|y|}\log p_{\theta\oplus\varphi}(y_t \mid y_{<t}, x)\Big],$$

где φ — набор LoRA-матриц ранга $r \ll d_{\mathrm{model}}$, а θ — замороженные базовые веса.

Онлайн-генерация. При решении новой задачи P модель загружается с выбранным адаптером $\varphi_{d(P)}$ (или без него, если используется унифицирующая версия) и выдаёт ответ $\mathcal{M}(P, \theta \oplus \varphi_{d(P)})$.

Особенности метода

- Параметрическая адаптация. В отличие от RAG, знания «впитываются» в веса и больше не требуют retrieval-фазы.
- LoRA-эффективность. Размер одного доменного адаптера φ — $\mathcal{O}(r \cdot d_{\text{model}})$, что на практике даёт экономию VRAM $\times$10–20 по сравнению с хранением полной копии модели.
- Требования к качеству синтетики. Ошибки «teacher»-модели напрямую переносятся на обучаемую LLM, вызывая шум и переобучение.
- Возможен *catastrophic forgetting.* При узком домене модель теряет универсальные навыки; смешанное (multi-domain) обучение частично решает проблему ценой большего TPU/GPU-времени.

3.3.3 Retrieval-Augmented Generation (RAG)

В отличие от SFT, где адаптация осуществляется *ex-situ* путём модификации весов модели, метод Retrieval-Augmented Generation адаптирует LLM *on-the-fly (в режиме реального времени)*, расширяя ее контекст документами, отобранными из специализированного корпуса знаний. Ключевое предположение метода состоит в том, что **релевантный доменный контекст задаёт для модели *экспертизу***: если задача относится, например, к динамическому программированию, то в итоговую инструкцию должны быть включены исключительно материалы, касающиеся динамического программирования, — тем самым LLM «становится экспертом» именно в этом узком разделе.

Формальная схема. Пусть $\mathcal{E} = \bigcup_{d \in \mathcal{D}} \mathcal{E}_d$ — корпус внешних документов, где каждая подколлекция $\mathcal{E}_d$ содержит тексты, помеченные доменом $d \in \mathcal{D}$ (согласно таксономии из раздела 3.2). Для каждой задачи P_i известен её домен d_i (либо набор доменов, если задача многотеговая).

1. Индексация. Каждый документ $c \in \mathcal{E}$ разбивается на фрагменты фиксированной длины (800 символов, перекрытие 200), после чего по каждому фрагменту $\tilde{c}$ формируется векторное представление $e(\tilde{c}) \in \mathbb{R}^m$ через модель Sentence-BERT. Векторные представления индексируются FAISS-графом HNSW (Hierarchical Navigable Small World) [2].

2. Доменный фильтр. При поиске рассматриваются только фрагменты из $\mathcal{E}_{d_i}$, гарантируя, что retrieved-контекст тематически соответствует задаче.

3. Поиск релевантного контекста. Запрос $q_i = e(P_i)$ формируется как векторное представление всего текста задачи (без примеров ввода/вывода). Функция

$$\mathcal{R}(q_i, k) = \underset{\tilde{c} \in \mathcal{E}_{d_i}}{\text{top-}k}\big(\langle q_i,\, e(\tilde{c})\rangle\big)$$

возвращает k наиболее близких фрагментов по косинусному сходству.

4. Формирование инструкции. Итоговый контекст $\mathcal{C}_i = \{\tilde{c}_1, \ldots, \tilde{c}_k\}$ добавляется к тексту задачи:

$$\text{prompt}(P_i) = \underbrace{\texttt{<SYS>} + \text{инструкция}}_{\text{общая}} \;\|\; \underbrace{\mathcal{C}_i}_{\text{домен}} \;\|\; \underbrace{P_i}_{\text{задача}},$$

где символ $\|$ обозначает конкатенацию сегментов.

5. Генерация решения. LLM выдаёт ответ $X_i = \mathcal{M}\big(\text{prompt}(P_i), \theta\big)$.

Оптимизационная задача. С учётом retrieval-шага функция, максимизируемая в разделе 3.1, переписывается как

$$\max_{\theta, \mathcal{R}} \; \mathbb{E}_{(P_i, S_i)}\big[\text{solve}\big(\mathcal{M}(P_i \| \mathcal{R}(P_i, k), \theta),\; S_i\big)\big],$$

где оптимизация по $\mathcal{R}$ сводится к выбору качественных векторных представлений, стратегии фильтрации и значения k.

Особенности метода

- «Мягкая» адаптация. Веса LLM остаются неизменными; адаптация достигается путём подстановки тематического контента прямо в инструкцию.
- Динамическая специализация. Для каждой задачи формируется *своя* экспертиза — модель может переключаться между доменами без перезагрузки весов или адаптеров.
- Минимальные требования к GPU-памяти. В оперативной памяти хранится только база эмбеддингов ($\approx$400k токенов $\Rightarrow$ несколько сотен МБ) и сама LLM, что делает метод особенно привлекательным при работе с крупными моделями (>30B параметров) на ограниченных ресурсах.

Ограничения. Метод RAG чувствителен к: (i) полноте корпуса $\mathcal{E}$, (ii) качеству семантических эмбеддингов, (iii) разбиению на фрагменты (сохранение целостности кода), и (iv) корректности доменной разметки. Влияние этих факторов подробно рассмотрено в разделе 4, посвящённом экспериментальной оценке.

4 Вычислительные эксперименты

4.1 Supervised Fine-Tuning

Для SFT-обучения требовался корпус вида *«задача → (пошаговое объяснение + корректное решение)»*. Данные собирались в два шага.

1. Предподготовка данных. Из `train`-части CodeContests выбирались только те задания, тэги которых однозначно указывали на соответствующий из целевых доменом согласно предложенной таксономии (таблица 1). Стоит отметить, что одна задача может пренадлежать к одному или нескольким доменам - тогда задача попадет сразу в несколько обучающих выборок.

2. Синтетическая разметка. Teacher-модель Gemma 3 27B instruct (запуск через vLLM на двух A100, 80 GB) получала на вход *текст задачи + эталонное C++-решение* и генерировала объяснение по схеме согласно описанию в пункте 3.2. Для каждой задачи бралось три эталоннах решения, которые прошли все тесты, и, соответственно, из одной задачи в исходном наборе данных получалось по три примера. Сгенерированный вывод принимался, *только если:*

   - код корректно распознавался компилятором `g++ -std=c++20`;
   - все публичные тесты авторов проходили без ошибок.

Статистика по собранному корпусу обучающих примеров приведена на рис. 2. Стоит отметить, что домены strings и geometry имеют сильно меньшее количество обучающих примеров в сравнении с другими доменами, что может сказаться на итоговом качестве решения задач методом SFT. Пример задачи из обучающей выборки приведён в Приложении C.

В качестве базовой LLM использовалась gemma-3-12B-instruct [44] [38]. Адаптация проводилась LoRA-методом (ранг $r = 128$, $\alpha = 256$, dropout 0.05) при замороженных базовых весах. Параметры адаптера выбирались на основании статьи [41], в которой авторы показали преимущество использования высокого ранга для адаптеров для задач генерации кода. В таблице 2 перечисляются ключевые гиперпараметры.

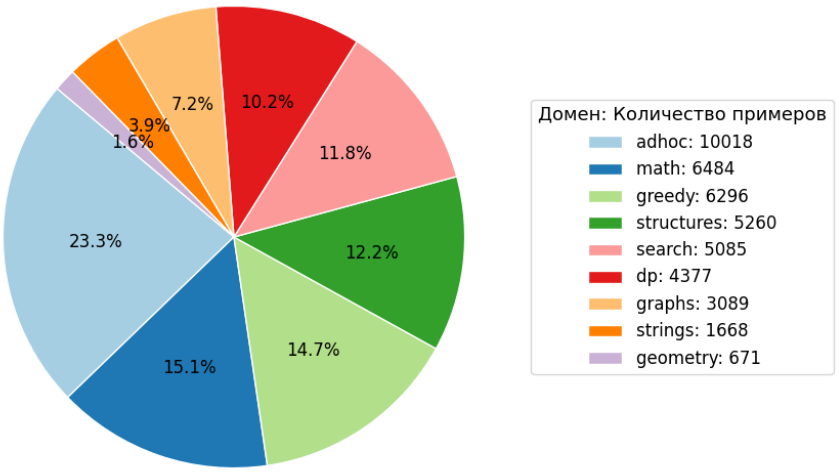

Рис. 2: Доменное распределение примеров в синтетическом датасете для обучения модели методом SFT. Крупнейшие доли занимают домены `adhoc`, `math` и `greedy`, совокупно покрывая более половины обучающей выборки. Домены `strings` и `geometry` представлены слабо, что может повлиять на качество генерации решений для задач этих классов.

Параметр	Значение	Комментарий
Оптимизатор	`adamw_torch_fused`	
Скорость обучения η	$2 \cdot 10^{-4}$	косинусное затухание
Warm-up	10 % шагов	
Batch size	1 пер-GPU	A100 $\times 2$, grad. accum. $= 16$
Эпохи	4	
Макс. последоват-ть	4 092 токена	с packing
Gradient checkpointing	вкл.	экономия VRAM
Max grad norm	1.0	
Weight decay	0.01	

Таблица 2: Гиперпараметры SFT-обучения.

Визуализация на Рис. 3 отображает ключевые метрики в процессе обучения модели методом SFT для домена greedy:

- **train/num_tokens**: Общее число обработанных токенов линейно растёт с числом шагов обучения, что соответствует стабильному прогрессу обучения без пропусков данных.
- **train/mean_token_accuracy**: Средняя потокенная сложность неуклонно растёт с 75% до более 95%, демонстрируя качественное обучение модели на

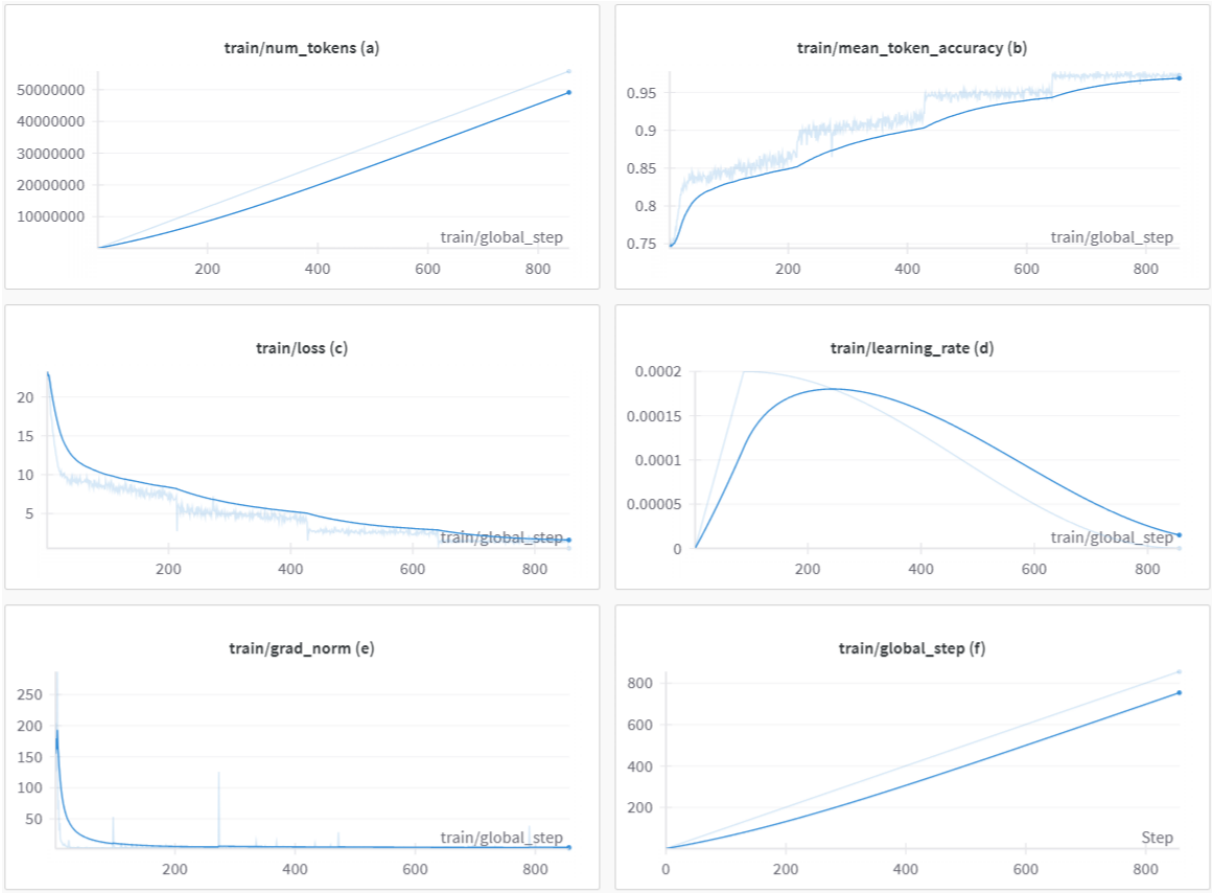

Рис. 3: Динамика ключевых метрик в процессе обучения модели методом SFT: **(a)** график количества токенов в процессе обучения — наблюдается линейный рост, что подтверждает стабильный прогресс обучения; **(b)** график средней потокенной точности — точность уверенно растёт от 75% до 95%, что свидетельствует об успешном обучении модели; **(c)** график функции потерь — loss быстро снижается на ранних этапах и достигает устойчивого минимума; **(d)** график изменения learning rate (скорости обучения) — реализована косинусная схема с фазой прогрева, после которой скорость плавно снижается; **(e)** график нормы градиента — резкий спад и последующая стабилизация указывают на отсутствие взрывных градиентов; **(f)** график глобального шага обучения — линейный рост шагов подтверждает непрерывное и корректное выполнение итераций.

уровне генерации каждого отдельного токена. Наличие шумов на кривой свидетельствует о чувствительности к батчам, особенно на ранних этапах.

- **train/loss**: Классическая убывающая кривая потерь указывает на успешное сходимое обучение. Видно, как loss быстро падает на первых сотнях шагов, после чего уменьшается медленнее, приближаясь к плато.

- **train/learning_rate**: Использовалась косинусная схема затухания с 10% warm-up шагов. На графике чётко видно рост до пика, после чего происходит плавное снижение learning rate в течение всей оставшейся тренировки.

- **train/grad_norm**: Градиентная норма резко падает на старте (свыше 250), стабилизируясь на низких значениях — это признак нормальной и безопасной динамики обучения, без взрывов градиентов.

- **train/global_step**: Линейный рост подтверждает стабильное прохождение шагов обучения без прерываний, а также подтверждает корректность логирования и отслеживания прогресса.

4.2 RAG

Основополагающим фактором для корректной работы данного подхода является качество источника внешних знаний. В качестве базы теоретической информации по алгоритмам был выбран репозиторий `cp-algorithms` [36]. Он содержит достаточный объём материала по теории алгоритмов, охватывающий все домены, представленные в ранее предложенной таксономии, и также для каждого документа была произведена разметка принадлежности к определенному домену, это происходило в автоматическом режиме используя проприетарную модель GPT-4.5 [45], решение о принадлежности происходило на основе метериалов документа и изначальной разбивке на главы (названиях категорий в которых лежат документы). В результате было собрано 152 документа, что составляет корпус объёмом около 400 тысяч токенов. Каждый документ был разбит на фрагменты длиной 800 символов с перекрытием в 200 символов. Такой размер был выбран исходя из необходимости сохранять во фрагментах цельные логические части теоретического текста и кода. Для предотвращения усечения блоков кода при разбиении применялась постобработка: при необходимости к фрагменту добавлялись оставшиеся строки кода, даже если итоговая длина превышала установленный предел.

Пример результатов поиска релевантного контекста с использованием алгоритма RAG приведён в приложении B. Первым ограничивающим фактором является сам объём корпуса: мы не можем гарантировать, что он покрывает всю необходимую теоретическую базу для решения всех типов задач. Вторым источником потенциальной ошибки выступает процесс подготовки данных: наличие технических артефактов (разметка, пунктуация, форматирование) может вносить шум, а разбиение на фрагменты может нарушать логическую целостность материала. Это может привести к тому, что важная мысль или часть объяснения окажется разделённой между несколькими фрагментами и не попадёт в контекст в полной форме.

Дополнительным фактором неопределённости является автоматическая разметка доменов с использованием LLM. Несмотря на то что структура репозитория изначально подразумевает деление на общие категории, а после сбора данных была проведена ручная проверка, нельзя полностью исключить небольшие отклонения. Все перечисленные факторы могут привести к включению нерелевантных документов в retrieved-контекст, что повышает вероятность возникновения галлюцинаций у модели и может привести к генерации некорректных решений.

4.3 Результаты экспериментов

В данном разделе проводится детальный сравнительный анализ полученных результатов экспериментов по ключевым метрикам качества генерации решений задач олимпиадного программирования (см. таблицы 3, 4, 5, 6). Для комплексной оценки эффективности каждого метода рассмотрены следующие аспекты: качество решения задачи (*pass@1*, *pass@10*), общее количество и уровни сложности задач, решенных и не решенных относительно базовой модели.

Метрика качества генерации решений оценивается с использованием стандартного показателя pass@k, который отражает долю задач, для которых хотя бы одно из первых k сгенерированных решений прошло все тесты. Для каждой задачи при фиксированной температуре генерации ($T = 0.5$) последовательно генерируется 10 решений с использованием различных значений зерна генератора случайных чисел от 0 до 9. Значение метрики pass@1 соответствует доле задач, в которых первое решение (с зерном = 0) успешно прошло все тесты. Аналогично, pass@k считается как доля задач, где хотя бы одно из первых k решений (по seed'ам от 0 до $k-1$) оказалось успешным. Такой подход позволяет воспроизводимо сравнивать методы адаптации модели, снижая влияние стохастичности генерации.

В рамках сравнительного анализа эффективности различных методов доменной адаптации были использованы следующие метрики, позволяющие комплексно охарактеризовать качество работы моделей на выбранных предметных областях:

- pass_count@k — абсолютное количество задач, для которых хотя бы одно из первых k сгенерированных решений прошло все тесты;

- delta(pass_count@k, +) — количество задач, которые модель научилась решать по сравнению с базовой моделью, среди первых k попыток;

- delta(pass_count@k, -) — количество задач, которые модель перестала решать по сравнению с базовой моделью, среди k попыток.

Метод	pass_count@k																	
	adhoc (64)		dp (40)		geometry (6)		graphs (31)		greedy (72)		math (84)		search (54)		strings (15)		structures (60)	
	k=1	k=10	k=1	k=10	k=1	k=10	k=1	k=10	k=1	k=10	k=1	k=10	k=1	k=10	k=1	k=10	k=1	k=10
Baseline	25	**44**	10	31	**1**	**4**	7	16	15	47	26	61	17	**39**	2	**10**	14	43
PE	21	39	16	32	**1**	2	11	15	17	38	35	62	13	33	4	9	14	39
RAG	**30**	38	**23**	**35**	**1**	2	**14**	**17**	**31**	43	**38**	54	**24**	31	**8**	9	**28**	43
SFT	19	43	16	31	**1**	3	7	16	19	**49**	27	**64**	1	7	1	3	14	**48**

Таблица 3: Суммарное количество решенных задач по всем доменам и подходам (метрики *pass_ count@1*, *pass_ count@10*). В скобках, рядом с названием домена, написано общее число задач для оценки качества. Жирным шрифтом выделены наилучшие результаты по каждой группе задач. Наибольшее число решённых задач наблюдается при использовании подхода RAG для *pass_ count@1*, что демонстрирует эффективность применения внешнего контекста для быстрой генерации решений. Для *pass_ count@10* наилучшие показатели распределены между методами, что указывает на важность разнообразия стратегий и подходов для устойчивого достижения успеха при множественных попытках генерации решений.

Метод	pass@k																	
	adhoc (64)		dp (40)		geometry (6)		graphs (31)		greedy (72)		math (84)		search (54)		strings (15)		structures (60)	
	k=1	k=10	k=1	k=10	k=1	k=10	k=1	k=10	k=1	k=10	k=1	k=10	k=1	k=10	k=1	k=10	k=1	k=10
Baseline	0.39	**0.69**	0.25	0.78	**0.17**	**0.67**	0.23	0.52	0.21	0.65	0.31	0.73	0.31	**0.72**	0.13	**0.67**	0.23	0.72
PE	0.33	0.61	0.40	0.80	**0.17**	0.33	0.35	0.48	0.24	0.53	0.42	0.74	0.24	0.61	0.27	0.60	0.23	0.65
RAG	**0.47**	0.59	**0.57**	**0.88**	**0.17**	0.33	**0.45**	**0.55**	**0.43**	0.60	**0.45**	0.64	**0.44**	0.57	**0.53**	0.60	**0.47**	0.72
SFT	0.30	0.67	0.40	0.78	**0.17**	0.50	0.23	0.52	0.26	**0.68**	0.32	**0.76**	0.02	0.13	0.07	0.20	0.23	**0.80**

Таблица 4: Вероятность успешного решения задач по всем предметным областям и подходам (метрики *pass@1*, *pass@10*). В скобках, рядом с названием домена, написано общее число задач для оценки качества. Жирным шрифтом выделены наилучшие результаты по каждой группе задач. Из таблицы следует, что метод RAG обеспечивает наибольшее значение метрики *pass@1* в большинстве предметных областей, подтверждая высокую эффективность внешнего контекста в улучшении качества первой генерации решения. Для метрики *pass@10* лучшие результаты распределены по всем подходам, что указывает на возможность как улучшения, так и деградации вариативности поиска при специализированном обучении.

Метод	delta(pass_count@k, +)																	
	adhoc (64)		dp (40)		geometry (6)		graphs (31)		greedy (72)		math (84)		search (54)		strings (15)		structures (60)	
	k=1	k=10	k=1	k=10	k=1	k=10	k=1	k=10	k=1	k=10	k=1	k=10	k=1	k=10	k=1	k=10	k=1	k=10
PE	5	2	9	3	0	**0**	6	2	10	5	20	4	6	**4**	2	1	8	4
RAG	**11**	3	**14**	**7**	**1**	**0**	**10**	**5**	**20**	**6**	**21**	**7**	**13**	3	**6**	**2**	**18**	**8**
SFT	2	**4**	9	4	**1**	**0**	2	2	11	3	10	6	1	1	0	0	7	5

Таблица 5: Количество задач, которые модель дополнительно научилась решать по сравнению с базовой моделью (*delta(pass_ count@k, +)*). В скобках, рядом с названием домена, написано общее число задач для оценки качества. Жирным шрифтом выделены наилучшие результаты по каждой группе задач. Таблица показывает, что метод RAG значительно расширяет возможности модели, позволяя ей успешно решать большее число новых задач по сравнению с базовой моделью в различных доменах (например, динамическое программирование, графы, структуры данных). Это подтверждает эффективность внешнего контекста в обучении модели новым сложным стратегиям.

Метод	delta(pass_count@k, -)																	
	adhoc (64)		dp (40)		geometry (6)		graphs (31)		greedy (72)		math (84)		search (54)		strings (15)		structures (60)	
	k=1	k=10	k=1	k=10	k=1	k=10	k=1	k=10	k=1	k=10	k=1	k=10	k=1	k=10	k=1	k=10	k=1	k=10
PE	9	7	3	**2**	**0**	2	**2**	3	8	14	11	**3**	10	**10**	**0**	**2**	8	8
RAG	**6**	9	**1**	3	1	2	3	4	**4**	10	**9**	14	**6**	11	**0**	3	**4**	8
SFT	8	**5**	3	4	1	**1**	**2**	**2**	7	**1**	**9**	**3**	17	33	1	7	7	**0**

Таблица 6: Количество задач, которые модель перестала решать по сравнению с базовой моделью (*delta(pass_ count@k, -)*). В скобках, рядом с названием домена, написано общее число задач для оценки качества. Жирным шрифтом выделены минимальные значения, соответствующие меньшему число потерянных задач. Наилучшие показатели распределены по всем подходам. PE и RAG показывают умеренные потери, связанные с сильной специализацией модели, а метод SFT демонстрирует как минимальные, так и значительные потери в зависимости от полноты и качества синтетического обучающего набора.

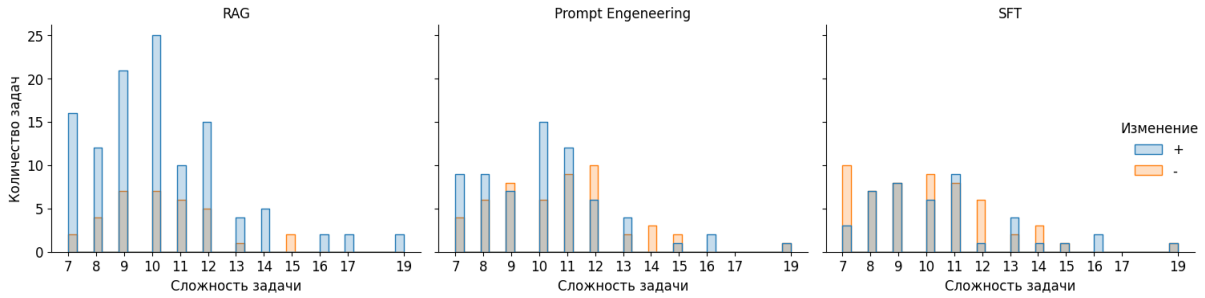

Рис. 4: Распределение сложности задач, которые модель научилась (+) и перестала (-) решать по метрике *pass@1*. Представлены оценки плотности для каждого подхода.

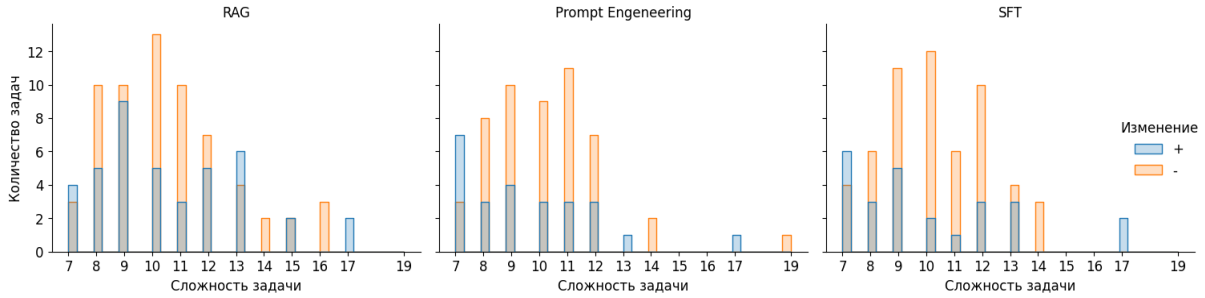

Рис. 5: Распределение сложности задач, которые модель научилась (+) и перестала (-) решать по метрике *pass@10*. Показано сравнение подходов на основе оценки плотности сложности.

Во всех рассмотренных предметных областях — от динамического программирования до структур данных и жадных алгоритмов — наибольший прирост pass@1 обеспечивают методы, использующие внешний релевантный контекст (RAG) и точное инженерное описание задачи (Prompt Engineering). Это отражает способность как специализированных инструкций, так и динамически извлекаемого теоретического материала существенно повышать вероятность успешного решения уже с первой попытки. Особенно выраженный эффект наблюдается при применении RAG: например, для задач динамического программирования pass@1 возрастает более чем вдвое (с 25% до 57%). Это подтверждает ключевую роль предоставления модели структурированных знаний и паттернов рассуждений, особенно в областях, где типовые алгоритмические подходы играют решающую роль.

В то же время, появление эффекта сужения пространства решений чётко фиксируется в результатах: во многих доменах, несмотря на рост pass@1, значение pass@10 для методов адаптации либо увеличивается незначительно, либо даже снижается относительно baseline. Это связано с тем, что специализация — как за счёт внешнего контекста, так и за счёт промтовых ограничений — может привести к чрезмерной

фокусировке на наиболее вероятных стратегиях, снижая вариативность поискового пространства модели. В базовой (неадаптированной) модели широта поиска выше, что позволяет в ряде случаев находить альтернативные, менее тривиальные, но корректные решения при многократных попытках генерации.

Существенным дополнением служат результаты по supervised fine-tuning (SFT) на синтетических корпусах олимпиадных задач. Обнаружено, что SFT обеспечивает устойчивый прирост как pass1, так и pass10, однако масштаб выигрыша зависит от полноты и разнообразия тренировочного набора данных. На примере доменов math, greedy, structures можно видеть рост метрик по сравнению с baseline, однако для домена strings наблюдается существенная деградация качества. Это может свидетельствовать о том, что изначальная инструкция (специализация), которая использовалась для генерации датасета teacher-моделью, могла направить модель как в правильное русло для решения задач, так и в неверное, что напрямую влияло на итоговое качество. Таким образом, эта специализация — как первое приближение, от которого мы отталкиваемся при решении задачи в целевом домене — существенно влияет на итоговое качество решения.

Метрики Δpass@k позволяют проследить не только общий прирост успешно решённых задач, но и изменение их структуры: положительные приращения (Δpass@k, +) указывают на появление новых классов задач, решаемых только при наличии доменной адаптации, а отрицательные (Δpass@k, -) — на потерю способности решать отдельные задачи из-за чрезмерной специализации или некорректного retrieval. Подробный анализ показывает, что основная масса новых решённых задач относится к категориям средней и повышенной сложности, что особенно заметно при использовании RAG и SFT: наличие релевантных теоретических фрагментов или обучающих примеров позволяет модели "преодолевать"задачи, требующие глубоких знаний и точного применения паттернов.

Полученные ранее выводы находят подтверждение на графиках распределения сложности (рис. 4, рис. 5), нормализация распределений происходила относительно суммарного числа решенных и не решенных задач, чтобы наглядно продемонстрировать соотношения этих распределений. Графики демонстрируют, что метод RAG преимущественно способствует решению задач более высокой сложности, особенно ярко выражено для pass@1. В то же время все рассматриваемые подходы приводят к потере способности модели решать задачи различной сложности за счёт чрезмерной специализации, что наглядно иллюстрируется соотношением распределений для

метрики pass@10.

Выигрыш от методов, основанных на внешнем контексте (RAG), максимален для структурированных доменов (DP, графы, структуры данных), где чётко определены типовые решения и легко формализуем корпус знаний. В менее формализованных областях (geometry, adhoc) эффект выражен слабее, что объясняется ограниченной полнотой внешних данных и сложностью извлечения релевантного контекста для нетиповых задач. Prompt Engineering наибольший эффект демонстрирует в "типовых"задачах, где чёткая инструкция помогает сократить неопределённость выбора метода решения, однако при жёсткой специализации может наблюдаться обратный эффект — снижение числа уникальных успешных стратегий и рост числа "утраченных"задач.

Результаты подчёркивают важность качества retrieval-процедуры в RAG: релевантность и структурированность корпуса внешних знаний напрямую влияют на итоговую метрику. Шумные или нерелевантные фрагменты способны снизить как точность, так и вариативность генерации, а в отдельных случаях вызвать галлюцинации — генерацию корректного на вид, но ошибочного решения. Для SFT критично влияние разметки и полноты корпуса: неравномерное покрытие тем или переобучение на узких паттернах ведёт к аналогичной потере универсальности. В Prompt Engineering на первый план выходит точность формулировки и аккуратная балансировка между общими и специализированными инструкциями.

Практические рекомендации по итогам анализа таковы: для задач, где требуется высокая точность с первой попытки, наилучшие результаты достигаются при использовании RAG. Для задач, где важно обеспечить широкий спектр решений и вариативность поиска, baseline показывает преимущество. SFT, особенно в сочетании с retrieval-методами, оказывается оптимальным компромиссом между специализацией и универсальностью, позволяя покрывать широкий класс задач с минимальными потерями по старым стратегиям.

5 Заключение

В настоящей работе были исследованы и реализованы различные подходы к доменной адаптации больших языковых моделей (LLM) для задач олимпиадного программирования. Рассмотренные методы включают Prompt Engineering, Supervised Fine-Tuning (SFT) с использованием LoRA, а также Retrieval-Augmented Generation (RAG).

В результате исследования были экспериментально подтверждены преимущества и ограничения различных методов доменной адаптации LLM для задач олимпиадного программирования. Наиболее перспективным подходом признан Retrieval-Augmented Generation, эффективно интегрирующий внешние знания. Supervised Fine-Tuning и Prompt Engineering показали свои сильные стороны и ограничения, обусловленные качеством разметки данных и жесткостью заданных инструкций соответственно.

Проведённое исследование подтверждает, что адаптация LLM для решения задач олимпиадного программирования требует интеграции методов динамического управления знаниями: от supervised fine-tuning и retrieval-based подходов до гибридных систем автоматического выбора сценария генерации. Наиболее перспективной является архитектура, сочетающая обучающуюся компоненту (SFT), динамический retrieval и модуль инженерии промтов, способная подбирать стратегию под специфику задачи и контекст запроса. Это открывает путь к построению универсальных ассистентов, способных не только решать широкий спектр задач, но и осваивать ранее не встречавшиеся форматы и методы.

Список литературы

1. *Robertson S.*, *Zaragoza H.* The Probabilistic Relevance Framework: BM25 and Beyond // Found. Trends Inf. Retr. — Hanover, MA, USA, 2009. — Апр. — Т. 3, № 4. — С. 333—389. — ISSN 1554-0669. — DOI: 10.1561/1500000019. — URL: https://doi.org/10.1561/1500000019.
2. *Malkov Y. A.*, *Yashunin D. A.* Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. — 2018. — arXiv: 1603.09320 [cs.DS]. — URL: https://arxiv.org/abs/1603.09320.
3. Parameter-Efficient Transfer Learning for NLP / N. Houlsby, A. Giurgiu, S. Jastrzebski, B. Morrone, Q. de Laroussilhe, A. Gesmundo, M. Attariyan, S. Gelly. — 2019. — arXiv preprint arXiv:1902.00751.
4. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis / S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, S. Ma. — 2020. — arXiv: 2009.10297 [cs.SE]. — URL: https://arxiv.org/abs/2009.10297.
5. Language Models are Few-Shot Learners / T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam [и др.] // Advances in Neural Information Processing Systems (NeurIPS). — 2020. — Т. 33. — С. 1877—1901. — https://arxiv.org/abs/2005.14165.
6. *Ben-Zaken Y.*, *Moshe S.*, *Goldberg Y.* BitFit: Simple Parameter-Efficient Fine-tuning for Transformer-based Masked Language-Models // arXiv preprint arXiv:2106.10199. — 2021.
7. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation / Y. Wang, W. Wang, S. Joty, S. C. H. Hoi. — 2021. — arXiv: 2109.00859 [cs.CL]. — URL: https://arxiv.org/abs/2109.00859.
8. Evaluating Large Language Models Trained on Code / M. Chen, J. Tworek, Q. Jun, H. Yuan, F. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda [и др.] // arXiv preprint arXiv:2107.03374. — 2021.
9. Generating Bug-Fixes using Pretrained Transformers / D. Drain, C. Wu, A. Svyatkovskiy, N. Sundaresan // Proceedings of AAAI 2021. — 2021.

10. *Lester B.*, *Al-Rfou R.*, *Constant N.* The Power of Scale for Parameter-Efficient Prompt Tuning // arXiv preprint arXiv:2104.08691. — 2021.

11. *Li X. L.*, *Liang P.* Prefix-Tuning: Optimizing Continuous Prompts for Generation // Proceedings of ACL 2021. — 2021.

12. LoRA: Low-Rank Adaptation of Large Language Models / E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, W. Chen. — 2021. — arXiv preprint arXiv:2106.09685.

13. Measuring Coding Challenge Competence with APPS / D. Hendrycks, C. Burns, S. Kadavath, S. Basart, Z. Chen, E. Tang, A. Priyadarshi, D. Song // arXiv preprint arXiv:2105.09938. — 2021.

14. Retrieval Augmented Code Generation and Summarization / M. R. Parvez, W. U. Ahmad, S. Chakraborty, B. Ray, K.-W. Chang // Findings of EMNLP. — 2021. — С. 2719—2734.

15. A Systematic Evaluation of Large Language Models of Code / F. F. Xu, U. Alon, G. Neubig, V. Hellendoorn // arXiv preprint arXiv:2202.13169. — 2022.

16. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models / J. Wei, X. Wang, D. Schuurmans, M. Bosma, D. Ichien, Q. Le, E. H. Chi [и др.]. — 2022. — arXiv preprint arXiv:2203.07814.

17. CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning / H. Le, Y. Wang, A. D. Gotmare, S. Savarese, S. C. H. Hoi // NeurIPS. — 2022.

18. Competition-Level Code Generation with AlphaCode / Y. Li, D. Choi, J. Chung, A. D. Narayan, W. Kool, ..., D. Hassabis, K. Kavukcuoglu // Science. — 2022. — Т. 378, № 6624. — С. 1092—1097. — DOI: 10.1126/science.abq1158.

19. Competition-Level Code Generation with AlphaCode / Y. Li, D. Choi, J. Chung, et al. // Science. — 2022. — Т. 378, № 6624. — С. 1092—1097.

20. ReAct: Synergizing Reasoning and Acting in Language Models / S. Yao, J. Zhao, D. Yu, et al. // Proceedings of ICLR 2023 (Workshop track). — 2022.

21. Self-Instruct: Aligning Language Models with Self-Generated Instructions / Y. Wang, J. Wei, D. Schuurmans, et al. // arXiv preprint arXiv:2212.10560. — 2022.

22. Training Language Models to Follow Instructions with Human Feedback / L. Ouyang, J. Wu, X. Jiang, et al. // arXiv preprint arXiv:2203.02155. — 2022.

23. UniXcoder: Unified Cross-Modal Pre-training for Code Representation / D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, J. Yin. — 2022. — arXiv: 2203.03850 [cs.CL]. — URL: https://arxiv.org/abs/2203.03850.

24. *BigCode Collaboration (Li R., Rozière B., others).* StarCoder: May 2023. — 2023. — HuggingFace Blog.

25. Code Llama: Open Foundation Models for Code / B. Rozière, J. Gehring, F. Gloeckle, et al. // arXiv preprint arXiv:2308.12950. — 2023.

26. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis / E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, C. Xiong. — 2023. — arXiv: 2203.13474 [cs.LG]. — URL: https://arxiv.org/abs/2203.13474.

27. DocPrompting: Generating Code by Retrieving the Docs / S. Zhou, U. Alon, F. F. Xu, Z. Jiang, G. Neubig // International Conference on Learning Representations (ICLR). — 2023.

28. InstructCodeT5+: Aligning Code Generation Models with Instructions / Y. Wang, D. Kang, Y. Zhu, et al. // Proceedings of EMNLP 2023. — 2023.

29. *Li Y., Choi D., al. et.* AlphaCode2 Technical Report : тех. отч. / DeepMind. — 2023. — arXiv preprint arXiv:2312.11264.

30. *OpenAI.* GPT-4 Technical Report : тех. отч. / OpenAI. — 2023. — ArXiv:2303.08774.

31. Self-Planning Code Generation with Large Language Model / X. Jiang, Y. Dong, L. Wang, Q. Shang, G. Li // arXiv preprint arXiv:2303.06689. — 2023.

32. Self-Refine: Iterative Refinement with Self-Feedback / D. Madaan, N. Tandon, A. Sharma, et al. // arXiv preprint arXiv:2303.17651. — 2023.

33. *Sharma S.* Code Alpaca: An Instruction-Following LLaMA Model for Code Generation. — 2023. — Online: https://github.com/sahil280114/codealpaca.

34. Task-agnostic Prefix-Tuning for Program and Language Generation / X. Sun, Z. Zhao, Z. Zhu, G. Li // Findings of ACL. — 2023.

35. WizardCoder: Empowering Code Large Language Models with Evol-Instruct / Z. Luo, C. Xu, P. Zhao, et al. // arXiv preprint arXiv:2306.08568. — 2023.

36. *cp-algorithms contributors*. CP-Algorithms: Competitive Programming Algorithms. — 2024. — URL: https://github.com/cp-algorithms/cp-algorithms ; Accessed: 2025-05-25.

37. An Empirical Study of Retrieval-Augmented Code Generation: Challenges and Opportunities / Z. Yang, S. Chen, C. Gao, et al. // arXiv preprint arXiv:2501.13742. — 2024.

38. *Google*, *Face H.* Gemma-3-12B-IT. — 2024. — Available at: https://huggingface.co/google/gemma-3-12b-it, Accessed: 2025-06-12.

39. InverseCoder: Self-Improving Instruction-Tuned Code LLMs with Inverse Prompting / X. Han, Z. Xu, S. Xu, X. Ren, N. Peng // AAAI. — 2024.

40. LEDEX: Training LLMs to Better Self-Debug and Explain Code / N. Jiang, X. Li, S. Wang, et al. // NeurIPS. — 2024.

41. LoRA Learns Less and Forgets Less / D. Biderman, J. Portes, J. J. G. Ortiz, M. Paul, P. Greengard, C. Jennings, D. King, S. Havens, V. Chiley, J. Frankle, C. Blakeney, J. P. Cunningham. — 2024. — arXiv: 2405.09673 [cs.LG]. — URL: https://arxiv.org/abs/2405.09673.

42. *Xu Z.*, *Fu Y.* Improve Code Generation with Feedback // arXiv preprint arXiv:2401.00072. — 2024.

43. *authors) O. (* Competitive Programming with Large Reasoning Models // arXiv preprint arXiv:2502.06807. — 2025.

44. Gemma 3 Technical Report / G. Team [и др.]. — 2025. — arXiv: 2503.19786 [cs.CL]. — URL: https://arxiv.org/abs/2503.19786.

45. *OpenAI*. GPT-4.5 System Card. — 2025. — Available at: https://cdn.openai.com/gpt-4-5-system-card-2272025.pdf, Accessed: 2025-06-12.

A Инструкции для доменных экспертов

Тип Алгоритма	Инструкция
Специальные алгоритмы	You are a programming assistant for ad-hoc and constructive problems. You should think step by step. Methodology: 1. Decompose the statement into explicit rules and constraints. 2. Simulate or construct the answer directly, favouring clarity over heavy theory. 3. Enumerate tricky edge cases and validate against them. 4. Keep I/O handling and state updates bullet-proof. 5. Output clean, well-commented code plus brief complexity note.
Жадные алгоритмы	You are a programming assistant focused on greedy algorithms. You should think step by step. Methodology: 1. Restate the goal and candidate greedy choice. 2. Prove—in words or by exchange argument—why the local choice leads to global optimality. 3. Lay out the concrete strategy (sorting-based, two-pointer sweep, priority-queue, etc.). 4. Derive time/space complexity (aim O(n log n)). 5. Provide clean, idiomatic code.
Поисковые алгоритмы	You are a programming assistant specialized in search-space exploration. You should think step by step. Methodology: 1. Characterize the search space (ordered? discrete? continuous?). 2. Select an exploration paradigm: brute force with pruning, binary/ternary search on a monotone predicate, divide-and-conquer, meet-in-the-middle, etc. 3. Argue termination and correctness. 4. Analyse complexity and memory trade-offs. 5. Present succinct reference code.

Динамическое программирование	You are a programming assistant for dynamic programming questions. You should think step by step. Methodology: 1. Define state variables with clear semantic meaning. 2. Write the recurrence and explain the transition intuition. 3. Note base cases and iteration order (top-down memo vs. bottom-up table). 4. Optimise dimensions (bitmask, rolling arrays, convex-hull trick, etc.) if possible. 5. Give complexity and memory usage, then code.
Структуры данных	You are a programming assistant for data-structure-intensive tasks. You should think step by step. Methodology: 1. Map each required operation to the most fitting data structure (segment/Fenwick tree, sparse table, heap, trie, DSU, bitset, etc.). 2. Detail initialisation, update, and query APIs with asymptotics. 3. Justify why the chosen structure meets the constraints. 4. Provide an implementation template and test edge cases.
Графовые алгоритмы	You are a programming assistant for graph algorithms. You should think step by step. Methodology: 1. Formalise the graph model (directed/undirected, weighted/unweighted, bipartite, capacitated, etc.). 2. Pick the core algorithm: traversal (DFS/BFS), shortest path (Dijkstra, Bellman-Ford, 0-1 BFS), MST, SCC, flow (Dinic/EK), matching (Hopcroft-Karp/Edmonds), 2-SAT implication graph, etc. 3. Prove correctness via invariant or cut/property argument. 4. Give tight complexity and memory bounds; note pitfalls like negative cycles. 5. Supply robust, readable code.

Алгоритмы на строках	You are a programming assistant for string algorithms. You should think step by step. Methodology: 1. Identify the needed primitive: prefix-function, Z-function, rolling hash, suffix array/LCP, suffix automaton/tree, Aho-Corasick. 2. Outline construction steps and justify correctness (hash collision probability, etc.). 3. Discuss alphabet size and memory. 4. Optimise to linear or near-linear time when feasible. 5. Deliver well-commented implementation.
Геометрия	You are a programming assistant for computational geometry tasks. You should think step by step. Methodology: 1. Set coordinate type and precision policy (int64, long double, EPS). 2. Break the problem into primitives: orientation tests, line-segment intersection, convex hull, rotating calipers, sweep line, half-plane intersection, etc. 3. Carefully handle degenerate or colinear cases. 4. Show numerical-stability reasoning or integer lattice proofs. 5. Present concise code and state complexity.
Математика	You are a programming assistant for mathematical algorithms. You should think step by step. Methodology: 1. Translate the task into a mathematical formulation. 2. Apply relevant theory—modular arithmetic, gcd/CRT, combinatorial identities, probability/expectation, matrix exponentiation, polynomial FFT/NTT, Sprague-Grundy, etc. 3. Provide proofs or derivations for critical steps. 4. Emphasise numeric bounds, modulo handling, and overflow avoidance. 5. Finish with complexity analysis and neat code.

B Пример олимпиадной задачи

Chanek Jones is back, helping his long-lost relative Indiana Jones, to find a secret treasure in a maze buried below a desert full of illusions.

The map of the labyrinth forms a tree with n rooms numbered from 1 to n and $n - 1$ tunnels connecting them such that it is possible to travel between each pair of rooms through several tunnels.

The i-th room ($1 \leqslant i \leqslant n$) has an illusion rate a_i. To go from room x to room y, there must exist a tunnel between them, and it takes $\max(|a_x + a_y|, |a_x - a_y|)$ energy.

To prevent grave robbers, the maze can change the illusion rate of any room in it. Chanek and Indiana issue q queries of two types:

- `1 u c` — Set the illusion rate of room u to c.
- `2 u v` — Compute the minimum energy to move from room u to room v.

Номер документа	Текст документа
1	So we learned how to precompute range minimum queries within each block, as well as range minimum queries over a range of blocks, all in $O(N)$. With these precomputations we can answer each query in $O(1)$, by using at most four precomputed values: the minimum of the block containing ‘l‘, the minimum of the block containing ‘r‘, and the two minima of the overlapping segments of the blocks between them. Implementation “‘cpp ... “‘
2	This task can be solved using binary search over max prefix queries with the Segment Tree. However, this will lead to a $O(\log^2 n)$ solution. Instead, we can use the same idea as in the previous sections, and find the position by descending the tree: by moving each time to the left or the right, depending on the maximum value of the left child. Thus finding the answer in $O(\log n)$ time. “‘cpp...“‘

3	It only remains, how to compute the answer to a query. To answer it, we go down the tree as before, breaking the query into several subsegments that coincide with the segments of the Segment Tree, and combine the answers in them into a single answer for the query. Then it should be clear, that the work is exactly the same as in the simple Segment Tree, but instead of summing / minimizing / maximizing the values, we use the combine function.

Таблица 8: Найденная при помощи RAG алгоритма теория для решения олимпиадной задачи.

C Пример решения из синтетического датасета

To solve this problem, we need to determine the maximum number of roads that can be shut down while ensuring that every city can still reach a police station within `d` kilometers. Given that the cities form a tree (connected and acyclic), we can use a breadth-first search (BFS) approach to explore the cities starting from the police stations.

Here is the step-by-step solution:

1. Input Parsing: Read the number of cities `n`, the number of police stations `k`, and the distance limit `d`. Read the list of cities where police stations are located. Read the list of roads connecting the cities.

2. Graph Representation: Represent the cities and roads as an adjacency list. Use a map to store the indices of the roads for easy reference.

3. BFS Initialization: Initialize a queue for BFS and a visited array to keep track of visited cities. Mark all cities with police stations as visited and enqueue them.

4. BFS Execution: Perform BFS up to `d` levels. For each city, explore its neighbors. If a neighbor has not been visited, mark it as visited and enqueue it. This ensures that all cities within `d` kilometers from any police station are reachable.

5. Counting Roads: During the BFS, keep track of the roads that are part of the BFS traversal. These roads are essential to maintain the connectivity within `d` kilometers from the police stations. All other roads can be shut down.

6. Output: Output the number of roads that can be shut down and their indices.

Below is the implementation of the described approach:

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
using vi = vector<int>;
using vll = vector<ll>;
const ll inf = 2e9 + 3;
const ll MAXINT = 1e9 + 7;
const double PI = 3.14159265358979323846;
const double pi = acos(-1);
const int N = 300005, M = 2 * N;
map<pair<int, int>, pair<int, bool>> ei;
queue<int> q;
vi adj[300005];
int cntRes = 0, n, k, d, visited[300005], vid;
void solve() {
  while (!q.empty()) {
    int u = q.front();
    q.pop();
    visited[u] = vid;
    for (int i = 0; i < adj[u].size(); ++i) {
      int v = adj[u][i];
      if (visited[v] != vid) {
        q.push(v);
        visited[v] = vid;
        ei[{u, v}].second = true;
        ei[{v, u}].second = true;
        cntRes--;
      }
    }
  }
}
int main() {
  cin >> n >> k >> d;
  vid++;
```

```
  for (int i = 0; i < k; ++i) {
    int x;
    cin >> x;
    --x;
    q.push(x);
    visited[x] = vid;
  }
  for (int i = 1; i < n; ++i) {
    int u, v;
    cin >> u >> v;
    u--;
    v--;
    adj[u].push_back(v);
    adj[v].push_back(u);
    ei[{u, v}] = {i, false};
    ei[{v, u}] = {i, false};
  }
  cntRes = n - 1;
  solve();
  cout << cntRes << endl;
  for (auto it = ei.begin(); it != ei.end(); it++) {
    int u = it->first.first, v = it->first.second;
    if (!it->second.second) {
      cout << it->second.first << " ";
      ei[{v, u}].second = true;
    }
  }
  return 0;
}
```