Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем
Системное программирование

Люлина Елена Сергеевна

Система анализа поведения студентов при обучении программированию

Выпускная квалификационная работа бакалавра

Научный руководитель:
доц. кафедры СП, к. т. н. Т. А. Брыксин

Рецензент:
генеральный директор
ООО “Цифровые образовательные решения”
Н. И. Вяххи

Санкт-Петербург
2020

SAINT-PETERSBURG STATE UNIVERSITY

Software and Administration of Information Systems
System Programming

Elena Lyulina

The analysis system of students' behavior while learning computer programming

Bachelor's Thesis

Scientific supervisor:
associate professor Timofey Bryksin

Reviewer:
CEO of Digital Learning Solutions LLC
Nilolay Vyahhi

Saint-Petersburg
2020

Оглавление

Введение

При обучении программированию очень важно наладить процесс решения задач, так как студенты должны научиться практически применять знания, полученные на лекциях. В процессе решения задач новички знакомятся с особенностями языков программирования, разными вариантами архитектуры программ, инструментами и средами программирования. Все эти знания крайне необходимы для дальнейшей работы и не могут быть усвоены на лекциях. При этом и сам код, и процесс его написания сильно зависят от опыта программиста. Важно понимать, в чем именно заключаются отличия процесса и качества кода новичка и опытного разработчика, чтобы эффективнее организовать преподавание программирования, уделяя внимание наиболее отличающимся моментам. В этой области было проведено много исследований поведения новичков, касающихся, например, процессов редактирования и написания кода [4, 20], его компиляции [11, 12], паттернов программирования [5], а так же различий поведения в средах разработки (Integrated Developement Environment, IDE) в зависимости от опыта [15]. Все эти работы основаны на записи определенных действий пользователя в среде программирования с их последующим анализом и описывают различные аспекты одного и того же действия: процесса программирования начинающими.

Также проводились различные исследования для выяснения того, насколько процесс написания кода и использование IDE сложны для новичков. В работе [14] проводился опрос большого количества студентов, и именно самостоятельное решение задач по программированию получило наивысшую оценку сложности, обойдя теоретические занятия и практические занятия в классе. При самостоятельном решении задач студентам зачастую необходима дополнительная помощь преподавателя, так как недостаток знаний и опыта не позволяют им создать правильное решение. Однако возможность получить квалифицированную помощь или ограничена ресурсами учителя, или вовсе отсутствует (в случае самостоятельного обучения или использования онлайн-курсов).

Эта проблема порождает большое количество исследований, нацеленных на автоматизацию помощи и подсказок новичкам. Среди них — помощь в моделировании [7] и парном программировании [21], сигнализация об ошибках в коде и их исправление [19], улучшение стиля кода [1] или его производительности [8].

Один из подходов к автоматизации подсказок посвящён исследованию процесса программирования. В нём для текущего состояния программы генерируется следующий шаг ее написания, приближающий студента к правильному решению задачи. Для этого используются данные о предыдущих решениях этой задачи другими студентами и построенное на их основе пространство решений. Такой способ автоматизации подсказок был использован для генерации возможного исправления решений задач на логическом языке программирования [3], в блоковом программировании [2], а также для решения небольших задач на языке Python [18]. Основное отличие первых двух подходов от последнего — использование не только финальных решений задач, но и промежуточных переходов, что позволяет применять алгоритмы поиска наилучшего пути в пространстве решений и положительно сказывается на результатах. Однако, отсутствие данных с промежуточными переходами, а также больш*а*я вариативность решений не позволяют расширить этот алгоритм для учебных произвольной сложности.

Данная работа призвана решить эти проблемы и расширить существующий метод генерации подсказок для языка Python. Для этого было решено создать инструмент для сбора данных с промежуточными решениями в виде расширения к популярным средам программирования. Данные, собранные в процессе работы, очень важны из-за широкой сферы применимости: анализ стратегий решений и исправления ошибок, непонятых концепций программирования, наиболее частых ошибок, паттернов программирования, среднего времени решения, сложности задач и так далее. Понимание всех перечисленных задач поможет преподавателям, составителям онлайн-курсов и разработчикам сред программирования для начинающих. В частности, результаты, полученные в ходе этой работы, планируется использовать в проекте

JetBrains под названием EduTools [6], нацеленном на обучение в среде программирования при помощи решений различных задач.

Постановка задачи

Целью работы является разработка алгоритма генерации подсказок для решения учебных задач на языке Python. Для достижения этой цели были поставлены следующие задачи:

- ознакомиться с предметной областью и сделать обзор существующих решений;
- собрать датасет с процессами решений задач программистами разного опыта;
- разработать алгоритм генерации подсказок, основанный на собранных данных;
- протестировать алгоритм и оценить релевантность подсказок.

1. Обзор

Во время решения задач у обучающихся может возникнуть множество проблем, вызванных самыми разными причинами от неумения пользоваться средой программирования и непонимания сообщений компилятора до неправильной архитектуры программы и оформления кода. С каждой из этих проблем студенту помогают справиться подсказки разного типа: объяснениями, что пошло не так, возможными вариантами исправления проблемы, подбадривающими советами, учебными материалами и так далее. Генерация подсказок иногда зависит от *текущего решения* студента — решения, которое написал студент в момент генерации подсказки. Авторы обзорной статьи [13] выделяют пять типов подсказок, каждый из которых имеет несколько подтипов:

1. Знание об ограничениях задачи:
    - подсказки о требованиях задачи (например, использовать определенный язык программирования или конструкцию);
    - подсказки о правилах решения задачи (содержат общую информацию о решении задачи, не опираясь на текущее решение студента).
2. Знание о понятиях:
    - подсказки с пояснением темы задачи;
    - подсказки с иллюстрациями примеров.
3. Знание об ошибках:
    - подсказки о не пройденных тестах;
    - подсказки об ошибках компилятора;
    - подсказки об ошибках в решении (например, о логических ошибках);
    - подсказки о проблемах со стилем кода (от ошибок с документацией и форматированием до ошибок в архитектуре программы);

- подсказки о производительности (использование ресурсов и время исполнения).

4. Знание о следующих действиях:

   - подсказки об исправлении ошибок;
   - подсказки о следующих шагах для решения задачи;
   - подсказки об улучшении решения (например, производительности или архитектуры).

Последний тип подсказок требует наибольшего участия преподавателя и поэтому особенно труден для автоматизации. В то же время, автоматизация именно этого типа подсказок может принести большую пользу в случае онлайн-курсов или самостоятельного решения задач, так как поможет студентам исправлять свои ошибки и решать задачи без помощи преподавателя. В связи с этим, в данной работе предлагается генерировать подсказки о действиях, приближающих к успешному решению задачи. Подсказка генерируется для текущего решения студента и представляет собой следующий фрагмент кода, который приблизит студента к решению задачи.

Существующие алгоритмы генерации подсказок о следующих шагах опираются на данные о предыдущих решениях этих задач. Именно от наличия этих данных, их объема и полноты во многом зависят подходы генерации подсказок. Например, данные могут содержать не только решения задач, но и отображать динамику решения. Для формального определения динамики решения введем несколько дополнительных определений. *Финальным решением* задачи будем называть фрагмент кода, который полностью решает задачу. *Промежуточным решением* задачи будем называть фрагменты кода, которые появлялись в процессе решения задачи и не являются финальными. Тогда сам процесс решения задачи можно представить *динамикой решения* задачи — графом, соединяющим все промежуточные решения в порядке их появления и финальное решение. (Рис. 1).

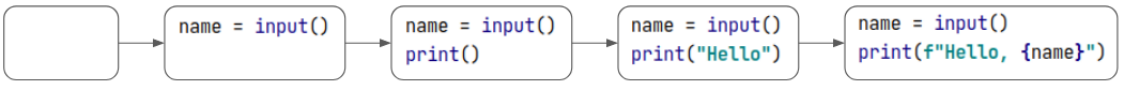


Рис. 1: Пример динамики решения задачи, соединяющей все последовательные промежуточные решения с финальным решением.

1.1. Генерация подсказок с использованием динамик решения задач

Основная идея подходов этого типа — объединение одинаковых вершин в динамиках решения задач различных студентов и получение общего графа, называемого *пространством решений* (Рис. 2). Вершинами в нем являются решения (промежуточные или финальные). Ребро между вершинами i и j существует, если существует студент, которые перешел от решения i к решению j.

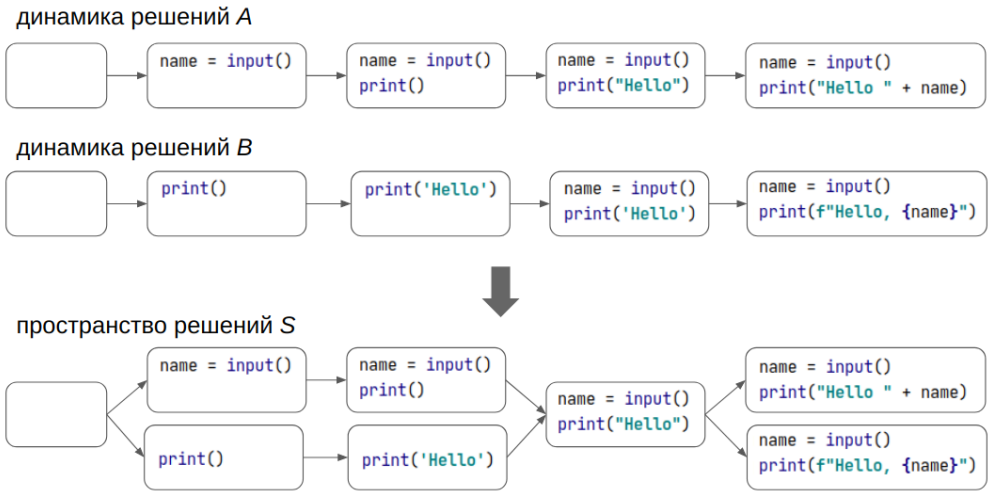


Рис. 2: Построение пространства решений S объединением одинаковых вершин динамик решения A и B.

Как показывают существующие подходы генерации подсказок [3, 2], при наличии данных с промежуточными решениями алгоритм генерации подсказок можно разбить на следующие шаги:

1. Определение *одинаковых* решений.

2. Определение, какие решения следует брать в качестве промежуточных (например, можно брать только синтаксически корректные фрагменты).

3. Объединение одинаковых вершин различных динамик решений в пространство решений.

4. Нахождение вершины, соответствующей текущему решению студента, в пространстве решений.

5. Поиск оптимального пути в пространстве решений от найденной вершины к финальному решению.

6. Генерация подсказки в виде следующей вершины после найденной на оптимальном пути.

Реализация этих шагов варьируется в зависимости от подходов, однако общий алгоритм остается одним и тем же. Далее приводится описание того, как эти подзадачи решают два различных подхода генерации подсказок, использующих промежуточные решения.

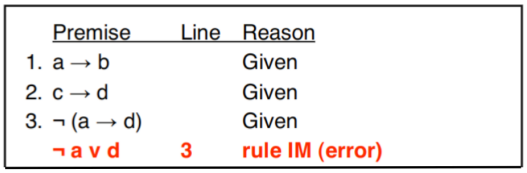
1.1.1. Марковский процесс принятия решений

Первый из алгоритмов, рассматриваемый в данном обзоре, основан на марковском процессе принятия решения и описан в статье [3], где он был применен к программам на логическом языке программирования. Каждая строка программы состоит из посылки, правила и номера строки, к которой оно было применено (Рис. 3).

Авторы определяют одинаковые решения через равенство наборов посылок, которые содержатся в этих решениях. При этом оцениваются результаты с использованием четырёх вариантов наборов:

- упорядоченный набор посылок;
- неупорядоченный набор посылок;
- упорядоченный набор без последней посылки;
- неупорядоченный набор без последней посылки.

В последних двух двух вариантах исключается последняя посылка для генерации подсказки в случае, если студент предпринял шаг, который до него еще никто не принимал.



Hint #	Hint Text
1	Try to derive: a ^ ¬ d
2	Use line 3, ¬ (a → d) to derive it
3	Use the rule: IM, implication
4	Enter a ^ ¬ d with ref. line 3 and rule IM implication

Рис. 3: Пример подсказки в алгоритме [3]. На верхнем рисунке приведена программа с неправильной четвертой строкой, выделенной красным цветом. Ниже представлена подсказка, приводящая корректный вывод, примененное правило и строку, к которой оно было применено.

В качестве промежуточных решений авторы используют решения с корректными строками — строками, содержащими посылку, номер строки и правило. Пользуясь таким определением, авторы строят пространство решений из всех доступных динамик решений, объединяя вершины с одинаковыми наборами посылок в соответствии с определенной метрикой.

Далее ищется одинаковая с текущим кодом студента вершина в пространстве решений. Если такой вершины не существует, то генерация подсказки невозможна.

Для поиска оптимального пути из найденной вершины к финальной авторы строят марковский процесс принятия решений (MDP) с наградами для каждого финального решения и штрафами для промежуточных. Переход по дугам тоже штрафуется для укорачивания пути к финальному решению. Подсказкой является следующая вершина после найденной на оптимальном пути.

Авторы взяли данные с динамиками решения задачи логического программирования, собранные за 4 семестра, в каждом из которых училось около 220 студентов. Эти данные содержали 523 решения задачи. Чтобы оценить работу алгоритма, эти данные разбивались на два набора: набор для обучения марковского процесса принятия решений и

набор для тестирования подсказок. Для последнего считались две метрики:

1. **Move mathches**: процент решений, для которых есть такая же вершина в пространстве решений, что дает оценку вероятности генерации подсказки.

2. **Unique mathches**: процент уникальных решений, для которых есть такая же вершина в пространстве решений, что дает оценку пересечения данных в тренировочном и тестовом наборах.

После этого авторы проанализировали, как влияет выбранный способ определения одинаковых решений и количество данных (взятых по семестрам) на выбранные метрики. Например, для каждого способа авторы натренировали MDP по данным 1-го, 2-х и 3-х семестров (Рис. 4).

Определение одинаковых решений	MDP, обученная по данным 1 семестра	MDP, обученная по данным 2 семестров	MDP, обученная по данным 3 семестров
упорядоченный набор посылок	72.79%	79.57%	82.32%
неупорядоченный набор посылок	79.62%	85.22%	87.26%
упорядоченный набор посылок без последней	79.88%	87.84%	91.57%
неупорядоченный набор посылок без последней	85.00%	91.50%	93.96%

Рис. 4: Значение метрики **Move matches** для каждого способа определения одинаковых решений в зависимости от размера тренировочного набора данных (данные за 1 семестр, 2 или 3).

Такая оценка показывает, что подсказка будет сгенерирована для большинства решений, однако, ничего не говорит о том, насколько хороша будет эта подсказка.

1.1.2. Использование политики решения задач

Данный алгоритм описан в статье [2] и автоматизирует генерацию подсказок для задач блокового языка программирования (Рис. 5). Ре-

шения задач являются одинаковыми, если состоят из одинаковых упорядоченных наборов блоков.

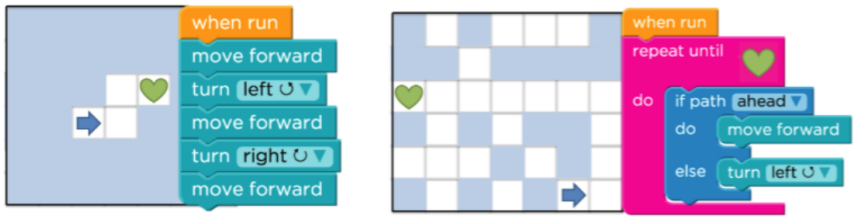


Рис. 5: Задачи на нахождение пути в лабиринте, использующиеся в статье [2]. Решения составлены из блоков (справа для каждой задачи) и представляют собой траекторию пути в лабиринте (слева для каждой задачи).

В качестве промежуточных решений берутся решения, которые студенты отправили на проверку. Так как между двумя последовательно отправленными решениями могут быть существенные различия, авторы интерполируют решения между двумя последовательными, делая различия между ними в одно изменение (изменением может считаться добавление или удаление блока и изменение переменной).

Пространство решений представляется в виде графа с объединенными динамиками решений. На рисунке (Рис. 6) представлено построенное пространство решений для одной из задач, при этом толщина дуг зависит от оптимальности перехода.

Во время исследований из-за небольшой вариативности перестановки блоков в задачах и достаточно большого объема данных для текущего решения студента всегда находилась одинаковая вершина.

Для поиска оптимального пути авторы вводят определение политики решения задачи.

Определение 1 *Пусть S — набор промежуточных решений. Политика решения задачи $\pi : S \to S$ определяет следующее промежуточное решение $s' = \pi(s)$ для всех $s \in S$.*

Теперь для поиска подсказки достаточно определить политику решения задачи. В статье рассмотрены несколько политик, здесь приведена наиболее успешная — политика Пуассоновского пути.

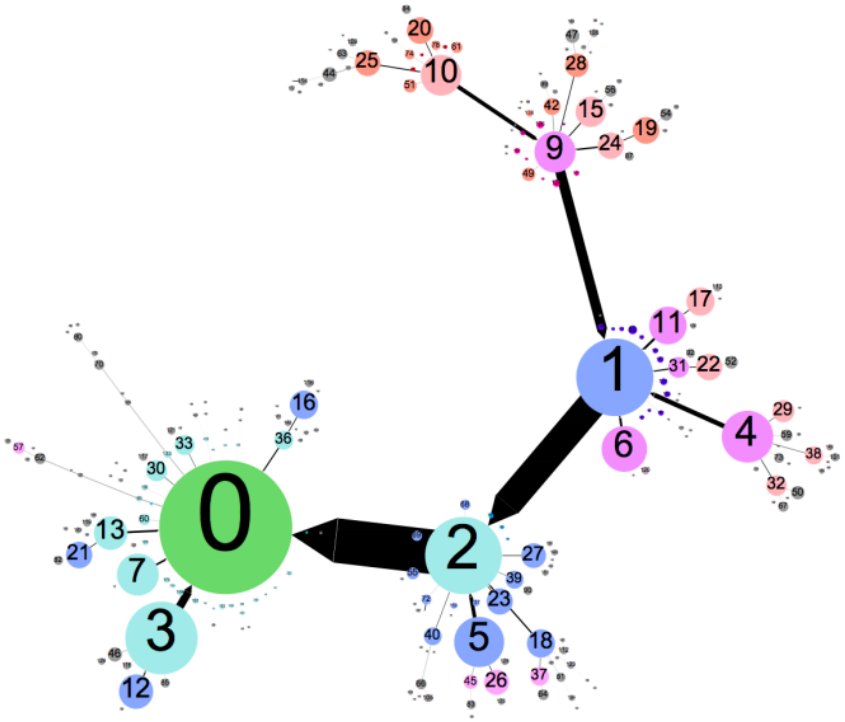


Рис. 6: Пространство решений для задачи блокового программирования [2]. Решение, помеченное меткой 0, является финальным.

Определение 2 *Пуассоновский путь*

$$\gamma(s) = \arg\min_{p \in Z(s)} \sum_{x \in p} \frac{1}{\lambda_x}$$

где $Z(s)$ *— все пути от промежуточного решения* s *к финальному,* λ_x *— количество посещений промежуточного решения* x *предыдущими студентами. Тогда политикой решения задачи* $\pi(s)$ *будет первое промежуточное решение на пути* $\gamma(s)$.

Такое определение обусловлено тем, что оптимальный путь — путь, занимающий меньше всего времени. Время, проведенное на каждом промежуточном решении, можно рассматривать как пуассоновский процесс. Его параметр можно получить из количества студентов, достигших этого промежуточного решения. Как следует из определения политики решений, подсказкой будет первое промежуточное решение на найденном оптимальном пути.

Данные включали в себя решения двух задач блокового программирования, собранных на платформе Code.org (Рис. 7). Эксперты вручную

создали в среднем по 250 подсказок для каждой задачи и использовали эти данные для *правильной* политики решения задач. Каждую из остальных предложенных политик авторы сравнивали с данной. Политика Пуассоновского пути показала наилучшие результаты, достигнув точности в 95.9% и 84.4% для двух задач соответственно.

Статистика	PA	PB
Количество студентов	509,405	263,569
Количество решений задач	1,138,506	1,263,360
Количество уникальных решений задач	10,293	79,553

Рис. 7: Статистика по собранным данным для задач PA и PB

1.2. Генерация подсказок без использования динамик решения задач

Данные с промежуточными решениями задач довольно трудно собрать, так как это требует непрерывной записи всех изменений решения, что редко реализовано в средах программирования или онлайн-курсах. Однако, финальные решения обычно сохраняются во время сдачи задачи, так что такие данные собрать легче. Авторы данной статьи [18] предлагают подход генерации подсказок для языка Python с использованием только финальных решений, без динамик решения задач.

Язык Python предлагает бо́льшую вариативность по сравнению с блоковыми и логическим языками программирования, поэтому существует больше вариантов решения для одной и той же задачи. Каждое решение можно представить в виде абстрактного синтаксического дерева (*abstract syntax tree, AST*), однако решения, одинаковые с точностью до названий переменных или порядка операндов сравнения, все

еще будут считаться разными. Авторы статьи пытаются обойти эту проблему, предложив алгоритм *каноникализации*. Его цель — унификация решений, что позволяет сжать пространство решений и обойти вариативность языка Python, сконцентрироваться на семантике и опустить способы реализации. Алгоритм каноникализации включает в себя следующие трансформации AST:

- анонимизация кода (переименование переменных и функций);
- упрощение математических операций;
- подстановка значений переменных;
- подстановка функций вместо их вызова;
- удаление недостижимого кода;
- упорядочение слагаемых;
- унификация логических выражений и операций сравнения.

Первым шагом алгоритма является анонимизация кода, которую применяют к исходному AST, получая *анонимизированное AST* (*aAST*). Затем к нему применяется остальной набор трансформаций для получения *каноникализированного AST* (*cAST*).

Между двумя AST существует набор изменений (*edits*), последовательное применение которых превратит первое дерево во второе. К таким изменениям относится добавление, удаление или перемещение вершины в AST. В то же время количество изменений между двумя деревьями служит подобием “расстояния” между деревьями: если изменений мало, то деревья “близки” или “похожи”. Авторы также предлагают возможность применения изменений с учетом трансформаций каноникализации, позволяя для двух деревьев сопоставлять вершины, одинаковые после применения трансформаций. Например, это позволяет сопоставлять переменные с разными именами, но одинаковым значением. Для этого перед применением изменений между двумя решениями они

сравнивают количество изменений между aAST этих решений с количеством изменений между cAST этих решений и применяют те, которых меньше.

Алгоритм нахождения оптимального пути можно разделить на несколько этапов:

1. Поиск оптимального финального решения.

   За неимением динамик решения для текущего cAST ищется ближайшее финальное решение по количеству изменений между cAST решений. Однако, может оказаться, что не все эти изменения необходимо применять, поэтому авторы используют перебор всех возможных изменений. Они рассматривают все элементы множества подмножеств данного набора, применяют их к текущему решению и оценивают результат с помощью тестов. Если результат является решением, он объявляется новым оптимальным финальным решением.

2. Определение оптимальных промежуточных вершин.

   Когда оптимальное финальное решение установлено, устанавливается и набор изменений, который необходимо применить к текущему решению для достижения найденного финального. Если последовательно применять изменения из этого набора, получатся некие промежуточные решения. Среди них рассматриваются такие, что они:

   - синтаксически корректны;
   - ближе к финальному решению, чем текущее решение;
   - улучшают результат тестов.

   Эти промежуточные решения оцениваются по формуле, учитывающей количество посещений решения, расстояние до финального решения, результат тестов и расстояние до текущего решения. В качестве первого промежуточного выбирается решение с наивысшей оценкой. Следующие решения выбираются по этой же оценке

среди тех, кто находится после (с точки зрения применения изменений) первого выбранного решения.

3. Генерация подсказки.

   Как только промежуточные вершины определены, первое из них выдается в качестве подсказки.

Авторы статьи использовали данные с решениями 15 задач от 15 студентов. Задачи были выбраны довольно простые, содержащие только операции сравнения и условные операторы. Всего было собрано 294 синтаксически корректных решения, которые были взяты для тестирования генерации подсказки. Для каждого фрагмента было оценено, смог ли алгоритм сгенерировать последовательный набор подсказок, которые привели к финальному решению, и сколько в среднем подсказок содержится в таком наборе. К финальному решению удалось привести 98.3% фрагментов, при этом все неудавшиеся попытки относились к самой сложной задаче. При этом, насколько хорошими являлись полученные подсказки и можно ли их советовать студентам, оценено не было.

Одним из плюсов работы [18], по мнению авторов, является возможность генерировать подсказки даже для тех состояний программ, которые ни разу не появлялись в предыдущих процессах решения. Однако в работе представлена апробация только для небольших задач (Рис. 8), причем на самой сложной задаче алгоритм показал наихудший результат, что делает его неприменимым для более сложных задач. В дополнение к этому для поиска промежуточных решений используется перебор булеана множества найденных изменений, экспоненциально увеличивая время работы при линейном увеличении размеров задач. Это можно обойти при наличии данных с промежуточными решениями, что также позволит использовать существующие подходы с применением пространства решений.

В данной работе предлагается собрать данные с динамиками решений задач и разработать алгоритм генерации подсказок для языка Python с применением пространства решений. Это позволит обойти пе-

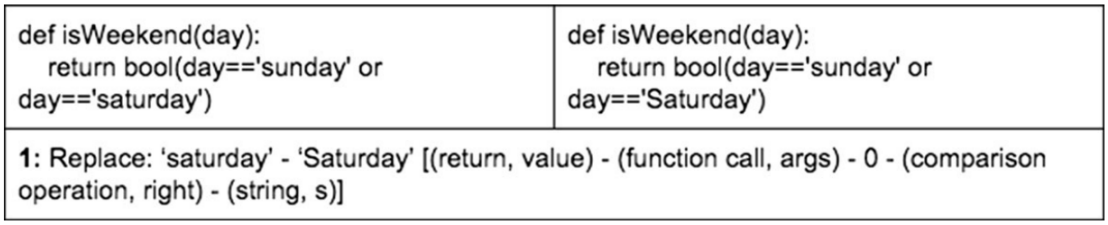


Рис. 8: Пример подсказки алгоритма [18]. Слева представлено решение с ошибкой, справа — правильное решение. Внизу представлена подсказка, помогающая исправить неправильное решение.

ребор булеана множества изменений, как в статье [18], и расширить алгоритм генерации подсказок для более сложного класса задач.

2. Сбор данных

Разработка нового алгоритма генерации подсказок требует данных с промежуточными решениями задач. На данный момент не существует датасетов с такими данными, поэтому было необходимо его получить. Для этого были выбраны 5 задач различного уровня сложности, которые предлагалось решить студентам и для которых впоследствии будут сгенерированы подсказки. Условия двух задач, самой простой и самой сложной, представлены на Рис. 9. Условия остальных задач можно найти в приложении А. Задачи можно решить, имея самые базовые навыки программирования, включающие знание о циклах и условных операторах, что делает задачи доступными даже начинающим программистам. Однако, даже самая простая задача превышает по сложности задачи, используемые в предыдущем подходе [18]. Более того, выбранные задачи примечательны большой вариативностью решений, что уже заметно по двум прикрепленным для каждой задачи решениям в приложении А. Это делает автоматическую генерацию подсказок особенно сложной и интересной задачей.

Название: Пирожки
Описание: Пирожок в столовой стоит *a* рублей и *b* копеек. Определите, сколько рублей и копеек надо заплатить за *n* пирожков.

Название: Скобки
Описание: В строке расставить между буквами открывающиеся и закрывающиеся скобки так, чтобы до середины шли открывающиеся, а после - закрывающиеся.
В случае нечётной длины: example → e(x(a(m)p)l)e
В случае чётной длины: card → c(ar)d, но не c(a()r)d

Рис. 9: Условия самой простой задачи “Пирожки” и самой сложной — “Скобки”.

2.1. Методика сбора данных

Основными участниками сбора данных являются школьники и студенты, поэтому важно было сделать процесс сбора данных максимально простым, удобным и понятным. В частности, важно, чтобы школьники решали программы в привычной им обстановке, пользуясь знакомыми инструментами, ведь иначе результаты могут не отражать реальный

процесс решения задач. Для облегчения работы с инструментом сбора данных необходим удобный пользовательский интерфейс, который поможет детям правильно использовать расширение.

С другой стороны, чем легче происходит процесс сбора данных для организаторов, тем больше данных они смогут собрать. Возможность дистанционного сбора данных позволяет сильно расширить аудиторию участников, так как не требует физического присутствия организаторов. В то же время, дистанционная отправка данных не должна ложиться на плечи участников, так как это ненадежно. Оптимальный вариант — использовать сервер, к которому будет подключаться инструмент сбора данных, получать условия задач в начале и отсылать данные в конце. Таким образом, все, что требуется от пользователя — это иметь доступ к интернету и решать задачи, ведь все остальные необходимые действия автоматизированы.

Основная цель сбора данных — собрать динамики решения задач. Эти данные можно получить из изменений кода в файлах, с которыми работает участник, подвергнув их обработке. Помимо динамик решения задач было решено собирать побочную информацию, которая может влиять на решения. Например, в зависимости от возраста и опыта программирования, участники могут создавать разные решения одной и той же задачи, поэтому каждому участнику надо предоставить анкету для заполнения этой информации. Удобно, если эта анкета будет предоставляться пользовательским интерфейсом инструмента, это позволит контролировать ее заполнение участником.

2.2. Разработка инструмента сбора данных

Расширение (plugin) для сред программирования позволяет решить все поставленные задачи: сбор данных с изменениями кода в файлах, привычная обстановка для решения задач, удобный пользовательский интерфейс, простая настройка и подключение к серверу. Для пользователя таким образом процесс сбора данных не мешает нормальной работе (Рис. 10), а для организаторов становится возможным удаленный

сбор данных.

В данной работе необходимы данные с промежуточными решениями на языке Python, однако разработанное расширение можно использовать с любой IDE, построенной на платформе IntelliJ [10], позволяя собирать данные для многих языков программирования. Расширение разработано на языке программирования Kotlin, его исходный код доступен в открытом доступе на GitHub[1].

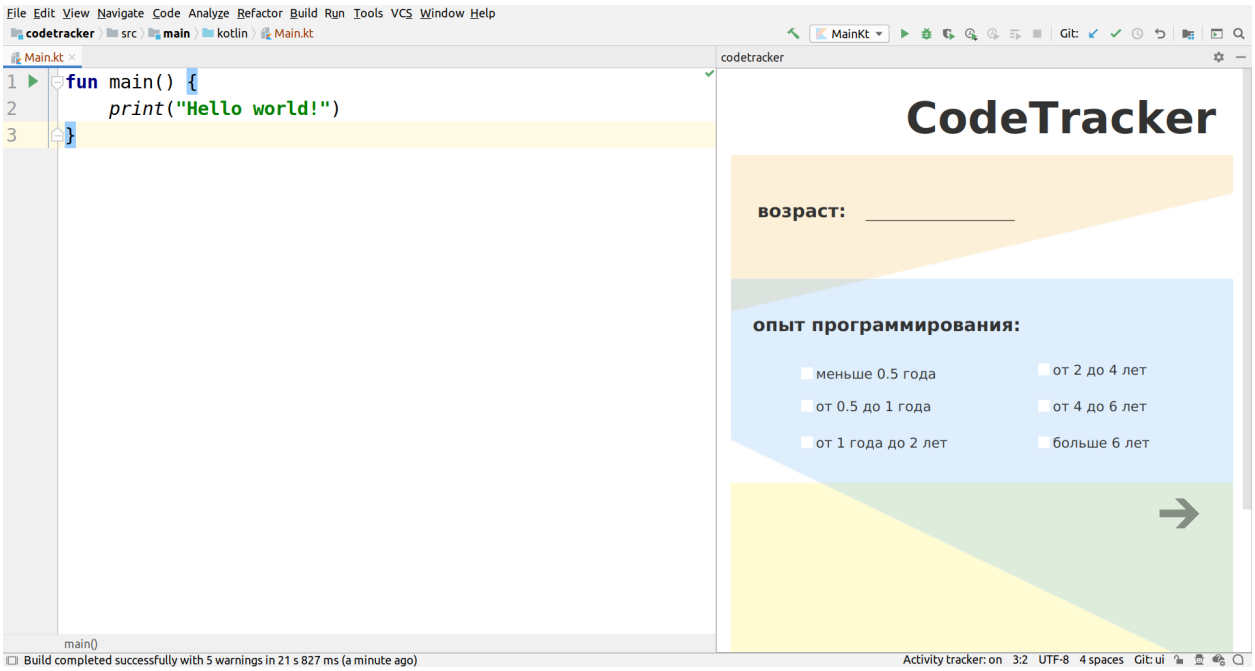


Рис. 10: Работа с расширением в среде программирования.

Архитектура расширения состоит из двух компонентов: модуля логирования решений задач и пользовательского интерфейса, которые взаимодействуют с IntelliJ Platform SDK (Рис. 11). Это позволяет запускать расширение вместе с запуском среды программирования, следить за изменениями в открытых документах и записывать таким образом динамики решений, встраивать пользовательский интерфейс в интерфейс среды.

Наблюдение за изменениями в открытых документах выполняет класс `PluginDocumentListener`, наследующийся от интерфейса `DocumentListener`. Он передает все изменения классу `DocumentLogger`, чья задача — записывать эти изменения в csv-файл. При закрытии последнего окна среды

[1] `https://github.com/JetBrains-Research/codetracker`

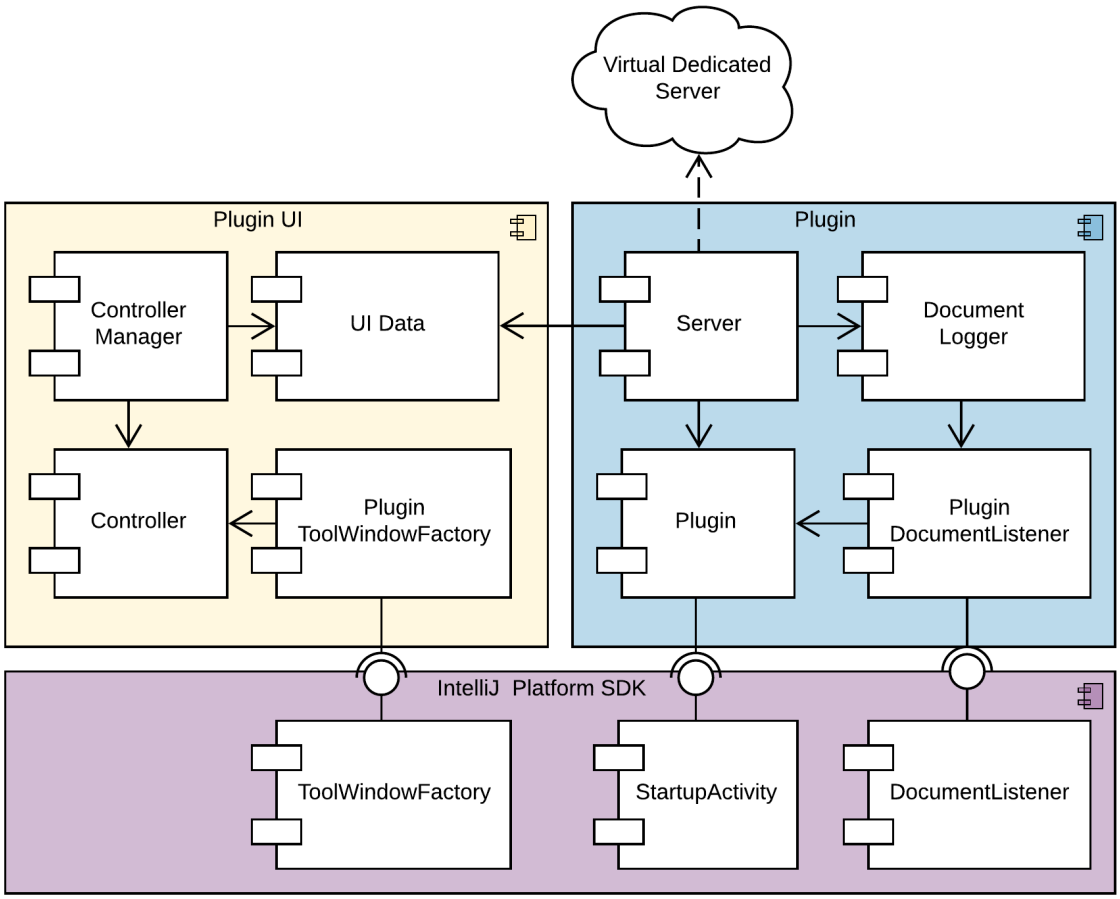


Рис. 11: Диаграмма компонентов разработанного расширения.

программирования или при удалении расширения эти файлы отправляются на сервер при помощи класса `Server`. Через него же инициализируется пользовательский интерфейс: в класс `UiData` передаются данные об условиях задачи, которые должны быть показаны пользователю. За встраивание пользовательского интерфейса в среду отвечает класс `PluginToolWindowFactory`, инициализирующий контроллер интерфейса и все его компоненты при помощи `UiData`. Класс `ControllerManager` обновляет интерйфейс расширения во всех открытых окнах среды в соответствии с изменениями в одном из них.

Сам пользовательский интерфейс расширения представляет собой три страницы, между которыми переключается пользователь: страница с анкетой, страница с выбором задачи и страница с условием выбранной задачи (Рис. 12). Переход с первой страницы на последующие невозможен до тех пор, пока не будут корректно заполнены все данные, что защищает от их случайного пропуска. Страница с условием зада-

чи содержит также примеры правильных входных/выходных данных, позволяющие пользователю копировать их и проверять правильность решения, не переключаясь на другие окна.

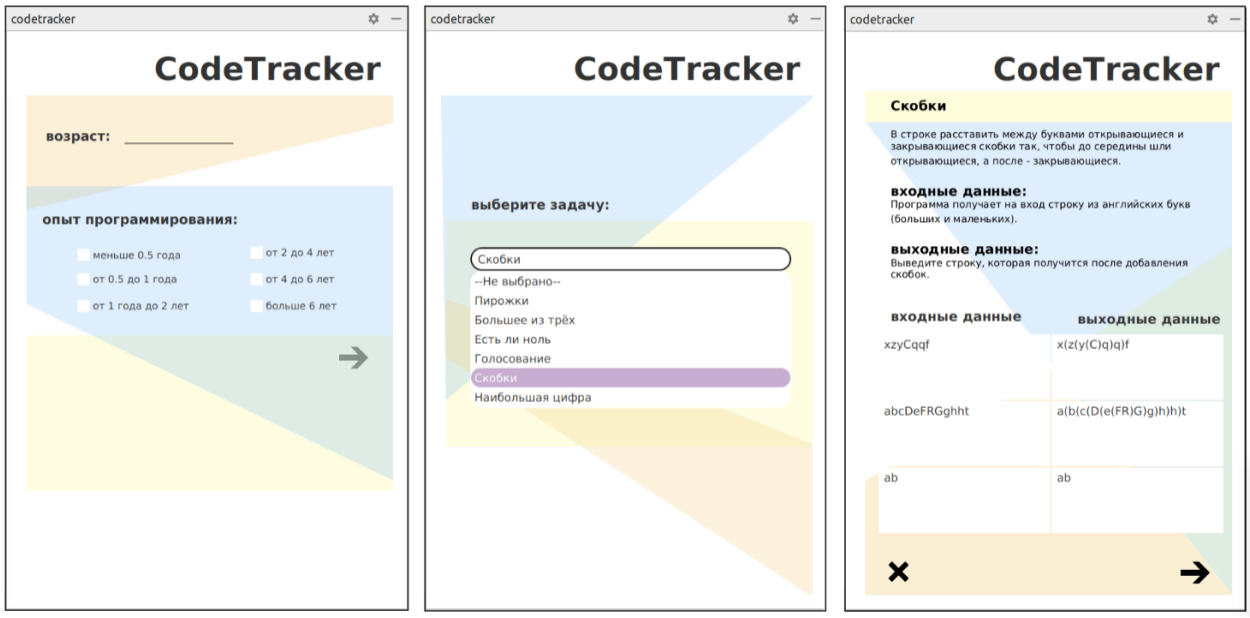


Рис. 12: Пользовательский интерфейс разработанного расширения для сбора данных.

2.3. Собранные данные

Важно было собрать данные о том, как решают задачи именно школьники и студенты, так как более опытные программисты могут создавать слишком сложные решения, не подходящие в качестве подсказок новичкам. Для этого были собраны данные у школьников лицея “Физико-техническая школа”, студентов курса “Введение в Android”, студентов Computer Science Center и студентов СПбГУ. Процесс сбора данных был анонимный, все участники дали явное согласие на сбор данных в рамках данного эксперимента. Всего участие приняли более 120 человек, средний возраст участника равен 16 годам, средний опыт — от 1 до 2 лет программирования. Участники решали задачи на 4-х языках программирования: Java, Kotlin, C++ и Python. Несмотря на то, что в данной работе используются данные только для языка Python, остальные данные представляют ценность в области анализа процессов

решения задач, поэтому было решено записывать и их тоже (Рис. 13).

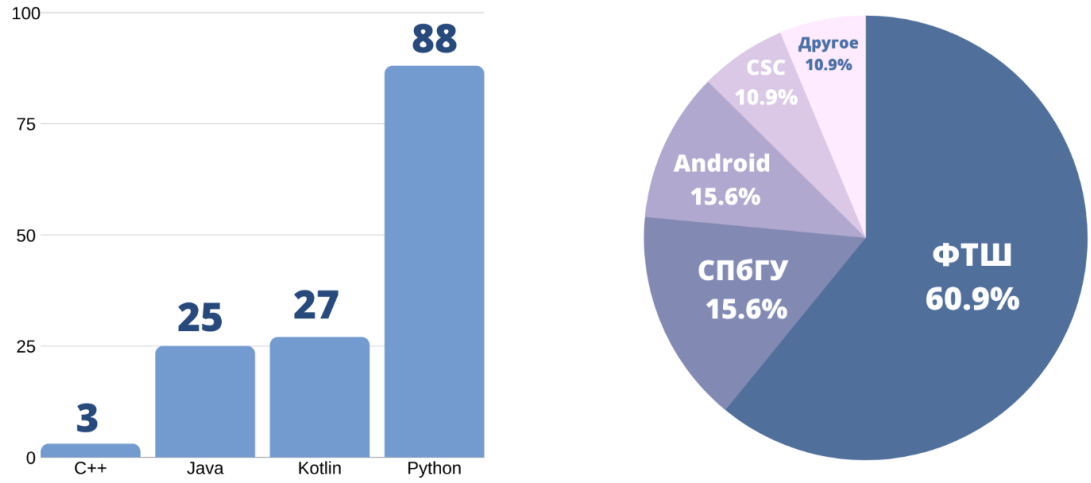


Рис. 13: Участники сбора данных в соответствии с выбранным для решения задач языком программирования (слева) и местом проведения сбора данных (справа).

Для языка Python было собрано более 150 динамик решения разных задач, приводящих к финальному решению. Среди них 43 динамики принадлежат задаче “Пирожки”, а 21 — задаче “Скобки” (Рис. 9).

2.4. Обработка данных

Необработанные данные выглядят как csv-файлы, в каждом из которых записаны все изменения, происходящие в конкретном файле во время работы с расширением. Каждое изменение записывается в виде фрагмента кода, который стал содержать файл после изменения. Таким образом, данные представляют собой последовательный набор фрагментов кода, которые содержал изменяемый файл. Вместе с изменениями записывается имя изменяемого файла, анкета пользователя и выбранная в данный момент задача. Однако, если студент решает все задачи по очереди в одном и том же файле, стирая предыдущее решение при переходе к следующей задаче, то все данные запишутся в один csv-файл. В такой ситуации необходимо разделять один файл с данными на данные по каждой задаче (Рис. 14). Сложность состоит в том, что

не всегда выбранная задача соответствует решаемой в данной момент: некоторые пользователи забывали выбрать новую задачу в пользовательском интерфейсе расширения или выбирали задачу слишком рано, продолжая исправлять решение предыдущей задачи. Кроме того, в качестве промежуточных решений было решено использовать только синтаксически корректный код, поэтому необходимо было отфильтровать некорректные промежуточные решения. Последним требованием было выявление финальных решений. Исходный код алгоритмов обработки данных можно найти в репозитории проекта на GitHub[2].

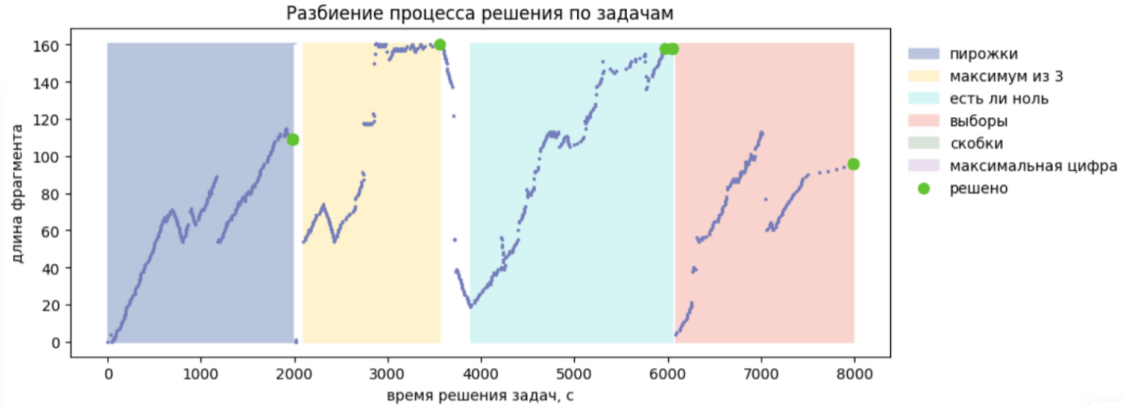


Рис. 14: Разбиение находящихся в одном файле решений в соответствии с решаемыми задачами. Цвет заднего фона соответствует определенной задаче, при этом фрагменты белого цвета обрезаются, так как содержат ненужные данные.

2.4.1. Определение языка программирования

Так как собранные данные включали в себя решения не только на языке Python, но и на других языках, необходимо было определять язык решения. Это можно определить по расширению изменяемого файла, имя которого записывается вместе с изменениями.

2.4.2. Проверка синтаксической корректности фрагментов

Для проверки синтаксической корректности использовался инструмент статической проверки типов MyPy [16], который также позволя-

[2] https://github.com/JetBrains-Research/codetracker-data

ет обнаружить ошибки форматирования, наименования переменных, неправильных импортов и множество других. В отличии от интерпретатора Python, MyPy проверяет корректность недостижимого кода, но упускает несколько важных ошибок, например, деление на ноль. Поэтому для проверки используется MyPy совместно с интерпретатором Python.

2.4.3. Запуск тестов на корректных фрагментах кода

Запуск тестов, проверяющих правильность вывода для данного ввода, позволяет определить, что за задачу решал студент и решил ли он ее. Для каждой задачи были составлены 8 тестов, которые были запущены на каждом из фрагментов. Если на фрагменте проходили все 8 тестов для какой-то задачи, то считалось, что эта задача решена корректно и фрагмент является финальным решением. Если было найдено финальное решение, то все фрагменты до него считались промежуточными решениями этой задачи, что позволило определить, какую задачу решал студент в каждый момент времени. Было выбрано именно такое количество тестов, так как с одной стороны, они учитывали все возможные типы ввода-вывода для каждой задачи, а с другой стороны занимали не очень много времени на проверку. Кроме того, для ускорения процесса запуска тестов проверялись только уникальные фрагменты, содержащие строки вывода (иначе вывод нельзя будет сравнить с правильным) и ограниченные снизу по размеру (нет смысла проверять маленькие фрагменты). С этой же целью тестирование фрагмента прекращалось после первого непройденного теста.

2.4.4. Фильтрация промежуточных решений

В качестве динамик использовались только те динамики, которые содержали синтаксически корректные решения и приводили к финальному решению. Определить их помогали предыдущие этапы обработки данных. Однако, не все синтаксически корректные фрагменты подходят для промежуточных решений. Например, корректными считаются

следующие фрагменты, которые не имеют никакого эффекта:

```
1  print
2
```

```
1  int
2
```

```
1  6
2
```

Для нахождения и фильтрации таких фрагментов был использован статический анализатор кода Pylint [17]. Кроме того, было решено исключить последовательные промежуточные решения с изменениями в одной и той же строке, чтобы динамики решений состояли только из решений с дописанными строками.

Из динамик также были удалены циклы, так как они являются признаком того, что пользователь написал неправильное решение и решил вернуться к своему предыдущему решению. Такие петли не должны быть в динамиках, иначе они могут предлагаться в качестве подсказки. Для этого для двух совпадающих aAST в динамике удалялись все деревья между ними. При этом среди cAST все еще могли оставаться циклы, так как разным aAST может соответствовать одно и то же cAST.

3. Алгоритм генерации подсказок

3.1. Описание алгоритма генерации подсказок

Основной идеей алгоритма является использование пространства решений для языка Python, что позволит генерировать подсказки для более сложных задач по сравнению с подходом в [18] благодаря исключению перебора булеана всех изменений между деревьями.

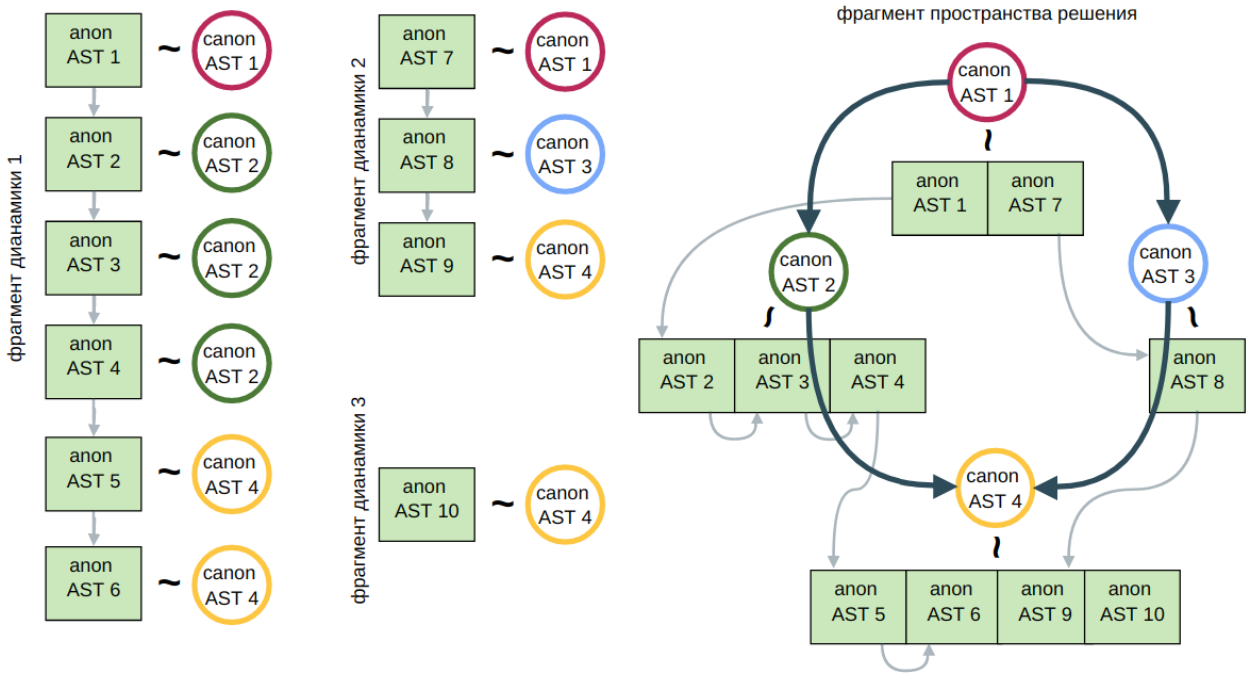


Рис. 15: Структура фрагмента пространства решений, полученного объединением фрагментов динамик 1, 2 и 3.

Пространство решений строится из динамик решения с промежуточными aAST посредством объединения решений с одинаковыми cAST. Так как одному cAST могут соответствовать разные aAST, то вершины в пространстве решений состоят из одного cAST и набора aAST, соответствующих ему. Также возможна ситуация, когда последовательные промежуточные aAST соответствуют одному cAST, поэтому переход между ними не будет учтен в пространстве решений. Чтобы учитывать этот переход, у каждого aAST хранится переход на следующее aAST (Рис. 15). Для каждого aAST также хранится список пользователей (вместе с опытом и возрастом), динамики которых объединились

в этой вершине.

Такая структура диктуется алгоритмом каноникализации: некоторые трансформации настолько меняют aAST, что в дополнение к cAST необходимо хранить aAST. Более того, были обнаружены ошибки в работе каноникализации из статьи [18], которые приводят к удалению некоторых вершин aAST. В то же время, этот алгоритм выполняет свою главную роль унификации решений, поэтому было решено все равно его использовать для сжатия пространства решений. Однако, в данной работе использовался другой способ подсчета изменений между AST, нежели в [18], так как он не всегда корректно работал на промежуточных решениях. В данной работе использовалась библиотека GumTree [9], позволяющая находить изменения между AST более точно.

Блок-схема предлагаемого алгоритма генерации подсказок с использованием пространства решений представлена на рис. 16. Сам алгоритм выглядит следующим образом:

1. Для текущего решения ищется вершина с таким же aAST. Вначале сравниваются все cAST в пространстве решений с таким же количеством вершин, что и у текущего cAST, и в случае совпадения сравниваются aAST, соответствующие найденному cAST. Если такое же aAST найдено, то рассматриваются все пути в графе из этого дерева и в качестве подсказки выдается “наилучшее” следующее aAST.

2. Если такого же дерева не существует, в пространстве решений ищется наиболее подходящее промежуточное aAST. Для ускорения работы алгоритма область поиска ограничивается N_c cAST и N_a соответствующим им aAST с похожим на текущее количество вершин. Кроме того, выбирается N_s aAST с структурой текущего решения. Чтобы учесть структуру, сравнивается количество условных операторов и операторов цикла. Затем среди этих деревьев выбирается “наилучшее” дерево.

3. После этого определяется, насколько найденное ближайшее решение близко к финальным решениям по количеству вершин, содер-

жащихся в их деревьях. Если решение далеко, то оно выдается в качестве подсказки.

4. Если решение близко к финальным, то среди них находится “наилучшее” для текущего решения пользователя.

5. Последним шагом определяется, стоит ли идти напрямую к найденному финальному решению (и тогда оно выдается в качестве подсказки) или же идти через пространство решений (и тогда выдается найденное ранее ближайшее решение). Для этого оценивается, какой процент пути к финальному решению уже пройден и насколько далеко от найденного ближайшего решения находится текущее решение.

Чтобы выбрать “наилучшее” для текущего решения среди нескольких решений, называемых кандидатами, оценивается несколько параметров:

- количество изменений между кандидатом и текущим решением;
- соответствие опыта и возраста пользователей, посещавших решение-кандидат, с опытом и возрастом текущего студента;
- “откатывание” текущего решения назад: кандидаты, которые находятся ближе к текущему решению, но отдаляют его от финального решения, штрафуются;
- “популярность” кандидата, определяемая количеством посещений пользователей;
- количество пройденных тестов: кандидаты, уменьшающие количество пройденных тестов, штрафуются;
- структура текущего дерева и дерева-кандидата, кандидаты с сильно различающейся структурой штрафуются.

Подсказкой, таким образом, является предложенное алгоритмом aAST. К текущему фрагменту можно применить изменения с учетом трансформаций каноникализации, взятых из статьи [18]. Тогда полученное

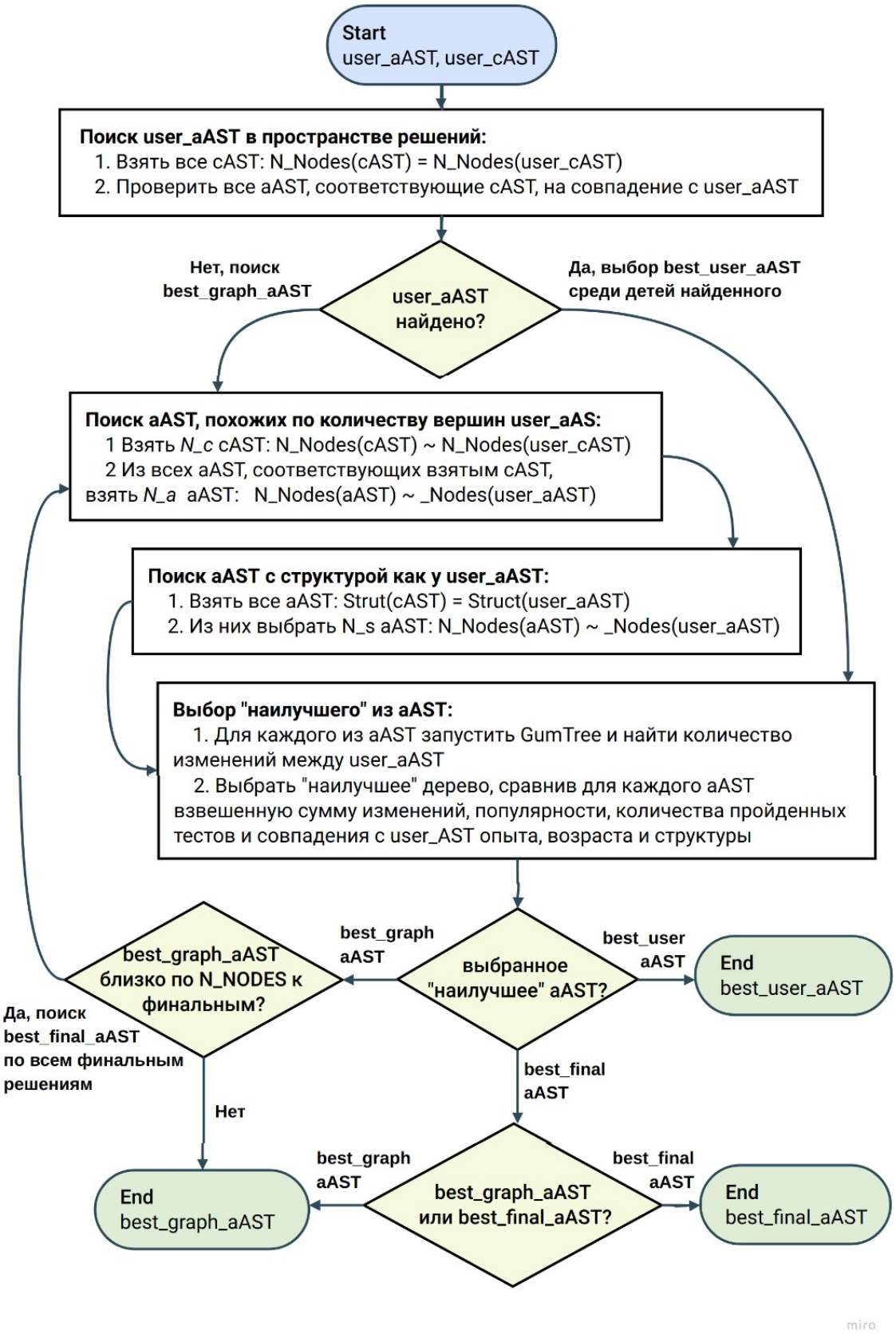


Рис. 16: Блок схема алгоритма генерации подсказок.

дерево будет структурно похоже на предложенное aAST, но при этом будет иметь имена и другие особенности, присущие текущему фрагменту. Это позволит использовать полученное дерево в качестве подсказки.

3.2. Оценка сложности алгоритма

Одним из преимуществ использования пространства решений является отсутствие необходимости перебирать булеан набора изменений для получения промежуточных решений, как это было предожено в [18]. Это позволяет ускорить время работы генерации подсказок. Рассмотрим при этом алгоритмическую сложность предложенного алгоритма. Пусть пространство решений предподсчитано, и для генерации подсказок используется десериализированное пространство решений. В нем предлагается для cAST хранить хэш-таблицы по количеству вершин в дереве: в одной таблице для промежуточных cAST, в другой — для финальных. Также в каждом aAST будем хранить его количество вершин. Будем хранить также хэш-таблицу по структурам aAST.

Определим следующие переменные:
N_{cAST} — количество cAST в пространстве решений,
N_{aAST} — количество aAST, соответствующих одному cAST,
N_{child} — количестве следующих aAST у одного aAST,
N_{nodes} — количество вершин в cAST или aAST,
N_{nodes_dict} — количество cAST для конкретного номера вершин, находящихся в хэш-таблице,
N_{struct_dict} — количество aAST для конкретной структуры, находящихся в хэш-таблице.
Также в алгоритме используются следующие константы:
N_c, N_a и N_s — количество cAST, aAST и структурно похожих aAST соответственно, выбираемых для поиска наилучшей вершины в пространстве решений.

Будем считать, что проверка на равенство двух AST происходит за $O(N_{nodes})$, так как необходимо сравнить каждую вершину в первом дереве с вершиной во втором, а сложность работы GumTree обозначим за

$O_{GT}(N_{nodes})$, предполагая, что сравнение двух деревьев зависит только от количества вершин в них.

Тогда сложность первого шага алгоритма — поиск соответствующего фрагменту aAST в пространстве решений — составляет

$$N_{nodes\_dict} \cdot N_{nodes} + N_{aAST} \cdot N_{nodes},$$

так как заключается в проверке на равенство cAST из хэш-таблицы с последующей проверкой всех aAST, соответствующих найденному cAST.

Если такое aAST не найдено, то выполняется второй шаг — поиск наиболее подходящего aAST в пространстве решений. Сложность поиска aAST, схожих по количеству вершин, составляет

$$\frac{N_c}{N_{nodes\_dict}} + N_c \cdot N_a + \frac{N_a}{N_{nodes\_dict}},$$

так как сначала из хэш-таблицы выбирается N_c cAST, после этого аналогичная хэш-таблица создается для соответствующих им aAST, из которой уже выбирается N_a aAST. Поиск деревьев, схожих по структуре, занимает

$$\frac{N_s}{N_{struct\_dict}}.$$

Проверка, насколько выбранное в пространстве решений aAST близко к финальным по количеству вершин, проверяется за константу. Четвертый шаг, поиск наиболее подходящего финального aAST, аналогичен по сложности второму шагу. Итоговое решение о выдаче алгоритмом промежуточного или финального дерева включает в себя три вызова GumTree и выполняется за $O_{GT}(N_{nodes})$.

Выбор “наилучшего” aAST из N кандидатов включает N вызовов GumTree.

Таким образом, общая сложность алгоритма зависит от того, на ка-

ком шаге нашлась подсказка и максимально составляет

$$(N_{nodes\_dict} + N_{aAST}) \cdot N_{nodes} + \frac{N_c + N_a + N_s}{N_{nodes\_dict}} + N_c \cdot N_a$$
$$+(N_a + N_s) \cdot O_{GT}(N_{nodes}) = O((N_{nodes\_dict} + N_{aAST}) \cdot N_{nodes}). \quad (1)$$

Интересно посмотреть, как изменится время работы алгоритма при увеличении решений в пространстве. Будем считать, что N_{nodes} — количество вершин в AST — не меняется при увеличении количество решений, так как все решения одной задачи зачастую содержат примерно одинаковое количество вершин. Несмотря на то, что сложность алгоритма не зависит напрямую от количества решений в пространстве, при разрастании пространства решений в k раз, во столько же меняется количество AST с конкретным количеством вершин в хэш-таблице и количество aAST, соответствующих одному cAST.

3.3. Примеры генерации подсказок

На Рис. 17 приведены примеры генерации подсказок для задачи “Пирожки” (условие в приложении А).

Автор первого фрагмента решает задачу, проверяя, не переполнилось ли количество копеек в стоимости n пирожков. Если переполнилось, то надо перевести соответствующее количество копеек в рубли. Однако, в текущем фрагменте есть несколько ошибок:

- переменная $g5$ объявляется внутри *if*, хотя используется потом и снаружи. Это может привести к ошибке исполнения программы в случае, если переменная не будет инициализирована;
- для нахождения остатка от деления лучше использовать операцию %, а не вычислять результат вручную.

Предложенное алгоритмом aAST исправляет обе ошибки текущего aAST.

Автор второго фрагмента не реализовал ввод данных, который необходим в этой задаче. Предложенное алгоритмом aAST предлагает заменить инициализацию переменных на ввод данных с клавиатуры. При

1.

aAST текущего фрагмента:

```
g4 = int(input())
g3 = int(input())
g2 = int(input())
g0 = (g4 * g2)
g1 = (g3 * g2)
if (g1 >= 100):
    g5 = (g1 // 100)
g1 = (g1 - (g5 * 100))
g0 = (g0 + g5)
print(g0, g1)
```

aAST, предложенное в качестве подсказки:

```
g4 = int(input())
g3 = int(input())
g2 = int(input())
g1 = (g4 * g2)
g0 = (g3 * g2)
if (g0 >= 100):
    g5 = (g0 // 100)
    g1 = (g1 + g5)
    g6 = (g0 % 100)
    g0 = g6
print(g1, g0)
```

2.

aAST текущего фрагмента:

```
a = int(10)
b = int(15)
n = int(2)
z = n*a
x = n*b
print()
```

aAST, предложенное в качестве подсказки :

```
g4 = int(input())
g3 = int(input())
g2 = int(input())
g1 = (g2 * g4)
g0 = (g2 * g3)
```

Рис. 17: Примеры предложенных алгоритмом подсказок в виде aAST для задачи “Пирожки”, сгенерированных для данных текущих фрагментов, представленных в виде их aAST.

этом, предложенное aAST предлагает удалить строку с пустым выводом, так как это действие неэффективно. Возможно, более логичным предложением было добавить в вывод переменные, которые необходимо будет вывести далее, а не удалять строку с выводом. Однако, в таком случае подсказка будет иметь слишком большой “шаг”, предлагая пользователю слишком большой переход. Поэтому данная подсказка считается хорошей.

3.4. Варианты использования пространства решений

Разработанный алгоритм включает построение пространства решений, которое помимо использования для генерации подсказок само по себе довольно интересно для изучения. Например, рассмотрим про-

странство решений, соответствующее задаче “Скобки”, представленное на Рис. 18.

Вершины покрашены в зависимости от количества пройденных тестов до первого некорректно завершённого теста. Красным отмечены вершины, которые не прошли ни одного теста, зелёным — которые прошли все тесты. Все финальные (зелёные) вершины соединены в одну пустую вершину для ускорения работы алгоритма при генерации подсказки. При помощи пространства решения можно оценить, как увеличивается количество пройденных тестов при приближении к финальному решению, и заметить некоторые интересные паттерны поведения участников при решении задач. Например, выделяется динамика красного цвета, но с зелеными решениями в середине. Если посмотреть на исходный код этих фрагментов, то будет ясно, что автор сначала решил задачу (и вершины окрасились в зелёный цвет), а потом решил улучшить архитектуру решения и вынес часть кода в отдельную функцию. Из-за этого рефакторинга динамика на какое-то время опять окрасилась в красный цвет. Также интересно посмотреть на фрагменты, которые пользуются особой популярностью у студентов. Например, в вершине 32, из которой расходится множество путей, находится фрагмент, соответствующий считыванию строки с клавиатуры. А вершина 37, находящаяся в середине графа над множеством вершин, выходящих из нее, содержит считывание строки и вычисление ее длины.

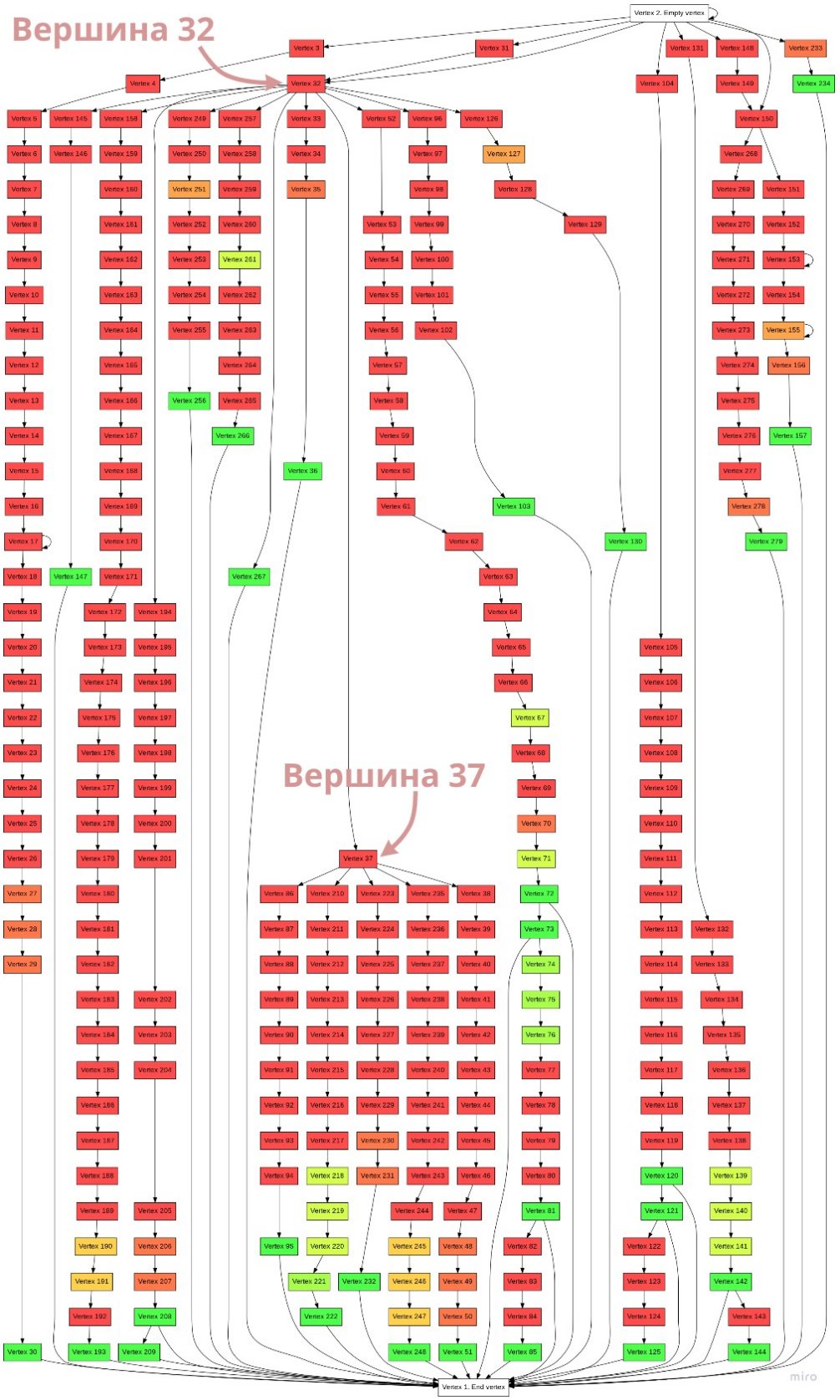


Рис. 18: Пространство решений, построенное для задачи “Скобки”.

4. Тестирование

Для проведения тестирования алгоритма рекомендации подсказок из 5 задач, которые были предложены студентам для решения, были выбраны 2 задачи: самая простая (“Пирожки”) и самая сложная (“Скобки”). Их условия представлены на Рис. 9. Для первой задачи было собрано 43 решения, для второй — 22.

4.1. Критерии оценки подсказок

Подсказки, сгенерированные алгоритмом, включают в себя предложенное им aAST и AST, полученное после применения изменений к текущему фрагменту, взятых из статьи [18]. Ожидалось, что применение изменений работает корректно, поэтому такой код можно выдавать в качестве подсказки. Однако, иногда применение изменений работало неправильно, хотя предложенное aAST выглядело подходящим. Поэтому были разработаны критерии оценки подсказок, рассматривающие в качестве подсказки предложенное aAST и отдельно оценивающие корректность применения изменений:

- Если фрагмент **не является** частью решения текущей задачи (такое могло быть, если участник сбора данных неверно выбрал задачу):
  - **Длина подсказки**:
    * пустая или начало решения;
    * конец решения или финальное решения.
- Если фрагмент **является** частью решения текущей задачи:
  - **Совпадение структуры** (блоки *if*, *for* и *while*) у фрагмента и подсказки:
    * структуры похожи, содержатся одни и те же блоки с незначительными для решения различия или в подсказке появляется новый блок, характерный для решения;

    * структуры не похожи, содержатся разные блоки или блоки в неправильном порядке, который меняет логику программы.

- **Приближение к финальному решению**:
    * приближает фрагмент к финальному решению, то есть либо исправляет существующие ошибки, либо предлагает дальнейшие шаги к решению;
    * отдаляет фрагмент от финального решения, то есть либо удаляет существующий правильный код, либо советует изменения, не характерные для финальных решений;
    * не изменяет фрагмент;
    * и не приближает, и не отдаляет (например, исправляет существующую ошибку, но предлагает новую).
- **Шаг подсказки**, то есть насколько большие изменения предлагает подсказка:
    * слишком маленький шаг для хорошей подсказки;
    * подходящей шаг;
    * слишком большой шаг для хорошей подсказки.
- **Логичность подсказки**:
    * хорошая подсказка, логичная, могла бы быть предложена экспертом;
    * подходящая подсказка, логичная, но с небольшими недочетами (например, можно было бы предложить чуть более логичную подсказку, чем эту), все еще может быть предложена;
    * плохая подсказка, эксперт бы не предложил.
- **Корректность применения изменений** к предложенному aAST:
    * изменения применены корректно;
    * изменения применены некорректно.

Среди собранных данных были как и успешные попытки решения задач, представляющие динамики решений с финальным решением на конце, так и неуспешные, авторам которых так и не удалось решить задачу. Эти данные было решено использовать для тестирования алгоритма. Для каждой задачи было выбрано случайным образом 50 фрагментов и сгенерированы для них подсказки. Так как некоторые критерии можно трактовать субъективно, сгенерированные подсказки были оценены вручную двумя независимыми оценщиками. В случае, если мнения расходились, подсказка или обсуждалась до тех пор, пока мнения оценщиков не совпадали, или пропускалась.

4.2. Результаты тестирования

Используя приведённые выше критерии, можно составить различные описания подсказок и найти вероятность их появления. Например, найти вероятность появления подсказок с конкретным шагом и логичностью. Это помогает оценить, насколько релевантные подсказки предлагались пользователям. Были составлены описания подсказок с разным уровнем релевантности. Например, подсказки, которые не только “хорошие” или “подходящие”, но еще и с похожей структурой, считаются более релевантными. Эти группы приведены в соответствии с повышением их релевантности:

- R1:
  - **логичность подсказки**: хорошая или подходящая.
- R2:
  - **логичность подсказки**: хорошая или подходящая;
  - **приближение к решению**: приближает фрагмент к решению.
- R3:
  - **логичность подсказки**: хорошая или подходящая;

– **приближение к решению**: приближает фрагмент к решению;

– **совпадение структуры**: структуры похожи.

- R4:

  – **логичность подсказки**: хорошая или подходящая;

  – **приближение к решению**: приближает фрагмент к решению;

  – **совпадение структуры**: структуры похожи;

  – **шаг подсказки**: шаг подходящей величины.

Вероятность появления подсказок из каждой группы была оценена для задач “Пирожки” и “Скобки”, результаты приведены на Рис. 19.

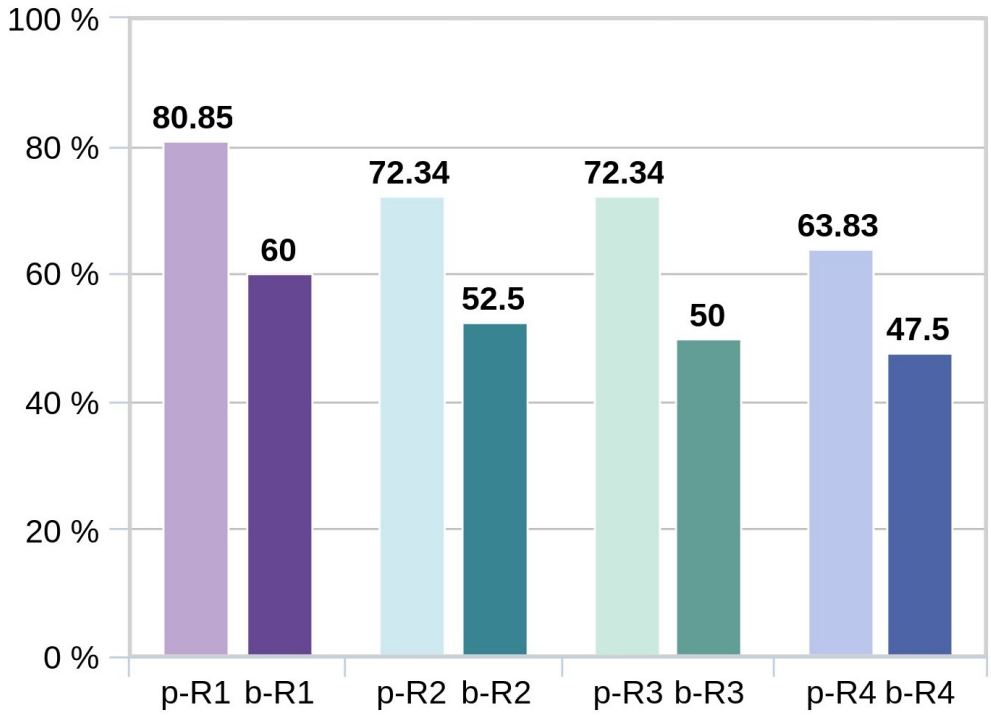


Рис. 19: Вероятность появления подсказок различной степени релевантности от R1 до R4 для задачи “Пирожки” (обозначается как p-Ri) и “Скобки” (обозначается как b-Ri).

Из рисунка видно, что для каждой группы результат на задаче “Пирожки” лучше, чем для задачи “Скобки”. Это объясняется тем, что вторая задача имеет большую вариативность решений, чем первая, и к

тому же сложнее. Из-за этого ее решило довольно мало участников эксперимента, так что еще одну роль в различии результатов играет разница в количестве собранных данных.

При этом алгоритм с достаточно большой вероятностью генерирует релевантные подсказки: 80.85% для первой задачи и 60% — для второй. С увеличением уровня релевантности подсказок эти значения ухудшаются, однако, не очень сильно: от R1 до R4 вероятность генерации подсказок упала на 21% и 20% соответственно. Это говорит о том, что большинство подсказок уровня R1 удовлетворяет критериям следующих уровней.

В процессе оценки предложенного алгоритмом aAST также оценивалась корректность применения изменений, которая составила 40.4% для первой задачи и 70.0% для второй задачи. Поэтому алгоритм применения изменений в соответствии с трансформациями требует доработки для использования его в качестве преобразования aAST в подсказку.

Отдельное тестирование было проведено для измерения времени работы алгоритма с использованием стандартной библиотеки timeit. Для каждой задачи было оценено среднее время генерации подсказки для каждого из 50 фрагментов. Усредненное по всем фрагментам время составило 16.8 секунд для задачи “Пирожки” и 17.3 — для задачи “Скобки”.

5. Результаты

В ходе данной работы были получены следующие результаты:

1. Проведен обзор существующих решений генерации подсказок: для блокового языка программирования, для логического языка программирования и языка Python. Первые два подхода не распространимы на языки с большей вариативностью, как Python, а последний подход не работает на больших задачах и не использует динамики решений.

2. Создан инструмент для удаленного сбора данных в виде расширения для сред программирования на основе платформы IntelliJ. Реализация выполнена на языке Kotlin.

3. Собран набор данных, включающий динамики решений задач по программированию для 6 задач на языках программирования Python, Java, Kotlin, C++. В него входят данные 120 человек разного возраста и опыта. Для данного проекта использовались только данные по решениям на языке Python.

4. Реализованы алгоритмы обработки данных, построения пространства решений (язык Python). Они включают в себя:

   - определение языка программирования;
   - фильтрацию некорректных фрагментов кода;
   - тестирование решений на правильность;
   - разделение всех фрагментов кода по задачам;
   - фильтрацию промежуточных решений.

5. Разработан алгоритм генерации подсказок для языка Python с использованием динамик решения и пространства решений. При генерации подсказки учитываются следующие факторы:

   - существует ли в пространстве решений текущее решение пользователя;

- насколько близко студент находится к финальным решениям и решениям в графе;
- количество предыдущих студентов, написавших найденное решение;
- опыт и возраст студента;
- увеличение количества пройденных тестов.

6. Проведено тестирование алгоритма генерации подсказок. Для него были сгенерированы пары код-подсказка, релевантность которых оценили эксперты по различным критериям. Оценка релевантности генерируемых подсказок для двух задач различного уровня сложности показала 80.85% и 60% соответственно, Это является хорошим показателем, учитывая, что для второй задачи, самой сложной, было собрано почти в 2 раза меньше решений, чем для первой задачи. Среднее время генерации подсказки составило 16.8 и 17.3 секунды для двух задач соответсвенно.

A. Условия задач для сбора данных

Название: Пирожки
Описание: Пирожок в столовой стоит *a* рублей и *b* копеек. Определите, сколько рублей и копеек надо заплатить за *n* пирожков.

Пример решения:

```
a = int(input())
b = int(input())
n = int(input())
print(a * n + (b * n) // 100, b * n % 100)
```

Пример решения:

```
a = int(input())
b = int(input())
n = int(input())
r = a * n
k = b * n
while k >= 100:
    k -= 100
    r += 1
print(r, k)
```

Название: Есть ли ноль
Описание: Проверьте, есть ли среди данных N чисел нули. Выведите YES, если среди введенных чисел есть хотя бы один ноль, или NO в противном случае.

Пример решения:

```
N = int(input())
arr = input()
l = list(map(int, arr.split(' ')))
out = 'YES' if 0 in l else 'NO'
print(out)
```

Пример решения:

```
N = int(input())
is_zero = False
for _ in range(N):
    a = int(input())
    if a == 0:
        is_zero = True
if is_zero:
    print('YES')
else:
    print('NO')
```

Название: Голосование
Описание: Даны три числа, каждое из которых равно 1 или 0. Определите, что среди них встречается чаще: 0 или 1, и выведите это число.

Пример решения:

```
if (int(input()) + int(input()) + int(input())) > 1:
    print(1)
else:
    print(0)
```

Пример решения:

```
a = int(input())
b = int(input())
c = int(input())

if a == 0:
    if b == 0:
        print(0)
    elif c == 0:
        print(0)
    else:
        print(1)
elif b == 0:
    if c == 0:
        print(0)
    else:
        print(1)
else:
    print(1)
```

Название: Большее из трёх
Описание: Программа получает на вход 3 целых числа, каждое записано в отдельной строке. Выведите одно число — наибольшее из данных трёх чисел.

Пример решения:

```
a = [int(input()), int(input()), int(input())]
print(max(a))
```

Пример решения:

```
a = int(input())
b = int(input())
c = int(input())
if a <= b <= c or b <= a <= c:
    print(c)
elif a <= c <= b or c <= a <= b:
    print(b)
elif b <= c <= a or c <= b <= a:
    print(a)
```

Название: Скобки
Описание: В строке расставить между буквами открывающиеся и закрывающиеся скобки так, чтобы до середины шли открывающиеся, а после - закрывающиеся.
В случае нечётной длины: example → e(x(a(m)p)l)e
В случае чётной длины: card → c(ar)d, но не c(a()r)d

Пример решения:

```
s = input()
l = len(s)
print('('.join(s[:(l + 1) // 2])[:l - l % 2] +
      ')'.join(s[l // 2:]))
```

Пример решения:

```
data = input()
length = len(data)
if length % 2 == 1:
    for i in range(length):
        if i < length // 2:
            print(data[i] + '(', end='')
        elif i != length-1:
            print(data[i] + ')', end='')
        else:
            print(data[i], end = '')
else:
    for i in range(length):
        if i < length / 2 - 1:
            print(data[i] + '(', end='')
        elif (i != length / 2 - 1) and (i != length-1):
            print(data[i] + ')', end='')
        else:
            print(data[i], end = '')
```

Список литературы

[1] AutoStyle: Toward Coding Style Feedback at Scale / Joseph Moghadam, Rohan Choudhury, H.Z. Yin, Armando Fox. — 2015. — 03. — P. 261–266.

[2] Autonomously Generating Hints by Inferring Problem Solving Policies / Chris Piech, Mehran Sahami, Jonathan Huang, Leonidas Guibas // Proceedings of the Second (2015) ACM Conference on Learning @ Scale. — L@S '15. — New York, NY, USA : Association for Computing Machinery, 2015. — P. 195–204. — Access mode: https://doi.org/10.1145/2724660.2724668.

[3] Barnes Tiffany, Stamper John. Toward Automatic Hint Generation for Logic Proof Tutoring Using Historical Student Data // Intelligent Tutoring Systems / Ed. by Beverley P. Woolf, Esma Aïmeur, Roger Nkambou, Susanne Lajoie. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2008. — P. 373–382.

[4] Blikstein Paulo. Using Learning Analytics to Assess Students' Behavior in Open-Ended Programming Tasks // Proceedings of the 1st International Conference on Learning Analytics and Knowledge. — LAK '11. — New York, NY, USA : Association for Computing Machinery, 2011. — P. 110–116. — Access mode: https://doi.org/10.1145/2090116.2090132.

[5] Bulmer Jeff, Pinchbeck Angie, Hui Bowen. Visualizing Code Patterns in Novice Programmers // Proceedings of the 23rd Western Canadian Conference on Computing Education. — WCCCE '18. — New York, NY, USA : Association for Computing Machinery, 2018. — Access mode: https://doi.org/10.1145/3209635.3209652.

[6] EduTools - plugin for IntelliJ IDEs. — Access mode: https://plugins.jetbrains.com/plugin/10081-edutools.

[7] From Objects-First to Design-First with Multimedia and Intelligent

Tutoring / Sally H. Moritz, Fang Wei, Shahida M. Parvez, Glenn D. Blank // Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education. — ITiCSE '05. — New York, NY, USA : Association for Computing Machinery, 2005. — P. 99–103. — Access mode: https://doi.org/10.1145/1067445.1067475.

[8] Gulwani Sumit, Radiček Ivan, Zuleger Florian. Feedback Generation for Performance Problems in Introductory Programming Assignments // Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. — FSE 2014. — New York, NY, USA : Association for Computing Machinery, 2014. — P. 41–51. — Access mode: https://doi.org/10.1145/2635868.2635912.

[9] GumTree, a framework to deal with source code as trees and compute differences between them. — Access mode: https://github.com/GumTreeDiff/gumtree.

[10] IntelliJ-based IDE's. — Access mode: https://www.jetbrains.com/products.html#type=ide.

[11] Jadud Matthew C. A First Look at Novice Compilation Behaviour Using BlueJ // Computer Science Education. — 2005. — Vol. 15, no. 1. — P. 25–40. — Access mode: https://doi.org/10.1080/08993400500056530.

[12] Jadud Matthew C. Methods and Tools for Exploring Novice Compilation Behaviour // Proceedings of the Second International Workshop on Computing Education Research. — ICER '06. — New York, NY, USA : Association for Computing Machinery, 2006. — P. 73–84. — Access mode: https://doi.org/10.1145/1151588.1151600.

[13] Keuning Hieke, Jeuring Johan, Heeren Bastiaan. A Systematic Literature Review of Automated Feedback Generation for Programming Exercises // ACM Trans. Comput. Educ. — 2018. —

Sep. — Vol. 19, no. 1. — Access mode: https://doi.org/10.1145/3231711.

[14] Lahtinen Essi, Ala-Mutka Kirsti, Järvinen Hannu-Matti. A study of the difficulties of novice programmers. — Vol. 37. — 2005. — 09. — P. 14–18.

[15] Mining Developer's Behavior from Web-Based IDE Logs / Pasquale Ardimento, Mario Bernardi, Marta Cimitile, Giuseppe Ruvo. — 2019. — 06. — P. 277–282.

[16] MyPy, an optional static type checker for Python. — Access mode: http://mypy-lang.org/.

[17] Pylint, a Python static code analysis tool. — Access mode: https://www.pylint.org/.

[18] Rivers Kelly, Koedinger Kenneth. Data-Driven Hint Generation in Vast Solution Spaces: a Self-Improving Python Programming Tutor // International Journal of Artificial Intelligence in Education. — 2015. — 10. — Vol. 27.

[19] Scaffolding Students' Learning Using Test My Code / Arto Vihavainen, Thomas Vikberg, Matti Luukkainen, Martin Pärtel // Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education. — ITiCSE '13. — New York, NY, USA : Association for Computing Machinery, 2013. — P. 117–122. — Access mode: https://doi.org/10.1145/2462476.2462501.

[20] Vihavainen Arto, Helminen Juha, Ihantola Petri. How Novices Tackle Their First Lines of Code in an IDE: Analysis of Programming Session Traces // Proceedings of the 14th Koli Calling International Conference on Computing Education Research. — Koli Calling '14. — New York, NY, USA : Association for Computing Machinery, 2014. — P. 109–116. — Access mode: https://doi.org/10.1145/2674683.2674692.

[21] Yusri N., Syed-Mohamad Sharifah, Abdul Rashid Nur'Aini. Recoop:

A collaborative tool to support teaching and learning programming. — 2015. — 09. — Vol. 9. — P. 2703–2710.