УДК 004.42 Дата подачи статьи: 16.11.21, после доработки: 31.01.22
DOI: 10.15827/0236-235X.137.065-074 2022. Т. 35. № 1. С. 065–074

Алгоритм поиска идиом в исходных текстах программ, использующий подсчет поддеревьев

Д.А. Орлов [1], *к.т.н., доцент, orlovdmal@mpei.ru*

[1] *Национальный исследовательский университет «МЭИ», г. Москва, 111250, Россия*

Статья посвящена разработке алгоритма извлечения программных идиом из корпуса исходных текстов программ. Программные идиомы – это фрагменты исходных текстов программ, которые встречаются в исходных текстах различных программ и служат для решения одной типичной задачи. В данной работе программная идиома рассматривается как поддерево абстрактного синтаксического дерева (Abstract Syntax Tree, AST) программы, обеспечивающее максимальное сокращение информации в исходном коде программы при замене всех его вхождений на отдельную синтаксическую конструкцию (например, на вызов функции).

Разработана метрика ценности поддерева в качестве идиомы, оценивающая сокращение количества информации от такой замены. Таким образом, поиск программных идиом сводится к поиску максимума функции ценности поддерева на множестве поддеревьев AST. Чтобы сократить перебор поддеревьев, поиск максимума функции ценности поддерева предлагается осуществлять методом наискорейшего спуска: на каждом шаге в поддерево добавляется узел, обеспечивающий наибольшее увеличение ценности поддерева. Для хранения поддеревьев используется структура, являющаяся обобщением префиксного дерева.

Предложен алгоритм ускоренного извлечения программных идиом. Ускорение достигается за счет повторного использования результатов поиска максимума функции ценности поддерева. Для программной реализации разработанных алгоритмов, а также для исследования выбран язык Python, поскольку он имеет большой корпус исходных текстов и удобные средства построения AST.

С помощью разработанной программной реализации проведен эксперимент по извлечению программных идиом из корпусов исходных текстов программ с открытым исходным кодом на языке Python. Полученные в результате программные идиомы являются осмысленными фрагментами исходных текстов программ. Показано, что применение разработанных алгоритмов к исходному коду одного проекта позволяет выявить варианты рефакторинга исследуемой программы.

Ключевые слова: *программная идиома, анализ программ, рефакторинг, извлечение данных, абстрактное синтаксическое дерево, Python.*

Программные идиомы – это фрагменты исходных текстов программ, которые встречаются в различном ПО и служат для решения одной типичной задачи. Термин «идиоматический код» означает исходный текст программы, написанный в таком виде, который другие опытные разработчики считают естественным [1]. Тем не менее формального определения термина «программная идиома» не существует.

В качестве примеров программных идиом можно привести фрагменты исходных текстов программ на языках C++ и Python:

```
а) with open($args,$keywords) as $id:
     $id.write($args)
б) try:
     import $name
     ...
except ImportError:
     $body
в) for path in (sys.argv[1::] or
sys.stdin.read().splitlines()):
     $body
г) for(auto it = $name.begin();
it != $name.end();){
     if($expr)
         it = $name.erase(it);
     else
         ++it;
}
```

Здесь последовательности, начинающиеся с символа $, означают аргументы идиомы, в исходных текстах программ на их месте могут находиться любые синтаксически допустимые фрагменты, многоточием обозначена любая синтаксически допустимая последовательность операторов.

Фрагменты представляют следующие идиомы: а) – открытие файла, запись данных в него (на языке Python3); б) – импорт модуля и обработка ошибки импорта (на языке Python3); в) – получение аргументов из командной строки и устройства стандартного ввода и выполнения для них некоторых действий (на языке Python3); г) – поиск в контейнере элементов, удовлетворяющих некоторому условию, и удаление их (на языке C++).

Программные идиомы дают дополнительную информацию для изучения языков программирования, так как являются готовыми решениями типичных задач, а также позволяют развивать язык программирования путем внесения в него дополнительных синтаксических конструкций либо расширения стандартной библиотеки. Применение алгоритмов поиска программных идиом к исходным текстам одной программы позволяет предложить варианты рефакторинга ПО [2], например, путем вынесения часто встречающихся фрагментов в отдельные функции. Кроме того, поиск программных идиом позволяет улучшить понимание программы при ее исследовании. Таким образом, задачу поиска программных идиом можно отнести к области понимания программ (program comprehension) [3], активно развивающейся в настоящее время.

В связи с ростом объемов исходных текстов программ и усложнением процесса их разработки задачи понимания программ и автоматического рефакторинга становятся более востребованными. В широкой эксплуатации отсутствуют средства поиска программных идиом, поэтому разработка алгоритмов и программных средств поиска программных идиом является актуальной.

Существующие подходы к автоматическому поиску программных идиом

Одной из первых публикаций, в которой сформулирована задача автоматического извлечения программных идиом из корпуса исходных текстов программ, является работа [1]. Авторы рассматривают программную идиому как фрагмент абстрактного синтаксического дерева (Abstract Syntax Tree, AST) программы, который встречается достаточно часто и имеет отдельное смысловое значение. Для выделения идиом применен аппарат байесовского анализа и вероятностных контекстно-свободных грамматик. Эта работа получила развитие в [4], поскольку методы, основанные на байесовском анализе, не учитывают семантику языка программирования, прежде всего их результатами будут короткие фрагменты исходного текста, которые сложно интерпретировать, при этом широко используемые идиомы не выделяются. Для получения идиом большей длины авторы работы [4] ограничились извлечением идиом, являющихся циклами. В [5] для поиска идиом используются последовательные преобразования синтаксических деревьев. В [6] авторы применяют подход, основанный на поиске часто встречающихся поддеревьев наибольшей длины, что приводит к появлению в результатах слишком специфичных фрагментов. Как отмечено в [1], поиск часто встречающихся поддеревьев дает неудовлетворительные результаты, поскольку более короткие поддеревья встречаются значительно чаще.

Таким образом, необходим подход, позволяющий выделять фрагменты исходного текста достаточно большой длины, при этом достаточно коротких, чтобы не считаться клонами (повторяющимися фрагментами исходного текста программ). Проблема автоматического поиска программных идиом в настоящее время исследована недостаточно. Большая часть публикаций представляет собой либо собрание идиом, полученных вручную [7], либо реализацию алгоритмов поиска уже известных идиом в программных проектах [8]. Автору не удалось найти сведения об аналогичных работах в России, однако есть публикации, посвященные решению связанных задач – поиску повторяющихся фрагментов исходного текста программ и автоматическому рефакторингу [9–11].

Формализация понятия «программная идиома»

Для целей исследования примем, что идиома является поддеревом AST и должна встретиться в исследуемом корпусе текстов программ несколько раз. Как уже было отмечено, короткие поддеревья не дают дополнительной информации о структуре программы, с другой стороны, длинные, но редко встречающиеся поддеревья также не могут считаться программными идиомами. Таким образом, необходима метрика ценности поддерева, которая зависит как от его длины, так и от частоты встречаемости.

Для создания подобной метрики применим аппарат теории информации. Идея состоит в следующем: идиома, будучи замененной на вызов функции или на отдельную синтаксическую конструкцию, должна снижать количество информации в исходном тексте программы, то есть должно происходить сжатие текста программы. В качестве оценки количества информации в тексте программы можно применить метрику Холстеда, которая вычисляется по формуле

$$H = N_1 \log_2 n_1 + N_2 \log_2 n_2, \quad (1)$$

где N_1 – количество операций в тексте программы; n_1 – количество уникальных операций; N_2 – количество операндов; n_2 – количество уникальных операндов.

Для исследования будем оценивать размеры не собственно исходных текстов программ, а AST, построенных из них. Некоторые узлы AST будут соответствовать операторам, некоторые – операндам.

Пусть поддерево AST, соответствующее программной идиоме, содержит L_1 операций и L_2 операндов, при этом встречается C раз в AST. Предположим, что программную идиому можно вынести в отдельную синтаксическую конструкцию (например, в функцию, базовый класс, шаблон, директиву препроцессора и т.д.). Будем считать, что модифицированное AST программы будет получено следующим образом. Сначала к AST добавляется описание программной идиомы, которому дается уникальный идентификатор, что соответствует, например, определению функции. Таким образом, количество уникальных операндов для модифицированной программы $n'_2 = n_2 + 1$. Будем считать, что объявление синтаксической конструкции, соответствующей программной идиоме, требует также одну дополнительную операцию в исходном тексте программы. Затем каждое вхождение программной идиомы будет заменено на два узла AST: узел, соответствующий вставке идиомы в AST (например, вызов функции), и узел, соответствующий имени программной идиомы. Аргументами программной идиомы будут являться дочерние поддеревья, не входящие в поддерево программной идиомы. Преобразование AST при замене идиомы на синтаксическую конструкцию показано на рисунке 1, где закрашенные

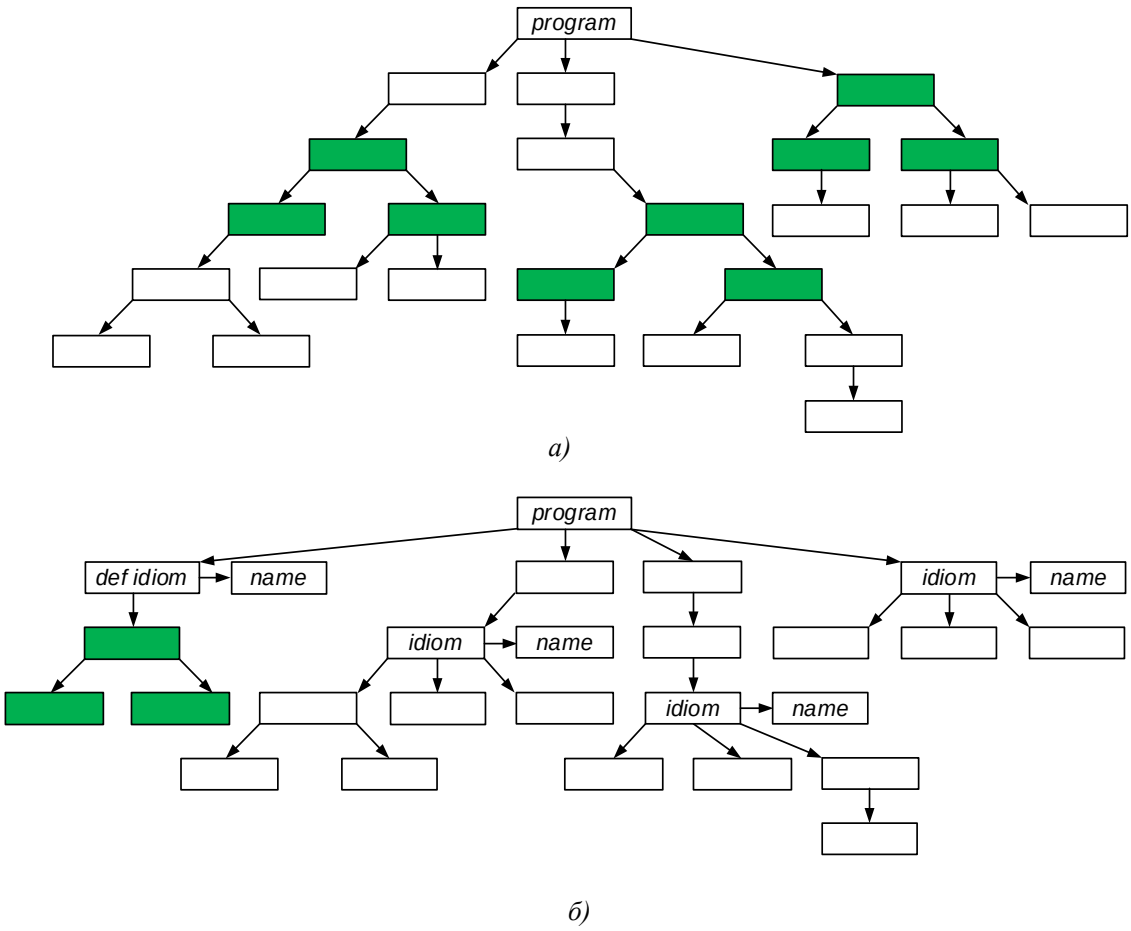

Рис. 1. Пример преобразования AST путем замены программной идиомы на отдельную синтаксическую конструкцию: а) до преобразования, б) после преобразования

Fig. 1. An example of converting an AST by replacing a programming idiom with a separate syntactic construct

прямоугольники обозначают узлы AST, соответствующие программной идиоме. Подписаны следующие узлы: *program* – корень AST, *def idiom* – узел, служащий для определения синтаксической конструкции, на которую может быть заменена идиома, *name* – узел, содержащий имя идиомы, *idiom* – узел, на который заменяется каждое вхождение идиомы.

Таким образом, количество операций и операндов в модифицированном AST составит соответственно:

$$N_1' = N_1 - CL_1 + C + L_1 + 1,$$

$$N_2' = N_2 - CL_2 + C + L_2 + 1.$$

Согласно формуле (1) количество информации в модифицированной программе составит $H' = (N_1 - CL_1 + C + L_1 + 1)\log_2 n_1 + (N_2 - CL_2 + C + L_2 + 1)\log_2(n_2 + 1)$.

Тогда метрикой ценности поддерева T в качестве программной идиомы будем считать значение, на которое сокращается количество информации в программе при оформлении поддерева T в виде отдельной синтаксической конструкции:

$$E(T) = H - H'. \quad (2)$$

Сформулируем следующее определение программной идиомы: поддерево T синтаксического дерева программы является программной идиомой тогда и только тогда, когда $E(T) \geq E(T_1)$, $E(T_2) \geq E(T)$ и $E(T) > 0$, где T_1 – поддерево T, а T – поддерево T_2.

Таким образом, программной идиомой является поддерево с наибольшим значением метрики ценности. Для заданного дерева количество поддеревьев в общем случае экспоненциально, поэтому, чтобы избежать комбинаторного взрыва, необходимо сократить количество рассматриваемых поддеревьев. Построим поддерево итеративно, на каждом шаге добавляя один узел, который обеспечивает максимальный прирост $E(T)$. Процесс построения дерева останавливается, когда $E(T)$ прекращает расти, то есть для поиска программной идиомы используется метод наискорейшего спуска. Таким образом, получен локальный максимум $E(T)$.

Структуры данных для представления поддеревьев

Структура данных для представления AST очевидна. Узел дерева будет содержать его имя (соответствует нетерминальному символу в грамматике языка программирования) и массив дочерних узлов. Будем считать, что лист дерева содержит только имена переменных (или констант), а также значения констант.

Структура данных для представления БД поддеревьев должна обеспечивать

– эффективное добавление новых поддеревьев;

– эффективное хранение поддеревьев, имеющих общие части;

– возможность быстрого получения количества вхождений поддерева в исходный текст программы $C(T)$.

Наиболее удобной структурой, обладающей подобными свойствами, является префиксное дерево. Рассмотрим обобщение префиксного дерева, использующегося для хранения деревьев [12]. Рассматриваемое дерево представляет собой ориентированный ациклический граф, который состоит из узлов двух типов: «ИЛИ»-узлы и «И»-узлы. Корнем такого дерева является узел типа «ИЛИ». Узлы типа «ИЛИ» используются для группировки различных поддеревьев, типа «И» – для представления поддеревьев, содержащих общую операцию. На рисунке 2 приведен пример префиксного дерева. Узлы типа «ИЛИ» обозначены окружностями, типа «И» – прямоугольниками. Представленное префиксное дерево построено для деревьев, соответствующих следующим выражениям некоторого языка программирования: $\{b*d, a + (b + c), a + d, b + (b + c)\}$.

На рисунке 2 терминальными символами являются $a, b, c, d, +, *$, нетерминальными – *var*, *opcode*, *expr*. AST каждого выражения имеет свой номер. Будем считать, что деревья нумеруются последовательно начиная с 0. Выходы узлов типа «ИЛИ» промаркированы множествами номеров выражений, которым принадлежит соответствующий дочерний узел.

На рисунке под а)–г) показаны деревья, соответствующие выражениям, под д) – построенное префиксное дерево.

При работе с набором данных, содержащим большое количество повторяющихся деревьев, целесообразно хранить количество вхождений для каждого сохраненного дерева. Это позволит сократить размеры множеств номеров деревьев, которые хранятся для каждого узла типа «ИЛИ». Следовательно, БД поддеревьев можно определить как $B = \{root, freq\}$, где *root* – корень префиксного дерева (вершина типа «ИЛИ»), *freq* – массив, содержащий $C(T)$ для дерева T с соответствующим индексом. Дерево T получает свой номер во время вставки в B. Номера деревьев растут последовательно вместе с количеством уникальных деревьев, добавленных в B.

Структуры данных, описывающие узлы типа «И» и «ИЛИ», алгоритмы вставки нового

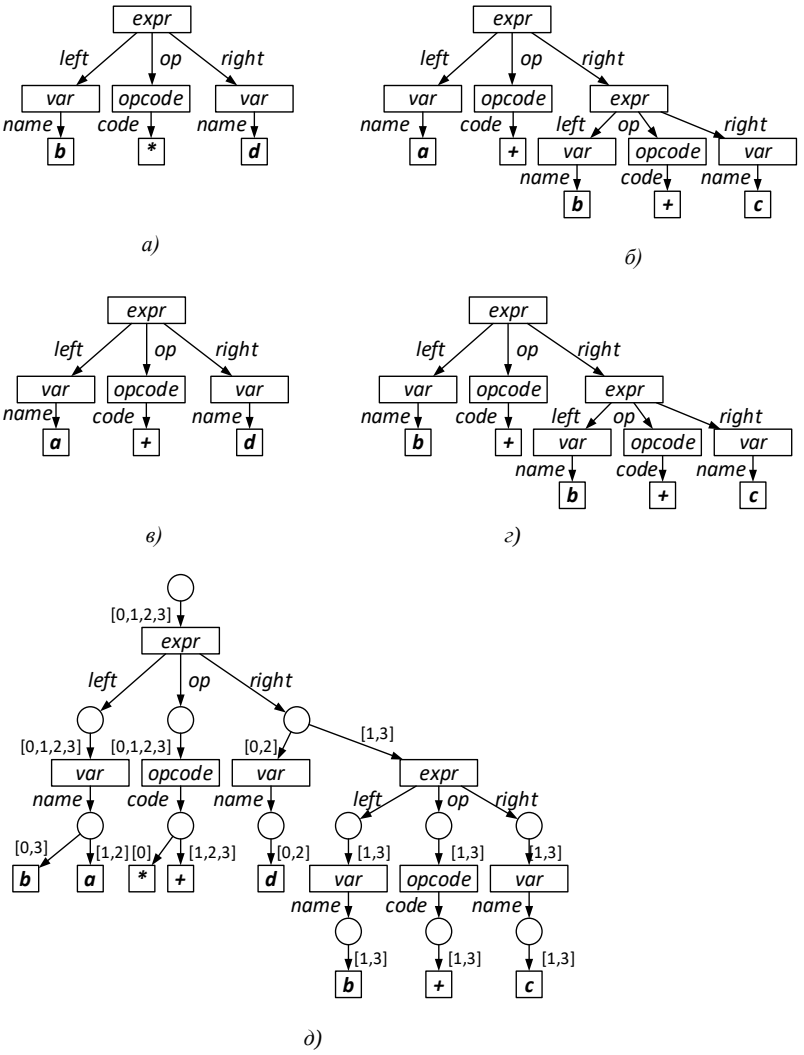

Рис. 2. Пример префиксного дерева

Fig. 2. A prefix tree example

дерева, проверки нахождения дерева в базе рассмотрены в [13].

Поиск максимума функции ценности на множестве поддеревьев

Пусть T – поддерево, построение которого производится. Поддерево T можно представить массивом его граничных узлов – *edgeNodes*. Этот массив содержит узлы типа «ИЛИ», а также листья префиксного дерева из B. Каждый элемент в массиве *edgeNodes* может быть раскрыт, а именно:

– если раскрывается узел типа «ИЛИ», то в поддерево T добавляется один узел, рассматриваемый узел типа «ИЛИ» извлекается из *edgeNodes* и заменяется массивом дочерних узлов;

– если раскрывается лист префиксного дерева, то в поддерево T добавляется лист, а рассматриваемый лист префиксного дерева удаляется из массива *edgeNodes*.

На каждом шаге проверим, раскрытие какого из элементов *edgeNodes* даст наибольшее значение $E(T)$. Добавляем этот узел в T. После добавления каждого узла необходимо сохранять поддерево и идентификаторы деревьев, поддеревом которых оно является. Назовем состоянием поиска кортеж из поддерева, номеров деревьев (*ids*), поддеревом которых оно является, и значения метрики ценности (*efficiency*) для данного дерева. Поддерево будем хранить в виде граничных узлов (*edgeNodes*). Таким образом, *SearchState* = {*edgeNodes*, *ids*, *efficiency*}.

Чтобы получить все возможные идиомы, проведем поиск максимума $E(T)$ для каждого из деревьев, сохраненных в B. Таким образом, будет получен простой алгоритм поиска идиом:

```
Algorithm FindIdioms(B):
begin
    for i in B.freq
    begin
        edgeNodes=[B.root]
        ids=B.freq
        searchState={edgeNodes, ids }
        while true
        begin
            v=ProcessSearchState(searchState, i)
            if v==∅
                break
            newSearchState={(edgeNodes\v) ∪
∪ Children(v,i), ids ∩ Ids(v)}
            if E(searchState)<=E(newSearchState)
               searchState=newSearchState
            else
                break
        end
        ProduceIdiom(searchState)
    end
end
```

В этом фрагменте псевдокода считаем, что $E(searchState)$ возвращает $E(T)$ для поддерева, заданного *searchState.edgeNodes*; *ProcessSearchState* выбирает из *edgeNodes* узел v, раскрытие которого дает наибольшее значение $E(T)$; $Children(v, id)$ возвращает массив узлов, дочерних узлу v, для дерева с номером id; $Ids(v)$ возвращает множество номеров деревьев, в которых имеется узел v; $ProduceIdiom(searchState)$ добавляет поддерево в хранилище идиом.

Данный алгоритм прост, его сложность составляет $O(N_T)$, где N_T – суммарное количество всех узлов рассматриваемых поддеревьев. Алгоритм может быть ускорен за счет повторного использования результатов максимизации $E(T)$. Пусть найденная идиома T для дерева с индексом i также является поддеревом для деревьев с индексами $I = \{i_1, ..., i_c\}$. Это означает, что поиск локального максимума функции ценности для деревьев с индексами $I \backslash i$ можно продолжить начиная с поддерева T. Данную идею можно обобщить. Рассмотрим дерево T_{-1}, из которого было получено дерево T путем добавления одного узла. Пусть дерево T_{-1} является поддеревом для деревьев с индексами I_{-1}, $I \subset I_{-1}$. Следовательно, поиск локального максимума функции ценности для деревьев с индексами $I_{-1} \backslash I$ можно продолжить начиная с поддерева T_{-1}.

Таким образом, если после добавления каждого узла сохранять поддерево и идентификаторы деревьев, поддеревом которых оно является (то есть состояние поиска), то можно избежать повторного раскрытия узлов. Состояния поиска будем хранить в стеке состояний поиска. Как только построение идиомы для всех деревьев, номера которых присутствуют в поле *ids* состояния поиска, будет завершено, состояние поиска извлекается из стека и производится переход к следующему состоянию в стеке. Как только в стеке не останется состояний, алгоритм будет завершен. Псевдокод данного алгоритма следующий:

```
Algorithm FindIdiomsFast(B):
begin
    readyIds = ∅
    searchState={[B.root], b.freq }
    statesStack=[searchState]
    while True
    begin
        i=NextFreeId(readyIds, statesStack)
        searchState=statesStack.top()
        if i==∅
            break
        while True
        begin
            v=ProcessSearchState(searchState, i)
            if v==∅
                break
            newSearchState={(edgeNodes\v) ∪
∪Children(v,i), ids ∩ Ids(v)}
            if E(searchState)<=E(newSearchState)
               statesStack.push(newSearchState)
            else
                break
        end
        ProduceIdiom(searchState)
        readyIds=readyIds ∪ i
    end
end
```

Внутри функции *NextFreeId* происходят выбор id дерева, для которого вычисления еще не произведены, а также удаление из стека тех состояний, для которых обработаны все id.

Представленный выше алгоритм обладает быстродействием $O(N_B)$, где N_B – суммарное количество всех узлов в БД поддеревьев. В наихудшем случае, когда все поддеревья различные, $N_B = N_T$, но на практике выделенные поддеревья имеют общие части, что позволяет достичь ускорения.

Качество выделяемых идиом можно улучшить, если продолжить поиск после достижения первого локального максимума $E(T)$. В этом случае целесообразно продолжить процесс наращивания поддерева T до тех пор, пока $C(T)$ для данного id не станет ниже порога частоты, а затем выбора поддерева, соответствующего максимуму $E(T)$. Порог частоты можно выбирать в зависимости от размера корпуса обрабатываемых исходных текстов и количества идиом, которое необходимо получить. В проведенных экспериментах в качестве порога использован двоичный логарифм количества выделенных поддеревьев.

Эксперименты, проведенные в [13], также показали, что выделенные идиомы нуждаются в фильтрации, то есть в удалении поддеревьев, которым соответствуют вложенные наборы идентификаторов. Пусть поддереву T_1 соответствует набор индексов I_1, поддереву T_2 – набор индексов I_2 и $I_1 \subset I_2$, тогда из набора должно быть исключено то из поддеревьев T_1, T_2, для которого $E(T)$ меньше. Такая фильтрация позволит привести выделенный набор идиом в соответствие определению программной идиомы во второй части данной статьи. Частично эту фильтрацию можно реализовать на этапе построения идиом, исключая из набора поддеревья, находящиеся в стеке состояний выше найденного поддерева.

Поиск программных идиом с использованием разработанного алгоритма

Алгоритмы построения базы поддеревьев и поиска идиом не зависят от исследуемого языка программирования. Таким образом, для

апробации разработанных алгоритмов необходимо выбрать язык программирования, обладающий большим корпусом исходных текстов, для которого задачи преобразования текста в AST и вывода результатов наиболее просты в реализации.

Для проведения эксперимента был выбран язык Python3, поскольку он в настоящее время популярен, использован в большом количестве проектов и в нем широко распространен идиоматический код. Кроме того, Python3 имеет удобные встроенные средства для работы с абстрактными синтаксическими деревьями для программ, написанных на Python.

Язык Python3 имеет контекстно-свободную грамматику, структуры данных для которой представлены в [14]. Эта структура данных практически готова для анализа, за исключением того, что некоторые узлы имеют изменяемое количество дочерних узлов (например, так представлены операторы, содержащиеся в теле цикла). Такие конструкции преобразуются в бинарные поддеревья. Для этого вводится особый тип узла (обозначим его *LIST*), который имеет два дочерних узла. Левый дочерний узел представляет первый элемент в списке, а правый – остальную часть списка. Процедура подготовки данных для анализа подробно описана в [13].

Чтобы получить все значимые идиомы и достичь приемлемой производительности, необходимо тщательно выбрать поддеревья. Ограничим тип поддеревьев, которые будут добавляться в базу. В работе [2] авторы рассматривают только идиомы, являющиеся циклами, в данном исследовании ослабим это ограничение. Пусть идиома может быть представлена только одним оператором (определением класса, определением функции, циклом, условным оператором, блоком with, блоком try/except или выражением). Части выражений не рассматриваются, так как деревья, соответствующие этим частям, имеют небольшой размер и неинформативны. Также не рассматриваются последовательности операторов, поскольку это ограничение серьезно ускоряет вычисления из-за отсутствия необходимости загружать поддеревья деревьев, полученных из списков. Это ограничение выводит из рассмотрения идиомы, являющиеся последовательностями операторов, однако эти последовательности могут быть выделены в поддереве, содержащем родительский оператор. Также ограничим минимальную длину программной идиомы. Это позволяет исключить небольшие, но неинформативные деревья.

Для выделения идиом были выбраны программы на языке Python3, доступные на ресурсе github. Для исследований взяты два набора данных:

– top30 наиболее высоко оцененных (most stars) программ на GitHub, написанных на языке Python3 (https://github.com/topics/python?l=python&o=desc&s=stars);

– библиотека html5lib (https://github.com/html5lib/html5lib-python).

Выбор библиотеки html5lib обусловлен тем, что она имеет достаточно большой объем и содержит значительное количество похожих фрагментов исходного кода, но не повторяющихся дословно. Использование ее для тестов позволит оценить применимость разработанных алгоритмов выделения идиом для рефакторинга программ. Из обоих наборов исключены тесты, так как исходный текст тестов не является исходным текстом типичных программ. Характеристики тестовых наборов представлены в таблице 1, а результаты обработки – в таблице 2.

Таблица 1

Характеристики тестовых наборов

Table 1

Test set characteristics

Набор	Количество файлов	Количество строк	Размер, Кб
top30	12 336	2 095 077	70 633
html5lib	52	15 419	436

Таблица 2

Результаты обработки

Table 2

Processing results

Набор	Количество поддеревьев	Время построения БД поддеревьев, с	Время вычисления идиом, с	Количество идиом
top30	564 220	2 108	122	7 606
html5lib	4 680	2,6	0,5	105

Отобраны 20 идиом с наибольшими значениями $E(T)$, выделенные из набора top30 (см. http://www.swsys.ru/uploaded/image/2022-1/2022-1-dop/14.jpg).

Автоматически выделенные программные идиомы обладают достаточной длиной, осмыслены, но требуют дополнительной фильтрации (например, идиомы 4 и 5 могут быть отождествлены). Выделенные фрагменты поддеревьев можно достаточно легко интерпретировать. Приведем несколько примеров автоматически выделенных идиом:

```
1.  $targets=os.path.join(...,...)
2.  await asyncio.gather(*$value,$keywords)
3.  while $expr $op $expr:
     ...
     $id+=1
4.  return {$id : $value for ($id,$id) in
    $value.items($args) if $ifs}
5.  $id=[$elt for (i,$id) in enumerate($args) if
    $ifs]
6.  for $id in $id:
     $id=$value
     if (not $operand):
      continue
     ...
```

где 1 – объединение частей пути к файлу в строку и присвоение ее некоторой цели; 2 – ожидание получения результатов асинхронного выполнения; 3 – цикл while, в котором также используется индекс; 4 – возврат из функции словаря, содержащего значения другого словаря, отфильтрованные по некоторому условию; 5 – присвоение некоторой цели массива, сгенерированного из другого массива; 6 – цикл по элементам контейнера, для некоторых из которых обработка не производится.

Разработанный инструментарий можно применить и к отдельным проектам. В этом случае будут выделены фрагменты, характерные для данного проекта. В целом выделенные фрагменты будут иметь большую длину, чем фрагменты, выделенные из нескольких проектов. Рассмотрим идиомы, выделенные из исходного текста библиотеки.

```
1.  self.tokenQueue.append({"type":token-
    Types[$value],"data":$value})
2.  class $name (Phase):
     def __init__(self,parser,tree):
      Phase.__init__(self,parser,tree)
      self.startTagHandler=_utils.Method-
    Dispatcher([$list])
      self.startTagHandler.default=self.start-
    TagOther
      self.endTagHandler=_utils.Method-
    Dispatcher([$list])
      self.endTagHandler.default=self.endTagOther
     def $name($args):
      $body
     def $name($args):
      $body
     def $name($args):
      $body
     def $name(self,token):
      $body
     ...
3.  if data $op $expr:
     $body
    else:
     if data == $s:
      $body
     else:
      if data == $s:
       self.tokenQueue.append({"type":token-
    Types[$value],"data":$s})
       ...
      else:
       if data $op $expr:
        self.tokenQueue.append({"type":token-
    Types["ParseError"],"data":$s})
        ...
      else:
       $orelse
4.  def __init__(self,parser,tree):
     Phase.__init__(self,parser,tree)
     self.startTagHandler=_utils.Method-
    Dispatcher([$list])
     self.startTagHandler.default=self.start-
    TagOther
     self.endTagHandler=_utils.Method-
    Dispatcher([$list])
     self.endTagHandler.default=self.endTagOther
5.  if data $op $expr:
     self.tokenQueue.append({"type":token-
    Types["ParseError"],"data":$s})
     self.currentToken["correct"]=False
     self.tokenQueue.append(self.currentToken)
     self.state=self.dataState
    else:
     $orelse
```

Они позволяют найти возможные варианты рефакторинга. Так, идиомы 1 и 5 могут быть вынесены в отдельные методы класса, идиома 2 использована для выделения суперкласса. Отметим, что идиома 4 содержится в идиоме 1, таким образом, необходимо улучшать алгоритмы фильтрации идиом.

Заключение

Статья посвящена автоматическому извлечению программных идиом из исходных текстов программ. В данной работе программная идиома рассматривается как поддерево T синтаксического дерева программы. Разработаны метрика ценности идиомы $E(T)$, основанная на метрике Холстеда, а также алгоритм автоматического извлечения идиом, основанный на поиске максимумов функции ценности поддерева методом наискорейшего спуска.

Для хранения поддеревьев для дальнейшей обработки алгоритмом использована структура данных на основе префиксного дерева.

Разработанный алгоритм извлечения программных идиом реализован на языке Python3. Программа получает на вход исходные тексты программ на языке Python3. Разработанная программа протестирована на наборах: top30 наиболее высоко оцененных (most stars) программ на GitHub, написанных на языке Python3; библиотека html5lib. Автоматически выделенные идиомы можно легко интерпретировать, однако полученный набор идиом содержит повторяющиеся почти идентичные фрагменты, таким образом, необходима дополнительная фильтрация идиом (отождествление близких идиом). Разработанная программа, примененная к исходному коду отдельной программы, может быть использована для рефакторинга, так как позволяет находить фрагменты, которые можно оформить как отдельные функции или классы.

Литература

1. Allamanis M., Sutton C. Mining idioms from source code. Proc. XXII ACM SIGSOFT Int. Symposium on FSE, 2014, pp. 472–483. DOI: 10.1145/2635868.2635901.
2. Nucci D.D., Pham H., Fabry J. et al. A language-parametric modular framework for mining idiomatic code patterns. Proc. XII SATToSE, 2019. URL: http://ceur-ws.org/Vol-2510/sattose2019_paper_3.pdf (дата обращения: 20.10.2021).
3. Белеванцев А.А., Велесевич Е.А. Анализ сущностей программ на языках Си/Си++ и связей между ними для понимания программ // Тр. ИСП РАН. 2015. Т. 27. № 2. С. 53–64. DOI: 10.15514/ISPRAS-2015-27(2)-4.
4. Allamanis M., Barr E.T., Bird C., Devanbu P., Marron M., Sutton C. Mining semantic loop idioms. IEEE Transactions on Software Engineering, 2018, vol. 44, no. 7, pp. 651–668. DOI: 10.1109/TSE.2018.2832048.
5. Iyer S., Cheung A., Zettlemoyer L. Learning programmatic idioms for scalable semantic parsing. Proc. EMNLP-IJCNLP, 2019, pp 5426–5435. DOI: 10.18653/v1/D19-1545.
6. Pham H., Nijssen S., Mens K., Nucci D.D., Molderez T., De Roover C. et al. Mining patterns in source code using tree mining algorithms. Discovery Science, 2019, pp. 471–480. DOI: 10.1007/978-3-030-33778-0_35.
7. Alexandru C., Merchante J., Panichella S., Proksch S., Gall H., Robles G. On the usage of pythonic idioms. Proc. ACM SIGPLAN Int. Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, 2018, pp. 1–11. DOI: 10.1145/3276954.3276960.
8. Merchante J., Robles G. From Python to Pythonic: Searching for Python idioms in GitHub. Proc. SATToSE, 2017. URL: http://sattose.wdfiles.com/local--files/2017:schedule/SATToSE_2017_paper_15.pdf (дата обращения: 20.10.2021).
9. Луговской Н.Л. Подход для проведения рефакторинга «Выделение функции» в инструменте Klocwork Insight // Тр. ИСП РАН. 2012. Т. 23. С. 107–132. DOI: 10.15514/ISPRAS-2012-23-7.
10. Саргсян С., Курмангалеев Ш., Белеванцев А., Асланян А., Балоян А. Масштабируемый инструмент поиска клонов кода на основе семантического анализа программ // Тр. ИСП РАН. 2015. Т. 27. № 1. С. 39–50. DOI: 10.15514/ISPRAS-2015-27(1)-3.
11. Осадчая А.О., Исаев И.В. Метод поиска клонов в программном коде // Научно-технический вестник информационных технологий, механики и оптики. 2020. Т. 20. № 5. С. 714–721. DOI: 10.17586/2226-1494-2020-20-5-714-721.
12. Ružička P., Prívara I. On tree pattern unification problems. Proc. IX FCT, 1993, pp. 418–429. DOI: 10.1007/3-540-57163-9_36.
13. Orlov D. Finding idioms in source code using subtree counting techniques. In: Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles, pp. 44–54. DOI: 10.1007/978-3-030-61470-6_4.
14. Abstract Syntax Trees. URL: https://docs.python.org/3/library/ast.html (дата обращения: 20.10.2021).

Software & Systems
DOI: 10.15827/0236-235X.137.065-074

Received 16.11.21, Revised 31.01.22
2022, vol. 35, no. 1, pp. 065–074

An algorithm of idiom search in program source codes using subtree counting

D.A. Orlov [1], *Ph.D. (Engineering), Associate Professor, orlovdmal@mpei.ru*

[1] *National Research University "Moscow Power Engineering Institute",*
Moscow, 111250, Russian Federation

Abstract. The paper is dedicated to programming idiom extraction algorithm design. Programming idiom is the fragment of source code which often occurs in different programs and used for solving one typical programming task. In this research the programming idiom is a source code fragment that often occurs in different programs and used for solving one typical programming task. In this research, the programming idiom is considered as the part of a program abstract syntax tree (AST), which provides maximum reduce of information quantity in a source code, when all of programming idiom occurrences are replaced with certain syntax construction (e.g., function call).

The developed subtree value metric estimates information amount reduce after such replace. Therefore, the idiom extraction is reduced to search of subtree value function maximum on AST subtree set. To reduce a

number of subtrees inspected, the authors use steepest descent method for subtree value function maximum search. At each step subtree is extended with one node, which provides maximum increase of a subtree value metric. Subtrees are stored in a data structure that is a generalization of a trie data structure.

The paper proposes an accelerated algorithm of idiom extraction. Programming idiom extraction speedup is achieved through reusing results of idiom efficiency maximum search. The paper also describes the implementation of the developed algorithms. The algorithms are implemented in Python programming language. The implementation extracts programming idioms from source code written in Python. This programming language is chosen due to a large corpus of texts written in such language; it also includes convenient tools for building AST.

The authors carried out an idiom extraction experiment using the developed implementation. The idioms were extracted from corpora of an open-source program source code. The extracted programming idioms are source code fragments with own meaning. It is also shown that applying developed algorithms to a source code of a single software project can reveal possibilities of investigated program refactoring.

Keywords: programming idiom, source code analysis, refactoring, data mining, AST, Python.

References

1. Allamanis M., Sutton C. Mining idioms from source code. *Proc. XXII ACM SIGSOFT Int. Symposium on FSE*, 2014, pp. 472–483. DOI: 10.1145/2635868.2635901.

2. Nucci D.D., Pham H., Fabry J. et al. A language-parametric modular framework for mining idiomatic code patterns. *Proc. XII SATToSE*, 2019. Available at: http://ceur-ws.org/Vol-2510/sattose2019_paper_3.pdf (accessed October 20, 2021).

3. Belevantsev A., Velesevich E. Analyzing C/C++ code entities and relations for program understanding. *Proceedings of ISP RAS*, 2015, vol. 27, no. 2, pp. 53–64. DOI: 10.15514/ISPRAS-2015-27(2)-4 (in Russ.).

4. Allamanis M., Barr E.T., Bird C., Devanbu P., Marron M., Sutton C. Mining semantic loop idioms. *IEEE Transactions on Software Engineering*, 2018, vol. 44, no. 7, pp. 651–668. DOI: 10.1109/TSE.2018.2832048.

5. Iyer S., Cheung A., Zettlemoyer L. Learning programmatic idioms for scalable semantic parsing. *Proc. EMNLP-IJCNLP*, 2019, pp 5426–5435. DOI: 10.18653/v1/D19-1545.

6. Pham H., Nijssen S., Mens K., Nucci D.D., Molderez T., De Roover C. et al. Mining patterns in source code using tree mining algorithms. *Discovery Science*, 2019, pp. 471–480. DOI: 10.1007/978-3-030-33778-0_35.

7. Alexandru C., Merchante J., Panichella S., Proksch S., Gall H., Robles G. On the usage of pythonic idioms. *Proc. ACM SIGPLAN Int. Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2018, pp. 1–11. DOI: 10.1145/3276954.3276960.

8. Merchante J., Robles G. From Python to Pythonic: Searching for Python idioms in GitHub. *Proc. SATToSE*, 2017. Available at: http://sattose.wdfiles.com/local--files/2017:schedule/SATToSE_2017_paper_15.pdf (accessed October 20, 2021).

9. Lugovskoy N.L. The refactoring approach used in Klocwork Insight toolkit. Proceedings of the Institute for System. *Proceedings of ISP RAS*, 2012, vol. 23, pp. 107–132. DOI: 10.15514/ISPRAS-2012-23-7 (in Russ.).

10. Sargsyan S., Kurmnagaleev S., Belevantsev A., Aslanyan H., Baloian A. Scalable code clone detection tool based on semantic analysis. *Proceedings of ISP RAS*, 2015, vol. 27, no. 1, pp. 39–50. DOI: 10.15514/ISPRAS-2015-27(1)-3 (in Russ.).

11. Osadchaya A.O., Isaev I.V. Search of clones in program code. *Sci.Tech. J. Inf. Technol. Mech. Opt.*, 2020, vol. 20, no. 5, pp. 714–721. DOI: 10.17586/2226-1494-2020-20-5-714-721 (in Russ.).

12. Ružička P., Prívara I. On tree pattern unification problems. *Proc. IX FCT*, 1993, pp. 418–429. DOI: 10.1007/3-540-57163-9_36.

13. Orlov D. Finding idioms in source code using subtree counting techniques. In: *Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles*, pp. 44–54. DOI: 10.1007/978-3-030-61470-6_4.

14. *Abstract Syntax Trees*. Available at: https://docs.python.org/3/library/ast.html (accessed October 20, 2021).

Для цитирования

Орлов Д.А. Алгоритм поиска идиом в исходных текстах программ, использующий подсчет поддеревьев // Программные продукты и системы. 2022. Т. 35. № 1. С. 065–074. DOI: 10.15827/0236-235X.137.065-074.

For citation

Orlov D.A. An algorithm of idiom search in program source codes using subtree counting. *Software & Systems*, 2022, vol. 35, no. 1, pp. 065–074 (in Russ.). DOI: 10.15827/0236-235X.137.065-074.