УДК 004.02

Сравнение исходных текстов программ путем выравнивания последовательностей токенов

© А.В. Дубанов

МГТУ им. Н.Э. Баумана, Москва, 105005, Россия

В настоящее время весьма актуальна проблема обнаружения заимствований в текстах. В данной работе был модифицирован один из известных алгоритмов выравнивания последовательностей биополимеров для того, чтобы сравнивать исходные тексты программ и выявлять в них похожие фрагменты. Входными данными этого алгоритма являются исходные тексты программ, которые рассматриваются как последовательности символов. Лексические домены при этом соответствуют алфавиту символов, составляющих эти последовательности. Алгоритм был реализован в виде программы, его работа продемонстрирована на фрагментах кода, написанных на языке Scheme. В статье обсуждаются перспективы и ограничения применения алгоритма.

Ключевые слова: *заимствование кода, выравнивание последовательностей, наибольшая общая подпоследовательность, динамическое программирование, лексический анализ, функциональное программирование.*

Введение. Проблема поиска некорректных заимствований (в том числе в исходных кодах программ) является весьма злободневной задачей [1]. В практике кафедры «Теоретическая информатика и компьютерные технологии» эта задача систематически возникает при проверке домашних заданий, результаты которых должны быть представлены в виде программ на языках программирования высокого уровня, и имеет целью выявление случаев «списывания». Для решения этой задачи требуется разработать средства для сравнения исходных текстов, позволяющие находить похожие или идентичные фрагменты в исходных кодах программ, оценивать степень обнаруженного сходства и наглядно представлять результат сравнения.

Очевидно, что такая задача является обобщением задачи нахождения наибольшей общей подпоследовательности в двух последовательностях символов. Различные методы нахождения наибольших общих или идентичных подпоследовательностей в исходном коде реализованы в ряде компьютерных программ «антиплагиата», наиболее известной из которых является MOSS [2]. Однако практическое применение программ «антиплагиата» имеет ряд ограничений как технических, так и лицензионных. Среди удачных приемов следует отметить сравнение не отдельных литер, а *токенов*. Это позволяет сделать метод нечувствительным к переименованию идентификаторов.

Для решения задачи поиска похожих фрагментов кода наше внимание привлекли *методы парного выравнивания последовательностей*. Эти методы широко применяются в биоинформатике для аннотации белковых и нуклеотидных последовательностей по сходству. Такая аннотация основана на выявлении сходных (необязательно идентичных) подпоследовательностей в аннотированной последовательности и в последовательности, охарактеризованной ранее. В биоинформатике такие подпоследовательности называют гомологичными или консервативными. Применительно к исходному коду будем называть их похожими или сходными. Весьма привлекательными особенностями алгоритмов локального выравнивания являются возможности их настройки, статистической оценки уровня значимости обнаруженного сходства, а также наглядного представления результатов выравнивания [3]. Методы локального выравнивания уже успешно применяются на практике для анализа научно-технических текстов [4]. Поэтому мы предложили использовать парное выравнивание для сравнения исходных кодов.

Целью данной работы были модификация алгоритма выравнивания последовательностей символов для сравнения исходных текстов программ и демонстрация его способности находить похожие фрагменты в исходных текстах программ. В дальнейшем этот алгоритм предполагается использовать для выявления случаев «списывания» домашних заданий студентами, поэтому он должен быть нечувствительным к переименованию идентификаторов (имен процедур и переменных), задаваемых программистом. В связи с этим в качестве выравниваемых последовательностей рассматривались последовательности токенов, полученных при лексическом анализе исходных текстов.

Алгоритм выравнивания последовательностей. Алгоритм был получен путем модификации одного из описанных в литературе алгоритмов выравнивания последовательностей биополимеров и реализован на языке Haskell редакции 2010 г. с использованием Haskell Platform 2013.2.0.0 в операционной системе OpenSUSE 13.1 на персональном компьютере с процессором Intel Core i5 (2,6 ГГц) и 4 Гбайт RAM. Разработанная программа включает в себя лексический анализатор, реализацию алгоритма выравнивания и средства вывода результата вычислений в виде HTML-файла, работающие последовательно. Тесты были выполнены на примерах фрагментов кода, написанных на ограниченном подмножестве языка Scheme 5-й редакции (R5RS).

За основу был взят алгоритм парного выравнивания, предназначенный для нахождения непересекающихся участков наибольшего сходства в двух последовательностях биополимеров, причем необяза-

тельно следующих в одинаковом порядке в обеих последовательностях [5]. Отличительной особенностью такого алгоритма является возможность обнаружения наиболее похожих подпоследовательностей. При этом не исключаются случаи многократного вхождения одной подпоследовательности в какую-нибудь из последовательностей. К другим достоинствам данного алгоритма относится возможность найти похожие фрагменты последовательностей и получить выравнивание за единственный проход по заполненной таблице, что позволяет сократить время вычислений.

В биоинформатике выравнивание обычно предусматривает разрывы (делеции) более короткой последовательности в пределах консервативных фрагментов, что обусловлено особенностями исследуемых объектов. Например, делеция в выравнивании может быть результатом мутации, не приводящей к существенным изменениям в структуре и свойствах белка. В противоположность этому применительно к исходному коду вставка или удаление токена из последовательности в типичном случае приводит к изменению смысла программы или к ошибочному коду. Поэтому мы сочли необходимым искать *непрерывные* непересекающиеся сходные подпоследовательности. Исходя из этой необходимости, можно предложить следующую модификацию алгоритма.

Пусть A и B — последовательности длиной m и n символов соответственно. Тогда для выравнивания этих последовательностей потребуется таблица T размером $h = n + 1$ строк на $w = m + 1$ столбцов. Обозначим индекс столбца как i, индекс строки как j и будем нумеровать столбцы и строки таблицы, начиная с 0, а символы в последовательностях, начиная с 1. Тогда значение в ячейке таблицы $T(i, j)$ при $i > 0$ и $j > 0$ будет содержать оценку сходства для пары символов: i-го символа последовательности A и j-го символа последовательности B.

Заполнение таблицы T начинается с верхнего левого угла:

$$T(0,0) = 0, \tag{1}$$

а дальнейшее заполнение осуществляется по столбцам слева направо, т. е. при заполнении таблицы индекс столбца i принимает значения от 0 до w. Каждый столбец заполняется сверху вниз, т. е. при заполнении столбца индекс строки j принимает значения от 0 до h. Правила заполнения таблицы могут быть выражены следующими формулами:

$$T(i,0) = \max \begin{cases} T(i-1,0); \\ T(i-1,j) - t, \ j \in [1,h]; \end{cases} \tag{2}$$

$$T(i,j)=\max\begin{cases}T(i-1,\ 0);\\ T(i-1,j-1)+s(a_i,b_j);\\ T(i,j-1)-g;\\ T(i-1,j)-g;\end{cases} \tag{3}$$

$$i\in[1,w],\ \ j\in[1,h], \tag{4}$$

где $T(i, j)$ — заполняемая ячейка таблицы ($T(i, 0)$ соответствует 0-й ячейке i-го столбца); $s(a_i, b_j)$ — оценка сходства (score — счет) между символами a_i и b_j; g — штраф за разрыв последовательности (делецию); t — некоторое пороговое значение, позволяющее отсечь случаи слабого сходства, если в этом возникнет необходимость.

Выравнивание последовательностей получается путем *поиска последовательности ячеек в таблице исходя из принципа максимизации оценки сходства*. Просмотр таблицы осуществляется справа налево, т. е. индекс столбца i изменяется от w до 0. Непрерывные сходные фрагменты представлены в таблице непрерывными последовательностями ячеек с ненулевыми значениями, расположенными по диагонали (далее диагоналями).

Поиск правого нижнего конца диагонали осуществляется путем поиска максимального значения в очередном, i-м столбце таблицы. Это может быть выражено следующими формулами:

$$p_0(i)=\begin{cases}(0,\ 0), & i=0, j=0;\\ p_0(i-1), & i=0, j\neq 0;\\ p_0(i-1), & i\neq 0, j\neq 0,\ T(i,j)=0;\\ (i,\ j), & i\neq 0, j\neq 0,\ T(i,j)\neq 0;\end{cases} \tag{5}$$

$$j=\min\left\{j\,|\,T(i,j)=\max\{T(i,j)\},\ j\in[0,h]\right\}, \tag{6}$$

где i — индекс столбца таблицы; j — индекс строки таблицы; $T(i, j)$ — значение в ячейке таблицы; $p_0(i)$ — функция для вычисления индексов правого нижнего конца очередной диагонали в таблице, расположенного в столбце i или левее.

Результатом вычисления $p_0(i)$ будет пара (i, j), такая, что $T(i, j)$ является правым нижним концом диагонали. Значение $p_0(i)=(0,0)$ указывает на то, что достигнут верхний левый угол таблицы, новых похожих подпоследовательностей не найдено. Аналогично $p_0(i)=(0,j)$ при любом допустимом j указывает на то, что достигнут

крайний левый столбец таблицы и новых сходных подпоследовательностей не может быть найдено. В случае если *i*-й символ не входит в подпоследовательность последовательности *A*, которой соответствует похожая на нее подпоследовательность в последовательности *B*, первое в столбце (при просмотре ячеек сверху вниз) максимальное значение будет найдено в ячейке с индексом $j = 0$. В этом случае поиск нижнего правого конца непрерывной диагонали продолжается в столбце с индексом $i - 1$.

После нахождения правого нижнего конца диагонали осуществляется *поиск входящих в диагональ ячеек*, который продолжается до тех пор, пока не будет встречено нулевое значение в ячейке с индексами $i > 0$ и $j > 0$. При этом правый нижний конец диагонали включается в последовательность пар символов, входящих в похожие последовательности *A* и *B*. В противном случае осуществляется поиск новой диагонали. Поиск завершается по достижении крайнего левого столбца таблицы, где $p(i, j)$ принимает значение ( ) (пустая последовательность). Такой поиск может быть описан следующей рекурсивной формулой:

$$p(i,j) = \begin{cases} (\,), & i = 0; \\ (p \circ p_0)(i-1), & i \neq 0,\ j = 0; \\ (p \circ p_0)(i-1), & i \neq 0,\ j \neq 0,\ T(i,j) = 0; \\ \big((i,j), p(i-1,j-1)\big), & i \neq 0,\ j \neq 0,\ T(i,j) \neq 0, \end{cases} \tag{7}$$

где $p(i, j)$ — функция вычисления очередной пары в выравнивании, следующей за парой (i, j) в общей последовательности $\big((i,j), p(i-1,j-1)\big)$ (показана в виде вложенных пар, в программе реализована в виде списка), соответствующей парам символов a_i и b_j, входящих в состав сходных подпоследовательностей последовательностей *A* и *B*.

Как было отмечено, поиск диагоналей в таблице осуществляется справа налево. Поскольку первая диагональ может быть найдена по формулам (5) и (6), то последовательность всех пар символов, входящих в непересекающиеся наиболее похожие подпоследовательности, двух последовательностей может быть задана формулой

$$P = \big(p \circ p_0\big)\big(w\big), \tag{8}$$

где *P* — последовательность пар индексов (i, j), которая и является искомым выравниванием.

Таким образом удается найти непрерывные непересекающиеся сходные подпоследовательности в двух последовательностях.

Оценка сходства всех найденных непрерывных подпоследовательностей может быть получена по формуле

$$s_{\max} = \max\left\{ s \middle| T(i,j),\ i \in [1,w],\ j \in [1,h] \right\}. \qquad (9)$$

Здесь поиск максимального значения осуществляется среди индексов символов последовательностей A и B. Максимальное значение соответствует нижнему правому концу крайней правой диагонали.

Пример заполнения таблицы T для строк ABCxDEFGx и DEFGoABCo (жирным шрифтом показаны ячейки, образующие диагонали, и соответствующие этим диагоналям похожие (в данном случае идентичные) подпоследовательности в последовательностях; горизонтальными и вертикальными стрелками показан поиск нижних правых концов диагоналей, диагональными стрелками — поиск ячеек, образующих диагональ):

		A	**B**	**C**	x	**D**	**E**	**F**	**G**	x
	0	0	1	2	3	3	4	5	6	7
D	0	0	1	2	3	**4**	4	5	6	7
E	0	0	1	2	3	3	**5**	5	6	7
F	0	0	1	2	3	3	4	**6**	6	7
G	0	0	1	2	3	3	4	5	**7**	7
o	0	0	1	2	3	3	4	5	6	7
A	0	**1**	1	2	3	3	4	5	6	7
B	0	0	**2**	2	3	3	4	5	6	7
C	0	0	1	**3**	3	3	4	5	6	7
o	0	0	1	2	3	3	4	5	6	7

Алгоритм находит соответствие между одинаковыми символами, подпоследовательностями ABC и подпоследовательностями DEFG в сравниваемых последовательностях.

В биоинформатике при выравнивании нуклеотидных последовательностей, алфавиты которых включают в себя только четыре символа, обозначающих нуклеотиды, $s(a_i, b_j)$ принимают равным 1 для совпадающих символов и −1 для несовпадающих, штраф за делецию устанавливается равным 2. При выравнивании аминокислотных последовательностей белков, алфавит которых составляют 20 символов стандартных аминокислотных остатков, причем замены одних на дру-

гие неравноценны, $s(a_i, b_j)$ находят в матрице замен (таблицы цены замены одного символа алфавита на другой символ), а значение (иногда и способ вычисления) g подбирается для конкретных матрицы замен, алгоритма выравнивания и задач исследования.

Во всех примерах, рассмотренных в данной статье, было принято $s(a_i, b_j) = 1$ при $a_i = b_j$ и $s(a_i, b_j) = -1$ в противном случае, $g = 2$ и $t = 0$. Такие значения $s(a_i, b_j)$ и g выбраны исходя из результатов предварительных тестов предложенного алгоритма. Эти тесты показали приемлемые результаты выравнивания коротких последовательностей символов. При таких параметрах алгоритм настроен на поиск непересекающихся идентичных подпоследовательностей в последовательностях A и B. Кроме того, такие значения позволяют легко интерпретировать результат сравнения: при запрете на делеции в участках локального сходства значение в ячейке таблицы в нижнем правом конце непрерывной диагонали из нескольких ячеек будет равно количеству символов, совпадающих в обеих последовательностях. Значение $s_{\max}$ при этом будет соответствовать количеству символов в наибольшей непрерывной общей подпоследовательности последовательностей A и B. Значение $t = 0$ в данной работе было выбрано так, чтобы оно не оказывало влияния на получаемый результат. Выбор значения t для практических целей нуждается в дополнительном исследовании и обсуждении.

Следует отметить, что выравнивание по данному алгоритму как операция над последовательностями является некоммутативной. Так, при выравнивании A к B и B к A в общем случае будут получаться различные таблицы, которые, соответственно, могут давать различные выравнивания, по крайней мере, при слабом сходстве. Эта особенность имеет место при использовании парного выравнивания в биоинформатике. Возможно, эту особенность будет необходимо учитывать и при сравнении исходных текстов программ с помощью выравнивания.

В настоящей работе при сравнении исходных текстов программ последовательностями символов являлись последовательности токенов, полученные при лексическом анализе. Токены считались одинаковыми, если совпадали их лексические домены. Таким образом, алфавитом являлось множество лексических доменов языка, распознаваемых лексическим анализатором. Благодаря этому методика сравнения оказывалась нечувствительной к замене идентификаторов.

Лексический анализатор распознавал следующие лексические домены подмножества языка Scheme 5-й редакции: идентификаторы, ключевые слова (else, =>, define, unquote, unquote-splicing, quote, lambda, if, set!, begin, cond, and, or, case, let, let*, letrec, do, delay,

quasiquote), открывающие скобки списков, открывающие скобки векторов, закрывающие скобки списков и векторов, числа (ограниченное подмножество), булевы значения, строки, символы (characters), символы цитирования и квазицитирования, точки в паттернах списков. В каждом примере присутствовали токены, принадлежащие по крайней мере четырем доменам.

Для визуализации результатов сравнения последовательностей применялась HTML-разметка исходных текстов. Разметка выполнялась путем отображения выравнивания на исходный код с использованием координат токенов (строка, столбец в исходном коде), найденных лексическим анализатором. Токены, входящие в непрерывные сходные фрагменты обоих исходных кодов (в данном случае идентичных), выделялись жирным шрифтом. Комментарии переносились из исходного текста в размеченный в неизменном виде и анализу не подвергались.

Рассмотрим **четыре примера сравнения исходных кодов программ на языке Scheme**, выполненных данным способом.

В этих примерах сравниваемые исходные тексты следуют один за другим, код показан моноширинным шрифтом, прочее — пропорциональным шрифтом. Совпадающие фрагменты кода выделены жирным шрифтом, внизу каждого листинга приведено значение оценки сходства, рассчитанное по формуле (9).

Пример 1

test-1.scm

```
(define (fctrl number)
  (if (zero? number) 1 (* number (fctrl (- number 1)))))
```

simple-factorials.scm

```
(define (factorial-classic n)
  (if (= n 0) 1 (* n (factorial-classic (- n 1)))))

(define (factorial-classic-short n)
  (if (zero? n) 1 (* n (factorial-classic-short (- n 1)))))

(define (factorial-with-do n)
  (define p 1)
  (do ((i 1 (+ i 1))) ((> i n) p) (set! p (* p i))))

(define (factorial-with-do-variant n)
  (do ((i 1 (+ i 1)) (p 1 (* i p))) ((> i n) p)))

(define (factorial-iterative n)
  (define (loop i)
    (if (< i n) (* i (loop (+ i 1))) n))
  (loop 1))
```

Score: 27

Пример 2

test-2.scm

```
(define (fctrl x)
  (do ((k 1 (+ k 1)) (product 1 (* k product))) ((> k x) product)))
```

simple-factorials.scm

```
(define (factorial-classic n)
  (if (= n 0) 1 (* n (factorial-classic (- n 1)))))

(define (factorial-classic-short n)
  (if (zero? n) 1 (* n (factorial-classic-short (- n 1)))))

(define (factorial-with-do n)
  (define p 1)
  (do ((i 1 (+ i 1))) ((> i n) p) (set! p (* p i))))

(define (factorial-with-do-variant n)
  (do ((i 1 (+ i 1)) (p 1 (* i p))) ((> i n) p)))

(define (factorial-iterative n)
  (define (loop i)
    (if (< i n) (* i (loop (+ i 1))) n))
  (loop 1))
```

Score: 38

Пример 3

insertion-sort.scm

```
(define (insert-ex pred? x xs)
  (if (null? xs)
      (list x)
      (if (pred? x (car xs))
          (cons x xs)
          (cons (car xs) (insert-ex pred? x (cdr xs))))))

(define (insertion-sort pred? xs)
  (if (null? xs)
      '()
      (insert-ex pred? (car xs) (insertion-sort pred? (cdr xs)))))
```

naive-lexer.scm

```
(define (tokenize xs)
  (define (tokenize-rest xs)
    (if (null? xs)
        (list 'paren-right)
        (let ((x (car xs)))
          (cond
            ((list? x)
             (append (list 'paren-left)
                     (tokenize-rest x)
                     (tokenize-rest (cdr xs))))
            ((vector? x)
             (append (list 'hash-paren-left)
                     (tokenize-rest (vector->list x))
                     (tokenize-rest (cdr xs))))
            ((symbol? x)
             (cons 'symbol
                   (tokenize-rest (cdr xs))))
            (else
             (cons 'other
                   (tokenize-rest (cdr xs)))))))))
  (cons 'paren-left (tokenize-rest xs)))
```

Score: 74

Пример 4

factorial-memoized.scm

```
(define factorial-memoized
  (let ((memo '()))
    (lambda (n)
      (let ((memoized (assq n memo))
            (factorial
             (lambda (m)
               (if (zero? m) 1 (* m (factorial-memoized (- m 1)))))))
        (if memoized
            (cadr memoized)
            (let ((n! (factorial n)))
              (set! memo (cons (list n n!) memo))
              n!))))))
```

naive-lexer.scm

```
(define (tokenize xs)
  (define (tokenize-rest xs)
    (if (null? xs)
        (list 'paren-right)
        (let ((x (car xs)))
          (cond
            ((list? x)
             (append (list 'paren-left)
                     (tokenize-rest x)
                     (tokenize-rest (cdr xs))))
            ((vector? x)
             (append (list 'hash-paren-left)
                     (tokenize-rest (vector->list x))
                     (tokenize-rest (cdr xs))))
            ((symbol? x)
             (cons 'symbol
                   (tokenize-rest (cdr xs))))
            (else
             (cons 'other
                   (tokenize-rest (cdr xs)))))))))
  (cons 'paren-left (tokenize-rest xs)))
```

Score: 81

Как следует из примеров 1 и 2, алгоритм уверенно обнаруживает короткие идентичные фрагменты кода, несмотря на переименование идентификаторов.

Пример 3 демонстрирует обнаружение повторяющихся паттернов, заданных определенным чередованием идентификаторов и круглых скобок. Очевидно, что такая идентификация сходных фрагментов является слишком грубой для практического применения. Кроме того, использованная разметка не позволяет наглядно оценить взаимное соответствие фрагментов кода.

В примере 4 показано обнаружение формально сходных фрагментов, имеющих, однако, различную семантику. На этом листинге представлены тексты процедуры для вычисления факториала с мемоизацией результата вычислений и процедуры лексического анализа выражений

на языке Scheme. Несмотря на то что в различных текстах сходные небольшие фрагменты следуют в разном порядке, при одноцветной разметке они воспринимаются как один большой общий фрагмент. Таким образом, этот листинг можно рассматривать как пример ложно положительного результата обнаружения сходства. Пример показывает, что целесообразно классифицировать и сравнивать различные ключевые слова как разные токены. Кроме того, в таких случаях существенную помощь при визуальной оценке результата сравнения может оказать кодирование непрерывных подпоследовательностей цветом.

Исходя из изложенного, можно сделать вывод о том, что алгоритм в большей степени пригоден для сравнения двух исходных текстов с уже обнаруженным сходством, чем для поиска похожих программ в некоторой выборке.

Заключение. В ходе работы был предложен алгоритм сравнения исходных текстов программ путем выравнивания последовательностей токенов и на ряде примеров показана его работоспособность. Дополнительно в статье приведено описание алгоритма в виде набора рекурсивных формул. Представление в виде таких формул правил заполнения таблицы традиционно используется для описания алгоритмов выравнивания последовательностей, основанных на динамическом программировании. В данной статье в виде формул представлены также правила поиска пути по этой таблице. Последнее удобно использовать для реализации алгоритма в рамках парадигмы функционального программирования.

Вместе с тем указанный алгоритм вряд ли готов к немедленному практическому применению. Так, для поиска не полной идентичности, а именно сходства кодов целесообразно применить матрицы замен, аналогичные используемым в биоинформатике при выравнивании последовательностей белков. Для каждого языка программирования, вероятно, потребуется разработать свою специфичную матрицу замен. Возможно, сравнению также должны подлежать значения констант. Параметры алгоритма (функция оценки сходства, штраф за делецию и само разрешение делеций, пороговое значение сходства) должны быть адаптированы к этим матрицам.

Остаются открытыми вопросы о пригодности данного алгоритма для поиска похожих программ в некоторой выборке и о статистической оценке значимости сходства, хотя потенциально эта возможность существует, так как в биоинформатике такая оценка успешно применяется. Далее, потребуется количественно охарактеризовать предложенный метод как бинарный классификатор (например, его способность отличать «списанное» решение от самостоятельного) для исходных кодов различной длины и для различных языков программирования.

До получения ответов на перечисленные вопросы практическое применение выравнивания для сравнения исходных кодов программ следует считать преждевременным. Однако получение ответов на эти вопросы не должно вызвать принципиальных затруднений.

Перспектива применения алгоритмов выравнивания последовательностей, на наш взгляд, привлекательна не только для выявления заимствований фрагментов кода, но и для выявления идиом, применяемых при написании программ на различных языках программирования. Выявление достаточно больших повторяющихся фрагментов позволит разработать расширения этих языков для сокращенной записи сравнительно больших и часто применяемых конструкций, что, в свою очередь, позволит увеличить степень повторного использования кода, краткость и выразительность исходных текстов программ.

Автор выражает благодарность заместителю заведующего кафедрой «Теоретическая информатика и компьютерные технологии» МГТУ им. Н.Э. Баумана С.Ю. Скоробогатову за совместное обсуждение задачи, рассмотренной в данной статье.

ЛИТЕРАТУРА

[1] Burrows S., Tahaghoghi S.M.M., Zobel J. Efficient plagiarism detection for large code repositories. *Softw. Pract. Exper*, 2007, no. 37(2), pp. 151–175.

[2] MOSS (Measure of Software Similarity). URL: http:// theoy.stanford.edu/~aiken/moss/ (дата обращения 02.10.2014)

[3] Agrawal A., Huang X. Pairwise statistical significance of local sequence alignment using sequence-specific and position-specific substitution matrices. *IEEE/ACM Trans Comput Biol Bioinform*, 2011, no. 8(1), pp. 194–205.

[4] Lewis J., Ossowski S., Hicks J., Errami M., Garner H.R. Text similarity: An alternative way to search MEDLINE. *Bioinformatics*, 2006, no. 22 (18), pp. 2298–2304.

[5] Durbin R., Eddy S.R., Krogh A., Mitchison G. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1998, 366 p.

Статья поступила в редакцию 03.10.2014

Ссылку на эту статью просим оформлять следующим образом:

Дубанов А.В. Сравнение исходных текстов программ путем выравнивания последовательностей токенов. *Инженерный журнал: наука и инновации*, 2014, вып. 9.

URL: http://engjournal.ru/catalog/it/hidden/1318.html

Дубанов Александр Вячеславович родился в 1975 г., окончил Московскую медицинскую академию им. И.М. Сеченова в 1998 г. Канд. биол. наук, доцент кафедры «Теоретическая информатика и компьютерные технологии» МГТУ им. Н.Э. Баумана. Автор 20 научных работ в области применения вычислительных методов и разработки программного обеспечения для медико-биологических исследований. e-mail: qrcs@mail.ru.

The comparison of program sources using the sequence alignment of tokens

© A.V. Dubanov

Bauman Moscow State Technical University, Moscow, 105005, Russia

Borrowing detection is a very actual problem now. In this work, one of the known algorithms of the biopolymer sequence alignment was modified to make it possible to compare program sources and detect similar snippets in these sources. The input data of this algorithm are the source codes treated as the sequences of symbols. The set of lexical domains correspond to the alphabet of symbols making up these sequences. The algorithm was implemented and demonstrated with some code fragments written in Scheme language. The perspectives and restrictions of the algorithm application are also discussed.

Keywords: *borrowing of code sequence alignment, the largest common subsequence, dynamic programming, lexical analysis, functional programming.*

REFERENCES

[1] Burrows S., Tahaghoghi S.M.M., Zobel J. Efficient plagiarism detection for large code repositories. *Softw. Pract. Exper*, 2007, no. 37(2), 151–175.

[2] MOSS (Measure of Software Similarity). Available at: http://theoy.stanford.edu/~aiken/moss/ (accessed on 02.10.2014).

[3] Agrawal A., Huang X. Pairwise statistical significance of local sequence alignment using sequence-specific and position-specific substitution matrices. *IEEE/ACM Trans. Comput. Biol. Bioinform.*, 2011, no. 8(1), pp. 194–205.

[4] Lewis J., Ossowski S., Hicks J., Errami M., Garner H.R. Text similarity: An alternative way to search MEDLINE. *Bioinformatics*, 2006, no. 22 (18), pp. 2298–2304.

[5] Durbin R., Eddy S.R., Krogh A., Mitchison G. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1998, 336 p.

Dubanov A.V., Ph. D., assoc. professor of the Computer Science and Technologies Department of Bauman Moscow State Technical University. Specializes in the application of the computational methods and software development for biomedical research. He is the author of 20 publications. e-mail: qrcs@mail.ru.