2. Кислов К. В., Гравиров В .В. Нейросетевой метод детектирования землетрясения при повышенном уровне шума // Электронный журнал "Исследовано в России". 2010. № 070. С. 824–836. http://zhurnal.ape.relarn.ru/articles/2010/070.pdf.

3. Яневиц Р. Б., Беспечная Л. Ю., Беспечный В. Н. Построение сейсмологических моделей осадочных отложений на основании данных сейсморазведки и ПГИС // Геология, геофизика и разработка нефтяных и газовых месторождений. 2006. № 10. С. 105–114.

4. Fukushima K. Neocognitron: a hierarchical neural network capable of visual pattern recognition // Neural Networks. 1988. 1 (2). P. 119–130.

5. Fukushima K. Neocognitron for handwritten digit recognition. // Neurocomputing. 2003. Vol. 51. P. 161–180. http://visiome.neuroinf.jp/modules/xoonips/detail.php?item_id=375.

6. Садыхов Р. Х., Ваткин М. Е. Алгоритм обучения нейронной сети «неокогнитрон» для распознавания рукописных символов // Изв. НАН Беларуси. Сер. физ.-техн. наук. 2002. № 3. С. 81–86.

7. Федяев О. И., Махно Ю. С. Программный эмулятор нейросети типа неокогнитрон для распознавания графических образов // Научные работы Донецкого национального технического университета. Сер. «Информатика, кибернетика и вычислительная техника». 2008. № 9 (132). С. 265–269.

8. Ивашко А. Г., Кугаевских А. В. Возможность применения ИНС «Неокогнитрон» для распознавания древнеегипетских иероглифов // Вестник Тюменского государственного университета. 2009. № 6. С. 209–214.

9. Duffner S. Face Image Analysis With Convolutional Neural Networks. // PhD Thesis. Albert-Ludwigs-Universität Freiburg im Breisgau, Germany, 2007.

10. Копыльцов А. В., Сорокин И. В. Вейвлет-анализ структурной энтропии файлов // Изв. РГПУ им. А. И. Герцена. 2011. № 138. С. 7–15.

I. V. Sorokin, A. V. Kopyltsov

NEOCOGNITRON FOR MALWARE RECOGNITION

The algorithm for using neocognitron for compare binary data is developed. To identify specific areas of the analyzed files, in this paper we describe the stages of training and testing neural network. It can help to detect malicious programs.

Pattern recognition, neural networks, neocognitron, malware

УДК 004.421.6

А. Р. Лисс, И. А. Андрианов

ИСПОЛЬЗОВАНИЕ РАЗРЕЖЕННЫХ СУФФИКСНЫХ ДЕРЕВЬЕВ ДЛЯ ЗАДАЧ ОБРАБОТКИ ТЕКСТОВ

Потребность в эффективных алгоритмах обработки строк возникает во многих практических задачах. Одной из классических задач в этой области является задача поиска наибольшей общей подстроки. Для ее решения предлагается использовать разреженные суффиксные деревья, приводится описание алгоритма. Такой подход позволит обрабатывать данные существенно большего размера, чем в случае использования обычных суффиксных деревьев. Также в статье рассматривается решение задачи поиска максимальной симметричной подстроки.

Разреженные суффиксные деревья, поиск по сходству, алгоритм Укконена

В настоящее время наблюдается значительный интерес к алгоритмам обработки строк и последовательностей. Это объясняется постоянно растущей потребностью в эффективном решении такого рода задач в связи с развитием областей их применения – распознавание образов, вычислительная молекулярная биология, информационно-поисковые системы и др. В большинстве строковых алгоритмов используется общий подход, который заключается в проведении предварительной обработки (предобработки) одной или нескольких исходных строк – т. е. построении индекса.

Одним из наиболее общих способов предобработки является построение для строки суффиксного дерева (СД). Суффиксные деревья могут применяться для эффективного поиска подстрок и более сложных шаблонов с подстановочными символами (вплоть до полноценных регулярных выражений), для поиска дубликатов в программном коде [1], для нечеткого поиска, поиска часто встречающихся подстрок, используются в реализации некоторых алгоритмов сжатия данных и др.

Суффиксным деревом T для m-символьной строки S называется дерево с корнем, имеющее ровно m листьев, занумерованных от 1 до m. Каждая внутренняя вершина, отличная от корня, имеет не меньше двух детей, а каждая дуга помечена дуговой меткой – парой чисел (i, j), задающих подстроку $S[i..j]$ строки S (говорят, что дуга помечена этой подстрокой). Никакие две дуги, выходящие из одной и той же вершины, не могут начинаться с одного и того же символа. Для каждого листа i конкатенация меток на пути к нему от корня составляет в точности суффикс S, начинающийся в позиции i [2, 5.2]. Пример СД для строки “banana$” показан на рис. 1.

Штрихами показаны суффиксные связи – специальные дуги, создающиеся в процессе работы алгоритмов построения СД и полезные также для решения некоторых задач. Суффиксной связью называется дуга из узла $\overline{vw}$ в узел $\overline{w}$, где v – непустая строка минимальной длины, такая, что узел $\overline{w}$ существует в дереве. Если такой строки не существует, то суффиксная связь идет в корень.

Существует несколько алгоритмов построения суффиксного дерева за линейное время, для большинства задач предпочтительным оказывается метод Укконена как наиболее простой в понимании и реализации. Построив дерево за линейное время, можно использовать его для решения целого ряда задач [2].

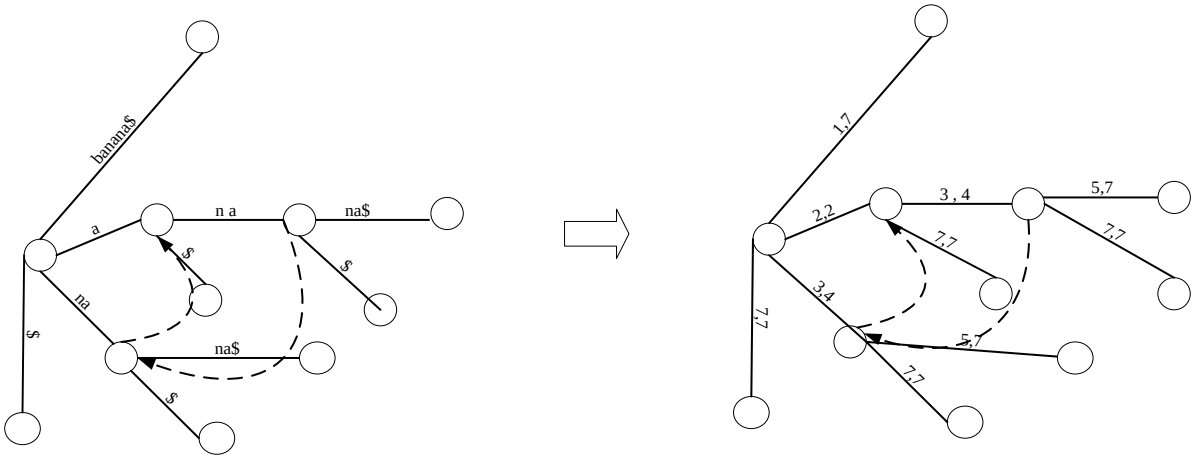

Рис. 1

Несмотря на несомненные достоинства, суффиксные деревья имеют серьезный недостаток – значительные требования к оперативной памяти. Хотя объем памяти, занимаемый суффиксным деревом, линейно зависит от длины исходной строки, коэффициент пропорциональности довольно высок – даже в лучших реализациях размер суффиксного дерева превышает размер исходной строки более чем в 10 раз [3]. В результате на типовой

вычислительной машине при еще достаточно скромных размерах входных данных (десятки мегабайт) суффиксное дерево перестает помещаться в физическую память, и введение в действие файла подкачки резко снижает скорость работы программы.

Один из путей решения данной проблемы заключается в том, что с целью уменьшения памяти, занимаемой деревом, можно хранить в нем не каждый суффикс исходной строки, а только некоторое их подмножество – в результате получается разреженное суффиксное дерево (РСД). Важным частным случаем является РСД, которое включает в себя лишь те суффиксы, позиции начала которых отстоят друг от друга на одинаковые расстояния (например, суффиксы, начинающиеся в 1, 3, 5, 7-й и т. д. позициях). Пример для строки “ьanana$” показан на рис. 2.

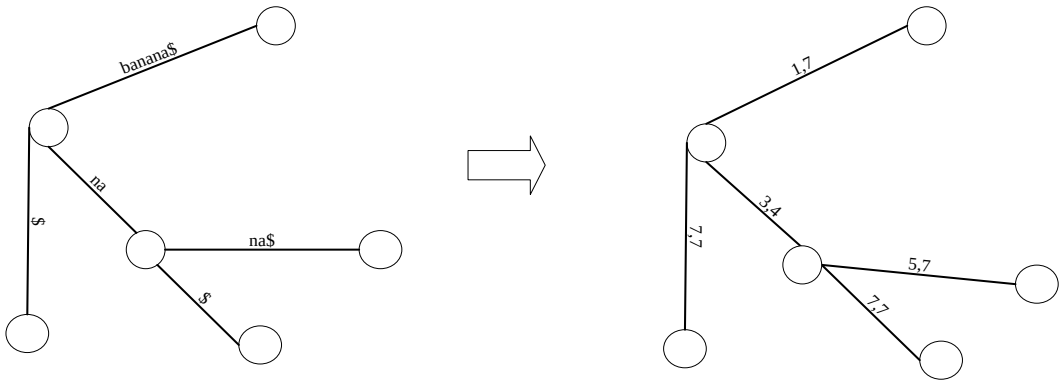

Рис. 2

Если дерево содержит каждый *k*-й суффикс, то при этом объем памяти уменьшается примерно в *k* раз. Полученную структуру Каркайнен и Укконен назвали равномерно разреженным суффиксным деревом (evenly spaced sparse suffix tree, РРСД) степени *k* [4]. В работе [4] показано, что можно модифицировать алгоритм Укконена так, чтобы он строил РРСД и при этом временная сложность алгоритма оставалась прежней.

Как следует из [4], РРСД введены авторами специально для решения задачи о поиске подстрок в заданной строке. Однако при этом время поиска экспоненциально зависит от *k*, что сильно ограничивает степень разреженности. Можно сделать вывод, что в работе [4] РРСД рассматриваются как эвристика для ускорения поиска при сравнительно небольших размерностях входных данных, когда полное СД уже не помещается в память, но еще не требуются действительно большие *k*.

Авторы рассматривали возможность применения данных структур к другим задачам обработки текстов на примере задачи поиска наибольшей общей подстроки двух строк. В результате получен алгоритм с временем работы $O(\min(mk, nk^2))$ и памятью $O(m/k + n)$, позволяющий решать задачи существенно большей размерности, чем описанные в литературе методы [2].

Сформулируем задачу о наибольшей общей подстроке. Даны две строки, требуется найти подстроку максимальной длины, присутствующую в обеих строках. Рассмотрим для начала алгоритм, предложенный в [2, 7.8]. Пусть даны две строки P и T длины m и n соответственно, $n \geq m$. Вначале строится суффиксное дерево S для P за время $O(m)$, используя алгоритм Укконена.

Далее используем S для построения специальной структуры данных, называемой статистикой совпадений [2]. Статистикой совпадений для строки T относительно строки P будет массив ms[1..|T|], где ms[i] – длина наибольшей общей подстроки T, начинающейся в позиции i и встречающейся также где-то в P.

Заполнение ms производится следующим образом. Вначале определяется ms[1] при движении от корня СД, построенного для P, в соответствии с символами строки T до обнаружения несовпадения. Длина совпавшей части T[1..j] даст ms[1]. Чтобы эффективно найти ms[2], используется тот факт, что позиция, соответствующая подстроке T[2..j], заведомо имеется в дереве, и для перехода к ней необходимо выполнить переход по суффиксной связи, после чего выполнить так называемую канонизацию (см [2]). Аналогично определяется ms[3] и т. д. Если СД для P уже построено, то сложность вычисления статистики совпадений составляет $O(|T|)$.

Таким образом, можно подняться до ближайшей внутренней вершины дерева v, перейти по суффиксной связи в вершину $s(v)$ и начинать поиск ms[2] с вершины $s(v)$, а не из корня. Как показано в [2], заполнение ms производится также за $O(m)$. Наконец, осталось найти максимальное значение в массиве ms – длину наибольшей общей подстроки.

Для более эффективного использования памяти предлагается следующее решение. Выберем k так, чтобы полученное РРСД занимало почти всю доступную физическую память. Введем еще одну характеристику РРСД – позицию первого суффикса t. Тогда в РРСД будут входить суффиксы, начинающиеся в позициях t, $t + k$, $t + 2k$ и т. д.

Переберем все значения t от 1 до k и для каждого из них построим разреженное суффиксное дерево с соответствующими параметрами. Введем еще одну величину t_1 – индекс первого элемента в ms, с которого начинаем корректировать значения в ms (t_1 также изменяется от 1 до $k - 1$). Построив РРСД(t, k), переберем все значения t_1 и для каждого из них выполним ранее описанную процедуру со следующими модификациями. Во-первых, каждый элемент массива ms будет обработан k раз, а не один. При этом будем не просто заносить значения в ms, а корректировать уже имеющиеся, заменяя текущее значение на новое в том случае, если оно больше текущего. Во-вторых, от элемента ms[i] будем переходить к элементу ms[$i + k$] и, соответственно, в РРСД будем переходить от подстроки P[u..v] к подстроке P[$u + k$..v].

Корректность алгоритма можно показать следующим образом. Пусть наибольшая общая подстрока, начинающаяся в T с позиции i, в строке P начинается с позиции j. Необходимо, чтобы на каком-нибудь шаге алгоритма при вычислении ms[i] проверялось совпадение с суффиксом P[j..m], находящимся в РРСД. Поскольку для данного РРСД рассматриваются все значения k от 1 до $k - 1$, а для каждого из них с шагом k доходим до конца массива, то тем самым обработаем все его элементы, включая ms[i], куда и будет записана длина совпавшей части.

Из приведенного описания алгоритма следует, что его временная сложность составляет $O(\min(mk, nk^2))$, пространственная сложность равна $O(m/k + n)$. В последнем выражении предполагаемая константа пропорциональности при m значительно превышает кон-

станту при n, поэтому в случае близких по длине строк получим выигрыш в памяти почти в k раз. В случае сильно отличающихся по длине строк все зависит от того, насколько велика максимальная из них. Если взять ее в качестве T и массив ms вместе с СД для P поместится в оперативную память, то проще использовать ранее описанный алгоритм. Если нет, то выгоднее использовать предложенный алгоритм, строя массив ms для меньшей строки и разреженные суффиксные деревья – для большей.

Сравнив временную сложность предложенного алгоритма с ранее описанным, видим, что она увеличивается в k^2 раз. Но при этом не следует забывать, что время выполнения одной операции чтения/записи для оперативной памяти и диска отличается на несколько порядков, и для небольших k увеличение числа операций с избытком компенсируется уменьшением их стоимости. Эксперименты показали, что на достаточно уже старом персональном компьютере с 512 Мбайт оперативной памяти данный способ позволяет решать задачи размерности порядка гигабайта, тогда как способ, описанный в [2, 7.8] – менее 100 Мбайт.

Проанализируем полученный алгоритм и попытаемся обобщить его, чтобы получить некоторые рекомендации для эффективного по памяти решения и ряда других задач обработки строк. Основная идея заключается в использовании общего принципа «разделяй и властвуй». Необходимо разбить исходную задачу на более мелкие подзадачи таким образом, чтобы каждую из них можно было решать по отдельности, после чего на основе их решений восстановить решение исходной задачи. Применительно к данному случаю критерием независимости подзадач является возможность разбиения множества суффиксов исходной строки на непересекающиеся подмножества (или с заведомо ограниченной степенью пересечения) так, чтобы подзадача на этом подмножестве имела смысл, а ее решение могло бы быть использовано далее. Отметим, что в общем случае не требуется равномерность дерева, поскольку к настоящему времени разработаны способы построения неравномерно разреженного суффиксного дерева за линейное время. Основываясь на этих соображениях, удалось получить эффективное по памяти решение еще одной задачи – поиска наибольшей подстроки в строке, являющейся палиндромом, за время $O(nk)$, используя РРСД разреженности k. Рассмотрим ее более подробно.

Задача о поиске наибольшего общего палиндрома звучит так. Дана строка, требуется найти в ней подстроку максимальной длины, которая читается одинаково как направо, так и налево. На первый взгляд данная задача кажется несколько далекой от практики, но на самом деле это не совсем так (достаточно заметить, что палиндром – просто случай симметрии, которая встречается повсеместно). В частности, эта задача возникает в вычислительной биологии при анализе ДНК.

Рассмотрим для начала алгоритм поиска наибольшего палиндрома, предложенный в [2]. Вкратце алгоритм выглядит так. Берем исходную строку, конкатенируем ее с перевернутой ее копией (перед этим дописав к концам строк по служебному символу) и над результатом строим суффиксное дерево.

Далее потребуется уметь решать следующую подзадачу (известную как задача о наименьшем общем предке): даны 2 узла в суффиксном дереве. Требуется за константное время найти их наименьшего общего предка, разрешается выполнить предварительную обработку дерева, потратив на это время порядка O (число узлов дерева).

Имеется несколько решений данной задачи, наиболее понятным и легко реализуемым считается алгоритм Шибера–Вишкина [2, гл. 8]. Имея алгоритм поиска наименьшего общего предка, легко можно ответить и на вопросы о наибольшем общем продолжении – даны две позиции i и j в строке (над которой построено суффиксное дерево). Требуется за константное время определить длину наибольшего общего префикса строк, начинающихся с позиций i и j. Для этого достаточно взять 2 листа, соответствующих префиксам, начинающимся с i и j, и найти их наименьшего общего предка. На рис. 3 показан пример поиска палиндрома в строке 'bananas'.

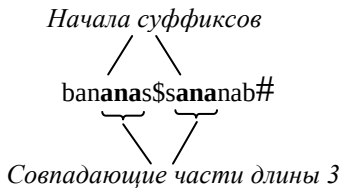

Рис. 3

После этого просто перебираем все возможные варианты позиции середины палиндрома, берем эту позицию в исходной и перевернутой строках (дописанной справа к исходной) и находим наибольшее общее продолжение. В ходе цикла храним лучшую длину и позицию, которая по окончании цикла и будет ответом. Здесь есть момент в отличии палиндромов четной и нечетной длин, который требует аккуратного программирования.

Таким образом, описанный алгоритм выполняет поиск за линейное время, но предъявляет весьма значительные требования к памяти как под собственно суффиксное дерево, так и под структуры данных, используемые в алгоритме Шибера–Вишкина. Предлагается следующий подход к решению данной проблемы.

Выберем степень неплотности k таким образом, чтобы полученное дерево вместе со структурами данных для алгоритма Шибера–Вишкина занимало почти весь объем доступной физической памяти. Аналогично предыдущей задаче о наибольшей общей подстроке введем смещение t ($1 \le t \le k$), которое определяет позицию первого суффикса. Переберем все значения t от 1 до k и для каждого из них создадим строку так, чтобы в построенное неплотное суффиксное дерево для каждого суффикса исходной строки вошел дополняющий его суффикс перевернутой строки. Для этого при конкатенации строк между ними придется вставить несколько служебных символов (не более чем $2k$).

Теперь осталось для каждого дерева применить алгоритм Шибера–Вишкина и найти палиндром максимальной длины, после чего выбрать максимальный уже среди них.

Из приведенного описания алгоритма сразу следует, что его временная сложность составляет $O(kn)$, пространственная – $O(n/k)$.

Таким образом, можно сделать вывод, что разреженные суффиксные деревья применимы для более широкого круга задач, чем та, для которой они были предложены, и позволяют обрабатывать такие объемы данных, которые было бы затруднительно обработать за приемлемое время имеющимися методами.

СПИСОК ЛИТЕРАТУРЫ

1. Лисс А. Р., Андрианов И. А. Анализ и разработка методов поиска дубликатов в программном коде // Изв. СПбГЭТУ «ЛЭТИ». 2010. Вып. 7. С. 55–61.

2. Гасфилд Д. Строки, деревья и последовательности в алгоритмах: Информатика и вычислительная биология / Пер. с англ. СПб.: Невский Диалект; БХВ-Петербург, 2003.

3. Giegerich R., Kurtz S., Stoye J. Efficient implementation of lazy suffix trees // In the Proc. of 3rd Intl. Workshop on Algorithm Engineering. Lecture Notes in Computer Science. 1999. Vol. 1668. P. 30–42.

4. Karkkainen J., Ukkonen E. Sparse suffix trees // In Proc. 2nd Ann. Intl. Conf. on Computing and Combinatorics (COCOON'96), LNCS 1090, 1996.

A. R. Liss, I. A. Andrianov

THE USE OF SPARSE SUFFIX TREE FOR WORD PROCESSING TASKS

Efficient algorithms for processing of strings and sequences required in many practical problems. Therefore, this area is being studied and developed. Find the longest common substring is one of the classical problems in the strings processing. To solve this problem, we propose to use sparse suffix trees and give a description of the algorithm. This approach allows to process data much larger than with conventional suffix trees. Also in this article, we consider the use of sparse suffix trees to find the longest symmetric substring.

Sparse suffix trees, similarity search, Ukkonen algorithm