УДК 004.051

ЭФФЕКТИВНАЯ СТРУКТУРА ХРАНЕНИЯ СЛОВАРЯ СТРОКОВЫХ КЛЮЧЕЙ С АССОЦИИРОВАННЫМИ ЗНАЧЕНИЯМИ

В.П. Гергель, Д.С. Скатов

Нижегородский госуниверситет им. Н.И. Лобачевского

danskatov@gmail.com

Поступила в редакцию 10.09.2012

Предложен способ реализации словаря на основе минимальных конечных автоматов, где ключу ставится в соответствие набор значений. Показаны его преимущества в сравнении с доступными решениями в виде СУБД и классов стандартной библиотеки C++: улучшение по времени поиска составляет от 20-40 раз до порядков. При этом существенно сокращается требуемая для хранения память.

Ключевые слова: визуальные словарь, эффективные структуры хранения, префиксное дерево, конечный автомат, вычислительная лингвистика, машинная морфология.

Введение

В статье рассматриваются встраиваемые реализации словарей, позволяющие по заданному строковому ключу получать множество ассоциированных значений. Под встраиваемыми понимаются реализации, не требующие сервера приложений (в сравнении с большинством реляционных СУБД).

Подобные структуры хранения востребованы, в первую очередь, в приложениях автоматической обработки текста. Так, при выполнении синтаксического разбора предложения (естественного языка) используется модуль словарной морфологии, по заданному слову определяющий множество его грамматических значений. Например, слову *«для»* соответствует два значения: от *«длить* (*ед.ч., несов.вид, непереходный, 2-е лицо*) и *«для»* (*предлог*). Этот же модуль требуется в информационно-поисковых системах для приведения слов из документов и запросов к начальной форме. К реализации модуля предъявляются высокие требования по скорости, поскольку он поставляет входные данные для более трудоёмких этапов обработки текста.

В статье рассмотрены четыре встраиваемые реализации словарей: 1) авторская реализация, основанная на минимальном конечном автомате и алгоритмах, адаптированных из [1], называемая далее *DawgMap*; 2) *std::map* из стандартной библиотеки C++ [4]; реализации на основе 3) БД *SQLite* [5] и 4) *Oracle BerkeleyDB* [6]. Сравнение выполнено на основе реальных лингвистических данных: использовались 1) набор записей машинной морфологии русского языка и 2) набор запросов к российской поисковой системе, несколько миллионов ключей в каждом наборе. Показаны преимущества авторской реализации в большинстве рассмотренных контекстов.

В [7] анализируется ряд свободно доступных словарей для хранения ключей (иногда – со значениями), выполненных отдельными исследователями за рамками стандартной библиотеки C++ и встраиваемых СУБД. Показано, что эти реализации непригодны для применения в коммерческом ПО (например, по причине некорректной работы при большом количестве ключей) или/и не обладают достаточным функционалом.

Далее используются следующие обозначения: A – конечный алфавит символов, $key \in A^+$ – добавляемый ключ, $val(key) \subseteq Vals \subseteq A^+$ – набор значений ключа key, $Vals$ – множество допустимых значений, $N = |Vals|$, $M = |A|$. Говорится также, что величина N определяет степень разнообразия значений. Количество вершин конечного автомата называется его мощностью.

1. Словарь строк как конечный автомат

Известно, что языку, образованному конечным множеством строк, можно поставить в соответствие ациклический конечный автомат, распознающий этот язык [3] (также в литературе: *DAWG – directed acyclic word graph*).

Префиксные деревья. Простейшим автоматом, распознающим язык из конечного числа строк, является *префиксное дерево* [3]. Для него

Кнутом Д. получена [8] оценка мощности как функции числа добавленных строк: если ключи равномерно распределены в алфавите мощности M, имеют максимальную длину s, и уже добавлено n таких ключей, то число состояний дерева в среднем есть величина $O(n/(s\ln M) = O(n)$. Для лингвистических данных этот закон подтверждается – префиксное дерево, образуемое 3 млн русских словоформ, удовлетворяет данной асимптотике. Насыщение мощности автомата произойдёт, когда добавлены все возможные строки такого типа, т.е.

$$1 + M + M^2 + ... + M^s = M\frac{M^s - 1}{M - 1}$$

строк. При M = 25, s = 10 это количество составляет порядка 10^{14}, так что для реального текстового материала насыщение никогда не будет достигнуто – в русском языке число известных словоформ не превышает 5 млн.

В префиксных деревьях цепочки вершин ближе к листьям часто имеют вырожденный вид, представляя собой пути из узлов, каждый из которых имеет ровно одно входящее и одно исходящее ребро. Это характерно для лингвистических данных, где многие строки имеют одинаковые окончания. Сворачивая каждый путь в одно ребро, помеченное цепочкой соответствующих символов, можно получить так называемое *patricia-дерево* (*radix-tree, compact prefix tree*).

Минимальные конечные автоматы. Негативный опыт использования различного рода деревьев, о котором сообщают исследователи, обусловлен их сильно ветвистой природой и большим объёмом. Это негативно сказывается на производительности кэша ЦП, а значит, и всей структуры хранения в целом.

Известно, что для любого детерминированного конечного автомата существует эквивалентный ему автомат наименьшей мощности. Следовательно, для конечного множества строк наибольшим по мощности автоматом является префиксное дерево, наименьшим – минимальный конечный автомат (*mafsa – minimal acyclic finite state automata*).

Способы получения минимальных представлений конечных словарей включают: 1) получение автоматов, по мощности занимающих промежуточное положение между префиксным деревом и минимальным автоматом; 2) построение префиксного дерева и его последующая минимизация; 3) минимизация промежуточного автомата после добавления очередной строки; 4) минимизация автомата «на лету», т.е. уже в процессе добавления строки.

Эти подходы являются компромиссами по времени, памяти и сложности реализации. Основная проблема возникает на этапе построения автомата: в некоторых подходах для получения минимальных автоматов по миллионам строк необходимы десятки гигабайт памяти для промежуточных представлений. В данной статье используется последний из упомянутых подходов, описанный, например, в [1]. В нём после каждой вновь добавленной строки результирующий автомат уже является минимальным. Подход достаточно сложен в реализации, но для словарей из миллионов строк требует на этапе построения нескольких десятков мегабайт памяти и нескольких минут (3 GHz ЦП, 8 Гб ОЗУ).

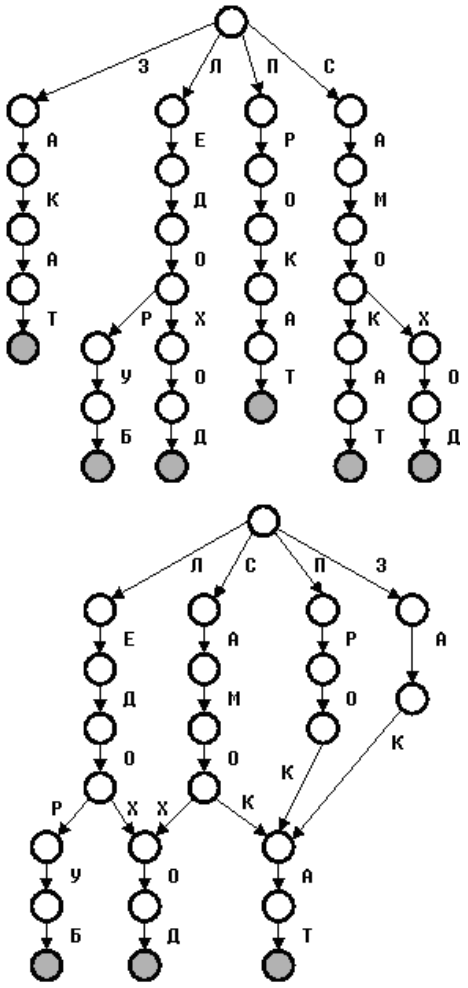

Рис. 1. Префиксное дерево (сверху, 32 вершины) и эквивалентный минимальный автомат (снизу, 20 вершин) для множества строк {«закат», «ледоруб», «ледоход», «прокат», «самокат», «самоход»}. Серым отмечены финальные состояния

Предельная мощность минимального конечного автомата после добавления всех строк длины, не более s, в алфавите мощности M не зависит от количества добавленных строк и составляет ровно s + 1 вершину. Действительно,

пусть все такие строки уже добавлены в автомат. Тогда в нём из вершины с номером i будет исходить ровно M дуг в вершину с номером $i + 1$, $i = 1, \ldots, s$, и все вершины будут финальными. Таким образом, можно рассчитывать, что стабилизация мощности автомата будет достигнута, в сравнении с префиксным деревом, при существенно меньшем количестве добавленных строк. Далее в работе выполнимость этого свойства показана на реальных лингвистических данных.

Успешность применения минимальных автоматов в обработке естественных языков объясняется тем, что общими в их строках часто являются не только префиксы, но и суффиксы, инфиксы (см. рис. 1).

2. Эффективная структура хранения словаря

2.1. Представление словаря с ассоциированными значениями ключей

Сохранение значений, ассоциированных с ключами. Конечные автоматы предназначены для бинарного различения входных строк: можно получить ответ на вопрос, принадлежит ли входная строка языку данного автомата. На практике со строками необходимо связать значения, которые могут в общем случае являться произвольными последовательностями байтов, а не только целыми числами фиксированной разрядности, как это предложено в некоторых реализациях. Для префиксного дерева и его производных очевидным решением является сохранение значения в листовом узле. В случае, когда размер значения достаточно велик, а различных значений много меньше ключей, разумно сохранять не само значение, а ссылку на него, при хранении значения в отдельной области памяти.

Пусть сохраняемому ключу можно присвоить любое из N различных значений. При минимизации конечного автомата, адаптированного для сохранения таких ключей и их значений, оказывается, что нужно различать не обычные и финальные состояния, а как минимум $N + 1$ класс состояний, из которых N классов соответствуют финальным состояниям с одним из N различных значений, а $(N + 1)$-й класс – это прочие состояния. Кроме того, следует позаботиться о возможности сохранения нескольких значений для одного ключа, так что размер узла должен быть динамическим по количеству значений.

Конечно-автоматный преобразователь как словарь. Другой подход к сохранению значений состоит в том, что автомат для представления словаря рассматривается как частный случай *конечно-автоматного преобразователя* (*finite-state trancducer*) [3], который по входной строке не только отвечает на вопрос о принадлежности его языку, но и порождает некоторую выходную строку в качестве отклика. Чтобы организовать такое поведение, получая значение как отклик на ключ, словарь вместо строк *key* будет наполняться строками вида key#value, где # – специальный разделительный символ, отделяющий ключ от значения.

В предложенном подходе поиск **всех** ассоциированных с ключом значений состоит в том, чтобы в автомате найти путь, целиком проходимый буквами *key*, и посмотреть, имеет ли достижимая по *key* вершина q_k ребро, помеченное #. Если да, тогда считается, что *key* есть в словаре, а множество его значений целиком определено автоматом со стартовой вершиной q_k. Подход применим, если символ # ранее не встречался в *key*. Обычно в качестве # используется символ с кодом '\0', в этом случае означенное условие выполнено.

Структура хранения DawgMap. Предложенная в работе реализация словаря строк (*DawgMap*) выполнена в виде библиотеки на C++. Помимо автоматической поддержки любого числа значений, ассоциированных с ключом (в том числе нулевого), *DawgMap* имеет следующие преимущества.

1. *Автоматическое определение общих значений и их наборов.* Для заданных *key* и $val(key) = \{v_1, \ldots, v_n\}$ на этапе построения автомата можно формально пополнить его строками $key\#v_1, \ldots, key\#v_n$. Не надо заботиться о том, чтобы одинаковые значения (и их группы) у разных ключей хранились в единственном экземпляре – это будет достигнуто автоматически в процессе построения автомата.

2. *Сжатие значений.* Часто оказывается, что не только ключи, но и значения делят общие префиксы, суффиксы, инфиксы. Алгоритм построения автомата поступит с такими значениями так же, как и с общими частями ключей – по мере возможности общие фрагменты значений будут объединены.

3. *Иерархии значений.* Иногда необходимо организовать вложенность значений, когда одно значение ключа имеет несколько подзначений: *key#value@subvalue*. Это можно организовать с помощью *DawgMap*, подбирая соответствующие разделительные символы для значений. Вместо них можно использовать явное задание длины идущего далее значения в его первом символе, но степень сжатия при этом обычно

снижается, т.к. некоторые префиксы перестают быть совпадающими у разных ключей.

4. *Типизация значений.* Используя разделительные символы, можно связывать с ключами значения разных типов. Так, с символом '\0' будет связан первый тип значений, с '\1' – второй и т.д. Получение значений конкретного типа сводится к обнаружению у вершины исходящих рёбер, помеченных разделительным символом этого типа.

2.2. Способы хранения состояний конечного автомата. Вершина графа *DawgMap* представляет собой набор переходов с указателями на вершины, в которые они ведут. Набор отсортирован в лексикографически возрастающем порядке символов, помечающих дуги переходов.

Сериализация. Имеется два представления автомата: на этапе его формирования (*динамическое*) и после формирования, для повторного использования в режиме чтения (*статическое*). Процесс сохранения структуры для повторного использования называют *сериализацией*. Сериализация *DawgMap* осуществляется в бинарный файл так, чтобы загрузка структуры на чтение состояла *только* в открытии бинарного файла. Тем самым реализация по удобству чтения находится на уровне встраиваемых БД. Известны два основных способа сериализации графов: с плотным и разреженным представлением переходов. В плотном представлении хранятся только переходы, действительно имеющиеся у вершины, а каждый переход представлен одним символом и одним указателем. В разреженном – каждая вершина представлена массивом из M элементов, i-й элемент соответствует i-й в лексикографическом порядке букве алфавита, а тем буквам, у которых в данной вершине переходов нет, назначается нулевой указатель.

Плотное представление вершины. Если применить плотное представление и использовать для отыскания перехода бинарный поиск, то сложность поиска ключа длины n составит $O(n\log M)$. Однако для лингвистических данных мощности алфавитов невелики, а полную крону размером порядка M элементов имеет менее половины вершин автомата. Можно предусмотреть адаптивную стратегию поиска, когда в вершинах с малым размером кроны он выполняется прямым перебором символов. Более того, на практике оказывается, что размер структуры имеет решающее значение перед вычислительной сложностью алгоритма поиска: основное время затрачивается в операциях перехода между состояниями, а не в поиске точки перехода.

Разреженное представление вершины. Таким образом, разреженное представление теоретически позволяет ускорить поиск, но требует больше места для хранения, что негативно сказывается на производительности структуры. В [2] предложен способ хранения графа с разреженными вершинами, учитывающий то обстоятельство, что у многих вершин крона неплотная. У таких вершин в массиве указателей между реально существующими переходами имеются пустые пространства, которые можно заполнить данными других вершин (рис. 2).

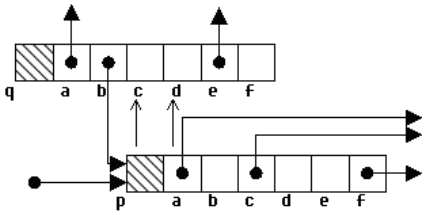

Рис. 2. Перекрытие хранимых состояний. Для алфавита $A = \{a,b,c,d,e,f\}$ вершина q имеет переходы только по $\{a,b,e\}$, p –только по $\{a,c,f\}$. Штрихованный квадрат содержит битовую маску. Состояние p можно сохранить не после f, а вставить его после b, тогда пропуски в q, соответствующие $\{c,d,f\}$, будут использованы эффективно

В этом виде уже нельзя приписать пустые указатели тем символам, что не имеют переходов, так что для каждой вершины дополнительно заводится битовая маска. В ней для буквы с номером $j \in \{1, \ldots, M\}$, имеющей переход, записан бит 1, для остальных – бит 0. В русском и английском алфавитах для маски достаточно 32 бит = 4 байт (все значащие буквы). Таким образом, надо получить максимально плотное по общей длине пересечение массивов с пустотами. Субоптимальное решение задачи достигается в [2] жадным алгоритмом.

Проведённые эксперименты показали, что плотное представление вершин по указанной схеме идентично по производительности разреженному, при этом приводит к структурам хранения бóльших размеров. Не дала улучшения производительности и смешанная оптимизация, когда вершины с плотной кроной хранятся в разреженном представлении и наоборот.

Два типа вершин. При сохранении лингвистических данных оказывается, что порядка 25% вершин образуют *пути* – цепочки состояний, имеющих ровно одно входящее и одно исходящее ребро. Для учёта этого факта в *DawgMap* при сериализации есть два типа вершин: раз-

вилки (вершины с двумя и более исходящими рёбрами) и пути.

2.3. Поиск и извлечение значений. Особенность *DawgMap* состоит в том, что значения в общем случае не записаны в нём в готовом виде, перед использованием их необходимо извлечь из графа (рис. 3). Поэтому при работе с *DawgMap* используются два поисковых алгоритма.

1. **Алгоритм *Follow***. Для извлечения набора значений по заданному ключу *key* необходимо сначала пройти по дугам из корневой вершины графа, следуя символам *key*. Успешным поиском является ситуация, когда просмотрены все символы *key*, а в графе было достигнуто состояние q_K с исходящей дугой, помеченной разделительным символом.

2. **Алгоритм *Fetch***. Все значения ключа *key* представлены путями, начинающимися с вершины, в которую из q_k ведёт достигнутый ранее в *Follow* переход с разделительным символом, и заканчивающимися в финальных вершинах графа. Алгоритм *Fetch* обходит все такие пути (в силу линейного упорядочения переходов по буквам – это будет сделано в лексикографическом порядке), тем самым извлекаются все значения $val(key) = \{v_1, \ldots, v_{|val(key)|}\}$.

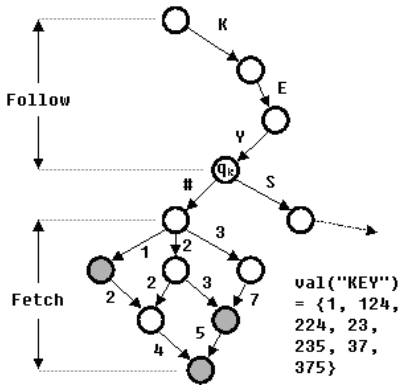

Рис 3. Алгоритмы Follow и Fetch

Переборный характер алгоритма *Fetch* приводит к вопросу о его производительности. Однако глубина его погружения ограничена длиной значений и их количеством — для лингвистических данных обычно не более десятка. На практике *Fetch* отнимает в худшем случае 4% от общего времени получения набора значений по ключу.

3. Сравнение реализаций

3.1. Способ тестирования. Пусть заданный словарь V с мощностью n ключей отсортирован в лексикографическом порядке. Для оценки объёма и производительности структур хранения выполняется следующая последовательность действий:

1) Из всего отсортированного словаря для заданного $\alpha \in (0, 100)$ выбирается порция ключей V_α объёмом α%, равномерная в следующем смысле: отобранные элементы словаря находятся на как можно более равном удалении от своих непосредственных соседей и границ набора элементов. Тем самым достигается равномерность порции по буквам алфавита. Например, для словаря $V = \{$«*a*», «*aa*», «*ab*», «*b*», «*ba*», «*bb*», «*c*»$\}$ и $\alpha = 70\%$ – может быть выбрано $V_\alpha = \{$«*a*», «*ab*», «*b*», «*bb*», «*c*»$\}$, $\alpha = 30\%$ – $V_\alpha = \{$«*a*», «*ba*»$\}$.

2) Для отобранного множества V_α строится словарь.

3) Из V_α по такой же схеме отбирается p ключей (их доля в V_α равна $p|V_\alpha|^{-1}\cdot 100\%$), по которым выполняется оценка производительности операций поиска (успешного и неуспешного). Этот набор перед поиском перемешивается. Одна серия измерений состоит в последовательном поиске всех ключей из перемешанного набора, и выполняется всего p таких серий. Замеряется общее время t выполнения всех серий. В качестве среднего времени поиска по словарю объёмом $|V_\alpha|$, образованному некоторыми строками заданного языка, полагается усреднённое время $t\cdot p^{-2}$.

4) Пробегаются значения $\alpha = j\cdot|V|^{-1}$ с заданным по j шагом Δ: $j = \Delta, 2\Delta, \ldots, |V|$.

Предложенный способ тестирования моделирует индексацию таких частей словаря растущего объёма, которые достаточно равномерны по составу использованных букв алфавита и тем самым похожи на некоторые, реально используемые, словари такого объёма.

В экспериментах полагалось $\Delta = 50000$, $p = 500$. Тестирование проводилось на компьютере с процессором 3 GHz Intel Core 2 Quad Q9650, 8 Гб ОЗУ. Компиляция всех модулей, в т.ч. встраиваемых БД, осуществлялась для платформы Windows 7 x64 в Visual C++ 2010 в режиме Release с поддержкой инструкций SSE2.

3.2. Тестирование на данных словарной морфологии. Для эксперимента использованы следующие морфологические данные: 2.3 млн различных словоформ, порядка 10 тыс. различных грамматических значений с начальными формами. Сохраняются не сами значения, а их числовые дескрипторы. Начальная форма сохраняется в виде кода *RnAx*, означающего, что для получения по заданному слову его началь-

ной формы необходимо отнять от него *n* букв с конца и добавить в качестве суффикса строку *x*. Так, некоторому дескриптору 253 соответствует грамматическое значение *(им.сущ., жен.р., ед.ч., им.п.)*. В качестве значения ключа используется строка из кода начальной формы, символа-разделителя и дескриптора. Одному ключу могут соответствовать до 18 различных значений.

Эксперимент №1: с сохранением множества значений. Данные сохраняются в multivalued-стиле: для каждого ключа *key* с *r* различными значениями $\{val_1, \ldots, val_r\}$ к языку автомата добавляется *r* строк $key\#val_j$, $j = 1, \ldots, r$. Результаты эксперимента показаны на рис. 4. По результатам наблюдается следующее:

1) *DawgMap* для построения требует наибольший объём времени (порядка 30 с для всего словаря) и приемлемый объём памяти (не более 100 Мб для данного словаря);

2) при успешном поиске *DawgMap* на два порядка эффективнее встраиваемых БД *SQLite* и *Oracle BerkeleyDB*, и сравним по порядку с *std::map* – для данного словаря *MDag* быстрее *std::map* в 30-40 раз;

3) в отличие от линейного роста размера, наблюдаемого у других структур, *DawgMap* демонстрирует следующее поведение: первую четверть обработанного объёма размер автомата растёт, к середине объёма достигает практически начального минимума (когда обработаны первые 50 тыс. элементов), и становится в 1.7 раза меньше максимального объёма (это также соответствует объёму, полученному после индексации первых 10% всего объёма словаря), когда все данные уже обработаны;

4) сериализованный вариант *DawgMap* для данного словаря компактен: его исходный объём в виде текста из ключей и списков значений составлял 45 Мб, результирующий объём структуры хранения – 2.1 Мб.

Схожий размер достижим и в случае, если сохранять не идентификаторы значений, а бинарное представление самих значений (объёмом порядка 50 байт каждое) – результирующий размер структуры в этом случае составит 3.5 Мб. Вероятно, возможность такого сжатия обусловлена тесной связью ключей и значений. Так, ключам, имеющим общие окончания, часто будут соответствовать одинаковые наборы значений. Кроме того, в самих значениях присутствует естественная иерархия: разбиение словоформ по частям речи, в рамках одной части речи – по соответствующим характеристикам. Эксперименты показали, что размер автомата может колебаться (1.5–2 раза) в зависимости от того, насколько хорошо пользователь библиотеки учёл иерархию в бинарном представлении значений.

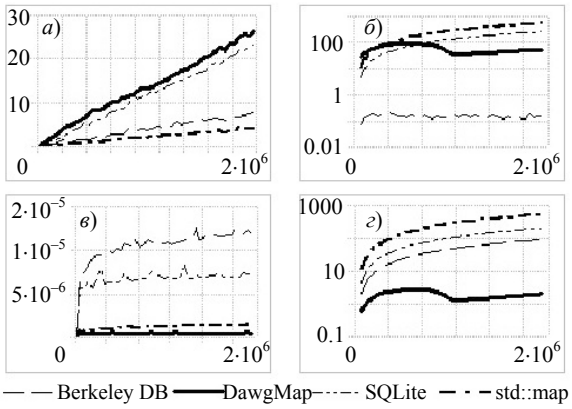

Рис. 4. Серия экспериментов для данных словарной морфологии: (*а*) – время построения словаря объёмом *n* ключей (*n* – абсцисса); (*б*) – объём памяти в Мб для формирования словаря (логарифмическая шкала); (*в*) – время успешного поиска в таком словаре; (*г*) – размер сериализованной структуры хранения в Мб (логарифмическая шкала).

При $p = 500$ средняя производительность для полного словаря из 2.3 млн элементов составила $4.86 \cdot 10^{-7}$ с, а на словаре из 50 тыс. элементов – $4.14 \cdot 10^{-7}$ с. Таким образом, простейший морфологический анализатор, построенный на *DawgMap*, может на некоторых входных наборах достигать производительности порядка 20 млн слов в секунду. Время успешного поиска, усреднённое по всем элементам словаря, составило 3 млн слов в секунду. Аналогичная средняя производительность была получена на текстовом корпусе из 100 Мб новостных сообщений. Также необходимо отметить следующее положительное свойство *DawgMap* – он слабо чувствителен по времени поиска при росте объёма словаря.

Эксперимент №2: с сохранением единственного значения. Во втором эксперименте для каждого ключа *key* со значениями $\{val_1, \ldots, val_r\}$ сохранялось значение $val_1@\ldots@val_r$. При извлечении это значение можно разделить на *r* частей по символу @. В такой постановке качественная картина для *DawgMap* сохранилась, время успешного поиска осталось таким же, но структура хранения заняла на 10% больше. Для встраиваемых БД имело место двукратное сокращение размера и полуторакратное ускорение операции поиска.

3.3. Тестирование на базе поисковых запросов. Использовалась выборка из 1 млн запросов одной из российских поисковых систем. В отличие от данных словарной морфологии

(средняя длина 7 символов), здесь часто встречаются длины порядка 15-20 символов, алфавитный состав богаче – используются буквы строчные и прописные, символы латинского алфавита, некоторые разделители. Для каждого добавляемого запроса генерировалось одно из 10 тыс. равномерно распределённых числовых значений, моделирующих сохранение частоты запроса.

Несмотря на более чем в 2 раза меньшее количество ключей, результирующий размер *DawgMap* составил 34 Мб против 2.1 Мб для данных машинной морфологии, а для его построения потребовалось в 3 раза больше памяти. Производительность успешного поиска *DawgMap* ухудшилась примерно в 2 раза, что обусловлено увеличением длины ключа.

Дополнительно была выполнена оценка времени неуспешного поиска, т.е. когда искомый ключ отсутствует в словаре. Для машинной морфологии более важно́ время успешного поиска, т.к. в текстах большинство слов правильные. Для подсказки в поисковой строке и исправления опечаток важны оба показателя. Качественная картина для *DawgMap* в сравнении с успешным поиском не меняется, отмечены две особенности: 1) *SQLite* отрабатывает неуспешный поиск в несколько раз быстрее, 2) увеличивается разрыв между *DawgMap* и *std::map* в пользу первого.

3.3. Асимптотика числа вершин автомата. Если автомат пополняется строками словаря по схеме, предложенной в п. 3.1, то сформулированная ранее в экспериментах динамика его мощности характерна также и для словарей случайно сгенерированных строк: имеет место увеличение мощности в начале и уменьшение к концу сеанса пополнения.

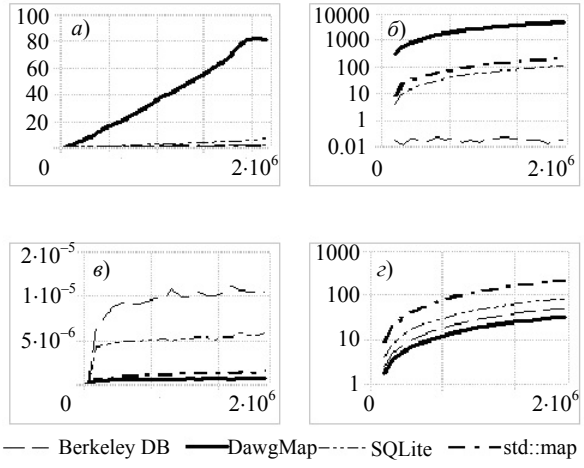

Рис. 5. Серия экспериментов для данных поисковых запросов. Содержание графиков то же, что на рис. 4

Пусть теперь граф пополняется строками словаря в строго лексикографическом порядке, и уже добавлено n строк. Тогда для минимального автомата, согласно эксперименту, наблюдается асимптотика числа вершин $O(n^{\beta})$, $0 < \beta < 1$ – как на случайных, так и на реальных данных, β зависит от словаря. Так, для эксперимента №1 из п. 3.2 кривая мощности описывается функцией $6.7x^{0.685}$ с погрешностью не более 3 тыс. единиц (при максимальных значениях порядка 100 тыс.).

Случай $\beta = 1$ соответствует асимптотике префиксного дерева.

Подтверждений приведённой гипотезе в доступных источниках автором не обнаружено.

Заключение

В статье дано описание *DawgMap* – реализации структуры хранения строковых ключей с наборами ассоциированных значений, эффективной по объёму памяти и скорости поиска по ключу. Дан обзор предметной области, охарактеризованы различные подходы к организации словарей. Даются объяснения конструктивным решениям, принятым в реализации *DawgMap*. На основе реальных лингвистических данных проведены эксперименты, по результатам которых разработанная структура оказывается в десятки раз эффективнее известных встраиваемых решений.

DawgMap используется в следующих коммерческих разработках: 1) модуль исправления опечаток в словах на русском языке, 2) модуль подсказки для строки ввода запроса в поисковой системе по вьетнамскому сегменту сети Интернет, 3) модуль извлечения из текста объектов специального вида (персон, адресов и т.д.), 4) машинная словарная и бессловарная морфология для линейки языков (русский и пр.).

Список литературы

1. Daciuk J., Mihov S., Watson B.W., Watson R.E. Incremental construction of minimal acyclic finite-state automata // Computational Linguistics. 2000. V. 26. N.1. P. 3–16, March ‘00.

2. Lucchesi C.L., Kowaltowski T. Applications of finite automata representing large vocabularies // Software-Practice and Experience. 1993. 23. 15–30.

3. Ахо А.В., Лам М.С., Сети Р., Ульман Д.Д. Компиляторы: принципы, технологии и инструментарий. 2-е изд. М.: Вильямс, 2008.

4. http://www.open-std.org/jtc1/sc22/wg21/docs/ papers/2005/n1905.pdf

5. http://www.*sqlite*.org/inmemorydb.html

6. http://www.oracle.com/technetwork/products/*berkeleydb*/overview/index.html

7. Скатов Д.С. Эффективные реализации строковых словарей для решения задач компьютерной лингвистики // Интеллектуальные системы и компьютерные науки: Сб. тр. X Международ. конф. М., 2011.

8. Кнут Д. Искусство программирования. Т. 3. Сортировка и поиск. 2-е изд. М.: Вильямс, 2007.

EFFICIENT STORAGE STRUCTURE OF A DICTIONARY WITH STRING KEYS AND ASSOCIATED VALUES

V.P. Gergel, D.S. Skatov

The article describes an implementation of a dictionary based on minimal finite-state automata in which a key is associated with a set of values. The advantages of the dictionary as compared to other available implementations such as DMBS or C++ standard class library are illustrated. The acceleration of the search process ranges from 20–40 times up to several orders of magnitude, while the memory volume required is considerably reduced.

Keywords: dictionary, efficient storage structures, prefix tree, finite-state automaton, computational linguistics, machine morphology.