РОССИЙСКАЯ ФЕДЕРАЦИЯ

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ

ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО И ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

ТЮМЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Институт математики, естественных наук и информационных технологий

Кафедра программного обеспечения

КУРСОВАЯ РАБОТА

Тема: Идеальное хеширование

Выполнил:

Студент группы 124-2

Кочеров Кирилл Дмитриевич

Проверил:

Пиджаков Святослав Игоревич

Тюмень – 2014

Аннотация.

В данной курсовой работе рассматривается алгоритм для построения минимальной идеальной хэш-функции. Сложность используемого алгоритма, а также количество необходимой памяти для его работы линейно зависит от размера входных данных, что свидетельствует о его оптимальности. Алгоритм реализован с использованием среды разработки Light Table и языка программирования Clojure.

Содержание.

[Введение 2](#_Toc380436943)

[Теория и терминология 3](#_Toc380436944)

[Описание алгоритма 3](#_Toc380436945)

[Шаг 1 Построение графа 4](#_Toc380436946)

[Шаг 2 Вычисление функции 6](#_Toc380436947)

[Шаг 3 Получение искомой хэш-функции. 7](#_Toc380436948)

[Используемые структуры данных. 7](#_Toc380436949)

[Анализ алгоритма 7](#_Toc380436950)

[Эмпирический анализ времени исполнения 7](#_Toc380436951)

[Потребление памяти 8](#_Toc380436952)

[Заключение 10](#_Toc380436953)

[Список литературы и источников 11](#_Toc380436954)

# Введение

Использование минимальных идеальных хэш-функций до недавних пор было ограничено приложениями, где множество хэшируемых объектов сравнительно невелико из-за ограничений существовавших алгоритмов. Однако со временем объем таких множеств многократно возрос, что сделало неэффективным - а порой даже и невозможным - хранение такого количества информации.

Выбор данного алгоритма позволяет, проведя предварительные линейные расчёты для построения функции, использовать построенную функцию для большего количество ключей (вплоть до сотен миллионов), благодаря существенно сниженному количеству памяти, требуемому для хранения объектов.

Минимальные идеальные хеш-функции оказались очень эффективным решением для задач, в которых требуется эффективное использование памяти и быстрый доступ к предметам из статичных множеств, таких как: слова в естественном языке, зарезервированные слова в языках программирования, uniform resource locators (URLs) в поисковых системах. Поэтому идеальное хеширование широко применяется в системах управления базами данных (СУБД), системах электронных платежей, компиляторах, операционных системах, и других.

Далее будет рассмотрен оптимальный алгоритм, который позволит для заданного набора слов построить минимальную идеальную хэш-функцию за линейное время. Данный алгоритм носит название CHM (по первым буквам фамилий авторов), который разработали Zbigniew J. Czech, George Havas и Bohdan S. Majewski в 1992 году.

# Теория и терминология

Рассмотрим множество из слов, каждое из которых состоит из последовательности символов упорядоченного алфавита .

**Хэш-функция** – функция , которая отображает множество слов на некоторый интервал целых чисел где – целое число (обычно ).

Два разных слова с одинаковыми значениями хэш-функции называются **синонимами**. В случае, когда функция порождает синонимы, говорят, что произошла **коллизия**. Сегодня известны различные подходы для разрешения коллизий, такие как: метод цепочек или метод открытой адресации.

Функция называется **идеальной**, если она инъективна на . Если , то функцию называют **минимальной** хэш-функцией.

Соответственно, функция, обладающая этими двумя свойствами, будет называться **минимальной идеальной хэш-функцией**.

В данной работе будет рассмотрен алгоритм, основанный на построении случайных графов для нахождения функции следующего вида:

где и – функции, которые отображают слова в числа, – функция, которая отображает числа в

# Описание алгоритма

Далее приведено краткое описание алгоритма:

1. Построить граф используя случайные функции и .
2. Если граф ациклический, то, используя обход в глубину (DFS), вычислить значения функции для каждой вершины . Иначе перейти к шагу 1.
3. Используя значения функции , составить минимальную идеальную хэш-функцию.

Приведенный выше будет линейно зависеть от размера входных данных в случае, если количество попыток на построение графа не будет достаточно велико. Как показывают авторы[1], если определить константу , отвечающую за количество вершин в графе как

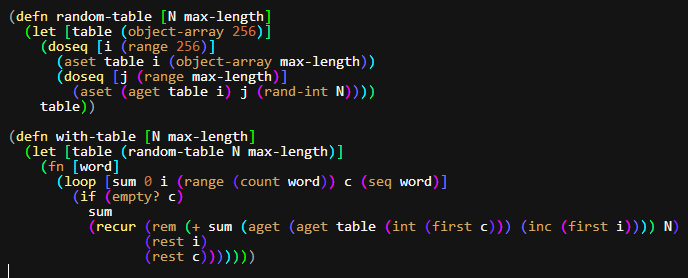
где – произвольная константа , то вероятность построить ациклический граф будет удовлетворять поставленным требованиям. В данной работе будет тожественно , что даст ожидаемое количество итераций для построения графа =

Рассмотрим теперь каждый шаг более подробно.

## Шаг 1 Построение графа

Построение графа выполняется итеративно. До тех пор, пока граф – ациклический, для очередного слова вычисляются значения двух функций: и . Затем в граф добавляется новое ребро . Если на каком-то этапе вычислений в графе образуется цикл, то построение начинается с начала с новыми функциями и .

Функции и можно определить следующим образом:

Где и - таблицы, где каждому символу алфавита ∑ для каждой из возможных позиций в слове поставлено в соответствие случайное целое число.

Листинг Выбор случайных функций

Существует эффективный способ проверки, содержит ли граф хотя бы один цикл, который не повлияет на сложность исходного алгоритма. Для этого следует воспользоваться структурой данных disjoint set. При каждом добавлении ребра в граф поместим вершины и в одно множество. Тогда при добавлении очередного ребра в граф появится цикл тогда и только тогда, когда обе вершины и уже принадлежат одному множеству. Данный алгоритм, реализованный с использованием структуры данных disjoint set имеет сложность , где – функция, обратная функции Аккермана. Несмотря на нелинейную теоретическую оценку, эмпирически время работы можно считать линейным, так как, по утверждению авторов[2], не превысит 4 для любого набора данных, который сможет предоставить человек.

Пример реализации на языке Clojure.

Листинг 2Выбор случайных функций

## Шаг 2 Вычисление функции

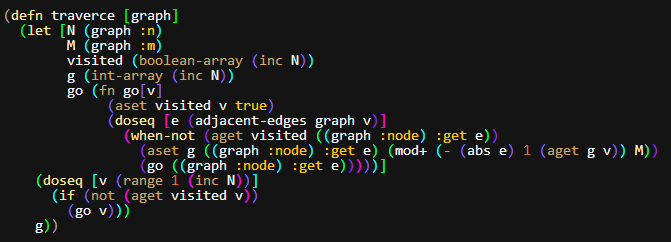
Для построения минимальной идеальной хэш-функции нам потребуется второстепенная вспомогательная функция . Она определена на множестве целых чисел множество её значений – . Данная функция вычисляется итеративно по построенному ранее графу по следующему алгоритму:

Предварительно пронумеруем все ребра построенного графа начиная с нуля в соответствии с порядком следования слов во входных данных программы, которым соответствуют ребра графа.

Для каждой компоненты связности выберем любую вершину, принадлежащую к ней . Путь значение функции для вершины равно нулю . Воспользуемся обходом в глубину (DFS) для построенного графа из вершины , и для каждой посещенной вершины , для всех вершин, смежных с , установим значение функции :

.

Где функция – номер ребра.

Пример реализации на языке Clojure.

Листинг 2Вычисление функции g

## Шаг 3 Получение искомой хэш-функции.

Вычислив функцию становится возможным определить результирующую минимальную идеальную хэш-функцию как:

## Используемые структуры данных.

При реализации алгоритма были использованы следующие структуры данных:

1. Edge-oriented graph. Описание данной структуры можно найти в [3].
2. Disjoint set[2].

# Анализ алгоритма

## Эмпирический анализ времени исполнения

Программа была протестирована для константы c = 3, со входными данными – словарем английских слов. Результаты приведены в таблице:

|  |  |
| --- | --- |
| m = n / c | Время, с |
| 1000 | 0.272224335 |
| 2000 | 0.554942234 |
| 4000 | 1.075503446 |
| 8000 | 2.168692476 |
| 16000 | 4.317165526 |
| 32000 | 8704.751519 |
| 64000 | 17346.05948 |

Временные характеристики в таблице – среднее значение из 25 тестов. Очевидно, эмпирический анализ подтверждает линейную теоретическую оценку сложности алгоритма.

## Потребление памяти

Рассмотрим потребление памяти приложением. Для построения минимальной идеальной хэщ-функции используется следующие структуры данных:

**Graph**

* FIRST – вектор, который хранит чисел integer, каждое из которых представляет собой номер первого ребра для каждой вершины. Следовательно, вектор FIRST требует байт.
* EDGES – вектор, представляющий ребра графа в программе. Каждое ребро задается парой целых чисел, означающих номер соответствующих вершин. Поэтому вектор EDGES требует байт
* NEXT – вектор, который вкупе с вектором FIRST представляет собой связный список из ребер, смежных для данной вершины . Поэтому вектор NEXT требует байт.

**Другие вспомогательные структуры данных:**

* visited – вектор из бит, каждый из которых определяет, было ли уже подсчитано значение функции соответствующее данной вершине. visited требует байт.
* – функция, которая представлена вектором из целых чисел. g требует байт.

Подводя итоги, для построения минимальной идеальной хэш-функции требуется:

байт.

А поскольку значение должно принимать значение не менее , то минимальное количество требуемой памяти:

Теперь подсчитаем необходимое количество памяти для хранения минимальной идеальной хэш-функции. Для представления построенной функции, нам необходим только вектор g, который занимает байт. Соответственно, для хранения минимальной идеальной хэш-функции требуется байт.

# Заключение

Алгоритм, описанный в данной курсовой работе был реализован на языке программирования Clojure. Результат – консольное приложение, предоставляющее функционал минимальной идеальной хэш-функции конечному пользователю.

Благодаря тому, что язык Clojure полностью интегрирован в JVM (по сути Clojure – библиотека для JVM), то дополнительно был реализован класс Pasher, который инкапсулирует всю функциональность приложения. При компиляции был получен файл Pasher.class, который может быть загружен в любой проект Eclipse или NetBeans.

# Список литературы и источников

1. Zbigniew J. Czech, George Havas, Bohdan S. Majewski. An optimal algorithm for generating minimal perfect hash functions. In Information Processing Letters, 43(5): 257-264, October 1992.
2. B. Bollobas and I. Simon. On the expected behavior of disjoint set union algorithms. In 17th Annual ACM Symposium theory of Computing STOC’85, pages 224-231, may 1985.
3. J. Ebert. A versatile data structure for edge-oriented graph algorithms. Communications of the ACM, 30(6):513-519, June 1987.
4. D. E. Knuth. The Art of Computer Programming, Vol. 3: Sorting and Searching. Addison-Wesley, Reading, Mass., 2nd edition, 1973.
5. Stuart Halloway, Aaron Bedra. Programming Clojure, 2nd edition. The Pragmatic Programmers. 296 pages , 2012.
6. Luke VanderHart, Stuart Sierra. Practical Closure (Expert’s Voice in Open Source). Apress, may 2010.