CS314 HW 2: Functional programming and Racket

Using Bloom Filters for Spell Checkers

Spring 2019

In this project, you will implement an "infrastructure" that allows you to create and experiment with different spell checkers that are based on hash functions. Hash functions map character strings (e.g. "hello") to integer values. Each hash function has a range of [0, size - 1], i.e., can map a character string to an integer value greater than or equal to 0, but smaller than size. The index generated by applying hash function h to a word w, i.e., $h(w)$, can be used to index into a data structure that stores information associated with w. For instance, symbol tables typically use hashing to map identifiers to their attributes.

Hashing can also be used to implement a spell checker. Instead of comparing a word w with entries in a dictionary of words, the word's hash value $h(w)$ is used to index into a bitvector. A bitvector is an array of booleans, i.e., each entry is either true (= 1) or false (= 0). If bitvector(h(w)) = false, w is spelled incorrectly. The bitvector is created by applying a hash function to each word w in the directory, setting the the bitvector to true (:= 1) for the computed hash value, i.e., bitvector(h(w)) := 1 for all w in the dictionary. Typically, the memory requirement of a dictionary of words is much larger than the memory requirement of its corresponding bitvector. In addition, indexing into the bitvector (or any other array data structure) is typically much faster than using any lookup technique based on string comparisons.

Unfortunately, the more compact representation and faster access has its price. Clearly, the choice of the hash function will determine the quality of the spell checker. If a misspelled word has the same hash value as a correctly spell word in the dictionary, the spell checker will fail to report the misspelled word. In order to increase the chances of mapping a misspelled word to a

bitvector location that is marked false, we can

- use a set of (different) hash functions $h_1, ..., h_n$ instead of just a single hash function h; instead of marking (checking) only $h(w)$ as true in the bitvector, we now mark (check) all $h_i(w)$ as true.
- increase the length (size) of the bitvector, allowing hash functions with larger ranges and thereby generating a sparser bitvector, i.e., a bitvector with many more 0 entries than 1 entries.

In this project, you will write a spell checker generator that allows the experimentation with different sets of hash functions and their ranges. Racket is a language well suited for "rapid prototyping", i.e., allows a quick implementation of a working prototype. We will represent a bitvector in Racket as a list of indices, i.e., a list of integer values for which the bitvector entry is '1' (e.g. vector [0 0 1 0 1 0 0] is represented as list '(2 4) – the indexing is 0-based). This is a typical representation of a bitvector where '1' entries are sparse, i.e., where there are many more '0' entries than '1' entries.

1 Hash Functions

Our hash functions consist of two main steps. In the first step, the input word w is mapped to an integer value, called its key. In the second step, the key is mapped to its final hash value. We will compute w's key by associating an integer value with each letter (symbol) in w, and then multiplying the resulting values. Note: In this project we only consider lower case letters and words.

We will use the following key function key(w) – please note that we are using pseudocode for functions here rather than the Racket's function notation:

```
key <- 5413
for each character c in w
    reading from right to left do
    key <- 29 * key + ctv(c)
end for
```

where ctv ("character-to-value") maps 'a' to 1, 'b' to 2, ... and 'z' to 26.

1.1 Division Method

In the division method, a key k is mapped into size slots by taking the remainder (modulo) of k divided by size. In other words, the hash functions have the form: $h(k) = k \mod size$. The size should be a prime number in order to generate a more uniform spread of the hash function values over the function's entire [0, size - 1] range.

1.2 Multiplication Method

The multiplication method for creating hash functions operates in two steps. First, we multiply the key k by a constant A (we choose $A = 0.6780227553$), and then extract the fractional part of kA. Then, we multiply this value with size and take the floor of the result. In other words, the hash functions have the form:

$$h(k) = \lfloor size \cdot (k \cdot A - \lfloor k \cdot A \rfloor) \rfloor$$

The desired uniform spread of the hash function over its entire range is not dependent on a particular selection of size.

2 Project Description

For this project, words are represented as lists of lower case symbols, e.g., the word "class" is represented as `'(c l a s s)`. The project consists of three parts:

1. Write the key function key which takes a word as input and maps it to its key using the provided cvt character to value mapping.

2. Write a function that generates hash functions based on the division method `gen-hash-division-method`, and a function that generates hash functions based on the multiplication method `gen-hash-multiplication-method`.

3. Write a function gen-checker that takes as input a list of hash functions and a dictionary of words, and returns a spell checker. A spell checker is a function that takes a word as input and returns either `#t` or `#f`, indicating a correctly or incorrectly spelled word, respectively.

Your implementation of gen-checker should generate the bitvector representation for the input dictionary exactly once. For the implementation of these functions, you can use Racket functions such as modulo and floor. You must use the reduce function at least once in your implementation in a useful way. The definition of reduce is given in file include.rkt.

Note that a hash function based on a multiplication method may return integer values of the form 17.0 instead of just 17 . Use the Racket "=" function (numeric equality) to test for equality. Do not use "eq?" or "equal?".

3 How To Get Started

The code for this project is in the hw2.tar file on Sakai.

Create your own directory on the ilab cluster, and copy the entire provided project hw2.tar to your own home directory or any other one of your directories. Say `tar xf hw2.tar` to extract the project files.

Make sure that the read, write, and execute permissions for groups and others are disabled (`chmod go-rwx <directory name>`). It is considered cheating if you do not protect your project files!

There are four files: spell.rkt, include.rkt, test-dictionary.rkt and dictionary.rkt. You are only allowed to modify the spell.rkt file. File test-dictionary.rkt contains a small dictionary consisting of only three words. Use it to debug your program. File dictionary.rkt contains a list of over 45,000 words allowing you to generate a realistic spell checker. You can switch between different dictionaries by requiring the corresponding .rkt file.

4 Grading

You will submit your version of file spell.rkt via the Assignments page on Sakai. No other file should be modified, and no additional files may be used.

Your programs will be graded based on programming style and functionality. You must not use assignment (set!) in Racket. Functionality will be verified through automatic testing on a set of test cases.