ELSEVIER

Theoretical Computer Science 182 (1997) 1–143

Theoretical Computer Science

Fundamental Study

Perfect hashing

Zbigniew J. Czech[a,*], George Havas[b], Bohdan S. Majewski[c]

[a] *Silesia University of Technology, 44-100 Gliwice, Poland*
[b] *University of Queensland, Queensland 4072, Australia*
[c] *University of Newcastle, Callaghan, NSW 2308, Australia*

Communicated by M. Nivat

Contents

* Corresponding author. E-mail: zjc@star.iinf.polsl.gliwice.pl, web:http://sun.zo.iinf.polsl.gliwice.pl./~zjc.

PII S0304-3975(96)00146-6

Preface

Let S be a set of n distinct keys belonging to some universe U. We would like to store the keys of S so that membership queries of the form "Is x in S?" can be answered quickly. This searching problem, also called the *dictionary problem*, is ubiquitous in computer science applications.

Various algorithms to solve the dictionary problem have been devised. Some of them are based on *comparisons of keys*. For example, binary search using a linear ordering of keys compares the given key x with a key x_i in the table. Depending on the result of the comparison ($x < x_i$, $x = x_i$, $x > x_i$), the search is continued in one of three different ways. The idea of comparisons between keys is also used in the methods involving binary search trees and AVL trees. Another approach to searching makes use of the representation of keys as sequences of digits or alphabetic characters. From the first digit of a given key a subset of keys which begins with that digit is determined. The process can be then repeated for the subset and subsequent digits of the key. This digitally governed branching procedure is called a *digital search*.

Yet another approach to the dictionary problem is to compute a function $h(x)$ which determines the location of a key in a table. This approach has led to a class of very efficient searching methods commonly known as *hashing* or *scatter storage* techniques. If a set of keys is static, then it is possible to compute a function $h(x)$ which enables us to find any key in the table in just one probe. Such a function is said to be a *perfect*

hash function. A perfect hash function which allows us to store a set of records in a minimum amount of memory, i.e. in a table of the size equal to the number of keys times the key size, is called a *minimal perfect hash function.*

Minimal perfect hash functions have a wide range of applications. They are used for memory efficient storage and fast retrieval of items from static sets, such as reserved words in programming languages, command names in interactive systems, or commonly used words in natural languages. Therefore there is application for minimal perfect hash functions in compilers, operating systems, language translation systems, hypertext, hypermedia, file managers, and information retrieval systems.

This work is a monograph on perfect and minimal perfect hashing. It is intended to serve researchers and professionals of computer science. Some parts of this text can be used as an introduction to the subject for students. All three authors have covered material from this work in Algorithms and Data Structures courses taught to higher level undergraduate or to starting postgraduate students.

Researchers will find in this work the current state of developments in perfect and minimal perfect hashing. Bibliographical remarks facilitate further reading. With practitioners in mind, we have included many examples of minimal perfect hash functions which can be readily implemented in practice. We also indicate where some implementations are available publicly on the Internet (see Section 6.6).

It is assumed that the reader possesses some background in the design and analysis of algorithms.

The work comprises seven chapters plus an appendix and a comprehensive bibliography. Ch. 1 is an introduction to perfect and minimal perfect hashing. Section 1.1 contains basic definitions. In Section 1.2 we present a simple example of finding a minimal perfect hash function by trial and error. This example shows the kinds of difficulties which are encountered in designing such a function. Section 1.3 discusses the space and time requirements for perfect hash functions. This section is included in the Introduction because it is fundamental but it is quite theoretical and difficult. The reader may safely skim it on first reading without causing difficulties in understanding later sections.

The remaining chapters constitute a detailed study of existing methods of perfect and minimal perfect hashing. Ch. 2 describes the algorithm proposed by Sprugnoli which came from consideration of theoretical properties of sets of integers. Five different descendants of it are then presented. Even though each tries to overcome drawbacks of its predecessor, we show that none of them has serious practical value. Ch. 3 discusses the methods based on rehashing and segmentation, i.e. breaking up the input set of keys into a number of smaller sets, for which a perfect hash function can be found. Ch. 4 presents Cichelli's method which uses a simple heuristic approach to cut down the search space. The solutions of Sprugnoli and Cichelli, despite of their flaws, inspired a number of improved methods capable of handling larger key sets. In the rest of Ch. 4 we describe Sager's algorithm which was developed as an attempt to optimize Cichelli's method. It was the basis of various improvements which led to algorithms which have practical application to larger sets. We describe further improvements to

Sager's algorithm which speed up generation of minimal perfect hash functions. Ch. 5 discusses several algorithms which construct perfect hash functions based on sparse table compression. The single and double displacement methods proposed by Tarjan and Yao are first presented. Then the modifications of the methods which make them suitable for letter-oriented keys are considered. In Ch. 6 we present three algorithms and their modifications that generate perfect hash functions by using probabilistic approach. Some advanced applications of these methods are also discussed. Finally in Ch. 7 we give an overview of dynamic perfect hashing. The Appendix contains the notation index.

Chapter 1. Introduction to perfect and minimal perfect hashing

1.1. Basic definitions

Let $U = \{0, 1, 2, \ldots, u-1\}$ be the universe for some arbitrary positive integer u, and let S be a set of n distinct elements (keys) belonging to U. A *hash function* is a function $h: U \to M$ that maps the keys from S into some given interval of integers M, say $[0, m-1]$. Given a key $x \in S$, the hash function computes an address, i.e. an integer in $[0, m-1]$, for the storage or retrieval of x. The storage area used to store keys is known as a *hash table*. (In practice, the goal is to handle the information contained in the record associated with a key. We simplify the problem by considering only the table of keys and assuming that once a key is located or placed in the table, associated information can be easily found or stored by use of pointers.)

Keys for which the same address is computed are called *synonyms*. Due to the existence of synonyms, a situation called *collision* may arise, in which two different keys have the same address. Various schemes for resolving collisions are known. A *perfect* (or 1-*probe*) *hash function* for S is an injection $h: U \to [0, m-1]$, i.e. for all $x, y \in S$ such that $x \neq y$ we have $h(x) \neq h(y)$, which implies that $m \geqslant n$. If $m = n$ and h is perfect, then we say that h is a *minimal perfect hash function*. As the definition implies, a perfect hash function transforms each key of S into a unique address in the hash table. Since no collisions occur, each key can be retrieved from the table in a single probe. We also say that a minimal perfect hash function *perfectly scatters* the keys of S in the hash table. The function h is said to be *order preserving* if for any pair of keys $x_i, x_j \in S$ $h(x_i) < h(x_j)$ if and only if $i < j$. In other words, we assume that keys in S are arranged in some sequential order, and the function preserves this order in the hash table. Such a function is called an *ordered minimal perfect hash function*.

Perfect and minimal perfect hashing is readily suitable only for static sets, i.e. sets in which no deletion and insertion of elements occurs. As defined above, the keys to be placed in the hash table are nonnegative integers. However, it is often the case that keys are sequences of characters over some finite and ordered alphabet Σ. To deal with such a key, a hashing scheme can either convert it to an integer by using the

ordinal numbers of characters in Σ, or by applying a character by character approach. An example of using the latter method is given in Section 1.2.

The question arises whether for a given set of keys an ordered minimal perfect hash function always exists. The answer to this question is positive as it is straightforward to prove that any two finite, equal size and linearly ordered sets, say X and Y, are isomorphic. This means that there exists an injective function $h: X \to Y$ transforming X into Y such that

$$\forall x_1, x_2 \in X \quad x_1 \leqslant x_2 \quad \Leftrightarrow \quad h(x_1) \leqslant^* h(x_2),$$

where $\leqslant$ and $\leqslant^*$ are the respective linear order relations.

In practice, finding any kind of perfect hash function, especially for large sets of keys, may not be easy as these functions are very rare. Knuth [66, p. 506] indicated that only one function in ten million is a perfect hash function for 31 keys mapped into 41 locations.

If we consider random placement of n keys into a hash table of size m ($m \geqslant n$) then the probability that no collisions occur is

$$\frac{m(m-1)\cdots(m-n+1)}{m^n}. \tag{1.1}$$

For $m = 13$ and $n = 10$ this probability is equal to 0.0074. The probability of placing n keys into an n-element hash table without collisions is

$$n!/n^n. \tag{1.2}$$

This probability is very small, even for relatively small values of n. For example, for $n = 10$ it is equal only to 0.00036. These probabilities decrease rapidly for larger n.

Expressions (1.1) and (1.2) can be treated as the probabilities that a function h randomly chosen from the set of functions $h: U \to [0, m-1]$ is a perfect hash function and a minimal perfect hash function, respectively.

The probability that a randomly chosen function is an ordered minimal perfect hash function is even smaller. This equals to $1/n^n$ which, for a 10-element set of keys, gives 10^{-10}.

A crucial issue in perfect hashing is the efficiency of the hashing function. The function should be easily computable, i.e. its value for any key should be evaluated in a small number of steps, ideally in fast constant time. Furthermore, it is desirable that the amount of memory required to store a program for the function evaluation is minimized. We discuss these problems in more detail in Section 1.3.

1.2. Trial and error

Suppose that we want to find a minimal perfect hash function for the set of mathematical function identifiers $S_\alpha = \{$`exp`, `cos`, `log`, `atn`, `tng`, `int`, `abs`, `sgn`, `sqr`, `sin`, `rnd`$\}$, $|S_\alpha| = 11$. Several approaches to such a task can be considered. For example, by use of one of the standard codes (ASCII, say), the set of identifiers can be translated into a set of distinct integer numbers from some range. Then, for the resulting

Table 1
Conflicts among identifiers

Function	σ_{i_1}	σ_{i_2}	σ_{i_3}	$\sigma_{i_2} + \sigma_{i_3}$	$\sigma_{i_1} + \sigma_{i_3}$
Identifiers in conflict	atn abs	cos log	cos abs	atn int	sin sgn

Table 2
Letter frequencies

Letter	a	b	c	e	g	i	l	n	o	q	r	s	t	x
Frequency	2	1	1	1	1	2	1	3	2	1	1	3	2	1

set, one of the methods for finding a minimal perfect hash function can be utilized. Another approach, which we shall follow in this section, is to determine an assignment of integers to the letters of the alphabet. Based on this assignment, the hash function converts the identifiers into distinct hash table addresses.

We shall view the identifiers as sequences of letters. Since all of the identifiers have the same length, the only attributes we can exploit in constructing the function are: the letters which constitute the identifiers; and positions of these letters within each identifier. To convert the members of S_α into the hash table addresses $0, 1, \ldots, 10$, we have to assign, by a trial and error method, an integer to each letter of the alphabet so that the addresses of keys are distinct. Let us denote $id = \sigma_{i_1}\sigma_{i_2}\sigma_{i_3}$, $id \in S_\alpha$, and let $\widehat{\sigma_{i_k}}$ be the integer assigned to letter σ_{i_k}, for $k = 1, 2, 3$ and $i = 0, 1, \ldots, 10$. The following functions can be considered as candidates for minimal perfect hash functions: $h(\sigma_{i_1}\sigma_{i_2}\sigma_{i_3}) = r$, where r is an expression from the list: $\widehat{\sigma_{i_1}}$, $\widehat{\sigma_{i_2}}$, $\widehat{\sigma_{i_3}}$, $\widehat{\sigma_{i_1}} + \widehat{\sigma_{i_2}}$, $\widehat{\sigma_{i_2}} + \widehat{\sigma_{i_3}}$, $\widehat{\sigma_{i_1}} + \widehat{\sigma_{i_3}}$, $\widehat{\sigma_{i_1}} + \widehat{\sigma_{i_2}} + \widehat{\sigma_{i_3}}$. This list contains expressions in increasing order of complexity. Surely, we would like to have as simple a hash function as possible, therefore we need a suitable r possibly from the beginning of the list. Some of the functions on the list have to be discarded however, because of "conflicts" among the letters which take part in computing the hash values. For example, for the function $h(\sigma_{i_1}\sigma_{i_2}\sigma_{i_3}) = \widehat{\sigma_{i_1}}$, there are unresolved conflicts among identifiers `atn` and `abs`, and `sin`, `sqr` and `sgn`. Table 1 shows some of these conflicts.

There are no conflicts for the given set S_α with either of the two possible candidate functions $h(\sigma_{i_1}\sigma_{i_2}\sigma_{i_3}) = \widehat{\sigma_{i_1}} + \widehat{\sigma_{i_2}}$ or $h(\sigma_{i_1}\sigma_{i_2}\sigma_{i_3}) = \widehat{\sigma_{i_1}} + \widehat{\sigma_{i_2}} + \widehat{\sigma_{i_3}}$.

Let us continue with the former candidate since it is simpler. Our task is to assign integers to the letters of the alphabet in such a way that $h(\sigma_{i_1}\sigma_{i_2}\sigma_{i_3}) = \widehat{\sigma_{i_1}} + \widehat{\sigma_{i_2}}$ is an injection. Table 2 shows the numbers of letter occurrences in the first and second positions in the identifiers. These numbers indicate, that to make our assignment task easier, we should begin with the most "sensitive" letters, i.e. from those with the highest frequencies. These letters are: `n`, `s`, `a`, `i`, `o` and `t`.

If we assign the value 0 to letter `n`, the value 1 to letters `s`, `i`, `o`, and the value 3 to letter `a`, the hash addresses 1 and 2 for identifiers `int` and `sin`, respectively, are fixed (see column 1 of Table 3, where □ denotes values not yet determined). By setting

Table 3
Computing the addresses

	$\widehat{\sigma_{i_1}}$	+	$\widehat{\sigma_{i_2}}$		$\widehat{\sigma_{i_1}}$	+	$\widehat{\sigma_{i_2}}$		$\widehat{\sigma_{i_1}}$	+	$\widehat{\sigma_{i_2}}$
exp	□	+	□	exp	□	+	□	exp	0	+	4
cos	□	+	1	cos	□	+	1	cos	4	+	1
log	□	+	1	log	□	+	1	log	5	+	1
atn	3	+	□	atn	3	+	0	atn	3	+	0
tng	□	+	0	tng	0	+	0	tng	0	+	0
int	1	+	0	int	1	+	0	int	1	+	0
abs	3	+	□	abs	3	+	□	abs	3	+	4
sgn	1	+	□	sgn	1	+	□	sgn	1	+	7
sqr	1	+	□	sqr	1	+	□	sqr	1	+	8
sin	1	+	1	sin	1	+	1	sin	1	+	1
rnd	□	+	0	rnd	□	+	0	rnd	10	+	0

$\widehat{\mathtt{t}} = 0$, we get $h(\mathtt{tng}) = 0$ and $h(\mathtt{atn}) = 3$ (column 2 of Table 3). Now we may assign arbitrary values to the remaining letters, as each of them is used only once in computing the hash function. This means that we can place all other words except `int`, `sin`, `tng` and `atn` in any position in the hash table. In column 3 of Table 3 we have placed the remaining identifiers in the order they appear in the input set. (Note that we generally need to assign values to each letter of the alphabet, not just to each letter which occurs in a used position in some key. This ensures that any potential key, not only correct keys, will be hashed to some address. For brevity we will often only list used letters, with other letters being assigned arbitrary values, typically zero.)

Note that, with the same ease, we can find an ordered minimal perfect hash function for the given set S_α. Namely, by assigning $\widehat{\mathtt{n}} = 3$, $\widehat{\mathtt{s}} = 7$, $\widehat{\mathtt{i}} = 2$, $\widehat{\mathtt{t}} = 1$ and $\widehat{\mathtt{a}} = 2$, we obtain $h(\mathtt{atn}) = 3$, $h(\mathtt{tng}) = 4$, $h(\mathtt{int}) = 5$ and $h(\mathtt{sin}) = 9$. We may then place with no difficulty `exp` at address 0, `cos` at address 1, `log` at address 2, and so on.

Obviously, we were successful in our job of finding a minimal perfect hash function quite quickly because the input set of keys was small. When this set gets larger, the frequencies of letters influencing hash addresses increase. The assignment of an integer value to a frequently used letter, say `a`, may fix the hash values for several keys. It may happen that some of these keys collide with each other in that the same address is obtained for them. In such a case we need to change the assignments previously done, and continue the process with the next value for letter `a`. Essentially, we face here a problem of an exhaustive search of a space of assignments. For large sets of keys, such a space can be enormous. Furthermore, for a given function, there may be no satisfactory assignment. This means that naive algorithms may fail after doing exponentially large amounts of work.

The second problem we may confront in finding a minimal perfect hash function concerns the family of candidate functions. If the example set of keys is slightly changed into $S_\alpha = \{\mathtt{exp}, \mathtt{cos}, \mathtt{log}, \mathtt{atn}, \mathtt{tan}, \mathtt{int}, \mathtt{abs}, \mathtt{sgn}, \mathtt{sqr}, \mathtt{sin}, \mathtt{rnd}\}$, then our list of functions becomes useless as the identifiers `tan` and `atn` exclude the previous candidates $h(\sigma_{i_1}\sigma_{i_2}\sigma_{i_3}) = \widehat{\sigma_{i_1}} + \widehat{\sigma_{i_2}}$ and $h(\sigma_{i_1}\sigma_{i_2}\sigma_{i_3}) = \widehat{\sigma_{i_1}} + \widehat{\sigma_{i_2}} + \widehat{\sigma_{i_3}}$. To overcome this

difficulty, some hashing schemes apply a single "general" function instead of a family of functions. Such a function is parametrized, and its "generality" comes from a high probability of successfully tuning its parameters to the given input key set.

1.3. Space and time requirements

Important issues concerning perfect hashing are:

- the space required for the program which computes the hash value,
- the evaluation time of this program.

By the size of a program we mean the length of the program, measured in bits.

Definition 1.1 (Mehlhorn [74]). Given the universe $U = \{0, 1, \ldots, u-1\}$, a class H of functions $h: U \to [0, m-1]$, is called (u, m, n)-perfect if for every $S \subseteq U$, $|S| = n$, there is $h \in H$ such that h is perfect for S.

Both lower and upper bounds are available on the length of a program for computing a perfect hash function. A lower bound is established by deriving a lower bound on $|H|$, i.e. on the number of different programs required. Based on this bound, the minimum length L of a binary string to code these programs is determined. Note that the number of different strings (programs) of length not exceeding L is less than 2^{L+1}.

Theorem 1.1 (Mehlhorn [74, Theorem III.2.3.6]). *Let $u, m, n \in N$ and let H be a (u, m, n)-perfect class of hash functions. Then*

(a)

$$|H| \geqslant \binom{u}{n} \Big/ \left(\frac{u}{m}\right)^n \binom{m}{n},$$

(b)

$$|H| \geqslant \frac{\log u}{\log m},$$

(c) *There is at least one $S \subseteq U$, $|S| = n$, such that the length of the shortest program which computes a perfect hash function for S is*

$$\max\left(\frac{n(n-1)}{2m \ln 2} - \frac{n(n-1)}{2u \ln 2}\left(1 - \frac{n-1}{u}\right), \log\log u - \log\log m\right) - 1.$$

Proof. (a) The number of distinct subsets of U of size n is $\binom{u}{n}$. Any perfect hash function can be perfect for at most $(\frac{u}{m})^n \binom{m}{n}$ subsets. This result comes from considering the number of elements that can be hashed into a given slot of a hash table by a given hash function, and then maximizing the product of these numbers for all m slots. There are $\binom{m}{n}$ ways to select n out of m places in the hash table. Denote by $h^{-1}(i)$ the subset of elements of U hashed by some hash function h into the ith location of the hash

table. Thus the number of subsets for which h can be perfect is bounded by

$$\sum_{0 \leqslant i_1 < i_2 < \cdots < i_n < m} |h^{-1}(i_1)| \times |h^{-1}(i_2)| \times \cdots \times |h^{-1}(i_n)|.$$

The above expression is maximal when all $h^{-1}(i)$'s are equal and set to be u/m. The maximal value is equal to $(\frac{u}{m})^n \binom{m}{n}$. Thus the number of distinct hash functions must be at least

$$|H| \geqslant \binom{u}{n} \Big/ \left(\frac{u}{m}\right)^n \binom{m}{n}.$$

(b) Consider a class of hash functions $H = \{h_1, h_2, \ldots, h_t\}$. It is possible to construct $U_i \subseteq U$, $0 \leqslant i \leqslant t$, such that for every $S \subseteq U$, $|S \cap U_i| \geqslant 2$ (i.e. S contains at least two elements from the set U_i), the functions $h_1, h_2, \ldots, h_i$ are not perfect for S. Consequently, we need $|U_t| \leqslant 1$. Let $U_0 = U$ and $U_{i+1} = U_i \cap h_{i+1}^{-1}(j)$ for $i < t$, where j is such that $|U_i \cap h_{i+1}^{-1}(j)| \geqslant |U_i \cap h_{i+1}^{-1}(k)|$ for every $k \in [0, m-1]$. Then $|U_i| > |U_{i+1}| \geqslant |U_i|/m$ and hence $|U_{i+1}| \geqslant u/m^{i+1}$. If $t \geqslant \log u / \log m$ then $|U_{t-1}| \geqslant u/m^{\log_m u} = 1$ and $|U_t| \geqslant 1/m$. Hence $t \geqslant \log u / \log m$ hash functions are required to hash perfectly any subset of U constructed accordingly to the above rules.

(c) Denote the lower bounds derived in parts (a) and (b) as b_1 and b_2. As already mentioned, there are at most 2^{L+1} different binary programs of length $\leqslant L$. Since a different program is required for each function, there is at least one set $S \subseteq U$, $|S| = n$, such that the shortest program computing a perfect hash function for S has length $\max(\log b_1, \log b_2) - 1$. We have

$$\log b_2 = \log \log u - \log \log m$$

and

$$\begin{aligned}
\log b_1 &= \log \left(\frac{u(u-1)\cdots(u-n+1)}{u^n} \Big/ \frac{m(m-1)\cdots(m-n+1)}{m^n} \right) \\
&= \left[\sum_{i=0}^{n-1} \ln \left(1 - \frac{i}{u}\right) - \sum_{i=0}^{n-1} \ln \left(1 - \frac{i}{m}\right) \right] \Big/ \ln 2 \\
&\geqslant \left[-\left(1 - \frac{n-1}{u}\right) \sum_{i=0}^{n-1} \frac{i}{u} + \sum_{i=0}^{n-1} \frac{i}{m} \right] \Big/ \ln 2 \\
&= \frac{n(n-1)}{2m \ln 2} - \frac{n(n-1)}{2u \ln 2} \left(1 - \frac{n-1}{u}\right).
\end{aligned}$$

In the last but one step of the above derivation the following estimates are used: $\ln(1 - i/u) \geqslant -(1 - (n-1)/u)i/u$ and $-\ln(1 - i/m) \geqslant i/m$ (cf. [74, Appendix]). □

Less formally, result (c) means that, for at least one $S \subseteq U$, the length of the shortest program which computes a perfect hash function is $\Omega((\beta/(2 \ln 2))n + \log \log u)$ bits. There is an initial cost of $\log \log u$ due to the size of the universe, and there is an incremental cost of $\beta/(2 \ln 2)$ bits per element stored, where $\beta = n/m$ is the load factor of the hash table.

An upper bound was proved by Mehlhorn [74, Theorem III.2.3.8] by explicitly constructing a program of size $O(\beta n + \log\log u)$ bits. Although this program almost achieves the established lower bound, it needs exponential space and time to evaluate the hash function, and as a consequence it is not practical.

To give a practical, constructive upper bound we describe a 1-probe hash scheme proposed by Fredman et al. [50], and slightly improved by Mehlhorn [74], which we denote by FKS. Its space complexity is $O(n \log n + \log\log u)$ and it allows evaluation of the hash function in $O(1)$ time.

We shall use a random access machine (RAM) with the uniform cost measure as the computing model [1, 29]. In our model every memory register contains a $(\log u)$-bit word. Words can be added, subtracted, multiplied, and (integer) divided. All these operations take unit time. Furthermore, an array access to a word is done in unit time, and index computations are permitted. In the following discussion we assume that S is a subset of universe $U = \{0, 1, \ldots, u-1\}$, with $u = |U|$ being prime.

The FKS method considers a class of linear congruential hash functions of the form $h : x \mapsto (ax \bmod u) \bmod m$, where $a \in [1, u-1]$ is a multiplier and m is the size of the hash table. Let $s_i = |S_i|$, where $S_i = \{x : x \in S \text{ and } (ax \bmod u) \bmod m = i\}$ for $i \in [0, m-1]$, i.e. s_i denotes the number of keys in S hashed onto the ith table location.

Theorem 1.2 (Fredman et al. [50, Lemma 1]). *Given $S \subseteq U$, $|S| = n$, when $m \geqslant n$ and x is restricted to S, then there exists $a \in [1, u-1]$ such that*

$$\sum_{i=0}^{m-1} \binom{s_i}{2} < \frac{n(n-1)}{m}.$$

Proof. We show that

$$\sum_{a=1}^{u-1} \sum_{i=0}^{m-1} \binom{s_i}{2} < \frac{un(n-1)}{m} \tag{1.3}$$

from which the theorem follows. Note that $\binom{s_i}{2}$ gives the number of colliding pairs of keys at the ith table location. The sum in (1.3) can be rewritten as

$$\begin{aligned} \sum_{a=1}^{u-1} \sum_{i=0}^{m-1} \binom{s_i}{2} &= \sum_{a=1}^{u-1} \sum_{i=0}^{m-1} |\{(x, y) : x, y \in S,\ x \neq y,\ h(x) = h(y) = i\}| \\ &= \sum_{x, y \in S,\, x \neq y} |\{a : a(x-y) \bmod u \in X\}| \end{aligned} \tag{1.4}$$

where $X = \{m, 2m, 3m, \ldots, u-m, u-2m, u-3m, \ldots\}$. Thus, the sum is the number of $a \in [1, u-1]$ satisfying the condition specified in (1.4). It is easy to see that for a single pair of distinct keys, x and y, the number of a's satisfying that condition cannot exceed $2u/m$. Taking the sum over the $\binom{n}{2}$ possible pairs, we get the bound $un(n-1)/m$ on the sum in (1.3). □

We use Theorem 1.2 in two cases: $m = n$; and $m = n(n-1) + 1 = O(n^2)$.

Corollary 1.1 (Fredman et al. [50, Corollary 1]). *There exists* $a \in [1, u-1]$ *such that for* $h: x \mapsto (ax \bmod u) \bmod n$, $\sum_{i=0}^{n-1} (s_i)^2 < 3n$.

Proof. By the result of Theorem 1.2 for $m = n$ we have

$$\sum_{i=0}^{n-1} (s_i)^2 < 2(n-1) + \sum_{i=0}^{n-1} s_i = 3n - 2$$

from which the inequality follows. □

Corollary 1.2 (Mehlhorn [74, Corollary III.2.3.10b]). *There exists* $a \in [1, u-1]$ *such that the function* $h: x \mapsto (ax \bmod u) \bmod (n(n-1)+1)$ *operates injectively on* S.

Proof. Theorem 1.2 for $m = n(n-1)+1$, when combined with the observation that $\sum_{i=0}^{n(n-1)} s_i = n$, implies that there exists $a \in [1, u-1]$ such that $\sum_{i=0}^{n(n-1)} (s_i)^2 < n+2$. Now note that $s_i \in N_0$ for all i. Thus $s_i \geqslant 2$ for some i gives $\sum_{i=0}^{n(n-1)} (s_i)^2 \geqslant n+2$, a contradiction. So we conclude that $s_i \leqslant 1$ for all i, i.e. h operates injectively on S. □

Note. It is easy to see that Corollary 1.2 can be proved in the same way for $m = n^2$ (cf. [50, Corollary 2]).

Given a set S, $S \subseteq U$, $|S| = n$, the FKS scheme for representing the set S works as follows. The *primary hash function* $h: x \mapsto (ax \bmod u) \bmod n$, where a is chosen as in Corollary 1.1, partitions S into a number of subsets called *collision buckets*, S_i, $i = 0, 1, \ldots, n-1$, $|S_i| = s_i$. Each collision bucket S_i is then resolved by using a *secondary perfect hash function* $h_i: x \mapsto (a_i x \bmod u) \bmod (s_i(s_i-1)+1)$ provided by Corollary 1.2. Let $c_i = s_i(s_i-1)+1$ be the size of the hash table for bucket S_i. Then the key $x \in S$ is stored in location $C_i + h_i(x)$ where $C_i = \sum_{j=0}^{i-1} c_j$. Corollary 1.1 enables us to store all n keys of S within a table of size $3n$, say $D[0 \ldots 3n-1]$.

Example 1.1. Let $S = \{2, 4, 5, 15, 18, 30\}$, $n = 6$, $u = 31$, $a = 2$. The representation for S consists of tables $A[0 \ldots 5]$, $c[0 \ldots 5]$, $C[0 \ldots 5]$ and $D[0 \ldots 7]$ (Fig. 1). Tables A and c contain the parameters a_i and c_i for the secondary hash functions. Table C contains the locations $C_i = \sum_{j=0}^{i-1} c_j$ which separate elements from different collision buckets in D. The keys from S are stored in table D. A query for 5 proceeds as follows. The primary hash function gives index $i = (2 \times 5 \bmod 31) \bmod 6 = 4$. Using this index we access tables A and c getting the parameters $a_4 = 7$ and $c_4 = 3$ for the secondary hash function, which gives $j = (7 \times 5 \bmod 31) \bmod 3 = 1$. The key 5 is found in D at location $C_i + j = 2 + 1 = 3$.

In the first step, using h, a bucket number in which x can be contained is computed. It requires three accesses to parameters, one multiplication, and two (integer) divisions of $(\log u)$-bit words. In the second step, the location $C_i + h_i(x)$ of x is determined. Provided that C_i is computed in advance, this step requires three accesses to parameters and array

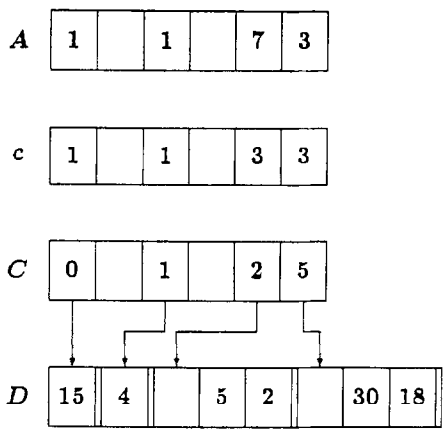

Fig. 1. The FKS scheme.

elements, two multiplications, two (integer) divisions, and three additive operations. All operations, as before, are carried out on $(\log u)$-bit words. Lastly, one probe for key x in the location $C_i + h_i(x)$ of table D is done. To conclude, a membership query for a key $x \in S$ with this scheme takes $O(1)$ time.

The FKS compound hash function $h_i \circ h$ requires storing the parameters a, u and n for the primary function h, the tables $A[0 \ldots n-1]$ and $c[0 \ldots n-1]$ containing the parameters a_i and c_i for the secondary functions h_i, and a table $C[0 \ldots n-1]$ listing the offsets C_i in D. Thus the description of this perfect function needs $(3n + O(1))$ locations, or $(3n + O(1)) \log u$ bits. In addition, $3n \log u$ bits in table D are required to store the keys of the set S. The space complexity of the function exceeds the lower bound $\Omega(n + \log \log u)$ bits given in Theorem 1.1. The FKS scheme can be modified however, so that its storage requirement is reduced. We discuss this modification later.

A disadvantage of the FKS approach, as described above, is that the time to construct deterministically the representation for S might be $O(nu)$ in the worst case. By making use of exhaustive search, we can find a suitable a for the h function in time $O(nu)$, and a_i's for the secondary functions in time $\sum_{i=0}^{n-1} s_i(u-1) = O(nu)$. The following variants of Corollaries 1.1 and 1.2 allow us to construct the representation for S in random time $O(n)$.

Corollary 1.3 (Fredman et al. [50, Corollary 3]). *For at least one half of the values* $a \in [1, u-1]$, *the following inequality is satisfied:* $\sum_{i=0}^{n-1} (s_i)^2 < 5n$.

Proof. Setting $m = n$ in the inequality (1.3) gives

$$\sum_{a=1}^{u-1} \left(\sum_{i=0}^{n-1} (s_i)^2 - n \right) < 2u(n-1). \tag{1.5}$$

Since, for all a's, the terms $\sum_{i=0}^{n-1} (s_i)^2 - n$ are nonnegative, at most one half of these terms in sum (1.5) can exceed twice the average value, $2(n-1)$. Thus for at least one

half of the a's between 1 and $u-1$ we must have $\sum_{i=0}^{n-1}(s_i)^2 - n < 4(n-1)$, which implies the corollary. □

Corollary 1.4 (Mehlhorn [74, Corollary III.2.3.10b]). *For at least one half of the values $a \in [1, u-1]$, the function $h: x \mapsto (ax \bmod u) \bmod (2n(n-1)+1)$ operates injectively on S.*

Proof. Setting $m = 2n(n-1)+1$ in the inequality (1.3) gives

$$\sum_{a=1}^{u-1}\left(\sum_{i=0}^{2n(n-1)}(s_i)^2 - n\right) < \frac{2un(n-1)}{2n(n-1)+1} < u. \tag{1.6}$$

We then proceed as in the proof of Corollary 1.3. □

Note. Corollary 1.4 also holds for $m = 2n^2$ (cf. [50, Corollary 4]).

Based on Corollaries 1.3 and 1.4, we represent the set S as before, except that we now allocate space $2s_i(s_i-1)+1$ to store the keys of bucket S_i. The multipliers a and a_i's are selected at random from the range $[1, u-1]$ until suitable values are found. Since the probability that a particular choice for a or a_i is suitable exceeds $\frac{1}{2}$, the expected number of tries to find each parameter is at most two. Thus the total random time to find a and a_i's for $i \in [0, n-1]$ is $O(n)$ (more precisely, $j \times n$ steps are executed with probability not greater than 2^{-j}).

It must be noted, however, that this short construction time is achieved at the expense of space. To store the keys of S we need now $\sum_{i=0}^{n-1}(2s_i(s_i-1)+1) = 2\sum_{i=0}^{n-1}(s_i)^2 - n = 9n \log u$ instead of $3n \log u$ bits in table D.

The following theorem allows us to bound the deterministic construction time by $O(n^3 \log u)$. As we shall see, the theorem is also useful for reducing the storage requirement of the scheme.

Theorem 1.3 (Fredman et al. [50, Lemma 2]). *Given a set $S \subseteq U$, $|S| = n$, there exists a prime $q < n^2 \log u$ such that the function $\zeta: x \mapsto x \bmod q$ is perfect for S, i.e. for any $x_i, x_j \in S$, $i \neq j$, $x_i \bmod q \neq x_j \bmod q$.*

Proof. Let $S = \{x_1, x_2, \ldots, x_n\}$, and let $t = \prod_{i<j}(x_i - x_j) \prod_i x_i$. Then $\log|t| \leqslant \binom{n+1}{2} \log u$. The prime number theorem gives $\log(\prod_{q<x,\, q \text{ prime}} q) = x + o(x)$, so we conclude that some prime $q < n^2 \log u$ cannot divide t. This prime q satisfies the theorem. □

As we have already mentioned, the worst case time to find the representation for S by a deterministic algorithm is $O(nu)$. If $u < n^2 \log u$, then $O(nu) = O(n^3 \log u)$. If $u \geqslant n^2 \log u$, then in time $O(nq) = O(n^3 \log u)$ we compute a prime q satisfying Theorem 1.3. Now the location where a key $x \in S$ gets stored in the representation for S is determined by using the value $\zeta(x) = x \bmod q$ instead of x. It is worth noting that, in effect, this preprocessing function ζ replaces the original universe $[0, u-1]$

with a smaller universe $[0, q-1]$, where $q < n^2 \log u$. Although the time to construct the representation for S is reduced to $O(n^3 \log u)$, membership query time is worse, as now a triple compound hash function $h_i \circ h \circ \zeta$ must be evaluated.

We are now ready to present the promised modification of the FKS scheme which brings down the representation space to $O(n \log n + \log \log u)$ bits. The process of representing a set S in the modified scheme consists of four basic steps [87]. In the first step, a preprocessing function ρ reduces the universe of keys from size u to n^2. The next two steps apply functions h and h_i as described before. The last step stores set S in a minimum amount of space, i.e. in $n \log u$ bits.

1. The function ρ maps S into $[0, n^2 - 1]$ without collisions. Theorem 1.3 and Corollary 1.2 show that we can find a suitable $\rho : x \mapsto (a_\rho x \bmod q) \bmod n^2$, with $a_\rho < q < n^2 \log u$, where q is prime, for this mapping. Due to this preprocessing, all numbers involved in the further steps are bounded by n^2, reducing their length from $O(\log u)$ to $O(\log n)$. Let $S' = \rho(S)$.
2. The primary hash function h maps S' into $[0, n-1]$ so that the sum of the sizes of the hash tables for the buckets is less than $3n$. Corollary 1.1 shows that it is possible to find $h : x \mapsto (ax \bmod p) \bmod n$, where p is any prime greater than $n^2 - 1$, and $a \in [1, p-1]$, satisfying that condition.
3. The secondary hash function h_i scatters perfectly the keys of each bucket S_i over the hash table of size $c_i = s_i(s_i - 1) + 1$. The appropriate function $h_i : x \mapsto (a_i x \bmod p) \bmod c_i$, where $a_i \in [1, p-1]$, is provided by Corollary 1.2. As shown in Corollary 1.1, given a key $x \in S$ the address $t_i = C_i + h_i(\rho(x))$ could be used to find x in the table of keys of size $3n \log u$ bits. However, to save space we reference with t_i a compression table $P[0 \ldots 3n-1]$ of only $3n \log n$ bits, which in turn gives us the index of x (if present) within the (compact) table of keys $D[0 \ldots n-1]$ of size $n \log u$ bits.
4. In this step, the keys of S are stored with no vacant locations in table $D[0 \ldots n-1]$. For each key $x \in S$ its index in D is stored in the element $P[C_i + h_i(\rho(x))]$ of the compression table.

Let us analyze the space requirement for this scheme. The function ρ requires the parameters a_ρ and q, each of $2 \log n + \log \log u$ bits. The primary hash function h needs the parameters a and p of total length $4 \log n$ bits. To compute the secondary hash functions h_i we need to access the tables $A[0 \ldots n-1]$ and $c[0 \ldots n-1]$ storing the parameters a_i and c_i, and then to use the table $C[0 \ldots n-1]$ listing the locations C_i, which separate elements from different buckets. The total space of tables A, c, and C is $3n \log n$ bits. The compression table P requires $3n \log n$ bits. Summing up, the description of this perfect hash function needs $6n \log n + 8 \log n + 2 \log \log u = O(n \log n + \log \log u)$ bits. In addition, we need $n \log u$ bits in table D to store the keys of S.

The perfect hash function defined by $h_i \circ h \circ \rho$ maintains $O(1)$ search time. Only ρ involves operations on $(\log u)$-bit words, the remaining functions use values bounded by n^2, i.e. of length $O(n)$ bits. The scheme can be constructed by making use of the deterministic algorithm in time $O(n^3 \log u)$ in the worst case.

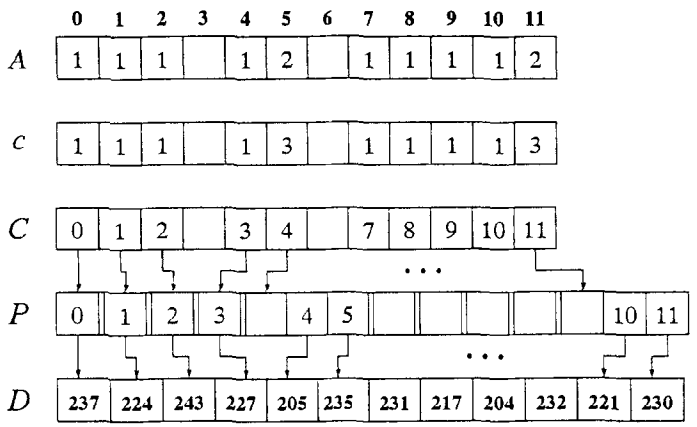

Fig. 2. The modified FKS scheme.

Example 1.2. Let us design the modified FKS scheme for the set containing three-letter abbreviations for the months of the year, $S_x = \{$JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC$\}$. The abbreviations are converted into unique integers $x \in S$, where $S = \{217, 205, 224, 227, 231, 237, 235, 221, 232, 230, 243, 204\}$, by adding ASCII codings of the three letters in the key. For key JAN, $x = 74 + 65 + 78 = 217$. As the preprocessing function in the first step we use $\rho : x \mapsto (14x \bmod 167) \bmod 144$ which transforms S into $S' = \{14, 134, 112, 10, 66, 6, 122, 70, 80, 52, 90, 120\}$. In the second step the primary hash function $h : x \mapsto (4x \bmod 149) \bmod 12$ partitions S', and thereby S, into the following collision buckets $S_0 = \{237\}$, $S_1 = \{224\}$, $S_2 = \{243\}$, $S_3 = \emptyset$, $S_4 = \{227\}$, $S_5 = \{205, 235\}$, $S_6 = \emptyset$, $S_7 = \{231\}$, $S_8 = \{217\}$, $S_9 = \{204\}$, $S_{10} = \{232\}$ and $S_{11} = \{221, 230\}$. In the third step the secondary functions, $h_i : x \mapsto (a_i x \bmod 149) \bmod c_i$, are found. For $i = 0, 1, 2, 4, 7, 8, 9, 10$, we select $a_i = 1$ and $c_i = 1$; for $i = 5, 11$, $a_i = 2$ and $c_i = 3$. The scheme is shown in Fig. 2.

In what follows, we shall present the method designed by Schmidt and Siegel [87] which implements a variation of the FKS scheme in optimal space and maintains O(1) evaluation time. We need the following lemma.

Lemma 1.1 (Slot and van Emde Boas [92, Lemma 5]). *For a given collection of disjoint sets S_j, $j = 1, 2, \ldots, t$, each of which is a subset of U, $|U| = u$, $t \leqslant u$, there exists a value a such that the functions $h^j : x \mapsto (ax \bmod u) \bmod c_j$, with $c_j = 2s_j(s_j - 1) + 1$, operate injectively on one half of the sets S_j.*

Proof. The proof of the lemma proceeds in a much the same way as the proof of Theorem 1.2. Given $a \in [1, u-1]$, we say that there is a collision between keys $x, y \in S_j$, $x \neq y$, if $h^j(x) = h^j(y) = k$ for some $k \in [0, c_j - 1]$. Let $X_1 = \{c_j, 2c_j, 3c_j, \ldots, u - c_j, u - 2c_j, u - 3c_j, \ldots\}$. We estimate the total number of collisions between keys for all

sets S_j and all values $a \in [1, u-1]$:

$$\sum_{a=1}^{u-1} \sum_{j=1}^{t} \sum_{k=0}^{c_j-1} |\{(x,y): x,y \in S_j,\ x \neq y,\ h^j(x) = h^j(y) = k\}|$$

$$= \sum_{j=1}^{t} \sum_{a=1}^{u-1} \sum_{k=0}^{c_j-1} |\{(x,y): x,y \in S_j,\ x \neq y,\ h^j(x) = h^j(y) = k\}|$$

$$= \sum_{j=1}^{t} \sum_{\substack{x,y \in S_j, \\ x \neq y}} |\{a: a(x-y) \bmod u \in X_1\}|$$

$$< \sum_{j=1}^{t} \frac{2u\binom{s_j}{2}}{2s_j(s_j-1)+1} < \frac{ut}{2}.$$

Since there exist $u-1$ possible values for a there must exist a value a which gives no collisions for at least $t/2$ sets S_j. This completes the proof. □

Lemma 1.1 enables us to reduce the number of multipliers a_i for the secondary hash functions used in the representation for S. Namely, there exists a value a_0 such that h_0^i operates injectively on at least one half of all S_i, $i = 0, 1, \ldots, n-1$. Again by Lemma 1.1 there exists a second multiplier a_1 such that h_1^i operates injectively on one half of those S_i that were not resolved using a_0. Proceeding in this way we obtain at most $\lfloor \log n \rfloor + 1$ different a_z multipliers, where a_z is the zth multiplier servicing about $1/2^{z+1}$ of S_i's.

The optimal scheme by Schmidt and Siegel uses the triple compound hash function $h_z \circ h \circ \rho$ as described before with two modifications: (a) the space assigned for each collision bucket S_i, $i = 0, 1, \ldots, n-1$, is $c_i = 2s_i(s_i-1)+1$; and (b) only $\lfloor \log n \rfloor + 1$ multipliers a_z for the secondary hash functions are stored. The crucial issue is how to represent the tables A, c, C and P in space $O(n)$ maintaining simultaneously $O(1)$ membership query time. To achieve this goal Schmidt and Siegel employ some compact encoding techniques. The following decoding operations are assumed to be performed in constant time on $O(\log n)$-bit words:

(1) Extract a subsequence of bits from a word.
(2) Concatenate two bit strings of altogether $O(\log n)$ bits.
(3) Compute the bit-index of the kth zero in a word.
(4) Count the number of consecutive 1's in a word, from the front.
(5) Count the number of 0's in a word.
(6) Access a few constants.

The operations (1), (2) and (6) are a matter of arithmetic. The remaining operations for a given $(\log n)$-bit word can be accomplished by breaking it into words of $(\log n)/2$ bits (padded with leading 0's), and by using these words to access decoder arrays of $2^{(\log n)/2} = \sqrt{n}$ words. In particular, operation (3) requires for each k, $0 < k \leqslant (\log n)/2$, an array of $\sqrt{n}$ words. Thus, to write down the decoder arrays for operation (3) we need $k\sqrt{n}\log n = O(n)$ bits. For operations (4) and (5) two decoder arrays of size $\sqrt{n}\log n$ bits each are needed. Note that a string of $2\log n$ bits which can be parted

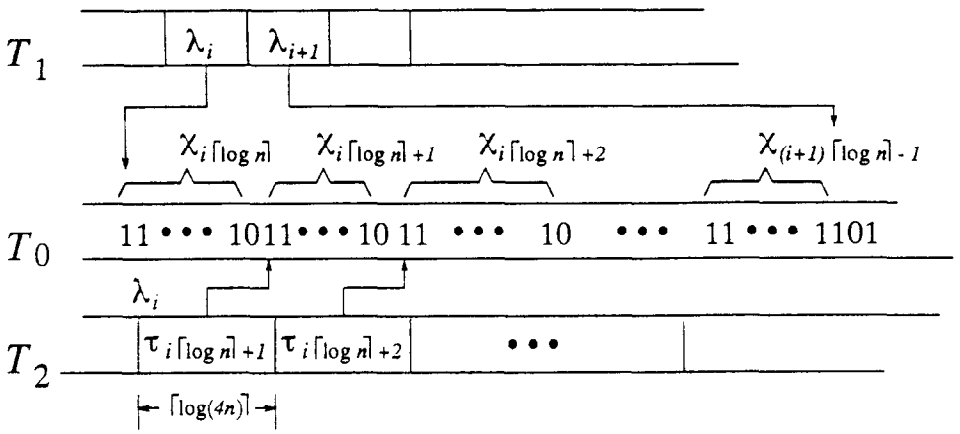

Fig. 3. Schmidt and Siegel's encoding for $\lambda_{i+1} - \lambda_i > \lceil \log(4n) \rceil^2$.

into two $(\log n)$-bit words requires up to four decoder array accesses to compute the location of the kth 0.

Now we show how the table C containing the values $C_i = \sum_{j=0}^{i-1} c_j = \sum_{j=0}^{i-1} (2s_j^h(s_j^h - 1) + 1)$ is encoded. Let χ_j denote the unary string of c_j 1's that encodes c_j. The strings χ_j are stored in table T_0, with 0's separating two consecutive χ_j's. As each χ_j requires exactly c_j bits, by Corollary 1.1, the length of T_0 is $\sum_{j=0}^{n-1} (|\chi_j| + 1) = \sum_{j=0}^{n-1} c_j + n \leqslant 4n$ bits. We assume that T_0 is stored as a sequence of $\lceil \log(4n) \rceil$-bit words and, because of operations (1) and (2), each bit in T_0 is addressable.

We divide the n strings encoded in T_0 into $\lceil n/\lceil \log n \rceil \rceil$ groups of $\lceil \log n \rceil$ strings each, except possibly the last group. (We modify the original encoding of Schmidt and Siegel dividing the strings in T_0 into $\lceil \log n \rceil$-string groups, and each group into $\lceil \log \log n \rceil$-string subgroups.) Let λ_i, $0 \leqslant \lambda_i < 4n$, be the address (index) of the starting point of the first string in the ith group, $\chi_{i\lceil \log n \rceil}$, $i = 0, 1, 2, \ldots, \lceil n/\lceil \log n \rceil \rceil - 1$.

A second table T_1 contains the indices λ_i, $i = 1, 2, \ldots, \lceil n/\lceil \log n \rceil \rceil$, in binary, stored as $\lceil \log(4n) \rceil$-bit words ($\lambda_0 = 0$ is not stored). The size of table T_1 is $\lceil n/\lceil \log n \rceil \rceil \times \lceil \log(4n) \rceil = O(n)$ bits. Since $\lambda_i = C_{i\lceil \log n \rceil} + i\lceil \log n \rceil$ we can use this equation and value λ_i from T_1 to compute $C_{i\lceil \log n \rceil}$. If $\lambda_{i+1} - \lambda_i \leqslant 2\lceil \log(4n) \rceil$, the addresses for intermediate c_j, $i\lceil \log n \rceil \leqslant j < (i+1)\lceil \log n \rceil$, which enables us to compute the corresponding C_j, can be decoded in $O(1)$ time via accesses to tables T_0 and T_1, and decoding operations (1)–(6).

In the case when $\lambda_{i+1} - \lambda_i > \lceil \log(4n) \rceil^2$ the third table T_2 (of $4n$ bits) is used. This stores, starting in bit location λ_i, $\lceil \log n \rceil - 1$ binary indices for the starting locations of χ_j in T_0, $i\lceil \log n \rceil < j < (i+1)\lceil \log n \rceil$ (Fig. 3; indices to individual χ_j's are represented by τ_j).

The last case, $2\lceil \log(4n) \rceil < \lambda_{i+1} - \lambda_i \leqslant \lceil \log(4n) \rceil^2$, is handled with an additional level of refinement. Namely, every group of strings is divided into $\lceil \lceil \log n \rceil / \lceil \log \log n \rceil \rceil$ subgroups, each with $\lceil \log \log n \rceil$ strings. Denote by $\eta_{i,j}$ the address in T_0 of the starting point of $\chi_{i\lceil \log n \rceil + j\lceil \log \log n \rceil}$, $j = 1, 2, \ldots, \lceil \lceil \log n \rceil / \lceil \log \log n \rceil \rceil - 1$. Then $\lambda_i < \eta_{i,j} < \lambda_{i+1}$.

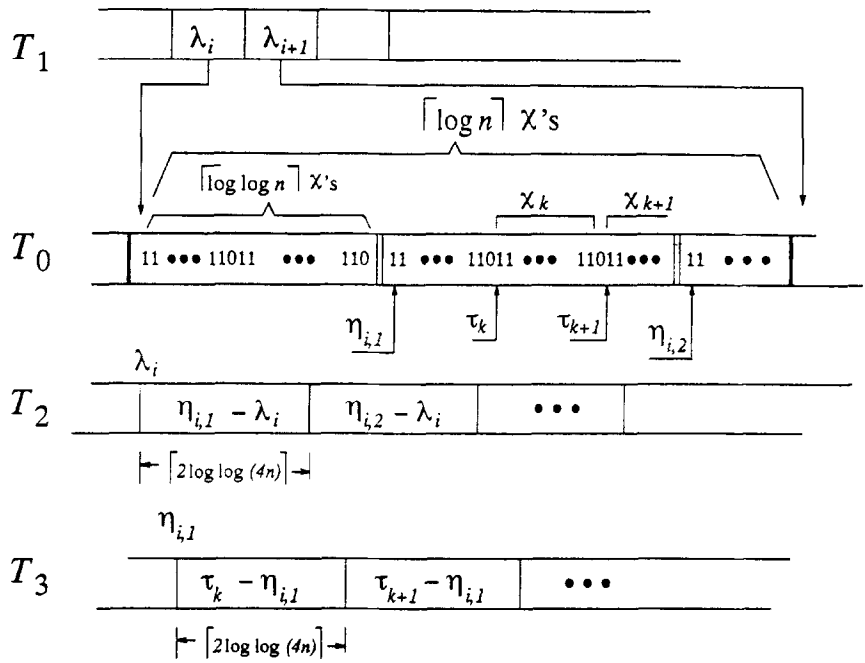

Fig. 4. Schmidt and Siegel's encoding for $2\lceil \log(4n) \rceil < \lambda_{i+1} - \lambda_i \leqslant \lceil \log(4n) \rceil^2$.

The binary offsets $\eta_{i,1} - \lambda_i$, $\eta_{i,2} - \lambda_i, \ldots$ are stored as $(2\lceil \log \log(4n) \rceil)$-bit numbers in T_2, starting at location λ_i. If $\eta_{i,j+1} - \eta_{i,j} \leqslant 2\lceil \log(4n) \rceil$ the information for intermediate c_k, $i\lceil \log n \rceil + j\lceil \log \log n \rceil < k < i\lceil \log n \rceil + (j+1)\lceil \log \log n \rceil$, can be easily decoded from table T_0. Otherwise, the offsets of size $2\lceil \log \log(4n) \rceil$ of all intermediate c_k are placed in a table T_3 (of $4n$ bits) starting at bit location $\eta_{i,j}$. This last encoding requires $(\lceil \log \log n \rceil - 1) \times 2\lceil \log \log(4n) \rceil \leqslant 2\lceil \log(4n) \rceil$ bits. (Fig. 4. illustrates both cases. For $\eta_{i,1} - \eta_{i,0} \leqslant 2\lceil \log(4n) \rceil$, $\eta_{i,0} = \lambda_i$, no additional levels are required. For the second segment, bounded by $\eta_{i,1}$ and $\eta_{i,2}$, offsets $\tau_k - \eta_{i,1}$ of length $2\lceil \log \log(4n) \rceil$ bits to individual χ_k's, are stored in table T_3.)

Example 1.3. In Fig. 5 we show the encoding of table C in the FKS scheme from Example 1.1. The values $c_0 = 1$, $c_1 = 0$, $c_2 = 1$, $c_3 = 0$, $c_4 = 3$ and $c_5 = 3$ are stored in unary in table T_0 of size $4n = 24$ bits. The coding strings are divided into $\lceil n/\lceil \log n \rceil \rceil = \lceil 6/\lceil \log 6 \rceil \rceil = 2$ groups of $\lceil \log n \rceil = \lceil \log 6 \rceil = 3$ strings each. Table T_1 contains the indices λ_i, $i = 1, 2$, to the starting point of strings χ_3 and χ_6 in T_0. Since both $\lambda_1 - \lambda_0$ and $\lambda_2 - \lambda_1$ are less than or equal to $2\lceil \log(24) \rceil = 10$, no further level of indirection is necessary.

It is easy to see that there is no need to store the parameters c_i, $i = 0, 1, \ldots, n-1$, in a separate table. These parameters can be fetched in constant time from the representation of C encoded as described above.

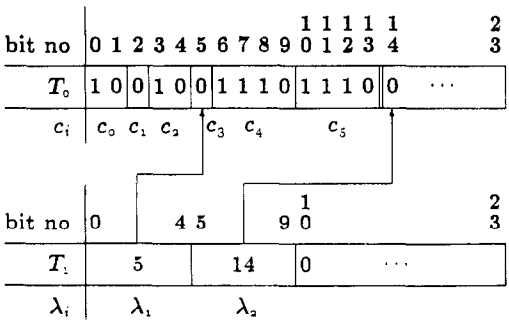

Fig. 5. Encoding of table C.

Tables P and A can be encoded in a similar way. In particular, the multipliers $a_0, a_1, \ldots, a_{\lfloor \log n \rfloor}$ are stored in a $(\lfloor \log n \rfloor + 1)$-word array. Thus, it consists of $(\lfloor \log n \rfloor + 1) \times 2 \log n = O(\log^2 n)$ bits. Table A_0 contains n strings in unary which encode the indices of multipliers assigned to each collision bucket. The ith sequence of bits encodes the integer $j_i \leqslant \lfloor \log n \rfloor$, if a_{j_i} is the multiplier assigned to bucket S_i. By Lemma 1.1, the first multiplier encoded by the string 0 is usable for at least half of the buckets, the second multiplier encoded by the string 10 is usable for at least $\frac{1}{4}$ of the buckets, etc. Thus, the sequence of bits in A_0 comprises at most $n/2$ 0's, $n/4$ 10's, etc., for a total length of no more than $2n$ bits. Using an encoding structure for A similar to that described above, we can retrieve a multiplier assigned to each collision bucket in $O(1)$ time.

To summarize, the space complexity of the hashing scheme of Schmidt and Siegel is $O(n + \log\log u)$. The scheme requires $O(n)$ bits to encode the offsets C_i (table C), the compression table P and the multipliers a_j (table A), and $O(2\log n + \log\log u)$ bits to store the parameters a_ρ and q of the preprocessing function. The scheme combined with the lower bound from Theorem 1.1 gives the following.

Theorem 1.4 (Schmidt and Siegel [87, Theorem 6]). *For a set S of n elements belonging to the universe $U = \{0, 1, \ldots, u-1\}$, there is a constant-time perfect hash function with space complexity $\Theta(n + \log\log u)$.*

Note that the optimal scheme by Schmidt and Siegel is mainly of theoretical importance. The scheme is hard to implement and the constants associated with the evaluation of the hash function are prohibitive.

A separate issue is the space required for perfect hash functions allowing arbitrary arrangement of keys stored in the hash table. We make a distinction between two types of hash functions which allow certain orders to be imposed on keys.

Hash functions of the first type allow only one, quite specific arrangement of keys in the hash table. These schemes are referred to as *ordered perfect hash functions* [19, 22]. The lower bound on the number of bits required to represent an ordered perfect hash function matches the bound for plain perfect hash functions. The best upper bound

known so far is $O(n \log n + \log \log u)$, which can be derived by combining the result of [22] and Theorem 1.3 together with Corollary 1.2.

The second type of hash functions are those that allow recording any permutation of keys in the hash table. We call them *arbitrary ordered perfect hash functions* omitting the word *arbitrary* whenever it is understood. A class of hash functions H_0 is called (u, m, n)-ordered perfect, if for every ordered subset S of universe U there exists $h \in H_0$ which is perfect for S, and $h(x) = i$ if x is the ith element of S.

Theorem 1.5 (Fox et al. [46]). *Let H_0 be a (u, m, n)-ordered-perfect class of hash functions. Then*

$$|H_0| \geqslant n! \binom{u}{n} \Big/ \left(\frac{u}{m}\right)^n \binom{m}{n}.$$

Proof. The number of distinct subsets of U of size n, when the order of elements counts, is

$$\frac{u!}{(u-n)!} = n! \binom{u}{n}.$$

By the argument presented in Theorem 1.1 any hash function can be perfect for at most $(\frac{u}{m})^n \binom{m}{n}$ different sets. Thus the number of distinct hash functions must be at least as large as stated in the theorem. □

From Theorem 1.5 it follows that there is at least one $S \subseteq U$, $|S| = n$, such that the length of the shortest program which computes an ordered perfect hash function for S is

$$\log |H_0| = \log(n!) + \log \left(\frac{\binom{u}{n}}{\left(\frac{u}{m}\right)^n \binom{m}{n}} \right)$$

bits. The factorial function can be estimated using Stirling's approximation

$$n! = \sqrt{2\pi n} e^{\vartheta/(12n)} \left(\frac{n}{e}\right)^n$$

where $0 < \vartheta < 1$ and ϑ depends on n. Thus we have

$$\log n! = \left(n + \frac{1}{2}\right) \log \left(\frac{n}{e}\right) + \frac{1}{2} \log(2\pi e) + O\left(\frac{1}{n}\right).$$

From this and Theorem 1.1 it follows that

$$\log |H_0| \geqslant \left(n + \frac{1}{2}\right) \log \left(\frac{n}{e}\right) + \frac{n(n-1)}{2m \ln 2} - \frac{n(n-1)}{2u \ln 2} \left(1 - \frac{n-1}{u}\right) + O(1).$$

Less formally, combined with Theorem 1.1, this means that $\Omega(n \log n + (\beta/(2 \ln 2))n + \log \log u)$ bits are required to store an ordered perfect hash function, where $n \log n$ is the factor introduced by the order-preserving property. The lower bound is matched by the upper. To see that consider any space optimal perfect hash function, say

h, $|h| = O(n + \log\log u)$, which implies $m = O(n)$. Let $\pi = \langle i_1, i_2, \ldots, i_n \rangle$, $i_j \in \{1,2,\ldots,n\}$ be such a permutation of numbers 1 through n that $\pi(i)$ is equal to the position we want to allocate to key $x_i \in S$, for $i = 1,2,\ldots,n$. Replace the original hash function by $h'(x) = R[h(x)]$, where R is a vector of length m, $R[j] = \pi(i)$ if $h(x_i) = j$. As the space required to record permutation π is no more than $|R| = m\lceil\log(n+1)\rceil = O(n\log n)$, the total space for such a constructed ordered hash function is $O(n\log n + |h|)$ bits.

1.4. Bibliographic remarks

Similar to the trial and error approach described in Section 1.2, Greniewski and Turski used a nonalgorithmic method for finding a perfect hash function to map the reserved words of the KLIPA assembler into an almost minimal hash table [53]. Their hash function took the form $h(x) = Ax + B$, $x \in S$, where A and B are suitable constants.

Mairson confirmed the $\Omega(n + \log\log u)$ lower bound on the length of a perfect hash function program established by Theorem 1.1 using a different argument [72, Theorems 3 and 4]. A few improvements on the space complexity of the FKS scheme were proposed. Fredman et al. presented a variation of the scheme which leads to a description size of $O(n\sqrt{\log n} + \log\log u)$ bits [50]. Slot and van Emde Boas achieved the optimal $O(n + \log\log u)$-bit space but increased the hash function evaluation time to $O(n)$ [92]. This improvement is described in Section 3.2. Jacobs and van Emde Boas reduced the space requirement of the FKS scheme to $O(n\log\log n + \log\log u)$ bits, while maintaining $O(1)$ access time [60].

Chapter 2. Number-theoretical solutions

2.1. Introduction

The solutions to the problem of generating perfect hash functions presented in this chapter came from consideration of theoretical properties of sets of integers. The described algorithms are mainly of historical interest, so we do not present them in detail, but give references to their source. However, we do indicate how their computations behave.

In Section 2.2 two methods for generating perfect hash functions proposed by Sprugnoli are discussed [93]. Section 2.3 presents a method of reciprocal hashing given by Jaeschke [61]. Chang's solution [19] based upon the Chinese remainder theorem is presented in Section 2.4. Sections 2.5–2.7 describe various modifications of Chang's approach. They are proposed by Chang and Lee [25], Chang and Chang [22], and Winters [100], respectively.

2.2. Quotient and remainder reduction

Sprugnoli presented two methods for generating perfect hash functions [93]. Let S be a set of nonnegative integer keys. The first method, called *quotient reduction*, searches

for two integers s and t such that the function $h(x) = \lfloor (x+s)/t \rfloor$, $x \in S$, is a perfect hash function. Sprugnoli proposed an algorithm [93, Algorithm Q] which, for the given form of hash function $h(x)$, finds s and t such that the elements of an arbitrary set S can be stored in the smallest hash table.

Example 2.1. Let $S = \{17, 138, 173, 294, 306, 472, 540, 551, 618\}$. The best [93] quotient reduction perfect hash function for S, in terms of hash table size, is $h(x) = \lfloor (x+25)/64 \rfloor$. The function, however, is not minimal as shown below:

x	17	138	173	294	306	472	540	551	618
$h(x)$	0	2	3	4	5	7	8	9	10

The quotient reduction method is simple and works well for small (up to about 12 keys) and uniformly distributed sets. However, for nonuniform sets this method can give rather sparse tables.

The second method, called *remainder reduction*, overcomes this drawback by scrambling the elements of S first, to get a more uniform distribution, and then applying a quotient reduction perfect hash function to the scrambled set. The remainder reduction method finds a perfect hash function of the form $h(x) = \lfloor ((d+qx) \bmod r)/t \rfloor$, where d, q, r and t are integer parameters. To compute the parameters Sprugnoli devised an algorithm [93, Algorithm T] which is faster than the one used in the quotient reduction method. Unfortunately, the algorithm is not guaranteed to work for all sets of keys. Besides, neither method ensures finding minimal perfect hash functions.

Example 2.2. Again suppose S_α is the set containing three-letter abbreviations for the months of the year: $S_\alpha = \{$JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC$\}$. Let $x_\alpha = \sigma_1\sigma_2\sigma_3$ and let x be the number obtained by taking the EBCDIC coding of σ_2 and σ_3 (using these as base 256 digits, with σ_2 the more significant digit). Thus, if we consider the EBCDIC codings of the second and third characters of every month, we get the set of integers shown in Table 4. It can be seen from the table that these integers are perfectly hashed into a hash table of minimal size by the remainder reduction function $h(x) = \lfloor ((4+3x) \bmod 23)/2 \rfloor$.

As already mentioned, the methods described above can be applied for small sets. For larger sets Sprugnoli suggested the use of *segmentation*, which means dividing the set of keys into a number of segments or buckets. Each bucket contains no more than 12 keys, so that a (quotient or remainder reduction) perfect hash function can be

Table 4
A perfect hashing for the month names

x_α	JAN	FEB	MAR	APR	MAY	JUN	JUL	AUG	SEP	OCT	NOV	DEC
x	49621	50626	49625	55257	49640	58581	58579	58567	50647	50147	55013	50627
$h(x)$	5	6	0	7	11	2	10	4	3	1	9	8

found to store and retrieve keys from the bucket. The distribution of keys into buckets is done by using a *primary* (or *grouping*) *function* h, which can be an ordinary hash function. The keys $x \in S$ for which $h(x) = i$ go into bucket i. For all keys in a bucket, a perfect hash function is generated and its description is associated with the bucket. We give a more detailed discussion of perfect hashing with segmentation in Section 3.

Using this approach Sprugnoli was able to develop a minimal perfect hash function for the 31 most common English words [93].

2.3. Reciprocal hashing

Jaeschke presented a method for creating minimal perfect hash functions, which he called *reciprocal hashing*, based upon the following theorem [61].

Theorem 2.1. *Given a finite set* $S' = \{x'_1, x'_2, \ldots, x'_n\}$ *of pairwise relatively prime integers, there exists a constant* C *such that* h *defined by*

$$h(x) = \lfloor C/x \rfloor \bmod n$$

where $x \in S'$, *is a minimal perfect hash function.*

As it cannot be expected that the keys in an input set are always pairwise relatively prime, Jaeschke gave the following lemma.

Lemma 2.1. *For any set* $S = \{x_1, x_2, \ldots, x_n\}$ *of positive integers there exist two integer constants* D *and* E *such that*

$$x'_1 = Dx_1 + E, \quad x'_2 = Dx_2 + E, \quad \ldots, \quad x'_n = Dx_n + E$$

are pairwise relatively prime.

He proposed two algorithms, which he called Algorithm C and Algorithm DE, which search for the corresponding constants. According to Jaeschke, in a random selection of S's, only two out of 3000 sets required determining the constants D and E. Note however, that in practice keys cannot be expected to be random. As reported by Jaeschke, with respect to running times his method is usually only slightly better than Sprugnoli's methods, but in the case of nonuniformly distributed sets it is essentially better (10 to 1000 times faster). The main disadvantage of Jaeschke's method is the exponential time complexity of Algorithms C and DE, which find the appropriate constants by using an exhaustive search. The algorithms are impractical for sets with more than about 20 elements. Furthermore, the constant C can become very large.

In the method, the input is assumed to be a set of integers. No conversion from character keys to integers was suggested. This task is relatively straightforward to carry out if large integers are acceptable. For example, character strings can be treated as numbers using a base equal to the size of the character set. It must be noted however, that large integers adversely impact the complexity of the calculations in the reciprocal hashing method.

Table 5
Another perfect hashing for the month names

x_α	JAN	FEB	MAR	APR	MAY	JUN	JUL	AUG	SEP	OCT	NOV	DEC
x	22	10	29	32	36	42	40	26	37	35	48	9
x'	269	197	311	329	353	389	377	293	359	347	425	191
$h(x')$	1	3	0	10	4	6	8	5	7	11	9	2

Example 2.3. Consider again the set of abbreviated month names S_α from Example 2.2. If we treat keys as numbers with base 26, we obtain the following set of integers: $S = \{6800, 4188, 8832, 1110, 8839, 7320, 7318, 1229, 12\,990, 10\,238, 9876, 2837\}$. Since the elements of S are not pairwise relatively prime, we compute the constants $D = 6$ and $E = 119$ using Algorithm DE. The resulting set of pairwise relatively prime integers is $S' = \{40\,919, 25\,247, 53\,111, 6779, 53\,153, 44\,039, 44\,027, 7493, 78\,059, 61\,547, 59\,375, 17\,141\}$. Algorithm C examined 13 520 possible C's before a suitable value $C = 1\,055\,357\,680$, giving the minimal perfect hash function for S', was found.

For these keys, we can use a more compact mapping between names and integers. By associating with each letter an integer code: $\hat{\mathrm{A}} = 0$, $\hat{\mathrm{B}} = 1$, and so on, we can transform the names of S_α into unique integers by simply adding the codes associated with each letter. The following set is obtained: $S'' = \{22, 10, 29, 32, 36, 42, 40, 26, 37, 35, 48, 9\}$. The keys in S'' are not pairwise relatively prime, thus we apply Algorithm DE to find suitable constants $D = 6$ and $E = 137$. The resulting set is $S'' = \{269, 197, 311, 329, 353, 389, 377, 293, 359, 347, 425, 191\}$. Next, we compute the constant $C = 6\,766\,425$. To find this constant Algorithm C executed 15 921 iterations. Note that the constants here are smaller. The minimal perfect hash function for S'' is shown in Table 5. Thus, $x_\alpha = \sigma_1\sigma_2\sigma_3$; $x = \widehat{\sigma_1} + \widehat{\sigma_2} + \widehat{\sigma_3}$, $x' = 6x + 137$; and $h(x') = \lfloor 6\,766\,425/x' \rfloor \bmod 12$.

2.4. A Chinese remainder theorem method

Chang [19] developed a similar scheme for generating ordered minimal perfect hash functions. This is based upon the Chinese remainder theorem, which can be stated as follows.

Theorem 2.2 (Chinese remainder theorem). *Given positive integers $x_1, x_2, \ldots, x_n$ and $l_1, l_2, \ldots, l_n$, a constant C can be found such that $C \equiv l_1 \pmod{x_1}$, $C \equiv l_2 \pmod{x_2}$, $\ldots, C \equiv l_n \pmod{x_n}$, if x_i and x_j are relatively prime for all $i \neq j$.*

To apply this theorem to perfect hashing, let us assume that x_i's are keys to be hashed and they are pairwise relatively prime. Let us also set $l_1 = 1$, $l_2 = 2$, and so on, to be the consecutive locations in the hash table. In this case, we may let the ordered minimal perfect hash function be $l_i = C \bmod x_i$, $i = 1, 2, \ldots, n$, or $h(x) = C \bmod x$, $x \in S$, $|S| = n$. Clearly, the function hashes keys from S in ascending order.

The first problem which arises in such a formulation is that we cannot assume that the keys are pairwise relatively prime. Thus Chang proposed transforming x_i's to prime numbers by making use of a prime number function. A function $p(x)$ is called a *prime number function* for $a \leqslant x \leqslant b$, where x, a, b are all positive integers, if $p(x)$ is a prime number for $a \leqslant x \leqslant b$ and $p(x_1) > p(x_2)$ if $x_1 > x_2$. Although no general prime number function is known, some such functions exist [19, 59, 71, 90]:

$p(x)$	Validity range
$x^2 - x + 17$	$1 \leqslant x \leqslant 40$
$x^2 - x + 41$	$1 \leqslant x \leqslant 40$
$x^2 - 81x + 1681$	$41 \leqslant x \leqslant 80$
$x^2 + x + 41$	$1 \leqslant x \leqslant 39$
$x^2 - 79x + 1601$	$40 \leqslant x \leqslant 79$

The second problem is how to find the constant C. Here Chang gave an algorithm, called Algorithm A, for computing the smallest positive constant C which satisfies the Chinese remainder theorem. The algorithm, based on results in number theory, finds this constant in $O(n^2 \log n)$ time. This is polynomial time, in contrast to the exponential time required by Jaeschke's algorithm. Unfortunately, the number of bits to represent C and some other variables required to calculate C is $O(n \log n)$. Even for quite small sets, C can be very large. For example, for the set of the first 64 primes, $C \approx 1.92 \times 10^{124}$ and the binary representation of C requires 413 bits. Besides, as already mentioned, no general method for finding a prime number function exists.

Example 2.4. Let $S = \{4, 5, 7, 9\}$. The elements of S are pairwise relatively prime thus, using Algorithm A, we compute $C = 1417$. We have: $1417 \bmod 4 = 1$, $1417 \bmod 5 = 2$, $1417 \bmod 7 = 3$, and $1417 \bmod 9 = 4$.

Example 2.5. Carrying on with Example 2.3, we subtract 8 from each element of S'', and sort the set in ascending order to obtain $S = \{1, 2, 14, 18, 21, 24, 27, 28, 29, 32, 34, 40\}$. The elements of S are not pairwise relatively prime, so we apply the prime number function $p(x) = x^2 - x + 41$, which conveniently includes S in its domain. Then using Algorithm A, we compute $C \approx 6.85744 \times 10^{30}$. To store this parameter, 103 bits are needed.

Let us observe that the letter-oriented methods for finding minimal perfect hash functions based on extracting distinct pairs of characters from the keys can be applied only for the sets of up to $26 \times 26 = 676$ keys. Also, there exists no algorithm that guarantees to produce distinct pairs for any given set of keys. In the case of larger sets two characters may not be sufficient to distinguish all keys. For example, there are roughly 500 COBOL reserved words and there do not exist two characters for each

word to form 500 distinct extracted pairs (for further discussion of this problem see Section 5.4).

2.5. A letter-oriented scheme

Chang and Lee modified Chang's approach to make it suitable for letter-oriented keys [25]. They assumed that an input consists of a set of keys, where each key is a string of characters over some ordered alphabet Σ, $x_\alpha = \sigma_1\sigma_2\ldots\sigma_l$. They also required that there exist i and j such that the ordered pairs formed by the ith and jth characters of the keys are distinct. Let a distinct pair of characters extracted from the keys be represented by (σ_i, σ_j). Then the minimal perfect hash function which is searched for has the form:

$$h(\sigma_i, \sigma_j) = d(\sigma_i) + (C(\sigma_i) \bmod p(\sigma_j)),$$

where $d(\sigma)$, $C(\sigma)$ and $p(\sigma)$ are integer mapping functions defined for each character σ in the alphabet Σ, and implemented as lookup tables.

Chang and Lee provided an algorithm which computes values of the mapping functions. The algorithm breaks into three steps. In step 1, all extracted pairs are divided into groups $G_1, G_2, \ldots, G_t$, so that in group G_s, $s = 1, 2, \ldots, t$, are all pairs (σ_i, σ_j) with the same σ_i, and the groups are arranged by lexical order of σ_i's. Then, if σ_i appears in G_s, $d(\sigma_i)$ is set to $d(\sigma_i) = \sum_{k=1}^{s-1} |G_k|$. The values of function d serve as offsets in the hash table.

In step 2, each distinct σ_j is associated with a unique prime number $p(\sigma_j)$. First, the σ_j's are arranged in descending order of frequency. Then, the prime 2 is assigned to the σ_j with greatest frequency, prime 3 is assigned to the σ_j with second greatest frequency, and so on.

Finally, in step 3, each σ_i is associated with $C(\sigma_i)$ determined by taking into account all of the accompanying σ_j's appearing together with σ_i. Let $p_1 < p_2 < \cdots < p_b$ be the prime numbers assigned to $\sigma_{j_1}, \sigma_{j_2}, \ldots, \sigma_{j_b}$, respectively. Then, using Chang's Algorithm A, a value $C(\sigma_i)$ can be found such that $C(\sigma_i) \equiv 1 \pmod{p_1}$, $C(\sigma_i) \equiv 2 \pmod{p_2}, \ldots, C(\sigma_i) \equiv b \pmod{p_b}$.

Chang and Lee reported success with several nontrivial sets of words, like 34 non-printable ASCII identifiers or 36 Pascal reserved words. However the assumption they made, that in a set of keys there must be i and j such that all pairs of characters (σ_i, σ_j) are distinct, is quite restrictive. For example, it fails even for such a small set as {aim, air, ham, him}. As in Chang's algorithm, values of $C(\sigma_i)$ can be large, although in general they are smaller due to the assignment of the prime numbers according to the frequencies of the σ_j's.

Example 2.6. Let us construct a minimal perfect hash function for the 31 most frequently used English words: A, AND, ARE, AS, AT, BE, BUT, BY, FROM, FOR, HAD, HE, HER, HIS, HAVE, I, IN, IS, IT, NOT, OF, ON, OR, THAT, THE, THIS, TO, WHICH, WAS, WITH and YOU. We extract the first letter and the third letter if the word contains

Table 6
Grouping the extracted pairs

Group	Extracted pair	Original key	Group	Extracted pair	Original key
1	(A, A)	A	5	(I, I)	I
	(A, D)	AND		(I, N)	IN
	(A, E)	ARE		(I, S)	IS
	(A, S)	AS		(I, T)	IT
	(A, T)	AT	6	(N, T)	NOT
2	(B, E)	BE	7	(O, F)	OF
	(B, T)	BUT		(O, N)	ON
	(B, Y)	BY		(O, R)	OR
3	(F, O)	FROM	8	(T, A)	THAT
	(F, R)	FOR		(T, E)	THE
4	(H, D)	HAD		(T, I)	THIS
	(H, E)	HE		(T, O)	TO
	(H, R)	HER	9	(W, I)	WHICH
	(H, S)	HIS		(W, S)	WAS
	(H, V)	HAVE		(W, T)	WITH
			10	(Y, U)	YOU

more than two letters, and the first and last letters otherwise. We may group the pairs extracted as shown in Table 6.

Once the grouping is done, values for function d can be computed. Since A appears in group G_1, we set $d(\mathtt{A}) = 0$. For F, for example, we compute $d(\mathtt{F}) = |G_1| + |G_2| = 5 + 3 = 8$. Therefore we have the following table:

σ	A	B	F	H	I	N	O	T	W	Y
$d(\sigma)$	0	5	8	10	15	19	20	23	27	30

In the second step, the σ_j's (the second letters of the pairs) are arranged in descending order of frequency. Let $f(\sigma)$ denote the frequency of letter σ. We obtain $f(\mathtt{T}) = 5$, $f(\mathtt{E}) = f(\mathtt{S}) = 4$, $f(\mathtt{I}) = f(\mathtt{R}) = 3$, $f(\mathtt{A}) = f(\mathtt{D}) = f(\mathtt{N}) = f(\mathtt{O}) = 2$, $f(\mathtt{F}) = f(\mathtt{U}) = f(\mathtt{V}) = f(\mathtt{Y}) = 1$. Thus, we assign prime number 2 to T, 3 to E, 5 to S, and so on. The values of $p(\sigma)$ are as follows.

σ	A	D	E	F	I	N	O	R	S	T	U	V	Y
$p(\sigma)$	13	17	3	29	7	19	23	11	5	2	31	37	41

In the last step, we group the primes associated with σ_j's having the same σ_i's. This gives us 10 groups: {A: 2,3,5,13,17}, {B: 2,3,41}, {F: 11,23}, {H: 3,5,11,17,37}, {I: 2,5,7,19}, {N: 2}, {O: 11,19,29}, {T: 3,7,13,23}, {W: 2,5,7} and {Y: 31}. Finally, using Algorithm A, we compute suitable values of $C(\sigma_i)$, $i = 1,2,\ldots,10$, giving

the following table:

σ	A	B	F	H	I	N	O	T	W	Y
$C(\sigma)$	6023	167	232	75892	1277	1	496	1108	17	1

Then the 31 most frequently used English words are hashed into the addresses 1 to 31 in the order: AT, ARE, AS, A, AND, BUT, BE, BY, FOR, FROM, HE, HIS, HER, HAD, HAVE, IT, IS, I, IN, NOT, OR, ON, OF, THE, THIS, THAT, TO, WITH, WAS, WHICH, YOU. Thus this minimal perfect hash function hashes (T, E) (word THE) as follows: $h(\text{THE}) = d(\text{T}) + (C(\text{T}) \bmod p(\text{E})) = 23 + (1108 \bmod 3) = 23 + 1 = 24$.

2.6. A Chinese remainder theorem based variation

Trying to overcome drawbacks of Chang's algorithm [19], Chang and Chang proposed a modified ordered minimal perfect hashing scheme [22]. They proved the following.

Theorem 2.3. *For a finite integer set* $S = \{x_1, x_2, \ldots, x_n\}$, $h(x) = \lceil C/n^{x_i-1} \rceil \bmod n$, $x \in S$, *is an ordered minimal perfect hash function if*

$$C = (n-1)n^{x_n-1} - \sum_{i=2}^{n-1} (n-i+1)n^{x_i-1}$$

and $x_1 < x_2 < \cdots < x_n$.

Example 2.7. Let $S = \{2, 4, 7, 9, 11\}$. Then

$$\begin{aligned} C &= (5-1) \times 5^{x_5-1} - \sum_{i=2}^{5-1} (5-i+1) \times 5^{x_i-1} \\ &= 4 \times 5^{x_5-1} - ((5-2+1) \times 5^{x_2-1} + (5-3+1) \times 5^{x_3-1} + (5-4+1) \times 5^{x_4-1}) \\ &= 4 \times 5^{11-1} - (4 \times 5^{4-1} + 3 \times 5^{7-1} + 2 \times 5^{9-1}) \\ &= 4 \times 5^{10} - 4 \times 5^3 - 3 \times 5^6 - 2 \times 5^8 = 38\,233\,875. \end{aligned}$$

Thus the hash function becomes $h(x) = \lceil 38\,233\,875/5^{x_i-1} \rceil \bmod 5$, and it hashes S as follows:

x	2	4	7	9	11
$h(x)$	0	1	2	3	4

This method removes the first impediment of Chang's algorithm, i.e. the necessity for finding an appropriate prime number function. Also the method of computing C is quite straightforward, and Chang and Chang showed that the time required to compute it is $O(n \log x_n)$. Unfortunately, as before, the value of C tends to be very large, and

the number of bits required to store C is $O(n \log n)$. Consequently, calculation of the hash function takes roughly $O(n \log x_n)$ time. Since $x_n \geqslant n$, this is slower than binary search on an ordered table. Due to the fast growth rate of the parameter C and the time for calculating the hash function, the algorithm has theoretical value only.

Example 2.8. Consider the set from Example 2.5, $S = \{1, 2, 14, 18, 21, 24, 27, 28, 29, 32, 34, 40\}$. For this set

$$\begin{aligned} C &= 11 \times 5^{x_{12}-1} - \sum_{i=2}^{11} (13-i) \times 12^{x_i - 1} \\ &= 11 \times 12^{39} - (11 \times 12 + 10 \times 12^{13} + \cdots + 2 \times 12^{33}) \\ &= 13\,472\,905\,210\,791\,188\,637\,278\,318\,893\,030\,892\,938\,198\,908. \end{aligned}$$

This constant requires 144 bits in binary representation.

2.7. Another Chinese remainder theorem based variation

Another variation of Chang's solution is an algorithm proposed by Winters [100]. One of the impediments in Chang's scheme is that it requires a method for transforming integers into prime numbers. As no such general method is known, ready application of the algorithm is limited to sets for which there is a prime number function. Winters exploited the fact that elements of the set do not have to be prime – it is sufficient if they are relatively prime. Consequently, Winters presented a method which transforms an arbitrary set of nonnegative integers into a set of nonnegative and relatively prime integers.

Theorem 2.4. *Given any set of nonnegative integers* $S = \{x_1, x_2, \ldots, x_n\}$, *the elements of the set* $S' = \{Mx_1 + 1, Mx_2 + 1, \ldots, Mx_n + 1\}$, *where* $M = \prod_{x_i > x_j}(x_i - x_j)$, *are pairwise relatively prime.*

The above theorem allows us to perform the mapping from an input set of nonnegative integers, S, into a set of relatively prime integers, S'. Then, the smallest constant C for S' such that $C \bmod (Mx + 1)$ is a minimal perfect hash function for $x \in S$ can be computed by using Chang's Algorithm A.

In addition, Winters proposed a probabilistic method for converting an input set $\{x_1, x_2, \ldots, x_n\}$ into a set $\{y_1, y_2, \ldots, y_n\}$, such that $\max_{1 \leqslant i \leqslant n} y_i = O(n^2)$. This conversion is useful when the values of x_i's are large, compared with their number (for example, when the original keys are character strings treated as numbers using the natural base). Since, in the transformation of an input set into a set of relatively prime integers, differences of elements are used, the conversion allows a reduction of the magnitudes of the hash function parameters and makes it independent of the values of the x_i's.

Unfortunately, despite all these modifications, the constant C produced by the algorithm is still enormous. Winters proved that the number of bits required to represent

C is $O(n^3 \log n)$ and the time necessary to compute it is roughly $O(n^4 \log n)$. Notice, that for $n = 1024$, the constant C may require more than 1 gigabyte of memory. Although the algorithm is guaranteed to stop after a polynomially bounded number of arithmetic steps, the magnitude of C makes this solution impractical. The time required to compute the value of the hash function for a given key is approximately $n^2 \log n$ times longer than the time required for a simple linear search in a table of keys, and the space required for C is greater by a factor of n^2 than that required for the table of keys.

Example 2.9. Let us compute parameters M and C for the set from Example 2.8, $S = \{1, 2, 14, 18, 21, 24, 27, 28, 29, 32, 34, 40\}$.

$$\begin{aligned} M &= \prod_{x_i > x_j} (x_i - x_j) \\ &= (2-1)(14-1)(14-2)(18-1)\cdots(40-32)(40-34) \\ &= 1 \times 13 \times 12 \times 17 \times \cdots \times 8 \times 6 \\ &= 2\,247\,347\,982\,474\,990\,500\,545\,110\,751\,169\,304\,868\,554\,381\,625\,500\,106\,752 \\ &\quad 000\,000\,000\,000. \end{aligned}$$

Now, using Chang's Algorithm A, the constant C is computed: $C \approx 1.83 \times 10^{810}$. The number of bits in the binary representation of C is 2692. The parameters M and C would be much larger if we treated character keys as base 26 numbers.

2.8. Bibliographic remarks

Some remarks on the implementation of the algorithms devised by Sprugnoli may be found in [4].

In [18] Chang considered the method of finding an ordered minimal perfect hash function of the form $h(x) = C \bmod T(x)$, $x \in S$, where C is computed using the Euler's theorem and $T(x)$ is a pairwise relatively prime transformation function on S. Chang proposed [20] a letter-oriented minimal perfect hash function based upon Jaeschke's reciprocal hashing. Given a distinct pair of characters extracted from the keys, (σ_i, σ_j), his hash function took the form $h(\sigma_i, \sigma_j) = d(\sigma_i) + \lfloor C(\sigma_i)/p(\sigma_j) \rfloor \bmod n(\sigma_i)$, where d, C, p and n are integer mapping functions defined for each character σ in the alphabet Σ. The C function is determined using Algorithm C given by Jaeschke. Chang and Wu considered the problem of how to design a minimal perfect hash function which is suitable for the Mandarin phonetic symbols system [26]. Inspired by Chang's scheme [20] they hashed 1303 Mandarin phonetic symbol transcriptions to 1303 locations preserving one-to-one correspondence.

Using Chang and Lee's results, Jan and Chang [63] developed a letter-oriented minimal perfect hash function which requires storing 2×26 integer values rather than 3×26 values ($d(\sigma_i), C(\sigma_i)$ and $p(\sigma_j)$) needed in Chang and Lee's scheme (cf. Section 2.5). Their hash function has the form $h(\sigma_i, \sigma_j) = d(\sigma_i) + \lfloor C(\sigma_i)/n(\sigma_i)^{f(\sigma_j)-1} \rfloor \bmod n(\sigma_i)$,

where d and C are integer mapping functions, $n(\sigma_i)$ is the number of keys with the same σ_i, and $f(\sigma_j)$ is an encoding function with the formula $f(\sigma_j) = \text{ord}(\sigma_j) - \text{ord}(\texttt{'A'}) + 1$. Of these hash function parameters only d and C need to be stored as the lookup tables. Another modification of Chang and Lee's method was given by Chang et al. [23]. They proposed a hash function of the form $h(\sigma_i, \sigma_j) = v_1(\sigma_i) + v_2(\sigma_j)$, where v_1 and v_2 are two integer mapping functions defined on the set of distinct pairs of characters extracted from the keys. Although the form of the hash function is quite simple, the time complexity of the algorithm to compute the values of v_1 and v_2 described in [23] is $O(m!m^2 + n^2)$, where m and n denote the number of distinct σ_i's and σ_j's. Inspired by method of Chang et al. [23], Chang proposed a composite perfect hashing scheme for large letter-oriented key sets [21] with the help of a hashing indicator directory (HID).

Chapter 3. Perfect hashing with segmentation

3.1. Introduction

An important technique which led to a number of methods of good quality is called *segmentation*. Segmentation divides the input set of keys into a number of subsets. For each subset a (minimal) perfect hash function is constructed separately.

In this chapter several methods based on segmentation are discussed. In Section 3.2 we present a variant of scheme of Fredman et al. proposed by Slot and van Emde Boas [92]. The scheme requires optimal space to represent the hash function, but the evaluation time of the function is linear in the number of keys in the input set. Section 3.3 describes the methods that hinges on rehashing and segmentation given by Du et al. [41]. These methods employ a series of hash functions selected randomly from the set of all functions that map the input keys into an address space. In Section 3.4 the modification of those methods proposed by Yang and Du that uses backtracking to find an appropriate series of functions is presented [102]. Section 3.5 describes a method of very small subsets devised by Winters [99]. The method utilizes the subsetting procedure that allows the separation of the input set of keys into subsets of cardinalities at most two, for which a minimal perfect hash function can be easily constructed.

3.2. Bucket distribution schemes

The FKS scheme developed by Fredman et al. [50], and slightly improved by Mehlhorn [74] was described in Section 1.3 The scheme divides the input set of keys into subsets in two steps. Using a primary hash function the initial division is done in the first step. A subset of the input set of keys mapped into the same location is said to form a collision bucket. Then in the second step, for all keys in a collision bucket, a secondary perfect hash function is constructed. Despite this two-step procedure, it offers the advantage that perfect hash functions are constructed for small subsets, thus

improving significantly the chances of finding such functions quickly. The effectiveness of the method strongly depends on how closely the primary hash function mimics a truly random function.

More specifically, given a set S of n distinct keys belonging to the universe $U = \{0, 1, \ldots, u-1\}$, a partition of S into a number of collision buckets S_i, $i = 0, 1, \ldots, n-1$, of size $s_i = |S_i|$, is obtained by using a primary hash function

$$h : x \mapsto (ax \bmod u) \bmod n$$

where $a \in [1, u-1]$ is a multiplier, and u is assumed to be prime. Then, for each collision bucket a secondary perfect hash function is constructed, of the form

$$h_i : x \mapsto (a_i x \bmod u) \bmod c_i$$

where $a_i \in [1, u-1]$ is a multiplier and $c_i = s_i(s_i - 1) + 1$ is the size of the hash table for bucket S_i.

We showed that the description of the FKS compound hash function $h_i \circ h$ needs $O(3n \log u)$ bits. In addition, $3n \log u$ bits are required to store the keys of S (cf. Section 1.3).

In what follows, we present a variant of the FKS scheme given by Slot and van Emde Boas [92]. The scheme requires optimal space to represent the hash function, i.e. $O(n + \log\log u)$ bits, but the evaluation time of the function is $O(n)$.

The method works as follows: initially the magnitude of keys is reduced by two preprocessing transformations. The first transformation, $\zeta : x \mapsto x \bmod q$, maps S into $S' \subset [0, q-1]$ without collisions, where $q < n^2 \log u$ is a suitable prime number given by Theorem 1.3. To store q we need $2 \log n + \log\log u$ bits. The second transformation, $\rho : x \mapsto (a_\rho x \bmod q) \bmod n^2$, maps S' into $S'' \subset [0, n^2 - 1]$, also without collisions. It follows from Corollary 1.2 that we can find a suitable multiplier $a_\rho \in [1, q-1]$ for that mapping. It is easy to see that the space needed for representing the parameters of these preprocessing transformations is $O(\log n + \log\log u)$ bits.

Once the keys from S are reduced to the range $[0, n^2 - 1]$, the scheme of Slot and van Emde Boas proceeds in the "standard" way. First, the primary hash function h maps S'' into $[0, n-1]$ so that the total amount of space for the bucket hash tables is less than $5n$ (cf. Corollary 1.3). Then, the secondary hash function h_i scatters perfectly the keys of buckets S_i, $i = 0, 1, \ldots, n-1$, over the hash tables of sizes $c_i = 2s_i(s_i - 1) + 1$ (cf. Corollary 1.4). It is assumed that the hash tables for the buckets S_i are parts of a table $D[0 \ldots 5n-1]$.

To minimize the space complexity of the scheme, we exploit the relation $\sum_{i=0}^{n-1} s_i = n$. This allows us to encode the values s_i in unary notation by a bit string of length $2n$, which consists of blocks of s_i 1's separated by 0's. During a membership query we need the s_i's twice. First, to compute the size of the hash table for the ith bucket, $c_i = 2s_i(s_i - 1) + 1$, and then to compute the offset $C_i = \sum_{j=0}^{i-1} c_j$ of the hash table of that bucket in table D. Clearly, both these values can be computed in $O(n)$ time using the encoding of s_i's mentioned above.

In the last step we need multipliers for the secondary hash functions. By Lemma 1.1, the number of these multipliers can be reduced to $\lfloor \log n \rfloor + 1$. The multipliers $a_0, a_1, \ldots, a_{\lfloor \log n \rfloor}$ are stored in a $(\lfloor \log n \rfloor + 1)$-word array which consists of $(\lfloor \log n \rfloor + 1) \times 2 \log n = O(\log^2 n)$ bits. The assignment of each multiplier to a subset of buckets must be defined. For this purpose another unary encoding is used, which allows us to determine the array index of the multiplier assigned to bucket S_i. We use a bit string zz of length at most $2n$ bits containing exactly n 1's. The string is divided into a number of segments of decreasing lengths. The first segment of n bits encodes which buckets are assigned to multiplier a_0. The second segment of length at most $n/2$ bits encodes which of the remaining buckets (not assigned to a_0) are assigned to a_1. The third segment of length at most $n/4$ bits specifies the buckets (assigned neither to a_0 nor to a_1) assigned to a_2, and so on. A function *multind*, given below, evaluates the index of the multiplier assigned to bucket S_i using the following variables:

pos – the index of the currently scanned bit in *zz*,
place – the index of the currently scanned bit within a segment,
segm – the index of the currently scanned segment,
size – the length of the current segment,
rank – the position of the bit within the current segment determining whether the index of this segment is equal to the index of the multiplier assigned to bucket S_i,
rem – the number of buckets whose multiplier index is not equal to the index of the current segment.

```
function multind(i: integer; zz: bit-string): integer;
  pos, place: integer := −1;
  segm, rem: integer := 0;
  size: integer := n; -- the length of the first segment
  rank: integer := i;
begin
  loop
    pos := pos + 1; place := place + 1;
    if zz[pos] = 0 then
      rem := rem + 1;
    end if;
    if place = rank then
      if zz[pos] = 1 then
        return(segm);
      else
        rank := rem − 1;
      end if;
    end if;
    if place = size − 1 then
      segm := segm + 1; size := rem;
```

```
        place := −1; rem := 0;
      end if;
    end loop;
  end multind;
```

The function scans the consecutive segments of zz. If the bit corresponding to bucket S_i in the current segment is 1, then the multiplier a_{segm} is assigned to S_i. In the loop, the variable *rem* is first updated. Then, if the currently scanned bit represents S_i, we test whether it is on. If so, the multiplier index for S_i is determined and we terminate. Otherwise the rank of the bit corresponding to S_i in the next segment is set. Finally, we test if the end of the current segment is reached, and if so the variables *segm*, *size*, *place* and *rem* are reset for the next execution of the loop. Note that we do not need any special markers in zz for separating the segments. The function *multind* can detect where the segments start. The function terminates for all bit strings zz containing at least n 1's. Clearly, since zz consists of at most $2n$ bits, the evaluation time of function *multind* is $O(n)$.

Example 3.1. Let us use the method of Slot and van Emde Boas to find the hash scheme for the set of 12 integers corresponding to the names of the months (cf. Section 2.3, Example 2.3), $S = \{6800, 4188, 8832, 1110, 8839, 7320, 7318, 1229, 12\,990, 10\,238, 9876, 2837\}$, $|S| = n = 12$. We begin by reducing the magnitude of the keys. The first preprocessing transformation, $\zeta : x \mapsto x \bmod q$, where $q = 167$ maps S into $S' = \{120, 13, 148, 108, 155, 139, 137, 60, 131, 51, 23, 165\}$. The second transformation, $\rho : x \mapsto (a_\rho x \bmod q) \bmod n^2$, for which we select $a_\rho = 13$, maps S' into $S'' = \{57, 2, 87, 68, 11, 137, 111, 112, 33, 18, 132, 141\}$. The primary function could be $h : x \mapsto (14 \times x \bmod 149) \bmod 12$ which splits S'' into the following collision buckets $S_0 = \{2, 132\}$, $S_1 = \{141\}$, $S_2 = \{87\}$, $S_3 = \{33\}$, $S_4 = \{111\}$, $S_5 = \{57, 11\}$, $S_6 = \{112\}$, $S_7 = \{18\}$, $S_8 = S_9 = \emptyset$, $S_{10} = \{68, 137\}$, $S_{11} = \emptyset$. The amount of space needed for the bucket hash tables $\sum_{i=0}^{11}(2s_i(s_i - 1) + 1) = 21 < 5n$. The bit string of length $2n = 24$ encoding the bucket sizes s_i is 110.10.10.10.10.10.110.10.10.0.0.110.0 (the dots separate the sizes). Now we have to find at most $\lfloor \log n \rfloor + 1 = 4$ multipliers for the secondary hash functions of the form $h_i : x \mapsto (a_i x \bmod 149) \bmod (2s_i(s_i - 1) + 1)$. We select $a_0 = 1$ which can be assigned to all buckets except S_0. To resolve S_0 we select $a_1 = 2$. The bit string zz encoding the assignment of multipliers a_0 and a_1 to buckets S_0 and S_{11} is 011111111111.1 (the dot separates the segments). The placement of the keys in the hash table is shown in Fig. 6. To test if some key $x \in S$ we proceed as follows:

1. Compute $x' = x \bmod 167$.
2. Compute $x'' = (13 \times x' \bmod 167) \bmod 144$.
3. Compute the bucket index $i = (14 \times x'' \bmod 149) \bmod 12$.
4. Compute $C_i = \sum_{j=0}^{i-1}(2s_j(s_j - 1) + 1)$ and s_i by scanning the bit string encoding the bucket sizes.
5. Determine the secondary hash function multiplier evaluating function *multind*.

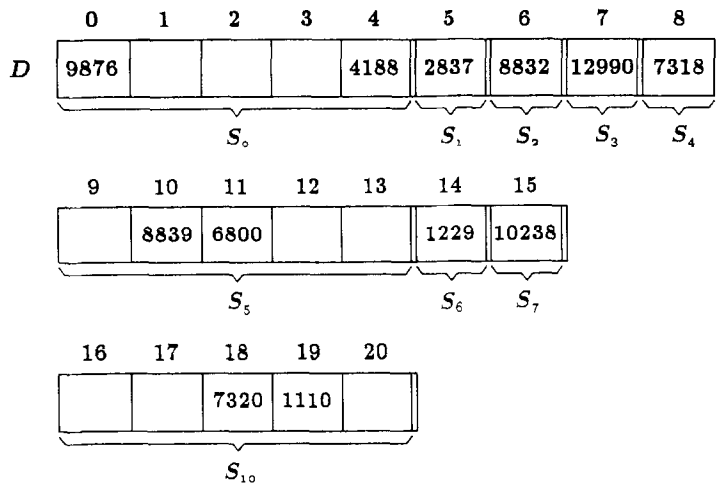

Fig. 6. The placement of keys in the hash table.

6. Compute the offset within the bucket, $\delta = (a_{seqm} \times x'' \bmod 149) \bmod (2s_i(s_i - 1) + 1)$.

7. Check the key stored in D at location $C_i + \delta$.

The key JUL is looked up as follows: $x' = 7318 \bmod 167 = 137$; $x'' = (13 \times 137 \bmod 167) \bmod 144 = 111$; $i = (14 \times 111 \bmod 149) \bmod 12 = 4$; $C_4 = \sum_{j=0}^{3}(2s_j(s_j - 1) + 1) = 5+1+1+1 = 8$ and $s_4 = 1$; $a_{multind(4,zz)} = a_0 = 1$; $\delta = (111 \bmod 149) \bmod 1 = 0$; $D[C_4 + \delta] = D[8] = 7318$, which corresponds to JUL.

3.3. A hash indicator table method

Methods based on *rehashing* and *segmentation* are due to Du et al. [41]. Their *composite hash function* h is formed from a number of hash functions $h_1, h_2, \ldots, h_s$, selected randomly from the set $F_{n \times m}$ of all functions that map n keys into an address space with m entries. A special table, called the *hash indicator table* (*HIT*), is used. It has the same number of entries as that of the address space. The entry $HIT[d]$ corresponds to the entry d in the address space. The contents of *HIT* is determined as follows. Given the input set S of keys, each key $x \in S$ that h_1 maps into a unique address of the address space is placed in this location of the hash table and its *HIT* is set to 1. We shall call such a (*key*, *address*) pair as *singleton*. Then the remaining keys of S are rehashed, i.e. the second hash function h_2 is applied to all keys that were not allocated places by the first hash function. Again, all keys that h_2 hashes into unique locations are put in appropriate places, i.e. $h_2(x)$, in the hash table, and their *HIT* values are set to 2. This process continues until all keys of S are inserted in the hash table or all s hash functions have been applied. If the process is successful then the *HIT* value specifies the index of the hash function which mapped the key (uniquely) to the hash table.

More formally, the composite hash function can be defined as follows:

$$h(x) = \begin{cases} h_j(x) = d & \text{if } HIT[h_r(x)] \neq r \text{ for } r < j \text{ and} \\ & HIT[h_j(x)] = HIT[d] = j, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The contents of the *HIT* table is computed by the following program:

$S := \{x_0, x_1, \ldots, x_{n-1}\}$;
Clear all entries of *HIT*;
for $j := 1$ **to** s **do**
  for all $x_r \in S$ **do**
    $HIT[d] := j$ if $HIT[d] = 0$ and $h_j(x_r) = d$ for one and only one $x_r \in S$;
    $S := S - \{x_r : x_r$ satisfies the above conditions$\}$;
  end for;
end for;
if $S \neq \emptyset$ **then**
  The program fails to find a perfect hash function for $\{x_0, x_1, \ldots, x_{n-1}\}$;
end if;

To check if a key x is in the hash table, the hash functions $h_1, h_2, \ldots, h_s$ are applied to x until $HIT[h_j(x)] = j$. If this condition fails, then the hash value for x is undefined.

Example 3.2. Let $S = \{x_0, x_1, \ldots, x_7\}$, and let the address space be from 0 to 8. Table 7 shows the hash values for the three hash functions h_1, h_2 and h_3 which form the composite hash function h. The table entries in bold indicate the singletons. We can see that x_1 and x_7 are placed by h_1, x_2, x_3 and x_6 by h_2, and x_0 by h_3. The mapping of h is undefined on the remaining keys x_4 and x_5. Fig. 7 shows the *HIT* table constructed by the program given above and the placement of the keys in the hash table D. A query for x_0 proceeds as follows: $HIT[h_1(x_0)] = HIT[3] = 0 \neq 1$, $HIT[h_2(x_0)] = HIT[4] = 1 \neq 2$, and finally $HIT[h_3(x_0)] = HIT[6] = 3$. Thus $h(x_0) = 6$.

Let us analyze the efficiency of the method. A hash function h_l is said to have k singletons if there are k entries in the address space with a single key hashed by h_l to each of them. Let h_l be a hash function drawn randomly from $F_{n \times m}$, and let $P_k(n, m)$ denote the probability of h_l having k singletons. Then [41, Theorem 1]

Table 7
The hash values for h_1, h_2 and h_3

Hash function	Key							
	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7
h_1	3	**4**	3	5	1	5	1	**8**
h_2	4	7	**0**	**1**	3	3	**5**	2
h_3	**6**	2	7	0	4	8	3	7

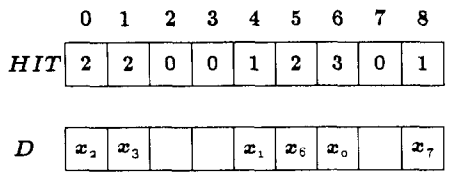

Fig. 7. The *HIT* and *D* tables.

$P_k(n, m) = e_k(n, m)/m^n$ where

$$e_k(n, m) = n!\binom{m}{k}\sum_{r=0}^{n-k}(-1)^r\binom{m-k}{r}\frac{(m-r-k)^{n-r-k}}{(n-r-k)!}.$$

The mean value of the number of singletons delivered by h_l is

$$E = \sum_{k=0}^{n} k \cdot P_k(n, m) = n\left(1 - \frac{1}{m}\right)^{n-1}.$$

Now let $Q_k^s(n,m)$ be the probability of obtaining k singletons by applying $h_1, h_2, \ldots, h_s$ in the way described before, where each h_l is randomly chosen from $F_{n\times m}$. This probability can be computed from the formulas [41, Theorems A1 and A2]:

$$Q_k^s(n,m) = \sum_{r=0}^{k} Q_r^{s-1}(n,m)\Delta P_{k-r}(n,m,r),$$

$$\Delta P_k(n,m,i) = \left(\frac{i}{m}\right)^{n-i}\sum_{j=k}^{n-i}\frac{\binom{n-i}{j}}{i^j}e_k(j, m-i)$$

with $Q_k^1(n,m) = P_k(n,m)$. $\Delta P_k(n,m,i)$ is defined as the probability of getting k more singletons by applying the sth function, provided that i singletons have been obtained by the successive application of $s-1$ functions.

Example 3.3. Let $n = 14$ and $m = 17$. The expectation of k for $Q_k^1(14,17)$ is 6.673471 and for $Q_k^7(14,17)$ is 13.280872. That is, while using a single random hash function we can expect less than 7 keys out of 14 to be placed with no collisions in the 17-element hash table. This expectation increases to over 13 keys if we use up to 7 hash functions.

Surely, if we expand the sequence of functions $h_1, h_2, \ldots, h_s$, the probability of the composite hash function being perfect will increase. However this increase will be slow, since the more singletons we already have, the fewer chances there are for the remaining keys to be mapped into unique and unoccupied locations of the hash table. Du et al. have found that instead of expanding the sequence of functions, better results can be achieved by making use of segmentation. In their modified scheme, the address space is divided into q segments. Each segment $\mathscr{D}_i$, $i = 0, 1, \ldots, q-1$, of size m_i has assigned to it a composite hash function h^i which maps the set of keys $\{x_0, x_1, \ldots, x_{n-1}\}$ into the address subspace $\{0, 1, \ldots, m_i - 1\}$. HIT_i is the hash indicator table of h^i, which

specifies the index j of function h_j^i which gives the value of h^i. All the tables HIT_i, $i = 0, 1, \ldots, q-1$, constitute the hash indicator table HIT. The composite hash function H in the modified scheme is defined as follows:

$$H(x) = \begin{cases} h_j^i(x) + \sum_{t=0}^{i-1} m_t & \text{if } i, j \text{ can be found such that } HIT_i[h_r^i(x)] \neq r \\ & \text{for } r < j, \text{ and } HIT_i[h_j^i(x)] = j, \text{ and} \\ & HIT_a[h_b^a(x)] \neq b \text{ for } a < i \text{ and } 1 \leqslant b \leqslant s \ (m_0 = 0), \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The definition implies that if a key x does not find its proper position in the first segment with h^0, we try the second segment with h^1, and so on. The following program computes the values of $H(x)$:

```
t := 0;
for i := 0 to q − 1 do
  for j := 1 to s do
    z := h_j^i(x);
    if HIT_i[z] = j then return(t + z); end if;
  end for;
  t := t + m_i;
end for;
return(undefined);
```

The program to compute the HIT values is given below. We assume that the partition of the address space into segments $(\mathscr{D}_0, \mathscr{D}_1, \ldots, \mathscr{D}_{q-1})$ of sizes $(m_0, m_1, \ldots, m_{q-1})$ and the hash functions $h_1^0, h_2^0, \ldots, h_s^0, h_1^1, h_2^1, \ldots, h_s^1, \ldots, h_1^{q-1}, h_2^{q-1}, \ldots, h_s^{q-1}$ are all given.

```
S := {x_0, x_1, ..., x_{n−1}};
Clear all entries of HIT_i, i = 0, 1, ..., q − 1;
for i := 0 to q − 1 do
  for j := 1 to s do
    for all x_r ∈ S do
      HIT_i[d] := j if HIT_i[d] = 0 and h_j^i(x_r) = d for one and only one x_r ∈ S;
      S := S − {x_r : x_r satisfies the above conditions};
    end for;
  end for;
end for;
if S ≠ ∅ then
  The program fails to find a perfect hash function for {x_0, x_1, ..., x_{n−1}};
end if;
```

Example 3.4. Let $S = \{x_0, x_1, \ldots, x_8\}$, and let the address space from 0 to 13 be partitioned into segments $(\mathscr{D}_0, \mathscr{D}_1)$ of sizes $(9, 5)$. Table 8 defines the hash values for the six hash functions $\{h_1^0, h_2^0, h_3^0\}$ and $\{h_1^1, h_2^1, h_3^1\}$ which form the composite hash

Table 8
The hash values of $\{h_1^0, h_2^0, h_3^0\}$ and $\{h_1^1, h_2^1, h_3^1\}$

Hash function	Key								
	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
h_1^0	3	**4**	3	5	1	5	1	**8**	**0**
h_2^0	4	7	8	**1**	3	3	0	1	7
h_3^0	**6**	5	0	4	1	**3**	8	7	4
h_1^1	0	2	1	3	1	2	1	4	2
h_2^1	1	3	4	1	4	4	**2**	0	2
h_3^1	4	1	**0**	0	**4**	3	0	2	1

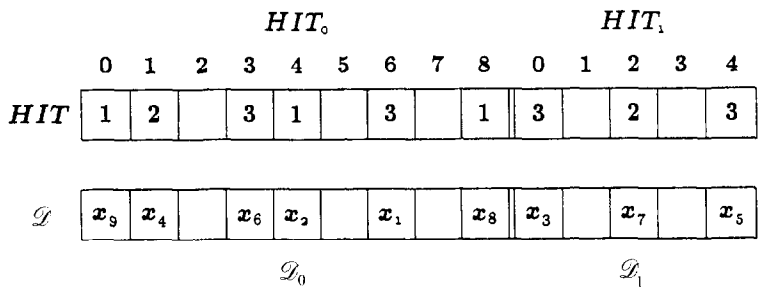

Fig. 8. The *HIT* and $\mathscr{D}$ tables.

function H. As before, the table entries in bold indicate the singletons. In Fig. 8 the *HIT* and $\mathscr{D}$ tables are shown. It can be seen that H is perfect for S.

Let $R_k^s(n, \overline{m}, q)$ denote the probability of H having k singletons. The vector $\overline{m} = (m_0, m_1, \ldots, m_{q-1})$ defines the partition of the address space. We denote $(m_0, m_1, \ldots, m_{r-1})$, $1 \leqslant r \leqslant q$, by $\overline{rm}$, and so $\overline{qm} = \overline{m}$. The probability $R_k^s(n, \overline{qm}, q)$ can be calculated from the formula [41, Theorem A4]:

$$R_k^s(n, \overline{qm}, q) = \sum_{k_{q-1}=0}^{k} \sum^{k_0+k_1+\cdots+k_{q-2}=k-k_{q-1}} \cdot \prod_{i=0}^{q-1} Q_{k_i}^s \left(n - \sum_{t=0}^{i-1} k_t, m_i\right)$$

$$= \sum_{k_{q-1}=0}^{k} R_{k-k_{q-1}}^s(n, \overline{(q-1)m}, q-1) \cdot Q_{k_{q-1}}^s(n - (k - k_{q-1}), m_{q-1})$$

with $R_k^s(n, \overline{1m}, 1) = Q_k^s(n, m_0)$.

Example 3.5. Let $n = 14$ and $m = 17$, as in Example 3.3, and let $\overline{m} = (13, 4)$. Then the probability $R_{14}^7(14, (13, 4), 2) = 0.978$, which means that we have a better than 97 percent chance to construct a perfect hash function mapping 14 keys into 17 addresses using a 2-segment hash scheme with 2×7 random hash functions in total.

As indicated in [69], the schemes of Du et al. described above have the following drawbacks: (i) selecting a suitable set of random hash functions is difficult (in fact, Du et al. do not discuss this problem at all), (ii) retrieving a key may require up to s accesses, or up to qs in the modified scheme, to the *HIT* table, and (iii) the hash function is not guaranteed to work for all sets of keys.

3.4. Using backtracking to find a hash indicator table

As we have seen in the previous section, the hash addresses $(e_0, e_1, \ldots, e_{n-1})$, or equivalently the *HIT* values, corresponding to keys $(x_0, x_1, \ldots, x_{n-1})$ are determined by considering the table of hash addresses for a series of functions $h_1, h_2, \ldots, h_s$ row by row (see Example 3.2). Let us denote this table M. First, all the singletons in the first row of M are selected (marked in bold). Each singleton defines a hash address for one key. Then, the singletons in the second row are selected, omitting the columns with singletons selected in the first row. The remaining rows of M are processed in the similar fashion.

Unfortunately, this procedure does not guarantee finding the hash addresses for a given table M, even when a solution does exist. Table 9 shows such a solution for the sample set of keys from Example 3.2.

A superficial analysis of the procedure described above reveals that it assigns the addresses to keys whenever it is possible. This means that all the singletons in every row place some keys in the hash table. However, there are cases where the solution can be obtained only by delaying specific address assignments or, in other words, by "combining" the singletons from different rows of M. Following this idea, Yang and Du [102] proposed a backtracking method which, for a given table M, finds (1) all the solutions, (2) the optimal solution with the lowest retrieval cost defined as $(\sum_{i=0}^{m-1} HIT[i])/n$, and (3) an answer "no solution" if no solution exists. We shall describe this method in the sequel.

Let $(e_0, e_1, \ldots, e_{n-1})$ be a sequence of the hash addresses, where each e_i is chosen from column i in M. A sequence is called a *solution* if it has two properties. The first property implies that no two addresses in the sequence are the same, as no two keys can occupy the same address in the hash table:

$$e_i \neq e_j \quad \text{for } i \neq j,\ 0 \leqslant i,\ j \leqslant n-1. \tag{3.1}$$

Table 9
A solution for the sample set of keys

Hash function	Key							
	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7
h_1	3	4	3	5	1	5	1	8
h_2	4	7	**0**	**1**	3	3	**5**	**2**
h_3	**6**	2	7	0	**4**	**8**	3	7

	x_0	...	x_j	...	x_i	...
h_1						
...						
h_l			δ_j		e_i	
...						
$h_{l'}$			e_j			
...						

Fig. 9. The second solution property.

Let $i \neq j$, $e_i = h_l(x_i)$, $e_j = h_{l'}(x_j)$ and $l < l'$. Then the second property is (see Fig. 9):

$$e_i \neq \delta_j \text{ where } \delta_j = h_l(x_j). \tag{3.2}$$

This property ensures that the retrieval process works properly.

Given a solution $(e_0, e_1, \ldots, e_{n-1})$, $e_i = h_{l_i}(x_i)$, $i = 0, 1, \ldots, n-1$, the corresponding *HIT* table is constructed by letting $HIT[h_{l_i}(x_i)] = l_i$ and setting the remaining entries to 0.

The problem we wish to solve can be expressed as follows. Given the table M, find all sequences of addresses $(e_0, e_1, \ldots, e_{n-1})$ which satisfy properties (3.1) and (3.2). The solutions are found by making use of an exhaustive search which systematically enumerates all solutions $(e_0, \ldots, e_{n-1})$ by considering all partial solutions $(e_0, \ldots, e_{i-1})$ that have the properties mentioned above. During the search we attempt to expand a partial solution $(e_0, \ldots, e_{i-1})$ into $(e_0, \ldots, e_{i-1}, e_i)$. If $(e_0, \ldots, e_{i-1}, e_i)$ does not have the desirable properties, then by induction, no extended sequence $(e_0, \ldots, e_{i-1}, e_i, \ldots, e_{n-1})$ can have these properties. In such a case we backtrack to the partial solution $(e_0, \ldots, e_{i-1})$ and then try to expand it with the next value of e_i.

For illustration, let us consider an example [102] of constructing a perfect hash function for a 2×3 table M. The set of keys is $S = \{x_0, x_1, x_2\}$ and the address space is 0..3. Fig. 10 shows the configurations as the expansion of the solution proceeds. In the beginning, an address is selected from the first column and row of M, i.e. from position [1, 0] (see Fig. 10(a)), so we have $(e_0, e_1, e_2) = (2, -, -)$. The next address is selected from position [1, 1], $(e_0, e_1, e_2) = (2, 2, -)$, as shown in Fig. 10(b). Since according to property (3.1) no two addresses in the solution can be the same, it is necessary to backtrack to the first column and select the address at position [2, 0], $(e_0, e_1, e_2) = (3, -, -)$ (Fig. 10(c)). Note that our search is pruned here as we need not check sequences (2, 2, 0) and (2, 2, 1). In Fig. 10(d) the address 2 at position [1, 1] is the same as the address at position [1, 0], so the property (3.2) is violated. We discard this address, and select the address 1 at position [2, 1], as shown in Fig. 10(e). Finally, the address 0 in column 3 at position [1, 2] is selected, and the complete solution is $(e_0, e_1, e_2) = (3, 1, 0)$. The corresponding *HIT* table has the form: $HIT = [1, 2, 0, 2]$.

The program which finds and prints all the solutions for a given table M is shown below [102]. It calls two boolean functions, *expand* and *check*. The first function

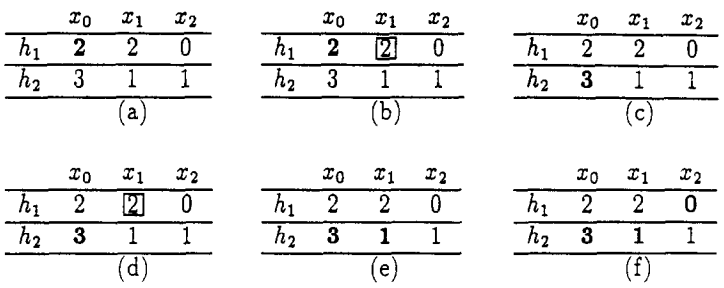

Fig. 10. Expanding the solution.

is used to expand a partial solution $(e_0,\ldots,e_{i-1})$ with the next value of the component e_i. It returns the value *true* if such an expansion can be done. Otherwise, the value *false* is returned, which causes backtracking in the main program. The second function, *check*, is used to test whether a partial solution $(e_0,\ldots,e_i)$ has the desirable properties. If so, the function returns the value *true* and the partial solution is expanded with the component e_{i+1}. Otherwise, the main program either expands the partial solution with the next value of e_i or performs backtracking.

```
M: array[1..s, 0..n − 1] of integer;
-- e is the vector of hash addresses corresponding
-- to the keys x0, x1, ..., xn−1 (a solution)
-- next_row[i] points to the next row to be tried in column i
e, next_row: array[0..n − 1] of integer;
-- row[i] records the row number of table M
-- from which the value e[i] was taken
row: array[0..n − 1] of integer;
HIT: array[0..m − 1] of integer; -- m denotes the size of the address space
-- back is the number of columns to go back during backtracking
i, k, back: integer;

function expand(i): boolean; -- expanding the solution
begin
    if next_row[i] ⩽ s then
        row[i] := next_row[i];
        e[i] := M[row[i], i];
        next_row[i] := next_row[i] + 1;
        return(true);
    else
        return(false); -- no hash addresses to try in a current column
    end if;
end;
```

```
function check(i): boolean; -- checking the properties
begin
    back:= 0;
    for k := 0 to i - 1 do
        if e[k] = e[i] then -- property (3.1) violated
            if row[k] = row[i] then
                if row[k] = 1 then
                    back := i - k; -- backtrack i - k columns
                else
                    back := 1; -- backtrack to the previous column
                end if;
            end if;
            return(false);
        else
            if (next_row[k] > next_row[i]) and (M[row[i], k] = e[i]) then
                return(false); -- property (3.2) violated
            end if;
        end if;
    end for;
    return(true);
end;

begin -- main program
    read(M);
    for i := 0 to n - 1 do
        next_row[i] := 1; -- start from the first row in each column
    end for;
    i := 0;
    while i ⩾ 0 do
        if (expand(i) and check(i)) then
            if i = n - 1 then -- compute HIT
                | Clear all entries of HIT;
                for k := 0 to n - 1 do
                    HIT[M[row[k], k]] := row[k];
                end for;
                print(e and HIT); -- solution found
            else
                i := i + 1;
            end if;
        else
            while back > 0 do
                next_row[i] := 1; -- backtracking
                i := i - 1;
```

```
            back := back − 1;
        end while;
        while next_row[i] > s do
            next_row[i] := 1; -- backtracking
            i := i − 1;
        end while;
    end if;
end while;
end;
```

Example 3.6. The program given above can be used to find all the solutions to the problem of constructing the perfect hash function presented in Example 3.2. There are four such solutions. The first one, of the minimum retrieval cost equals 2.375, is shown in Table 9. The others are listed in Table 10.

Yang and Du evaluated empirically the effectiveness of the method described above employing the random functions $h_1, h_2, \ldots, h_s$. The hash values of the functions were uniformly distributed over the address space $0..m-1$. Given the number of keys $n = 25$ and the loading factor of the hash table $\beta = n/m = 0.8$, the probability of getting a perfect hash function was measured for 100 test data sets. The probability was found to be around 0.3 for $s = 3$, and 0.97 for $s = 7$. The experiments also showed that the probability of getting a perfect hash function decreased quite rapidly as n increased. This difficulty can be overcome by using segmentation, as suggested by Du et al. (see Section 3.3).

3.5. A method of very small subsets

Winters observed that it is relatively easy to construct a minimal perfect hash function for sets of integer keys of small cardinality [99]. Therefore one possible solution to the problem of obtaining such a function for larger sets is to utilize a mutually exclusive and exhaustive subsetting of the input key set such that no subset has cardinality greater than the maximum that is computationally acceptable for the type of minimal perfect hash function being constructed. Subsequently, when a given key is to be retrieved, the key's subset is determined and the appropriate hash function is applied. The subsetting method proposed by Winters is suitable for the sets of keys

Table 10
The remaining solutions for the sample set of keys

e_0	e_1	e_2	e_3	e_4	e_5	e_6	e_7	cost
6	7	0	1	4	8	3	2	2.500
6	2	0	1	4	8	5	7	2.625
6	2	0	1	4	8	3	7	2.750

whose values are bounded above by n^2, where n is the number of keys in the input set. The keys which do not satisfy this condition must be reduced into the interval $[0, n^2 - 1]$. The reduction can be accomplished either deterministically, as shown in Section 1.3, or probabilistically. Winters showed that it is very likely that there exists an integer z, $z \leqslant n^2$, such that each key of the input set gives a unique value modulo that integer. Denoting by $\Pr(z \leqslant n^2)$ the probability of existing such an integer, we have [99, 100]

$$\Pr(z \leqslant n^2) \geqslant 1 - \left(1 - \left(\frac{n^2 - n - 2}{n^2 - n}\right)^{(n^2-n)/2}\right)^{(n^2+n+2)/2}.$$

Since for $n = 10$ it holds $\Pr(z \leqslant n^2) \geqslant (1.0 - 10^{-10})$, and because this is an increasing function of n, the probability of existing an integer which permits reducing the values of keys into the desired interval is very high.

The separation of the input set of keys $S = \{x_i\}$ into subsets of small cardinality is accomplished by a series of divisions. At level one, each key value x_i is divided by n. Each division generates an integer quotient q_i, $q_i \in [0, n-1]$, and an integer remainder r_i, $r_i \in [0, n-1]$. Clearly, every pair (q_i, r_i) is unique within S. The values q_i and r_i determine the subset into which the key x_i is placed. All the keys having a given q_i value constitute one subset, provided that the number of these keys does not exceed $\sqrt{n}$. If this number exceeds $\sqrt{n}$, the subset placement of keys having that q_i is based on those keys' r_i values. That is, all such keys with the same r_i value are in the same subset. A q_i or r_i value that is used to define a subset is called a *subset generator*. The following theorem proves that since no q_i generated subset can have more than $\sqrt{n}$ keys, also no r_i generated subset can have more than $\sqrt{n}$ keys.

Theorem 3.1 (Winters [99, Theorem 2.1]). *Let $\{y_j\}$ be the set of input keys (less than n^2) such that, when any one is divided by n, it gives an integer quotient having more than $\sqrt{n}$ keys mapping to it. Let $\{r_j\}$ be the set of integer remainders generated by $\{y_j\}$ mod n. No r_j can have more than $\sqrt{n}$ keys mapping to it.*

Proof. Assume there exists an r_j with more than $\sqrt{n}$ keys mapping to it. Clearly, no two keys which generated this r_j value could have generated the same quotient value, as every pair (q_j, r_j) is unique. This means that each of the keys mapping to r_j had a different quotient value. Therefore, there were more than $\sqrt{n}$ quotient groups, each with more than $\sqrt{n}$ elements mapping to r_j. This implies that there were more than n original input keys, a contradiction. □

It follows from the Theorem 3.1 that the input set of keys can be split at level one into subsets such that no subset can have more than $\sqrt{n}$ keys. At this point, some subsets may have cardinality small enough to allow minimal perfect hashing while others may require further subsetting at subsequent levels. The magnitude of keys in the latter subsets can be bounded above by n by determining a suitable reducer z.

Consequently, each such subset created at level one can be then split at level two using the same process employed at level one with the modification that the divisor used is $\sqrt{n}$ rather than n. By the argument given above, subsets produced at the second level contains no more than $\sqrt[4]{n}$ elements, and each element can be bounded by $\sqrt{n}$. The process can be repeated similarly at subsequently higher levels with appropriate divisors, until every subset is small enough. We make a conservative assumption that acceptable minimal perfect hash functions may be constructed only for subsets with no more than two elements.

The input set of keys is represented in the scheme by a series of tables of pointers. As each (sub)set is split, the system creates a table of pointers. Each created pointer corresponds to one of the possible subset generator values, q_i or r_i, and it is set to one of three values depending on the generated subset. If q_i or r_i does not generate a subset, i.e. too many (more than $\sqrt{n}$) or no elements map to it, the pointer is set to zero. If the subset generated by the associated generator value contains a single key, the pointer gives the address of that key. If the subset contains two keys, the pointer gives the address of hashing parameters to be used in computing the addresses of these keys. Finally, if the associated subset is too large to be hashed, i.e. its cardinality is greater than two but less than $\sqrt{n}$, the pointer gives the address of the start of a pointer table at the next higher level. This next level table is used with subset generators obtained when the subset associated with this current level pointer is further split at the next higher level. Thus, we have subsets being iteratively split, with pointers indicating what further processing the new subset should undergo, until all subsets have become small enough to be minimally perfectly hashed.

Example 3.7. Let us apply the method to the set of keys from the example in Section 3.2, $S = \{x_i\} = \{6800, 4188, 8832, 1110, 8839, 7320, 7318, 1229, 12\,990, 10\,238, 9876, 2837\}$, $|S| = n = 12$. To reduce the magnitude of keys into the interval $[0, n^2 - 1]$ we use the integer $z = 141$ obtaining $S' = \{x_i\} \bmod 141 = \{32, 99, 90, 123, 97, 129, 127, 101, 18, 86, 6, 17\}$. Dividing each key in S' by 12 gives the following (quotient, remainder) pairs: $32 \rightarrow (2, 8)$, $99 \rightarrow (8, 3)$, $90 \rightarrow (7, 6)$, $123 \rightarrow (10, 3)$, $97 \rightarrow (8, 1)$, $129 \rightarrow (10, 9)$, $127 \rightarrow (10, 7)$, $101 \rightarrow (8, 5)$, $18 \rightarrow (1, 6)$, $86 \rightarrow (7, 2)$, $6 \rightarrow (0, 6)$, $17 \rightarrow (1, 5)$. Observe that each quotient subset has less than $\lceil \sqrt{n} \rceil = \lceil \sqrt{12} \rceil = 4$ elements, hence all the quotients, given in Fig. 11(a), are the subset generators (for readability the quotient values that do not generate subsets are not shown). An entry of P_i indicates that a single key maps to this pointer, and so the pointer gives the address of that key. An entry of H_i indicates that exactly two keys map to this pointer, and so the pointer gives an address of a hash table. Each hash table contains a hash parameter a or b, and two more cells to keep the original keys. The hash functions used in this example have the forms $h(x) = x \bmod a$ and $h(x) = \lfloor x/b \rfloor$. An entry $T_{2,i}$ indicates that table i at level two is to be used for further splitting of the more than two keys that map to this pointer. The resulting structure for the set S is shown in Fig. 11(a)–(d).

(a) level 1

q or r	q-ptrs	r-ptrs
0	P_1	0
1	H_1	0
2	P_2	0
7	H_2	0
8	$T_{2,1}$	0
10	$T_{2,2}$	0

(b) level 2 $T_{2,1}$ (reducer = 97)

q or r	q-ptrs	r-ptrs
0	H_3	0
1	P_3	0
2	0	0
3	0	0

(c) level 2 $T_{2,2}$ (reducer = 123)

q or r	q-ptrs	r-ptrs
0	P_4	0
1	H_4	0
2	0	0
3	0	0

(d)

P_1: 9876 H_1: $a = 2$, 12990, 2837
P_2: 6800 H_2: $b = 87$, 8832, 10238
P_3: 1229 H_3: $b = 2$, 4188, 8839
P_4: 1100 H_4: $b = 5$, 7320, 7318

Fig. 11. The subsetting structure.

Suppose now that we want to retrieve the key 4188. The reduced value of the key is 4188 mod 141 = 99, which divided by 12 gives quotient 8 at level one. Items in this subset are processed next by table 1 at level two. Here 99 is reduced to 2 (mod 97), which is then divided by $\lceil \sqrt{n} \rceil = 4$. The pointer H_3 in $T_{2,1}$ associated with the obtained quotient 0 determines the parameter $b = 2$ of the hash function $h(x) = \lfloor x/b \rfloor$. Applying this function to the reduced key gives the address $h(0) = \lfloor 0/2 \rfloor = 0$ which holds the key 4188.

The time complexity of the membership query for a key in Winters' scheme depends on the number of levels of processing and the evaluation time of the associated hash function.

Theorem 3.2 (Winters [99, Theorem 3.1]). *There will be no more than* $1 + \log\log n$ *levels in the subsetting structure.*

Proof. By Theorem 3.1 and the fact that $n, \sqrt{n}, \sqrt[4]{n}, \ldots$ are used as divisors, it follows that no table at level i will contain more than $n^{(1/2)^{i-1}}$ elements. A minimal perfect hash function with acceptable parameter size can be found for any subset consisting of two elements. Therefore, the structure will never have more than s levels, where $n^{(1/2)^{s-1}} = 2$ or equivalently, $n = 2^{2^{s-1}}$. Thus we have $s - 1 = \log\log n$. □

By Theorem 3.2 and the assumption that the hash function can be evaluated in constant time, the overall complexity of retrieval in the scheme is $O(\log\log n)$.

It has been also proved in [99] that with some modifications to the construction phase, the overall space complexity of the scheme that can be achieved is $O(n \log\log n)$. The time complexity of the construction phase is $O(n^2 \log\log n)$.

3.6. Bibliographic remarks

Motivated by the approach of Fredman et al., Cormack et al. devised a scheme which allows both insertion and deletion of keys, uses $n+O(1)$ space, but may require $O(n^n/(n-1)!)$ time to construct a perfect hash function [30]. Note, that if we relax the restriction that each entry of a hash table can receive only one key and instead allow it to keep $b > 1$ keys (where b is a constant), an efficient dynamic perfect hash function can be designed. That variant has been studied by Larson and Ramakrishna [68, 82] in the context of accessing a file kept on a disk. Because a single disk access retrieves a block of t keys, a hash function is said to be perfect if no block receives more than $b \leqslant t$ records. Larson and Ramakrishna, using segmentation and a hash function similar to those applied by Fredman et al., implemented an efficient dynamic perfect hashing for external files. The load factor achieved by their scheme exceeded 70%.

Another thread was investigated by Dietzfelbinger and Meyer auf der Heide [39] and by Dietzfelbinger et al. [35]. By using certain classes of universal hash functions they show that the FKS probabilistic method can construct a perfect hash function in $\Theta(n)$ time, with the probability $1 - O(n^{-\varepsilon})$, for some constant $\varepsilon > 0$ (see Section 6.2).

Chapter 4. Reducing the search space

4.1. Mapping, ordering, searching

The algorithms presented in this chapter generate a (minimal) perfect hash function by searching through the space of all possible functions from S, $|S| = n$, into $[0, m-1]$. As mentioned in Section 1.1, the total number of such functions is m^n and only $m(m-1)\cdots(m-n+1)$ of them are perfect. Therefore any method using such an approach must employ techniques that substantially reduce the search space, or otherwise it is doomed to failure.

The majority of the methods we present here use a *Mapping, Ordering, Searching* (MOS) approach, a description coined by Fox et al. (cf. [48]). In this approach, the construction of a (minimal) perfect hash function is accomplished in three steps.

First, the mapping step transforms a set of keys from the original universe to a universe of hash identifiers. A *hash identifier* is a collection of selected properties of a key, such as symbols comprising the key, their positions of occurrence in the key, the key length, etc. The mapping step has to preserve "uniqueness", i.e. if two keys are distinguishable in the original universe, they must also be distinguishable in the universe of hash identifiers. In general, this requirement is hard to achieve and may need a significant effort.

The second step, ordering, places the keys in a sequence which determines the precedence in which hash values are assigned to keys. Keys are divided into subsets, $W_0, W_1, \ldots, W_k$, such that $W_0 = \emptyset$, $W_i \subset W_{i+1}$, and $W_k = S$, for some k. The hash values

must be assigned to all the members of a subset simultaneously, since assignment of a hash value to any of them determines hash values for all others.

The third step, searching, tries to extend the desired hash function h from the domain W_{i-1} to W_i. This is the only step of potentially exponential time complexity, since if the searching step encounters W_i for which h cannot be extended, it backtracks to earlier subsets, assigns different hash values to the keys of these subsets and tries again to recompute hash values for subsequent subsets.

In this chapter only character keys are considered. However, for simplicity, we denote keys and their subsets by x and S, respectively, instead of x_α and S_α. A key is treated as a one-dimensional table of characters, $x[1 \ldots l]$. Thus, $x[i]$, $1 \leqslant i \leqslant l$, denotes the ith character of x, and $|x| = l$ is the length of a key.

In Section 4.2 we present Cichelli's algorithm which uses an exhaustive backtrack search and some heuristics to limit the number of hash value collisions occurring during the search. Section 4.3 describes Sager's mincycle algorithm that grew from an attempt to optimize Cichelli's method. Sections 4.4–4.6 examine three algorithms that are substantial improvements over Sager's mincycle algorithm. In Section 4.4 the algorithm which exploits a highly skewed vertex degree distribution in the dependency graph devised by Fox et al. is presented. Sections 4.5 and 4.6 describe the improvements to Sager's algorithm given by Czech and Majewski. Finally, in Sections 4.7 and 4.8 the enhancements of the algorithms for minimal perfect hashing proposed by Fox et al. are discussed.

4.2. *Using letter frequency*

One of the first successful algorithms for generating minimal perfect hash functions was presented by Cichelli [28]. This algorithm can be viewed as an early representative of the MOS approach. In Cichelli's algorithm the *mapping* step is very simple. The mapping functions are defined as follows: $h_0(x) = |x| = l$, $h_1(x) = x[1]$ and $h_2(x) = x[l]$. The form of Cichelli's hash function is

$$h(x) = h_0(x) + g(h_1(x)) + g(h_2(x))$$

where g is a function, implemented as a lookup table, which maps characters into integers. The values of g are computed employing an exhaustive backtrack search, whose execution time is potentially exponential in the size of the key set. Cichelli proposed two ordering heuristics on a set of keys that cause hash value collisions to occur during the search as early as possible, thus effectively pruning the search tree and speeding up the computations. These heuristics are:

1. Arrange the keys in a list in decreasing order of the sum of frequencies of occurrence of each key's first and last letter.
2. Modify the order of the keys obtained such that any key whose hash value has already been determined (because its first and last letters have appeared in keys previous to the current one) is placed next in the list.

The backtrack search attempts to find the values of the g table by processing the keys in the order defined by the above heuristics. The values of g are restricted to a predefined range $[0, b]$, where b is an arbitrarily chosen integer. (Cichelli proposed no method of choosing a value for b. This problem is important since b determines the branching factor of the search tree.) Cichelli's search procedure works as follows [28]:

> "If both the first and last letter of the key already have associated values, try the key. If either the first or last letter has an associated value, vary the associated value of the unassigned character from zero to the maximum allowed associated value, trying each occurrence. If both letters are as yet unassociated, vary the first and then the second, trying each possible combination. (An exception test is required to catch situations in which the first and last letters are the same.) Each "try" tests whether the given hash value is already assigned and, if not, reserves the value and assigns the letters. If all keys have been selected, then halt. Otherwise, invoke the search procedure recursively to place the next key. If the "try" fails, remove the key by backtracking."

Example 4.1. Consider the following set of the city names: GLIWICE, BRISBANE, CANBERRA, KATOWICE, MELBOURNE, DARWIN, WARSZAWA, WROCLAW, SYDNEY and KRAKOW. The frequencies of occurrence of the first and last letters of these names are:

Letter	A	B	C	D	E	G	K	M	N	S	W	Y
Frequency	2	1	1	1	4	1	2	1	1	1	4	1

The list of names ordered by decreasing sum of frequencies of occurrence of first and last letters is (the sum is given in parentheses): WROCLAW (8), KATOWICE (6), WARSZAWA (6), KRAKOW (6), GLIWICE (5), BRISBANE (5), MELBOURNE (5), CANBERRA (3), DARWIN (2) and SYDNEY (2). After rearrangement of the list according to Cichelli's second heuristic we get: WROCLAW (8), KATOWICE (6), KRAKOW (6), WARSZAWA (6), GLIWICE (5), BRISBANE (5), MELBOURNE (5), CANBERRA (3), DARWIN (2) and SYDNEY (2). During the search the hash value $h(\mathtt{WROCLAW}) = |\mathtt{WROCLAW}| + g(\mathtt{W}) + g(\mathtt{W}) = 7$ is assigned by setting $g(\mathtt{W}) = 0$. Then, we can set $g(\mathtt{K}) = g(\mathtt{E}) = 0$, which gives $h(\mathtt{KATOWICE}) = |\mathtt{KATOWICE}| + g(\mathtt{K}) + g(\mathtt{E}) = 8$. For the next name, KRAKOW, both g values for the first and last letters have been already determined, but the location computed for this name is empty, so we can assign $h(\mathtt{KRAKOW}) = |\mathtt{KRAKOW}| + g(\mathtt{K}) + g(\mathtt{W}) = 6 + 0 + 0 = 6$. Continuing this process we obtain the following table defining the associated value for each letter:

Letter	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
g		1	3	4	0	0	0	3	0	0	0	0	0	3	8	0	0	0	0	0	0	0	0	0	0	9	0

and the contents of the hash table for addresses from 6 to 15 is: KRAKOW, WROCLAW, KATOWICE, WARSZAWA, GLIWICE, BRISBANE, MELBOURNE, CANBERRA, DARWIN and SYDNEY.

Example 4.2 (Cichelli [28]). Consider a set of 36 reserved words for Pascal. The following table defines the associated value for each letter:

Letter	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
g	1	1	15	1	0	0	15	3	15	13	0	0	15	15	13	0	15	0	14	6	6	14	10	6	0	13	0

The corresponding hash table, with hash values running from 2 to 37, is as follows: DO, END, ELSE, CASE, DOWNTO, GOTO, TO, OTHERWISE, TYPE, WHILE, CONST, DIV, AND, SET, OR, OF, MOD, FILE, RECORD, PACKED, NOT, THEN, PROCEDURE, WITH, REPEAT, VAR, IN, ARRAY, IF, NIL, FOR, BEGIN, UNTIL, LABEL, FUNCTION and PROGRAM.

As reported by Cichelli, finding a solution for the Pascal reserved words took about seven seconds on a PDP-11/45 using a straightforward implementation of his algorithm.

The advantages of Cichelli's method are that it is simple, effective and machine independent, because the character codes used in a particular machine never enter the hash value calculations. Unfortunately, mainly because of the exponential time for the searching step, the algorithm is practical only for relatively small sets (up to about 45 keys). Furthermore, no two keys can have the same length and the same first and the last letter.

4.3. Minimum length cycles

A novel solution to the problem of generating minimal perfect hash functions was suggested by Sager [85]. Sager's algorithm, called the *mincycle* algorithm, grew from an attempt to optimize Cichelli's method. The mincycle algorithm can be viewed as another example of the MOS approach. It introduces better mapping functions and, more importantly, proposes an effective key ordering heuristic. The mincycle algorithm searches for a minimal perfect hash function of the form:

$$h(x) = (h_0(x) + g(h_1(x)) + g(h_2(x))) \bmod n,$$

where h_0, h_1 and h_2 are auxiliary pseudorandom functions, g is a function, implemented as a lookup table, whose values are established during an exhaustive backtrack search, and n is the size of the hash table. The pseudorandom functions which are used in the mapping step are defined as follows:

$$h_0(x) = |x| + \sum_{j=1 \text{ by } 3}^{|x|} \text{ord}(x[j]),$$

$$h_1(x) = \left(\sum_{j=1 \text{ by } 2}^{|x|} \text{ord}(x[j]) \right) \bmod r,$$

$$h_2(x) = \left(\left(\sum_{j=2 \text{ by } 2}^{|x|} \text{ord}(x[j]) \right) \bmod r \right) + r.$$

The integer r is a parameter of the mincycle algorithm. It determines the size of table g, $|g| = 2r$, containing the description of the minimal perfect hash function. Therefore it is desirable for g to be as small as possible. On the other hand, the larger r is, the greater the probability of finding a minimal perfect hash function. Sager chose r to be the smallest power of 2 greater than $n/3$. The functions defined above convert each key x into a (hopefully) unique hash identifier which is a triple of integers $(h_0(x), h_1(x), h_2(x))$. If the triples for all keys from the input set are not unique, the functions h_0, h_1 and h_2 must be modified. However, Sager did not provide a general method suitable to accomplish such a modification.

In the ordering step, keys are divided into subsets, $W_0, W_1, \ldots, W_k$, such that $W_0 = \emptyset$, $W_i \subset W_{i+1}$, and $W_k = S$, for some k. The sequence of these subsets is called a *tower*, and each subset $X_i = W_i - W_{i-1}$ in the tower is called a *level*. To construct the tower, the dependency graph $G = (R, E)$, $R = \{h_1(x) : x \in S\} \cup \{h_2(x) : x \in S\}$, $E = \{\{h_1(x), h_2(x)\} : x \in S\}$ for the input set of keys S is built. Each key x corresponds to edge $e = \{h_1(x), h_2(x)\}$. The dependency graph, which is bipartite and consists of at most $|R| = 2r$ vertices and $|E| = n$ edges, represents constraints among keys. Observe that assigning a location in the hash table for key x requires selecting the value $Q(x) = g(h_1(x)) + g(h_2(x))$. There may exist a sequence of keys $(x_0, x_1, \ldots, x_{\tau-1})$ such that $h_1(x_i) = h_1(x_{i+1})$ and $h_2(x_{i+1}) = h_2(x_{(i+2) \bmod \tau})$, for $i = 0, 2, 4, \ldots, \tau-2$. Once keys $x_0, x_1, \ldots, x_{\tau-2}$ are assigned to locations in the hash table, both $g(h_1(x_{\tau-1}))$ and $g(h_2(x_{\tau-1}))$ are set (note that $h_1(x_{\tau-2}) = h_1(x_{\tau-1})$ and $h_2(x_{\tau-1}) = h_2(x_0)$). Hence, key $x_{\tau-1}$ cannot be assigned an arbitrary location, but must be placed in the hash table at location

$$h(x_{\tau-1}) = (h_0(x_{\tau-1}) + Q(x_{\tau-1})) \bmod n.$$

In our sequence, the keys $x_0, x_1, \ldots, x_{\tau-2}$ are independent, i.e. they have a choice of a location in the hash table (through their $Q(x_i)$ values), whereas the key $x_{\tau-1}$ is dependent, i.e. it has not such a choice. These keys are called *canonical* and *noncanonical*, respectively. It is easy to see that

$$Q(x_{\tau-1}) = g(h_1(x_{\tau-1})) + g(h_2(x_{\tau-1})) = \sum_{p \in path(x_{\tau-1})} (-1)^p Q(x_p)$$

where $path(x_{\tau-1})$ is a sequence of words $\langle x_0, x_1, \ldots, x_{\tau-2} \rangle$, and thus

$$h(x_{\tau-1}) = \left(h_0(x_{\tau-1}) + \sum_{p \in path(x_{\tau-1})} (-1)^p Q(x_p) \right) \bmod n. \tag{4.1}$$

If the place $h(x_{\tau-1})$ is occupied, a collision arises and no minimal perfect hash function for the selected values of g can be found. In such a case, the search step backtracks and tries to find different values of g that do not lead to a collision.

This dependency of keys is reflected in the dependency graph by cycles. There may be, and usually are, many cycles. Each of them corresponds to a sequence of keys similar to those described. The core of Sager's heuristic in the ordering step is to find an order of keys such that keys without a choice are processed by the searching

```
Construct a dependency graph G = (R, E),
  where R = {h1(x) : x ∈ S} ∪ {h2(x) : x ∈ S}, E = {{h1(x), h2(x)} : x ∈ S};
Associate a list of keys x for which {h1(x), h2(x)} = e, with each edge e ∈ E;
k := 0;
W0 := ∅;
while E is not empty do
  Choose the edge e = {u, v} from E lying on the maximum number
    of minimal length cycles in G;
  k := k + 1;
  Xk := {x : x is associated with the edge e};
  Select an arbitrary key in Xk to be canonical;
  Form path(x) for the remaining (noncanonical) keys x in Xk;
  Wk := Wk-1 ∪ Xk;
  Remove e from E;
  Merge u and v into a new vertex u';
end while;
```

Fig. 12. The ordering step of the mincycle algorithm.

step as soon as possible. The ith level of the tower, $X_i = W_i - W_{i-1}$, is selected by the following method (see Fig. 12). Choose an edge (possibly a multiple edge, i.e. an edge to which more than one key is mapped) lying on the maximum number of minimal length cycles in the dependency graph. Let X_i be the set of keys associated with the chosen edge. Remove the edge from the graph and merge its endpoints. Repeat this procedure until all edges are removed from the dependency graph. In each set X_i an arbitrary key is selected to be canonical, and the remaining become noncanonical. A single edge always corresponds to a canonical key.

The maximum number of repetitions of the above procedure is n, and an edge lying on the maximum number of cycles of minimal length can be selected in $O(r^3) = O(n^3)$ time [84]. Consequently, the time complexity of the ordering step is $O(n^4)$.

The heuristic (which gave the name to the whole algorithm) tries to ensure that each time an edge is selected, it is done in such a way that the maximum number of dependent keys is placed in a tower. Moreover, because the selected edge not only lies on a cycle of minimal length but also on the maximum number of such cycles, it ensures that in subsequent steps the selection will be optimal or at least close to optimal.

Example 4.3. Consider the set of the three-letter abbreviations for the months of the year. The values of the mapping functions for it are given in Table 11. The dependency graph is constructed so that for each name a corresponding edge is created. The endpoints of each edge are defined by the values of the h_1 and h_2 functions. The dependency graph is shown in Fig. 13(a).

In each iteration of the ordering step, an edge lying on the maximum number of cycles of minimal length in the dependency graph is chosen. Thus the first edge selected is $\{0, 13\}$ and $X_1 = \{\texttt{FEB}, \texttt{JUN}, \texttt{AUG}\}$. Suppose that the key FEB is selected to be canonical in X_1, i.e. its hash value is computed as

$$h(\texttt{FEB}) = (h_0(\texttt{FEB}) + Q(\texttt{FEB})) \bmod 12 = (9 + Q(\texttt{FEB})) \bmod 12.$$

Table 11
Values of functions h_0, h_1 and h_2 for the month abbreviations ($r = 8$)

Month	h_0	h_1	h_2
JAN	1	0	9
FEB	9	0	13
MAR	4	7	9
APR	4	3	8
MAY	4	6	9
JUN	1	0	13
JUL	1	6	13
AUG	4	0	13
SEP	10	3	13
OCT	6	3	11
NOV	5	4	15
DEC	7	7	13

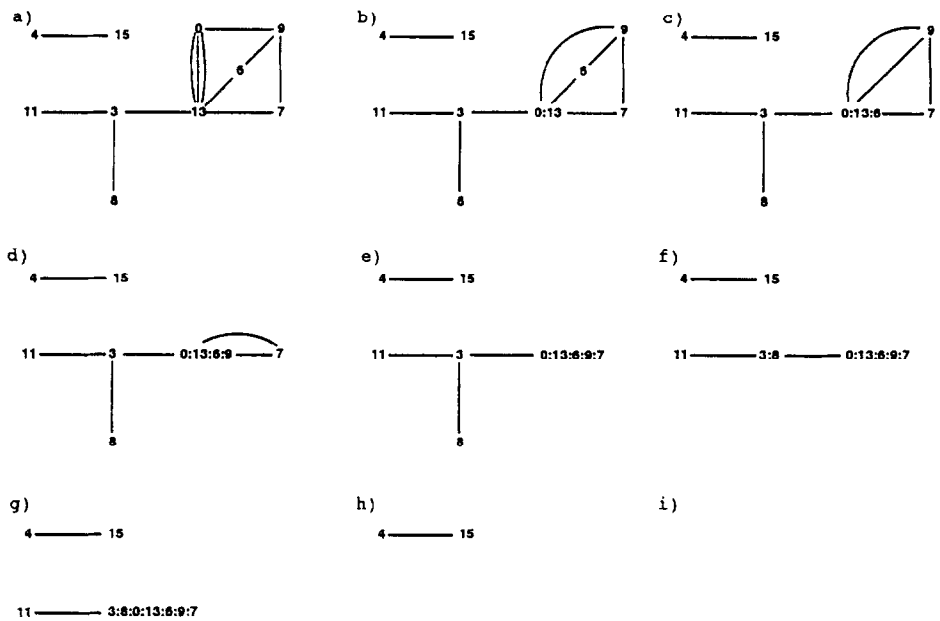

Fig. 13. The ordering step for the month abbreviations.

Then for the noncanonical keys JUN and AUG the following cycles of length $l = 2$ exist: (x_0, x_1) = (FEB, JUN) and (x_0, x_1) = (FEB, AUG). According to (4.1) the hash values for keys x_1 are computed as

$$h(x_1) = \left(h_0(x_1) + \sum_{p \in path(x_1)} (-1)^p Q(x_p) \right) \bmod n$$

where $path(x_1) = \langle 0 \rangle$, which for JUN and AUG gives

$$h(\mathtt{JUN}) = (h_0(\mathtt{JUN}) + Q(\mathtt{FEB})) \bmod 12 = (1 + Q(\mathtt{FEB})) \bmod 12,$$

$$h(\mathtt{AUG}) = (h_0(\mathtt{AUG}) + Q(\mathtt{FEB})) \bmod 12 = (4 + Q(\mathtt{FEB})) \bmod 12.$$

In the modified graph, formed by merging the endpoints of the selected edge, there are five edges each belonging to a cycle of length 3 (Fig. 13(b)). Hence we may choose any of these five edges. Assume that the next edge is (6, 13), therefore $X_2 = \{\mathtt{JUL}\}$. Now in the modified graph a multiple edge has appeared (Fig. 13(c)). Consequently, it is picked in the next step and the keys $\{\mathtt{JAN}, \mathtt{MAY}\}$ form X_3. Assume that JAN is selected to be the canonical key in X_3. Then for the key MAY the cycle exists: $(x_0, x_1, x_2, x_3) = (\mathtt{JAN}, \mathtt{FEB}, \mathtt{JUL}, \mathtt{MAY})$, and $path(x_3) = \langle 0, 1, 2 \rangle$. The hash value for $x_3 = \mathtt{MAY}$ is computed as follows:

$$h(\mathtt{MAY}) = (h_0(\mathtt{MAY}) + Q(\mathtt{JAN}) - Q(\mathtt{FEB}) + Q(\mathtt{JUL})) \bmod 12$$

$$= (4 + Q(\mathtt{JAN}) - Q(\mathtt{FEB}) + Q(\mathtt{JUL})) \bmod 12.$$

After the deletion of the multiple edge $\{9, 13\}$ another one is formed (Fig. 13(d)). This new multiple edge is selected in the next iteration, and $X_4 = \{\mathtt{MAR}, \mathtt{DEC}\}$. If MAR is selected as the canonical key in X_4 then

$$h(\mathtt{MAR}) = (h_0(\mathtt{MAR}) + Q(\mathtt{MAR})) \bmod 12 = (4 + Q(\mathtt{MAR})) \bmod 12,$$

and

$$h(\mathtt{DEC}) = (h_0(\mathtt{DEC}) + Q(\mathtt{FEB}) - Q(\mathtt{JAN}) + Q(\mathtt{MAR})) \bmod 12$$

$$= (7 + Q(\mathtt{FEB}) - Q(\mathtt{JAN}) + Q(\mathtt{MAR})) \bmod 12.$$

The remaining graph (Fig. 13(e)) is acyclic and the sequence in which the edges are selected is arbitrary. We choose $X_5 = \{\mathtt{APR}\}$, $X_6 = \{\mathtt{SEP}\}$, $X_7 = \{\mathtt{OCT}\}$ and $X_8 = \{\mathtt{NOV}\}$.

The searching step of the mincycle algorithm attempts to extend h incrementally from $W_0 = \emptyset$ to $W_k = S$, where k is the height of the tower. Thus, we have to find $Q(x_i) \in [0, n-1]$, $i = 1, 2, \ldots, k$, such that values $h(x_i) = (h_0(x_i) + Q(x_i)) \bmod n$, for canonical keys $x_i \in Y$, and $h(x_j) = (h_0(x_j) + \sum_{p \in path(x_j)} (-1)^p Q(y_p)) \bmod n$, for noncanonical keys $x_j \in S - Y$ are all distinct, i.e. for any $x_1, x_2 \in S$, $h(x_1) \neq h(x_2)$.

The $Q(x_i)$ values, denoted $Q(i)$ in the algorithm, are found during the exhaustive search at every level X_i of the tower (Fig. 14). A $Q(i)$ value corresponds to a canonical key which appears as the first key in each set X_i. The keys at the consecutive levels of the tower, $X_1, X_2, \ldots, X_k$, are numbered with $j = 0, 1, \ldots, |W_k| - 1$.

Example 4.4. For the last example, the searching step has to find the Q values for canonical keys: $Q(\mathtt{FEB})$, $Q(\mathtt{JUL})$, $Q(\mathtt{JAN})$, $Q(\mathtt{MAR})$, $Q(\mathtt{APR})$, $Q(\mathtt{SEP})$, $Q(\mathtt{OCT})$ and

```
for i := 1 to k do
  Q(i) := n;
end for;
i := 1;
while i ∈ [1, k] do
  Q(i) := (Q(i) + 1) mod (n + 1);
  if Q(i) = n then
    i := i − 1; -- backtrack
  else
    noconflict := true;
    j := |W_{i−1}|;
    while noconflict and j < |W_i| do
      h(x_j) := (h_0(x_j) + Σ_{p∈path(x_j)} (−1)^p Q(p)) mod n;
      if ∀y ∈ [0, j − 1], h(x_y) ≠ h(x_j) then
        j := j + 1;
      else
        noconflict := false;
      end if;
    end while;
    if noconflict then
      i := i + 1;
    end if;
  end if;
end while;
success := i > k;
```

Fig. 14. The searching step of the mincycle algorithm.

$Q(\texttt{NOV})$, such that the hash addresses for all keys are different. The exhaustive search for these values is done at every consecutive level of the tower, starting with the initial value 0 (see Fig. 14). Thus, setting $Q(\texttt{FEB}) = 0$ at the first level of the tower gives $h(\texttt{FEB}) = (9 + Q(\texttt{FEB})) \bmod 12 = (9 + 0) \bmod 12 = 9$, $h(\texttt{JUN}) = (1 + Q(\texttt{FEB})) \bmod 12 = (1 + 0) \bmod 12 = 1$ and $h(\texttt{AUG}) = (4 + Q(\texttt{FEB})) \bmod 12 = (4 + 0) \bmod 12 = 4$. These hash values do not collide with each other so $Q(\texttt{FEB}) = 0$ is accepted. The assignment $Q(\texttt{JUL}) = 0$ at the second level of the tower results in collision because the hash address $h(\texttt{JUL}) = (h_0(\texttt{JUL}) + Q(\texttt{JUL})) \bmod 12 = (1 + 0) \bmod 12 = 1$ has been already taken by the key JUN. The next value tried is $Q(\texttt{JUL}) = 1$ which gives $h(\texttt{JUL}) = (h_0(\texttt{JUL}) + Q(\texttt{JUL})) \bmod 12 = (1 + 1) \bmod 12 = 2$. Since this location is empty the value of $Q(\texttt{JUL}) = 1$ is accepted. For level three both $Q(\texttt{JAN}) = 0$ and $Q(\texttt{JAN}) = 1$ lead to conflict. The proper value is $Q(\texttt{JAN}) = 2$, since the address $h(\texttt{JAN}) = (h_0(\texttt{JAN}) + Q(\texttt{JAN})) \bmod 12 = (1 + 2) \bmod 12 = 3$ is not occupied. For the noncanonical key at this level we get

$$\begin{aligned} h(\texttt{MAY}) &= (4 + Q(\texttt{JAN}) - Q(\texttt{FEB}) + Q(\texttt{JUL})) \bmod 12 \\ &= (4 + 2 - 0 + 1) \bmod 12 = 7. \end{aligned}$$

```
procedure traverse(i: vertex);
begin
  mark(i) := true;
  for j := 0 to 2r − 1 do
    if Qmat(i, j) < n and not mark(j) then
      g(j) := (Qmat(i, j) − g(i)) mod n;
      traverse(j);
    end if;
  end for;
end traverse;

begin -- FINDg
  for i := 0 to 2r − 1 do
    mark(i) := false;
  end for;
  for i := 0 to 2r − 1 do
    if not mark(i) then
      g(i) := 0;
      traverse(i);
    end if;
  end for;
end;
```

Fig. 15. Finding the g function.

As location 7 is empty, the search can be continued at the next level. The final Q values for canonical keys and the placement of keys in the hash table T are as follows:

key	FEB	JUL	JAN	MAR	APR	SEP	OCT	NOV
Q(key)	0	1	2	1	4	0	5	7

address	0	1	2	3	4	5	6	7	8	9	10	11
T[address]	NOV	JUN	JUL	JAN	AUG	MAR	DEC	MAY	APR	FEB	SEP	OCT

If the searching step is successful in extending the function h to S then all that remains is to find a function g such that $h(x) = (h_0(x) + g(h_1(x) + g(h_2(x))) \bmod n$ is a minimal perfect hash function for $x \in S$. The $O(n^2)$ algorithm FINDg of Fig. 15 finds such a function. (Indeed the algorithm can be easily improved to run in $O(n)$ time by changing its data structures.) The algorithm uses a two-dimensional table *Qmat* computed by the ordering step. For each canonical key x_i such that $\{h_1(x_i), h_2(x_i)\} = \{i, j\}$, the value $Qmat(i, j) = Qmat(j, i) = Q(x_i)$. All other elements of *Qmat* are set to n.

Example 4.5. To complete the last two examples, consider computing the g values by the FINDg algorithm. The Q values for canonical keys found in the searching step are

inserted into the *Qmat* table. For example, the elements *Qmat*(0, 9) and *Qmat*(9, 0) are assigned the value $Q(\texttt{JAN})=2$ as $h_1(\texttt{JAN})=0$ and $h_2(\texttt{JAN})=9$. The same is done for all other canonical keys. The FINDg algorithm using the *Qmat* table performs a depth-first search of the dependency graph given in Fig. 13(a) and computes the g values for all visited vertices. First, after setting $g(0) = 0$, it computes the g value for vertex 9:

$$g(9) = (Qmat(0, 9) - g(0)) \bmod 12 = (2 - 0) \bmod 12 = 2.$$

The next vertex visited is 7, for which we get

$$g(7) = (Qmat(9, 7) - g(9)) \bmod 12 = (1 - 2) \bmod 12 = 11.$$

Then, in the same fashion, the remaining g values are computed. We obtain the following table for g:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$g(i)$	0	0	0	0	0	0	1	11	4	2	0	5	0	0	0	7

Based on experimental results, Sager claimed that the time required to complete the ordering step dominates the potentially exponential time of the searching step and, hence, the algorithm runs in $O(n^4)$ time. He gave no proof of this claim but mentioned a formal proof which shows that for certain parameters, for which the running time of the algorithm is $O(n^6)$, the ordering step can always be expected to dominate the searching step (but see Section 6.7). Experimental evidence shows that, although the mincycle algorithm performs well for sets of size up to about 300 keys, it is impractical for larger sets. The reasons are:

1. The running time of the algorithm is closer to hours than minutes for sets of more than 300 keys, even for fast machines (cf. Table 13).
2. The memory requirements of the algorithm are very high. The mincycle algorithm, even when implemented very carefully, needs $O(n^2)$ space, with quite a large constant factor. For example, for relatively small sets (about 700 keys) it uses more than 2 MB of memory.
3. Poor pseudorandom functions that do not yield distinct triples when used with sets of several hundred keys.

These drawbacks were addressed in subsequent improvements of the mincycle algorithm and are discussed in the following sections.

4.4. Skewed vertex degree distribution

In the following three sections we examine three algorithms that are substantial improvements over Sager's mincycle algorithm [32, 33, 48, 49]. Their structure is the same as that of the mincycle algorithm. Instead of changing the basic idea, the authors enhanced some steps of it, making their solutions applicable for key sets of significant cardinality – greater than 1000 keys.

Fox et al. (FHCD) observed that the vertex degree distribution in the dependency graph is highly skewed [48, 49]. This fact provided the inspiration to carry out the ordering of keys so that the searching step can be executed more efficiently. Furthermore, FHCD proposed the use of randomness whenever possible. They selected the more complicated but guaranteed to work mapping functions defined as

$$h_0(x) = \left(\sum_{i=1}^{|x|} T_0[i, x[i]] \right) \bmod n,$$

$$h_1(x) = \left(\sum_{i=1}^{|x|} T_1[i, x[i]] \right) \bmod r,$$

$$h_2(x) = \left(\left(\sum_{i=1}^{|x|} T_2[i, x[i]] \right) \bmod r \right) + r, \tag{4.2}$$

where the integer parameter r is typically $n/2$ or less, and T_0, T_1 and T_2 are two-dimensional tables of random integers defined for each character of the alphabet Σ and each position of a character within a key. The tables are repeatedly generated in the mapping step until each key is mapped into a unique triple of integers.

Let $t = nr^2$ be the size of the universe of triples, and assume that the triples $(h_0(x), h_1(x), h_2(x))$, $x \in U$, are random. Then, the probability of distinctness for n triples chosen uniformly at random from t triples is [49]

$$p(n, t) = \frac{t(t-1)\cdots(t-n+1)}{t^n} \sim \exp\left(-\frac{n^2}{2t}\right).$$

For $t = nr^2$ and $r = n/2$, we have

$$p(n, t) \sim \exp\left(-\frac{2}{n}\right).$$

This probability goes quickly to 1 for increasing n. We can also prove that on the average it will never be necessary to generate the random tables more than twice. Thus, if the size of the alphabet Σ and the maximum key length are fixed, then the mapping step runs in $O(n)$ expected time.

The observation concerning the vertex degree distribution which led to a better ordering step is as follows. For a random bipartite dependency graph, the probability p that an edge is adjacent to a given vertex v is $p = 1/r$. Let D be the random variable that equals the degree of v. Then D is binomially distributed with parameters n and p, and the probability of D having the value d is

$$\Pr(D = d) = \binom{n}{d} p(1-p)^{n-d}.$$

For large n, we can use the Poisson approximation to obtain

$$\Pr(D = d) \sim e^{-\lambda} \lambda^d / d!$$

```
Empty VHEAP;
i := 0;
while some vertex of nonzero degree is not SELECTED do
  Insert a vertex w of maximum degree in the
    dependency graph into VHEAP;
  while VHEAP is not empty do
    i := i + 1;
    SELECT a vertex v_i of maximum degree in VHEAP and
      delete it from VHEAP;
    for all w adjacent to v_i do
      if w is not SELECTED and w is not in VHEAP then
        Insert w into VHEAP;
      end if;
    end for;
  end while;
end while;
```

Fig. 16. The ordering step of FHCD's algorithm.

where $\lambda = np = n/r$. Since $n = 2r$, the expected number of vertices of degree d is

$$n\Pr(D = d) \sim \frac{ne^{-\lambda}\lambda^d}{d!} = \frac{ne^{-2}2^d}{d!}.$$

Consequently, the expected number of vertices of degree 0, 1, 2, 3 and 4 are approximately $0.135n$, $0.271n$, $0.271n$, $0.18n$ and $0.09n$, respectively. As we can see, a random dependency graph contains vertices of low degree more likely than vertices of high degree. Based on this observation, FHCD suggested the following heuristic algorithm for building the tower. Select a vertex of maximum degree in the dependency graph. Beginning from this vertex traverse the graph always selecting as next vertex to visit a vertex of the maximum degree among neighbors of already visited vertices. If the currently selected vertex is v, then add on a level of the tower corresponding to vertex v, $X(v)$, all keys for which associated edges have one endpoint equal to v and the other belonging to the set of visited vertices.

Note that the above heuristic algorithm by visiting vertices with high degree first, makes the dependent keys to be placed at the initial levels of the tower. Fig. 16 shows the algorithm for the ordering step. It maintains the unselected vertices in a heap *VHEAP* ordered by degree. The algorithm actually produces an ordering of the nonzero degree vertices. From the vertex ordering, the levels of the tower are easily derived. Let the vertex ordering for a connected component of the dependency graph be $v_0, v_1, \ldots, v_t$. Then the level corresponding to a vertex v_i, $X(v_i)$, $i = 1, 2, \ldots, t$, is the set of keys associated with the edges incident both to v_i and to a vertex earlier in the ordering. The number of keys in any set $X(v_i)$ cannot be larger than the degree of v_i – typically it is smaller.

Example 4.6. Given the graph of Fig. 13(a), the ordering of vertices belonging to the larger connected component, and the levels of the tower obtained by the above algorithm are as follows: $v_0 = 13$, $v_1 = 0$, $v_2 = 3$, $v_3 = 9$, $v_4 = 6$, $v_5 = 7$, $v_6 = 8$, $v_7 = 11$, and $X(0) = \{\texttt{FEB}, \texttt{JUN}, \texttt{AUG}\}$, $X(3) = \{\texttt{SEP}\}$, $X(9) = \{\texttt{JAN}\}$, $X(6) = \{\texttt{MAY}, \texttt{JUL}\}$, $X(7) = \{\texttt{MAR}, \texttt{DEC}\}$, $X(8) = \{\texttt{APR}\}$, $X(11) = \{\texttt{OCT}\}$.

Since each heap operation can be accomplished in $O(\log 2r) = O(\log n)$ time, the complexity of the ordering step is $O(n \log n)$. Because the vertex degrees of a random dependency graph are mostly small, the heap operations can be optimized to run faster. This optimization consists in implementing *VHEAP* as a series of stacks and one heap of bounded size. The stacks are provided for vertices of degree 1, 2, 3 and 4 – a single stack for each degree. The heap *VHEAP* contains all vertices of degree $\geqslant 5$. Since most vertices have degree between 1 and 4, the size of the heap is kept below a constant. Depending on the degree, a vertex w is added either to the appropriate stack or inserted into *VHEAP*. Both these operations take constant time, and thus the time for the ordering step incorporating this optimization is actually $O(n)$.

At every iteration of the searching step, the keys of one level of the tower are to be placed in the hash table. Each level $X(v_i)$ corresponds to a vertex v_i. Consider, for example, a vertex $v_i \in [r, 2r-1]$ (a similar argument, which follows, can be presented for vertices $v_i \in [0, r-1]$). Every key $x \in X(v_i)$ has the same $h_2(x)$ value, and thus the same $g(h_2(x))$ value equals $g(v_i)$. Since the $g(h_1(x))$ values for $x \in X(v_i)$ have been already selected in the previous iterations of the searching step, and $h_0(x)$ values are computed in the mapping step, $h(x)$ is determined by the selection of $g(v_i)$ value. Let $b(x) = h_0(x) + g(h_1(x))$. Then

$$h(x) = (b(x) + g(v_i)) \bmod n.$$

The $b(x) \bmod n$ values for $x \in X(v_i)$ yield offsets from $g(v_i) \bmod n$ to the hash values of x. FHCD call the set of $b(x) \bmod n$ values a *pattern* (mod n). The pattern may be considered as being in a circle, or a disk, of n sectors and subject to *rotation* by the value $g(v_i)$. The disk is identified with the hash table of n locations, some of which may be already occupied. Given the keys of a level, $x \in X(v_i)$, the searching step must select a value $g(v_i)$ that puts all the $b(x) + g(v_i)$ values in empty locations of the hash table simultaneously. FHCD referred to this process as *fitting a pattern into a disk*. When a value $g(v_i)$ is to be selected, there are usually several choices that place all keys of $X(v_i)$ into empty locations in the hash table. FHCD noticed that an appropriate value for $g(v_i)$ should be picked at random rather than, for example, picking the smallest acceptable value. Therefore, while looking for $g(v_i)$ the searching step uses a pseudorandom probe sequence to access the locations of the hash table. Fig. 17 shows the algorithm for the searching step. It processes only a single connected component (of $t+1$ vertices) of the dependency graph. The remaining components are processed in the same fashion. The algorithm generates a set of 20 small primes – or fewer if n is small – that do not divide n. Each time a consecutive level is to be placed

```
Generate a set R of 20 small primes;
Assign an arbitrary value to g(v_0);
for i := 1 to t do
  Establish a random probe sequence s_0 = 0, s_1 = q, s_2 = 2q,
    ..., s_{n-1} = (n-1)q, by choosing q at random from R;
  j := 0;
  repeat
    collision := false;
    if v_i ∈ [0, r-1] then
      for all x ∈ X(v_i) do
        h(x) = (h_0(x) + g(h_2(x)) + s_j) mod n;
      end for;
      if any place h(x) is occupied then
        collision := true;
      end if;
    else -- v_i ∈ [r, 2r-1]
      for all x ∈ X(v_i) do
        h(x) = (h_0(x) + g(h_1(x)) + s_j) mod n;
      end for;
      if any place h(x) is occupied then
        collision := true;
      end if;
    end if;
    j := j + 1;
    if j > n - 1 then
      fail;
    end if;
  until not collision;
end for;
```

Fig. 17. The searching step of FHCD's algorithm.

into the hash table, one of the primes, q, is chosen at random to be s_1 and is used as an increment to obtain the remaining s_j, $j \geqslant 2$. In effect, the pseudorandom sequence is $0, q, 2q, 3q, \ldots, (n-1)q$. Although the randomness of such a sequence is limited, FHCD found it sufficient in the experimental tests. The search carried out in the algorithm of Fig. 17 is not exhaustive. Once $n-1$ unsuccessful trials to place a level into the hash table is done, the algorithm detects a failure. According to FHCD, for a large value of n and an appropriate choice of r, it is very unlikely to occur. A reasonable response to this rare event, as suggested, is to restart the minimal perfect hash function algorithm with new values of h_0, h_1 and h_2, or to use backtrack which assigns new g values to earlier vertices and tries again to recompute g values for successive subsets.

Example 4.7. Let us illustrate the searching step for the subset of three-letter abbreviations for the months of the year. The subset comprises the keys associated with the edges of the largest connected component in Fig. 13(a). The values of the functions

h_0, h_1 and h_2 are given in Table 11, and the ordering of vertices and the levels of the tower produced by the ordering step are in Example 4.4. The searching step assigns the g values to the vertices $v_0 = 13$, $v_1 = 0$, $v_2 = 3$, $v_3 = 9$, $v_4 = 6$, $v_5 = 7$, $v_6 = 8$ and $v_7 = 11$, in that order. The value for $v_0 = 13$ is assigned arbitrarily, so let $g(v_0) = g(13) = 0$. Then the first level $X(0) = \{\texttt{FEB}, \texttt{JUN}, \texttt{AUG}\}$ is considered. The hash values for these keys are computed as follows:

$$h(\texttt{FEB}) = (h_0(\texttt{FEB}) + g(h_1(\texttt{FEB})) + g(h_2(\texttt{FEB}))) \bmod 12$$

$$= (9 + g(0) + g(13)) \bmod 12 = (9 + s_j + 0) \bmod 12,$$

$$h(\texttt{JUN}) = (1 + s_j + 0) \bmod 12,$$

$$h(\texttt{AUG}) = (4 + s_j + 0) \bmod 12.$$

This level gives a pattern $\{9, 1, 4\}$ and is easily placed with $s_j = 0$, and so $s_j = g(v_1) = g(0) = 0$, $h(\texttt{FEB}) = 9$, $h(\texttt{JUN}) = 1$ and $h(\texttt{AUG}) = 4$. The next level $X(3) = \{\texttt{SEP}\}$ consists of a single key. The key of such a level can be placed in any location in the hash table, since having

$$h(\texttt{SEP}) = (10 + g(3) + g(13)) \bmod 12 = (10 + s_j + 0) \bmod 12,$$

we may set s_j arbitrarily. Let $s_j = g(3) = 9$, which gives $h(\texttt{SEP}) = 7$. The same applies to the following level $X(9) = \{\texttt{JAN}\}$ which comprises a single key too. $g(v_3) = g(9) = 10$ places JAN in location $h(\texttt{JAN}) = (1 + 0 + 10) \bmod 12 = 11$. The level $X(6) = \{\texttt{MAY}, \texttt{JUL}\}$ makes the pattern $\{2, 1\}$ since we have

$$h(\texttt{MAY}) = (4 + s_j + 10) \bmod 12,$$

$$h(\texttt{JUL}) = (1 + s_j + 0) \bmod 12.$$

By setting $s_j = g(6) = 1$ the level is placed at locations $h(\texttt{MAY}) = 3$ and $h(\texttt{JUL}) = 2$. For level $X(7) = \{\texttt{MAR}, \texttt{DEC}\}$ the pattern $\{2, 7\}$ is obtained as

$$h(\texttt{MAR}) = (4 + s_j + 10) \bmod 12,$$

$$h(\texttt{DEC}) = (7 + s_j + 0) \bmod 12.$$

This pattern is placed with $g(7) = 3$ at locations $h(\texttt{MAR}) = 5$ and $h(\texttt{DEC}) = 10$. The remaining keys in the tower – APR and OCT – can be placed in any of the unoccupied locations of the hash table, as remarked before.

Timings for each step of the algorithm are given in Table 12 [49]. Each entry in the table is a result of a single experiment. They were measured using a Sequent Symmetry with 32 MB of main memory, running on a single 80386 processor running at about 4 MIPS. The parameter r was fixed to $r = 0.3n$, i.e. the space requirement for the g table was 0.6 words/key (the g table contains $|g| = 2r$ words). The

Table 12
Timings in seconds for FHCD's algorithm

n	Bits/key	Mapping	Ordering	Searching	Total
32	3.0	0.37	0.02	0.03	0.42
64	3.6	0.75	0.03	0.07	0.85
128	4.2	0.60	0.05	0.08	0.73
256	4.8	0.97	0.05	0.18	1.20
512	5.4	1.23	0.08	0.35	1.67
1024	6.0	1.50	0.17	0.67	2.33
2048	6.6	2.42	0.30	1.67	4.38
4096	7.2	3.47	0.62	3.13	7.22
8192	7.8	5.53	1.27	5.92	12.72
16384	8.4	9.87	2.52	12.05	24.43
32768	9.0	18.78	5.05	24.62	43.45
65536	9.6	35.68	10.20	50.02	95.90
131072	10.2	69.70	20.15	101.08	190.93
262144	10.8	137.97	40.30	201.57	379.83
524288	11.4	275.25	81.23	406.58	763.07

second column of the table shows the same space requirement expressed in bits/key, calculated as $(|g| \log n)/n = 0.6 \log n$. Note that each value in the g table is from the range $[0, n-1]$, thus it can be stored on $\log n$ bits. The number of stacks used in the ordering step was 12. FHCD underlined that although there was some variation in processing time because of the probabilistic nature of the operations, with an appropriate value for r the algorithm was finding a minimal perfect hash function with high probability.

Based on the results from Table 12 it was asserted that the searching step requires $O(n)$ time on average to complete its task. No rigorous proof of this requirement, however, was given. As the mapping step runs in $O(n)$ expected time, and the optimized ordering step needs $O(n)$ time, FHCD made a claim that their minimal perfect hash function algorithm runs practically in $O(n)$ time, however see Section 6.7.

4.5. *Minimum length fundamental cycles*

Czech and Majewski (CM) proposed three major enhancements to the mincycle algorithm [32]. They were introduced in the two last steps of Sager's algorithm, i.e. in the ordering and searching steps. The initial step, mapping, was left unchanged.

The first enhancement came from the observation that the collisions among the keys can be reflected by a fundamental set of cycles with respect to a spanning tree of the dependency graph. Since the colliding keys are to be inserted into the tower as soon as possible, it seems to be desirable to perform this task based on the fundamental set of cycles of minimum total length. Unfortunately, the problem of finding a spanning tree of a graph which induces the fundamental set of cycles of minimum total length is known to be $\mathcal{NP}$-complete [34]. Deo et al. proposed some heuristic algorithms to

solve this problem [34]. The algorithms perform a modified breadth-first search (BFS) through a graph, to generate a suboptimal spanning tree T for which $L(T) - L(T_{\min})$ is hopefully small, where $L(\,)$ denotes the total length of the fundamental cycle set. The suggested heuristic search deviates from general BFS in the criteria used to select a new vertex to explore from. Deo et al. say:

> Whenever the partial tree is to be extended, the new vertex to be explored is not the "oldest" unexplored vertex, as in a straightforward BFS, but is selected according to some function of the degrees of the vertices.

Following this approach, CM developed a heuristic search algorithm which performs better than those described in [34], with regard to the total length of the fundamental cycle set obtained. The search in this algorithm proceeds as follows. Perform a BFS through a graph beginning with the vertex of maximum degree. Whenever a new vertex is considered, it is a vertex with the largest number of neighbors in the partial spanning tree, determined at the time the vertex was inserted in the partial spanning tree (with ties broken arbitrarily).

CM noticed that the keys of S which correspond to cycles and multiple edges in the dependency graph should be considered first while searching for a minimal perfect hash function. Thus the ordering step takes care of them before the other keys. In fact, all other keys are *free*, i.e. arbitrary hash values can be assigned to them. (Refer to Example 4.4 in which the keys `SEP`, `JAN`, `APR` and `OCT` – making up the separate levels – could be assigned any location in the hash table.) Therefore these keys are put on the last levels of the tower.

Consequently, after generating the dependency graph G, the subgraph $G_1 = (R_1, E_1)$ that contains only the multiple edges and the edges lying on cycles is constructed. For this purpose the biconnected components of G are found. The edges of the biconnected components of size greater than 1 are placed into E_1. Once G_1 is found, its spanning forest is formed by using the heuristic search algorithm defined above, for each connected component of G_1. The cycles induced by the forest are then passed to the ordering step which builds the tower trying to maximize the cardinalities of its initial levels. This is achieved by using a *signature* $\mathcal{S}(e) = [n_2, n_3, \ldots, n_{\max}]$ which indicates for each edge e on how many cycles of length between 2 and max the edge lies on. While building the tower, a key associated with an edge e with the largest signature (lexicographically, with respect to n_i's), the "most conflicting" one, is put into the tower and then the edge is deleted from all cycles it lies on. Since the lengths of the cycles decrease, the signatures of the edges of the cycles to which e belongs must be modified. If edge e is deleted from cycle $(e, e_1, e_2, \ldots, e_{l-1})$ of initial length l, then for all other edges e_i, $1 \leqslant i \leqslant l-1$, the value of n_l of their signatures should be decreased by 1, and n_{l-1} increased by 1. This process is repeated until all keys associated with the edges of the fundamental cycles are moved into the tower. Then, the remaining (nonconflicting) keys are placed into the tower.

Fig. 18 presents the algorithm for the ordering step (edges and the corresponding keys are used interchangeably). The algorithm maintains a heap *EHEAP* of edges

```
Mark all edges e ∈ E1 as not selected;
k := 0; -- current level of the tower
Empty EHEAP;
while E1 is not empty do
  Insert an edge e ∈ E1 with the largest signature S(e)
    into EHEAP;
  Mark e as selected;
  E1 := E1 − {e};
  while EHEAP is not empty do
    e := del_max(EHEAP); -- take an edge with the largest
                         -- signature S(e) from EHEAP;
    if is_path(h1(e), h2(e)) then -- noncanonical edge;
      Form path(e);
      Add a key corresponding to edge e on level t of the tower
        such that t = max{i : Q(i) is on path(e)};
    else -- canonical edge
      k := k + 1;
      Add a key corresponding to edge e on level k of the tower;
      Insert e into the spanning forest;
    end if;
    Delete e from all the fundamental cycles it belongs to
      and modify the signatures of these cycles;
    for all the fundamental cycles of length 1 do
      Insert edge e of the cycle into EHEAP
        provided it is marked as not selected;
      Mark the inserted edge e as selected;
      E1 := E1 − {e};
    end for;
  end while;
end while;
Add free keys into the tower;
```

Fig. 18. The ordering step of CM's algorithm.

ordered by signatures. It builds the spanning forest containing the canonical edges that have already been inserted in the tower. The forest allows us to decide whether an edge should be placed on a current, a lower, or the next level of the tower. The forest is also used to form the *paths* for noncanonical edges. A function *is_path* finds a path (if any) between vertices $h_1(e)$ and $h_2(e)$ in the forest and calculates *path*(e). The theoretical analysis and the experimental tests showed that the time complexity of the ordering step is $O(n^2)$ [32].

The next two modifications enhance the searching step of Sager's algorithm. The first introduces the search tree pruning. When it happens that a key collides with some other key, an immediate backtrack is executed to the highest placed key in the tower for which assigning a different location in the hash table resolves the collision. The second modification is derived from careful analysis of the method of

Table 13
Timings in seconds for the mincycle and CM's algorithms

		Mincycle algorithm			CM's algorithm		
n	Bits/ key	Mapping & ordering	Searching & FINDg	Total	Mapping & ordering	Searching & FINDg	Total
25	5.9	0.224	0.011	0.235	0.226	0.017	0.243
50	7.2	1.444	0.013	1.457	0.249	0.017	0.266
100	8.5	12.555	0.019	12.574	0.289	0.020	0.309
200	9.8	130.897	0.029	130.926	0.380	0.029	0.409
300	7.0	83.194	0.051	83.245	0.965	0.044	1.009
500	9.2	n/a	n/a	n/a	1.308	0.056	1.364
1000	10.2	n/a	n/a	n/a	3.266	0.102	3.368
1500	7.2	n/a	n/a	n/a	10.865	0.264	11.129
2000	11.2	n/a	n/a	n/a	0.564	0.208	9.772
2500	9.2	n/a	n/a	n/a	20.152	0.319	20.471

Note: n/a: not available due to excessive execution times

placement of canonical keys into the hash table. Thus, the searching step of Sager's algorithm starts the address assignment to those keys with all $Q(i)$ values equal 0, $i = 1,2,\ldots,k$ (see Fig. 14). This means that an attempt is made to place the keys at locations determined by the h_0 values of the keys. Since the randomness of the h_0 function is weak, it was observed that the addresses of the canonical keys concentrated in two clusters in the hash table. CM proposed placing the canonical keys with a random sample of $Q(i)$ values. These two modifications sped up the searching step substantially.

Table 13 contains the execution times of the algorithms for randomly chosen sets of words from the UNIX dictionary. Each value in the table is an average over 100 experiments. The tests were conducted on a Sun 3/60 workstation. The parameter r was set to the smallest power of 2 greater than $n/3$, which defines the space requirements of the minimal perfect hash function shown in the second column.

The experimental results indicate that the execution time of the searching step of CM's algorithm is dominated by the execution time of the first two steps of the algorithm. Therefore CM argued that the algorithm requires $O(n^2)$ time. However, like the FHDC solution, no formal proof of this complexity has been provided and see Section 6.7.

4.6. *A linear-time algorithm*

In a further development of the minimum length fundamental cycles algorithm, Czech and Majewski [33] set the number of vertices of the dependency graph to $2r = 2n$. With this assumption they proved that the algorithm finds a minimal perfect hash function with $O(n \log n)$ bit space description in expected time $O(n)$.

The mapping step of the algorithm transforms the input keys into distinct triples of integers $(h_0(x), h_1(x), h_2(x))$, applying the following formulas:

$$h_0(x) = \left(\sum_{i=1}^{|x|} T_0[i_s, x[i]] \right) \bmod n,$$

$$h_1(x) = \left(\sum_{i=1}^{|x|} T_1[i_s, x[i]] \right) \bmod r,$$

$$h_2(x) = \left(\left(\sum_{i=1}^{|x|} T_2[i_s, x[i]] \right) \bmod r \right) + r,$$

where T_0, T_1 and T_2 are two dimensional tables of random integers defined as before, r is a parameter equals n, $i_s = ((|x|+i) \bmod |x|_{\max})+1$ determines the starting position for fetching numbers from the tables, and $|x|_{\max}$ is the maximum key length. The formulas slightly differ from their predecessors (4.2), as the starting position for fetching the random numbers from the tables depends on the key length. This ensures more even utilization of the table elements and increases the randomness of dependency graphs that are constructed from triples. The transformation of keys into triples takes $O(n)$ expected time.

The values $h_1(x)$ and $h_2(x)$ define the bipartite dependency graph G in which each key is associated with edge $e = (h_1(x), h_2(x))$. As have been already noticed, the constraints among keys are reflected by cycles in this graph. The structure of the cycles is used for ordering keys into the tower so that the dependent (noncanonical) keys are placed at the lowest possible levels of the tower. The algorithm for the ordering step is shown in Fig. 19. The order of keys in the tower is found heuristically. CM's heuristic for building the tower is simple. Like in the minimum length fundamental cycles algorithm (see Section 4.5), the graph G_1 (which consists of the multiple edges and the edges lying on cycles in the dependency graph G) is formed. The multiple edges are then put into the tower beginning with the edges of highest multiplicity. These edges are then deleted from G_1 by merging their endpoints, and a fundamental set of cycles for the resultant graph is found by using a breadth-first search. The fundamental cycles are considered in order of increasing length. For this purpose a heap *CHEAP* containing the cycles ordered by length is maintained. The keys corresponding to the edges of the shortest cycle are put into the tower one at a time. Each edge is then deleted from all the cycles it lies on, and the next shortest cycle is considered. As the last, free keys are added into the tower. While building the tower a spanning forest that consists of the edges corresponding to the canonical keys in the tower is constructed. The forest is used to compute the *path*s for noncanonical keys.

To determine the time complexity of the ordering step the following lemmas estimating the expected total length of cycles, the expected number of cycles, and the

```
k := 0; -- current level of the tower
for each multiple edge E in G1 do
  k := k + 1; -- begin the next level of the tower
  Add to the tower a key corresponding to an arbitrary edge e ∈ E as canonical
    and the keys corresponding to the remaining edges of E as noncanonical;
  Delete E from G1 by merging the endpoints of E;
  Insert e into the spanning forest;
  Form path's for noncanonical keys;
end for;
Find a fundamental cycle set for G1 − {E} and structure
  the cycles into CHEAP;
Form the sets of cycles members[e] each edge e in G1 − {E} belongs to;
while CHEAP not empty do
  c := del_min(CHEAP); -- take a shortest cycle from CHEAP
  for all edges e of c do
    if a key corresponding to edge e is not in the tower then
      if is_path(h1(e), h2(e)) then -- noncanonical edge
        Form path(e);
        Add a key corresponding to edge e on a current level of the tower;
      else -- canonical edge
        k := k + 1;
        Add a key corresponding to edge e on a current level of the tower;
        Insert e into the spanning forest;
      end if;
      for j ∈ members[e] do
        Decrease length of cycle[j];
        Restore CHEAP; -- move cycle[j] up
      end for;
    end if;
  end for;
end while;
Add free keys into the tower;
```

Fig. 19. The ordering step of CM's algorithm.

expected number of multiple edges in a random bipartite graph of $2r = 2n$ vertices and n edges are used.

Lemma 4.1. *Let C_{2d} denote the number of cycles of length $2d$. Then the expected total length of cycles in the graph is $\sum_{d=1}^{n/2} 2d \times E(C_{2d}) = O(\sqrt{n})$.*

Proof. To build a cycle of length $2d$, we select $2d$ vertices and connect them with $2d$ edges in any order. There are $\binom{n}{d}^2$ ways to choose $2d$ vertices out of $2n$ vertices of a graph, $d!d!/2d$ ways of connecting them into a cycle, and $(2d)!$ possible orderings of the edges. The cycle can be embedded into the structure of the graph in $\binom{n}{n-2d}n^{2(n-2d)}$ ways. Hence, the number of graphs containing a cycle of length $2d$ is $\binom{n}{d}^2((d!)^2/2d)(2d)!\binom{n}{n-2d}n^{2(n-2d)}$. Therefore, the expected number of cycles of length

$2d$ in the graph is

$$E(C_{2d}) = \frac{\binom{n}{d}^2 \frac{(d!)^2}{2d}(2d)!\binom{n}{n-2d}n^{2(n-2d)}}{n^{2n}}$$

$$= \frac{1}{2d}\left(\frac{n(n-1)\cdots(n-d+1)}{n^d}\right)^2 \frac{n(n-1)\cdots(n-2d+1)}{n^{2d}}.$$

Using an asymptotic estimate from Palmer [77]:

$$\frac{(n)_i}{n^i} = \frac{n(n-1)\cdots(n-i+1)}{n^i} \sim e^{-\frac{i^2}{2n}-\frac{i^3}{6n^2}},$$

the sum can be approximated with an integral

$$\sum_{d=1}^{n/2} 2d \times E(C_{2d}) = \sum_{d=1}^{n/2}\left(\frac{n(n-1)\cdots(n-d+1)}{n^d}\right)^2 \frac{n(n-1)\cdots(n-2d+1)}{n^{2d}}$$

$$\sim \int_1^{n/2} \mathrm{e}^{-3d^2/n} dd \leqslant \sqrt{\frac{n}{3}}\int_0^{\infty} \mathrm{e}^{-z^2} dz = \sqrt{\frac{n}{3}}\frac{\sqrt{\pi}}{2} \approx \frac{1}{2}\sqrt{n}$$

$$= \mathrm{O}(\sqrt{n}). \quad \square$$

Lemma 4.2. *The expected number of cycles in the graph is* $\sum_{d=1}^{n/2} E(C_{2d}) = \mathrm{O}(\ln n)$.

Proof.

$$\sum_{d=1}^{n/2} E(C_{2d}) = \sum_{d=1}^{n/2} \frac{1}{2d}\left(\frac{n(n-1)\cdots(n-d+1)}{n^d}\right)^2 \frac{n(n-1)\cdots(n-2d+1)}{n^{2d}}$$

$$\sim \sum_{d=1}^{n/2} \frac{1}{2d} e^{-3d^2/n} = \frac{1}{2}\left[\sum_{d=1}^{\sqrt{n}} \frac{e^{-3d^2/n}}{d} + \sum_{d=\sqrt{n}+1}^{n/2} \frac{e^{-3d^2/n}}{d}\right]$$

$$\leqslant \frac{1}{2e^{3/n}} H_{\sqrt{n}} + \frac{1}{2e^3}\left(H_{n/2} - H_{\sqrt{n}}\right) \sim \frac{1}{4}\ln n + \frac{1}{80}\ln n = \mathrm{O}(\ln n). \quad \square$$

Lemma 4.3. *Let* e_j *denote the number of multiple edges of multiplicity* j *in the graph. Then* $\lim_{n\to\infty} \sum_{j=3}^{n} E(e_j) = 0$.

Proof. The expected number of multiple edges of multiplicity j is $E(e_j) = \mathcal{N} \cdot p_j$, where $\mathcal{N} = n^2$ is the number of possible places an edge can be inserted in the graph, and p_j is the probability that j edges are inserted in a given place. Since p_j is determined by the binomial distribution

$$p_j = \binom{n}{j}\left(\frac{1}{\mathcal{N}}\right)^j \left(1 - \frac{1}{\mathcal{N}}\right)^{n-j}$$

we get

$$E(e_j) = \mathcal{N}\binom{n}{j}\left(\frac{1}{\mathcal{N}}\right)^j\left(1-\frac{1}{\mathcal{N}}\right)^{n-j} \leqslant \binom{n}{j}\frac{1}{\mathcal{N}^{j-1}} \leqslant \frac{n^j}{j!}\frac{1}{n^{2(j-1)}} = \frac{1}{j!\,n^{j-2}}$$

which gives the lemma. □

Using the above lemmas we can prove

Lemma 4.4. *The expected time complexity of the ordering step is* $O(n)$.

Proof. The ordering step comprises finding the fundamental cycles and building the tower. Parallel edges of multiplicity 2 are cycles of length 2 so using Lemmas 4.1 and 4.3 all multiple edges can be omitted in the analysis. The cost of finding the fundamental cycles of G_1 is proportional to the total length of these cycles. By Lemma 4.1 this length cannot exceed $O(\sqrt{n})$. While placing the edges of the fundamental cycles in the tower, a heap of cycles is maintained. The following operations are executed: (i) selecting the shortest cycle in the heap; (ii) finding a path in a spanning forest between the end vertices of a noncanonical edge, and making the path list for it; (iii) restoring the heap. Let v denote the expected number of the fundamental cycles. The operation (i) takes time $O(v \log v)$, that by Lemma 4.2 is $O(\ln n \log(\ln n))$. Finding the *path*s for all noncanonical edges in operation (ii) requires at most $O(n_1 v) = O(\sqrt{n} \ln n)$ time, whereas making the path lists is done in $O(n_1) = O(\sqrt{n})$ time (n_1 denotes the number of edges lying on the fundamental cycles). The cost of operation (iii) is at most $O(n_1 \log v) = O(\sqrt{n} \log(\ln n))$. All these costs imply that the expected time complexity of the ordering step is less than $O(n)$. □

In the searching step, a set of values $Q(x_i)$, $i = 1,2,\ldots,k$, is to be determined. The $Q(x_i)$ values guarantee that the hash addresses $h(x_i) = (h_0(x_i) + Q(x_i)) \bmod n$, for canonical keys $x_i \in Y$, and $h(x_j) = (h_0(x_j) + \sum_{p \in path(x_j)}(-1)^p Q(y_p)) \bmod n$, for noncanonical keys $x_j \in S - S_a - Y$ are all distinct, i.e. for any x_1, $x_2 \in S - S_a$, $h(x_1) \neq h(x_2)$ (S_a is a set of keys corresponding to free edges, see Section 4.5). The $Q(x_i)$'s are found during the exhaustive search at every level X_i of the tower. The search starts with $Q(x_i) = 0$ for each canonical key x_i, i.e. an attempt is made to place it at position $h_0(x_i)$ in the hash table. Note that since values h_0 are random, all the canonical keys of the tower are placed in a random probe of places. Once the hash value for the canonical key x_i on a given level of the tower is found, the value of $Q(x_i)$ is known. It enables us to compute the hash values for the noncanonical keys at the level. Following [49], let us call the set of the hash values of keys at a given level X_i a *pattern* of size $s = |X_i|$. Clearly, if all places defined by the pattern are not occupied, the task on a given level is done and the next level can be processed. Otherwise, the pattern is moved up the table modulo n until the place where it fits is found. Except for the first level of the tower, this search is conducted for a hash table that is partially filled. Thus, it may happen that no place for the pattern is found. In

```
i := 1;
while i ∈ [1, k] do
  Place the canonical key x_i ∈ X_i in the next table location (beginning at
      position h_0(x_i)), and compute Q(x_i) = (h(x_i) − h_0(x_i)) mod n;
  if x_i not placed then
    i := i − 1; −− backtrack
  else
    for each noncanonical key x_j ∈ X_i do
      h(x_j) = (h_0(x_j) + Q(x_i)) mod n;
    end for;
    if all locations h(x_j) are empty then
      i := i + 1;
    end if;
  end if;
end while;
Compute h values for free keys;
```

Fig. 20. The searching step of CM's algorithm.

such a case the searching step backtracks to earlier levels, assigns different hash values for keys on these levels, and then again recomputes the hash values for successive levels (Fig. 20).

Having $Q(x_i)$'s, the values of table g can be computed by making use of the $O(n)$ procedure FINDg presented in [85] (see Section 4.3).

Example 4.8. Consider the set of names of some popular alcoholic beverages: $S = \{$`beer`, `brandy`, `champagne`, `cider`, `cognac`, `liqueur`, `porto`, `rum`, `sherry`, `vodka`, `whisky`, `wine`$\}$. Table 14 contains the values of the functions h_0, h_1 and h_2 generated in the mapping step for the keys from S, where $r = n = |S| = 12$. Fig. 21 shows the corresponding bipartite dependency graph defined by the h_1 and h_2 values. Note that the graph is almost acyclic due to its sparsity. It contains only one multiple edge or, in other words, one cycle of length 2. Based on this graph the ordering step creates the tower of keys. The keys associated with the multiple edge $\{8, 23\}$ are put on the first level of the tower: $X_1 = \{x_0, x_1\} = \{$`vodka`, `whisky`$\}$. Let $x_0 =$ `vodka` be the canonical key at this level. Then $path(x_1) = \langle 0 \rangle$ (cf. Example 4.3). Once edge $\{8, 23\}$ is deleted from the dependency graph (see Fig. 19), it becomes acyclic. Therefore the rest of the keys can be put on the consecutive levels of the tower in any order, a single key at each level.

Let us assume that the tower has the form: $X_1 = \{$`vodka`, `whisky`$\}$, $X_2 = \{$`beer`$\}$, $X_3 = \{$`brandy`$\}$, $X_4 = \{$`champagne`$\}$, $X_5 = \{$`cider`$\}$, $X_6 = \{$`cognac`$\}$, $X_7 = \{$`liqueur`$\}$, $X_8 = \{$`porto`$\}$, $X_9 = \{$`rum`$\}$, $X_{10} = \{$`sherry`$\}$ and $X_{11} = \{$`wine`$\}$. In the searching step (see Fig. 20), the key `vodka` is placed at position $h($`vodka`$) = h_0($`vodka`$) = 2$ in the hash table, and the value $Q($`vodka`$) = (2 - 2) \bmod 12 = 0$ is computed. For the noncanonical key `whisky` we get the hash address $h($`whisky`$) = (h_0($`whisky`$) +$

Table 14
Values of the mapping functions h_0, h_1 and h_2

Key	h_0	h_1	h_2
beer	11	5	23
brandy	7	8	22
champagne	9	11	17
cider	1	9	20
cognac	6	0	19
liqueur	2	4	17
porto	11	5	12
rum	5	4	14
sherry	8	2	15
vodka	2	8	23
whisky	10	8	23
wine	11	8	13

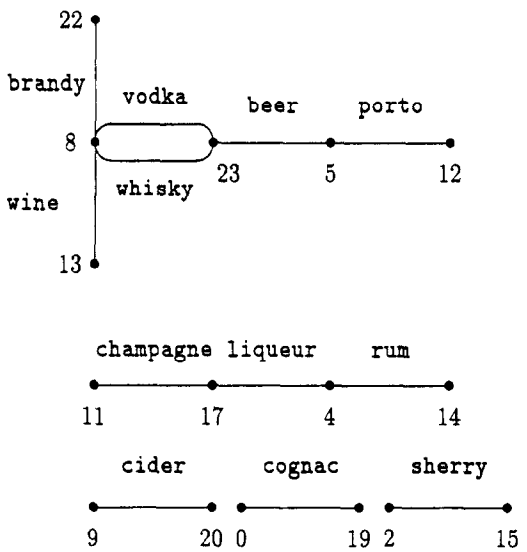

Fig. 21. The bipartite dependency graph.

$Q(\texttt{vodka})) \bmod 12 = (10 + 0) \bmod 12 = 10$. The keys of the next five levels of the tower, beer, brandy, champagne, cider and cognac, are placed with no conflicts in the hash table at positions defined by their h_0 values. The key liqueur cannot be placed at position $h_0(\texttt{liqueur}) = 2$, since this position has already been taken by the key vodka. So it is placed at the next free position in the hash table by setting $Q(\texttt{liqueur}) = 1$:

$$h(\texttt{liqueur}) = (h_0(\texttt{liqueur}) + Q(\texttt{liqueur})) \bmod 12 = (2 + 1) \bmod 12 = 3.$$

Table 15
The hash addresses and the Q values

key	beer	brandy	champagne	cider	cognac	liqueur
h(key)	11	7	9	1	6	3
Q(key)	0	0	0	0	0	1

key	porto	rum	sherry	vodka	whisky	wine
h(key)	0	5	8	2	10	4
Q(key)	1	0	0	0	–	5

Table 16
The g values

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
$g(i)$	0	0	0	0	0	0	0	0	0	0	0	0	1	5	0	0	0	1	0	0	0	0	0	0

In the same fashion the conflicts for the keys `porto` and `wine` are resolved. Table 15 shows the hash addresses and the Q values for the keys from S.

Having the Q values for canonical keys we compute the g values applying algorithm FINDg (see Fig. 15 and Example 4.3). The obtained g values are listed in Table 16.

The exhaustive search applied in the algorithm has a potential worst-case time complexity exponential in the number of keys to be placed in the hash table. However, if this number is small as compared to the table size, the search is carried out in a table that is mostly empty, and can be done faster (see E_search in Table 17).

Lemma 4.5. *The expected time complexity of the searching step is* $O(\sqrt{n})$.

Proof. The keys to be placed in the hash table during the search correspond to multiple edges and the edges of fundamental cycles in graph G_1. By Lemmas 4.3 and 4.1, as $n \to \infty$ the number of multiple edges goes to 0, and the number of edges on fundamental cycles $n_1 = O(\sqrt{n})$. Among the latter, $n_1 - v$ edges are canonical. They constitute patterns of size 1 and are placed independently of each other at the positions $h_0(x)$. The other v edges are noncanonical (dependent) and form patterns of size $s > 1$. Assuming that i random elements of the hash table are occupied, the probability of successfully placing in one probe a pattern of size $s > 1$ is

$$p_s = \frac{\binom{n-i}{s} s!}{n^s} = \frac{(n-i)(n-i-1)\cdots(n-i-s-1)}{n^s}.$$

Since $i < n_1$, $s \leqslant v$, $n_1 \sim \sqrt{n}$, and $v \sim \ln n$, it follows that $\lim_{n\to\infty} p_s = 1$. Thus, we can neglect the existence of noncanonical edges. Treating these edges as canonical, the search can be approximated by the task of placing n_1 keys (edges) in a hash table of size n using $h_0(x)$ as the primary hash function and linear probing to resolve collisions.

Table 17
Timings for CM's linear time algorithm

n	Map	Order	E_search	Search	Total
1000	0.058	0.238	0.000	0.023	0.318
2000	0.098	0.282	0.000	0.047	0.427
3000	0.142	0.322	0.000	0.070	0.534
4000	0.183	0.366	0.000	0.091	0.641
5000	0.224	0.409	0.000	0.115	0.747
6000	0.264	0.452	0.000	0.140	0.855
7000	0.305	0.495	0.000	0.160	0.960
8000	0.347	0.534	0.000	0.186	1.067
9000	0.392	0.577	0.000	0.209	1.178
10 000	0.439	0.619	0.000	0.232	1.290
12 000	0.539	0.722	0.000	0.288	1.549
16 000	0.697	0.867	0.000	0.374	1.938
20 000	0.870	1.027	0.000	0.470	2.367
24 000	1.043	1.187	0.000	0.567	2.797
28 000	1.365	1.374	0.001	0.666	3.405
32 000	1.556	1.544	0.001	0.766	3.866
36 000	1.748	1.696	0.001	0.866	4.311
40 000	1.951	1.866	0.001	0.966	4.783
45 000	2.207	2.091	0.000	1.089	5.387
50 000	2.437	2.260	0.001	1.200	5.897
100 000	4.486	4.018	0.000	2.405	10.909

The expected number of probes to place a single key in a table containing i keys is [66]

$$C_i' \approx \frac{1}{2}\left[1 + \left(\frac{1}{1-\alpha_i}\right)^2\right]$$

where $\alpha_i = i/n$ is the load factor of the hash table which varies between 0 and $(n_1 - 1)/n = O(1/\sqrt{n})$. The total number of probes to place all n_1 keys is then

$$\sum_{i=0}^{n_1-1} C_i' \leqslant \frac{n_1}{2}\left[1 + \left(\frac{1}{1-1/\sqrt{n}}\right)^2\right] \sim n_1 = O(\sqrt{n}).$$

Thus, as $n \to \infty$ each key is placed in the hash table in constant time, and the search is done in expected time $O(\sqrt{n})$. □

Practical experiments have shown that n_1 is bounded by $\frac{1}{4}\sqrt{n}$ (Fig. 22), which is half the theoretical result (Lemma 4.1). Furthermore, $v < \frac{1}{4}\ln m$, and in fact for $n = 1000..100\,000$, v can be treated as a small constant (Fig. 23). For hash tables of size $n = 100\,000$ the average number of keys to be placed in the table was 53.66, so the tables were almost empty indeed when the search for these keys was executed. The column E_search of Table 17 contains the execution time of the exhaustive search. We can see that the search time is negligible compared to the total execution time of the algorithm.

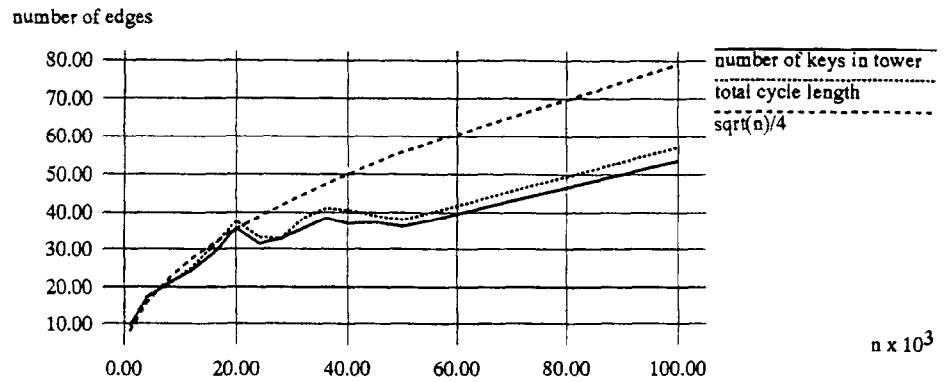

Fig. 22. The average number of keys in the tower and the average total length of fundamental cycles, n_1.

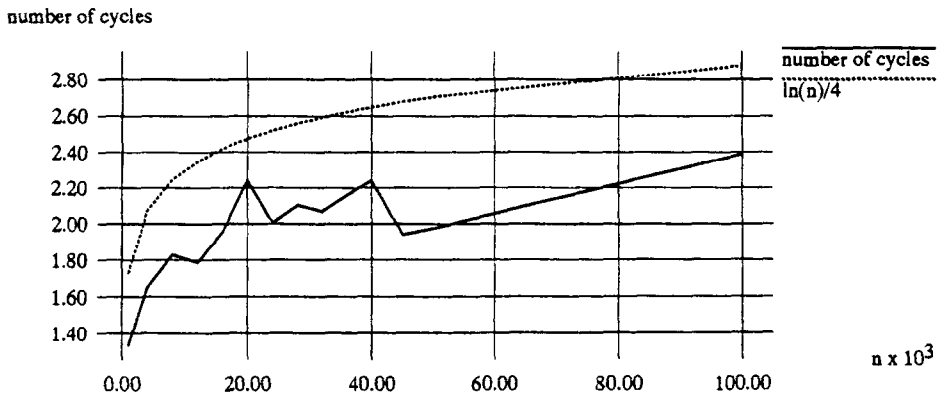

Fig. 23. The average number of fundamental cycles, v.

Once the hash values for n_1 words are determined, the hash values for free keys are computed (see Fig. 20). It is accomplished by a single pass through the hash table and placing successive keys in empty locations. Clearly, it takes $O(n)$ time.

Theorem 4.1. *Given $|g| = 2r = 2n$, the expected time complexity of the algorithm to find a minimal perfect hash function is $O(n)$.*

Proof. The theorem follows from Lemmas 4.4 and 4.5, and the observation that the mapping step and the FINDg procedure run in $O(n)$ expected time. □

The results of timing experiments are summarized in Table 17. All experiments were carried out on a Sun SPARC station 2 running under the SunOS^{tm} operating system. Each entry in the table is an average over 200 trials. Keys were chosen from the 24 692 key UNIX dictionary. For $n > 24\,692$, sets of keys consisting of random letters were generated. The values of n, Map, Order, E_search, Search and Total are the number

of keys, time for the mapping step, time for the ordering step, time for the exhaustive search, time for the whole searching step, and total time for the algorithm, respectively. All times are in seconds. Since $r = n$, the space requirement for g table is $2n \log n$ bits.

The experimental results back the theoretical considerations. The total execution time of CM's algorithm grows approximately linearly with n.

4.7. *Quadratic minimal perfect hashing*

Continuing their work on minimal perfect hashing, Fox et al. [46, 49, 47] developed two other schemes. The schemes maintained the general concept of the MOS approach. However, due to some modifications, improvements in space and time requirements were achieved.

Fox, Chen, Heath and Dauod (FCHD) proposed an algorithm which uses $O(n)$ bits to represent a minimal perfect hash function [46, 49]. It involves no dependency graph so the ordering and searching steps are simplified. The algorithm finds a quadratic minimal perfect hash function. FCHD maintain that this type of a minimal perfect hash function, as a variation of quadratic hashing, has the advantage of eliminating both primary and secondary clustering in the hash table. It has the form

$$h(x) = \begin{cases} (h_0(x)g(h_1(x)) + h_2(x)g^2(h_1(x))) \bmod n & \text{if } b(h_1(x)) = 1, \\ g(h_1(x)) & \text{if } b(h_1(x)) = 0, \end{cases}$$

where h_0, h_1 and h_2 are pseudorandom functions defined as $h_0, h_2 : U \to [0, n-1]$, $h_1 : U \to [0, r-1]$, $r = cn/\log n - 1$ and c is a constant typically less than 4, b is a mark bit vector of r bits, and g is a function defined as $g : [0, r-1] \to [0, n-1]$.

The minimal perfect hash function is represented by the tables of the pseudorandom functions h_0, h_1 and h_2, which require a constant number of bits, the mark bit vector b and the array g, which need $cn/\log n$ and cn bits, respectively. Consequently, the description of the function needs $O(cn(1+1/\log n))+O(1) = O(n)+O(1)$ bits, typically $< 4n$ bits.

In the mapping step of the algorithm, the functions h_0, h_1 and h_2 are repeatedly generated, until each key is uniquely identified by a triple $(h_0(x), h_1(x), h_2(x))$. Also as a part of this step, the mark bits are set as per definition given below. In the ordering step, keys are partitioned into sets $\mathscr{X}_i = \{x : h_1(x) = i\}$, for $i \in [0, r-1]$. A mark bit $b(i)$ is set if $|\mathscr{X}_i| > 1$ and is reset if $|\mathscr{X}_i| \leqslant 1$. The searching step, by making use of the exhaustive search, determines a value $g(i)$ that fits all keys of $\mathscr{X}_i$ in empty locations of the hash table simultaneously. The sets $\mathscr{X}_i$ are processed in descending order of $|\mathscr{X}_i|$. Thus, larger sets of keys that are more difficult to place in the hash table are processed before smaller sets. When all sets of size $|\mathscr{X}_i| > 1$ are processed, assigning the remainder of sets of size $|\mathscr{X}_i| = 1$ is done by setting $g(i)$ to the addresses of the remaining empty locations in the hash table.

The experimental results showed that the quadratic minimal perfect hash function algorithm was approximately two times faster for large key sets than the skewed vertex

Table 18
Total running times in seconds for the skewed vertex distribution (SVD) and quadratic algorithms

n	SVD	Quad.
32	0.10	2.18
1024	1.33	2.95
131 072	189.28	98.93
262 144	374.09	194.43
524 288	808.34	383.57

Table 19
Timings in seconds for quadratic algorithm

n	Bits/key	Map	Order	Search	Total
32	2.28	2.15	0.02	0.02	2.19
1024	3.19	2.58	0.05	1.13	3.76
131 072	3.24	58.18	7.43	14 569.12	14 634.73
262 144	3.61	117.63	15.82	4606.75	4740.20
524 288	3.60	228.10	34.43	14 129.87	14 831.73

distribution algorithm described in Section 4.4 (see Table 18). The space requirement was 0.6 words/key. Table 19 shows the results of pushing the quadratic minimal perfect hash function algorithm to require as few bits to store the g table as possible. In both experiments only internal memory of a Sequent Symmetry machine was used.

The algorithm proved to be also suitable for very large key sets for which external memory had to be used. The generation of a minimal perfect hash function for 3 875 766 keys with a requirement of 4.58 bits/key took 33313 s (about 9 h) on a NeXT workstation with a 68030 processor and 12 MB of main memory.

4.8. *Large and small buckets*

In a further attempt to improve the methods of generating minimal perfect hash functions, Fox, Heath and Chen (FHC) increased the probability of success in the searching step by producing a nonuniform distribution of keys during the mapping step [47]. Similarly to the previous solution, the input keys are partitioned into $b = cn/\log n$ buckets, which correspond to sets $\mathcal{X}_i$ (cf. Section 4.7). However, roughly 60% of keys are forced to go into 30% of buckets, and the remaining 40% of them are mapped into 70% of buckets. (These 60–40% and 30–70% key partition proportions were established experimentally.) In effect, two types of groups of keys are obtained: small, with few keys (possibly 1 or 2), and large, with many keys. Such a partition is fine since large groups of keys can be managed if dealt with early in the searching step.

A minimal perfect hash function from a class of hash functions

$$h(x) = (h_{20}(x, d) + g(i)) \bmod n$$

is searched for, where $h_{20} : U \times \{0, 1\} \to [0, n-1]$ is a pseudorandom function, i is a bucket number for x, and d is a value of a bit allocated to each bucket. As 0 and 1 can be the value for d, h_{20} can generate two different sets of integers for keys in bucket i. This adds some more freedom to the searching step, avoiding failures by changing the d values as needed. For key x a bucket number is obtained as

$$i = \begin{cases} h_{11}(h_{10}(x)) & \text{if } h_{10}(x) \leqslant \lfloor 0.6n \rfloor - 1, \\ h_{12}(h_{10}(x)) & \text{otherwise,} \end{cases}$$

where h_{10}, h_{11} and h_{12} are pseudorandom functions defined as follows: $h_{10} : U \to [0, n-1]$, $h_{11} : [0, \lfloor 0.6n \rfloor - 1] \to [0, \lfloor 0.3b \rfloor - 1]$ and $h_{12} : [\lfloor 0.6n \rfloor, n-1] \to [\lfloor 0.3b \rfloor, b-1]$.

With each bucket i a value of $g(i)$ is associated. Once it is set, locations in the hash table are determined for all keys in the bucket. A value of $g(i)$ can be accepted if for each key in the bucket the hash function h yields an index to a location in the hash table unoccupied by any other key. The probability that this holds is inversely proportional to the number of keys already allocated in the hash table and the number of keys in the bucket. However, due to the nonuniform distribution of keys obtained in the mapping step, large groups of keys can be handled when relatively small number of locations in the hash table are occupied, and then, when the majority of locations have been assigned, small groups are processed. This substantially increases the probability of success in the searching step.

In order to speed up the process of assigning unused locations in the hash table, FHC designed an auxiliary index data structure, which allows finding these locations in constant time. In a further push to cut the hash function memory requirements, the authors used Pearson's method [78] to map a character key into an integer. This reduced the size of tables representing the pseudorandom functions, which in the previous algorithms were defined for each character of the alphabet and each position of a character within a key, to 128 bytes per pseudorandom function.

The algorithm has been used for a 3 875 766 key set to obtain minimal perfect hash functions with 2.4 up to 3.5 bits per key. The timing results are given in Table 20. A NeXT workstation with 68040 processor and 64 MB of main memory was used for conducting the experiments. Note that the algorithm was able to find a minimal perfect hash function for the 3.8 million key set in about 6 h. This result is better than the previous findings of FHC with the quadratic minimal perfect hash function algorithm.

4.9. Bibliographic remarks

Jaeschke and Osterburg have shown a number of sets of keys for which Cichelli's mapping function can behave pathologically [62]. A Monte Carlo study of Cichelli's method showed that the probability of generating a minimal perfect hash function tends to 0 as n approaches 50 [6]. Some heuristics for computing the character weights in a Cichelli-style, minimal perfect hash function is given in [98].

Table 20
Timing results for FHC's algorithm

Bits/key	Map	Order	Search	Total
2.4	1890	5928	35889	43706
2.5	1886	5936	25521	33343
2.6	1887	5978	18938	26802
2.7	1887	6048	14486	22421
2.8	1897	6170	11602	19669
2.9	1894	6088	9524	17506
3.0	1905	6108	8083	16095
3.1	1894	6119	6998	15011
3.2	1885	6141	6110	14136
3.3	1884	6224	5436	13544
3.4	1886	6197	4958	13041
3.5	1886	6191	4586	12663

Several attempts were undertaken to overcome the drawbacks of Cichelli's algorithm. Cook and Oldehoeft improved Cichelli's backtrack algorithm by introducing a more complex ordering step [29]. The keys were partitioned into groups and, instead of a single key, a group of keys with the same last letter was handled at each step. Cercone et al. developed an interactive system that allows a user to specify a set of character positions within a key which should be taken into account while computing the hash value, and whether or not to include the key length into the hash function [17]. Their algorithm used a nonbacktrack intelligent enumerative search to find the character value assignments. Haggard and Karplus generalized Cichelli's hash function by considering potentially every character position in a key [55]: $h(x) = |x| + \sum_i g_i(x[i])$. Their algorithm finds a set of g_i functions, called selector functions, that uniquely identify each key. Various heuristics are then used to speed up both the letter value assignments for each selector function and the backtracking performed while finding these assignments. Another variation was suggested by Brain and Tharp [10]. Instead of using a single letter to address table g, they proposed using leading and ending letter pairs (triplets, etc.) to lower the probability of collision. Yet another modification to Cichelli's algorithm was suggested by Gori and Soda [52]. They used a method called an algebraic approach. Their hash function had the form: $h(x_i) = \Gamma^{\mathrm{T}} P_i^{\mathrm{T}} C$, for $i = 1, 2, \ldots, n$, $|S| = n$, where Γ is a vector of size l (the maximum key length) representing weights of letters, P_i is a two-dimensional matrix associated with the ith key, and C is a vector of size $|\Sigma|$ representing the letter codes.

Chapter 5. Perfect hashing based on sparse table compression

5.1. Introduction

This chapter presents the methods for perfect hashing based on sparse table compression. The first two sections of the chapter discuss the single and double displacement

methods proposed by Tarjan and Yao [97]. The tables considered are sparse but not too sparse, which means that the size of the universe of keys, u, is bounded by n^2, where n is a number of keys in the table. The former method using Ziegler's compression technique works well for tables with the so-called harmonic decay property. The latter method, though more complicated, gives good results for arbitrary tables. In Section 5.4 a modification of the single displacement method for letter-oriented keys suggested by Brian and Tharp is presented [11]. Section 5.5 describes another letter-oriented scheme given by Chang and Wu [27].

5.2. A single displacement method

Let S be a set of n keys belonging to the universe $U = \{0, 1, \ldots, u-1\}$. Assume that u is a perfect square, i.e. $u = t^2$ for some integer t, and that $n \geqslant \max(2, t)$. In looking for a perfect hash function for S we can represent this set as a $t \times t$ array A, so that location $[i, j]$ in the array corresponds to a key $x \in S$, where $i = \lfloor x/t \rfloor + 1$ and $j = x \bmod t + 1$. If x is present in S then location $[i, j]$ in A contains x, otherwise it contains zero. The single displacement method enables us to compress A into a one-dimensional hash table C with fewer locations than A, by giving a mapping from locations in A to locations in C such that no two nonzeros in A are mapped to the same location in C. The mapping is defined by a displacement $r[i]$ for each row i. Location $[i, j]$ in A is mapped into location $r[i] + j$ in C. The idea is to overlap the rows of A so that no two nonzeros appear in the same column in A, or equivalently, that no two nonzeros are placed in the same location in C.

Example 5.1. Let $S = \{1, 4, 5, 6, 8, 10, 12, 18, 20, 24\}$. This set can be represented by a 5×5 array A as in Fig. 24(a) (zeros are denoted by dots). Locations in A are numbered row by row starting from zero. In Fig. 24(b) the row displacements such that no two nonzeros have the same location in table C are shown.

To look up a key x, we compute $i = \lfloor x/t \rfloor + 1$ and $j = x \bmod t + 1$, and then check whether $C[r[i] + j]$ contains x. The access time of this perfect hash function is $O(1)$, and the storage required is t words for the row displacements plus $\max_i r[i] + t$ words for the locations in table C.

To apply this method, a set of displacements which gives table C of small size must be found. Unfortunately the problem of finding a set of displacements that minimizes the size of C, or even that comes within a factor of less than 2 of minimizing the size of C, is $\mathcal{NP}$-complete [51, p. 229].

Ziegler proposed the following *first-fit method* to obtain the row displacements [104]. Compute the row displacements for rows 1 through t one at a time. Select as the row displacement $r[i]$ for row i the smallest value such that no nonzero in row i is mapped to the same location as any nonzero in a previous row. An improved method, also suggested by Ziegler, is to apply the first-fit method to the rows sorted in decreasing order by the number of nonzeros they contain (see Fig. 24). This *first-*

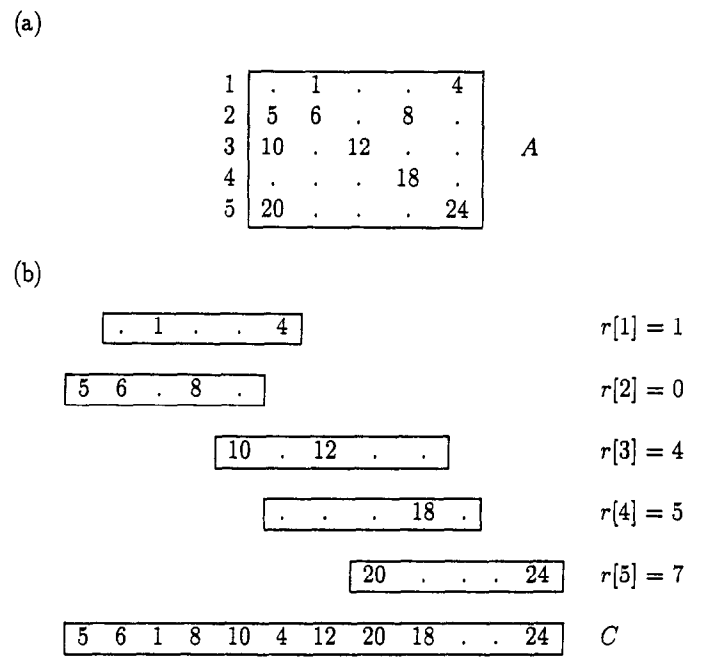

Fig. 24. (a) The representation for S; (b) The row displacements for A.

fit-decreasing method compresses A effectively if the distribution of nonzeros among the rows of A is reasonably uniform. Tarjan and Yao provided an analysis of the effectiveness of the compression [97]. Define $n(\tau)$, for $\tau \geqslant 0$, to be the total number of nonzeros in rows with more than τ nonzeros. The following theorem shows that if $n(\tau)/\tau$ decreases fast enough as τ increases, then the first-fit-decreasing method works well.

Theorem 5.1 (Tarzan and Yao [97, Theorem 1]). *Suppose the array A has the following harmonic decay property*:

$$H: \quad \textit{For any } \tau,\ n(\tau) \leqslant \frac{n}{\tau+1}.$$

Then every row displacement $r[i]$ computed for A by the first-fit-decreasing method satisfies $0 \leqslant r[i] \leqslant n$.

Proof. Consider the choice of $r[i]$ for any row i. Let $r[i]$ contain $\tau \geqslant 1$ nonzeros. By H, the number of nonzeros in previous rows is at most n/τ. Each such nonzero can block at most τ choices for $r[i]$. Altogether at most n choices can be blocked, and so $0 \leqslant r[i] \leqslant n$. □

The theorem implies that if array A has the harmonic decay property then the storage required for table C is $n+2t = O(n)$ words, or $O(n \log u)$ bits. Note that an array with

```
Initialize:
    list(i) := empty, i ∈ [1, t]; -- a list of nonzero columns in row i
    count[i] := 0, i ∈ [1, t]; -- a count of nonzero columns in row i
    bucket(l) := empty, l ∈ [0, n]; -- buckets used for sorting the rows
    entry[k] := free, k ∈ [1, n + t]; -- the table shows if a location in C is free
for each nonzero location A[i, j] do
    count[i] := count[i] + 1;
    Put j in list(i);
end for;
-- Sort the rows by their number of nonzeros;
for i := 1 to t do
    Put i in bucket(count[i]);
end for;
for l := n downto 0 do
    for each i in bucket(l) do
        r[i] := 0;
        check overlap: for each j in list(i) do
            if entry[r[i] + j] not free then
                r[i] := r[i] + 1;
                go to check overlap;
            end if;
        end for;
        for each j in list(i) do
            entry[r[i] + j] := taken;
        end for;
    end for;
end for;
```

Fig. 25. The first-fit-decreasing algorithm.

the harmonic decay property has at least half the nonzeros in rows with only a single nonzero. In addition, no row of the array can have more than $\sqrt{n}$ nonzeros.

Fig. 25 shows the first-fit-decreasing algorithm. The input is the array A with the harmonic decay property. This assumption is used when the *entry* table is allocated (only) $n+t$ words. The algorithm computes the row displacements $r[i]$ so that no two nonzeros appear in the same column in A.

It is easy to see that the execution time of the first-fit-decreasing algorithm is $O(n^2)$. The initialization step requires $O(n)$ time. Let row i, $1 \leqslant i \leqslant t$, contain τ_i nonzeros. Then the time to compute the displacement for row i is $O(n\tau_i)$, and the total time to compute the displacements is $\sum_{i=1}^{t} n\tau_i + t = O(n^2)$.

5.3. A double displacement method

In practice, it cannot be expected that all tables A will have the harmonic decay property. To overcome this difficulty we need a way to smooth out the distribution of nonzeros among the rows of A before computing the row displacements by using the first-fit-decreasing method. Towards this goal, Tarjan and Yao proposed applying

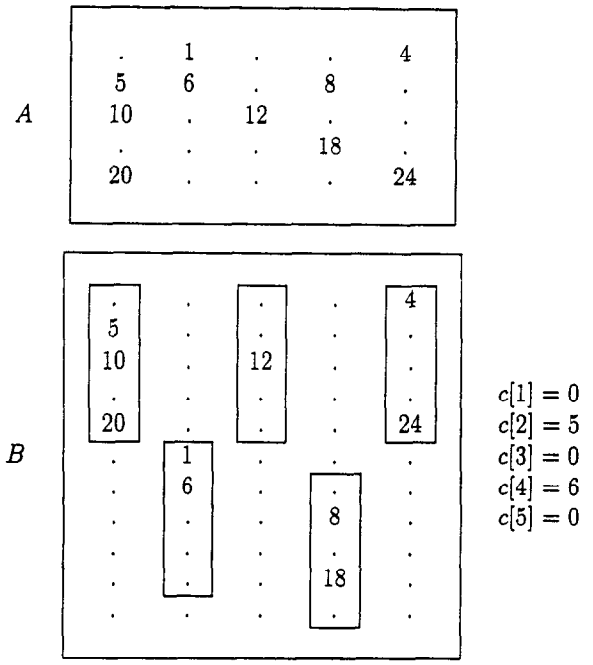

Fig. 26. The column displacements for A.

a set of column displacements $c[j]$ which map each location $[i, j]$ into a new location $[i + c[j], j]$. This transforms the array A into a new array B with an increased number of rows, $\max_j c[j] + m$, but with the same number of columns (see Fig. 26).

The transformation is expected to make the compression of the array B to be more efficient than for the original array A, despite the initial expansion caused by transforming A into B. Now we need a good set of column displacements which on the one hand gives us as few rows with many nonzeros as possible, and on the other hand keeps the array B relatively small. Tarjan and Yao proposed an *exponential decay* condition as a criterion for choosing the column displacements, defined as follows. Let B_j be the array consisting of the first j shifted columns of A. Let n_j be the total number of nonzeros in B_j. Let $n_j(i)$, for $i \geqslant 0$, be the total number of nonzeros in B_j that appear in rows of B_j containing more than i nonzeros. Then the exponential decay condition is

$$E_j: \quad \text{For any } i \geqslant 0, \ n_j(i) \leqslant \frac{n_j}{2^{i(2-n_j/n)}}.$$

The first-fit method can be employed to compute the column displacements to satisfy E_j for all j as follows. Compute the displacements for columns 1 through m one at a time. Select as the column displacement $c[j]$ for column j the smallest value such that B_j satisfies E_j (see Fig. 26).

Once the column displacements defining the transformed array B are computed, we find a set of row displacements $r[i]$ for B by using the first-fit-decreasing method. As

a result, the table C is obtained. To look up a key x, we compute $i = \lfloor x/t \rfloor + 1$ and $j = x \bmod t + 1$, and then check whether $C[r[i + c[j]] + j]$ contains x.

The theorem below gives the bound for expansion caused by using the column displacements (for a proof see [97, Theorem 2]).

Theorem 5.2. *The set of column displacements $c[j]$ computed by the first-fit method to satisfy E_j for all j is such that $0 \leqslant c[j] \leqslant 4n \log\log n + 9.5n$ for $1 \leqslant j \leqslant m$.*

With the double displacement method, the access time of the perfect hash function is O(1). The storage requirement is t words for the column displacements plus $4n \log\log n + t + 9.5n$ words for the row displacements (by Theorem 5.2) plus $n + t$ words for C (by Theorem 5.1), which amount to $4n \log\log n + 3t + 10.5n = O(n \log\log n)$ words, or $O(n \log\log n \log u)$ bits. (This storage requirement can be improved into $O(n)$ by packing several small numbers into a single word of storage, though it complicates the scheme. Furthermore, the double displacement method can be combined with the trie structure to obtain a static table storage scheme which requires $O(n)$ words of storage and allows $O(\log_n u)$ access time, for arbitrary values of n and u [97].)

It is straightforward to implement the first-fit method so that it computes column displacements to satisfy E_j for all j in $O(n^2)$ time. Since the same time is necessary for computing the row displacements, the time to construct the double displacement storage scheme is $O(n^2)$.

5.4. A letter-oriented method

Brian and Tharp [11] modified Tarjan and Yao's method to make it suitable for letter-oriented keys. In their solution the first and last letters of a key are used as indices to a two-dimensional array of integers $\mathscr{A}$. The integers in the array are the positions of keys in the key list.

Example 5.2. Consider a subset of the Pascal's reserved words, S_α = {AND, BEGIN, CHAR, CONST, ELSE, END, EOF}. In Fig. 27 the words are hashed into a 5×26 array $\mathscr{A}$ (empty locations are represented by dots). To retrieve the word AND, the first and the last letter of the word are used as the row and column indices in $\mathscr{A}$, respectively. At the location addressed by A and D is the value 1, which points to the word AND in the word list.

The perfect hash function constructed in this way has O(1) access time, but its memory usage is inefficient because many entries in $\mathscr{A}$ are empty. To solve the space inefficiency we can use the single displacement technique discussed in the previous section to compress $\mathscr{A}$ into a one-dimensional hash table $\mathscr{C}$.

Example 5.3. Applying the first-fit-decreasing algorithm to the array from the last example gives the result shown in Fig. 28. The first-fit-decreasing algorithm used here has been slightly modified, so that the negative displacements are allowed.

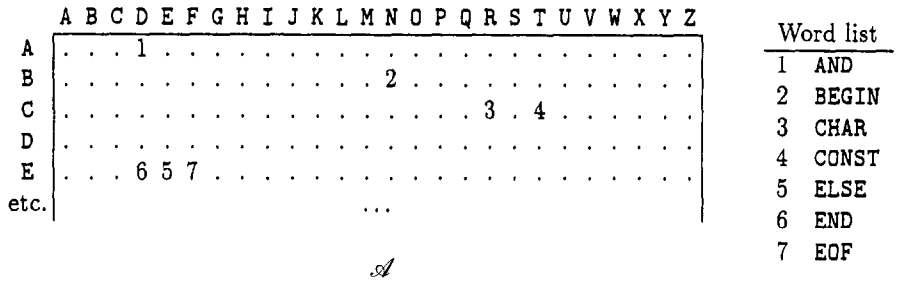

Fig. 27. A perfect hash function using two-dimensional array $\mathscr{A}$.

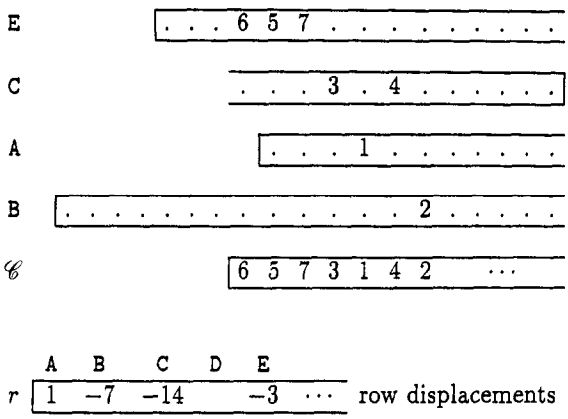

Fig. 28. Compressing array $\mathscr{A}$ into hash table $\mathscr{C}$.

Given a character string key, $x = \sigma_1\sigma_2 \ldots \sigma_l$, the perfect hash function has the form

$$h(\sigma_1, \sigma_l) = r[\sigma_1] + \text{ord}(\sigma_l),$$

where $\text{ord}(A) = 1$, $\text{ord}(B) = 2$, etc. The value of $h(\sigma_1, \sigma_l)$ is an index to the hash table $\mathscr{C}$.

Brian and Tharp conducted experiments to test the efficiency of the compressing algorithm for randomly filled sparse arrays $\mathscr{A}$ (see Table 21). In the table, array $\mathscr{A}$ was sparsely filled with the number of values indicated in the *number of entries* column. Ten runs were performed for each test set shown in the table, and the average of space was taken. The *average hash table size* column contains the size of table $\mathscr{C}$ once array $\mathscr{A}$ was packed in it. The percentages at the end of each line show the overhead caused by unused locations in $\mathscr{C}$ as compared to active entries.

The results indicate that arrays which are large and extremely sparse (e.g. a 5000×5000 array containing 1000 elements) require relatively more space when compressed than do less sparse arrays. Also, as the array fills with more and more elements (e.g. a 100×100 array containing 1000 elements) the space efficiency of the

Table 21
Performance characteristics of the compressing algorithm

Array size	Number of entries	Average hash table size
26 × 26	34	34.900 (+2.6%)
26 × 26	136	169.200 (+24.4%)
26 × 26	272	436.600 (+60.5%)
100 × 100	100	112.100 (+12.1%)
100 × 100	500	572.000 (+14.4%)
100 × 100	1000	1457.400 (+45.7%)
1000 × 1000	1000	1234.700 (+23.5%)
1000 × 1000	2000	2411.600 (+20.6%)
1000 × 1000	3000	3519.200 (+17.3%)
5000 × 5000	1000	4713.700 (+371.4%)
5000 × 5000	2000	4918.000 (+145.9%)
5000 × 5000	5000	6520.600 (+30.4%)

compressing algorithm decreases. This phenomenon can be explained via the harmonic decay property discussed in the previous section.

The perfect hashing method described above requires that the pairs formed by the first and last letters of the keys in the input set S_α are distinct. (In general, a pair can be formed from the ith and jth letter of a key.) Unfortunately, the probability that no pair collisions occur decreases rapidly as the number of keys in S_α increases. Table 22 shows the probability of pair collision versus the number of keys being placed into a 26 × 26 array $\mathscr{A}$ of 676 elements. If two random keys are to be placed into $\mathscr{A}$, the probability of them not colliding at the same location is $675/676 = 0.9985$, so the probability of collision is 0.0015. If three random keys are placed in $\mathscr{A}$, the probability of no collision decreases to $(674/676) \times (675/676) = 0.9955$, and thus the probability of collision is 0.0045. It can be seen from Table 22 that the chances of placing successfully 70 keys into a 26 × 26 array are slim. To solve the pair collision problem, Brain and Tharp suggested using the first letter pair and the last letter pair (or the first and last triplet, etc.), which increases the size of $\mathscr{A}$. Using letter pairs allows keys to be placed into a 676 × 676 array $\mathscr{A}$ of 456 976 elements, which allows perfect hashing of 200 keys with only a 0.043 probability of a pair collision, or 676 keys with a pair collision probability of 0.4. For larger key sets, letter triplets can be employed. For example, to perfectly hash 5000 keys, the first and last letter triplet can be used. However, such an approach produces an extremely large array $\mathscr{A}$, as $(26 \times 26 \times 26) \times (26 \times 26 \times 26) = 17\,576 \times 17\,576 = 308\,915\,776$. This space requirement can be cut by using the modulo function to reduce the size of $\mathscr{A}$. For example, if the first and last letter triplet's values are computed modulo 5000, then array $\mathscr{A}$ becomes 5000 × 5000. Unfortunately, this space reduction increases again the probability of getting a pair collision.

Using the modulo reduction, Brian and Tharp generated a minimal perfect hash function for 5000 keys in 283 s on a PS/2/70. Because a 5000 × 5000 array was used, 5000 words for storing the row displacements were needed.

Table 22
Probability of pair collision in a 26 × 26 array

Number of keys	Probability of collision
2	0.0015
3	0.0045
10	0.07
20	0.26
30	0.50
40	0.71
50	0.85
60	0.93
70	0.97

5.5. *Another letter-oriented method*

Chang and Wu's scheme assumes that the unique signatures of the keys from the input set can be obtained [27]. In the simplest case, a signature is a pair of the letters of a key, or 2-tuple. In general, more letters can be extracted from a key, so a signature may become a z-tuple consisting of z letters. In the first step of Chang and Wu's scheme, the extracted distinct z-tuples are mapped to a set of entries of a two-dimensional array $\mathcal{M}$ of values 0 or 1. The tuples are divided into two parts which, converted into integers, are indices into the $r \times s$ array $\mathcal{M}$.

Example 5.4. Let S_σ = {IRENA, MALGOSIA, KASIA, LYN, KATIE, LISA, BOZENA, DIANA, MARYSIA, BONNIE, EWA, KARI}. By extracting the first and third letters (2-tuple) from these names, a 26 × 26 binary array $\mathcal{M}$ is produced as shown in Fig. 29 (0's are represented by dots).

The array $\mathcal{M}$ is then compressed by using a technique devised by Chang and Wu, and subsequently it is decomposed into a set of triangular-like row vectors. The following functions are used for the compression of array $\mathcal{M}$.

CheckRow(i, j) – If both the ith and the jth row vectors of $\mathcal{M}$ have 1's at the same position then return *false*, otherwise return *true*.

CheckCol(i, j) – If both the ith and the jth column vectors of $\mathcal{M}$ have 1's at the same position then return *false*, otherwise return *true*.

MoveRow(i, j) – Move the jth row vector to the ith row vector in $\mathcal{M}$ and set all elements of the jth row vector to 0.

MoveCol(i, j) – Move the jth column vector to the ith column vector in $\mathcal{M}$ and set all elements of the jth column vector to 0.

MergeRow(i, j) – Place all the 1's of the jth row vector into the same position of the ith row vector in $\mathcal{M}$ and set all elements of the jth row vector to 0.

MergeCol(i, j) – Place all the 1's of the jth column vector into the same position of the ith column vector in $\mathcal{M}$ and set all elements of the jth column vector to 0.

The $r \times s$ array $\mathcal{M}$ is compressed into a $p \times q$ array $\mathcal{M}_1$ by merging first its rows and then its columns. The rows and columns merged cannot have 1's at the same position.

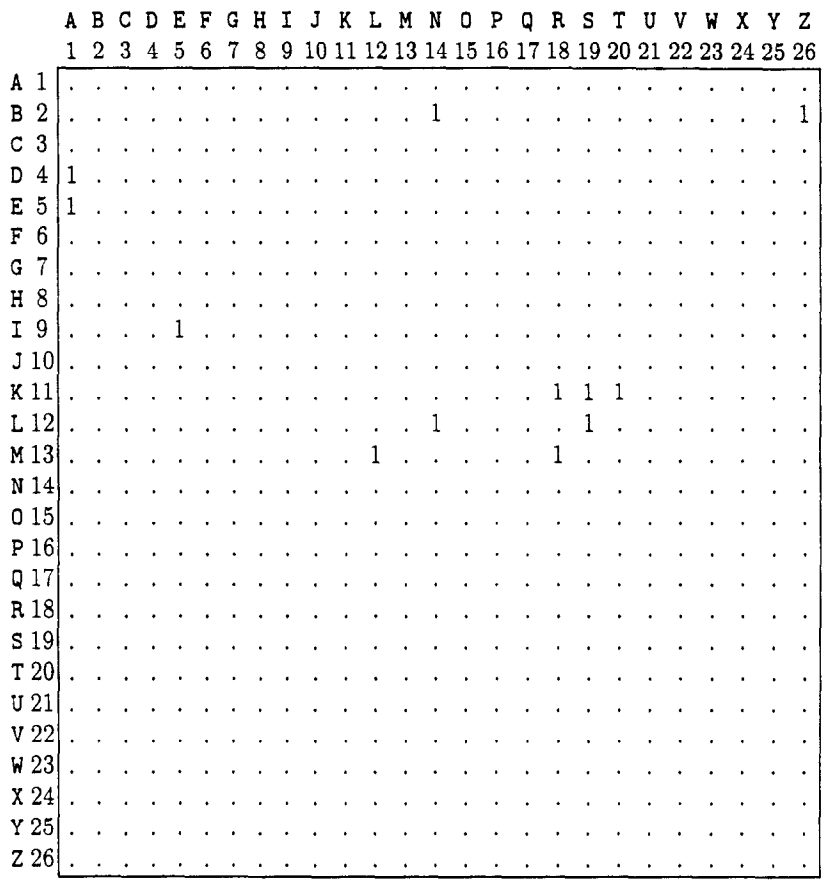

Fig. 29. The $\mathcal{M}$ array.

The algorithm for compressing, given below, computes the arrays $row[1 \ldots r]$ and $col[1 \ldots s]$, where $row[i]$ ($col[j]$) is the index of the row (column) vector into which the ith (jth) row (column) vector of $\mathcal{M}$ is merged. Consequently, the element $[i, j]$ in $\mathcal{M}$ takes the place $[i_1, j_1] = [row[i], col[j]]$ in $\mathcal{M}_1$.

```
-- Initialization of rf and cf vectors which are row and column flags,
-- respectively. They indicate if rows and columns are available for check.
∀i=1,2,...,r rf[i] := true;
∀j=1,2,...,s cf[j] := true;

-- Merge rows.
p := 1;
for i := 1 to r do
  if rf[i] = true then
    for j := i + 1 to r do
      if (rf[j] = true) and (CheckRow(i, j) = true) then
```

```
        MergeRow(i, j);
        rf[j] := false;
        row[j] := p;
      end if;
    end for;
    MoveRow(p, i);
    row[i] := p;
    p := p + 1;
  end if;
end for;
p := p − 1; -- p is the number of rows of M1.

-- Merge columns.
q := 1;
for i := 1 to s do
  if cf[i] = true then
    for j := i + 1 to s do
      if (cf[j] = true) and (CheckCol(i, j) = true) then
        MergeCol(i, j);
        cf[j] := false;
        col[j] := q;
      end if;
    end for;
    MoveCol(q, i);
    col[i] := q;
    q := q + 1;
  end if;
end for;
q := q − 1; -- q is the number of columns of M1.
```

Fig. 30 illustrates the result produced by the compressing algorithm. The more compact array $\mathcal{M}_1$ is then decomposed into two triangular-like parts, $\mathcal{U}$ and $\mathcal{L}$ (see Fig. 31), as follows. Let $[i_1, j_1]$ be an element of $\mathcal{M}_1$. If $j_1 \geqslant \lfloor qi_1/p \rfloor$ then the element belongs to $\mathcal{U}$, otherwise it belongs to $\mathcal{L}$. As a result, each original row vector of $\mathcal{M}_1$ is decomposed into two row vectors. Every row vector is associated with its index which is used for determining the displacements of row vectors such that no two 1's have the same location in a one-dimensional hash table $\mathcal{C}$. Let $R_{i_1}^{\mathcal{U}}$ be a row vector in $\mathcal{U}$, and let $R_{i_1}^{\mathcal{L}}$ be a row vector in $\mathcal{L}$. Both these vectors constitute the i_1th original vector of $\mathcal{M}_1$. We define the index of $R_{i_1}^{\mathcal{U}}$ to be $k = i_1$, and the index of $R_{i_1}^{\mathcal{L}}$ to be $k = i_1 - \lceil 2p/q \rceil + p + 1$. In effect, $\mathcal{M}_1$ is decomposed into $2p - \lceil 2p/q \rceil + 1$ row vectors in total. Once the decomposition of array $\mathcal{M}_1$ is done, the displacements $rd[k]$, for $k = 1, 2, \ldots, 2p - \lceil 2p/q \rceil + 1$, of the row vectors are computed using the first-fit-decreasing method given by Ziegler (see Section 5.2).

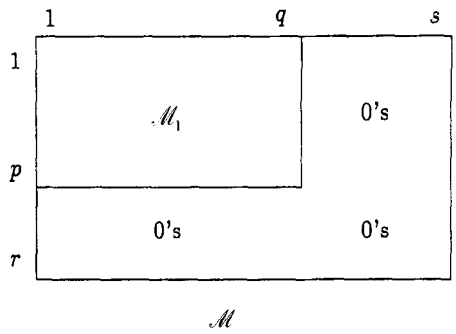

Fig. 30. The result of compressing the $\mathcal{M}$ array.

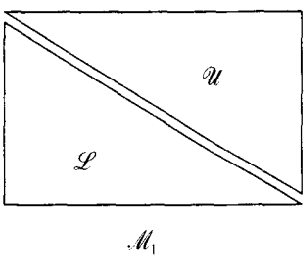

Fig. 31. Triangular-like decomposition of the $\mathcal{M}_1$ array.

Now, let the extracted z-tuple be mapped to the position $[i, j]$ in the array $\mathcal{M}$. Denoting $i_1 = row[i]$ and $j_1 = col[j]$, we obtain a perfect hash function of the form

$$h(i,j) = \begin{cases} rd[i_1] + j_1 - (\lfloor qi_1/p \rfloor - 1) \\ \quad = rd[row[i]] + col[j] - \lfloor q \times row[i]/p \rfloor + 1 \\ \quad \text{if } j_1 = col[j] \geqslant \lfloor qi_1/p \rfloor = \lfloor q \times row[i]/p \rfloor, \\ rd[i_1 - \lceil 2p/q \rceil + p + 1] + j_1 \\ \quad = rd[row[i] - \lceil 2p/q \rceil + p + 1] + col[j] \qquad \text{otherwise.} \end{cases}$$

Example 5.5. Fig. 32(a) shows the 2×7 array $\mathcal{M}_1$ ($p = 2$, $q = 7$) produced by the compressing algorithm for array $\mathcal{M}$ from the last example. The values of $row[i]$ and $col[i]$ arrays are specified in Table 23. In Fig. 32(b) the triangular-like decomposition of $\mathcal{M}_1$ is depicted. Table 24 lists the row displacements computed by Ziegler's method. Consider the key KASIA from S_σ. The extracted 2-tuple is (K, S) which corresponds to the element $[i, j] = [11, 19]$ in $\mathcal{M}$, and $[i_1, j_1] = [row[11], col[19]] = [1, 5]$ in $\mathcal{M}_1$. Thus the hash address for that key is

$$h(11, 19) = r[i_1] + j_1 - \lfloor qi_1/p \rfloor + 1 = 0 + 5 - \lfloor 7/2 \rfloor + 1 = 3.$$

The complete hash table for the keys from S_σ is shown in Table 25.

The description of the perfect hash function proposed by Chang and Wu requires storing the arrays of indices $row[1 \ldots r]$ and $col[1 \ldots s]$, and the array of displacements

(a)

1	1	1	1	1	1	1
1	1	1	1	1	0	0

(b)

index									index
3	←	1	1	1	1	1	1	1	→ 1
4	←	1	1	1	1	1	0	0	→ 2

Fig. 32. (a) The $\mathcal{M}_1$ array; (b) The decomposition of $\mathcal{M}_1$.

Table 23
The *row*[*i*] and *col*[*i*] values

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
row[*i*]	1	1	1	1	2	1	1	1	1	1	1	2	2	1	1	1	1	1	1	1	1	1	1	1	1	1
col[*i*]	1	1	1	1	2	1	1	1	1	1	1	2	1	3	1	1	1	4	5	6	1	1	1	1	1	7

Table 24
The row displacements *rd*[*k*]

k	1	2	3	4
rd[*k*]	0	0	10	5

Table 25
The hash table $\mathcal{C}$

address	1	2	3	4	5	6	7	8	9	10	11	12
key	BONNIE	KARI	KASIA	KATIE	BOZENA	EWA	MALGOSIA	LYN	MARYSIA	LISA	DIANA	IRENA

rd[1 ... 2*r*]. Let κ be the cardinality of the set of characters appearing in all extracted *z*-tuples, and let us assume that the tuples are divided into two roughly equal parts used as indices to the two-dimensional array $\mathcal{M}$. Then the number of elements (memory words) in arrays *row*, *col* and *rd* do not exceed $\kappa^{\lfloor z/2 \rfloor}$, $\kappa^{z-\lfloor z/2 \rfloor}$ and $2\kappa^{\lfloor z/2 \rfloor}$, respectively. Thus the space complexity of the perfect hash function is $O(4\kappa^{\lceil z/2 \rceil})$. Note that the magnitude of *z* highly depends on the method for extracting distinct *z*-tuples.

The time to find the perfect hash function in the proposed scheme is dominated by the time to compute the row displacements by applying Ziegler's first-fit-decreasing algorithm. This time is $O(n^2)$ (see Section 5.2), which is the time complexity of Chang and Wu's scheme.

5.6. Bibliographic remarks

The requirement of storing and accessing sparse tables arises in many applications. Examples include LR parsing [3] and sparse Gaussian elimination [95]. Various methods of storing and accessing sparse tables are discussed in [7, 65, 91, 94]. In this chapter we presented such methods based on perfect hashing. In [12] a perfect hashing technique that uses array-based tries and a simple sparse matrix packing algorithm was introduced. The technique was successfully used to form an ordered minimal perfect hash function for the entire 24481 element UNIX word list without resorting to segmentation. Chang et al. [24] proposed a refinement of the letter-oriented perfect hashing scheme given by Chang and Wu (see Section 5.5). The refinement achieves a more efficient storage utilization.

Chapter 6. Probabilistic perfect hashing

6.1. Introduction

In recent years, probabilistic algorithms have become as commonplace as conventional, deterministic algorithms. A number of efficient techniques have been developed for designing such algorithms. An excellent survey, providing an overview of five such methods, illustrated using twelve randomized algorithms and including a comprehensive bibliography can be found in [54].

In this chapter we present three algorithms and their modifications for generating minimal perfect hash functions that use a probabilistic approach. By a *probabilistic algorithm* we understand an algorithm which contains statements of the form $y :=$ *outcome of tossing a fair coin*. In general, such a coin is an s-sided coin, $s \geqslant 2$. Examples of the above statement, which often take a slightly disguised form, are [54]: "randomly select an element y from a set $\mathscr{S}$" or "randomly choose a process with which to communicate". As randomness is incorporated explicitly into the methods, alternative terms such as *randomized* or *stochastic* algorithms are often used. Correspondingly to the average running time of deterministic algorithms, probabilistic algorithms are characterized by two parameters: the expected running time and the probability that the running time will exceed the expected running time. A good example is provided by the randomized quicksort algorithm [81]. One can prove that its expected running time is no more than $(4/3)n \log n + O(n)$ on every input of size n. Moreover, with probability $1 - O(n^{-6})$ the randomized quicksort terminates in fewer than $21n \log n$ steps [81, Theorems 1.2 and 1.3]. Another important issue in the theory of probabilistic algorithms is the degree of likelihood that some event $A = A(n)$ occurs. An example of such event may be generating a random acyclic graph on n vertices. The larger n the more likely the event should be, as for small problems we may be willing to execute quite a few attempts, however for large n this may become too costly. A particularly desirable quality is if the probability of some favorable event $A = A(n)$ to occur tends to 1, polynomially fast with n. More formally, we say that an event $A = A(n)$ occurs

```
repeat
  choose at random a y from 𝒮;
until y has the property 𝒫;
```

Fig. 33. Random search algorithm.

with n-dominant probability, or equivalently, high probability, if $\Pr(A) = 1 - O(n^{-\varepsilon})$, for some $\varepsilon > 0$. An algorithm is considered to be *solid* if it solves instances of size n within stated complexity bounds with n-dominant probability.

One of the most recognized representatives of the field is Rabin's algorithm for primality testing [80]. Rabin observed that each composite number has witnesses to its compositeness. By selecting a proper type of witnesses, i.e. witnesses that constitute a constant fraction of the set of all numbers greater than one and smaller than the tested number, we can efficiently test for primality of a given number. The test simply relies on selecting a random number and checking if it is a witness to the compositeness of the number in question. If a random y is a witness with probability p, by repeating the above process k times we are either *certain* that the number is composite, or with confidence $1-(1-p)^k$ assert that the number is probably prime. Rabin proved that such a test based on Fermat's little theorem identifies a composite number with probability greater than 0.5 [80]. (This is a lower bound, which in practice is much better; see [54, Fig. 2].)

Primality testing uses a technique called *random search* [54]. Suppose that we are asked to search a vast space $\mathscr{S}$ for an element having a property $\mathscr{P}$. If $\mathscr{S}$ is too large to be searched exhaustively, and there is no known technique that allows us to move between elements of $\mathscr{S}$ in such a way that with every move we get closer to finding an element of $\mathscr{S}$ having $\mathscr{P}$, we might be at loss. For example, if 37 does not divide evenly 396 831 797 how do we find out that 11 443 does? However, if $\mathscr{S}$ fulfills the following three conditions: (i) choosing a random element from $\mathscr{S}$ can be done quickly, (ii) property $\mathscr{P}$ is easily verified, and (iii) $\mathscr{S}$ is abundant with elements having property $\mathscr{P}$, we have an easy solution. Simply pick a $y \in \mathscr{S}$ at random and verify if y has $\mathscr{P}$. Repeating the selection a number of times will result in finding a y having $\mathscr{P}$ with substantial probability (see Fig. 33). The favorable event $A = A(|y|)$ is now defined as singling out a y (of size $|y|$) having $\mathscr{P}$.

Let p be the probability that a randomly selected y from $\mathscr{S}$ has property $\mathscr{P}$, i.e. p is the probability that $A = A(|y|)$ occurs. Let X be a random variable counting the number of iterations of the loop in Fig. 33. X is said to have the geometric distribution, with

$$\Pr(X = i) = (1 - p)^{i-1}p. \tag{6.1}$$

The probability distribution function of X, $F_X(i)$, is defined as

$$F_X(i) = \sum_{j=1}^{i} \Pr(X = j) = 1 - (1 - p)^i. \tag{6.2}$$

The expected value of X, which is the expected number of iterations executed by the random search algorithm is

$$E(X) = \sum_{j=1}^{\infty} j \Pr(X = j) = \frac{1}{p}. \tag{6.3}$$

The probability that the random search algorithm executes no more than i iterations is $\Pr(X \leqslant i) = F_X(i)$. For ε, $0 < \varepsilon < 1$, the smallest i for which the algorithm in Fig. 33 terminates in i or less iterations with probability at least $1 - \varepsilon$ is

$$Q_X(\varepsilon) = \min_i \{F_X(i) \geqslant 1 - \varepsilon\} = \left\lceil \frac{\ln \varepsilon}{\ln(1-p)} \right\rceil. \tag{6.4}$$

Example 6.1. Consider the experiment in which we throw an unbiassed die, until a 6 comes up. In terms of random search we randomly select a number from space $\mathscr{S} = \{1, 2, 3, 4, 5, 6\}$, and the property $\mathscr{P}$ is defined as the number equal to 6. Here, an event is selection of a number and the favorable event is throwing a 6. While saying that each number has a property of being 1, 2, 3, etc. might seem cumbersome, we still need to verify what is the number that just came up. Often property $\mathscr{P}$ might be more involved, and the distinction between the favorable event and the property is useful for describing the complexity of verifying $\mathscr{P}$, versus the probability of the favorable event.

The probability p of obtaining a 6 is $1/6$, and thus the expected number of throws before a 6 comes up is $E(X) = 1/p = 6$, while the minimum number of throws necessary to obtain 6 with probability at least $0.99 = 1 - 1/100$ is the smallest i for which $F_X(i) \geqslant 1 - 1/100$, which is $Q_X(0.01) = 26$.

Tradeoffs are usually involved in the use of probabilistic algorithms. By using randomness we often obtain simpler, faster and easier to understand and implement algorithms. On the other hand, we may have to accept that a given algorithm gives a wrong answer with a small probability. In general, we agree to trade absolute correctness for some limited resources (e.g. time or space). This trade-off leads to a distinction between two types of probabilistic algorithms: *Monte Carlo* algorithms, which are always fast and probably correct, and *Las Vegas* algorithms, which are always correct and probably fast. One can always convert a Las Vegas algorithm into a Monte Carlo algorithm, and vice versa (cf. [54]). Which type we choose depends on how much risk we are prepared to take. (Notice that the conversion is strict. For example, a Monte Carlo algorithm of Rabin [80] cannot be converted into a Las Vegas type method for primality testing. The reason for that is that what we call probabilistic primality testing is in fact probabilistic compositeness testing. Hence we can obtain a Las Vegas algorithm for compositeness testing, but not one for primality testing.)

In the following sections we give a few examples of algorithms for generating perfect hash functions that use random search. We will see that these algorithms are both simple and highly practical.

Section 6.2 discusses the probabilistic aspects of the FKS algorithm (cf. Section 1.3) and its improvements proposed by Dietzfelbinger et al. A random graph approach introduced by Czech et al. is described in Section 6.3. The random hypergraph generalization of this approach and its modifications are discussed in Sections 6.4 and 6.5. Advanced applications of the probabilistic methods from Sections 6.3 and 6.4 are presented in Section 6.6. Section 6.7 which concludes this chapter indicates the properties of graphs which may cause some graph-based algorithms to fail.

6.2. *The FKS algorithm and its modifications reinterpreted*

The FKS method described in Section 1.3 may be viewed as an example of a probabilistic algorithm. It consists of two steps, each step using random search to achieve its goal. In the first step we are looking for a function $h : x \mapsto (ax \bmod u) \bmod n$ such that $\sum_{i=0}^{n-1} |\{x : h(x) = i\}|^2 < 5n$. In terms of random search we have a space of size $u - 1$ of congruential polynomials of degree 1, and we are interested in finding a polynomial that distributes the input set more or less evenly, among the n available addresses. Similarly to primality testing, if $(a'x \bmod u) \bmod n$, $a' \in \{1, 2, \ldots, u-1\}$, is not a suitable polynomial, we do not have an easy, deterministic way of discovering a constant a'', such that $(a''x \bmod u) \bmod n$ fulfills the above-stated condition. Fortunately for us, the space of polynomials of degree 1 is rich enough in "good quality" hash functions. As indicated by Corollary 1.3, we have more than a 50% chance that a randomly selected polynomial is good enough. In the second step, we are searching for perfect hash functions for groups of keys that were mapped into the same address. As before, we can expect a random congruential polynomial of degree 1 to be suitable, with probability more than $\frac{1}{2}$ (see Corollary 1.4).

The algorithm of Fredman et al. [50] is not solid. It executes $j \geqslant 1$ iterations with probability 2^{-j}. On the positive note, the algorithm takes no more than $O(\log n)$ steps with n-dominant probability. The question as to whether the reliability can be improved was positively answered by Dietzfelbinger et al. [35]. For $d \geqslant e$ let $\mathscr{H}^{d,e} = \{h_a : a = \langle a_e, a_{e+1}, \ldots, a_d \rangle \in U^{d-e+1}\}$ be a class of congruential polynomials of some fixed degree d, where

$$h_a : x \mapsto \left(\sum_{i=e}^{d} a_i x^i \right) \bmod u.$$

For brevity $\mathscr{H}^{d,0}$ is denoted as $\mathscr{H}^d$. Mapping into a smaller range M is effected by using a function $\zeta : U \to M$. In the simplest instance $M = \{0, 1, \ldots, m-1\}$ and $\zeta(x) = x \bmod m$. Now $\mathscr{H}_m^d$ is defined as $\{\zeta \circ g : g \in \mathscr{H}^d\}$. Let $h \in \mathscr{H}_m^d$ and define

$$S_i^h = h^{-1}(i) \cap S, \quad i \in M, \quad s_i^h = |S_i^h|, \quad \text{and} \quad B_k^h = \sum_{i \in M} \binom{s_i^h}{k}.$$

(We shall omit the superscript h whenever it can be deduced from the context.) Thus B_2 is the number of collisions caused by h when applied to some fixed set $S \subseteq U$. The following theorem can be proved (cf. [35, Theorem 4.6]).

Theorem 6.1. *Let $S \subseteq U$ be fixed and let $h \in \mathscr{H}_m^d$ be random, $d \geqslant 3$. Then $B_2 < 3|S|^2/m$ with $(|S|^2/m)$-dominant probability.*

Notice that in terms of the FKS algorithms, the primary hash function h is good for set S if $\sum_{i\in M}(s_i)^2 = O(|S|)$. The above theorem is used to show how the FKS algorithm can be made to run in linear time with high probability [35, Section 6]. In the first step the authors suggested choosing the primary hash function from $\mathscr{H}_m^3$, with $m = n = |S|$. By Theorem 6.1 the first tried function is good with probability $1-O(n^{-1})$. Therefore the first step terminates in $O(n)$ time with n-dominant probability. The second step remains unmodified, and for each bucket i a hash function $h_i \in \mathscr{H}_{2(s_i)^2}^{1,1}$ is selected independently. If we denote by t_i the number of trials needed to find a suitable h_i, then we have $\Pr(t_i = 1) \geqslant 0.5$ (by Corollary 1.4), and thus $E(t_i) \leqslant 2$ and $\mathrm{Var}(t_i) \leqslant 2$. Let T_i be the number of operations required to find a suitable function h_i. Then $T_i = O(t_i s_i)$, and hence $E(T_i) = O(s_i)$ and $\mathrm{Var}(T_i) = O((s_i)^2)$. As selecting h_i for bucket i is done independently of other choices, we have the following estimates for the total number of operations $T = \sum_{i\in M} T_i$ necessary to complete step two, $E(T) = \sum_{i\in M} O(s_i) = O(n)$ and $\mathrm{Var}(T) = \sum_{i\in M} \mathrm{Var}(T_i) = \sum_{i\in M} O((s_i)^2) = O(n)$. Now it follows from Chebyshev's inequality that $T = O(n)$ with n-dominant probability (see [35, Fact 3.4]).

The above result has an important theoretical consequence. It shows that random search is successful with high probability even when only limited randomness is available. A similar result was obtained in an earlier paper by Dietzfelbinger and Meyer auf der Heide [39, Theorem 5.2]. There, however, a more complicated and space consuming function is used. The authors proposed and analyzed the following class of universal hash functions.

Definition 6.1 (Dietzfelbinger and Meyer auf der Heide [39, Definition 4.1]). For $r, m \geqslant 1$ and $d_1, d_2 \geqslant 1$ define

$$\mathscr{R}(r, m, d_1, d_2) = \{h : U \to \{1, \ldots, m\} : h = h(f, g, a_1, \ldots, a_r)\}$$

for $f \in \mathscr{H}_r^{d_1}$, $g \in \mathscr{H}_m^{d_2}$, $a_1, \ldots, a_r \in \{1, \ldots, m\}$, where $h = h(f, g, a_1, \ldots, a_r)$ is defined as

$$h(x) = (g(x) + a_{f(x)}) \bmod m$$

for $x \in U$. Even if $h(f, g, a_1, \ldots, a_r)$ and $h(f', g', a_1', \ldots, a_r')$ are extensionally equal, we consider them different if $(f, g, a_1, \ldots, a_r) \neq (f', g', a_1', \ldots, a_r')$.

Dietzfelbinger and Meyer auf der Heide suggested choosing $m = n$ and $r = n^{1-\delta}$, for some $\delta > 0$. For such choice, and sufficiently large d_1 and d_2, many probability bounds obtained for a random member of $\mathscr{R}(r, m, d_1, d_2)$ are close to those obtained by uniform and independent mapping, thus mimicking truly random hash functions (see [39, Theorem 4.6]). Informally, the structure of $\mathscr{R}(r, m, d_1, d_2)$ can be explained as follows [35, Section 4]. Function $f \in \mathscr{H}_r^{d_1}$ splits S into r buckets $S_i^f = f^{-1}(i) \cap S$, $0 \leqslant i < r$. In the FKS scheme, all keys from bucket S_i^f would be mapped into a unique

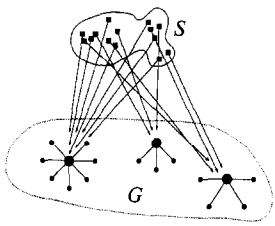

Fig. 34. A graph-theoretic interpretation of the FKS method.

space of size $2|S_i^f|^2$. Here, instead, a second hash function, $(g(x)+a_i)$, is applied, with all buckets sharing the common range $[1,m]$. Dietzfelbinger and Meyer auf der Heide proved the following theorem.

Theorem 6.2 (Dietzfelbinger and Meyer auf der Heide [39, Fact 2.4]). *Let* $0 < \delta < 1$ *be fixed, and let* $r = n^{1-\delta}$. *Let* $f \in \mathscr{H}_r^d$ *be randomly chosen. Then*

$$\Pr(|S_i^f| \leqslant 2n^\delta \text{ for all } i) \geqslant 1 - O(n^{1-\delta-\delta d/2}).$$

Thus for $r = n^{1-\delta}$ and $d_1 > 2(1-\delta)/\delta$, with high probability all buckets will have size close to the average, which is n^δ. For sufficiently large d_2 and $m = O(n)$ the second hash function g maps each $x \in S_i^f$ into a unique address, again with high probability. This ensures that keys that belong to the same S_i^f are unlikely to collide. Moreover, the independence of the offsets a_i ensures that collisions between keys belonging to different buckets are governed by the laws that apply to truly random functions.

It is worth noting that the constructions presented in [39, 35] allow us to construct a dynamic perfect hashing that uses linear space, has $O(1)$ worst case lookup time and $O(1)$ insertion time with high probability, see Ch. 7.

6.3. A random graph approach

The FKS algorithm and its modifications can be viewed as a method that, for a given input set S, constructs a mapping between S and a special acyclic graph. The characteristic feature of that graph is that each edge must have a unique vertex. Thus the primary hash function h maps each key into some primary vertex. We require that the degrees of the primary vertices are small, in the sense defined above. In the next step, the secondary function h_i, selected for the ith primary vertex, maps the group of keys that were mapped into that vertex into a number of secondary vertices. This time we insist that the degrees of secondary vertices are 1. The resulting structure is an acyclic graph G, which is a union of star-shaped trees, as shown in Fig. 34.

The specific structure of the graph in the FKS method allows direct construction of the hash table from the graph (cf. Section 1.3). On the other hand, the peculiar shape of the graph necessitates rather large memory requirements for the method. In terms of random search we are trying, in a clever way, pick a graph consisting of star-shaped trees from the space of all random graphs. As such graphs are not very common, the

price we pay is space. On the other hand, exploiting some properties of other, possibly more common, types of graphs may lead to a successful algorithm. This idea forms the basis of the next method.

Czech et al. considered the following problem [31]: For a given undirected graph $G = (V,E)$, $|E| = \mu$, $|V| = v$, find a function $g : V \to \{0,1,\ldots,\mu-1\}$ such that the function $h : E \to \{0,1,\ldots,\mu-1\}$ defined as

$$h(e) = (g(a) + g(b)) \bmod \mu$$

is a bijection, where edge $e = \{a,b\}$. In other words, we are looking for an assignment of values to vertices so that for each edge the sum of values associated with its endpoints taken modulo the number of edges is a unique integer in the range $[0,\mu-1]$.

The above problem, called the *perfect assignment problem*, is clearly a simplified version of the task that the searching steps of the algorithms presented in Ch. 4 are trying to solve. The key observations that lead to an efficient algorithm for generating minimal perfect hash functions are the following: (i) the perfect assignment problem can be solved in optimal time if G is acyclic, and (ii) acyclic graphs are common. The solution to the perfect assignment problem for an acyclic graph G takes the form of the following simple procedure. Associate with each edge a unique number $h(e) \in [0,\mu-1]$ in any order. For each connected component of G choose a vertex v. For this vertex, set $g(v)$ to 0. Traverse the graph using a depth-first search (or any other regular search on a graph), beginning with vertex v. If vertex b is reached from vertex a, and the value associated with the edge $e = \{a,b\}$ is $h(e)$, set $g(b)$ to $(h(e) - g(a)) \bmod \mu$. Apply the above method to each component of G (see Fig. 35).

The perfect assignment problem asks, in a sense, for a minimal perfect hash function for the edges of graph G. By establishing a 1–1 mapping between the input set S and the edge set of an acyclic graph G, we can effectively find a perfect hash function for S. This phase, called the *mapping step*, can be done in the following way. Choose two random, independent hash functions, $f_1 : U \to \{0,1,\ldots,v-1\}$ and $f_2 : U \to \{0,1,\ldots,v-1\}$. For each key $x \in S$ compute two "vertices", $f_1(x)$ and $f_2(x)$. (For the purposes of the algorithm we assume that if $f_1(x) = f_2(x)$ we modify $f_2(x)$ by adding a random number in the range $[1,v-1]$. This way we avoid creating self-loops.) Thus, set S has a corresponding graph G, with $V = \{0,1,\ldots,v\}$ and $E = \{\{f_1(x), f_2(x)\} : x \in S\}$. Unfortunately, the graph is not guaranteed to be acyclic. In order to guarantee acyclicity we may use random search. The algorithm repeatedly selects f_1 and f_2 until the corresponding graph, $G = G(f_1, f_2)$, is acyclic (Fig. 36).

To verify that the method given in Fig. 36 constitutes a legitimate and logical use of random search we need to check if (i) selecting an element of the search space, i.e. a random graph, can be done quickly, (ii) testing the property in question, i.e. acyclicity, can be done efficiently, and finally (iii) the space is rich in elements having the desired property. Clearly the answer for both (i) and (ii) is yes, as, assuming f_1 and f_2 are computable in constant time, generating a random graph and testing for its

```
procedure traverse(v : vertex);
begin
  visited[v] := true;
  for w ∈ neighbors(v) do
    if not visited[w] then
      g(w) := (h(e) − g(v)) mod μ;
      traverse(w);
    end if;
  end for;
end traverse;

begin
  visited[v ∈ V] := false;
  for v ∈ V do
    if not visited[v] then
      g(v) := 0;
      traverse(v);
    end if;
  end for;
end;
```

Fig. 35. The assignment step.

```
repeat
  Set V = {0, 1, ..., ν − 1} and E = ∅;
  Select at random functions f1, f2 : U → {0, 1, ..., ν − 1};
  for x ∈ S do
    E := E ∪ {f1(x), f2(x)};
  end for;
until G is acyclic;
```

Fig. 36. Mapping a set of keys onto the edge set of an acyclic graph.

acyclicity can be done in $O(\mu)$ and $O(\nu + \mu)$ time, respectively. For the solution to be useful we must have $\mu = n = |S|$ and $\nu = O(\mu)$, thus both operations take similar time to complete. The remaining question is if for $\nu = c\mu$, for some constant c, acyclic graphs dominate the space of all random graphs.

Research on random graphs began with the paper by Erdős and Rényi [43]. They used the following model to study the structural properties of random graphs. A graph with ν vertices is given. At time 0 it has no edges in it. Then the graph gains new edges at random, one at a time (hence the number of edges may be considered as time elapsed since the beginning of "life" of the graph). A "rather surprising" result discovered [43] was that for a number of fundamental structural properties $\mathscr{P}$ there exists a function $\mathscr{P}(\nu)$, tending monotonically to ∞, for $\nu \to \infty$, such that the probability that a random graph has property $\mathscr{P}$ tends to 1, if $\lim_{\nu\to\infty} \mu(\nu)/\mathscr{P}(\nu) = \infty$, and to 0, if $\lim_{\nu\to\infty} \mu(\nu)/\mathscr{P}(\nu) = 0$. Here $\mu(\nu)$ is the number of edges of the graph as a

function of v. For example, if $\mu(v) \sim cv \log v$ with $c \leqslant \frac{1}{2}$, a random graph is almost surely connected, but a sparser graph, with $\mu(v) \sim o(\log v)v$ edges, is almost certain to contain more than one connected component [44]. The most sudden change in structure of a random graph happens when the number of edges becomes greater than half the number of vertices. Before that moment a random graph has a nonzero probability of being acyclic and most of its components are trees, with the largest component being a tree with $O(\log v)$ vertices. However, as soon as the number of edges becomes larger than $\frac{1}{2}v$ a giant component emerges, with approximately $v^{2/3}$ vertices, and the graph is cyclic with probability 1 for $v \to \infty$.

For us, the crucial fact is that the probability of a random graph being acyclic is a nonzero constant, as long as $\mu < \frac{1}{2}v$. More specifically, if f_1 and f_2 are random and independent, hence each edge $\{f_1(x), f_2(x)\}$ is equally likely, and the probability of this edge being selected does not depend on the previous choices, the following theorem can be proved:

Theorem 6.3 (Havas et al. [58]). *Let G be a random graph with v vertices and μ edges obtained by choosing μ random edges with repetitions. Then if $v = c\mu$ holds with $c > 2$ the probability that G is acyclic, for $v \to \infty$, is*

$$p = e^{1/c}\sqrt{\frac{c-2}{c}}.$$

Proof. The probability that a random graph has no cycles, as $v = c\mu$ tends to infinity, has been shown to be [43]

$$\exp(1/c + 1/c^2)\sqrt{\frac{c-2}{c}}.$$

As the graphs generated in the mapping step may have multiple edges, but no loops, the probability that the graph generated in the mapping step is acyclic is equal to the probability that there are no cycles times the probability that there are no multiple edges. The jth edge is unique with probability

$$\left(\binom{v}{2} - j + 1\right) \Big/ \binom{v}{2}.$$

Thus the probability that all μ edges are unique is [77, p. 129]

$$\prod_{j=1}^{\mu} \left(\binom{v}{2} - j + 1\right) \Big/ \binom{v}{2}^{\mu} \sim \exp(-1/c^2 + o(1)).$$

Multiplying the probabilities proves the theorem. □

The presented algorithm is of Las Vegas type, as when it stops it outputs a correct perfect hash function for S, and is likely to complete its task fast. For c as small as 2.09 the probability of a random graph being acyclic exceeds $p > \frac{1}{3}$. Consequently, for such c, the expected number of iterations of the algorithm in Fig. 36

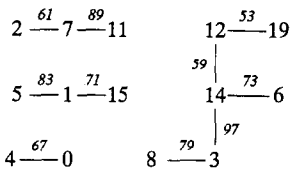

Fig. 37. An acyclic graph corresponding to the set of ten primes.

Table 26
Table g for the set of ten primes

i	0	1	2	3	4	5	6	7	8	...	11	12	...	14	15	...	19	20
$g(i)$	0	0	0	0	3	7	6	2	6	...	6	2	...	9	4	...	8	0

is $E(X)<3$. The probability that the algorithm executes more than j iterations is $F_X(j) \leqslant 1-(2/3)^j$ and with probability exceeding 0.999 the algorithm does not execute more than $Q_X(0.001) = 18$ iterations (see Section 6.1).

Example 6.2. Let S be the set of the ten prime numbers greater than 50 and smaller than 100, $S = \{53, 59, 61, 67, 71, 73, 79, 83, 89, 97\}$. Our objective is to provide a structure that allows us to verify if a number between 50 and 100 is prime in constant time. A 1–1 correspondence between S and acyclic graph G, shown in Fig. 37, is established using two mapping functions

$$f_1(x) = ((34x + 11) \bmod 101) \bmod 21,$$
$$f_2(x) = ((29x + 18) \bmod 101) \bmod 21.$$

These functions were found by repeatedly selecting at random two coefficients, a_1 and a_0, for each function from the set $\{1, 2, \ldots, 101 - 1\}$.

We have decided to store the ith prime in the $(i-1)$th location. For example, number 83, which is the 8th prime in the set, is kept at address 7, i.e. $h(\{5,1\}) = 7$. In the second step, we solve the perfect assignment problem, considering one connected component at a time. Thus we start with the single edge $\{4,0\}$, assigning 0 to $g(0)$ and 3 to $g(4)$, as 67 is the 4th prime in S. Next we move to the component consisting of two edges $\{5,1\}$ and $\{1,15\}$. First we assign 0 to $g(1)$ (here, 1 is the entry vertex v). Then we set $g(5)$ to 7 and $g(15)$ to 4. The process continues, by traversing the remaining two connected components. The complete table g is shown in Table 26. (For clarity, the entries for the vertices not belonging to any edges, which are set to 0, were left unspecified.) To check if 59 is a member of S, we compute $f_1(59) = 14$ and $f_2(59) = 12$, thus 59 has the corresponding edge $\{14, 12\}$. Next we compute $h(\{14,12\}) = (g(14) + g(12)) \bmod 10 = (9+2) \bmod 10 = 1$. Checking the second location (address 1) of the hash table confirms that 59 is a prime from S.

6.4. Random hypergraphs methods

The assignment problem can be generalized [58]. Standard graphs, where each edge connects exactly two vertices are just a member of infinite family of r-graphs, with $r = 2$. An r-graph is defined by a set of vertices $V = \{0, 1, \ldots, v-1\}$ and a set of edges $E = \{e\colon e = \{v_1, v_2, \ldots, v_r\}\}$. Thus each edge $e \in E$ contains, or using standard graph terminology, connects exactly r vertices of V. For $r = 1$ edges are equivalent to vertices, for $r = 2$ we obtain standard graphs, and for $r \geqslant 3$ we get regular hypergraphs. In such settings, the assignment problem is formulated as follows. For a given r-graph $G = (V, E)$, $|E| = \mu$, $|V| = v$, where each $e \in E$ is an r-subset of V, find a function $g : V \to \{0, 1, \ldots, \mu - 1\}$ such that the function $h : E \to \{0, 1, \ldots, \mu - 1\}$ defined as

$$h(e) = (g(v_1) + g(v_2) + \cdots + g(v_r)) \bmod \mu$$

is a bijection, where $e = \{v_1, v_2, \ldots, v_r\}$.

Similarly to 2-graphs, only acyclic r-graphs are guaranteed to have a linear time solution to the perfect assignment problem [57, 73]. However acyclicity of r-graphs, especially for $r \geqslant 3$, may be defined in many ways [42]. It turns out that the strongest definition of the cycle leads to the proper, in our context, definition of acyclicity. Each vertex in a cycle in a 2-graph has degree 2. More generally, a cycle in a 2-graph is a subgraph with all vertices of degree at least 2. Thus we may say that an r-graph is acyclic if it does not contain a subgraph with minimum degree 2. An equivalent definition of acyclicity of r-graphs is:

Definition 6.2. An r-graph is acyclic if and only if some sequence of repeated deletions of edges containing vertices of degree 1 yields a graph with no edges.

Unfortunately, the acyclicity test suggested directly by the definition for r-graphs with $r > 1$ may take $O(\mu^2)$ time. A solution that runs in optimal time has the following form [58]. Initially mark all the edges of the r-graph as not removed. Then scan all vertices, each vertex only once. If vertex v has degree 1 then remove from the r-graph the edge e such that $v \in e$. As soon as edge e is removed, check if any other of its vertices has degree 1 now. If yes, then for each such vertex remove the unique edge to which this vertex belongs. Repeat this recursively until no further deletions are possible. After all vertices have been scanned, check if the r-graph contains edges. If so, the r-graph has failed the acyclicity test. If not, the r-graph is acyclic and the reverse order to that in which they were removed is one we are looking for (a stack can be used to arrange the edges of G in an appropriate order for the second phase; Fig. 38).

Theorem 6.4. *The recursive acyclicity test given in Fig.* 38 *runs in* $O(r\mu + v)$ *time.*

Proof. Consider an r-graph $G = (V, E)$, represented by a bipartite 2-graph $H = (V_1 \cup V_2, E')$ such that $V_1 \cap V_2 = \emptyset$, where $V_1 = V$ and $V_2 = E$ and $E' = \{\{a, b\}\colon a \in V, b \in E, a \in b\}$. In this representation both edges and vertices of the r-graph become vertices

```
procedure peel_off(e : edge; w : vertex);
begin
  removed[e] := true;
  push(STACK, e);
  for v ∈ e, v ≠ w do
    Remove e from list of edges adjacent to v;
  end for;
  for v ∈ e, v ≠ w do
    if degree(v) = 1 and not removed[b : v ∈ b] then
      peel_off(b, v);
    end if;
  end for;
end peel_off;

begin
  for e ∈ E do
    removed[e] := false;
  end for;
  STACK := ∅; -- empty the stack
  for v ∈ V do
    if degree(v) = 1 and not removed[e : v ∈ e] then
      peel_off(e, v);
    end if;
  end for;
  if |STACK| = μ then -- all e ∈ E are on STACK
    return ACYCLIC;
  else
    return CYCLIC;
  end if;
end;
```

Fig. 38. A linear-time acyclicity test algorithm.

of the bipartite graph. An edge in the bipartite graph connects two vertices $u \in V_1$ and $v \in V_2$ if and only if u, a vertex in the r-graph, belongs to v, an edge, in the r-graph. During the acyclicity test each vertex in V_1 is tested at least once, by the loop which looks for vertices of degree 1. Once some edge e is deleted (which is a vertex in V_2) we test each of the other vertices in e as to whether it now has degree 1. This corresponds to traveling in graph H through some edge $\{e,u\}$, where u belongs to e in G. Regardless of the result of the test we never use that edge (of H) again. Consequently, the number of tests performed on vertices in V_1 is at most $\sum_{v \in V_1} \mathrm{dg}(v)$. As we access vertices in V_2 only once (when we delete them) the whole process takes at most $|V_1| + |V_2| + \sum_{v \in V_1} \mathrm{dg}(v) = v + \mu(r+1)$ constant time steps. □

To obtain a solution to the generalized assignment problem we proceed as follows. Associate with each edge a unique number $h(e) \in \{0, \ldots, \mu - 1\}$ in any order. Consider the edges in reverse order to the order of deletion in an acyclicity test, and assign values

```
-- STACK is a stack of edges created during a successful acyclicity test
for v ∈ V do
  assigned[v] := false;
end for;
while |STACK| > 0 do
  pop(STACK, e);
  s := 0;
  for v ∈ e do
    if not assigned[v] then
      w := v;
      g(v) := 0;
      assigned[v] := true;
    else
      s := s + g(v);
    end if;
  end for;
  g(w) := (h(e) − s) mod μ;
end while;
```

Fig. 39. The generalized assignment step.

to each as yet unassigned vertex in that edge. Each edge, at the time it is considered, will have one (or more) vertices unique to it, to which no value has yet been assigned. Let the set of unassigned vertices for edge e be $\{v_1, v_2, \ldots, v_j\}$. For edge e assign 0 to $g(v_2), \ldots, g(v_j)$ and set $g(v_1) = (h(e) - \sum_{i=2}^{r} g(v_i)) \bmod \mu$ (Fig. 39).

To prove the correctness of the method it is sufficient to show that the values of the function g are computed exactly once for each vertex, and for each edge we have at least one unassigned vertex by the time it is considered. This property is clearly fulfilled if G is acyclic and the edges of G are processed in the reverse order to that imposed by the acyclicity proof.

As we have seen, the random search step can be carried out efficiently for any r-graph with $r = O(1)$. It remains to show that there exists a constant $c_r = c(r)$ such that if $v \geqslant c_r \mu$ the space of random r-graphs is dominated by acyclic r-graphs. We have seen that this holds for $r = 2$. As before, we assume that the functions f_i are random and independent. (Hence for a fixed set S each edge $e = \{f_1(x), f_2(x), \ldots, f_r(x)\}$ for $x \in S$ is equally probable.) The following results come from [58, 79].

Theorem 6.5. *Let G be a random r-graph for $r \geqslant 1$. The threshold for the moment when the space of random r-graphs becomes dominated by acyclic r-graphs is when $\mu \leqslant v/c_r$ where*

$$c_r = \begin{cases} \Omega(\mu) & \text{for } r = 1, \\ 2 + \varepsilon, \varepsilon > 0 & \text{for } r = 2, \\ r\left(\max\limits_{y>0}\left\{\dfrac{y}{(1-e^{-y})^{r-1}}\right\}\right)^{-1} & \text{otherwise.} \end{cases}$$

Proof. We have seen the proof of the above theorem for $r = 2$. Thus it remains to show that the above holds for $r = 1$ and $r \geqslant 3$. For $r = 1$, under the assumption that f_1 is totally random, the ith key is mapped into a unique location with probability $(v - i + 1)/v$. Thus the probability that all μ keys are mapped into unique locations is $p_1 = v^{-\mu} \prod_{i=1}^{\mu}(v - i + 1) \sim \exp(-\mu^2/(2v) - \mu^3/(6v^2))$. Choosing $v = O(\mu^2)$ guarantees that the sum of the terms in the argument of the exponential function is bounded by $O(1)$, and consequently p_1 becomes a nonzero constant. In the case of limited randomness Corollaries 1.2 and 1.4 may be used.

The moment when the space of random r-graphs, with $r \geqslant 3$, becomes dominated by acyclic hypergraphs was identified by Pittel et al. [79]. The argument given in [79] is primarily devoted to a discussion about graphs, i.e. 2-graphs. Here is a summary of how to make the appropriate modification for r-graphs.

Let H be a randomly chosen r-graph. Define $V_0 = V(H)$ (the vertex set of H), and for $j \geqslant 0$ define by induction V_{j+1} to be the set of vertices of V_j that have degree at least 2 in H_j, which we define as the restriction of H to V_j. That is, we derive H_{j+1} from H_j by peeling off those vertices of degree less than 2. This process eventually terminates when $V_{j+1} = V_j = V_\infty$ say for some j. H has a subgraph of minimum degree at least 2 if and only if $V_\infty \neq \emptyset$, since V_∞ will be the vertex set of the maximal subgraph of H of minimum degree at least 2. Suppose H was selected by choosing each edge independently with probability $\alpha(r-1)!/v^{r-1}$, where α is some constant. Define

$$p_{i+1} = (1 - e^{-\alpha p_i})^{r-1}, \qquad q_{i+1} = (1 - e^{-\alpha p_i})^r.$$

Pittel et al. showed that $|V_j| \sim q_j v$ almost surely as $v \to \infty$ with j fixed [79]. Thus if $\lim_{j\to\infty} q_j = 0$ then for all $\varepsilon > 0$ we get $|V_j| < \varepsilon v$ almost surely for j sufficiently large. A separate argument (see [79]) then shows that in this case there is almost surely no subgraph of minimum degree 2.

To see how these claims imply the stated value for the threshold, consider that in the limit $p_i = p_{i+1} = p$ where

$$p = (1 - e^{-\alpha p})^{r-1}.$$

Then

$$\alpha = \frac{\alpha p}{(1 - e^{-\alpha p})^{r-1}}$$

and so

$$\alpha = \frac{y}{(1 - e^{-y})^{r-1}}$$

for some $y > 0$. Thus the minimum positive value of $y/(1 - e^{-y})^{r-1}$ is the minimum α for which $\lim_{j\to\infty} q_j \neq 0$. This gives the threshold as stated above. □

Interestingly enough $c = c(r)$ has a minimum for $r = 3$. (It remains to be seen if it is a local or the global minimum, with latter option being much more likely.) For

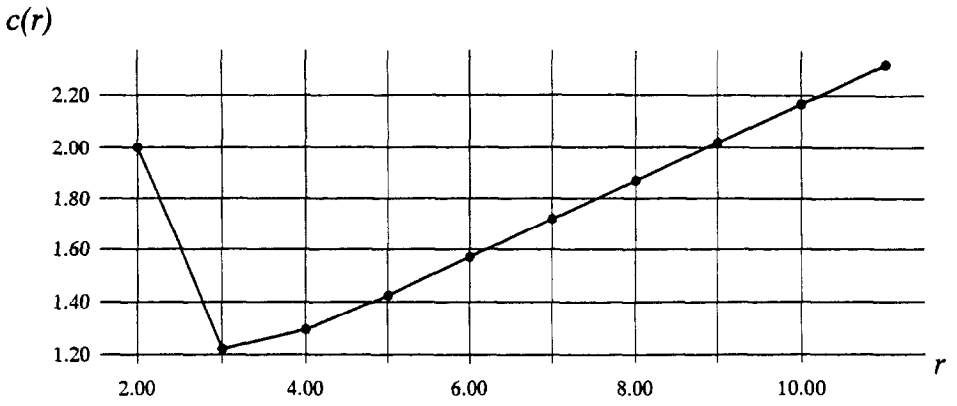

Fig. 40. Threshold points for the random r-graphs, $r \in \{2,\ldots,11\}$.

$r=2$ we have $c_2 = 2+\varepsilon$, with $c_2 = 2.09$ being a practical choice. For $r \in \{3,4,5\}$ the following approximations have been obtained by Maple^{TM} V: $c_3 \approx 1.221793133$, $c_4 \approx 1.294867415$ and $c_5 \approx 1.424947449$. A plot of threshold points for $2 \leqslant r \leqslant 11$ is presented in Fig. 40. For $r > 2$ the method becomes solid. Both theoretical [79] and practical evidence [58] indicate that the probability of a random r-graph for $r > 0$ tends to 1 for increasing v (naturally, we have $\mu = n$ and $v \geqslant c_r\mu$). For example, it is reported in [58] that for $n = \mu \geqslant 2^{16}$ the method for $r=3$ consistently executed, in all 1000 experiments, only one attempt to find acyclic 3-graphs. The average time required by the method ran on Sun SPARC station 2 to generate an order preserving minimal perfect hash function for 2^{19} character keys was less than 30 s.

Recapturing the essence of the above paragraphs, for a fixed r, the algorithm comprises two steps: mapping and assignment. In the mapping step the input set is mapped into an r-graph $G=(V,E)$, where $V=\{0,1,\ldots,v-1\}$, $E=\{\{f_1(x),f_2(x),\ldots,f_r(x)\} : x \in S\}$, and $f_i : U \to \{0,1,\ldots,v-1\}$. The number of edges μ is set to the number of keys $n=|S|$, and the number of vertices v is set to be $c_r\mu = c_r n$. The step is repeated until graph G passes the test for acyclicity. Then the assignment step is executed. Generating a minimal perfect hash function is reduced to the assignment problem as follows. As each edge $e=\{v_1,v_2,\ldots,v_r\}$, $e \in E$, corresponds uniquely to some key x (such that $f_i(x)=v_i$, $1 \leqslant i \leqslant r$) we simply set $h(e)=i-1$ if x is the ith key of S, yielding the order preserving property. Then values of function g for each $v \in V$ are computed by the assignment step (which solves the assignment problem for G). The function h is an order preserving minimal perfect hash function for S.

To complete the description of the algorithm we need to define the mapping functions f_i. Ideally the f_i functions should map any key $x \in S$ randomly into the range $[0,v-1]$. The theoretical deficiency of the above presented family is that total randomness is not efficiently computable. However, in practice, limited randomness is often as good as total randomness [15,86]. A large collection of mapping functions is necessary, so that if a given tuple fails, we can easily select an alternative set. For that purpose we use universal hashing. In universal hashing we deal with an entire class, $\mathscr{H}$, of

```
f := 0;
for i ∈ [1, |x_α|] do
  f := T[f xor σ_i];
end for;
return f;
```

Fig. 41. The hash function of Pearson.

hash functions. We require that class $\mathscr{H}$ is abundant with good quality hash functions, that can be selected fast. The general idea works as follows. Instead of fixing a hash function and using it for all possible inputs, a random member of $\mathscr{H}$ is selected before each "run". Since, according to our assumption about $\mathscr{H}$, picking a "bad" hash function is unlikely, the running time averaged over many runs is expected to be close to optimal. This design is ideal for our goals. A "run" becomes a single iteration of the mapping step, and we repeatedly select r mapping functions from some class $\mathscr{H}$ until the resulting graph is acyclic. For integer keys the functions f_i may be selected from the classes $\mathscr{H}_v^d$ for any fixed d or $\mathscr{R}(r,s,d_1,d_2)$ (defined in Section 6.2). In practice, polynomials of degree 1 or 2 are often reliable enough (see the discussion following the experimental results in this section).

Character keys however are more naturally treated as sequences of characters, so we define two more classes of universal hash functions, designed specially for character keys. (The first class has been used by others including Fox et al. [49].) We denote the length of the key x_α by $|x_\alpha|$ and its ith character by σ_i. A member of the first class, a function $f : \Sigma^* \to \{0, 1, \ldots, v-1\}$ is defined as

$$f(x_\alpha) = \left(\sum_{i=1}^{|x_\alpha|} T[i, \sigma_i] \right) \bmod v$$

where T is a table of random integers modulo v for each character and for each position of a character in a key. Selecting a member of the class is done by selecting (at random) the mapping table T. The performance of this class is satisfactory for alphabets with more than about 8 symbols. Another class was suggested by Pearson, and is especially convenient for small numbers of keys, with $n \leqslant 2^k$, for $k \leqslant 8$ [78]. For a table T of 2^k random bytes, with k selected so that the maximum code among the characters in alphabet Σ is no greater than $2^k - 1$, the hash function is computed by the program given in Fig. 41.

Again, selecting a specific function f is done by generating a random permutation of 2^k numbers between 0 and 2^{k-1}, and recording it in table T.

Both of the above classes allow us to treat character keys in the most natural way, as sequences of characters from a finite alphabet Σ. However we must remember that for any fixed maximum key length L, the total number of keys cannot exceed $\sum_{i=1}^{L} |\Sigma|^i \sim |\Sigma|^L$ keys. Thus either L cannot be treated as a constant and $L \geqslant \log_{|\Sigma|} n$ or, for a fixed L, there is an upper limit on the number of keys. In the former case, strictly speaking, processing a key character by character takes nonconstant time. Nevertheless, in practice

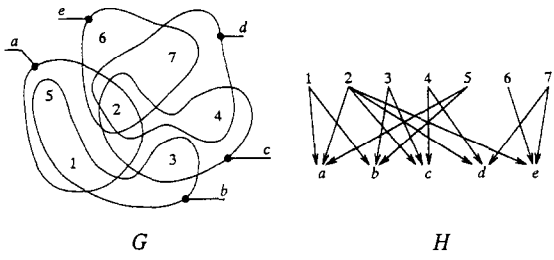

Fig. 42. An example of acyclic 3-graph.

it is often faster and more convenient to use the character by character approach than to treat a character key as a binary string. Other hashing schemes [49, 78, 85] use this approach, asserting that the maximum key length is bounded. This is an abuse of the RAM model [1, 29], however it is a purely *practical* abuse. To avoid it we would need to convert character keys into integers. However, in practice the schemes designed specially for character keys have superior performance.

Example 6.3. Fig. 42 shows a 3-graph G, together with its bipartite 2-graph representation H. Graph G is acyclic, which can be shown by applying the algorithm presented in Fig. 38.

Initially all edges, $a, b, \ldots, e$, are marked as not removed. Next the algorithm scans vertices, starting with vertex 1, until it encounters a vertex of degree one. In our case this is vertex 6. The unique edge e that contains vertex 6 is removed, and placed on stack *STACK*. Next, vertices 2 and 7 are tested if their degree is one. This is the case for vertex 7, and recursively we call procedure *peel_off* for edge d. After removing edge d from the list of adjacency of vertices 2 and 4, we discover that the degree of vertex 4 became one, and thus we call procedure *peel_off* for edge c. Removing edge c causes vertex 3 to become uniquely associated with edge b, a removal of which causes the degrees of vertices 5 and 2 to drop to one, hence enabling us to erase the last edge in G.

We carried out more than 5000 experiments to test the performance of two members of the described family of algorithms. For integer keys we randomly selected n keys from the universe $U = \{0, 1, \ldots, 100\,000\,007 - 1\}$. For each n, we executed two algorithms, for $r = 2$ and $r = 3$, 250 times on a DEC Alpha 3000/500 and calculated the average number of iterations, the average times for the mapping step, assignment step and total execution time. The mapping functions were selected from $\mathscr{H}_n^{3,1}$. The results are presented in Table 27 (times are given in seconds). As Alpha's allow 64-bit integer arithmetic and no key required more than 27 bits, all computations were carried out using integers. Our experiments confirmed the theory presented in [35], where it is indicated that polynomials of degree 1 and 2 lack reliability. In particular, the method based on 2-graphs recorded an increasing number of iterations for increasing n, reaching an average of 5.5 iterations for 100 000 keys, when the f_i's were selected from

Table 27
Performance figures of the two members of the family for integer keys

	$r = 2$, $c = 2.09$				$r = 3$, $c = 1.23$			
n	Iterations	Map	Assign	Total	Iterations	Map	Assign	Total
1000	2.688	0.029	0.003	0.032	2.212	0.055	0.001	0.056
5000	2.568	0.130	0.016	0.146	1.332	0.164	0.006	0.169
10 000	2.824	0.289	0.043	0.332	1.168	0.306	0.014	0.321
25 000	2.984	0.793	0.133	0.926	1.044	0.748	0.063	0.811
50 000	3.300	1.792	0.313	2.105	1.000	1.503	0.161	1.664
75 000	2.700	2.238	0.511	2.749	1.000	2.301	0.267	2.568
100 000	2.988	3.311	0.713	4.024	1.000	3.119	0.383	3.502
250 000	3.076	8.730	1.914	10.646	1.000	7.543	0.813	8.266
500 000	3.012	17.232	3.977	21.209	1.000	15.147	1.748	16.896

Table 28
Performance figures for the two members of the family for character keys

	$r = 2$, $c = 2.09$				$r = 3$, $c = 1.23$			
n	Iterations	Map	Assign	Total	Iterations	Map	Assign	Total
1000	2.512	0.013	0.003	0.016	2.372	0.028	0.001	0.029
5000	2.608	0.072	0.022	0.094	1.444	0.093	0.003	0.096
10 000	2.604	0.144	0.043	0.188	1.212	0.174	0.013	0.187
24 691[a]	3.012	0.383	0.141	0.524	1.084	0.395	0.039	0.434
25 000	2.840	0.408	0.150	0.558	1.048	0.420	0.044	0.464
50 000	3.048	0.905	0.339	1.244	1.016	0.907	0.111	1.018
75 000	2.896	1.322	0.541	1.863	1.000	1.396	0.189	1.585
100 000	2.752	1.720	0.748	2.468	1.000	1.901	0.270	2.170
212 563[b]	2.992	3.926	1.680	5.606	1.000	4.376	0.656	5.032
250 000	3.040	4.781	2.001	6.782	1.000	4.964	0.794	5.759
500 000	2.852	9.239	4.143	13.382	1.000	10.218	1.717	11.935

[a] The standard UNIX™ dictionary.
[b] Qiclab dictionary.

$\mathscr{H}_n^{2,1}$. For $c_2 = 2.09$ the average should be around 2.99. For practical use however, even polynomials of degree 1 seem to be a good choice. Since the effect of this on the speed of generation of a perfect hash function is hardly an issue, the gain of a much faster computable hash function seems to outweigh the few extra iterations the generation algorithm needs to execute. The degree of the polynomials seems to have much less impact on the method based on 3-graphs (in fact, r-graphs, for $r \geqslant 3$).

In a similar setting we conducted experiments for character keys. We chose to use the first class of mapping functions, based on tables defined for each character and position of the character in a key. Randomly generated strings, 3 to 15 characters long, were fed to the algorithm, and 250 repetitions for each n were executed. The averages obtained are shown in Table 28 (times are given in seconds).

Table 29
The observed worst case performance

	$r = 2$, $c = 2.09$			$r = 3$, $c = 1.23$		
Dictionary (size)	Iterations	Map	Assign	Iterations	Map	Assign
UNIX™ (24691)	15	1.650	0.183	3	0.983	0.067
RANDOM (24691)	15	1.800	0.167	2	0.717	0.067
Qiclab (212563)	19	23.732	1.883	1	5.116	0.733
RANDOM (212563)	14	16.833	1.933	1	4.750	0.783

While executing the algorithm for random keys provides some information about expected performance, the real test comes from more difficult sets. We selected two such sets, one the standard UNIX™ dictionary and the other a publicly available dictionary obtained from `qiclab.scn.rain.edu`. We restricted our attention to the words longer than 3 characters that can be distinguished by their first 15 characters. This reduced the number of words in the UNIX dictionary to 24 691, and to 212 563 in the Qiclab dictionary. We ran, as for random sets, 250 experiments. In addition to the average time and number of repetitions, shown in Table 28, we measured the longest time and the largest number of iterations required for those sets. The results are summarized in Table 29. Interestingly enough, the results for random dictionaries (entries denoted RANDOM) of the same size are similar. (For $c_2 = 2.09$ the probability that the number of iterations exceeds 19 is less than 0.0005; see Eq. (6.2).)

6.5. *Modifications*

The method presented is a special case of solving a set of $n = \mu$ linearly independent integer congruences with a larger number of unknowns. These unknowns are the entries of array g. We generate the set of congruences probabilistically in $O(n)$ time. We require that the congruences are consistent and that there exists a sequence of them such that "solving" $i - 1$ congruences by assignment of values to unknowns leaves at least one unassigned unknown in the ith congruence. We find the congruences in our mapping step and such a solution sequence in our acyclicity test.

One of the consequence of the above setup is that we do not need $h(e_1) \neq h(e_2)$ for two distinct edges e_1 and e_2, in order to be able to solve the perfect assignment problem. In fact, we do not need to put any restrictions on the values of function h. More specifically, we have the following theorem.

Theorem 6.6. *Let G be an acyclic graph, with μ edges and v vertices, and let $h : E \to M$ be a function, which assigns to each edge of G a not necessary unique integer from $M = \{0, 1, \ldots, m-1\}$. Then there exists at least one function $g : V \to M$ such that*

$$\forall e \in E, \quad e = \{v_1, v_2, \ldots, v_r\} : h(e) \equiv \left(\sum_{v_i \in e} g(v_i) \right) \bmod m.$$

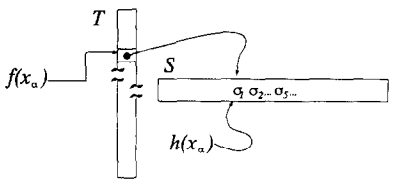

Fig. 43. Two methods of accessing a hashed string $x_\alpha = \sigma_1\sigma_2 \ldots \sigma_5 \ldots$

Hence any function h can be modeled by corresponding function g, and for acyclic r-graph such function can be found in optimal time.

This property offers some advantages. Consider a dictionary for character keys. In a standard application the hash value is an index into an array of pointers that address the beginning of the hashed strings (a typical application could be recognizing the set of keywords of a programming language). This means that on top of the space required for the hash function we need n pointers. Using Theorem 6.6 and the described algorithms we can generate a hash function that for any given character key x_α points directly to the beginning of that string, saving the space required for the pointers (Fig. 43). Other benefits include the ability to store a dictionary in any specific order we desire (sorted, grouped by verbs, nouns, adjectives, etc.), implementing a total ordering not otherwise directly computable from the data elements and an efficient algorithm for TRIE compression (for problem definition see [74, pp. 108–118]). The following section gives examples of dictionaries, where the additional flexibility of the described hashing scheme leads to increased efficiency.

The algorithm that operates on 2-graphs may be amended slightly. During the mapping step, edges of graph G are generated one by one. If some edge closes a cycle in G the whole process should be stopped. No matter what the other edges are, the graph G already has a cycle, hence it will be rejected by the acyclicity test. Flajolet et al. [45, Corollary 3] showed that, depending on the model of random graph we choose, the expected number of edges when an evolving graph obtains its first cycle varies between $\frac{1}{3}v$ (for random multigraphs) and $\frac{1}{2}(1 - \hat{p}_3)v$, where $\hat{p}_3 \approx 0.12161$. The variance is in both cases $O(v^2)$. The expected number of iterations in the mapping step (for multigraphs) executed by the algorithm is $\sqrt{c/(c-2)}$, hence the expected number of failures to generate an acyclic graph is $\sqrt{c/(c-2)} - 1$. If we could detect cycles immediately then we would take $\frac{1}{3}\beta(\sqrt{c/(c-2)} - 1)n + \beta n$ time to complete the mapping step, where β is the overhead of additional operations introduced by the fast cycle detection. Clearly, if c is close to 2 the gain can be substantial, providing β is small.

In order to immediately detect a cycle all we have to do is to check if there is a path between the vertices of the just created edge. If so, the new edge closes a cycle. Otherwise the graph remains acyclic. A convenient and fast method is offered by the solution to the set union problem. There we start with v sets, each set containing just one element. Then a sequence of intermixed operations of the following two kinds is

```
function find(x);
begin
  y := x;
  while p(p(y)) ≠ p(y) do
    p(y) := p(p(y));
    y := p(y);
  end while;
  return p(y);
end find;
```

Fig. 44. The halving variant of the *find label* operation.

executed:

- *find label*(v): Return the label of the set containing element v,
- *unite*(u,v): Combine sets containing elements u and v into a single set, whose label is the label of the old set containing element u. This operation requires that elements u and v initially be in different sets.

Now the method is clear. Initially we put the vertices into v separate sets. Then whenever a new edge is generated, say edge $\{u,v\}$, we carry out two *find label* operations, on u and v. Let the labels returned by the operations be Y and Z, respectively. If $Y=Z$ then u and v belong to the same set, hence there exists a path between u and v. That means that the new edge closes a cycle. However if $Y \neq Z$ then u and v are in separate components. Therefore the new edge does not close a cycle, and we add it to the graph. As now we have a path between vertices u and v we perform the *unite*(u,v) operation, to record that fact.

Tarjan and Van Leeuwen provided a comprehensive study of various solutions which implement the set union problem [96]. They indicated that possibly the fastest method is based on path halving. For each element v of some set V a set is created by setting $p(v) := v$, *label*$(v) := v$ and *size*$(v) := 1$. To find the label of the set that contains x we execute $y := x$, and then $y := p(y)$ until $p(y) = y$ and then return *label*(y). The path between x and y such that $y = p(y)$ is called the find path. Its length determines the complexity of the *find label* operation. To unite two sets with elements u and v we compare their sizes and then attach the smaller set to the larger one. Suppose that *size*$(v) >$ *size*(u). Then we execute $p(u) := v$, *size*$(v) :=$ *size*$(v) +$ *size*(u). Obviously, the only operation with nonconstant execution time is *find label*. In order to reduce the time required for subsequent searches, the *find label* operation compresses the find path each time it is performed. In its halving variant, during a find we make every other node along the find path (except the last and the next-to-last) point to the node two past itself (see Fig. 44).

Tarjan and Van Leeuwen [96] proved that path halving for n sets and m find/unite operations runs in $O(n + m\alpha(m+n,n))$ amortized time, where $\alpha(x,y)$ is the functional inverse of Ackermann's function, defined as $\alpha(x,y) = \min\{i \geqslant 1 : A(i, \lfloor x/y \rfloor) > \log y\}$. Tarjan and Van Leeuwen define Ackermann's function slightly different from the

"standard" definition, namely

$$A(x,y)=\begin{cases}2^y & \text{for } x=1 \text{ and } y\geqslant 1,\\ A(x-1,2) & \text{for } x\geqslant 2 \text{ and } y=2,\\ A(x-1,A(x,y-1)) & \text{for } x\geqslant 2 \text{ and } y\geqslant 2.\end{cases}$$

In the usual definition $A(1,y)=y+1$, and the explosive growth of the function does not occur quite so soon. However, this change only adds a constant to the inverse function α.

Employing a set union algorithm gives us a theoretically inferior solution. However, as pointed out in [96] "for all practical purposes, $\alpha(x,y)$ is a constant no larger than four." In fact

$$\alpha(x,x)\leqslant 4 \quad \text{if } x\leqslant 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}},$$

with fifteen 2's in the exponent. Moreover, linear time performance of set union algorithms is expected on the average [9, 67, 103]. The benefit we get is that unsuccessful attempts are interrupted as soon as possible. It has been observed in [73] that this type of solution indeed gives better performance of the algorithm, especially for c close to 2. Also path halving seems to offer the fastest method for the set union problem, as conjectured in [96].

In the discussion so far, we assumed that function h is computed as the sum of r elements of g modulo $n=\mu$. However, addition modulo n is not the only choice available. Equally we could define $h(e)$ for $e=\{v_1,v_2,\ldots,v_r\}$ as exclusive or of $g(v_i)$, $i=1,2,\ldots,r$. This choice results in two benefits. Firstly, it avoids overflows, as the result of exclusive or of two operands, each k bits long, is k bit long. Secondly, we remove the division (modulo) operation, hence the resulting function can be evaluated more rapidly.

In formal terms each element of g is a member of set M. To evaluate function h we repeatedly use a binary operation on M^2 which is a map $\diamond$ from $M\times M$ to M. The following theorem characterizes the class of legitimate binary operations that may be used in order to evaluate function h.

Theorem 6.7. *Let M be a set and $\diamond$ be a binary operation. The binary operation $\diamond$ is suitable for hash function $h: M^r \to M$ if set M together with the binary operation $\diamond$ form a group, that is the following three conditions are satisfied:*

1. $\exists o\in M\ \forall d\in M: o\diamond d=d\diamond o=d$.
2. $\forall d\in M\ \exists d^{-1}\in M: d\diamond d^{-1}=d^{-1}\diamond d=o$.
3. $\forall d_1\in M\ \forall d_2\in M\ \forall d_3\in M: (d_1\diamond d_2)\diamond d_3=d_1\diamond(d_2\diamond d_3)$.

Proof. The proof is a standard verification that the above properties are all that is required to generate and evaluate hash function h. □

It is easy to verify that both, addition modulo n and exclusive or form a group together with $M = \{0, 1, \ldots, m-1\}$, for any integer $m \geqslant 0$.

6.6. *Advanced applications*

The two examples presented in this section are intended to illustrate versatility of the described family of algorithms. They intentionally do not represent typical applications of hashing, as they use "atypical" properties of the presented family. We restrict our attention to only two members of it, for $r \in \{2,3\}$. The attractiveness of the solution for 2-graphs comes from its simplicity and the fact that it offers the more rapidly computable of the two hash functions (at the expense of greater space complexity).

As the first task we are asked to implement a constant time method of returning precomputed values of a hard to evaluate functions $\Psi : N_0 \rightarrow N_0$, at unevenly spaced points $S = \{x_1, x_2, \ldots, x_n\}$. As an example we take $\Psi(x_i)$ to be the largest prime factor of x_i, and the ith point is taken to be the ith Carmichael number. So far factoring an integer into a product of primes has eluded any attempts at polynomial time solutions, thus we may assume that the thus defined Ψ is indeed intractable. A Carmichael number ζ is a composite number, for which Fermat's congruence, $x^{\zeta-1} \equiv 1 \pmod{\zeta}$, is true. Carmichael numbers are rather rare and unevenly spaced. Consequently, evaluating function Ψ for each point x_i once, and storing the results in a dictionary seems to be the best option. A traditional approach would call for placing the obtained results in a table T, and designing a hash function ρ, such that $T[\rho(x_i)]$ stores $\Psi(x_i)$. With the above presented family we may design a hash function h, such that $h(x_i)$ is equal to the largest prime factor of the ith Carmichael number, i.e. $\forall x_i \in S : h(x_i) = \Psi(x_i)$. This saves us one lookup operation and the space necessary for the table T.

For the purpose of our example we choose the first nine Carmichael numbers: $S = \{561, 1105, 1729, 2465, 2821, 6601, 29\,341, 172\,081, 278\,545\}$, $S \subset \{0, 1, \ldots, 278\,549 - 1\}$. The largest prime factors for these numbers are: 17, 17, 19, 29, 31, 41, 61, 61 and 113, respectively. First, we decide on a class of universal hash functions from which the two mapping functions, f_1 and f_2, will be selected. We choose to use $\mathscr{H}_{19}^{1}$ (hence $v = c_2\mu$, with $c_2 = 2\frac{1}{9} > 2$). In the first step, we repeatedly select two polynomials of degree 1, at random, until the graph generated by them is acyclic. Suppose that the first selected pair of functions has the form:

$$f_1(x) = ((5667x + 101) \bmod 278\,549) \bmod 19,$$

$$f_2(x) = ((701x + 880) \bmod 278\,549) \bmod 19.$$

Thus $x_1 = 561$ is mapped into the edge $\{15, 14\}$, $x_2 = 1105$ into $\{1, 13\}$, $x_3 = 1729$ has the corresponding edge $\{8, 14\}$ and $x_4 = 2465$ is associated with the edge $\{0, 6\}$. The graph so far consists of three acyclic components. The next key, however, $x_5 = 2821$ is mapped into the same edge as x_2, $\{1, 13\}$. As this constitutes a cycle of length 2 the mapping step is interrupted and a new pair of the mapping functions is selected.

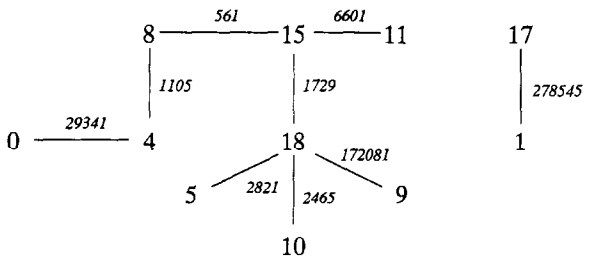

Fig. 45. An acyclic graph generated for the first nine Carmichael numbers.

This time we select:

$$f_1(x) = ((667x + 111) \bmod 278\,549) \bmod 19,$$
$$f_2(x) = ((172x + 8821) \bmod 278\,549) \bmod 19.$$

The process is restarted, and we discover that the second pair of mapping functions generates an acyclic graph, with two connected components (see Fig. 45). The second phase of the algorithm solves the perfect assignment problem. As we decided to have $h(x_i) = \Psi(x_i)$, the largest prime factor of the key associated with each edge defines the value of function h for that edge. The value of s is taken to be the largest prime factor plus one, which is $\Psi(x_9)+1 = 114$. We start with the larger component first, choosing vertex 0 as the entry point. Hence $g(0) := 0$. Next $g(4) := (h(\{0,4\}) - g(0)) \bmod s = (\Psi(x_7) - 0) \bmod 114 = 61$. From vertex 4 we move to vertex 8, and set $g(8) := (\Psi(x_2) - g(4)) \bmod 114 = 70$. Next in turn is vertex 15, and thus $g(15) := (\Psi(x_1) - g(8)) \bmod 114 = 61$. In the next step we choose to follow the edge $\{15, 11\}$, and we assign $(\Psi(x_6) - g(15)) \bmod 114 = 94$ to $g(11)$. Now we must backtrack to vertex 15 and explore the remaining path. Upon reaching vertex 18 we set $g(18) := (\Psi(x_3) - g(15)) \bmod 114 = 72$. In the next three steps we assign $(\Psi(8) - g(18)) \bmod 114 = 0$, $(\Psi(x_4) - g(15)) \bmod 114 = 82$, and $(\Psi(x_5) - g(15)) \bmod 114 = 84$, to $g(9)$, $g(10)$ and $g(5)$, respectively. Finally, we assign 0 to $g(1)$ and $\Psi(x_9) = 113$ to $g(17)$.

A disadvantage of the above technique is that table g must now accommodate numbers as large as $m-1$, while in traditional applications only $O(\log n)$ bits per entry are necessary. However quite often either $m = O(n)$ or $m-1$ fits quite comfortably into the default machine word, hence does not incur any additional memory costs. If m becomes too large we may choose to design a two way function, $f : M \to \{0, 1, \ldots, O(n^k)\}$, for some small fixed k, and work on $y = f(z)$, $z \in M$, only at the end regaining the original value of z by computing $f^{-1}(y)$.

Our second example comes from biology, where all species are known by two names, one commonly used in everyday language and the proper, Latin name which classifies a given species. Suppose we are asked to design a dictionary that stores some data about Australian marsupials. Each species must be accessible by its two names. For example koala should be accessible by the name `koala` and `phascolarctos cinereus`. In a traditional solution we would have to provide some intermediate structure that maps two different addresses obtained for each name into one common location. Such a

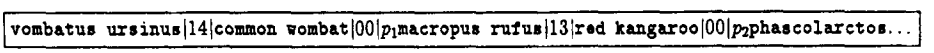

Fig. 46. The initial part of the hash table for the selected Australian marsupials.

structure is not necessary with the scheme described for the presented family. For an animal x known by its two names α and β we may simply design a hash function such that $h(\alpha) = h(\beta)$. To further illustrate the power of the method we will assume that the description of each marsupial is stored in the following format: each Latin name is followed by the corresponding common name and a 4 byte pointer to the record containing specific information associated with a given animal. Both, the Latin and common names end with a single byte that indicates the offset to the pointers. Clearly, the offset for common names is always 0, while for Latin names the offset is equal to the length of the following common name plus 1 (see Fig. 46; offset bytes are enclosed between two | symbols, while p_i denotes the ith pointer).

We choose to store the records in the following order: `common wombat` (`vombatus ursinus`), `red kangaroo` (`macropus rufus`), `koala` (`phascolarctos cinereus`), `brushtail possum` (`trichosurus vulpecula`), `sugar glider` (`petaurus breviceps`) and `desert bandicoot` (`perameles gunnii`). The above setup calls for the following values of the hash function, listed in the same order as the names: 0, 35, 67, 100, 143 and 178. While the suggested solution, to make the hash values equal for the two names of each animal, may be compelling, it leads to some inefficiencies. As the Latin name always precedes the common name, accessing a record by the common name causes this name to be first unsuccessfully compared with the Latin name, only then to find a match. Thus, instead of making the hash values equal for both names for each animal, we choose to set the hash function in such a way that the hash value of a Latin name points to the first byte of that name in the hash table and, similarly, so does the hash value for any of the six common names. For such a designed dictionary to access information related to animal named α we need to compute $h(\alpha)$ and compare α with the string starting at the address $h(\alpha)$. If the comparison is successful and the name ends with byte 0, we fetch the following 4 bytes and treat them as a pointer. Otherwise we skip the number of bytes indicated by the byte following the name and then fetch the 4 bytes of the pointer.

For the six selected animals we need the following 6 pairs of addresses: $\{0, 17\}$, $\{35, 50\}$, $\{67, 90\}$, $\{100, 122\}$, $\{143, 161\}$ and $\{178, 195\}$. One more observation required (mainly in order to save some space) is that each name can be uniquely identified by the first and third characters. Armed with all that information we execute the algorithm. (The following results were obtained by a modified version of the `perfes` program, available via anonymous ftp at ftp.cs.uq.oz.au in directory pub/TECHREPORTS/department.) The number of vertices is set to 17, which is the first prime number greater than $1.23 \times 12 = 14.76$. The size of the hash table s is set to 216, which is the total length, in bytes, of all names, offsets and pointers.

In the first step, we repeatedly generate three tables of random integers and check if the resulting 3-graph is acyclic. In our case, the fifth iteration produced the tables

T_1

a	b	c	d	g	i	k	m	p	r	s	t	u	v
	4	7	13			2	4	11	13	13	13		6
5		6	7	10	5		0		11	2	13	10	

T_2

a	b	c	d	g	i	k	m	p	r	s	t	u	v
	6	15	0			16	10	2	8	10	1		3
16		16	10	5	5		6		0	16	6	7	

T_3

a	b	c	d	g	i	k	m	p	r	s	t	u	v
	13	13	8			10	5	3	2	16	6		0
1		12	15	7	2		3		15	11	14	4	

Fig. 47. Tables generated by the mapping step of the random hypergraph algorithm.

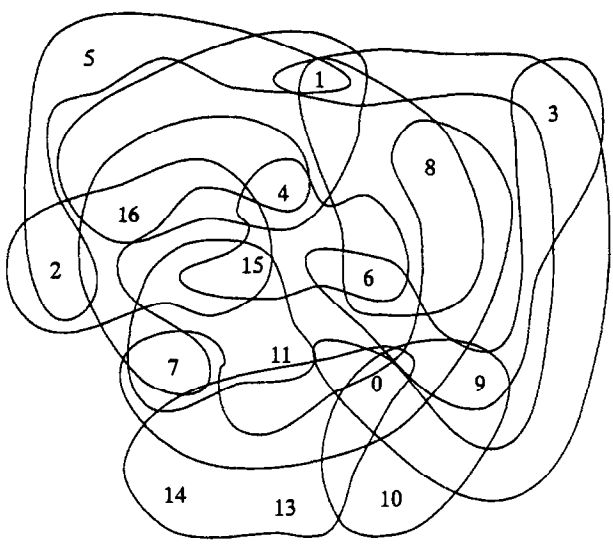

Fig. 48. Hypergraph generated by the first step of the random hypergraph method.

shown in Fig. 47. (In order to save space unused entries are either blank or not shown.) The tables from Fig. 47 together with the three mapping functions project the input set into the hypergraph shown in Fig. 48. For example, `common wombat` is mapped into the edge $\{(T_1[1,'c'] + T_1[2,'m']) \bmod 17, (T_2[1,'c'] + T_2[2,'m']) \bmod 17, (T_3[1,'c'] + T_3[2,'m']) \bmod 17\} = \{7, 4, 16\}$. Its Latin counterpart, `vombatus ursinus`, enjoys a unique bond with the edge $\{6, 9, 3\}$. To show that the hypergraph is indeed acyclic, meaning does not contain a subgraph with a minimum degree 2, we repeatedly peel off edges that contain unique vertices until no edges in the hypergraph are left. We start with edge $\{5, 2, 1\}$, with the unique vertex 5. (Equally we could start with edge

Table 30
Table g for the six Australian marsupials

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$g(i)$	155	202	52	125	122	140	0	90	114	91	5	0	0	0	183	0	143

$\{14, 13, 0\}$ or $\{10, 9, 0\}$.) By removing this edge the degree of vertex 2 decreases to 1, and consequently we can remove edge $\{15, 16, 2\}$. Unfortunately, both 15 and 16 still have degrees exceeding 1, thus we move to another edge with unique vertices, $\{14, 13, 0\}$. As vertex 0 belongs to 3 other edges, we need to find another edge with unique vertices, which is $\{10, 9, 0\}$, with vertex 10 having degree 1. This time the degree of vertex 9 drops to 1, enabling us to remove edge $\{6, 9, 3\}$. Now vertex 3 belongs exclusively to edge $\{3, 1, 0\}$, which is removed next. Next in turn is edge $\{7, 8, 0\}$ with the now unique vertex 0. It is followed by edge $\{1, 6, 8\}$, being the only remaining edge with vertex 8. 1 becomes the next unique vertex, thus we promptly remove edge $\{16, 1, 4\}$. This causes both vertices 16 and 4 to obtain degree 1, and we choose to remove firstly edge $\{7, 4, 16\}$. Next to follow is edge $\{7, 15, 11\}$, with 15 being the only vertex with higher degree than 1. Finally, the last edge $\{4, 15, 6\}$ is removed, proving that the hypergraph is acyclic.

The assignment step uses the edges in the reverse order to the order in which they were peeled off in the mapping step. Thus we start with edge $\{4, 15, 6\}$ (`sugar glider`). According to our setup, function h should evaluate to 122. Thus we assign 0 to $g(15)$ and $g(6)$ and set $g(4) := (h(e) - \sum_{i=2}^{3} g(v_i)) \bmod s = (122 - 0 - 0) \bmod 216 = 122$. Next goes edge $\{7, 15, 11\}$ (`koala`), for which $h(\{7, 15, 11\}) = 90$ is enforced by setting $g(11) := 0$ and $g(7) := (90 - 0) \bmod 216 = 90$. For the following edge, $\{7, 4, 16\}$, also known as `common wombat`, we set $g(16) := (17 - 90) \bmod 216 = 143$, and thus $h(\{7, 4, 16\}) = 17$. `Phascolarctos cinereus` is mapped into address 35, by setting $g(1) := (35 - 143 - 122) \bmod 216 = 202$. The process continues filling in the remaining entries in table g. A complete table g is shown in Table 30.

6.7. Graph-theoretic obstacles

We have presented various algorithms with different time complexities for constructing perfect or minimal perfect hash functions. The most important algorithms are based on mapping the set S into a graph, which then is processed to obtain a perfect hash function. These include Sager's mincycle algorithm (Section 4.3), its modifications due to Fox et al. (Section 4.4) and Czech and Majewski (Sections 4.5 and 4.6) and probabilistic methods due to Czech et al. (Sections 6.3 and 6.4). Even the FKS scheme can be viewed as a method of constructing and coding a special type of acyclic graph (Section 6.3).

In this section, which is based on [57], we concentrate our attention on the properties of graphs used by the algorithms. We formally justify the significance of such properties as being bipartite or being acyclic in relation to perfect and order preserving perfect

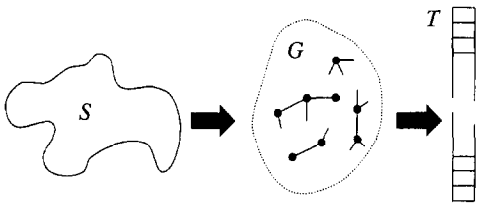

Fig. 49. A graph-based method of computing a perfect hash function.

hash functions. We identify a graph-theoretic cause for failure of some of the methods. We provide examples for which methods of Sager, of Fox et al. and of Czech and Majewski fail and do so in exponential time.

The general approach (see Fig. 49) is that first a set of keys is mapped explicitly into a graph. Then, using different methods, an attempt is made to solve the perfect assignment problem (Section 6.3): For a given undirected graph $G = (V, E)$, $|E| = \mu$, $|V| = v$ find a function $g : V \to \{0, 1, \ldots, \mu - 1\}$ such that the function $h : E \to \{0, 1, \ldots, \mu - 1\}$ defined as

$$h(e) = (g(u) + g(v)) \bmod \mu$$

is a bijection, where edge $e = \{u, v\}$.

Different graph properties are used in the effort to achieve the goal of finding a perfect hash function. However the perfect assignment problem is not always solvable. We provide a few examples of infinite classes of graphs for which we prove that no solution exists and we consider the impact this has on some of the algorithms.

A very simple example of a graph for which there is no solution to the assignment problem is a unicyclic graph, C_{4j+2}, $j > 1$. Such a graph consists of a single cycle and has $4j + 2$ edges and $4j + 2$ vertices. The edges are assigned numbers from 0 to $n - 1 = 4j + 1$ (for *minimal* perfect hashing). Hence the total sum of the values assigned to the edges is $(4j + 2)(4j + 1)/2 = (2j + 1)(4j + 1)$. This number is clearly odd. On the other hand, each vertex is adjacent to two edges. Thus the sum of values on the edges is $\sum_{\{u,v\}\in E} (g(u) + g(v)) \bmod \mu = (2 \sum_{v\in V} g(v)) \bmod \mu$ which must be an even number, a contradiction.

Another infinite family is formed by complete graphs on $8k + 5$ vertices, $k > 0$. Again, by the parity argument, we can show that there is no assignment of values $0 \ldots \mu - 1$ to edges such that a suitable function g can be found. The essence of the above examples can be stated as the following theorem.

Theorem 6.8. *For any graph G with $\mu = 4j - 2$ edges, $j > 0$, such that all the vertices of G have even degrees, there is no solution to the perfect assignment problem.*

Proof. Consider the sum of the values assigned to the edges, $\sum_{\{u,v\}\in E} (g(u) + g(v)) \bmod \mu$. If the number of edges is even, that is $\mu = 2k$, and the degrees of all the vertices of G are even, clearly the above sum must be even. On the other hand this sum must be equal to $\mu(\mu - 1)/2 = k(2k - 1)$. In order to make the assignment problem

unsolvable, the last expression needs to be odd, that is k must be odd, $k = 2j - 1$. Hence $\mu = 4j - 2$, and for such μ we always get a parity disagreement. □

An elegant test can be constructed to verify if the assignment problem can be solved for a cycle with particular values assigned to its edges. Given a cycle $C_k = (v_1, v_2, \ldots, v_k)$, let the edges of cycle C_k be assigned the values $t_1, t_2, \ldots, t_k$, so that the edge $\{v_j, v_{j+1}\}$ is assigned t_j. For vertex v_1 construct any permutation $\pi(v_1) = \langle i_1^1, i_2^1, \ldots, i_k^1 \rangle$, of the numbers from 0 to $k-1$. For vertex v_2 compute the permutation $\pi(v_2) = \langle i_1^2, i_2^2, \ldots, i_k^2 \rangle$, such that $i_j^2 = (t_1 - i_j^1) \bmod k$. For vertex v_p, $1 < p \leqslant k+1$, construct the permutation $\pi(v_p) = \langle (t_{p-1} - i_1^{p-1}) \bmod k, \ldots, (t_{p-1} - i_k^{p-1}) \bmod k \rangle$. The cycle has a solution to the assignment problem if there exists a j such that $i_j^{k+1} = i_j^1$. Tracing the whole process back we quickly discover that the cycle has a solution if there is a j such that $(t_k - t_{k-1} + \cdots + (-1)^{k-1} t_1 + (-1)^k i_j) \bmod k = i_j^1$. To show that the last condition is both sufficient and necessary assume that the ith vertex of cycle C_k is assigned the value $g(v_i)$. The values on the edges are $(g(v_i) + g(v_{i \bmod k+1})) \bmod \mu = t_i$, for $\mu > k$ and $1 \leqslant i \leqslant k$. We observe that $g(v_{i \bmod k+1}) = t_i - g(v_i) \pmod{\mu}$, $1 \leqslant i \leqslant k$. In particular we have

$$g(v_j) = t_{j-1} - g(v_{j-1}) = \sum_{p=1}^{j-1} (-1)^{j-1-p} t_p + (-1)^{j+1} g(v_1) \pmod{\mu}$$

for $2 \leqslant j \leqslant k$. Using the above to evaluate $g(v_k)$ in $(g(v_k) + g(v_1)) = t_k \pmod{\mu}$ yields

$$\sum_{p=1}^{k-1} (-1)^{k-1-p} t_p + (-1)^{k+1} g(v_1) + g(v_1) = t_k \pmod{\mu}$$

so

$$(-1)^{k+1} g(v_1) + g(v_1) = \sum_{p=1}^{k} (-1)^{k-p} t_p \pmod{\mu}.$$

Hence we have the following technical result, which is used to derive a practical corollary and effective applications.

Theorem 6.9. *For a cycle $C_k = (v_1, v_2, \ldots, v_k)$ and an integer μ there exists a function g which maps each vertex into some number in the range $[0, \mu - 1]$ such that for the edge $\{v_p, v_{p+1}\}$, $(g(v_p) + g(v_{p+1})) \bmod k = t_p$, where each t_p is a number in the range $[0, \mu - 1]$, if and only if for some integer $j \in [0, \mu - 1]$*

$$\left(\sum_{p=1}^{k} (-1)^{k-p} t_p + (-1)^k j \right) \equiv j \pmod{\mu}.$$

From the above theorem we easily deduce the following corollary.

Corollary 6.1. *Let $s = \sum_{p=1}^{k} (-1)^{k-p} t_p$, where $t_p \in \{0, \ldots, m-1\}$ for $1 \leqslant p \leqslant k$ are the values assigned to the edges of some cycle C_k. If k is even then there exists*

a solution to the perfect assignment problem if and only if $s \equiv 0 \pmod{\mu}$. *If* k *is odd then a solution exists if and only if* s *is even or both* s *and* μ *are odd.*

The above corollary provides a quick and easy test. Unfortunately, it does not give us a method for choosing t_p's so that the final condition is fulfilled. Notice that cycles of even length are in some sense more difficult to deal with than cycles of odd length. However, once the condition on the alternating sum for an even length cycle is fulfilled, it is much easier to assign values to its vertices. A similar procedure to that presented for acyclic graphs may be used (see Section 6.3). For odd length cycles, it is much easier to satisfy the condition for the alternating sum, but then there are at most two valid assignments to the vertices.

Corollary 6.1 provides us with another proof of Theorem 6.8 for the case of unicyclic graphs. First we notice that any alternating sum can be split into two parts, the positive part P and the negative part N. From any sum, to obtain a different sum we must exchange at least one element from the positive part with another element from the negative part. Call these elements ω and ξ. The initial sum is equal to $P-N$, while the modified sum will be equal to $P-N+2(\xi-\omega)$. Thus the minimum change must equal 2 or be a multiple of 2. Now consider the maximum sum for the case of a unicyclic graph C_{4j+2}. The maximum alternating sum is equal to $(6j+2)(2j+1)/2-(2j+0)(2j+1)/2 = (2j+1)^2$. This sum is odd. Hence it is not divisible by $4j+2$. Moreover, as we may modify it only by exchanging elements between the positive and the negative part, all other sums will be odd too. Consequently none of them is divisible by $4j+2$. Hence by Corollary 6.1 there is no solution for unicyclic graphs if $\mu = k$ and $k = 4j+2$. Without actually trying to do so, we have discovered the fact that an alternating sum of the integers $\{0, 1, \ldots, k-1\}$, for any permutation of them, must either always be odd (if $k = 4j+2$ or $k = 4j+3$) or always be even (if $k = 4j$ or $k = 4j+1$).

We can actually prove a stronger version of Theorem 6.9. Before we do this we need some definitions from graph theory. A *path* from v_1 to v_i is a sequence $(v_1, e_1, v_2, e_2, \ldots, e_{i-1}, v_i)$ of alternating vertices and edges such that for $1 \leqslant j < i$, e_j is incident with v_j and v_{j+1}. If $v_1 = v_i$ then a path is said to be a *cycle*. In a simple graph (a graph with no self-loops or multiple edges), a path or a cycle $(v_1, e_1, v_2, e_2, \ldots, e_{i-1}, v_i)$ can be more simply specified by the sequence of vertices $(v_1, \ldots, v_i)$. If in a path each vertex appears once, then the sequence is called a *simple* path. If each vertex appears only once except that $v_1 = v_i$ then a path is called a *simple* cycle. Because we are interested only in simple paths and cycles, these are abbreviated and the word "simple" is understood. A graph is *connected* if there is a path joining any pair of its distinct vertices. A vertex v is said to be an *articulation point* of a connected graph G if the graph obtained by deleting v and all its adjacent edges is not connected. G is said to be *biconnected* if it is connected and it contains no articulation points, that is, any two vertices of G are connected by at least two distinct paths. Two cycles, C_1 and C_2, can be added to form their sum, $C_3 = C_1 \oplus C_2$, where operation $\oplus$ is the operation of ring sum. The ring sum of two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, is the graph $((V_1 \cup V_2), ((E_1 \cup E_2) - (E_1 \cap E_2))$. The set of all

cycles is closed under the operation of ring sum, forming the *cycle space*. A *cycle basis* of G is a maximal collection of independent cycles of G, or a minimal collection of cycles on which all cycles depend. In other words, no cycle in a cycle basis can be obtained from the other cycles by adding any subset of them. There are special cycle bases of a graph which can be derived from spanning trees of a graph. Let F be a spanning forest of a graph G. Then, the set of cycles obtained by inserting each of the remaining edges of G into F is a *fundamental cycle set* of G with respect to F.

Theorem 6.10. *Let $G = (V,E)$ be a biconnected graph with $\mu = |E| > 2$ edges. Let each edge $e \in E$ be assigned a value $t_e \in [0, \mu - 1]$. Let $\mathscr{C}$ be a cycle basis of G with the additional property that each cycle in $\mathscr{C}$ goes through a selected vertex in G, say v_1. If there exists an integer $j \in [0, \mu - 1]$ such that for every cycle $C_k \in \mathscr{C}$, $C_k = (e_1, e_2, \ldots, e_k)$, $(\sum_{p=1}^{k}(-1)^{k-p}t_{e_p} + (-1)^k j) \equiv j \pmod{\mu}$, then there exists a function $g : V \to \{0, \ldots, \mu - 1\}$ such that for each edge $e = \{u, v\}$, $e \in E$, the function $(g(u) + g(v)) \bmod \mu$ is equal to t_e.*

Proof. The above result is easy to prove by induction. By Theorem 6.9, it is true if G consists of one cycle. Suppose it is true if $\mathscr{C}$ of G contains $i > 1$ cycles. Add a new cycle to $\mathscr{C}$ that has at least one edge, but at most one path in common with some cycle in $\mathscr{C}$. (In the following argument we restrict our attention to cycles with the above specified property. In doing so we lose no generality, as a cycle that has more than one disjoint path in common with some other cycle or cycles can always be modeled by a number of cycles, such that each of them shares just one path with one other cycle. The case when a new cycle has just one vertex in common with a basis cycle is trivial, therefore we omit it in our analysis.) If the sum test fails for the new cycle then G has no solution to the assignment problem. Hence suppose that the sum test is true. Graph G has no solution to the assignment problem if and only if there exists a cycle that does not belong to $\mathscr{C}$ for which the sum test fails. To show that this is impossible consider the graph in Fig. 50.

Cycles C_1 and C_2 are basis cycles, cycle C_3 can be created by adding C_1 and C_2. The alternating sum for C_1 is $s_1 = \sum_{k=0}^{p-1}(-1)^k t_{p-k}$ and the alternating sum for C_2 is $s_2 = \sum_{k=0}^{q-j-1}(-1)^k t_{q-k} + \sum_{k=0}^{i-1}(-1)^{q-j+k}t_{i-k}$. The alternating sum for C_3 is $s_3 = \sum_{k=0}^{p-i-1}(-1)^k t_{p-k} + \sum_{k=j+1}^{q}(-1)^{p-i-1+k-j}t_k$. If the length of cycle C_1 is odd and the length of cycle C_2 is even then $s_3 = s_1 + s_2$. By Corollary 6.1 and the statement of Theorem 6.10 we know that s_1 was either even or of the same parity as μ, and s_2 was a multiple of μ. Cycle C_3 must be of odd length and naturally $s_1 + s_2$ is either even or of the same parity as μ. If the lengths of both cycles are of the same parity, then $s_3 = s_1 - s_2$ and again the condition on s_3 is fulfilled. It is also easy to see that the combination even/odd ($s_3 = s_1 + s_2$, again) preserves the condition on the alternating sum of C_3. Therefore, if all cycles in a basis pass the sum test, so will any nonbasis cycle. □

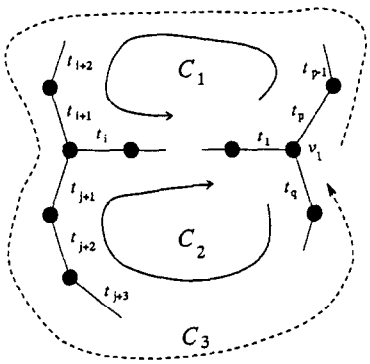

Fig. 50. The alternating sum for $C_3 = C_1 \oplus C_2$.

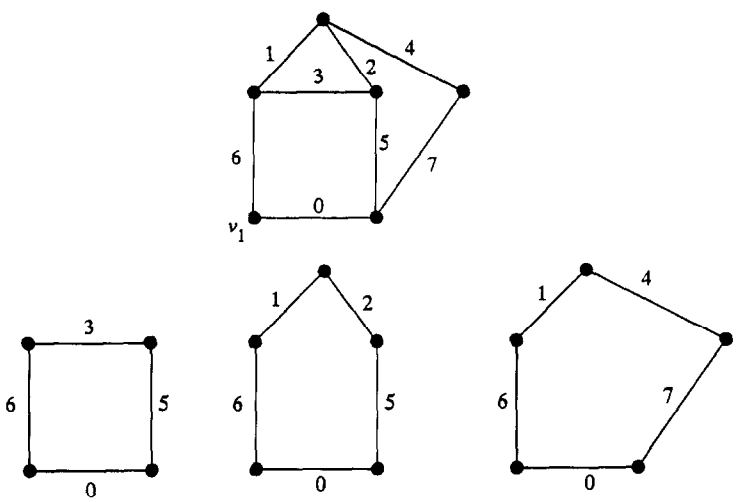

Fig. 51. A sample graph and its cycle basis.

Example 6.4. Consider the graph in Fig. 51. Its cycle basis based on v_1 consists of three cycles, shown at the bottom. To verify if for the given values on the edges it is possible to find the appropriate values for vertices we need to calculate the alternating sums for the three basis cycles. They are equal to $6-3+5-0=8$, $6-1+2-5+0=2$ and $6-1+4-7+0=2$. Using Theorem 6.10 we promptly discover that for all three cycles a suitable j is 1 or 5. Hence we conclude that there exists a suitable assignment for the graph in Fig. 51. Indeed, we may even give a method for obtaining such an assignment. We start by setting $g(v_1)$ to j. Then we perform a regular search on the graph, say depth-first search, starting with vertex v_1. Whenever we visit a new vertex u, which we have reached via the edge $e=\{v, u\}$ we set $g(u) := (t_e - g(v)) \bmod \mu$. The method is a straightforward modification of the method given for acyclic graphs

(see Section 6.3). Using this method we can easily find function g for the graph in Fig. 51.

The requirement that each cycle in $\mathscr{C}$ has to run through a specific vertex is necessary because of the odd length cycles. For each of them we not only need the permutation to agree on some position, but additionally two permutations for two different cycles have to agree on the same position. One way to ensure that is to make all cycles have at least one vertex in common, hence for all of them the starting permutation is the same. If a graph is a bipartite graph, that is all its cycles are of even length, then we may restrict our attention only to its cycles. Once for each cycle the condition on the alternating sum is satisfied, we may freely choose the values for edges that do not belong to any cycle and still be able to solve the assignment problem. This is expressed in the next theorem which we state without a proof, as it follows easily from the two theorems and the corollary given before.

Theorem 6.11. *Let $G = (V_1 \cup V_2, E)$ be a bipartite graph with $\mu = |E|$ edges. Let each edge $e \in E$, $E \subseteq V_1 \times V_2$, be assigned a unique value $t_e \in [0, \mu - 1]$. If for every biconnected component H_i of G and each cycle $C_k = (e_1, e_2, \ldots, e_k)$, $C_k \in \mathscr{C}_i$, where $\mathscr{C}_i$ is a cycle basis of H_i, $\sum_{p=1}^{k} (-1)^{k-p} t_{e_p} \equiv 0 \pmod{\mu}$, and G has no multiple edges, then there exists a solution to the assignment problem for G.*

Theorem 6.11 justifies the use of bipartite graphs by the algorithms of Sager [85], Fox et al. [49], and Czech and Majewski [32]. All of them consider cyclic edges independently of the other edges. A solution of the perfect assignment problem for all of the cyclic components can be trivially extended to acyclic parts of the dependency graph if the graph is bipartite. If the graph has cycles of odd length it might be impossible to do so. Hence the process of finding a perfect hash function is greatly simplified when bipartite graphs are used. Otherwise the whole structure needs to be considered.

In the case of order preserving hash functions, acyclicity is a sufficient condition for the function h to be order preserving. In the presence of cycles the order preserving property cannot be guaranteed. Some graphs (like K_4) resist some assignments, while allowing others. Others, C_6 might serve as a simple example, allow no solution to the perfect assignment problem. The class of graphs which are unsolvable for some assignments is much larger than the class of graphs for which there is no solution to the assignment problem. Note that inability for a graph to have arbitrary numbers assigned to its edges excludes it from the class of graphs suitable for generating an order preserving perfect hash function. The probabilistic method used in Section 6.3, by rejecting these types of graphs, can always succeed in solving the assignment problem for the graph output by the mapping step.

By Theorem 6.8 we know that if the number of edges in the dependency graph is $4j - 2$, $j > 0$, and all vertices have even degrees there is no solution to the assignment problem. This allows us to construct examples of dependency graphs for which the methods of Sager [85], Fox et al. [49], and Czech and Majewski [32] are certain to

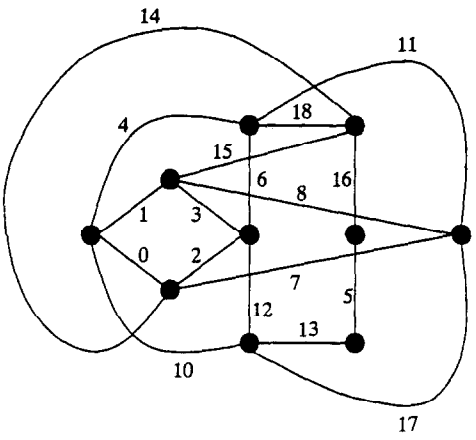

Fig. 52. An even graph.

fail. One such an example, with $\mu = 18$ edges and $v \approx 0.6m = 10$ vertices, is given in Fig. 52. We ran an exhaustive search along the lines of those specified by Sager [85] and Czech and Majewski [32] on this example. It made 16 259 364 iterations before discovering that there is no solution to the assignment problem for this graph. An upper bound on the number of iterations executed by the exhaustive search procedure in the above-mentioned algorithms is $\sum_{i=1}^{\mu-v+p(G)} \mu!/(\mu - i)!$, where $p(G)$ is the number of connected components of the dependency graph G.

We have also found an optimal perfect mapping for this graph, shown in Fig. 52. The mapping maps the 18 edges into the 19 element range [0, 18]. It is easy to verify that for any cycle of the graph in Fig. 52, the given assignment meets the conditions specified by the previous theorems.

Let us investigate the probability that a dependency graph generated in the mapping step belongs to the class of graphs characterized by Theorem 6.8. From now on we assume that the number of edges μ is even but not divisible by 4. Notice that our question can be reformulated as what is the probability that each connected component of a random graph has an Eulerian cycle. Such graphs are called even graphs.

Even graphs have been enumerated by Read [83]. Modifying his approach for bipartite graphs with $v = 2r$ vertices, r vertices in each part, we get that the number of even graphs is given by the following expression:

$$E(G_b) = 2^{-2r} \sum_{s=0}^{r} \binom{r}{s} \sum_{t=0}^{r} \binom{r}{t}$$

$$\times \sum_{j=0}^{m} (-1)^j \binom{s(r-t) + t(r-s)}{j} \binom{st + (r-s)(r-t)}{\mu - j}$$

for bipartite graphs, and

$$E(G_b') = 2^{-2r} \sum_{s=0}^{r} \binom{r}{s} \sum_{t=0}^{r} \binom{r}{t} \times \sum_{j=0}^{m} (-1)^j \binom{s(r-t)+t(r-s)+j-1}{j} \times \binom{st+(r-s)(r-t)+\mu-j-1}{\mu-j}$$

for bipartite multigraphs with no loops. The above expression allow us to compute that there are 1 479 792 175 even bipartite multigraphs with 18 edges and 10 vertices. Consequently, finding the one in Fig. 52 was a relatively simple task. However, the even bipartite multigraphs constitute only about 0.42% of the entire population of bipartite multigraphs with 18 edges and 10 vertices.

It is difficult to find a reasonable general approximation of the formula for $E(G_b')$. However, for $2r \leqslant \mu$, we observe that

$$E(G_b') \Big/ \binom{r^2+\mu-1}{\mu} \approx 2^{-2r+2}.$$

This approximation can be justified by the following simple argument. The probability that a given vertex v in a bipartite graph with r vertices in each part has degree d is

$$\Pr(\mathrm{dg}(v) = d) = \binom{\mu}{d} \left(\frac{1}{r}\right)^d \left(1 - \frac{1}{r}\right)^{\mu-d}.$$

Consequently, the probability that this degree is even is

$$\Pr(\mathrm{dg}(v) \bmod 2 = 0) = \sum_{d \geqslant 0} \binom{\mu}{2d} \left(\frac{1}{r}\right)^{2d} \left(1 - \frac{1}{r}\right)^{\mu-2d}.$$

As $(x+y)^m = \sum_{k=0}^{\mu} \binom{\mu}{k} x^k y^{\mu-k}$, selecting $x = 1/r$ and $y = (1 - 1/r)$ and $x = -1/r$ and $y = (1 - 1/r)$ and adding both equations gives

$$\left(\frac{1}{r} + \left(1 - \frac{1}{r}\right)\right)^m + \left(\frac{-1}{r} + \left(1 - \frac{1}{r}\right)\right)^m = 2 \sum_{k \geqslant 0} \binom{\mu}{2k} \left(\frac{1}{r}\right)^{2k} \left(1 - \frac{1}{r}\right)^{\mu-2k},$$

$$\frac{1 + \left(1 - \frac{2}{r}\right)^m}{2} = \sum_{k \geqslant 0} \binom{\mu}{2k} \left(\frac{1}{r}\right)^{2k} \left(1 - \frac{1}{r}\right)^{\mu-2k}.$$

Thus, as $r \to \infty$, any vertex has approximately 1 in 2 chances of having an even degree. However, once the degrees for $r-1$ vertices in each part of a bipartite graph are fixed, the degree of the last vertex in each part is determined. (Actually, "fixing" the degree of any vertex modifies the probabilities for the remaining vertices. But as long as we have "enough" edges, the above argument gives a reasonable approximation.)

As a consequence, the claimed polynomial time complexities of the algorithms of Sager [85], Fox et al. [49], and Czech and Majewski [32] are formally incorrect. The

expected time complexity of these algorithms is $O(n^n/2^n)$ rather than polynomial. Even graphs, however sparse, are still too common for these algorithms. A simple solution that avoids the above problem is provided by a probabilistic approach. If the claimed complexity of the algorithm is, say n^k, for some constant k, the algorithm should count the number of operations executed in the searching step. As soon as this number exceeds, say, n^{2k} the algorithm returns to the mapping step and, by changing some parameters, generates a new dependency graph. Notice that the trouble caused by even graphs was very unlikely to be detected by experimental observations. Fox et al. provide experimental data for $\mu \geqslant 32$. For 32 keys they would require close to a quarter of a million experiments to have a 50% chance of observing the phenomenon.

6.8. *Bibliographic remarks*

Probabilistic algorithms entered the computer science scene with publication of Rabin's seminal paper "Probabilistic Algorithms" [80], although Shallit traces back some of the concepts of randomized algorithms to the native American society of the Naskapi and the central African society of the Azande [88]. Also works of Laplace and Kelvin are pointed out as containing probabilistic approaches. Recently a great deal of attention has been paid to the field of probabilistic algorithms. Various surveys, including [13, 54, 56, 74, 81], have been published. For a more complete list see [54, Section 1].

Hashing can be seen as a direct application of the principles of randomized methods (and more specifically, input randomization). Each key from some set S receives a random address, hence the chances of $x, y \in S$ colliding are small. Interestingly enough, as we have seen in this chapter, applying randomized algorithms to generate a hash function results in deterministic addresses. While not every algorithm uses probabilistic techniques, many such concepts are present in a number of algorithms [46, 49, 47]. Another dimension is added by *universal hashing* [14–16, 35, 39, 74, Section III.2.4, 89], which is an example of *control strategy randomization* [54]. The technique called *input randomization*, a method of rearranging the input to rid it of any existing patterns [54], has found applications in distributed perfect hashing. In a distributed environment, with p independent processors, a distributed dictionary will operate in optimal time, providing each processor receives roughly the same number of keys, $O(n/p)$. The idea behind the method of [5, 38] is to use a good quality hash function to distribute the keys among the processors, and to replace it whenever performance is not satisfactory. Any deterministic distribution could not cope with some, malevolently selected inputs. By applying a good quality hash function, the input set is spread throughout the network of processors more or less evenly. Even if an adversary manages to detect a set of keys that causes deterioration in performance, by changing the distributing function, we effectively foil the adversary's plan.

Some interesting applications of minimal and order preserving perfect hash functions are discussed in [101]. For example, Witten et al. show that in certain applications, by using order preserving minimal perfect hash functions a significant space savings can

be obtained. Some benefits and disadvantages of perfect hash functions in managing large documents are outlined.

Random graphs have been studied in numerous papers. Our knowledge about certain events in the "life" of a random 2-graph is quite accurate. The moment when the number of edges approaches and then crosses the barrier of half the number of vertices is one such an example [64]. Detailed studies of random graphs can also be found in [43–45] and in the excellent books by Bollobás [8] and Palmer [77].

Chapter 7. Dynamic perfect hashing

7.1. Introduction

While discussing various method for building perfect hash functions, we restricted so far our attention to the static case, where the set of keys S is not only given beforehand, but does not change in the future. In this chapter we relax these constraints, and consider the case where a dictionary accessed by a perfect hash function in addition to the *lookup* operation, *Is* $x \in S$?, facilitates deletion, *Remove* x *from* S, and insertion, *Add* x *to* S.

Aho and Lee [2] were the first to propose a scheme that achieves constant worst case lookup time and constant amortized expected insertion and deletion times. Their algorithm, however, requires that the elements inserted are chosen uniformly at random from the universe U. This restriction was overcome by Dietzfelbinger et al. [36]. Their method achieves the same time and space bounds but inserted elements may be selected according to any distribution, without affecting the performance by more than a constant factor. (Aho and Lee's method relies on having a random sequence of operations processed by a deterministic dictionary. By randomizing the solution we free ourself from making assumptions about the input.) The only requirement is that the adversary, who is inserting elements of U into the table, has no knowledge of random choices made by the algorithm, and thus cannot use it in order to determine which item to insert into the table. It is interesting to observe that for any deterministic solution to the dictionary problem, i.e. when the adversary can predict the exact behaviour of the algorithm upon encountering any $x \in U$, an $\Omega(\log n)$ lower bound on the amortized worst case time complexity can be proved [36, Theorem 7], which matches the bounds for deterministic solutions, such as 2–3 trees [1]. Furthermore, if the worst case lookup time is to be bounded by some constant k, the amortized worst case complexity is $\Omega(kn^{1/k})$ [36, Theorem 8]. This indicates that the performance of the randomized solution suggested in [36] cannot be matched by any deterministic solution.

7.2. The FKS-based method

Dietzfelbinger et al. proposed a simple extension to the FKS scheme to the dynamic situation, wherein membership queries are processed in constant worst case time,

```
procedure RehashAll;
begin
  Let S be the set of all elements currently kept in the table;
  n := |S|;
  𝒩 := (1 + c)n;
  m := (4/3)√6 𝒩; -- m = O(𝒩) = O(n)
  repeat
    Select at random h ∈ ℋ^d_m;
    S_i := {x ∈ S : h(x) = i};
  until Σ_{i=1}^{m} 2(s_i)^2 < 8𝒩^2/m + 2𝒩;
  for i ∈ [0, m − 1] do
    -- search for a perfect hash function for subset S_i
    repeat
      Select random a_i ∈ U − {0};
    until h_i : x ↦ (a_i x mod u) mod 2(m_i)^2 is injective on S_i;
  end for;
end RehashAll;
```

Fig. 53. Reconstruction procedure for dynamic perfect hashing.

insertions and deletions are processed in constant amortized expected time and the storage used at any time is proportional to the number of elements stored in the table [36].

The primary hash function $h \in \mathscr{H}_m^d$ (cf. Section 6.2) partitions set S into m buckets, $S_i = \{x \in S: h(x) = i\}$, for $i \in [0, m-1]$, where $m = O(n)$ will be specified later. Let s_i denote the size of the ith bucket, $s_i = |S_i|$. Buckets are served by linear congruential hash functions, $h_i \in \mathscr{H}^{1,1}_{2m_i \times m_i}$, with bucket i resolved by the function $h_i(x) = (a_i x \bmod u) \bmod 2(m_i)^2$, where $s_i \leqslant m_i \leqslant 2s_i$ and $a_i \in U - \{0\}$. Each bucket has its own private memory block M_i of size $2(m_i)^2$, within which all keys $x \in S_i$ are stored without collisions. The algorithm maintains the count of keys stored, n, and the maximum number of keys that can be kept in the table, $\mathscr{N}$. Initially $\mathscr{N} = (1+c)n$ for some constant c. Whenever n exceeds $\mathscr{N}$ or n falls below $\mathscr{N}/(1+2c)$, the old table is destroyed, and a new table is constructed in $O(n)$ time using procedure *RehashAll* (Fig. 53), which is a variant of the FKS algorithm (cf. Section 1.3). Operations *insert*(x), *delete*(x) and *lookup*(x) are implemented as follows:

lookup(x): In order to check if $x \in U$ is currently stored in the hash table, we compute the number of the bucket to which x may belong, $i = h(x)$. Next the address within the memory block allocated to the ith bucket is computed, $j = h_i(x)$. If $M_i[j] = x$ the location of x is returned; otherwise, a signal that x is not in the table is returned.

delete(x): In order to delete key x we execute a *lookup*(x). If x is not in the table, no action is taken. Otherwise, n is decremented by 1 and the suitable position in block $M_{h(x)}$ is marked as empty. If the number of keys falls below the acceptable minimum, that is n less than $\mathscr{N}/(1+2c)$, the entire table is compacted by *RehashAll*, which generates a new hash table for $\mathscr{N} = (1+c)n$ keys, and records currently kept n keys in the new table.

insert(x): First, we execute *lookup*(x), to check if x is already in the table. If so, no action is taken. Otherwise, we add 1 to n, increase s_i, $i = h(x)$, compute $j = h_i(x)$ and execute one of the following actions:

$s_i \leqslant m_i \wedge M_i[j]$ is empty: Place x at the location j of block M_i.

$s_i \leqslant m_i \wedge M_i[j]$ is occupied: Collect all keys from bucket i together with x, and place them on list L_i. Next, using random search, find a linear congruence, h_i, that maps keys from L_i injectively into M_i.

$s_i > m_i$: Double the size of block i, $m_i = 2 \times m_i$. If a new block M_i of size m_i cannot be allocated, or the total size of all blocks violates the condition

$$\sum_{i=0}^{m-1} 2(s_i)^2 < \frac{8\mathcal{N}^2}{m} + 2\mathcal{N}, \tag{7.1}$$

reconstruct the entire table calling *RehashAll*. Otherwise, collect all keys from bucket i together with x, and place them on list L_i. Using random search, find a linear congruence, h_i, that maps keys from L_i injectively into M_i of the new size m_i.

Theorem 7.1 (Dietzfelbinger et al. [36, Theorem 1]). *Dynamic perfect hashing uses linear space, has* O(1) *lookup time and* O(1) *expected amortized insertion and deletion cost.*

Proof. Consider the space required by the algorithm between two successive calls to *RehashAll*. The pessimistic case occurs when only insertions are performed. (Notice that, as initially $\mathcal{N} = (1 + c)n$, and a new table is constructed whenever $n > \mathcal{N}$, no more than cn keys may be inserted.) The space used by the blocks M_i before the second call to *RehashAll* is bounded by

$$\begin{aligned}\sum_{i=0}^{m-1} 2\left((m_j)^2 + \left(\frac{m_j}{2}\right)^2 + \left(\frac{m_j}{4}\right)^2 + \cdots\right) &\leqslant \sum_{i=0}^{m-1} 2(m_j)^2 \sum_{j=0}^{\infty} \left(\frac{1}{2^j}\right)^2 \\ &= \sum_{i=0}^{m-1} 2\frac{4}{3}(m_j)^2 \leqslant \frac{16}{3} \sum_{i=0}^{m-1} 2(s_i)^2 \\ &\leqslant \frac{16}{3}\left(\frac{8(1+c)^2n^2}{m} + 2(1+c)n\right).\end{aligned}$$

The last transformation is derived using inequality (7.1). Setting $m = \frac{4}{3}\sqrt{6}(1 + c)n$, the space used by the blocks M_i is bounded by

$$\left(\frac{16}{3}\sqrt{6} + \frac{32}{3}\right)(1 + c)n.$$

The space required for the header table, that stores a_i, s_i, m_i and the pointer to M_i, is

$$4m = \frac{16}{3}\sqrt{6}(1 + c)n.$$

Hence, by allocating the total of

$$\left(\frac{32}{3}\sqrt{6}+\frac{32}{3}\right)(1+c)n+\mathrm{O}(1)<37(1+c)n$$

space for the whole scheme at the time when n elements are rehashed, the method is guaranteed to successfully perform up to cn insertions, without running out of space, providing that the primary hash function selected during rehashing satisfies inequality (7.1). By Corollary 1.3 we know that such a function can be found in O(1) expected time.

To establish the claimed time bounds we proceed as follows. Assume that the bounds were shown to hold up to some particular time τ at which *RehashAll* is called. At the time τ, n keys are stored in the table capable of accommodating $(1+c)n$ elements. We show that the expected number of calls to *RehashAll* before the table needs to be reconstructed due to n exceeding $\mathcal{N}$ is 2. Let h be a linear congruential function, $h \in \mathcal{H}_m^{1,1}$. We say that a random $h \in \mathcal{H}_m^{1,1}$ is *good* for set S if inequality (7.1) holds. By Corollary 1.3 a randomly selected h is *good* for $(1+c)n$ keys with probability exceeding $\frac{1}{2}$, and thus a function that is good for the currently stored n elements, as well for the next cn elements to be inserted or deleted will be found after an average of 2 attempts. Therefore, during the next cn insertions or deletions, the expected number of calls to *RehashAll* by *insert* is bounded by 2, with each call taking $\mathrm{O}(n)$ time to complete.

Now consider the ith subtable. Once s_i reaches m_i, the function selected for the newly created subtable is perfect for any set of size $2m_i$ keys with probability $\frac{1}{2}$. Therefore, the expected number of rehashings of a subtable needed to accommodate the current m_i and the next m_i keys to be inserted is 2. A single reconstruction takes $2(m_i)^2$ constant time steps. By charging the cost of reconstructions to the last $m_i/2$ keys inserted, we obtain a constant amortized insertion cost. This argument also holds for the first time subtable i reaches size m_i since the algorithm initiates $m_i = 2s_i$. Combining the argument establishing the expected number of calls to *RehashAll* with the above argument proves the time bounds, thus completing the proof of the theorem. □

Example 7.1. Suppose that at the time of the last call to *RehashAll* set S had 10 keys, $S = \{45, 78, 988, 11\,671, 344, 567, 12, 17, 999, 5633\}$. The parameters are set to $n = 10$, $c = 0.5$, $\mathcal{N} = 15$ and $m = 49$. Let the primary hash function be

$$h : x \mapsto (111x \bmod 11\,677) \bmod 49$$

which splits set S into 49 buckets, ten of them being not empty: $S_9 = \{12\}$, $S_{12} = \{5633\}$, $S_{14} = \{999\}$, $S_{17} = \{344\}$, $S_{18} = \{988\}$, $S_{25} = \{17\}$, $S_{34} = \{78\}$, $S_{35} = \{11\,671\}$, $S_{44} = \{567\}$ and $S_{46} = \{45\}$. Clearly,

$$20 = \sum_{i=0}^{48} 2(s_i)^2 < \frac{8 \times 15^2}{49} + 2 \times 15 \approx 66.73.$$

As each bucket contains zero or one key, all h_i are set to be

$$h_i : x \mapsto (a_i x \bmod 11\,677) \bmod 8$$

where a_i is a random number between 1 and 11 676. For brevity, we specify only two of them: $a_{34} = 1448$ and $a_{18} = 3604$. Suppose that next we insert the following seven keys: 147, 5601, 344, 4746, 981, 7001 and 133. The first key is mapped into the 34th bucket, for which we have $s_{34} = 1$, $m_{34} = 2$. We verify that $s_{34} + 1 = 2 \leqslant m_{34}$ and proceed to compute the index of 147 within M_{34}, $j = (1448 \times 147 \bmod 11\,677) \bmod 8 = 6$. As the other key, 78, that was in bucket 34 occupies position 3, no reconstruction is necessary. Next comes key 5601 which is mapped into bucket 37. As no keys occupy that bucket, we place 5601 at the $h_{37}(5601)$th position of M_{37}. The following key, 344, is already in the table, which is detected by the lookup operation in O(1) time. Key 4746, destined for bucket 18, collides with the resident key 988, as

$$(3604 \times 988 \bmod 11\,677) \bmod 8 = (3604 \times 4746 \bmod 11\,677) \bmod 8 = 0.$$

Random search is then used to find a_{18} such that 988 and 4746 do not collide. One such an a_{18} is 6577. Keys 981 and 7001 are mapped without collisions into the 25th and 10th bucket, respectively. Finally, the last key causes n, now equal 16, to exceed $\mathcal{N} = 15$, and initiates the general reconstruction of the table, by calling *RehashAll*. The new parameters of the hash function are set to: $n = 16$, $\mathcal{N} = 24$, $m = 79$.

The basic method, as described above, requires close to $40n$ space, whereas many static methods use no more than $2n$ space. It is possible however to trade off the expected amortized cost against space requirements. In particular Dietzfelbinger et al. state the following theorem, without a proof.

Theorem 7.2 (Dietzfelbinger et al. [36, Theorem 2]). *For every $\varepsilon > 0$ there are dynamic perfect hashing schemes with* O(1) *worst-case lookup cost,* O(1) *expected amortized update cost and space requirement $(6 + \varepsilon)n$ and $(1 + \varepsilon)n$, respectively.*

7.3. A real-time dictionary

A further improvement to the method was suggested by Dietzfelbinger and Meyer auf der Heide [39]. They developed a real-time dictionary, based on class $\mathcal{R}(r, m, d_1, d_2)$ of universal hash function, as defined in Definition 6.1. The dictionary is characterized by the following theorem:

Theorem 7.3 (Dietzfelbinger et al. [39, Theorem 1.1]). *There is a probabilistic (Monte Carlo type) algorithm for implementing a dictionary with the following features. Let $k \geqslant 1$ be constant.*

(i) *If after executing a sequence of instructions n keys are stored in the dictionary, then the data structure occupies space* O(n) *at this time.*

(ii) *Performing one lookup takes constant time in the worst case.*

(iii) *Performing* $\frac{1}{2}n$ *instructions in a dictionary of size* n *takes constant time per operation.*

(iv) *The algorithm is always correct; if fails to fulfill* (iii) *with probability* $O(n^{-k})$. (*The constants in the time bounds and space bound depend on* k.)

The dictionary possesses an interesting feature; it is immune to the attacks of "clocked adversaries" [70]. The idea of clocked adversary was introduced by Lipton and Naughton. They showed that several schemes for implementing dynamic hashing strategies by use of chaining are susceptible to attacks by adversaries that are able to time the algorithm. Even though the adversary has no knowledge of random choices made by the algorithm, he can draw some conclusions about the structure of the hash function being used from the time execution of certain instructions takes, and subsequently choose keys to be inserted that make the algorithm perform badly. A still open question is whether the first algorithm presented in this section [36] is susceptible to such attacks. The immunity of the real-time dictionary due to Dietzfelbinger and Meyer auf der Heide is based on the fact that the algorithm processes all instructions in constant time, giving no information to the adversary. From time to time the algorithm will fail to meet the constant time requirement. However such an event is considered to be a major failure, and the algorithm rebuilds the entire structure of the dictionary, making new, independent of the old ones, random choices.

The real-time dictionary is built from a collection of four dictionaries (Fig. 54). The dictionary operates in phases. Assume that after phase $j-1$ there is the total of n keys stored in the structure. The next phase then comprises $n/2$ rounds, executing a constant number of instructions per round. In phase j, instructions of the form (x, Op), where Op can be insert, delete or lookup, are received by dictionary D_{NEW}. At the same time the contents of dictionary D_{OLD}, that was receiving instructions during phase $j-1$, and of dictionary D_{OLDBACK}, that contains complete information stored during phases 1 to $j-2$, are transferred to D_{NEWBACK}. By the end of phase j, dictionary D_{NEW} reflects the changes made during the jth phase, dictionaries D_{OLD} and D_{OLDBACK} are empty, while dictionary D_{NEWBACK} contains complete information gathered by the end of phase $j-1$. Before the new, $(j+1)$th, phase starts, the roles of D_{NEW} and D_{OLD} as well as the roles of D_{NEWBACK}, D_{OLDBACK} are swapped. This can be achieved by simply renaming them.

To execute a lookup one needs to check dictionaries D_{NEW}, D_{OLD} and D_{OLDBACK}, in this order. Insertions are recorded in D_{NEW}, while deletions are performed by inserting the appropriate key mark "deleted" into D_{NEW}.

The real time dictionary is constructed from three different types of dictionaries.

A dictionary of type H is serviced by a congruential polynomial $h \in \mathscr{H}_n^d$ for some constant d. It can store up to n^ε keys, $0 < \varepsilon < 1$, and requires dn space, d slots for each address $i \in [0, n-1]$. Key x is inserted into one of the d slots at the $h(x)$th position of the table. A single instruction takes $O(d) = O(1)$ time. The following can be proved.

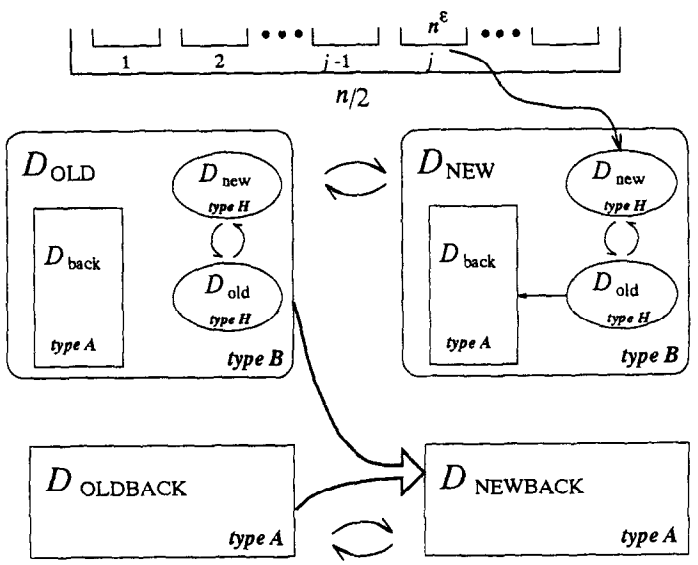

Fig. 54. The structure of the real-time dictionary.

Corollary 7.1 (Dietzfelbinger and Meyer auf der Heide [39, Corollary 2.3]). *A dictionary of type H uses dn space and with probability exceeding* $1 - O(n^{\varepsilon-(1-\varepsilon)d})$ *executes a sequence of* n^{ε} *instructions in such a way that each one takes constant time (in the worst case).*

In other words, for $d \geqslant (1+\varepsilon)/(1-\varepsilon)$, with probability $1-O(n^{-k})$, $k \geqslant 1$, no address will receive more than d keys. Starting with clear memory, setting up a dictionary of type H takes constant time required to choose $h \in \mathscr{H}_n^d$.

A dictionary of type A, designed to store up to n keys in $O(n)$ space, is serviced by a hash function η selected from class $\mathscr{R}(n^{1-\delta}, n, d_1, d_2)$. If (x_1, Op_1), (x_2, Op_2), ..., (x_n, Op_n) is the sequence of operations to be performed on a dictionary of type A, the sequence is split into $n^{1-\varepsilon}$ phases, with n^{ε} operations executed in each phase, $0<\varepsilon<1$. The structure of the dictionary resembles that of the static dictionary suggested by Fredman et al. (see Section 1.3). A key x is placed in bucket i, with $i = \eta(x)$ and $\eta \in \mathscr{R}(n^{1-\delta}, n, d_1, d_2)$ defined as

$$\eta : x \mapsto (g(x) + a_{f(x)}) \bmod n$$

where $a_i \in U - \{0\}$, $0 \leqslant i \leqslant n^{1-\delta}$, $f \in \mathscr{H}_{n^{1-\delta}}^{d_2}$, $g \in \mathscr{H}_n^{d_1}$. Each bucket S_i^{η} is serviced by a congruential polynomial of degree one, $h_i \in \mathscr{H}_{c(s_i^{\eta})^2}^{1,1}$, for $c \geqslant 2$. Let $S(j)$ be the set of keys inserted in phase j. Let index set $I = \{i : x \in S(j) \wedge \eta(x) = i\}$ be the set of indices of buckets S_i^{η} that received one or more keys from $S(j)$. For each $i \in I$ the algorithm constructs a list L_i that is a concatenation of keys already in S_i^{η} and those keys in $S(j)$ that were mapped into the ith bucket. Next for each L_i, $i \in I$, a perfect hash function is constructed. Dietzfelbinger and Meyer auf der Heide proved that the

total cost of the construction takes $O(n^\varepsilon)$ constant time steps, with high probability for sufficiently large d_1 and d_2 [39, Theorem 5.2]. Setting up a dictionary of type A takes constant time, if we assume that initially the memory is clear and only functions f and g are chosen at the beginning. The offsets a_i are fixed only when the first key x with $f(x) = i$ appears, for each $i \in \{0, 1, \ldots, n^{1-\delta}\}$.

Finally, a dictionary of type B can hold n keys in space $O(n)$. A single operation performed on a type B dictionary takes $O(1)$ time with high probability. In order to meet the constant time per instruction requirement, n, the number of keys to be inserted, must be known beforehand. It utilizes two interchangeable dictionaries of type H and one dictionary of type A set up to accommodate n keys. During the jth phase consisting of n^ε instructions of the type (x_i, Op_i), $jn^\varepsilon \leqslant i < (j+1)n^\varepsilon$, dictionary D_{new} of type H receives requests Op_i for lookups, deletion and insertions. At the same time the content of dictionary D_{old} is transferred to D_{back} of type A. At the end of phase j, D_{old} is empty, D_{new} contains information about phase j, while D_{back} reflects the state of the dictionary after $j-1$ phases. By Corollary 7.1 and subadditivity of probabilities each instruction in *every* of the $n^{1-\varepsilon}$ phases takes constant time with probability exceeding

$$1 - O(n^{1-\varepsilon} n^{\varepsilon} n^{(1-\varepsilon)d}) = 1 - O(n^{-k})$$

for d large enough. At the end of each phase the roles of D_{old} and D_{new} are swapped. Setting up a dictionary takes constant time using the initialization procedures specified for dictionaries of type H and A.

During the jth phase, which consists of $n/2$ instructions, dictionary D_{NEW} operates as the interface between the input and the real-time dictionary. By the description given above, as $n/2$ is known, we can build a dictionary of type B that processes all $n/2$ instructions in constant worst case time with high probability. At the same time, in the background, a transfer between dictionaries D_{OLD} and D_{OLDBACK} and dictionary D_{NEWBACK} occurs. The transfer may be conducted using the algorithm shown in Fig. 55.

Each key present in D_{OLD} can be in either of two states: *found* – if the key was inserted, using *insert* command or *deleted* – when the key was deleted using *delete* command (which inserts a key with a tag "deleted"). In transferring key x to D_{NEWBACK} from D_{OLDBACK} we need to check if, during the $(j-1)$th phase any changes occurred to x. We are faced with three possibilities.

1. Key x was deleted, in which case the transfer does not occur.
2. Key x was inserted, in which case the updated information associated with x is placed in D_{NEWBACK}.
3. Key x was neither inserted nor deleted. Under this condition the copy stored in D_{OLDBACK} is placed in D_{NEWBACK}.

Once transfer for D_{OLDBACK} is complete we need to add all new keys stored in D_{OLD} during the $(j-1)$th phase. Building lists L_{j-1} and L_{j-2} takes $\Theta(n)$ time. Merging them into one consistent list L takes again $\Theta(n)$ time. Finally, constructing a perfect hash function for L, $\frac{1}{2}n \leqslant |L| \leqslant \frac{3}{2}n$, takes $\Theta(n)$ time with high probability. Clearing the two old dictionaries takes $\Theta(n)$ time. Thus the algorithm in Fig. 55 takes $\Theta(n)$ time, with high probability. As during the jth phase the total of $n/2$ instructions are processed, by

```
L := ∅;
L_{j-1} := the list of keys in D_OLD;
L_{j-2} := the list of keys in D_OLDBACK;
for x ∈ L_{j-2} do
  case lookup(D_OLD, x) of
    when not found: place x on L;
    when found    : place updated x on L;
                    mark x in D_OLD as "deleted";
    when deleted  : -- do nothing; x is discarded;
  end case;
end for;
for x ∈ L_{j-1} do
  case lookup(D_OLD, x) of
    when not found: signal an error;
    when found    : place x on L;
                    mark x in D_OLD as "deleted";
    when deleted  : -- do nothing; x is discarded;
  end case;
end for;
-- L is the updated list of keys in the dictionary after the (j − 1)th phase;
Set up dictionary D_NEWBACK for |L| keys;
Build a perfect hash function for keys in L and record it in D_NEWBACK;
Make D_OLD and D_OLDBACK unavailable for access and reset them;
```

Fig. 55. A transfer of D_{OLD} and D_{OLDBACK} to D_{NEWBACK}.

executing an appropriately selected constant number of steps of the transfer algorithm, by the end of phase j, dictionary D_{NEWBACK} will correctly reflect the state of the real time dictionary at the end of the $(j-1)$th phase.

So far we assumed that there is some $n > 0$ keys in the dictionary. Thus it remains to show how to start the construction. The solution is quite simple. The first phase ends as soon as a fixed constant number of keys are in the dictionary. Consequently, the size of the dictionary is $O(1)$ during the first phase and $O(n)$, if n keys are stored in the dictionary, for the remaining phases.

7.4. Bibliographic remarks

Mehlhorn et al. considered a game on trees that is related to the dictionary problem [75]. Two players, A and B, take part. Player A models the user of the dictionary and player B models the implementation of it. The goal of A is to maximize the cost of each insertion. The goal of B is to minimize it. The authors showed that there exists a strategy for player A, such that n turns (operations on the dictionary) take $\Omega(n \log\log n)$ time, and that there always is a strategy for player B, which keeps the cost within $O(n \log\log n)$.

The schemes presented in [36, 50] naturally extend to a distributed environment. A single processor services one or more buckets. However, unlike in the single

processor environment, one needs a much more even distribution of keys among the buckets, as one needs to consider the expected time of the slowest processor.

In [37] a parallel perfect hashing strategy was implemented on a PRAM. Dietzfelbinger and Meyer auf der Heide show that n instructions can be executed in $O(n/q)$ time on q processors, as long as $n \geqslant q^{1+\varepsilon}$ and polynomials of higher than the first degree are used. By increasing the degree of the polynomial used by the primary hash function the value of ε can be made arbitrarily small. A more detailed and updated version of this paper appeared in [40].

In [38], a distributed version of a parallel dictionary, implemented on a complete network of processors without shared memory was presented. The processor that services the next request is selected on a semi-random basis, using the primary hash function selected from $\mathscr{R}(r, m, d_1, d_2)$. This ensures that as long as $n > q^{1+\varepsilon}$ each processor receives $O(n/q)$ requests, with high probability.

By using a new (the so-called OR-PRAM model) of computation, Bast et al., Dietzfelbinger and Hagerup developed a dictionary that executes each instruction in constant time, with high probability [5]. The same method, on a more common model, CRCW-PRAM, is slowed down by the factor $O(\log^{(k)} n)$ with $k \in N$ being fixed but arbitrarily large.

An overview of some pre-1991 dynamic hashing strategies can be found in [76].

Appendix A. Notation index

This appendix describes the notation and symbols that are used consistently throughout the work. The rest of the symbols we use may have different meanings depending on the chapter in which the symbols occur.

Z	the set of integer numbers, $Z = \{-\infty, \ldots, -1, 0, 1, \ldots, +\infty\}$
N	the set of natural numbers (excluding 0)
N_0	the set of natural numbers (including 0)
U	the set of all possible keys (the universe), $U = \{0, 1, 2, \ldots, u-1\}$, $U \subset N_0$
$\|X\|$	cardinality: the number of elements in set X; here X represents *any* set
Σ	a finite ordered alphabet of characters used to encode character keys
$\widehat{\sigma}$	an integer number assigned to character σ, $\sigma \in \Sigma$
h	the hash function, $h : U \to [0, m-1]$
$h^{-1}(i)$	the subset of elements of U hashed by a hash function h into the ith hash address
S (S_α)	the set of n integer (character) keys, $S \subseteq U$
x (x_α)	an integer (character) key; $x \in S$; $x_\alpha = \sigma_1 \sigma_2 \ldots \sigma_l$, $\|x_\alpha\| = l$ is the length of character key, $x_\alpha \in S_\alpha$

S_i	the set of keys $x \in S$ for which $h(x) = i$, $i \in [0, m-1]$; $S_i = \{x \in S: h(x) = i\}$, $\lvert S_i \rvert = s_i$
β	the load factor of the hash table defined as n/m
$\log n$	logarithm base 2 of n, $\log_2 n$
$\ln n$	logarithm base e of n, $\log_e n$
$f(n) = O(g(n))$	big-oh of $g(n)$; let $f, g : N \to R^+$, where R^+ denotes the set of positive real numbers; then $f(n) = O(g(n))$ if there exists $c \in R^+$ and $n_0 \in N$ such that for all $n > n_0$, $f(n) \leqslant cg(n)$
$f(n) = \Omega(g(n))$	big-omega of $g(n)$; $f(n) = \Omega(g(n))$ if there exists $c \in R^+$ and $n_0 \in N$ such that for all $n > n_0$, $f(n) \geqslant cg(n)$
$f(n) = \Theta(g(n))$	big-theta of $g(n)$; $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
$f(n) = o(g(n))$	small-oh of $g(n)$; $f(n) = o(g(n))$ if $\lim_{n\to\infty} (f(n)/g(n)) = 0$
$f(n) \sim g(n)$	$f(n)$ is asymptotically equal to $g(n)$ if $\lim_{n\to\infty} (f(n)/g(n)) = 1$
$\lfloor n \rfloor$	floor of n, greatest integer function: $\max_{k \leqslant n} k$
$\lceil n \rceil$	ceiling of n, least integer function: $\min_{k \geqslant n} k$
$(y)_i$	the falling factorial y, $(y)_i = y(y-1)\cdots(y-i+1)$
$\binom{n}{k}$	the number of k element subsets of an n element set
$\Pr(\ldots)$	probability of event described by ... to occur
$E(Y)$	the expected value of random variable Y
$\mathrm{Var}(Y)$	the variance of random variable Y

Acknowledgements

Z.J. Czech is grateful to the Department of Computer Science at the University of Queensland for supporting his visit to the department in May–June 1994. During the visit vast progress in preparation of the work was made. Z.J. Czech is also indebted to the Embassy of Australia in Poland for their help in supporting his travel to Australia. G. Havas and B.S. Majewski were partially supported by Australian Research Council grants. This work evolved from the Ph.D. thesis of the third author, who was initially supervised by the first author in Poland and then by the second author in Australia.

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms* (Addison-Wesley, Reading, MA, 1974).

[2] A.V. Aho and D. Lee, Storing a dynamic sparse table, in: *Proc. 27th Ann. Symp. on Foundations of Computer Science – FOCS'86* (Toronto, Canada, October 1986) 55–60.

[3] A.V. Aho and J.D. Ullman, *Principles of Compiler Design* (Addison-Wesley, Reading, MA, 1977).

[4] M.R. Anderson and M.G. Anderson, Comments on perfect hashing functions: A single probe retrieving method for static sets, *Comm. ACM* **22**(2) (February 1979) 104–105.

[5] H. Bast, M. Dietzfelbinger and T. Hagerup, A perfect parallel dictionary, in: *Proc. 17th Symp. on Mathematical Foundation of Computer Science – MFCS'92*, Prague, Czechoslovakia, August 1992, Lecture Notes in Computer Science, Vol. 629 (Springer, Berlin) 133–141.

[6] R.C. Bell and B. Floyd, A Monte Carlo study of Cichelli hash-function solvability, *Comm. ACM* **26**(11) (November 1983) 924–925.
[7] J. Bentley, *Programming Pearls* (Addison-Wesley, Reading, MA, 1986).
[8] B. Bollobás, *Random Graphs* (Academic Press, London, 1985).
[9] B. Bollobás and I. Simon, On the expected behaviour of disjoint set union algorithms, in: *Proc. 17th Ann. ACM Symp. on Theory of Computing – STOC'85* (May 1985) 224–231.
[10] M.D. Brain and A.L. Tharp, Near-perfect hashing of large word sets, *Software – Practice and Experience* **19**(10) (October 1989) 967–978.
[11] M.D. Brain and A.L. Tharp, Perfect hashing using sparse matrix packing, *Inform. Systems* **15**(3) (1990) 281–290.
[12] M.D. Brain and A.L. Tharp, Using tries to eliminate pattern collisions in perfect hashing, *IEEE Trans. Knowledge Data Eng.* **6**(2) (1994) 239–247.
[13] G. Brassard and P. Bratley, *Algorithmics: Theory and Practice* (Prentice-Hall, Englewood Cliffs, NJ, 1988).
[14] J.L. Carter and M.N. Wegman, Universal classes of hash functions, in: *Proc. 9th Ann. ACM Symp. on Theory of Computing – STOC'77* (1977) 106–112.
[15] J.L. Carter and M.N. Wegman, New classes and applications of hash functions, in: *Proc. 20th Ann. Symp. on Foundations of Computer Science – FOCS'79* (1979) 175–182.
[16] J.L. Carter and M.N. Wegman, Universal classes of hash functions, *J. Computer System Sci.* **18**(2) (1979) 143–154.
[17] N. Cercone, J. Boates and M. Krause, An interactive system for finding perfect hash functions, *IEEE Software* **2**(6) (November 1985) 38–53.
[18] C.C. Chang, An ordered minimal perfect hashing scheme based upon euler's theorem, *Inform. Sci.* **32** (1984) 165–172.
[19] C.C. Chang, The study of an ordered minimal perfect hashing scheme, *Comm. ACM* **27**(4) (April 1984) 384–387.
[20] C.C. Chang, Letter-oriented reciprocal hashing scheme, *Inform. Sci.* **38** (1986) 243–255.
[21] C.C. Chang, A composite perfect hashing scheme for large letter-oriented key sets, *J. Inform. Sci. Engng.* **7**(2) (1991) 173–186.
[22] C.C. Chang and C.H. Chang, An ordered minimal perfect hashing scheme with single parameter, *Inform. Process. Lett.* **27**(2) (February 1988) 79–83.
[23] C.C. Chang, C.Y. Chen and J.K. Jan, On the design of a machine independent perfect hashing scheme, *Comput. J.* **34**(5) (1991) 469–474.
[24] C.C. Chang, H.C. Kowng and T.C. Wu, A refinement of a compression-oriented addressing scheme, *BIT* **33**(4) (1993) 530–535.
[25] C.C. Chang and R.C.T. Lee, A letter-oriented minimal perfect hashing scheme, *Comput. J.* **29**(3) (June 1986) 277–281.
[26] C.C. Chang and H.C. Wu, A fast chinese characters accessing technique using mandarin phonetic transcriptions, *Comput. Process. Chinese Oriental Languages* **3**(3–4) (1988) 275–307.
[27] C.C. Chang and T.C. Wu, A letter-oriented perfect hashing scheme based upon sparse table compression, *Software – Practice Experience* **21**(1) (January 1991) 35–49.
[28] R.J. Cichelli, Minimal perfect hash functions made simple, *Comm. ACM* **23**(1) (January 1980) 17–19.
[29] C.R. Cook and R.R. Oldehoeft, A letter oriented minimal perfect hashing function, *SIGPLAN Notices* **17**(9) (September 1982) 18–27.
[30] G.V. Cormack, R.N.S. Horspool and M. Kaiserwerth, Practical perfect hashing, *Comput. J.* **28**(1) (February 1985) 54–55.
[31] Z.J. Czech, G. Havas and B.S. Majewski, An optimal algorithm for generating minimal perfect hash functions, *Inform. Process. Lett.* **43**(5) (October 1992) 257–264.
[32] Z.J. Czech and B.S. Majewski, Generating a minimal perfect hashing function in $O(m^2)$ time, *Archiwum Informatyki Teoretycznej i Stosowanej* **4** (1992) 3–20.
[33] Z.J. Czech and B.S. Majewski, A linear time algorithm for finding minimal perfect hash functions, *Comput. J.* **36**(6) (1993) 579–587.
[34] N. Deo, G.M. Prabhu and M.S. Krishnamoorthy, Algorithms for generating fundamental cycles in a graph, *ACM Trans. Math. Software* **8**(1) (March 1982) 26–42.
[35] M. Dietzfelbinger, J. Gil, Y. Matias and N. Pippenger, Polynomial hash functions are reliable, in: *Proc. 19th Internat. Colloq. on Automata, Languages and Programming – ICALP'92*, Vienna, Austria, July 1992, Lecture Notes in Computer Science, Vol. 623 (Springer, Berlin, 1992) 235–246.

[36] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert and R.E. Tarjan, Dynamic perfect hashing: upper and lower bounds, in: *Proc. 29th Ann. Symp. on Foundations of Computer Science – FOCS'88*, White Plains NY, October 1988; Reprinted in *SIAM J. Comput.* **24**(4) (August 1994) 738–761.

[37] M. Dietzfelbinger and F. Meyer auf der Heide, An optimal parallel dictionary, in: *Proc. ACM Symp. on Parallel Algorithms and Architectures*, Santa Fe, Mexico (1989) 360–368.

[38] M. Dietzfelbinger and F. Meyer auf der Heide, How to distribute a dictionary on a complete network, in: *Proc. 22nd ACM Symp. on Theory of Computing – STOC'90* (1990) 117–127.

[39] M. Dietzfelbinger and F. Meyer auf der Heide, A new universal class of hash functions, and dynamic hashing in real time, in: *Proc. 17th Internat. Colloq. on Automata, Languages and Programming – ICALP'90*, Warwick University, England, July 1990, Lecture Notes in Computer Science, Vol. 443 (Springer, Berlin, July 1990) 6–19.

[40] M. Dietzfelbinger and F. Meyer auf der Heide, An optimal parallel dictionary, *Inform. and Comput.* **102** (1993) 196–217.

[41] M.W. Du, T.M. Hsieh, K.F. Jea and D.W Shieh, The study of a new perfect hash scheme, *IEEE Trans. Software Eng.* **SE-9**(3) (May 1983) 305–313.

[42] R. Duke, Types of cycles in hypergraphs, *Ann. Discrete Math.* **27** (1985) 399–418.

[43] P. Erdős and A. Rényi, On the evolution of random graphs, *Publ. Math. Inst. Hung. Acad. Sci.* **5** (1960) 17–61; see also J.H. Spencer, ed., The art of counting: selected writings, in: *Mathematicians of Our Time* (MIT Press, Cambridge, MA, 1973) 574–617.

[44] P. Erdős and A. Rényi, On the evolution of random graphs, *Bull. Inst. Internat. Statis.* **38** (1961) 343–347; see also J.H. Spencer, ed., The art of counting: selected writings, *Mathematicians of Our Time* (MIT Press, Cambridge, MA, 1973) 569–573.

[45] P. Flajolet, D.E. Knuth and B. Pittel, The first cycles in an evolving graph, *Discrete Math.* **75** (1989) 167–215.

[46] E.A. Fox, Q.F. Chen, A.M. Daoud and L.S. Heath, Order preserving minimal perfect hash functions and information retrieval, *ACM Trans. Inform. Systems* **9**(3) (July 1991) 281–308.

[47] E.A. Fox, Q.F. Chen and L.S. Heath, A faster algorithm for constructing minimal perfect hash functions, in: *Proc. 15th Ann. Internat. ACM SIGIR Conf. on Research and Development in Information Retrieval – SIGIR'92* (Copenhagen, Denmark, June 1992) 266–273.

[48] E.A. Fox, L.S. Heath and Q.F. Chen, An $O(n \log n)$ algorithm for finding minimal perfect hash functions, Tech. Report TR-89-10, Virginia Polytechnic Institute and State University, Blacksburg, VA, April 1989.

[49] E.A. Fox, L.S. Heath, Q.F. Chen and A.M. Daoud, Practical minimal perfect hash functions for large databases, *Comm. ACM* **35**(1) (January 1992) 105–121.

[50] M.L. Fredman, J. Komlós and E. Szemerédi, Storing a sparse table with O(1) worst case access time, *J. ACM* **31**(3) (July 1984) 538–544.

[51] M.R. Garey and D.S. Johnson, *Computers and Intractibility: A Guide to the Theory of NP-completeness* (W.H. Freeman, San Francisco, 1979).

[52] M. Gori and G. Soda, An algebraic approach to Cichelli's perfect hashing, *BIT* **29**(1) (1989) 2–13.

[53] M. Greniewski and M. Turski, The external language KLIPA for the URAL-2 digital computer, *Comm. ACM* **6**(6) (June 1963) 322–324.

[54] R. Gupta, S. Bhaskar and S.A. Smolka, On randomization in sequential and distributed algorithms, *ACM Comput. Surveys* **26**(1) (March 1994) 7–86.

[55] G. Haggard and K. Karplus, Finding minimal perfect hash functions, *ACM SIGCSE Bull.* **18** (1986) 191–193.

[56] D. Harel, *Algorithmics: The Spirit of Computing* (Addison-Wesley, Reading, MA, 1987).

[57] G. Havas and B.S. Majewski, Graph theoretic obstacles to perfect hashing, *Congressus Numerantium* **98** (1993) 81–93; see also *Proc. 24th Southeastern Internat. Conf. on Combinatorics, Graph Theory and Computing.*

[58] G. Havas, B.S. Majewski, N.C. Wormald and Z.J. Czech, Graphs, hypergraphs and hashing, in: *Proc. 19th Internat. Workshop on Graph-Theoretic Concepts in Computer Science (WG'93)*, Utrecht, 1993, Lecture Notes in Computer Science, Vol. 790 (Springer, Berlin, 1994) 153–165.

[59] L.G. Hua, *An Introduction to Number Theory* (Sheng-Deng Reading, Taiwan, 1975) (in Chinese).

[60] C.T.M. Jacobs and P. van Emde Boas, Two results on tables, *Inform. Process. Lett.* **22** (1986) 43–48.

[61] G. Jaeschke, Reciprocal hashing: A method for generating minimal perfect hashing functions, *Comm. ACM* **24**(12) (December 1981) 829–833.
[62] G. Jaeschke and G. Osterburg, On Cichelli's minimal perfect hash functions method, *Comm. ACM* **23**(12) (December 1980) 728–729.
[63] J. Jan and C. Chang, Addressing for letter-oriented keys, *J. Chinese Inst. Eng.* **11**(3) (1988) 279–284.
[64] S. Janson, D.E. Knuth, T. Łuczak and B. Pittel, The birth of the giant component, *Random Struct. Algorithms* **4** (1993) 233–358.
[65] D.E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms* (Addison-Wesley, Reading, MA, 2nd ed., 1973).
[66] D.E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching* (Addison-Wesley, Reading, MA, 2nd ed., 1973).
[67] D.E. Knuth and A. Schönhage, The expected linearity of a simple equivalence algorithm, *Theoret. Comput. Sci.* **6** (1978) 281–315.
[68] P. Larson and M.V. Ramakrishna, External perfect hashing, in: *Proc. ACM SIGMOD*, Austin TX (June 1985) 190–200.
[69] T.G. Lewis and C.R. Cook, Hashing for dynamic and static internal tables, *Computer* **21** (1988) 45–56.
[70] R.J. Lipton and J.F. Naughton, Clocked adversaries for hashing, *Algorithmica* **9**(3) (March 1993) 239–252.
[71] C.T. Long, *Elementary Introduction to Number Theory* (Heath, Boston, 1965).
[72] H.G. Mairson, The program complexity of searching a table, in: *Proc. 24th Ann. Symp. on Foundations of Computer Science – FOCS'83*, Tuscon, AZ (November 1983) 40–47.
[73] B.S. Majewski, Minimal perfect hash functions, Ph.D. Thesis, University of Queensland, Department of Computer Science, November 1992.
[74] K. Mehlhorn, *Data Structures and Algorithms 1: Sorting and Searching* (Springer, Berlin, 1984).
[75] K. Mehlhorn, S. Näher and M. Rauch, On the complexity of a game related to the dictionary problem, *SIAM J. Comput.* **19**(5) (October 1990) 902–906.
[76] F. Meyer auf der Heide, Dynamic hashing strategies, in: *Proc. 15th Symp. on Mathematical Foundations of Computer Science – MFCS'90*, Banska Bystrica, Czechoslovakia (1990) 76–87.
[77] E.M. Palmer, *Graphical Evolution: An Introduction to the Theory of Random Graphs* (Wiley, New York, 1985).
[78] P.K. Pearson, Fast hashing of variable-length text strings, *Comm. ACM* **33**(6) (June 1990) 677–680.
[79] B. Pittel, J. Spencer and N.C. Wormald, The sudden emergence of a giant k-core in a random graph, *J. Combin. Theory, Ser. B*, submitted.
[80] M.O. Rabin, Probabilistic algorithms, in: J.F. Traub, ed., *Algorithms and Complexity: Recent Results and New Directions* (Academic Press, New York, April 1976) 21–39.
[81] P. Raghavan, *Lecture Notes on Randomized Algorithms* (IBM Research Division, Yorktown Heights, NY, 1990).
[82] M.V. Ramakrishna and P. Larson, File organization using composite perfect hashing, *ACM Trans. Database Systems* **14**(2) (June 1989) 231–263.
[83] R.C. Read, Euler graphs on labelled nodes, *Canad. J. Math.* **14** (1962) 482–486.
[84] T.J. Sager, A new method for generating minimal perfect hash functions, Tech. Report CSc-84-15, University of Missouri-Rolla, Department of Computer Science, 1984.
[85] T.J. Sager, A polynomial time generator for minimal perfect hash functions, *Comm. ACM* **28**(5) (May 1985) 523–532.
[86] J.P. Schmidt and A. Siegel, The analysis of closed hashing under limited randomness, in: *Proc. 22nd Ann. ACM Symp. on Theory of Computing – STOC'90*, Baltimore, MD (May 1990) 224–234.
[87] J.P. Schmidt and A. Siegel, The spatial complexity of oblivious k-probe hash functions, *SIAM J. Comput.*, **19**(5) (October 1990) 775–786.
[88] J. Shallit, Randomized algorithms in "primitive cultures", *SIGACT News* **23**(4) (1992) 77–80.
[89] A. Siegel, On universal classes of fast high performance hash functions, their time-space trade-off, and their applications, in: *Proc. 30th Ann. Symp. on Foundations of Computer Science – FOCS'89* (October 1989) 20–25.
[90] W. Sierpinski, *A selection of Problems in the Theory of Numbers* (Pergamon, London, 1964).
[91] B. Singh and T.L. Naps, *Introduction to Data Structures* (West Publishing, St. Paul, MI, 1985).

[92] C. Slot and P. van Emde Boas, On tape versus core; an application of space efficient perfect hash functions to the invariance of space, in: *Proc. 16th Ann. ACM Symp. on Theory of Computing – STOC'84*, Washington, DC (May 1984) 391–400.
[93] R. Sprugnoli, Perfect hashing functions: a single probe retrieving method for static sets, *Comm. ACM* **20**(11) (November 1977) 841–850.
[94] T.A. Standish, *Data Structures Techniques* (Addison-Wesley, Reading, MA, 1980).
[95] R.E. Tarjan, Graph theory and gaussian elimination, in: J.R. Bunch and D.J. Rose, eds., *Sparse Matrix Computations* (Academic Press, New York, 1976) 3–22.
[96] R.E. Tarjan and J. Van Leeuwen, Worst-case analysis of set union algorithms, *J. ACM* **31**(2) (April 1984) 245–281.
[97] R.E. Tarjan and A.C.C Yao, Storing a sparse table, *Comm. ACM* **22**(11) (November 1979) 606–611.
[98] J.A. Trono, An undergraduate project to compute minimal perfect hashing functions, *SIGCSE Bull.* **24**(3) (1992) 53–56.
[99] V.G. Winters, Minimal perfect hashing for large sets of data, in: *Internat. Conf. on Computing and Information – ICCI'90* (Niagara Falls, Canada, May 1990) 275–284.
[100] V.G. Winters, Minimal perfect hashing in polynomial time, *BIT* **30**(2) (1990) 235–244.
[101] I.H. Witten, A. Moffat and T.C. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images* (Van Nostrand Reinhold, New York, 1994).
[102] W.P. Yang and M.W. Du, A backtracking method for constructing perfect hash functions from a set of mapping functions, *BIT* **25**(1) (1985) 148–164.
[103] A.C. Yao, On the expected performance of path compression algorithms, *SIAM J. Comput.* **14** (February 1985) 129–133.
[104] S.F. Ziegler, Smaller faster table driven parser, Unpublished Manuscript. Madison Academic Computing Center, University of Wisconsin, Madison, Wisconsin, 1977.