Constructing Minimal Perfect Hash Functions via Independent Sets and Exact Cover

Marlon Dittes

November 20, 2026

4164834

Master Thesis

at

Algorithm Engineering Group Heidelberg
Heidelberg University

Supervisor:
Univ.-Prof. PD. Dr. rer. nat. Christian Schulz

Co-Supervisor:
Dr. Ernestine Großmann and Dr. Kenneth Langedal

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Prof. Dr. Christian Schulz, for his constant guidance and support throughout this thesis and my academic journey.

A special thanks to Ernestine Großmann and Kenneth Langedal for always providing constructive feedback and insightful suggestions. I met with both of them every week, and their advice, patience and the many discussions in these meetings helped me a great deal with this work.

Finally, I would like to express my heartfelt gratitude to my family and friends for their unwavering support, love, and encouragement throughout my life. Their constant belief in me and their presence have been invaluable in helping me complete this work.

I hereby confirm, Marlon Dittes, Matr. No. 4164834, that this thesis has been written independently and without unauthorised external assistance. I have not used any sources or aids other than those specified, and I have identified all passages that have been taken verbatim or paraphrased from published or unpublished works, including those from digital or AI-based sources. I confirm that any use of AI tools was discussed in advance with the supervisor of the thesis and that the agreed rules were followed. I take full responsibility for the academic quality and content of this paper, the methodology chosen and the process of its creation, as well as the literature cited. I agree that my thesis will be checked for plagiarism and possible misuse of automated text and code generation as part of university procedures.

Heidelberg, November 20, 2026



Marlon Dittes

Abstract

A minimal perfect hash function maps n static keys bijectively to $\{0, \ldots, n-1\}$ and needs at least $\log_2 e \approx 1.44$ bits per key. We study a construction that splits the keys into small parts, distributes the keys of a part into buckets and searches for one seed per bucket such that all keys of the part receive different slots. The choice of seed is expressed as an independent set or exact cover problem. We solve it with dancing links and repeat the search with new hash functions when no solution exists. We evaluate the search and the complete construction experimentally. The success probability of a single attempt depends heavily on the number of seed bits per key, and re-rolls make it possible to use seeds close to the lower limit for a part. With a simple prefix-sum layout, the complete function needs between 2.0 and 2.3 bits per key, against the lower bound of 1.44; the seeds take about 1.5 bits per key, and most of the remaining overhead comes from the offsets of the parts. The construction scales linearly in the number of keys and builds a function for 10^7 keys in about six minutes. Dancing links decide the instances of a part about three orders of magnitude faster than an ILP solver and decide more of them within the same time limit. A limit of 10^5 search nodes per attempt makes the construction roughly 2.6 times cheaper than 10^6 nodes at nearly the same space. We discuss how compact offsets and better encodings could bring the space closer to the state of the art.

Declaration on the Use of AI Tools

During the work on this thesis I used the AI assistant Claude (Anthropic, model Claude Sonnet 5.5) as a tool. All ideas, the problem formulation, the core algorithm and its implementation, the experimental design, and the decisions about what to include were mine, with the guidance of my supervisors. The assistant was used for the following tasks:

- **Writing support.** AI was used as a helper for the structure and the wording of some chapters. It suggested outlines and formulations for some passages, which I checked, revised and rewrote in my own words. The content, the experiments and the conclusions are my own work.
- **LaTeX constructs.** It generated LaTeX code for tables and figures (for example the tabular environments and the inclusion of the figures), which I then checked and adapted.
- **Program code for the experiments.** It helped create code for the analysis of the data structure: the scripts that run the experiments and the notebooks that compute the tables and figures from the measured data. I checked all code by hand, ran all experiments myself on my own machines and checked the results.
- **Analysis and explanations.** It helped to compute and interpret statistics (for example confidence intervals and interpolated thresholds), explained concepts (for example Elias–Fano coding and the CONSENSUS approach) and pointed to literature.
- **Literature.** It suggested references and summarised parts of papers. I read the cited works myself and checked every reference and statement before using it.

I checked all numbers in the thesis against the raw measurements, and I take full responsibility for the content of this thesis, including the parts that were produced with the help of the assistant.

Heidelberg, November 20, 2026



Marlon Dittes

Contents

Abstract v

Declaration on the Use of AI Tools vii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Our Contribution . . . 2
- 1.3 Structure . . . 3

2 Fundamentals 5
- 2.1 Notation and Conventions . . . 5
- 2.2 Perfect Hashing . . . 5
- 2.3 Space Lower Bound . . . 7
- 2.4 Independent Sets and Exact Cover . . . 7
  - 2.4.1 Algorithm X . . . 8
  - 2.4.2 The Dancing-Links Data Structure . . . 8
- 2.5 Probabilistic Tools . . . 9

3 Related Work 11
- 3.1 Foundations . . . 11
- 3.2 Families of Constructions . . . 12
  - 3.2.1 Through Retrieval . . . 12
  - 3.2.2 Through Fingerprinting . . . 12
  - 3.2.3 Through Brute Force and Recursive Splitting . . . 12
- 3.3 Bucket Placement . . . 13
- 3.4 k-Perfect Hashing . . . 15
- 3.5 Solver-Based Approaches . . . 16
- 3.6 Positioning of This Thesis . . . 16

4 Our Construction 17
- 4.1 Overview . . . 17

4.2 Problem Formulation . . . 18
4.2.1 Setting and Notation . . . 18
4.2.2 The Conflict Graph . . . 19
4.2.3 From Independent Set to Exact Cover . . . 20
4.2.4 How Many Solutions to Expect . . . 21
4.3 Solving the Exact Cover Instance . . . 22
4.3.1 A Worked Example . . . 23
4.4 Re-rolls and Restarts . . . 24
4.4.1 Why an Attempt Can Fail . . . 24
4.4.2 Defining Re-rolls . . . 25
4.4.3 The Attempt Loop . . . 26
4.4.4 Re-roll Costs . . . 27
4.4.5 Choosing a Cutoff . . . 27
4.5 Parts, Query and Space . . . 27
4.5.1 Why Split into Parts . . . 27
4.5.2 Combining the Parts . . . 28
4.6 Discontinued Approaches . . . 30
4.6.1 Independent Set Solvers on the Conflict Graph . . . 30
4.6.2 Making the Instances Easier . . . 31
4.6.3 Checking that Solutions Exist . . . 32
4.6.4 Relaxation to k-Perfect Hashing . . . 32
4.6.5 Turning to Small Parts . . . 33

5 Experimental Evaluation **35**
5.1 Experimental Setup . . . 36
5.2 Re-roll Cost and Space on Small Instances . . . 36
5.3 Success Rate of a Single Attempt . . . 38
5.4 Dancing Links Against an ILP Solver . . . 42
5.5 The Whole Construction on Large Key Sets . . . 45
5.6 Parameters of the Search . . . 51

6 Discussion **57**
6.1 Interpretation of the Results . . . 57
6.2 Conclusion . . . 58
6.3 Future Work . . . 59

Abstract (German) **61**

Bibliography **63**

CHAPTER 1

Introduction

1.1 Motivation

Many applications need to retrieve values associated with a fixed set of keys: the words of a dictionary, the k-mers of a genome, the entries of a database table. The standard solution is a hash table. A hash table stores the keys themselves, which for long keys such as strings or DNA sequences can be by far the largest part of its memory. A *minimal perfect hash function* (MPHF) avoids this. It gives each of the n keys of a static set its own number between 0 and $n-1$. The values can be kept in a plain array without collisions and without storing a single key. Moreover, the function requires little space: state-of-the-art constructions only need about 1.5 bits per key, independent of the length of the keys [23]. However, this approach has one limitation: the function is only defined for the keys of the original set.

Perfect hashing functions see use in multiple different areas, for example static hash tables and retrieval structures, approximate membership queries, databases, bioinformatics and text indexing. Many of these applications share the following: the structure is built once for a very large set and gets queried very often. Therefore, space and query time matter the most. For a more thorough review of the application areas, we refer to the survey of Lehmann et al. [23].

Every minimal perfect hash function needs at least $\log_2(n^n/n!) \approx n \log_2 e$ bits, i.e. asymptotically $\log_2 e \approx 1.44$ bits per key (Section 2.3). Recent constructions come close to this limit: the survey reports that 1.444 bits per key can be reached in practice, and ShockHash reaches 1.489 bits per key for 10^7 keys [23, 21].

One approach to reaching the theoretical space limit is to distribute the keys into small parts and to search for each part hash functions that map its corresponding keys to distinct slots. Finding such a function for all keys at once is infeasible for large sets, because the search time grows exponentially with the number of keys, whereas small parts can be solved quickly and independently. The search is faster for smaller parts, but this introduces

more space overhead. In this thesis we investigate how the search for small parts can be done by an exact method and what it costs in space and time when it is used for a complete function.

1.2 Our Contribution

We construct minimal perfect hash functions from small parts: the keys of each part are first distributed into buckets and we search for each bucket a seed such that all keys of the part land in different slots. Selecting one collision-free seed per bucket can be formulated as an independent set problem in a conflict graph or, equivalently, as an exact cover problem. We use dancing links to solve the various independent parts. If no solution exists, we repeat the search with new hash functions (re-rolls). Our contributions are as follows:

- We formulate the choice of the seeds of a part as an independent set and exact cover problem and implement an exact search with dancing links (Chapter 4).
- We measure how the success probability of one attempt depends on the number of seed bits per key and on the part size, compare it with a lower limit for the part size, and show how re-rolls trade the space for the seeds against the space for the number of attempts (Sections 5.2 and 5.3).
- We compare dancing links with an ILP solver. Dancing links is faster by about three orders of magnitude and decides more instances within the same time limit (Section 5.4).
- We implement the complete construction using a simple prefix-sum layout and evaluate it on sets of up to 10^7 keys. The resulting function needs between 2.0 and 2.3 bits per key; the seeds need about 1.44 to 1.6 bits per key, and most of the remaining overhead comes from storing the offsets of the parts. The work per key is constant in n and the construction scales nicely to 128 threads (Section 5.5).
- We examine the parameters of the search, particularly the limit on the number of nodes per attempt, and show that a limit of about 10^5 nodes makes the construction about 2.6 times cheaper at the same space (Section 5.6).

Our function does not yet reach the space of the best known constructions; Chapter 6 describes where the remaining overhead comes from and how it could be reduced.

1.3 Structure

The remainder of this thesis is organized as follows. Chapter 2 introduces the notation, the lower bound for minimal perfect hashing and the exact cover problem with dancing links. Chapter 3 discusses related work on minimal perfect hashing. Chapter 4 describes our construction: the partition into parts and buckets, the formulation as an independent set and exact cover problem, the search with re-rolls, and the layout of the data structure. Chapter 5 evaluates the construction experimentally. Chapter 6 concludes with a summary and an outlook on future work.

CHAPTER 2

Fundamentals

2.1 Notation and Conventions

We write $[k] = \{0, \ldots, k-1\}$ and $\log = \log_2$. The universe of keys is $U = [2^{64}]$, and $S \subseteq U$ denotes the key set with $|S| = n$. Keys are 64-bit integers. We assume longer keys to be reduced to 64 bits beforehand. For the space bounds we count the bits of the data structure excluding the key set itself. In Table 2.1 we list the symbols used throughout the thesis. We define the construction of the data structure itself in Chapter 4.

2.2 Perfect Hashing

Definition 1 (Perfect and minimal perfect hash function)
Let $S \subseteq U$ with $|S| = n$ and let $R \in \mathbb{N}$. A function $f\colon S \to [R]$ is called a perfect hash function *(PHF) for S if it is injective. The* load factor *is $\alpha = n/R$. If $R = n$, i.e. $\alpha = 1$, f is bijective and it is called a* minimal *perfect hash function (MPHF).*

A PHF is built once for a known set S and its behaviour on keys outside of S is undefined. We measure the quality of a PHF by the space of its representation, usually in bits per key, by the construction time, and by the query time.

Generalisation. Sometimes we are interested in a slot not being a single cell but a rather a block that can hold several keys, for example a bucket of a hash table. In this case, it is enough that no slot receives more than k keys.

Definition 2 (k-perfect hash function)
Let $S \subseteq U$ with $|S| = n$, and let $k, R \in \mathbb{N}$ with $kR \geq n$. A function $f\colon S \to [R]$ is a k-perfect hash function *for S if every slot receives at most k keys, i.e.*

$$\left|f^{-1}(j)\right| \leq k \qquad \text{for all } j \in [R].$$

Symbol	Meaning	Defined in
U, S, n	universe, key set, number of keys $n = \lvert S \rvert$	Sec. 2.2
α, R	load factor and range size $R = \lceil n/\alpha \rceil$; $\alpha = 1$ is minimal	Sec. 2.2
m	number of buckets of a part, $m = \lceil n/\lambda \rceil$; over all parts, $m = \sum_p m_p$	Sec. 4.2, 4.5
P, S_p, n_p, R_p, m_p	number of parts, keys of part p and its size, range and number of buckets	Sec. 4.5
β, B_i	bucket function and the i-th bucket, $i = 0, \ldots, m-1$	Sec. 4.2
λ	average number of keys per bucket	Sec. 4.2
Σ, σ	seed set $\{0, \ldots, \sigma - 1\}$ and its size	Sec. 4.2
h_s, g, fx	hash function of seed s, global seed, multiplicative mixing function	Sec. 4.2
π, f_π	seed assignment (one seed per bucket) and the resulting function	Sec. 4.2
(i, s)	candidate: bucket i with seed s	Sec. 4.2
$G = (V, E)$	conflict graph on the candidates	Sec. 4.2
b_i, z_j, $A_{(i,s)}$	bucket element, slot element, and the set $\{b_i\} \cup \{z_j : j \in h_s(B_i)\}$ of a candidate	Sec. 4.2.3
a, a_p, w_a	attempt index, the one stored for part p, bits to store it	Sec. 4.4, 4.5.2
O_p, C_p	prefix sums of the ranges and of the bucket counts	Sec. 4.5.2
$\oplus$, hi64, lo64	bitwise XOR; high and low 64 bits of a 128-bit product	Sec. 4.4, 4.5.2

Table 2.1: Symbols used throughout the thesis.

The load factor *here is* $\alpha = n/(kR) \in (0, 1]$*, which may be used to increase the number of bins, resulting in extra capacity. The function is* minimal *if* $R = n/k$*, i.e. if* $\alpha = 1$*; else we call it* non-minimal*. For* $k = 1$ *this is exactly the definition of a perfect hash function.*

Idealised hash functions. The analysis in this thesis relies on heuristics whenever it makes probabilistic claims about hash values. We assume the *random hash model*: for every seed s the function $h_s \colon U \to [R]$ behaves like a uniformly random function, independent of all other seeds. Since real hash functions are not random, the model is an idealisation. Nonetheless, the correctness of a constructed function does not ever rely on it. The construction itself directly verifies injectivity of the final function.

n	10	20	30	40	50	60	10^2	10^3	10^4
α_n	1.143	1.268	1.317	1.343	1.360	1.371	1.396	1.436	1.442

Table 2.2: The information-theoretic limit $\alpha_n = \frac{1}{n}\log_2 \frac{n^n}{n!}$ of bits per key for n keys. It tends to $\log_2 e \approx 1.443$.

2.3 Space Lower Bound

There is an information-theoretic lower bound of $n \log_2(e)$ bits for representing a minimal perfect hash function on n random keys [26]. To build some intuition, consider an arbitrary hash function mapping n keys to $[n]$. The probability that such a function is minimal perfect is $n!/n^n$. Consequently, distinguishing one minimal perfect hash function from all possible functions requires at least

$$\log_2\left(\frac{n^n}{n!}\right) \approx n \log_2(e)$$

bits of information. Among existing constructions, CONSENSUS-RecSplit [22] comes closest to this lower bound, achieving space usage as low as 1.444 bits per key.

Table 2.2 shows that this bound depends on n. For a small set of keys it lies well below $\log_2 e$ and approaches it only slowly: $\alpha_n = 1.143$ bits per key for $n = 10$, 1.396 for $n = 100$ and 1.442 for $n = 10^4$. This matters for our construction, because it splits the keys into small parts that are solved independently (Chapter 4). For a part of n_p keys, the relevant limit is α_{n_p}, which is smaller than the limit $\log_2 e$ for the whole set. The values α_{n_p} are the reference for the success rates of a single part in Section 5.3.

2.4 Independent Sets and Exact Cover

Independent Set. Let $G = (V, E)$ be a graph. A set $I \subseteq V$ is *independent* if no two vertices of I are adjacent. Deciding whether G has an independet set of a given size is NP-complete [8]. Therefore, finding a maximum independent set is NP-hard.

Exact Cover. Let X be a finite set of *columns* and let $\mathcal{A}$ be a family of subsets of X, which we call *rows*. An *exact cover* is a subfamily $\mathcal{S} \subseteq \mathcal{A}$ such that every column lies in exactly one row of $\mathcal{S}$. In matrix form, rows are $0/1$ vectors and we select rows such that every column has exactly one 1. Deciding whether an exact cover exists is NP-complete [18]. In a useful generalisation we may distinguish *primary* columns, which have to be covered exactly once, from *secondary* columns, which may be covered at most once.

Algorithm 1: SEARCH(): Algorithm X on the linked matrix.

```
if no primary column is left then
    return success;
c ← primary column with the fewest remaining rows (ties broken by the chosen
  rule);
if c has no remaining row then
    return failure;
COVER(c);
for each row r in column c do
    add r to the partial solution;
    for each node j ≠ r in row r, from left to right do
        COVER(column of j);
    if SEARCH() succeeds then
        return success;
    for each node j ≠ r in row r, from right to left do
        UNCOVER(column of j);
    remove r from the partial solution;
UNCOVER(c);
return failure;
```

2.4.1 Algorithm X

Algorithm X is a straightforward backtracking algorithm to find an exact cover (Algorithm 1), presented by Knuth [19]. It picks a primary column c that has not yet been covered. Some row containing c always has to be part of the solution, so the algorithm tries each such row r in turn: it adds r to the partial solution, removes from the problem all columns covered by r together with all rows that intersect r, since choosing any such row later would cover a column twice. It then recurses on the reduced problem. If the recursion fails, the removal is undone and the algorithm tries the next row. The search succeeds when no primary column remains uncovered. Which column to pick is not fixed by the algorithm; a standard choice is a column with the fewest remaining rows able to cover it.

2.4.2 The Dancing-Links Data Structure

It remains to show how to implement the removal and undoing efficiently. Copying the matrix would be way too expensive, which is where the dancing-links data structure [19] comes in. We store the matrix sparsely: every 1-entry is a *node*, and every node has four pointers L, R, U, D that connect to its neighbors in a circular doubly linked list, horizontally within its row (L and R) and vertically within its column (U and D). Each

column additionally has a *header* node at the top of its list, which stores the *size* of a column, i.e. the number of remaining rows that contain it. The headers of the primary columns are linked in a circular list through a *root* node so that all columns that have not yet been covered can be traversed easily.

Deleting a node x from a doubly linked list is done as follows:

$$L[R[x]] \leftarrow L[x], \qquad R[L[x]] \leftarrow R[x].$$

This leaves the pointers of x unchanged, so x still remembers its former neighbors. If all deletions are undone in reverse order, we can put x back by

$$L[R[x]] \leftarrow x, \qquad R[L[x]] \leftarrow x,$$

restoring the list to its previous state. Hence, we need no copies and no stored history besides the recursion stack. The nodes are unlinked on the way down and relinked on the way up, which is why Knuth calls the data structure *dancing links*.

The two operations $\textsc{Cover}(c)$ and $\textsc{Uncover}(c)$ are built on this idea.

- $\textsc{Cover}(c)$ unlinks the header of column c from the header list. Then, for every row r in column c (top to bottom) and for every other node j in row r (left to right), it unlinks j from its column list and decrements the size of its column. This removes all rows that contain c from all the other columns they cover.
- $\textsc{Uncover}(c)$ reverses the process: by going through the rows in column c from bottom to top and the nodes in each row from right to left, it relinks all nodes back into their columns and increments the sizes. Finally, it relinks the header of column c back into the header list.

The cost of both operations is proportional to the number of nodes that are removed or restored, not to the size of the matrix.

2.5 Probabilistic Tools

We collect elementary facts used in the analysis of the construction, all of which are textbook material.

Theorem 1 (Injectivity of a random function)
Let h be a uniformly random function from a set of k keys to $[R]$. Then

$$\Pr[h \text{ injective}] \;=\; \prod_{j=0}^{k-1}\Big(1-\frac{j}{R}\Big) \;\geq\; 1-\frac{k(k-1)}{2R}.$$

Proof. Insert the keys one after another; the j-th key avoids the j slots taken so far with probability $1-j/R$, which gives the product. For the bound, any two keys collide with probability $1/R$, and the union bound over the $\binom{k}{2}$ pairs bounds the probability that some pair collides by $\binom{k}{2}/R$. For $k = R = n$ the product is $n!/n^n$.

Restarts. Suppose we repeat a randomized procedure independently until it succeeds. Let attempt i have cost X_i and let N be the number of attempts up to and including the first success. If the attempts are independent and identically distributed, then N is geometrically distributed with mean $1/p$ for the success probability p of one attempt. *Wald's identity* then gives

$$\mathbb{E}\Big[\sum_{i=1}^{N} X_i\Big] \;=\; \mathbb{E}[N]\;\mathbb{E}[X_1].$$

CHAPTER 3

Related Work

Minimal perfect hashing has extensive history; the literature of the last years is especially large. A recent survey [23] gives a nice overview, and we will follow its classification of the approaches. We start with theoretical foundations, describe the different main families of construction approaches and discuss in more detail the family that our construction belongs to, *bucket placement*. We also briefly go over k-perfect hashing, solver-based approaches, and finally how this thesis relates to the existing work.

3.1 Foundations

Perfect hash functions were introduced in order to obtain static dictionaries with constant worst-case lookup time. In the work of Fredman and Komlós [7], they store a set of n keys in $O(n)$ words with constant worst-case access. Showing how many bits a perfect hash function needs is done via counting arguments. Mehlhorn [26] shows that a minimal perfect hash function needs $n \log_2 e - O(\log n) \approx 1.44\, n$ bits. This holds independently of how the function is constructed (Section 2.3). On the upper bound side, Hagerup and Tholey [15] give a construction with space of $(1 + o(1))$ times this lower bound, linear construction time as well as constant query time.

Why practice looks different. The lower bound states that $\Theta(1)$ bits per key are necessary and sufficient. However, it does not state how we can find such a function. A random function on n keys is a bijection with probability about e^{-n} (Section 2.5). Thus, any approach that chooses its function from a family of at most 2^s candidates needs $s \gtrsim n \log e$. The various practical approaches differ in how they split this search into pieces that are small enough to be solved: by building on a system of equations, by layering, by brute force on small sets, or by a search per bucket.

3.2 Families of Constructions

The survey groups modern constructions into three classes [23]: perfect hashing *through retrieval*, through *fingerprinting*, and through *brute force*. The last class contains the bucket-placement strategy that this thesis builds upon. We therefore treat it separately in Section 3.3.

3.2.1 Through Retrieval

Retrieval based strategies are the oldest. Here, every key gets mapped to several positions of a table by independent hash functions. This defines a random graph or hypergraph on the table positions with the keys as edges. If the structure is sparse enough, it can be *peeled*. Values are then assigned to the vertices such that, for every key, a simple combination of the values at its vertices (a sum modulo n, or an index of one of its vertices) is the key's rank. The scheme of Czech et al. [4], which was generalised to hypergraps by Majewski et al. [25], stores the hash values themselves and thus needs $\Theta(\log n)$ bits per key. There also exists a *multiple-choice* variant storing only *which* of a small number of candidate positions a key uses. This brings the space to a small constant number of bits per key. Examples are Botelho et al. [3, 2], which use 3-hypergraphs, and the faster construction of Genuzio et al. [9]. SicHash [20] uses small irregular cuckoo tables. These schemes are fast and robust, with a space of around two bits per key.

In our approach, the structure to be found is also a *choice*: a small value per key or bucket from which the position follows. In the retrieval approach, peeling resolves the choice, i.e. a linear-time elimination that exists as long as the graph has no dense subgraph. There is no such elimination order in our setting as the buckets interact through slots in an arbitrary way. We solve the choice by search (Sections 4.2.3 and 4.3).

3.2.2 Through Fingerprinting

Fingerprinting schemes work layer by layer as follows: they hash all keys into a bit array, keep the keys that are alone in their position whilst re-hashing the colliding ones into a further array in the next layer. A query probes the layers until the key's bit is set. BBHash [24] is the best known representative, reaching about e bits per key [23] with the construction being simple and parallel. Later variants reduce the space further. Fingerprinting doesn't store per-key search results. Nevertheless, the approach is still relevant to us, since the same idea is used in order to handle keys that cannot be placed.

3.2.3 Through Brute Force and Recursive Splitting

Brute force. The simplest way to reach the lower bound is to draw random seeds until the hash function with that seed happens to be bijective on the n keys. The method meets the lower bound but it is applicable only to very small sets.

Splitting into small sets. RecSplit [5] makes brute force work in practice by first splitting the key set recursively into small buckets and then into leaves of a few keys, which are then solved by brute force. The seeds of the leaves and the split information are ultimately stored with careful encoding. The approach reaches 1.56 bits per key, which is within $8.3\,\%$ of the lower bound, and the construction takes less than $2\,\text{ms}$ per key. Increasing the leaf size trades construction time for space. CONSENSUS [22] combines the search for the seeds with the existing encoding and reaches 1.444 bits per key, with a construction time that grows only like n/ε for a space overhead of ε instead of exponentially in $1/\varepsilon$ as for RecSplit.

ShockHash. ShockHash [21] employs a better-structured search in place of brute-force: it draws two hash functions h_0, h_1 and looks for an assignment $f\colon S \to \{0,1\}$ such that $x \mapsto h_{f(x)}(x)$ is a minimal perfect hash function. Considering a graph setting, every key is an edge between its two positions and the pursued assignment exists if and only if the random graph is a pseudoforest. If this is the case, cuckoo hashing yields the assignment. The expected number of trials is $(e/2)^n$ up to polynomial factors instead of e^n, and the assignment f is stored in one bit per key by a retrieval structure. Used inside the RecSplit framework, it constructs functions with 1.489 bits per key for 10^7 keys in about half an hour.

All schemes in this subsection obtain their space efficiency from a *search*. We also search, however, at a different scale: our unit is a bucket with its seed, and the solutions of the buckets must fit together.

3.3 Bucket Placement

The scheme. All approaches of this family share the following structure, which is also the structure of our construction (Section 4.2). The keys are first mapped to buckets by a hash function. Every bucket stores a small value, called a *displacement*, *pilot* or *seed*, which is combined with a hash of every key of the bucket to give the key's position in the table. During construction, buckets are processed one after another: one looks for a value such that all keys of the bucket land on positions that are still free. The query first computes the bucket of the key, looks up the corresponding value and hashes. The space is measured by the cost of the values: for $m = n/\lambda$ buckets of expected size λ and values with b_v bits on average, we get b_v/λ bits per key, which directly corresponds $\log \sigma/\lambda$ of Section 4.5.2 for fixed-width values. Increasing the average elements per bucket λ reduces the space per key, but a value that fits λ keys at once becomes rarer. Thus, the search takes longer and the values become larger. Table 3.1 summarises the space of these constructions. Our approach lies in the range of the pilot- and seed-based schemes such as PtrHash, whereas the schemes that come closest to the lower bound, RecSplit and ShockHash-RS, pay for it with much longer construction times.

Lineage. *Hash and displace* of Pagh [27] first introduced the idea of choosing a displacement per bucket, handling buckets in decreasing order of their size, obtaining a minimal perfect hash function with constant evaluation time overall. Fox et al. [6] (FCH) chose a non-uniform bucket distribution: a large share of the keys is put into a small share of the buckets. They aim to place the large buckets first, while the table is still mostly empty, and the many small buckets last, using the remaining free slots with little search. The scheme of Belazzougui et al. [1] (CHD) once again returns to uniformly distributed buckets, processing them by size, and compressing the sequence of displacements. For a table of size $1.01n$ it reaches about 1.98 bits per key. CHD is slow to query because the variable-length codes have to be decoded. The authors also state that the method extends to k-perfect hashing.

PTHash [28] combines the query speed of FCH with the space efficiency and fast construction of CHD. It uses the skewed FCH distribution of keys (about $60\,\%$ of the keys in about $30\,\%$ of the buckets), processes the buckets in decreasing size, and for every bucket takes the first pilot $0, 1, 2, \ldots$ for which all keys of the bucket hash to free positions. In order to avoid a long tail of searches when the table is almost full, PTHash searches in a table of $n' = n/\alpha$ positions for a load factor $\alpha < 1$. Keys that land on positions $\geq n$ are remapped to free positions below n via a compactly stored array. Small pilots are likely early on, making the pilot table very compressible, and several encoders trade query time against space.

PHOBIC [16] improves PTHash in multiple steps. The heuristic bucket distribution is replaced by one that is argued to be asymptotically optimal, a bucket assignment function $\beta^*(x) = x + (1 - x)\ln(1 - x)$ of the normalised hash value $x \in [0, 1)$ of a key. At the same time, it uses much smaller partitions and an interleaved coding of the pilots: the i-th bucket of every partition shares an encoder, so that every encoder is adapted to the pilot distribution of its bucket position. It saves about 0.17 bits per key compared to PTHash at the same query time and construction throughput.

PtrHash [10] builds on PTHash and PHOBIC, aiming at construction and query speed. The pilots are now stored with a *fixed width of 8 bits*, finding them with a construction close to cuckoo hashing (a bucket that cannot be placed evicts others). Every part has the same number of buckets and slots so that no prefix sums are needed. Additionally, keys hashed to positions above n are remapped with a single table and queries are answered in batches with prefetching. It uses 2.0 bits per key with the default parameters, and answers queries in $12\,\text{ns}$ per key for 10^9 integer keys.

Design dimensions. Within the family, the schemes differ in the following aspects, and we will refer to them when we describe our own choices.

- *Bucket sizes.* Uniform buckets (CHD), a heuristic skew towards a few large buckets (FCH, PTHash), or a skew derived from an analysis (PHOBIC).
- *Displacement function.* Additive (FCH, PHOBIC) or XOR-based (PTHash, and ours).

Method	Bits per key	Remarks (as reported by the authors; settings differ)
Lower bound	1.443	counting argument (Section 2.3)
CHD	1.98	table of size $1.01n$; ≈ 1.4 for $1.23n$ [1]
PTHash	2.17 to 3.23	10^9 keys; 101 ns to 37 ns per query [28]
PtrHash	2.0	fixed 8-bit pilots, default parameters [10]
RecSplit	1.56	less than 2 ms construction per key [5]
ShockHash-RS	1.489	10^7 keys in about half an hour [21]

Table 3.1: Space of selected constructions for minimal perfect hashing.

- *Search order and fallback.* Sequential first-fit in a static order by bucket size (CHD, PTHash, PHOBIC), or with cuckoo-style eviction of already placed buckets (PtrHash). Placement is greedy, with at most a local repair; if no pilot is found, the load factor is lowered or a bucket is moved.
- *Load factor and remapping.* A table larger than n with a remap of the overflowing positions (PTHash, PHOBIC, PtrHash) avoids the costly final phase in which the last free slots have to be hit.
- *Encoding of the values.* Variable-length codes of the pilots with encoders adapted to their distribution (CHD, PTHash, PHOBIC) versus a fixed width (PtrHash, and ours).
- *Partitioning.* One global table, partitions of a few thousand keys (PHOBIC), or equally sized partitions that avoid prefix sums (PtrHash).

3.4 k-Perfect Hashing

A k-perfect hash function allows for up to k keys per output value, and is minimal if the range has size $\lceil n/k \rceil$. The lower bound for the minimal case drops quickly with k: from 1.443 bits per key for $k = 1$ to 0.589 for $k = 4$ and 0.300 for $k = 10$ [17]. Applications are external-memory dictionaries, where an output value is the number of a page, and in-memory tables in which a cache line holds k keys. The paper of Hermann et al. [17] gives four new constructions: threshold-based bumping with fingerprints, a generalisation of bucket placement, a variant of RecSplit, and a k-perfect version of PaCHash. The generalisation of bucket placement uses an optimised bucket-assignment function derived for $k > 1$ and chooses for each bucket a seed such that no bin receives more than k keys. Some constructions need prefix sums, others avoid them. For the non-minimal case with a load factor below one and $k \geq 2$, Groot Koerkamp et al. [11] prove a tight lower bound whilst extending PtrHash to a k-perfect version, which they use for fast static hash sets.

3.5 Solver-Based Approaches

Using general solvers for minimal perfect hashing is barely explored when compared to the schemes above. Weaver and Heule [29] encode the construction as a SAT problem, which is feasible for dictionaries of up to 40 keys. One of their constructions aims at the information-theoretic limit, while a second one reaches about 1.83 bits per key. While the sizes are tiny compared to the other schemes, the idea is closely related to ours: a combinatorial solver is used for the choice in place of a greedy procedure.

The formulation of this thesis connects to two further bodies of work, which we use in Sections 2.4 and 4.6. Exact cover and the dancing-links algorithm [19] are classical tools of combinatorial search. The independent set formulation of Section 4.2 links the problem to the heuristics and reductions for (weighted) maximum independent set; a recent solver is the concurrent local search CHILS [13], which we used as an alternative (Section 4.6).

3.6 Positioning of This Thesis

Our construction belongs to the bucket-placement family. The query path (part, bucket, seed lookup, hash with the seed), the XOR-based hash with a seed, the fixed-width values and the combination of parts all have counterparts in PTHash, PHOBIC and PtrHash, and the cost of $\lceil \log \sigma \rceil / \lambda$ bits per key is the same trade-off. The differences are in how the seeds are *found*.

- *Joint instead of greedy.* The existing schemes place the buckets one after another in a static order, with at most a local repair such as an eviction; when a bucket finds no pilot, they use a larger table or a remap. We state the choice of all seeds of a part as one combinatorial problem, an independent set in the conflict graph, or equivalently an exact cover instance (Sections 4.2 and 2.4), and solve it by backtracking search with the most constrained column first (Section 4.3). The branching order is dynamic and decisions can be undone.

- *Restarts instead of fallbacks.* When an attempt fails or takes too long, we change the hash functions and try again, with a cutoff chosen from the measured distribution of attempt costs (Section 4.4).

- *Target load factor.* The exact-cover view covers $\alpha = 1$ directly, without a remap table, at the price of harder instances; for $\alpha < 1$ the slots are secondary columns.

To our knowledge, the global exact-cover formulation of the seed choice, with search and restarts, has not been used in the cited work.

CHAPTER 4

Our Construction

4.1 Overview

Our construction rests on the following idea: instead of searching for a single hash function that is injective on the whole key set, we fix a large family of cheap candidate functions and choose from it, bucket by bucket, so that the choices fit together in the end. We only store the choices, taking only a few bits per bucket.

The pipeline has six steps, see Figure 4.1.

1. *Parts.* An unseeded hash function distributes the keys to P independent parts. Parts keep every later step small, can be solved in parallel, and can be retried independently (Section 4.5).

2. *Buckets.* Within a part, the keys are grouped into buckets of about λ keys.

3. *Candidates.* For every bucket and every seed from a set of σ seeds, we compute where the bucket's keys would go. A seed is a *candidate* for a bucket if the keys do not collide with each other (Section 4.2).

4. *Choice.* We must pick one candidate per bucket such that no two chosen candidates use the same slot. This is an independent set problem in a conflict graph, which can be stated equivalently as an exact cover problem (Sections 4.2 and 4.2.3).

5. *Search.* We solve it with Knuth's dancing links under a search limit. If an attempt fails, we change the hash functions (a *re-roll*) and try again (Sections 4.3 and 4.4).

6. *Storage.* We store one seed per bucket, a few numbers per part, and for each part the index of the successful attempt. A query computes the part, the bucket, looks up the seed, and hashes the key (Section 4.5).

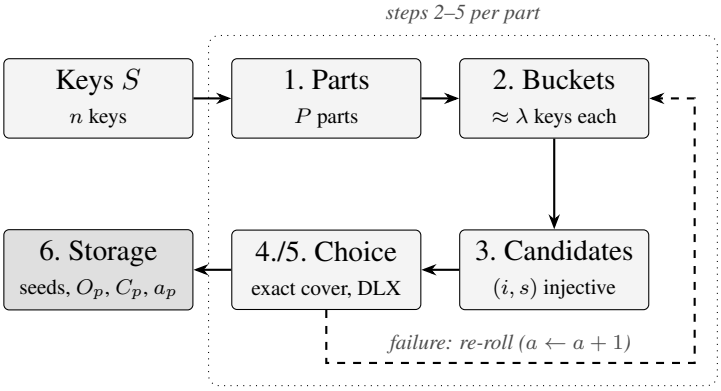


Figure 4.1: The construction pipeline. Steps 2–5 run independently for every part. If DLX fails on a part, only that part is re-rolled.

The core trade-off. Storing one seed costs $\log \sigma$ bits and covers λ keys. Hence, the structure takes about $\log \sigma / \lambda$ bits per key. While large buckets and few seeds are cheap in space, they make conflict-free choices rarer and the search harder. Each minimal perfect hash function on n keys needs at least $\log_2(n^n/n!)$ bits in total, which is about $n \log_2 e$ bits, or $\log_2 e \approx 1.443$ bits per key (Section 2.3). The question of this thesis thus becomes how close we can get to this lower bound via a search-based construction in practice, while keeping construction and query time as low as possible.

Reading guide. Sections 4.2 to 4.2.4 describe the problem and what to expect from it, Sections 4.3 and 4.4 the solver and the handling of failures, Section 4.5 the assembly of the final data structure and its space, and Section 4.6 the alternatives that we tried and abandoned. Definitions that are standard (perfect hashing, exact cover, dancing links, the lower bound, probability tools) are collected in Chapter 2.

4.2 Problem Formulation

We start by fixing notation for a single part and showing how we construct a perfect hash function by finding an independent set on a conflict graph. How the various parts are combined is the topic of Section 4.5.

4.2.1 Setting and Notation

Let $S \subseteq U$ be the set of keys of one part, $|S| = n$, and let $R = \lceil n/\alpha \rceil$ for a load factor $\alpha \in (0, 1]$. We construct a perfect hash function $f\colon S \to \{0, \ldots, R-1\}$ (Section 2.2); for $\alpha = 1$ its is minimal. In Sections 4.2 to 4.4, S, n, m and R refer to a single part. In Section 4.5, where several parts are combined, we index them by p (S_p, n_p, m_p, R_p), and n and m denote totals.

Buckets. A bucket function $\beta\colon S \to \{0, \ldots, m-1\}$ with $m = \lceil n/\lambda \rceil$ partitions S into m buckets $B_0, \ldots, B_{m-1}$ of about λ keys each, so $m \approx n/\lambda$. Since β is a hash function, the buckets have λ keys on average, but their sizes vary.

Seeds. Let $\Sigma = \{0, \ldots, \sigma - 1\}$ be a set of σ *seeds*. Each seed $s \in \Sigma$ defines a hash function $h_s\colon U \to \{0, \ldots, R-1\}$ with

$$h_s(x) = \big(\mathrm{fx}(x) \oplus \mathrm{fx}(s \oplus g)\big) \bmod R,$$

where fx is a multiplicative mixing function, $\oplus$ is the bitwise XOR operator and g a global seed. For a bucket B we write $h_s(B) = \{h_s(x) : x \in B\}$.

Building the perfect hash function. A *seed assignment* is a map $\pi\colon \{0, \ldots, m-1\} \to \Sigma$ which selects exactly one seed per bucket. It defines

$$f_\pi(x) = h_{\pi(\beta(x))}(x).$$

In order to evaluate f_π it takes one bucket computation, one lookup of the stored seed and one hash evaluation. The data structure only stores π; i.e. it stores one seed value for each of the buckets. The task is to find an assignment π for which f_π is injective.

4.2.2 The Conflict Graph

We may only use a seed for a bucket if it doesn't cause a collision inside this bucket. The pair (i, s) of a bucket index i and a seed s is called a *candidate* if h_s is injective on B_i.

Definition 3 (Conflict graph)
Let $V = \{(i, s) \in \{0, \ldots, m-1\} \times \Sigma : h_s$ is injective on $B_i\}$ be the set of candidates. The conflict graph $G = (V, E)$ *has the edge set $E = E_{\text{bucket}} \cup E_{\text{slot}}$ with*

$$E_{\text{bucket}} = \big\{\{(i, s), (i, s')\} : s \neq s'\big\},$$
$$E_{\text{slot}} = \big\{\{(i, s), (j, s')\} : i \neq j,\ h_s(B_i) \cap h_{s'}(B_j) \neq \emptyset\big\}.$$

There is a conflict between two candidates either if they select different seeds for the same bucket (bucket edges), or if they would place keys of different buckets into the same hash slot (slot edges). We have $|V| \leq m\sigma$. Obtaining the bucket edges is trivial as the candidates of one bucket form a clique. Obtaining the slot edges efficiently is done as follows: For every slot list all candidates that map a key to it. We now get a slot edge for every pair of candidates from a different bucket.

Theorem 2
Let $I \subseteq V$ be an independent set of a conflict graph G. Then

1. *I contains at most one candidate per bucket, hence $|I| \leq m$;*

2. *if $|I| = m$, then $I = \{(i, \pi(i)) : i = 0, \ldots, m-1\}$ for a seed assignment π, and f_π is injective;*

3. *conversely, if π is a seed assignment for which f_π is injective, then $\{(i, \pi(i)) : i = 0, \ldots, m-1\}$ is an independent set of size m in G.*

Proof. (1) Any two candidates of the same bucket are adjacent due to the bucket edges, so an independent set contains at most one candidate for each bucket.

(2) By (1), an independent set of size m contains exactly one candidate $(i, \pi(i))$ for every bucket i, which defines π. Let $x \neq y$ be keys with $f_\pi(x) = f_\pi(y)$. If $\beta(x) = \beta(y) = i$, then $h_{\pi(i)}$ is not injective on B_i, contradicting that $(i, \pi(i))$ is a candidate. If $\beta(x) = i \neq j = \beta(y)$, the slot $f_\pi(x)$ lies in $h_{\pi(i)}(B_i) \cap h_{\pi(j)}(B_j)$, so $(i, \pi(i))$ and $(j, \pi(j))$ share a slot edge, contradicting that I is an independent set.

(3) If f_π is injective, each $h_{\pi(i)}$ is injective on B_i, so $(i, \pi(i)) \in V$. Two chosen candidates $(i, \pi(i)), (j, \pi(j))$ with $i \neq j$ sharing a slot would give two different keys with the same hash value, contradicting injectivity. They share no bucket edge either, since the buckets differ. □

Theorem 3
Given buckets $B_0, \ldots, B_{m-1}$ and seed set Σ, an injective function of the form f_π exists if and only if the maximum independent set of the conflict graph G has size m. Every maximum independent set of that size produces such a function.

Proof. Immediate from Theorem 2. □

Constructing a perfect hash function in this way is a special case of the maximum independent set problem, which is NP-hard in general (Section 2.4). The construction yields a highly structured graph: the vertices are partitioned into m cliques and any edges between cliques arise from slot collisions. The next section exploits this structure.

4.2.3 From Independent Set to Exact Cover

The conflict graph describes the problem *negatively*: its edges list the pairs of candidates that may not be chosen together. We obtain a solution by avoiding all forbidden combinations. For every candidate that is considered at any step, we have to look up whether it conflicts with the candidates chosen so far. We can instead formulate the problem *positively*, by saying what each candidate *uses*, in order to build valid combinations directly.

Let the ground set U_G consist of one element b_i per bucket and one element z_j per slot,

$$U_G = \{b_0, \ldots, b_{m-1}\} \dot\cup \{z_0, \ldots, z_{R-1}\},$$

which are all distinct, and let the candidate (i, s) stand for the set

$$A_{(i,s)} = \{b_i\} \cup \{z_j : j \in h_s(B_i)\},$$

that is, the bucket it serves and the slots it occupies. Any two candidates are adjacent in the conflict graph G exactly if their sets intersect: they either share the bucket element (bucket edge) or a slot element (slot edge). Hence, G is the intersection graph of the sets $A_{(i,s)}$. An independent set of size m is a family of m pairwise disjoint sets that covers each bucket exactly once and each slot at most once.

For $\alpha = 1$ we have $R = n$ and the chosen sets cover $m + n$ elements in total, which is the size of the whole ground set. Every bucket and every slot is therefore covered exactly once, so the problem is an *exact cover* (Section 2.4). For $\alpha < 1$ some slots stay unused. In this case, buckets are then primary columns which have to be covered exactly once, while slots are secondary columns that may be covered at most once.

The positive view simplifies the search in two different ways.

- *No conflict checks.* When selecting a candidate, every candidate that shares one of its elements can be removed immediately. Thus, we skip the pairwise conflict check against earlier choices. We also don't need to build the conflict graph G, which can become much larger than the instance itself.

- *Points of choice.* We obtain a point of choice for each element: it is natural to branch on buckets or slots with few remaining candidates. An element with no remaining candidate signals a dead end. By branching on the element with the fewest remaining candidates we detect infeasible partial choices early.

This is the core idea behind Knuth's dancing-links algorithm [19], which the solver of Section 4.3 implements.

4.2.4 How Many Solutions to Expect

Before turning to the solver, we briefly want to look at for which parameters a solution can be expected at all. This directly depends on the number of candidates per bucket. We assume the random hash model of Section 2.2: every h_s behaves like an independent uniformly random function.

By Theorem 1, a seed is a candidate for a bucket of k keys, i.e. h_s is injective on it, with probability

$$\Pr[h_s \text{ injective on } B_i] \;=\; \prod_{j=0}^{k-1}\Big(1 - \frac{j}{R}\Big) \;\geq\; 1 - \frac{k(k-1)}{2R}.$$

For $R \approx n \gg \lambda^2$ almost every seed is a candidate for a bucket of average size. The expected number of candidates per bucket is therefore close to σ. Larger buckets are much more likely to have few or no candidates. If a single bucket has no candidate at all, the instance has no solution. This is why the implementation re-rolls the hash functions and the bucket assignment after a failed attempt (Section 4.4).

4.3 Solving the Exact Cover Instance

Section 4.2.3 describes the reduction of constructing a perfect hash function to an exact cover problem. We solve it with Knuth's Algorithm X, implemented with the dancing-links data structure [19], which we refer to as DLX. Both are introduced in Section 2.4. Here, we describe how they are applied to our instances: the instance layout, the secondary columns for $\alpha < 1$, the branching rule with its tie-breaking variants, and the search limits.

Instance layout. For a part with m buckets and range R, the matrix has $m+R$ columns. The first m are bucket columns, followed by the slot columns. Each candidate (i, s) is a row with $1 + |B_i|$ nodes (the bucket and the slots of its keys). With up to σ candidates per bucket, the matrix has at most

$$\sum_{i=0}^{m-1} \sigma\,(1 + |B_i|) \;=\; \sigma\,(n+m) \;\approx\; \sigma\,n\Big(1 + \frac{1}{\lambda}\Big)$$

nodes.

Secondary columns. For $\alpha = 1$ all columns are primary, meaning every slot has to be covered exactly once. For $\alpha < 1$, only the m bucket columns are primary and the slot columns become secondary. We don't add a secondary header in the header list through the root, thus never selecting it for branching. Its rows are still covered and uncovered like before. This enforces that every slot is used by at most one candidate without requiring that it is used at all.

Branching rule. In each node of the search tree, we choose the primary column with the fewest remaining rows, as proposed for Algorithm X by Knuth [19]. Any column with a single remaining row forces the choice of that row: for a bucket column, this is the only seed left for this bucket; for a slot column, only one candidate can occupy this slot. A column without any remaining row signals a dead end, so we can backtrack immediately. Selecting a column takes $O(c)$ time, where c is the number of primary columns, i.e. $c = m + R$ for $\alpha = 1$ and $c = m$ for $\alpha < 1$.

Tie-breaking. Often times multiple columns have the same minimal size. We implement five rules for choosing among them: `first` and `last` column in header order, a uniformly `random` column (reservoir sampling, seeded), and `bucket` or `slot`, which prefer a tied bucket column over a tied slot column or the reverse. The choice changes the shape of the search tree and is experimentally evaluated in Section 5.6.

Table 4.1: Slots $h_s(B_i)$ of the example. An entry marked with $\times$ is not a candidate, because both keys of the bucket get the same slot.

	$s = 0$	$s = 1$	$s = 2$	$s = 3$
$B_0 = \{a, b\}$	$\{2, 4\}$	$\{1, 1\}^{\times}$	$\{0, 3\}$	$\{0, 5\}$
$B_1 = \{c, d\}$	$\{1, 4\}$	$\{0, 3\}$	$\{2, 2\}^{\times}$	$\{3, 5\}$
$B_2 = \{e, f\}$	$\{1, 2\}$	$\{0, 4\}$	$\{3, 3\}^{\times}$	$\{2, 5\}$

Search limits. In the worst case, backtracking search can take exponentially many steps, which is why every run is limited. The search counts the visited nodes of the search tree and stops after a given limit. Optionally, one may use a wall-clock time limit instead. When a limit is reached, the search returns immediately with failure. The limits are what make *restarts* possible, which are described in the Section 4.4.

4.3.1 A Worked Example

We illustrate the construction on a tiny part with $n = 6$ keys $a, \ldots, f$, load factor $\alpha = 1$ (so $R = 6$), bucket size $\lambda = 2$ and $\sigma = 4$ seeds. The bucket function gives $B_0 = \{a, b\}$, $B_1 = \{c, d\}$ and $B_2 = \{e, f\}$. The slots that each seed assigns to the two keys of a bucket are shown in Table 4.1. The values are chosen by hand for illustration and are not the output of the actual hash function.

Candidates and conflict graph. Three seeds are not injective, so $|V| = 9$ of the $m\sigma = 12$ pairs are candidates. The conflict graph has 9 bucket edges (three triangles) and 15 slot edges. For example, $(0, 0)$ with slots $\{2, 4\}$ and $(1, 0)$ with slots $\{1, 4\}$ share slot 4 and are adjacent.

Exact cover. Table 4.2 shows the instance as a matrix. The ground set consists of the buckets b_0, b_1, b_2 and the slots $z_0, \ldots, z_5$, and every candidate (i, s) is the row with a 1 in the column of its bucket and in the columns of its slots, e.g. $(0, 2) \mapsto \{b_0, z_0, z_3\}$. Two candidates conflict exactly if their rows have a 1 in the same column. A solution is a set of rows such that every column contains exactly one 1.

DLX branches on a column with the fewest remaining rows. Column z_1 has only two, $(1, 0)$ and $(2, 0)$, so one of them must be chosen.

- Choosing $(1, 0)$ covers b_1, z_1 and z_4 and removes all rows that have a 1 in one of these columns, namely $(1, 1)$, $(1, 3)$, $(2, 0)$, $(0, 0)$ and $(2, 1)$. Only $(0, 2)$, $(0, 3)$ and $(2, 3)$ remain. Column z_2 now contains only $(2, 3)$, so this row is forced. It covers b_2, z_2 and z_5 and removes $(0, 3)$. Then $(0, 2)$ is the only remaining row of bucket 0, and it covers the last columns b_0, z_0 and z_3.

Table 4.2: The example as an exact cover instance. Each row is a candidate (i, s), each column a bucket b_i or a slot z_j, and a 1 means that the candidate uses it. Two candidates conflict exactly if their rows have a 1 in the same column. The highlighted rows form the solution: every column contains exactly one 1 among them.

	b_0	b_1	b_2	z_0	z_1	z_2	z_3	z_4	z_5
$(0,0)$	1	·	·	·	·	1	·	1	·
$(0,2)$	1	·	·	1	·	·	1	·	·
$(0,3)$	1	·	·	1	·	·	·	·	1
$(1,0)$	·	1	·	·	1	·	·	1	·
$(1,1)$	·	1	·	1	·	·	1	·	·
$(1,3)$	·	1	·	·	·	·	1	·	1
$(2,0)$	·	·	1	·	1	1	·	·	·
$(2,1)$	·	·	1	1	·	·	·	1	·
$(2,3)$	·	·	1	·	·	1	·	·	1

- Choosing $(2,0)$ covers b_2, z_1 and z_2 and removes $(2,1)$, $(2,3)$, $(1,0)$ and $(0,0)$. Column z_4 had 1s only in $(0,0)$, $(1,0)$ and $(2,1)$, so it is now empty. No solution can cover it, and the search backtracks.

The only solution is therefore $\pi = (2,0,3)$ with slots $\{0,3\}$, $\{1,4\}$ and $\{2,5\}$, which is a bijection to $\{0,\dots,5\}$. These are the highlighted rows in Table 4.2.

Storage and query. The structure stores three seeds with $\lceil \log_2 4 \rceil = 2$ bits each (6 bits for 6 keys, i.e. 1 bit per key), plus the part constants. To query key $d \in B_1$, we compute its bucket 1, read $\pi(1) = 0$ and return $h_0(d)$, which is slot 1 or 4 according to Table 4.1.

4.4 Re-rolls and Restarts

Any single instance, i.e. a fixed assignment of keys to buckets and a fixed family of hash functions, may be unsolvable or too hard to solve within a reasonable time. Instead of searching for a solution in vain, we change the hash functions and the bucket assignment after a failed attempt and try again. We call this a *re-roll*. This section explains why attempts fail, how we re-roll the instance, and how good cutoffs can be chosen.

4.4.1 Why an Attempt Can Fail

An attempt on one part ending without a solution boils down to one of the following reasons:

1. *Empty bucket.* If all σ seeds of a bucket produce a collision inside the bucket, there is no candidate for this bucket and the instance is unsolvable. This is unlikely for buckets of average size but may happen for larger buckets.

2. *Infeasible instance.* There are candidates for every bucket, but there is no combination of candidates that avoids all collisions. The search then ends after exhausting the whole search tree, proving that the instance is unsolvable.

3. *Search limit reached.* A solution may exist, but DLX does not find it within the given limit and the attempt is aborted.

Whether the second case is frequent depends on the parameters: the closer we get to the lower bound of $\log_2 e \approx 1.443$ bits per key, the more likely it is that no solution exists. The chance of success is a property of the random draws of the hash functions and the bucket assignment, not of the key set. Hence, a new draw has a fresh chance of success, which is the idea behind re-rolling.

4.4.2 Defining Re-rolls

Each part keeps track of an *attempt index* $a = 0, 1, 2, \ldots$, which determines two independent re-rolls.

Slot re-roll. The slot hash explained in Section 4.2 obtains an additional argument a that is mixed into the key as follows:

$$h_s^{(a)}(x) = \big(\mathrm{fx}(x \oplus \mu(a)) \oplus \mathrm{fx}(s \oplus g)\big) \bmod R,$$

where $\mu(a)$ is obtained by multiplying a with a fixed odd 64-bit constant modulo 2^{64}. Multiplying by an odd constant is a bijection on 64-bit values, so different values of a give different mixes, and these are spread over the whole 64-bit range. A new value of a changes the hash values of all keys, and therefore all candidates and conflicts, resulting in a new instance. For $a = 0$ we have $\mu(0) = 0$, recovering the original hash function.

Bucket re-roll. The bucket of a key x in a part with m buckets is computed from the hash $h^{(a)}(x) = \mathrm{fx}(x \oplus \nu(a))$ in two steps. First, we take the low bits of the product with the number of parts,

$$u^{(a)}(x) = \frac{\mathrm{lo64}\big(P,\, h^{(a)}(x)\big)}{2^{64}} \in [0, 1),$$

which is the fractional part of $P\, h^{(a)}(x)/2^{64}$. For $a = 0$ the high bits of the same product choose the part, so this value uses the part of the hash that the part index does not use. Then the bucket is

$$\beta^{(a)}(x) = \big\lfloor m \cdot u^{(a)}(x) \big\rfloor \in \{0, \ldots, m-1\}.$$

Algorithm 2: SOLVEPART: attempts with re-rolls for one part.

Input : keys S_p of the part, parameters σ, λ, α, limits, maximum number of attempts
Output: attempt index a and seeds π_p, or failure
$R \leftarrow \lceil |S_p|/\alpha \rceil$;
for $a = 0, 1, \ldots$ *up to the maximum number of attempts* **do**
  split S_p into buckets $B_0, \ldots, B_{m_p-1}$ using the bucket re-roll a;
  $V \leftarrow \{(i, s) : h_s^{(a)}$ is injective on $B_i\}$;
  if *some bucket has no candidate in* V ***or*** $R < |S_p|$ **then**
    continue;
  run DLX on the instance given by V, with node and time limit;
  if *DLX found a solution* **then**
    return (a, π_p);
return failure;

Here, $\nu(a)$ is a multiplicative mix of a similar to $\mu(a)$. It uses a different constant than μ, so that the bucket assignment and the slot hashes change in an unrelated way. Optionally, a monotone function $l\colon [0, 1) \to [0, 1)$ is applied to $u^{(a)}(x)$ before the second step, which makes bucket sizes non-uniform. The default is the identity. A bucket re-roll changes which keys are assigned to which buckets, and therefore the bucket sizes. For $a = 0$ we have $\nu(0) = 0$, which recovers the original bucket assignment β.

The part is never re-rolled. The part of a key comes from the unseeded hash $\mathrm{hi64}(P, \mathrm{fx}(x))$ and does not depend on a. A query has to find the part first in order to look up the attempt index stored for it, so the part cannot depend on that index.

4.4.3 The Attempt Loop

Algorithm 2 shows how one part is solved. Parts are independent, so this loop runs once per part.

Every attempt splits the part into buckets and builds the candidate set V for the current re-roll value, which takes $O(\sigma n_p)$ hash evaluations, and then runs DLX (Section 4.3). The outcome of an attempt is one of the following.

- *Solved:* DLX found a solution. The attempt index and the seeds are stored.
- *Infeasible:* the search tree was exhausted without a solution.
- *Aborted:* the node limit or the time limit of the attempt was reached.

All other outcomes lead to the next attempt. The loop ends when an attempt is solved, when the total time budget of the part is exhausted, or after a maximum number of attempts. Each attempt is limited by its node limit and its time limit, where the latter is additionally capped by the time that remains in the total budget of the part.

4.4.4 Re-roll Costs

For every part we only need to store the attempt index a_p of the corresponding successful attempt. With w_a bits for an attempt index, the cost of all parts is $P\,w_a$ bits. This is cheap in space as an index of w_a bits allows for up to 2^{w_a} attempts per part, each additional bit doubling the number of attempts that can be used.

4.4.5 Choosing a Cutoff

Running a single attempt for a very long time may often be worse than abandoning it and starting anew. Whether this is worth doing depends on the distribution of the attempt cost. If the distribution is heavy-tailed, i.e. if most attempts are cheap but some take very long, restarts are more effective.

4.5 Parts, Query and Space

In the previous section we described how a single part is solved. This section explains why we split the keys into parts, how the independently solved parts form one overall function and how many bits we need.

Notation. The key set is S with $|S| = n$ keys. It is split into P parts $S_0, \ldots, S_{P-1}$ with $n_p = |S_p|$ keys, range $R_p = \lceil n_p/\alpha \rceil$ and $m_p = \lceil n_p/\lambda \rceil$ buckets, so $\sum_p n_p = n$. We write $m = \sum_p m_p \leq n/\lambda + P$ for the total number of buckets.

4.5.1 Why Split into Parts

We distribute the keys into P parts via an unseeded hash function. Every part is solved on its own with range R_p and m_p buckets. This has multiple advantages:

- *Smaller instances.* The size of the matrix is about $\sigma(n_p + m_p)$ nodes (Section 4.3). The part size bounds the memory and time spent on building an instance.
- *Parallelism.* Parts are independent and can be solved concurrently.
- *Local retries.* A failed attempt (Section 4.4) only needs to repeat the work of one part, instead of the work of the whole key set.

At the same time, this approach introduces new costs. Since part sizes are random, the ranges have to be stored as prefix sums. Every part therefore gets its own overhead of a few numbers.

4.5.2 Combining the Parts

What is stored.

- the seed $\pi_p(i) \in \Sigma$ of every bucket i of every part p;
- the prefix sums O_p and C_p for $p = 0, \ldots, P$ (defined below), from which the range R_p and the number of buckets m_p of each part follow as consecutive differences;
- for every part p the index a_p of the successful attempt, from which the bucket re-roll and the slot re-roll follow (Section 4.4);
- global constants: n, P, λ, σ, and the global seed g.

Query. Every part is solved on its own with local slots $\{0, \ldots, R_p - 1\}$. To obtain one function, the local slots of part p are shifted to a disjoint interval by the *prefix sum* of the ranges of all previous parts,

$$O_p = \sum_{q<p} R_q,$$

and the seeds of all parts are kept in one array, where part p starts at the prefix sum of the bucket counts

$$C_p = \sum_{q<p} m_q.$$

We can then query a key x as follows:

1. compute the part $p = \mathrm{hi64}(P, \mathrm{fx}(x))$, where hi64 is the high 64 bits of the 128-bit product of P and $\mathrm{fx}(x)$;
2. compute the bucket i of x within part p, using the bucket re-roll of attempt a_p;
3. look up the seed $s = \pi_p(i)$ from position $C_p + i$ in the seed array;
4. return $O_p + h_s(x)$, evaluated with the slot re-roll of attempt a_p and range R_p.

Figure 4.2 illustrates the query.

There are two details to consider. First, the part sizes are random since a hash function assigns about n/P keys to each part. This may fluctuate in the order of $\sqrt{n/P}$, due to which we can't compute O_p as $p \cdot n/P$ and have to store the prefix sums explicitly. Second, C_p is not simply O_p/λ: for $\alpha = 1$ we have $O_p = \sum_{q<p} n_q$, but in general $\sum_{q<p} \lceil n_q/\lambda \rceil \neq \lceil \sum_{q<p} n_q/\lambda \rceil$ because of the rounding in every part. Obtaining C_p therefore also requires a sum over all p earlier parts. Alternatively, each part can keep its own seed array.

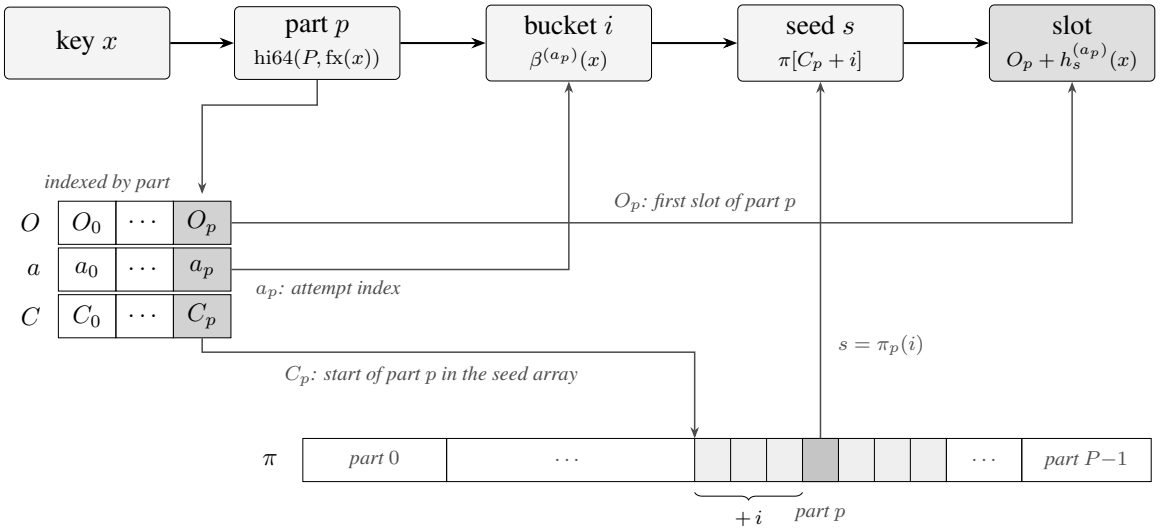


Figure 4.2: A query. The key determines its part p, which selects the entries O_p, C_p and a_p. The bucket i is computed with the re-roll a_p, the seed is read at position $C_p + i$ of the seed array, and the slot is the local hash value shifted by O_p.

Seeds. Since every seed is stored with a fixed width of $\lceil \log_2 \sigma \rceil$ bits, the seeds take exactly $m\lceil \log_2 \sigma \rceil$ bits, i.e.

$$\frac{\lceil \log_2 \sigma \rceil}{\lambda} \text{ bits per key}$$

up to the term caused by rounding the last bucket of each part. For example, $\sigma = 64$ seeds require 6 bits per bucket. Larger buckets lower the overall cost per key. However, larger buckets result in a smaller probability that a seed is a candidate and a harder search due to more collisions between buckets.

Total. Combining the above, the total space usage is

$$m\lceil \log_2 \sigma \rceil + P\left(2\lceil \log_2 n \rceil + w_a\right) \;\approx\; \frac{n\lceil \log_2 \sigma \rceil}{\lambda} + O(P \log n) \text{ bits},$$

where w_a is the number of bits to store the attempt index a_p of each part.

Smaller encodings. Fixed-width seeds spend $\lceil \log \sigma \rceil$ bits on every bucket, even if the solver prefers some seeds over others. If the distribution of the chosen seeds is skewed, an entropy-based encoding of the seed array that still supports random access can lower the space.

4.6 Discontinued Approaches

Before settling on exact cover with dancing links, we explored other approaches to solving the choice problem described in Section 4.2. We describe the approaches in the order in which we tried them. For each one we say what we expected, what happened and what we learned. In brief, we first tried general solvers for the independent set problem on large conflict graphs, then tried to make these instances easier, then relaxed the problem to non-minimal k-perfect hashing, and finally returned to minimal perfect hashing with very small parts, which is the approach adopted in this thesis.

4.6.1 Independent Set Solvers on the Conflict Graph

The project began with the idea of formulating the choice of seeds as an independent set problem. The conflict graph has one vertex per candidate, i.e. per pair of a bucket and a seed, and an edge between two candidates that cannot be chosen together; a perfect hash function corresponds to an independent set that contains one candidate per bucket. We hoped that the powerful solvers available for independent set problems would also work well for these instances and produce functions close to the lower bound. We started with key sets of the size that state-of-the-art constructions handle, about 10^5 to 10^6 keys, and used two types of solvers.

ILP (Gurobi). We formulated the problem as an integer linear program with one binary variable per candidate and solved it with Gurobi [14]. We used Gurobi version 12.0.2 with one thread per instance and otherwise default settings. The ILP has one binary variable per candidate (bucket, seed). Two kinds of constraints forbid choosing two candidates together: candidates of the same bucket are in conflict, since only one seed per bucket may be chosen, and candidates of different buckets are in conflict if at least one key of each is hashed to the same slot, which we compute in advance and pass to the ILP as constraints. We encode the first type of conflict using one clique constraint per bucket (at most one candidate may be selected per bucket), rather than pairwise constraints. This improved the solver's efficiency. Fixing the solver to use the interior point method made no difference.

Local search (CHILS). As an alternative we used the local search solver CHILS [13], which is designed for large independent set instances. CHILS clearly outperformed Gurobi on our initial runs on about 10^5 to 10^6 keys. All runs of this subsection had a time limit of $120\,\mathrm{s}$.

Results on large graphs. For $20\,000$ keys with 32 seeds and 4.0 keys per bucket the conflict graph has about $160\,000$ vertices and 13 million edges, and a valid hash function needs 5000 buckets to be placed; CHILS placed 4596. For $100\,000$ keys with 32 seeds and one key per bucket, CHILS placed $99\,616$ of the $100\,000$ buckets. With $200\,000$ keys, 4

seeds, 8 parts and 4 keys per bucket, CHILS placed 4874 of the 6250 buckets of a part, Gurobi only 3855. In none of these settings was a complete solution found.

Number of seeds. We also varied the number of seeds on 20 000 keys in 8 parts with 4 keys per bucket, where 625 buckets per part have to be placed. More seeds help, but slowly, and the goal was never reached:

seeds	4	8	16	32	64	128	256
placed buckets (of 625)	487	530	562	585	591	591	597

A load factor $\alpha = 0.9$ (a range slightly larger than the number of keys) brought the values closer by about 15 buckets, but never to 625.

4.6.2 Making the Instances Easier

Since the solvers did not find complete solutions, we tried to change the instances.

Graph reductions. Data reductions simplify an independent set instance before it is solved. We applied the reductions of the KarlsruheMIS project [13] before passing the graph to Gurobi or CHILS, on graphs of 20 000 keys with 4 to 128 seeds in 8 parts and 4 keys per bucket, expecting the reductions to produce smaller graphs and thus better solutions. For 4 seeds the original graph has 2500 vertices and 23 586 edges; after the reductions it has 2414 vertices and 47 578 edges. CHILS with the reductions placed 462 of the 625 buckets, and without them 487. The same trend was observed in the other instances, indicating that the reductions made the instances harder for us to solve.

Different distributions of the bucket sizes. Constructions such as PHOBIC and PtrHash do not distribute the keys uniformly over the buckets, and we tested whether this makes the conflict graph easier. With CHILS we compared the uniform distribution with the ones of PHOBIC, PtrHash and an x^x-shaped one. Non-uniform distributions led to larger solutions: of 5000 buckets, CHILS placed 4234 for the uniform distribution, 4554 for x^x, 4604 for the PHOBIC and 4636 for the PtrHash distribution. Still, none of them reached a complete solution, and the final construction uses the uniform distribution. We stayed with the uniform distribution as we noticed this worked best with the setup in this thesis.

An iterated approach. We also tried to keep the graphs small by solving in rounds: build the conflict graph for 64 seeds per bucket, let CHILS find a partial solution, then continue with 128 seeds for the buckets that are still unplaced, and so on until all buckets are placed. This did not work either.

4.6.3 Checking that Solutions Exist

At some point, we began to question whether the instances had any solutions at all. We checked this with PtrHash, which can build a function in the same setting. For $10\,000$ keys, 256 seeds, 3 keys per bucket, one part and $\alpha = 1.00$ it found a solution, so solutions exist. None of the independent set solvers we tested found one: neither CHILS nor Gurobi, also not with HyperMIS [12] or the data reductions before. We also constructed a small data set of instances with known solutions obtained from PtrHash and formulated them as independent set problems. The solvers failed to find these solutions as well. This was the first strong indication that general independent set solvers are not suitable for this problem. A likely explanation is that the instances are large and dense, while solutions are rare.

4.6.4 Relaxation to k-Perfect Hashing

After the realization that independent sets did not work, we looked at non-minimal k-perfect hashing: the keys are mapped to bins of capacity k with a load factor $\alpha < 1$. This is an easier problem and has a much lower bound on the space.

ILP. We used 10^6 keys, $k = 16$, $\alpha = 0.8$ and one part, where the lower bound is 0.027 bits per key. With an ILP similar to the one for the perfect hash function, all keys could be placed, but we never got closer than 0.05 bits per key, twice the lower bound.

Iterated approach. In an iterated variant, every bucket has exactly one available seed per iteration: either its keys can be placed with it, or the bucket is marked as not placeable in this iteration, and the next iteration tries a different seed per bucket. For $100\,000$ keys, one part, $k = 16$, $\alpha = 0.8$ and 10 keys per bucket, the first iteration placed 9047 of the $10\,000$ buckets and 100 iterations placed 9981. For more keys per bucket the share of placed buckets was lower: 4379 of 5000 for 20 keys per bucket, 2100 of 2500 for 40 and 1364 of 1667 for 60.

We experimented with several variations of this scheme: limiting the number of keys placed in each bin during the first iteration (for example, 12 out of 16) and allowing more keys in subsequent iterations; and increasing the number of available seeds per iteration (1, then 3, 7, and 15), both with and without increasing the bin capacities. None of these approaches managed to place all buckets, even after many iterations.

Greedy completion. The first iteration already places $90\,\%$ of the keys with one seed bit per bucket, which looked promising. We therefore tried applying a greedy method to the remaining buckets after a few iterations, increasing the seed until each bucket could be placed. This was too expensive in space: for 40 keys per bucket we place $90\,\%$ of the keys with one bit per bucket, but adding the greedy remainder only reduced the space to 0.06 bits per key, still far from the lower bound.

4.6.5 Turning to Small Parts

After finding that we could make no further progress with k-perfect hashing, we returned to minimal perfect hash functions and the approach that became the subject of this thesis. We reduced the size of the parts drastically, to about 100 keys, and first used Gurobi. This was the first time we could find solutions reliably. We then continued to approach the theoretical lower bound with these small parts, which led to the exact cover formulation and dancing links (Chapter 4).

What we took from this. Our experiments show that general solvers for the independent set problem are not suitable for this problem. On the large conflict graphs, neither the ILP solver nor the local search found complete solutions, even though PtrHash proved that solutions exist, and reductions, other bucket distributions and iterated variants did not change this. Relaxing the problem to non-minimal k-perfect hashing made the instances easier, but still did not get close to the lower bound. What made the problem tractable was reducing the instance size: with parts of about 100 keys, solutions can be found reliably. This made an exact search feasible and led to the exact cover formulation and dancing links. A smaller lesson concerns the ILP formulation: writing the constraints between the candidates of a bucket as cliques and not pairwise made the solver considerably more efficient.

CHAPTER 5

Experimental Evaluation

This chapter evaluates the construction of Chapter 4 experimentally. We follow the construction from the inside out: we start with a single part and finish with the complete data structure on up to 10^7 keys. Section 5.2 measures what the re-rolls cost in space, and Section 5.3 the success probability of a single attempt, which explains how many re-rolls are needed. Section 5.4 compares dancing links with an ILP solver on the same instances. Section 5.5 then looks at the complete construction: its space, its dependence on the size of the parts and on the number of keys, and its running time on many threads. Finally, Section 5.6 examines the parameters of the search itself, mainly the node limit per attempt and the rule that breaks ties between columns. Throughout, we are interested in two quantities: the number of bits per key, and the time that the construction needs to reach it.

5.1 Experimental Setup

All experiments of this chapter ran on one machine with an AMD EPYC 9754 processor (one socket, 128 cores, 2 hardware threads per core, i.e. 256 threads; clock up to 3.1 GHz; 1 MiB L2 cache per core, 256 MiB L3 cache) and 755 GiB of RAM under Linux 6.8. The code was compiled with g++ 13.3.0 as a CMake release build (`-O3 -DNDEBUG`, C++17, OpenMP). The keys are n distinct random 64-bit integers: we draw $1.05\,n$ values from a Mersenne Twister (`std::mt19937_64`, seeded with the seed of the run), sort them, remove duplicates (if fewer than n remain, the gaps are filled), keep n of them and shuffle them.

Time measures. We distinguish between two measures of execution time. The *wall time* is the elapsed time of the solving phase, i.e. of constructing all parts with the given number of threads; generating the keys and checking the result are not included. The *CPU time* is the sum of the solving times of all parts, measured separately for every part, so it is the total work. Where it is given per key (“CPU/key”), the CPU time is divided by the number of keys n. With a single thread, wall time and CPU time are equal. As the number of threads increases, wall time decreases, while CPU time remains approximately constant. It increases slightly with the number of threads due to contention for shared caches and memory bandwidth (Table 5.8). The wall time shows how long a construction takes, the CPU time how much work it needs and how well that work is distributed over the threads. As a machine-independent measure of the work we also count the search nodes of the dancing-links solver.

5.2 Re-roll Cost and Space on Small Instances

Our first experiment evaluates the space overhead introduced by the re-rolls described in Section 4.4. For a single part of n keys, we ask: how many attempts (re-rolls) are needed

so that all instances of a configuration are solved? At the same time we want to look at how many bits per key the structure then uses together with the bits for the attempt index. We compare these results against the information-theoretic lower bound $\alpha_n = \frac{1}{n}\log_2 \frac{n^n}{n!}$ introduced in Section 2.3.

Setup. We use $P = 1$ part, load factor $\alpha = 1$ (minimal), and $n \in \{40, 50, 60\}$ random 64-bit keys. A configuration is given by the number b of bits per bucket, so $\sigma = 2^b$ seeds with $b \in \{2, \ldots, 6\}$, and by the number of keys per bucket λ, which determines the number of buckets $m = \lceil n/\lambda \rceil$. The seeds then take

$$x = \frac{m\,b}{n}$$

bits per key. We use x as the horizontal axis of the figures of this chapter. For every configuration we solve 1000 instances (instance seeds $1, \ldots, 1000$). An instance is allowed at most 4096 attempts, each attempt is limited to 10^6 DLX nodes, and the instance to 60 s in total. Every attempt re-rolls both the buckets and the slot hash (Section 4.4), and ties in the branching rule are broken by the first column. Each instance is solved on a single thread. We call an instance *solved* if an attempt succeeds within the given limits. We record the attempt index a of every solved instance and define

$$b_a = \left\lceil \log_2(\max a + 1) \right\rceil$$

as the number of bits of a fixed-width attempt index that suffices for all instances of the configuration; it is only defined if all 1000 instances are solved. The total space per key is

$$\frac{m\,b + b_a}{n}.$$

The 1000 instances can be read as 1000 parts that share one attempt index width. Thus, w_a is the width needed by the worst of 1000 parts.

Attempts grow exponentially when seeds are saved. Figure 5.1 shows the number of attempts needed for all instances (solid) and for half of them (dashed) against the bits per key x spent on seeds. Fewer seed bits mean more attempts, roughly exponentially. For $n = 50$ and $b = 4$, each of the last two removed buckets (4 bits each) multiplies the number of attempts needed for all instances by about 10: 15 attempts for $m = 18$, 41 for $m = 17$, 407 for $m = 16$ and 3547 for $m = 15$. For $b = 5$ it is a factor of about 27 per removed bucket (89 attempts for $m = 13$, 2395 for $m = 12$). Below the limit $x < \alpha_n$, some configurations are not solved completely within the 4096 attempts: $n = 50$ with $b = 2$ (502 of 1000 solved for $m = 34$, and fewer for fewer buckets), $n = 60$ with $b = 4$ and $m = 18$ (705 solved), and $n = 60$ with $b = 6$ and $m = 12$ (707 solved; all 293 unsolved instances are time-outs).

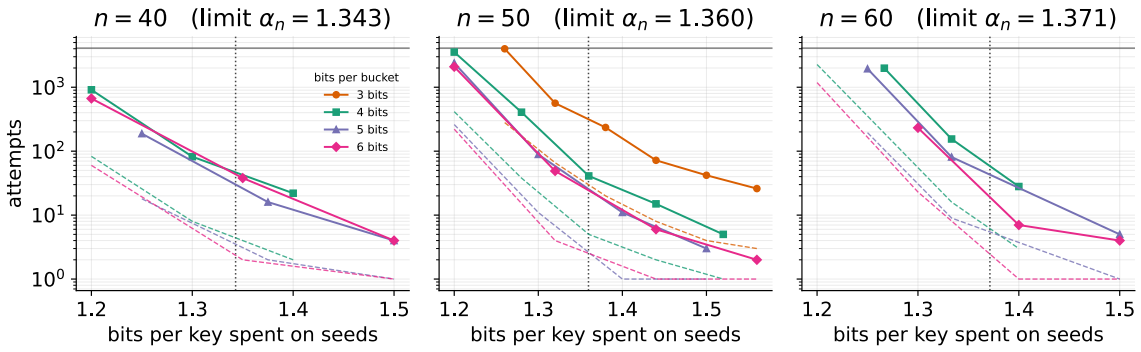


Figure 5.1: Attempts needed against the bits per key spent on seeds. Solid: attempts for all 1000 instances (maximum); dashed: median; dotted: limit α_n.

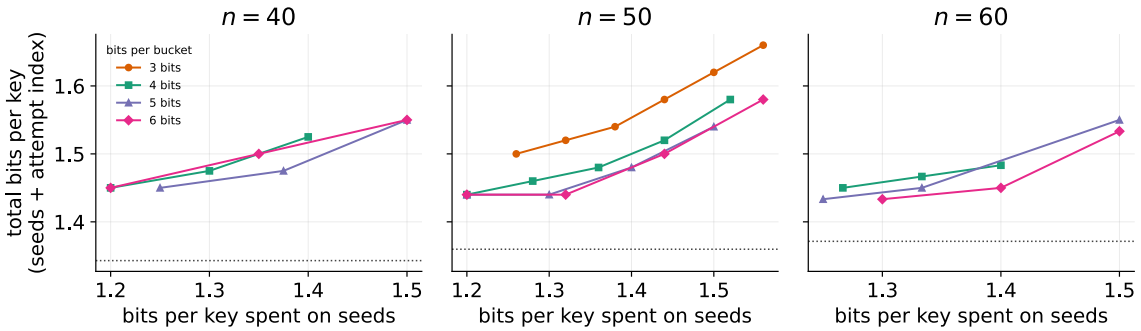


Figure 5.2: Total bits per key (seeds and attempt index) against the bits per key spent on seeds. Dotted: limit α_n.

Total space. Figure 5.2 presents the total space consumption per key. Saving seed bits does not pay off below a certain point: the bits saved reappear in the attempt index. For $n = 50$, $b = 4$ and $m = 15$ buckets, the seeds take 60 bits and the attempt index 12 bits, 72 bits in total, against $n\alpha_n = 68.0$ bits. Several configurations achieve the same minimum total space. For $n = 50$, five configurations attain 1.440 bits per key. Table 5.1 shows one best configuration per size. The best total space is 8.0 %, 5.9 % and 4.5 % above the limit for $n = 40$, 50 and 60.

5.3 Success Rate of a Single Attempt

The previous experiment allowed re-rolls: every instance was retried until it was solved, and we measured how many attempts this took and what the attempt index costs in space. This experiment looks at a single attempt, without re-rolls, and asks: for a part of n keys, with σ seeds per bucket and about λ keys per bucket, how often does a conflict-free choice of seeds exist at all? This success rate p explains the number of attempts of the previous

n	limit α_n	$\log \sigma$	buckets m	bits/key	gap
40	1.343	4	12	1.450	8.0 %
50	1.360	4	15	1.440	5.9 %
60	1.371	5	15	1.433	4.5 %

Table 5.1: Best total space per size (seeds and attempt index for all 1000 instances). Several configurations tie; one is shown.

experiment: a success rate p means that about $1/p$ attempts are needed (Section 4.4).

Setup. We use a single attempt without re-rolls, parts of $n \in \{50, 100\}$ random keys, $\sigma \in \{2, 4, \ldots, 256\}$ seeds and several numbers of buckets $m = \lceil n/\lambda \rceil$, so that the seeds take $x = m \log_2 \sigma / n$ bits per key. For every configuration we solve up to 1000 instances with DLX (Section 4.3). An instance is a *success* if a choice of one seed per bucket without a slot conflict is found, and *infeasible* if the search is exhausted without a solution, i.e. no such choice exists. The success rate $\hat{p}$ is the number of successes divided by the number of finished instances; we show 95 % Wilson intervals in the figure. Each instance has a time limit of 3600 seconds; instances that exceed it are not counted as finished (see the limits below).

The threshold moves with σ. Figure 5.3 shows the success rate against x. For every σ the rate rises from nearly 0 to nearly 1 within a range of about 0.2 bits per key, and the position of this range depends strongly on σ. For $n = 50$ and $\sigma = 16$ the rate is 0.016 at $x = 1.28$, 0.153 at 1.36, 0.431 at 1.44, 0.841 at 1.6 and 1 at 2.24. With $\sigma = 2$ no instance of $n = 50$ succeeds for any $x \leq 2.22$, and with $\sigma = 4$ the rate stays below 0.05 (at most 0.006 for $n = 100$) in this range. Table 5.2 gives the number of bits per key at which the rate crosses 50 %. Doubling σ lowers this value, but with diminishing returns: for $n = 50$ it falls from 1.68 ($\sigma = 8$) to 1.46, 1.39, 1.38, and then stays at 1.36 for $\sigma = 128$ and 256, which equals the limit $\alpha_{50} = 1.360$.

Larger parts need more bits. At the same σ, a larger n needs more bits per key (Table 5.2): the 50 % point moves from 1.68 to 1.75 for $\sigma = 8$, from 1.46 to 1.52 for $\sigma = 16$ and from 1.39 to 1.45 for $\sigma = 32$ when n grows from 50 to 100. Part of this is the limit itself, which grows from 1.360 to 1.396. The distance to the limit grows as well: from 0.03 to 0.05 bits for $\sigma = 32$, from 0.10 to 0.12 for $\sigma = 16$ and from 0.32 to 0.35 for $\sigma = 8$. A fixed number of seeds therefore gets less effective as the part grows.

Consistency with the first experiment. The two experiments examine the same process from two different perspectives. Section 5.2 re-rolls until an instance is solved and records how many attempts the median instance needed. This section measures the success probability p of one attempt. If successive attempts are independent and each succeeds with

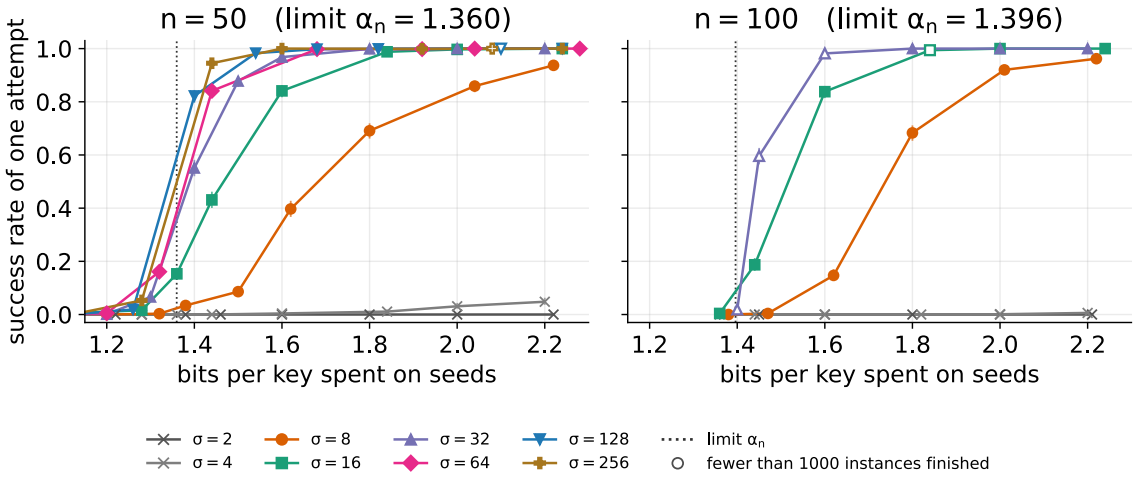


Figure 5.3: Success rate of one attempt against the bits per key spent on seeds, one curve per number of seeds σ, with 95 % Wilson intervals. Dotted: limit α_n; hollow markers: fewer than 1000 instances finished.

probability p, the number of attempts until the first success is geometrically distributed. Its median is the smallest k with $1-(1-p)^k \geq \frac{1}{2}$, i.e. $k = \lceil \ln 2 / \ln \frac{1}{1-p} \rceil \approx \ln 2/p$. We therefore predict the median reported in Section 5.2 from the rate $\hat{p}$ measured here, for the same $n = 50$, the same number of seeds σ and the same number of buckets m, and compare the two (Table 5.3). Only configurations in which all 1000 instances were solved in the first experiment and at least one attempt succeeded in this experiment are included in the comparison. The prediction interval is obtained by inserting the ends of the 95 % Wilson interval of $\hat{p}$.

For 11 of the 13 configurations the measured median lies inside the interval, and in all configurations with $\hat{p} \geq 0.05$ except the 3-bit ones it equals the prediction or lies next to it (for example 5 against 5 for $\sigma = 16$, $m = 17$; 11 against 11 for $\sigma = 32$, $m = 13$; 4 against 4 for $\sigma = 64$, $m = 11$). For $\hat{p} > \frac{1}{2}$ both experiments give a median of one attempt, as they must. Where $\hat{p}$ is small the interval is wide (1 success in 1000 allows medians from 123 to 3926), so these rows confirm the order of magnitude only. The agreement supports three statements: both measurements describe the same quantity; consecutive attempts behave like independent draws, and re-rolling the buckets and the slot hash yields attempts that are as good as fresh random instances.

Two rows show discrepancies, both with 3 bits per bucket. For $m = 22$ the prediction 231 (79 to 679) lies above the measured 66, but the rate rests on 3 successes only. For $m = 25$ the prediction is 8 (7 to 10) and the measured median is 4, i.e. the re-rolled attempts succeeded faster than independent random instances would. For $m = 22$ the deviation is plausibly due to sampling error, since the prediction rests on only 3 successes. For $m = 25$ the measured median is lower than the interval allows, so sampling error does not explain it. This deviation is favourable, as fewer attempts are required than predicted, and therefore

σ	$n = 50$	$n = 100$
8	1.68	1.75
16	1.46	1.52
32	1.39	1.45
64	1.38	–
128	1.36	–
256	1.36	–
limit α_n	1.36	1.40

Table 5.2: Bits per key at which the success rate of one attempt crosses 50 % (interpolated on the logit scale between the two measured rates around it). Dashes: no two measured rates with at least 900 finished instances enclose 50 %.

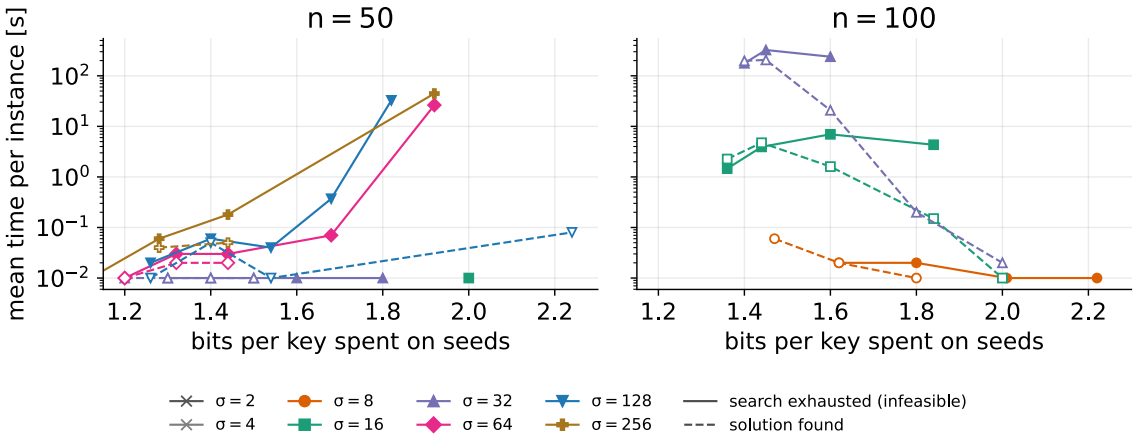


Figure 5.4: Mean time per instance of searches that found a solution (dashed, hollow) and of searches that were exhausted without one (solid), against the bits per key spent on seeds. Values of 0.00 s are not shown.

does not weaken the conclusions regarding space consumption.

Failing is expensive. Figure 5.4 shows the mean time per instance of successful and of exhausted searches. For $n = 50$ both are cheap (mean at most 0.4 s, with the exception of three configurations whose few exhausted searches took 26 to 45 s on average). For $n = 100$ the times grow strongly with σ. With $\sigma = 16$ an exhausted search takes 1.5 to 7 s, but with $\sigma = 32$ at $x = 1.4$ to 1.6 proving that no solution exists takes 177, 324 and 239 s on average, and even a successful search takes 197, 207 and 21 s. Near the threshold, an unsuccessful attempt costs as much as a successful one or more, so the search limit per attempt (Section 4.4) has to be small compared to these times: an attempt that is not cut off early can cost more than many re-rolled attempts.

$\log \sigma$	m	$\hat{p}$	predicted	95 % interval	measured
3	22	0.003	231	[79, 679]	66*
3	23	0.034	21	[15, 29]	20
3	25	0.086	8	[7, 10]	4*
4	16	0.016	43	[27, 70]	38
4	17	0.153	5	[4, 5]	5
4	18	0.431	2	[2, 2]	2
5	12	0.001	693	[123, 3926]	261
5	13	0.066	11	[8, 13]	11
5	14	0.551	1	[1, 1]	1
5	15	0.878	1	[1, 1]	1
6	10	0.005	139	[60, 324]	221
6	11	0.161	4	[4, 5]	4
6	12	0.841	1	[1, 1]	1

Table 5.3: Median number of attempts predicted from the success rate $\hat{p}$ of one attempt (this section) and measured with re-rolls (Section 5.2), $n = 50$. The interval follows from the 95 % interval of $\hat{p}$. *: the measured value lies outside the interval.

Limits. Each instance was subject to a time limit of 3600 seconds. Instances that hit the limit are not counted as finished. For $n = 50$ at most 2 of 1000 instances per configuration are missing, and for $n = 100$ at most 32 are missing. The unfinished instances are not a random subset: they are the instances for which the search took the longest, presumably the lengthy exhausted searches near the threshold. Rates close to the threshold at $n = 100$ are therefore rates among the instances that finished early, and the mean times underestimate the true mean. Rates with fewer than 1000 finished instances are drawn with hollow markers in Figure 5.3, and rates with fewer than 900 are not used in Table 5.2. The experiment is limited to parts of at most 100 keys because, with our solver, instances near the threshold cannot be decided within an hour for larger parts. DLX uses the first tie-breaking rule and a single attempt per instance.

5.4 Dancing Links Against an ILP Solver

Before dancing links, we solved the choice problem of Section 4.2 as an integer linear program with the solver Gurobi (Section 4.6). This section compares both solvers on the same instances. It asks two questions: how much faster is DLX, and how many more instances does this speed let us decide within the time limit?

Setup. The instances are those of Section 5.3: parts of $n \in \{50, 100\}$ random keys, σ seeds and several numbers of buckets, 1000 instances per configuration. The ILP has one binary variable per candidate (bucket, seed), requires that exactly one candidate per

bucket is chosen and that no slot is used by two chosen candidates (candidate collisions are calculated beforehand and given to the solver as constraints). We use Gurobi version 12.0.2 with one thread per instance, 3600 seconds timelimit and otherwise default settings. We compare 85 configurations that were run with both solvers (60 for $n = 50$ with $\sigma \leq 128$, 25 for $n = 100$ with $\sigma \leq 32$). An instance is *decided* if the solver reports a solution or proves infeasibility within the time limit; all other instances are *undecided*. The time of a configuration is the mean time per decided instance. The times of the DLX are printed with two decimals, so 0.00 s means less than 5 ms.

Both solvers give the same answers. Across all 85 configurations, the ILP finds no more solutions and proves no more instances infeasible than DLX. In the configurations where the ILP leaves instances undecided, DLX decides these instances, and the counts add up exactly. For $n = 50$, $\sigma = 128$ and $x = 1.40$ the ILP finds 720 solutions and proves 147 instances infeasible, 133 are undecided; DLX finds 822 and 178, i.e. 102 solutions and 31 infeasible instances more, 133 in total. The same holds for $n = 100$, $\sigma = 16$ ($x = 1.44$: 168 undecided, DLX finds 33 more solutions and 135 more infeasible instances; $x = 1.6$: 86 undecided, 36 and 50). Thus, the two solvers agree on every instance that both decide, up to the resolution of the reported counts. This provides a useful consistency check for both implementations.

DLX significantly outperforms the ILP solver. Figure 5.5 shows the mean time per decided instance. The ILP needs about 2 s per instance even for trivial instances: for $n = 50$ and $\sigma \leq 8$ its mean time lies between 1.7 and 2.2 s, independent of the success rate, while DLX needs less than 5 ms. We assume this time is due to a constant measurement overhead rather than the solving process itself. The time was measured in the old experiment code around the entire call and may include reading the instance, constructing the model, and checking the Gurobi license. We therefore exclude the speedups of these easy configurations. For larger σ the time of the ILP grows with the difficulty of the instance (at $n = 50$ from about 3 to 17 s for $\sigma = 32$, up to about 225 s for $\sigma = 64$ and 600 s for $\sigma = 128$), while DLX stays below 0.1 s. Table 5.4 gives the median speedup per σ over all configurations in which both solvers decided at least 90 % of the instances and the ILP needs at least 10 s on average, so that an overhead of about 2 s changes the ratio by at most 25 %. For $n = 50$ the median speedup is about 1 700 for $\sigma = 32$, 3 200 for $\sigma = 64$ and 7 000 for $\sigma = 128$, with a maximum of about 30 000. Over all 20 such configurations, the median speedup is about 2 700. For $n = 100$, only one configuration meets these criteria ($\sigma = 16$), with a speedup of 18.

A faster solver decides more instances. In total, the ILP leaves 1580 of the 85 000 instances undecided, DLX 43. For $n = 50$ these are 201 against 2 of 60 000 instances, for $n = 100$, 1379 against 41 of 25 000. The undecided instances are concentrated near the threshold at larger σ (Figure 5.6): in five configurations the ILP leaves at least 5 % of the

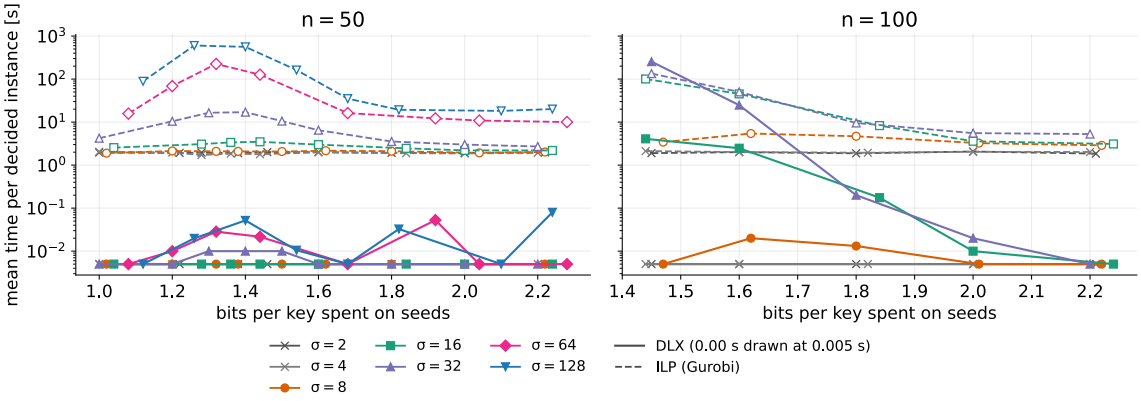


Figure 5.5: Mean time per decided instance of DLX (solid) and of the ILP solver (dashed, hollow) against the bits per key spent on seeds. DLX times of less than 0.00 s are drawn at 5 ms.

σ	median speedup, $n = 50$	median speedup, $n = 100$
2	≥ 390	≥ 380
4	≥ 379	≥ 403
8	≥ 417	575
16	≥ 554	203
32	1 054	277
64	3 193	–
128	7 018	–

Table 5.4: Median speedup of DLX over the ILP solver (ratio of the mean times per decided instance) over all configurations in which both solvers decided at least 90 % of the instances. $\geq$: DLX times below 5 ms in all configurations, so the speedup is a lower bound.

instances undecided, namely 133 for $n = 50$, $\sigma = 128$ and $x = 1.40$, 168 and 86 for $n = 100$, $\sigma = 16$ at $x = 1.44$ and 1.6, and 883 and 225 for $n = 100$, $\sigma = 32$ at $x = 1.45$ and 1.6. DLX decides all instances of the first three of these and all but 32 and 8 of the last two. At $n = 100$, $\sigma = 32$ and $x = 1.45$ the ILP decides only 11.7 % of the instances, DLX 96.8 %. The mean time shows the opposite order in this configuration (134 s for the ILP against 254 s for DLX), but it is taken over different instances: the ILP decided only the easiest 117 of them, so its mean time is too small. This is why the speedups of Table 5.4 are only reported where both solvers decided at least 90 % of the instances, and why the share of decided instances, not the mean time, is the right measure near the threshold.

Limits. The ILP running times include a constant overhead of approximately 2 s, the source of which we have not been able to identify. For this reason, we only compare

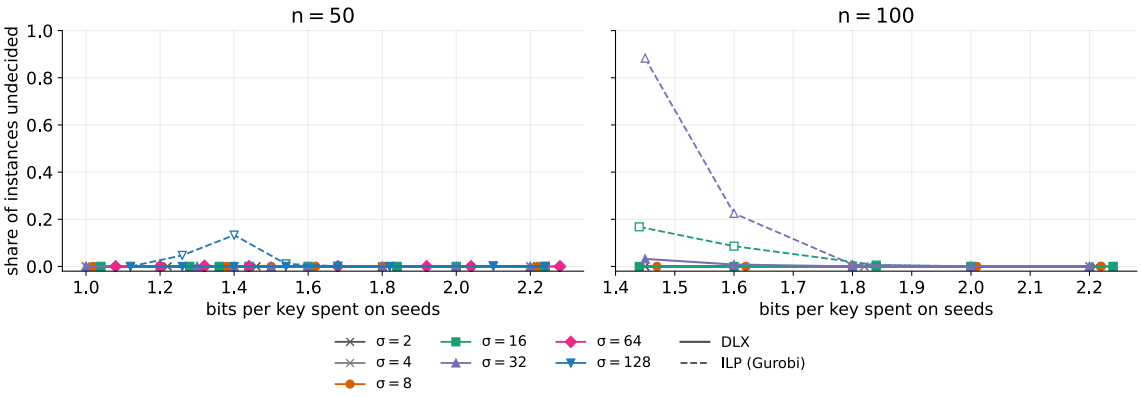


Figure 5.6: Share of instances that were not decided within the time limit, for DLX (solid) and for the ILP solver (dashed, hollow).

speedups for instances where the ILP solver requires at least $10\,\mathrm{s}$. Our evaluation uses a single ILP formulation, one solver, and its default settings; alternative formulations or configurations may yield better performance. The mean times are taken over decided instances only, so the times of the ILP are too small in the configurations with many undecided instances, and the speedups there would be larger. The comparison covers parts of at most 100 keys, because larger parts are not decided within the time limit by either solver close to the threshold. The speedup does not change the success rates of Section 5.3, which are the same for both solvers; it changes how large a part can be before the search time becomes impractical, and how many instances of a given size can be decided within a time limit.

5.5 The Whole Construction on Large Key Sets

The previous sections looked at a single part. Here we run the complete construction (Section 4.5) with the simple prefix-sum layout on up to 10^7 keys and ask three questions. How many bits per key does the finished function need, and where do they go? How does this depend on the size of the parts? How long does the construction take, and how well does it use many threads?

Setup. The keys are n distinct random 64-bit integers. Every part is solved independently with dancing links and re-rolls (Section 4.4) with σ seeds per bucket and λ keys per bucket on average. A part gets at most 4096 attempts of at most 10^6 search nodes each and at most $60\,\mathrm{s}$ in total; a part that exceeds one of these limits makes the run fail (exceptions are stated). The stored data are the seeds ($\lceil \log_2 \sigma \rceil$ bits per bucket), the attempt index of each part (fixed width, as many bits as the largest index needs), and the two prefix-sum arrays of the layout (slot offsets and seed offsets, each $P+1$ entries of fixed width). After the construction we check that all keys are mapped to distinct values in $[0, n)$; this check suc-

keys/part	seeds	attempt idx	prefix	total	EF est.	CPU/key [μs]	wall [s]
32	1.670	0.229	1.219	3.118	2.305	101	1.8
48	1.651	0.139	0.813	2.602	2.081	243	2.3
64	1.638	0.094	0.609	2.341	1.966	1 960	11.2
80	1.630	0.067	0.488	2.184	1.896	6 708	32.1
96	1.625	0.059	0.406	2.091	1.851	15 089	67.8
128	1.619	0.070	0.305	1.994	1.822	56 322	254.9

Table 5.5: Space (bits per key) and cost by part size, $n = 10^6$, $\sigma = 32$, $\lambda = 3.125$. Space: mean of three runs with different seeds (128 keys: one run, with at most 10^5 attempts and 600 s per part; all others 4096 and 60 s). Time: mean of two runs with 256 threads (128 keys: one run); "CPU" and "wall" as defined in Section 5.1.

ceeded in all 57 runs that finished. The space is computed from the widths of these arrays and is therefore exact; it agrees with the formulas of Section 4.5. If not stated otherwise, $n = 10^6$, $\sigma = 32$ and $\lambda = 3.125$, i.e. $5/3.125 = 1.6$ seed bits per key. Every configuration was run with one to three seeds. To measure the query time, we query all keys one after another on a single thread, in the random order of the shuffled key array, after a warm-up of up to 10^5 queries, and divide the elapsed time by the number of queries.

Prefix sums dominate the overhead. Figure 5.7 (left) and Table 5.5 show the space for different part sizes. The seeds need 1.62 to 1.67 bits per key, which is the intended 1.6 plus a small increase for the buckets that are empty or larger than average. The attempt index costs between 0.23 bits per key (parts of 32 keys) and 0.06 to 0.07 (parts of 96 and 128 keys). The space required for the prefix sums is given by $(P+1)(w_O+w_C)/n \approx 39/n_p$ bits per key for $n = 10^6$, where $w_O = 20$ and $w_C = 19$ bits per entry. This amounts to 1.22 bits per key for parts of 32 keys, 0.61 for 64 keys, and 0.41 for 96 keys. With 32 keys per part the total is 3.12 bits per key, more than twice the lower bound of 1.44. It falls to 2.34 at 64 keys, 2.09 at 96 and 1.99 at 128 keys, which is still 45 % and 38 % above the bound, respectively. The result is the same for the second ratio we tried ($\lambda = 2.7778$, 1.8 seed bits per key): the total is 3.31, 2.77, 2.53, 2.38 and 2.26 bits per key for 32 to 96 keys per part, always about 0.2 above the first series because the seeds cost more.

So in our implementation the overhead is a constant: the offsets cost about $2 \log_2 n$ bits per part, and parts that can be solved have at most about one hundred keys. Using fixed-width offsets is inefficient because consecutive offsets differ by only approximately n_p and could therefore be encoded much more compactly. We did not implement such an encoding. As an estimate we computed what Elias–Fano coding of the two offset sequences would need (column "EF" of the table; $2 + \lceil \log_2(U/\text{count}) \rceil$ bits per element): 1.97 bits per key at 64 keys per part, 1.85 at 96 and 1.82 at 128. This would remove about 0.38 bits per key at 64 keys per part and 0.17 at 128.

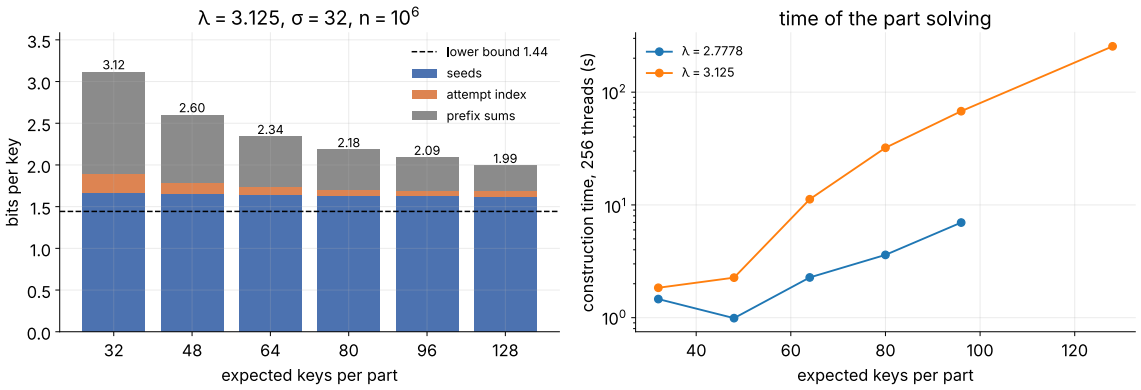


Figure 5.7: Left: space in bits per key by part size for $n = 10^6$, $\sigma = 32$, $\lambda = 3.125$; the dashed line is the lower bound 1.44. Right: time of the whole construction with 256 threads (log scale) for both ratios.

The price of larger parts. The space savings achieved by using larger parts come at the cost of increased running time (Figure 5.7, right). The summed time per key grows by a factor of 2.4 from 32 to 48 keys per part, then 8.1, 3.4 and 2.3 for each further step of 16 keys (and 3.7 from 96 to 128 keys per part); with 256 threads the whole construction of 10^6 keys takes 2 to 3 s up to 48 keys per part, 11 s for 64, 68 s for 96 and 255 s for 128 keys. Going from 64 to 96 keys per part saves 0.25 bits per key (11 %) and costs about 7.7 times the work. The time per part grows because the solver explores more nodes for every attempt, not because more attempts are needed: the mean attempt index stays below 0.45 for all sizes up to 96 keys, and only the 128-key parts need more (3.2 on average, at most 293). We therefore consider part sizes of 64 to 96 keys to be the practical range. For larger parts, construction time rather than space becomes the limiting factor.

Seeds against attempts. Table 5.6 varies σ and λ for parts of 64 keys ($n = 10^6$, one run each) and of 50 keys ($n = 2 \cdot 10^5$, three runs each), all with 128 threads. For every σ there are two choices of λ: one with 1.6 seed bits per key, and one with 1.44 bits per key, i.e. with seeds at the threshold of Section 5.3. At 64 keys, the second choice saves 0.16 bits per key in the seeds, and the attempt index costs only 0.016 more: the total falls from 2.31–2.35 to 2.17–2.20 bits per key (6 to 7 %) for every $\sigma \in \{16, 32, 64\}$. The price is the running time, which grows by a factor of 4.3 ($\sigma = 16$), 7.2 ($\sigma = 32$) and 8.1 ($\sigma = 64$), because the success probability of a single attempt falls and the mean attempt index rises from 0.24–0.32 to 0.76–1.43. This is the trade-off of Section 5.2. Here, we save almost all the bits because the attempt index is cheap.

For parts containing 50 keys, we explored even lower space requirements. For $\lambda = 2.9412$ ($\sigma = 16$) and $\lambda = 3.6765$ ($\sigma = 32$) the seeds take 1.36 bits per key, which is the lower bound for a part of 50 keys ($\log_2(50^{50}/50!)/50 = 1.36$). All three runs of both configurations succeeded, with a mean attempt index of 3.1 and 1.1 and a largest index of

n_p	σ	λ	ok	seeds	att. idx	prefix	total	mean idx	max idx	CPU/key [μs]
64	16	2.7778	1/1	1.470	0.094	0.609	2.173	1.43	54	2 253
64	16	2.5000	1/1	1.625	0.078	0.609	2.312	0.29	23	521
64	32	3.4722	1/1	1.479	0.109	0.609	2.198	0.76	81	6 329
64	32	3.1250	1/1	1.638	0.094	0.609	2.341	0.24	47	873
64	64	4.1667	1/1	1.486	0.109	0.594	2.189	0.98	127	8 076
64	64	3.7500	1/1	1.644	0.094	0.609	2.347	0.32	57	1 001
50	16	2.7778	3/3	1.478	0.107	0.700	2.284	0.87	36	248
50	16	2.9412	3/3	1.397	0.147	0.700	2.244	3.15	228	644
50	32	3.6765	3/3	1.409	0.167	0.680	2.256	1.14	642	2 417
50	32	3.9063	0/3	—	—	—	—	10.57	872	16 300

Table 5.6: Space (bits per key) for different seed counts and bucket sizes; the first block has $n = 10^6$ and $n_p = 64$, the second $n = 2 \cdot 10^5$ and $n_p = 50$. "ok" counts the successful runs; space columns are means over the successful runs, attempt statistics and time over all runs. All runs used 128 threads; a failed run has no space. The attempt statistics of the failed runs are those of the parts that were solved.

228 and 642. At $\lambda = 3.9063$ (1.28 bits per key, below that bound) all three runs failed, in 5, 9 and 6 of the 4000 parts. These parts are not out of attempts (the largest attempt index reached is 872 of the 4096 allowed) but out of time: they did not finish within the 60 s limit of a part, and the mean attempt index of the runs is 10.6. The total space for all these configurations is between 2.2 and 2.3 bits per key, since, for such small parts, the prefix sums alone require 0.70 bits per key.

Scaling with n. Table 5.7 gives the results for $n = 10^5$, 10^6 and 10^7. The space grows slowly with n because the offsets need $\log_2 n$ bits: for 96 keys per part the total is 2.010 bits per key at $n = 10^5$, 2.084 at 10^6 and 2.188 at 10^7. For $n = 10^7$ the construction took 352 s with 256 threads and the verification 0.8 s. Of the 2.188 bits per key, 1.625 are seeds, 0.479 prefix sums and 0.083 the attempt index. The attempt index is larger than at 10^6 (0.052 bits per key) only because of a single part: its largest index is 198, which forces 8 bits for all 104 167 parts instead of 5 bits. Using variable-length or escape coding for rare large indices would save approximately 0.03 bits per key in this case.

The work per key does not depend on n. The dancing-links solver explores about 5 700 to 6 100 search nodes per key for 96-key parts at all three sizes (6 098, 5 808 and 5 739 at $n = 10^5$, 10^6 and 10^7) and 760 to 860 for 64-key parts (Table 5.7, column "nodes/key"). The number of explored nodes does not depend on the machine, so the construction is linear in n. The summed time per key varies more (13.0, 15.0 and 8.8 ms at 256 threads for the 96-key parts). The wall time of the 96-key runs grows from 66 s to 352 s from $n = 10^6$ to 10^7. A query takes 44 to 58 ns in all runs of this table (53 ns for 10^7 keys). The structure for 10^7 keys needs 2.7 MB and fits into the cache, so these numbers say little about the query time for much larger inputs.

n	n_p	threads	total [bits/key]	wall [s]	CPU [s]	nodes/key	query [ns]	attempt bits
10^5	64	128	2.216	2.4	100	857	43.8	5
10^6	64	128	2.341	9.0	874	758	46.6	6
10^5	96	256	2.010	14.7	1 300	6 098	57.6	5
10^6	96	256	2.084	66.1	15 039	5 808	46.1	5
10^7	96	256	2.188	352.2	88 150	5 739	53.0	8

Table 5.7: Scaling with the number of keys, $\sigma = 32$, $\lambda = 3.125$. "nodes/key" is the number of search nodes of the dancing-links solver over all attempts, divided by n. "CPU": total CPU time in seconds. The row for $n = 10^7$ was run with at most 10^5 attempts and 600 s per part (see text).

The run for $n = 10^7$ could not be completed using the standard limits. Our first attempt with 60 s per part stopped after 521 s, because one part of the 104 167 had used up its 60 s without a solution. With 600 s and 10^5 attempts per part the same instance (same seed) completed: the part that had failed needs 199 attempts and 58.4 s, just below the former limit (runtime variance). We examine this heavy tail in the following section.

The time per part has a heavy tail. Figure 5.8 shows the time of every part in the runs with 10^6 keys (64 keys per part) and 10^7 keys (96 keys per part). For 10^6 keys the median part needs 2.2 ms, but the mean is 71 ms, the slowest percent of the parts needs more than 1.5 s, and the slowest part 5.8 s (it has 84 keys and needs 5 attempts). For 10^7 keys the median is 0.13 s, the mean 0.85 s, the 99th percentile 6.6 s, and the slowest part takes 58.4 s (128 keys, 199 attempts). Two effects cause this. First, the sizes of the parts vary (the part sizes lie between 33 and 100 keys for an average of 64, and between 58 and 137 for an average of 96), and the mean time of a part grows by a factor of 1.14 per additional key at $n_p = 64$ and 1.07 at $n_p = 96$ (right panel), so most of the time is spent in the large parts. The zigzag in the right panel repeats about every 3 keys. A part gets one more bucket every $\lambda = 3.125$ keys. Between two such steps the buckets get larger, which makes the part harder, and with the next bucket they become smaller again. Second, among parts of the same size the number of attempts varies (Section 5.2). Still, the total is not dominated by a few parts: the five slowest parts account for 1.9 % of the summed time at 10^6 keys and 0.2 % at 10^7 keys.

Parallelism. Since the parts are independent, the construction is straightforward to parallelize, and we distribute them dynamically across the threads. Table 5.8 and Figure 5.9 show the solving time for $n = 10^6$ and 64 keys per part for 1 to 256 threads. Up to 8 threads the speedup is nearly ideal (efficiency 97 to 102 %; values above 100 % are measurement noise). It then falls to 91 % at 16, 84 % at 32, 77 % at 64 and 64 % at 128 threads, where the solving takes 10.8 s instead of 876 s. The CPU time increases only moderately over this range, from 876 s with one thread to 1 065 s with 128 threads. Thus, most of the efficiency

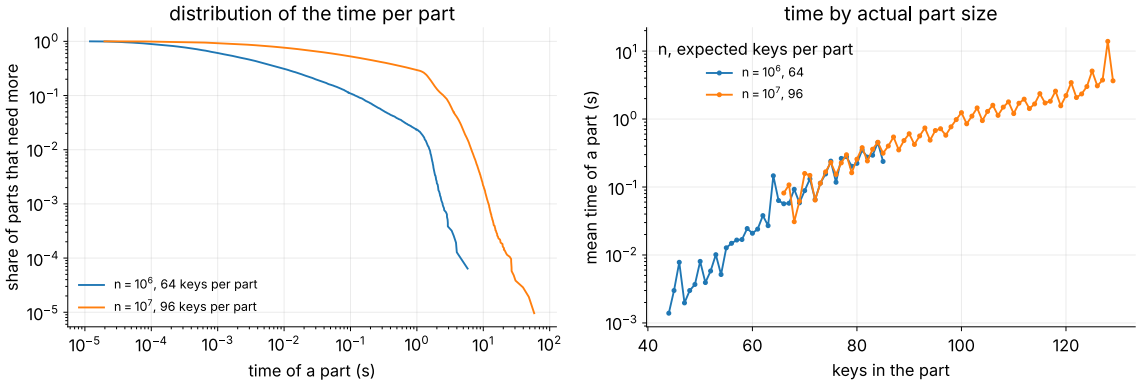


Figure 5.8: Left: share of the parts whose solving time exceeds a given value, for $n = 10^6$ with 64 keys per part (15 625 parts) and $n = 10^7$ with 96 keys per part (104 167 parts). Right: mean time of a part by its actual number of keys (sizes with at least 20 parts). The zigzag has a period of about 3 keys: each additional bucket makes the buckets smaller and the part easier.

loss is not caused by additional work, but by threads remaining idle towards the end of the run. With 128 threads, the average thread performs 8.3 s of work, while the total runtime is 10.8 s. This is caused by the long tail discussed in the previous paragraph. The slowest single part takes 5.8 s, and no number of threads can bring the time below that value (Figure 5.9, dotted line); 10^6 keys do not offer enough work for 256 threads. At 256 threads the solving takes 7.0 s (efficiency 49 %), only 1.2 times the time of the slowest part. The idle time at the end could be reduced by solving the largest parts first. The size of a part is known before the solving starts, and the time of a part grows strongly with its size, so sorting the parts by size in decreasing order and distributing them dynamically over the threads would let the small parts fill the gaps at the end of the run. The size is only a rough predictor, because the time also depends on the number of attempts among parts of the same size, so the slowest part can still start late.

Summary. With the simple layout, the complete function needs between 2.0 and 2.3 bits per key for parts of 64 to 128 keys, against the lower bound of 1.44, for n up to 10^7. Roughly 0.2 bits per key are due to the choice of the ratio of seeds, 0.06 to 0.1 to the attempt index, and 0.3 to 0.6 to the prefix sums, which are the part that is easiest to improve. Larger parts reduce the prefix-sum overhead but considerably increase construction time. Seeds at the threshold save about 0.16 bits per key (6 to 7 %) at the price of 4 to 8 times the construction time; the larger attempt index costs only 0.016 bits per key. The construction is linear in n (about 6 minutes for 10^7 keys with 256 threads, 876 s of work for 10^6 keys). Parallel speedup is nearly ideal up to 8 threads and is limited by the slowest parts beyond that.

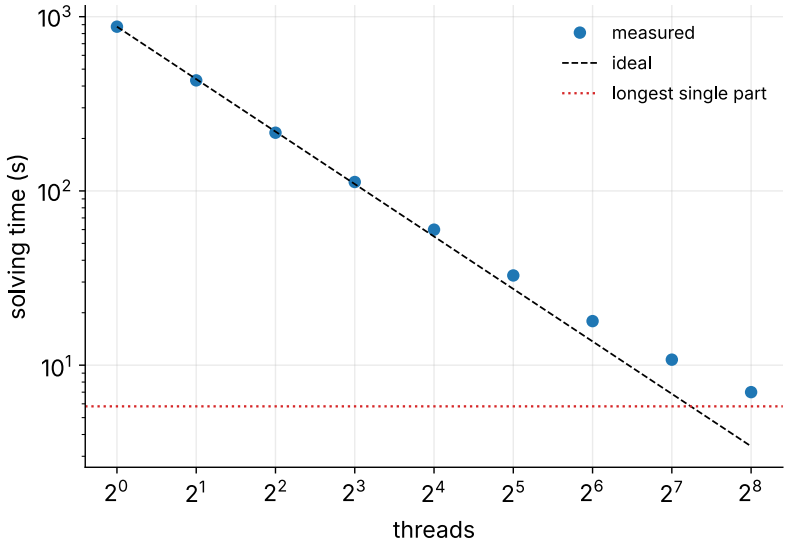


Figure 5.9: Solving time for $n = 10^6$, $n_p = 64$ by number of threads (log–log). The dashed line is the ideal speedup from the run with one thread; the dotted line is the time of the slowest single part.

5.6 Parameters of the Search

Most of the previous experiments used the same limits for the dancing-links search: at most 10^6 search nodes per attempt, at most 4096 attempts and at most $60\,\mathrm{s}$ per part, and the first of the columns with the fewest remaining entries as the next column to branch on (Section 4.3). This section examines these choices. It asks how the node limit per attempt influences the cost and the space of the whole construction, how the rule that breaks ties between columns compares, and whether the limits on the number of attempts and on the time per part matter.

Setup. We use the setting of Section 5.5 with $n = 10^6$ keys in parts of 64 keys, $\sigma = 32$ and 128 threads on the same machine, and two bucket sizes: $\lambda = 3.125$ (1.6 seed bits per key, “easy”) and $\lambda = 3.4722$ (1.44 seed bits per key, “hard”; seeds at the threshold of Section 5.3). Each configuration was run with two seeds. One run solves $15\,625$ parts, which gives a large sample for each setting. Space (bits per key) is exact; as a measure of the work we use the CPU time (Section 5.1), which is proportional to the number of search nodes (between 1.1 and $1.4\,\mu\mathrm{s}$ per node across all runs in this section).

Shorter attempts are cheaper. Table 5.9 and Figure 5.10 show the node limit per attempt from 10^3 to 10^7; for these runs the limit on the number of attempts was raised to 10^5. The work grows with the limit in both configurations, although the limit applies only

threads	wall [s]	CPU [s]	speedup	efficiency
1	876.3	876.3	1.0	1.00
2	431.0	861.6	2.0	1.02
4	215.8	861.4	4.1	1.02
8	112.5	895.6	7.8	0.97
16	59.9	950.0	14.6	0.91
32	32.7	1 014.1	26.8	0.84
64	17.9	1 036.9	49.0	0.77
128	10.8	1 064.8	81.5	0.64
256	7.0	1 117.5	125.2	0.49

Table 5.8: Strong scaling for $n = 10^6$, $n_p = 64$, $\sigma = 32$, $\lambda = 3.125$. Speedup and efficiency (speedup divided by the number of threads) are relative to the run with one thread. The result (bits, attempts) is identical for all thread counts.

to attempts that do not find a solution: for the easy configuration the CPU time is 46 s at a limit of 10^3 nodes, 348 s at 10^5, 894 s at 10^6 and $1\,588$ s at 10^7; for the hard configuration it is 530 s, $2\,336$ s, $6\,220$ s, and the run did not finish for 10^7 (see below). A smaller limit makes the search cheaper because the cost of an attempt has a heavy tail, like the cost of a part in Section 5.5: most attempts succeed or fail quickly, and a few use up the whole limit without success. Cutting these attempts off early and starting a new one (Section 4.4) costs less than waiting for them. Compared with the limit of 10^6 used so far, the work is smaller by a factor of 19.6 (easy) and 11.7 (hard) at a limit of 10^3.

The trade-off is a larger attempt index, since each part requires more attempts.: at a limit of 10^3 the mean attempt index is 3.0 (easy) and 32 (hard), against 0.24 and 0.76 at 10^6, and the largest index is 254 and $3\,562$. The attempt index is stored once per part, with as many bits as the largest index needs, so the cost is small: the width grows from 6 to 8 bits (easy) and from 7 to 12 bits (hard), i.e. by 0.031 and 0.078 bits per key, since one bit per part costs $1/64 = 0.016$ bits per key. The seeds do not change. The total space therefore rises from 2.341 to 2.373 bits per key (easy) and from 2.198 to 2.276 (hard). At limits of 10^5 (easy) and $3 \cdot 10^5$ (hard) and above, the space is the same as at 10^6. In the hard configuration at 10^5 nodes the construction needs 2.7 times less work than at 10^6 for 0.016 bits per key more, and at $3 \cdot 10^4$ nodes it needs 4.5 times less for 0.023 bits more; in the easy configuration the limit 10^5 is 2.6 times cheaper at no cost in space, and 10^4 is 8.1 times cheaper for 0.016 bits per key. A limit of around 10^5 nodes is therefore a good choice for both configurations; below 10^4 nodes the time still falls, but more slowly for the hard configuration (596 s at $3 \cdot 10^3$, 530 s at 10^3), while the space keeps growing. We did not test limits below 10^3.

At the largest limit (10^7) in the hard configuration, both runs failed: 6 parts in one run and 1 in the other exceeded the per-part time limit of 60 s. These failures were caused by the time limit rather than by the number of attempts. A single attempt of 10^7 nodes takes approximately 12 s, so just five attempts are sufficient to reach the 60 s limit.

	easy ($\lambda = 3.125$)				hard ($\lambda = 3.4722$)			
limit	CPU [s]	mean idx	bits	total	CPU [s]	mean idx	bits	total
10^3	46	2.95	8	2.373	530	32.1	12	2.276
$3 \cdot 10^3$	64	1.55	8	2.373	596	13.3	11	2.261
10^4	110	0.86	7	2.357	881	6.15	10	2.245
$3 \cdot 10^4$	193	0.55	7	2.357	1 378	3.26	9	2.222
10^5	348	0.37	6	2.341	2 336	1.76	8	2.214
$3 \cdot 10^5$	567	0.29	6	2.341	3 796	1.09	7	2.198
10^6	894	0.24	6	2.341	6 220	0.76	7	2.198
$3 \cdot 10^6$	1 226	0.22	6	2.341	8 906	0.63	7	2.198
10^7	1 588	0.22	6	2.341	($\geq$ 11 876)	0.57	—	—

Table 5.9: Influence of the node limit per attempt ($n = 10^6$, 64 keys per part, $\sigma = 32$; mean of two runs). "CPU" is the CPU time of Section 5.1, "mean idx" the mean attempt index, "bits" the width of the attempt index, "total" the bits per key of the whole function. For 10^7 nodes in the hard configuration both runs failed (parts exceeded 60 s); the CPU time is a lower bound.

Tie breaking. Whenever several columns have the fewest entries, the search has to choose one of them. Table 5.10 compares five rules with the limits of the earlier sections: the first of the tied columns in the order of the matrix (`first`), the last (`last`), a random one (`random`), a bucket column if there is one (`bucket`) and a slot column if there is one (`slot`). The rule does not change the space, since all successful runs of a configuration give the same total number of bits. It changes the cost: `first` and `bucket` need the same number of search nodes in all runs, which is explained by the order of the columns (the bucket columns come first, so `first` always chooses a bucket column when one is tied). `random` needs 1.4 times as many nodes and takes 1.3 to 1.4 times as long, and `last` and `slot` need almost twice as many nodes (1.9 to 2.0 times) and take 1.6 to 1.8 times as long. The mean number of attempts per part is almost the same for all rules (easy: 1.24 to 1.28; hard: 1.76 to 2.02), so the rule changes mainly the cost of an attempt and not its success probability. We chose `first` for all experiments, and these results confirm that choice; we did not investigate why preferring slot columns is worse.

The limits on attempts and time per part. These two limits only determine when a part is abandoned, and both depend on the node limit. At the standard node limit, no run came close to reaching the limit of 4096 attempts: the largest attempt index is 102 for all tie-breaking runs, 293 for the parts of 128 keys in Section 5.5 and 872 for the failing parts of 50 keys. Only with very small node limits does the limit on the attempts matter: for the hard configuration at 10^3 nodes the largest index is 3 562, close to 4096, which is why these runs used a limit of 10^5. The time limit of 60 s binds earlier. With about 1.25 μs per node it allows around $5 \cdot 10^7$ nodes per part, while 4096 attempts of 10^6 nodes would be

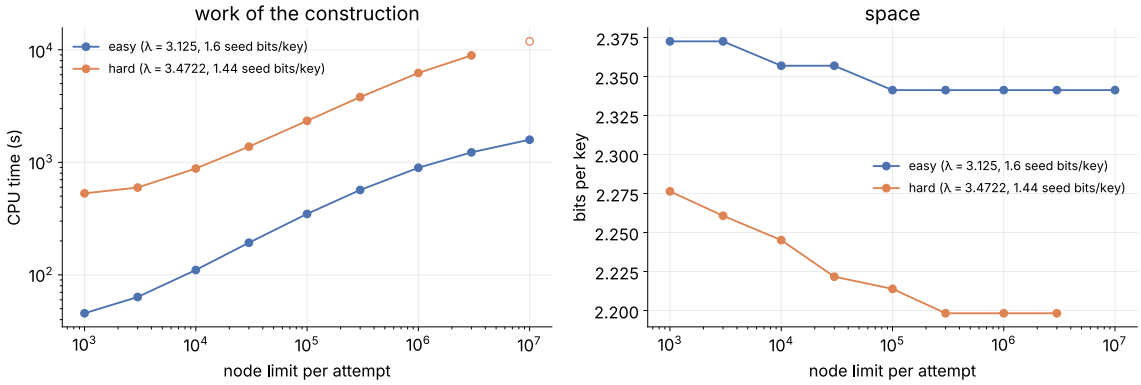


Figure 5.10: Left: CPU time of the construction (log–log) by node limit per attempt; the hollow marker is a lower bound (the runs failed). Right: bits per key of the whole function. The curves of the easy and the hard configuration have different levels because the seeds of the hard configuration are smaller.

	easy ($\lambda = 3.125$)			hard ($\lambda = 3.4722$)		
rule	CPU [s]	nodes	attempts/part	CPU [s]	nodes	attempts/part
`first`	894	1.00	1.24	6 219	1.00	1.76
`bucket`	897	1.00	1.24	6 235	1.00	1.76
`random`	1 224	1.44	1.26	8 198	1.39	1.87
`last`	1 530	1.97	1.27	10 255*	1.88*	2.02*
`slot`	1 572	2.02	1.28	10 241	1.87	2.02

Table 5.10: Tie breaking in the column choice ($n = 10^6$, 64 keys per part, $\sigma = 32$, node limit 10^6, mean of two runs). "nodes" is the number of search nodes relative to `first`. *: one of the two runs did not finish (a part exceeded 60 s), so its values are lower bounds. The space is identical for all rules.

$4 \cdot 10^9$ nodes. All failures in our experiments (parts of 50 keys with 1.28 bits per key, the first run for $n = 10^7$, and the node limit of 10^7 above) are failures of the time limit, not of the number of attempts. The run for $n = 10^7$ succeeded with 600 s per part (Section 5.5). We did not test whether a higher time limit would allow the parts of 50 keys with 1.28 bits per key to succeed.

Summary. The node limit is the key parameter: setting it to around 10^5 nodes makes the construction 2.6 to 2.7 times cheaper than with a limit of 10^6, at an additional cost of at most 0.016 bits per key. Smaller limits further reduce the work but result in a larger attempt index. Among the five tie-breaking rules tested, first-column tie breaking (equivalent to preferring bucket columns) performs best; rules that prefer slot columns require about twice as many nodes. The limits on the attempts and on the time of a part only decide when a part is given up; the time limit is the one that binds. We did not use this smaller limit in

the earlier sections: they were run with the limit of 10^6 nodes, chosen as a safe default we determined early before these experiment, and we did not repeat them. The space results in those sections remain unaffected, since the space at 10^6 nodes is the same as at $3 \cdot 10^5$ nodes (hard) and 10^5 nodes (easy). However, the construction times would be about 2.6 times lower.

CHAPTER 6

Discussion

6.1 Interpretation of the Results

The search works, and trades time for space. For parts of approximately 50 to 100 keys, the success probability of a single attempt increases from nearly 0 to almost 1 within about 0.2 bits per key, close to the lower bound for the part size. Re-rolls allow us to exploit this transition at the cost of storing an attempt index, which costs only 0.06 to 0.2 bits per key (Section 5.2). The bits saved in the seeds are therefore mostly kept, and the cost shows up as running time: seeds at the threshold save 0.16 bits per key and make the construction 4 to 8 times slower (Section 5.5). Thus, the number of seed bits and the node limit are the two main parameters governing this trade-off.

The remaining space is a matter of layout. At 64 keys per part, the seeds take 1.64 bits per key, the attempt indices 0.09 and the offsets of the parts 0.61, so the total of 2.34 is 0.9 bits above the limit. Most of this overhead is caused by our simple layout rather than the search procedure itself. Larger parts reduce the offset cost (0.31 bits per key at 128 keys), but the time per part grows by a factor of 1.07 to 1.14 for every additional key and has a heavy tail, which also limits the parallelism (Section 5.5). For these reasons, parts of 96 keys represent the upper end of the practical range. A more compact encoding of the offsets could remove a large part of the gap (Elias–Fano estimate: 1.8 to 2.0 bits per key), but we did not implement it.

Exact search against an ILP solver. Dancing links were much faster than an ILP solver (a median factor of about 2 700 where the ILP needs at least 10 s) and decided more instances within the same limit (Section 5.4). One possible explanation is that the exact cover formulation allows the solver to exploit forced choices and detect dead ends directly. However, our comparison was limited to a single formulation and the solver's default settings.

State of the art. With 2.0 to 2.3 bits per key, our function is in the range of PtrHash [10] (2.0) and CHD [1] (1.98) and above RecSplit [5] (1.56) and ShockHash-RS [21] (1.489) (Table 3.1). These numbers are taken from the respective papers, so we make no claim about construction or query time relative to them.

Limitations. Currently, the failure to solve even a single part within its resource limits causes the construction of the entire function to fail, as we do not have a fallback mechanism for such cases. This is why the largest instances, containing 10^7 keys, required a higher time limit. All experiments ran on one machine with uniformly random keys and mostly one to three seeds, so small differences between rows are not significant. The ILP timings contain a constant overhead of about 2 s, and the query time was measured on structures that fit into the cache.

6.2 Conclusion

Summary. In this thesis we looked at minimal perfect hash function constructions where seed information comes close to the information-theoretic lower bound of $\log_2 e \approx 1.44$ bits per key. We split the overall key set into small parts. For every part the keys are first distributed into buckets. We then search a seed for each bucket such that all keys of the part go to different slots. We formulate this search as an exact cover problem, solve it with dancing links, and use re-rolls of the hash functions if any attempt fails.

Main results. Our results lead to four main conclusions. First, the success probability of a single attempt depends strongly on the seed bits per key, and re-rolls make seeds near the lower limit usable at the cost of a small attempt index. Second, dancing links solve the instances much faster than an ILP solver (median speedup about 2 700 where the ILP needs at least 10 s) and decide many more of them within the same time limit (43 against 1 580 undecided out of 85 000 instances). Third, with a simple prefix-sum layout the complete function needs 2.0 to 2.3 bits per key for parts of 64 to 128 keys, of which the seeds take about 1.5 and the offsets of the parts most of the rest. The work grows linearly in the number of keys (10^7 keys took 352 s on 256 threads). Fourth, reducing the node limit from 10^6 to 10^5 makes the construction approximately 2.6 times cheaper while maintaining nearly the same space usage. Among the five tie-breaking rules evaluated, selecting the first column consistently produces the best results.

6.3 Future Work

There are several directions that we should consider. We order them roughly by how much space they could save.

Compact offsets and k-perfect hashing. The most straightforward improvement would be to use a more compact encoding for the part offsets. We looked at an estimate with Elias–Fano coding that could give 1.97 bits per key at 64 keys per part and 1.85 at 96, a saving of up to 0.4 bits per key (Section 5.5). However, we did not implement this. An alternative would be to adopt an approach similar to ShockHash-Flat [21]. They first compute a minimal k-perfect hash function mapping the keys to parts of *exactly* k keys, and then solving them independently as we do. If all parts have the same size, offsets would follow from the index and we do not need to store this information anymore. The space saved by removing the offsets is instead used to store the k-perfect hash function. This would at the same time handle another issue of our construction: our parts have random sizes (between 33 and 100 keys for an average of 64). We recognized that the time of a part grows by a factor of about 1.14 per additional key, meaning the slowest parts are the largest ones (Figure 5.8). By having parts of equal size, we would shorten this tail. To determine whether this approach is more efficient in practice, however, the space requirements of the k-perfect hash function would need to be analyzed.

Closer to the lower bound. In order to be competitive with state-of-the-art constructions, our total space has to come much closer to the limit of 1.44 bits per key. ShockHash reports values as low as 1.489 bits per key for 10^7 keys, while our complete function currently needs between 2.0 and 2.3. We notice that compact offsets are not enough: even with 1.85 bits per key our function would remain above this value. There are two main directions for improvement: allowing longer construction times and developing more efficient encodings. A longer construction would mean seeds way closer to the threshold, possibly allowing for more attempts, and larger parts. All three cost time, which grows steeply with the part size (Section 5.5). Better encodings on the other hand would mean using less than the fixed width for the seeds and attempt indices, for example via variable-length codes. Another possibility is to adopt the CONSENSUS [22] approach. Storing the smallest successful seed for each bucket is geometrically distributed and expensive to encode. CONSENSUS instead stores a short number of fixed width and chooses these numbers so that the whole sequence is cheap to store. Overall, this brings the space close to the information that is actually needed, at the price of needing to search longer. We have not yet investigated how easily this approach could be integrated into our construction.

Unsolved parts. We call a part *unsolved* if none of its attempts find a solution within the limits on the number of attempts and on the time. In our current implementation, a single unsolved part fails the whole construction. Since we want a practical data structure, this is not acceptable, since we cannot use a function that fails simply by bad luck. A fallback mechanism would introduce only a small overhead if unsolved parts are sufficiently rare. For example, one could use the last attempt index as a marker for a failed part. We then find

a solution for this part with a more expensive data structure that always succeeds; this could be a function with more seed bits per key. The additional space required by this approach is proportional to the fraction of unsolved parts. In our experiments this fraction was at most $0.2\,\%$. Even if the fallback needs 2 bits per key more, it only costs 0.005 bits per key on average. With such a fallback mechanism, the construction would always succeed. The choice of limits would then affect only the time and space requirements, rather than whether a solution can be constructed at all.

More tuning. The parameter analysis we present can be extended. We tested combinations of the number of seeds per bucket σ and the bucket size λ which we found to be working best. However, there could be better combinations for each part size, especially when combining it with the search for a good node limit. Similarly, the DLX search itself offers further opportunities for optimization. These include exploring alternative tie-breaking rules and experimenting with different restart strategies.

Closing. Our results show that a search over small parts with an exact solver can bring the seed information of a minimal perfect hash function close to the information-theoretic limit, and that the remaining distance of the complete function comes mostly from how the parts are combined and not from the search. Closing this distance by compact encodings, and making the search for larger parts cheaper, are the most promising steps towards constructions that are both small and fast to build.

Zusammenfassung

Eine minimale perfekte Hashfunktion bildet n statische Schlüssel bijektiv auf $\{0, \ldots, n-1\}$ ab und benötigt mindestens $\log_2 e \approx 1{,}44$ Bit pro Schlüssel. Wir untersuchen eine Konstruktion, die die Schlüssel in kleine Teile aufteilt, die Schlüssel eines Teils auf Buckets verteilt und für jeden Bucket einen Seed sucht, sodass alle Schlüssel des Teils unterschiedliche Plätze erhalten. Die Wahl der Seeds wird als Problem der unabhängigen Menge bzw. als Exact-Cover-Problem formuliert. Wir lösen es mit Dancing Links und wiederholen die Suche mit neuen Hashfunktionen, wenn keine Lösung existiert. Wir bewerten die Suche und die vollständige Konstruktion experimentell. Die Erfolgswahrscheinlichkeit eines einzelnen Versuchs hängt stark von der Zahl der Seed-Bits pro Schlüssel ab, und durch erneutes Ziehen der Hashfunktionen (Re-Rolls) lassen sich Seeds nahe an der unteren Grenze für einen Teil verwenden. Mit einem einfachen Präfixsummen-Layout benötigt die vollständige Funktion zwischen $2{,}0$ und $2{,}3$ Bit pro Schlüssel, gegenüber der unteren Schranke von $1{,}44$; die Seeds belegen etwa $1{,}5$ Bit pro Schlüssel, und der größte Teil des übrigen Overheads stammt von den Offsets der Teile. Die Konstruktion skaliert linear in der Anzahl der Schlüssel und erzeugt eine Funktion für 10^7 Schlüssel in etwa sechs Minuten. Dancing Links entscheidet die Instanzen eines Teils etwa drei Größenordnungen schneller als ein ILP-Löser und entscheidet innerhalb desselben Zeitlimits mehr davon. Ein Limit von 10^5 Suchknoten pro Versuch macht die Konstruktion bei nahezu gleichem Platzbedarf etwa $2{,}6$-mal günstiger als 10^6 Knoten. Wir diskutieren, wie kompakte Offsets und bessere Kodierungen den Platzbedarf näher an den Stand der Technik bringen könnten.

Bibliography

[1] Djamal Belazzougui, Fabiano C. Botelho, and Martin Dietzfelbinger. Hash, displace, and compress. In Amos Fiat and Peter Sanders, editors, *Algorithms - ESA 2009*, pages 682–693, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-04128-0.

[2] Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. Simple and space-efficient minimal perfect hash functions. In Frank Dehne, Jörg-Rüdiger Sack, and Norbert Zeh, editors, *Algorithms and Data Structures*, pages 139–150, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-73951-7.

[3] Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. Practical perfect hashing in nearly optimal space. *Information Systems*, 38(1):108–131, 2013. ISSN 0306-4379. doi: https://doi.org/10.1016/j.is.2012.06.002. URL https://www.sciencedirect.com/science/article/pii/S0306437912000944.

[4] Zbigniew J. Czech, George Havas, and Bohdan S. Majewski. An optimal algorithm for generating minimal perfect hash functions. *Information Processing Letters*, 43 (5):257–264, 1992. ISSN 0020-0190. doi: https://doi.org/10.1016/0020-0190(92)90220-P. URL https://www.sciencedirect.com/science/article/pii/002001909290220P.

[5] Emmanuel Esposito, Thomas Mueller Graf, and Sebastiano Vigna. *RecSplit: Minimal Perfect Hashing via Recursive Splitting*, pages 175–185. doi: 10.1137/1.9781611976007.14. URL https://epubs.siam.org/doi/abs/10.1137/1.9781611976007.14.

[6] Edward A. Fox, Qi Fan Chen, and Lenwood S. Heath. A faster algorithm for constructing minimal perfect hash functions. In *Proceedings of the 15th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '92, page 266–273, New York, NY, USA, 1992. Association for Computing Machinery. ISBN 0897915232. doi: 10.1145/133160.133209. URL https://doi.org/10.1145/133160.133209.

[7] Michael L. Fredman and János Komlós. On the size of separating systems and families of perfect hash functions. *SIAM Journal on Algebraic Discrete Methods*, 5

(1):61–68, 1984. doi: 10.1137/0605009. URL https://doi.org/10.1137/0605009.

[8] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, San Francisco, 1979.

[9] Marco Genuzio, Giuseppe Ottaviano, and Sebastiano Vigna. Fast scalable construction of minimal perfect hash functions. In *Proceedings of the 15th International Symposium on Experimental Algorithms - Volume 9685*, SEA 2016, page 339–352, Berlin, Heidelberg, 2016. Springer-Verlag. ISBN 9783319388502. doi: 10.1007/978-3-319-38851-9_23. URL https://doi.org/10.1007/978-3-319-38851-9_23.

[10] Ragnar Groot Koerkamp. Ptrhash: Minimal perfect hashing at ram throughput. volume 338, pages 21:1–21:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi: 10.4230/LIPICS.SEA.2025.21. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.SEA.2025.21.

[11] Ragnar Groot Koerkamp, Stefan Hermann, Peter Sanders, and Stefan Walzer. Non-minimal k-perfect hashing: Tight lower bounds and an application to fast static hash tables. volume 388, pages 20:1–20:23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2026. doi: 10.4230/LIPICS.ESA.2026.20. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ESA.2026.20.

[12] Ernestine Großmann, Christian Schulz, Darren Strash, and Antonie Wagner. Data reductions for the strong maximum independent set problem in hypergraphs. *arXiv preprint arXiv:2602.10781*, 2026. URL https://arxiv.org/abs/2602.10781.

[13] Ernestine Großmann, Kenneth Langedal, and Christian Schulz. Accelerating reductions using graph neural networks and a new concurrent local search for the maximum weight independent set problem, 2025. URL https://arxiv.org/abs/2412.14198.

[14] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2026. URL https://www.gurobi.com.

[15] Torben Hagerup and Torsten Tholey. Efficient minimal perfect hashing in nearly minimal space. In Afonso Ferreira and Horst Reichel, editors, *STACS 2001*, pages 317–326, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. ISBN 978-3-540-44693-4.

[16] Stefan Hermann, Hans-Peter Lehmann, Giulio Ermanno Pibiri, Peter Sanders, and Stefan Walzer. Phobic: Perfect hashing with optimized bucket sizes and interleaved coding, 2024. URL https://arxiv.org/abs/2404.18497.

[17] Stefan Hermann, Sebastian Kirmayer, Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer. Engineering minimal k-perfect hash functions, 2025. URL https://arxiv.org/abs/2504.20001.

[18] Richard M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972. ISBN 978-1-4684-2001-2. doi: 10.1007/978-1-4684-2001-2_9. URL https://doi.org/10.1007/978-1-4684-2001-2_9.

[19] Donald E. Knuth. Dancing links, 2000. URL https://arxiv.org/abs/cs/0011047.

[20] Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer. *SicHash - Small Irregular Cuckoo Tables for Perfect Hashing*, pages 176–189. doi: 10.1137/1.9781611977561.ch15. URL https://epubs.siam.org/doi/abs/10.1137/1.9781611977561.ch15.

[21] Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer. Shockhash: Near optimal-space minimal perfect hashing beyond brute-force, 2024. URL https://arxiv.org/abs/2310.14959.

[22] Hans-Peter Lehmann, Peter Sanders, Stefan Walzer, and Jonatan Ziegler. Combined search and encoding for seeds, with an application to minimal perfect hashing, 2025. URL https://arxiv.org/abs/2502.05613.

[23] Hans-Peter Lehmann, Thomas Mueller, Rasmus Pagh, Giulio Ermanno Pibiri, Peter Sanders, Sebastiano Vigna, and Stefan Walzer. Modern minimal perfect hashing: A survey, 2026. URL https://arxiv.org/abs/2506.06536.

[24] Antoine Limasset, Guillaume Rizk, Rayan Chikhi, and Pierre Peterlongo. Fast and Scalable Minimal Perfect Hashing for Massive Key Sets. In Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman, editors, *16th International Symposium on Experimental Algorithms (SEA 2017)*, volume 75 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-036-1. doi: 10.4230/LIPIcs.SEA.2017.25. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.SEA.2017.25.

[25] Bohdan Majewski, Nicholas Wormald, George Havas, and Zbigniew Czech. A family of perfect hashing methods. *Comput. J.*, 39:547–554, 01 1996.

[26] Kurt Mehlhorn. On the program size of perfect and universal hash functions. In *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*, pages 170–175, 1982. doi: 10.1109/SFCS.1982.80.

[27] Rasmus Pagh. Hash and displace: Efficient evaluation of minimal perfect hash functions. In Frank Dehne, Jörg-Rüdiger Sack, Arvind Gupta, and Roberto Tamassia, editors, *Algorithms and Data Structures*, pages 49–54, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-48447-9.

[28] Giulio Ermanno Pibiri and Roberto Trani. Pthash: Revisiting fch minimal perfect hashing. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '21, page 1339–1348. ACM, 2021. doi: 10.1145/3404835.3462849. URL http://dx.doi.org/10.1145/3404835.3462849.

[29] Sean Weaver and Marijn Heule. Constructing minimal perfect hash functions using sat technology, 2019. URL https://arxiv.org/abs/1911.10099.