Karlsruhe Institute of Technology

Cuckoo-PTHash: Exploring Cuckoo Hashing in the PTHash Framework

Bachelor's Thesis of

Benedikt Thomas Waibel

At the KIT Department of Informatics
ITI Institute for Theoretical Informatics

First examiner:	Prof. Dr. rer. nat. Peter Sanders
Second examiner:	T.T.-Prof. Dr. Thomas Bläsius
First advisor:	M.Sc. Hans-Peter Lehmann
Second advisor:	M.Sc. Stefan Hermann

06. February 2024 – 06. July 2024

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Abstract

A minimal perfect hash function (MPHF) is a hash function that maps a known set S of n keys to numbers from $0, \ldots, n-1$ without collision. Construction time, space usage, and query time are relevant performance metrics of MPHF construction. Known methods do not achieve the best results in all dimensions simultaneously. PTHash is a method of constructing a (minimal) PHF. The algorithm first divides keys into small buckets. Afterwards, it searches for one seed per bucket that maps keys from this bucket to yet unused numbers from the output range. PTHash especially focuses on fast queries. Cuckoo hashing is a technique that uses two hash functions to calculate two candidate positions for each key. This allows more flexibility to avoid collisions. Recent advances in the field of retrieval data structures enable space efficient usage of cuckoo hashing for PHF construction. Currently, some methods use this to construct many small PHFs.

In this work we present Cuckoo-PTHash, a (minimal) PHF that uses cuckoo hashing to map keys to the output range. Its construction algorithm builds upon the structure of PTHash, but introduces an additional construction step to resolve the flexibility of cuckoo hashing after completing the search by chosing a candidate for each key. We design a new bucket distribution that aims at reducing the entropy of bucket seeds. Additionally, we present union-find data structures in combination with a filter to be able to use cuckoo hashing efficiently during search.

Our approach shows that cuckoo hashing can also be used for construction of a large PHF, not only for small PHF partitions. Our approach significantly reduces the number of seeds that need to be tried compared to PTHash. We compare different techniques for each construction step and show that the new bucket distribution results in almost minimal seed entropy. The designed data structures and solution procedures can be applied to other methods, as well.

Zusammenfassung

Eine Minimale Perfekte Hashfunkion (MPHF) ist eine Hashfunktion, die eine bekannte Menge S an n Keys ohne Kollisionen auf die Zahlen $\{0, \ldots, n-1\}$ abbildet. Die Konstruktionszeit, der Speicherbedarf und die Zugriffszeit sind relevante Leistungsmaße von MPHF Konstruktionsmethoden. Bekannte Methoden erreichen nicht in allen Dimensionen zeitgleich die besten Ergebnisse. PTHash ist eine Methode zur Konstruktion einer (minimalen) PHF. Der Algorithmus teilt die Keys zunächst in kleine Buckets und sucht anschließend für jeden Bucket einen Seed, der Keys aus diesem Bucket auf bisher unbelegte Werte aus dem Ausgabebereich abbildet. PTHash fokussiert sich besonders auf schnelle Zugriffe. Cuckoo hashing ist eine Technik, die zwei Hashfunktionen nutzt, um zwei Kandidaten für die Position jedes Keys zu berechnen. Dies ermöglicht mehr Flexibilität bei der Vermeidung von Kollisionen. Durch jüngste Fortschritte bei Retrieval-Datenstrukturen lässt sich Cuckoo Hashing platzeffizient für Perfect Hashing nutzen. Bisher nutzen dies einige Methoden für die Konstruktion vieler kleiner PHFs.

In dieser Arbeit präsentieren wir Cuckoo-PTHash, eine (minimale) PHF, die Cuckoo Hashing verwendet, um Keys auf den Ausgabebereich abzubilden. Dessen Konstruktionsalgorithmus baut auf der Struktur von PTHash auf, ergänzt aber einen weiteren Konstruktionsschritt, um die Flexibilität von Cuckoo Hashing nach Abschließen der Suche durch Wahl eines Kandidaten für jeden Key aufzulösen. Wir entwerfen eine Bucket-Verteilung, um die Entropie der Bucket-Seeds zu verringern. Außerdem präsentieren wir Union-Find Datenstrukturen in Kombination mit einem Filter, um Cuckoo Hashing effizient bei der Suche nutzen zu können.

Unser Ansatz zeigt, dass Cuckoo Hashing auch für die Konstruktion einer großen PHF geeignet ist, nicht nur für kleine PHF-Partitionen. Dabei muss unser Ansatz im Vergleich zu PTHash bedeutend weniger Seeds durchprobieren. Wir vergleichen verschiedene Techniken für jeden Konstruktionsschritt und zeigen, dass die neue Bucket-Verteilung nahezu minimale Seed-Entropie liefert. Die von uns entworfenen Datenstrukturen und Lösungsverfahren lassen sich auch für andere Methoden nutzen.

Contents

1. Introduction

A *Perfect Hash Function (PHF)* is a hash function that maps a known set of keys S to numbers from $\{0, \dots, m-1\}$ without collisions. Multiple practical algorithms for constructing a PHF are known. Known construction algorithms focus either on construction time, query time or space efficiency. While there are methods that perform well in multiple dimensions, nonoe of them achieves the best results in each dimension simultaneously.

Cuckoo Hashing uses 2 hash functions h_0, h_1 to resolve collisions. Each key can be stored at one of its candidate positions $h_0(k), h_1(k)$. Cuckoo hashing combines both functions into a single hash function h by choosing a function for each key k. To build a PHF with cuckoo hashing, both hash functions h_0, h_1 map into the same output range, and the chosen hash function $d(k) \in \{0, 1\}$ that avoids collisions needs to be stored.

A *retrieval data structure* maps each key k from a set S to an r-bit value $f(k) \in \{0, 1\}^r$. Recent developments of succinct retrieval data structures enable a promising new approach for PHF construction. Using a retrieval data structure, a PHF constructed with cuckoo hashing can be encoded compactly.

This work combines ideas from two perfect hashing methods, PTHash which has good query times and ShockHash which is space efficient. PTHash maps keys to buckets, sorts the buckets, and performs a brute-force search for a displacement value, called *pilot*, for each bucket. PTHash fills the output range bucket by bucket. A pilot for a bucket is a successful choice if there are no collisions with previously inserted keys and within the new bucket. ShockHash constructs a space efficient PHF by filling small cuckoo hash tables.

We present a new perfect hashing approach. Construction uses the 3 step framework of PTHash, and an additional step to choose the hash function for each key out of the cuckoo hashing candidates. Keys of a bucket are mapped to a position using the seed of the bucket and cuckoo hashing.

1.1. Minimal Perfect Hashing

Given a hash function $h \colon U \to \{0, \dots, m-1\}$, $S \subseteq U$ is a known subset of n keys from the universe U and m is the size of the output range with $n \leq m$. The function h is a *Perfect Hash Function* (PHF) if it maps all keys in S without collision, i.e. h is injective on S. For $n = m$ a PHF h is further called a *Minimal Perfect Hash Function* (MPHF). In this case, the MPHF h is a bijection on S. We will write $\{0, \dots, m-1\}$ as $[m]$ for brevity. We will often write $h \colon S \to [m]$, as we allow the hash function to have any value for keys $k \notin S$. We call

$\alpha = n/m$ the *load factor* of h, analogous to the load factor of a hash table. For an MPHF, we have $n = m$ and $\alpha = 1$. In this case, we will use n as the number of keys and the size of the output range.

For small n, an MPHF can be obtained by trying hash functions by brute force until a function without collisions is found. This approach does not scale well because the number of hash functions that need to be tried using this approach grows exponentially with n. Instead, practical algorithms for high n need a different approach, and they consist of more than finding one seed that hashes every key without collision.

After construction, a PHF needs to be encoded. Because practical solutions result in more complex constructions than selection of a single seeded hash function, storing a PHF might require significantly more storage space. To achieve an efficient encoding, we exploit the fact that behaviour on keys $k \notin S$ is not defined. The hash function can return any value for such keys. Assuming there are no collisions, we do not need to store keys $k \in S$. The hash value for each key is unique and can be used as identifier in applications. A space optimal encoding only needs to store a minimal representation of the PHF. The asymptotic minimal space usage of an MPHF encoding is $\log_2 e \approx 1.44$ bits/key [18, 27].

Ideally, a solution to the perfect hashing problem should construct a PHF that can be queried in constant time. Additionally, we desire space efficient encoding of a found PHF. For certain smaller load factors, efficient algorithms for PHF construction are known. For $\alpha \to 1$, efficient construction is increasingly challenging. Over years of research, different solutions for this problem have been developed. Because construction is easier for higher storage space or higher query times, solutions usually have a trade-off between being more compact or having a more efficient construction. Finding algorithms with better space-time trade-offs is still a focus of current research. We discuss notable algorithms for finding a PHF in **??**.

(Minimal) perfect hashing has a wide range of applications. It is especially useful if space overhead for the data structure is limited, and if the number of query accesses is high compared to re-constructions. For example, it is used in compressed full-text indexes [2], databases [5], prefix-search data structures [3] and indexes for DNA [29].

1.2. Contribution

The use of a relatively new succinct retrieval data structure [9] enables new approaches in perfect hashing. This work explores the use of cuckoo hashing in the PTHash framework. It contributes the following:

- A configurable (minimal) perfect hash function implementation
- Design of a new bucket mapping function for our setting, reducing the entropy of seeds
- Significant reduction of required brute-force through use of cuckoo hashing

- Design of reversible union-find data structures and a lazy filter to efficiently test if a graph is a pseudo-forest
- An experimental evaluation measuring the effectiveness of the designed algorithms in practice and comparing the MPHF against its predecessor PTHash and against SicHash

1.3. Outline

The outline of this thesis is as follows. Chapter 2 explains the fundamentals of cuckoo hashing, ShockHash, and PTHash, as well as the used retrieval data structure and used encodings. Chapter 3 discusses related work in the fields of cuckoo hashing and (minimal) perfect hashing. Chapter 4 presents the designed algorithms in depth. The next Chapter 5 discusses the implementation and names notable implementation details. The implementation is evaluated and compared against other algorithms. This evaluation is grouped in Chapter 6. Finally, we present the results of this work with a conclusion and ideas for future work in Chapter 7.

2. Preliminaries

In this chapter, we introduce fundamental concepts that are used by our approach. We start by discussing cuckoo hashing, the core technique of our approach. Afterwards we discuss ShockHash and PTHash, two perfect hashing methods that provide ideas for our new method. Lastly, we discuss retrieval and encodings. Both are necessary to store a PHF compactly after construction.

2.1. Cuckoo Hashing

Cuckoo hashing [28] is a well-known method for collision resolution in hash tables. The original method uses two hash functions h_0, h_1, each hashing into a different table of $\frac{m}{2}$ cells. This gives each key two cells where it can be stored. Lookup is possible in constant time. To find the cell of a key, one simply has to evaluate both hash functions and look up the determined cells. Deletion is possible in constant time in a similar manner, unless the table sizes need to be scaled down.

Upon insertion, we evaluate both hash functions, resulting in two candidate cells. If one of the cells is empty, we place the value there and the insertion is done. Otherwise, we store the key in its candidate cell from the first table. We now need to store the key that was previously stored in that cell somewhere else. We do this by evaluating the other hash function for the key, and store it in its alternative cell in the other table. This process of kicking out the previously stored values gives the method its name. We perform a recursive insertion with kicked out keys until we find a free cell. This process is depicted in Figure 2.1.

Insertion fails if we reach a maximum number of recursions, or if we try to insert a key into a cell with the same hash function for a second time. If we did not try the second hash function for our new key yet, we can retry starting with the resulting second candidate cell.

In the case of failure, we need to rehash the tables with different hash functions. Rehashing can be done by looping over the tables, deleting and re-inserting every key that is not at the correct location with the normal procedure. Rehashing with different hash functions on failure is usually done by choosing a different seed for two seedable hash functions.

Up to a certain load factor α, the success probability of insertion tends to 1 for $m \longrightarrow \infty$ [10, 15, 16, 26]. The asymptotic load threshold is the supremum c where the above is true for any $\alpha < c$. Also, for $\alpha > c$, the success probability of insertion tends to 0. For cuckoo hashing

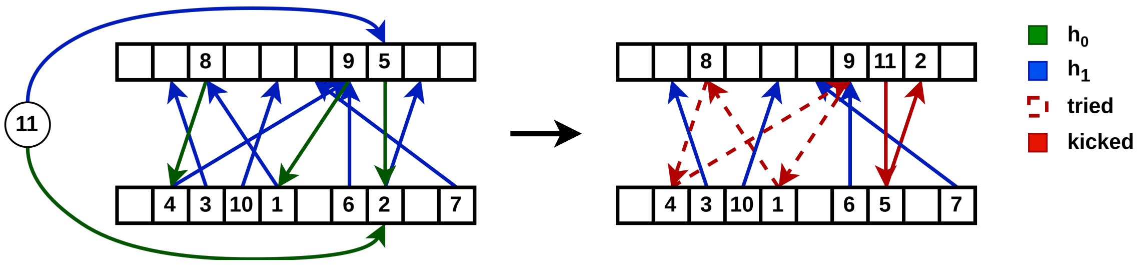


Figure 2.1.: Insertion process of a cuckoo hash table. Left: before the insertion. Right: after key 11 is inserted successfully. Keys that were visited or kicked out to their alternative position are marked in red.

with two hash functions, this threshold is $c = 0.5$. When a cuckoo hash table is filled above the load threshold, the success probability quickly drops.

Cuckoo Hashing [28] is a well-known method for handling collisions in hash tables, and variations of the method have been researched. The original work also mentions a variant where both hash functions map into the same table. It further proposes an asymmetric scheme where the first table is larger than the second. There are generalizations of cuckoo hashing that allow multiple elements per cell up to a constant number of elements [7]. Cuckoo hashing can also be generalized to use more than two hash functions [14]. For distinction, the case with two hash functions is also called *binary* cuckoo hashing. It is also possible to choose a different number of hash functions for each key. This is called *irregular* cuckoo hashing [8].

For perfect hashing, we will use a variant of cuckoo hashing with a single table of size m. The theoretical properties are similar to the variant with two tables. The most notable difference is that both hash functions could map to the same cell, leaving us with only one candidate cell. This event is less likely for high m. The hash function chosen for a key cannot be encoded by storing the key with the cell. This would hinder efficient encoding of the hash function. Encoding this information using another hash table would not be efficient either. Therefore, a different approach is needed. We will apply a retrieval data structure to store which of the two hash functions is used (see Section 2.4).

2.1.1. Graph interpretation

The state of a cuckoo hash table can be interpreted as a graph $G = (V, E)$ [6, 10]. We will use this interpretation as it is helpful for checking if a cuckoo hash table can be constructed and for theoretical considerations. The vertices V are the set of cells. For each key k in the hash table, there is an edge $\{h_0(k), h_1(k)\}$ in the graph. The graph can have multi-edges as multiple keys could have the same candidate cells. In case of the variant with one large table, it can also have self-loops. An example for an equivalent table and graph are shown in Figure 2.2.

Above, the graph is given in its undirected form. The undirected form of the graph is equivalent to an abstraction of a cuckoo hash table. We only consider the candidate cells for each key, but do not choose the candidate. We call this abstraction *cuckoo graph*. It is advantageous for perfect hashing as we keep the flexibility of cuckoo hashing, but have enough information to test whether collisions can be avoided. The directionality of the graph carries information. Directing an edge is equivalent to choosing the cell of a key. Afterwards, the edge points at the cell of the key.

A cuckoo graph is *without conflict* if it can be used for construction of a cuckoo hash table. The problem of testing a cuckoo graph for this property is then equivalent to testing if the graph can be directed such that the in-degree of each node is at most 1. We call such a directed graph *1-oriented*. A node with an in-degree of ≥ 2 would be equivalent to a cell with a collision. A graph can be 1-oriented if and only if it is a *pseudo-forest*. A pseudo-forest is a graph where each component is a *pseudo tree*. A pseudo tree is either a tree or a tree with one cycle. A pseudo tree can equivalently be defined as a graph where an edge e exists such that: $G' = (V, E \setminus \{e\})$ is a tree. Linear time algorithms for testing if a graph is a pseudo-forest exist. This can be done by testing that the number of edges e and nodes v fulfil $e \leq v$ for each component [33].

2.2. ShockHash

ShockHash [24] is an algorithm for building an MPHF. The name ShockHash is short for: **S**mall, **h**eavily **o**verloaded cu**ck**oo **Hash** tables. ShockHash uses binary cuckoo hashing to build PHFs, storing the hash function choice in a 1-bit retrieval data structure.

Cuckoo hashing allows for a graph interpretation of the perfect hashing problem: finding a PHF is equivalent to finding an n-edge graph that is a pseudo-forest. Asymptotically, the success probability of finding a PHF with cuckoo hashing is high for load factors below $c = 0.5$. For higher load factors, the success probability tends to zero. For a small table size n, it is feasible to retry building the hash table with a different seed. ShockHash fills hash tables of size n with n elements. In expectation, construction succeeds after $(e/2)^n \text{poly}(n)$ seeds. The success probability is higher than the plain brute-force construction, by a factor of 2^n. That means, ShockHash has to try a significantly smaller number of seeds before succeeding.

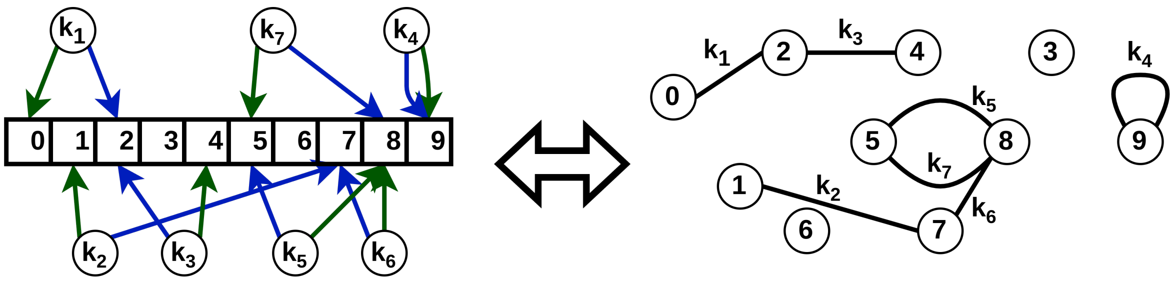


Figure 2.2.: Left: a cuckoo hash table with one table and the candidate position for each key. Right: the equivalent cuckoo graph

As result, ShockHash produces an MPHF $h_{f(x)}(x)\colon S \longrightarrow [n]$, where $f\colon S \longrightarrow [2]$ specifies which hash function is used for keys in S. Successful seeds are recorded, requiring $0.44n{+}o(n)$ bits with Golomb-Rice code [20, 31]. The function f is stored as a 1-bit retrieval data structure, which requires $n + o(n)$ bits. In total, ShockHash is very close to the asymptotic minimal space usage with $1.44n + o(n)$ bits.

ShockHash has provable theoretic properties. The paper [24] gives two lower bounds for the success probability of a single seed. The first lower bound is less sharp by a factor of $O(\sqrt{n})$ and therefore omitted here. Arguments are made using the graph representation. A seed corresponds to a random graph G which may contain self-loops and multi-edges. We say "the seed is successful" exactly if the graph is 1-orientable, i.e. if the graph is a pseudo-forest.

Theorem 2.2.1 *Let $h_0, h_1\colon S \longrightarrow [n]$ be uniformly random functions. The probability that there exists $f\colon S \longrightarrow [2]$ such that $x \mapsto h_{f(x)}(x)$ is bijective is at least $(e/2)^{-n}e^{-1}\sqrt{\pi}$.*

For the proof, please refer to the original paper [24]. From Theorem 2.2.1 follows that in expectation $(e/2)^n e/\sqrt{\pi}$ seeds need to be tried to succeed. Testing whether a seed generates a pseudo tree can be tested in linear time, resulting in construction time of $O((e/2)^n n)$.

ShockHash provides a building block for hash functions as an alternative to brute-force. It is applied as base case of RecSplit (see **??**). The paper [24] suggests applying ShockHash to hash functions based on Hash-and-Displace like PTHash (see Section 2.3). This can be done by searching for pseudo-forests instead of directly searching for PHFs. We realize this suggestion in this work.

2.3. PTHash

PTHash [30] is an algorithm for minimal perfect hashing. For construction, the algorithm splits the input keys into buckets and uses a technique called *hash-and-displace* on the subproblems. The name means **P**ilot **T**able **Hash**ing. During construction, the algorithm searches for a seed, called *pilot*, for each bucket to successfully displace the values in a bucket. The construction can be described as a three step framework of *mapping*, *ordering* and *searching*.

In the first step, *mapping*, the algorithm maps keys to buckets using a hash function and a global seed. This mapping uses a skewed distribution where the first 30% of the buckets hold roughly 60% of the keys. This results in a front part with few large buckets and a back part with many smaller buckets. The algorithm introduces a parameter c to control the balance between storage space and construction time. It reserves $b = \lceil \frac{cn}{\log_2 n+1} \rceil$ buckets, for higher c more space is needed. The parameter c can be used to calculate the expected average bucket size $\lambda = \frac{\log_2 n+1}{c}$. Because this parameter is more intuitive and because we already use c for the load threshold, we will use the parameter λ from here on. A key k is mapped to

$$bucket(k) = \begin{cases} h(k,s_1) \mod p_2, & h(k,s_1) \mod n < p_1 \\ p_2 + (h(k,s_1) \mod (b - p_2)), & \text{otherwise,} \end{cases}$$

where $p_1 = 0.6n$, $p_2 = 0.3b$, s_1 is a global seed, and h is a seeded hash function.

In the *ordering* step, the buckets are sorted by non-increasing size. Since the success probability for finding a PHF decreases with a growing load factor, sorting ensures that large buckets are processed first when the success probability is still high.

The *searching* step takes up the majority of the construction time. The algorithm uses an additional global seed s_2 and ensures that hash codes $h(k, s_2)$ are distinct within each bucket. If this is not the case, the algorithm is restarted with a new global seed s_2. Otherwise, the algorithm searches for an integer p_i for each bucket B_i. This integer p_i is called the *pilot* of the bucket and determines the displacement of keys. The position of a key can be calculated as $position(k) = (h(k,s_2) \oplus h(p_i,s_2)) \mod m$. A pilot is successful if none of the positions it maps keys from the bucket to is taken by a key from a previous bucket. Pilot candidates are the natural numbers including 0. PTHash tries these numbers successively in increasing order.

To improve space efficiency, PTHash encodes the pilots in the final representation. Different encoding methods result in a different space-time trade-off. For encoding, PTHash supports compact encoding, a dictionary-based encoding, Elias-Fano and the Simple Dense Coding (SDC) (see Section 2.5). Different encodings can be chosen for the front and the back (the few larger and many smaller buckets). The buckets in the front part contain a lot of keys and have lower expected seeds. Using a different encoding method for each part allows for encoding the front part with a simpler encoding, resulting in a lower request overhead for many keys. A PHF with $\alpha < 1$ is not minimal. PTHash can convert a PHF with an α close to 1 into an MPHF by mapping the additional positions to free cells in the minimal output range.

The authors give an extensive evaluation of PTHash based on experiments and benchmarks. For the detailed results, refer to the original paper [30]. The performance of PTHash depends on the parameters λ and α. For higher λ and α, the construction time increases and the required space decreases. PTHash does not get close to the theoretical lower bound of 1.44 bits/key because the search costs grow exponentially. Configurations resulting in < 2 bits/key have impractical construction times. Giving PTHash more storage space reduces the construction time. The query time can be reduced by choosing a slightly less space efficient encoding with quicker decoding.

2.4. Retrieval

A *retrieval data structure* (or *static function data structures*) represents a function $f : S \to \{0,1\}^r$, where $n = |S|$. For each value $k \in S$ it returns the r-bit value $f(k)$, for values

$k \notin S$ it may return an arbitrary r-bit value. Because the behaviour on values $k \notin S$ is not specified, the retrieval data structure does not need to encode the set S. This way, the minimal theoretic space requirement is rn bits. Practical algorithms achieving $rn + O(rn)$ with linear construction time and constant time queries are known.

For our case of $r = 1$, we can use *Bumped Ribbon Retrieval (BuRR)* [9]. BuRR also support $r > 1$. It has a space overhead of about 1%. Queries are performed by evaluating a hash function $h(k)$, calculating the XOR with a segment of rows from a precomputed table, and returning the parity of the result. Construction of the table is possible by solving a near diagonal system of linear equations. The system results in a matrix wich has all 1-entries in a narrow *ribbon* close to the diagonal. *Bumping* refers to the practice of encoding a small part of the keys in an additional layer retrieval data structure. Keys that would result in an unsolvable system can be bumped to a later layer. BuRR requires a constant number of bumping layers and thereby maintains constant access time.

2.5. Encodings

Encodings define how a list of values are represented in bytes. A single value can be retrieved from the encoded form by evaluating the corresponding decoding algorithm. Encodings can reduce the required storage space compared to simply storing the values as is in their initial order. A list of values usually has redundant information like repeated values or similarities. Encodings can use such relations to achieve a more compact representation. Most encodings perform better on inputs with lower entropy.

A very simple encoding method is *compact encoding* [30]. Compact encoding means encoding each value with the number of bits required to store the maximum value. *Dictionary-based* encoding makes use of repeated values in the input [30]. The dictionary-based method stores distinct values in an array, the *dictionary*, and stores each value as an index into this array. *Elias-Fano* [11, 13] encoding can efficiently store a monotonic sequence of integers. The method can be used to encode the prefix sum of seeds, which is such a monotonic sequence. *Simple Dense Coding (SDC)* [19] is a compression method that sorts values by frequency and encodes values with higher frequency with less bits than values with low frequency. PTHash supports the above mentioned encoding algorithms. *Golomb-Rice* [20, 31] encoding can efficiently store integer values that have a geometric distribution.

The input can be split into multiple partitions that are encoded separately. Partitioning can be beneficial if distributions of values are different, or if the entropy of values in a partition is expected to be lower than the entropy on the whole input. Partitioning can use a fixed number of partitions, fixed size partitions or dynamic parameters based on the input. PTHash also implements *partitioned compact* encoding, a partitioned variant of compact encoding using a fixed maximum input size for partitions.

3. Related Work

Many different MPHF algorithms have been developed in the past. A few notable ones are mentioned here.

This work is based on the ideas of PTHash [30]. There are other algorithms that are based on a hash-and-displace strategy as well. PTHash is based on an algorithm from *Fox, Chen and Heath (FCH)* [17]. FCH structures the construction into mapping, ordering and searching as well. However, FCH uses addition instead of XOR for displacement and reserves a space budget at the start to store displacement values in an array. Compared to our method, FCH achieves fast query times, but has high construction times for a given space budget. Another algorithm based on the hash-and-displace strategy is *compressed hash-and-displace (CHD)* [1]. Different to PTHash, the CHD algorithm uses a uniform bucket distribution. CHD also stores the seed for each bucket as compressed sequence, but its queries are slow in comparison to other methods.

The recently presented algorithm PHOBIC [21] improves several aspects of PTHash while maintaining its low query time. The method uses an optimized bucket function to reduce the seed entropy. The bucket function of PHOBIC is designed for the setting of PTHash. We apply the idea to our setting and design new bucket functions.

ShockHash (see Section 2.2) is not the only technique that uses cuckoo hashing and retrieval to build a PHF. SicHash [25] uses irregular cuckoo hashing and overloads small hash tables. To build a PHF from many small tables, an offset for each table is stored as a prefix sum of table sizes. In contrast, the method described here builds one large table from buckets of keys.

The algorithm RecSplit [12] solves the perfect hashing problem by recursively splitting the keys into two groups. It does this until a hash function for the remaining keys can be found efficiently with brute-force. This approach increases the query time because the splits need to be calculated to determine the hash function for a key. However, the technique can reach a space usage close to the lower bound with feasible construction time. Additionally, the base case of RecSplit can be replaced with Shockhash. This variant is called Shockhash-RS [24]. In this case, the algorithm builds many small cuckoo hash tables and needs to try a smaller number of seeds in expectation.

4. Design

In this thesis, we present a new PHF. The new hash function builds upon ideas of PTHash. We take the three construction steps (*Mapping*, *Ordering* and *Searching*) from that method. We redesign the search completely to employ cuckoo hashing. We rework the other existing steps or extend them by alternative approaches. Moreover, we introduce an additional step, called *Directing*, to facilitate the usage of cuckoo hashing. This fourth step chooses the used hash function for each key, thereby directing the edges in the *cuckoo graph*.

The design of the hash function enables configuration of the input parameters *load factor* α and average bucket size λ. Since the construction of an MPHF is the focus of this work, the design is laid out for the case $\alpha = 1$. Smaller values of α can be chosen, but result in a non-minimal PHF. The choice of λ impacts the tradeoff between construction time and required space. A higher value usually increases search costs, but often also results in more compact hash functions. In addition to the choice of input parameters, alternatives for different steps can freely be combined.

The design of the new PHF aims for the following goals:

- utilize the advantage of an increased success probability for inserting a key through cuckoo hashing
- enable fast queries, avoid deterioration compared to PTHash where possible
- enable compact encoding of the hash function, keep the entropy of seeds low

4.1. Framework

The construction of the hash function is divided into four steps, *Mapping*, *Ordering*, *Searching* and *Directing*. Each step serves a fixed purpose. At the same time, a choice of alternative approaches is presented for each step.

The *Mapping* step maps each key k to a bucket $b(h_k)$. Input for *Mapping* is an initial hash $h_k = h(k, s_1)$, where s_1 denotes a global seed. The number of buckets calculates as $b_{\max} = \lfloor \lambda n \rfloor$. Caused by the pseudo-random input, the output is a random distribution of bucket sizes. We can freely chose the expected bucket sizes. This choice is an optimisation problem. We design multiple distributions as explained subsequently. The first distribution maps keys uniformly random to the buckets. An idea of PTHash is a skewed distribution. The buckets are split into a larger and a smaller subset. A large part of the keys is mapped to the smaller subset of buckets. We provide a skewed distribution with adjusted parameters.

Because of the load threshold of $c = 0.5$ for cuckoo hashing, we map 50% of the keys to 5% of the buckets. Because a high success probability on the first key half is to be expected with cuckoo hashing, we can use an even smaller portion of the buckets for these keys. Moreover, we design multiple distributions that aim at balancing the expected work per bucket. This approach is discussed in detail in Section 4.2.

The *Ordering* step sorts buckets by desired criteria. After *Mapping*, buckets are sorted by their ID. This step can specify a new bucket order. In the following steps, this will be the order of processing. A supported option for ordering the buckets is sorting them by decreasing bucket size. This is the order chosen by PTHash. The success probability for a seed decreases with growing load. The idea behind this order is to reduce the search costs by processing larger buckets first when the probability is still high. In this work, we sort secondary by bucket ID. This enables a secondary criterion by giving the bucket ID a special meaning. Alternatively, the step can also be skipped. In this case, the buckets are passed on in their initial order. This variant is easy to implement and saves the sorting costs. Initial measurements show that skipping this step is not worth it. The saved run time of sorting algorithms is small compared to the additional search costs without *Ordering*. This variant only emphasizes the importance of this step. In the following, this alternative is neither used nor discussed further.

The third construction step is called *Searching*. We search for a seed $s_{2,b(h_k)}$ per bucket that successfully maps this bucket. The crucial difference compared to other approaches is the use of cuckoo hashing for this mapping. This assigns two candidate positions $p_d = h_d(h_k, s_{2,b(h_k)}), d \in \{0, 1\}$ to each key. The whole search process keeps these candidate positions to fully utilize the flexibility introduced by cuckoo hashing. The final position of a key is later chosen from these candidates. A new bucket extends the previous *cuckoo state*. This state can be used to test if construction of a PHF is possible with a seed or if it results in a conflict. Testing if a seed is successful is more complicated in comparison to PTHash. While PTHash can simply test multiple positions in a bitmap, with cuckoo hashing, we need to test if a seed leads to successful hash table construction. This check is equivalent to testing if the *cuckoo graph* is a *pseudo forest* [24]. For this test, the candidate positions of keys from the current bucket need to be added to the state. In case of a conflict, the previous state needs to be used to retry with a new seed. For this purpose, we design three different union-find data structures that support a reversion. These variants are explained in detail in Section 4.3. Alternative approaches for the test are insertion into a cuckoo hash table or a connected components algorithm. These alternatives are not supported here because they are not promising for our construction. The former gives up the flexibility of cuckoo hashing for previous buckets, the latter does not preserve state between iterations. Initial measurement showed that accesses to the union-find data structure make up the majority of the search time. First passing seeds through a filter reduces the amount of expensive accesses. The filter rejects a part of the seeds that will with certainty result in a conflict. Details on the filter are given in Section 4.4. The search considers seeds in ascending order, starting with 0. The first successful seed is chosen. Our approach therefore uses a *greedy* strategy. Because of the order of considering seeds, the probability for a smaller seed is higher and the entropy is reduced. This is important for encoding bucket seeds after construction. In contrast to PTHash, we do not use a *displacement* strategy. The goal of such a strategy is to be able to try many different positions for a new bucket with few hash function evaluations. Initial measurement showed that evaluating hash functions only makes up a small part of the search time. For this reason, we do not employ such a strategy.

We introduce the step *Directing* to be able to use cuckoo hashing efficiently. During search we keep track of both candidate positions for each key. When reaching this step, each bucket has a seed so that the *cuckoo graph* is a *pseudo forest*. In this step, we assign a specific position to each key by chosing one of the hash functions to map it to a cell. Chosing a hash function for each key is equivalent to directing the *cuckoo graph*. For directing, we support an insertion algorithm that inserts the keys into a cuckoo hash table. This option is easy to implement, but its run time depends on the component sizes. This step concludes the construction.

After construction, bucket seeds and chosen hash functions per key are encoded. This step aims at reducing the space consumption of the PHF. For the encoding of the seeds, we keep the encoding algorithms used by PTHash. A division into *front* and *back* encoding is possible as well, but our approach splits both parts at half of the keys. Which hash function was chosen for each key can be represented with a single bit per key. Retrieving the correct bit information for each key is no trivial problem, but it can be solved with a retrieval data structure. By using *BuRR* [9], the storing of this information takes 1 bit/key with an overhead below one percent.

4.2. Improved Bucket Distributions

Until now, we assumed that we can freely choose the distribution of bucket sizes. More precisely, we chose a bucket function $b : U \rightarrow \{0, \dots, b_{\text{max}}\}$. The specific chosen bucket function b determines the pseudo-random distribution of keys to buckets. In the following, we will discuss the bucket function. The bucket function impacts the expected bucket sizes, the success probability of a bucket and the expected number of retries. A well chosen bucket function can decrease the expected search costs, and it can result in seed distributions with a more compact encoding. We discuss properties of a good bucket function and calculate a good bucket function for Cuckoo-PTHash.

Let $\alpha : [0, 1] \rightarrow [0, 1]$ be the function mapping portion of processed buckets to the *load* of a hash table[21]. Then, the derivative α' is the function of bucket sizes as portion of the table size. The probability $p(\alpha(x))$ of successful insertion of a single key with a seed can be expressed using α. This probability decreases with increasing *load*. We will use the shorthand $y = \alpha(x)$ for the *load*. Because the probability is the highest at the start of each bucket, we get an upper bound for the successful insertion of a bucket as: $p(y)^{\alpha'(x)}$. The multiplicative inverse of the probability is the expected number of tries to find a successful seed. We obtain a lower bound for this value: $p(y)^{-\alpha'(x)}$. Because we try seeds in ascending order, starting with 0, we obtain the expected seed by subtracting 1 from the expected tries. Our goal is a low entropy of bucket seeds because this is favorable for encoding. For this goal, a bucket function is optimal if it keeps the work per bucket constant. We obtain the equation $p(y)^{-\alpha'(x)} \overset{!}{=} c, c \in \mathbb{R}$. This is a differential equation due to the occurence of α in base and exponent. Specific solutions depend on the probability function p. To map a hash value to its bucket, we require the inverse function s^{-1} of a solution s. For a normalized solution, we get $b(h_k) = s^{-1}(\frac{h_k}{h_{\max}})b_{\max}$.

PHOBIC [21] calculates and presents such an optimal bucket function for the setting of PTHash. The probability for PTHash is $p(y) = 1 - y$. This yields the optimal bucket function:

$$s^{-1}(x) = x + (1 - x) \log_e(1 - x) \tag{4.1}$$

For the detailed deduction and proofs of optimality refer to the original paper.

The probability function grows more complex through the use of cuckoo hashing. One option for modeling the probability is to determine if other keys with a shared candidate position can evade to their alternative position. This depends on the number of keys that share a candidate position and the probability that an alternative cell is empty. This approach results in a recursive definition over a probability distribution. Finding a closed form of a solution of the differential equation using this function is impractical. The probability is equivalent to the probability that one of both candidate cells does not belong to a pseudo tree. The probability of a single cell belonging to a pseudo tree is exactly the portion of pseudo tree cells. As a result, we get $p(y) = 1 - p_{\text{pseudo}}(y)^2$. Calculating the probability p_{pseudo} can result in similar problems as in the first approach.

Instead of precisely calculating the expected number of pseudo tree cells, we measure this value experimentally during construction. We can estimate the expected value using samples. We observe that the portion of pseudo tree cells is close to 0 for loads below 0.5. For higher loads, this portion increases, but is always below the identity $f(y) = y$. The actual function can be approximated with polynomials f. For $y < 0.5$ we approximate as $f(y) = 0$. This approximation is equivalent to the assumption that nearly no retries are required on the first half of keys. We assign a small portion of the buckets for this half. The chosen portion is 5% of the buckets. Keys in this half are uniformly distributed to buckets.

For $y \geq 0.5$, the approximations vary depending on the degree of the polynomial. Boundary conditions are chosen for the development of the number of pseudo tree cells observed in experiments. For degree 1, we obtain $f(y) = 2y - 1$ with $f(0.5) = 0, f(1) = 1$. For degree 2, we get $f(y) = -2y^2 + 5y - 2$ with the additional condition $f'(1) = 1$. Both polynomials map inputs from the interval $[0.5, 1]$. Because we control the input, we normalize inputs to the interval $[0, 1]$. This results in the transformed polynomials $f(y) = y$ and $f(y) = -\frac{1}{2}y^2 + \frac{3}{2}y$. Inserting the polynomials into $p(y)$ and solving the differential equation yields solutions for the bucket function. We get Equation (4.2) for the first degree approximation and Equation (4.3) for the second degree approximation:

$$s^{-1}(x) = 2\tanh^{-1}(x) - 2x - x \log_e(1 - x^2) \tag{4.2}$$

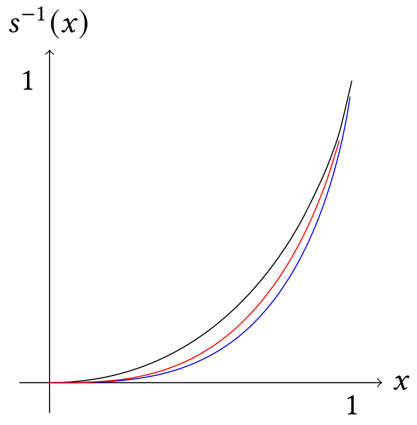


Figure 4.1.: Bucket Function Solutions for PHOBIC and Cuckoo-PTHash. Black: PHOBIC. Blue: solution for first degree approximation. Red: solution for second degree approximation.

$$\begin{aligned} s^{-1}(x) = &- \frac{1}{2}(3+\sqrt{17})\log_e(3+\sqrt{17}-2x) - \log_e(1-x) - 2\log_e(2-x) \\ &+ (18 + \frac{32}{663+161\sqrt{17}} - \frac{72}{\sqrt{17}})\log_e(-3+\sqrt{17}+2x) \\ &- 4x + x\log_e(1 - \frac{(-3+x)^2x^2}{4}) \end{aligned} \tag{4.3}$$

The normalized Solutions for PHOBIC and Cuckoo-PTHash are displayed in Figure 4.1 We can see that increasing the degree of the polynomial results in a sizable increase of summands in the solution. Additionally, the cost of normalizing the solution increases. At the same time, the curves of different solutions are similar to each other. For this reason, polynomials of higher degree than 2 were not considered. Cuckoo-PTHash supports the bucket function of PHOBIC and the bucket functions resulting from the approximations. Additionally, we design a variant that uses large buckets for the first half of keys. For the second half, it uses the solution function of PHOBIC. We call this variant Cuckoo-PHOBIC. This variant is designed to utilize the similarity of the curve despite the much simpler formula for PHOBIC.

4.2.1. Secondary Bucket Order

The bucket functions determine the expected bucket size for each bucket. Depending on specific keys and the chosen global seed, the actual size of a bucket can differ from the expected value. It is impossible to avoid this fully, not least because expected bucket sizes can take any positive number while actual bucket sizes are always integers. One of the designed *Ordering* variants allows for defining a secondary sorting criterion by chosing the initial bucket order. The variant then sorts secondary by bucket ID. The bucket function determines the initial bucket order and can be chosen differently. Cuckoo-PTHash starts with buckets with the lowest expected bucket size. Consequently, for two buckets of the same actual size, the bucket with the lower expected size is processed first. Buckets with a higher actual size than expected size will likely need more retries than planned for. At the same time, for multiple buckets of the same actual size the expected work is the highest for the last bucket. This is caused by the decreasing success probability. The order mentioned above aims at balancing out both effects. The advantage of this order is especially visible for buckets with an expected size close to 0. Often, these buckets contain no key, can be processed without retries and can be encoded very compact. If one such bucket does contain a key, the seed needs to be encoded explicitly. Depending on the chosen encoding, this can also increase the encoding cost of neighboring buckets. For these reasons, we want to minimize the seed for such buckets by processing them first.

4.3. Union-Find

A union-find data structure maintains disjoint subsets of an underlying set M. It supports the operations *union* and *find*. At the start, each element in M is contained in its own subset. The operation *union* joins to subsets. The operation *find* returns a representative element for a subset. By comparing the representative, one can determine if two elements currently belong to the same subset. The time complexity of a sequence of m *union* and *find* operations is in $O(m\alpha(m))$. Here, the function α denotes the inverse Ackermann function. The function value of the inverse Ackermann function grows extremely slowly and is $\alpha(m) \leq 5$ for all input sizes feasible in practice.

We use a union-find data structure to represent the state of the *cuckoo graph*. The set of cells is the underlying set M. The partition of the set M corresponds to the partition of cells by the graph component they belong to. Components can either be trees or pseudo trees. We express this by labeling each component as tree or as pseudo tree. On insertion of an edge, four different cases are possible. Inserting an edge between a tree and a (pseudo) tree results in a larger (pseudo) tree. An edge between two cells of the same tree component converts the component into a pseudo tree. An edge between two different pseudo trees or between two cells of the same pseudo tree both result in a conflict. The behaviour for different cases is displayed in Figure 4.2.

The idea to use a union-find data structure for construction of a PHF that uses cuckoo hashing is mentioned by ShockHash [24]. The idea is not used there because construction of the union-find data structure is to expensive compared to alternatives when used for small cuckoo hash tables. With our approach, the usual input size is significantly higher. Edges are inserted bucket by bucket. If insertion of an edge results in a conflict, the current bucket needs to be retried with a new seed. Edges of previous buckets stay the same. Instead of constructing the union-find data structure up from the ground each time, the state can be reused between multiple iterations. To facilitate this, it is necessary that changes from the current bucket can be reversed.

Union-find cannot be reset without further ado. We design three variants of such an extension to the data structure. We introduce the additional operations *new*, *reverse* und *commit*. The operation *new* signals the start of a new bucket. The data structure needs to

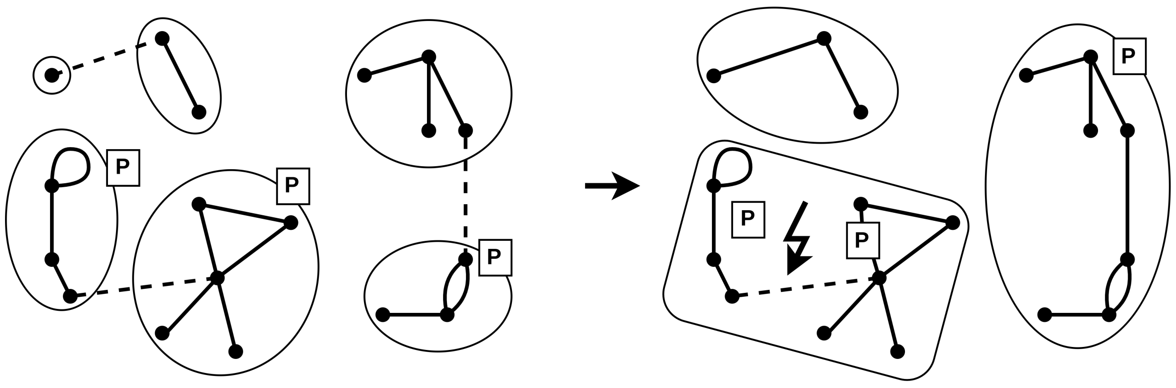


Figure 4.2.: Behaviour of the union-find data structure on insertion of edges. Components that are stored by the data structure are illustrated as borders. The label "P" indicates a pseudo tree.

be reset to this state in case of a conflict. The operation *reverse* performs a reset of the state. The operation *commit* signals the successful insertion of a full bucket. With the used *greedy* strategy, edges existing at this time are not removed in the future.

The first union-find variant creates a copy of the data structure at the start of a *new* bucket. Changes are only applied to the copy. On a *reverse*, the copy is discarded. On a *commit*, the copy replaces the backup state. This variant is simple to implement but has a high memory consumption.

Alternatively, the second variant maintains a list of changes. This list is empty at the start of each *new* bucket. Along with each *union* operation changes are recorded. On a *reverse*, changes are undone in reverse order. A *commit* clears the list of changes. The idea of this variant is to compactly store changes for buckets that are small in comparison to the number of cells.

The third variant applies changes to an additional data structure. This data structure is empty at the start of a *new* bucket. For this variant, the operations *union* and *find* operate on two levels. At first, *find* determines the representative without changes of the current bucket. Afterwards, it determines the new representative after changes of the current bucket are applied. *Union* operates only on the additional data structure. A new edge either changes the label of a component or replaces the representative of one subset with the representative of the new superset. The additional data structure thereby represents changes on representatives. The additional data structure is cleared on *reverse*. A *commit* applies the changes to the main data structure. The goal of this variant are compact storage of changes and only applying changes to the main data structure that do not need to be undone.

4.4. Filter

The goal of the filter is to reduce expensive accesses to the union-find data structure for seeds that fail for certain. We allow one-sided errors (similar to a Bloom filter). It is permitted that the filter lets seeds through that result in a conflict. For every filtered seed, it must be certain that it would have resulted in a conflict. Additionally, we require filter accesses to be quicker than accesses to the union-find data structure.

The filter only uses information on the *cuckoo state* without knowledge about changes caused by the current bucket. Therefore, the filter can only be fully precise on buckets of size 1. An edge that does not cause a conflict could convert a tree to a pseudo tree or it could connect a tree to a pseudo tree. On a bucket with multiple keys, such changes can result in a conflict for a later inserted edge. Conflicts that are only caused by the changes of multiple edges are not detected by our filter. This results in a higher filter precision on smaller buckets.

An edge results in a conflict if both end points are part of a pseudo tree (possibly the same). We can use the graph interpretation to detect a part of the conflicts in advance. We use a bit mask with a 1 for every cell in a pseudo tree and a 0 for other cells. A filter query can be handled with two bit lookups. If the filter is activated, we first query the filter for each edge. If a conflict is found, the seed is rejected.

We allow for an additional inaccuracy in our filter. The filter can contain 0 for cells that are part of a pseudo tree. We call a filter *lazy* if it makes use of this option. This change reduces the precision of the filter. Without this change, usage of the filter is only possible if we know for each cell whether its component is a tree or a pseudo tree. Insertion of an edge can introduce new pseudo tree cells if both end points are part of the same tree or if a tree is connected to a pseudo tree. Without *laziness*, both cases require an update in the bit mask for each tree node.

5. Implementation

This work implements the presented PHF in C++ [32]. The implementation uses CMake as build system. SimpleRibbon [23] is included as dependency. This is a BuRR implementation with CMake support. Additionally, the implementation includes an efficient hash table [22] and a collection of C++ data structures and algorithms [4]. The encoding algorithms are inherited from PTHash [30] and adapted for Cuckoo-PTHash.

The PHF is implemented as include library fully in header files. This increases the effort at compile time, but results in machine code optimized for the application that includes the PHF. The PHF enables extensive configuration by implementing variants for each step as described in Chapter 4. Different variants are implemented as template types with a shared interface. This static polymorphism is resolved by instancing at compile time. These implementation choices aim at avoiding overheads on important execution paths. Optimizations or outsourcing to compilation reduce overheads at runtime.

The implementation is divided into the core construction algorithm and algorithms for construction phases and encoding. The core algorithm first produces a non-encoded result. This way, a PHF can be encoded differently with one execution of construction. Queries can be performed on an encoded PHF.

5.1. Union-Find and Filtering

The implementations of the union-find variants have a few things in common. They each use *union-by-rank* and *path compression* to efficiently implement the primitive operations `union` and `find` of a union-find data structure. Additionally, the shared interface of the data structures encompasses the operations `insert`, `newBucket`, `reverse` and `commitBucket`. The operation `insert` inserts a new edge between two given cells or signals a conflict via the return value. The remaining operations work as described in Section 4.3. The data structures can be initialized to an empty state with a specified number of cells or from a starting state. This makes switching the used variant during construction possible. This capability is currently not used by the construction algorithm.

For our usage we require the additional information if a component belongs to a pseudo tree. This information is not provided by usual union-find data structures. Affiliation to a graph component is represented by a parent pointer. Representative nodes can usually be distinguished by a circular reference pointing to the node itself instead of the non-existant parent. Instead, the parent pointer of a representative can be used differently as long as representatives stay distinguishable from other nodes. If the *most significant bit (MSB)* is set, the node is a representative carrying additional information on the graph component. This halves the maximum size of the data structure. Such sizes are not reached in practice. The maximum value for the parent pointer signals that a graph component is a pseudo tree. Other special values can be used if required. An alternative version of the PHF uses these values to store the size of tree components.

The union-find variants differ in their implementation of reversion operations. The variant `CopyUnionFind` mostly works like a usual union-find data structure with the exception that it copies the whole data structure at the start of a new bucket. Because a *reverse* resets the data structure to the last backup, newly compressed paths are lost in the process. A small Optimization of the variant applies changes to the backup until a first edge is inserted successfully. `ReversibleUnionFind` requires a structure that uniquely describes changes. Each change specifies the previous representative, as well as information if rank or pseudo tree status changed. If *path compression* is performed, this information is not sufficient. For this reason, the variant compresses accessed paths only on the next `commitBucket`. The `TwoLevelUnionFind` variant implements the small additional data structure with multiple small hash tables. They store changes of representatives or changes to the rank or pseudo tree status of a representative. For the efficiency of the data structure this variant requires an efficient hash table. For this purpose, the included hash table implementation is used. Because changes are only applied to the main data structure on `commitBucket`, changes to this level never need to be reversed. Consequently, *path compression* is always possible on this level.

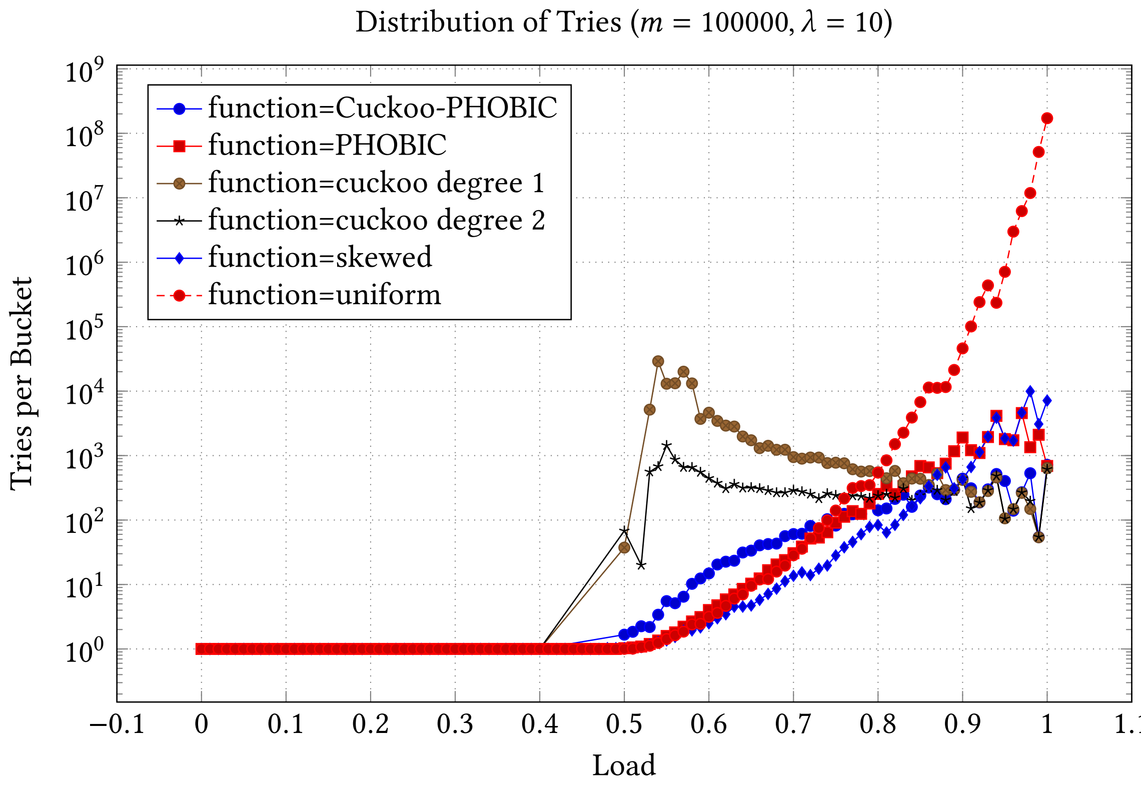
In our implementation, the filter is merged with the union-find data structure. The union-find data structure incidentally updates the filter bitmap. The interface of the filter is `isProblematic` and returns `true` exactly if both passed cells belong to a pseudo tree. In our implementation, the filter is *lazy*. Cells do not always point directly at their current representative. When a component becomes part of pseudo tree, the data structure sets bits in the bitmap along with *path compression*. The variant `TwoLevelUnionFind` also updates the filter for seeds that might fail.

6. Evaluation

In this chapter, we measure the implemented PHF and the effects of the designed algorithms. For the experiments, we use an Intel i7−11700 CPU, with 64 GB DDR4-RAM, running Ubuntu `22.0.4` as operating system. The CPU has 8 cores, with 48 KiB L1 and 512 L2 data cache each. To compile the code, we used GCC in version `11.4.0` with the compiler options `-march=native` und `-O3` activated.

In the first part, we compare the different algorithms designed for each step with each other. Moreover, we evaluate the impact of the filter. With some of the compared algorithms, large inputs are not feasible. For this reason, unless declared otherwise, the used input size is 100000 Keys and $\lambda = 10$. In the second part, we compare the PHF with related methods. For these benchmarks, the input size is 100 million keys. Λ is specified for each result. The inputs are random strings of length 10 to 50 characters. Because most hash functions initially hash inputs and proceed with these hashes, construction is mostly independent of the specific keys. All presented results are an average of three seperate executions.

6.1. Bucket Function



At first, we observe the effects of the designed variants for the *Mapping* step (see Section 4.1). The goal of more complex bucket functions is balancing the expected work per bucket [21]. How well each function achieves this goal can be evaluated by observing the seed distribution during construction. This is depicted in Section 6.1. In principal, the probability to successfully insert a key decreases with higher load. Therefore, the expected number of tries grows exponentially. For equal load, the success probability is higher for smaller buckets. This causes a reduction of tries on transition to the next smaller bucket size. With cuckoo hashing, the success probability is close to 1 for $\alpha < 0.5$. Thus, tries only increase significantly for $\alpha > 0.5$.

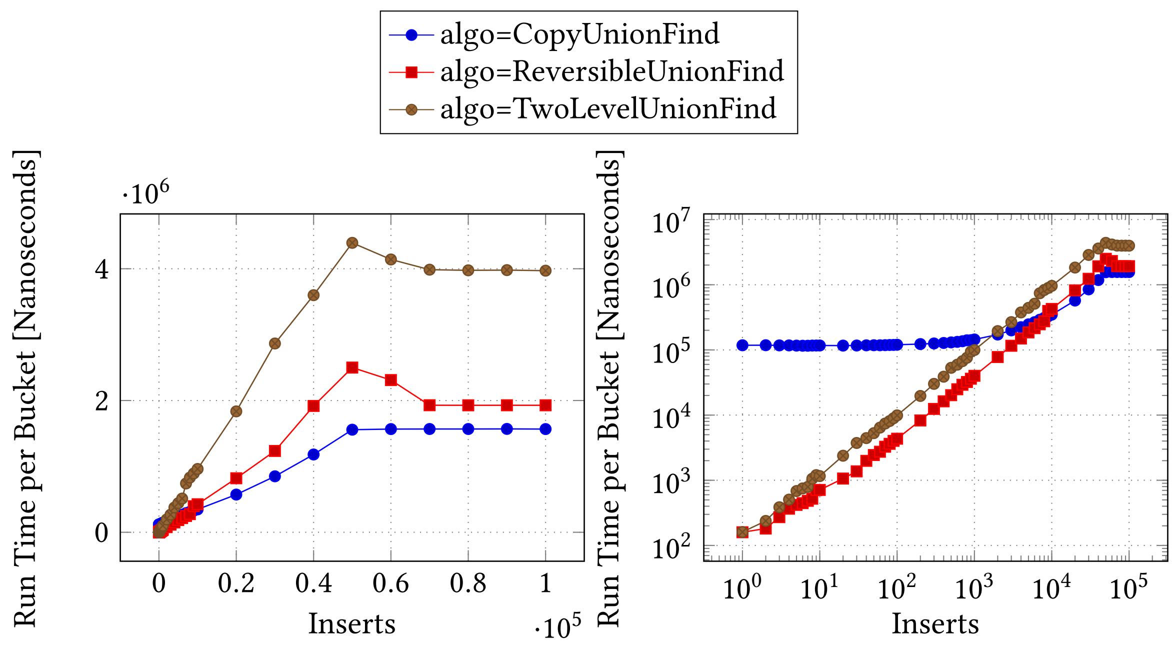
The uniform distribution of bucket sizes does not counteract the exponential growth of tries. Consequently, the number of tries actually increases exponentially. Due to the exponential growth, this method is only feasible for small inputs. The skewed distribution, as used in PTHash, uses few, large buckets for the first half of the keys. Hence, for the same λ, the method can use smaller buckets than the uniform distribution on the second half of the keys. In the measurements, this is visible as an exponential growth with a reduced growth rate. The bucket function of PHOBIC aims at keeping the expected work per bucket constant. Because the function is not designed for use with cuckoo hashing, the tries are not constant for Cuckoo-PTHash. The necessary tries are at first higher than with the skewed distribution for $\alpha > 0.5$, but are smaller on the last buckets. Switching to Cuckoo-PHOBIC is closer to a uniform distribution of tries on the second half. It uses large buckets on the first half and applies the function of PHOBIC only to the second half. This results in even more tries at the start of the second half, but less tries on the last fifth. The total number of tries at the end is lower. The number of tries still grows continuously. The tries are not constant. The approximation of the pseudo tree nodes by a polynomial of first degree results in a bucket function that starts with a high number of retries on the second half. From this point, the tries decrease nearly linear for the following buckets. At last, by use of an approximation of second degree, a near constant number of tries per bucket can be observed.

The choice of the variant for the *Mapping* step impacts the total construction time and space usage. The exponential growth of the uniform distribution results in the highest construction time and the highest space usage by far with 3.83 bits/key. The run time and the space usage of the other variants, approximations excluded, are each sorted in descending order. At first comes the skewed distribution with 2.49 bits/key, next the bucket function of PHOBIC with 2.27 bits/key and last Cuckoo-PHOBIC with 2.17 bits/key. The construction time is halved when switching from the bucket function of PHOBIC to Cuckoo-PHOBIC. The space usage of the approximation of first degree is 2.22 bits/key. The space usage of the approximation of second degree is with 2.16 bits/key slightly lower than the space usage of Cuckoo-PHOBIC. The construction time for the approximation of first degree is slightly higher than for the skewed distribution, the construction time for the approximation of second degree is 50% higher than for Cuckoo-PHOBIC. The high run time for the approximation of first degree is caused by an insufficient approximation. The function chooses too large buckets at the start of the second half, costing construction time. The higher construction time for the approximation of second degree is caused by a higher average seed compared to Cuckoo-PHOBIC.

The bucket function of PHOBIC has a similar development of tries compared to the approximations. In practice, the number of tries for Cuckoo-PHOBIC is smaller than for the latter for most of the buckets. On the last buckets, the number of tries is similar for these variants. This results in a similar, even slightly better space usage. At the same time, the construction time is significantly smaller. For this reason, unless stated otherwise, the bucket function Cuckoo-PHOBIC is used for construction.

6.2. Union-Find

In this section, we compare the designed union-find data structures that are used during the *Search* step (see Section 4.3). Union-find data structures help with the search by maintaining the state of the *cuckoo graph* between iterations and enabling checks for conflicts. For comparison, the variants were measured in experiments. These experiments simulate conditions similar to the usage during construction. At the start, the union-find data structures can be filled up to a desired load. Afterwards, we measure the processing time for one bucket. We define this time as the timespan between invocation of `newBucket` and `commitBucket` or `reverse`, respectively. A bucket contains a desired number of pseudo-random keys. Keys can lead to collisions. In this case, the insert sequence ends with a



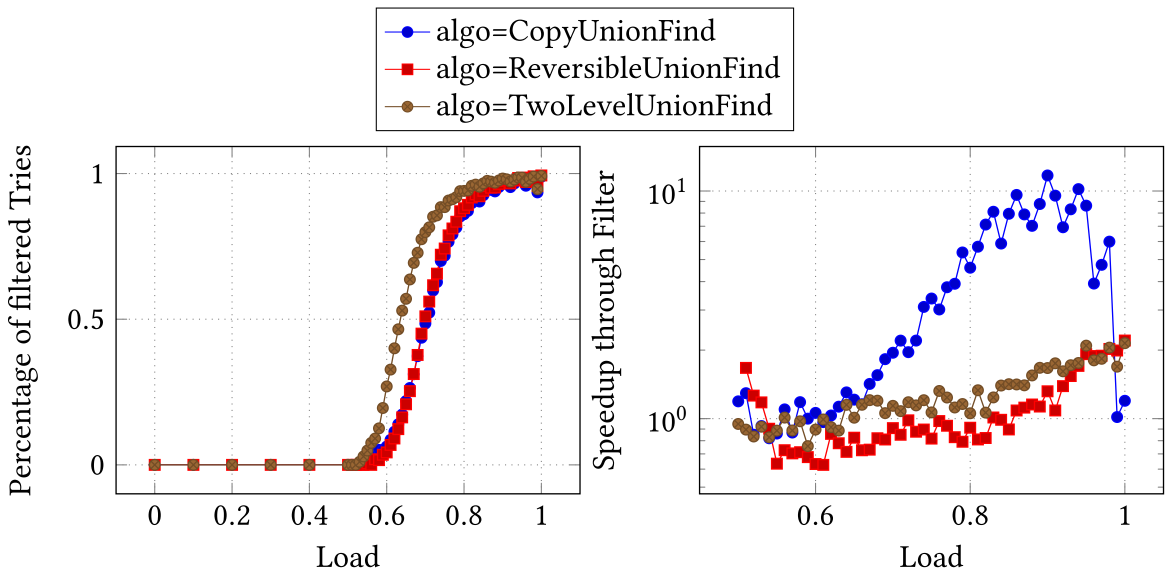
(a) Union Find Benchmark ($m = 100000, \alpha = 0$) **(b)** Union Find Benchmark ($m = 100000, \alpha = 0$)

Figure 6.1.: Time measurements of bucket insertion

`reverse` after the conflict. The displayed measurements were performed on empty union-find data structures. We measured the run time per bucket over growing bucket size. The increment of the measured bucket sizes was increased in regular intervals.

The results of the experiments are displayed with linear scale in Figure 6.1a. This representation helps to verify the setting of the experiments. For bucket sizes above half of the size of the data structure, conflicts during insertion are more likely. Due to conflicts, the run time per bucket should increase less strongly beyond this threshold. In effect, a sharp decline of the growth rate validates this expactation. For buckets larger than half the size of the data structure, the run time per bucket is nearly constant. The growth up to buckets of this size is linear for every variant. `CopyUnionFind` has the smallest growth rate, followed by `ReversibleUnionFind`. The variant `TwoLevelUnionFind` has the largest growth rate. Differences on small buckets are not discernible with this scale.

For this reason, Figure 6.1b shows the same result with a logarithmic scale for both axes. Thereby, differences on small buckets become apparent. For buckets below a certain size, the run time of `CopyUnionFind` does only decrease marginally. For the chosen parameters, the threshold are buckets of around 1000 keys. This matches with expectations as this variant needs to create a copy of the data structure. The minimal run time per bucket grows with the size of the data structure. For this reason, the variant is not suitible in practice. The majority of the buckets is small and nearly all of the retries arise for these small buckets. Without eliminating the constant costs, `CopyUnionFind` is infeasible for small buckets and for large data structures. For very large buckets this variant can still be useful if the variant is switched afterwards for smaller buckets. The other two variants show a linear growth starting from buckets of size 1. At that, `TwoLevelUnionFind` is only faster



(a) Construction (n=100000, avg. bucket 10) **(b)** Construction (n=100000, avg. bucket 10)

Figure 6.2.: Filter plots

than `ReversibleUnionFind` on this minimal bucket size. Because the growth rate is higher for `TwoLevelUnionFind`, `ReversibleUnionFind` is faster for almost every bucket size. With deactivated filter, using `ReversibleUnionFind` is most efficient.

6.3. Filter

The filter is designed to reject a majority of conflicting seeds with accesses to the used union-find data structure. Initial measurements showed that the major part of the search time is spent on these accesses. The goal of the filter is to reduce the search time, since less of these expensive accesses are necessary. In this section, we measure the actual impact of the filter.

Figure 6.2a shows the portion of the tries that are blocked by the filter. This measurement is created from data that was recorded during construction. For the first half of the keys, the filtered percentage is 0%. This has two reasons. First, in this half, most buckets succeed on the first seed. On the other hand, the filter is only effective if the affiliation of a cell to a pseudo tree is also recorded for the filter. With a *lazy* filter, this recording is often delayed. For a load above 0.5, the portion of the filtered Tries starts to grow. This portion develops differently depending on the union-find implementation. This is caused by the filter implementation that is coupled to the union-find variants. The portion of the filtered tries grows towards 1 for all variants. For decreasing bucket sizes there are less failed attempts that the filter cannot detect. Additionally, cells that lie in a pseudo tree gradually get detected. The filter reaches higher accuracies earlier for `TwoLevelUnionFind`. For `CopyUnionFind` and `ReversibleUnionFind`, the filter can only be updated after a seed is successful. The variant

`TwoLevelUnionFind` does not need to reverse changes to the first level. As a consequence, the filter can be updated for any access to this level. Only cells that only lie in pseudo tree after the change of the current bucket are not captured by this. That explains the earlier growth of the accuracy.

The filter is designed to have cheap accesses in comparison to accesses to the union-find data structure. If a seed is filtered, an access to the union-find data structure is not necessary anymore. An unsuccessful try that is not filtered has an increased run time because both the filter and the union-find data structure are accessed. The speedup that is achieved by using the filter is displayed in Figure 6.2b. Because the filter is only effective for loads above 0.5, the speedup is only displayed for this half. For `CopyUnionFind` and `ReversibleUnionFind`, the speedup is below 1 at first, because the filter causes additional costs initially. For `TwoLevelUnionFind`, the speedup reaches 1 early because this variant starts filtering tries earlier. The speedup grows for every variant. The speedup is the highest for `CopyUnionFind` with values up to 10. This arises as a result of the high constant costs per bucket that can be avoided by filtering an unsuccessful seed. On the last buckets, the speedup sinks again. This is caused by an optimization built for this variant that already avoids the high copying costs for conflicts that occur on the first edge and for buckets of size 1. The speedup of `TwoLevelUnionFind` is slightly higher than the speedup of `ReversibleUnionFind` for many buckets. The speedup on the last buckets is similar for both variants. The highest reached speedup is around 2.2. The progression of the speedup for both variants follows from the progression of the filtered tries. The portion of filtered tries is higher at first for `TwoLevelUnionFind`, but grows towards the same limit.

The speedup for every variant is close to 1 or higher for most buckets of the second half. Hence, the filter is worth it on this half for every variant, especially so for `CopyUnionFind` and `TwoLevelUnionFind`. For the half that is excluded here, no tries get filtered. The filter can be deactivated for this half to avoid additional costs. All variants have lower construction times through use of the filter. The speedup is 4.84 for `CopyUnionFind`, 1.15 for `ReversibleUnionFind` and 1.31 for `TwoLevelUnionFind`. Moreover, it must be noted that `TwoLevelUnionFind` even achieves a lower construction time than `ReversibleUnionFind` if both use the filter. This variant is therefore used for construction by default unless stated otherwise.

6.4. Performance of each step

Because this approach supports a range of algorithms for each step, the steps can scale differently and the optimal algorithm can vary for different input sizes. To detect performance issues and possibilities for future work, we display the performance of each step in the current default configuration in Figure 6.3.

This plot makes it visible if the default algorithm for a step scales well. Algorithms that do not scale well could be replaced with a better alternative. Steps that perform well on inputs below a threshold but drop off above the threshold, might require an alternative for larger

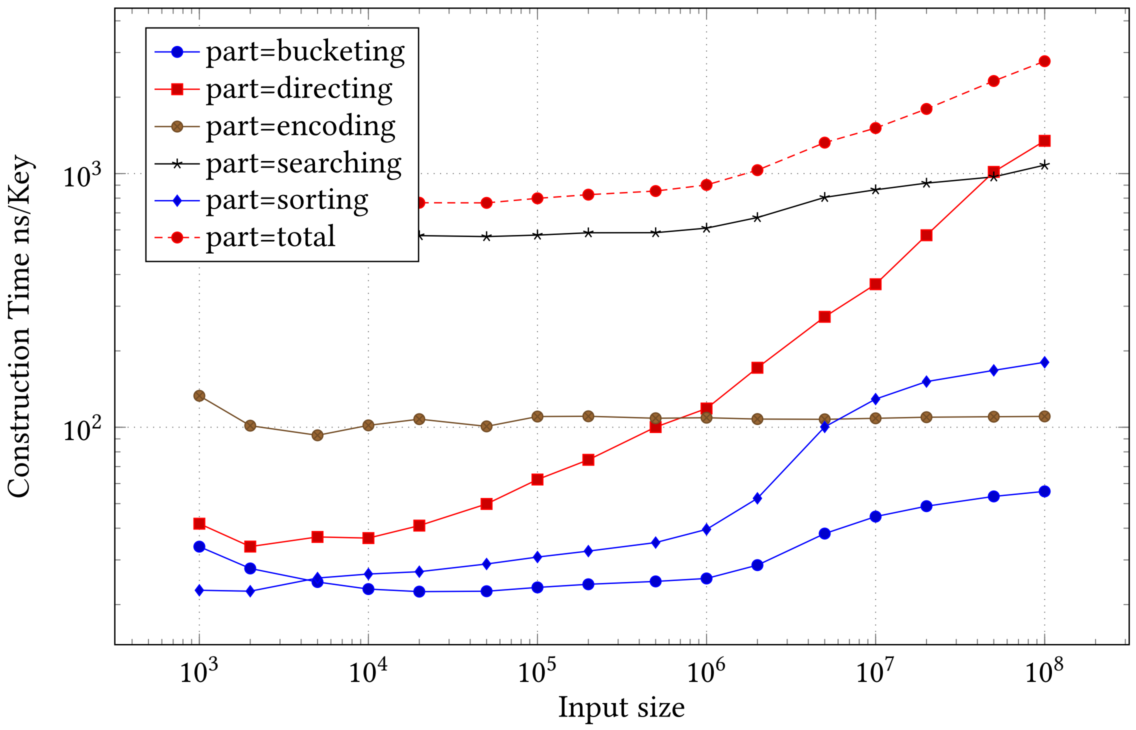


Figure 6.3.: Performance of each step over growing input sizes

inputs. The step with the best scaling is the *encoding* step. The time needed for this step is linear, even for large inputs. The *searching* step takes the majority of the construction time. The bucketing and sorting step each make up a small part of the construction time. The runtime per key for each of these steps is mostly stable for many input sizes. For input sizes above a million keys, the construction time per key increases for each of these steps. This makes the size a promising threshold for partitioning. Because the growth for input sizes above a million keys is high, an alternative sorting algorithm for such inputs would help. The most important step for improvements is the *directing* step. This step shows an exponential growth over every input size. It surpasses the time needed for searching for inputs above 50 million keys. This limits the performance of the approach for large inputs. A better scaling algorithm for this step would greatly increase the performance of the construction in total.

6.5. Comparison to related methods

We compare our new approach against related methods from the field of minimal perfect hashing. Especially, we compare with PTHash [30]. Our approach is a collection of modifications and extensions to this method. Additionally, we compare with SicHash [25], a (minimal) PHF using irregular cuckoo hashing. Every benchmark is performed on an input size of 100 million keys. Because our approach runs purely sequential, existing parallel

Table 6.1.: Performance of different methods on 100 million keys

Method	Space (bit/key)	Query (ns/query)	Construction (ns/key)
Cuckoo-PTHash $\lambda = 10, \alpha = 1.0$, PC	2.12	72	3173
Cuckoo-PTHash $\lambda = 10, \alpha = 1.0$, EF	2.06	85	3173
Cuckoo-PTHash $\lambda = 6, \alpha = 1.0$, PC	2.43	74	2384
Cuckoo-PTHash $\lambda = 8, \alpha = 1.0$, PC	2.23	73	2618
Cuckoo-PTHash $\lambda = 12, \alpha = 1.0$, PC	2.04	72	3258
SicHash, $\alpha = 0.9, p_1 = 21, p_2 = 78$	2.41	72	129
SicHash, $\alpha = 0.97, p_1 = 45, p_2 = 31$	2.08	64	179
PTHash, $\lambda = 4.0, \alpha = 0.99$, C-C	3.19	35	298
PTHash, $\lambda = 5.0, \alpha = 0.99$, EF	2.11	54	532
PTHash, $\lambda = 6.4, \alpha = 0.99$, EF	1.99	52	1550

variants of other methods are excluded. For comparability, every method is compared in its sequential implementation.

For the same space usage, our approach is currently multiple times slower than PTHash. On one side, this shows that our method can not handle the chosen input size equally well. Section 6.4 reveals potentials for improvements on inputs of above a million keys. On the other hand, our approach is the first attempt to use cuckoo hashing for the construction of large hash tables. It might be the case, that a different approach is necessary for better results on large input sizes. Although the construction time is higher for our approach, we can reach a similar space usage before the construction time becomes infeasibly large. This shows a significant decrease in bruteforce complexity and search complexity. Our method already uses about 1 bit/key for the retrieval data structure. A similar space usage in total is only possible with a reduction in space usage for the bucket seeds. This is caused by the usage of cuckoo hashing. It greatly reduces the number of tries per key. The construction time is approximately 20 times higher than for SicHash. SicHash uses cuckoo hashing for the construction of small irregular cuckoo hash tables and is more efficient with this approach. The query time of Cuckoo-PTHash is higher compared to the queries of PTHash, as well. This is expected because both methods use the same algorithms for the encoding of the seed. Additionally, our approach needs to query the retrieval data structure. The query times are comparable to SicHash that also needs to query a retrieval data structure but can query seeds a bit more efficiently. In total, our approach is less efficient in comparison with the compared methods. At the same time, the difference is small enough to motivate further improvements to this method or exploring similar approaches with cuckoo hashing in future work.

7. Conclusion

Our approach enables construction of a (minimal) PHF with different tradeoffs. The approach uses the three construction steps of PTHash, but uses cuckoo hashing during search to map keys to cells. To utilize cuckoo hashing we designed a new construction step (*Directing*). With this addition, the flexibility of cuckoo hashing can be maintained during the whole search step. For most of the steps, we designed alternative algorithms that can be chosen for construction.

For the *Mapping* step we designed new variants with the goal to reduce the entropy of the seeds. This is especially successful for the approximation of the number of pseudo tree cells by a polinomial of second degree. The seeds are nearly constant on the second half of keys. The solution procedure can also be transferred to other methods. The new variants improve the space-time-tradeoff.

For the search step, we designed three variants of revertible union-find data structures. Additionally, we designed a filter that reduces the necessary accesses to these data structures. The variants are evaluated and discussed in detail and advantages and disadvantages are compared. Two of the three variants are suitable for large input sizes.

Different configurations of our approach are compared with related methods in an experimental evaluation. The performance of our approach is worse compared to these methods. However, the difference is small enough to motivate further study. Our approach greatly reduces the necessary bruteforce compared to PTHash.

Our approach provides ideas for future work. In the scope of this work, utilizing partitions or developing a parallel implementation of designed algorithms was not possible. Both partitions and parallelization result in performance boosts on related methods. Similar speedups might be possible for our approach. Some of the currently used algorithms scale worse for inputs above a certain size. For this reason, limiting the size of a partition can be beneficial. A different aspect of future work is a new focus for the cuckoo hashing approach. This work focuses on efficient construction of a MPHF using a greedy strategy. If cuckoo hashing is used, different successful seeds are not necessarily equivalent to each other. A clever choice of seeds can result in performance improvements. A more detailed discussion of this idea is given in Appendix A.1. Such ideas replace the greedy strategy with an alternative. Backtracking is an additional option to achieve more optimal seeds. Because such changes aim at achieving a more compact encoding, possibly at the cost of construction time, fully focusing on minimal space usage is an option.

Bibliography

[1] Djamal Belazzougui, Fabiano C. Botelho, and Martin Dietzfelbinger. “Hash, displace, and compress”. In: *ESA*. Vol. 5757. Springer. 2009, pp. 682–693. DOI: `10.1007/978-3-642-04128-0_61`.

[2] Djamal Belazzougui and Gonzalo Navarro. “Alphabet-independent compressed text indexing”. In: *ACM Trans. Algorithms* 10.4 (2014), pp. 1–19.

[3] Djamal Belazzougui et al. “Fast prefix search in little space, with applications”. In: *ESA*. Vol. 6346. Springer. 2010, pp. 427–438.

[4] Timo Bingmann. *TLX: Collection of Sophisticated C++ Data Structures, Algorithms, and Miscellaneous Helpers.* 2018. URL: `https://github.com/tlx/tlx` (visited on 06/03/2024).

[5] Chin-Chen Chang and Chih-Yang Lin. “Perfect Hashing Schemes for Mining Association Rules”. In: *The Computer Journal* 48.2 (Jan. 2005), pp. 168–179. DOI: `10.1093/comjnl/bxh074`.

[6] Luc Devroye and Pat Morin. “Cuckoo hashing: Further analysis”. In: *Information Processing Letters* 86.4 (2003), pp. 215–219.

[7] Martin Dietzfelbinger and Christoph Weidling. “Balanced allocation and dictionaries with tightly packed constant size bins”. In: *Theoretical Computer Science* 380.1-2 (2007), pp. 47–68.

[8] Martin Dietzfelbinger et al. “Tight thresholds for cuckoo hashing via XORSAT”. In: *Automata, Languages and Programming: 37th International Colloquium, ICALP 2010, Bordeaux, France, July 6-10, 2010, Proceedings, Part I 37.* Springer. 2010, pp. 213–225.

[9] Peter C. Dillinger et al. “Fast Succinct Retrieval and Approximate Membership using Ribbon”. In: *20th International Symposium on Experimental Algorithms (SEA 2022).* 2022, 4:1–4:20. DOI: `10.4230/LIPIcs.SEA.2022.4`. URL: `https://github.com/lorenzhs/BuRR` (visited on 12/18/2023).

[10] Michael Drmota and Reinhard Kutzelnigg. “A precise analysis of cuckoo hashing”. In: *ACM Trans. Algorithms* 8.2 (2012), pp. 1–36.

[11] Peter Elias. “Efficient storage and retrieval by content and address of static files”. In: *Journal of the ACM (JACM)* 21.2 (1974), pp. 246–260.

[12] Emmanuel Esposito, Thomas Mueller Graf, and Sebastiano Vigna. “RecSplit: Minimal perfect hashing via recursive splitting”. In: *ALENEX*. SIAM. 2020, pp. 175–185.

[13] Robert Mario Fano. *On the number of bits required to implement an associative memory.* Massachusetts Institute of Technology, Project MAC, 1971.

[14] Dimitris Fotakis et al. "Space efficient hash tables with worst case constant access time". In: *STACS 2003: 20th Annual Symposium on Theoretical Aspects of Computer Science Berlin, Germany, February 27–March 1, 2003 Proceedings 20*. Springer. 2003, pp. 271–282.

[15] Nikolaos Fountoulakis, Megha Khosla, and Konstantinos Panagiotou. "The multiple-orientability thresholds for random hypergraphs". In: *Combinatorics, Probability and Computing* 25.6 (2016), pp. 870–908.

[16] Nikolaos Fountoulakis and Konstantinos Panagiotou. "Sharp load thresholds for cuckoo hashing". In: *Random Structures & Algorithms* 41.3 (2012), pp. 306–333.

[17] Edward A Fox, Qi Fan Chen, and Lenwood S Heath. "A faster algorithm for constructing minimal perfect hash functions". In: *SIGIR*. 1992, pp. 266–273.

[18] Michael L. Fredman, János Komlós, and Endre Szemerédi. "Storing a Sparse Table with 0(1) Worst Case Access Time". In: *J. ACM* 31.3 (June 1984), pp. 538–544. ISSN: 0004-5411. DOI: 10.1145/828.1884. URL: https://doi.org/10.1145/828.1884.

[19] Kimmo Fredriksson and Fedor Nikitin. "Simple compression code supporting random access and fast string matching". In: *Experimental Algorithms: 6th International Workshop, WEA 2007, Rome, Italy, June 6-8, 2007. Proceedings 6*. Springer. 2007, pp. 203–216.

[20] Solomon Golomb. "Run-length encodings (corresp.)" In: *IEEE transactions on information theory* 12.3 (1966), pp. 399–401.

[21] Stefan Hermann et al. *PHOBIC: Perfect Hashing with Optimized Bucket Sizes and Interleaved Coding*. 2024. arXiv: 2404.18497 [cs.DS].

[22] ktprime. *Fast and memory efficient c++ flat hash map/set*. 2019. URL: https://github.com/ktprime/emhash (visited on 06/03/2024).

[23] Hans-Peter Lehmann. *SimpleRibbon*. 2022. URL: https://github.com/ByteHamster/SimpleRibbon/ (visited on 06/03/2024).

[24] Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer. "ShockHash: Towards Optimal-Space Minimal Perfect Hashing Beyond Brute-Force". In: *arXiv preprint arXiv:2308.09561* (2023).

[25] Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer. "SicHash - Small Irregular Cuckoo Tables for Perfect Hashing". In: *ALENEX*. SIAM. 2023, pp. 176–189.

[26] Marc Lelarge. "A new approach to the orientation of random hypergraphs". In: *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*. SIAM. 2012, pp. 251–264.

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

[28] Rasmus Pagh and Flemming Friche Rodler. "Cuckoo hashing". In: *Journal of Algorithms* 51.2 (2004), pp. 122–144. DOI: https://doi.org/10.1016/j.jalgor.2003.12.002.

[29] Giulio Ermanno Pibiri. "Sparse and skew hashing of k-mers". In: *Bioinformatics* 38.Supplement_1 (2022), pp. i185–i194.

[30] Giulio Ermanno Pibiri and Roberto Trani. "PTHash: Revisiting FCH Minimal Perfect Hashing". In: *SIGIR*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 1339–1348. DOI: 10.1145/3404835.3462849. URL: https://github.com/jermp/pthash (visited on 06/03/2023).

[31] Robert F Rice. *Some practical universal noiseless coding techniques.* Tech. rep. 1979.

[32] Benedikt Waibel. *Cuckoo-PTHash.* 2024. URL: https://gitlab.kit.edu/ucilx/cuckoo-pthash (visited on 07/06/2024).

[33] Walter Whiteley. "The Union of Matroids and the Rigidity of Frameworks". In: *SIAM Journal on Discrete Mathematics* 1.2 (1988), pp. 237–255. DOI: 10.1137/0401025.

A. Appendix

A.1. Optimality of a seed

The previous approaches always take the first seed with successful insertion (greedy strategy). When instead multiple successful seeds are compared, a new measure is necessary. We need to be able to judge how optimal a seed is in respect of expected future work. Otherwise, alternative strategies and backtracking decisions might only decrease performance.

For PTHash, successful pilots do not impact the success probability of future buckets differently. The success probability depends on the number of free cells. When cuckoo hashing is used, that is not true anymore. The success probability for future buckets depends on the current graph structure. Multiple successful seeds can have highly different impact on future work. This becomes clear when we look at the graph representation. When a seed is tested for conflicts, we test if each component of the graph still is a pseudo tree. Components in an instance without conflicts can be pseudo trees or trees. Also, the size of components can vary from a single cell to all cells in one component. More pseudo tree nodes directly reduce the success probability because we cannot insert edges between pseudo tree components. Additionally, states that have a higher likeliness of adding many pseudo tree nodes are worse for the success probability, as well. That means, tree components of smaller size are better in this respect. A large tree could become a pseudo-tree with one additional edge.

In conclusion, an optimality measure for Cuckoo-PTHash can assign a better score to states with less pseudo tree nodes. The indirect effect of large tree components can be considered by assigning a better score to uniform component sizes. We can not only look at component sizes and ignore pseudo tree nodes. If we would only insert loops, each component would have size 1, but the success probability is the worst. We can not look at the average component size either, because it is equal for every state with the same number of pseudo-tree nodes.

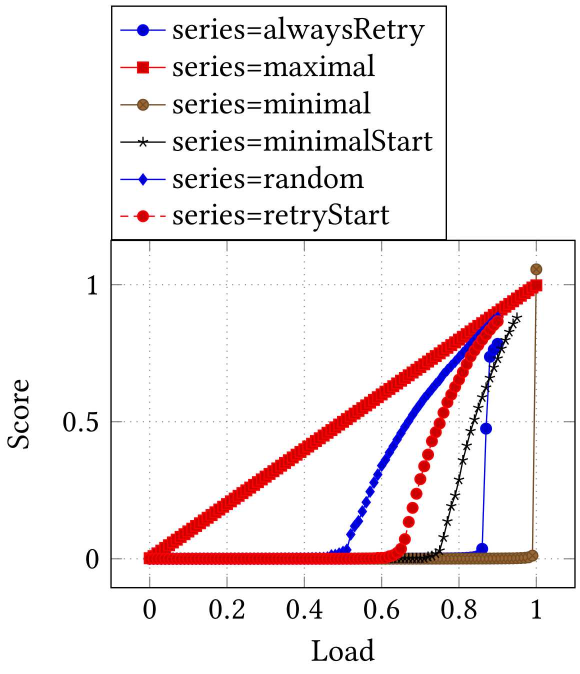
Taking this into account, we can score states by first comparing the number of pseudo tree nodes. Second, the sum of squares of tree component sizes can compared, a lower sum being better. In this case, a lower number of pseudo tree nodes is always counted as better score. Alternatively, one score could be calculated from both values. Having a few more more pseudo tree nodes might balance out with very balanced tree components. This alternative allows to represent this in the score calculation. Future work could explore different score calculations and weights for both measures.

To calculate a score, the number of pseudo tree nodes and the size of each tree component need to be known. For an efficient calculation, this information cannot be calculated anew for every iteration. The first can just be stored and updated for each succesful seed. The second requires keeping track of component sizes after each insert. For the union-find approach, the rank is not sufficient as approximation. Adding components of smaller rank increases the component size but never the rank. Our implementation allows to use the parent pointer of representatives for additional information. This value can be used to store the size of a tree component. This is done as an example on a dedicated branch of the implementation.

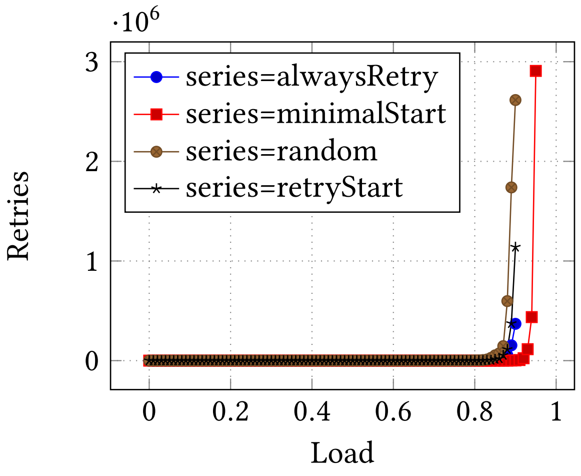
The chosen strategy impacts the probability of choosing a cuckoo graph, the success probability, and the expected future work. Because the strategy impacts the expected future work, the optimal bucket function for each strategy can differ. Exploring possible strategies and modifying the PHF for a new strategy goes beyond the scope of this work. Examples of promising strategies are described in the following. Instead of always choosing the first succesful seed, the search could continue until a certain number of successful seeds are available. Also, the search could continue until a number of retries is reached. Both strategies would continue with the succesful seed with the best score. Alternatively, the search could reject seeds below a certain score threshold. Additionally, trying additional seeds could be done with awareness of the encoding. For example, partitioned compact encoding has a predictable space usage. If a previous bucket in the same partition has a higher seed, the search could continue up to this seed size for the current bucket.

To judge if alternative strategies can improve the performance, we measure a few strategies in an experiment. This experiment simulates construction on a choice of strategies. The bucket size is fixed at 10 and the number of cells is $m = 10000$. We measure the average score for each strategy and the number of retries. The score is calculated as a sum of the number of pseudo tree nodes times the square of the number of cells, and the sum of squares of tree component sizes. We compare with the greedy strategy. Another strategy, called `retryStart`, compares 16 tries for loads $\alpha < 0.5$. The strategy, called `alwaysRetry`, retries until 16 successful seeds are found, regardless of the load. Additionally, we measure a scenario, called `minimalStart`, that artificially generates the optimal cuckoo graph up to a load of $\alpha = 0.5$. For higher load, the greedy strategy is used. Because the bucket size is fixed at 10, the experiment is stopped when retries grow infeasibly large.

The results of this experiment are shown in Figure A.1. The greedy strategy is called `random` in the results because it chooses a random cuckoo graph without conflict. First, we discuss the average score for the measured strategies. This value is displayed in Figure A.1a. The results include a calculated minimal and maximal score for each load. The score is low for every strategy in the first half. This is due to the score formula. Differences of tree component sizes have small impact compared to the impact of pseudo tree nodes. For higher loads, the scores of the strategies increase. For $\alpha \longrightarrow 1$, the scores tend to the maximal score. The score first starts growing significantly for the greedy strategy, next for the `retryStart` strategy and last for the `minimalStart` strategy. The `alwaysRetry` has the lowest score for a long time but passes the `minimalStart` strategy with a rapid growth around $\alpha = 0.5$.



(a) Optimality Score over load



(b) Retries over load

Figure A.1.: Measurement results for different strategies.

The number of retries are shown in Figure A.1b. Every strategy has an exponential growth of retries. This is caused by the fixed bucket size. The start of the exponential growth is different for each strategy. The retries start growing exponentially in the following order: the greedy strategy, the `retryStart` strategy, the `alwaysRetry` strategy, and the `minimalStart` strategy.

The measurements show that considering alternative seeds on the first half can benefit the performance on the second half. An artificial choice of an optimal cuckoo graph for the first half delays the growth of the retries the most, even though the greedy strategy is used for the second half. Admitting a number of retries on the first half, can already delay the growth of score and retries noticable on the second half. The `alwaysRetry` has the lowest score up to a load of around $\alpha = 0.85$, but grows rapidly afterwards. The growth of retries starts earlier than for the artificial strategy. This shows that below a load or score threshold, the exponential growth is not triggered. Also, the chosen score function combined with the `alwaysRetry` strategy seem to result in a few large tree components. When these grow to large, the strategy can not avoid converting them to pseudo trees, resulting in the rapid growth.

In conclusion, the experiment shows that alternatives to a greedy strategy might benefit the performance of PHF methods using cuckoo hashing. The chosen strategy could greatly reduce the size of the seeds, at least up to a certain load factor. A bit of extra work on the first half could reduce the search costs on the second half by a lot. This might result in a decreased total search time and a more space efficient PHF.