Karlsruhe Institute of Technology

Compacting Minimal Perfect Hashing using Symbiotic Random Search

Bachelor's Thesis of

Jonatan Ziegler

At the Department of Informatics
Institute of Theoretical Informatics (ITI)

Reviewer:	Prof. Dr. rer. nat. Peter Sanders
Second Reviewer:	T.T.-Prof. Dr. Thomas Bläsius
Advisors:	Dr. Stefan Walzer
	M.Sc. Hans-Peter Lehmann

June 10, 2024 – October 10, 2024

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself. I have not used any other than the aids that I have mentioned. I have marked all parts of the thesis that I have included from referenced literature, either in their original wording or paraphrasing their contents. I have followed the by-laws to implement scientific integrity at KIT.

Karlsruhe, October 10, 2024

. . . . . .
(Jonatan Ziegler)

Abstract

Minimal perfect hash functions (MPHFs) are data structures that map a set of n keys to $\{1, \ldots, n\}$ bijectively. To store them, at least $n \cdot \log_2 e \approx 1.4427n$ bits are necessary. Previous MPHFs reach as low as 1.489 bits per key by finding seeded hash functions that split the keys into two sets of equal size repeatedly and handling a base case. However, storing all those seeds individually—even with an entropy optimal encoding—wastes up to $\log_2 e$ bits per split. We introduce Symbiotic Random Search (SRS) which solves this problem by utilizing a special search pattern. By bounding the number of attempts to find a splitting hash function in one go and backtracking if none was found, SRS creates a combined encoding of all splitting seeds. SRS can waste arbitrarily little $\omega > 0$ bits per split on average. The expected work required for SRS to find such an encoding is linear in $1/\omega$. SRS can be applied whenever a data structure shall be found by trying candidates, this search can be performed step by step and using little space is a goal. SRS-RecSplit—SRS applied to RecSplit—reaches a space usage of 1.442 91 bits per key in practice with reasonable construction time—just 0.000 21 bits overhead. For similar construction time, it reaches as low as $1/192$ the space overhead of the previous record holder. For a similar overhead, SRS-RecSplit's construction is up to 101 times faster. Furthermore, SRS-RecSplit's theoretical construction time being linear in the inverse overhead makes it reach even smaller overheads more easily than its competitors, which have an exponential relationship between these metrics.

Zusammenfassung

Minimale perfekte Hashfunktionen (MPHFs) sind Datenstrukturen, die eine Menge aus n Schlüsseln bijektiv auf $\{1, \ldots, n\}$ abbilden. Um diese abzuspeichern werden mindestens $n \cdot \log_2 e \approx 1.4427n$ Bits benötigt. Bisherige MPHFs erreichen bis zu 1.489 Bits pro Schlüssel, indem sie geseedete Hashfunktionen finden, welche die Schlüssel wiederholt in zwei gleich große Teile aufteilen und einen Basisfall gesondert handhaben. Allerdings werden durch das individuelle Abspeichern der Seeds bis zu $\log_2 e$ Bits pro Unterteilung verschwendet, selbst bei Verwendung einer entropieoptimalen Codierung. Wir präsentieren Symbiotic Random Search (SRS) welches dieses Problem durch ein spezielles Suchmuster löst. Durch das Beschränken der Anzahl an Versuchen eine teilende Hashfunktion in einem Anlauf zu finden und durch Backtracking, falls keine Hashfunktion gefunden wurde, erzeugt SRS eine gemeinsame Codierung aller aufteilender Seeds. SRS kann im Durchschnitt beliebig wenige $\omega > 0$ Bits pro Teilung verschwenden. Die erwartete Arbeit dafür ist linear in $1/\omega$. SRS kann immer dann angewandt werden, wenn eine Datenstruktur durch zufälliges Ausprobieren gefunden werden soll, diese Suche Schritt für Schritt vorgenommen werden kann und ein geringer Platzverbrauch ein Ziel ist. SRS-RecSplit – SRS angewandt auf RecSplit – erreicht in Praxis einen Platzverbrauch von 1.442 91 Bits pro Schlüssel bei zumutbarer Konstruktionszeit – nur 0.000 21 Bits Overhead. Für eine ähnliche Konstruktionszeit erreicht es bis zu $1/192$ des Overheads des bisherigen Rekordhalters, für einen ähnlichen Overhead ist SRS-RecSplits Konstruktion bis zu 101-mal schneller. Des Weiteren erlaubt SRS-RecSplits theoretische Konstruktionszeit – linear im Kehrwert des Overheads – kleinere Overheads einfacher zu erreichen als es bei seinen Konkurrenten mit einem exponentiellen Zusammenhang der Fall ist.

Acknowledgments

I want to thank my supervisors Stefan and Hans-Peter for providing this interesting research topic and the idea behind Symbiotic Random Search and some of the proofs. I am grateful for their advice and feedback from reading through early versions of this thesis many times. I appreciate their detailed critique and high standards that from time to time were cumbersome but in the end, led to a better version of this thesis.

Contents

1. Introduction

Minimal perfect hash functions (MPHFs) are data structures, which map a set of n keys (arbitrary objects) bijectively to $\{1, \ldots, n\}$. Their applications range from networking [LPB06] to database systems [FHCD92] and bioinformatics [CLM16]. To store an MPHF, at least $n \cdot \log_2 e \approx 1.4427n$ bits are necessary (see Subsection 2.5.1 for more details). Apart from practical relevance, improving MPHFs is an interesting algorithmic exercise, trying to get closer and closer to a theoretical space limit. In practice, the previous most compact approach—bipartite ShockHash-RS [LSW24]—reaches 1.489 bits per key for an acceptable construction time (more details on this approach in Subsection 3.4.2).

This approach is based on RecSplit [EGV19], which uses seeded hash functions with codomain $\{0, 1\}$ to recursively split the keys into equally sized parts. RecSplit splits until only small groups of keys are left, which form the leafs of a splitting tree. In these leafs, RecSplit searches an MPHF directly using a seeded hash function with a larger codomain. In the end, all the seeds are near optimally encoded and stored (more details in Section 3.3). We will show that splitting an MPHF in parts and storing a seed for each part separately wastes up to $\log_2 e$ bits per seed, despite encoding each of them optimally (Subsection 4.2.3).

To avoid these losses, we introduce Symbiotic Random Search (SRS, Section 4.3). SRS searches seeds cleverly, which allows for a more compact representation. In general, SRS provides a scheme for searching successes in a sequence of Bernoulli processes—called tasks—each consisting of a sequence of Bernoulli trials that can be successful or not. SRS represents a successful trial in each task using only an arbitrarily small overhead compared to when merging all tasks into one (which is successful when all singular tasks would be) and representing the geometrically distributed first successful trial. Not merging tasks has the advantage of expecting to find successful trials faster. This is why RecSplit breaks the problem of finding an MPHF into smaller parts. To achieve its space advantage over the individual encoding of successful trials, SRS limits the number of successive trials performed for one task (or hash functions tested for one splitting when applied to RecSplit). If a successful trial was found, SRS advances to the next task, where the specific trials performed will depend on the successful trial found for the previous task. If no successful trial was found in the limited number of tries, SRS backtracks to the previous task to find another successful trial there before continuing with the current task. In essence, SRS can be applied whenever a randomized data structure shall be found by trying out candidates, and whose search can be split into parts, but optimal space usage shall be retained.

Applying SRS to RecSplit yields SRS-RecSplit (Section 4.4). Apart from using SRS to search splitting seeds, SRS-RecSplit also uses binary splittings all the way down, omitting leafs of larger sizes. SRS-RecSplit reaches space usage as low as 1.442 91 bits per key, which is just 0.000 21 bits above the lower bound (Chapter 7). For similar construction time, we reach as low as 1/192 of the space overhead of the previous record. For a similar overhead, SRS-

RecSplit's construction is up to 101 times faster. What is even more remarkable is that SRS-RecSplit's theoretical runtime is linear in $1/\omega$ for any overhead of $\omega > 0$ bits above the lower bound. This also makes it for SRS-RecSplit easier to reach a lower space overhead in practice compared to previous approaches, which have an exponential relationship between space overhead and construction time [EGV19 | LSW24]. In detail, SRS-RecSplit achieves construction in expected time $\mathcal{O}\left(\frac{n^{3/2}\log n}{\omega}\right)$ and $\mathcal{O}(\log n)$ query time for input size n and a space overhead of ω bits per key (Section 5.3). These are suboptimal as SRS-RecSplit—as presented here—omits an initial partitioning step the way original RecSplit does. However, this step can still be added on top to reach expected linear construction and constant query time.

1.1. Outline

This thesis is structured the following: First, we arrange a common baseline and understanding of notation in Chapter 2 *Theoretical Foundations*. We recite some useful basic theorems and warm up with the geometric probability distribution and its entropy, hash functions in general, and minimal perfect hash functions in particular. In Chapter 3 *Related Work*, we give an overview of some other MPHF algorithms and describe RecSplit as a basis for SRS-RecSplit in more detail. In Chapter 4 *Algorithm*, we first introduce the abstract problem SRS solves, present two obvious but suboptimal approaches, and then propose SRS as a solution (Section 4.3). Then, we use SRS to construct SRS-RecSplit (Section 4.4). Afterward, in Chapter 5 *Analysis*, we provide proof for the properties of SRS and SRS-RecSplit stated before. Chapter 6 *Implementation* provides and discusses implementation details. Then, we provide experimental results for our implementation of first SRS on a generic problem and then SRS-RecSplit in Chapter 7 *Experiments*. We will also compare SRS-RecSplit to RecSplit and bipartite ShockHash-RS. Finally, in Chapter 8 *Conclusion*, we sum up our results and look out to what remains to be explored.

2. Theoretical Foundations

This chapter provides an overview of some fundamentals useful for following along with this thesis. First, we introduce some notations used throughout this thesis and bring to mind some useful basic theorems. Then, we take a look at the geometric probability distribution that we will meet a couple of times throughout this thesis. We also introduce entropy as it will be an important tool for analysis. Finally, we introduce hashing and in particular minimal perfect hash functions (MPHFs).

2.1. Notation

We use the following notation throughout this thesis:

- $\mathbb{N}^+$ are the natural numbers *excluding* zero. $\mathbb{N}_0 := \mathbb{N}^+ \cup \{0\}$.
- $[n] := (0, n] \cap \mathbb{Z} = \{1, \dots, n\}$ for $n \in \mathbb{N}_0$
  $[n]_0 := [0, n) \cap \mathbb{Z} = \{0, \dots, n-1\}$ for $n \in \mathbb{N}_0$
  Note: Brackets $[\cdot]$ are also used for array indexing and the Iverson bracket.
- Iverson Bracket $[P] = \begin{cases} 1 & P \\ 0 & \neg P \end{cases}$ for some statement P [Knu92].
- Big O notation for parameters that have an upper bound like $p \in (0, 1)$ will be bound above instead of below, e.g. for $p \in (0, 1)$, $n \in \mathbb{N}_0$, it is $p < p_0$, not $p > p_0$ in $g \in \mathcal{O}(f) \iff \exists \alpha > 0, n_0, p_0 \; \forall n > n_0, p < p_0 : g(n, p) \leq \alpha f(n, p)$.
  This way, the same notation can also be used for behavior close to zero.
  Furthermore, $\mathcal{O}(f)$ is considered a set and thus set notation is used.
- $A \oplus B := \{a \oplus b \mid a \in A, b \in B\}$, $a \oplus B := \{a\} \oplus B$ for commonly used $\oplus \in \{+, -, \cdot, \dots\}$.
- $X \sim$ Distr means random variable X has probability distribution Distr.
- $\mathbb{P}(X)$ denotes the probability of event X happening, $\mathbb{E}[Y]$, $H[Y]$ denote the expected value and entropy of a random variable Y, respectively. *i.i.d* means independently identically distributed. $\bar{X}$ is the complement of event X.
- $\mathcal{R}_{\text{Alg}}$ denotes the runtime or work of algorithm Alg, $\mathcal{S}_{\text{Alg}}$ its space usage.
- $f|_S$ means f restricted to domain S, for $f : X \to Y, S \subseteq X$
- $\binom{U}{n} := \{S \subseteq U \mid |S| = n\}$ for a set U, $n \in \mathbb{N}_0$
  Reasonable, as for finite set U we have $|\binom{U}{n}| = \binom{|U|}{n}$.
- $f(a, \cdot)$ is understood as $x \mapsto f(a, x)$.

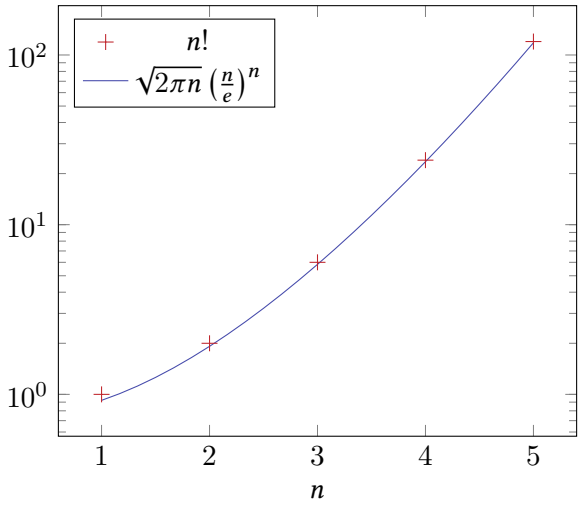


Figure 2.1.: Stirling's approximation for the factorial. Stirling's approximation is close, even for small n.

2.2. Basic Theorems

This section provides some well-known bounds that will be used throughout this thesis.

Stirling's Approximation. We will often want to approximate terms containing factorials, for example, the probability of finding a minimal perfect hash function at random, as we will see later on, is $\frac{n!}{n^n}$. For that, we will make use of Stirling's approximation [e.g., Rom00]. It states for $n \in \mathbb{N}^+$

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \in \Theta\left(\sqrt{n}\frac{n^n}{e^n}\right)$$

where $a_n \sim b_n$ means $\frac{a_n}{b_n} \to 1$ $(n \to \infty)$. As shown in Figure 2.1, Stirling's approximation is close even for small n. Applying this to $\frac{n^n}{n!}$ we get

$$\frac{n^n}{n!} \sim \frac{n^n}{\sqrt{2\pi n}(\frac{n}{e})^n} = \frac{e^n}{\sqrt{2\pi n}} \in \Theta\left(\frac{e^n}{\sqrt{n}}\right).$$

Bernoulli's Inequality. For some proofs, it is helpful to recall Bernoulli's inequality [Einb]: For $a \in [-1, \infty)$, $n \in \mathbb{N}_0$ it holds

$$(1+a)^n \geq 1 + na.$$

2.3. Geometric Distribution and Entropy

When searching for minimal perfect hash functions at random, we will regularly encounter the geometric distribution. The geometric distribution $\mathrm{Geo}_1(p)$ describes the number of times a Bernoulli trial (success or fail) with success probability $p \in (0, 1]$ needs to be per-

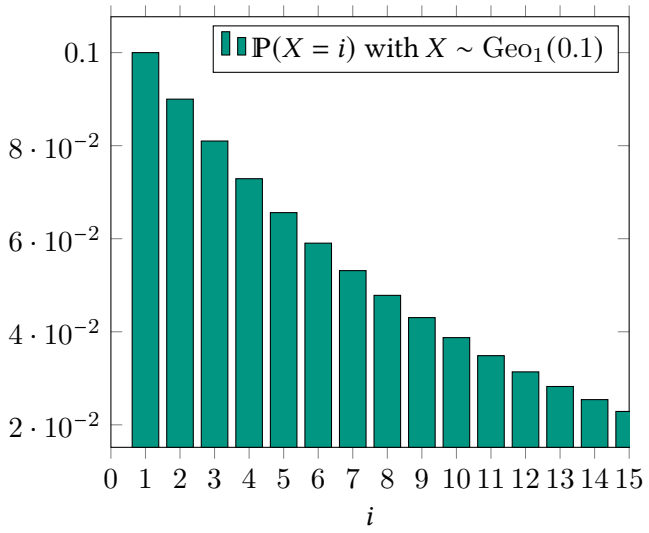


Figure 2.2.: Probability mass function of the Geo_1 distribution.

formed until and including the first success [Kle20]. More formally, if $X \sim \mathrm{Geo}_1(p)$ then $\mathbb{P}(X = i) = (1-p)^{i-1}p$, $i \in \mathbb{N}^+$, see Figure 2.2. Furthermore, it has an expected value of $\mathbb{E}[X] = \frac{1}{p}$.

We write Geo_1 because often it is also interesting to consider the number of times a Bernoulli trial has to be repeated *before* the first success occurs, which is a distribution on $\mathbb{N}_0$ instead of $\mathbb{N}^+$. If 0 is included in the outcomes, we write Geo_0. As a result, if $X \sim \mathrm{Geo}_1(p)$ then $X - 1 \sim \mathrm{Geo}_0(p)$.

2.3.1. Entropy

Apart from the expected value, another important property of a random variable is its entropy $H[X]$ [Sha48]. When X can take on $k \in \mathbb{N}^+$ different values with probabilities p_i, $i \in [k]$, its entropy is defined as $H[X] = -\sum_{i\in[k]} p_i \log_2 p_i \in [0, \infty)$. Entropy can be interpreted in many ways:

First, it can be seen as the expected information gained knowing the value of a random variable. The higher the entropy the more information can be expected to gain from the variable. For example, for $X \sim \mathrm{Uniform}([n])$ for $n \in \mathbb{N}^+$, its entropy is

$$H[X] = \sum_{i=1}^{n} -\frac{1}{n} \log_2 \frac{1}{n} = \log_2 n.$$

With a uniform distribution, we gain an equal amount of information from each possible value of $X \in [n]$. When $n = 1$, we now that always $X = 1$ and thus we do not learn anything new when "evaluating" X—we get $H[X] = 0$. As n gets larger, we can "encode" more different outcomes for X while each singular outcome becomes less likely—X carries more information.

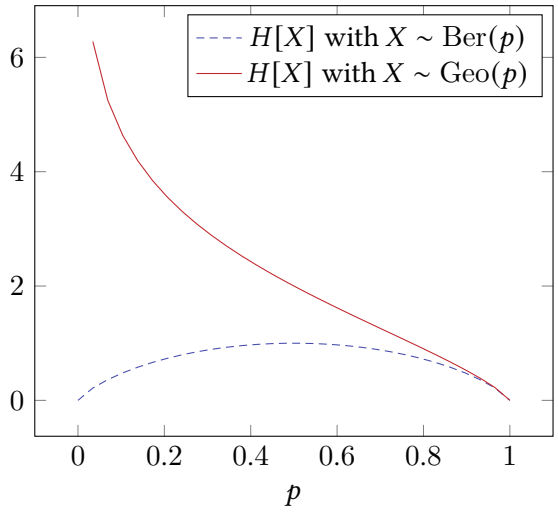


Figure 2.3.: Entropy of the Bernoulli and geometric distribution.

This leads to the second interpretation of entropy: the expected number of bits necessary to store X. This interpretation is motivated by Shannon's source coding theorem [Sha48], which effectively states that given a sequence of i.i.d. random variables $(X_i)_{i\in[m]}$, $m \in \mathbb{N}^+$ with entropy $H[X_i]$, $i \in [m]$, at least $m \cdot H[X_1]$ bits are required in expectation to losslessly encode $(X_i)_{i\in[m]}$. As a result, we will use entropy as a measure for required storage throughout this thesis. Moreover, this is justified as there exist codes that come close to this bound—in particular Golomb Rice codes for geometrically distributed X_i [Gol66 | RP71 | GvV75].

Note that when we only store a single geometrically distributed variable X by storing its binary representation with a variable length, we just use $\mathbb{E}[\log_2 X]$ as expected space usage. This only works for single values where we can assume to know the length of the binary representation through other means (e.g., having the total size of the data structure stored somewhere). Otherwise, for multiple values, we would not know where one value ends and another one starts.

Looking at further examples, we can determine the entropy of a Bernoulli trial $X \sim \mathrm{Ber}(p)$, $p \in [0, 1]$ to be $H[X] = -p \log_2 p - (1-p) \log_2(1-p)$, see Figure 2.3. The aforementioned geometric distributions $X \sim \mathrm{Geo}_0(p), \mathrm{Geo}_1(p)$ both have entropy [GvV75]

$$H_{\mathrm{Geo}}(p) := H[X] = -\log_2 p - \tfrac{1-p}{p} \log_2(1-p).$$

Subsection 4.2.3 and Section 5.1 will take a closer look at some properties of H_{Geo}.

2.4. Hashing

In essence, hashing allows for a mapping of data points, called *keys*, to codes, called *hash codes* or *hashes*, often of a fixed size [Kno75]. This is mainly done for easier or more efficient handling of the data itself. As the space of hash codes is smaller than the sometimes even infinite data space, some data points will get assigned the same hash. This is called a *hash col-*

lision. Typically, it is required that hash codes are assigned in a jumbled way. Requirements range from an even distribution of hashes to similar data getting assigned dissimilar hashes, or even the inability to find collisions efficiently in cryptographic applications [MvOV18]. A function providing such a mapping is called a *hash function.*

One big field for applications of non-cryptographic hash functions is hash tables [MS08]. These data structures use hashing to associate keys with values, so that—given a key—a value can be queried efficiently. For that, hash tables commonly use the hash code assigned to a key as an index into an array, where then possible collisions are handled in some way. Here, it is important that hash functions can be evaluated quickly while hashes are distributed evenly. In practice, accessing a value given a key often is possible in expected constant time, but collisions lead to a worse worst-case bound.

For this thesis, we assume having a randomly drawn *seeded hash function* $(f_i : U \to [n])_{i \in \mathbb{N}_0}$ with seeds $i \in \mathbb{N}_0$ for some universe of keys U and $n \in \mathbb{N}^+$, such that hash codes are uniformly and independently distributed:

$$\forall i \in \mathbb{N}_0,\ x \in U : f_i(x) \overset{i.i.d.}{\sim} \text{Uniform}([n]).$$

We assume we do not need to store anything for such a sequence of seeded hash functions and that they can be evaluated in constant time.

2.5. Minimal Perfect Hash Functions (MPHFs)

A *perfect* hash function on $X \subseteq U$ is a data structure that is constructed for a set of keys X. It provides a hash function $f : U \to [n]$ that is injective on X: $\forall x_1, x_2 \in X : x_1 \neq x_2 \implies f(x_1) \neq f(x_2)$. A *minimal perfect hash function (MPHF)* is a perfect hash function with $n = |X|$, making f bijective on X [FCH92]. Using an MPHF to get the hash for a key is called a *query.*

Interesting properties of MPHFs are construction time, query time, and space usage. Optimal values for that are linear construction time, constant query time, and $\log_2 e$ bits per key space usage. In this thesis, we will mostly focus on space usage with acceptable characteristics in the other categories and explain, how they can further be improved.

Perfect hash functions have the advantage, that they have no collisions for any keys in X. This allows for use in hash tables with worst-case constant query time [FKS84]. MPHFs further allow for a minimally sized backing array in extremely tight memory constraints, creating updatable retrieval data structures [MSSZ14]. Their downside is, however, that MPHF construction requires the key set X to be known at construction and most MPHFs cannot be easily updated.

One easy way to provide such a minimal perfect hash function would be to store every element of X in order in an array. To calculate $f(x)$ for $x \in X$, an algorithm could search for x in the array and return the index as $f(x)$ (see Algorithm 2.1).

Algorithm 2.1: Wasteful MPHF

```
Data: x ∈ X, array A of elements X
Result: f(x)
for i ∈ [n] do
    if A[i] = x then
        return i
```

This requires nK space to store the array A, where K is the size of the elements in X. One goal is to store f using as little space as possible. As $f(x)$ can be arbitrary for $x \notin X$, it is possible to store much less than an array of X.

2.5.1. Lower Bound

Let there be a finite universe U with $|U| = u$ and $n \in \mathbb{N}^+$. In the following, we will consider how many $f : U \to [n]$ are at least necessary so that for every $X \in \binom{U}{n}$ at least one of them is an MPHF. If we require at least k different hash functions, we thus would need at least $\log_2 k$ bits to differentiate between them [Meh84].

To estimate the minimal number k of hash functions required, we consider an upper bound on the number of sets $X \in \binom{U}{n}$ a single $f : U \to [n]$ can be an MPHF for. Let $x_i := |f^{-1}(\{i\})|$ be the number of times f takes on $i \in [n]$. f is an MPHF for all X with one element in $f^{-1}(\{1\})$, one in $f^{-1}(\{2\})$ and so on. This yields a total number of $\prod_{i\in[n]} x_i$ sets X. Using the AM-GM inequality [Ste04] we see that this product is maximal when all x_i are equal, as

$$\frac{u}{n} = \frac{1}{n}\sum_{i\in[n]} x_i \geq \sqrt[n]{\prod_{i\in[n]} x_i}$$

with equality iff all x_i are equal. Thus, we get

$$\prod_{i\in[n]} x_i \leq \left(\frac{u}{n}\right)^n.$$

Now, we can use this information to find a lower bound on the minimal number k of hash functions required to have an MPHF for every possible set $X \in \binom{U}{n}$. We simply divide the number of sets by the maximum number one hash function can cover to get the number of hash functions at least required.

$$k \geq \frac{\binom{u}{n}}{\left(\frac{u}{n}\right)^n} = \frac{u!}{(u-n)!\cdot n!}\frac{n^n}{u^n} \geq \frac{(u-n+1)^n}{u^n}\frac{n^n}{n!} = \left(1-\frac{n-1}{u}\right)^n \frac{n^n}{n!}.$$

Assuming $u \in \Omega(n^2)$, thus $u \geq \alpha n^2$ for some $\alpha > 0$ and for almost all $n \in \mathbb{N}^+$, we can further simplify

$$k \geq \left(1-\frac{n-1}{u}\right)^n \frac{n^n}{n!} \geq \left(1-\frac{n}{\alpha n^2}\right)^n \frac{n^n}{n!} = \underbrace{\left(1-\frac{1/\alpha}{n}\right)^n}_{\to e^{-1/\alpha}\ (n\to\infty)} \frac{n^n}{n!}$$

Algorithm 2.2: Finding a MPHF by brute force.

Data: X, Sequence of hash functions $(f_i : U \to [n])_{i \in \mathbb{N}_0}$ $(n = |X|)$
Result: Minimal perfect hash function for X

```
for i ∈ ℕ_0 (in ascending order) do
    if f_i|_X is bijective then
        return i
```

In the end, we need $\log_2 k$ bits to differentiate between k different hash functions. Thus, we need at least per key in bits:

$$\begin{aligned}
\frac{\log_2 k}{n} &\geq \frac{\log_2 \left(\left(1 - \frac{1/\alpha}{n}\right)^n \frac{n^n}{n!} \right)}{n} \\
&= \frac{\log_2 (\left(1 - \frac{1/\alpha}{n}\right)^n \beta_n \frac{e^n}{\sqrt{2\pi n}})}{n} && (\beta_n := \underbrace{\frac{n^n}{n!} \frac{\sqrt{2\pi n}}{e^n}}_{\to 1 \ (n \to \infty)} \text{ by Stirling}) \\
&= \log_2 e + \log_2 \left(1 - \frac{1/\alpha}{n}\right) + \frac{1}{n} \left(\log_2 \frac{\beta_n}{\sqrt{2\pi}} - \frac{1}{2} \log_2 n \right) \\
&\to \log_2 e \approx 1.44270 && (n \to \infty) \qquad (2.1)
\end{aligned}$$

In the limit for large n, we at least need $\log_2 e \approx 1.44270$ bits per key. This bound is tight, as we will see in Subsection 2.5.2.

This is in fact a lower bound: Suppose there exists an algorithm $\mathcal{A}$ that produces MPHFs of size $\log_2 k' < \log_2 k$ bits ($k' \in \mathbb{N}$) for any $X \in \binom{U}{n}$. This algorithm can at most produce $k' = 2^{\log_2 k'} < 2^{\log_2 k} = k$ different MPHFs. This contradicts the definition of k as the minimal number of such functions needed to have an MPHF for every $X \in \binom{U}{n}$.

2.5.2. Naive MPHF

Now that we know a lower bound on space usage for MPHFs, we will look at a naive construction, which matches this lower bound.

Given a seeded hash function $(f_i : U \to [n])_{i \in \mathbb{N}_0}$ $(n \in \mathbb{N}^+)$, each independently and uniformly chosen from the set of all functions $[n]^U = \{f \mid f : U \to [n]\}$ (think of a seeded hash function) and a set $X \in \binom{U}{n}$, one strategy to find a *minimal perfect* hash function is to iterate over $i \in \mathbb{N}_0$ until $f_i|_X$ is bijective (see Algorithm 2.2).

Let Y describe the number of functions checked for bijectivity until a bijective one is found. It turns out $Y \sim \mathrm{Geo}_1(p)$ with the probability p of a single hash function $f = f_i$ being bijective (for any $i \in \mathbb{N}_0$) is:

$$p = \frac{\#\text{of hash functions bijective on } X}{\#\text{of hash functions}} = \frac{\{f \in [n]^U \mid f|_X \text{ is bijective}\}}{|[n]^U|} = \frac{n^{u-n} n!}{n^u} = \frac{n!}{n^n}$$

The number of hash functions bijective on X is given by the observation, that with $x \in U \setminus X$ $f(x)$ can be chosen freely in $[n]$. However for $X = \{x_0, \ldots, x_{n-1}\}$, $f(x_0)$ can be chosen arbitrarily from $[n]$, $f(x_1)$ can be chosen from $[n] \setminus \{f(x_0)\}$ and so on, resulting in:

$$|[n]|^{|U \setminus X|} \cdot |[n]| \cdot |[n] \setminus \{f(u_0)\}| \cdot \cdots \cdot 1 = n^{u-n} n!$$

As a result, the expected number of checks is $\mathbb{E}[Y+1] = \frac{1}{p} = \frac{n^n}{n!} \in \Theta\left(\frac{e^n}{\sqrt{n}}\right)$ as Y is geometrically distributed. Thus, the expected runtime of Algorithm 2.2 grows exponentially in n, rendering it unfeasible for large n.

Considering the required space, this approach looks much better: If we simply store the index i that Algorithm 2.2 returned using a binary encoding, we need expected bits:

$$\mathbb{E}[\log_2 i] \leq \log_2 \mathbb{E}[i] = \log_2 \mathbb{E}[Y] \leq \log_2 \frac{1}{p} = \log_2 \frac{n^n}{n!}.$$

Looking at the derivation of Equation 2.1 we see that this approach reaches minimal space usage in expectation.

To sum up, constructing an MPHF by brute force searching through hash functions provides optimal space usage but has impractically exponential construction time. This thesis will improve this approach, based on RecSplit [EGV19], to gain better construction time while retaining near-optimal space usage.

3. Related Work

In this chapter, we look at various MPHF implementations using different approaches to get a picture of the state of research. First, we take a look at an older technique of bucket placement. After that, we explore fingerprinting for MPHF construction. Then we describe RecSplit [EGV19] on which the most recent, most space-efficient constructions are based on and which also form a basis for the MPHF introduced in this thesis. Finally, we take a look at some approaches using cuckoo hashing. Figure 3.1 gives an overview of the here presented MPHF algorithms.

It shall be noted, that the provided numbers for space usage only point in a rough direction and are not necessarily directly comparable. Those numbers state the best-achieved result for a realistic runtime that was stated in each paper. Thus, they were not performed on the same hardware and had different runtime constraints. Providing an accurate comparison between all of them is out of the scope of this thesis.

3.1. Bucket placement

Some approaches try to construct MPHFs incrementally by partitioning the input keys into buckets and finding injective hash functions on each bucket that do not collide with the previous ones. Apart from mentioning [BBD09], we want to present one lineup of MPHFs that build upon each other:

FCH. Fox et al. (FCH) [FCH92] construct an MPHF in three steps: First, they partition the input keys into multiple buckets unevenly, such that 60% of the keys get mapped to 30% of the buckets. Then, they order the buckets in decreasing size. Then, they assign hashes to each key—bucket by bucket in the order of the previous step—that collide with previously assigned hashes. For that, a global hash function is searched first that does not produce a collision in any bucket. Afterward, for each bucket, FCH randomly search a displacement value that is added to the hashes of each key until no collision occurs. To further increase the chance of success, a second global hash function may be tried. In the end, two seeds for partitioning and hash assignment, and for each bucket the displacement value (and whether the second global hash function was used) are stored. Their approach reaches 2.5 bits per key while having fast queries.

PTHash. PTHash [PT21] modifies FCH's approach by changing the last step. Instead of a displacement value that gets added on, PTHash applies a seeded random offset to each hash. The advantage of this approach is that seeds for the offsets do not need to be searched

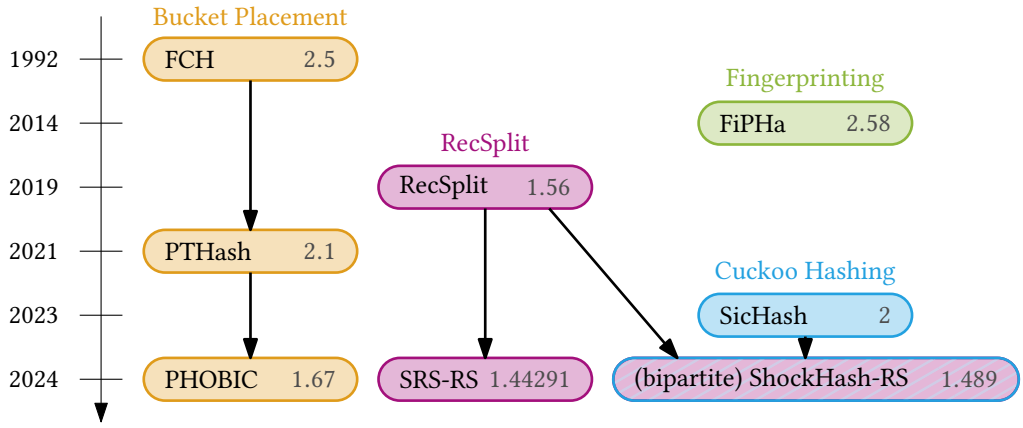


Figure 3.1.: Non-exhaustive overview about MPHF algorithms as they are presented in this chapter. Year of publication is shown on the left. Reached space usage per key is shown next to the names. Arrows indicate how the algorithms got improved. Colors indicate the concepts they are based on.

at random but can be tried out in order. Thus, they are compressible which then decreases space usage. For compression, PTHash uses—amongst others—a lookup table linking buckets to potentially multiply used seeds. PTHash reaches 2.1 bits per key while maintaining fast queries.

PHOBIC. PHOBIC [Her+24] improves on PTHash by optimizing the non-uniform bucket size distribution. Furthermore, PHOBIC performs another uniform partitioning step in the beginning to encode the seed for the ith bucket of each partition together, as they have the same statistical distribution. PHOBIC reaches as little as 1.67 bits per key.

3.2. Fingerprinting

Other approaches use fingerprinting for MPHF construction. **FiPHa** [MSSZ14] reaches 2.58 bits per key. It calculates a fingerprint (hashes in $[n]$ for input size n) for each key. For each fingerprint that has no collision, a bit in a bit vector is set. All keys that cause a collision get moved to the next layer. Here, the same is repeated for the remaining keys with a new fingerprinting function. This is performed for a fixed number of layers when a fallback MPHF handles the remaining keys to achieve constant query time—or until no keys are left for logarithmic query time. In the end, the bit vectors are concatenated and stored together with a rank data structure [NP12].

To query a key, FiPHa calculates its fingerprint. If the bit indexed by the fingerprint is set, FiPHa returns the rank of this bit. Otherwise, it repeats the same for the second layer staring at the nth bit, which contains $n - \text{rank}(n)$ keys.

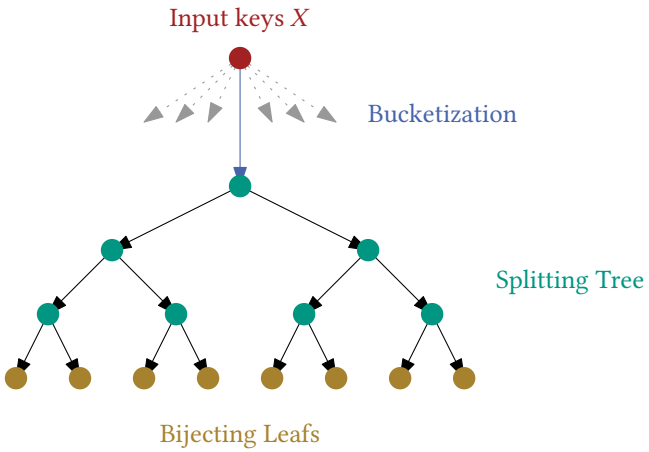


Figure 3.2.: Structure of RecSplit. First, the input keys are partitioned into buckets of expected constant size. Then, they are split recursively until they have a specified leaf size in each node, where a bijection is searched directly. For simplicity, the splitting tree for only one bucket is shown.

3.3. RecSplit

As the MPHF proposed in this thesis is based on RecSplit [EGV19], we describe it in more detail. It allows for space usage down to 1.56 bits per key.

Let $X \subseteq U$ be the input key set of size $|X| = n \in \mathbb{N}^+$ that should be mapped bijectively to $[n]$. First, RecSplit partitions the keys X into buckets of expected constant size b. This partitioning is done using an ordinary hash function with codomain $[\frac{n}{b}]$ where keys with the same hash are grouped together. Then, RecSplit constructs a separate splitting tree for each of these buckets as follows:

Using seeded hash functions this time, RecSplit again splits the buckets' keys into two parts (or more, depending on a splitting strategy). This time, however, an even split is achieved by trying out seeds, until one is found where each hash value occurs equally often (a difference of one is allowed for an odd number of keys). RecSplit then splits these two parts again and again, until they reach a leaf size l or smaller, see Figure 3.2. For these leafs, an MPHF is searched using brute force similar to Subsection 2.5.2. All the seeds found are then encoded using a Golomb–Rice encoding [Gol66 | RP71] and concatenated in preorder of the splitting tree.

Afterward, these bit strings for each bucket are concatenated as well. Furthermore, the prefix sums of bucket sizes and the bucket starting positions inside the bit string are encoded using an Elias–Fano encoding [Eli74 | Fan71]. In the end, the encoded seeds, the starting positions of the buckets therein and the prefix sum of bucket sizes form the MPHF data structure.

To query the MPHF, a key is first hashed into its bucket and then the corresponding splitting tree is descended, each time hashing the key with the current node's seed to decide which node to visit next. While traversing, RecSplit keeps track of the number of keys that lie left of the current node. This way, RecSplit obtains an index of the queried key inside the bucket, and using the prefix sum of bucket sizes it obtains the final hash value.

SRS-RecSplit (Section 4.4) mainly differs from RecSplit in the way it searches the splitting hash functions, and that there are no big leafs—binary splittings are searched until all keys are separated. The latter is possible because our approach does not waste up to $\log_2 e$ bits per splitting, as RecSplit does.

3.4. Cuckoo Hashing Based

Some approaches for MPHF construction are based on cuckoo hashing. **Cuckoo hash tables** [PR04] achieve constant lookup time by avoiding collisions. In their basic form, they achieve this by using two hash functions and two arrays instead of one as described in Section 2.4. On access, cuckoo hashing probes the array cell in each array that is indexed by the hash of the queried key under the respective hash function. Cuckoo hash tables guarantee, that if a key is contained in the hash table, it is in one of those two cells.

To insert a key in the hash table, cuckoo hashing tries to insert it in its cell in the first array. If this cell is not empty, cuckoo hashing relocates the element stored there to its cell in the second array. If this cell is occupied as well, the same is repeated there, moving the value to the first array. This kicking out (that got cuckoo hashing its name) is repeated until either an empty cell is eventually reached or a maximum number of iterations is exceeded. In the latter case, the hash table needs to be rebuilt from the ground up with new hash functions. Rebuilding happens rarely enough to achieve expected constant insertion time if both arrays together are less than half full.

3.4.1. SicHash

SicHash [LSW23] makes use of a form of cuckoo hashing to construct an MPHF. It allows for space usage down to 2 bits per key but achieves faster construction than, e.g., RecSplit for larger space usage, especially for non-minimal perfect hash functions. SicHash first partitions the input keys into buckets of the same expected size, similar to RecSplit. For each bucket, a generalized irregular cuckoo hash table is constructed at full capacity, that uses d different hash functions instead of 2. Furthermore, it uses different d for different keys, decided by another hash function. As cuckoo hash table construction may fail, after some retries, each cuckoo hash table has a seed for which construction succeeded. Then, for each key, SicHash stores which of the d hash functions was used in the cuckoo hash table using a retrieval data structure [DHSW22] that can represent static functions with low space usage. In the end, SicHash stores a prefix sum of bucket sizes, the seeds for the cuckoo hash tables, and the retrieval data structure.

To query the MPHF, SicHash first determines the bucket the key belongs to. Then, using the retrieval data structure, it determines which of the d hash functions was finally used for the cuckoo hash table with the stored seed. Using this hash function, the key's unique index inside the bucked is determined, which then gets added to the sum of all previous bucket sizes to get the final hash.

3.4.2. ShockHash

ShockHash [LSW24] uses ideas of SicHash to provide an alternative to finding an MPHF using brute force, as done in RecSplit's leafs, that requires less work by an exponential factor. For that, it constructs an ordinary (only two hash functions) cuckoo hash table at full capacity and stores the seed of the successful cuckoo hash table, as well as which of the two hash functions was used for which key using a retrieval data structure. Before cuckoo hash table construction, ShockHash performs an efficient check whether every hash is hit in the first place to filter out hash functions that cannot lead to a successful construction early on. Then, it checks whether construction can be successful using an algorithm to detect pseudoforests—graphs with at most one edge more per component than a regular forest.

Unlike the cuckoo hash tables as described above, ShockHash (and SicHash) uses one combined array, into which both hash functions index. A further improvement—bipartite ShockHash—goes back to using separate arrays and hash functions. This allows for keeping a pool of hash functions that hit every hash. Out of this pool, ShockHash then searches two hash functions that together successfully form a cuckoo hash table.

To be efficient on large inputs, (bipartite) ShockHash-RS combines ShockHash with RecSplit and reaches as low as 1.489 bits per key.

In theory, it would be possible to combine ShockHash-RS with SRS (Section 4.3) to achieve lower space usage than ShockHash-RS on its own. However, as SRS-RecSplit (Section 4.4) already reaches excellent space usage by splitting down to single keys, this would mainly add further complexity. Yet, it is thinkable that using larger leafs would allow for better cache efficiency and faster queries. In Section 7.3 we experimentally compare SRS-RecSplit to bipartite ShockHash-RS as it represents the state of the art regarding low-space MPHFs.

4. Algorithm

This chapter describes what Symbiotic Random Search (SRS) can be used for, motivates why its approach is necessary, and introduces the underlying algorithm. Furthermore, we will apply SRS to constructing MPHFs, creating SRS-RecSplit.

4.1. Problem

Before we introduce SRS, we want to define the problem it is useful for and establish some terminology. Given a sequence of tasks $[m]$, $m \in \mathbb{N}^+$, for each task $j \in [m]$ there is a Bernoulli process $(X_j^i)_{i \in \mathbb{N}_0}$ where $X_j^i \overset{i.i.d.}{\sim} \mathrm{Ber}(p_j)$ for every *seed* $i \in \mathbb{N}_0$, $p_j \in (0, 1]$. Let $S_j = \{i \in \mathbb{N}_0 \mid X_j^i = 1\} \subseteq \mathbb{N}_0$ be the *successful seeds* for task j.

The goal is to find and represent successful seeds $i_1, \ldots, i_m$ where $\forall j \in [m] : i_j \in S_j$ while trying to minimize expected space usage and expected construction time. For that, *testing* one seed—that is, one membership query on S_j—shall take work at most W_j ($j \in [m]$).

Example. This problem occurs in the RecSplit splitting tree (see Section 3.3 for more details). Here, tasks $j \in [m]$ represent nodes of the splitting tree, p_j is the probability of finding a splitting/bijection at node j, X_j^i indicates whether the seeded hash function h_i provides a valid splitting/bijection for seed i and thus S_j is the set of all splitting/bijecting seeds. In the end, i_j ($j \in [m]$) are represented and stored individually using a Golomb–Rice coding.

4.2. Two Obvious Solutions

There are two obvious approaches to solving this problem. In the following, we will consider finding a successful seed for each task independently (IND) and one seed for all combined (UNI), see Figure 4.1. We will see that IND has low work requirements, while UNI has good space usage. In detail, we will see that IND, as used by RecSplit, wastes up to $\log_2 e$ bits per task compared to UNI.

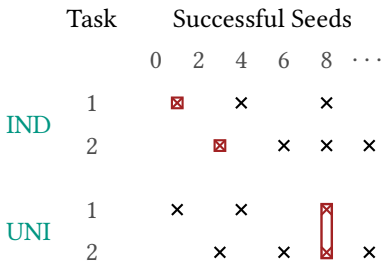


Figure 4.1.: Schematic representation of which successful seeds IND and UNI choose. Successful seeds are marked as a cross. Chosen seeds are highlighted with a red box. IND chooses the first successful seed for each task, UNI the first that is successful for both tasks.

4.2.1. Independent Search (IND)

Maybe the most natural approach, as done by RecSplit, is to consider each task $j \in [m]$ independently and then search through the seeds using brute force:

$$i_j = \min S_j \sim \mathrm{Geo}_0(p_j)$$

As a result, the **work** required is

$$\mathbb{E}[\mathcal{R}_{\mathrm{IND}}] \leq \mathbb{E}\left[\sum_{j\in[m]} W_j(i_j+1)\right] = \sum_{j\in[m]} W_j \mathbb{E}\left[i_j+1\right] = \sum_{j\in[m]} \frac{W_j}{p_j}.$$

For the required **space usage** using an entropy optimal encoding, we get in bits

$$\mathbb{E}[\mathcal{S}_{\mathrm{IND}}] = \sum_{j\in[m]} H[i_j] = \sum_{j\in[m]} H_{\mathrm{Geo}}(p_j)$$

where $H_{\mathrm{Geo}}(p) = -\log_2 p - \frac{1-p}{p}\log_2(1-p)$ is the entropy of the $\mathrm{Geo}(p)$ distribution ($p \in (0,1]$), see Section 5.1. This may seem like a good solution, as runtime is linear in m (for constant $p = p_j,\ j \in [m]$). However, considering another approach, we will see that space usage is not optimal.

4.2.2. Unified Search (UNI)

In this approach, we try to find a single seed $i \in \mathbb{N}_0$ that is successful for all tasks, that is, $\forall j \in [m] : i \in S_j$. We again evaluate them in ascending order by using brute force. In other words,

$$i_j = i = \min \bigcap_{j\in[m]} S_j \sim \mathrm{Geo}_0\left(\prod_{j\in[m]} p_j\right).$$

For the **work** required for finding this value, we get the following bounds (considering that when one task fails, we immediately can try the next seed without evaluating the remaining tasks):

$$\mathbb{E}[\mathcal{R}_{\mathrm{UNI}}] \begin{cases} \leq \mathbb{E}[i+1] \cdot \sum_{j\in[m]} W_j = \prod_{j\in[m]} \frac{1}{p_j} \cdot \sum_{j\in[m]} W_j \\ \geq \mathbb{E}[i+1] \cdot \min_{j\in[m]} W_j = \prod_{j\in[m]} \frac{1}{p_j} \cdot \min_{j\in[m]} W_j \end{cases}$$

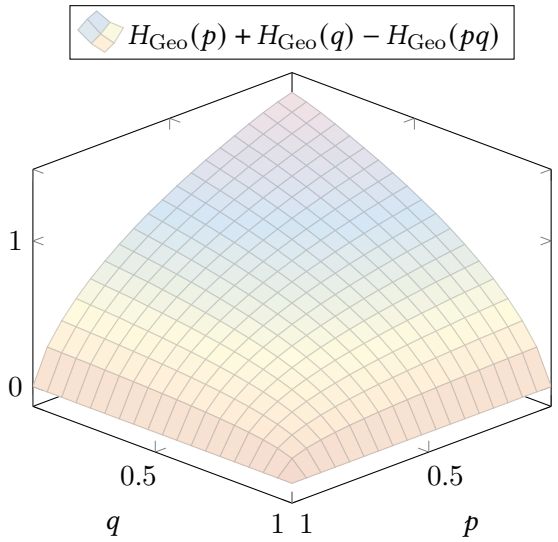


Figure 4.2.: Discrepancy between $H_{\text{Geo}}(p) + H_{\text{Geo}}(q)$ and $H_{\text{Geo}}(pq)$. For small success probabilities $p, q \to 0$, this approaches $\log_2 e \approx 1.44$.

As we only need to store one seed, **space usage** using an entropy optimal encoding is

$$\mathbb{E}[\mathcal{S}_{\text{UNI}}] = H[i] = H_{\text{Geo}}\left(\prod_{j\in[m]} p_j\right).$$

In the following, we will discover that this approach requires around $\log_2 e$ fewer bits per task than IND. However, UNI requires work exponential in m.

4.2.3. Entropy Discrepancy between IND and UNI

To compare the space usage of UNI and IND, we need to look closer at the entropy of a geometric distribution. For that, we compare the entropy of two individual geometrically distributed random variables $X \sim \text{Geo}_0(p)$ and $Y \sim \text{Geo}_0(q)$ with parameters $p, q \in (0, 1]$ to a single variable $Z \sim \text{Geo}_0(pq)$. We will find that X and Y together have higher entropy than Z.

For that, we analyze the entropy of a geometrically distributed variable as presented in Section 5.1. Proofs are shown in Section 5.1.

Lemma 4.1: *For $p \in (0, 1]$ it holds*

$$H_{\text{Geo}}(p) = -\log_2 p - \frac{1-p}{p}\log_2(1-p) \in -\log_2 p + \log_2 e - \log_2 e \cdot \frac{p}{2} - \mathcal{O}\left(p^2\right).$$

With this big O approximation, which tells us about the behavior of $H_{\text{Geo}}(p)$ for small p, we compare the combined entropy of X and Y with the entropy of Z.

Corollary 4.2: *For $p, q \in (0, 1]$ it holds*

$$H_{\mathrm{Geo}}(p) + H_{\mathrm{Geo}}(q) \in H_{\mathrm{Geo}}(pq) + \log_2 e \left(1 - \frac{p + q - pq}{2}\right) + \mathcal{O}\left(p^2 + q^2\right).$$

We find that for small success probabilities p, q there are up to $\log_2 e$ bits overhead, see also Figure 4.2. Applying this finding inductively, we get that n geometrically distributed variables loose up to $(n-1)\log_2 e$ bits compared to a single geometrically distributed variable with the combined probability:

Corollary 4.3: *It holds for $p_i \in (0, 1],\ i \in [n],\ n \in \mathbb{N}^+$*

$$\sum_{i \in [n]} H_{\mathrm{Geo}}(p_i) \in H\left(\prod_{i \in [n]} p_i\right) + \log_2 e \cdot \left(n - 1 - \frac{\sum_{i \in [n]} p_i - \prod_{i \in [n]} p_i}{2}\right) + \mathcal{O}\left(\sum_{i \in [n]} p_i^2\right). \quad (4.1)$$

With these findings, we can compare the space usage of IND and UNI. As IND stores m separate geometrically distributed seeds and UNI only one, we get with Corollary 4.3

$$\mathbb{E}[\mathcal{S}_{\mathrm{IND}}] \in \mathbb{E}[\mathcal{S}_{\mathrm{UNI}}] + \log_2 e \cdot \left(m - 1 - \frac{\sum_{j \in [m]} p_j - \prod_{j \in [nm]} p_j}{2}\right) + \mathcal{O}\left(\sum_{j \in [m]} p_j^2\right).$$

Thus, IND loses up to $\log_2 e$ bits per additional task compared to UNI for small p_j ($j \in [m]$). This motivates us to search for another solution to the problem stated in Section 4.1, that achieves IND's linear work requirements and UNI's space usage.

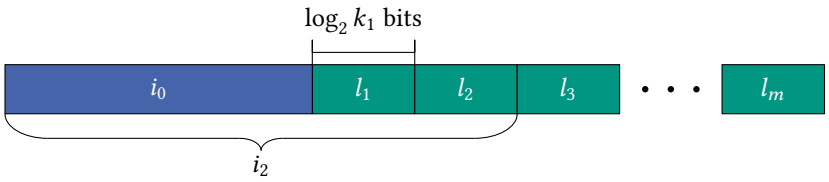


Figure 4.3.: Storage representation for indices allowing for quick access of the seeds.

4.3. Symbiotic Random Search (SRS)

We now finally get to introduce Symbiotic Random Search (SRS). SRS solves the discrepancy in space usage between IND and UNI while still only requiring work linear in m. More precisely, SRS can reach space usage per task arbitrarily close to the one of UNI for large enough m.

In principle, SRS approaches the problem like IND. It searches for a successful seed for each task, one at a time. However, it limits the number of seeds tested for one task in one go to $k_j \in \mathbb{N}^+$. If a successful seed is found, SRS continues with the next task. If not, SRS backtracks to the previous task and tries to find another successful seed there. In essence, SRS lets tasks help each other out if they do not find a successful seed within their k_j tries. This gives SRS its name.

On its own, retrying the same k_j seeds after the previous task found another successful seed is not helpful as just the same failing seeds would be tested. However, in SRS, the concrete tested seeds i_j consist of the last successful seed i_{j-1} of the previous task $j-1$ and an *index* $l_j \in [k_j]_0$ by "concatenating" them (see also Figure 4.3). We use η to represent concatenation:

$$\begin{aligned} i_j &:= i_{j-1} \cdot k_j + l_j && (j \in [m]) \\ &= \Big(\cdots \big((i_0 \cdot k_1 + l_1) \cdot k_2 + l_2\big) \cdot \ldots\Big) \cdot k_j + l_j =: \eta(i_0, l_1, \ldots, l_j) \in \mathbb{N}_0. \end{aligned}$$

This way, new seeds are always tested when retrying. In practice, it suffices to only consider the least significant w bits of the seeds i_j for some large enough $w \in \mathbb{N}_0$ (e.g., $w = 64$), allowing for efficient computation. Subsection 5.2.4 motivates why this is the case.

As stated above, only k_j seeds can be tested without finding another seed for a previous task. Thus, SRS requires only $\log_2 k_j$ bits for task $j \in [m]$ to store l_j. Furthermore, an unlimited *root seed* $i_0 \in \mathbb{N}_0$ is required to determine i_1 and allow for an unlimited number of retries. This root seed will require only constant expected space, independent of m. Nevertheless, SRS can find a solution in expected time linear in m if k_j are chosen correctly. Moreover, this choice is possible for space usage per task arbitrarily close to UNI for large enough m.

In the end, SRS only stores the root seed i_0 and the sequence of indices $(l_j)_{j\in[m]}$ (which altogether are equivalent to the last seed i_m). Thus, the seeds i_j need to be calculable easily. This calculation is especially convenient when k_j are powers of two, as then i_0, $(l_j)_{j\in[m]}$ can be just concatenated in their binary representation without wasting any space, see Figure 4.3.

Thus, we will try to use k_j to bound the number of immediate retries that are powers of two. Later, Subsection 5.2.5 shows that

$$\log_2 k_j = \lceil \sigma(j) \rceil - \lceil \sigma(j-1) \rceil, \quad j \in [m]$$
$$\sigma(j) := \sum_{j'=1}^{j} \left(\omega - \log_2 p_{j'}\right), \; j \in [m] \cup \{0\}$$

is a good choice for any $\omega \in (0, 1]$, which introduces a runtime–space tradeoff. σ keeps track of the total desired bits used up to task j. This may not be a natural number, but by rounding, the required bits are "averaged out" along the neighboring tasks.

Algorithm 4.1 describes SRS in detail. For each task (l. 3) we try out k_j indices (l. 4). We combine an index l_j with the previous task's seed—consisting of all previous indices and the root seed i_0—to the seed i_j for this try (l. 5). If this seed is a successful seed (l. 6) we continue with the next task (l. 9). Once we find a successful seed in the final task, we are done (l. 7,8). If none of the k_j indices lead to a successful seed (l. 10), we give up for now and go back to the previous task where we continue with the next index. Should the latter happen in the first task, we retry with a new root seed.

In abstract, SRS can be thought of as a depth-first search on a directed acyclic graph (DAG), see Figure 4.4. For each task, there is a node for every possible seed. Out of each circular node with a successful seed for its task (except for the last tasks), there are k_{j+1} edges leading to nodes for the next task. Shown on the left with squares are additional nodes to allow for an infinite number of retries which represent the possible root seeds i_0. SRS now tries to find a path from the leftmost pink node () to any node with a successful seed of the final task, here shown with a checkmark on the right.

We will now quickly summarize SRS work and space requirements, as they are analyzed in Section 5.2. Storing SRS = $(i_0, l_1, \ldots, l_m)$ requires **space usage**

$$\mathbb{E}[\mathcal{S}_{\text{SRS}}] = \mathbb{E}[\log_2 i_0] + \sum_{j \in [m]} \log_2 k_j \in \mathcal{O}\left(\log_2 \frac{1}{\omega}\right) + m\omega - \sum_{j \in [m]} \log_2 p_j,$$

see Subsection 5.2.3 and Subsection 5.2.5. Remembering

$$\mathbb{E}[\mathcal{S}_{\text{UNI}}] = H_{\text{Geo}}\left(\prod_{j \in [m]} p_j\right) \in -\log_2 \prod_{j \in [m]} p_j + \mathcal{O}(1) = -\sum_{j \in [m]} \log_2 p_j + \mathcal{O}(1),$$

we can see that we can choose ω such that SRS's space usage per task gets arbitrarily close to the one of UNI if m is sufficiently large.

Analyzing the **work** required is less trivial, but in Section 5.2 we essentially get

$$\mathbb{E}[\mathcal{R}_{\text{SRS}}] \in \mathcal{O}\left(\frac{1}{\omega} \sum_{j \in [m]} \frac{W_j}{p_j}\right)$$

which is linear in m.

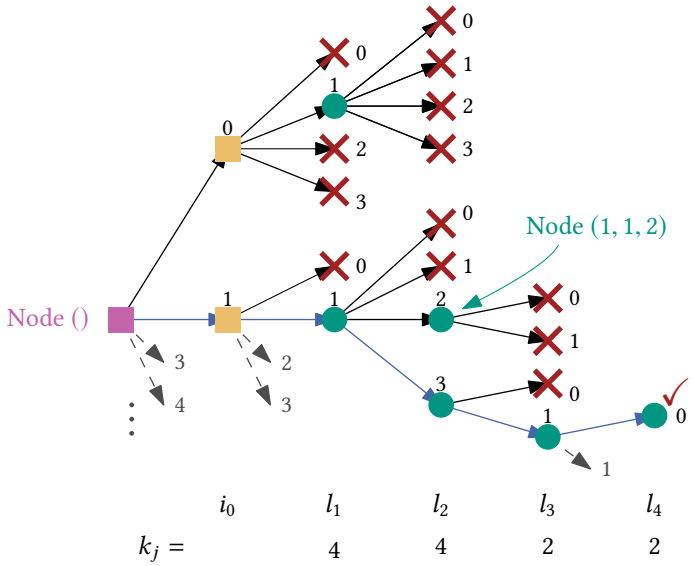


Figure 4.4.: SRS can be thought of as a depth-first search through a random DAG with nodes $V = \bigcup_{j=0}^{m}(\mathbb{N}_0 \times [k_1]_0 \times \cdots \times [k_j]_0) \cup \{()\}$ and edges $E = \{((i_0, l_1, \ldots, l_{j-1}), (i_0, l_1, \ldots, l_{j-1}, l_j)) \in V^2 \mid i_j = \eta(i_0, l_1, \ldots, l_{j-1}, l_j) \in S_j,\ j \in [m]\} \cup \{()\} \times \mathbb{N}_0$, starting at $()$ and trying to reach any node in $\mathbb{N}_0 \times [k_1]_0 \times \cdots \times [k_m]_0$. Neighbors are discovered in increasing order of their integer value. In the end, SRS will store $(i_0, l_1, l_2, l_3, l_4) = (1, 1, 3, 1, 0)$ as $1\,01\,11\,1\,0_2$. Circle nodes are associated with an SRS task. Not to be confused with the RecSplit splitting tree in Figure 4.5.

Algorithm 4.1: Construction of the SRS data structure. With $i_j = \eta(i_0, l_1 \ldots, l_j)$, the seeds can be retrieved.

Data: $m \in \mathbb{N}^+,\ (k_j \in \mathbb{N}^+)_{j\in[m]},\ (S_j \subseteq \mathbb{N}_0)_{j\in[m]}$
Result: $(i_0, l_1, \ldots, l_m) \in \mathbb{N}_0 \times [k_1]_0 \times \cdots \times [k_m]_0$

```
for i_0 ∈ ℕ_0 do
    if not ⊥ return FindSeedTask(1, (i_0, l_1, ..., l_m))

Function FindSeedTask(j ∈ [m], (i_0, l_1, ..., l_{j-1}) ∈ ℕ_0 × [k_1]_0 × ··· × [k_{j-1}]_0):
    Result: (i_0, l_1, ..., l_m) ∈ ℕ_0 × [k_1]_0 × ··· × [k_m]_0 or ⊥
    for l_j ∈ [k_j]_0 do
        i_j := η(i_0, l_1, ..., l_j)                    // Integer representation
        if i_j ∈ S_j then
            if j = m then
                return (i_0, l_1, ..., l_m)
            if not ⊥ return FindSeedTask(j + 1, (i_0, l_1, ..., l_j))
    return ⊥
```

4.4. Applying SRS to MPHF Construction (SRS-RecSplit)

In this section, we use SRS to construct an MPHF. We use the recursive splitting idea of RecSplit but search and store splitting indices using SRS. Furthermore, we will perform binary splittings all the way down to avoid leafs of larger sizes. In practice, SRS-RecSplit reaches a space usage as low as $1.442\,91$ bits per key with reasonable construction time. For similar construction time we beat the previous record holder—bipartite ShockHash-RS [LSW24]—by having $1/192$ of its overhead.

More importantly, the theoretical expected construction time of SRS-RecSplit is linear in $\frac{1}{\omega}$ were ω is the overhead per key—as opposed to most previous approaches having an exponential characteristic therein [EGV19 | LSW24]. As we present SRS-RecSplit without bucketization, it has expected construction time in $\mathcal{O}\left(n^{3/2} \log n\right)$ and query time in $\mathcal{O}(\log n)$. We mention using bucketization to gain expected linear construction and constant query time briefly in Subsection 4.4.3.

In the following, we first consider only input sizes that are powers of two, as they allow for easier explanation. Then, we look at the changes necessary to also handle non-power-of-two inputs. Finally, we will briefly mention bucketization to reach optimal construction and query time.

4.4.1. Construction for Power-of-Two Input Sizes

For simplicity, we will first look at cases where we want to construct an MPHF on $X \subseteq U$ with $|X| = n = 2^r$ for some $r \in \mathbb{N}^+$. SRS-RecSplit implicitly constructs a splitting tree, where in each node, the input is split into two equally sized parts using appropriate hash functions. This is repeated recursively until all keys are separated from each other. These hash functions $h_s : U \to \{0, 1\}$ ($s \in \mathbb{N}_0$) are searched using SRS, where each node corresponds to one task. In the end, only the information gathered by SRS will be stored[1]. Figure 4.5 shows how the input set is split and the order in which SRS visits the splitting tasks. Section 5.3 shows that construction for n keys has **runtime**

$$\mathbb{E}\left[\mathcal{R}_{\text{SRS-RecSplit}}\right] \in \mathcal{O}\left(\frac{n^{3/2} \log n}{\omega}\right)$$

while having **space usage**

$$\mathbb{E}\left[\mathcal{S}_{\text{SRS-RecSplit}}\right] \in \mathcal{O}\left(\log_2 \frac{1}{\omega}\right) + (n-1) \cdot \omega + \log_2 \frac{n^n}{n!}$$

for any $\omega \in (0, 1]$, which is roughly the overhead per key. Comparing with the naive MPHF construction in Subsection 2.5.2, we can see that we can reach space usage per key arbitrarily close to the lower bound by choosing a small ω. However, we achieve this without the exponential construction time of previous approaches. To perform a query—evaluating the MPHF for a given key—the splitting tree is traversed again, collecting all the hashes for each node along the path to the corresponding leaf. Thus, queries require work in $\mathcal{O}(\log_2 n)$.

[1] In practice, also storing the input size and chosen overhead parameter ω is helpful. These are, however, not accounted for in the bit-per-key space usage metric.

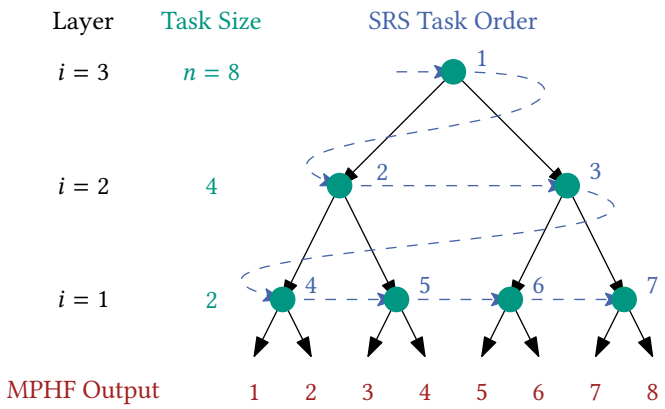


Figure 4.5.: Splitting tree implicitly constructed by SRS-RecSplit. Black non-dashed arrows show how the input keys are split in each layer. SRS tasks are represented by green markers. The order in which SRS traverses these tasks is marked by blue dashed arrows. Not to be confused with the SRS search tree in Figure 4.4.

We will now briefly describe the parameters that we have to supply to SRS. First, we have $m = n - 1$ splitting tasks, forming r layers $i \in r, \ldots, 1$ (see Figure 4.5). In each layer i (not to be confused with the seeds i_j as used in Section 4.3), tasks have the same parameters $p_i = \frac{2^i!}{2^{2^i}(2^{i-1}!)^2}$ (see Subsection 5.3.3). On layer i, S_j will be the set of seeds s for which the hash function h_s forms an even splitting on the keys corresponding to this task. Note, that even though the keys corresponding to a task depend on which seeds were successful for all ancestor tasks in the splitting tree, the sets of successful seeds S_j are nevertheless stochastically independent as we assume fully random hash functions, which we never[2] evaluate for the same key and seed twice. This dependence necessitates ordering an SRS task after all its ancestors.
W_i will be in $\mathcal{O}(2^i)$ as for testing a seed we need to evaluate h_s on each key corresponding to this task. To keep track of which keys correspond to which task, we partition the keys accordingly in the place we store them for construction after finding an even spitting seed for one task. Then, when we are in task c of layer i with task size 2^i, we know the cth chunk of 2^i keys belongs to this task. For the theoretical runtime, we can just use the same ω on each layer. By using cleverly rounded bounds k_j as described in Subsection 5.2.5 over all layers, we only have a size-dependent space overhead of $(n-1) \cdot \omega$. In practice, it can be beneficial to alter ω over the layers, using $\omega_i = \omega\sqrt{2^i}$ on layer i. Subsection 5.3.2 goes into detail why this is advantageous in practice.

4.4.2. Non-Power-of-Two Input Sizes

Only being able to per construct MPHFs for power-of-two input sizes is not very useful. Luckily, we can apply the same approach for non-power-of-two input sizes. However, instead of splitting non-power-of-two inputs in a task into equally sized parts, we split them into the largest possible power-of-two and the rest. This way, on each layer we have at most one *uneven task* with a non-power-of-two input size, the rest being *ordinary tasks*, see Figure 4.6.

[2] If we limit seeds to the least significant w bits that is not quite true, Subsection 5.2.4 motivates why this is still not a problem in practice.

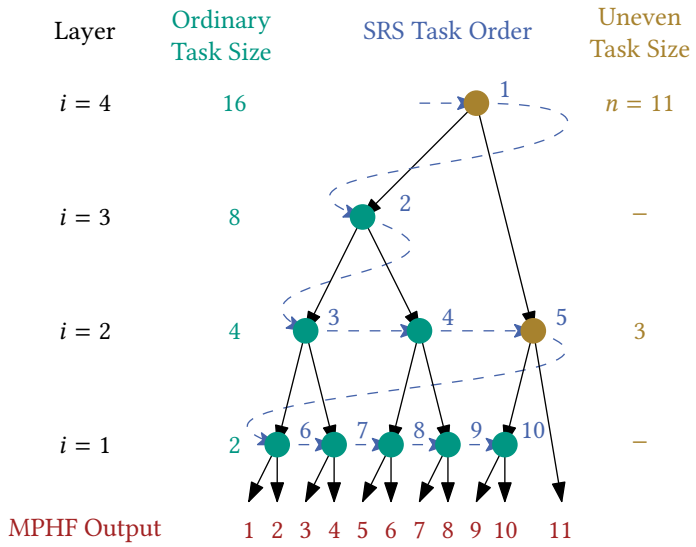


Figure 4.6.: Splitting tree for non-power-of-two input sizes. Some layers have an uneven task where an uneven split is performed, separating out the next smaller power of two.

This fact is important for calculating the cleverly rounded bounds efficiently, as we do not want to iterate through all tasks to access a specific seed when performing queries. Now, calculating how many SRS tasks are required and the concrete splitting probabilities is a bit more complicated, but still sufficiently efficiently possible. Subsection 5.3.3 shows that we still need $m = n - 1$ tasks and find uneven splits of size $\tilde{n}$ with $p(\tilde{n}) \approx 1/\sqrt{2\pi\tilde{n}q(1-q)}$ where $q := \frac{2^{\lfloor \log_2 \tilde{n} \rfloor}}{\tilde{n}}$.

4.4.3. Bucketization

Apart from space usage, the other important metrics of MPHFs are construction and query time. For SRS-RecSplit to reach (expected) optimal results here, we can apply bucketization as RecSplit does as well. We would choose a constant targeted bucket size b and then partition all keys into n/b buckets of expected size b using a hash function with codomain n/b. As the input size to SRS-RecSplit as described above would be expected constant, we would get expected construction time linear in n/b and thus n. Query time would be expected constant. By setting a hard limit on maximal bucket size and choosing a bucketization hash function accordingly, worst-case constant query time might also be possible. We could then concatenate all the data SRS-RecSplit generates for each bucket. However, we then would need to store starting positions for each bucket in this long bit string of all the SRS data. Furthermore, we would need to keep a prefix sum of bucket sizes to calculate the final hashes. These could be encoded using an Elias–Fano coding [Eli74 | Fan71] for monotone sequences. As SRS-RecSplit reaches far smaller space overheads than RecSplit, it is unclear how large we would need to choose buckets to not waste too much space on starting positions and bucket sizes. Scaling bucket sizes with $1/\omega$—which leads to the number of buckets scaling with ω—seems promising to decrease the overheads of storing starting positions and bucket sixes for small omega as well. Further analyzing SRS-RecSplit with bucketization is out of scope for this thesis and remains future work.

5. Analysis

This chapter provides more details on the conclusions and choices made in Chapter 4. First, we will revisit the entropy function for geometrically distributed random variables in Section 5.1, as this motivated the use of SRS in the first place. Then, we will prove the runtime and space usage of SRS for various cases from easy to complex in Section 5.2. This will be the largest part of this chapter. Finally, we will use these results to analyze runtime and space usage of SRS-RecSplit in Section 5.3 for easy power-of-two input sizes, and also show what changes for non-power-of-two cases.

5.1. Entropy of the Geometric Distribution

This section contains proofs for the lemmas given in Subsection 4.2.3 about the entropy of the geometric distribution, $H_{\text{Geo}}(p) = -\log_2 p - \frac{1-p}{p}\log_2(1-p),\ p \in (0,1]$.

First, we approximate this function in big O notation.

Lemma 4.1: *For $p \in (0,1]$ it holds*

$$H_{\text{Geo}}(p) = -\log_2 p - \frac{1-p}{p}\log_2(1-p) \in -\log_2 p + \log_2 e - \log_2 e \cdot \frac{p}{2} - \mathcal{O}\left(p^2\right).$$

Proof. Let $f(x) = -\frac{1-x}{x}\ln(1-x),\ x \in (0,1]$.
As $\ln(1-x) = -\sum_{k=1}^{\infty}\frac{x^k}{k},\ x \in [0,1)$ [ASM65, p. 68],

$$\begin{aligned} f(x) &= -\left(1-\frac{1}{x}\right)\sum_{k=1}^{\infty}\frac{x^k}{k} = -\sum_{k=1}^{\infty}\frac{x^k}{k} + \sum_{k=1}^{\infty}\frac{x^{k-1}}{k} = -\sum_{k=1}^{\infty}\frac{x^k}{k} + \sum_{k=0}^{\infty}\frac{x^k}{k+1} \\ &= -\sum_{k=1}^{\infty}\frac{x^k}{k} + 1 + \sum_{k=1}^{\infty}\frac{x^k}{k+1} = 1 - \sum_{k=1}^{\infty}\left(\frac{x^k}{k} - \frac{x^k}{k+1}\right) = 1 - \sum_{k=1}^{\infty}\left(\frac{1}{k} - \frac{1}{k+1}\right)x^k \\ &= 1 - \sum_{k=1}^{\infty}\frac{1}{k(k+1)}x^k \in 1 - \frac{x}{2} - \mathcal{O}\left(x^2\right), \quad x \in [0,1) \end{aligned}$$

Thus for $p \in (0,1]$

$$\begin{aligned} H_{\text{Geo}}(p) &= -\log_2 p - \frac{1-p}{p}\log_2(1-p) = -\log_2 p + \log_2 e \cdot f(p) \\ &\in -\log_2 p + \log_2 e - \log_2 e \cdot \frac{p}{2} - \mathcal{O}\left(p^2\right). \end{aligned}$$

□

Next, we use this approximation to look at the difference in entropy between two separate geometric distributions and one combined.

Corollary 4.2: *For $p, q \in (0,1]$ it holds*

$$H_{\mathrm{Geo}}(p) + H_{\mathrm{Geo}}(q) \in H_{\mathrm{Geo}}(pq) + \log_2 e \left(1 - \frac{p+q-pq}{2}\right) + \mathcal{O}\left(p^2 + q^2\right).$$

Proof. Let $f(x) = -\frac{1-x}{x}\ln(1-x),\ x \in (0,1]$. Using Lemma 4.1 we get

$$\begin{aligned} H(p) + H(q) - H(pq) &= -\log_2 p - \log_2 q + \log_2(pq) + \log_2 e\,(f(p) + f(q) - f(pq)) \\ &\in \log_2 e \left(1 - \frac{(p+q-pq)}{2}\right) - \mathcal{O}\left(p^2 + q^2\right). \end{aligned}$$

□

We lose up to $\log_2 e$ bits for small p, q. We then apply this inductively, to get a similar statement for n geometrically distributed variables. Again, we lose up to $\log_2 e$ bits for each additional variable.

Corollary 4.3: *It holds for $p_i \in (0,1],\ i \in [n],\ n \in \mathbb{N}^+$*

$$\sum_{i\in[n]} H_{\mathrm{Geo}}(p_i) \in H\left(\prod_{i\in[n]} p_i\right) + \log_2 e \cdot \left(n - 1 - \frac{\sum_{i\in[n]} p_i - \prod_{i\in[n]} p_i}{2}\right) + \mathcal{O}\left(\sum_{i\in[n]} p_i^2\right). \quad (4.1)$$

Proof. Let $p_i \in (0,1]$ ($i \in \mathbb{N}^+$). For $n = 1$ the statement is trivial. For the other n proof by induction over $n \in \mathbb{N}^+ \setminus \{1\}$: For $n = 2$ the statement follows from Corollary 4.3 (COR).

Assume the statement holds for some fixed but arbitrary $n \in \mathbb{N}^+ \setminus \{1\}$ (IH). It holds

$$\begin{aligned} &\sum_{i\in[n+1]} H(p_i) = \sum_{i\in[n]} H(p_i) + H(p_{n+1}) \\ &\overset{\text{IH}}{=} H\left(\prod_{i\in[n]} p_i\right) + H(p_{n+1}) + \log_2 e \cdot \left(n - 1 - \frac{\sum_{i\in[n]} p_i - \prod_{i\in[n]} p_i}{2}\right) + \mathcal{O}\left(\sum_{i\in[n]} p_i^2\right) \\ &\overset{\text{COR}}{=} H\left(\prod_{i\in[n+1]} p_i\right) + \log_2 e \left(1 - \frac{\prod_{i\in[n]} p_i + p_{n+1} - \prod_{i\in[n]} p_i \cdot p_{n+1}}{2}\right) + \mathcal{O}\left(\prod_{i\in[n]} p_i^2 + p_{n+1}^2\right) \\ &\qquad + \log_2 e \cdot \left(n - 1 - \frac{\sum_{i\in[n]} p_i - \prod_{i\in[n]} p_i}{2}\right) + \mathcal{O}\left(\sum_{i\in[n]} p_i^2\right) \\ &= H\left(\prod_{i\in[n+1]} p_i\right) + \log_2 e \cdot \left((n+1) - 1 - \frac{\sum_{i\in[n+1]} p_i - \prod_{i\in[n+1]} p_i}{2}\right) + \mathcal{O}\left(\sum_{i\in[n+1]} p_i^2\right). \end{aligned}$$

□

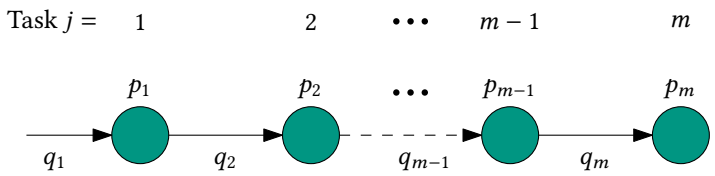


Figure 5.1.: Tasks in the order in which SRS visits them. p_j is the probability that a single seed in task j is successful. q_j is the probability that FindSeedTask returns successfully from task j.

5.2. Analysis of SRS

We now analyze the runtime and space usage of SRS. We start with the work required for a simpler case with p_j, k_j constant (Subsection 5.2.1) and then relax this requirement to only requiring $p_j k_j \geq 1 + \varepsilon$ for some $\varepsilon > 0$ (Subsection 5.2.2). Then, we look at the space usage of these cases (Subsection 5.2.3). Afterward, we motivate why it is enough to consider the last w bits of the seeds in practice (Subsection 5.2.4). In the end, we consider cleverly rounded bounds k_j (Subsection 5.2.5) to get to the final space usage as advertised in Section 4.3.

When looking at Algorithm 4.1 in Section 4.3, the following observations can be made:

- If a single call to the recursive function FindSeedTask returns *successfully* ($\neq \perp$), calls to all following tasks must have returned successfully and all calls to previous tasks will return successfully directly after. The main work is done while descending to task $j = m$.
- When FindSeedTask *fails* (returns $\perp$), some time after it is re-entered again with new previous indices $(i_0, l_1, \ldots, l_{j-1})$.

Let $q_j = \mathbb{P}\big(\text{FindSeedTask}(j, i_{j-1}) \neq \perp\big)$ for $j \in [m]$ (see Figure 5.1). We set $q_{m+1} := 1$ for convenience as the mth task has no subsequent tasks its success depends on. This probability is independent of the seed $i_{j-1} \in \mathbb{N}_0$, as $(i \in S_j)$ is i.i.d. for $i \in \mathbb{N}_0$. Let Q_j be the work done for a *fixed* task $j \in [m]$, that is, the collective work of all calls to FindSeedTask$(j, \cdot)$ without work in subsequent calls to FindSeedTask$(j+1, \cdot)$.

Furthermore, let Y_j be the number of times $i_j \in S_j$ has to be tested until FindSeedTask$(j, \cdot)$ returns successfully. As $Y_j \sim \text{Geo}_1(p_j q_{j+1})$, we find that $\mathbb{E}\big[Q_j\big] = \mathbb{E}\big[W_j Y_j\big] = \frac{W_j}{p_j q_{j+1}}$. Note, that we do not consider work done for task $j+1$ here.

Considering the probability that FindSeedTask$(j, \cdot)$ fails $(1 - q_j)$, we get

$$q_j = 1 - (1 - p_j q_{j+1})^{k_j} \quad (j \in [m]).$$

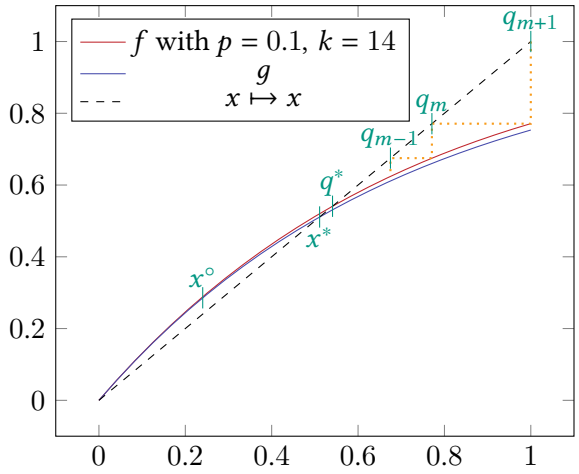


Figure 5.2.: Node success recursion function $f(x) = 1-(1-px)^k$ with its first two iterations applied to $1 = q_{m+1}$: $q_m = f(1)$, $q_{m-1} = f(f(1))$, bound below by $g(x) = 1 - e^{-pkx}$, their fixed points q^*, x^* respectively, visible as intersection with $x \mapsto x$, and the lower bound x° thereof.

5.2.1. Constant p, k

For simplicity, we will first assume that $\forall j \in [m] : p_j = p,\ k_j = k,\ W_j = W$. The lemmata referenced in the following are provided at the end of this subsection. We can now define

$$f(q) = 1 - (1 - pq)^k \quad (q \in [0, 1])$$

which gives us $q_j = f(q_{j+1}) = f^{m-j+1}(q_{m+1}) = f^{m-j+1}(1)$. One of the main observations that makes SRS work is that for correctly chosen p, k, the success probability of task $j = 1$ (whose success depends on the success of all the following tasks up to $j = m$), and therefore of the entire algorithm, is

$$q_1 = f^m(1) \to q^* > 0 \quad (m \to \infty).$$

Here, q^* is the fixed point of f such that $f(q^*) = q^* \in (0, 1]$ (see Figure 5.2), which exists according to Lemma 5.1 when $pk > 1$. As $f(q) < q$ for $q > q^*$ (Lemma 5.2), we get

$$1 = q_{m+1} \geq q_m \geq \cdots \geq q_1 \geq q^* > 0.$$

To approximate q^* we can use $g(x) = 1 - e^{-pkx} \leq f(x),\ x \in [0, 1]$ (see Lemma 5.4). g's fixed point x^*, if it exists, is $x^* \leq p^*$ (Corollary 5.3). We then can determine a lower bound x° to g's fixed point. By looking where $g'(x^\circ) = 1$ we get $x^\circ = \frac{\ln(pk)}{pk}$ requiring $pk > 1$ for the fixed point to exists (Lemma 5.5). We could do the same for f directly but using the simpler g will result in a simpler lower bound and will be useful later. In the end, we will get

$$0 < x^\circ \leq x^* \leq q^* \leq q_j \ (j \in [m]).$$

For the work required for SRS, we thus obtain for any $\varepsilon \in (0, 1]$ with $pk \geq 1 + \varepsilon > 1$ and $\omega := \log_2(1 + \varepsilon)$

$$\mathbb{E}[R_{\mathrm{SRS}}] = \sum_{j \in [m]} \mathbb{E}[Q_j] \leq \sum_{j \in [m]} \frac{W_j}{p_j q_j} \leq \frac{mW}{px^\circ} = \frac{mW}{p} \frac{pk}{\ln(pk)} \leq \frac{mW}{p} \frac{1+\varepsilon}{\ln(1+\varepsilon)} \in \mathcal{O}\left(\frac{mW}{p\omega}\right).$$

We will now prove the lemmas that were required for this conclusion. First, we show that the function f (which on iteration on $q_{m+1} = 1$ yields the success probabilities q_j of all the tasks) has in fact a fixed point, and some other properties that are useful later on.

Lemma 5.1: *Let* $f : [0,1] \to [0,1],\ q \mapsto 1-(1-pq)^k,\ p \in (0,1],\ k \in \mathbb{N}^+,\ pk > 1$.
Then, f *satisfies the requirements of Lemma 5.2 and has a fixed point* $q^* \in (0,1]$.

Proof. Let p, k, f be as required.
It holds

$$\begin{aligned} f(0) &= 1-(1-0)^k = 0 \\ f'(q) &= pk(1-pq)^{k-1} \\ f'(0) &= pk > 1 \\ f''(q) &= -p^2k(k-1)(1-pq)^{k-2} \leq 0 \quad \text{as } k \geq pk > 1. \end{aligned}$$

As $f(1) = 1-(1-p)^k \leq 1$, there either is a fixed point at $q^* = 1$ or at $q^* \in (0,1)$ by intermediate value theorem. □

Now we show, that if we are above the fixed point of f, we iterate closer and closer towards it. This allows us to state $q_{j+1} \leq q_j$ and also helps in a later argument. Furthermore, we show that this fixed point lies above the point where f has slope 1. We use this point as an approximation for the actual fixed point.

Lemma 5.2: *Let* $f \in C^2(\mathbb{R}_0^+),\ f(0) = 0,\ f'(0) > 1,\ f'' \leq 0$.
Given $\exists x_f \in \mathbb{R}^+ : f(x_f) = x_f$ *it holds:*

1) $f(x) < x$ *for* $x > x_f$

2) $x_f > \sup\{x \in \mathbb{R}_0^+ \mid f'(x) \geq 1\}$

Proof. Let f be as required. We will show that f, starting from $f(0) = 0$, rises above $id : x \mapsto x$. If it has a fixed point $x_f > 0$, which is another intersection with id, f will dive below id and stay there. The first phase above id is the only time f might reach $f'(x) = 1$, which it will if it comes back down, compare Figure 5.3.

Assume $x_f \in \mathbb{R}^+$ such that $f(x_f) = x_f$.

1) Let $\tilde{f}(x) := f(x) - x,\ x \in \mathbb{R}_0^+$. Then

$$\tilde{f}(x_f) = 0,\ \tilde{f}'(0) > 0,\ \tilde{f}''(x) \leq 0.$$

As $\tilde{f}'(0) > 0\ \exists \varepsilon \in (0, x_f) : \tilde{f}(\varepsilon) > 0$. Using the mean value theorem we get

$$0 > \underbrace{\tilde{f}(x_f)}_{=0} - \tilde{f}(\varepsilon) = \tilde{f}'(\xi)\underbrace{(x_f - \varepsilon)}_{>0} \quad \text{for a } \xi \in (\varepsilon, x_f) \implies \tilde{f}'(\xi) < 0.$$

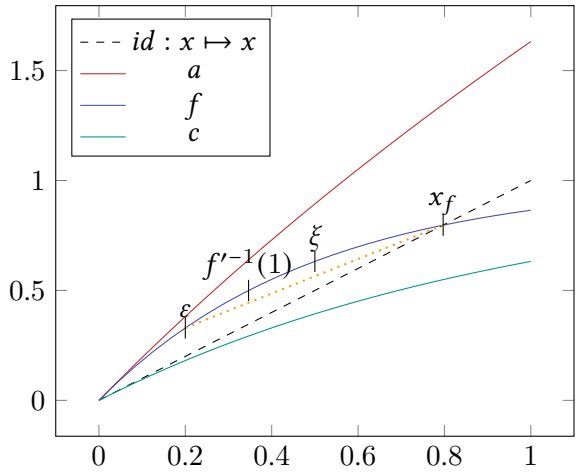


Figure 5.3.: Three functions with $a(0) = b(0) = f(0) = 0,\ a'', b'', f'' \leq 0$, and $a'(0), f'(0) > 1$ but not for except for b. a has no positive fixed point (intersection with id), but f has x_f and thus fulfills all the criteria. The plot also shows examples for ε, ξ from the proof. Note that the proof uses $\tilde{f}(x) := f(x) - x$ instead of f for simpler notation.

$$\implies \tilde{f}(x) = \tilde{f}(x_f) + \int_{x_f}^{x} \left(\underbrace{\tilde{f}'(\xi)}_{<0} + \int_{\xi}^{x'} \underbrace{\tilde{f}''(x'')}_{\leq 0}\, dx'' \right) dx' < \tilde{f}(x_f) = 0 \quad \text{for } x > x_f$$

Thus, $f(x) < x$ for $x > x_f$.

2) It holds

$$\tilde{f}'(x_f) = \tilde{f}'(\xi) + \int_{\xi}^{x_f} \underbrace{\tilde{f}''(x)}_{\leq 0}\, dx \leq \tilde{f}'(\xi) < 0.$$

Thus,

$$\tilde{f}'(x) = \tilde{f}'(x_f) + \int_{x_f}^{x} \underbrace{\tilde{f}''(x)}_{\leq 0}\, dx \leq f'(x) < 0 \quad x \geq x_f.$$

As a result,

$$(x \geq x_f \implies f'(x) < 1) \iff (f'(x) \geq 1 \implies x < x_f).$$

□

From this behavior of f around its fixed point, we conclude: If we find a function g that bounds f from below and if g also has a fixed point, the fixed point of g must be smaller than the one of f.

Corollary 5.3: *Let $f, g \in C^2(\mathbb{R}_0^+),\ g \leq f,\ f(0) = 0,\ f'(0) > 1,\ f'' \leq 0$. It holds:*

$$\exists x_f, g_f \in \mathbb{R}^+ : f(x_f) = x_f,\ g(x_g) = x_g \implies x_g \leq x_f$$

Proof. Let f, g be as required. Let $x_f, x_g \in \mathbb{R}^+$ such that $f(x_f) = x_f,\ g(x_g) = x_g$. Following Lemma 5.2, $f(x) < x$ for $x > x_f$. Thus, $g(x) \leq f(x) < x$ for $x > x_f$. As a result, g cannot have a fixed point at $x > x_f$, thus $x_g \leq x_f$. □

We now show that $g(x) = 1 - e^{-pkx}$ is such a function that bounds f from below, and thus the previous statement on its fixed point applies.

Lemma 5.4: *Let $x, p \in [0, 1]$, $k \in \mathbb{N}$. It holds: $1 - (1 - px)^k \geq 1 - e^{-pkx}$*

Proof. Let x, k, p be as required. It holds

$$(1 - px)^k = (1 - \frac{pxk}{k})^k \overset{n \geq k}{\leq} (1 - \frac{pkx}{n})^n \overset{n \to \infty}{\to} e^{-pkx}$$

as $1 \geq px \implies n \geq k \geq pkx$ thus the exponential sequence $(1 - \frac{a}{n})^n$ is increasing monotonically in n ($a \in \mathbb{R}$) [Eina]. □

Finally, we show in which cases g also has a fixed point and again approximate this fixed point from where g has slope one. This will then give us a bound of f's fixed point and thereby the probability that SRS succeeds at any of its tasks.

Lemma 5.5: *For $g_a : \mathbb{R}^+ \to \mathbb{R}^+, x \mapsto g_a(x) = 1 - e^{-ax}$ with $a \in \mathbb{R}^+$ the following holds:*

$$\exists x^* \in \mathbb{R}_0^+ : g_a(x^*) = x^* \iff a > 1 \tag{5.1}$$

$$\text{In that case } x^* > \frac{\ln a}{a} \tag{5.2}$$

Proof. Let $a,\ g_a$ be as required. It holds

$$\begin{aligned} g \text{ has a fixed point } x^* > 0 &\iff g(\epsilon) > \epsilon \text{ for some } \epsilon > 0 && \text{as } g(x) \to 1 < \infty \text{ for } x \to \infty \\ &\iff g'(0) = a > 1 && \text{as } g(0) = 0. \end{aligned}$$

Suppose g has a fixed point x^*. Following Lemma 5.2, $x^* > \max f'^{-1}(\{1\})$. It holds

$$f'(x) = ae^{-ax} \overset{!}{=} 1 \iff e^{-ax} = a^{-1} \iff ax = \ln a \iff x = \frac{\ln a}{a}.$$

Thus, $x^* > \frac{\ln a}{a}$. □

To sum up, we showed that if we keep $p_j = p,\ k_j = k$ constant and limit $pk \geq 1 + \varepsilon > 1$, all the success probabilities q_j that SRS succeeds on task j and all the following tasks are bound by $q_j \geq \frac{1+\varepsilon}{\ln(1+\varepsilon)}$. This makes SRS require work $\mathbb{E}[R_{\text{SRS}}] \in \mathcal{O}\left(\frac{mW}{p\omega}\right)$ in those cases. Next on, we will relax the conditions and show that our analysis also covers varying p_j, k_j as long as they all fulfill the condition $pk \geq 1 + \varepsilon > 1$ individually.

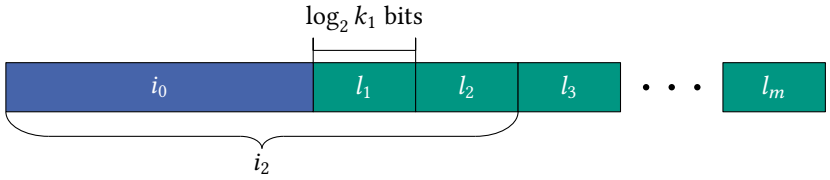


Figure 5.4.: Storage representation for indices allowing to access seeds quickly. Already shown as Figure 4.3 before.

5.2.2. Supporting Varying Task Probabilities p_j

As g only depends on the product pk, we immediately see that the analysis of Subsection 5.2.1 also applies for varying $p_j,\ k_j,\ j \in [m]$, as long as $p_j k_j \geq 1 + \varepsilon$ for all $j \in [m]$.

In detail, we can define

$$f_j(q) = 1 - (1 - p_j q)^{k_j} \quad (j \in [m])$$

and then can bound each individual f_j by $g(x) = 1 - e^{-(1+\varepsilon)x}$ with

$$q_j = f_j(f_{j+1}(\cdots f_m(1)\cdots)) \geq g^{m-j+1}(1) \geq x^* \geq \frac{\ln(1+\varepsilon)}{1+\varepsilon}.$$

Here, we again approximated the fixed point x^* of g in the same way as in Subsection 5.2.1 and get work

$$\mathbb{E}[R_{\mathrm{SRS}}] \leq \frac{1+\varepsilon}{\ln(1+\varepsilon)} \sum_{j\in[m]} \frac{W_j}{p_j} \in \mathcal{O}\left(\frac{1}{\omega} \sum_{j\in[m]} \frac{W_j}{p_j}\right).$$

5.2.3. Seed Representation

Now that we have an understanding of the work required for SRS, we want to take a look at its space usage. For that, we need to revisit how seeds are represented. We need to ensure that when FindSeedTask$(j, \cdot)$ for some fixed $j \in [m]$ is reentered, not the same—and thus useless—tests $i_j \in S_j$ are performed. For that, the tested seed i_j can not just be $l_j \in [k_j]_0$ but somehow must depend on the previously found seed i_{j-1}. To achieve this dependence, we set

$$\begin{aligned} i_j &:= i_{j-1} \cdot k_j + l_j && (j \in [m]) \\ &= \Big(\cdots\big((i_0 \cdot k_1 + l_1) \cdot k_2 + l_2\big) \cdot \ldots\Big) \cdot k_j + l_j =: \eta(i_0, l_1, \ldots, l_j) \in \mathbb{N}_0. \end{aligned}$$

In the end, we only store $(i_0, l_1, \ldots, l_m)$ with a fixed-length binary encoding for the indices l_j and a variable-length binary encoding for the root seed i_0.

To ensure seeds i_j can be calculated in $\mathcal{O}(1)$ we will only use k_j that are powers of two. This way, η just concatenates the binary representation of the indices l_j and the root seed i_0 in the order they are already stored, see Figure 5.4.

For that, we set k_j to the smallest power of two that satisfies $k_j p_j \geq 1 + \varepsilon$, as required by Section 5.2:

$$k_j = 2^{\left\lceil \log_2 \frac{1+\varepsilon}{p_j} \right\rceil} \begin{cases} \geq \frac{1+\varepsilon}{p_j} \\ \leq 2^{\log_2 \frac{1+\varepsilon}{p_j} + 1} = 2\frac{1+\varepsilon}{p_j} \end{cases}$$

As a result, storing SRS $= (i_0, l_1, \ldots, l_m)$ needs

$$\begin{aligned} \mathbb{E}[\mathcal{S}_{\text{SRS}}] &= \mathbb{E}[\log_2 i_0] + \sum_{j \in [m]} \log_2 k_j \overset{*}{\leq} \log_2 \mathbb{E}[i_0] + \sum_{j \in [m]} \left\lceil \log_2 \frac{1+\varepsilon}{p_j} \right\rceil \\ &\leq \log_2 \frac{1}{x^\circ} + \sum_{j \in [m]} \left(\log_2 \frac{1+\varepsilon}{p_j} + 1 \right) \\ &= \log_2 \frac{1+\varepsilon}{\ln(1+\varepsilon)} + \sum_{j \in [m]} \left(-\log_2 p_j + 1 + \log_2(1+\varepsilon) \right) \\ &= \log_2(1+\varepsilon) - \log_2 \ln(1+\varepsilon) + \sum_{j \in [m]} \left(-\log_2 p_j + 1 + \log_2(1+\varepsilon) \right) \end{aligned}$$

bits using Jensen's inequality [Jen06] for $*$ and where $x^\circ = \frac{1+\varepsilon}{\ln(1+\varepsilon)}$ was the bound on q_1 found in Subsection 5.2.1. Recalling IND's space usage (Subsection 4.2.1) and applying the big O approximation for the entropy of the geometric distribution (Subsection 4.2.3) to it, we can reformulate it for better comparison:

$$\mathbb{E}[\mathcal{S}_{\text{IND}}] = \sum_{j \in [m]} H[i_j] = \sum_{j \in [m]} H(p_j) \in \sum_{j \in [m]} \left(-\log_2 p_j + \log_2 e - \mathcal{O}(p_j) \right).$$

Knowing $\log_2 e \approx 1.44 > 1$, we see that SRS can already get better than IND for small enough ε, large enough m, and small p_j. Moreover, the +1 for every $j \in [m]$ can be avoided by a more intelligent rounding strategy, see Subsection 5.2.5. Next, we will motivate why we can just use the w least significant bits of each seed in practice.

5.2.4. Windowed Seeds

Just using powers of two for k_j is not enough for $\mathcal{O}(1)$ seed access as with increasing m, the bit length of the seeds increases linearly. Thus, we no longer can expect that operations on them can be performed in $\mathcal{O}(1)$. To solve this, we will only consider the last (least significant) w bits of the concatenated indices as the actual seeds i_j and call this modification of SRS the *windowed case*.

This modification leads to a couple of problems with our analysis. Firstly, it is now theoretically possible that windowed SRS cannot find a solution when none of the now only 2^w testable seeds could be successful, if only with a small probability. In practice, this is no problem because w can be set large enough (e.g., $w = 64$) so that not all 2^w possible seeds can be tested in any reasonable amount of time.

Another problem is that seeds might now be tested twice where they would not have been in the non-windowed case. This happens when SRS backtracks enough tasks to cower w bits, and then finds indices that result in exactly the same w as before, see Figure 5.5. In the non-windowed case, this is not a problem, as the seeds differ in some bit in front of the last w bits

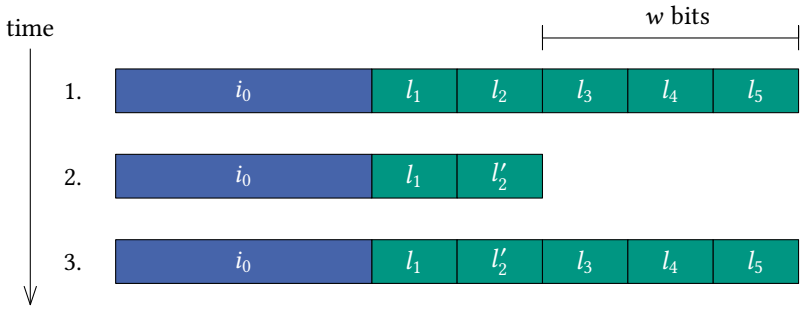


Figure 5.5.: A seed collision occurred. After backtracking (2.), the same indices for all tasks covering the last w bits were found again (3.). Windowed SRS will test the same seed consisting solely of l_3, l_4, l_5 again. Non-windowed SRS will not, as l_2 and l'_2 differ.

as SRS always increments some index. However, in the windowed case, only the last—now identical—w bits are considered for the seed, and so the same seed is tested again. We call this a *seed collision*. Seed collisions are a problem because SRS' runtime analysis assumes that new seeds are always tested with a success probability *independent* of every test before.

For simplicity, we now will again only consider the case where p, k, W do not vary among the tasks. In the following, we will sketch a proof that for some $w \in \Theta(\log_2 m)$, seed collisions are unlikely enough. We then argue, that if a seed collision should happen, the algorithm could be restarted with some new randomness and this would not change expected work. We further will consider p and ω to be a constant for our big O analysis and only let m be variable.

Let one step of SRS be the part of the execution between testing two indices ($i_j \in S_j$). One step thus requires work W. Non-windowed SRS requires $\mathbb{E}[\text{Steps non-windowed SRS}] \in \mathcal{O}\left(\frac{m}{p\omega}\right) = \mathcal{O}(m)$ steps as shown in Subsection 5.2.1 for non-varying p, k, W. Let C denote the event that a seed collision occurs during the first m^2 steps. Let $\mathcal{T}$ denote the number of steps of an algorithm.

Lemma 5.6: *For some $w \in \Theta(\log m)$ in non-windowed SRS with non-varying p, k, W where p, ω are considered constants, it holds for non-windowed SRS $\mathbb{P}(C) \in \mathcal{O}\left(\frac{1}{m}\right)$.*

Proof Sketch. Considering non-windowed SRS, for a seed collision to happen in one step, SRS needs to have been at this task before, and now find another seed with the same last w bits. For that, SRS needed to have backtracked at least $t := \frac{w}{\log_2 k}$ tasks (assume we chose w as a multiple of $\log_2 k$ for simplicity), see Figure 5.5. Then, SRS needs to find the exact same t indices as before. As C only considers the first m^2 steps, we get

$$\mathbb{P}(C) \leq m^2 \cdot \mathbb{P}(\text{colission in a specific step}) \leq m^4 \cdot \mathbb{P}(\text{specific last } w \text{ bits reached}).$$

We use that in the first m^2 steps there are at most m^2 indices that we could have found for the same task before. This is a very generous approximation but sufficient for our purposes. We get

$$\mathbb{P}(\text{specific last } w \text{ bits reached}) = p^t = p^{w/\log_2 k}.$$

Thus, with $w = 5 \log_2 k \log_2 m \in \Theta(\log m)$ we have

$$\mathbb{P}(C) \leq m^4 \cdot p^{-w/\log_2 k} = \frac{1}{m} \in \mathcal{O}\left(\frac{1}{m}\right).$$

□

Now that we have argued that a seed collision happening in the first m^2 steps is unlikely, we want to motivate why seed collisions are not a problem in practice. For that, we provide an adaptation of SRS that has runtime linear in m, regardless of any negative effect seed collisions could have on the runtime of SRS. To be specific, we will run windowed SRS, but restart after m^2 steps. The idea is that when no seed collision occurred, it is unlikely SRS reaches m^2 steps before returning successfully. As seed collisions are rare, we only restart SRS a constant expected number of times.

Corollary 5.7: *Windowed SRS*

- *when restarted after m^2 steps without ending successfully*
- *with non-varying p, k, W*
- *and p, ω are considered constants,*

requires expected number of steps $\mathbb{E}[\mathcal{T}] \in \mathcal{O}(m)$.

Proof Sketch. Let M be the event, that SRS reaches m^2 steps. It holds for windowed SRS:

$$\begin{aligned}\mathbb{P}(M) &= \mathbb{P}(M, C) + \mathbb{P}\left(M, \bar{C}\right) = \mathbb{P}(M \mid C) \cdot \mathbb{P}(C) + \mathbb{P}\left(M \mid \bar{C}\right) \cdot \mathbb{P}\left(\bar{C}\right) \\ &\leq 1 \cdot \mathbb{P}(C) + \mathbb{P}\left(M \mid \bar{C}\right) \cdot 1 \\ &\in \mathcal{O}\left(\frac{1}{m}\right) + \mathcal{O}\left(\frac{1}{m}\right) = \mathcal{O}\left(\frac{1}{m}\right).\end{aligned}$$

Thereby we use Lemma 5.6 for $\mathbb{P}(C) \in \mathcal{O}\left(\frac{1}{m}\right)$. As SRS without seed collisions is equivalent to non-windowed SRS, we get $\mathbb{P}\left(M \mid \bar{C}\right) \in \mathcal{O}\left(\frac{1}{m}\right)$ by using the definition of the expected value operator

$$\mathcal{O}(m) \ni \mathbb{E}[\mathcal{T}_{\text{non-windowed SRS}}] = \sum_{t=0}^{\infty} t \cdot \mathbb{P}(\text{non-windowed SRS takes } t \text{ steps}) \geq m^2 \cdot \mathbb{P}(M).$$

Now that we know that restarting is rare, be it because we had a seed collision (which voids our runtime analysis from before) or just by chance, we can consider the total number of steps required for windowed SRS with restarting (SRS-win-rest). We call it a new *round* when we restart SRS.

Let Σ be the event that SRS-win-rest is successful in less than m^2 steps. For the number of steps $\mathcal{T}_{\text{SRS-win-rest}}$ we get for sufficiently large m

$$\begin{aligned}
\mathbb{E}[\mathcal{T}_{\text{SRS-win-rest}}] &= \sum_{r=1}^{\infty} \mathbb{E}[\mathcal{T}_{\text{SRS-win-rest}} \mid r \text{ rounds}] \cdot \mathbb{P}(r \text{ rounds}) \\
&\leq \sum_{r=1}^{\infty} (m^2(r-1) + \mathbb{E}[\mathcal{T}_{\text{SRS-win}} \mid \Sigma]) \cdot \mathbb{P}(r \text{ rounds}) \\
&= \mathbb{E}[\mathcal{T}_{\text{SRS-win}} \mid \Sigma] + m^2 \sum_{r=1}^{\infty} (r-1)(\frac{\alpha}{m})^{r-1} = \mathbb{E}[\mathcal{T}_{\text{SRS-win}} \mid \Sigma] + m^2 \sum_{r=0}^{\infty} r(\frac{\alpha}{m})^{r} \\
&= \mathbb{E}[\mathcal{T}_{\text{SRS-win}} \mid \Sigma] + m^2 \frac{\alpha/m}{(1-\alpha/m)^2} = \mathbb{E}[\mathcal{T}_{\text{SRS-win}} \mid \Sigma] + \alpha m \frac{1}{(1-\alpha/m)^2} \\
&\in \mathbb{E}[\mathcal{T}_{\text{SRS-win}} \mid \Sigma] + \mathcal{O}(m)
\end{aligned}$$

using $\sum_{r=0}^{\infty} r x^r = \frac{x}{(1-x)^2}$ for $0 < x < 1$. We reach r rounds with probability less than $\left(\frac{\alpha}{m}\right)^{r-1}$ as we need to reach m^2 rounds $r-1$ times (each requiring m^2 steps), as $\mathbb{P}(M) \in \mathcal{O}(1/m)$ (which provides the $\alpha > 0$) as shown above.

All that now remains is to evaluate the expected number of steps that windowed SRS will reach given that it succeeded in less than m^2 steps. We intuitively get

$$\begin{aligned}
\mathbb{E}[\mathcal{T}_{\text{SRS-win}} \mid \Sigma] &= \mathbb{E}[\mathcal{T}_{\text{SRS-win}} \mid C, \Sigma] \cdot \mathbb{P}(C \mid \Sigma) + \mathbb{E}\left[\mathcal{T}_{\text{SRS-win}} \mid \bar{C}, \Sigma\right] \cdot \mathbb{P}\left(\bar{C} \mid \Sigma\right) \\
&\in m^2 \cdot \mathcal{O}(1) + \mathcal{O}(m) \cdot 1 = \mathcal{O}(m).
\end{aligned}$$

We used that having a seed collision in less than m^2 steps is even less likely than in m^2. Furthermore, without a seed collision, we already know non-windowed SRS requires $\mathcal{O}(m)$ steps in expectation, which only gets smaller when considering only executions with less than m^2 steps until success.

Combining these results, we get $\mathbb{E}[\mathcal{T}_{\text{SRS-win-rest}}] \in \mathcal{O}(m)$. □

To summarize, we argued that in SRS a seed collision rarely happens, and we theoretically could restart SRS after m^2 steps if it did not finish successfully. We would still require only $\mathcal{O}(m)$ steps, regardless of the runtime we would get when having a seed collision. We should note, that this is purely a theoretical construct and may not be implemented in practice. Next, we will use a more clever choice for k_j to use even less space.

5.2.5. Clever Rounding of Bounds k_j

In Subsection 5.2.3 we analyzed SRS' space usage for power-of-two bounds k_j that satisfy $p_j k_j \geq 1 + \varepsilon > 1$. However, requiring powers of two leads to having to choose bounds up to twice as large as strictly necessary. Although not using powers of two with a clever encoding scheme would be possible, we will choose another route to avoid this overhead that is simpler to implement but may be harder to analyze. Instead, we will vary k_j such that only on average, we get $k_j p_j \approx 1 + \varepsilon$. This has the effect, that some bounds will be larger and some will be smaller, thus we cannot simply use our analysis from before. In the following, we

will analyze SRS' work for cases where the bounds are b-periodic for some $b \in \mathbb{N}^+$. The idea is, that while in a period, some k_j might be too small, but overall average out and provide enough tries to SRS to not lose its linear work requirement. We also show that choosing b-periodic bounds is not a limiting requirement, by proving that one can for every targeted ε (remember $\omega = \log_2(1+\varepsilon)$) choose a smaller $\tilde{\varepsilon}$ that leads to periodic bounds but does not increase work asymptotically. Enforcing periodic bounds is again purely a theoretical tool for analytical purposes and not implemented in practice.

Before coming to these analytical tricks, we will start simple with the choice of the bounds, as it also will be implemented. As stated, we do want on "average" get $k_j = \frac{1+\varepsilon}{p_j}$, but only use powers of two. For that, we thus define

$$\sigma(j) := \sum_{j'=1}^{j} \log_2 \frac{1+\varepsilon}{p_{j'}} \in \mathbb{R}_0^+ \quad j \in [m] \cup \{0\}$$

to be the fractional number of bits we would have liked to spend for tasks $j' = 1$ up to $j' = j$. Then we can set

$$\log_2 k_j = \lceil \sigma(j) \rceil - \lceil \sigma(j-1) \rceil \in \mathbb{N}_0 \quad j \in [j] \tag{5.3}$$

which ensures $\sum_{j'=1}^{j} \log_2 k_j = \lceil \sigma(j) \rceil \approx \sigma(j)$ for the total space usage of all indices and k_j to be powers of two.

Space usage

Having made a choice for the bounds, we can now analyze the resulting space usage. By adding up the expected space required for the root seed and the indices, we get

$$\begin{aligned}
\mathbb{E}[\mathcal{S}_{\text{SRS}}] = \mathbb{E}[\log_2 i_0] + \lceil \sigma(m) \rceil &\leq \log_2 \mathbb{E}[i_0] + 1 + \sum_{j \in [m]} -\log_2 \frac{1+\varepsilon}{p_j} \\
&\leq \log_2 \frac{1}{q_1} + m \log_2(1+\varepsilon) + 1 + \sum_{j \in [m]} -\log_2 p_j \\
&= \log_2 \frac{1}{q_1} + m \log_2(1+\varepsilon) + 1 + \sum_{j \in [m]} -\log_2 p_j \\
&\leq \log_2 \frac{1}{\alpha\varepsilon} + m \log_2(1+\varepsilon) + 1 + \sum_{j \in [m]} -\log_2 p_j \\
&\in \mathcal{O}\left(\log_2 \frac{1}{\omega}\right) + m\omega - \sum_{j \in [m]} \log_2 p_j
\end{aligned}$$

with q_1 being the probability for SRS to succeed on the first task. In the following work analysis we will show $q_1 \geq q^\dagger \in \Omega(\varepsilon)$, thus $q_1 \geq \alpha\varepsilon$ for a constant $\alpha > 0$. Also note that $\varepsilon \in \Theta(\omega)$ as $\omega = \log_2(1+\varepsilon)$.

We recall the space usage of OPT (Subsection 4.2.2):

$$\mathbb{E}[\mathcal{S}_{\text{OPT}}] = H\left(\prod_{j \in [m]} p_j\right) \in -\log_2 \prod_{j \in [m]} p_j + \mathcal{O}(1) = \mathcal{O}(1) - \sum_{j \in m} \log_2 p_j.$$

We can see that SRS' space usage per task can get arbitrarily close to the one of UNI, by choosing a small enough ω, given a sufficiently large m. It now remains to show that $q_1 \in \Omega(\varepsilon)$ and SRS still retains a linear work requirement. This is the difficult part.

Work analysis

Next, we endearingly analyze the work required for SRS with the above-stated choice of bounds k_j. As stated above, $\forall j \in [m] : p_j k_j \geq 1 + \varepsilon$ does not hold anymore, as some k_j are rounded down. For simplicity, let again be $p_j = p,\ j \in [m]$. We have to revisit our runtime analysis from Subsection 5.2.1 to verify that for the success probabilities of SRS in total still holds $q_1 \to q^* \geq q^\dagger > 0$ for $m \to \infty$ and for each individual task $\forall j \in [m] : q_j \geq q^\dagger$ for some $q^*, q^\dagger > 0$ independent of m.

We will proceed as follows: First, we will make the observation that if we choose ε in a particular way, we will get b-periodic k_j from the choice in (5.3). Assuming b-periodic bounds k_j, we will set up a recursion function for q_j similar to f from Subsection 5.2.1, this time however applying it b times recursively with different k_j for one period. We then analyze its fixed point in a fashion similar to Subsection 5.2.1. This fixed point gives us a bound to every b-th q_j. We now need to also approximate every q_j in between to get a lower bound on every q_j. Having made a specific choice for ε and with it b we will see that $q_j \in \Omega(\varepsilon)$. This leads us to a runtime linear in $1/\varepsilon$ in the end, as we hoped for. We will also go a step further by showing that restricting our choices to ε as stated above is not a problem. For that, given any $\varepsilon \in (0, 1]$, we will choose a smaller $\tilde{\varepsilon} < \varepsilon$ leading to less space usage that does meet the requirements for periodicity. We then will use this $\tilde{\varepsilon}$ instead for our analysis but in the end show work still linear in $1/\varepsilon$.

We will start by showing how b and $\tilde{\varepsilon}$ have to be chosen to make our analysis work. To achieve b-periodic bound k_j, we require the *targeted* bits per task $\log_2 \frac{1+\varepsilon}{p}$ to be a rational number with denominator $b \in \mathbb{N}^+$. A good choice will be

$$b = \left\lceil \frac{2}{\log_2(1+\varepsilon)} \right\rceil$$

because this will lead to our $1/\varepsilon$ work we want to achieve using the approximation we achieve later on.

For an arbitrary $\varepsilon \in (0, 1]$, there exists $a \in \mathbb{N}^+, \xi \in [0, \frac{1}{b})$ such that

$$\log_2 \frac{1+\varepsilon}{p} = \frac{a}{b} + \xi.$$

Thus we try to choose $\tilde{\varepsilon} \in (0, 1]$ such that

$$\log_2 \frac{1+\tilde{\varepsilon}}{p} = \frac{a}{b} = \log_2 \frac{1+\varepsilon}{p} - \xi \in \mathbb{Q}.$$

We then get

$$\begin{aligned}\varepsilon \geq \tilde{\varepsilon} &= (1+\varepsilon) \cdot 2^{-\xi} - 1 > (1+\varepsilon) \cdot 2^{-\frac{1}{b}} - 1 = (1+\varepsilon) \cdot 2^{-1/\lceil 2/\log_2(1+\varepsilon) \rceil} - 1 \\ &\geq (1+\varepsilon) \cdot 2^{-\frac{\log_2(1+\varepsilon)}{2}} - 1 = \sqrt{1+\varepsilon} - 1 > 0\end{aligned}$$

which we need for later. We now use $\tilde{\varepsilon}$ for the analysis instead of ε, and as a result are more space efficient. We will see, that the runtime will still be linear in $\frac{1}{\varepsilon}$. Thus, we get

$$\sigma(j) = \sum_{j'=1}^{j} \log_2 \frac{1+\tilde{\varepsilon}}{p} = j\frac{a}{b} \quad j \in [m],$$

$$\sigma(b) = a \in \mathbb{N}^+.$$

We will quickly show that k_j are in fact b-periodic:

$$\begin{aligned}
\log_2 k_{j+b} &= \lceil \sigma(j+b) \rceil - \lceil \sigma(j+b-1) \rceil = \left\lceil (j+b)\log_2 \frac{1+\tilde{\varepsilon}}{p} \right\rceil - \left\lceil (j+b-1)\log_2 \frac{1+\tilde{\varepsilon}}{p} \right\rceil \\
&= \left\lceil j \log_2 \frac{1+\tilde{\varepsilon}}{p} + a \right\rceil - \left\lceil (j-1)\log_2 \frac{1+\tilde{\varepsilon}}{p} + a \right\rceil \\
&= \left\lceil j \log_2 \frac{1+\tilde{\varepsilon}}{p} \right\rceil + a - \left\lceil (j-1)\log_2 \frac{1+\tilde{\varepsilon}}{p} \right\rceil - a \\
&= \log_2 k_j \\
\implies k_{j+b} &= k_j \quad j \in [m],\ j \le m-b
\end{aligned}$$

Now that we got that out of the way, we will start with the core part of our analysis. We want to show that each task gets only visited an expected number of times in $\mathcal{O}(1/\tilde{\varepsilon})$, which in particular is independent of m. We will achieve this by showing that $q_j \in \Omega(\tilde{\varepsilon})$, as the rest follows with the expected value of a geometric distribution (see also Subsection 5.2.1 for more details).

We remember that in Subsection 5.2.1 we had $q_{j+1} = f(q_j) := 1 - (1-pq_j)^{k_j}$. We again approximate $f(x) = 1-(1-px)^k \ge 1 - e^{-pkx} =: g_{pk}(x)$ for $x \in (0,1]$ (Lemma 5.4). For our periodic case, we now consider for one period

$$\tilde{g}(x) := \left(\bigcirc_{i=1}^{b} g_{pk_{b-i+1}} \right)(x) = g_{pk_b}(g_{pk_{b-1}}(\cdots g_{pk_1}(x)\cdots)).$$

Thereby, we get

$$q_{ib+b} \ge \tilde{g}(q_{ib}) \quad i \in \mathbb{N}^+,\ ib \le m-b$$

where $q_j = \mathbb{P}(\text{FindSeedTask}(j,\cdot) \neq \perp)$ for $j \in [m]$. Then, we determine a lower bound to the fixed point x^* of $\tilde{g}$ by finding where $h \le \tilde{g}$ has $h'(x^\circ) = 1$. Using Lemma 5.8 (see below) we get

$$\tilde{g}(x) \le \tilde{a}x - \frac{x^2}{2}\tilde{a}\underbrace{\sum_{j=1}^{b}\prod_{i=j}^{b} pk_{b-i+1}}_{=:\Sigma} =: h(x), \qquad \tilde{a} := \prod_{j=1}^{b} pk_b.$$

Thus,

$$1 \overset{!}{=} h'(x^\circ) = \tilde{a} - \tilde{a}x^\circ\Sigma \iff x^\circ = \frac{1}{\Sigma}\frac{\tilde{a}-1}{\tilde{a}}.$$

With Lemma 5.2 and Corollary 5.3 from earlier we get

$$\frac{1}{\Sigma}\frac{\tilde{a}-1}{\tilde{a}} = x^\circ \le x^* \le q_{ib} \quad i \in \mathbb{N}^+,\ ib \le m.$$

We now want to reformulate this approximation for q_{ib} in terms of $\tilde{\varepsilon}$. Taking a closer look at $\tilde{a}, \Sigma$, we find

$$\prod_{j'=1}^{j} k_{j'} = \exp2\left(\log_2 \prod_{j'=1}^{j} k_{j'}\right) = \exp2\left(\sum_{j'=1}^{j} \log_2 k_{j'}\right) = \exp2(\lceil \sigma(j) \rceil) \quad j \in [m]$$

$$\implies \prod_{j=1}^{b} k_b = \exp2(\lceil \sigma(b) \rceil) \geq \exp2\left(\sum_{j=1}^{b} \log_2 \frac{1+\tilde{\varepsilon}}{q}\right) = \left(\frac{1+\tilde{\varepsilon}}{p}\right)^b$$

$$\implies \tilde{a} = \prod_{j=1}^{b} pk_b \geq (1+\tilde{\varepsilon})^b$$

with $\exp2(x) = 2^x,\ x \in \mathbb{R}$ and

$$\begin{aligned}
\Sigma &= \sum_{j=1}^{b} \prod_{i=j}^{b} pk_{b-i+1} = \sum_{j=1}^{b} p^{b-j+1} \prod_{i=1}^{b-j+1} k_i = \sum_{j=1}^{b} p^{b-j+1} \exp2(\lceil \sigma(b-j+1) \rceil) \\
&\leq \sum_{j=1}^{b} p^{b-j+1} \exp2(\sigma(b-j+1)+1) = \sum_{j=1}^{b} p^{b-j+1} 2 \left(\frac{1+\tilde{\varepsilon}}{p}\right)^{b-j+1} = 2\sum_{j=1}^{b} (1+\tilde{\varepsilon})^{b-j+1} \\
&\leq 2b \cdot (1+\tilde{\varepsilon})^b.
\end{aligned}$$

In the end, we get

$$q_{ib} \geq x^\circ \geq \frac{(1+\tilde{\varepsilon})^b - 1}{2b(1+\tilde{\varepsilon})^{2b}} \geq \frac{\tilde{\varepsilon} b}{2b(1+\tilde{\varepsilon})^{2b}} = \frac{\tilde{\varepsilon}}{2(1+\tilde{\varepsilon})^{2b}} > 0 \text{ for } \tilde{\varepsilon} > 0,\ i \in \mathbb{N}^+,\ ib \leq m.$$

Now that we have an approximation for every bth task success probability—for each task after a period—we now need to look inside a period and gain an approximation for $q_{ib-1}, \ldots, q_{ib-b+1}$. Here, we expect success probabilities to fall a bit for some tasks, as some of them get fewer bits to work with. For these tasks, we revisit the recursion condition on $q_{j+1}\ j \in [m-1]$ and get

$$\begin{aligned}
q_{j+1} &= 1 - (q - pq_j)^{k_j} \geq 1 - \exp(-pk_j \cdot q_j) \\
&\iff \exp(-pk_j \cdot q_j) \geq 1 - q_{j+1} \iff -pk_j \cdot q_j \geq \ln(1-q_{j+1}) \\
&\iff q_j \geq \frac{-\ln(1-q_{j+1})}{pk_i} \overset{(*)}{\geq} \frac{q_{j+1}}{pk_1}
\end{aligned}$$

with

$$(*) \quad -\ln(1-x) \geq x \iff 1 - x \leq e^{-x} \overset{y=-x}{\iff} 1 + y \leq e^y \quad x \in \mathbb{R}.$$

Recursively applying this we get

$$\begin{aligned}
q_{ib-s} &\geq \left(\prod_{j=ib-s}^{ib-1} pk_j\right)^{-1} q_{ib} = p^{-s}\frac{\exp2(\lceil\sigma(ib-s-1)\rceil)}{\exp2(\lceil\sigma(ib-1)\rceil)}q_{ib} \\
&= p^{-s}\exp2\left(\left\lceil(ib-s-1)\frac{a}{b}\right\rceil - \left\lceil(ib-1)\frac{a}{b}\right\rceil\right)q_{ib} \\
&\geq p^{-s}\exp2\left((ib-s-1)\frac{a}{b} - (ib-1)\frac{a}{b} - 1\right)q_{ib} \\
&= p^{-s}\exp2\left(-s\frac{a}{b} - 1\right)q_{ib} = \frac{q_{ib}}{p^s\left(\frac{1+\tilde{\varepsilon}}{p}\right)^s \cdot 2} = \frac{q_{ib}}{2(1+\tilde{\varepsilon})^s} \qquad s \in [b-1] \\
&\geq \frac{q_{ib}}{2(1+\tilde{\varepsilon})^b}.
\end{aligned}$$

Combining this result with the bound for q_{ib} and reformulating in terms of ε, we get

$$\begin{aligned}
q_j &\geq \frac{\tilde{\varepsilon}}{4\cdot(1+\tilde{\varepsilon})^{3b}} \geq \frac{\sqrt{1+\varepsilon}-1}{4\cdot((1+\varepsilon)\cdot 2^{-\xi})^{3b}} \geq \frac{\sqrt{1+\varepsilon}-1}{4\cdot((1+\varepsilon)\cdot 2^{0})^{3b}} \geq \frac{\sqrt{1+\varepsilon}-1}{4\cdot(1+\varepsilon)^{3b}} \\
&\geq \frac{\sqrt{1+\varepsilon}-1}{4\cdot(1+\varepsilon)^{3\left(\frac{2}{\log_2(1+\varepsilon)}+1\right)}} \geq \frac{\sqrt{1+\varepsilon}-1}{4\cdot(1+\varepsilon)^{3\frac{2}{\log_2(1+\varepsilon)}}(1+\varepsilon)^3} \geq \frac{\sqrt{1+\varepsilon}-1}{4\cdot 2^{6\frac{\log_2(1+\varepsilon)}{\log_2(1+\varepsilon)}}(1+\varepsilon)^3} \\
&\geq \frac{\sqrt{1+\varepsilon}-1}{2^8\cdot(1+\varepsilon)^3} =: q^\dagger \\
&\overset{\varepsilon\leq 1}{\geq} 2^{-11.5}\cdot(1+\varepsilon-1) = 2^{-11.5}\varepsilon\begin{cases}>0\\ \in\Omega(\varepsilon)\end{cases} \qquad j\in[m]
\end{aligned}$$

as $\sqrt{x} = \frac{x}{\sqrt{x}} \geq \frac{x}{\sqrt{2}}$ for $x \in [0,2]$. We already plugged in $\varepsilon \geq \sqrt{1+\varepsilon}-1$, which now tells us that choosing the a bit smaller $\tilde{\varepsilon}$ which allowed for b-periodic bounds k_j is easily possible and still get $q_j \in \Omega(\varepsilon)$. Inserting this in the work equation from Subsection 5.2.1 we finally get

$$\mathbb{E}[\mathcal{R}_{\text{SRS}}] \leq \frac{1}{q^\dagger}\sum_{j\in[m]}\frac{W_j}{p} \in \mathcal{O}\left(\frac{1}{\varepsilon}\sum_{j\in[m]}\frac{W_j}{p}\right) = \mathcal{O}\left(\frac{1}{\omega}\sum_{j\in[m]}\frac{W_j}{p}\right).$$

To sum up, we have shown that for any targeted $\omega = \log_2(1+\varepsilon)$ and non-varying $p = p_j$ we can choose a smaller $\tilde{\varepsilon}$ which leads to periodic bounds k_j given by (5.3). In this periodic case, we get success probabilities $q_j \in \Omega(\tilde{\varepsilon}) \subseteq \Omega(\varepsilon)$. This should suffice to convince one that also directly using ε that leads to non-periodic (or such with way larger period) would lead to the same work requirement. Thus, we will not enforce using ε that leads to periodic k_j in practice.

We still need to prove one lemma used when approximating the recursion function for one period of q_{ib}.

Lemma 5.8: *Let $g_a = 1 - e^{-ax}$ for $a > 0$, $x \in (0,1]$, $a_i \in (0,4]$ for $i \in [b]$, $b \in \mathbb{N}^+$. Let $\tilde{a} = \prod_{i=1}^{b} a_i$. It holds*

$$\left(\bigcirc_{i=1}^{b} g_{a_i}\right)(x) = g_{a_1}(g_{a_2}(\cdots g_{a_b}(x)\cdots)) \geq \tilde{a}x - \frac{x^2}{2}\tilde{a}\sum_{j=1}^{b}\prod_{i=j}^{b} a_i \quad x \in (0,1]. \tag{5.4}$$

Proof. Let $g_a, a_i\ i \in \mathbb{N}^+,\ x$ be as required. Proof by induction over $b \in \mathbb{N}^+$.

Let $b = 1, a := a_1$. It holds (Induction Base IB):

$$\begin{aligned}
g_a(x) &= -\sum_{i=1}^{\infty}\frac{(-ax)^i}{i!} = \sum_{i=1}^{2}\frac{-(-ax)^i}{i!} + \sum_{i=3}^{\infty}\frac{-(-ax)^i}{i!} \\
&= \sum_{i=1}^{2} -\frac{(-ax)^i}{i!} + \sum_{j=2}^{\infty}\left(\frac{-(-ax)^{2j-1}}{(2j-1)!} + -\frac{(-ax)^{2j}}{(2j)!}\right) \\
&= \sum_{i=1}^{2} -\frac{(-ax)^i}{i!} + \sum_{j=2}^{\infty}\frac{-(-ax)^{2j-1}}{(2j-1)!}\left(1 - \underbrace{\frac{ax}{2j}}_{\leq 1}\right) \\
&\geq ax - \frac{x^2}{2}a^2 = a\left(x - \frac{x^2}{2}a\right) = ax\left(1 - \frac{ax}{2}\right)
\end{aligned}$$

Let (5.4) hold for an arbitrary, but fixed $b \in \mathbb{N}^+$ (Induction Hypothesis IH). In the following, $\cdot$ shall bind weaker than $\prod$. It holds:

$$\begin{aligned}
\left(\bigcirc_{i=1}^{b+1} g_{a_i}\right)(x) &= \left(\bigcirc_{i=1}^{b} g_{a_i}\right)(g_{a_{b+1}}(x)) \\
&\overset{\text{IH}}{\geq} \prod_{i=1}^{b} a_i \cdot g_{a_{b+1}}(x) - \frac{g_{a_{b+1}}(x)^2}{2}\prod_{i=1}^{b} a_i \cdot \sum_{j=1}^{b}\prod_{i=j}^{b} a_i \\
&\overset{\text{IB}}{\geq} \prod_{i=1}^{b} a_i \cdot a_{b+1}\left(x - \frac{x^2}{2}a_{b+1}\right) - \frac{1}{2}\left(a_{b+1}x\left(1 - \frac{xa_{b+1}}{2}\right)\right)^2 \prod_{i=1}^{b} a_i \cdot \sum_{j=1}^{b}\prod_{i=j}^{b} a_i \\
&= \prod_{i=1}^{b+1} a_i \cdot \left(x - \frac{x^2}{2}a_{b+1}\right) - \frac{x^2}{2}a_{b+1}^2 \underbrace{\left(1 - \frac{xa_{b+1}}{2}\right)^2}_{\leq 1}\prod_{i=1}^{b} a_i \cdot \sum_{j=1}^{b}\prod_{i=j}^{b} a_i \\
&\geq x\prod_{i=1}^{b+1} a_i - \frac{x^2}{2}a_{b+1}\prod_{i=1}^{b+1} a_i - \frac{x^2}{2}a_{b+1}\prod_{i=1}^{b+1} a_i \cdot \sum_{j=1}^{b}\prod_{i=j}^{b} a_i \\
&= x\prod_{i=1}^{b+1} a_i - \frac{x^2}{2}\prod_{i=1}^{b+1} a_i \cdot \left(a_{b+1} + \sum_{j=1}^{b} a_{b+1}\prod_{i=j}^{b} a_i\right) \\
&= x\prod_{i=1}^{b+1} a_i - \frac{x^2}{2}\prod_{i=1}^{b+1} a_i \cdot \sum_{j=1}^{b+1}\prod_{i=j}^{b+1} a_i.
\end{aligned}$$

□

Varying p_j

We now want to briefly think about how this result can be generalized for varying p_j. When p_j vary for $j \in [m]$, k_j can in general not be made periodic by altering ε slightly. When p_j are blockwise constant, that is,

$$p_1 = \cdots = p_{s_1},\ p_{s_1+1} = \cdots = p_{s_1+s_2},\ p_{s_1+s_2+1} = \cdots\ \cdots = p_{s_1+s_2+\cdots s_r}$$

for block sizes $s_i \in \mathbb{N}^+,\ i \in [r],\ \sum_{i\in[r]} s_i = m$, we could just calculate the bounds k_j for each block separately and thus would only waste one bit per block due to rounding. We will have such a case later when applying SRS to MPHF construction. Then, however, we will just assume our result holds even when calculating bounds for all tasks together and not waste any precious bits.

5.2.6. Summary

We now want to give a quick summary of all the cases we analyzed SRS for. We started with the most simple case of constant success probabilities p and retry bounds k for all m tasks. We showed that, $pk \geq 1 + \varepsilon > 1$ is necessary for SRS to require work in $\mathcal{O}\left(\frac{mW}{\omega p}\right)$ where W is the work required to check one seed and $\omega = \log_2(1+\varepsilon)$. Then, we showed that the same analysis can be applied when p_j, k_j vary, as long as $p_j k_j \geq 1+\varepsilon$ holds for each of them. Afterward, we discussed how to choose k_j such that we can store and retrieve seeds i_j easily. To begin, we choose to just use the smallest power of two that fulfills the aforementioned requirement. This makes us, however, waste up to one bit per task, while still being a bit better than individually testing and encoding seeds for small p_j. We then discussed using only the last w bits of a seed (windowed case) for efficient operations on them, and argued why that should not be a problem in practice. At last, we pushed space usage even further, arbitrarily close (per task) to searching for one seed that is successful for all tasks (UNI). We achieved this by cleverly rounding the bounds k_j up and down. For this, however, we needed to reanalyze the work requirement and considered periodic k_j. Next, we will use these results to analyze SRS applied to MPHF construction—SRS-RecSplit.

5.3. Analysis of SRS-RecSplit

Now that we analyzed SRS extensively, we can use the results to easily analyze the runtime and space usage of SRS-RecSplit, as described in Section 4.4. We first analyze SRS-RecSplit for power-of-two inputs. Then, we show why varying the overhead parameter ω can be helpful for practical performance. Finally, we provide some details on what changes when we also allow non-power-of-two inputs.

5.3.1. Power-of-Two Input Size

For the simpler case of power-of-two input sizes, we will first analyze construction time, then we will look at space usage.

Theorem 5.9: *The power-of-two construction of SRS-RecSplit for $X \in U$, $|X| = n$ and overhead parameter $\omega \in (0, 1]$, as described in Subsection 4.4.1, has runtime*

$$\mathbb{E}\left[\mathcal{R}_{\text{SRS-RecSplit}}\right] \in \mathcal{O}\left(\frac{n^{3/2} \log n}{\omega}\right).$$

Proof. From Section 4.3 we know SRS has runtime $\mathcal{O}\left(\frac{1}{\omega} \sum_{j \in [m]} \frac{W_j}{p_j}\right)$ for m tasks, each with success probability p_j and work requirement W_j to test a seed. We now need to figure out the value of these parameters to get a construction time for SRS-RecSplit.

We will start with the number of tasks m. When an input set of *keys* $X \subseteq U$ from some universe U with a power of two size $n := |X| = 2^r$, $r \in \mathbb{N}^+$ is given, the splitting tree requires r layers $i \in \{r, \dots, 1\}$, each containing 2^{r-i} splitting nodes of size 2^i. This results in

$$m = \sum_{i=1}^{r} 2^{r-i} = \sum_{i=0}^{r-1} 2^i = 2^r - 1 = n - 1$$

tasks.

We now continue to determine the success probabilities p_j. In our case, success means to successfully find a seed, such that it splits the task's keys into two parts of equal size. As they are the same for each task in one layer of the splitting tree, we determine p_i on layer $i \in r, \dots, 1$. Let there be seeded hash functions $h_s : U \to \{0, 1\}$ for seeds $s \in \mathbb{N}_0$ with $\mathbb{P}(h_s(x) = 1) = \frac{1}{2}$ independently for all $x \in U$, $s \in \mathbb{N}_0$. Hash function h_s *splits* an input $X_{\tilde{n}} \subseteq U$ of size $|X_{\tilde{n}}| = \tilde{n}$ *evenly*, iff exactly half of the keys in $X_{\tilde{n}}$ get hashed to 0:

$$\#0_{\tilde{n}} := \left|h_s^{-1}(\{0\}) \cap X_{\tilde{n}}\right| = \frac{\tilde{n}}{2}.$$

As $\#0_{\tilde{n}} \sim \text{Bin}(\tilde{n}, \frac{1}{2})$, it holds (using Stirling's approximation in the second line)

$$\begin{aligned}\mathbb{P}\left(\#0_{\tilde{n}} = \frac{\tilde{n}}{2}\right) &= \binom{\tilde{n}}{\frac{\tilde{n}}{2}} \left(\frac{1}{2}\right)^{\tilde{n}/2} \left(1 - \frac{1}{2}\right)^{\tilde{n}/2} = 2^{-\tilde{n}} \binom{\tilde{n}}{\frac{\tilde{n}}{2}} = 2^{-\tilde{n}} \frac{\tilde{n}!}{(\tilde{n}/2)!^2} \\ &\in \Omega\left(2^{-\tilde{n}} \frac{\sqrt{\tilde{n}} \tilde{n}^{\tilde{n}} e^{-\tilde{n}}}{(\sqrt{\tilde{n}/2} (\tilde{n}/2)^{\tilde{n}/2} e^{-\tilde{n}/2})^2}\right) = \Omega\left(\frac{1}{\sqrt{\tilde{n}}}\right).\end{aligned}$$

Therefore, we get the probability of finding a splitting in layer i with $\tilde{n} = 2^i$ keys of

$$p_i = \mathbb{P}\left(\#0_{2^i} = \frac{2^i}{2}\right) = \frac{2^i!}{2^{2^i}(2^{i-1}!)^2} \in \Omega\left(\frac{1}{\sqrt{2^i}}\right) = \Omega\left(2^{-i/2}\right).$$

Next up, we determine W_i which is again the same on layer i. Testing whether a seed splits an input of size $\tilde{n}$ equally, requires evaluating h_s for each element and counting the results, resulting in work $\mathcal{O}(\tilde{n})$. Thus, on layer i we have

$$W_i \in \mathcal{O}\left(2^i\right).$$

This, however, does not cover all the work required. To keep track of which key belongs to which task, we partition them after finding a successful seed. This partitioning requires work in $\mathcal{O}(\tilde{n})$ for $\tilde{n}$ keys. We therefore can just include this work in W_i and no longer have to think about it. More details on the exact partitioning scheme can be found in Chapter 6.

We now plug these values into SRS' work equation. By summing over all layers of the splitting tree and therein over every task we get

$$\mathbb{E}\left[\mathcal{R}_{\text{SRS-RecSplit}}\right] \in \mathcal{O}\left(\frac{1}{\omega}\sum_{i=1}^{r}\sum_{j=1}^{2^{r-i}}\left(\frac{W_i}{p_i}\right)\right) = \mathcal{O}\left(\frac{1}{\omega}\sum_{i=1}^{r}\frac{2^{r-i}2^i}{\sqrt{2^{-i}}}\right) = \mathcal{O}\left(\frac{1}{\omega}\sum_{i=1}^{r}\frac{n\sqrt{2^i}}{\omega}\right) = \mathcal{O}\left(\frac{n^{3/2}\log n}{\omega}\right).$$

□

Now we consider space usage.

Theorem 5.10: *The power-of-two construction of SRS-RecSplit for $X \in U$, $|X| = n$ and overhead parameter $\omega \in (0, 1]$, as described in Subsection 4.4.1, has space usage*

$$\mathbb{E}\left[\mathcal{S}_{\textit{SRS-RecSplit}}\right] \in \mathcal{O}\left(\log_2\frac{1}{\omega}\right) + (n-1)\cdot\omega + \log_2\frac{n^n}{n!}.$$

Proof. According to Section 4.3, we get **space usage**

$$\begin{aligned}\mathbb{E}\left[\mathcal{S}_{\text{SRS-RecSplit}}\right] &\in \mathcal{O}\left(\log_2\frac{1}{\omega}\right) + m\omega - \sum_{i=1}^{r}\sum_{j=1}^{2^{r-i}}(\log_2 p_i) = \mathcal{O}\left(\log_2\frac{1}{\omega}\right) + (n-1)\cdot\omega - \log_2\prod_{i=1}^{r}p_i^{2^{r-i}} \\ &= \mathcal{O}\left(\log_2\frac{1}{\omega}\right) + (n-1)\cdot\omega + \log_2\frac{n^n}{n!}\end{aligned}$$

with $\frac{n!}{n^n} = \prod_{i=1}^{r} p_i^{2^{r-i}}$ being the probability to find an MPHF for n elements, given by recursive application of the invariance property presented in [EGV19]. □

Considering the space usage per task, we can see that it is arbitrarily close to the lower bound when choosing small ω. For that, we remember the space usage of the naive MPHF implementation shown in Subsection 2.5.2. Together with the derivation of the lower bound (Subsection 2.5.1) we argued, that a space usage of $\log_2\frac{n^n}{n!}$ bits is optimal. When we use small ω, the overhead dependent on n decreases. For large n, the per task overhead caused by $\mathcal{O}\left(\log_2\frac{1}{\omega}\right)$ gets arbitrary small as well.

5.3.2. Adaptive Overheads ω

There is one more trick we can apply. In practice, varying ω across layers in the splitting tree improves construction time. Although we do not get a better asymptotic construction time, we will still provide a short analysis to understand why this is the case. The idea behind this modification is that we want to negate the smaller probability of finding an even split on the upper layers with larger task sizes. As we chain together multiple blocks of SRS tasks with constant success probabilities p as mentioned at the end of Subsection 5.2.5, nothing prevents us from choosing different overhead parameters ω in each block—each layer.

If we would set $\omega_i = \sqrt{2^i}\omega$ on layer i for some $\omega \in (0, 1]$, this would cancel out $p_i \in \Omega\left(2^{-i/2}\right)$. If we would continue the construction time analysis with these values for ω we would reach construction time in $\mathcal{O}\left(\frac{n \log n}{\omega}\right)$ (with even $\mathcal{O}\left(\frac{n}{\omega}\right)$ possible for a different choice of ω_i). We would then show, that the total overhead over all tasks is still linear in ω (as there are many more tasks with small ω), on which also our runtime depends. Sadly, there is a catch. In our analysis for SRS, we restricted ω to $(0, 1]$. If we increase n here, we also get more layers and thus larger i making ω_i grow without bounds. Restricting ω to $(0, 1]$ for SRS' analysis is necessary because when we make ω much larger, we do not get a linear work improvement beyond some point. If we have already enough tries to find a successful seed—the bounds k_j being large enough—providing more tries just does not make that much of a difference. Thus, if we do not bound ω from above, we no longer could state $\mathbb{E}[\mathcal{R}_{\text{SRS}}] \in \mathcal{O}\left(\frac{mW}{p\omega}\right)$ as then big O notation would also need to apply for $\omega \to \infty$.

In practice, as long as $\omega_i = \sqrt{2^i}\omega \leq \sqrt{n}\omega \overset{!}{\leq} 1$ we still can use these adaptive ω_i. ω would then no longer be the actual overhead per key, but a fraction of that:

$$\sum_{i=1}^{r} 2^{r-i}\omega_i = \sum_{i=1}^{r} 2^{r-i}2^{i/2}\omega = n\omega \sum_{i=1}^{r} \left(\frac{1}{\sqrt{2}}\right)^i \leq n\omega \sum_{i=1}^{\infty} \left(\frac{1}{\sqrt{2}}\right)^i = n\omega \frac{1}{1 - 1/\sqrt{2}} \approx 3.41n\omega.$$

This allows us to choose larger ω that lead to the same overhead but better construction time. We use such adaptive overheads in our implementation, see Chapter 6. Further analyzing possible advantages that could be gained by adaptive overheads ω remains future work.

5.3.3. Non-Power-of-Two Input Size

We will now consider SRS-RecSplit for non-power-of-two input sizes. As described in Subsection 4.4.2 we will split a set of non-power-of-two many keys into the next smallest power of two and the rest. In the following, we will calculate the number of tasks required and what the new splitting probabilities are.

Number of Tasks

First, we need to consider how many SRS tasks are necessary for an input $X \subseteq U$ with size $|X| = n$: We now have $r := \lceil \log_2 n \rceil$ layers $i \in \{r, \ldots, 1\}$ with each $\left\lfloor \frac{n}{2^i} \right\rfloor$ *ordinary* tasks of size 2^i, and possibly one remaining *uneven* task of size $n - 2^i \left\lfloor \frac{n}{2^i} \right\rfloor$ iff its size would be $> 2^{i-1}$. This

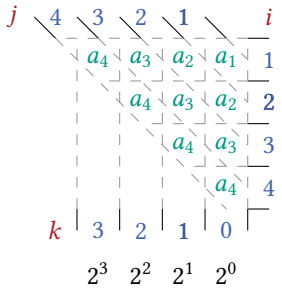


Figure 5.6.: This figure motivates the re-indexing to simplify the number of SRS tasks necessary for SRS-RecSplit given a non-power-of-two input sizes. In each row i we see the binary representation of n shifted to the right by i bits. Each column represents the value of each bit (2^k). The diagonal index j determines which bit of n we sum over (a_j). Given a sum over i and j, we want to sum over j, k.

is the case iff for one layer, there remains some of the input that is not covered by ordinary tasks, but is more than a single ordinary task on the next layer can cover and thus needs to be split beforehand. Thus, we get for the total number of tasks

$$m = \sum_{i=1}^{r} \left(\left\lfloor \frac{n}{2^i} \right\rfloor + \left[n - 2^i \left\lfloor \frac{n}{2^i} \right\rfloor > 2^{i-1} \right] \right) \tag{5.5}$$

where $[P] := \begin{cases} 1 & P \\ 0 & \neg P \end{cases}$ for some statement P is the Iverson bracket.

We now want to simplify this expression. Looking at the binary representation of $n = \sum_{j=0}^{r} 2^j a_j,\ a_j \in \{0,1\}$, we can rewrite the first part of the sum as

$$\begin{aligned} \sum_{i=1}^{r} \left\lfloor \frac{n}{2^i} \right\rfloor &= \sum_{i=1}^{r} \sum_{j=i}^{r} 2^{j-i} a_j = \sum_{1 \le i \le j \le r} 2^{j-i} a_j \\ &= \sum_{0 \le j \le r,\ 0 \le k \le j-1} 2^k a_j = \sum_{j=0}^{r} a_j \sum_{k=0}^{j-1} 2^k = \sum_{j=0}^{r} a_j (2^j - 1) = n - \sum_{j=0}^{r} a_j \end{aligned}$$

as $\Phi : \{(i,j) \in \mathbb{N}_0^2 \mid 1 \le i \le j \le r\} \to \{(j,k) \in \mathbb{N}_0^2 \mid 0 \le j \le r, 0 \le k \le j-1\}, (i,j) \mapsto (j, k := j - i)$ is bijective. Figure 5.6 provides an intuition behind this re-indexing.

For the second part of the sum of (5.5), considering again the binary representation of n, the bracket equals 1 for layer i iff the bits $a_{i-1} = a_{i'} = 1$ for some $i' < i - 1$ are set. If n is not a power of two, thus $a_r = 0$ ($r = \lceil \log_2 n \rceil$), we get 1 in the sum for every i with $a_{i-1} = 1,\ i \in \{r, \dots, 1\}$ except the last (least significant) set bit, thus

$$\sum_{i=1}^{r} \left[n - 2^i \left\lfloor \frac{n}{2^i} \right\rfloor > 2^{i-1} \right] = \sum_{i=0}^{r-1} a_i - 1.$$

As a result, for non-power-of-two n, we also get

$$m = n - \sum_{j=0}^{r} a_j + \sum_{i=0}^{r-1} a_i - 1 = n - 1 - a_r = n - 1.$$

Uneven tasks

Next, we will consider the probability for a seed to perform an uneven split for tasks that contain a non-power-of-two number of keys. These tasks split their keys into the next smaller power of two and the rest.

More precisely, for task size $\tilde{n}$ with $\log_2 \tilde{n} \notin \mathbb{N}_0$, we need to split into parts of size $2^{\lfloor \log_2 \tilde{n} \rfloor}$ and rest. For that, we need a hash function $h_s : U \to \{0,1\}$ for seeds $s \in \mathbb{N}_0$ with $\mathbb{P}(h_s(x) = 0) = \frac{2^{\lfloor \log_2 \tilde{n} \rfloor}}{\tilde{n}} =: q$ independently for all $x \in U,\ s \in \mathbb{N}_0$. For a given input set $X_{\tilde{n}} \subseteq U$ with size $|X_{\tilde{n}}| = \tilde{n}$ we get for the number $\#0_{\tilde{n}} \in \mathbb{N}_0$ of keys hashed to 0 by h_s:

$$\begin{aligned}
\mathbb{P}\left(\#0_{\tilde{n}} = \tilde{n}q\right) &= \binom{\tilde{n}}{\tilde{n}q} q^{\tilde{n}q}(1-q)^{\tilde{n}(1-q)} = \frac{\tilde{n}!}{(\tilde{n}(1-q))!(\tilde{n}q)!} q^{\tilde{n}q}(1-q)^{\tilde{n}(1-q)} \\
&\sim \frac{\sqrt{2\pi\tilde{n}}\tilde{n}^{\tilde{n}}/e^{\tilde{n}} \cdot q^{\tilde{n}q}(1-q)^{\tilde{n}(1-q)}}{\sqrt{\tilde{n}(1-q)}(\tilde{n}(1-q))^{\tilde{n}(1-q)}/e^{\tilde{n}(1-q)} \cdot \sqrt{2\pi\tilde{n}q}(\tilde{n}q)^{\tilde{n}q}/e^{\tilde{n}q}} = \frac{1}{\sqrt{2\pi\tilde{n}q(1-q)}} \\
&\geq \frac{2}{\sqrt{2\pi\tilde{n} \cdot (1-1/\tilde{n}) \cdot 1/2}} = \frac{1}{\sqrt{\pi(\tilde{n}-1)}} \in \Omega\left(\frac{1}{\sqrt{\tilde{n}}}\right)
\end{aligned}$$

For the second line we applied Stirling approximation (see Section 2.2). Again, $a_n \sim b_n$ means the quotient $\frac{a_n}{b_n} \to 1$ $(n \to \infty)$. For the third line we used $q \in \left[\frac{1}{2}, 1 - \frac{1}{\tilde{n}}\right]$ which we get because we split of at most half of the keys and at least $1/\tilde{n}$ when $\tilde{n}$ is just one larger than a power of two.

Splitting probabilities being in $\Omega\left(\tilde{n}^{-1/2}\right)$ as they are in the power-of-tow case is important for the construction time. We can apply a similar analysis as in Subsection 5.3.1 to get construction time in $\mathcal{O}\left(\frac{n^{3/2}}{\omega}\right)$. Furthermore, we can apply the same analysis for space usage to get $\mathcal{O}\left(\log_2 \frac{1}{\omega}\right) + (n-1) \cdot \omega + \log_2 \frac{n^n}{n!}$ bits.

6. Implementation

To validate the theoretical findings, we implemented SRS-RecSplit in Rust [Zie24]. The following presents some noteworthy considerations and implementation details. First, we look at how the splittings in each SRS task are performed in practice and how and which hash functions are used.

6.1. Hashing and Splitting

In SRS-RecSplit, in each SRS task, the input keys get split into two parts using a hash function. This section explains how these splittings are performed in practice.

Master Hash Codes. First, to not have to hash larger keys multiple times, we create a 64-bit *master hash code* for each key, as also done by other implementations [EGV19]. This master hash code is then used instead of the keys in every further step. We will call this set of master hash codes $W = \{0,1\}^{64}$. We use WyHash2 to generate these master hash codes [Yi24 | Tho22].

Splitting Hash Functions. To perform splittings, we need a seeded hash function $h_s^P : W \to \{0,1\}$ with different probabilities $P := \mathbb{P}(h_s(x) = 0)$ independently for all $x \in U,\ s \in \mathbb{N}_0$ for even splits—but also for uneven splits for non-power-of-two input size cases. Hashes under this hash function decide whether a key belongs to the left or right child.

For even splits where each child has the same size—requiring $P = 1/2$—we construct $h_s^{1/2}$ by considering only one bit of a uniform hash function $\bar{h} : W \to W$, and XORing ($\oplus$) the seed s to the master hash code: $h_s^{1/2}(x) = \bar{h}(s \oplus x)\ \&\ 1$. $\&$ marks the bitwise AND operation.

For uneven splits with $P = \frac{2^{\lfloor \log_2 n \rfloor}}{n}$, we use the same scheme, but using fixed point inversion [EGV19] instead of bit masking:

$$h_s^P(x) = \begin{cases} 0 & \left((\bar{h}(s \oplus x) \cdot n) \gg \log_2 |W|\right) < 2^{\lfloor \log_2 n \rfloor} \\ 1 & \text{otherwise} \end{cases}.$$

$\gg$ is the bit-shift operator.

Hash Algorithm. Now, a choice of $\bar{h}$—independent of the choice made for the master hash codes—still remains. As SRS-RecSplit spends most of its time on hashing, using a fast hash function that is crucial. Wyhash2 [Yi24 | Tho22] has proven to be fastest under all tested,

Table 6.1.: SRS-RecSplit construction time comparison using various hashing algorithms with $n = 2^{10}$, $\omega = 0.1$. Fxhash does not terminate in any reasonable amount of time.

algorithm	avg.	std. div.
xxhash [Dou24]	3.64 ms	335 µs
std::hash [Com]	3.13 ms	257 µs
ahash [Kai24]	1.14 ms	41.1 µs
wyhash2 [Tho22]	1.13 ms	34.2 µs
fxhash [Chr24]	-	-

while still being of good enough quality for SRS to work (see Table 6.1). Fxhash produces too similar hashes (the last bit is the same) given a seed just one larger, thus leading to SRS not terminating in any reasonable amount of time.

Calculation of splitting probabilities. To figure out the number of bits $\log_2 k_j$ provided for one task j, it is necessary to know the probability p_j of a splitting in the splitting tree for any seed to be successful, see Section 4.3.

For even splittings with the splitting probability p_i determined in Subsection 4.4.1, we can calculate its logarithm more efficiently with

$$\log_2 p_i = \log_2 \frac{2^i!}{2^{2^i}(2^{i-1}!)^2} = \log_2 \prod_{t=1}^{2^{i-1}} \frac{2^{t-1}+t}{4t} = \sum_{t=1}^{2^{i-1}} \log_2 \left(\frac{2^i}{8t} + \frac{1}{4}\right)$$

for every task j on layer $i \in [\log_2 n]$ for power-of-two input sizes n. These values can be precomputed and stored to cover any reasonable power-of-two input size.

For non-power-of-two input sizes n, some uneven splittings—splitting a non-power-of-two input into the next smaller power-of-two and rest—are also necessary. The probabilities of finding a successful splitting here, see Subsection 4.4.2, need to be computed on the fly as storing them for any possible input size is unfeasible. For efficient calculation, we use Stirling's approximation and simplify it to get

$$\log_2 p(n) \approx -\frac{\log_2\big(2\pi nq(1-q)\big)}{2}$$

where $q = 2^{\lfloor \log_2 n \rfloor}/n$.

Partitioning. After a successful split is found in one SRS task, its keys need to be partitioned by their hash value of the current task so that they will be correctly assigned to the child tasks of the splitting tree. For that, we apply Hoare partitioning [Hoa62] as it is done in textbook Quicksort. We iterate from both ends of the key array until we find one key on each side that belongs to the opposite side—as dictated by its hash value—and then swap them. This algorithm requires runtime $\mathcal{O}(M)$ for M keys.

6.2. Other

There are a few other implementation details to note.

Floating Point Operations. To calculate how many bits to use for each task, SRS needs to keep track of fractional bits, noted as σ in Subsection 5.2.5. For that, it is necessary to work with non-integer values. Using 32-bit floating point numbers led to numerical errors that caused these calculations to lead to different results in the construction and hashing of keys afterward. Using 64-bit floating point numbers deferred this issue, for all tested input sizes, however, using fixed point arithmetic would be possible. For a more stable algorithm, the same equations (i.e., not replacing products for sums) should be used in the future.

Window Size. To allow for $\mathcal{O}(1)$ hashing, in practice only the last w bits of seeds are used to seed the hash functions. In Subsection 5.2.4 we argued that this should not be a problem. The implementation uses $w = 64$ bits, which allows for sufficiently many possible splittings that cannot be exhausted in any reasonable amount of time.

Adaptive Overheads. In our implementation, we use adaptive overheads $\omega_i = \frac{\omega}{3.4}\sqrt{2^i}$ on each layer, as presented in Subsection 5.3.2. The rescaling factor of 3.4 is used to in the end get approximately ω bits per key overhead when providing overhead parameter ω. We also limit $k_j \leq 64$ as larger values can occur in the first layers for large ω and n because of these adaptive overheads. Because of adaptive overheads, this prediction of space overhead is not perfect and worse for small input sizes. In our experiments, we always show the real overhead.

Layer Calculation. To calculate the layer $i \in \lceil \log_2 n \rceil, \ldots, 1$ in the splitting tree given a task index $j \in [n-1]$, we use the following formula: $i = \left\lceil \log_2 \left\lceil \frac{n}{j} \right\rceil \right\rceil$. To calculate a zero-based index of the task inside the current layer, we use $c = j - \left\lfloor \frac{n}{2^i} \right\rfloor + [\mathrm{trailingzeros}(n) \geq i] - 1$. $[P] \in \{0, 1\}$ is again the Iverson bracket with $[P] = 1 \iff P$ for a statement P. $\mathrm{trailingzeros}(n)$ is the number of trailing zeros (starting at the least significant bit) in the binary representation of n.

Iterative Implementation. In practice, implementing SRS iteratively instead of with recursive function calls as shown in Algorithm 4.1 in Section 4.3 is beneficial for several reasons: First, a recursive implementation would lead to stack overflows on default stack sizes for thousands of tasks, as required for SRS-RecSplit. Furthermore, the iterative implementation needs fewer function calls. In preliminary experiments, we got a 30% shorter construction time. However, the iterative implementation still uses a custom stack data structure for storing intermediate indices.

7. Experiments

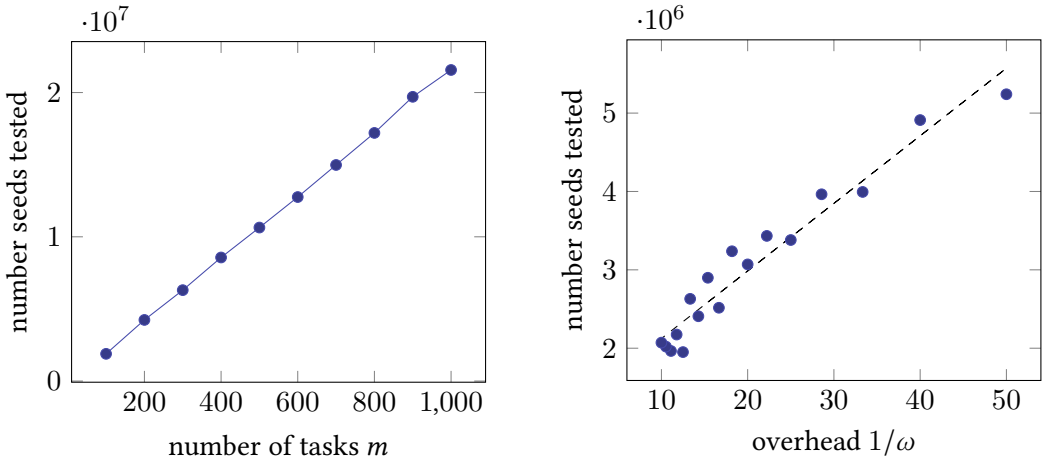
This chapter evaluates the practical usability of SRS-RecSplit. First, we compare some of our predictions about SRS made in Section 5.2 to real-world results for a problem simpler than constructing MPHFs. Afterward, we test SRS-RecSplit's runtime in various scenarios. Finally, we compare SRS-RecSplit to state-of-the-art MPHF implementations.

Experimental Setup. All timed experiments were performed on an Intel® Core™ i7-11700 CPU with 8 cores and 16 threads, 16 MiB cache, and a base frequency of 2.50 GHz with 4.9 GHz turbo. Furthermore, 64 GB of DDR4 memory were available. We compiled our implementations with rustc 1.81.0 in release mode. For the comparison in Section 7.3 we used the reference implementations of RecSplit [Vig24] and ShockHash [Leh24] compiled with g++ version 11.4.0 with -O3. There, we used uniformly random 16-character alphanumerical strings, as strings are the only type all implementations support natively. For other experiments, we use random 8 byte integers as keys.

7.1. SRS

First, we show some experimental results for a simple application of SRS with constant success probabilities $p_j = p$ and clever bounds k_j. Although not of particular relevance, tasks for this implementation consisted of finding an MPHF of size $t = 10$ by brute force, as it might be required in RecSplit's leafs. All tasks together thus can convert a t-perfect hash function, with at most t keys that get mapped to a single hash, into a perfect hash function ($t = 1$).

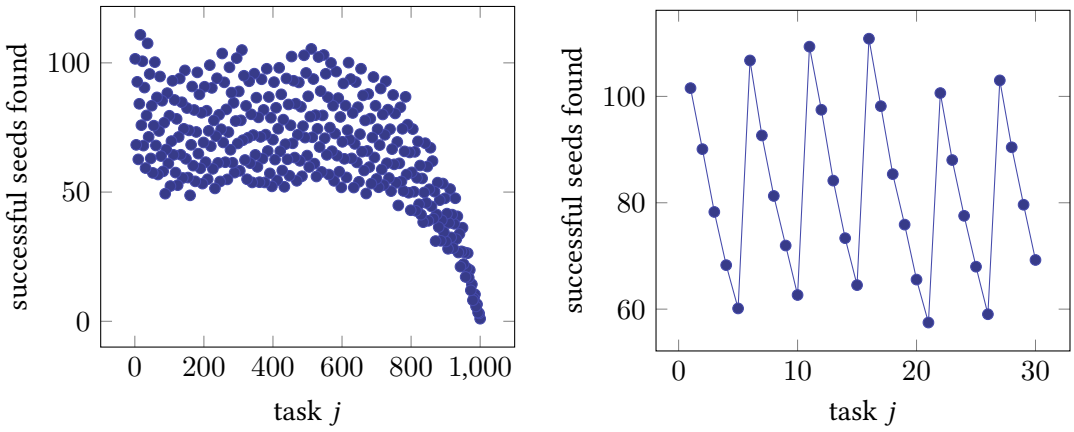
First, in Figure 7.1 we see that runtime is—as expected—linear in the number of tasks and the inverse of the overhead $1/\omega$. We remember, that when SRS finds a successful seed for one task, but afterward fails to find one for the next task in its limited k_j tries, SRS tries to find another successful seed in the previous task. In Figure 7.2a we can see that the number of successful seeds SRS needed to find for a single task does not increase arbitrarily the farther away from the last task we get. Instead, this number is bound by a constant (dependent on ω). This makes SRS require work linear in m. We also see a strong variation in the number of seeds found for tasks close together. This variation is caused by the clever rounding of the bounds k_j (see Subsection 5.2.5). Rounding leads to some tasks having more tries because of rounding up, while others have fewer tries because their bound got rounded down. Thus, the latter tasks cause more retries for their previous task. Figure 7.2b shows the number of successful seeds found for tasks 1 to 30 in more detail. We can see jumps caused by the clever rounding once in a while, providing one bit more to store a bound. The repeating pattern also motivates the runtime analysis for periodic bounds, as shown in Subsection 5.2.5.



(a) $\omega = 0.1$ with varying m

(b) $m = 100$ with varying ω

Figure 7.1.: Number of seeds tested for SRS to be successful for different number of tasks (7.1a) and overhead parameters (7.1b). Number of tested seeds in linear in both m and $1/\omega$, as expected.



(a)

(b) Zoomed in version of 7.2a on tasks 1 to 30.

Figure 7.2.: Number of successful seeds found in each task for $m = 1000, \omega = 0.01$ for constant p_j with cleverly rounded bounds k_j, averaged over 100 samples. Cleverly rounding the bounds to a next power of two causes the visible dips in Figure 7.2b. The last task always requires only one successful seed, as then SRS terminates.

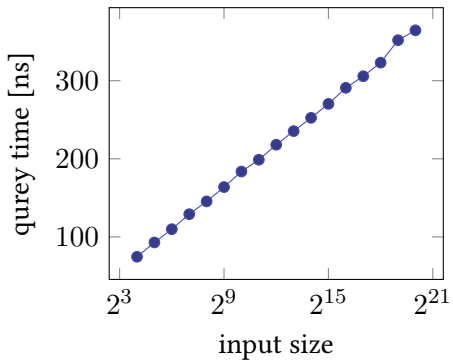


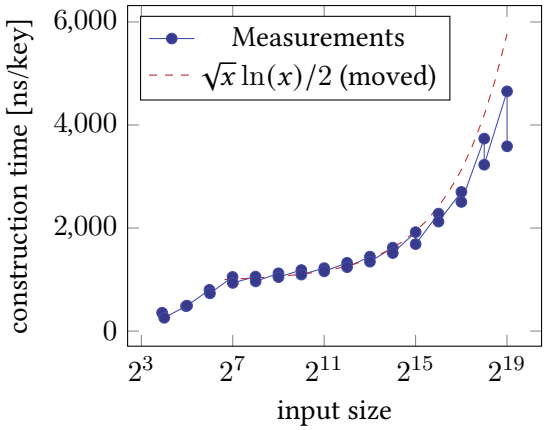
Figure 7.3.: Query time of SRS-RecSplit for different input sizes.

7.2. SRS-RecSplit

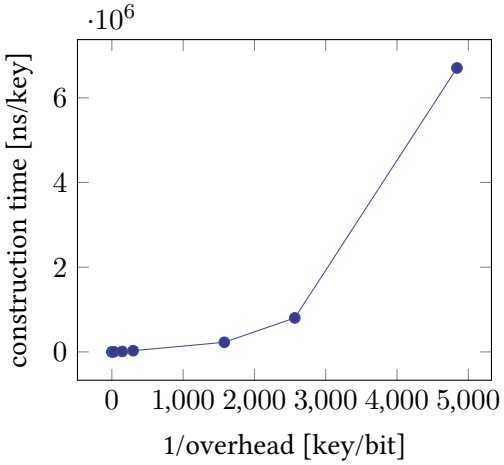
We will now evaluate SRS-RecSplit as described in Section 4.3. Note that we do not use bucketization. In Figure 7.4a we see SRS' construction time per key for various input sizes that are powers of two and not. This creates visible jumps in the measurements, as non-power-of-two inputs are less efficient and less optimized. As SRS-RecSplit has construction time in $\mathcal{O}\left(\frac{n^{3/2}\log n}{\omega}\right)$, we expect to see per key construction time proportional to $\sqrt{n}\log n$, which we do. In Figure 7.4b we do not quite see the expected runtime linear in 1/overhead, especially considering the largest measurement. This may be due to cache inefficiencies as SRS might backtrack out of the cached indices, but only considering the number of hash function evaluations in Figure 7.4d, we do see a linear relationship. Figure 7.3 shows the query time for different input key set sizes, where the expected logarithmic relationship is visible.

7.3. Comparison with Other MPHF Implementations

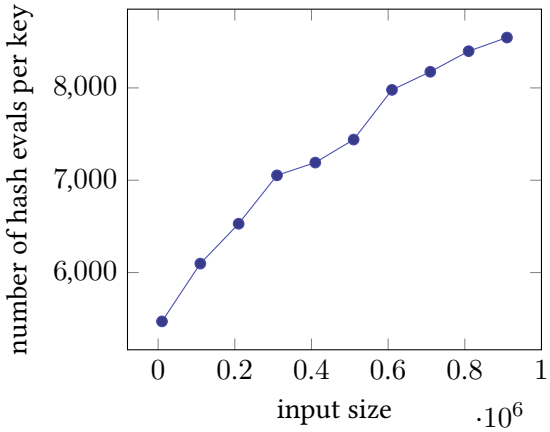
We will now compare SRS-RecSplit to the previously best MPHF implementation regarding space usage—bipartite ShockHash-RS—and RecSplit on which SRS-RecSplit is based. In Figure 7.5 we see the Pareto frontiers of each MPHF algorithm regarding construction time and space usage. A Pareto frontier consists of measurements for different configurations that are not dominated by any other measurement of the same algorithm. A point (a, b) is Pareto dominant over (c, d) iff $a \leq c,\ b \leq d$ where $\leq$ means "better" and at least one of the inequalities is strict. Towards the top left is better, as this means a higher construction throughput (keys per second) and lower space overhead regarding the lower bound of $\log_2 e$ bits per key. For a smaller input size of $n = 2^{15}$ keys (Figure 7.5a), SRS-RecSplit beats ShockHash-RS for even larger overheads and reaches a way smaller overhead, as low as 0.000 21 bits per key. This leads to a space usage of 1.442 91 bits per key. Considering a larger input size of $n = 2^{20}$ (Figure 7.5b), SRS-RecSplit is only competitive for very small overheads because its super-linear construction time in the input size becomes a problem. This can, however, easily be solved in the future by applying a bucketization scheme similar to what ShockHash-RS and RecSplit use.



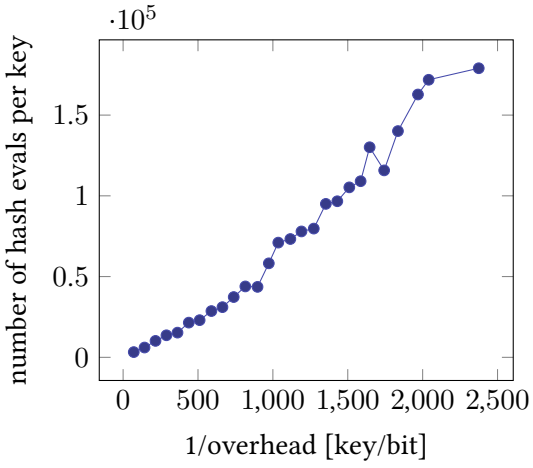
(a) $\omega = 0.1$. The plot shows measurements for input sizes of powers-of-two and one smaller to show the impact of the non-power-of-two construction. This causes the jumps in construction time, as non-power-of-two input sizes are slightly less efficient. Construction time per key is $\mathcal{O}(\sqrt{n}\log n)$ as expected.



(b) $n = 2^{15}$. Practical construction time is not quite linear in 1/overhead, however, the number of hash function evaluations are, see Figure 7.4d.

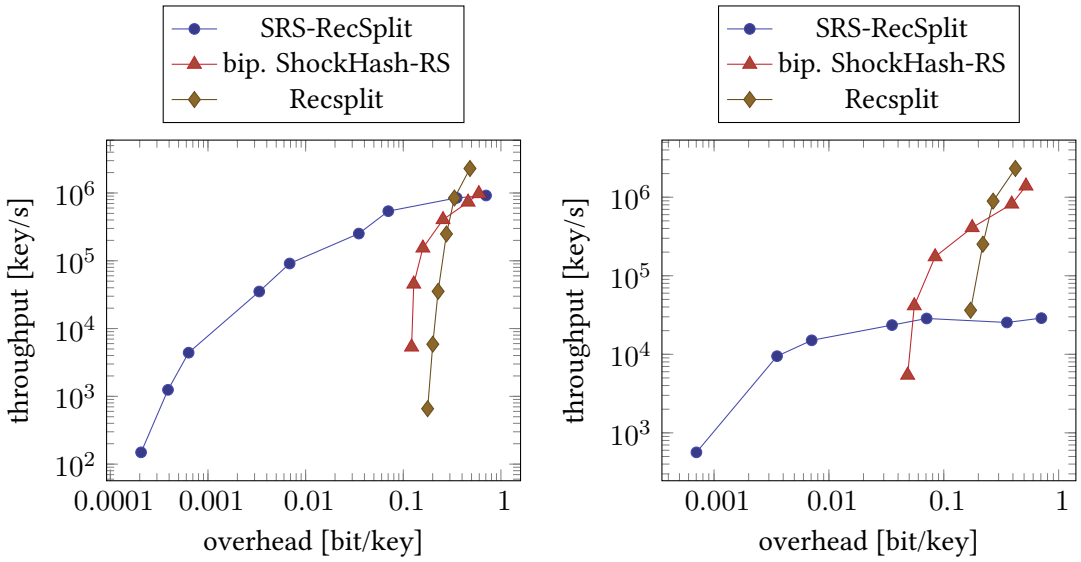


(c) $\omega = 0.01$. Number of hash function evaluations for different input sizes.



(d) $n = 100\,000$. Number of hash function evaluations for different overheads.

Figure 7.4.: Construction time (top) of SRS-RecSplit and number of hash function evaluations necessary for construction (bottom) for different input sizes (left) and overheads (right).



(a) $n = 2^{15}$ **(b)** $n = 2^{20}$

Figure 7.5.: Pareto frontiers of MPHF algorithms, showing construction time vs. space overhead above lower bound of $\log_2 e$. Both axes are logarithmic. The upper left is better.

Table 7.1.: Construction time and space measurements for each of the tested algorithms with roughly comparable construction time or overhead for $n = 2^{15}$. All values are per key. b is bucket size, l is leaf size.

(a) Measurements with comparable construction time. SRS-RecSplit reaches 1/192 the space overhead as bipartite ShockHash-RS uses.

	time	overhead	space
RecSplit $b = 2000,\ l = 14$	170 µs	0.201 bits	1.643 bits
bip. ShockHash-RS $b = 2000,\ l = 128$	187 µs	0.122 bits	1.567 bits
SRS-RecSplit $\omega = 0.001$	227 µs	0.000 634 bits	1.443 33 bits

(b) Measurements with comparable space overhead. SRS-RecSplit needs only 1/101 the construction time as bipartite ShockHash-RS uses.

	time	overhead
RecSplit $b = 2000,\ l = 16$	1520 µs	0.178 bits
bip. ShockHash-RS $b = 2000,\ l = 128$	187 µs	0.122 bits
SRS-RecSplit $\omega = 0.1$	1.85 µs	0.070 bits

Table 7.1 shows some measurements out of the Pareto frontiers for roughly comparable construction times or overheads, for $n = 2^{15}$. SRS-RecSplit reaches 1/192 the space overhead as bipartite ShockHash-RS uses for similar construction time and only needs 1/101 the construction time for the same overhead.

8. Conclusion

In this thesis, we introduced Symbiotic Random Search (SRS) to solve the entropy discrepancy of up to $\log_2 e$ bit, existing between two geometrically distributed random variables and one combined one. SRS requires work linear in the number of tasks m and in the inverse of ω—the overhead per task towards the entropy of a single combined task.

To achieve these results, we analyzed SRS for the case of tasks with varying success probabilities p_j and retry bounds k_j as long as $p_j k_j \leq 1 + \varepsilon = 2^{\omega}$ using non-windowed seeds (Subsection 5.2.1 and Subsection 5.2.2). We provided a sketch of a proof that work does not change asymptotically for the case of constant p, k with seeds limited to a window of w bits for some $w \in \Theta(\log m)$ (Subsection 5.2.4). Furthermore, we proved that we get the same runtime for tasks with equal success probabilities p and rounding the retry bounds k_j to powers of two, such that on average we spend at most ω bits per task more than minimum (Subsection 5.2.5). We argued, that this result can at least be applied for blockwise constant p with losing one additional bit per block. We take this as a strong hint that the same relationship could also apply when combining all these relaxations of requirements.

Afterward, we applied SRS to MPHF construction and RecSplit in particular to get SRS-RecSplit. We explained and analyzed a simple version that only works for power-of-two inputs, and also shortly provided some details on the adaptation to make this version work for non-power-of-two inputs. We then tested our implementation and reached space usage as low as 1.442 91 bits per key. Moreover, we reached $1/192$ of the overhead towards the optimum of the previous record holder for a similar construction time. For similar overheads, constructing SRS-RecSplit is 101 times faster.

Future Work. There are a couple of ideas that can be developed further. First, integrating bucketization into SRS-RecSplit to achieve expected linear construction and constant queries remains one of the most important and promising tasks. Only this would truly make SRS-RecSplit practically usable for large input sizes and competitive regarding construction and lookup time. Additionally, further exploiting adaptive overhead parameters ω could provide some improvements. Finally, we would like to see a more elegant proof for the work SRS requires that may cover all cases.

Bibliography

[ASM65] Milton Abramowitz, Irene A. Stegun, and David Miller. "Handbook of Mathematical Functions with Formulas, Graphs and Mathematical Tables (National Bureau of Standards Applied Mathematics Series No. 55)". In: *Journal of Applied Mechanics* Volume 32 (1965), pp. 239–239.

[BBD09] Djamal Belazzougui, Fabiano C. Botelho, and Martin Dietzfelbinger. "Hash, Displace, and Compress". In: *Algorithms - ESA 2009*. Edited by Amos Fiat and Peter Sanders. Vol. 5757. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 682–693. ISBN: 978-3-642-04127-3. DOI: *10.1007/978-3-642-04128-0_61*.

[Chr24] Christopher. *Cbreeden/Fxhash*. Sept. 17, 2024. URL: *https://github.com/cbreeden/fxhash* (visited on 09/29/2024).

[CLM16] Rayan Chikhi, Antoine Limasset, and Paul Medvedev. "Compacting de Bruijn Graphs from Sequencing Data Quickly and in Low Memory". In: *Bioinformatics* Volume 32 (June 15, 2016), pp. i201–i208. ISSN: 1367-4803. DOI: *10.1093/bioinformatics/btw279*.

[Com] Rust Community. "Rust-Lang/Rust: Empowering Everyone to Build Reliable and Efficient Software." URL: *https://github.com/rust-lang/rust* (visited on 09/29/2024).

[DHSW22] Peter C. Dillinger, Lorenz Hübschle-Schneider, Peter Sanders, and Stefan Walzer. "Fast Succinct Retrieval and Approximate Membership Using Ribbon". In: *20th International Symposium on Experimental Algorithms (SEA 2022)*. Edited by Christian Schulz and Bora Uçar. Vol. 233. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 4:1–4:20. ISBN: 978-3-95977-251-8. DOI: *10.4230/LIPIcs.SEA.2022.4*.

[Dou24] Douman. *DoumanAsh/Xxhash-Rust*. Sept. 21, 2024. URL: *https://github.com/DoumanAsh/xxhash-rust* (visited on 09/29/2024).

[EGV19] Emmanuel Esposito, Thomas Mueller Graf, and Sebastiano Vigna. "RecSplit: Minimal Perfect Hashing via Recursive Splitting". In: *2020 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*. Society for Industrial and Applied Mathematics, Dec. 17, 2019, pp. 175–185. DOI: *10.1137/1.9781611976007.14*.

[Eina] Manfred Einsiedler. "Die Exponentialfunktion – Analysis I (Kap. 1-9)". Analysis I. URL: *https://wp-prd.let.ethz.ch/analysis19/chapter/die-exponentialfunktion/* (visited on 07/15/2024).

[Einb] Manfred Einsiedler. "Summen Und Produkte – Analysis I (Kap. 1-9)". Analysis I. URL: *https://wp-prd.let.ethz.ch/analysis19/chapter/summen-und-produkte/#x1-79001r5* (visited on 09/24/2024).

[Eli74] Peter Elias. "Efficient Storage and Retrieval by Content and Address of Static Files". In: *Journal of the ACM* Volume 21 (Apr. 1974), pp. 246–260. ISSN: 0004-5411, 1557-735X. DOI: *10.1145/321812.321820*.

[Fan71] Robert Mario Fano. *On the Number of Bits Required to Implement an Associative Memory*. Massachusetts Institute of Technology, Project MAC, 1971.

[FCH92] Edward A. Fox, Qi Fan Chen, and Lenwood S. Heath. "A Faster Algorithm for Constructing Minimal Perfect Hash Functions". In: *Proceedings of the 15th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. New York, NY, USA: Association for Computing Machinery, June 1, 1992, pp. 266–273. ISBN: 978-0-89791-523-6. DOI: *10.1145/133160.133209*.

[FHCD92] Edward A. Fox, Lenwood S. Heath, Qi Fan Chen, and Amjad M. Daoud. "Practical Minimal Perfect Hash Functions for Large Databases". In: *Communications of the ACM* Volume 35 (Jan. 2, 1992), pp. 105–121. ISSN: 0001-0782. DOI: *10.1145/129617.129623*.

[FKS84] Michael L. Fredman, János Komlós, and Endre Szemerédi. "Storing a Sparse Table with *0(1)* Worst Case Access Time". In: *Journal of the ACM* Volume 31 (June 26, 1984), pp. 538–544. ISSN: 0004-5411, 1557-735X. DOI: *10.1145/828.1884*.

[Gol66] S. Golomb. "Run-Length Encodings (Corresp.)" In: *IEEE Transactions on Information Theory* Volume 12 (July 1966), pp. 399–401. ISSN: 1557-9654. DOI: *10.1109/TIT.1966.1053907*.

[GvV75] R. Gallager and D. van Voorhis. "Optimal Source Codes for Geometrically Distributed Integer Alphabets (Corresp.)" In: *IEEE Transactions on Information Theory* Volume 21 (1975), pp. 228–230. DOI: *10.1109/TIT.1975.1055357*.

[Her+24] Stefan Hermann, Hans-Peter Lehmann, Giulio Ermanno Pibiri, Peter Sanders, and Stefan Walzer. "PHOBIC: Perfect Hashing with Optimized Bucket Sizes and Interleaved Coding". Apr. 29, 2024. arXiv: *2404.18497*.

[Hoa62] C. A. R. Hoare. "Quicksort". In: *The Computer Journal* Volume 5 (Jan. 1, 1962), pp. 10–16. ISSN: 0010-4620. DOI: *10.1093/comjnl/5.1.10*.

[Jen06] J. L. W. V. Jensen. "Sur Les Fonctions Convexes et Les Inégualités Entre Les Valeurs Moyennes". In: (Nov. 30, 1906). DOI: *10.1007/bf02418571*.

[Kai24] Tom Kaitchuck. *Tkaitchuck/aHash*. Sept. 26, 2024. URL: *https://github.com/tkaitchuck/aHash* (visited on 09/29/2024).

[Kle20] Achim Klenke. "Basic Measure Theory". In: *Probability Theory: A Comprehensive Course*. Cham: Springer International Publishing, 2020, pp. 1–51. ISBN: 978-3-030-56402-5. DOI: *10.1007/978-3-030-56402-5_1*.

[Kno75] G. D. Knott. "Hashing Functions". In: *The Computer Journal* Volume 18 (Jan. 1, 1975), pp. 265–278. ISSN: 0010-4620. DOI: *10.1093/comjnl/18.3.265*.

[Knu92] Donald E. Knuth. *Two Notes on Notation*. 1992. arXiv: *math/9205211*.

[Leh24] Hans-Peter Lehmann. *ByteHamster/ShockHash*. Sept. 19, 2024. URL: *https://github.com/ByteHamster/ShockHash* (visited on 09/29/2024).

[LPB06] Yi Lu, Balaji Prabhakar, and Flavio Bonomi. "Perfect Hashing for Network Applications". In: *2006 IEEE International Symposium on Information Theory*. 2006 IEEE International Symposium on Information Theory. July 2006, pp. 2774–2778. DOI: *10.1109/ISIT.2006.261567*.

[LSW23] Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer. "SicHash - Small Irregular Cuckoo Tables for Perfect Hashing". In: *2023 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*. Society for Industrial and Applied Mathematics, Jan. 2023, pp. 176–189. DOI: *10.1137/1.9781611977561.ch15*.

[LSW24] Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer. "ShockHash: Near Optimal-Space Minimal Perfect Hashing Beyond Brute-Force". June 5, 2024. arXiv: *2310.14959*.

[Meh84] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*. Springer Berlin Heidelberg, 1984. ISBN: 978-3-540-13302-5.

[MS08] "Hash Tables and Associative Arrays". In: *Algorithms and Data Structures: The Basic Toolbox*. Edited by Kurt Mehlhorn and Peter Sanders. Berlin, Heidelberg: Springer, 2008, pp. 81–98. ISBN: 978-3-540-77978-0. DOI: *10.1007/978-3-540-77978-0_4*.

[MSSZ14] Ingo Müller, Peter Sanders, Robert Schulze, and Wei Zhou. "Retrieval and Perfect Hashing Using Fingerprinting". In: *Experimental Algorithms*. Edited by Joachim Gudmundsson and Jyrki Katajainen. Vol. 8504. Cham: Springer International Publishing, 2014, pp. 138–149. ISBN: 978-3-319-07958-5. DOI: *10.1007/978-3-319-07959-2_12*.

[MvOV18] A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 2018. ISBN: 978-0-429-88132-9.

[NP12] Gonzalo Navarro and Eliana Providel. "Fast, Small, Simple Rank/Select on Bitmaps". In: *Experimental Algorithms*. Edited by Ralf Klasing. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 295–306. ISBN: 978-3-642-30850-5.

[PR04] Rasmus Pagh and Flemming Friche Rodler. "Cuckoo Hashing". In: *Journal of Algorithms* Volume 51 (2004), pp. 122–144. ISSN: 0196-6774. DOI: *10.1016/j.jalgor.2003.12.002*.

[PT21] Giulio Ermanno Pibiri and Roberto Trani. "PTHash: Revisiting FCH Minimal Perfect Hashing". In: *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. New York, NY, USA: Association for Computing Machinery, July 11, 2021, pp. 1339–1348. ISBN: 978-1-4503-8037-9. DOI: *10.1145/3404835.3462849*.

[Rom00] Dan Romik. "Stirling's Approximation for *n!*: The Ultimate Short Proof?" In: *The American Mathematical Monthly* Volume 107 (June 2000), pp. 556–557. ISSN: 0002-9890, 1930-0972. DOI: *10.1080/00029890.2000.12005235*.

[RP71] R. Rice and J. Plaunt. "Adaptive Variable-Length Coding for Efficient Compression of Spacecraft Television Data". In: *IEEE Transactions on Communication Technology* Volume 19 (1971), pp. 889–897. DOI: *10.1109/TCOM.1971.1090789*.

[Sha48] C. E. Shannon. "A Mathematical Theory of Communication". In: *The Bell System Technical Journal* Volume 27 (1948), pp. 379–423. DOI: *10.1002/j.1538-7305.1948.tb01338.x*.

[Ste04] J. Michael Steele. "The AM-GM Inequality". In: *The Cauchy-Schwarz Master Class: An Introduction to the Art of Mathematical Inequalities*. Cambridge: Cambridge University Press, 2004, pp. 19–36.

[Tho22] Jack Thomson. *JackThomson2/Wyhash2*. Aug. 25, 2022. URL: *https://github.com/JackThomson2/wyhash2* (visited on 09/12/2024).

[Vig24] Sebastiano Vigna. *Vigna/Sux*. Sept. 20, 2024. URL: *https://github.com/vigna/sux* (visited on 09/29/2024).

[Yi24] Wang Yi. *Wangyi-Fudan/Wyhash*. Aug. 30, 2024. URL: *https://github.com/wangyi-fudan/wyhash* (visited on 09/12/2024).

[Zie24] Jonatan Ziegler. "Worldofjoni/Srs-Rs-Rs". 2024. URL: *https://github.com/worldofjoni/srs-rs-rs* (visited on 09/29/2024).

A. Appendix

A.1. Table of Symbols

In this section, we provide an overview of the most important symbols used in this thesis.

A.1.1. MPHF and SRS-RecSplit

Symbol	Description
$n \in \mathbb{N}^+$	MPHF size.
U	Universe of keys.
$X \subseteq U$	Set of input keys.
$f_s : U \to C$	Seeded hash function for seed $s \in \mathbb{N}_0$. Codomain C depends on context.
$r = \lceil \log_2 n \rceil$	Number of layers in SRS-RecSplit's splitting tree.
$i \in r, \ldots, 1$	Layer indices in SRS-RecSplit's splitting tree.
p_i, W_i, ω_i	Same meaning as for SRS (see below), but applying for each task in layer i.

A.1.2. SRS

Symbol	Description
$m \in \mathbb{N}^+$	Number of SRS tasks.
$j \in [m]$	Task indices.
$i_j \in \mathbb{N}_0$	Seeds.
$S_j \subseteq \mathbb{N}_0$	Set of successful seeds.
$p_j \in (0, 1]$	Success probability for $i_j \in S_j$.
W_j	Work required for one test $i_j \in S_j$.
$k_j \in \mathbb{N}_0$	Bound on number of seeds tested in one go.
$\sigma(j) \in \mathbb{R}_0^+$	Targeted bits to use for tasks 1 to j.
$l_j \in [k_j]_0$	Indices of tries in each task, from which seeds are constructed: $i_j = \eta(i_0, l_1, \ldots, l_j)$.
η	Binary concatenation function.
$i_0 \in \mathbb{N}_0$	Root seed required to achieve infinite retries.
$w \in \mathbb{N}^+$	Seed window size used in practice for efficient handling of seeds.
$\omega \in (0, 1]$	Overhead per Task in bits. $\omega = \log_2(1 + \varepsilon)$.
$\varepsilon \in (0, 1]$	Bound on $p_j k_j \geq 1 + \varepsilon$ ($j \in [m]$) for convergence of SRS' success.
$q_j \in [0, 1]$	Probability, that SRS returns successful from task j including all following.