

**FACULTY
OF MATHEMATICS
AND PHYSICS**
Charles University

BACHELOR THESIS

Vojtěch Gaďurek

Data Structures for Sketching Dynamic Sets

Computer Science Institute of Charles University

Supervisor of the bachelor thesis: Pavel, Veselý, Mgr. Ph.D.

Study programme: Computer Science

Prague 2024

I declare that I carried out this bachelor thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In date

Author's signature

I want to thank all who were helpful namely Mgr. Jiří Beneš, Mgr. Radek Zikmund, especially my supervisor, Mgr. Pavel Veselý, Ph.D. for all he has done and all the consultations he provided. I am very glad to have him behind my back.

We used AI tools quite extensively in this project. We primarily used them to refactor, transform, or rewrite already-written text or code. We have not primarily used AI for idea generation. The tools we used were ChatGPT, Grammarly, and GitHub Copilot. This list is not enumerative and specifies only the most used.

Title: Data Structures for Sketching Dynamic Sets

Author: Vojtěch Gadurek

Institute: Computer Science Institute of Charles University

Supervisor: Pavel, Veselý, Mgr. Ph.D., Computer Science Institute of Charles University

Abstract: Our work explores algorithms for finding the symmetric differences between two large sets that differ only in a few elements. We focus on performance and modularity from an implementation perspective and memory complexity from an algorithmic perspective. We achieved fast and modular code using advanced C# features, mainly expression trees, allowing us to achieve update speeds of 152 MOp/s k . We compared different hash functions, finding tabulation to have the best properties for recovery. Finally, the algorithms were applied to structured data, improving the ratio of the memory size to the number of elements in the difference from 1:0.8 to 1:1.5 for k -mers.

Keywords: Set Recovery, Expression trees, Bloom Filters, k -mers

Název práce: Datové struktury pro sketching dynamických množin

Autor: Vojtěch Gadurek

Ústav: Informatický ústav Univerzity Karlovy

Vedoucí bakalářské práce: Pavel, Veselý, Mgr. Ph.D., Computer Science Institute of Charles University

Abstrakt: Naše práce se zabývá algoritmy pro nalezení symetrických rozdílů mezi dvěma velkými množinami, které se liší pouze v několika prvcích. Zaměřujeme se na výkon a modularitu z hlediska implementace a pamětovou složitost z hlediska algoritmů. Rychlého a modulárního kódu jsme dosáhli pomocí pokročilých funkcí C#, zejména Expression trees, což nám umožnilo dosáhnout rychlosti aktualizace 152 MOp/s k . Porovnali jsme různé hashovací funkce. Nakonec byly algoritmy aplikovány na strukturovaná data, což zlepšilo poměr velikosti paměti k počtu prvků v rozdílu z 1:0.8 na 1:1.5 pro k -mery.

Klíčová slova: Rekonstrukce množiny, Expression trees, Bloomovy Filtry, k -mery

Contents

Introduction	8
1 Preliminaries	10
1.1 Randomized Algorithms	10
1.2 Hypergraphs	11
1.2.1 Basics	11
1.2.2 2-cores in Random Hypergraphs	12
1.3 Hashing	13
1.3.1 Problem of Dictionary	13
1.3.2 Hash Function Families	13
1.3.3 Examples of Hash Function Families	14
1.3.4 Tabulation	14
1.3.5 Choosing the Right Hash Function	15
1.4 Strings	15
1.5 Bioinformatics	15
1.5.1 DNA and kMers	15
1.5.2 Masked Superstrings	15
2 Set Recovery Problems	16
2.1 Introduction to Streaming algorithms	16
2.1.1 Motivation	16
2.1.2 Use Cases for Streaming Algorithms	17
2.1.3 Number of Distinct Elements in MultiSet	18
2.2 Statement of the Problem	19
2.3 From Simple solutions to IBLT's	20
2.3.1 Solving 1-SetRecoveryProblem	20
2.3.2 From 1 to k -SetRecoveryProblem Solution	21
2.3.3 Invertible Bloom Lookup Table	22
2.4 HPW Algorithm	24
3 Dotnet and C# Preliminaries	26
3.1 Dotnet and Just-In-Time Compilation	26
3.2 Memory	27
3.2.1 Heap and Stack	27
3.2.2 Value and Reference Types	27
3.2.3 Garbage Collector	29
3.3 Abstraction, Virtual Methods, Delegates, Interfaces	30
3.3.1 Inheritance and Virtual Methods	30
3.3.2 Interfaces	31
3.3.3 Generics	32
3.4 Expression Trees	33
3.4.1 A Simple example	33
3.4.2 Construct-Compile	34
3.4.3 Base API	34
3.4.4 Issues with Expression Trees	37

3.5	Compiler Optimizations	37
3.5.1	When the Compiler Inlines	40
3.6	Constant Optimization	40
4	LittleSharp	41
4.1	Motivation	41
4.2	How to use	41
4.2.1	Naming Conventions	42
4.2.2	Debugging	42
4.3	Implementation details	43
4.3.1	Architecture	43
4.3.2	SmartExpression operators	44
4.3.3	Scopes	45
4.4	Future work	45
5	FlashHash	47
5.1	Motivation	47
5.2	How to Use	47
5.3	Implementation	47
5.3.1	Simplest Solution	47
5.3.2	Possible solution via Runtime Types with Struct Delegates	48
5.3.3	Solution using Expression Trees (LittleSharp)	50
5.3.4	Comparing Approaches	50
5.4	Future work	51
6	RedaFasta	52
6.1	Motivation	52
6.2	How to use	52
6.2.1	Disposing	53
6.3	Future work	53
7	SymmetricDifferenceFinder	54
7.1	Motivation	54
7.2	Overview	54
7.2.1	Encoding-Decoding Pipeline	54
7.2.2	Hash Functions	55
7.2.3	Encoder	55
7.2.4	Decoders	55
7.2.5	HWPDecoder	55
7.2.6	HyperGraphDecoder	55
7.2.7	Decoders with oracles	55
7.2.8	Data sources	56
7.3	Building a Custom Solution	56
7.3.1	Creating a table	56
7.3.2	Choosing Hash Functions	56
7.3.3	Choosing Encoder	57
7.3.4	Using Encoder	57
7.3.5	Sketch	58
7.3.6	Choosing Decoder	59

7.4	Buffers size	60
7.5	Future Work	60
8	Experiments	61
8.1	Comparison of Hash Functions	61
8.2	Decoding with oracle	63
8.2.1	Experiments	65
8.3	Performance	70
8.3.1	Hash functions	70
8.4	Updates	71
8.4.1	Update Cost Depending on the Size of the Sketch	71
8.4.2	Encoding k -mer sets	72
	Conclusion	75
8.5	Comparison of HPW to IBLT	75
	Bibliography	76
	List of Figures	78
	List of Tables	79
	List of Abbreviations	80

Introduction

Motivation

Advances in technology in the last decade led to a huge accumulation of data. This has created the need for algorithms that can work with large data in small memory. In our work, we focus on the problem of finding the symmetric difference of two large sets that differ only in a small number of elements. Finding the symmetric difference between two sets has many applications from keeping instances of distributed databases in sync to comparing closely related genomes.

Basic Approach

The state-of-the-art solution to such a problem is to use an Invertible Bloom Lookup Table (IBLT). See section 2.3.3. This data structure is very close to a hash table. It differs mainly in the way how it deals with collisions. It just adds the values of the colliding keys and keeps the count of keys in each bucket. This means that when an item is added to the table and then removed, the table will be the same as the table without these two operations. This property is very useful in our case because we can add all elements from the first set to the table and then remove all elements from the second set; only the items in the difference remain in the table. We then may try to recover the difference from the table, returning all values in buckets where only one key was hashed. This, however, leads to some keys being unrecovered if multiple keys were hashed to the same bucket. Another crucial difference in the table is that we do not use one hash function but k hash functions, for example, $k = 3$. So, each key is stored in k buckets, improving the probability no two keys are hashed into the same buckets. We may use such a structure to find the symmetric difference of two sets in space linear in the size of the difference. The authors Houen, Pagh, and Waltzer built on the IBLT and proposed a new data structure that improves the space consumed by a constant. [1]. See Section 2.4 Their improvement lies in replacing the summation of values in the buckets with the XOR operation and not keeping the count of keys in the buckets. This may lead to the recovery of keys, not in the difference, but the authors successfully showed such an event does not happen with high probability.

Implementation

We expected a typical use case to be run on large data, therefore we highly prioritized performance. By combining low-level optimization and parallelism, we achieved a speed of 130 million encoded k -mers per second on 4 cores with 32GB of RAM. See Table 8.4.2

However, we also made the library highly modular, allowing users to replace each part easily. Our main and only programming language was C#. We used its runtime code generation features extensively, namely Expression Trees.

We use them in our custom-made library, FlashHash, which provides access to many hash functions from two hash function families. These hash functions are

compiled during runtime for better performance. See Chapter 5.

Working with Expression Trees may be tedious and cumbersome. As such, we developed a small library—LittleSharp—to make working with them more enjoyable and safe. It provides a reasonable type of safety, making many otherwise runtime errors a compile-time error. See Chapter 4.

Lastly, we implemented a library for reading superstrings of k -mers called RedaFasta. See Chapter 6.

Improvements

Recovery with oracle

In Bioinformatics, one often work with k -mer sets instead of DNA strings. k -mer is DNA substring of length k . Finding symmetric difference of two k -mer sets is useful for closeness of different species. We were able to use the fact that data in such sets are not entirely random, and from the existence of some k -mer, we may infer the existence of other k -mers. We called such a process a recovery with an oracle. With this, we beat the theoretical bound of 1:1 ratio by the ratio 1:1.52 for 31-mer sets. We found this value experimentally. See Figure 8.4.

Other results

We found no difference between IBLT and HPW regarding the probability of recovery. However, encoding to IBLT is much more expensive.

We found that different hash function families significantly influence recovery and performance. The best hash function family for recovery with Oracle was tabulation, which was, however, more than 10 times slower than the fastest hash function family multiply-shift. They, however, had a comparable success rate for native recovery.

1 Preliminaries

1.1 Randomized Algorithms

We are going to need some tools to work with mean and variance. The simplest is:

Theorem 1. (*Markov Inequality*) *Given a nonnegative random variable X with expectation $E[X]$, it holds that*

$$\Pr[X > kE[X]] \leq \frac{1}{k}$$

In simplicity, this theorem says that only the $\frac{1}{k}$ fraction of the people can be k -times richer than the mean.

Definition 1. (*Variance*) *Let X be random variable, then*

$$\text{Var}[X] := E[(E[X] - X)^2]$$

.

One can, with little work, get a Chebichev Inequality from Markov Inequality.

Theorem 2. (*Chebyshev Inequality*) *Given a random variable X with expectation*

$$\Pr[|X - E[X]| > k] \leq \frac{\text{Var}[X]}{k^2}$$

The previous theorems work for all random variables. We may get a better bound by requiring more from the random variables. We will require that they are independent and are the sum of Bernoulli random variables.

Theorem 3. (*Multiplicative Chernoff bound*) *Given independant Bernoulli random variable $X_1, X_2 \dots X_n$ such that $X = \sum_{i=1}^n X_i$ and $E[X] = \mu$, then for $0 \leq \beta \leq 1$*

$$\Pr[X \leq (1 - \beta)\mu] \leq \exp\left(\frac{-\beta^2\mu}{2}\right)$$

We have two main types of randomized algorithms: Las Vegas and Monte Carlo. Las Vegas algorithms are guaranteed to return correct answers, but they are not guaranteed to end. Monte Carlo does not always return the correct answer, but the probability of failure is low. Often, the value is set not to be greater $\frac{1}{3}$. A correct answer does not have to be exactly the right value; sometimes, we allow some amount of error and answers close enough, which we count as success.

The number $\frac{1}{3}$ is arbitrary. We will show that any algorithm with a probability of success more than $\frac{1}{2}$ can be modified to increase its probability of success to any value smaller than 1. This trick is often called the Median trick.

Median Trick

Let us say we have some randomized algorithm A with a probability of success p . Now we run k instances of A . We then pick the median of all results. This approach fails when the median instance is a failure. This happens only if more than $\frac{k}{2}$ instances were failures.

We can now use Multiplicative Chernoff bound to determine how many instances of A we must run to get some desired success rate of p' . We set X as the number of successful instances. We need $(1 - \beta)\mu$ to equal $\frac{k}{2}$.

That is,

$$(1 - \beta)\mu = (1 - \beta)kp = \frac{k}{2}$$

$$\beta = \frac{2p + 1}{2p}$$

Then we get:

$$\Pr[X \leq \frac{n}{2}] \leq \exp\left(\frac{-(2p + 1)n}{2}\right) = 1 - p'$$

This means the probability of failure decreases exponentially with the number of instances run. So, when we want an error rate δ , then we need to run $\mathcal{O}(\log(\frac{1}{\delta}))$ instances.

This is a very useful trick as it allows us to strive for randomized algorithms with low success rates, and we are punished just by logarithmic slowdown.

Some algorithms are more likely to succeed with larger data sets. If we can prove that the limit in the infinity of such probability is one, we say that the algorithm succeeds with high probability. An abbreviation "whp" is often used.

1.2 Hypergraphs

A graph is a tuple of two sets. One set contains vertices and one edge. Normally, edges may connect only two vertices. Formally, edges are sets containing exactly two vertices, but we may generalize this concept to allow more vertices be connected by one edge.

1.2.1 Basics

Definition 2. Let V be a set and $E \subseteq P(V)$, then the tuple (V, E) is called a hypergraph. Elements of V are called vertices, and aspects of E are called edges.

One should easily see that every graph is also a hypergraph. The other way is not true. Some terms are connected to graphs, which would be useful to state also for hypergraphs. Definitions are going to be very similar to their graph equivalents. We begin with the subgraph:

Definition 3. Let $G := (V, E)$ be a hypergraph, then $G' := (V', E')$ is a subgraph of G if $V' \subseteq V$ and $E' \subseteq E$, and we may write that as $G' \subseteq G$.

Definition 4. Let G be some hypergraph and its G' subgraph, then G' is a k -core if every vertex of G' has a degree of at least k and G' is the largest such subgraph of G .

Sometimes, we would like to have edges that are only the same size. This concept is called k -uniformity.

Definition 5. Let $G := (V, E)$ be a hypergraph, then G is k -uniform, if $\forall e \in E : |e| = k$.

We define degree:

Definition 6. Let $G := (V, E)$ be a hypergraph, and be v some vertex from V then $\Delta(v) = |\{e \in E; v \in e\}|$. We call this property degree of vertex.

and used them to define k -cores

Definition 7. k -core is the largest subgraph H of graph G such that all vertices have a degree of at least k .

1.2.2 2-cores in Random Hypergraphs

We start by defining k -uniform random hypergraph with m edges.

Definition 8. Be V set of vertices and A set of hypergraphs on V . Then, the random graph $G(V, A)$ on A is a graph uniformly picked from A .

Definition 9. Be V set of vertices and A set of k -uniform hypergraph on V with m edges, then k -uniform random hypergraph with m edges is defined as $G_k^m := G(V, A)$

Now, we may state an important bound proved by Molloy [2]. He proved that random 3-uniform hypergraphs with n vertices and m edges do not contain a 2-core with high probability if $\frac{n}{m} \geq 1.222 \dots$. This solution is tight; thus, when the ratio is less than $1.222 \dots$, then 2-core is found with high probability.

Finding 2-cores in a hypergraph is quite easy.

Algorithm 1 Step $G : (V, E) \rightarrow (V, E)$

```

1: procedure STEP( $G = (V, E)$ )
2:    $ToBeRemoved \leftarrow \{e \in E \mid (\exists v \in e)(\text{degree of } v \text{ is } 1)\}$ 
3:   return  $(V, E \setminus ToBeRemoved)$ 
4: end procedure

```

Greedy algorithms have a bad tendency not to work. However, it works for this problem. If some vertices lie in only one edge, we can pair these two together and remove such edges. If there exists none such vertex, then we found 2 – core. Algorithm 2.

Algorithm 2 Run $G : (V, E) \rightarrow \text{False or True}$

```
1: procedure RUN( $G = (V, E)$ )
2:    $(V, \text{New}) \leftarrow \text{STEP}(G)$ 
3:   if  $E = \text{New}$  then
4:     return False
5:   end if
6:   if  $E = \{\}$  then
7:     return True
8:   end if
9:   return RUN( $(V, \text{New})$ )
10: end procedure
```

1.3 Hashing

1.3.1 Problem of Dictionary

The first time one meets with hash functions is when faced with the need for a data structure with an $O(1)$ time cost for insert and get by key. Suppose the keys are from $[n]$, where the number of inserts $O(n)$, then the solution is simple. We can create an array of length n and index into it.

This solution, however, depends on the size of the universe. If the universe U has a size of M , then we need to use a table of size M . This may not and often is not feasible. For example, there are 2^{32} 32-bit integers. The main solution to this problem is to have some "good" hash function $f : U \rightarrow [n]$, where n is some desired table size. What means "good"? When we insert some item a with key k in the table, we insert at $f(k)$ index. When there is already a value in the table, we get a hash conflict. There are many possible solutions to this. The simplest solution is to keep more values in the same field.

1.3.2 Hash Function Families

We would like to minimize hash conflicts. There are a few properties that may be useful in determining such "goodness." Before we look at them, we need to address one problem. If we choose a hash function deterministically, the enemy may choose data hash with very few values. We may prevent this by picking a hash function randomly from some hash function family.

Definition 10. A family F of hash functions from $X \rightarrow Y$, where $|Y| = n$, is **c-universal** if

$$\forall(x, y \in X) \left(P(f(x) = f(y)) \leq \frac{c}{n} \right),$$

where f is chosen uniformly randomly from F and $x \neq y$.

Universality guarantees that hashes are more or less evenly distributed. This is a useful property. It allows us to state that the expected number of items with the same hash is $E(\frac{c}{n})$.

One problem with universality is that it allows for a hash function family where all hash functions map 1 to 0. This is prevented by being a hash function

family (c, k) -independent. This property is sometimes called strong universality in the literature.

Definition 11. A family F of hash functions from $X \rightarrow Y$, where $|Y| = n$, is **(c, k) -independent** if

$$\forall(x_1, \dots, x_k \in X, y_1, \dots, y_k \in Y) \left(P(f(x_1) = y_1 \wedge \dots \wedge f(x_k) = y_k) \leq \frac{c}{n^k} \right),$$

where f is chosen uniformly randomly from F and all x_i all distinct.

1.3.3 Examples of Hash Function Families

Linear Congruence

One of the simplest hash functions is linear congruence. This function is defined as:

Definition 12. Let p be a prime number, n be from $\mathbf{N}$, a a number from $[n - 1]$, and b a number from $[p]$. Then, the hash function family is defined as:

$$f_{a,b}(x) = (ax + b) \mod p \mod n$$

Let H_p^n be set of all such functions with prime p and size n , then we H_p^n a linear congruence family.

Linear congruence family is also useful in practice. Dictionaries in C# are actually implemented using a linear congruence family. The main advantage of this family is that if we choose the size of the universe, we are hash to, then we get a function from a linear congruence family practically for free. The linear congruence family is $(2, 4)$ -independent.

Multiplyshift

The multiply shift family is another often used hash function family. For example, murmurhash is based on it. It has one advantage over linear congruence. It uses only logical shifts and multiplications. However, it guarantees only 2-universality. It is defined as

Definition 13. Let $a \in \{0, 1\}^n$ and $r < n$. Then the multiplyshift hash functions is defined as:

$$f_{a,r}(x) := \text{rightshift}(ax, n - r)$$

is a Multiplyshift hash function from $\{0, 1\}^n$ to $\{0, 1\}^r$

1.3.4 Tabulation

Tabulation is a hash function family with many good properties. It is 3-independent. For us, Invertible Bloom Lookup tables may use a hash function from the tabulation hash function family instead of fully random hash functions. [3] Tabulation is also very performant as it requires only table lookups and xor.[3] We lastly show a definition of tabulation:

Definition 14. *Be L set of size n and be $x := (x_1, \dots, x_m)$ of size m . Than let we have m lookup tables h_i of size n and be these lookup tables filled with random values from some universe U , than:*

$$h(x) = h_1(x_1) \oplus \dots \oplus h_m(x_m)$$

than h is tabulation hash function from L^n to U .

1.3.5 Choosing the Right Hash Function

In the previous section, we described some basic properties that are often required from hash functions. However, one should remember that the more independent the hash family is, the better it is. Sometimes, using universal hash function families instead of independent ones may, in reality, give good enough results. This is especially useful in performance-critical applications as less independent hash function families are often faster.

1.4 Strings

We will look at string very intuitively as a finite sequence of symbols. Let A be some string, then A_i is i -th symbol, $A[i : k]$ is substring of A starting at i and before the k -th. When i equals 0 or k to the length of the string, we usually do not write them; thus, $A[0 : k] = A[:k] = A$, if the length of A is k .

1.5 Bioinformatics

1.5.1 DNA and kMers

For our purposes, one may imagine DNA as a string of four letters: A, C, G, and T. Two and two form pairs, namely being A with T and C with G. To every DNA string S , there exists its complement C . This complement may be obtained by first replacing all letters with the other letter in pairs and then reversing the string. This means "AAAT" has a complement "ATTT".

DNA strings may be long, and getting them may be hard. As such, we often work on sets of k -mer. k -mer is a substring of length k . Similarly, as DNA has a complement, k -mers also have complements. We would sometimes need to have just one representation of a k -mer and its complement; thus, we define a canonical kMer as being lower in lexicographical ordering.

1.5.2 Masked Superstrings

When we have a set of keys, it is k -times more expensive to store them than a DNA string. This string may, however, be hard to get or even impossible. As Masked Superstring is used. The main idea is to encode the k -mers into one long string. We then get all the k -mers, by taking all substrings of length k of the string. However, creating such a string could be hard or impossible, so some k -mers are marked as not in the set. The simplest solution is to use upper and lower symbols, where upper means that k -mer starting with such symbol is in the set, and lower means such k -mer is not in the set. [4]

2 Set Recovery Problems

2.1 Introduction to Streaming algorithms

2.1.1 Motivation

The main idea behind streaming algorithms is that they are very similar to natural streams. After the water passes, there is no way to get it back. Similarly, streaming algorithms may view every data piece just once, never going back. They are often very limited in memory, which they can use; the maximum allowed is, at most, polylogarithmic to the size of data or to the size of the universe from which the data is taken. One may ask whether such harsh memory bounds are needed as memory is plentiful and cheap.

In the early days of computing, memory was a significant hurdle. It was both limited and expensive. However, this has changed dramatically. Today, we can store vast amounts of data inexpensively. As a result, people often opt to add more storage rather than spend time deleting unnecessary files.

This environment has led to a massive accumulation of data, sometimes exceeding what local machines can handle. One example might be some network router that processes large quantities of data. This is a prime example of a problem that requires the use of a streaming algorithm. The router gets some data, then performs some operations with them, and lastly, it flushes them and never sees them ever again. This may be useful for finding heavy hitters and thus deciding which IP addresses to keep in the cache.

Another challenge is limited throughput. Imagine having two distant instances of a massive database; it would be ideal for synchronizing them. However, sending all the data over the network to check if they are the same isn't always feasible due to bandwidth constraints. We would like to have an algorithm minimizing such costs.

At first glance, these two problems do not seem very similar. However, the opposite is true. When we have a good algorithm for processing large data in small memory, we can also use it to solve the second example. Suppose we have two databases, S_1 and S_2 . We want to perform operation o on $S_1 \cup S_2$. We have access to some algorithm A that can do o with memory of M , and A does not care about the order in which it consumes data, and it does not expect to see data again. Then we may first do operation o on instance S_1 , save the state of the algorithm, send it to instance S_2 , and restart the process. We had to use at most $\mathcal{O}(M)$ bits of information to ship the content of working memory to the other database.

The way an internet router accesses data may be formalized by the concept of the data stream. We may imagine the data stream as an oracle holding some sequence of words internally, which are inputs to our program. We may ask the oracle to provide us with the next item in the sequence. This is the only way to access the input data, and there is no way back. So if we get the k -th item from the sequence, there is no way to see the $(k - i)$ -th item again for some natural i . More formally:

Definition 15. *Data Stream is tuple (A, i, Access) , where A is a sequence of*

unknown length N of finite words from universe U , i is a counter, which may only increase in value. `Access()` returns the value of A_i and increases the value of a counter i by one.

This may not seem to be limiting as we might create a large array and copy elements from the Data Stream to it and then continue with normal computation as in RAM. This leads us to the limitation of memory usage.

When designing algorithms, we prefer those with polylogarithmic space complexity over the size of the universe and the number of items in the Data Stream.

In computer science, the $\mathcal{O}$ notation is often used to describe algorithms' time or space complexity. While this simplification is convenient and helps us to compare algorithms easily, it may lead to wrong conclusions as algorithms with worse $\mathcal{O}$ complexity may perform better in the real world than algorithms with better $\mathcal{O}$ complexity.

One reason may be that our theoretical model of computation RAM (Random Access Machine) hides some constants from us. If our only need is to find asymptotic complexity, then hiding constants is not an issue, but in the real world, these hidden costs may cause significant performance problems. For example, random access to memory is much more expensive than sequential read.

Consequently, algorithms with the same time complexity can differ significantly in performance, sometimes by orders of magnitude. For example, IBLT (Invertible Bloom Lookup table) outperforms the naive algorithm using hash table for finding the symmetric difference of two large sets by using memory linear to the size of the symmetric difference. In contrast, the naive algorithm requires linear memory to the size of both sets combined.

2.1.2 Use Cases for Streaming Algorithms

At first glance, it is unclear how strong Streaming Algorithms are and for what purposes they may be used. In this section, we give some examples of such problems and Streaming Algorithms that solve them:

Counting

As humans, we are often used to count on our fingers. The basic strategy of raising one finger for every increase leads to the highest count of 10. This may be significantly improved by using fingers as bits and counting in binary. We can get at most $2^{10} - 1$. Is there any upper bound on the maximum? One should see that for describing 2^{10} states, we need at least 10 bits. If we decided to count only even numbers, we could go higher. This has one little problem: it is not clear how to add one to our count. The solution is to abandon the deterministic algorithm and go for a probabilistic one. We will use s for the true sum and s' for the sum by the returned algorithm.

We can use *Update* Algorithm 3 to increase the count by one. It is not hard to see that the expected value is the same as the true value. More specifically, s' is going to have a binomial distribution. Conversely, it is unclear how close the algorithm estimates are to the sum s . We accept all s' in $[s - \epsilon s, s + \epsilon s]$ as successes. We will use the Chebychev inequality to set bounds on probability. We have to know the variance of random variables to use this inequality. Luckily, it is

Algorithm 3 Update $c : \mathbb{N} \rightarrow \mathbb{N}$

```
1: procedure UPDATE( $c$ )
2:   Let  $s$  be a uniform pick from  $[0, 1]$ 
3:   if  $s > 0.5$  then
4:     return  $c + 1$ 
5:   else
6:     return  $c$ 
7:   end if
8: end procedure
```

Algorithm 4 Query $s : \mathbb{N} \rightarrow \mathbb{N}$

```
1: function QUERY( $s$ )
2:   return  $2s$ 
3: end function
```

common knowledge that binomial distribution has a variance with value $np(p-1)$. Now we just input the value to the Chebychev inequality for $p = \frac{1}{2}$:

$$\Pr[|s' - s| > \epsilon s] = \Pr[|X - pn| > \epsilon np] \leq \frac{np(p-1)}{(\epsilon np)^2} = \frac{(1-p)}{p\epsilon^2 n}$$

So, we can see that the probability of failure decreases as δ or n gets larger. The latter is important as it implies that the algorithm succeeds with high probability. When we set parameters δ and n , so the probability of failure is less than $\frac{1}{2}$, we may use the median trick to get a failure rate of δ , by running $\mathcal{O}(\log(\frac{1}{\delta}))$ instances of the algorithm.

This algorithm may be usable, but it allows us to double a range; we can count on this being probably not worthwhile. Getting the same result would cost us only one bit of memory. So, running multiple instances of this algorithm would essentially be worthless. Also, we are losing accuracy and determinism, which are great costs.

Morris Counter

We modify this algorithm to multiply the value exponentially. We change the meaning of increasing the counter by one from "add two" to "multiply by two." We also change the probability of increasing the counter appropriately.

The algorithm we get is called the Morris Counter. It is regarded as the first true Streaming Algorithm ever discovered. [5] It needs just $\mathcal{O}(\log(\log(n)))$ memory, where n is the highest number counted to.

2.1.3 Number of Distinct Elements in MultiSet

We are going to move to a different problem. Imagine having a very large multiset where some elements may be multiple times, which is given to us as a Data Stream. Our goal is to find the number of different elements. In this subsection, we will show just a sketch.

Algorithm 5 Update $c : \mathbb{N} \rightarrow \mathbb{N}$

```
1: Let  $s$  be a uniform pick from  $[0, 1]$ 
2: if  $s > 2^{-c}$  then
3:   return  $c + 1$ 
4: else
5:   return  $c$ 
6: end if
```

Algorithm 6 Query $s : \mathbb{N} \rightarrow \mathbb{N}$

```
1: return  $2^s$ 
```

We start by cheating. We assign each element a random number from some uniform distribution so that the same elements have the same number. We clearly do not have space for that, but we will solve this issue later.

Now, we find k different smallest random numbers associated with some elements from the elements (we expect the numbers to be linearly ordered). We are going to call this value v . Let n be the true number of distinct elements in the set, and R be the size of the universe from which we pick random numbers. Let v be the largest of the smallest numbers, then our estimate is $n' := \frac{R(k-1)}{v_0}$

We show this algorithm mainly as a showcase for using a hash function as a random number generator. Any hash function may be used as a random number generator. However, we should not forget that the randomness gained this way may not have the best properties. For example by using 3-independant hash function, we may get very close to 3-independant random variables. We mention just close, as 3-independant for hash function is defined only asymptotically.

The hash function may be used to assign random numbers, and we can keep the k -smallest hashes in memory. Now, we should check how good our approximation is. We may see that some distinct values may have the same hashes. This can be easily overcome, but we are not going to, and we just assume that all values have different hashes. We would also assume that the hashes are independent of each other. (These are clearly contradictory, but again, we are not going to care).

It remains to prove that our approximation is good enough. This is not hard, however we omit the proof. We just remark, that Chebychev Inequality is again used.

The algorithm described is called KVM (k -minimum values) and was introduced by Bar-Yossef et al. [6]. This algorithm can be improved, which leads to HyperLogLog being the algorithm used in industry. [7]

We refer interested readers to Small Summaries for Big Data [8] for finding proofs or more streaming algorithms.

2.2 Statement of the Problem

We will return to the example from the introduction. Imagine we have two instances of one database, and we would like to find if they differ, and if they do, then make corrections so they become the same. Also, we would like to exchange as little data between the instances as possible.

We can model this problem by having two large sets A, B such that their symmetric difference is small (the symmetric difference of A, B contains elements that are just in one of the sets). The elements of A and B are given to us as a Data Stream, first we get elements from A , then from B .

Definition 16. *k -SetRecoveryProblem of sets A and B is the problem of finding the symmetric difference of A, B of size at most k , where elements of A and B are provided to us using a data stream, first giving elements of A , then sending a dividing symbol, and then sending all elements of B . If not said otherwise, we would use N for the number of elements in both sets and M for the universe size from which items of the Data Stream are taken.*

We will expect that elements of both sets have some unique binary representation. If we talk about some arithmetic operations over these elements, we are actually doing these operations over their binary representations.

One may imagine that sets contain just integers that are then mapped to the actual elements.

We can solve any k -SetRecoveryProblem in $\mathcal{O}(n)$ space and time, where n is the number of elements in the Data Stream. We build a HashSet, adding all elements from A and then removing all elements from B . This solves our problem, but our space complexity is linear to the size of both sets. The best-known algorithm has $\mathcal{O}(k)$ space complexity, where k is the size of the symmetric difference.

2.3 From Simple solutions to IBLT's

It is not easy to see how to proceed to get an algorithm better than the naive version. In this chapter, we will try to build Invertible Bloom Lookup Tables that solve the problem of the aforementioned space complexity.

2.3.1 Solving 1-SetRecoveryProblem

Let us start by simplifying the problem. We would limit the size of the symmetric difference to size 1.

The solution is not hard. The construction is quite simple; we will remember two numbers C and S :

$$C := |A| - |B|$$

$$S := \sum_{i \in A} i - \sum_{i \in B} i$$

It is trivial to see that there exists a Streaming Algorithm returning values C and S for any sets A and B .

If C is 1, then the symmetric difference is $\{S\}$. If C is -1 , then the symmetric difference is $\{-S\}$, if $C = 0$, then the symmetric difference is empty.

It should be clear that C may be just $-1, 0$, or 1 .

This algorithm may not be the best in terms of memory efficiency, as with S, C may consume $\mathcal{O}(\log(N) + \log(M))$ memory, but this may be improved to $\mathcal{O}(\log(M))$.

We are going to reuse this algorithm in the next section; thus, it would be nice to give it a name.

2.3.2 From 1 to k -SetRecoveryProblem Solution

We now use SumOneFinder to find a solution to the 2-SetRecoveryProblem. The sets may differ by up to two elements. For simplicity, we assume that they differ by exactly two elements, a and b , and that $A \subset B$.

First, we use a hash function $f : A \cup B \rightarrow 0, 1$ to divide all elements into two buckets. If we are lucky and a and b end up in different buckets, we only need to solve two 1-Set Recovery Problems. However, if a and b are hashed into the same bucket, recovery becomes impossible. We only know that there are two elements in such buckets because we keep a count. Unfortunately, the probability of failure is $\frac{1}{2}$, which means we cannot use a median trick. However, by dividing the elements into more parts, we increase the probability that a and b will be in different parts.

We can generalize this approach to solve the k -SetRecoveryProblem. For any k , we need to find an appropriate number l of parts so that the probability of each element being in a different part is greater than $\frac{1}{2}$.

When we have k elements and l parts, the probability that no two elements hash to the same part is:

We see that our problem is equivalent to the Birthday Paradox. It is known that for large k , l needs to be in $\mathcal{O}(k^2)$. This gives us a solution to our problem in $\mathcal{O}(\log(\frac{1}{\delta})k^2)$ with a success probability of $1 - \delta$. Note that the logarithm arises from using a median trick. We call this algorithm, without using the median trick, KSumOneFinder.

This solution has two drawbacks: it uses quadratic space relative to the size of the symmetric difference, and there is a nonzero probability of failure.

The main idea is that by running multiple instances of KSumOneFinder, we can use the results from each instance to improve the others. Suppose we have two instances of KSumOneFinder, G_1 and G_2 , and both fail to recover k elements but recover two sets of elements, K_1 and K_2 . We can then take K_1 and remove these elements from G_2 . If we are fortunate, some unrecovered elements in G_2 will be in the same buckets as those in K_2 , allowing us to recover them. If we recover all elements or none, we stop; otherwise, we use elements in K_2 to recover more elements from G_1 .

Algorithm 7 SymDifference $A, B : (A, B) \rightarrow C$

```

1:  $G1 \leftarrow \text{Encode}(A, B)$ 
2:  $G2 \leftarrow \text{Encode}(A, B)$ 
3:  $K1 \leftarrow \text{Recover}(G1)$ 
4:  $K2 \leftarrow \text{Recover}(G2)$ 
5: Let  $Recovered \leftarrow K1 \cup K2$ 
6: while true do
7:    $G1 \leftarrow \text{Recover}((G1 - Recovered) \cup Recovered)$ 
8:    $G2 \leftarrow \text{Recover}((G2 - Recovered) \cup Recovered)$ 
9:   if no element was newly recovered then
10:    return  $Recovered$ 
11:   end if
12: end while
```

Before we analyze this algorithm, we first show some intuition. To recover an

element we need to match it with some bucket. Every element lies in m buckets, where m is the number of instances run. As such, we may look at buckets as vertices and elements as edges in some m -uniform hypergraphs. We are not able to recover some elements if some 2-core exists. We made a little change to our algorithm; we let our instances of KSumOneFinder share one single set of buckets.

Here, we may call a theorem from preliminaries ?? about the threshold for the appearance of 2-core in 3-uniform hypergraph. As such, we get a high probability that we can recover all elements from the symmetric difference of the two sets A and B if the number of buckets to the size of the symmetric difference is approximately at least $1.222\dots$. This is the best we can get for any k -uniform hypergraph. We do not need to limit ourselves to uniform hypergraphs, as better bounds were found for non-uniform ones.

2.3.3 Invertible Bloom Lookup Table

The algorithm is equivalent to a known data structure called IBLT (Invertible Bloom Lookup Table). This algorithm is described in [9]. This structure may also work as a dictionary, supporting recovery of key-value pairs. There is no use case for this for us; we will describe only a simplified version. Let U be the universe of possible keys. Keys must be a binary representation; we will work just with it and finally be B , the sets of buckets. We start by picking a set of random hash functions $U \rightarrow B$ named H and some helper hash function D .

The buckets are very similar to SumOneFinder: This is practically the same

Algorithm 8 Definition of Bucket

```

1: type Bucket:
2:     SumOfKeys
3:     SumOfHashes
4:     Count

```

as the fields of SumOneFinder, but we added a new field called SumOfHashes. This field will be important for allowing the removal of keys not present in the Sketch.

Now, we describe three possible operations that are done with the IBLT data structure.

Algorithm 9 Add Key : (Value) $\rightarrow$ void

```

1: procedure ADD KEY(Value)
2:   for every hash function  $h$  in  $H$  do
3:      $bucket \leftarrow (h(\text{Key}))$ 
4:      $bucket.\text{SumOfKeys} += \text{Key}$ 
5:      $bucket.\text{SumOfHashes} += D(\text{Key})$ 
6:      $bucket.\text{Count} += 1$ 
7:   end for
8: end procedure

```

The SumOfHashes is used to address a specific problem in hashing where the count of items hashed to a bucket may not accurately reflect the actual number

Algorithm 10 Remove Key : (Value) $\rightarrow$ void

```
1: procedure REMOVE KEY(Value)
2:   for every hash function  $h$  in  $H$  do
3:      $bucket \leftarrow (h(\text{Key}))$ 
4:      $bucket.\text{SumOfKeys} -= \text{Key}$ 
5:      $bucket.\text{SumOfHashes} -= D(\text{Key})$ 
6:      $bucket.\text{Count} -= 1$ 
7:   end for
8: end procedure
```

Algorithm 11 IsPure bucket : Bucket $\rightarrow$ bool

```
1: function ISPURE(bucket)
2:   if  $bucket.\text{Count} \neq 1$  and  $bucket.\text{Count} \neq -1$  then
3:     return false
4:   end if
5:   if  $bucket.\text{SumOfHashes} \times bucket.\text{Count} \neq D(bucket.\text{SumOfValues}) \times$ 
    $bucket.\text{Count}$  then
6:     return false
7:   end if
8:   return true
9: end function
```

Algorithm 12 Recovery : $\rightarrow$ Values

```
1: procedure RECOVERY
2:   repeat
3:     for every bucket do
4:       if ISPURE(bucket) then
5:         if  $bucket.\text{Count} = 1$  then
6:           REMOVE(bucket.SumOfKeys)
7:         else if  $bucket.\text{Count} = -1$  then
8:           ADD(bucket.SumOfKeys)
9:         end if
10:      end if
11:    end for
12:  until some bucket contains non-zero value
13: end procedure
```

of elements present. Without an additional hashing function, D , there's a risk of mistakenly recovering elements that are not part of the symmetric difference. The SumOfHashes helps mitigate this issue by reducing the likelihood of such errors. This reduction in probability is proportional to the size of the universe that D hashes into, providing a more reliable method for identifying the correct elements in the symmetric difference.

2.4 HPW Algorithm

IBLT needs $1.222\dots$ buckets with three main hash functions and one helper function. We are not going to improve that. However, the size of the bucket can be improved. If we take an IBLT bucket, we see that it has three fields. In a trivial implementation using, for example, a 32-bit integer for them, we would actually need three times the memory. This may be improved as Count and SumOfHashes may have smaller sizes. We will not dive deep into these methods as they are out of scope. There is actually an algorithm needing just one field per bucket. This algorithm was discovered by Jakob Bæk Tejs Houen, Rasmus Pagh, Stefan Walzer. [1] As the authors did not name their algorithm for the purpose of this text, we are going to use the abbreviation HPW.

The main idea is simple: instead of the sum of values and counts, we will use the xor of values hashed to the bucket. The interesting property of xor is that it is commutative, and $a \oplus a = 0$. This means that if some element a belongs to both sets A and B , the corresponding Sketch will stay the same.

Algorithm 13 Toggle Value : (Value) $\rightarrow$ void

```

1: procedure TOGGLE VALUE(Value)
2:   for every hash function  $h$  in  $H$  do
3:      $bucket \leftarrow (h(\text{Key}))$ 
4:      $bucket.\text{Value} \oplus = \text{Key}$ 
5:   end for
6: end procedure

```

Algorithm 14 LooksPure bucket : Bucket $\rightarrow$ bool

```

1: function LOOKSPURE(bucket)
2:    $value \leftarrow bucket.\text{Value}$ 
3:   if  $value = \text{null}$  then
4:     return false
5:   end if
6:   for every hash function  $h$  in  $H$  do
7:     if  $(h(value)) = bucket$  then
8:       return true
9:     end if
10:  end for
11:  return false
12: end function

```

Algorithm 15 Recovery : $\rightarrow$ Values

```
1: procedure RECOVERY
2:    $pure \leftarrow \{b \in B \mid \text{LooksPure}(b)\}$ 
3:    $recovered \leftarrow \{\}$ 
4:   while true do
5:      $pureNext \leftarrow \{\}$ 
6:     for each  $b \in pure$  do
7:       if  $recovered.Contains(b.Value)$  then
8:         remove  $b$  from  $recovered$ 
9:       else
10:        add  $b.Value$  to  $recovered$ 
11:        TOGGLE VALUE( $b.Value$ )
12:        for every hash function  $h$  in  $H$  do
13:          add  $(h(b.Value))$  to  $pureNext$ 
14:        end for
15:      end if
16:    end for
17:     $pure \leftarrow pureNext$ 
18:    if  $pure = \{\}$  then
19:      return success if all bucket.Values are null
20:    end if
21:  end while
22: end procedure
```

We will not explain why the algorithm works in depth, but we will try to give some insight into it. We start by imagining an idealized world, where LookPure returns true only when a value in the bucket corresponds to some element of the symmetric difference. In this world, when our algorithm succeeds, it returns all the elements in the symmetric difference and not one more. Sadly, not all dreams may be fulfilled, and we must face the problem of LooksPure not returning a correct answer. It may happen that the xor x of some set of values K that hashes to one bucket looks pure. If this happened during recovery, then the value x would be recovered despite not being an element of the symmetric difference. This does not mean it stays there; it can be removed and probably will as the value was toggled back to the sketch. We are going to call such keys outside keys. What may also happen is that some set of outside keys cancel themselves out, and then we have no chance of removing them. It is possible to show that these issues are unlikely happen. As such, this HPW returns with high-probability the right answer if the number of buckets is large enough.

3 Dotnet and C# Preliminaries

We chose C# as the main programming language. This may seem like a weird choice, as many consider it a slow language. C# is probably a little slower than compiled languages due to being a JITted and garbage collected. This is, however, little price for benefits we get. We do not need to worry about freeing memory, and users do not need to compile the source code for their machines.

One main strength of C# is that the compiler was rewritten with the idea of the compiler-as-a-service (CaaS) in mind. Originally, the compiler was written in C++, but later, Microsoft decided to rewrite it to C#. Now, developers may use the compiler from the code itself. This means compiling code during runtime and then using the newly compiled binaries is possible. Such a possibility is especially useful when working with data unknown during compile-time. Another example is the possibility of writing custom compilation errors. The new compiler is called Roslyn.

We used these features extensively in this project to achieve better performance. For example, we speed up hash functions 2 to 4 times (depending on the machine used) by using runtime-generated code 8.4.2. Mainly due to these optimizations, we achieved about 152 million updates per second while maintaining high code genericity.

We do not expect a common reader to know all the little nuances of C# or DotNet. Thus, we provide readers with an introduction to them. However, we assume familiarity with some C-type language.

3.1 Dotnet and Just-In-Time Compilation

Every programming language needs to be translated into machine code. There are two main approaches: compiling and interpreting. Both approaches have their benefits and drawbacks regarding performance, security, and portability. By portability, we mean the ability to run our software easily on many different machines and architectures.

Compiled languages are translated to the assembly before the program starts. This has two advantages. First, security: we may send our clients just the machine code for their specific machine, keeping the source code private. Second, performance: the processor gets the code already translated and does not need to do any additional work. The compiler can also optimize the code to run faster. The main problem is that not all processors have the same instruction sets, so we have to compile our software for each of them.

Interpreted languages use an interpreter to read their instructions and translate them to machine code during runtime. This costs time. However, any machine equipped with such an interpreter can run the native code.

There is also a third option: JIT (Just-In-Time) compilation, which lies somewhere in the middle. It involves both compiling and interpreting. It first takes our code written in the language and translates it to IL (Intermediate Language). IL contains most of the instructions used by modern processors. However not all machines have those, thus the IL code still needs to be translated to the true machine code instruction. This is done during runtime when our

program first needs the code.

C# is a JIT language. Native code is first translated to CIL (Common Intermediate Language), and then when the program runs, the CLR (Common Language Runtime) does the heavy work of compiling from CIL to machine code during runtime.

3.2 Memory

3.2.1 Heap and Stack

There are two main types of memory: heap and stack. One should take care not to be confused by the name heap, as it has very little to do with the data structure used for repeatedly finding the minimum in logarithmic time.

Stack

Stack and heap work quite differently. Stack functions as we would expect a stack to function: data is pushed to the top and also taken from the top. This is useful for recursive parts of our program and is mainly used by methods. One should easily see that methods are, in their nature, recursive. Methods have basically only two options: call some other method or return. When we call a method, we first save the values held by accumulators (registers used by the processor) so they are free to be used by the called method. These values are saved on the top of the stack. Then, after the called method finishes, the saved data is loaded back. Methods may also put their own data on the stack, but it is important to remember that every piece of data on the stack is not safe to use after the method that placed it there has returned. As the stack pointer is now under these data, any method called later can and probably will overwrite them.

Heap

Sometimes, we want to save data between method calls. The heap is used for this purpose. We request memory of a certain size from the heap, and then we are given such memory that is guaranteed not to be used. When we no longer need this memory, we return it to the heap. This process is called allocating and deallocating memory.

This sounds simple, but there are two problems. The first is that we may run out of memory if we do not deallocate. The second is that when we free some memory, some part of the program may still treat that part of the memory as if it still held original data, but that may or may not be true. These are real problems and may cause dangerous bugs and security issues, especially the latter.

3.2.2 Value and Reference Types

We started with heap and stack for a reason. In C#, there are many different types, but there are only two main kinds of them: value and reference types. They both serve the same purpose of bundling data and operations together; they only differ in the place where they are located in memory. Value types live on the stack and reference types on the heap. As heap and stack differ greatly, value

and reference types behave differently. It is important to know these nuances as they may cause performance issues or unexpected behavior leading to bugs. Also, before we explain the main differences, we take a little detour to terminology. When we are speaking about instances of type, we use term values for instances of value types and objects for instances of reference types.

Value Types

The best example of a value type is an integer or any other numerical type in C#. However, not all numerical types are implemented in C#. For example, complex numbers may be missing or some other. As such, C# provides us with the possibility to define new value types using the struct keyword:

```
struct ComplexNumber{
    public double Real = 0;
    public double Imaginary = 0;
    public ComplexNumber(){
        //Constructor
    }
}
```

The main intuition behind structs is that they are merely syntax sugar for grouping fields together.

```
//This is the same ->
double cReal = 0;
double cImaginary = 0;
//To this ->
var c = new ComplexNumber();
```

This leads to some interesting properties of value types. When we call the method, values are passed by copy into it. This means every single field of the value being passed into the method has to be copied. We can see this as a new instance of such a value type created. We show an example:

```
var x = new ComplexNumber(){Real = 1, Imaginary = 1};
var y = new ComplexNumber(){Real = 1, Imaginary = 1};
var z = Add(x, y);

ComplexNumber Add(ComplexNumber a, ComplexNumber b){
    a.Real += b.Real;
    a.Imaginary += b.Imaginary;
    return a;
}
// z.Real is 2
// x.Real is 1
```

In this example, `z.Real` is 2, but `x.Real` is still 1. This is because the fields were copied; thus, when we added `b.Real` to `a.Real`, `x.Real` was not changed.

This behavior is caused by the fact that values live on the stack. Thus, values' lifespan is limited by the life of the method creating them.

For example, if we passed the value type by a pointer to the place where such a value lies on the stack, we could not use the value safely after the method that created it returned, as it could be overwritten at any time.

As copying the whole value type is required, we may sometimes face bottlenecks when using them, especially when the values copied contain many fields.

Lastly, value types are always initialized. For numerical types, C# guarantees contain 0; if some value type contains a reference type field, it will hold a null value.

Reference Types

We use reference types to model more complex data. They are often useful for modeling non-generic objects with global states, such as persons, database entries, and database connections.

Value types are constrained by the lifespan of the method creating them. This is not an issue for reference types as space on the heap is guaranteed to them until deallocation. As such, copying their fields is unnecessary when passing them to methods. We just need to pass information about where the object lies on the heap. This information is called reference, and thus this process is called argument passing by reference

Similarly to value types, we might want to define our reference types using the keyword `class`. We will use nearly the same example as when discussing value types. We only changed the structure to `class` in the definition of `ComplexNumber`.

```
class ComplexNumber{
    public double Real;
    public double Imaginary;
    public ComplexNumber(){
        //Constructor
    }
}

var x = new ComplexNumber(){Real = 1, Imaginary = 1};
var y = new ComplexNumber(){Real = 1, Imaginary = 1};
var z = Add(x, y);

ComplexNumber Add(ComplexNumber a, ComplexNumber b){
    a.Real += b.Real;
    a.Imaginary += b.Imaginary;
    return a;
}

// z.Real is 2
// x.Real is 2
```

Although the code is the same, the result is not. `z.Real` is still 2, but `x.Real` is now 2. The reason is that `x` and `z` are just references pointing to the same object.

Passing by reference has the advantage that when we pass an object to a method, the only cost is a copy of the reference. One may ask why we should use value types when they are more expensive to pass to methods. The answer is that allocations to the heap may be problematic and come with their own costs.

3.2.3 Garbage Collector

The first section was about the problems with the heap, especially the problem with deallocating memory. Luckily for us, C# is a memory-safe language. This means it solves both problems for us by using a garbage collector (GC). The garbage collector keeps track of all objects allocated by our program.

Remark. This is technically not completely true, as C# allows opting out of the safe context. However, this is something that is used very rarely and only when performance or the need for interop with other languages is required. One seeking such behavior should check `unsafe` context and `Memory<T>`.

When the program needs more memory, it enters the garbage-collecting phase, during which some objects with no reference points are found, and the memory used by them is freed. The word *some* is important, as the process is not deterministic, and we cannot be sure whether or when some unused objects will be collected.

This process is also expensive. This is normally not a problem, but when we allocate a lot in a hot path, it will probably hurt us in terms of performance. (Allocating a new object is also not costless). Also, some garbage collectors need to stop all computations. This may not be feasible in some applications where even short delays are not acceptable, for example, in high-frequency trading.

This problem has only one solution: allocate less. It is possible to achieve this in many ways. Some of the solutions are used quite often. Using values instead of objects is very common; one just needs to consider that passing by copy might be expensive for values having many fields. The next trick in the book is to recycle objects after they will no longer be used. This is usually done via pools, and for some objects that are generally expensive to allocate, we are given an implementation in the base library. Such tools are provided by `ArrayPool<T>` or `ThreadPool<T>`.

3.3 Abstraction, Virtual Methods, Delegates, Interfaces

3.3.1 Inheritance and Virtual Methods

Imagine implementing a probabilistic algorithm. This algorithm has a very low chance of cycling and never ending. We aim to use some heuristics to determine whether it is a good idea to continue computation. A very simple heuristic is to set a maximum number of steps. We can implement the algorithm like this:

```
class ProbAlg {
    //Some fields ...
    int _nRounds = 0;
    int _maxNRounds = 100;

    bool StopCondition(){
        return _nRounds < _maxNRounds;
    }

    void Run(){
        while (StopCondition()){
            //Some computation
            _nRounds += 1;
        }
    }
}
```

Later we might want to implement a different stop condition and compare how the algorithms perform. One option would be to use virtual methods and

inheritance.

```
class ProbAlg {
    //Some fields ...
    int _nRounds = 0;
    int _maxNRounds = 100;

    virtual internal bool StopCondition(){
        return _nRounds < _maxNRounds;
    }

    void Run(){
        while (StopCondition()){
            //Some computation
            _nRounds += 1;
        }
    }
}

class ProbAlgBetterStopCondition : ProbAlg {
    override bool StopCondition(){
        return SuperGoodHeuristic();
    }
}
```

The virtual keyword marks the method as virtual, and the internal keyword marks the method as internal, so they may be used by classes inheriting from ProbAlg but not by any other class. Virtual methods' implementations may be overridden by their children. This is important as, during compile time, the compiler does not know which implementation will be used beforehand. This may cause some performance issues as the compiler cannot perform some of its optimizations.

Inheritance on Value Types

Value types cannot be inherited. This makes sense as values are passed using copies, but children may declare new fields, thus needing more memory than their parents. This is not a problem for reference types as references always have the same size.

3.3.2 Interfaces

Let us say we need to work with groups but do not want to implement a special function for each group type. In this case, interfaces are the way to go.

```
interface IGroupElement {
    IGroupElement Add(IGroupElement a);
    IGroupElement Inverse();
}

struct Z5 : IGroupElement {
    public readonly int Value;
    // Constructor
    public Z5(int value) {
        Value = value;
    }
}
```

```

    IGroupElement Add(IGroupElement a) {
        return new Z5((Value + ((Z5)a).Value) % 5);
    }

    IGroupElement Inverse() {
        return new Z5((5 - Value) % 5);
    }
}

IGroupElement AddAll(List<IGroupElement> elementsToAdd) {
    // Aggregate is C#'s version of fold
    return elementsToAdd.Aggregate((a, b) => a.Add(b));
}

```

‘AddAll’ is going to work for any implementation of ‘IGroupElement’. However, there are some issues. The first is that ‘List<IGroupElement>’ may hold any value or object implementing ‘IGroupElement’. This can result in runtime errors since we haven’t defined behavior for adding elements from two different groups.

There’s also a performance issue regarding using interfaces over value types. ‘List<IGroupElement>’ holds references. This is problematic because value types live on the stack and not on the heap, meaning they have to be moved to the heap, creating a new object. This process, called boxing, goes against the reason for using value types: to minimize heap allocations.

3.3.3 Generics

This problem has a solution called generics. Generics in C# are similar to templates in C++. Let’s look at how to solve this problem using generics.

```

interface IGroupElement<T> {
    T Add(T a);
    T Inverse();
}

struct Z5 : IGroupElement<Z5> {
    public readonly int Value;
    // Constructor
    public Z5(int value) {
        Value = value;
    }

    Z5 Add(Z5 a) {
        return new Z5((Value + a.Value) % 5);
    }

    Z5 Inverse() {
        return new Z5((5 - Value) % 5);
    }
}

T AddAll<T>(List<T> elementsToAdd) where T : IGroupElement<T> {
    // Aggregate is C#'s version of fold
    return elementsToAdd.Aggregate((a, b) => a.Add(b));
}

```

We have now defined a generic version of ‘AddAll’. ‘T’ can be substituted by any type that fulfills all constraints. Currently, we have defined only one constraint: that ‘T’ implements ‘IGroupElement<T>’. Now, ‘List<T>’ can hold only objects or values of one type or their children. This solves the issue of different types in ‘List<IGroupElement>’ causing runtime errors.

Let’s look at how generics behave for value and reference types. For every reference type, there is just one shared CIL code. Every value type has its own CIL code. The code is compiled for the specific type when it is first needed during runtime. This is important because the value type does not need to be boxed. [Generics1]

3.4 Expression Trees

Expression Trees are very useful C# API. They allow us to get duck-typing in C# without needing reflection (reflection is expensive and sometimes it may not be feasible due to its costs). This was especially useful before the addition of generic math. Expression trees allow us to build code during runtime and then compile and execute it. This may be useful when, for example, we would like to optimize our function for some parameters or we just do not know how the data will look; we want as much performance as possible.

3.4.1 A Simple example

If we look at code, we may see it having a tree structure (generally, it is DAG), where nodes are some functions and leaves are constant or parameters. For example:

```
int IfOddAddOne(int input) {  
    if (input % 2 == 1){  
        return input + 1  
    }  
    return input;  
}
```

Might get translated to this:

```
//Constant nodes  
one <- CONSTANT(1)  
two <- CONSTANT(2)  
//Parameter nodes  
input <- PARAMETER()  
//Operator nodes  
moduloByTwo <- MODULO(input, two)  
addOne <- ADD(input, 1)  
isOdd <- EQUALS(one, moduloByTwo)  
ifOddAddOne <- IF_ELSE(isOdd, addOne, input)  
//Function node  
F <- FUNCTION(input, ifOddAddOne)  
//Now we can use F as node  
two <- F(one)
```

C# allows us to build such expressions during runtime and then to compile them and use them as any other delegate. If we wanted an expression tree for the previous code, we could do that with something like this:

```
//This code was partially generated by ChatGpt
using System.Linq.Expressions
//Parameters
var input = Expression.Parameter(typeof(int), "input");
//Constants
var constOne = Expression.Constant(1);
var constTwo = Expression.Constant(2);
// mod = input % 2
var mod = Expression.Modulo(input, constTwo);
// if-else statement
var condition = Expression.Equal(mod, constOne);
var addOne = Expression.Add(input, constOne);
var ifTrue = addOne;
var ifFalse = input;
//Condition
var ifElse = Expression.Condition(condition, ifTrue, ifFalse);
Expression<Func<int, int>> lambda
= Expression.Lambda<Func<int, int>>(ifElse, input);
Func<int, int> compiled = lambda.Compile();
```

One can see that no output has been assigned. This is because, in Expression Trees, the returned value is always the value of the last expression.

3.4.2 Construct-Compile

We work with expression trees in two phases. In the first phase, we construct the tree from smaller expressions or other expression trees. expression trees are immutable, and this construction may be done during runtime. Then we may compile the expression tree to get a lambda. Compiling is an expensive operation, and we should try to reuse compiled lambdas as much as possible, but the code is going to be optimized in the same way the code was written beforehand.

3.4.3 Base API

We are going to explain some of expression trees' basic APIs. We are not planning to cover the topic completely but to give a reader a little overview of the concept. There are several different types of expressions: Constants, Parameters, Scopes, Operators, Callers, and Flow Controllers.

Constants

Constants represent a constants. The value of the constant needs to be known when we compile the expression tree.

```
//Creates a constant expression of type string
var text = Expression.Constant("text");
//Creates a constant expression of type int
var one = Expression.Constant(1);
//We may specify the type explicitly
one = Expression.Constant(1, typeof(int));
//Those fail during the compile phase as "q" has a type string and
//no implicit conversion to int
text = Expression.Constant("q", typeof(int));
```

Parameters

Parameters may be both parameters and variables. They represent an expression to which a value can be written or read; thus, they work as variables in normal C# code. There are no differences between parameters and variables currently, and they may be used interchangeably.

```
//Create a parameter expression with type int
var x = Expression.Parameter(typeof(int));
// We may assign a value
var assignment = Expression.Assign(x, Expression.Constant(1));
//It is important to remember, that assignment represents
//is a expression
```

Scopes

Scopes are important; they group expressions together. There are basically two types of them Block and Lambdas. They both contain a list of parameters and a list of expressions. Expressions are executed sequentially and may use only parameters in their own parameter list or in any other parameter list of Scope transitively containing it. Example:

```
var x = Expression.Parameter(typeof(int));
var assignment = Expression.Assign(x, Expression.Constant(1));

//Scope behavior
//Works correctly
var firstScope = Expression.Block(new {x}, new {assignment});
//May fail during the compile phase as assignment second
//scope does not contain x
var secondScope = Expression.Block(new {}, new {assignment});
//Works correctly the containing scope provides such a variable
var thirdScope = Expression.Block(new {x}, new {secondScope});
//Fails during compile phase as x already is already declared
//in fourthScope
var fourthScope = Expression.Block(new {x}, new {firstScope});

//Lambdas
//It is useful to remember, that the last
//executed expression is also going to be the returned value
var f = Expression.Lambda<Func<int>>(firstScope);
//This returns the function always returns 1
Func<int> F = f.Compile();
var f2 = Expression.Lambda<Func<int,int>>(secondScope, x);
/////This returns function that always returns 1
Func<int, int> F2 = f2.Compile();
```

Operators

Operators allow us to work with all C# operators like =, ==, +, -, « etc. For example:

```
var x = Expression.Parameter(typeof(int));
var one = Expression.Constant(1);
var two = Expression.Constant(2);
```

```
// x = 1
var assign = Expression.Assign(x, one);
//x += 1
var addAssign = Expression.AddAssign(x, one);
// 1 * 2
var multiply = Expression.Multiply(one, two);
// (1 * 2) + 2
var add = Expression.Add(multiply, two);
// ((1 * 2) + 2) - x
var minus = Expression.Minus(multiply, x);
```

Callers

Sometimes, we need to work with classes or structures. They may have some fields, properties, or methods. Then, we may use expressions like `Call`, `Field`, or `Property` to call a method or get some value. It should be noted that in C#, properties might be both methods and fields. This may lead to some bugs if one is unaware of this behavior. One can use the `PropertyOrField` expression to solve this.

```
//Calls parameterless constructor
var list = Expression.New(typeof(List<int>));
//Call a method with name Add -> a.Add(1)
var listAdd = Expression.Call(list, new {int}, "Add", 1);
//We can get a value from property or field by using PropertyOrField
var listCount = Expression.PropertyOrField(list, "Count");
```

Flow Controls

Controlling flow is the basis of our programming. As it allows us to use expressions like `if`, `then`, `loop`, and `goto`. We show an example of a simple loop.

```
var i = Expression.Parameter(int);
var zeroToI = Expression.Parameter(0);
//Assign 0 to i
var condition = Expression.LessThan(i, Expression.Constant(10));
//Add one to i
var actionToDo = Expression.AddAssign(i, Expression.Constant(1));
var breakLabel = Expression.Label("LoopBreak");
Expression loop = Expression.Loop(
    Expression.IfThenElse(
        condition,
        actionToDo,
        Expression.Break(breakLabel)
    ),
    //Defines break label to jump to
    breakLabel
);

var scope = Expression.Scope(new {i}, new {zeroToI, loop });
```

This is equivalent to the following code:

```
int i = 0;
while(true){
    if(i < 10) break;
```

```

        else i += 1;
    }

```

3.4.4 Issues with Expression Trees

One very visible issue with them is that they contain a lot of boilerplate code and are not very readable. If we look at the two previous examples, the latter is clearly more readable. This is a recurring problem with expression trees. Also, the previous example is part of the project's codebase, but it took many iterations to get it right. The native C# version was written in seconds with no bugs. This is not only a boilerplate problem but also a problem of no or very low static checking. This means that our code is mainly going to fail during runtime. We give an example of some possible issues that may lead to runtime errors.

```

using System.Linq.Expressions
//Parameters
var input = Expression.Parameter(typeof(int), "input");
//Constants
var constOne = Expression.Constant(1);
var constA = Expression.Constant('A');
var add = Expression.Add(constOne, constA);

Expression<Func<int, int>> lambda
= Expression.Lambda<Func<int>>(add);

lambda.Compile() //Fails there

// but x = 1 + 'A' does not compile

```

One very recurring error is forgetting to increment a value in a while loop, looking like this:

```

int i = 0;
while(i < 10){
    //Do something
    i++;
    //Easily forgotten
}

```

Normally, we get a warning that some of our code is unreachable, as we will be forever stuck in the loop. It is, however, not possible to do our code generated using Expression Tree, which exists only in runtime when it is too late to fix such a problem.

3.5 Compiler Optimizations

As C# has a compiler, one should be aware of some optimizations and potential performance bottlenecks.

Cost of Method Call

Calling methods is not free. We need to do some preparations, mainly saving values from accumulators to the stack and then loading them back. This cost is very small but can add up if the method body is small. For example:

```

int AddTwo(int x) {
    return AddOne(AddOne(x));
}

int AddOne(int x) {
    return x + 1;
}

```

Inlining

The compiler can solve this problem by inlining. It may replace the method call with its body, saving the cost of the method call. So we get something like this:

```

int AddTwo(int x) {
    return x + 2;
}

int AddOne(int x) {
    return x + 1;
}

```

Inlining is very powerful as it allows us to write very small methods without losing performance. However, not all methods can be inlined. A method can only be inlined when the compiler knows which method is going to be called.

Simple Example

```

List<int> FilterOdd(List<int> input) {
    var answer = new List<int>();
    foreach(var number in input) {
        if (IsOdd(number)) answer.Add(number);
    }
    return answer;
}

```

This code works, but what if we now need to implement a filter that filters prime numbers? We might modify the preceding code:

```

List<int> FilterPrime(List<int> input) {
    var answer = new List<int>();
    foreach(var number in input) {
        if (IsPrime(number)) answer.Add(number);
    }
    return answer;
}

```

If we look closely, we see that the code of ‘FilterPrime’ and ‘FilterOdd’ are very similar. Therefore, it would be nice to abstract the behavior of the filter. We can do something like this:

Solution Using Delegates

```

List<int> Filter(
    List<int> input,
    Func<int, bool> filterCondition
)

```

```

) {
    var answer = new List<int>();
    foreach(var number in input) {
        if (!filterCondition(number)) continue;
        answer.Add(number);
    }
    return answer;
}

```

Now, ‘filterCondition’ may still be a very short method, but the compiler does not know it at the time of compilation, so it cannot be inlined. Similarly, this would happen if we were using an interface.

Solution Using Interface

```

interface IFilter {
    bool Invoke(int number);
}

List<int> Filter(
    List<int> input,
    IFilter filterCondition
) {
    var answer = new List<int>();
    foreach(var number in input) {
        if (!filterCondition.Invoke(number)) continue;
        answer.Add(number);
    }
    return answer;
}

```

There is a solution to not inlining. We’ll first show the code and then explain why it works.

Solution Using Generic Over Struct Interface Method

```

interface IFilter {
    bool Invoke(int number);
}

List<int> Filter<Filter>(
    List<int> input,
    Filter filterCondition
) where Filter : struct, IFilter {
    var answer = new List<int>();
    foreach(var number in input) {
        if (!filterCondition.Invoke(number)) continue;
        answer.Add(number);
    }
    return answer;
}

```

We used a struct constraint to enforce that ‘Filter’ is a value type. This means we get value behavior on generics, thus generating unique CIL code for every type of filter. Also, since ‘filter’ is a struct, it cannot have children, so there is no way to override the ‘Invoke’ method. This means the compiler knows there is only one possible implementation of such a method, thus it can inline it.

3.5.1 When the Compiler Inlines

The .NET compiler uses heuristics to determine whether to inline and where to inline, as inlining may not always be beneficial. There is no way to enforce inlining in C#, but we can give hints to the compiler. This is done using attributes. One can use `[MethodImpl(MethodImplOptions.AggressiveInlining)]` to increase the chance of a method being inlined and to forbid inlining for that method: `[MethodImpl(MethodImplOptions.NoInlining)]` .

3.6 Constant Optimization

When we use constants, the compiler may optimize their use. For example:

```
int AddTwo(int a) {  
    return a + 1 + 1;  
}
```

Is optimized to:

```
int AddTwo(int a) {  
    return a + 2;  
}
```

This saves us some unnecessary calculations that would otherwise need to be repeated. Modern compilers are quite advanced and can replace some expensive arithmetic operations, such as division and modulo, with less expensive ones, like multiplication. This can result in a significant performance boost.

Sometimes, we do not know the value to divide with beforehand, but we would still like to benefit from compiler optimizations, as we expect to do many divisions with the same divisor. We know only one method to achieve such functionality without too many obstacles. It is expression trees. One could also try source generation or `System.Reflection.Emit` namespace to emit CIL during runtime. However, we are not going to describe those techniques. As we decided not to use them.

4 LittleSharp

4.1 Motivation

We used expression trees extensively in the project, but they have a few issues. The first and foremost is that the native Expression Tree library is not statically checked. This leads to many hard-to-debug errors. Also, code written in Expression Trees is hard to read, making debugging such code much more complex. As such, we decided to create a wrapper around Expression Trees that would provide some static checking but not at the cost of versatility. We created a standalone library called LittleSharp. Even though its main purpose was to serve the needs of our project, we think it may be useful for others. However, it is mainly limited to operations with numeric types. The library is at GitHub: <https://github.com/vojtechgadurek/LittleSharp>.

4.2 How to use

In this section, we introduce the library briefly. First, we will write a short function that declares c with value 5, adds a, b , and c , and returns the sum.

```
// Declare a function with two parameters, a and b
var f = CompiledFunctions
    //Create<TIn1, ... , TInn, TOut>(out var input1_, ...)
    //creates a function with arguments with types TIn1 to TInn
    //and return type TOut
    //when one wants to create action,
    //they should use CompiledActions class
    .Create<int, int, int>(out var a_, out var b_);
// a and b have types Variable<int>
// f.S is a scope that is going to be executed when f is called
// S is an abbreviation for Scope
// c = 5;
f.S.DeclareVariable<int>(out var c_, 5)
// c have also type Variable<int>
// return a + b + c
    .Assign(f.Output, a.V + b.V + c.V);
// f.Output is the value that is going to be returned
// .Assign(a, b) is the same as a = b;
// a.V is equal to SmartExpression<T>, V is abbreviation for Value
Expression<Func<int, int, int>> addExpression = f.Construct();
Func<int, int, int> add = addExpression.Compile();
//By calling Compile, the expression tree is then compiled
```

We may also use flow control statements, like while and if. Imagine having a buffer B and some number N to determine the number of values in the buffer. We would like to sum all values together, which can be done as follows:

```
// Declare a function with two parameters, a and b
var f = CompiledFunction
    .Create<int[], int, int>(out var B_, out var N_);
f.S.DeclareVariable(out var i_, 0)
    //We declared variable i with value 0 of type int
    //It is possible to declare variable without explicit
```

```

//initialization, however this is not recommended as
//it may lead to unexpected behaviour
.While(
    i_.V < N.V,
    new Scope()
        .DeclareVariable(
            out var value,
            // By calling ToTable<int>(), we hint compiler
            // that the underlying type is a table,
            // thus we may use [] syntax to access fields
            // B_.V.ToTable<int>() is going to have
            // type ArrayAccess<int, int[]>
            B_.V.ToTable<int>()[i_.V]
        )
        .Assign(f.Output, f.Output.V + value.V)
        .Assign(i_, i_.V + 1)

Func<int[], int, int> F = f.Construct().Compile()

```

This code is equivalent to following:

```

// Declare a function with two parameters a and b

var F = (int[] B, int N) => {
    int i = 0;
    int output = 0;
    while(i < N){
        int value = B[i];
        output += value;
    }
    return output;
}

```

4.2.1 Naming Conventions

As we introduced some new concepts, we also introduced naming conventions for them. Variables should end with underscore `_`; when the variable is converted to some special type, it should end with `__NAME` or its abbreviation. For example:

```

var a = CompiledAction.Create<int[]>(out var table_)
a.S.Macro(var out table_T, table_.V.ToTable<int>());

```

4.2.2 Debugging

Debugging expression trees is hard. There are Visual Studio extensions for that. Two libraries can be used to visualize them: Readable Expressions and Expression Tree Visualizer. **[ExpressDebug]** So, one see the actually generated code. Sadly, Readable Expression does not work with LittleSharp. One can also use some tools provided by the library itself. For example, it is possible to use `Print()`, to log some data. For example:

```

// Logger has to implement method WriteLine(string line)
class Logger{
    public List<string> Data = new ();
    void WriteLine{string line}{

```

```

        Data.Add(line);
    }
}

var logger = new Logger();

//This is going to change a logger for all instances of Scopes
// As such it should be used only for debugging
// When DebugOutput is not set,
//than it prints to standard output
Scope.SetDebugOutput(logger);

var a = CompiledAction.Create()
a.S.Print("Start")
    .Declare(out var v_, 0)
    .Print(v_.ToStringExpression())
    //It is important not call ToString() as
    //then we would call ToString on v_, what is Variable<int>
    // not the underlying type

a.Construct().Compile()()
// Logger.Data contains "Start" and "0"

```

4.3 Implemenation details

4.3.1 Architecture

There are three main types: Variables, SmartExpression, Scopes

Variables

Variable wraps two possible expressions: ParameterExpression and Variable Expression. This means a Variable should be seen as an expression that holds a value, and a value might be assigned to it. There is a generic version of Variable, and this is Variable<T>. This should be preferred to use as it contains information about the type of variable that is held by Variable<T>.

SmartExpression

SmartExpression wraps all expressions. As Expressions are the main building block of Expression Trees, SmartExpression is going to fit this function also as a main building block. SmartExpression also has its generic version, SmartExpression<T>. This generic version also holds an expression of type T.

Scope

Scopes are the most interesting and complex of the types provided. They hold together SmartExpressions and Variables. One should imagine as a normal scope in C#, would look like. They should make it easier not to use variables from different scopes. A generic version, Scope<T>, also has a return type of T.

4.3.2 SmartExpression operators

As our goal is to provide safe arithmetic operations, we simplified SmartExpression to hold some binary numeric type. As such, we overloaded nearly all binary operators to work on the same types. We show an implementation of an overload for addition.

```
public class SmartExpression<TValue>{
    public readonly Expression Expression;
    //More stuff
    public static SmartExpression<TValue> operator+
    (
        SmartExpression<TValue> a,
        SmartExpression<TValue> b
    )
    {
        //Expression.Add( ome_expression, another_expression)
        //is from expression trees API
        return new SmartExpression<TValue>(
            Expression.Add(a.Expression, b.Expression)
        );
    }
    //More stuff
}

//This allows us to do this
SmartExpression<int> a = 1;
SmartExpression<int> b = 2;
SmartExpression<int> result = a + b;
//In the language of expression trees,
//this can be translated to the following
var a = Expression.Constant(1);
var b = Expression.Constant(2);
var result = Expression.Add(a, b);
```

It may not be clear why the latter approach is so bad that developing a new library is well motivated. The problem becomes visible with more complex code. The more expressions in the code, the more problems. The problems are mainly with variables, as one needs to make sure they all have the right types and are all initiated. Our library solves many of these problems. Typing gives more safety and partially solves problems with assigning values of bad types and then having to deal with runtime errors. For example:

```
var S = new Scope();
S.DeclareVariable(a_, 0);
S.DeclareVariable(b_, 1);
S.DeclareVariable(c_, "a");
S.DeclareVariable(d_, 01);
// d_ has SmartExpression<long> type

a_ + b_; //Compile error
a_.V + b_.V; // Compiles
a_.V + c_.V; // Compile error as + is overloaded only for
//SmartExpression<T>s with the same T
a_.V + d.V; // Compile error for the same reason
a_.V.Convert<long>() + d.V; // Compiles
//This could be translated to (long) a + d

a_.V.ForceType<long>() + d.V;
```

```
// Compiles, fails during runtime
// ForceType cast only SmartExpression<T> to different type,
// but not the underlying expression
a_.Assign(S, b_.V); //Compiles
a_.Assign(S, d_.V); //Does not compile as to Variable<T>
// may be only assigned value with type T
```

This behavior provides enough safety to catch most errors, especially bad assignments.

4.3.3 Scopes

Scopes are more interesting. They hold both variables and expressions to be executed. We may imagine that they hold a list of expressions and a list of variables. Expressions are executed in the order in which they were added to the scope. They are not executed when we do not add an expression to the scope. All variables are declared before all expressions. When declaring a variable with some assigned value, one should not forget that this assigned value is, in reality, an expression. This means we are doing two things: declaring a variable and assigning a value. Example:

```
var S = new Scope();
S
    .DeclareVariable(a_, 0)
    .Macro(out var x_, a_ + 10)
    .DeclareVariable(b_)
    .Function( (x) => x + 1, x_, out var z_)
    //The function is never executed
    //as z_ is then never used.
    .Function( (x) => x + 13, x_, out var y_)
    .Assign(b_, y_)

//First we declared variables a_ and b_
//Then a_ = 0
//Then b_ = 13
```

Scopes may be contained in other scopes and may also contain them. Variables may be used only from the scope they are declared in or any scope contained in that scope. The property being contained is transitive. This behavior is the same as expected from normal scopes in C#.

Scopes may or may not have a return value.

4.4 Future work

This library is mostly experimental and serves more as a glimpse into a concept than a fully functional library. This makes sense as it was a side project created out of a need for better API for Expression Trees.

There is some technical debt in architecture. The main problem is that when a Scope cannot be added as a Scope to another Scope, it needs to be made a SmartExpression first, which is problematic as it makes analyzing such code much harder. The solution to this problem is decoupling compilation from expression trees via visitor pattern.

LittleSharp would benefit enormously from using Roslyn more extensively, especially source generators and analyzers, to build a less error-prone experience. Analyzers could be used to write custom compile errors, thus helping users debug code written using SmartExpression better. Code Generators could be used to create a more advanced type system supporting custom types.

5 FlashHash

5.1 Motivation

The project required access to families of fast hash functions. Sadly, no libraries were offering such. The base C# library provides only cryptographic hashing functions. Other libraries offered hashing only from byte arrays. This meant we had to convert integer values to byte arrays, which caused performance bottlenecks. This led to the decision to implement, such a library ourselves. The library is at GitHub: <https://github.com/vojtechgadurek/FlashHash>.

5.2 How to Use

FlashHash implements multiple hash function families - Linear Congruence, Polynomial, Tabulation, and Multiply Shift. There are two important concepts to understand: (Hash Function) Schemes and (Hash Function) Families.

Schemes

Schemes describe one hash function. One may build a hash function out of it:

```
var scheme = new LinearCongruenceScheme(10, 5, 7)
// This scheme yields the following hash function:
// (10*x + 5) % Prime % 7
// Prime represents the closest Mersene prime
// greater than the size (7)

Expression<Func<ulong, ulong>> expression = scheme.Create()
// Building an expression tree is not computationally expensive

Func<ulong, ulong> hashFunction = expression.Compile();
// However, the compilation is resource-intensive
```

Families

Families are merely factories of schemes:

```
var family = new LinearCongruenceFamily();
family.SetRandomness(new Random(42));
// Custom randomness can be set
ulong size = 10;
Func<ulong, ulong> hashFunction =
    family.GetScheme(size).Create().Compile()
// Retrieves a random hashing function
// from the linear congruence family
```

5.3 Implementation

5.3.1 Simplest Solution

There is a very simple solution:

```

var CreateLinearCongruenceHashingFunction =
    (long size) =>
    {
        ulong a = Random.GetRandomUInt64();
        ulong b = Random.GetRandomUInt64();
        ulong p = Random.GetGoodPrime(size);
        return (ulong x) => (a * x + b) % size % prime;
    }

```

This approach has one very large advantage: it is simple. On the other hand, we pay for this simplicity by performance, as constant optimization and inlining are not possible. One can easily see that linear congruence uses modulo operators. These are quite expensive, and even more so if they are not constant optimized. This implementation is circa 2 – 4 times slower, depending on the processor, than when `a`, `b`, `size`, and `prime` were known in advance.

There are two possible solutions to this in C#: Expression Trees and Runtime Types with Struct Delegates. We talked about these in Chapter 5 d. We chose a solution via expression trees, but to interested readers, we first describe the solution by Runtime Types with Struct Delegates.

5.3.2 Possible solution via Runtime Types with Struct Delegates

First, we have this implementation:

```

//For sharplab
using SharpLab.Runtime;

interface IConstant{
    ulong Get();
}

//Constants are going to look something like this
struct One : IConstant
{
    public ulong Get() => 1L;
}

struct Six : IConstant
{
    public ulong Get() => 6;
}

struct Prime : IConstant
{
    public ulong Get() => 11;
}

//This is just for sharplab
[JitGeneric(typeof(One), typeof(Six), typeof(Prime),typeof(Six)) ]
struct HashingFunction<TA, TB, TSIZE, TPRIME>
where TA : struct, IConstant
where TB : struct, IConstant
where TSIZE : struct, IConstant
where TPRIME : struct, IConstant
{
    public HashingFunction(){

```

```

readonly TA a = default(TA);
readonly TB b = default(TB);
readonly TSIZE size = default(TSIZE);
readonly TPRIME prime = default(TPRIME);
public ulong Invoke(ulong value)
    =>
    (a.Get() * value + b.Get())% prime.Get() %size.Get();
}

// Then hf.Invoke = (x) => (1*x + 6) % 11 % 6;
// But compiler knows this during compilation

```

This works. If we look at the generated IL code, we see that all values got inlined:

There is one small problem: We need access to families of hashing functions, but we cannot implement a constant type for every number. The solution to this is IL code and runtime types. It is possible in C# to create types during runtime, and then we may use reflection to build hashing functions out of the created constant. Implementation of the Linear Congruence family would thus look like this:

```

static class ConstantFactory(){
    Type GetConstant(ulong value){
        //Magic
    }
}

class LinearCongruenceFamily {
    IHashingFunction GetHashingFunction(size){
        //Prepare
        ulong a = Random.GetRandomUInt64();
        ulong b = Random.GetRandomUInt64();
        ulong prime = GetGoodPrime(size);
        Type A = ConstantFactory.Get(a);
        Type B = ConstantFactory.Get(b);
        Type PRIME = ConstantFactory.Get(prime);
        Type SIZE = ConstantFactory.Get(SIZE);
        //Reflection
        var maker = typeof(LinearCongruence<>).MakeGenericType(
            new {A,B, PRIME, SIZE}
        );
        return (IHashingFunction) Activator.CreateInstance(maker);
    }
}

```

Constant Factory

The implementation of such a library function may look something like this:

```

public static Type CreateRuntimeConstant(long value)
{
    //Assembly to which we add the type
    AssemblyName assemblyName = new AssemblyName("DynamicAssembly");
    AssemblyBuilder assemblyBuilder =
        AssemblyBuilder.DefineDynamicAssembly(assemblyName, AssemblyBuilderAccess.RunAndSave);
    ModuleBuilder moduleBuilder =
        assemblyBuilder.DefineDynamicModule("DynamicModule");
}

```

```

//Type builder with struct semantics
TypeBuilder tb =
    moduleBuilder.DefineType(
        $"long_{value}",
        TypeAttributes.Public | TypeAttributes.Sealed |
        TypeAttributes.SequentialLayout
        | TypeAttributes.BeforeFieldInit,
        typeof(ValueType),
        new Type[] { typeof(IConstant) }
    );

    var method = tb.DefineMethod("GetValue",
        MethodAttributes.Public |
        MethodAttributes.Virtual |
        MethodAttributes.HideBySig, typeof(long), Type.EmptyTypes);
    var il = method.GetILGenerator();
    il.Emit(OpCodes.Ldc_I8, value);
    il.Emit(OpCodes.Ret);
    tb.DefineMethodOverride(
        method,
        typeof(IConstant).GetMethod("GetValue")
    );
    return tb.CreateType();
}

```

This solution has a small problem. It does not work, and we do not know why. We suppose Roslyn treats Runtime-generated types differently, but we have not investigated further.

5.3.3 Solution using Expression Trees (LittleSharp)

We are going to use LittleSharp for work with expression trees. Reasons for developing this library are given in the chapter about LittleSharp. See Chapter 4.

Using LittleSharp the simplest solution looks like this:

```

var CreateLinearCongruenceHashingFunction =
    (long size) =>
    {
        ulong a = Random.GetRandomUInt64();
        ulong b = Random.GetRandomUInt64();
        ulong p = Random.GetGoodPrime(size);
        var f = CompiledFunctions.Create<ulong, ulong>
            (out var input_, out var hash_);
        f.S.Assign(f.Output, (a * input_ + b) % size % prime);
        return f.Construct().Compile();
    }
//Returns Func<ulong, ulong>
//This code is constant optimized

```

5.3.4 Comparing Approaches

Both approaches add significant complexity to the project. The main issue is that switching the context between runtime-generated and normal code is not smooth. One needs to use reflection calls in case of IL-generated code or delegates calls for Expression Trees. Also, using runtime-generated code will cause type

safety issues. This is because types cannot be checked before the code is generated during runtime. This may cause many hard-to-find bugs that may cause runtime errors.

Both approaches are comparable. They lead to constant optimized code and allow inlining with additional work. Expression Trees are easier to work with because their API is simpler and requires much less reflection code. Some type of safety can also be added; this is discussed in the Chapter 4. Finally, expression trees are probably used more by the community, which means there are more sources to learn about them.

Secondly, and most importantly, we are unsure whether we could make the approach with struct delegates work. We could possibly do some hacks, maybe using a source generation or modifying Roslyn itself. Considering this, we have chosen the approach using Expression Trees.

5.4 Future work

We see the library as very complete, but it could provide more hash function families. It could also possibly provide a possibility to hash from any integer type to any integer type. This would be very simple to implement. Some sort of caching for the compiled expression tree hashing function could be useful.

It would probably be interesting to investigate why exactly struct delegates approach does not work. It is possible, that some optimization is not turned off for runtime types; however, we are not sure. This was pointed to us by Mgr. Radek Zikmund.

6 RedaFasta

6.1 Motivation

Our goal is to write a reader of Masked Supestrings files that does not cause bottlenecks to our main algorithm. As such, it needs to be reasonably fast. We are willing to sacrifice genericity for performance. The library is on GitHub <https://github.com/vojtechgadurek/RedaFasta>.

6.2 How to use

The library provides two classes, `FastaFile` and `FastaFileReader`. `FastaFile` serves to initialize and recover configurations from FASTA files. `FastaFileReader` is the main class doing all reading, buffering, and parsing kmers.

Canonical Order

Reader returns canonical kMers.

Opening a `FastaFileReader`

The first line of the fasta file is expected to be a configuration. There must be two parameters `l` and `k`. `l` is the number of chars in sequence, and `k` is the number of chars in a kmer. then it is expected to be a line with all chars in sequence. The sequence should contain only A,C,G,T or a, c, t, g. The reader does not check this; different symbols may be treated as any char. A reader reads only `l` characters from the line.

```
//Open fasta file
string path = "file.fasta"
//Open TextReader
TextReader tr = new StringReader(path);
//Get configuration
var config = FastaFile.Open(tr);
//Open reader
var fastaReader =
    new FastaFileReader(
        config.KMerSize,
        config.nCharsInFile,
        tr
    );
```

Users may use `FastaFileReader` in two ways: using its own buffer or borrowing and then returning a buffer. Both methods have advantages and disadvantages. It is not recommended to mix both methods, as this may lead to highly underfilled buffers. `FastaFileReader` does not guarantee that KMer is read in order.

Using own buffer

This method always tries to fill the buffer as much as possible. This may be beneficial if we need more consistent counts of numbers in the buffer. This is,

however, at the cost of an additional copy, causing it to be less performant, than borrowing

```
FastaFileReader fastaReader;  
//We may use our own buffer and let it fill  
var buffer = new ulong[1024];  
int nKMerInBuffer = fastaReader.FillBuffer(buffer);  
//0 means there is no KMer to be read and the stream ended
```

Borrowing a buffer

This buffer returns some buffer. Buffer has three fields: Data, Size, and Used. Size the number of KMer in the buffer; Used is the number of KMer already used. This means when Used has value N, the Data[N] is the first valid value. Used should be 0, but using the method FillBuffer() may cause some buffers to have Used fields with different values

```
FastaFileReader fastaReader;  
//We may borrow a buffer from the reader  
var borrowedBuffer = fastaReader.Borrow();  
//We then should return the buffer back to the fasta reader  
//If do not return a buffer,  
//fasta file reader may be forced to allocate a new one.  
//This may cause performance issues.  
if (borrowedBuffer is not null)  
    fastaReader.Recycle(borrowedBuffer);
```

6.2.1 Disposing

FastaFileReader does not dispose of the TextReader it was provided with. The user should do this themselves by calling the Dispose() method.

6.3 Future work

This library has a very specific purpose, and it fits that purpose well. It could be improved by adding a possibility to check whether files are in the correct format.

7 SymmetricDifferenceFinder

7.1 Motivation

We aimed to develop a library to test different approaches for solving set recovery problems, especially finding symmetric differences between two sets. We decided we would like both performance and modifiability, even at the cost of speed and ease of development. Performance is essential because the expected use case may deal with massive datasets. We need modifiability to be able to change or replace components easily. The library may be found here <https://github.com/vojtechgadurek/SymmetricDifferenceFinder>.

7.2 Overview

SymmetricDifferenceFinder is our project's core library. It is a small framework for building solutions to the set recovery problem. We will now do a little tour to introduce potential users to the library. The most important concept is a Sketch. One should imagine a sketch as some magic set and some operations with them. These operations may be Toggle(key) (adds the key if the key is not in the sketch; otherwise, removes the key) or a combination of Add(key) and Remove(key).

When we add some value to the sketch, we should imagine this value being there. However, we do not have access to it. If we remove it, it stops being in the Sketch.

Our implementation of Sketches is a little more low-level. We are I will assume this magical set is a set of buckets, and we may access these buckets by keys. This means the general interface is Toggle(index, key).

For some purposes, it may be useful to implement just the *Add(index, key)*; we call those Tables.

For example Tables implement this interface:

```
public interface ITable
{
    void Add(Hash key, Key value);
    int Size();
}
```

Sketches should also implement a method for finding symmetric differences between the items they hold and other sketches of the same type. This is important as not all sketches may be compatible.

7.2.1 Encoding-Decoding Pipeline

There are two main phases: encoding and decoding. During encoding, we encode the values to the sketches. There should be one sketch for each set. During decoding, we first take sketches of sets we want to find symmetric differences, merge them, and then run a decoding algorithm on the result of merging. This step may fail. If it fails, we then need to change the parameters (mostly the size of the table or use some better decoding method if available for the data we have) and rerun the algorithm from the start.

7.2.2 Hash Functions

Hash functions are used in both phases. One must use the same set hashing function for encoding and decoding. This set is expected to implement `IEnumerable<Expression<Func<ulong, ulong>>` interface. This may, for example, be a `List<Expression<Func<ulong, ulong>>`. Order is not important. Some basic hashing functions are provided by the `FlashHash` library. One can also implement their own.

7.2.3 Encoder

Encoders are used in the encoding phase. There are two implementations of Encoders: `Encoder<TTable>` and `NoConflictEncoder<TTable>`. `TTable` has to implement an `ITable` interface and be a value type (this is to get inline). `NoConflictEncoders` expect that no two hashing functions that have intersecting ranges exist. There are two possible interfaces for implementing `IEncoder` and `IParallelEncoder`.

7.2.4 Decoders

Decoders are used in the decoding phase. They should be given a sketch and then return decoded values and information on whether decoding was successful. There are two possible decoders: `HPWDecoder`, based on the HWP algorithm, and `HyperGraphDecoder`, which uses standard hypergraph decoding. It should be noted that these two decoders may not be used interchangeably because they need different contracts from `Sketch`.

7.2.5 HPWDecoder

`HPWDecoder` is based on the HPW algorithm. It allows the removal of items that are not present in the set or that are not going to be preset after the encoding ends. Due to its nature, it cannot distinguish whether some element in the symmetric difference of sets A and B was an element of A or an element of B .

7.2.6 HyperGraphDecoder

It uses a [algorithm] to find the largest 2 core of the hypergraph. In its base version, it cannot remove items that are not present in the sketch or that will be present. We also implemented an improved version of such an algorithm that supports such operations.

7.2.7 Decoders with oracles

Decoders with oracles are useful if one can guess the existence of some other keys in a sketch from the already decoded ones. Our algorithm expects that such data form an ordered graph. Currently, these methods are provided only for `HPWDecoder`, and such an oracle needs to conform to `IStringFactory` and be a structure. They are especially useful when only a few items remain undecoded.

Their effectiveness depends largely on how good our guesses are. However, they are currently more of an experimental feature with a very limited API.

7.2.8 Data sources

The library implements multiple possible data sources. RedaFasta may be used to provide a MaskedSuperString file [4] as a data source and multiple random data generators are available for testing purposes.

7.3 Building a Custom Solution

This section will describe how to use this library to create a custom solution to set recovery problems. For example, we would like to improve a solution using HyperGraphDecoder using a space-saving dictionary.

7.3.1 Creating a table

A table needs to conform to the ITable interface interface. We are going to implement a simpler version of the table. We expect that items will only be removed from tSketchch during the decoding phase. This is true if we have two sets A and B and $B \subseteq A$.

```
//Tables need to be structs
struct IBLTableImprovedMemory: ITable
{
    ulong[] _keySums;
    SpaceSavingDictionary _counts;
    public IBLTableImprovedMemory(int size){
        _data = ulong[size];
        _counts = new SpaceSavingDictionary(size);
        _hashSum = new SpaceSavingDictionary(size);
    }
    // Hash is allias for ulong
    // Key is allias for ulong
    // This means writing ulong or Hash or Key is equivalent
    void Add(Hash key, Key value){
        _keySums[key] += value;
        _counts.Add(1);
    }
}
```

7.3.2 Choosing Hash Functions

After choosing a table or writing its implementation, one should select a set of hashing functions. This set should implement an IEnumerable<T> interface, where T is Expression<Func<ulong, ulong>>. The hash function range should not exceed the table size; otherwise, the program fails during runtime.

```
using FlashHash;
//Simplest solution is this
ulong size = 10;
//This creates an array holding one hash function using modulo
//This approach has a problem with performance
```

```

//One should rather FlashHash library
Expression<Func<ulong, ulong>>[] hfs = [(x) => x % size];
//Solution using FlashHash library
//This solution gets us two Linear Congruence hash function
//with not conflicting ranges
hfs=
[
    HashingFunctionProvider
        .Get(typeof(LinearCongruenceFamily), size / 2, 0).Create(),
    HashingFunctionProvider
        //5 sets an offset to the hash function
        //It means all values of hashes are going to be increased by 5
        .Get(typeof(LinearCongruenceFamily), size / 2, 5).Create(),
]

```

7.3.3 Choosing Encoder

The basic library currently offers two encoders: Encoder and NoConflictEncoder. NoConflictEncoder expects that no hash functions intersect in ranges. This allows for some optimizations during encoding; namely, it will enable parallel writing to the table.

```

Expression<Func<ulong, ulong>>[] hfs;
int size = 10;
var config =
    EncoderConfiguration<IBLTTableImprovedMemory>(hfs, size);
//Factory does most of the compiling, the user
// seeking performance should reuse Factory as much
// as possible
var factory = EncoderFactory<IBLTTableImprovedMemory>(
    config,
    //This is table factory
    (size) => new IBLTTableImprovedMemory(size)
)
IDecoder decoder = factory.Create()

```

7.3.4 Using Encoder

The user must write their method to wrap a data source and encoder. Here, we provide an example of using RedaFasta to read the files in [SomeFormat].

```

//Opening a [SomeFormat] file
string fastaFilePath = "test.test";
var config = FastaFile.Open(new StreamReader(fastaFilePath));
var reader = new FastaFileReader(
    config.kMerSize, config.nCharsInFile, config.textReader
);

//Some instances implementing the IEncoder interface
IEncoder encoder;

//It may be a good idea to set the buffer to some size
//that is large enough
var buffer = new ulong[1024 * 1024];
while (true)
{

```

```

var data = reader.BorrowBuffer();
if (data is null)
{
    break;
}
//ParallelEncode may speed the encoding
encoder.ParallelEncode(buffer, data.Size);
reader.RecycleBuffer(data);
}

encoder.GetTable() //Returns table

```

7.3.5 Sketch

We should revisit the definition of IBLTTableImprovedMemory. We decided to use HyperGraphDecoder. HyperGraphExpect a sketch to implement IHyperGraphDecoderSketch<TSketch>. Now, we have two choices implement a brand new Sketch and some method transforming Table to the new Sketch or extend IBLTTableImprovedMemory to implement ISketch interface. We are going to use the second way. We will not implement it entirely, but to show important parts.

```

struct IBLTTableImprovedMemory:
    ITable,
    IHyperGraphDecoderSketch<IBLTTableImprovedMemory>
{
    ulong[] _keySums;
    SpaceSavingDictionary _counts;
    public IBLTTableImprovedMemory(int size){
        _data = ulong[size];
        _counts = new SpaceSavingDictionary(size);
        _hashSum = new SpaceSavingDictionary(size);
    }
    void Add(Hash key, Key value){
        _keySums[key] += value;
        _counts.Add(1);
    }
    //This allow the decoding algorithm
    //to remove items from buckets
    void Remove(Hash key, Key value){
        _keySums[key] -= value;
        _counts.Add(-1);
    }
    int GetCount(Hash key) => _counts.Get(key);
    //Now, there is little bit a tricky part
    //we need to implement public looks pure
    //if this method is not defined decoder fails during runtime
    static Expression<Func<ulong, IBLTTable, bool>>
    GetLooksPure(HashingFunctions hashingFunctions)
    {
        var f = CompiledFunctions
            .Create<ulong, IBLTTable, bool>(
                out var key_,
                out var sketch_);

        f.S.Assign(f.Output, false)
            .DeclareVariable(
                out var count_,

```

```

        sketch_.V.Call<int>("GetCount", key_.V)
    )
    .IfThen(
        //Bucket is pure when there is only one item
        //hashed to it
        !(count_.V == 1),
        new Scope().GoToEnd(f.S)
    )
    return f.Construct();
}
}

```

7.3.6 Choosing Decoder

If we decided to implement an `IHyperGraphDecoderSketch<TSketch>`, we might use only a decoder that can use such Sketch. There is only one implementation, and this is currently a `HyperGraphDecoder`. We first need to merge the two sketches of the two Sets; we would like to find symmetric differences.

```

//Using a hypergraph decoder
IBLTTableImprovedMemory sketch1;
IBLTTableImprovedMemory sketch2;
//Know we want to merge these into sketches
//Every sketch needs to implement SymmetricDifference()
//So, imagine we implemented such a method
//This implementation could loop something like this:
public IBLTTable SymmetricDifference(IBLTTable other)
{
    if (other.Size() != Size())
    {
        throw new InvalidOperationException(
            "Sketches do not have same sizes"
        );
    }
    IBLTCell[] data = new IBLTCell[Size()];
    _table.CopyTo(data, 0);
    //This means if some a was encoded in A and also B
    //Then it is not encoded in the merged sketch
    for (int i = 0; i < data.Length; i++)
    {
        IBLTCell otherCell = other.GetCell((ulong)i);
        _table[i].KeySum -= otherCell.KeySum;
        _count.Add(i, -otherCell.GetCount(i));
    }
    return new IBLTTable(data);
}
//Know we just need to merge the two sketches
sketch1.SymmetricDifference(sketch2);

```

After we get a merged sketch, we may create a decoder and use it.

```

IEnumerable<Expression<Func<ulong, ulong>>> hfs;
var factory = HypergraphFactory(hfs);
var decoder = factory.Create(sketch);
decoder.Decode()
//We may examine the decoding state by
decoder.EncodingState
//This property may have many possible states

```

```

//there are three most useful:
// - *Success*
// - *Failure*
// - *Shutdown* (have not finished)

decoder.GetDecodedValues()
// Returns a hash set of values in the symmetric difference
//We may get some decoded values even
//when the algorithm is in a fail state.

```

7.4 Buffers size

The size of buffers may influence performance greatly, especially when using the RedaFasta library to read MaskedSuperString files. Something between $1024 \cdot 16$ and $1024 \cdot 1024$ is optimal for our machine. The work with buffers may be parallized via Tasks to small sizes of buffers that may lead to too large overhead for calling those. For encoders, a number of partitions may be set by calling `SetPartitions(int n)`.

```

Encoder encoder = ...;
//Changes the number of partitions to 8
encoder.SetPartions(8);

```

We wrote some basic benchmarks that may be modified to find the optimal parameter setting for concrete machines. However, they were not meant for this purpose in the first place, and some work is to be done.

7.5 Future Work

We would like to write a console application using this library to find Symmetric Differences between files and, more exactly, MaskedSuperString files to provide a more seamless experience for possible users with no need to build their own Symmetric Difference Finder.

Writing code with LittleSharp is still an unpleasant experience. However, we think it is much better than writing code just with Expression Tree. There is a lot of potential for improving LittleSharp and, with it, the overall readability of the code written using LittleSharp.

The abstraction is currently a little bit problematic as it supports only symmetric differences, but IBLT may support a finding in which set each element lies. This would allow Oracle decoders to write using IBLT.

Oracle Decoders are very experimental features, and they would benefit greatly from fewer random constants in the code and greater parametricity. In simple words, some refactoring is required.

8 Experiments

8.1 Comparison of Hash Functions

We decided to compare different hash function families and their effect on the recovery of IBLT and HPW algorithms. All tests were run for a sketch size of 10000; tests started at the ratio buckets: items of 1:0.72 and ended with ratio 1:0.85. We chose a step of 0.001 and ran 100 tests for each ratio. The data encoded were random k -mer strings more thoughtfully described in the Section 8.2.

We look at the graph comparing all possible combinations together Figure 8.1. We should see three groups: the worst is polynomial hashing, then linear hashing, and the last group comprises tabulation and multiply-shift. We may clearly see that there is no or little difference between HPW and IBLT algorithms. This is not very surprising as both rely on the same bound of the probability of the existence of 2-core in the random 3-universal hypergraph.

However, what is interesting is that polynomial hashing of order 2 is weaker than Linear Congruence. We have no explanation for this behavior; exploration on a more diverse data should probably be done.

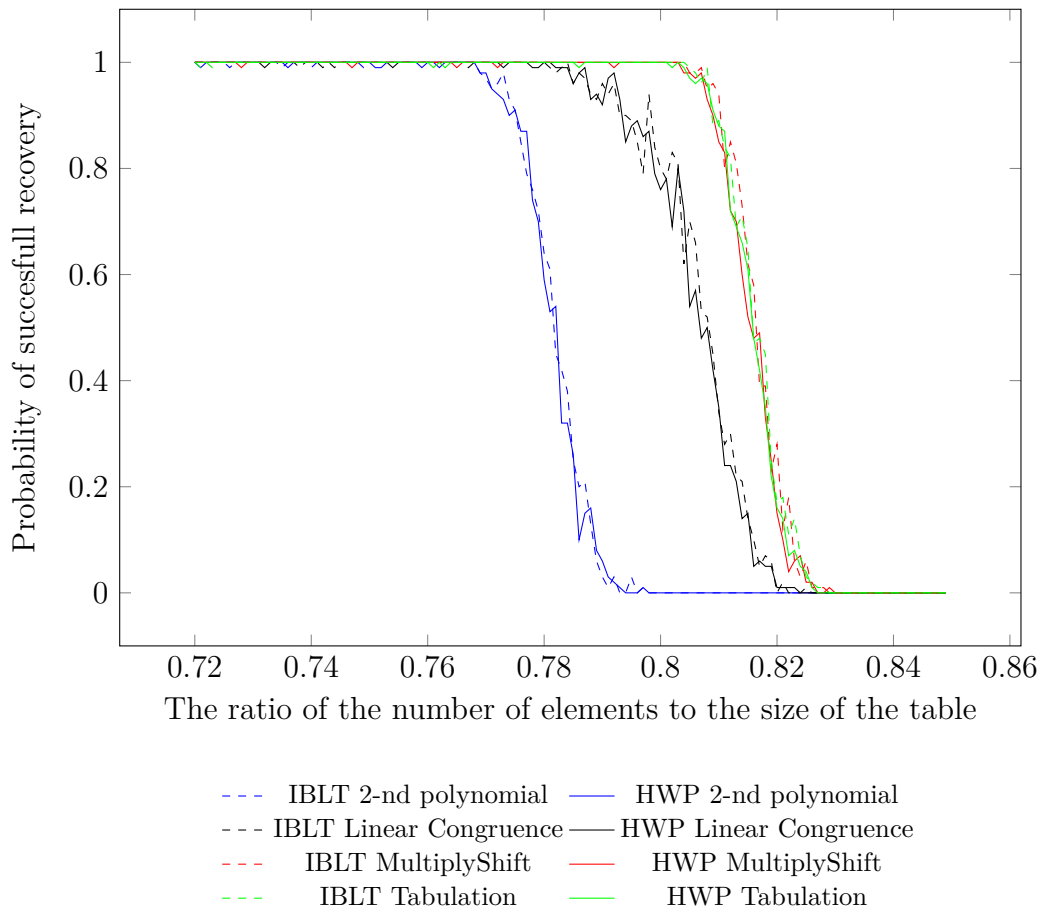


Figure 8.2 Graph of the probability of recovery depending on recovery algorithm, hash function family, and ratio of buckets to elements encoded in 10000 buckets

8.2 Decoding with oracle

HPW and IBLT algorithms have one inherent problem due to their nature: it is impossible to get a better ratio of the buckets to the size of a symmetric difference than 1. This comes from the simple fact that we must pair elements to buckets. However, we could circumvent this problem by pairing multiple elements with each bucket.

We first show an example and then describe the method more formally. Imagine going to the supermarket and buying *bread*, *ham*, and *butter*.

For some reason, you decided to encode all the items using the HPW algorithm to some sketch. For simplicity, we set the number of hash functions to 2.

The resulting sketch looks like this ($\{\text{bread}\}$, $\{\text{bread, ham, butter}\}$, $\{\text{butter, ham}\}$). After some time, we decide to recover items from the sketch. We first recover bread as there exists a bucket bread is also. After that, we are stuck as we have a 2-core. However, we remember that we often buy cheese with bread. We thus toggle cheese into the sketch. This, however, did not help us as it was not located in the sketch. Again, we search in memory, remembering that we often buy ham together with bread. After toggling out the ham, we see that only butter is in the sketch. So, we successfully recovered the whole sketch.

The approach given in the example is very similar to our algorithm for recovery with Oracle. However, there are a few differences.

Definition 17. *Be f a function from A to 2^A , then we call f an oracle over A .*

Definition 18. *Be f a oracle over some nonempty set A , then*

$$G_A^f := (A, \{(a, b); a \in A \wedge b \in f(a)\})$$

is called graf induced by oracle f over A .

This is something close to our implementation:

Algorithm 16 Recover sketch $F : \text{Sketch} \rightarrow \text{Oracles} \rightarrow A$

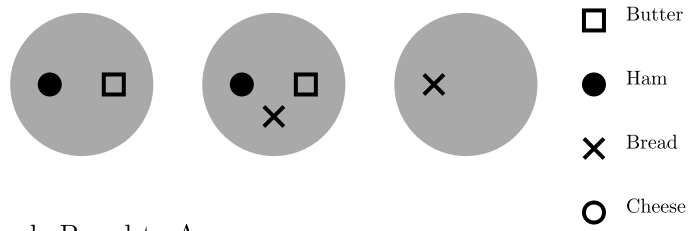
```

1: Initial phase of the algorithm:
2: do  $n$  steps of HPW algorithms
3: Recovery with oracle:
4: for  $i \leftarrow 1$  to  $l$  do
5:    $A \leftarrow$  recovered values
6:   Oracle prediction, pick some values that are "close" to already recovered
7:    $H \leftarrow \{\text{for every } a \in A \text{ pick uniformly randomly some } h \text{ from } f(a) \text{ with probability } p \text{ for every oracle } f \in F\}$ 
8:   Toggle  $H$  into the sketch
9:   do  $k$  steps of HPW algorithm
10:  Toggle  $H$  into the sketch
11:  do  $k$  steps of HPW algorithm
12: end for

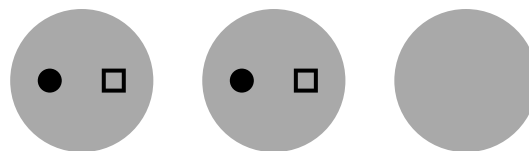
```

Good domains for such recovery are k -mer sets. When there is some k -mer *AATT* three, there is a great chance that one of *ATTA*, *ATTT*, *ATTC*, or *ATTG* also exists. As such, we define oracle f_k for k -mer sets.

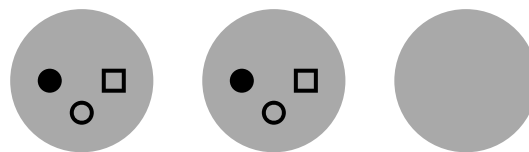
A - Beginning state



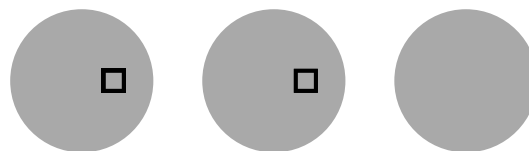
B - Toggle Bread to A



C - Toggle Cheese to B



D - Toggle Ham to B



E - Toggle Bread to D



Figure 8.3 Recovery process from the example

Definition 19. *Be A set canonical k -mers and be $C := \{'A', 'C', 'G', 'T'\}$, than:*

$$f_k(X) := \{\text{cannonical}(X[:k-1] + c); c \in C\}$$

and the other direction.

$$r_k(X) := \{\text{cannonical}(c + X[1:]); c \in C\}$$

We should see that $G_{f_k}^A$ for k -mer set created from DNA string is a way. If we have two k -mer sets created from DNA strings and A is their symmetric difference, then $G_{f_k}^A$ made up of many paths of length at least k . This is, however, true only for sets that do not enforce canonicity. When we enforce canonicity, some of those paths may be shortened. See Algorithm 16

8.2.1 Experiments

We have run the experiments for two hash function families, Linear Congruence and Tabulation. The test started at a ratio of 1 : 1.4 buckets items and ended with 1 : 1.6; we ran 40 tests 20 times each time, increasing the ratio by 0.01. The number of buckets was 10000. We run 100 steps during the initial recovery phase, then we run up to 1000 recovery rounds with up to 10 steps. We set our oracle set to f_k, r_k .

We have not used real k -mer data, but we created random string data. As such, we generated a set of S of random DNA strings of length n , then created a k -mer set from each DNA string and merged them into one large set. We also converted all k -mers to canonical k -mers. We run the experiment for string lengths of 5, 10, 100, and 1000.

We see in Figure 8.4, that tabulation is a better to hash family than linear congruence as it guarantees better behavior. If we look at the graphs, we see that tabulation recovers all sketches until some point, while Linear Congruence starts failing earlier. We can also see, that recovery improves with increased size of strings, this is logical as we longer strings, we have greater chance of getting some k -mer in it recovered.

In Figure 8.5, we compare the recovery of the same algorithm over both canonized k -mer set and non-canonized. We see that the algorithm performs similarly well.

We have not written our tests to measure performance. However on Figures 8.6 and 8.7, we may see some interesting behavior where there is a window of a slowdown around a point where the test sketch is not being recovered. This is probably explainable by the two facts: if we recover many items in the first phase of the algorithm, then the algorithm quickly recovers the remainder, and when we recover too few items, recovery with oracle is very fast as there are not many items to work with.

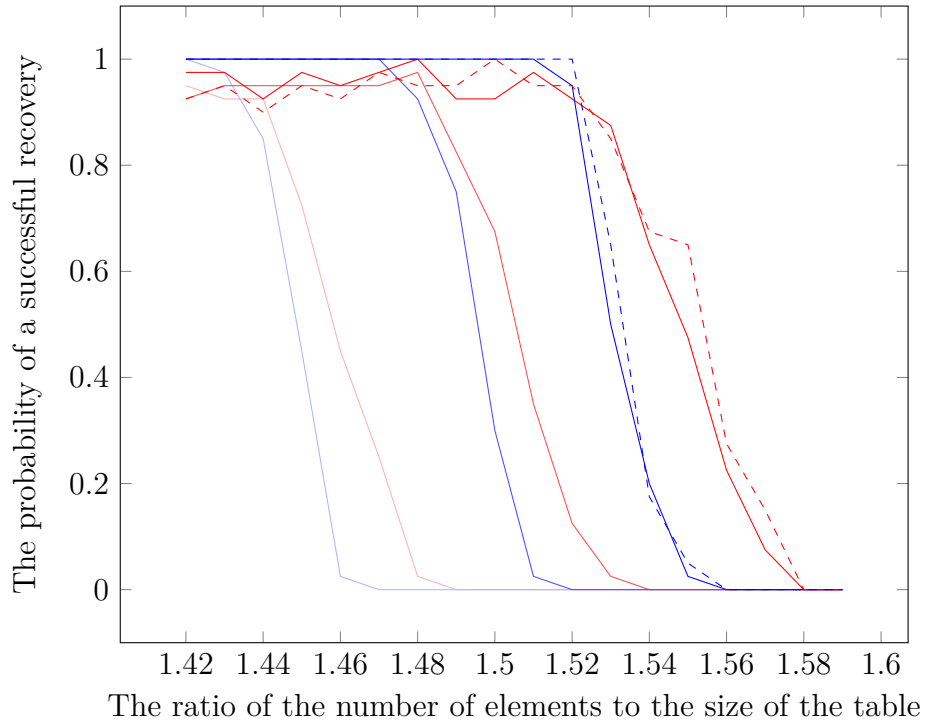


Figure 8.4 Graph of the probability of recovery with Oracle based on HWP algorithm for different lengths of strings for tabulation and linear congruence and on the number of elements to the number of buckets over 10000 buckets

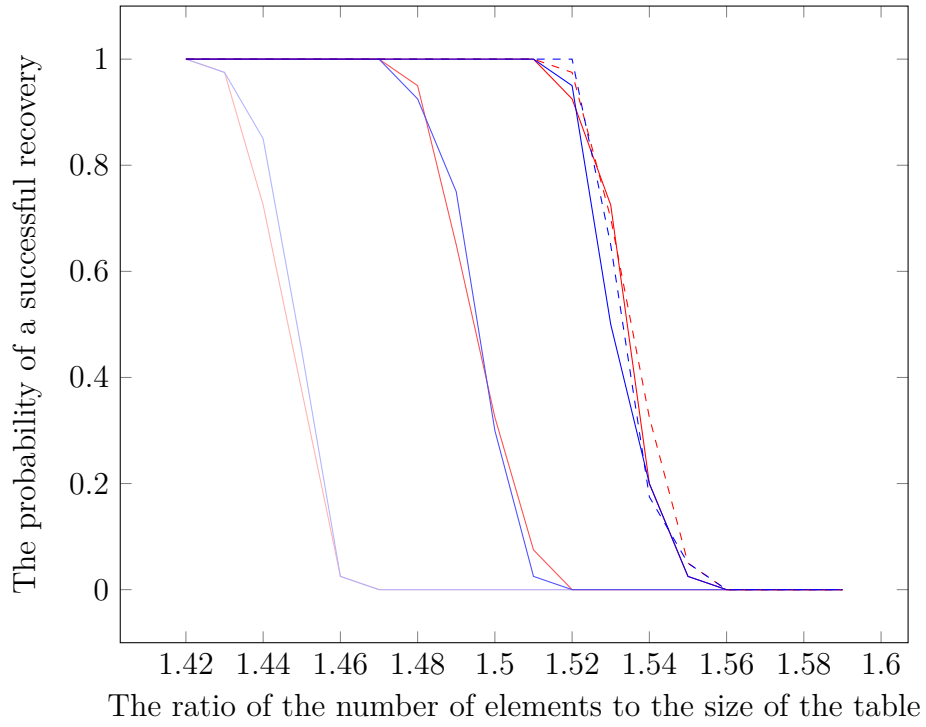


Figure 8.5 Graph of the probability of recovery with Oracle based on HWP algorithm for different lengths of strings for Cannonic and Non-Canonical sets on the number of elements to the number of buckets over 10000 buckets

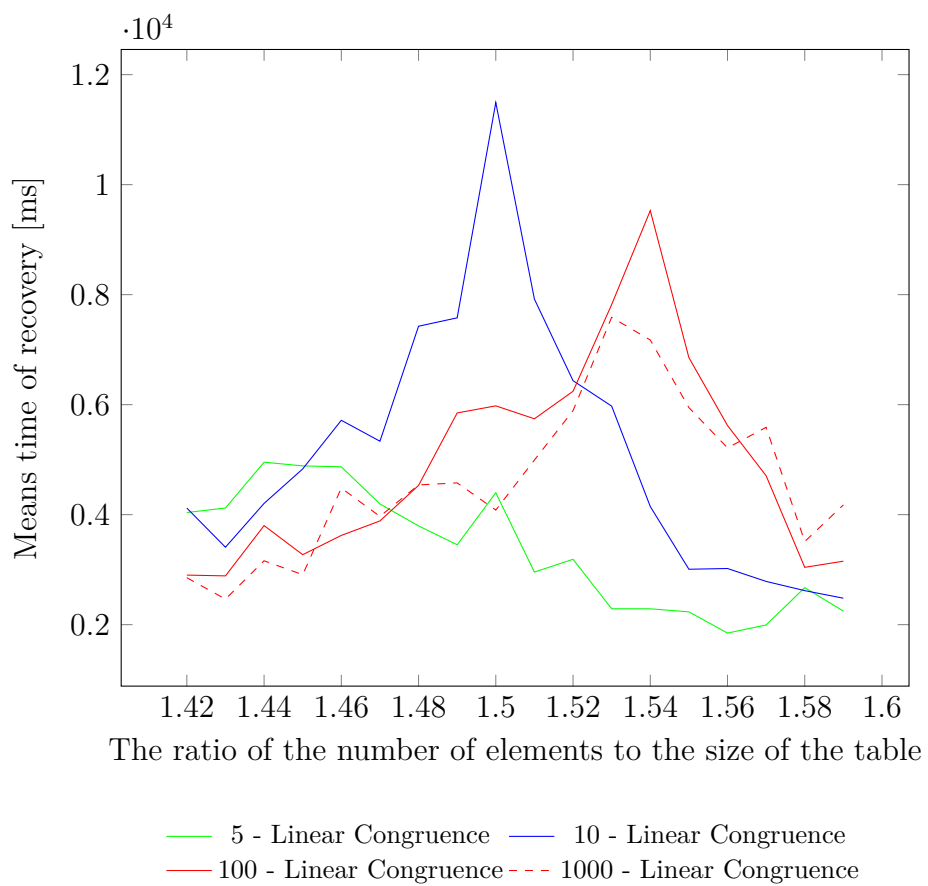


Figure 8.6 Graph of the mean time of recovery with Oracle based on HWP algorithm over Linear Congruence for different lengths of strings and on the number of elements to the number of buckets over 10000 buckets

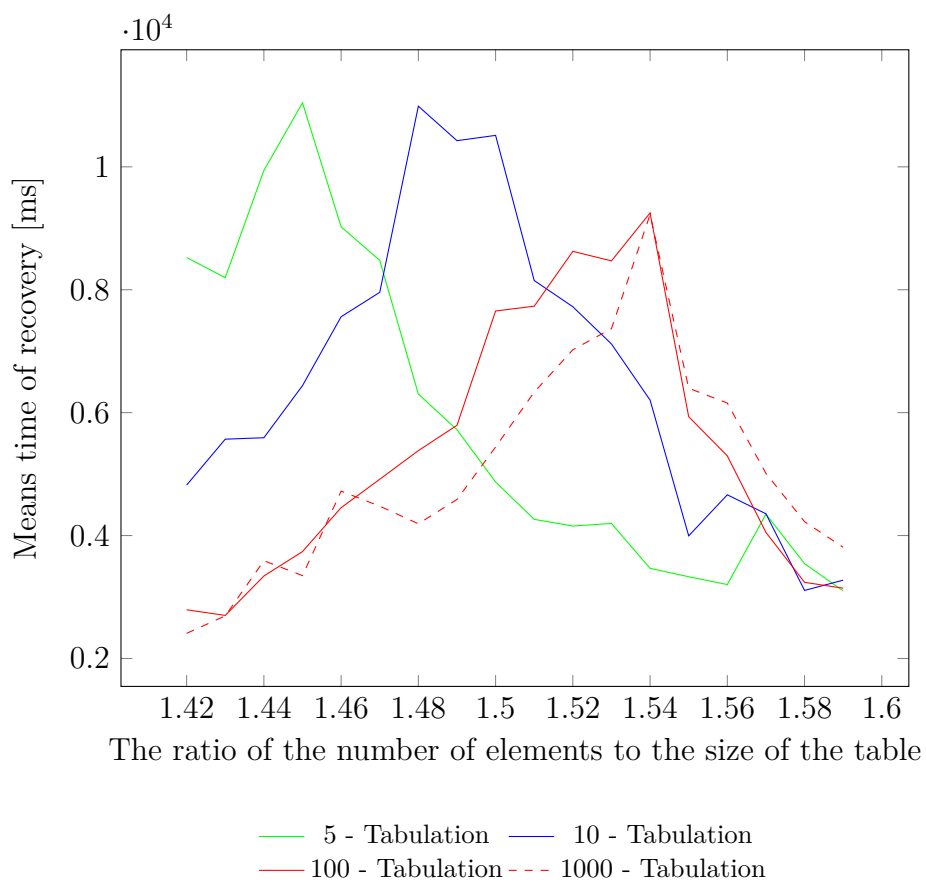


Figure 8.7 Graph of the mean time of recovery with Oracle based on HWP algorithm over Tabulation for different lengths of strings and on the number of elements to the number of buckets over 10000 buckets

8.3 Performance

We did some performance testing using primarily BenchmarkDotNet for such purpose.

8.3.1 Hash functions

We tested the performance of the hash function with this benchmark:

```
[Benchmark]
public ulong BenchmarkHashFunction()
{
    ulong sum = 0;
    ulong[] buffer = new ulong[BufferLength];
    //BufferLength was set at 4096
    ulong[] answerBuffer = new ulong[BufferLength];

    //FillBuffer was called 1024 times
    //Data was created beforehand

    //HOTPATH
    while (Stream!.FillBuffer(buffer) > 0)
    {
        f(buffer, answerBuffer, 0, BufferLength);

        for (int j = 0; j < BufferLength; j++)
        {
            sum += answerBuffer[j];
        }
    }
    //END HOPATH
    return sum;
}
```

Data was created beforehand, and there were circa 4 million numbers hashed ($1024 \cdot 4096$). Modulo is just the remainder of division by a certain size; polynomial had order 2.

Hashing Scheme	Mean	Error	StdDev	Median	MH/s
LinearCongruence	14.514 ms	0.2823 ms	0.4477 ms	14.388 ms	288.97
Modulo	8.828 ms	0.1628 ms	0.4202 ms	8.848 ms	475.12
MultiplyShift	8.779 ms	0.1732 ms	0.3034 ms	8.808 ms	477.66
Polynomial-2	16.787 ms	0.5540 ms	1.5625 ms	16.181 ms	249.91
Tabulation	80.691 ms	1.8565 ms	5.3265 ms	79.234 ms	51.97

Table 8.1 Benchmark results for various hash function families for circa 4 million ulong entries over BenchmarkDotNet v0.13.12, Ubuntu 20.04.6 LTS (Focal Fossa) (container) AMD EPYC 7763, 1 CPU, 2 logical cores and 1 physical core .NET SDK 8.0.301 [Host] : .NET 8.0.6 (8.0.624.26715), X64 RyuJIT AVX2 Job-ABHFES : .NET 8.0.6 (8.0.624.26715), X64 RyuJIT AVX2

When compared with results of Pătraşcu and Thorup [10]. See Table 8.2. We may see that our implementation of Polynomial is much faster; it has just 2 times

slowdown to multiply shift, but theirs shows more than 10. Such a difference is very surprising; we do not understand why.

One possibility is that we tested hashing schemes for some fixed size in our tests. We had to pay for a modulo operation or its equivalent. We are not sure whether the authors of the paper did similarly [10].

Another possibility may be different instruction costs on different machines. This may not be the reason entirely. However, tabulation is circa 30 percent faster on our local machine, while MultiplyShift is nearly 2 times slower.

Hashing Scheme	MH/s
univ-mult-shift	787.40
3-indep-Mersenne-prime	58.14
tabulation	206.61

Table 8.2 Results of Pătraşcu and Thorup for various Hashing Schemes

8.4 Updates

When we look at the cost of updates, there are two main costs: Hashing and Random Access to a table. Sadly, we do not live in a world where RAM is a reality, and the larger an array is, the higher the cost for random access to it. This is mainly caused by the architecture of memory; as such, for some size, the array does not fit into the cache, and we get more cache misses.

8.4.1 Update Cost Depending on the Size of the Sketch

We tested encoding for sketches of different sizes, and as a hash function family, we chose multiply-shift. We show the benchmark used.

```
[Benchmark]
public ulong[] BenchmarkDecoder()
{
    var config = new EncoderConfiguration<XORTable>(
        new[] {
            HashingFunctionProvider.Get(
                hashingFunctionFamily,
                TableSize,
                0
            )},
        (int)TableSize
    );
    var factory = new EncoderFactory<XORTable>(
        config,
        size => new XORTable(new ulong[size])
    );
    var encoder = factory.Create();
    //HOTPATH
    foreach (var buffer in stream)
    {
        encoder.Encode(buffer, buffer.Length);
    }
}
```

```

//END HOTPATH
return encoder.GetTable().GetUnderlyingTable();
}

```

Size of the table	HPW MOp/s	IBLT MOp/s
2^{10}	278.33	181.47
2^{11}	274.58	178.53
2^{12}	274.17	175.93
2^{13}	273.79	174.42
2^{14}	263.55	154.11
2^{15}	253.25	147.43
2^{16}	238.48	139.84
2^{17}	221.54	135.77
2^{18}	215.58	130.44
2^{19}	205.39	124.66
2^{20}	190.34	74.28
2^{21}	182.21	47.15
2^{22}	102.49	44.05
2^{23}	87.68	40.08
2^{24}	81.75	36.90
2^{25}	74.19	34.88
2^{26}	58.45	27.98

Table 8.3 Benchmark results for different table sizes (powers of two) for HPW and IBLT in MOp/s. BenchmarkDotNet v0.13.12, Ubuntu 20.04.6 LTS (Focal Fossa) (container) AMD EPYC 7763, 1 CPU, 4 logical and 2 physical cores .NET SDK 8.0.203 [Host] : .NET 8.0.3 (8.0.324.11423), X64 RyuJIT AVX2

Firstly, this comparison is inherently unfair. As IBLT is tested for 2 hash functions. This, however, does not impact the latter cost of consuming 3-times memory. As we may see, IBLT is up 2 times slower for very large tables as the cost of random access increases.

We may see two interesting facts: the overhead cost for the multiply shift is approximately two times the cost of hashing. However, as the sizes increase, the performance drops significantly. This is, however, in line with our model. One should note that for different machines, the exact values are going to differ, probably mainly depending on cache architecture. If we consider the performance of hash functions, we may clearly see that tabulation will be very expensive. We should also see that the performance of hash functions is much more important for small sizes, as their effect is greater.

8.4.2 Encoding k -mer sets

The test comprised reading a file containing 100M k -mers and encoding using 3 hash functions from the multiply shift family to a table of size 10M. We parallelized the encoding and used this benchmark:

```

[Benchmark]
public object ParallelEncode()

```

```

{
    var hfs = HashingFunctionCombinations
        .GetFromSameFamily(
            3,
            new MultiplyShiftFamily()
                .GetFactory()((int)TableSize)
                .Select(h => LittleSharp
                    .Utils
                    .Buffering
                    .BufferFunction(h).Compile())
                .ToList());
    var GetEncoder = () =>
        new Encoder<XORTable>((
            new XORTable((int)TableSize),
            hfs,
            1024));
    string fastaFilePath = "test.test";
    var config = FastaFile.Open(new StreamReader(fastaFilePath));
    var reader = new FastaFileReader(
        config.kMerSize,
        config.nCharsInFile,
        config.textReader,
        1024 * 1024,
        8
    );

    var CreateTask = () => new Task<XORTable>(() =>
    {
        var encoder = GetEncoder();
        while (true)
        {
            //HOTPATH
            FastaFileReader.Buffer? data;
            lock (reader)
            {
                data = reader.BorrowBuffer();
            }
            if (data is null)
            {
                break;
            }
            encoder.EncodeParallel(data.Data, data.Size);
            lock(reader){
                reader.RecycleBuffer(data);
            };
            //END HOTPATH
        }
        return encoder.GetTable();
    });
    var tasks =
        Enumerable
            .Range(0, 4)
            .Select(_ => CreateTask())
            .ToArray();
    foreach (var task in tasks)
    {
        task.Start();
    }
}

```

```
Task.WaitAll(tasks.ToArray());  
return tasks.Aggregate(  
new XORTable((int)TableSize),  
(t, x) => x.Result.SymmetricDifference(t)  
);  
}
```

We got the mean time of 645 **ms**, translating to **152 MOp/s**. One should also note the cost of reading from the k -mer file is not insignificant as it consumes relatively many instructions. The test was performed this machine: BenchmarkDotNet v0.13.12, Ubuntu 20.04.6 LTS (Focal Fossa) (container) AMD EPYC 7763, 1 CPU, 4 logical and 2 physical cores .NET SDK 8.0.203 [Host] : .NET 8.0.3 (8.0.324.11423), X64 RyuJIT AVX2 DefaultJob : .NET 8.0.3 (8.0.324.11423), X64 RyuJIT AVX2

Conclusion

8.5 Comparison of HPW to IBLT

We have found that HPW did not have a worse recovery in experiments than IBLT. However, it consumes much less memory, making it much more performant in the encoding phase. We have not measured the decoding phase rigorously, but from some experiments, we predict that IBLT is a little bit faster than HPW. As such, finding symmetric differences between sets with larger symmetric differences when using IBLT may provide an advantage. We implemented only the original version of IBLT. However, more memory-efficient variations were described. [11]

Improvements

We increased the recovery ratio from 1:0.8 to 1:1.52 for the recovery of k -mer sets. When we ran some further experiments, we found that when Oracle provides fewer possibilities (i.e. predictions with greater correct), then the recovery ratio may be further improved.

It may be futile to explore possibilities and ways to improve oracle prediction. One idea is to encode more information into the sketch. Thus improving the prediction of oracle.

Another avenue for further research could be to find rigorous bounds on such decoding. However, we fear such an endeavor may not be easy.

Another question is whether there are other domains for which recovery with Oracle would be beneficial.

There is also a known approach to increase the ratio to 1:0.92 by using non-uniform hash functions as Michael Rink's experimental result shows [12]. We could incorporate such solution to our, however this would be at cost to performance as it requires to use up to 21 hash functions.

Implementation

We achieved a fast but highly generic codebase, allowing users to easily swap even low-level components with no or little cost to overall performance.

In the chapters dedicated to each library, we further discussed possible work on it. To sum up, there is still work to do on SymmetricDifferenceFinder.

Some technical debt needs to be resolved, largely in the more experimental parts of the project, such as recovery with oracles.

During the project, we met and fought successfully with the language's limitations. The main issue is the lack of support for recompiling functions for some known parameters. For example, have a generic $Divide(a, b) = a/b$; if we would want many operations for the same $b := 7$, we consider recompiling such a function by calling $Divide(a) = Recompile(Divide, b = 7)$ during runtime. We circumvented that by using Expression Trees; however, this is far from a pleasant experience.

Bibliography

1. HOUEN, Jakob Bæk Tejs; PAGH, Rasmus; WALZER, Stefan. Simple Set Sketching. In: *2023 Symposium on Simplicity in Algorithms (SOSA)*. [N.d.], pp. 228–241. Available from DOI: 10.1137/1.9781611977585.ch21.
2. MOLLOY, Michael. The pure literal rule threshold and cores in random hypergraphs. In: MUNRO, J. Ian (ed.). *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004, New Orleans, Louisiana, USA, January 11-14, 2004*. SIAM, 2004, pp. 672–681. Available also from: <http://dl.acm.org/citation.cfm?id=982792.982896>.
3. THORUP, Mikkel. Fast and powerful hashing using tabulation. *Commun. ACM*. 2017, vol. 60, no. 7, pp. 94–101. ISSN 0001-0782. Available from DOI: 10.1145/3068772.
4. SLADKÝ, Ondřej; VESELÝ, Pavel; BŘINDA, Karel. Function-Assigned Masked Superstrings as a Versatile and Compact Data Type for k-Mer Sets. *bioRxiv*. 2024. Available from DOI: 10.1101/2024.03.06.583483.
5. MORRIS, Robert. Counting large numbers of events in small registers. *Commun. ACM*. 1978, vol. 21, no. 10, pp. 840–842. ISSN 0001-0782. Available from DOI: 10.1145/359619.359627.
6. BAR-YOSSEF, Ziv; JAYRAM, TS; KUMAR, Ravi; SIVAKUMAR, D; TREVISAN, Luca. Counting distinct elements in a data stream. In: *Randomization and Approximation Techniques in Computer Science: 6th International Workshop, RANDOM 2002 Cambridge, MA, USA, September 13–15, 2002 Proceedings*. Springer, 2002, pp. 1–10.
7. FLAJOLET, Philippe; FUSY, Éric; GANDOUET, Olivier; MEUNIER, Frédéric. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. *Discrete mathematics & theoretical computer science*. 2007, no. Proceedings.
8. CORMODE, Graham; YI, Ke. *Small summaries for big data*. Cambridge University Press, 2020.
9. GOODRICH, Michael T.; MITZENMACHER, Michael. Invertible bloom lookup tables. In: *2011 49th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. 2011, pp. 792–799. Available from DOI: 10.1109/Allerton.2011.6120248.
10. PĂTRAȘCU, Mihai; THORUP, Mikkel. Twisted tabulation hashing. In: *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*. New Orleans, Louisiana: Society for Industrial and Applied Mathematics, 2013, pp. 209–228. SODA '13. ISBN 9781611972511.
11. FLEISCHHACKER, Nils; LARSEN, Kasper Green; OBREMSKI, Maciej; SIMKIN, Mark. *Invertible Bloom Lookup Tables with Less Memory and Randomness* [Cryptology ePrint Archive, Paper 2023/918]. 2023. Available also from: <https://eprint.iacr.org/2023/918>. <https://eprint.iacr.org/2023/918>.

12. RINK, Michael. On Thresholds for the Appearance of 2-cores in Mixed Hypergraphs. *CoRR*. 2012, vol. abs/1204.2131. Available from arXiv: 1204.2131.

List of Figures

8.1	All together	62
8.2	Graph of the probability of recovery depending on recovery algorithm, hash function family, and ratio of buckets to elements encoded in 10000 buckets	62
8.3	Recovery process from the example	64
8.4	Graph of the probability of recovery with Oracle based on HWP algorithm for different lengths of strings for tabulation and linear congruence and on the number of elements to the number of buckets over 10000 buckets	66
8.5	Graph of the probability of recovery with Oracle based on HWP algorithm for different lengths of strings for Cannonic and Non-Canonized sets on the number of elements to the number of buckets over 10000 buckets	67
8.6	Graph of the mean time of recovery with Oracle based on HWP algorithm over Linear Congruence for different lengths of strings and on the number of elements to the number of buckets over 10000 buckets	68
8.7	Graph of the mean time of recovery with Oracle based on HWP algorithm over Tabulation for different lengths of strings and on the number of elements to the number of buckets over 10000 buckets	69

List of Tables

8.1	Benchmark results for various hash function families for circa 4 million ulong entries over BenchmarkDotNet v0.13.12, Ubuntu 20.04.6 LTS (Focal Fossa) (container) AMD EPYC 7763, 1 CPU, 2 logical cores and 1 physical core .NET SDK 8.0.301 [Host] : .NET 8.0.6 (8.0.624.26715), X64 RyuJIT AVX2 Job-ABHFES : .NET 8.0.6 (8.0.624.26715), X64 RyuJIT AVX2	70
8.2	Results of Pătraşcu and Thorup for various Hashing Schemes . . .	71
8.3	Benchmark results for different table sizes (powers of two) for HPW and IBLT in MOp/s. BenchmarkDotNet v0.13.12, Ubuntu 20.04.6 LTS (Focal Fossa) (container) AMD EPYC 7763, 1 CPU, 4 logical and 2 physical cores .NET SDK 8.0.203 [Host] : .NET 8.0.3 (8.0.324.11423), X64 RyuJIT AVX2	72

List of Abbreviations

MOp/s mega operations per seconds

MH/s mega hashes per seconds

whp with high probability

HWP Houen-Waltzer-Pagh recovery algorithm

IBLT Invertible Bloom Lookup Table

CLR Common Language Runtime

GC Garbage Collector

CIL Common Intermediate Language