

Software Engineering Department  
ORT Braude College

Capstone Project Phase A

**K-Mismatch Algorithm Optimization and Application**

Project Number

**24-1-D-13**

Supervisor: Dr. Zakharia Frenkel

Vladimir Brustinov ([vladimir.brustinov@e.braude.ac.il](mailto:vladimir.brustinov@e.braude.ac.il))

Nik Lerman ([nik.lerman@e.braude.ac.il](mailto:nik.lerman@e.braude.ac.il))

Git: <https://github.com/keepthegoal/24-1-D-13>

Table of Contents

[**Abstract 3**](#_heading=h.iv9e3rbdh86g)

[**Introduction 4**](#_heading=h.s27pfgm5igmd)

[1. Problem Definition 4](#_heading=h.q4k6nuup7qew)

[2. Proposed Solution 4](#_heading=h.kt99s876yow1)

[**Related Work 5**](#_heading=h.t264e0uhe1s1)

[1. Virtual Calls Amortization 5](#_heading=h.wxpuilr9qdyf)

[2. STL Containers Optimization and Redefinitions 7](#_heading=h.cxielv6npcx4)

[3. Operator Optimization and Redefinitions 7](#_heading=h.9gqcfapsbja5)

[4. Object Pooling 7](#_heading=h.aa5nakvjjrc7)

[5. Using References Instead of Pointers 7](#_heading=h.2bhlpc8wl0ct)

[6. Limit Exception Handling 8](#_heading=h.fed3qa3cfob5)

[7. Avoiding Runtime Type Identification (RTTI) 8](#_heading=h.bzgxrqbbdbxt)

[8. Avoiding iostream 8](#_heading=h.facbfaw0u1oj)

[9. Evaluating Alternative Libraries 8](#_heading=h.1jribrdry24i)

[10. Initialization Over Assignment Preference 9](#_heading=h.qp1uykweowux)

[11. Preference for Prefix Operators Over Postfix 9](#_heading=h.d9s25z54chy1)

[12. Explicit Constructors Use 9](#_heading=h.qb95cshde5z1)

[13. Inlining Functions 10](#_heading=h.yaqvgoy1x7bo)

[14. Compiling Parameters Optimizations 10](#_heading=h.4mcqndhql0gv)

[15. Compiler-Based Functions Optimizations 10](#_heading=h.1sds20v7n0jn)

[16. Cache-Based Optimizations 10](#_heading=h.g286r2lifprf)

[17. Memory Pooling 11](#_heading=h.o5zql74n0md)

[**K-Mismatch Background 12**](#_heading=h.vw6jobgkwy4l)

[1. Early Developments 12](#_heading=h.1auaqwce564f)

[2. Addressing Mismatches 12](#_heading=h.nr25xsrmend0)

[3. Memory Efficiency: A Cornerstone of Algorithm Design 12](#_heading=h.8qstbbpzqkjo)

[4. Searching by given filters optimization 13](#_heading=h.kbt3oscpgdxf)

[5. Optimization path 14](#_heading=h.4y717cpbpbyt)

[**Expected Achievements 16**](#_heading=h.6g2x8j5g6hsi)

[1. Criteria for Success 16](#_heading=h.na0ujewnsw22)

[**The Process 17**](#_heading=h.7i0udmvl8ih7)

[1. Research 17](#_heading=h.xfsvimp45o39)

[2. Methodology and Development Process 18](#_heading=h.26rxpakwlq1u)

[**Product 19**](#_heading=h.zh53xccn0ues)

[1. Functional Requirements 19](#_heading=h.uisy3616jnvh)

[2. Non-functional Requirements 19](#_heading=h.fbt7o6gc49da)

[3. Use Case Diagram 19](#_heading=h.61qctxhllmux)

[4. Architecture Overview 20](#_heading=h.4zm6fymy93y6)

[4.1. Naive Search 20](#_heading=h.r5dfkcf420oe)

[4.2. Generate MCS 21](#_heading=h.ulp0wm4xhucd)

[**Verification and Evaluation 22**](#_heading=h.xmfk63d38x9l)

[1. Evaluation 22](#_heading=h.luxjc88d88n7)

[2. Verification 22](#_heading=h.bywanpd6bb6s)

[**References 23**](#_heading=h.nr6rb6sw6p25)

# Abstract

The increasing complexity of string matching in computational biology, cybersecurity, and text processing applications underscores the importance of developing more efficient algorithms capable of handling mismatches effectively. The new k-mismatch algorithm based on spaced q-grams developed in our department, represents a critical advancement in string searching algorithms, offering appropriate speed for applications requiring nuanced text analysis.

Whereas the problem has an appropriate algorithmic solution, the problem of effective implementation still holds on the ground. The problem includes memory and CPU efficient methods such as minimizing cache misses, optimizing data structures for speed, reducing memory overhead, and handling large datasets efficiently. It was observed that different implementations can significantly vary in speed and RAM requirements.

The goal of the problem is to investigate an existing solutions for the similar implementation problems to build a robust platform for finding of necessary tricks and tips for proper algorithmic techniques, and to make a primary research toward finding of the best k-mismatch search implementation solution.

**Chapter 1**

# Introduction

The escalating complexity of string matching in many domains highlights the critical necessity for the development of more efficient algorithms capable of effectively handling mismatches. The K-Mismatch algorithm, which permits the inclusion of up to 'k' mismatches within the pattern matching process, signifies a significant advancement in string search algorithms. The development project of code optimization will focus on runtime reduction, memory and CPU- efficient implementation of the K-Mismatch algorithm.

### Problem Definition

The k-mismatch problem is a computational problem in the field of string matching, where the objective is to find all occurrences of a pattern string within a text string, allowing for up to k mismatches. Here is a plain text description of the problem:

*Given:*

* A text string T of length n.
* A pattern string P of length m.
* An integer k, representing the maximum number of allowed mismatches.

*Definition:* For each position i in the text string T, determine whether there is an occurrence of the pattern string P starting at position i, such that the number of positions at which the corresponding characters in P and the substring of T starting at i differ is at most k. In other words, for each position i in T, we want to check if there exists a substring of T starting at i that matches the pattern P with at most k mismatches.

*Output:* A list of all starting positions i in the text T where the pattern P occurs with at most k mismatches.

### Proposed Solution

Utilizing the power of C++, this project implements the K-Mismatch algorithm to address efficiency challenges. By leveraging C++'s efficiency in memory management and its powerful standard library, the implementation aims to provide a fast, reliable, and scalable solution for string matching problems with specific mismatch allowances. By focusing on proper implementation without compromising on speed, this approach aims to make the K-Mismatch algorithm more applicable and efficient for real-world applications.

**Chapter 2**

# Related Work

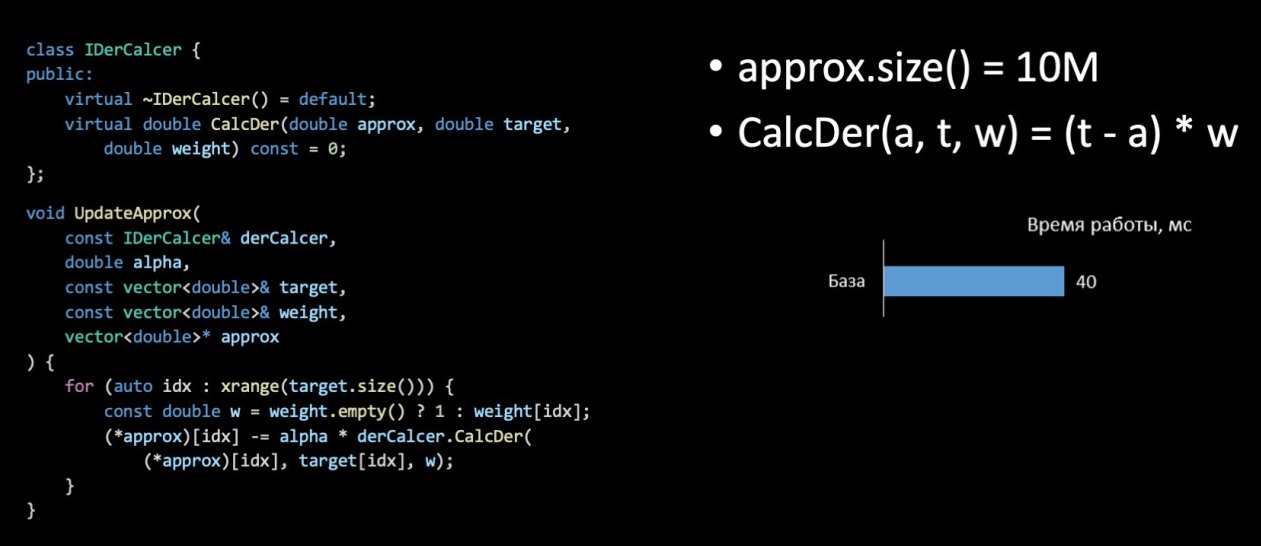
During our problem exploration phase, we have encountered a couple of data sources offering proven optimization methodologies and practices aimed at achieving high-performance code implementation. These sources cover a wide range of C++-specific techniques, such as strategies for minimizing the use of temporary objects, hints for function inlining, relative efficiency of the Standard Template Library (STL), and proven operator performance. Additionally, these resources provide numerous general guidelines and tricks to help developers achieve optimal results, including tips on stack redefinition, utilizing processor vector operators, and leveraging I/O stream features.

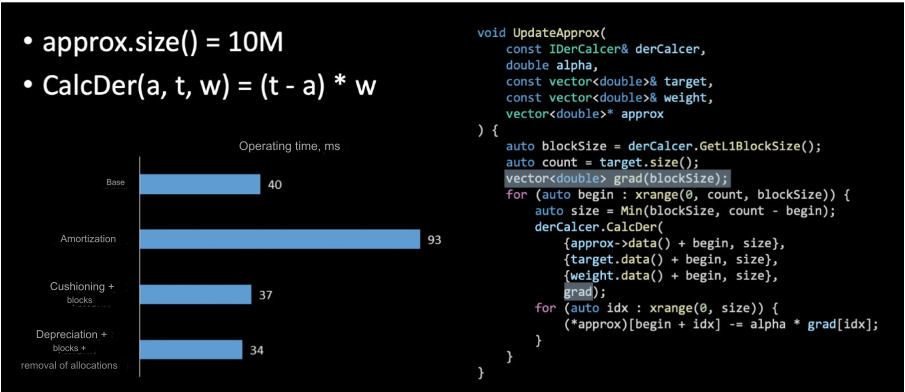
Understanding and implementing these optimization methodologies can significantly enhance the performance of C++ code - reducing unnecessary computational overhead, eliminating the overhead of function calls, more memory-efficient code that executes faster, especially for arithmetic operations. Overall, these techniques and practices are crucial for developers looking to write efficient and high-performance C++ code, helping algorithm implementation to be more attractive for usage in complicated projects and systems.

For exact code performance bottleneck identification there is profiling required. Code profiling purpose is to ensure it is optimized, resulting in high application performance. It analyzes the memory, CPU, and network utilized by each software component or routine. But some proven C++ optimization approaches can be implemented during early development phases and there are number of them:

### Virtual Calls Amortization[[1]](#footnote-1)

Virtual function calls can be more costly than direct function calls due to the indirection needed to support polymorphism. This indirection typically involves a lookup in a virtual table (vtable) to find the correct function to execute based on the actual object type at runtime.







In come cases manual devirtualization via redesigning class hierarchies to minimize the use of polymorphism can improve code performance.

### STL Containers Optimization and Redefinitions[[2]](#footnote-2)

The use of Standard Template Library (STL) containers is a cornerstone in C++ programming, offering robust data structures like vectors, lists, and maps. While these containers are highly versatile, their default behaviors and memory allocation strategies might not be optimized for all scenarios, leading to potential inefficiencies. Customizing these containers, either by directly modifying their behavior or choosing specialized versions that better align with specific performance needs, can significantly enhance performance. Techniques such as pre-allocating memory with `reserve()` in vectors to avoid frequent reallocations, or redefining container operations to minimize complexity, can lead to noticeable improvements in both speed and memory usage.

### Operator Optimization and Redefinitions[[3]](#footnote-3)

Operator overloading in C++ allows for clean and intuitive syntax but can introduce hidden performance costs if not carefully optimized. For instance, overloading arithmetic operators without careful consideration of argument passing and return types can lead to unnecessary object copies. Optimizing these operators by ensuring arguments are passed by reference, particularly in the case of large objects, and by returning references where appropriate, can significantly reduce overhead. Additionally, redefining operators to minimize their computational footprint, especially in performance-critical sections of code, can contribute to overall application efficiency.

### Object Pooling[[4]](#footnote-4)

Object pooling is a specific form of memory pooling where instances of a class are reused instead of being frequently created and destroyed. This is particularly useful for objects that are expensive to construct.

The concept of object pools is discussed in the book "Patterns in Game Design" by Björk and Holopainen, which, while focused on game development, offers valuable insights into efficient resource management applicable in other domains.

### Using References Instead of Pointers[[5]](#footnote-5)

In C++, references provide a more user-friendly alternative to pointers, with the added benefit of not being nullable, which enhances code safety and readability. While references and pointers can often be used interchangeably in terms of functionality, references offer a syntax that makes the intentions of code clearer and prevents some common errors associated with pointer arithmetic and dereferencing. However, it's important to note that the choice between pointers and references might not always lead to direct performance improvements but can contribute to cleaner code, which is easier to optimize and maintain.

### Limit Exception Handling[[6]](#footnote-6)

Exception handling is a powerful feature of C++ that enables robust error management through stack unwinding and object cleanup. However, the convenience of exceptions comes at a cost, introducing runtime overhead that can impact the performance of critical code paths. By limiting the use of exceptions to truly exceptional conditions and favoring explicit error checking in performance-sensitive areas, developers can maintain the benefits of robust error handling while mitigating its performance impact. This approach encourages the use of alternative mechanisms, such as error codes or condition flags, in hot code paths where efficiency is paramount.

### Avoiding Runtime Type Identification (RTTI)[[7]](#footnote-7)

RTTI provides mechanisms for dynamically identifying the type of an object at runtime, facilitating dynamic casting and type checks. However, these features come with a performance penalty due to the overhead of managing type information and performing runtime checks. In performance-critical applications, it's often advantageous to avoid RTTI and rely on alternative design patterns, such as static polymorphism (using templates) or the visitor pattern, which can provide similar functionality with less overhead. This not only enhances performance but also encourages cleaner and more maintainable code by avoiding dynamic type dependencies.

### Avoiding iostream[[8]](#footnote-8)

The convenience and flexibility of the `iostream` library for input and output operations in C++ are well-known. However, its performance is often outmatched by more specialized I/O libraries or the older, but faster, C standard I/O functions (`stdio`). For applications where I/O performance is critical, such as high-throughput logging or data processing, bypassing `iostream` in favor of direct system calls or buffered I/O operations can result in significant performance improvements. This optimization is particularly relevant in contexts where I/O operations are a bottleneck.

### Evaluating Alternative Libraries[[9]](#footnote-9)

While the C++ Standard Library provides a broad range of functionalities, certain applications may benefit from third-party or specialized libraries that are optimized for specific tasks. For instance, scientific computing applications might leverage libraries optimized for matrix operations, while networked applications might benefit from libraries that offer more efficient event handling or I/O multiplexing. Actively evaluating and integrating these specialized libraries can lead to substantial performance gains by leveraging domain-specific optimizations that are not available in more general-purpose libraries.

### Initialization Over Assignment Preference[[10]](#footnote-10)

In C++, object initialization directly constructs an object with its intended value, whereas assignment constructs an object and then assigns it a new value. This distinction is crucial for performance, as direct initialization can avoid the overhead associated with default construction and subsequent assignment. This is particularly relevant for objects of classes with complex construction logic or resource allocation. By preferring initialization over assignment, developers can reduce the number of operations performed and potentially take advantage of compiler optimizations, leading to more efficient code.

### Preference for Prefix Operators Over Postfix[[11]](#footnote-11)

The distinction between prefix and postfix increment/decrement operators in C++ is subtle but significant in terms of performance. While prefix operators (`++i`, `--i`) increment or decrement the value of a variable and then return the result, postfix operators (`i++`, `i--`) create a temporary copy of the variable, modify the original, and then return the copy. This additional copying can be inefficient, especially for complex data types or in tight loops. Preferring prefix operators eliminates this overhead, leading to more efficient code, particularly in iterative operations over containers or sequences.

### Explicit Constructors Use[[12]](#footnote-12)

In C++, constructors that can be called with a single argument implicitly define a conversion from the argument type to the class type. Marking these constructors as `explicit` prevents such implicit conversions, which can prevent unintended conversions that might lead to performance issues or bugs. Using explicit constructors helps ensure that object conversions are intentional, improving code clarity and safety. This practice is particularly important in performance-critical code where unintended conversions could introduce significant overhead.

### Inlining Functions[[13]](#footnote-13)

Function inlining is a compiler optimization that replaces a function call with the function's body, eliminating the overhead associated with the call and return process. While inlining offers the potential for significant performance improvements, especially in small, frequently called functions, it must be used judiciously to avoid code bloat, which can have adverse effects on cache performance and overall speed. The decision to inline should be based on profiling and understanding the application's performance characteristic.

### Compiling Parameters Optimizations[[14]](#footnote-14)

Modern C++ compilers provide a wide range of optimization flags that can fine-tune the performance of the generated binary. These flags can control optimizations related to code speed, size, and specific processor features. Experimenting with these options can uncover significant performance improvements but requires careful consideration of the trade-offs involved. For example, aggressive optimization for speed (`-O3` in GCC and Clang) can increase binary size, potentially impacting cache performance. Tailoring compilation parameters to the specific needs of the application and the target hardware is essential for achieving optimal performance.

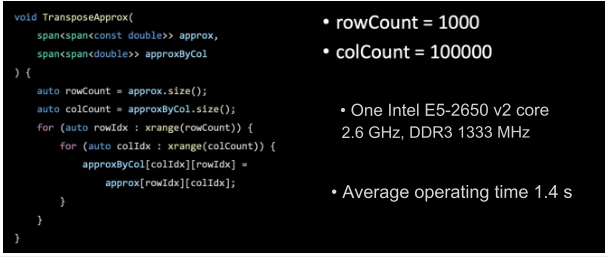
### Compiler-Based Functions Optimizations[[15]](#footnote-15)

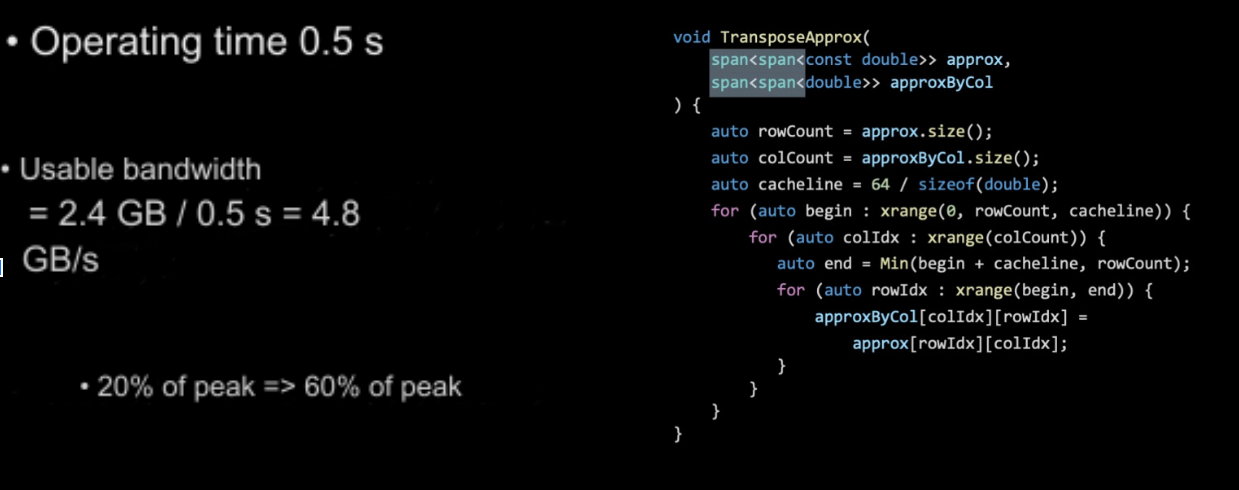
Compiler intrinsics allow direct access to specialized processor instructions that can significantly accelerate certain operations, such as mathematical calculations, memory manipulation, or SIMD (Single Instruction, Multiple Data) operations. Using intrinsics requires a deep understanding of the target hardware and the specific performance characteristics of the operations being optimized. While they can offer substantial performance boosts, intrinsics also reduce code portability and readability, making them most suitable for low-level optimization of performance-critical code sections.

### Cache-Based Optimizations[[16]](#footnote-16)

That technique aims to optimize performance bottlenecks due to non-optimal memory access patterns. To optimize this, it is critical to align data writes with the architecture's cache line size. This is achieved by reorganizing the data writing process, ensuring it aligns with a cache line size. This approach significantly reduces cache misses and enhances performance by improving data locality and ensuring efficient use of the cache.

In the given example it reaches +400% code speedup.





### Memory Pooling[[17]](#footnote-17)

Memory pooling involves pre-allocating a large block of memory and then manually managing allocation and deallocation within this block. This reduces the overhead associated with frequent calls to new and delete.

**Chapter 3**

# K-Mismatch Background

The journey through the landscape of string matching algorithms reveals a lot of features related to innovation and adaptation, underscored by the quest to meet the ever-evolving demands of data processing and analysis. This section aims to describe the historical development of these algorithms, focusing on the challenges and limitations beyond the way for the K-Mismatch algorithm. Understanding this progression helps to understand the critical role of memory efficiency in algorithm design, which significantly influences performance and usability across diverse application domains.

## Early Developments

The inception of string matching algorithms can be traced back to simple brute-force techniques. These methods, while straightforward, demonstrated significant inefficiencies, especially with large texts and patterns, due to their exhaustive search approach. The realization of these inefficiencies spurred the development of more refined algorithms, such as the Knuth-Morris-Pratt (KMP) algorithm[[18]](#footnote-18), which introduced the concept of preprocessing the pattern to identify potential matches without re-examining each character of the text.

## Addressing Mismatches

Despite the advancements, a critical challenge remained: handling mismatches effectively. Traditional algorithms excelled in exact matching scenarios but faltered when mismatches were allowed—a common requirement in fields like bioinformatics, where genetic sequences do not always align perfectly. This gap underscored the need for algorithms that could tolerate discrepancies between the text and the pattern, leading to the exploration of more flexible and sophisticated approaches.

## Memory Efficiency: A Cornerstone of Algorithm Design

A crucial aspect of the evolution of string matching algorithms, and particularly the K-Mismatch algorithm, is the focus on memory efficiency. The efficiency with which an algorithm utilizes memory resources directly impacts its performance, scalability, and usability, especially in resource-constrained environments or with exceptionally large datasets. Memory-efficient algorithms can process large volumes of data more swiftly and with fewer resources, making them unexceptable part of today's data-intensive application domains. As part of preprocessing we need to work with MCS (Minimal Coverage Set)[[19]](#footnote-19). MCS is created by considering all combinations of matches and mismatches for a query, ensuring that each combination contains at least one filter from the set. The process involves denoting matches as "1" and mismatches (or don't care) as "0". During the process of creating an MCS with an example where the first filter containing four matches is selected based on a criterion that ensures each combination is "covered" by at least one filter from the set. This approach allows for the generation of a minimal set of filters (MCS) that can cover all combinations considered in the table, thus optimizing the search process by reducing the required memory resources and increasing the speed of search under certain conditions.

## Searching by given filters optimization

Search action in C++ can be significantly enhanced by utilizing data structures that are tailored for efficient lookup operations, in our case suffix trees seem most suitable. This is a data structure that provides extremely fast substring searching capabilities, making it ideal for applications that require frequent text searches, such as text editors, DNA sequence analysis, and pattern matching algorithms.

A suffix tree is constructed from a given text and represents all possible suffixes of that text in a compressed form. Once built, the suffix tree allows for rapid searches of any substring by traversing downward from the root of the tree through edges that match the characters of the substring being searched. This traversal typically operates in O(m) time, where m is the length of the substring, regardless of the size of the original text. This makes suffix trees particularly useful for scenarios where a static text is searched repeatedly for various patterns.

In C++, implementing a suffix tree requires efficient memory management and the intricacies of tree construction, but the performance benefits in search-intensive applications often outweigh these challenges. Using suffix trees, applications can achieve significant performance improvements by reducing the time complexity of search operations from O(n⋅m) in naive implementations to O(m) with the suffix tree, where n is the length of the text. This optimization is crucial for real-time and performance-critical applications.

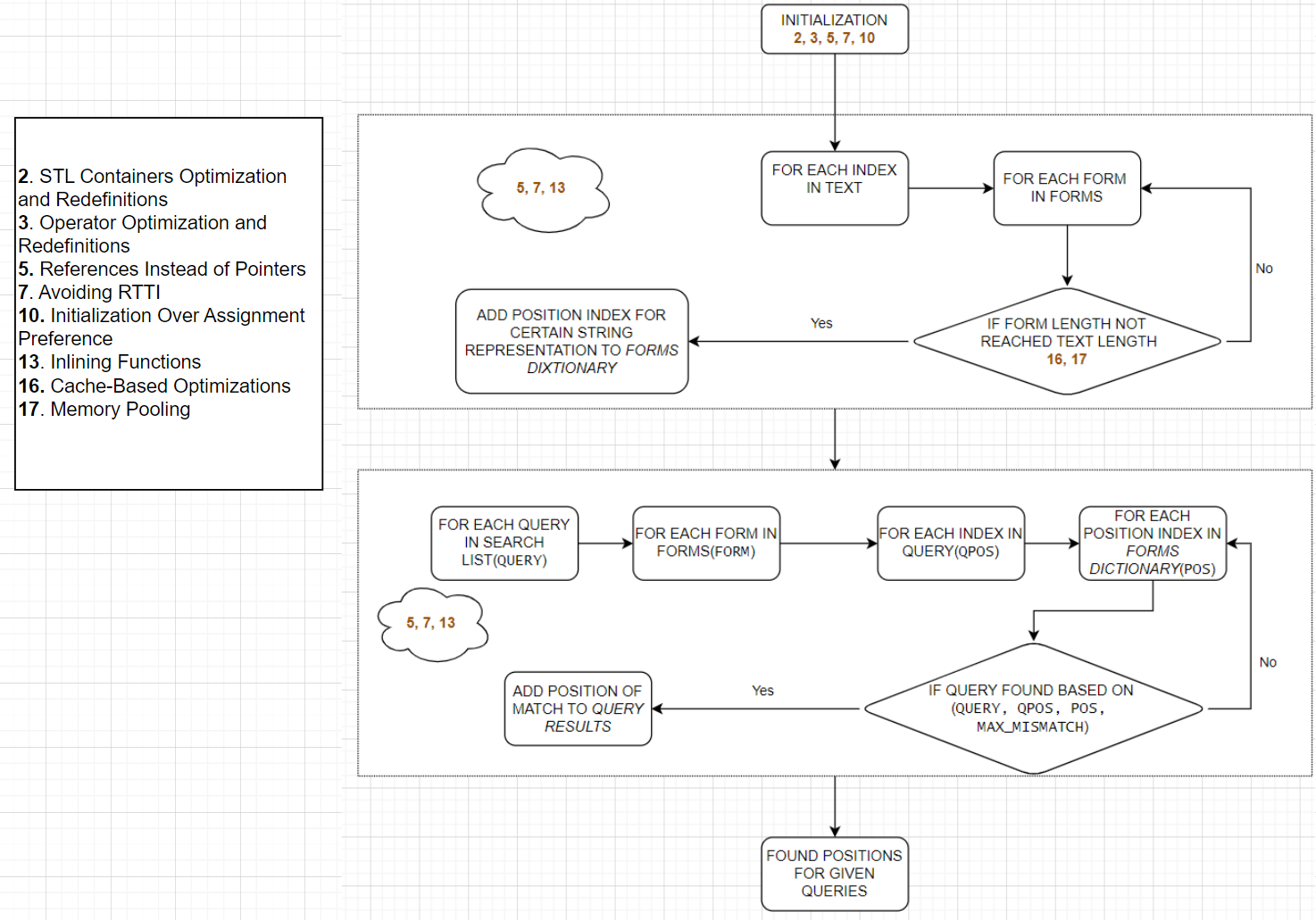
## 

## 

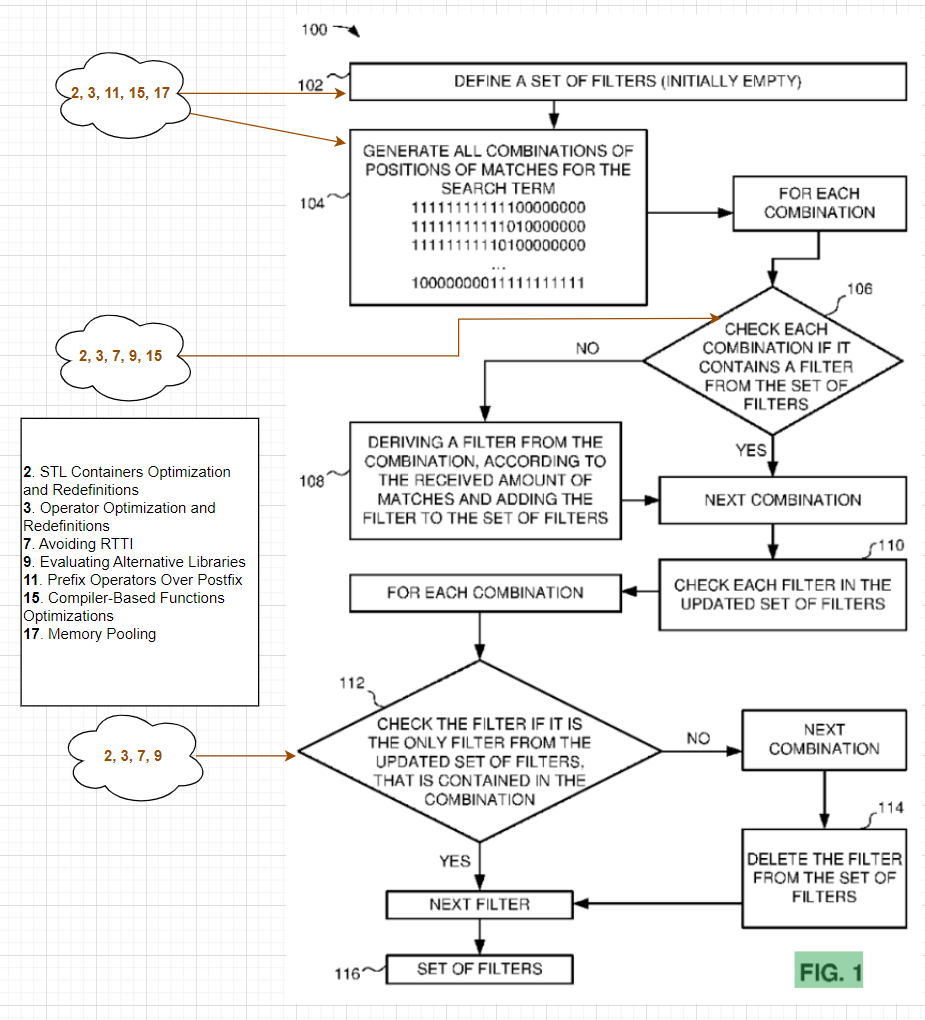
## 

## Optimization path

There are couple of known optimization techniques aren’t directly depending on partial code functionality. Thus, numerous of these methods could be applied directly to whole codebase, together with domain-related techniques. For instance, STL container and operator optimizations are example of full codebase application. In particular cases, possible methods can be applied following the block-scheme below. Indexes are taken from ‘Related Works’ enumeration. Here’s MCS search optimization breakdown:



Similar plan could be applied for building MCS:



# 

**Chapter 4**

# Expected Achievements

Our result expectation is an algorithm implementation that achieves significant reductions in memory usage, enhanced processing speed during handling mismatches. By carefully optimizing memory allocation and management, we’d like to reduce the overall memory footprint of the algorithm, ensuring more efficient use of resources. This optimization not only may benefits the performance of the algorithm but also would allow it to run more smoothly on systems with limited memory capacity.

In addition to memory improvements, our implementation focuses on enhancing processing speed. Through techniques such as algorithmic optimizations, parallel processing, and efficient data structures, we are planning to significantly reduce the time required for the algorithm to execute. This improvement in processing speed is crucial for applications where real-time or near-real-time performance is essential, ensuring that the algorithm can keep up with the demands of the system.

## Criteria for Success

* *Efficient Execution Time and Memory Usage:* The final product must implement the K-Mismatch algorithm with a specific performance target. Memory usage should not exceed 1.5 times the input size, ensuring the algorithm can run on standard desktop environments without requiring excessive resources. Performance should be significantly raised above naive implementation results: *42min 34sec* for *100M* length text, *1000* queries, *25* chars query length, *15%* mismatch factor, *10* alphabet size.
* *Utilization of Advanced C++ Features:* Code optimization should involve specific advanced C++ features such as templates for generic programming, use of the Standard Template Library (STL) for efficient data handling, and smart pointers for memory management. Additionally, implement parallel processing using C++17's standard library features or third-party libraries like Intel TBB for multithreaded execution.
* *Performance Benchmarking:* Include benchmark results comparing this implementation’s performance across versions under various conditions (different lengths of strings, different values of 'K', and varying system specifications). Benchmarks should clearly demonstrate a performance improvement in terms of both speed and memory usage.
* *Robust Error Handling:* The application should include explicit error handling that can manage out-of-range parameters for 'K', non-standard character inputs, and unexpected end-of-file (EOF) markers. Errors should trigger meaningful feedback to the user, explaining the nature of the error and how to resolve it.

**Chapter 5**

# The Process

## Research

When delving into the complex domain of performance optimization, it's crucial to grasp a variety of foundational concepts before one can fully comprehend how performance enhancement tools operate, from the underlying hardware to the software layers. Key topics we are exploring include:

* Study of Established Optimization Techniques: Examine existing and time-tested optimization strategies within the broader scope of programming, with a specific focus on C++. This includes understanding common patterns and algorithms that can enhance performance.
* Optimization in C++ Specific Contexts: Investigate C++-specific optimization practices such as template metaprogramming, efficient use of STL containers, and techniques to minimize overhead in object-oriented programming.
* Libraries and Their Impact on Development: Evaluate various libraries, both standard and third-party, to understand their performance characteristics and suitability for different types of projects. This includes measuring how well different libraries integrate with existing systems and their efficiency under various conditions.
* Compilers and Their Influence on Performance: Examine how different compilers (e.g., GCC, Clang, MSVC) implement optimizations such as loop unrolling, inline expansion, and code vectorization, and how these can be utilized or enhanced in application code.
* Advanced Hardware Architecture Understanding: Gain a deep knowledge of modern hardware architectures which is crucial for advanced optimization. This includes an understanding of how different processor architectures handle computations, explicitly in following context:
  + Caching Mechanisms and Memory Hierarchy: Study the detailed workings of caching mechanisms including L1, L2, and L3 caches, and how data is stored and retrieved in multi-level cache systems. Understanding this can help in designing algorithms that maximize cache hits and minimize cache misses.
  + Processor Vector Instructions and Their Optimization Potential: Learn about specific processor vector instructions such as SIMD (Single Instruction, Multiple Data) that allow parallel processing of data. Knowing how to leverage these can significantly boost performance for data-heavy applications.

## Methodology and Development Process

Development process divides to following steps:

1. **Conceptualization:**
   1. *Requirement Analysis:* Begin with a thorough analysis of requirements including performance targets, usability needs, and typical use cases.
   2. *Algorithm Selection:* Research and evaluate various algorithms and their modifications that solve the K-Mismatch problem. As soon as we have basic implementation, our goal is to explore search suffix tree.
2. **Design:**
   1. *Architecture Planning:* Design the software architecture focusing on modularity and extensibility. Decide on the core components such as the input parser, the K-Mismatch processor, and the output handler.
   2. *Interface Design:* Design a user-friendly command-line interface that allows for easy input of parameters and displays results clearly. Ensure that the interface supports batch processing for efficiency in handling multiple test cases.
   3. *Algorithm Design:* Outline the chosen algorithm's logic in a detailed flowchart or pseudocode, focusing on handling edge cases and optimizing for the worst-case scenarios.
3. **Implementation:**
   1. *Environment Setup:* Set up a development environment with necessary tools and libraries. Choose a C++ compiler that supports C++17 or later for advanced features.
   2. *Core Development:* Start coding the main algorithm using C++. Implement the algorithm as designed, making use of advanced C++ features for performance and memory management (e.g., move semantics, STL containers).
   3. *Interface Implementation:* Develop the command-line interface as per the design, integrating with the core algorithm.
4. **Testing and Evaluation:**
   1. *Sample Text Generation:* Generate a set of predefined random texts varying by length, following appropriate context rules, such as required text complexity, continuity, alphabet, etc., that will be used for further code testing.
   2. *Unit Testing:* Write comprehensive unit tests for each component of the software, particularly focusing on the main functionality. Use a testing framework like Google Test for C++.
   3. *Integration Testing:* Test the integration ability of the solution to ensure they work together with external project as expected. This includes testing the interface with the core algorithm functionality.
   4. *Performance Testing:* Conduct systematic performance tests to measure execution time and memory usage under various conditions. Use profiling tools to identify bottlenecks and optimize them.
5. **Benchmarking:** Compare the performance of the implementation against existing solution using standardized datasets to validate improvements.

**Chapter 6**

# Product

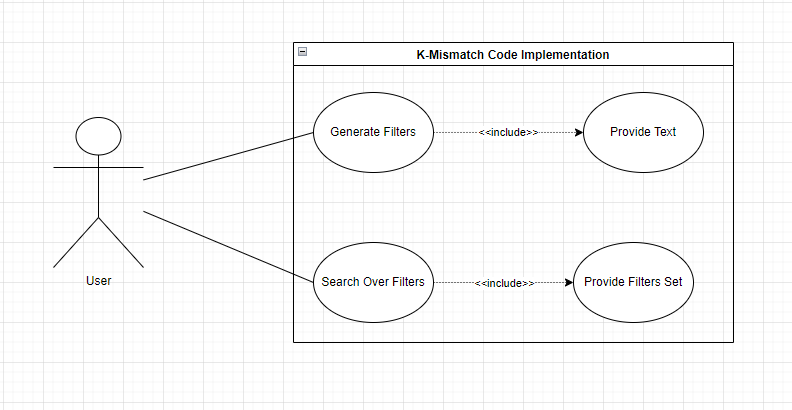
## Functional Requirements

|  |  |
| --- | --- |
| **1** | The system is based on the existing algorithm |
| **2** | The system shall be designed as an exportable C++ library |
| **3** | The system can be used at any PC |
| **4** | The system stores corpus text in memory |
| **5** | The system provides search interface for given corpus text |
| **6** | The system shows the benchmark details of each execution |

## Non-functional Requirements

|  |  |
| --- | --- |
| **1** | Easy to use CLI interface for the user |
| **2** | Easy to integrate to external code |
| **3** | Supports last updated C++ standards |
| **4** | Well-documented and maintainable code |
| **5** | For cloud purposes, will be deployed in Israel region |

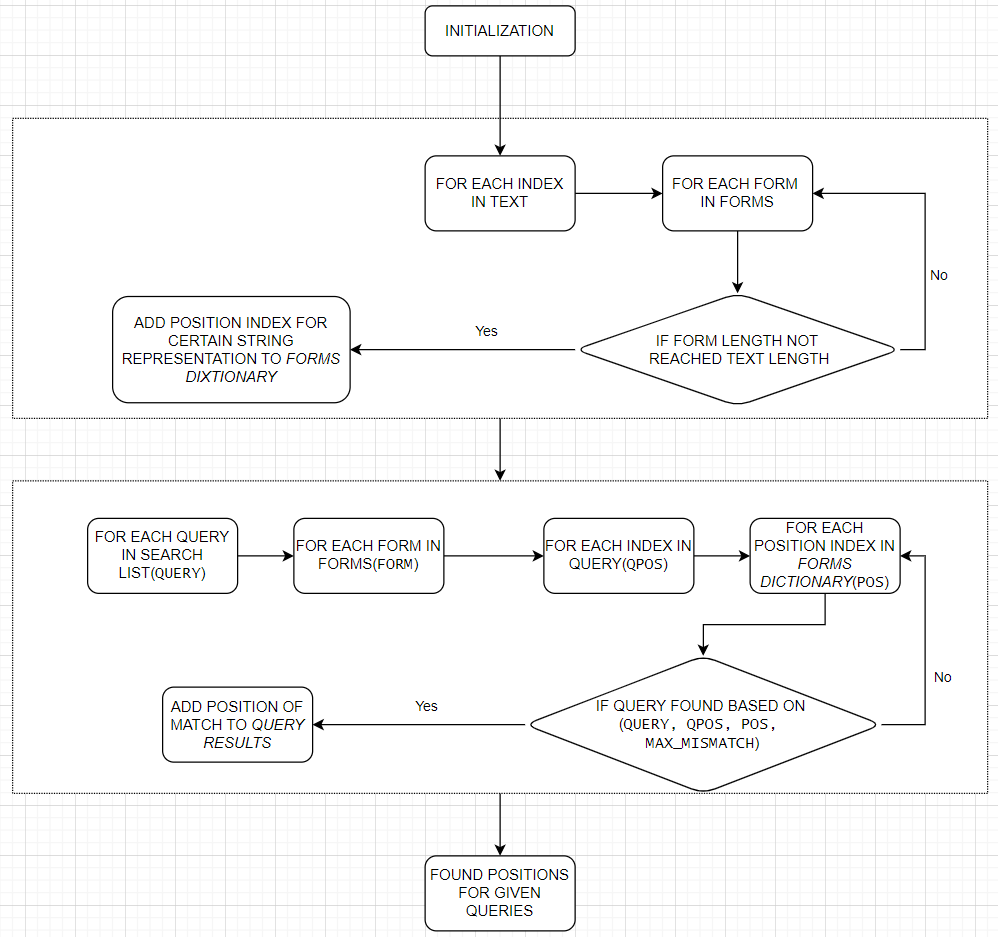
## Use Case Diagram

****

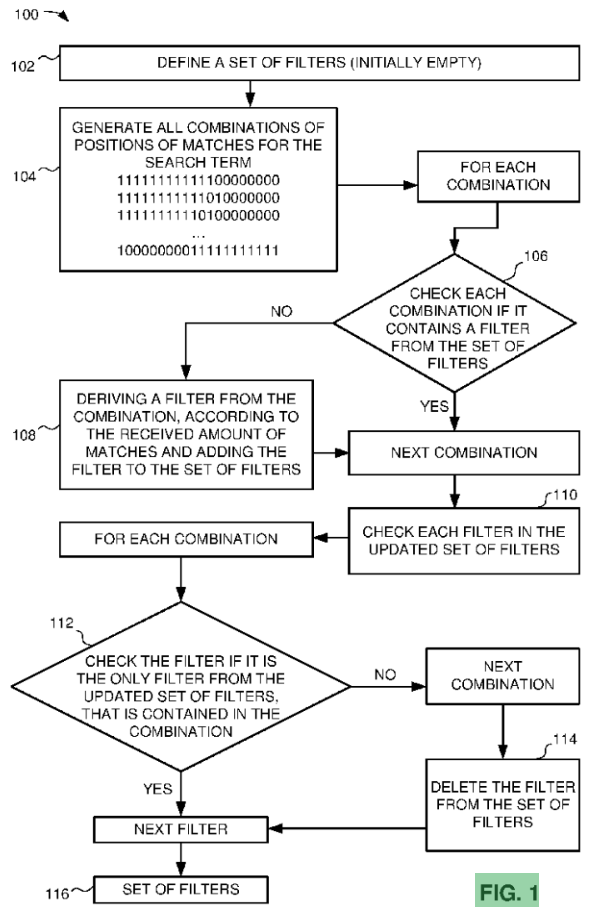
## Architecture Overview

## 

### Naive Search

****

### Generate MCS



**Chapter 7**

# Verification and Evaluation

## Evaluation

The evaluation will be primarily concentrated on the initial execution of the code, ensuring it functions effectively across different hardware platforms, regardless of the specific configurations. The main issue we’re interested to evaluate is successful running of the program at reasonable time in for user purpose. Additionally, another parameter will be checked is portability of the code as an external library following C++ import standards. Together with proper working CLI it will allow to get full control over the provided functionality.

## Verification

Verification process including how the implementation was tested for correctness, reliability, and adherence to the specified requirements. First, we will implement unit tests to validate each function and class independently, focusing on the correctness of individual components and their interactions. These tests will help us verify the logic and functionality at the smallest units of the codebase, catching errors early in the development cycle.

Next, we will incorporate functional testing to ensure that the software behaves as expected when all components work together. We will use manually generated and confirmed input-output pairs for each algorithm step, from initial text to set of filters and, finally, search output, and compare them with code results.

In addition to functional correctness, we will enforce strict type checking during the development phase. This will involve strict checks of the types of variables to prevent type-related errors and ensure data integrity throughout the application. We will leverage C++'s strong typing system to catch any mismatches or inappropriate usage of variable types at compile time.

Moreover, we will focus on the correct export mechanisms of libraries. This entails ensuring that our software components are modular and that external and internal libraries are correctly linked, avoiding issues like symbol conflicts or missing dependencies. Proper management of library exports is crucial for the stability and portability of the software.

Overall, our verification plan is going to cover all critical aspects of the software, from the smallest unit to the complete system, ensuring high-quality, reliable, and working software.

# References

1. Knuth, D. E., Morris, J. H., & Pratt, V. R. (1977). *Fast pattern matching in strings*. SIAM Journal on Computing, 6(2), 323–350. <https://doi.org/10.1137/0206024>
2. Frenkel, Z., Volkovich, Z., & Ltd, O.-. E. R. a. D. (2017, March 19). WO2018173042A1 - *System and method for generating filters for k-mismatch search*. Google Patents. <https://patents.google.com/patent/WO2018173042A1>
3. Kluev, L. (2020, October 15). *C++ optimization: combining speed and high level. Yandex report.* Habr. <https://habr.com/ru/companies/yandex/articles/522900/>
4. *XL C/C++ for AIX 13.1.3.* (n.d.). <https://www.ibm.com/docs/en/xl-c-and-cpp-aix/13.1.3?topic=guide-coding-your-application-improve-performance>
5. Isensee, P. *C++ Optimization Strategies and Techniques.* (n.d.). <https://www.tantalon.com/pete/cppopt/main.htm>
6. *Improving performance with SIMD intrinsics in three use cases - Stack Overflow*. (2020, July 8). <https://stackoverflow.blog/2020/07/08/improving-performance-with-simd-intrinsics-in-three-use-cases/>
7. Qrdl. (2021, February 14). *FizzBuzz like senior.* Habr. <https://habr.com/ru/articles/540136/>
8. Holopainen, J., Bjork S. *Patterns in Game Design (Game Development Series) (2004, December 21)*.

1. C++ optimization: combining speed and high level. Yandex report - [link](https://habr.com/ru/companies/yandex/articles/522900/) [↑](#footnote-ref-1)
2. Pete Isensee - [Take Advantage of STL Containers](https://www.tantalon.com/pete/cppopt/design.htm#STLContainers) [↑](#footnote-ref-2)
3. Pete Isensee - [Pass Class Parameters by Reference](https://www.tantalon.com/pete/cppopt/asyougo.htm#PassClassParametersByReference) [↑](#footnote-ref-3)
4. Patterns In Game Design. Staffan Bjork, Jussi Holopainen [↑](#footnote-ref-4)
5. Pete Isensee - [Consider Using References Instead of Pointers](https://www.tantalon.com/pete/cppopt/design.htm#PreferReferencesInsteadOfPointers) [↑](#footnote-ref-5)
6. Pete Isensee - [Limit Exception Handling](https://www.tantalon.com/pete/cppopt/design.htm#ExceptionHandling) [↑](#footnote-ref-6)
7. Pete Isensee - [Avoid Runtime Type Identification](https://www.tantalon.com/pete/cppopt/design.htm#RTTI) [↑](#footnote-ref-7)
8. Pete Isensee - [Prefer stdio to iostream (printf vs. cout)](https://www.tantalon.com/pete/cppopt/design.htm#StdioVsIOStream) [↑](#footnote-ref-8)
9. Pete Isensee - [Evaluating Alternative Libraries](https://www.tantalon.com/pete/cppopt/design.htm#AlternativeLibraries) [↑](#footnote-ref-9)
10. Pete Isensee - [Prefer Initialization over Assignment](https://www.tantalon.com/pete/cppopt/asyougo.htm#PreferInitialization) [↑](#footnote-ref-10)
11. Pete Isensee - [Relative Costs of Common Programming Operations](https://www.tantalon.com/pete/cppopt/appendix.htm#AppendixA) [↑](#footnote-ref-11)
12. Pete Isensee - [Use Explicit Constructors](https://www.tantalon.com/pete/cppopt/asyougo.htm#UseExplicitConstructors) [↑](#footnote-ref-12)
13. Pete Isensee - [Inline Functions](https://www.tantalon.com/pete/cppopt/final.htm#InlineFunctions) [↑](#footnote-ref-13)
14. Pete Isensee - [Compiler Optimization](https://www.tantalon.com/pete/cppopt/compiler.htm) [↑](#footnote-ref-14)
15. Stackoverflow. Konstantin - [Improving performance with SIMD intrinsics in three use cases](https://stackoverflow.blog/2020/07/08/improving-performance-with-simd-intrinsics-in-three-use-cases/) [↑](#footnote-ref-15)
16. C++ optimization: combining speed and high level. Yandex report - [link](https://habr.com/ru/companies/yandex/articles/522900/) [↑](#footnote-ref-16)
17. IBM Documentation - [Managing memory with multiple heaps](https://www.ibm.com/docs/en/xl-c-and-cpp-aix/13.1.3?topic=heaps-managing-memory-multiple) [↑](#footnote-ref-17)
18. Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. “Fast Pattern Matching in Strings.” - [link](https://epubs.siam.org/doi/10.1137/0206024) [↑](#footnote-ref-18)
19. Zakharia FRENKEL, Zeev VOLKOVICH. “System and method for generating filters for k-mismatch search” - [link](https://patents.google.com/patent/WO2018173042A1/en#:~:text=This%20is%20evaluated%20as%20a%20function%20of%20the%20number%20of%20filters%20in%20the%20minimal%20filter%20set%20(MCS).) [↑](#footnote-ref-19)