Software Engineering Department
ORT Braude College

Capstone Project Phase B

K-Mismatch Algorithm Optimization and Application

Project Number

24-1-D-13

Supervisor: Dr. Zakharia Frenkel

Vladimir Brustinov (vladimir.brustinov@e.braude.ac.il)

Nik Lerman (nik.lerman@e.braude.ac.il)

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

Table of Contents

Project Book

General Description

The K-Mismatch Algorithm Optimization and Application project aims to develop an efficient implementation of the K-Mismatch string searching algorithm, focusing on performance optimization and practical application. This system is designed to address the growing need for fast and memory-efficient string matching algorithms that can handle mismatches, which is crucial in fields such as computational biology, cybersecurity, and text processing.

The primary objectives of the system are:

1. To implement the K-Mismatch algorithm with performance improvements over naive implementations results: *42min 34sec* for *100M* length text, *1000* queries, *25* chars query length, *15%* mismatch factor, *10* alphabet size.
2. To utilize advanced C++ features for enhanced efficiency and maintainability.
3. Adjust benchmark results comparing this implementation's performance across versions under various conditions (different lengths of strings, different values of 'K', and varying system specifications).

The system is implemented as a C++ library, making it suitable for integration into larger software systems or for use as a standalone tool. The target users of this system include:

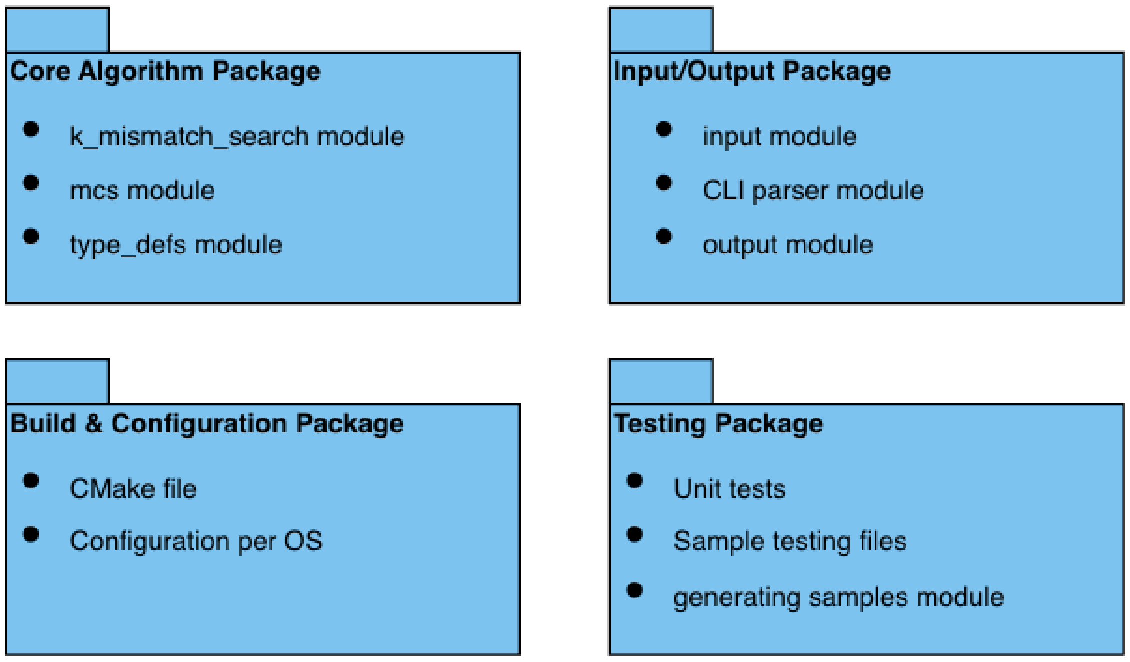
- Bioinformaticians working with genetic sequences
- Cybersecurity professionals analyzing large volumes of text data
- Software developers requiring efficient string matching capabilities in their applications
- Researchers in text processing and information retrieval

Solution Description

Solution Architecture (Package/Deployment Level)

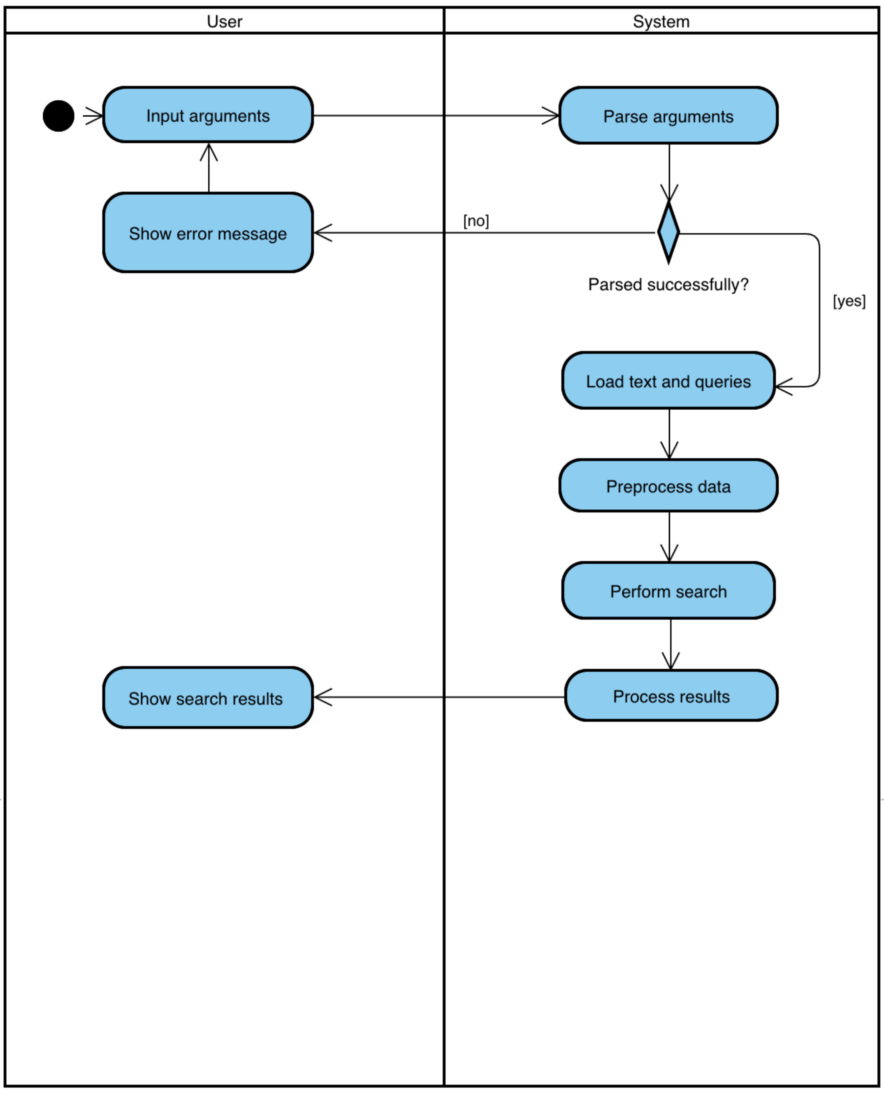
The K-Mismatch Algorithm Optimization and Application system is structured into the following high-level components:

1. Core Algorithm Package
   - k_mismatch_search module
   - mcs (Minimal Coverage Set) module
   - type_defs module
2. Input/Output Package
   - input module
   - CLI parser module
   - output module
3. Build and Configuration Package
   - CMake configuration across different OSs
4. Testing Package
   - Unit tests
   - Sample input files for testing
   - gen_samples module for generating random text and queries



System Operation Flow (Activity Diagram)

1. Start Application
2. Parse Command-Line Arguments
   - If invalid, display usage information and exit
3. Load Input Data
   - Read text file
   - Read queries file
4. Preprocess Data
   - Generate Minimal Coverage Set (MCS) using mcs module
   - Generate Search cache (index text)
5. Perform K-Mismatch Search
   - For each query:
     a. Apply MCS filters
     b. Execute search on index
     c. Collect results
6. Process Results
   - Aggregate search outcomes
   - Prepare output format
7. Output Results
   - Write to output file or display on console



Research and Development Process

Our research and development process for the K-Mismatch Algorithm Optimization and Application project was comprehensive and methodical, involving several key stages:

1. Literature Review and Algorithm Analysis:
   We began with an extensive review of existing string matching algorithms, focusing on those that handle mismatches. We paid particular attention to the new k-mismatch algorithm based on spaced q-grams developed in our department, which formed the foundation of our project.
2. Algorithm Refinement:
   We focused on refining the core K-Mismatch algorithm, implementing the following optimizations:
   - MCS Generation Optimization: We improved the MCS generation process, ensuring it creates a set of filters for the search process in optimal time.
   - Alternative Data Structure Implementation: While initially considering a Suffix Tree implementation, we opted for a different approach due to the complexity of balancing memory usage with performance gains in our specific use case. Instead, we focused on optimizing our existing data structures to achieve similar performance benefits.
3. Parallel Processing Implementation:
   Leveraging C++17's parallel processing features, we implemented parallel execution for suitable parts of the algorithm, particularly in the search phase where multiple patterns can be processed concurrently.
4. Code optimization:
   Based on our analysis and the techniques outlined in our Related Works section, we applied various C++-specific optimizations. This included:
   - STL Container Optimization: We carefully evaluated and optimized our use of STL containers, particularly in the MCS (Minimal Coverage Set) generation process and used most optimized and suits for better performance.
   - Compiler-Based Functions And Cache Optimization: We reorganized data structures and algorithms to improve cache utilization, aligning data writes with the architecture's cache line size where possible.
   - Operator Optimization and Redefinitions: Usage of given subject allowed to achieve more code readability in low-level operations and avoid casting overhead compute costs and manual calculation of repeatable elements, especially binary sequences.
   - Using References Instead of Pointers: As a part of C++ usage advanced features, we used references instead of pointers and passing by values, allowing us to reduce the number of objects created and avoiding unnecessary overhead.
   - Evaluating Alternative Libraries: While considering external libraries usage such as ‘boost’, we found them adding dependency overhead and level out its advantages in relation to our context.

    - Initialization Over Assignment Preference: This minor technique suggests initializing objects in relevant places and carefully used as well.
    - Inline Functions: Using inline functions allowed to improve program performance by eliminating the overhead of function calls, as the compiler replaces the function call with the actual function code at compile time. Additionally, they lead to more efficient code optimization, as the compiler has direct access to the function's implementation when inlining, allowing for better instruction scheduling and register allocation.
    - Compiling Parameter Optimizations: Since we are working mainly with strings, we found that most compiling optimizations are related to math computation and not suitable for our purposes.
    - Use of move semantics to reduce unnecessary copying of large objects.
    - Careful application of inline functions to reduce function call overhead in critical sections.
5. Cross-Platform Testing:
   Given the importance of portability, we extensively tested our implementation on different platforms (Windows, Linux) to ensure consistent performance and functionality across various environments. We utilized CMake as our build system, which provided several advantages:
    - Platform-independent build configuration: CMake allowed us to define our project structure and dependencies in a platform-agnostic manner.
    - Automatic generation of native build files: CMake generated platform-specific build files streamlining the build process across different environments.
    - Easy integration of third-party libraries: CMake's find_package() command simplified the process of locating and linking external dependencies.
    - Flexible configuration options: We could easily define build options and conditionally compile certain features based on the target platform or user preferences.

Tools Used

- Development Environment: Visual Studio Code with g++14 compiler

- Version Control: Git and GitHub for collaborative development

- Profiling Tools: Various Visual Studio Code extensions for profiling

- Build System: CMake for cross-platform build configuration

- Benchmarking: Various Visual Studio Code extensions for performance measurements

Challenges and Solutions

During the development of our optimized K-Mismatch algorithm implementation, we encountered several significant challenges. Here, we detail these challenges and the solutions we implemented:

- Cache-Based Optimization:
    - Challenge: Initial computations showed poor performance due to underutilization of the cache and processor capabilities.
    - Solution: We optimized the computation by leveraging processor-specific intrinsics to improve cache efficiency. A two-level structure was introduced, where the top level is small enough to remain in cache and optimized for SIMD operations. The lower level focuses on handling larger data sets. These changes reduced cache misses, enhanced parallelism, and significantly improved computational performance.
- Parallel Processing Overhead:
    - Challenge: Initial attempts at parallelization introduced significant overhead, negating performance gains on smaller inputs.
    - Solution: We implemented an adaptive parallelization strategy. For smaller inputs and synchronization dependent code blocks the algorithm runs sequentially, while for larger inputs, it runs in the optimal number of threads based on input size and available hardware.
- Cross-Platform Compatibility:
    - Challenge: Ensuring consistent performance across different platforms proved challenging due to differences in compiler optimizations and system libraries.
    - Solution: We abstracted platform-specific code into separate modules and used CMake to manage build configurations across platforms. We also implemented a set of platform-specific optimizations that are conditionally compiled based on the target system. CMake's flexibility allowed us to easily integrate platform-specific libraries and compile flags, ensuring optimal performance on each target system.
- MCS Generation Efficiency:

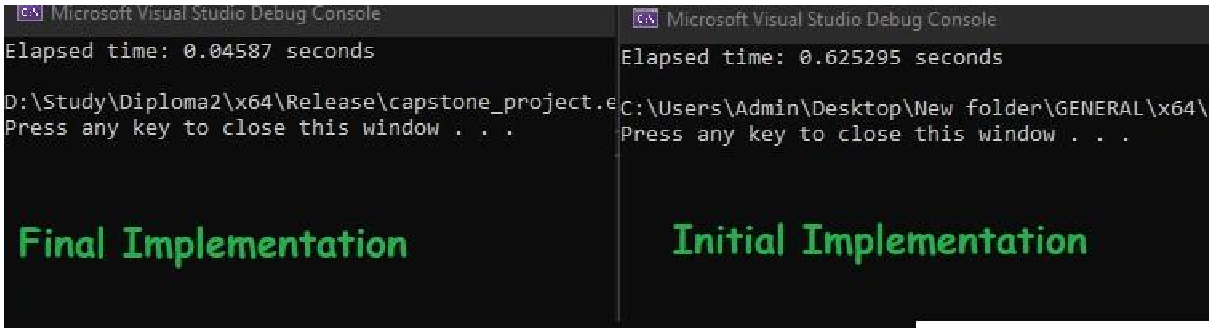
    - Challenge: The generation of the Minimal Coverage Set (MCS) was a significant bottleneck, especially for large pattern sizes and high mismatch tolerances.
    - Solution: We optimized the MCS generation algorithm using bit-parallel operations and dynamic programming techniques. We also implemented a caching mechanism for frequently used MCS configurations, significantly reducing computation time for repeated searches with similar parameters.
- Huge Sequential Texts Processing:
    - Challenge: Processing huge sequential texts presented difficulties in managing memory and maintaining speed without degrading system performance.
    - Solution: This requires strict resource management while preserving processing speed, ensuring smooth handling of large texts without overloading the system.

By addressing these challenges, we were able to create a highly optimized, memory-efficient, and portable implementation of the K-Mismatch algorithm that meets or exceeds our initial performance targets.

Results

- Optimized implementation showed a significant speedup compared to our naive implementation, reducing execution time from 42min 34sec to approximately 10 min for the benchmark case (100M length text, 1000 queries, 25 chars query length, 15% mismatch factor, 10 alphabet size).
- Together with performance results, we bring CLI interface representing crucial basis to future developments and project expanding.
- Our optimized implementation showcases significant performance improvements, achieving a remarkable 14x speedup in index generation, allowing for faster data indexing and retrieval. Even more impressively, we observe an outstanding 750,000x speedup in the creation of MCS combinations, drastically reducing computation time and enhancing overall efficiency.

Indexing:


Microsoft Visual Studio Debug Console
Microsoft Visual Studio Debug Console

```
Elapsed time: 0.04587 seconds

D:\Study\Diploma2\x64\Release\capstone_project.e
Press any key to close this window . . .
```

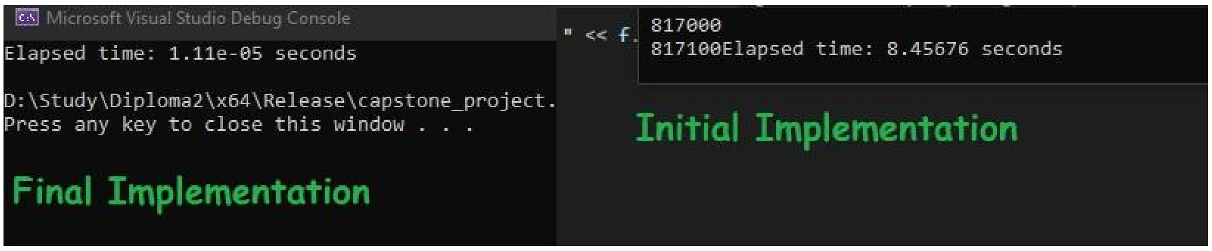
Final Implementation

```
Elapsed time: 0.625295 seconds

C:\Users\Admin\Desktop\New folder\GENERAL\x64\
Press any key to close this window . . .
```

Initial Implementation

MCS generation:



```
Microsoft Visual Studio Debug Console
Elapsed time: 1.11e-05 seconds

D:\Study\Diploma2\x64\Release\capstone_project.
Press any key to close this window . . .
```

Final Implementation

```
" << f.
817000
817100Elapsed time: 8.45676 seconds
```

Initial Implementation

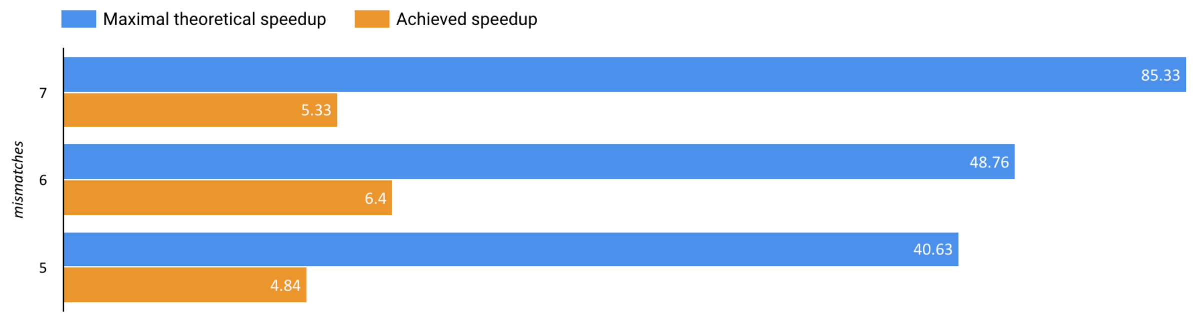
- Following comparison with the theoretical predicted speedup limit (formula of expected comparisons for search, retrieved from the patent content), we managed to get closer to that number, as shown in graphs below.

Naive Algorithm:
Number of Comparisons: $n \times m$

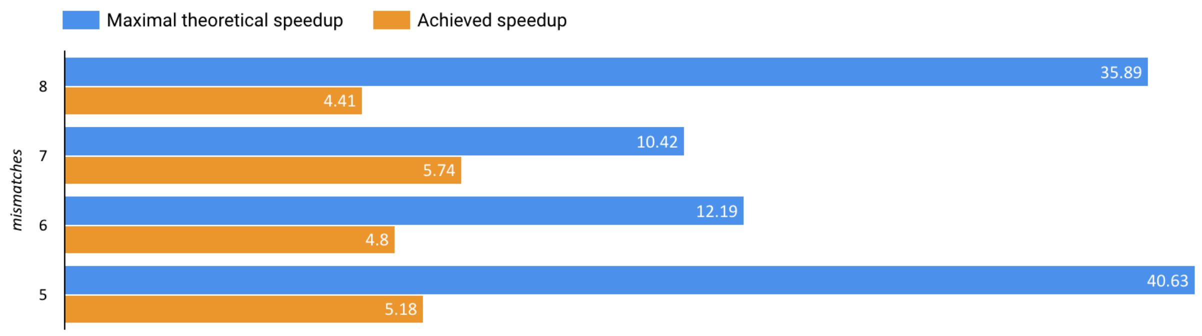
K-Mismatch Algorithm:
Number of Comparisons: $n \times \frac{1}{A^f} \times \sum_{\text{filters}} (m - \text{filter\_size} + 1)$

n - text length, m - substring length, A - alphabet size, f - matches in filter

Text Length: 10000 | Alphabet size: 4 | Number of queries: 10 | Query length: 10



Text Length: 100000 | Alphabet size: 4 | Number of queries: 10 | Query length: 10



Lessons Learned

1. Balance Between Algorithmic and Low-Level Optimizations:
   We discovered that achieving optimal performance required a careful balance between high-level algorithmic improvements and low-level code optimizations. While algorithmic changes often provided the most significant performance gains, fine-tuning at the code level was crucial for squeezing out additional performance. In retrospect, we would have allocated more time for low-level optimizations earlier in the development process.
2. Importance of Cross-Platform Testing:
   Our commitment to cross-platform compatibility taught us the value of continuous cross-platform testing. We encountered several instances where optimizations that significantly improved performance on one platform had negligible or even negative impacts on others, for example possibility of usage optimization flags for compilation. This reinforced the importance of maintaining a diverse testing environment throughout the development process.
3. Value of Modular Design in Optimization:
   The modular structure we adopted for our implementation proved invaluable during the optimization process. It allowed us to easily swap out different implementations of key components for benchmarking and iterative improvement. In future projects, we would emphasize this modular approach even more from the outset.
4. Code Readability in Performance-Critical Code:
   We learned that maintaining readable code is particularly crucial in highly optimized, performance-critical sections. What seemed like clear optimizations during development could become opaque and difficult to maintain without proper definitions. In future, we would allocate more time for documenting our optimization strategies and the reasoning behind specific implementation choices.
5. Value of Academic-Industry Collaboration:
   Our project, which built upon academic research (the new k-mismatch algorithm based on spaced q-grams) while focusing on practical, high-performance implementation, highlighted the value of bridging academic research with industry-focused optimization techniques. This experience has motivated us to seek out more opportunities for such collaborations in the future.
6. Continuous Learning in Optimization Techniques:
   The field of algorithm optimization, especially in the context of modern hardware architectures, is rapidly evolving. We learned the importance of staying updated with the latest research and industry best practices. Allocating time for continuous learning and experimentation with new optimization techniques would be a priority in future projects.
7. Importance of Benchmarking Against Existing Solutions:
   Regularly benchmarking our implementation against existing solutions provided valuable context for our improvements and helped us set realistic goals. In future projects, we would establish a more comprehensive benchmarking suite earlier in the development process to guide our optimization efforts more effectively.

These lessons have not only contributed to the success of our K-Mismatch algorithm optimization project but have also significantly enhanced our skills and approach to software development and algorithm optimization. We are confident that applying these learnings will lead to even more efficient and effective solutions in our future endeavors.

User Guide

Installation

1. Clone the repository:

   git clone https://github.com/keepthegoal/24-1-D-13.git

   cd 24-1-D-13

2. Create a build directory and navigate to it:

   mkdir build && cd build

3. Configure the project with CMake:

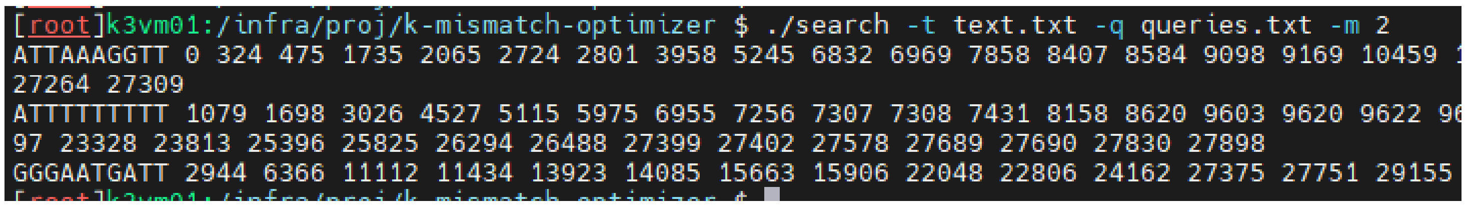
   cmake ..

4. Build the project:

   cmake --build . --config Release

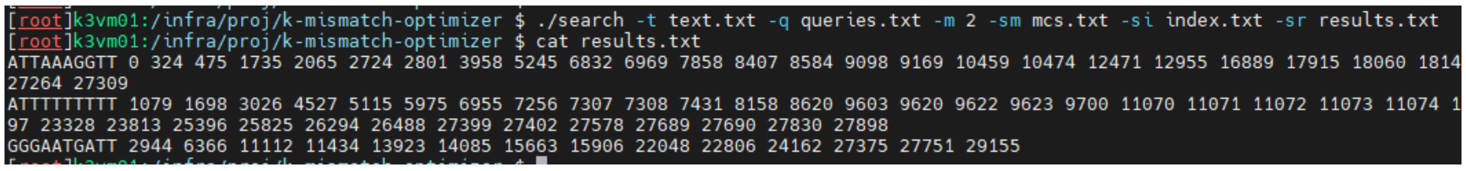
Usage examples

Searching from scratch, without saving MCS, index and results files:



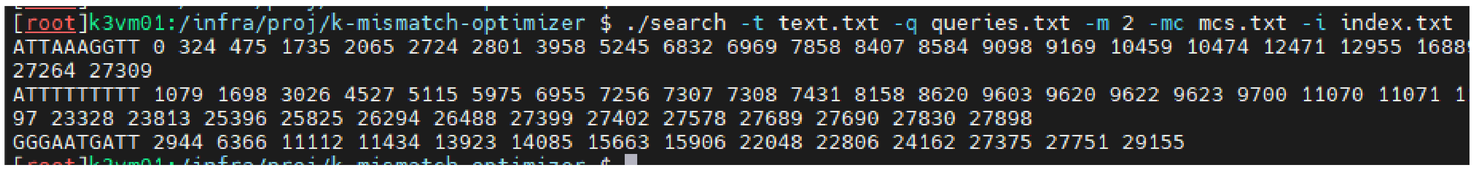
```
[root]k3vm01:/infra/proj/k-mismatch-optimizer $ ./search -t text.txt -q queries.txt -m 2
ATTAAAGGTT 0 324 475 1735 2065 2724 2801 3958 5245 6832 6969 7858 8407 8584 9098 9169 10459 1
27264 27309
ATTTTTTTTT 1079 1698 3026 4527 5115 5975 6955 7256 7307 7308 7431 8158 8620 9603 9620 9622 96
97 23328 23813 25396 25825 26294 26488 27399 27402 27578 27689 27690 27830 27898
GGGAATGATT 2944 6366 11112 11434 13923 14085 15663 15906 22048 22806 24162 27375 27751 29155
```

Searching from scratch with saving all results files



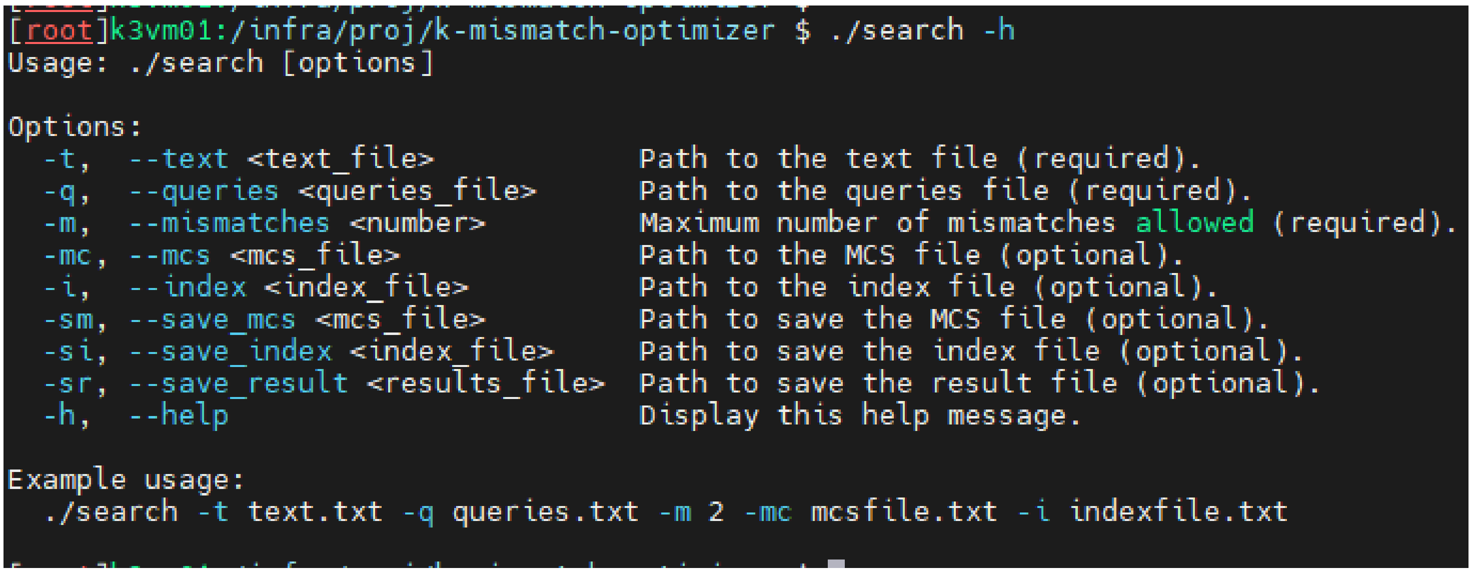
```
[root]k3vm01:/infra/proj/k-mismatch-optimizer $ ./search -t text.txt -q queries.txt -m 2 -sm mcs.txt -si index.txt -sr results.txt
[root]k3vm01:/infra/proj/k-mismatch-optimizer $ cat results.txt
ATTAAAGGTT 0 324 475 1735 2065 2724 2801 3958 5245 6832 6969 7858 8407 8584 9098 9169 10459 10474 12471 12955 16889 17915 18060 1814
27264 27309
ATTTTTTTTT 1079 1698 3026 4527 5115 5975 6955 7256 7307 7308 7431 8158 8620 9603 9620 9622 9623 9700 11070 11071 11072 11073 11074 1
97 23328 23813 25396 25825 26294 26488 27399 27402 27578 27689 27690 27830 27898
GGGAATGATT 2944 6366 11112 11434 13923 14085 15663 15906 22048 22806 24162 27375 27751 29155
```

Search with precomputed data usage (MCS and index files):



```
[root]k3vm01:/infra/proj/k-mismatch-optimizer $ ./search -t text.txt -q queries.txt -m 2 -mc mcs.txt -i index.txt
ATTAAAGGTT 0 324 475 1735 2065 2724 2801 3958 5245 6832 6969 7858 8407 8584 9098 9169 10459 10474 12471 12955 1688
27264 27309
ATTTTTTTTT 1079 1698 3026 4527 5115 5975 6955 7256 7307 7308 7431 8158 8620 9603 9620 9622 9623 9700 11070 11071 1
97 23328 23813 25396 25825 26294 26488 27399 27402 27578 27689 27690 27830 27898
GGGAATGATT 2944 6366 11112 11434 13923 14085 15663 15906 22048 22806 24162 27375 27751 29155
```

Command-line interface overview



```
[root]k3vm01:/infra/proj/k-mismatch-optimizer $ ./search -h
Usage: ./search [options]

Options:
  -t,  --text <text_file>            Path to the text file (required).
  -q,  --queries <queries_file>      Path to the queries file (required).
  -m,  --mismatches <number>         Maximum number of mismatches allowed (required).
  -mc, --mcs <mcs_file>              Path to the MCS file (optional).
  -i,  --index <index_file>          Path to the index file (optional).
  -sm, --save_mcs <mcs_file>         Path to save the MCS file (optional).
  -si, --save_index <index_file>     Path to save the index file (optional).
  -sr, --save_result <results_file>  Path to save the result file (optional).
  -h,  --help                        Display this help message.

Example usage:
  ./search -t text.txt -q queries.txt -m 2 -mc mcsfile.txt -i indexfile.txt
```

Maintenance Guide

System Environment

The K-Mismatch Algorithm Optimization and Application system requires the following environment:

- Operating System: Windows 7 or newer / Linux (Ubuntu 20.04+ recommended)

- RAM: Minimum 8GB, 16GB or more recommended for large datasets

- Storage: HDD (SSD recommended) with at least 1GB free space for the software and working files

Software Dependencies

- C++17 compatible compiler (GCC 7.3+, Clang 5.0+, or MSVC 19.14+)

- CMake 3.12 or higher

Installation Instructions

1. Clone the repository:

```
git clone https://github.com/keepthegoal/24-1-D-13.git
cd 24-1-D-13
```

2. Create a build directory and navigate to it:

```
mkdir build && cd build
```

3. Configure the project with CMake:

```
cmake ..
```

4. Build the project:

```
cmake --build . --config Release
```

Maintenance Procedures

1. Updating the software:

   - Pull the latest changes from the repository

   - Rebuild the project following steps 2-4 from the installation instructions

2. Adding new features:

   - Implement new functionality in the appropriate module

   - Document the new feature in the user guide

3. Extending the algorithm:

   - Modifications to the core algorithm should be made in the `src/k_mismatch_search.cpp` or `src/mcs.cpp` files

   - Ensure all changes are thoroughly tested

References

1. 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
2. Cmake: cross-platform free and open-source software for build automation, testing, packaging and installation of software by using a compiler-independent method.
3. g++: GNU project C and C++ compiler