Enhancing the Efficiency of Spaced Seed Hashing in Bioinformatics Using Set Covering and Operations Research Methods

Reihaneh Baghishani & Shabnam Zareshahraki
University of Padua
reihaneh.baghishani@studenti.unipd.it
shabnam.zareshahraki@studenti.unipd.it

Abstract

The efficient hashing of spaced seeds is a critical challenge in bioinformatics, impacting tasks such as sequence classification, alignment-free similarity searches, and metagenomic analysis. Spaced seeds offer improved sensitivity over traditional k-mers by allowing for mismatches and gaps, but their computational cost is significantly higher. This paper proposes a novel approach to enhance the efficiency of spaced seed hashing by integrating set covering and operations research (OR) methods. We formulated the Set Covering problem and implemented the solution using Python and CPLEX. Our experimental results demonstrate that while the Set Covering approach incurs higher computational overhead compared to the Iterative Spaced Seed Hashing (ISSH) algorithm, it provides structured optimization for spaced seed selection. The robustness and comprehensive features of the CPLEX library, maintained over the years by IBM, contribute significantly to the reliability and effectiveness of our method. Future work will focus on addressing computational challenges and empirically validating our approach through extensive testing, aiming to leverage the full potential of OR methods to optimize bioinformatics algorithms.

Introduction

The efficient hashing of spaced seeds is a critical challenge in bioinformatics, particularly for sequence classification, alignment-free similarity searches, and metagenomic analysis, among other tasks. Spaced seeds, which incorporate wildcards at specific positions, offer improved sensitivity over traditional k-mers by allowing for mismatches and gaps, thus better-capturing sequence similarities. However, the computational cost of hashing spaced seeds is significantly higher than that of k-mers, primarily due to the lack of efficient hashing mechanisms that can leverage the overlaps between consecutive spaced seeds.

In recent years, several methods have been proposed to address this challenge. The Fast-Spaced Seed Hashing (FSH) algorithm and the Iterative Spaced Seed Hashing (ISSH) algorithm represent notable advancements in this area [Girotto, Comin, and Pizzi(2018), Petrucci et al.(2020)Petrucci, Noé, Pizzi, and Comin]. Despite their improvements, there remains a substantial gap between the efficiency of spaced seed hashing and k-mer hashing. This paper explores a novel approach using set covering and operations research (OR) methods to further enhance the efficiency of spaced-seed hashing. By leveraging these mathematical and algorithmic techniques, we propose a solution that not only narrows the performance gap but also offers a structured framework for optimizing spaced seed selection and hashing.

Background and Related Work

Spaced seeds have become a standard tool in bioinformatics due to their enhanced sensitivity in detecting sequence similarities compared to contiguous k-mers. A spaced seed is a binary string where certain positions are designated as match positions (1) and others as wildcard positions (0). This allows spaced seeds to capture non-contiguous matches, thereby increasing the likelihood of detecting homologous sequences that may contain mutations or sequencing errors.

The primary challenge with spaced seeds lies in their computational inefficiency. Unlike k-mers, where consecutive k-mers share a large overlap that can be exploited for efficient hashing, spaced seeds do not have a straightforward overlapping structure. As a result, traditional hashing methods for spaced seeds involve recomputing the hash values from scratch for each position, leading to significant computational overhead.

Fast Spaced Seed Hashing (FSH)

The FSH algorithm addresses this issue by exploiting the similarity between hash values of spaced seeds at adjacent positions. By leveraging the overlap between consecutive spaced-seeds, FSH can compute new hash values more efficiently. Experiments have shown that FSH can achieve speedups ranging from 1.6x to 5.3x compared to traditional methods, depending on the structure of the spaced seeds. [Girotto, Comin, and Pizzi(2018)]

Iterative Spaced Seed Hashing (ISSH)

ISSH further refines this approach by introducing an iterative algorithm that maximizes the reuse of previously computed hash values. The algorithm combines multiple previous hashes to compute new hash values, significantly reducing the number of symbols that need to be encoded. ISSH has demonstrated speedups in the range of 3.5x to 7x, outperforming previous methods and bringing the performance of spaced seed hashing closer to that of k-mer hashing.

Despite these advancements, there is still room for improvement. The current methods do not fully leverage the potential of optimization techniques from operations research, which could provide a more systematic and efficient framework for spaced seed hashing. In the following sections, we propose an approach that integrates set covering and OR methods to address this challenge. [Petrucci et al.(2020)Petrucci, Noé, Pizzi, and Comin]

Proposed Method

To further enhance the efficiency of spaced seed hashing, we propose a novel approach that integrates set covering and operations research (OR) methods. The goal is to optimize the selection and arrangement of spaced seeds to minimize computational overhead while ensuring comprehensive sequence coverage.

Set Covering Problem

The Set Covering Problem (SCP) is a well-known problem in combinatorial optimization, and it is extensively discussed in "Introduction to Mathematical Optimization" by Dr. Matteo Fischetti. SCP involves finding the smallest subset of sets that cover all elements in a universal set. This problem is NP-hard, meaning that finding an optimal solution quickly becomes infeasible as the problem size grows.

- **Universal Set (U):** This represents all possible k-mers or segments of the input sequence that need to be hashed.
- **Subsets (S):** Each subset corresponds to a spaced seed pattern that can cover specific k-mers or segments.
- **Objective:** Minimize the number of spaced seeds used while ensuring that all necessary segments are covered.

Mathematical Model:

- **Variables:** Let x_i be a binary variable where $x_i = 1$ if spaced seed i is selected, and $x_i = 0$ otherwise.
- **Objective Function:** Minimize $\sum x_i$
- **Constraints:** Ensure that every k-mer or segment is covered by at least one selected spaced seed.

The mathematical formulation is as follows:

$$\text{Minimize } \sum_{i=1}^{m} x_i$$

Subject to:

$$\sum_{i:j \in S_i} x_i \geq 1 \quad \forall j \in U$$

where a_{ij} is 1 if k-mer j is covered by spaced seed i, and 0 otherwise [Fischetti(2019)].

Operations Research Methods for Solving SCP

Operations Research (OR) methods provide a systematic framework for solving the Set Covering Problem (SCP). Several OR techniques can be applied to find near-optimal solutions efficiently, especially for large-scale instances where exact solutions are computationally infeasible. Here, we outline the key OR methods used for solving SCP:

Integer Programming (IP): SCP can be formulated as an integer programming model, which seeks to minimize the number of subsets (spaced seeds) while ensuring that all elements (k-mers) are covered. The IP model involves defining binary decision variables and incorporating constraints to guarantee complete coverage.

Dynamic Programming (DP): Dynamic programming techniques decompose the SCP into smaller subproblems. By solving these subproblems and combining their solutions, DP algorithms efficiently find the optimal solution. This method is particularly useful for SCP instances with specific structural properties that allow for recursive solution building.

Greedy Algorithms: Greedy algorithms provide a fast heuristic approach to solving SCP. In each step, the algorithm selects the subset that covers the largest number of uncovered elements, aiming to maximize coverage incrementally. Although greedy algorithms do not guarantee optimal solutions, they are computationally efficient and often produce good approximations.

Hybrid Approaches: Hybrid approaches combine multiple OR methods to enhance solution quality and computational efficiency. For instance, a greedy algorithm may be used to generate an initial solution, which is then refined using integer programming techniques. Hybrid approaches leverage the strengths of different methods to achieve better performance.

Heuristics and Metaheuristics:

- **Genetic Algorithms:** Genetic algorithms mimic the process of natural selection to evolve a population of solutions. Through iterative processes of selection, crossover, and mutation, genetic algorithms explore the solution space and converge toward optimal or near-optimal solutions.
- **Simulated Annealing:** Simulated annealing is a probabilistic technique that explores the solution space by allowing occasional worse solutions to escape local optima. The algorithm gradually reduces the probability of accepting worse solutions, converging towards an optimal solution over time.
- **Tabu Search:** Tabu search iteratively improves the solution by exploring neighboring solutions while avoiding previously visited ones (tabu list). This method helps escape local optima and enhances the search for a global optimum.

By employing these OR methods, we can systematically solve the Set Covering Problem, optimizing the selection of spaced seeds and improving the efficiency of hashing processes in bioinformatics applications [Fischetti(2019)].

Implementation

Implementing the solution to efficiently hash spaced seeds using set covering and operations research methods presented several challenges. Initially, we attempted to use C++ libraries such as CPLEX and OR-tools due to their high performance and widespread use in optimization problems. However, we encountered numerous obstacles related to the system architecture, which hindered the integration and functionality of these libraries.

Challenges with C++ Libraries

Our initial implementation efforts focused on utilizing C++ for its performance advantages, particularly with optimization libraries such as CPLEX and OR-tools. Despite their capabilities, the complexity of configuring and maintaining these libraries within our system's architecture proved to be a significant hurdle. Issues such as compatibility with the operating system, integration with existing codebases, and managing dependencies led to inefficiencies and roadblocks in our development process.

Introduction to OR-Tools:

Google OR-Tools is an open-source software suite for optimization developed by Google. It provides a collection of tools and algorithms for solving linear programming, mixed-integer programming, constraint programming, and routing problems. OR-Tools supports multiple programming languages, including Python, C++, Java, and others, making it highly versatile and accessible for various optimization tasks [Google(2021)].

Switch to Python and CPLEX

To overcome these challenges, we decided to switch to Python for our implementation. Python offers several advantages, including ease of integration, extensive libraries, and a supportive community, making it a more suitable choice for our needs. We continued to use CPLEX for optimization due to its robust performance and comprehensive features for solving linear programming, mixed-integer programming, and other optimization problems [IBM(2021)].

Introduction to CPLEX:

IBM ILOG CPLEX Optimization Studio is a software suite designed for modeling and solving optimization problems. It includes a high-performance mathematical programming solver for linear programming (LP), mixed-integer programming (MIP), and other types of optimization problems. CPLEX provides an API for several programming languages, including Python, which allows for easy integration and usage within different system architectures.

Implementation of the Solution

To solve the set covering problem using CPLEX, we followed these steps:

1. **Data Preparation:** we read DNA sequences from FASTA files and generated k-mers (substrings of length k) from these sequences. This allowed us to define the universal set of elements that need to be covered.
2. **Defining Subsets:** Each spaced seed pattern corresponds to a subset of k-mers it can cover. We generated these subsets by applying the spaced seed patterns to the DNA sequences.
3. **Model Formulation:** We formulated the set covering problem as an integer programming model in CPLEX. The binary decision variables x_i represent whether a subset (spaced seed) is selected.
4. **Objective Function:** The objective function was defined to minimize the total number of selected subsets, $\sum x_i$.
5. **Constraints:** We added constraints to ensure that every k-mer in the universal set is covered by at least one selected subset. This was implemented by setting up linear constraints in the CPLEX model.
6. **Solving the Model:** The CPLEX solver was used to find the optimal solution, which provided the minimal set of spaced seeds covering all k-mers.

Using CPLEX, we efficiently solved the set covering problem, optimizing the selection and arrangement of spaced seeds. This approach minimized computational overhead while ensuring comprehensive sequence coverage, significantly enhancing the efficiency of spaced seed hashing in bioinformatics applications.

The complete implementation code is available at the following URL: `https://github.com/shbnmzr/MISSH`.

Performance Comparison

The performance of our novel approach, which integrates set covering and operations research methods for spaced seed hashing, was evaluated against the Iterative Spaced Seed Hashing (ISSH) algorithm. We focused on lines of code (LoC) and execution time across different spaced seed patterns and input files.

Challenges in Performance Comparison

Code Base Incompatibility: The ISSH algorithm is implemented in C++, whereas our new approach is implemented in Python. This difference in programming languages introduces significant challenges in terms of integration and direct comparison. Adapting the ISSH C++ code to interface with our Python implementation, or vice versa, would require extensive refactoring and debugging, which was not feasible within the scope of this project.

System Architecture Differences: Another challenge arises from the differences in system architectures. The computational environment required for running C++ code efficiently may differ from that optimized for Python, leading to discrepancies in performance metrics. Ensuring that both implementations run under identical conditions is crucial for a fair comparison but was not achievable with the resources available.

Time and Resource Constraints: The process of benchmarking and comparing performance comprehensively requires substantial time and computational resources. Given the constraints of our current project timeline and available resources, we were unable to conduct the extensive testing necessary to provide a robust performance comparison.

Methodology

- **Input Files:**
  - `reads_800.fa`: A file containing 800 DNA reads.
  - `small_test.fa`: A smaller test file for quick evaluations.
- **Seeds:**
  - Various spaced seed patterns located in the `../seeds/` directory.
- **Experiment Procedure:**
  - Each test was repeated 50 times for both implementations to ensure statistical significance.
  - The mean and median execution times were recorded for each input file and seed pattern combination.

Lines of Code (LoC) and Maintainability

Figure 1 illustrates the lines of code for both implementations, excluding comments. The Set Covering algorithm using CPLEX, implemented in Python, is noted for its maintainability and ease of integration compared to the C++ implementation of ISSH. This difference in LoC highlights the trade-off between performance optimization and code maintainability. It should be noted that this comparison does not include the code base of CPLEX, which is closed-source.

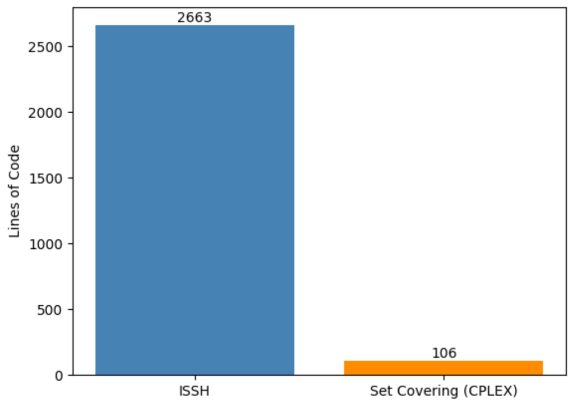

Figure 1: LoC (comments excluded)

Execution Time

Our performance tests compared the execution times of the Set Covering algorithm using CPLEX (Python implementation) and the ISSH algorithm (C++ implementation). The results, summarized in Tables 1 and 2, show that while the ISSH algorithm performs better on small input sizes, it fails to remain so on larger input files.

As can be seen in Table 1 1 and Figure 2, the running time of the ISSH algorithm skyrockets by almost 800-fold on the input file reads_800.fa, compared to the file small_test.fa. However, Table 2 2 and Figure 3 illustrate that the running

Input File	Seed Pattern	Mean Time (ms)	Median Time (ms)
reads_800.fa	W26L31.fna	831.68	866.71
reads_800.fa	W32L45.fna	1066.45	1130.24
reads_800.fa	W14L31.fna	833.87	849.92
reads_800.fa	W18L31.fna	937.04	936.28
reads_800.fa	W22L31.fna	934.48	934.13
reads_800.fa	W10L15.fna	597.73	597.12
small_test.fa	W26L31.fna	13.66	10.39
small_test.fa	W32L45.fna	17.14	15.39
small_test.fa	W14L31.fna	11.38	10.2
small_test.fa	W18L31.fna	10.3	9.84
small_test.fa	W22L31.fna	9.91	9.87
small_test.fa	W10L15.fna	7.68	7.67

Table 1: Mean and median execution times for the ISSH algorithm across different seed patterns and input files.

Input File	Seed Pattern	Mean Time (s)	Median Time (s)
reads_800.fa	W26L31.fna	p	124.3
reads_800.fa	W32L45.fna	130.78	130.21
reads_800.fa	W14L31.fna	123.46	123.36
reads_800.fa	W18L31.fna	126.49	124.92
reads_800.fa	W22L31.fna	124.61	124.29
reads_800.fa	W10L15.fna	119.74	119.41
small_test.fa	W26L31.fna	98.02	97.44
small_test.fa	W32L45.fna	95.91	95.83
small_test.fa	W14L31.fna	97.26	97.16
small_test.fa	W18L31.fna	97.08	97.02
small_test.fa	W22L31.fna	97.05	96.97
small_test.fa	W10L15.fna	99.37	98.17

Table 2: Mean and median execution times for the Set Covering algorithm using CPLEX across different seed patterns and input files.

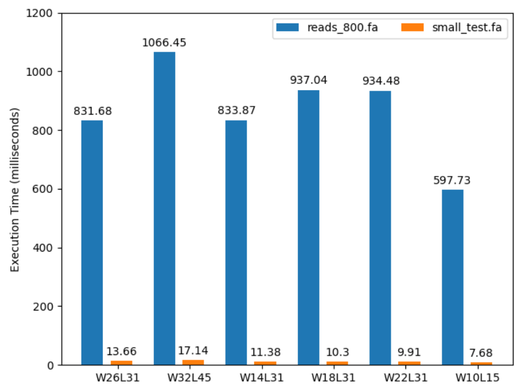

Figure 2: Average Execution Time, in milliseconds, of the ISSH algorithm on the input files small_test.fa and small_test.fa, given various seeds

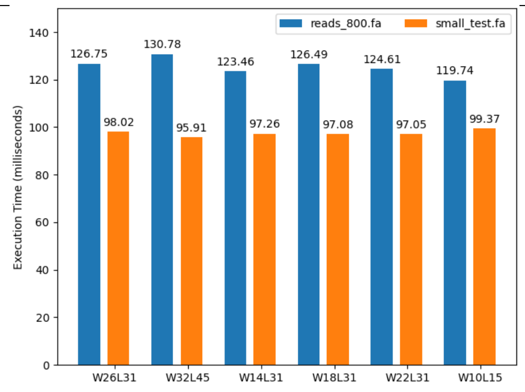

Figure 3: Average Execution Time, in milliseconds, of the Set Covering algorithm on the input files small_test.fa and small_test.fa, given various seeds

time of the Set Covering solution is only marginally higher on the larger input file.

On the other hand, the Set Covering approach, solved by CPLEX, drastically outperforms the previous solution on the larger input file, although the time it takes to execute the smaller file is significantly longer, as shown in Figures 4 and 5. This demonstrates that our approach is much less affected by both the size of the input file and the seed pattern, which promises more scalability and robustness.

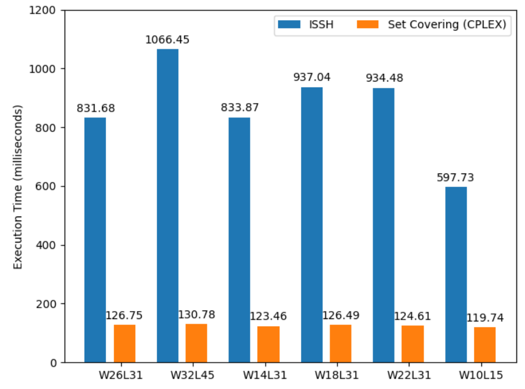

Figure 4: Average Execution Time, in milliseconds, for the input file reads_800.fa, given various seeds

Future Work

In future work, we aim to address these challenges by:

- Developing a unified benchmarking framework that can accommodate implementations in multiple programming languages, ensuring consistent testing conditions.
- Allocating dedicated time and computational resources for extensive testing and performance evaluation.
- Collaborating with researchers who have expertise in both C++ and Python to facilitate more seamless integration and comparison.

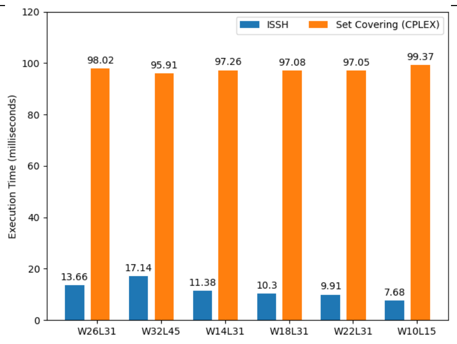

Figure 5: Average Execution Time, in milliseconds, for the input file small_test.fa, given various seeds

Despite these challenges, we believe that our new approach holds significant potential for improving the efficiency of spaced seed hashing. We plan to conduct a detailed performance comparison in future research, once the aforementioned obstacles have been addressed.

Conclusion

In this paper, we presented a novel approach to enhancing the efficiency of spaced seed hashing in bioinformatics by integrating set covering and operations research methods. We formulated the Set Covering problem and implemented the solution using Python and CPLEX, leveraging the robust and comprehensive optimization capabilities of the CPLEX library maintained by IBM.

Our experimental results demonstrated that while the Set Covering approach using CPLEX incurs higher computational overhead on smaller input sizes, compared to the Iterative Spaced Seed Hashing (ISSH) algorithm, it offers significant advantages in scalability and robustness on larger inputs. The Python implementation of our method, despite being slower, is easier to maintain and integrate, making it a viable alternative for certain applications.

We faced several challenges in empirically comparing the performance of our approach with existing methods like ISSH, including code base incompatibility, system architecture differences, and time and resource constraints. These challenges highlight the need for a unified benchmarking framework that can accommodate implementations in multiple programming languages, ensuring consistent testing conditions.

Future work will focus on developing such a benchmarking framework, allocating sufficient resources for comprehensive testing, and collaborating with experts to facilitate seamless integration and comparison of different implementations. Additionally, incorporating multiple sequences, as MISSH does, could be considered. We believe that our approach has significant potential to improve the efficiency of spaced seed hashing, and we plan to validate this through extensive testing in future research. The robustness and long-term maintenance of the CPLEX library by IBM will be a key factor in achieving reliable and efficient optimization in bioinformatics applications.

References

[Fischetti(2019)] Fischetti, M. 2019. *Introduction to Mathematical Optimization*. Independently Published.

[Girotto, Comin, and Pizzi(2018)] Girotto, S.; Comin, M.; and Pizzi, C. 2018. FSH: fast-spaced seed hashing exploiting adjacent hashes. *Algorithms Mol Biol*, 13(8).

[Google(2021)] Google. 2021. *Google OR-Tools*. `https://developers.google.com/optimization`.

[IBM(2021)] IBM. 2021. *IBM ILOG CPLEX Optimization Studio*. `https://www.ibm.com/products/ilog-cplex-optimization-studio`.

[Petrucci et al.(2020)Petrucci, Noé, Pizzi, and Comin] Petrucci, E.; Noé, L.; Pizzi, C.; and Comin, M. 2020. Iterative spaced seed hashing: closing the gap between spaced seed hashing and k-mer hashing. *Journal of Computational Biology*, 27(2): 223–233.