Improved Sequence Alignment For Merging Functions

Sean Stirling - s1641210

MInf Project (Part 1) Report
Master of Informatics
School of Informatics
University of Edinburgh

2020

Abstract

Within optimising compilers, the code reduction pass finds ways in which the binary size of compiled code can be reduced as best as possible. This is highly valued by devices with limited hardware including IoT sensors, embedded devices, and even smartphones. Function merging, a subset of code reduction, analyses functions inside code to potentially merge them into a smaller but semantically equivalent function.

Production compilers merge functions that are deemed to be identical while state of the art compilers make use of control flow graphs to find exploitable structural similarities to determine whether two functions can be merged. A new technique emerged in 2019 [1] that does not have the same shortcomings as previous approaches. Borrowing a technique from BioInformatics, it finds similarities and aligns two functions in such a way that the optimal alignment is achieved, thereby merging the similarities and separating the dissimilarities. A new system utilising the same methods hopes to improve upon this idea, however, it still uses the same algorithm for finding similarities between functions.

This algorithm is quadratic in both time and space with respect to function size, limiting the potential of this technique. In this thesis, I present implementations of alternative algorithms borrowed from BioInformatics which can, on average, improve upon code reduction by 1.4% or time by 5%. The chosen final implementation achieves a 5% speedup in compilation overhead with no detriment to code reduction attained by the system. Each algorithm was evaluated by analysing the compilation time and resulting binary size reduction on the SPEC2006 benchmarking suite.

Acknowledgements

I'd like to extend my appreciation to multiple people that aided me throughout the development of this project. Firstly, I'd like to thank my supervisor for providing insight, keeping me grounded to my work, and giving me feedback on my report, alongside the PhD student working on the system for answering my emails whenever I had questions about how to operate it or why it did certain things. Special thanks to my parents who support me while I study, as well as my friends who I would work alongside in AT until the building got shutdown. Thanks also to the algorithm that picked me for this project considering it was my number one choice. Last, but not least, I have to thank all the artists out there for the music I would listen to: hundreds of hours spent listening to music, making the countless hours spent staring at a screen far more enjoyable.

Table of Contents

Chapter 1

Introduction

1.1 Project Description

Binary size is money when dealing with small, resource hungry devices like IoT sensors, embedded devices, and even smartphones. Strangely, compilers do not do a very good job on that front. Function merging by sequence alignment [1] is a promising new technique trying to change that. This approach identifies pairs of similar functions and replaces them with a single merged function where identical statements are merged and non-identical statements are executed only when the corresponding function would be called. The merged function ends up requiring less space but having behaviour identical to that of the original functions.

The most computationally expensive part of this technique is sequence alignment. Sequence alignment is borrowed from bio-informatics where it is used to identify regions of similarity in different DNA sequences. When given a sequence of instructions, it identifies the best way to merge the two sequences without changing the internal order of each sequence.

A new system named SalSSA is hoping to utilise the methods behind FMSA but do so with improvements in both code reduction and time. The existing implementation uses the Needleman-Wunsch algorithm as in FMSA, a dynamic programming algorithm that finds globally optimal alignments but has $O(MN)$ worst case performance, and $O(MN)$ memory complexity. The aim of this project is to apply other sequence alignment algorithms which could increase code reduction, decrease memory usage, and reduce the time cost of sequence alignment by using other, potentially approximate, algorithms. Approximating the optimal alignment should affect function merging negatively, so part of this project will be to evaluate the trade-off between alignment quality and speed and develop run-time heuristics to control this trade-off.

1.2 Motivation

Duplication of code is a common problem pertaining to beginners and professional software developers alike. Code repetition can be caused by: software stacks, where

each library in the stack re-implements pre-existing functions; clean code practices leading to similar functions with only minor differences in their functionality; working within teams and on large code-bases; and templating. It is not uncommon for a code-base to include multiple functions all performing very similar tasks while containing minor differences. An increase in function count will inevitably result in a larger compiled binary size. Function merging, a technique used to analyse code and remove duplication within code, is the solution to this problem.

Unfortunately, production compilers offer little help beyond dead-code elimination or the merging of identical functions. Experimental state-of-the-art compilers analyse function control flow graphs (CFGs) to determine whether merging is possible while maintaining the semantics of the original separate functions. Even the previous state-of-the-art does not manage to make a significant difference in code size reduction. This is likely due to the rigid, overly restrictive algorithms used to find duplicates.

In 2019, Function Merging by Sequence Alignment [1] emerged as a new technique to merge similar functions. This method has significant benefits over other merging techniques, in that it is possible to merge any function with another function; whether they have varying CFGs or signatures. Sequence alignment provides flexible merging possibilities, allowing for a greater number of functions to be identified as similar and subsequently merged. FMSA, as the new state of the art, utilised an algorithm which is quadratic in both time and space, and now the currently developing system also uses this algorithm. It is by far the slowest part of the process: increasing alignment speed would greatly reduce the compilation overhead introduced by the system.

Aside from the quadratic nature of the algorithm which makes it ill-suited for large functions, it has the shortcoming in that it relies on code generation to maintain function semantics. The reason this can negatively affect code reduction is that a merged function calculated this way may open up many if statements, and therefore more code, to select which function is being called. For example, a function in which there is quite a lot of similarity but has many extra randomly placed individual functions requires an if statement to be generated for each one of these extra instructions so that the merged function correctly calls them when needed. The overhead in file size brought about by the if statement may not actually be worthwhile for the one or two instructions placed inside it. Code generation can, therefore, have a negative effect on the actual size of the compiled code, and Needleman-Wunsch can occasionally fall victim to this.

1.3 Objectives

The aim of this project is to implement, analyse, and compare various sequence alignment algorithms that provide improvements in code reduction, memory usage, and time. Through analysis of the trade-offs of each algorithm, heuristics can be developed to ensure that the appropriate algorithm is selected for alignment.

1.4 Contributions and Results Summary

In short, the adopted algorithms and their resulting performances:

- Smith-Waterman - 14.8% in code reduction, 2% improvement in time
- Gotoh Global - 14.6% in code reduction, no change in time
- Gotoh Local - 14.0% in code reduction, 1% improvement in time
- Myers and Miller - 14.3% in code reduction, no change in time
- FOGSAA - 13.4%, 51% increase in time
- MUMMER - 13.7% in code reduction, 3% improvement in time
- BLA(S)T - 13.4% in code reduction, 5% improvement in time

  The chosen final implementation, BLA(S)T, achieves an average code reduction equivalent to the current algorithm, 13.4%, with an average speedup of 5%, capable of reaching heights up to greater than 15% improvement.

Chapter 2

Background Review

2.1 Function Merging

Optimising compilers have been around since not long after the development of the compiler itself. They use clever techniques to minimize a program's execution time, memory requirement, power consumption, and binary size. Code optimisation is an integral part of a compiler's purpose: removal of unnecessary computations and intermediate values will result in far less work for the CPU, cache, and memory. This is especially fruitful for embedded systems as optimised code will bring a lower product cost. However, optimising compilers will often increase the code size easily if it deems it to help performance, going against what the code reduction step is attempting to achieve.

Code reduction can fall into two categories: the replacement of code with smaller but semantically equivalent code, such as peephole optimisation [7]; and the removal of redundant code, such as dead code elimination, block merging, or function merging. There are many distinct techniques [8] involved that code compaction can utilise. Redundant code elimination follows the idea that computation within a program is redundant at a program point if it has been previously computed and its result is guaranteed to be available at that point. Any such computations should be eliminated without affecting the behaviour of the program. Additionally, unreachable code elimination attempts to find code fragments where there is no control flow path to it. This "dead code" can easily be removed without any effect on a program's execution. Strength reduction, a technique which refers to the first type of code reduction, replaces instructions by an equivalent but cheaper sequence. This doesn't always mean a reduction in code size, however, the benefits for code reduction arise in situations where the replacement sequence happens to be shorter than the original sequence.

Function merging, where functions are equivalent, is known as Identical Code Folding (ICF). Google implemented ICF into their ELF gold linker [9] [10] which merges identical functions into a single copy. According to Google's implementation of ICF, they can achieve a reduction in file size of 6%. Through their analysis of C++ programs, identical code is particularly common with heavy use of templates. Templated classes can be instantiated as different types and each of them gets separate copies of various

repeated functions. When the size of the pointer types are the same, the bodies of these repeated functions are identical and can be merged into one instance.

Before FMSA, the state-of-the-art [11] utilised structural similarity between functions to merge non-identical functions. Their algorithm determines two functions to be similar if both their function signatures and control flow graphs are equivalent. Functions have equivalent signatures if they agree on the number, order, and types of arguments as well as their linkage type, and other compiler-specific properties. CFGs are equivalent if and only if there exists a directed edge-preserving bijection between the two graphs, also known as graph isomorphism. Simply put, graph isomorphism is an equivalence relation on graphs, where their number of components (vertices and edges) are the same and their edge connectivity is retained. The number of basic blocks within the isomorphic CFGs must also be equal, with each instruction having equivalent resulting types. However, each instruction may differ in their opcodes and the number and type of inputs. When performed on the SPEC2006 code base, the resulting code reduction reaches 3.9% on the x86 architecture, while only introducing an extra 5.4% overhead to compilation time.

Code factoring, a related technique to function merging, involves finding a multiply-occurring sequence of instructions, making a representative sequence that can be used in place of all occurrences, and then arranging for each occurrence to execute the representative instead of the occurrence.

Function merging by sequence alignment [1] is a novel technique in the field of compiler optimisations. It does not suffer from any of the major limitations of existing solutions, outperforming them by more than 2.4x in terms of code reduction. With this method, an average reduction of 6% is seen while only introducing 15% overhead. With improvements upon the compilation time, memory, or code reduction, this technique has the potential to become usable within production compilers.

2.2 Sequence Alignment

Sequence alignment is the poster child of BioInformatics, the field of combining biology, computer science, and mathematics to analyse and interpret biological data. The computational problem of sequence alignment was perhaps born in 1966 with the definition of the edit distance between two strings as the minimum number of edit operations - insertions, deletions, and letter substitutions - needed to transform one string into another, known as the Levenshtein distance [12]. The subsequent literature on sequence alignment has been enormous and includes seminal papers such as the original Needleman-Wunsch dynamic programming solution [13], the introduction of affine gaps, multiple sequence alignment approaches, and genome-scale similarity search [14]. Regardless of the rising complexity, sequence alignment boils down to matching characters in a string, in order, in such a way that the highest score, with respect to a scoring scheme, is achieved. This can be attained by inserting gaps in either strings, or matching two characters from each string whether they are equivalent (matching) or nonequivalent (mismatching). This principle is shown below in Figure 2.1.

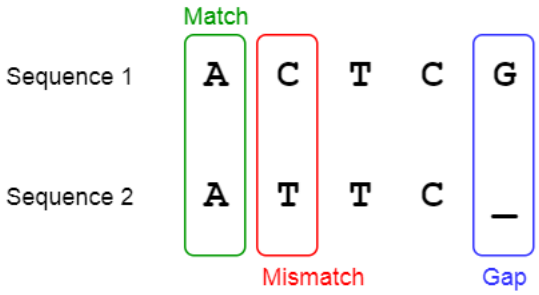

Figure 2.1: Sequence Alignment [15]

2.2.1 Alignment Basics

After the initial discovery of the structure of DNA in the 1950s [16], the 1970s saw rise to DNA sequencing. DNA sequencing, the process in which DNA is extracted and examined so its nucleic acid sequence is determined, soon brought about the demand for algorithms which could effectively compare the sequences of DNA. The ability to efficiently compare DNA sequences is highly sought after in BioInformatics. It gives BioInformaticians the power to establish the differences between sequences of DNA that vary between species or between the members of species themselves. Further allowing the discovery of Phylogenetic trees, that is, trees which display the relationship between genes (sequence of DNA which codes for a particular protein) and their evolutionary distances between species. In short, one could determine how specific genes had evolved between species and how closely related these species, through these specific genes, are to one another.

Discovering the function of genes is an important task in the decoding of genome sequences. Comparative analysis of sequences, that are related through evolution, can provide valuable information for inferring gene function and regulatory controls, necessitating significantly enlightening alignments. The necessity for finding alignments that are the most biologically reasonable brought about the first optimal global alignment algorithm. Published in 1970 by Saul B. Needleman and Christian D. Wunsch [13], the Needleman-Wunsch algorithm, was one of the first dynamic programming algorithms which could align biological sequences. The purpose of the algorithm is to find the arrangement of the two sequences such that, according to a scoring scheme, the optimal score is achieved. Scoring schemes can be broken down into three components: a Match score, a Mismatch score, and a Gap score.

- The Match score describes the score given when a character from each sequence are aligned and are the same
- The Gap score describes the score given when a character from one sequence is aligned with a gap in the other sequence
- The Mismatch score describes the score given when a character from each se-

quence are aligned but are not the same

The scoring scheme allows the user to define what they would prefer. For instance, in biology, one may expect to find sequences which are more closely related to one another with gaps being more present than mismatches. In this case, the mismatch score would be quite low, whereas, the gap score would be higher so as to represent the desire for aligning sequences with gaps over mismatches. The understanding behind this scoring scheme is that nucleotides, the components of DNA (AGTC), could have been inserted in, or deleted from, gene sequences during evolutionary mutation. Relating this to function merging, a sequence of computer instructions (the function) may contain some extra instructions that are not present in a similar function. Aligning these two functions would very likely place gaps in the positions of the extra instructions while matching the equivalent instructions.

2.2.2 Local vs Global

Alignments created by algorithms developed to compare biological sequences will usually fall into these two categories:

- Global
- Local

Global alignment techniques attempt to align every character in each sequence, which can be most useful when the sequences are similar and of roughly equal size. Local alignment techniques attempt to find highly similar regions between each sequence and align those, which can be useful for mostly dissimilar sequences or when one sequence is much shorter than the other. The Needleman-Wunsch algorithm is a global alignment algorithm, whereas, the Smith-Waterman algorithm [2] is a local one. A hybrid alignment exists, known as semi-global or ”glocal”, however, these techniques were not explored in this project.

Considering function merging, a global aligner would take the entirety of each sequence of instructions and find the best way to align them. A local aligner will find the highest scoring segment between the two functions, possibly finding an incredibly similar section between them, and not attempting to accommodate the rest. This could potentially result in an increase in code reduction due to the local aligner not being hindered by taking into account the entirety of each sequence, allowing it to focus on only the best possible matching segment. Though, this could also result in a decrease in code reduction since it will throw out many potentially matching instructions in favour of achieving the highest score possible.

2.2.3 Affine Gaps

Through experimental evidence, it has been found that mutated DNA sequences are more likely to have long stretches of inserted, or deleted, nucleotides instead of random insertions, or deletions, throughout the sequence. To accommodate for this, alignment algorithms were developed which allow for specifying a gap opening cost and a gap

extension cost (eq. 2.1), describing the cost to open a gap and then extend it. This type of gap cost scheme is commonly known as affine gaps.

$$Gap\ Run\ Cost = Gap\ Open\ +\ Gap\ Extension*(k) \tag{2.1}$$

Where k is equal to the length of the gap.

As with DNA, you are probably more likely to see functions which differ by stretches of extra instructions instead of randomly placed additional instructions. This type of gap cost could help better represent the similarities between functions, potentially increasing the amount of code reduction provided by the system. Affine gaps are considered to be accurate in understanding the evolutionary nature of DNA sequences and should also be sufficient to represent the implemented nature of functions written by developers. Aside from constant and affine, other gap cost functions exist, including linear, convex, and profile-based. However, these other gap cost functions may bring about increased time complexities as they are more complicated to solve for and may not provide an alignment for which you are looking for.

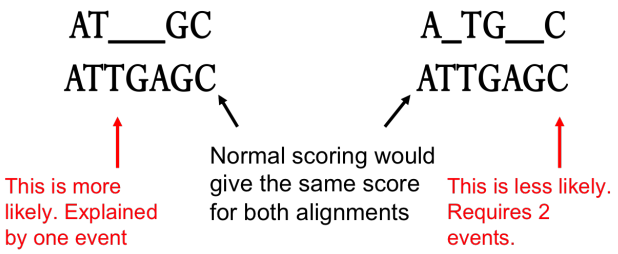

Figure 2.2: Affine Gaps [17]

2.2.4 Alignment techniques

The process of sequence alignment can be solved with two broad methods:

- Dynamic Programming
- Heuristic Approach

Dynamic programming techniques, such as the Needleman-Wunsch algorithm, guarantee the user an optimal alignment, i.e. it gives the alignment achieving the highest score possible from the scoring scheme. Dynamic programming techniques are guaranteed to find the optimal alignment as they extensively iterate through the search space. Every dynamic programming sequence alignment technique follows a motif of calculating an alignment matrix which can then be used to find the optimal alignment. A common misunderstanding is that substitution/similarity matrices [18] are believed to be this constructed matrix, when in fact, substitution/similarity matrices are what are used to define the scoring scheme. In BioInformatics, this is necessary for protein alignments, as there are 20 possible amino acids, known as codons [19], which have varying degrees of probability that one could mutate into the other.

The grid, as seen in Figure 2.3, created by dynamic programming algorithms refer to the score given to an alignment as each character from a sequence is aligned with

the other sequence. For example, the second row of the grid refers to aligning each character of the top sequence, AACG, with purely gaps, whereas, the diagonal of the alignment matrix represents aligning each character with its counterpart in the second sequence, whether it be a match or mismatch. One way to think of it is, by having pointers, P1 and P2, that point towards the to-be aligned character of each sequence. Incrementing P1 and P2 describes a match or mismatch, an increase in P1 is a gap in sequence 2, and an increase in P2 is a gap in sequence 1.

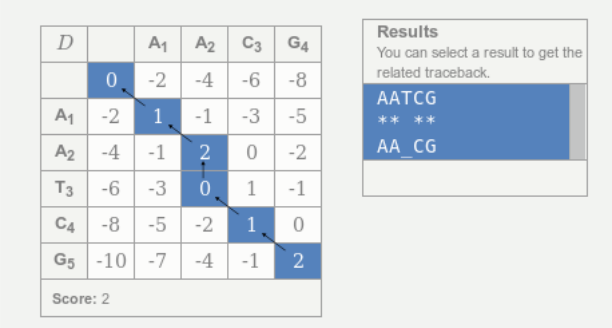

Figure 2.3: Alignment Grid (Match = +1, Mismatch = -1, Gap = -2) [20]

Each cell in the grid represents the score given to a character alignment based on the maximum of the three possible cells to come before it and a similarity score. The score of a cell depends on the maximum of the score above, to the left, or to the top left of it. Adding these scores to a similarity score which describes the operation to get to the cell in question from the cell it depends on achieves the score for that cell. Basically, each cell can be thought of as the total alignment score, if the alignment stopped at that cell. With this idea, it is trivial to see how the Needleman-Wunsch algorithm operates. Simply start at the bottom-right cell and follow the path that obtained the score in that cell. So, the trace-back procedure should be thought of as retracing your steps finding out which cell was used to give the current cell its score.

Obviously, creating the alignment matrix requires *O(MN)* time to create, while the trace-back procedure is linear *O(M+N)*. Where M and N are the lengths of the sequences. The memory usage is also *O(MN)*.

On the other hand, heuristic approaches attempt to find an optimal or highly scoring alignment using approximating methods. Unfortunately, they are often unable to find the optimal alignment. Trading off alignment accuracy for speed has become a staple of the alignment process, where these days, database searching and multiple sequence alignment is widely used. The NCBI tool [21], available to the public, allows users to search for similarities between their query sequence and millions of sequences stored within a database, with sizes greater than a million characters in each sequence. Clearly, performing dynamic programming alignment on each sequence with the query to find the one with the highest score is infeasible. The BLAST algorithm [14] provides a way of doing this with a massive speedup in time. It is not guaranteed to find the optimal alignment, however, BioInformaticians consider it to still be highly accurate.

Most heuristic methods follow an approach of quickly finding ”anchors” which are then used as starting points for a dynamic programming algorithm to align from, thereby reducing the search space that it has to search through. These anchors can be found by multiple methods but most use some form of tree/graph which is then very quickly searched through to find the specific patterns and anchors that the algorithm is looking for. Other heuristic methods exist, including one I have implemented here which employs a branch and bound methodology that lets it search through possible alignment paths and prune branches that are deemed not to be worth it's time to search through.

Chapter 3

Adopted Algorithms and their Implementation

3.1 Software Design

Before I get into the meat of what I've done and how, I'll briefly describe how the software is designed and why. Most of these principles were developed before my contributions, however, I have modified a couple of things which I'll note here.

- Each algorithm inherits from a sequence aligner super class that provides methods for returning variables like the scoring scheme and the match function used to match instructions, as well as providing the pure virtual function for returning the alignment. I altered this class to allow any alignment algorithm to force a global alignment. Quite simply done by taking the two sequences, the current local alignment, and the positions in each sequence where the alignment sits, and sticking on the remaining sequence data to the front and ends without losing function ordering. Attempting to force a global alignment on one that is already global will result in no change.
- A small class defining the scoring scheme, storing the gap, match and mismatch costs has been modified to accommodate affine gaps by providing a new constructor and new variables which store the gap open and gap extension costs.
- An alignment class defines how the alignment is actually stored. This is what is used by the system to generate the merged code. I have not altered this.
- Heavy use of templating is present, which was pre-existing before I began contributing. I carried on the practice of templating as it allowed me to test out my implementations on strings in a very quick manner without having to test on the final system. Being able to test on DNA strings had the added benefit of being able to compare with results from actual BioInformatics results.

3.2 Smith-Waterman [2]

The Smith-Waterman algorithm was the first algorithm I implemented because of its high similarity to the Needleman-Wunsch algorithm and, consequently, low complexity. Moreover, as soon as you begin to research about sequence alignment, two algorithms are presented and explained to you: those being the Needleman-Wunsch and Smith-Waterman algorithms. It is often considered to be more famous than the Needleman-Wunsch algorithm, while only being a variation, due to its disposition for finding highly similar segments within DNA sequences.

The Smith–Waterman algorithm performs local sequence alignment; that is, for determining similar regions between two strings of nucleic acid sequences or protein sequences. First proposed in 1981, as a variation of the Needleman-Wunsch algorithm, it finds optimal local alignments with respect to the scoring scheme. Negative scoring cells in the alignment matrix that would be created by the Needleman-Wunsch algorithm are instead set to 0. This indicates that any alignment's path through this point should end because carrying on would cause the score to drop below, and not be able to rise back up to, the current alignment score. When tracing backwards to align the sequences, the procedure starts from the highest scoring cell in the alignment matrix and continues until it comes across a 0, at which point the alignment ends. This results in the highest scoring alignment segment of the two sequences.

Due to its similarity to Needleman-Wunsch, this algorithm was fairly simple to implement by imitating the same structure used by the implementation of NW. Altering the way each cell in the alignment matrix is calculated, keeping track of the highest scoring matrix cell as it is built, and tracing backwards from this max position until a 0 is come across, is sufficient to have a working Smith-Waterman implementation. As you may be wondering, how can local alignments represent two functions when it can disregard many instructions in the resulting alignment? My solution to this is to 'stick' on the remaining instructions to the local alignment. This means that, before the alignment's position in each function, the instructions up to that point are aligned with just gaps. The same is for after the alignment. This results in an alignment looking like Figure 3.1, where the section of matches is the local alignment, and the rest is the remaining sequences having been stuck on.

```
# Smith-Waterman:
AGCTTCAG-GCTGA------------
        ||||
--------AGCTG-GATCGATCGATG
Time Taken: 80 microseconds
```

Figure 3.1: Forcing A Global Alignment

This allows me to represent the nature of a local alignment algorithm while actually forcing it into a global one so it can be used by the system.

3.3 Gotoh (Local and Global) [3]

The dynamic programming approach by Michael S. Waterman, Temple F. Smith and William A. Beyer [22] computes optimal global alignments while allowing for an arbitrary scoring of consecutive gaps. It permitted the use of gaps with different types of gap scoring functions such as logarithmic or affine gaps, to help better understand and represent similarities between DNA sequences. Because of this extra functionality, it brought with it a much larger time complexity, $O(N^3)$.

This is obviously infeasible for large DNA sequences, and so there was a demand for a more efficient algorithm which could compute alignments with either one or many gap scoring functions. In 1982, Osamu Gotoh showed that global optimal alignments with affine gaps could be computed in $O(N^2)$ time. The idea behind his method is to, instead of computing one alignment matrix, compute three alignment matrices which keep track of the cost for the alignment of prefixes, alignment of prefixes that end with a gap in sequence B, and alignment of prefixes that end with a gap in sequence A. What this means is that an entry in the first alignment matrix, M, represents the best score for the alignment of the prefixes $a_{1..i}$ with $b_{1..i}$, while an entry in the other matrices, I_x, I_y, represent the best score under the additional constraint that the alignment ends in a gap within sequence A and B, respectively.

Due to Gotoh being another dynamic programming approach, it again followed a very similar implementation structure of calculating the alignment matrices and then a trace-back procedure. Calculating the alignment matrices is very simple, however, trace-back is not. I was not able to find an explanation on how the trace-back procedure worked in Gotoh so I came up with my own way. Beginning in matrix M, I trace-back until I find that I need to traverse into either matrix I_x or I_y. As soon as I traverse into a new matrix an enum variable keeping track of which matrix I am in is switched to this new matrix. This must be done because certain scores can actually seem like they came from any matrix, and so figuring out the matrix it truly came from relies on knowing what matrix I am currently in. This is calculable because I know that I cannot jump from matrix I_x to I_y or vice-versa. There are also a few edge cases involved with being inside one of the I_x or I_y matrices that require special attention, such as reaching the edge of a I_x or I_y matrix and then not being able to move to the end of the alignment.

```
# Needleman-Wunsch:
AGCT---TC-A--G-GCTGA
||||   || |  |   ||
AGCTGGATCGATCGA--TG-
Time Taken: 114 microseconds

# Global Gotoh:
AGCT---TC-----AGGCTGA
||||   ||     |   ||
AGCTGGATCGATCGA---TG-
Time Taken: 198 microseconds
```

Figure 3.2: Affine Gaps

Gotoh's algorithm is an optimal global alignment algorithm, however, it can be altered into an optimal local alignment algorithm by imitating the Smith-Waterman approach.

Changing the rules of how matrix M is calculated to that like Smith-Waterman is sufficient, as well as altering the trace-back procedure to how Smith-waterman works by starting at the highest scoring cell until 0 is reached.

Well after I had implemented, but before I began integrating the Gotoh algorithms, and any other affine gap algorithm, I wrongfully believed that they would become obsolete. This is because mismatches cannot be represented in the system since an instruction cannot be aligned with another instruction when they are not equivalent. So, at first, I believed the extra merging capabilities provided by these algorithms would be lost, and that they would become more computationally expensive versions of the Needleman-Wunsch. However, as a natural consequence to the way the algorithms prefer long stretches of gaps, if we do not allow mismatches at all, we end up with alignments that prefer stretches of matches and stretches of gaps. What makes this great is that when considering code generation, an opening of a gap stretch in an alignment would signify code being generated to select a specific function. This addition of code to the final merged function will increase the file size of the merged function, and for alignments with many gaps being placed randomly throughout the alignment, many if statements will be generated and inserted. This principle can be seen in Figure 3.2, where Needleman-Wunsch has 6 gap stretches representing 6 potential if statements, whereas Gotoh has 4 gap stretches, and so only 4 potential if statements. The hope is that this reduction in code generation will outweigh the loss of a few singular matches.

3.4 Myers and Miller [4]

The goal of Myers-Miller's algorithm was to "give Hirschberg's idea the visibility it deserves by developing a linear-space version of Gotoh's algorithm" [4]. Released in 1988, the paper utilises the 1975 computer science paper by Hirschberg [23] and a realisation that the matrices used in Gotoh, M, I_x and I_y, depend only on previous rows of each matrix. Meaning that a handful of row-size vectors are sufficient to compute successive rows.

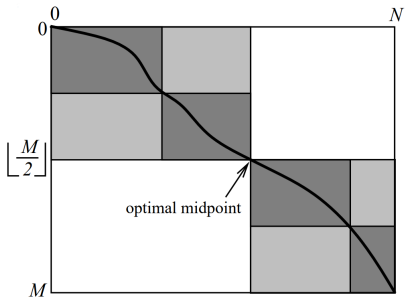

Figure 3.3: Hirschberg Subdivision Process [24]

Named after Dan Hirschberg, Hirschberg's algorithm [23], is a dynamic programming algorithm that finds the global optimal sequence alignment between two sequences. While it is a different sequence alignment algorithm to Needleman-Wunsch, the technique it employs of divide and conquer does not differentiate itself enough, and so it is considered to be a more space efficient version of the Needleman-Wunsch algorithm. It is a clever modification of the NW algorithm which still takes $O(MN)$ time but requires only $O(M+N)$ space. It works by firstly making a forward score-only pass and stopping at the middle row. Then, a backwards

score-only pass stopping at the middle row. For each point along the middle row, we now have the optimal score from the beginning to that point and the optimal score from that point to the end. Simply adding those numbers we have collected gives the optimal score along all paths from the beginning to the end, that pass through that point. Sweeping along the middle row and checking those sums can determine a point on the middle row at which the optimal path is guaranteed to pass through.

This idea can then be applied recursively to find all optimal points on the alignment path, which can then be aligned accordingly, returning an optimal alignment in only linear space. You can see this in Figure 3.3. The reason the memory complexity is linear is because the memory requirement per subdivision is half that of the parent problem. If we assume T to be the original memory requirement, then it follows that the total size of all problems is at most $T + \frac{1}{2}T + \frac{1}{4}T + ... = 2T$ [24]. Hirschberg's algorithm, without affine gaps, had already been implemented before I began contributing, so there was no need to create it myself.

To perform recursion on sequences of instructions, a class which holds references to the original sequences, but defines new starting and end positions, allows me to declare the divided sequences used in the recursive function. Since Hirschberg's algorithm had already been implemented before my contributions, this class was already pre-existing and I could use its simple functionality of referencing the sequences with alternative offsets.

3.5 FOGSAA [5]

By this point in time, I had implemented these dynamic programming algorithms which improve upon alignment quality, and memory requirement, but none had improved on time. The algorithm I present here is called FOGSAA (Fast Optimal Global Sequence Alignment Algorithm). In essence, it is a branch and bound algorithm that, at first, greedily goes down the "alignment space" (possible alignment paths) and then tries other promising paths that it has seen on its greedy journey, repeating this process until it knows that it cannot do any better than what it has currently achieved.

Branch and bound algorithms are a design paradigm for solving discrete and combinatorial optimisation problems, as well as mathematical optimisation [25][26]. First proposed in 1960 by Ailsa Land and Alison Doig [27], it has become the most widely used tool for solving NP-hard optimisation problems. They are reasonably understandable in what they are doing and why, however, they have a long and complicated history of figuring out their complexities [28].

FOGSAA begins by greedily aligning each nucleotide pair, whether that be aligning a match, mismatch, or gap. As it does so, each alignment point (or node in the tree) and the possible other routes are given two fitness scores, T_{max} and T_{min}, based on their current score, PrS, and possible worst and best future scores, F_{max} and F_{min}, (eqs 3.1 and 3.2). At each node, the alternative paths are saved. The alternative paths are the alignment types, whether that be a gap in A or B, or a match(mismatch), that are less than one of the other alignment types. The alignment type that gives the best fitness score, T_{max}, for that moment in time is what is travelled down.

$$T_{max} = PrS + F_{max} \tag{3.1}$$

$$T_{min} = PrS + F_{min} \tag{3.2}$$

When the initial greedy search has been completed, based on the fitness scores, the next best possible path is taken to see how it performs. Throughout traversing down the alignment search space, the algorithm checks to see whether this particular alignment point has been done better before, and if so, it prunes the current branch and searches down a better path. The idea is that there cannot be more than MN nodes searched in the branch and bound tree. Therefore, the worst case running time is bounded by $O(MN)$, though, on average, it is much lower. The best case is when FOGSAA finds the optimal alignment within its initial greedy search, resulting in a best case complexity of $O(M+N)$ time. This shows that FOGSAA has varying complexities depending on how similar the sequences being aligned are, whereas, Needleman-Wunsch is $O(MN)$ best-case, average-case, and worst-case. FOGSAA will also result in the exact same alignment quality as the Needleman-Wunsch algorithm. A detailed proof of this is given in the paper but, in short, FOGSAA will resort to searching the entire search space which is the same as what Needleman-Wusnch does anyway. The principles behind FOGSAA can be seen in Figures 3.4 and 3.5. Figure 3.4 displays the initial greedy search that reaches the full depth of an alignment, tries a new alternative path and realises that this new path has been done just as well or better before. In Figure 3.5 you can see the final alignment highlighted in blue and all the alternative paths that had been searched and pruned.

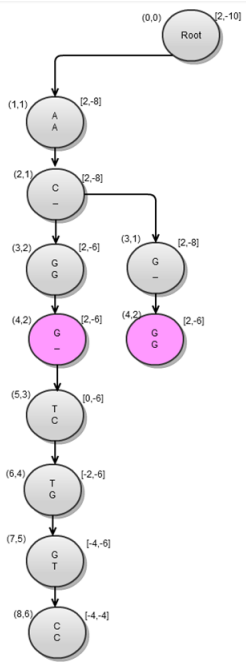

Figure 3.4: FOGSAA Initial Greedy Search and Pruning a Branch [5]

Efficient implementation of this algorithm requires that a hashed priority queue is used to save the possible alternative paths. Unfortunately, no description of how this is done is given, other than that their max fitness score, T_{max}, represents their hash value and the nodes with the same T_{max} values are sorted by their minimum fitness score, T_{min}. I was also unable to find a description or implementation of a hashed priority queue online, so I resorted to creating my own. At first, I implemented the algorithm using just a C++ standard library priority queue to store the possible alternate paths, however, because of having to sort each newly inserted node the running time of the algorithm was ridiculously slow. The necessity for this hashed priority queue was huge.

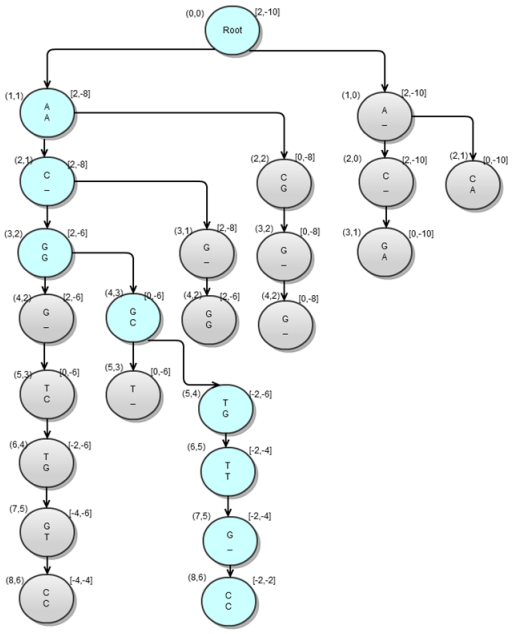

Figure 3.5: Full Alignment With FOGSAA and The Paths Searched [5]

With a hashed list, you must know what the hash value is to be able to index into the array correctly. However, knowing that I should be using the fitness scores to index into the array, how do I know what fitness score represents the top score in the list? My solution, and I suspect that this is what the original authors did too, is to keep track of the current highest fitness score I have seen as I've added to the hashed priority queue. This maxPointer, as new nodes are saved, will be updated with a nodes fitness score if it happens to have a better score. The only other problem is that when I pop a node off the hashed priority queue, maxPointer possibly no longer points towards the top node. So, as I pop an element, I decrease maxPointer until I find a node it can point towards.

My implementation of this hashed priority queue is simply an array of C++ standard library priority queues, where the maxPointer allows me to index into the array where the elements are a queue which contains nodes sorted by their minimum fitness score. This vastly improved the alignment speed of the algorithm.

Like the initial use of just one priority queue, another naive implementation of this algorithm had me storing the resulting alignment inside each node, with the expectation that once I had reached the end of the search process, the last node would contain the final alignment. This is very bad for memory usage and computation efficiency. Instead, having each node storing an enum of the alignment type that occurred to get to it would allow me to take the final node and backtrack towards the beginning, aligning along the way.

Unfortunately, no matter the number of optimisations I have made, I have not been able to have this algorithm run faster than NW in any instance, except from highly similar sequences (almost exactly similar) and ones that are very short (10 characters or less). From my analysis, it seems like it just takes far too much work done per node search. In the final implementation, as a further improvement to speed, I only save the best possible alternative path instead of both alternative paths. This actually results in a much faster alignment speed while not reducing the alignment quality by much at all. Yet, even with this, it still underperforms. Perhaps someone with greater knowledge of optimising code could help me reduce the work done.

3.6 MUMmer [6]

The MUMmer algorithm was the final algorithm I had implemented. The reason I have left a couple of algorithms to the end is that some decisions that went into those came about after finalising this one.

This was perhaps the most complicated algorithm that I had implemented due to its use of suffix trees [30]. The idea behind MUMmer is to construct a generalised suffix tree of the two DNA sequences concatenated with one another so that Maximal Unique Matches can be found as anchor points, and then the gaps (not to be confused with gaps in an alignment) between them are bridged together using a dynamic programming algorithm. This should result in a far smaller search space for the dynamic programming algorithm used. This principle can be seen in Figure 3.6. A shows what a dynamic algorithm may achieve, B displays the anchors that have been found by the suffix tree, C shows the chosen anchors and how the path would traverse between them, and finally, D shows the final alignment using the anchors and bridging the gap between them. From D, you can see, with the rectangular boxes, that the dynamic programming algorithm used to bridge the gap does not have to traverse through the entire search space of the two sequences.

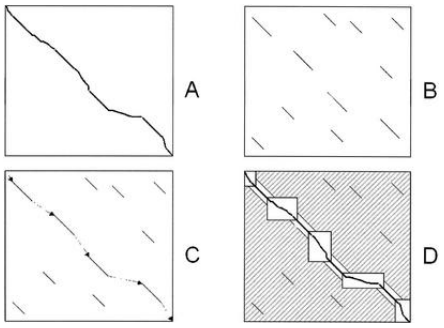

Figure 3.6: MUMmer Reduced Search Space [29]

The MUMmer algorithm was designed for aligning extremely large sequences containing millions of nucleotides. It assumes that the sequences are closely related, and using this assumption can compare entire chromosomes as large as the human chromosome, i.e. several hundred million base pairs. It's had many revisions over the years, culminating in MUMmer4 [31] but, in this section, I focus on the original algorithm proposed in 1999.

3.6.1 Suffix Trees

I'll begin by describing suffix trees, their construction and what they're used for. Not to be confused with suffix tries or suffix arrays, though they are all related, suffix trees are an extremely elegant data structure with many applications in text processing. Some of these include substring checking, pattern matching, finding distinct substrings, finding the longest palindrome, finding the longest common substring etc. These uses all have vast applications such as plagiarism checking, DNA matching and text searching. As the name suggests, suffix trees represent every suffix within a string S. A suffix tree of "banana" will look something like Figure 3.7:

As you can see in Figure 3.7, searching for the word "ana" within the suffix tree is fast

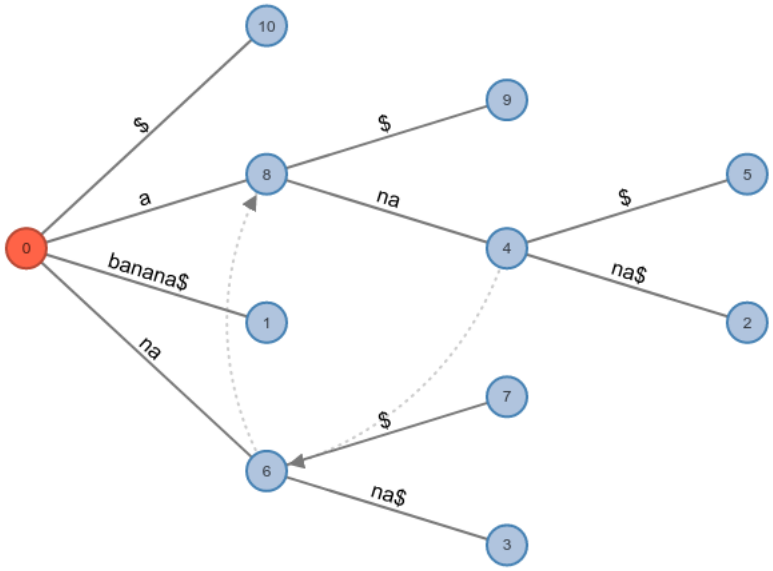

Figure 3.7: Source: https://brenden.github.io/ukkonen-animation/

and trivial. Simply traverse down the tree with ”ana”, matching characters until you reach the end of your query string, showing that it is indeed a substring of ”banana”.

With this power it comes as a surprising fact that suffix trees can be constructed in linear time, $O(N)$, where N is the size of the string. In situations where it is expected that many searches of a long text will be performed, a suffix tree would repay the cost of construction and more.

Suffix trees were first introduced in 1973 by Peter Weiner [32] in the seminal paper. A simple brute-force algorithm to construct suffix trees would require quadratic time which is no faster than dynamic programming techniques. However, it is possible to build a suffix tree in linear time [32][33][34]. In fact, the suffix tree construction algorithm used by MUMmer was the one presented by McCreight, while I used Ukkonen’s algorithm. An article written by Robert Giegerich and Stefan Kurtz analyses and explains the differences between these linear time construction algorithms if you’re interested [35].

Esko Ukkonen devised a linear-time algorithm for constructing a suffix tree that may be the conceptually easiest linear-time construction algorithm. It has space saving improvements over Weiner’s algorithm and has a property known as ”online”, where it processes its input piece-by-piece in a serial fashion [36]. While it may be the easiest linear-time construction algorithm to understand, it is still quite complex and requires some thought to fully grasp it’s inner workings. There are some fantastic explanations online like this famous stack-overflow answer [37], this 6 article long series by GeeksforGeeks [38], and many others [39][40][41], alongside this amazing visualisation tool [42] allowing you to see how the suffix tree looks during the stages of it’s construction.

I won’t attempt to explain the algorithm here, I’ll instead describe the problems I faced in its implementation and how I solved them. The first problem encountered was that each node in the tree has the potential to have every possible child in the

alphabet. In a string, the alphabet defines what characters could be present in this string. For ASCII this would be 128, for DNA this would be just 4. However, an alphabet of instructions? The system assumes that the alphabet is possibly infinite for the function merging optimisation. The alphabet can be considered to be the union of instruction labels with the intersection of opcodes and the data types, $'label' \cup (InstructionOpCodes \cap DataTypes)$, which becomes essentially infinite because written code can have infinitely many user-defined types. My solution to this is that I know for sure that the maximum potential number of distinct instructions is bounded by the size of the input sequence. Knowing this, I can use a dynamically allocated array storing pointers to nodes which have been created. This allows me to store the children nodes while keeping memory requirement small by using pointers which only actually point to a node when that node has been explicitly created by the algorithm.

Another major problem that I faced was that I needed a way to define what position in the children array was the specific instruction I was looking for. For instance, in DNA, you could define that in each node's children array, 'A' can be found in position 0, 'G' in 1, 'T' in 2, 'C' in 3, such that the child node had been created. In my case, I know that the number of children is bounded by the size of the input sequence, but I don't know at what position in the children array each instruction should be placed in. I solved this by using an alphabet vector which, during construction, continuously pushes new instructions that it hasn't seen before, defining the position in the children array as the position it can be found in this alphabet. This solves my problem quite nicely, however, during construction I am constantly searching/looping through this array to find the position a certain instruction should be found at, and for sequences with large alphabets, this could result in large search times.

Other solutions to this problem included pre-scanning the sequences beforehand to build the alphabet vector before the construction algorithm proceeds or using a form of hash function which could take the specific instruction and assign it a value representing its position in the array. The former method still ran into the same problem of large search times as in my final solution since I will still have to look through this array to find the instruction I'm looking for and what index it belongs to. The latter solution requires a hash function which could take an instruction and assign it an index value; I had no idea how to come up with such a function.

As I mentioned above, MUMmer creates a generalised suffix tree, which it uses to find Maximal Unique Matches. A generalised suffix tree is a variation on a suffix tree in which the suffixes of two (or more) distinct strings are stored, not just the suffixes of one string. Generalised suffix trees allow for finding the longest common substring between strings, or for finding MUMs between strings. A generalised suffix tree can be created by concatenating strings together, separating them by a dummy character often signified by #, and ended by another dummy character, $. This would look something like *Seq1#Seq2$*. The problem with this is that I cannot just create dummy instructions to place between and at the end of the two sequences. A sequence in the system is stored as *llvm::SmallVectorImpl<Value*,8>*, that is, an llvm specific version of a vector of pointers. This means that I can add nullptrs to represent the dummy characters. With a modification to the way instructions are matched, I can assure that

nullptrs are treated like dummy characters, by checking the position of the sequence and whether I had seen a nullptr before during construction. Surprisingly, this works quite well for generalised suffix trees. Unfortunately, construction of non-generalised suffix trees does not seem to behave as expected. I believe this to be the main cause, though, I have not been able to fully diagnose this issue.

Lastly, due to the use of pointers within suffix tree construction and storage, memory allocation must be carefully handled. A great number of implementations of suffix trees that I have seen cause memory leakage to occur. This obviously cannot be tolerated and so proper deletion of the tree and it's nodes, and any pointers used within construction is an absolute necessity. Deletion of the tree can be handled in a very simple way: traverse down the tree in a depth-first manner until you reach a leaf node, and delete that node. As you move back up the tree, delete internal nodes as you go. Construction of the tree requires that there be pointers that point towards the current end of a nodes suffix. Many other nodes can rely on this value and so changing it and having the other nodes see this change is necessary. Sadly, raw pointers are not acceptable because, if you want to delete one of these pointers, you can never be sure whether it is in use by another node. Smart pointers allowed me to solve this issue very quickly and easily. Using the C++ standard library shared_ptr, these pointers which are no longer in use by any node are deleted without my supervision. This worked perfectly, and I have come away with a suffix tree construction algorithm that does not leak any memory.

3.6.2 MUMmer

Now that I have explained suffix trees, I will be able to describe how the MUMmer algorithm works. As eluded to earlier, MUMmer finds Maximal Unique Matches (MUMs) by searching a generalised suffix tree constructed by concatenating the two DNA sequences. As the name suggests, a MUM is a maximally sized unique match, meaning a maximal match that does not occur more than once in either sequence, only once. As described in the paper [6], "every unique matching sequence is represented by an internal node with exactly two child nodes, such that the child nodes are leaf nodes from different genomes". They also mention that "MUMmer... determines whether a match is maximal based on pointers used to construct the suffix tree". This quite cryptic description is actually eluding to suffix links. Suffix links are simply pointers between internal nodes, allowing construction to lower from $O(N^2)$ to $O(N)$. So long as a potential MUM is not the suffix link of another potential MUM, then it is definitely maximal. If it is the suffix link of another MUM, then based on how suffix links work, this potential MUM is not maximal. Traversing through all the trees nodes, I can find the potential MUMs and then remove MUMs which are not actually maximal by checking their suffix links.

The MUMs found by the suffix tree are known as "anchors" which are then joined together by a dynamic programming algorithm. This should result in a decrease in the search space for the dynamic programming algorithm as it needs to align between anchors and not entire sequences. Though, how do we decide which MUMs should be used for anchors? The original MUMmer paper employs a technique of finding the longest increasing subsequence [43]. Firstly, they sort the MUMs by their position in

genome A, then consider their positions in genome B and the lengths of each MUM. The longest increasing subsequence problem is solvable in $O(NlogN)$. I was not able to find their implementation or description of their LIS (Longest Increasing Subsequence) algorithm and simply resorted to implementing the basic algorithm that does not take into account their lengths. This still results in reasonable accuracy, however, and so I don't believe it to be a major issue. The MUMs that are found during the LIS process are then taken to be the anchors which will be aligned and bridged together.

Bridging the gap between these remaining MUMs is the final stage in the algorithm. MUMmer employs a variety of algorithms to bridge the gap based upon analysis of the to be joined MUMs, however, for simplicity's sake, I use Needleman-Wunsch to bridge the gap. Another reason for using Needleman-Wunsch to bridge the gaps is that it allows me to maintain a sort of consistency since I am trying to compare MUMmer with Needleman-Wunsch, and using MUMmer to reduce the search space for NW is the important aspect to focus on here. Not to mention that NW is the most basic and should be the fastest dynamic approach I have currently implemented.

The construction cost, sorting and LIS cost, outweighs that of the standard Needleman-Wunsch algorithm. This means that, on average, NW will outperform MUMmer in terms of speed and efficiency. However, for large sequence lengths, MUMmer will become more appropriate. So as a heuristic built into the algorithm itself, I perform MUMmer only when the combined sequence lengths are above a certain sweet spot that I found via bench-marking. Otherwise, I use Needleman-Wunsch to align the sequences.

I should note that, on occasion, MUMs cannot be found by the suffix tree because of the possibility that there are no maximal **unique** matches. When this happens, I return the longest common substring that can be found by the suffix tree, which is then bridged towards the end and towards the beginning. There is also a small issue where sometimes the suffix tree returns a MUM that seems to be completely incorrect, as in it has a negative index into a sequence. So, before I return the list of MUMs, I vet each one to make sure its positions in the sequences are valid. I've only seen this happen almost once in a thousand times and so I would not call it a big issue. You may then wonder about the validity of the other MUMs, yet, from my analysis, it seems like it works perfectly up until this one in a thousand case.

3.7 BLA(S)T

The most famous paper in BioInformatics is undoubtedly the one released in 1990 by Altschul et al [14], amassing over 58,000 citations. It describes a technique for genome scale searching, allowing BioInformaticians to take a query DNA or protein sequence and search a database of other sequences for highly similar ones. Through learning about how integral this algorithm is to the world of BioInformatics, I decided to attempt to apply it's methods to pairwise alignment.

Right off the bat, BLAST (Basic Local Alignment Search Tool) is a search tool and not a pairwise alignment tool. It finds patterns, known as words, within the query sequence and the database of millions of other sequences. Each word match found in a sequence

is expanded until it cannot expand any further without causing a mismatch. This is done for each sequence where a word match was found, and those words whose score is above a certain threshold are called a High Scoring Segment Pair (HSP), or Maximal Segment Pair (MSP). A report is then given detailing each HSP where they can then be aligned properly by a dynamic programming algorithm or aligned from the HSP endpoints to the beginning and ends of each sequence.

Using BLAST's methods as a pairwise alignment tool has actually been done before [44], however, I had implemented my version before I became aware of how it had been done before. Plus, the full details of how they do it is not fully explained, they do things slightly differently, and are specific to DNA and protein analysis.

BLAST begins by taking your query sequence, splitting it up into overlapping k-mers, where k is the length, and matching them to the database sequences. For example, a sequence like AGTC as 2-mers would be: AG, GT, TC. To find these matches, BLAST describes using "a deterministic finite automaton or finite state machine". Suffix Trees or the Aho-Corasick [45] algorithm would be appropriate for this case, and I believe a variation of this algorithm is what is implemented today. At first, I had implemented pattern matching in a quadratic way, by simply taking each word and scanning it through the other sequence for matches, which was incredibly slow. As an improvement to speed, I realised I could take these words and scan the second sequence, incrementing by the size of the word length. This meant matching the words to **non-overlapping** k-mers, thereby, greatly reducing the amount of time spent scanning the second sequence while hardly reducing the ability to find matches.

As I mentioned in the MUMmer section, some decisions that went into this algorithm came about after finalising MUMmer. One of those decisions came about here. Comparing to non-overlapping k-mers improves the runtime of the algorithm, however, it is still quadratic in nature. Suffix trees can also be used for pattern matching by traversing the tree and recording where exact matches are found. Unfortunately, as mentioned before in the suffix tree section, construction of non-generalised suffix trees seems to be incorrect, which is what is needed for pattern matching. I actually have a working pattern matcher for strings, but when I tried to apply it to instructions, it would return matches that were absolutely incorrect. So, I reverted this stage to use the method of comparing words with non-overlapping k-mers. BLAST's principles can be seen in Figure 3.8.

Another thing to consider for this stage is what an appropriate word size should be. Based on what I had seen during my research of BLAST and testing on my data sets, I have decided to use the base two logarithmic function on the shortest sequence to define what the word size should be. Although, if the size of the shortest sequence is greater than 500, then I set the word size to 11. The reason I chose this function is that I felt it allowed me to achieve a high amount of sensitivity to finding matches while keeping the word size quite high, especially for shorter sequences where I want the word size to always be greater than 1 and a significant portion of the total sequence length. I also chose 11 as a hard-coded word size when the sequence size is greater than 500 because this value is what is commonly used in BLAST, and I wanted to

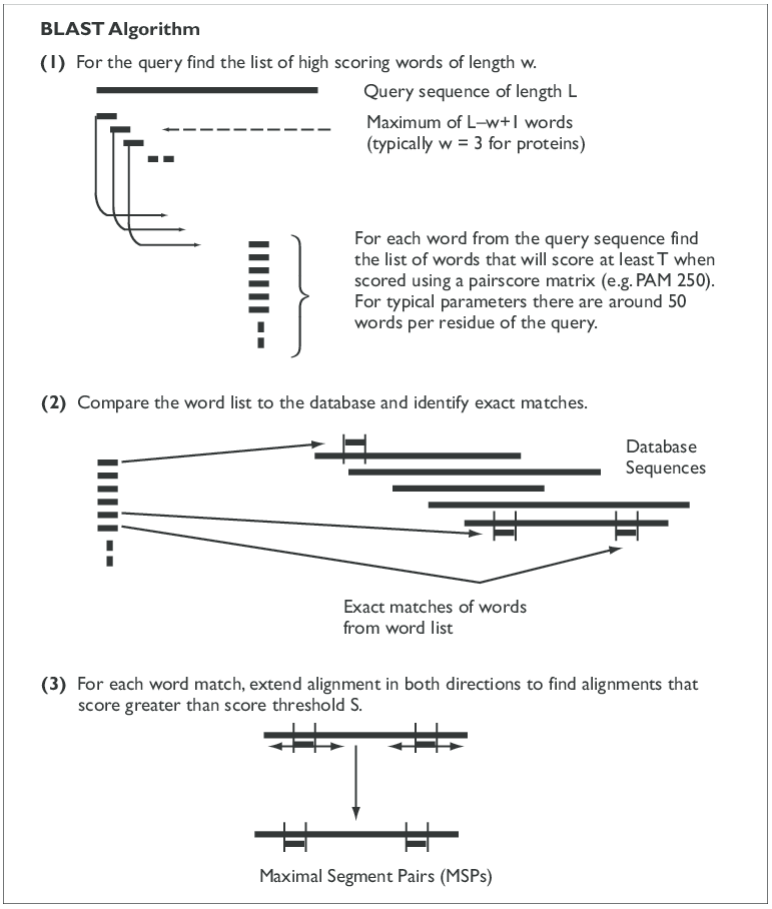

Figure 3.8: BLAST Algorithm [46]

represent BLAST as best I could. The base two logarithmic function works quite well with the 500 idea, as by that point the word size achieved by this function comes close to achieving a word size of 11.

Next, BLAST takes these word matches and begins expanding upon them until they can't be expanded any further, otherwise they would begin mismatching. I do the same by taking the words I have matched and attempt to decrease and increase their positions in each sequence and see whether that still leaves me with a matching pattern. Expanding in one direction does not affect the expansion in the other direction allowing for expansion that can reach its maximum potential. Any words that overlap during the expansion process are merged together which also greatly reduces the amount of time spent expanding, and means fewer words need to be checked for gap bridging. Once a word has expanded fully, I save it for later, which are known as HSPs in the context of BLAST. Other methods of saving could be utilised such as checking whether the fully expanded word passes a certain threshold, but I felt that saving all the fully expanded words would further decrease the amount of time spent using the dynamic programming algorithm when it comes to bridging the gaps.

Now BLAST would report which sequences contain these maximal matches and give a full alignment of the best ones. I instead take the HSPs and attempt to perform the exact same alignment procedure as I do with MUMmer, by using them as anchors

and aligning the gaps between them. As in MUMmer, I sort them by their position in sequence A, perform LIS on them for their position in sequence B, and then bridge the gaps between them. The only difference in this process is that I do not align to the ends of each sequence so as to represent local alignments, which is what BLAST is used for. As with Smith-Waterman, I force it into a global alignment by sticking on the leftovers of each sequence.

3.8 Alternative Routes

In this subsection, I'll give a quick list and description of algorithms that were implemented or researched but were not taken further to be evaluated.

- By 1997, the BLAST programs had become widely used for searching protein and DNA databases for sequence similarities. At which point a new paper was released [47] describing refinements to algorithmic and statistical methods which substantially decrease execution time while enhancing sensitivity to finding weakly similar regions. They describe a new criterion for allowing the extension of word hits, stating that words should only be extending so long as the hit is within a certain distance of another hit lying on the same diagonal. The diagonal essentially means the same relativistic position on both sequences for two hits. Positions (1,1) and (5,5) are on the same diagonal, (3,2) and (5,4) are too, whereas, (1,1) and (4,6) are not. As with BLAST, I took the ideas behind this paper and tried to apply them to purely pairwise alignment. Unfortunately, the gap extension criterion does not apply well to aligning two sequences as very few word hits are extended, which means that the work for the dynamic programming algorithm used to bridge the gap is increased dramatically. I fully implemented this algorithm but found that its performance in code reduction and time were abysmal and so I saw no reason to take it further to be evaluated. Not to mention that it was just a more complex version of BLAST but performs far worse when compared to it.

- A dynamic programming approach by Abdullah N. Arslan, Ömer Eğecioğlu and Pavel A. Pevzner (2001) [48] computes the best normalized local alignments of two sequences. It pinpoints the best local alignment with the maximal degree of similarity (e.g. maximal percent of matches). This technique seems promising at first since local alignments seem to do really well for code reduction, however, the algorithm's runtime is $O(N^2 logN)$ and is purported to be 3-5x slower than Smith-Waterman in practice. I decided that the extra time introduced by this algorithm was not worth the potential code reduction.

- After I had implemented the Myers-Miller algorithm, the next on my list was another dynamic programming approach [49] which alleges to be a time-efficient algorithm that produces the k best "non-intersecting" local alignments for any chosen k. This has the prospective of trying out multiple local alignments to see which one produces the best resulting code reduction. This would come with increased time complexity and, by this point, I hadn't implemented any algorithm which improved upon time. And so, I decided it was not in my best

interest to implement it.

3.9 Final Policy

In the original project description, I was to develop these algorithms and then come up with any heuristics that would allow the system to dynamically choose which algorithm to use. Most heuristics are not feasible because of the amount of overhead that they bring to the total time when compared to simply picking an algorithm and running with it always. Even fast heuristics like $O(N)$ would bring about too much overhead. However, due to the function merging optimisation which selects pairs of functions to be merged, I know that a heuristic has already selected two functions which are quite similar. This would be very promising for an algorithm like FOGSAA which thrives on sequences which are similar, it's just a shame it's still far too slow even for the selected functions. If only I knew how similar these functions are then I'd be able to tell whether FOGSAA would make a reasonable time gain.

The only possible fast heuristic I could come up with was sequence length. As in MUMmer, where I select NW for short sequences and MUMmer for long sequences, I could select an algorithm which runs faster for shorter sequences and another for longer sequences. As you will see in the Evaluation section, Smith Waterman, MUMmer and BLAST run the fastest with no loss in quality of code reduction, with MUMmer and Smith Waterman slightly improving upon it. So, I looked at utilising sequence length to decide when to use either of these algorithms. For instance, I could choose Smith Waterman for very short sequences, BLAST for medium length ones, and MUMmer for the very long sequences. I tried multiple ways in which this interaction could work, even removing either MUMmer or BLAST from the equation, yet I could not manage to achieve a time reduction more significant than the results achieved by BLAST by itself.

Because BLAST is found to be the fastest algorithm, it would then make sense to just use BLAST as is. Though, you may remember how both MUMmer and BLAST sit on top of NW, considering that they both make use of it to bridge the gaps in their alignments. And since I've found Smith-Waterman to outperform NW in every way, shouldn't it then make sense to use Smith-Waterman in place of NW. Or, I could take it even further, get really meta, and use BLAST or MUMmer to fill in the gaps within their own alignments. This could result in a sort of recursive nature where an algorithm uses itself to bridge the gaps in its own alignments.

Firstly, this recursive idea is impossible since it breaks the template recursion depth. I didn't have much hope for it anyway since, at some point, anchoring recursively would result in a far increased time than simply aligning via a dynamic solution. I believed that the anchoring algorithms like MUMmer and BLAST would benefit greatly from using Smith-Waterman or a dynamic approach which runs faster than NW. Through benchmarking, I found this not to be the case. It seems as though this reduces the efficiency of each algorithm and they're better off left with using NW to bridge gaps. The results from using SW to bridge gaps often came close in time (behind by $<1\%$), and actually provided an increase in code reduction by about 0.1%. The slight increase

in code reduction does not seem worth the larger decrease in time.

Therefore, the final policy is simply the BLAST algorithm using NW to bridge the gaps between the anchors in its alignment.

Chapter 4

Evaluation

For each algorithm, I will present the results for code reduction against the results obtained by the state of the art, FMSA using Needleman-Wunsch, and the results achieved by the new system, SalSSA using NW. This allows me to show the improvements made over the state of the art while emphasising how my contributions played a part in that improvement. For compilation time, the results are normalised to the time achieved by the system using Needleman-Wunsch so as to show the difference between the two algorithms, (Eq 4.1). FMSA performs far worse than the current system and so skews the resulting time graphs, understating the performance improvements that can be seen in each one. This is why I have decided not to include FMSA in each time graph.

$$normTimeAlg = \frac{timeAlg - timeNW}{timeNW} \tag{4.1}$$

The state of the art and the current system both make use of variable exploration values which define how many functions a specific function should be aligned with so as to pick the best one. I have decided not to analyse how these values affect the computation time or code reduction because I'm only trying to show the improvements of my contributions over the state of the art and in the new system. They don't affect this in any way. Also, the FMSA system that I have used seems to have some errors where it would actually occasionally fail on some benchmarks, ending the whole benchmark process. Making it a nightmare to try and test it for high t values where it is more likely to fail, ruin the benchmark results, and waste a considerable amount of time.

When it comes to code reduction and compilation time, code reduction is only done once because it will always return the same value, whereas compilation time is measured three times so as to gain an average since it can vary slightly between measurements. So, when I present the compilation time results, they are the average of 3 separate runs, which should minimise any variation. This allows me to gain a decent understanding of how well my contributions perform in terms of code reduction and time.

The presented results were collected using a dedicated server intended to minimise measurement noise. The specs are:

- Cpu: Intel Xeon CPU E5-2560 v2 @2.6GHz, 16 cores
- Memory: 64GB
- OS: Linux ”Ubuntu” v18.04.3 (Bionic Beaver), kernel 4.15.0-69-generic

The function merging optimisation runs as a pass within llvm, version 11.0.0, with clang, version 6.0.0-1ubuntu2.

The scoring scheme can play a part in how each algorithm performs. So, to represent the desire for matches over gaps and disallow any mismatches, I use a scoring scheme of +2 for a match, -1 for a gap, and essentially $-\infty$ for mismatches. However, for algorithms in which affine gaps are used, the chosen scoring scheme is +1 for a match, -3 for a gap open, -1 for a gap extension, and $-\infty$ for mismatches. This affine scoring scheme was chosen because the gap opening and extension costs would require a stretch of 4 matches for it to be worth closing a current gap stretch. Scoring schemes can be altered to find ones which produce higher resulting code reduction, however, I did not explore this in great detail, only trying a few scoring schemes and choosing the best one.

4.1 Smith-Waterman

Beginning with the Smith-Waterman algorithm, it is clear to see in Figure 4.1 that it outperforms the state of the art, FMSA, in code reduction by a considerable amount and outperforms the current system using Needleman-Wunsch by a marginal amount. It is incredibly shocking to see a greater than 50% code reduction in the 447.dealIII benchmark, while FMSA only achieved half that. Averaging a 14.8% reduction in file size compared to the 13.4% achieved by the Needleman-Wunsch algorithm, totalling an improvement of 1.4% in code reduction. Against the state of the art, FMSA, an 8.2% increase in code reduction is seen. To reach 8.2% average code reduction would be very successful, but for there to be an increase in 8.2% over the current highest achieved average reduction is amazing. It goes to show that merging functions by sequence alignment is incredibly prosperous for code reducing compilers.

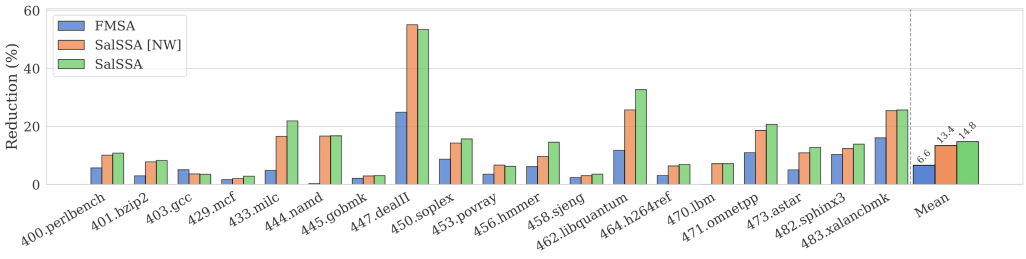

Figure 4.1: Code Reduction of Smith-Waterman

What these results suggest to me is that finding high scoring local alignments or segments of high similarity between the functions is better than attempting to align entire sequences when wanting to merge them in such a way that code generation is reduced

and code reduction is increased. Attempting to align the instructions that are ”stuck on” in the forcing process could possibly result in an alignment filled with gap openings which, in turn, result in extra code generation, when it would be more reasonable to just stick them on the ends. Or, it may be hindering global alignment algorithms to even consider these instructions when it would do much better if it didn't have to. I would, therefore, recommend that local alignments or processes which find segments of high similarity between functions be used for the function merging optimisation.

In Figure 4.2, you can see that SalSSA utilising Smith-Waterman outperforms Needleman-Wunsch in terms of time. An overall 2% improvement is seen using the Smith-Waterman algorithm over the current system. I believe the time difference between the algorithms used in SalSSA could be attributed to how Smith-Waterman only aligns segments, and so the trace-back procedure is often far shorter than the trace-back procedure in Needleman-Wunsch. The extra instructions that are unaligned are stuck on quickly and easily, requiring no calculations like when optimally tracing backwards. While Smith-Waterman often outperforms Needleman-Wunsch in terms of time, it can be slower as seen in benchmark 444.namd. The few extra instructions per cell making sure no cell goes below zero and saving the cell at which the highest score is found could culminate in a longer time to construct the matrix. The shortened trace-back procedure often benefits the system so that the total time is reduced, but with 444.namd, this does not seem to be the case.

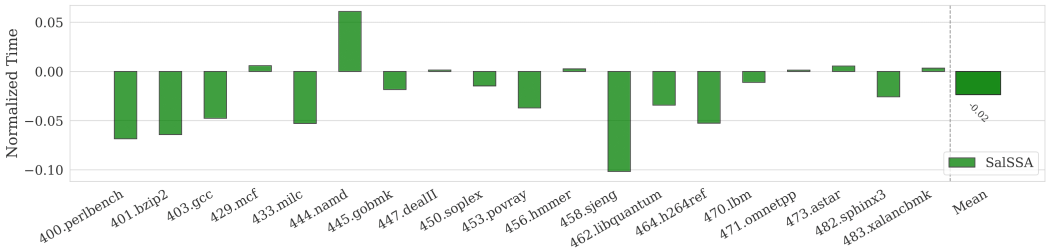

Figure 4.2: Relative Time of Smith-Waterman to NW

4.2 Gotoh (Global and Local)

With Gotoh, for both the local and global variations, the hope was that minimising the number of gap stretches/gap openings would reduce the amount of code generation affecting the merged function file size, while maintaining a standard of finding similar matches between them. And looking at the results in Figures 4.3 and 4.4 it is clear to see that this was a success. In both the local and global variations, they outperform the state of the art and the current system using Needleman-Wunsch. Surprisingly, neither outperform Smith-Waterman but what these results show is that minimising code generation is again very fruitful for the system.

Looking at the improvements from Global Gotoh, and how it outperforms NW, I fully expected that Local Gotoh would follow suit and outperform Smith-Waterman in the

same way. I believed that the combination of the power of Smith-Waterman and the reduction of code generation through reducing gap openings would bring about even further code reduction with Local Gotoh, but it seems that this was not the case. Local Gotoh under-performs when compared to Smith-Waterman and the Global variation of Gotoh.

What is quite interesting to see here is that Global Gotoh has massive benefits when used upon certain benchmarks such as 447.dealII, where you can see it nearly reaches a 60% reduction. However, it has detrimental effects seen in benchmarks like 444.namd, 458.sjeng, and 464.h264ref. I am not entirely sure why this may be the case but it's possible that the scoring scheme used to disadvantage gap openings has actually negatively affected the number of matches that an alignment can achieve, and thus code reduction is reduced overall.

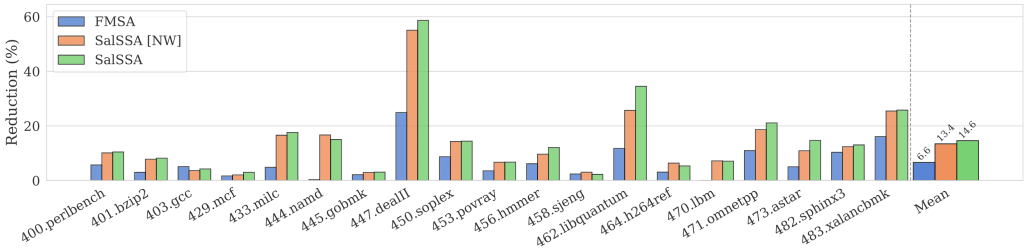

Figure 4.3: Code Size Reduction of Global Gotoh

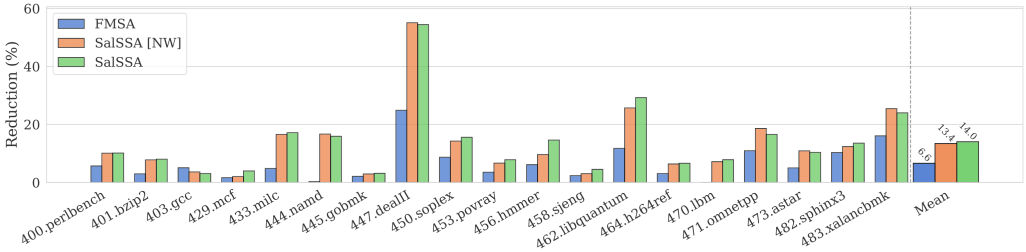

Figure 4.4: Code Size Reduction of Local Gotoh

In total, Global Gotoh achieves a mean code reduction of 14.6%, an increase of 1.2% over Needleman-Wunsch, and an increase of 8% over the state of the art, FMSA. On the other hand, Local Gotoh achieves a mean code reduction of 14%, beating Needleman-Wunsch by 0.6% and FMSA by 7.4%. The global variation of Gotoh is the clear winner here.

With the compilation time results of both algorithms, it is clear to see how the improvements in code reduction comes at a cost. The extra instructions necessary to calculate all three matrices and trace-back have taken an effect on the total time necessary to align sequences. For both algorithms, it seems that the overhead to this functionality is about 2% Since Smith-Waterman outperforms these algorithms, and does so in a

quicker time, I would recommend that Smith-Waterman be preferred over Gotoh for uses in function merging.

Looking at the time results of both algorithms, I'm surprised to see how neither increases the overhead when the extra calculations per loop should sum to an increased time per pairwise alignment. With Global Gotoh there appears to be very similar results with one benchmark, 470.lbm, taking significantly more time than NW at an extra 6%. The other benchmarks sit around the same time as NW, with a couple actually reaching an improvement. It seems as though the extra calculations don't have a notable consequence on the total time.

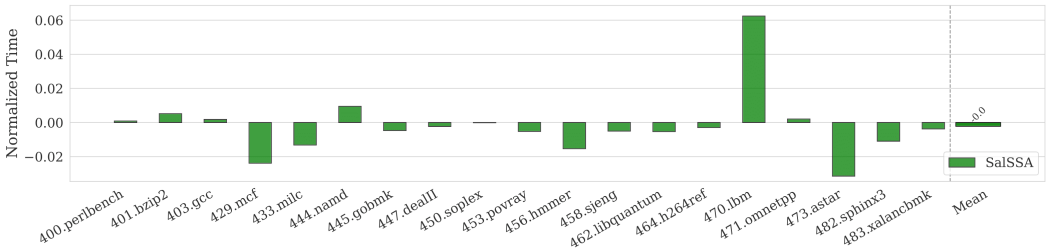

Figure 4.5: Relative Time of Global Gotoh to NW

With the time results of Smith-Waterman, I kind of expected Local Gotoh to have a similar time to Global Gotoh but at a slightly reduced time. This is quite clear to see with the 1% improvement on NW, but doesn't overtake SW because of the extra calculations involved in Local Gotoh.

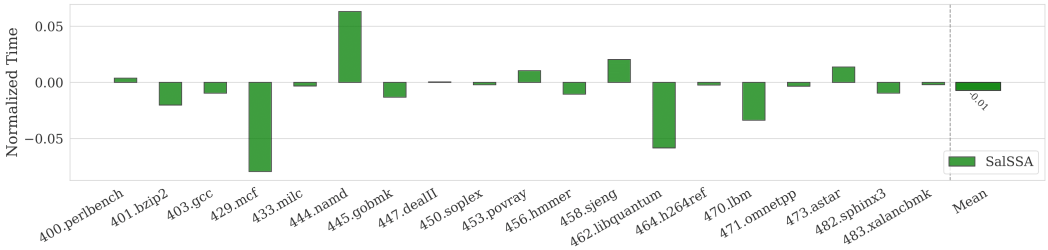

Figure 4.6: Relative Time of Local Gotoh to NW

4.3 Myers-Miller

Myers-Miller is an algorithm only intended to provide the functionality of Gotoh, while keeping memory linear. This should result in a similar amount of code reduction but do so with a performance cost. There shouldn't be a great deal to analyse here since Myers-Miller works to find an optimal alignment, the same as Gotoh, but do so with a slight computation cost. Except for the fact that Myers-Miller doesn't actually achieve

the same code reduction as Global Gotoh and does so with a very similar compilation time.

As for the slight variation in code reduction between them, I believe the reason for this is that they both find optimal alignments, however, an optimal alignment does not imply an optimal code reduction. It may require more than just an optimal alignment to reduce the file size as far as possible. Some alignments may work better than others, even though these alignments achieve the same optimal score. Though, the general idea of Myers-Miller providing Gotoh functionality can be seen by the high similarity in results between them in benchmarks, and the slight difference in their mean code reduction.

Looking at the time results of Myers-Miller, I am very surprised to see that it seems to have little to no negative effects when compared to NW. There are a few benchmarks where the extra computation cost is transparent, such as 433.milc, 458.sjeng, and 482.sphinx3. However, the degree at which these benchmark increase is not that significant, barely reaching 5% extra time. The other benchmarks seem to all sit right around how NW performs with a couple reaching a 2% improvement on time. I'm not sure what to make of these results, but if I was to take a guess, I would say that in benchmarks where the time has actually increased, these benchmarks contain many small sized functions where the discrepancies in time of each pairwise alignment begins to add together into an overall increased time.

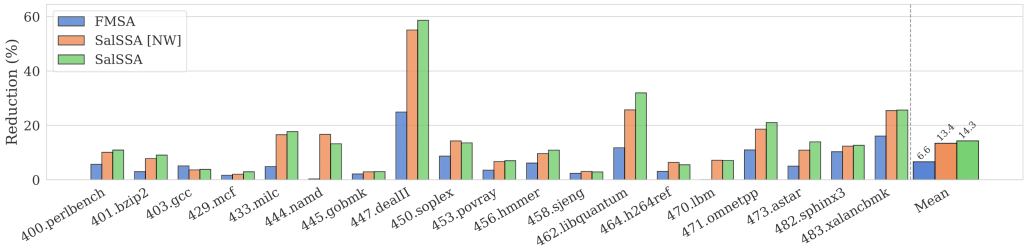

Figure 4.7: Code Size Reduction of Myers Miller

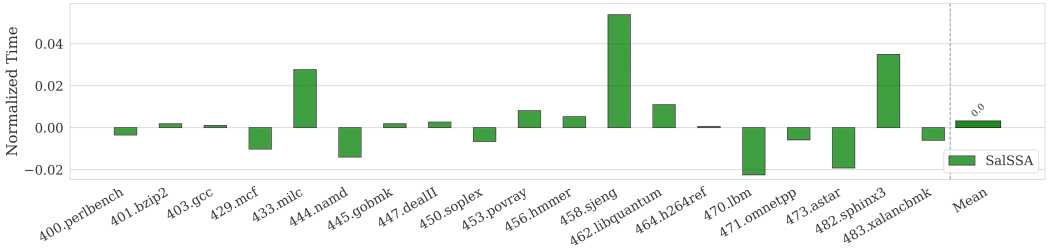

Figure 4.8: Relative Time of Myers-Miller to NW

4.4 FOGSAA

As I mentioned before in Section 3.5, FOGSAA did not turn out as well as I'd hoped with it's supposed time gains. And to re-iterate again here, it really is disappointing. Looking at the code reduction in Figure 4.9, you can see that it, on average, performs just as well as Needleman-Wunsch, which is to be expected considering that it attempts to beat Needleman-Wunsch in time only. On occasion, it outperforms and underperforms when compared to NW. I believe this occurs for the same reason as to why Myers-Miller does not exactly achieve the same code reduction as Gotoh. They both find optimal alignments, but those optimal alignments may not maximise code reduction as well as other optimal alignments. However, you can see that FOGSAA performs exactly as expected, with regards to code reduction, as it achieves the same mean code reduction as Needleman-Wunsch.

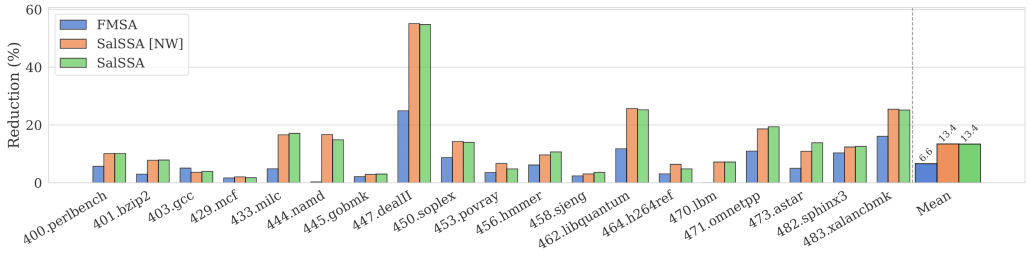

Figure 4.9: Code Size Reduction of FOGSAA

With the performance of FOGSAA in terms of time, it is really bad. For some benchmarks, it seems to perform just as well as NW like 429.mcf, 433.milc, 462.libquantum, 470.lbm, 471.omnetpp, and 473.astar. Aside from these benchmarks, it underperforms massively, nearly reaching heights of being 2.5x slower than NW in the 400.perlbench benchmark. On average, it increased time by 51% which is completely unacceptable.

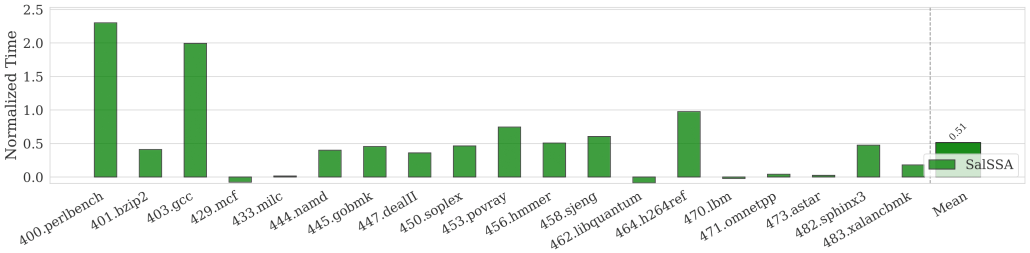

Figure 4.10: Relative Time of FOGSAA to NW

When it comes to the time performance of FOGSAA, there is an important aspect to consider. FOGSAA includes a fail-safe, where, if the algorithm detects that the sequences are not similar by a certain threshold, then it quits and returns the current alignment found along it's searching process. This means that it could return an alignment which is not optimal because it considers it not worthy of it's time to keep at-

tempting to search other alternative paths. The results above are from using a fail-safe of 30% where if it detects a similarity below that then the alignment process is quit and the current best alignment is returned. As you can see from those figures, this does not greatly affect the alignment quality as it has achieved the same mean reduction as NW. I tried changing this fail-safe value, but it did not help FOGSAA to become an algorithm worthy of considering. As the fail-safe threshold increases, the code reduction begins to decrease largely while the time taken still remains at a ridiculously high amount when compared to NW.

4.5 MUMmer

After reading about the power of suffix trees, I believed MUMmer would be the fast alignment algorithm I had been looking for. Because it was designed for aligning very large DNA sequences, I thought that applying it to only large functions would allow it to thrive. Looking at the results in Figures 4.11 and 4.12, I can see that I was correct in my thinking. MUMmer is actually able to provide a modest increase in code reduction of around 0.3% and does so with an average reduced compilation overhead of about 3% compared to Needleman-Wunsch.

The increase in code reduction goes against the idea that heuristic methods, which approximate the optimal alignment, would align the functions in such a way that code reduction is reduced when compared to aligning optimally. It may be a slight increase, but it goes to show that optimal alignments aren't always optimal for the function merging optimisation. I'm sure this ties into how the differences between Gotoh and Myers-Miller, or FOGSAA and NW, in code reduction can be attributed to the fact that some optimal alignments outperform other optimal alignments when it comes to maximising code reduction.

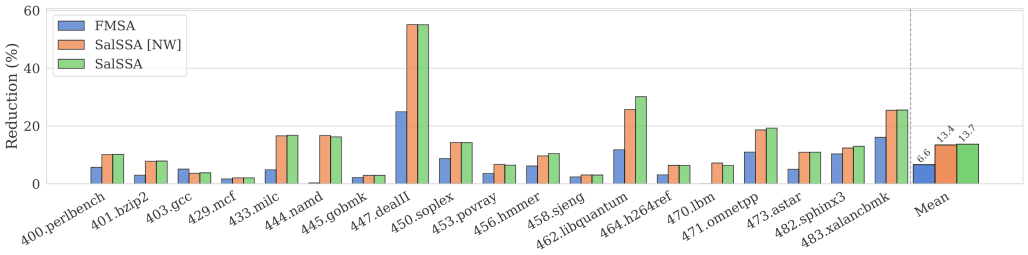

Figure 4.11: Code Size Reduction of MUMmer

From the time results seen in Figure 4.12, the effect of switching between MUMmer for large sequences and NW for short ones is unmistakable. Most benchmarks sit with little to no change in the time from NW, but multiple benchmarks have seen a significant reduction from anywhere between 5% to an astounding near 25% improvement. Because of this, it seems as though the average 3% improvement isn't that dramatic yet, in code-bases with large functions, the improvement over NW can be massive.

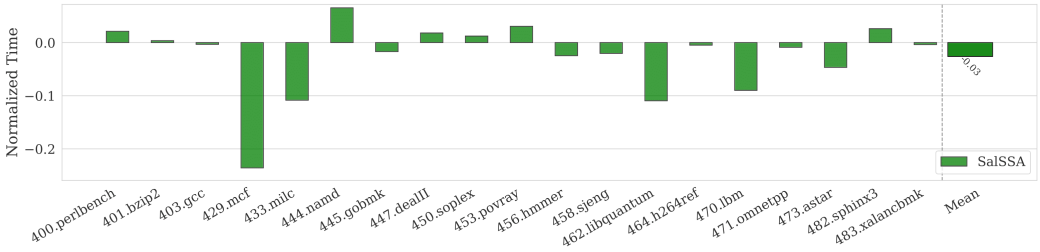

Figure 4.12: Relative Time of MUMmer to NW

4.6 BLA(S)T

Unfortunately, with my implementation of BLA(S)T, I was not able to have pattern matching via suffix trees working for instructions, yet when looking at the results in Figures 4.13 and 4.14, it shows that it was not such a big deal. Achieving exactly the same mean code reduction as NW while providing it at a 5% speedup.

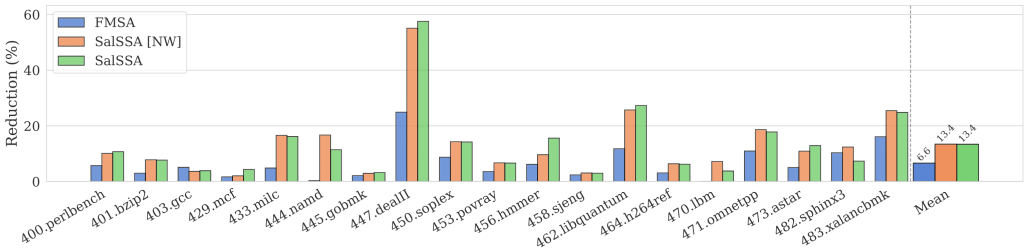

Figure 4.13: Code Size Reduction of BLA(S)T

The combination of pattern matching, anchoring, and locally aligning seems to have done a great job at improving the performance of sequence alignment. And it does so at no cost to the code reduction provided by the system. A significant number of benchmarks reach an improvement of 5% or more, and multiple manage to come close to or push past the 10% improvement hurdle. It is clear to see that BLAST will often outperform but never under-perform when compared to NW. The use of non-overlapping pattern matching has worked well in reducing time for many of the benchmarks but doesn't manage to reduce the time in a few benchmarks, however, it maintains the level that is provided by NW. With BLAST, it seems like you are always going to get a time equivalent or less than that provided by NW. And considering that it provides this speedup at no detriment to code reduction, BLAST should be far more preferred than NW.

The few benchmarks in which there is no reduction in time overlap with the ones in MUMmer where NW is chosen. This suggests to me that benchmarks like 445.gobmk, 447.dealII, 450.soplex, 453.povray, and 456.hmmer are code bases which include many small sized functions. For the effects of BLAST and MUMmer to be seen, the se-

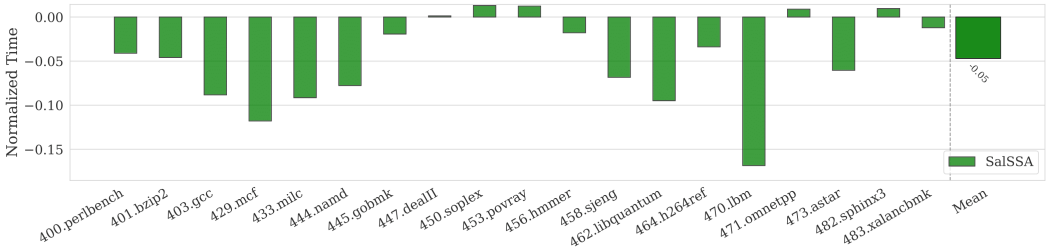

Figure 4.14: Relative Time of BLA(S)T to NW

quences must be quite large, and so, with these benchmarks, any performance improvement is insignificant.

4.7 Conclusion

To conclude the results section, I personally believe that the project was a success. When comparing to the current system, I have managed to achieve an increase in code reduction of up to 1.4% at no cost to compilation overhead. In fact, I achieve this with a reduction in compilation overhead by about 2%. I also have managed to achieve a decrease in compilation overhead by 5% while not discounting the code reduction by any amount.

Comparing to the state of the art, FMSA utilising the Needleman-Wunsch algorithm, I achieve over double the code reduction. This was not entirely from my contributions, however, I have managed to bring the code reduction from a 6.8% increase to an 8.2% increase over the state of the art.

Based on the results I can conclude that local alignments which are forced into global ones should be considered to be more valuable over naturally globally ones. I believe the increase in code reduction comes from the fact that the local alignment algorithms can focus on finding highly similar segments which should undoubtedly be merged, and are not held back by having to accommodate for the rest of the sequences which are not as similar. It seems to be more fruitful for the system to not even attempt to align these instructions but instead have them stuck on, possibly reducing the code generation brought about by these instructions.

I can also conclude that optimal alignments don't imply optimised code reduction. This can be seen quite clearly in the differences in Myers-Miller and Gotoh. They both achieve optimal alignments, however, they don't always agree on the amount of code reduction. This was also seen with MUMmer outperforming NW in terms of code reduction when MUMmer is built to approximate the optimal alignment, not actually achieve it. Even BLA(S)T which I built to also approximate the optimal alignment, achieves a mean code reduction equal to the amount provided by NW. The same is seen in FOGSAA, which does actually achieve an equivalent mean code reduction to NW, but varies slightly between the code reduction amount seen in benchmarks when compared to Needleman-Wunsch.

Perhaps analysis upon alignments and their resulting code reduction values could be

the key to finding alignments which actually result in optimal code reduction. This could open up alternative avenues for alignments which need not be optimal, in the sequence alignment sense, but instead optimise code reduction. It is also very beneficial for the system that it does not need optimal alignments to achieve higher reductions in code as this allows the system to use faster algorithms which will bring down the compilation overhead to a far lower degree.

If I was to take a guess as to why optimal alignments can vary in code reduction, I would say that this is caused by a combination of how local alignments need not consider dissimilar parts of a sequence and that code generation introduced into the final merged function negatively impacts the file size. There could be more to it but I believe these reasons to be the main sources. I'm not well versed in the code generation aspect of the system but could become something I look into next year.

Chapter 5

Discussion

Obtaining high quality sequence alignment while minimising the runtime is a challenge in BioInformatics. It's not a totally solved problem. In this field, fast heuristics dramatically improve the running time but they dearly compromise on alignment quality. This usually brings into question the alignment and whether it is of such a calibre that theories, predictions, and whatever else can be made off of them. But it seems in my case, that this isn't such an issue. Non-optimal alignments can perform just as well as optimal ones, sometimes better. Even optimal alignments which are equivalent in terms of their score aren't as equal as I thought they would be. In BioInformatics, optimal alignments are usually considered to be just as valuable as one another, yet, as a function merger, the code reduction achieved can shift drastically.

This discrepancy reveals a new idea I have not previously seen: optimal alignments are not necessarily optimal for systems which make use of them, especially outwith the context of DNA analysis. Altering variables like the scoring scheme will affect the finalised results, as to be expected. But when no variables have been changed and the resulting alignments are calculated to have the same score, why is it that the results can still differ? In the case of function merging, I reason that code generation plays a significant part, if not the only part. I suspected this to be the case for alignments produced by algorithms which do not take gap openings into account. Where a gap opening represents code generation maintaining function ordering so that when the merged code targets a specific function, it only runs those instructions specific to that function. However, as seen with the results of Myers-Miller and Global Gotoh, accounting for this still does not bring about equivalent results.

Moreover, the fact that heuristically based algorithms, designed to make trade offs on code reduction in favour of efficiency, have achieved equivalent or better code reduction is very unexpected. This leaves me without having to make any justification for using an algorithm which runs 5% faster and does so with no detriment to the quality of the merged code. Considering how my implementation of BLAST and MUMmer work, I can draw another conclusion as to why they perform just as well as optimal algorithms. Each one finds maximal matches, which must work quite well from a reduction in code generation standpoint.

I previously explained how MUMmer finds maximal matches in the algorithm description, however, with BLAST I didn't spell it out. Because BLAST is constantly trying to expand matched words, they eventually become fully expanded and are then used as anchors. BLAST's anchors are therefore maximal. I believe that maximal matches result in reduced code generation. Assuming that an optimal alignment achieves a set of matches and gaps where the average lengths of these matches are shorter than the set of matches/anchors obtained by MUMmer or BLAST, then I would say that the probability of gap openings in these heuristic algorithms is reduced. The more instructions taken up by stretches of matches should result in fewer gap openings. This does not mean fewer gaps as NW optimises for the least number of gaps possible, but it does not optimise for the least possible number of gap openings. Though, MUMmer and BLAST both rely on NW to bridge the gap between their anchors, which could open up the opportunity for the gap openings to be created by NW.

I also find it strange that certain dynamic programming solutions which should run slower than or equivalent to NW, do so at such a minimal scale or actually outperform it. Algorithms like Global Gotoh and Myers-Miller seem to have made little to no impact on the time that their increase in code reduction makes them far more favourable. Smith-Waterman and Local Gotoh beat NW in time that it makes it seem like they are doing something special in comparison but they still run with the same overall time complexities as NW. The extra calculations per loop seem to play a minimal part in the overall time of each algorithm, conceivably allowing for more complex algorithms with the same time complexity. The difference in the time spent aligning sequences is most likely what is the cause for the speedup. With local alignments, they have the advantage of only needing to trace-back as far as their alignment requires, with the rest being stuck on in a quick and trivial manner. I'm sure BLAST also takes advantage of this considering it does not align to the ends of the sequences like in MUMmer. This is probably why it has managed to come out on top as the fastest algorithm.

Another thing to note is that MUMmer and BLAST seem to improve upon time in similar benchmarks. They share an overlap of improved benchmarks, suggesting to me that the improvements in time come from large functions where algorithms like MUMmer and BLAST can thrive. Time gains in small functions are essentially negligible and so benchmarks comprising of many small functions are likely not going to allow for much improvement. But in benchmarks where this is possible, BLAST and MUMmer kick in and produce dramatic upgrades in time. In the benchmarks in which there is little to no change, algorithms like Local Gotoh and Smith-Waterman improve upon a couple of these suggesting that there is some room for improvement with the small sized functions. However, when I attempted to access this potential time gains, I would end up with far worse average timings, often underperforming when compared to NW. There is probably some kind of interaction here which I am not aware of causing this.

What's most important to consider is that this technique is in it's infancy and has a great deal of room for improvement. In just a year of development, since FMSA, the system has achieved over double the code reduction provided by FMSA. And with my time improvements, it does so at a vastly reduced time compared to the state of the art. BioInformatics is also ever developing and maturing, and further optimisations in each field could be of benefit to one another.

In this thesis, I have managed to utilise heuristically based sequence alignment algorithms to maintain a level of code reduction provided by an optimal algorithm, while doing so at a far greater speed. I have also taken alternative dynamic programming approaches to increase code reduction with little to no negative impact on the compilation overhead.

Chapter 6

Improvements and Extensions

6.1 Improvements to current project

Firstly, I feel fixing the pattern matching within my suffix tree implementation so that my BLA(S)T implementation will run faster for large sequences should be one of my main priorities. Even though I was unsuccessful in fixing this problem earlier, I can't imagine this to be a burdening task. It would just be a matter of whether it is worth it or not to even try, considering it runs quite fast using the quadratic method anyway.

FOGSAA [5] would absolutely be the next improvement on my list, maybe even coming as the top priority. It's a shame that I spent such a huge amount of time dedicated to this one algorithm trying to squeeze every bit of performance out of it as I could, yet it just never seemed to perform as well as described by the original authors. Perhaps, with a fresh mind, I could revisit the algorithm and figure out why it runs so much slower than expected, but for now, I'm stumped.

Another improvement that could be implemented is that, in MUMmer and BLAST, I do not make use of the lengths of the sequences to decide which anchors should be taken for the final alignment. This is not hugely important since I still achieve very good accuracy without it, but it would be very interesting to see how it affects the final alignment quality and resulting code reduction. Speaking of MUMmer, I should find out why the one in one thousand occurs where a MUM has a bad index in the sequences.

Also, there are a couple more algorithms that I feel would have been interesting to implement and evaluate. These include: Myers bit-vector algorithm [50] utilising Ukkonen's banded algorithm [51], combined with Hirschberg's linear space algorithm [23] to obtain the optimal alignment, as implemented by Edlib [52]; ACANA [53] which utilises the heuristic anchoring method, however, proposes a Smith-Waterman-like dynamic programming algorithm to recursively identify near-optimal regions as anchors; and LAGAN [29] utilising the CHAOS algorithm [54] to find the anchoring regions and then constructing a rough global alignment map used to chain the anchors together. Because of how using anchors as heuristic methods will under-perform with small sized sequences, I believe Myers bit-vector algorithm to be the most promising

though it is still quadratic in nature.

6.2 Extending to Fifth Year

Throughout the development of the project, I thought of a few possible extensions that would allow me to contribute further into my fifth year. These are:

- In Chapter 4, I talked a little about possibly analysing alignments of functions and determining which alignments and why produce the best resulting code reduction. I believe that code generation can play a significant part in this and is something I'd need to thoroughly understand to be able to make decisions on this. A clear understanding of the relationship between functions and alignment algorithms will also allow me to exploit potential correlations. This could be something that opens up avenues for creating new algorithms which optimise for code reduction specifically instead of as an optimisation of the scoring scheme. This type of understanding should prove to be quite invaluable to the project as a whole.
- BLAST is predominantly a search tool. Currently, the system employs a lightweight ranking infrastructure that uses a fingerprint of the functions to evaluate the similarity between functions. By comparing the opcode frequencies of two individual functions and the type frequencies, the system calculates upper and lower bounds for the two functions and uses these to determine which function pairs would be best to merge. BLAST, on the other hand, is used for taking a DNA or protein sequence and comparing it to a database filled with millions of other sequences and finds the one that it best aligns to. If you consider a database of DNA sequences to be like the code-bases list of functions, then you could potentially take BLAST and use it to find the pairs of functions that are best aligned and return the alignment at the same time. This process could even be parallelised, and so multiple functions are being searched and aligned at the same time.
- As mentioned in the improvements section, there were a few extra algorithms that I was interested in implementing. These would make a nice addition to the collection of algorithms already implemented by myself and the original authors.
- I have noticed through my testing that even exactly equivalent instructions are not matched. I saw this through attempting to match a function with itself, which should return that every instruction matches and thus an alignment of matches that spans the entirety of the sequence would be made. This is not the case. I noticed that it would occasionally deem two equivalent instructions as unequal. Perhaps as a future improvement, I could look into the match function used and fix any sort of error like this. This is not to say that this is not already under scrutiny and being fixed as I write this, but it's still a problem that I found which needs fixing.

Chapter 7

Reflections

When I look back at the entire project I can definitely say that I enjoyed most of it. It introduced me to a whole new side of computing, BioInformatics, which is utterly fascinating, forced me to work with and learn templates which I had no prior reason to, and gave me a closer insight into full system development. This is valuable experience that I am very glad to have been given.

Perhaps the strangest aspect of learning an entirely new topic and then working on it as a year-long project, is that, at first, I thought the initial things I was trying to do were complicated, difficult and certainly not straightforward. However, when I look back at my early stages now, I feel like these topics were simple or easy and should not have caused me any trouble. This was compounded by the fact that heuristic level sequence alignment techniques seemed to be locked away from most teaching and that my only sources became purely academic papers. The stuff that was readily taught at first allowed me to learn quickly and expand my knowledge but as soon as that dried out, I was left with a far harder experience. Certain techniques could only be found through references from other papers, leading me down a rabbit hole of new and exciting potential methods which I could try to apply to my project. At first, sequence alignment is taught at, what seems like now, a very basic level. It is not until you go through the academic literature that you realise how in-depth and vast this topic can get. Obviously, this should be the case, considering DNA analysis is incredibly important in the world of genomics, but it just felt like, when I first started learning, a bunch of online articles and lecture slides would give me most of the knowledge I would need to know. This was not the case.

Searching for methods with the goal of speed improvements was a hard process, aside from BLAST, which is so readily taught that it actually gets in the way of discovering other methods. It was easier to find one that is only vaguely related, and then follow the links through their references to get to what you actually want. I even posted on a forum asking BioInformaticians for their recommendations of fast sequence alignment methods, as well as emailing a professor at this school who then informed me of the MUMmer algorithm.

Because of the ”openness” of the project, I often didn’t know how much would be

enough or when to stop. I had plans to implement a few more algorithms or at least provide extra features to some existing ones, but implementation difficulties would always hold me back. I'm very happy with the effort I put in, and the amount of work I have managed to get done, yet at each stage of development I always wanted to do something more. "One more algorithm" seemed to be something I would tell myself whenever I finalised an implementation. Because of only finding new methods through academic papers, I feel that I was discovering better and better methods continuously, but without the time to implement them. I even had ideas of coming up with my own algorithm purely designed for speed and efficiency.

The topic can extend far into other ones likes approximate string match, tree building, and edit distance calculations. It may seem obvious to see that they're all related and should be researched for my purpose, but when I first started, the only keywords I knew were function merging and sequence alignment. And it wasn't until far into the development of the project that I began discovering some of this stuff.

With all this said, I really did enjoy learning all the new concepts and ideas. It felt really good to be looking through papers released in the past 5 years and thinking I was on the cutting edge. Not to mention how pleased I am that I can take this learned knowledge and apply it a second time around.

Bibliography

[1] Rodrigo Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, and Hugh Leather. Function Merging by Sequence Alignment. In *Proceedings of the 2019 International Symposium on Code Generation and Optimization*, pages 149–163, United States, 2 2019. Institute of Electrical and Electronics Engineers (IEEE). Date of Acceptance: 29/10/2018.

[2] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[3] Osamu Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162(3):705–708, 1982.

[4] Eugene W. Myers and Webb Miller. Optimal alignments in linear space. *Bioinformatics*, 4(1):11–17, 03 1988.

[5] Chakraborty Angana and Bandyopadhyay Sanghamitra. FOGSAA: Fast Optimal Global Sequence Alignment Algorithm. *Scientific Reports*, 3(1):1746, 2013.

[6] Robert D. Fleischmann Jeremy Peterson Owen White Steven L. Salzberg Arthur L. Delcher, Simon Kasif. *Alignment of whole genomes*, 14-03-2020. http://mummer.sourceforge.net/MUMmer.pdf.

[7] Pinaki Chakraborty. Fifty years of peephole optimization. *Current Science*, 108(12):2186–2190, 2015.

[8] Saumya Debray, William Evans, Robert Muth, and Bjorn De Sutter. Compiler techniques for code compaction. *ACM Trans. Program. Lang. Syst.*, 22:378–415, 03 2000.

[9] Chris G. Demetriou. Safe icf : Pointer safe and unwinding aware identical code folding in the gold linker. 2010.

[10] Ian Lance Taylor. A new elf linker.

[11] Tobias J. K. Edler von Koch, Björn Franke, Pranav Bhandarkar, and Anshuman Dasgupta. Exploiting function similarity for code size reduction. In *LCTES '14*, 2014.

[12] VI Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, 1966.

[13] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.

[14] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[15] *Pairwise Sequence Alignment using Biopython*, 15-03-2020. https://towardsdatascience.com/pairwise-sequence-alignment-using-biopython-d1a9d0ba861f.

[16] James D. Watson and Francis H. Crick. Molecular structure of nucleic acids: A structure for deoxyribose nucleic acid. *Nature*, 171:737–738, 1953.

[17] *Advanced Sequence Alignment*, 15-03-2020. http://www.csbio.unc.edu/mcmillan/Comp555S16/Lecture14.html.

[18] S Henikoff and J G Henikoff. Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences*, 89(22):10915–10919, 1992.

[19] Wikipedia. *DNA codon table*, 28-01-2020. https://en.wikipedia.org/wiki/DNA_codon_table.

[20] *Needleman-Wunsch*, 15-03-2020. http://rna.informatik.uni-freiburg.de/Teaching/index.jsp?toolName=Needleman-Wunsch.

[21] NCBI. *Basic Local Alignment Search Tool*, 28-01-2020. https://blast.ncbi.nlm.nih.gov/Blast.cgi.

[22] M.S Waterman, T.F Smith, and W.A Beyer. Some biological sequence metrics. *Advances in Mathematics*, 20(3):367–387, 1976.

[23] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343, June 1975.

[24] *Notes on Dynamic-Programming Sequence Alignment*, 15-03-2020. http://globin.cse.psu.edu/courses/spring2000/DP.pdf.

[25] Wikipedia. *Branch and Bound*, 14-03-2020. https://en.wikipedia.org/wiki/Branch$_a$nd$_b$ound.

[26] GeeksforGeeks. *Branch and Bound Algorithm*, 14-03-2020. https://www.geeksforgeeks.org/branch-and-bound-algorithm/.

[27] A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.

[28] RJLipton and KWRegan. *Branch And Bound—Why Does It Work?*, 14-03-2020. https://rjlipton.wordpress.com/2012/12/19/branch-and-bound-why-does-it-work/.

[29] Michael Brudno, Chuong B. Do, Gregory M. Cooper, Michael F. Kim, Eugene Davydov, NISC Comparative Sequencing Program, Eric D. Green, Arend Sidow, and Serafim Batzoglou. Lagan and multi-lagan: efficient tools for large-scale multiple alignment of genomic dna. *Genome research*, 13(4):721–731, Apr 2003. 12654723[pmid].

[30] Wikipedia. *Suffix tree*, 14-03-2020. https://en.wikipedia.org/wiki/Suffix$_t$*ree*.

[31] Guillaume Marçais, Arthur L. Delcher, Adam M. Phillippy, Rachel Coston, Steven L. Salzberg, and Aleksey Zimin. Mummer4: A fast and versatile genome alignment system. *PLOS Computational Biology*, 14(1):1–14, 01 2018.

[32] Peter Weiner. Linear pattern matching algorithm. pages 1–11, 11 1973.

[33] Edward M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, April 1976.

[34] Ukkonen E. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[35] Giegerich R. and Kurtz S. From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction. *Algorithmica*, 19(3):331–353, 1997.

[36] Wikipedia. *Online algorithm*, 14-03-2020. https://en.wikipedia.org/wiki/Online$_a$*lgorithm*.

[37] jogojapan edited by TeWu. *Ukkonen's suffix tree algorithm in plain English*, 15-03-2020. https://stackoverflow.com/questions/9452701/ukkonens-suffix-tree-algorithm-in-plain-english.

[38] GeeksforGeeks. *Ukkonen's Suffix Tree Construction*, 15-03-2020. https://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-1/.

[39] Mark Nelson. *Fast String Searching With Suffix Trees*, 15-03-2020. https://marknelson.us/posts/1996/08/01/suffix-trees.html.

[40] Duke. *Suffix Trees and its Construction*, 15-03-2020. https://www2.cs.duke.edu/courses/fall14/compsci260/resources/suffix.trees.in.detail.pdf.

[41] Ezo Onukwube. *The Wonders of the Suffix Tree through the Lens of Ukkonen's Algorithm*, 15-03-2020. https://humanreadablemag.com/issues/0/articles/the-wonders-of-the-suffix-tree-through-the-lens-of-ukkonen%E2%80%99s-algorithm/.

[42] Brenden Kokoszka. *Visualization of Ukkonen's Algorithm*, 15-03-2020. https://brenden.github.io/ukkonen-animation/.

[43] Wikipedia. *Longest increasing subsequence*, 15-03-2020. https://en.wikipedia.org/wiki/Longest$_i$*ncreasing*$_s$*ubsequence*.

[44] Kent W James. BLAT–the BLAST-like alignment tool. *Genome research*, 12(4):656–664, April 2002.

[45] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, June 1975.

[46] *BLAST Algorithm*, 15-03-2020. https://www.researchgate.net/figure/Schematic-illustration-of-the-BLAST-algorithm-Originally-published-in$_fig1_1$1875325.

[47] Stephen F. Altschul, Thomas L. Madden, Alejandro A. Schäffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research*, 25(17):3389–3402, 09 1997.

[48] Abdullah N. Arslan, Ömer Eğecioğlu, and Pavel A. Pevzner. A new approach to sequence comparison: normalized sequence alignment . *Bioinformatics*, 17(4):327–337, 04 2001.

[49] Xiaoqiu Huang and Webb Miller. A time-efficient, linear-space local similarity algorithm. 1991.

[50] Gene Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *J. ACM*, 46(3):395–415, May 1999.

[51] Esko Ukkonen. Algorithms for approximate string matching. *Inf. Control*, 64(1–3):100–118, March 1985.

[52] Martin Sosic and Mile Sikic. Edlib: a c/c ++ library for fast, exact sequence alignment using edit distance. *Bioinformatics (Oxford, England)*, 33(9):1394–1395, May 2017. 28453688[pmid].

[53] Weichun Huang, David M. Umbach, and Leping Li. Accurate anchoring alignment of divergent sequences. *Bioinformatics*, 22(1):29–34, 11 2005.

[54] Michael Brudno, Michael Chapman, Berthold Göttgens, Serafim Batzoglou, and Burkhard Morgenstern. Fast and sensitive multiple alignment of large genomic sequences. *BMC Bioinformatics*, 4(1):66, 2003.