UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

EVALUATING LANGUAGES FOR BIOINFORMATICS: PERFORMANCE, EXPRESSIVENESS AND ENERGY

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

By

RANDY J. RAY
Norman, Oklahoma
2022

EVALUATING LANGUAGES FOR BIOINFORMATICS: PERFORMANCE, EXPRESSIVENESS AND ENERGY

A THESIS APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY THE COMMITTEE CONSISTING OF

Acknowledgments

This effort would have never been possible without the support and encouragement of many others.

Firstly, I would like to thank my adviser and the chair of the thesis committee, Dr. Sridhar Radhakrishnan. He was the first point of contact when I initially became interested in the Master's program at OU and had me excited about the potential from the very start. I would also like to thank the remaining members of the committee, Dr. Christan Grant and Dr. Le Gruenwald, for their role in this and for the courses I took from them.

Secondly, I would like to thank the members of the C++ and Rust communities on Reddit. Of the languages used in this research, these were the least-familiar to me and the ones I sometimes struggled with. Without the ready answers to my many questions, I would have had a much more difficult time getting the final results that I did. I would like to especially thank Andrew Gallant of the Rust community for his extensive advice and help with Rust in general and several of the programs in particular.

Next, I would like to thank my mother, whose words of support and encouragement often came at truly opportune times.

I would next like to thank my father, who passed away before I could begin this program but who would have been exceedingly proud of what I have been able to accomplish. Without his support, I would have barely managed to finish my undergraduate years at OU, let alone been in a position to return for a higher degree later in life.

Lastly, and most importantly, I thank my dear wife Eugenie. I truly believe that, without her help and support, I would not have gotten this far. She was my steady support, my cheerleader, my proofreader, my copy-editor, my sounding board for ideas, my rubber duck, and my project manager. But most of all, she was my partner. She kept me focused and kept me moving forward, even in the times I sat at our dining table and doubted my ability to finish this effort.

She never lost faith or confidence in me, and for that (among many many reasons) I dedicate this work to her.

Abstract

One of the fastest growing concerns in the technology sector is the increased demand for power in the world's data centers. Global data center electricity use in 2021 was estimated as between 220 and 320 terawatt-hours (TWh), as much as 1.3% of global electricity demand. As the data center industry continues to expand, so too will power usage, and therefore the need for increased energy efficiency in software development.

This thesis introduces a methodology that evaluates a set of programming languages based on three key metrics: performance, expressiveness, and energy use, demonstrating a fair consideration of each language's strengths and weaknesses. The framework presented creates a collection of string-matching algorithms used on DNA sequences to demonstrate the capabilities of each language, and draw out their distinctiveness.

DNA sequencing was chosen due to its growing uses and applications as technology evolves and makes such sequencing faster and less expensive. This in turn has lead to a growing percentage of compute-time being spent on this field. Using the methodology presented here it will be shown that using a newer language, like Rust, has advantages that help it balance speed, ease of use, and power consumption when used for advanced scientific computing.

A key part of this work introduces a novel approximate-matching algorithm to aid in this evaluation process. This new algorithm differs from current algorithms in use, in its ability to hold the gap between nucleotides to a specific maximum while allowing other gaps to exist. It will offer an alternative technique to other current approximate-matching algorithms and hopes to offer researchers another tool to consider for sequence-matching problems.

The expectation is that this research will show how testing and evaluating via performance, expressiveness and energy use metrics allows for rating and ranking programming languages in a consistent and reproducible manner. This will enable developers to make educated choices when selecting a language for a project. The methods described here will be applicable to other languages as well, given similar data to work with. This research will benefit the programming field by providing methods and techniques that can be used in the language selection process, particularly when energy efficiency is as important as overall performance.

Contents

List of Figures

List of Tables

Source Code Listings

1 Introduction

When evaluating a programming language for use on a project, programmers are faced with an ever-growing array of choices. These choices range from long-lived, well-established languages such as C or Fortran, to the very latest offerings such as Julia or Swift. Selecting a language is a process that is generally rooted in a combination of factors: suitability to the target platform, performance, expressiveness, and (often most importantly) the programmer's familiarity and comfort with the language.

This thesis will be a comparison of the relative strengths of five different programming languages when applied to five different algorithms. To explore this, the problem of large-scale string matching will be examined with a focus on matching DNA-like strings.

1.1 String Matching

The topic of string matching has long been a popular area of research in computer science. Before the paper by Knuth, Morris and Pratt in 1977 [20] there was already considerable work being done. In the same year, Boyer and Moore [10] published an improvement over the Knuth-Morris-Pratt algorithm with enhancements such as starting the match from the tail of the pattern rather than the head, and allowing for greater right-ward jumps through the string being searched. Even earlier, a 1975 paper by Aho and Corasick [1] described a method of searching for multiple patterns simultaneously in a given target string, trading a longer preparation time for the benefit of a construct that could be used over and over on different target strings.

String matching algorithms take many different forms, from simple indexing-based to suffix trees, character comparisons to bit vector operations, and serial to massively-parallel driven by the latest in GPU advances. A search through the Google Scholar service during the preparation of this paper counted nearly 60,000 results matched across diverse disciplines that employ string matching as part of their algorithmic problem-solving processes.

But matching strings means extensive reading and manipulation of strings. These strings are blocks of allocated memory, which can lead to program errors and vulnerabilities. A large percentage of security vulnerabilities discovered in programs are traced back to memory-related issues; in [15], Google software engineers are quoted as attributing roughly 70% of serious security bugs in Chrome to memory management and safety bugs. The article goes on to report that analysis from Microsoft echoes this number. As such, a process to write programs that are more stable and secure must include careful consideration of memory-related challenges.

1.2 DNA Strings

For this study, string matching will be applied to the problem of finding sub-sequences within strings created to emulate DNA (Deoxyribonucleic Acid) sequences. DNA sequence strings have interesting properties, in that they can be *extremely* long but at the same time the alphabet is limited to just four characters ("A", "T", "C", and "G"), called "bases".

Figure 1: DNA and the four bases

In [18] Heather and Chain say, "It is hard to overstate the importance of DNA sequencing to biological research". Today's researchers use ever-increasing computing resources to process this data faster and in a more complete fashion. String-matching algorithms, mathematical models and other tools have become vital to these research fields, as the size and quantity of data produced by sequencing has also grown. In just the two decades since the completion of the Human Genome Project, current sequencer technology has advanced to the point of being capable of producing as much as a terabyte of data per day [11].

1.3 Comparison Bases

Implementations of the selected algorithms will be developed in five languages: C, C++, Rust, Perl, and Python. Each language's implementations will be evaluated against the others on three bases:

1. **Performance**: Run-times for each solution will be gathered using existing timer mechanisms. Time-measurements will be somewhat coarse, as overhead operations such as I/O will necessarily be included in the times.

2. **Expressiveness**: Each solution will be measured on several source-level metrics in an effort to evaluate the expressiveness of the code.

3. **Energy efficiency**: Energy usage will be measured for each solution using the Running Average Power Limit (RAPL) tools available on Intel processors. RAPL will be outlined in greater detail in section 4.1.4.

These three bases cover modern concerns in software development: the general performance of an application, the readability/maintainability of the application, and the power consumption of the system running the application. Where the first two criteria are well-known and common, the last has been chosen based on steadily-growing concern over power consumption in the data center industry and in the mobile computing field [26].

2 Motivations and Prior Work

This research began initially as an effort to demonstrate the suitability of the Rust programming language for the bioinformatics field. An exploration of the Rust-Bio project [21] led to finding the SeqAn[1] project for C++. Further investigation led to an understanding of the ongoing popularity of languages like Perl and Python in this area, as well. It was decided that, rather than focus specifically on Rust and its potential, this effort would instead pursue an understanding of the relative power of a selected set of languages on the metrics described.

It is believed that these three measurements can evaluate the languages with enough clarity that a programmer could make an informed choice as to which would better meet their needs.

2.1 Programming Languages

Programming languages have a long history. The first commercially-available compiled language was FORTRAN, first appearing in 1956. But there were languages before FORTRAN, languages that were highly specialized and often relied on obscure syntax. Languages evolve and new languages emerge as the applications of computing and the needs of software grow and expand. Some languages, such as C and later versions of Fortran, persist even as new languages intended to replace them fall out of favor and die off. When examining what makes a language successful, it is necessary to look at multiple factors:

- How well does it perform? How fast are the programs written in the language?
- What aspects of problem-solving are made easier by the language? What aspects are made harder?

[1]SeqAn: `https://www.seqan.de/`

- How difficult is it to develop software in the language? How difficult is it to maintain?

A comparison of languages is not only predicated on their speed but also on readability, expressiveness, and capability. A language must be able to perform, but it must also be understandable. Jokes about the relative readability and maintainability of different languages date back to the APL language if not earlier than that. Setting aside endeavors such as obfuscated code competitions, some languages are simply harder to read than their peers. Perl and Python are often compared in this regard, for example. Perl's syntax relies heavily on the use of non-alphabetic symbols (referred to as "sigils") in using and referencing variables. Contrast this with Python's comparatively clean syntax, which is closer in style to that of C and other similar languages.

A full treatment on the discipline of programming language design is outside the scope of this writing. Instead, issues of the more aesthetics-oriented language differences will be addressed by examining some static aspects of the code, aspects that are completely independent of the running of the programs themselves.

2.2 Performance, Expressiveness, Energy

The experiments that will be described in this paper were designed to focus on a trio of aspects of concern to modern software developers: how well the code performs, how easy it is to read and maintain the code, and (more recently) how the code ranks in terms of energy efficiency.

The overall performance of programs is an issue often discussed when languages are compared directly. Languages such as C and C++ offer high performance, while interpreted languages like Perl and Python have comparatively poor performance. And yet, Python holds great popularity in many sub-fields such as data science, machine learning, data visualization, and task automation. It leads to the question: why would a language so much slower would be so popular?

While there are many varied reasons why people developing software like or dislike a given language, certain aspects often rise up in conversations. These aspects include the *friendliness* of the language, the *ease of use* it offers, and the *readability* of the language. Aspects like this are sometimes referred to as the *expressiveness* of a language: the breadth of ideas that can be represented in that language, and the degree to which they can be understood and communicated.

Expressiveness can be a significant factor in language selection and use. In [7], Berkholz looks at measuring expressiveness by looking at how many lines of code change in an average version control commit for projects written in a range of languages. He found functional languages such as Lisp and Haskell to be the most expressive, and domain-specific languages to be biased towards high levels of expressiveness.

In addition to performance and expressiveness, the energy usage of software is rapidly becoming more important as data-centers try to reduce carbon footprints and developers target battery-driven mobile devices.

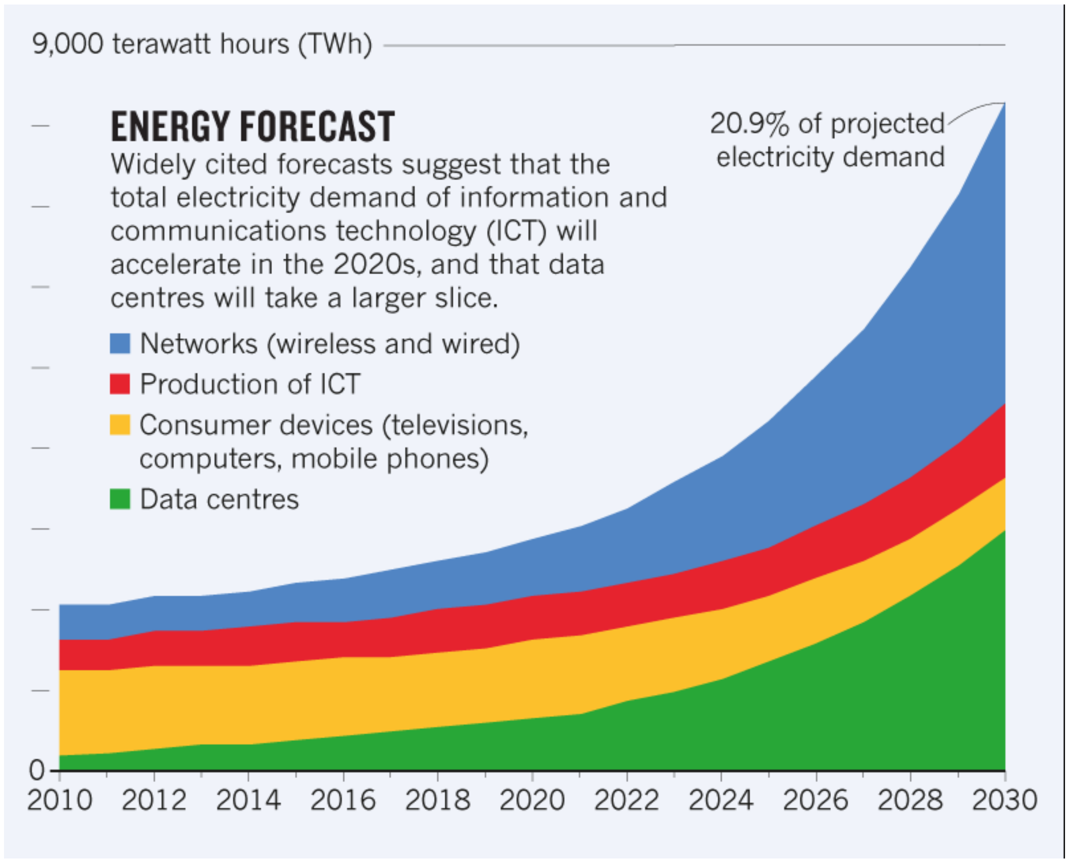


Figure 2: Projected energy demands through 2030 (original source [17])

In [17], the author reports that the U.S. data center industry alone "consumed around 196 to 400 terawatt-hours (TWh)" in 2020. And as their graph above predicts, this could become significantly higher by 2030. But in [22] the authors point out that actual server energy use is on average only 43% of data center usage. And yet, if accurate, this means that server energy could account for anywhere from 84 to 172 TWh. Even a small reduction in server energy could have an impact.

In [26], Pereira, et al did an extensive analysis of energy efficiency at the programming language level. Upon seeing the results and the methods used, it became clear that this should be used with the previous two metrics to evaluate a set of languages in even broader terms.

2.3 Prior Work

Some of the papers that informed this research include:

Rahate and Chandak [28] performed a study focused on algorithm performance similar to what is planned here. From this paper, it was determined that there should be at least five algorithms under consideration and that algorithms such as Knuth-Morris-Pratt [20] would make good candidates.

In [13], Chen and Nguyen describe an approach to string matching over DNA data with k differences. Their technique was based primarily on *edit distance*, and lent ideas to the development of the k-gap approximate-matching algorithm that will be described in 3.5.

Neamatollahi, et al [24] describe three pattern matching algorithms that are specifically targeted at searches on large DNA sequences. While the first is a more traditional character-based matching algorithm, the second and third take advantage of aspects of the CPU such as word-width to speed up comparisons.

The concept of multiple-pattern-matching for exact matches is explored by Bhuka and Somayajulu in [8]. Their approach is based on the use of pair indexing for both the sequence and the pattern. This paper gave weight to the idea of multi-pattern matching and led to the decision to take the Aho & Corasick algorithm [1] as one of the evaluation algorithms.

In [14], Cheng, et al describe a novel data structure and use it in two new parallel approximate matching algorithms. Ultimately, this was not used directly as the multiple-machine clusters would have greatly increased the complexity of taking energy usage measurements.

3 Selected Algorithms

Guided by [28], five algorithms in total were chosen to be used in providing the basis for evaluating the languages under scrutiny. Of the five, four are exact-matching algorithms and one is an approximate-matching algorithm. One algorithm matches multiple patterns in a single examination of a sequence, while the remaining algorithms match only single patterns.

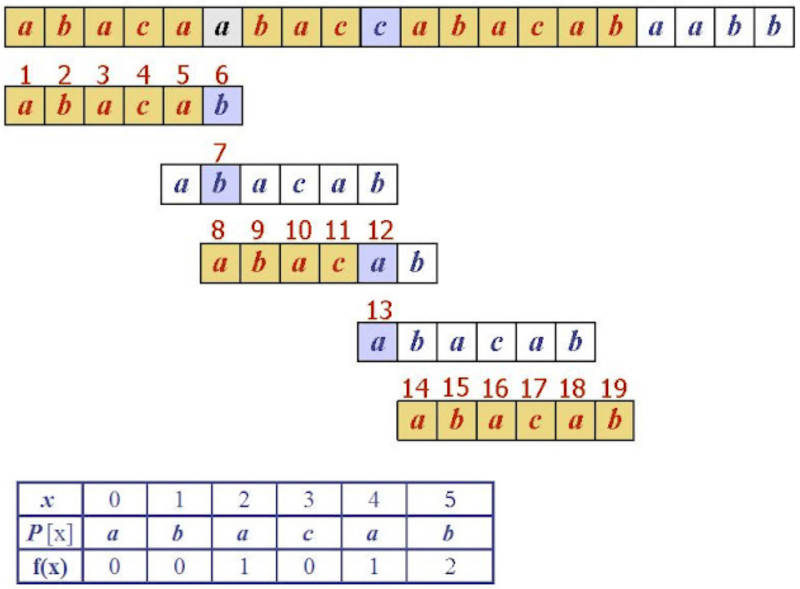
In this section, these algorithms are introduced and the first four briefly explained. The fifth algorithm will receive a more in-depth treatment due to it being a new approach. An understanding of the underlying mechanics of the algorithms will be helpful when later evaluating their implementations.

During the process of running experiments and analyzing the results, it was decided to add a variation of the fifth algorithm to the existing set. As a variation, it is not covered in the same depth as the fifth and original algorithm is. Instead, it is dealt with in greater detail in section 4.5.6.

3.1 Knuth, Morris, and Pratt

The Knuth-Morris-Pratt [20] algorithm is one of the foundational algorithms in the area of string matching and text searching. It is still used in the present day, with implementations as recent as the Rust-Bio project [21]. This was chosen primarily for historical significance, but also for ease of implementation.

Knuth-Morris-Pratt is an *exact matching* algorithm, meaning that it matches the desired pattern exactly or not at all. It finds all instances of the pattern within the target string, including overlapping instances, in time-complexity linear to the sum of the pattern length and the target length.



x	0	1	2	3	4	5
P[x]	*a*	*b*	*a*	*c*	*a*	*b*
f(x)	0	0	1	0	1	2

Figure 3: Example of Knuth-Morris-Pratt (original source [30])

Generally when matching, the pattern is aligned with positions in the target string. Shifting is done when a character mismatch is found between an index in the target and the corresponding index in the pattern. Where a naive implementation might shift the pattern by one place after each failed match, resulting in a time complexity approaching O(mn), the Knuth-Morris-Pratt approach is built on the concept of an auxiliary table (referred to as the “next” table) that instructs the matching algorithm on how much to shift the pattern over the target stream. Based on repetition within the pattern itself, the table may call for a shift of the pattern by more than one character for a given mismatch.

The computation of this table is shown in the paper to require O(m) steps, and the process of matching the pattern to the target takes at most an additional $2n$ steps. This is due to the fact that, at each step of the matching process, only one of the text pointer or the pattern pointer are moved (each of which can only move n times at most). This results in a worst-case run-time bounded by $O(m + n)$.

As an example, consider a search for the pattern "CTAGC" in a sequence that starts with "CGCCTAGCG". The first step is to compute the "next" table according to the algorithm, shown in figure 4.

j	0	1	2	3	4	5
$next(j)$	-1	0	0	0	-1	1

Figure 4: Knuth-Morris-Pratt next-table

With the table in hand, the process moves to matching. The matching algorithm goes through the following steps:

1. i and j both initialize to 0
2. $p_0 = s_0$, so *next* is not consulted
3. i and j increment, both to 1 and 1
4. $p_1 \neq s_1$, so $i = next[1]$ and becomes 0
5. $p_0 \neq s_1$, so $i = next[0]$ and becomes -1
6. i and j increment, to 0 and 2 respectively
7. $p_0 = s_2$, so *next* is not consulted
8. i and j increment
9. $p_1 \neq s_3$, so $i = next[1]$ and becomes 0
10. $p_0 = s_3$, so *next* is not consulted
11. i and j increment
12. p_i continues to match s_j as both variables increment. When $i = 5$, the algorithm detects a match.

Here, the Knuth-Morris-Pratt algorithm has found the match in 10 character comparisons, 5 of which were required to verify the full match.

3.2 Boyer and Moore

In the same year that Knuth, Morris and Pratt published their paper, Boyer and Moore published as well [10]. This algorithm is also an exact matching approach, that is based on the research of Knuth, et al. The Boyer-Moore performance improvements are based on searching from the end of the pattern rather than the beginning, and computing two tables to use in optimizing the jumps through the sequence string when mismatches are discovered. This algorithm was chosen as an example of refinement and improvement of another sample algorithm.

Boyer and Moore postulated that, "more information is gained by matching the pattern from the right than from the left." For example: If the target character that corresponds to the current location of the last character from the pattern is not only a mismatch but also does not appear in the pattern at all, the pattern may then be shifted right by its full length. They refer to this algorithm as being "usually sublinear," meaning that when finding the location of the pattern within the target the number of compared characters is usually less than $i + m - 1$ (where m is the pattern length and i is the position within the target where the match of the pattern begins).

The first of the two tables is the simplest to compute. It is the size of the alphabet of the pattern and sequence[2], and it tracks the number of positions by which the pattern can be moved down the sequence without additional checking for matches. Boyer and Moore define each entry in this table as being m (the pattern's length) when the character does not appear in the pattern at all, and $m - i$ otherwise (where i is the right-most index within the pattern where the character does occur). Using the same example pattern and sequence as in the previous section, the first table (referred to as $delta_1$ in the paper and *bad_char* in the implementations) is shown in figure 5. Only the entries for the four characters that appear in patterns are shown.

	A	C	G	T
$delta_1$	2	4	1	3

Figure 5: Boyer-Moore $delta_1$ table

The second table (referred to as $delta_2$ in the paper and $good\_suffix$ in the implementations) is more complex to calculate, as it first requires calculation of suffixes within the pattern. In the paper, it is described as the distance that the pattern can be slid down in

[2]In these implementations, to avoid constantly translating the four characters into values between 0 and 3, the alphabet-size was set to 128 for convenience.

order to align the discovered sub-match (in the target) with the last $m - j$ characters of the pattern (where j is the index within $delta_2$), plus the additional distance that the pointer within the target must be moved so as to restart the matching process at the right end of the pattern. This table is shown in figure 6.

j	0	1	2	3	4
$delta_2$	4	4	4	4	1

Figure 6: Boyer-Moore $delta_2$ table

When using the two tables to select the amount to shift, it is possible that the $delta_1$ table's value may be negative. Because of this a *max* operator is applied to the two potential values and the largest possible shift is chosen. With both tables computed, the matching process begins with the pattern aligned to the starting character of the target string. The algorithm then goes through the following steps (where m is the pattern length and n is the sequence length):

1. j (the pointer within the sequence) initializes to 0
2. i (the pointer within the pattern) is set to $m - 1$ (4)
3. $p_4 \neq s_4$, so the tables are consulted
4. $delta_2(4)$ is 1, and $(delta_1(T) - m + 1 + i) = 3$
5. j advances by 3
6. i is set to $m - 1$ (4)
7. $p_4 = s_7$, so i is decremented
8. p_i continues to match s_{i+j} until i becomes -1
9. A match is recorded and j advances by $delta_2(0)$, to 7

While the Knuth-Morris-Pratt algorithm had made 10 character comparisons to find the match, Boyer-Moore makes only 6, 5 of which were required to verify the match.

3.3 Bitap

The Bitap algorithm (sometimes known as “Shift-Or”, or “Shift-Add”) was initially developed by Bálint Dömölki in 1964. In 1989 it was re-invented by Ricardo Baeza-Yates and Gaston H. Gonnet, and published in [4] in 1992. Here, it has been chosen for the distinctive approach when compared to the other algorithms.

In terms of time complexity, the algorithm is on the same terms as the previous ones, having a preprocessing time of $O(m + \sigma)$ (the length of the pattern plus the size of the alphabet) and a running time of $O(n)$. However, the operations it performs are all bit-oriented: shifts, complements, bitwise-and and bitwise-or. This resulted in this algorithm running significantly faster than either of Knuth-Morris-Pratt or Boyer-Moore on the experiment input data.

The structure of the algorithm is based on encoding the pattern in a vector of bitmaps. The vector has length equal to the alphabet size, and each element of the vector is a bit-field of width W, where W is the size in bits of an unsigned integer. W also limits the length of the pattern in this implementation, though it is possible to implement the algorithm for longer patterns.

The vector is initialized in the preprocessing phase first to an all-1’s value in each slot, and then modified through a single pass over the pattern. For each character in the pattern, the slot corresponding to that character has the bit that corresponds to the pattern position flipped to 0. The resulting vector fully encodes the pattern. Using the example pattern from the previous two algorithms (“CTAGC”), imagine that the elements of the vector (S) are exactly as wide as the pattern (5 bits) and that there are only the four elements corresponding to the restricted alphabet of the pattern. This is shown in figure 7.

$$\begin{aligned} \mathrm{S(A)} &= \texttt{11011} \\ \mathrm{S(C)} &= \texttt{01110} \\ \mathrm{S(G)} &= \texttt{10111} \\ \mathrm{S(T)} &= \texttt{11101} \end{aligned}$$

Figure 7: Bitap shift positions vector

Here, it is clear that the letter “C” appears twice in the pattern as there are two 0-values in S(C). Each of the other three letters appear just once. Examining the bits of each S-value shows exactly where each letter appears in the pattern.

During the process of computing the S vector, a “limit” value is also calculated: it starts out as 0, and has a number of bits set equal to the full width of the pattern. At the end of the loop that calculates S, the value of *limit* is shifted to the right by one bit, then subjected to a bit-complement operation. The result is a value that can be directly used in comparison, when determining if a match has been found.

The process of searching for a match begins with a *state* value of W bit-width, set to all 1's. The target string is read one character at a time. For each iteration of this loop, *state* is shifted to the left one bit (introducing a 0) and then `or`'d with the S value of the character under the index. If, after this operation (the "shift-or"), the value of *state* is less than the value of *limit* then a match has been found and will be reported.

j	s_j	$S(s_j)$	*state*	result
			`11111`	
0	C	`01110`	`11110`	`11110`
1	G	`10111`	`11100`	`11111`
2	C	`01110`	`11110`	`11110`
3	C	`01110`	`11100`	`11110`
4	T	`11101`	`11100`	`11101`
5	A	`11011`	`11010`	`11011`
6	G	`10111`	`10110`	`10111`
7	C	`01110`	`01110`	`01110`
8	G	`10111`	...	

Figure 8: Bitap matching process

Figure 8 shows the process of matching the example pattern to the same sequence used in previous algorithm examples. The j column indicates the index of the character in the sequence, s_j is the character at j, and $S(s_j)$ is the S-value for that character. The *state* column shows the value of that variable at each iteration, while the "result" column shows the value of *state* after the `or`-operation. Not shown is the *limit* value, which in this example is `10000` (16 decimal). On the row where $j = 7$, we see that the high bit of *state* is a 0 for the first time. This signals a match (*state* $<$ *limit*), and the matching index is $j - m + 1$, or 3.

In this example the Bitap algorithm found the match after 8 rounds of the shift-or operation. No direct character-level comparisons were made, nor were any equality comparisons made. The process of finding the match involved only the two bit-level operations and a single "less-than" comparison for each iteration of the main loop.

3.4 Aho and Corasick

Alfred V. Aho and Margaret J. Corasick published their algorithm for searching multiple patterns at once in 1975 [1]. In their algorithm, a finite number of patterns are merged into a single deterministic finite automaton (DFA) which can then be applied over any number of target strings. The DFA is capable of finding all locations of all patterns in the set, with overlapping, in a single pass through the target string. This makes the algorithm significantly faster than the others for the simple reason that Aho-Corasick scans each target string once, regardless of the number of patterns. By contrast, each of the single-pattern algorithms scans the target string once per pattern. The choice of this algorithm was driven by an interest in comparing a multi-pattern algorithm to the single-pattern options, and an interest in exploring the use of a finite automaton for the matching process.

Aho and Corasick designed the pattern matching machine as a collection of three functions:

- A "goto" function g, which maps transitions from one state to the next based on the character being examined
- A "failure" function f, which maps a state into another state whenever g reports that there is no transition for the current state and current character
- An "output" function *output*, which maps states to sets of patterns matched at the specific state

Construction of these three functions is accomplished through two supporting algorithms. The first fully constructs g and partially computes *output*. The second fully constructs f and completes the computation of *output*. The first algorithm takes only the set of patterns (referred to there as K) as input, while the second algorithm takes the resulting g and *output* from the first as input.

For an example, let K be the list $\{ACTG,\ CTG,\ AAGT\}$. Constructing g by the algorithm given in the paper yields the DFA given by the figures 9 through 11.

Starting with figure 9, the goto function g provides the finite-state machine that lays out the transitions between states based on the character under consideration. Any state that does not have a transition for a given character is assumed to "fail" on that character, which is where the failure function f comes into use. While it is given in the algorithm that there are no failure transitions for state 0, this is made explicit in the figure by including a self-referencing transition for "G" and "T".

In figure 10, the failure function f shows how the processing of the DFA may jump around to implement a form of back-tracking. State 2 will move to state 5 on a failure transition,

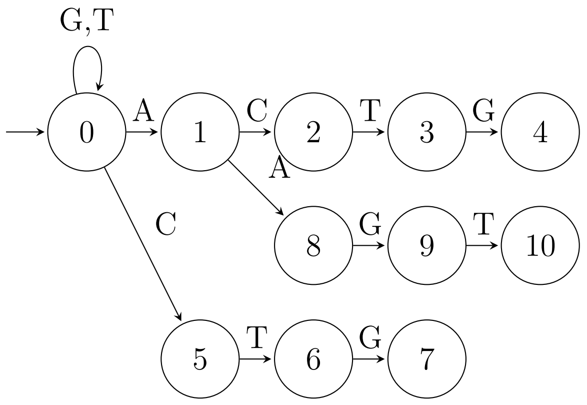


Figure 9: Aho-Corasick goto function

i	1	2	3	4	5	6	7	8	9	10
$f(i)$	0	5	6	7	0	0	0	1	0	0

Figure 10: Aho-Corasick failure function

reflecting that the character previous to the state ("C") could instead be the start of matching the "CTG" pattern. Because there are no failing transitions in state 0, that state is not represented in the figure.

i	0	1	2	3	4	5	6	7	8	9	10
$output(i)$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{0,1\}$	$\emptyset$	$\emptyset$	$\{1\}$	$\emptyset$	$\emptyset$	$\{2\}$

Figure 11: Aho-Corasick output function

In figure 11, the *output* function is represented as an indexed array of sets. Most states (including state 0) have the empty set as output. The non-empty sets represent the found patterns by their index in the original list. State 10 has only a value of 2, representing the pattern "AAGT". Of note is state 4, whose set includes both the index 0 and the index 1. This reflects the fact that the three characters of pattern 1 overlap the last three characters of pattern 0.

To illustrate the process of this algorithm, consider the target string that begins with the sequence "AACTG...". Figure 12 shows the progression of states as the pointer advances through the first five characters.

Starting at the 0 state, the first "A" moves the DFA to state 1 after which the second "A" moves it to state 8. From 8, the character "C" is a failure transition, so the value of $f(8)$ is read. The value is 1, so the DFA immediately moves to state 1 and looks for a transition on

	A		A		C		T		G		...
0		1		8		2		3		4	
				1							

Figure 12: Example of Aho-Corasick state progression

"C". It exists, and the DFA moves to states 2, 3 and 4 in turn. Upon reaching state 4 the value of *output*(4) is found to not be the empty set, so the machine signals a match of the patterns "ACTG" and "CTG".

Here, the Aho-Corasick algorithm has positively identified the presence of two of the three patterns encoded by the DFA. Only 5 characters from the target string were processed, and because the DFA indexes by the ordinal values of the characters no direct comparisons were made. Further, no back-tracking was required to find the second pattern.

3.5 Approximate Matching with Gaps

For the fifth algorithm in the suite of experiments, it was decided to implement an approximate-matching algorithm. In this case, the algorithm chosen is a new algorithm that takes a different approach to approximate-matching: building a DFA with additional states that allow for measured gaps between the characters of the pattern.

The idea of using a DFA to represent the pattern is inspired primarily by [1] and [5]. Becchi and Crowley, in particular, address the problem of representing the concept of *counting* states in an extended definition of a finite automaton. In their work, they put forward a means of having special states that track the number of times they've been entered in sequence, only allowing exit from the state when the given number (count) of visits is satisfied.

From this, a different type of approximate-matching can be pictured: one that allows for well-defined, finite "gaps" between elements of the pattern being matched. Given a value k that is a positive integer, gaps that are up to k characters in length can be tolerated while still regarding the pattern as matching a point in the target sequence. And when $k = 0$, the approach becomes the naive exact-matching approach that is of time-complexity $O(mn)$.

Current algorithms for approximate matching in DNA sequences use a variety of approaches, such as edit distance. But methods based on edit distance can result in all changes or differences being clumped together as opposed to distributed across the pattern in a more balanced fashion. For example, given the same "CTAGC" pattern used for the previous single-pattern algorithms and $k = 3$ this algorithm would allow as many as 12 additions to the pattern in order to match a position in the sequence. Simply allowing an edit distance of

12 in a different algorithm could potentially lead to 12 additions being made between just two of the pattern characters. The algorithm described here is designed to prevent such uneven distribution of the gap letters in the matched segment of the sequence.

The basis of the algorithm is the notion that such an approximate match as described here can be expressed in terms of a regular expression that supports the specification of ranges of matches for a given class or symbol in the expression. Such a specification is generally encoded as $\{a, b\}$ in an expression, where a and b are integers and $b \geq a$. Most compatible engines support several variant forms of this:

$\{a\}$: matches exactly a occurrences

$\{a,\}$: matches at least a occurrences, with no upper limit

$\{,b\}$: matches at most b occurrences, with as few as zero

$\{a, b\}$: matches at least a occurrences and at most b occurrences

Using the fourth construct and the "CTAGC" pattern, an example regular expression for a value of $k = 3$ could look like:

```
C.{0,3}T.{0,3}A.{0,3}G.{0,3}C
```

However, this not correct because the dot (`"."`) matches all characters. This is not desirable, as any "T" encountered after the initial "C" should move to that part of the pattern rather than possibly being subsumed by the dot. Additionally, in a general regular expression engine the dot will match *all* characters, including those not part of the DNA alphabet. Restricting the counting-states to just the alphabet of the problem, and eliminating the target letter from the gap consideration gives us:

```
C[ACG]{0,3}T[CGT]{0,3}A[ACT]{0,3}G[AGT]{0,3}C
```

This, given that k is known at the time of the construction, can be readily transformed into a DFA.

As an initial naive approach to the algorithm, the constructed DFA emulates the counting-states by creating k additional states for each of the first $m - 1$ characters of the pattern being searched for. It first creates states 0 and 1, representing the initial state and the the state reached by encountering the first character of the pattern (p_0). These two states constitute the beginning of the "trunk" of the DFA. Then the algorithm loops from 1 to $m - 1$, performing the following steps for each loop iteration:

1. A new state is created on the trunk ("`new_state`")
2. The previous head of the trunk is linked to the new state by a transition on p_i
3. The previous head is recorded in a temporary variable ("`last_state`")
4. A second loop runs from 1 to k that creates the branch states and their transitions to `new_state`
5. The head of the trunk becomes `new_state`
6. The value of `new_state` is advanced by k

At the end of the outer loop, the head of the trunk is recorded as the (sole) acceptance state for the DFA, and state 0 is recorded as the starting state.

This DFA (using the example pattern and $k = 3$) is illustrated in figure 13. In this diagram, the branches alternate between left and right of the trunk so as to allow the labels of the transitions to be more clearly readable. The numbering of the states reflects their order of creation, as detailed above and as will be outlined later in algorithm 1.

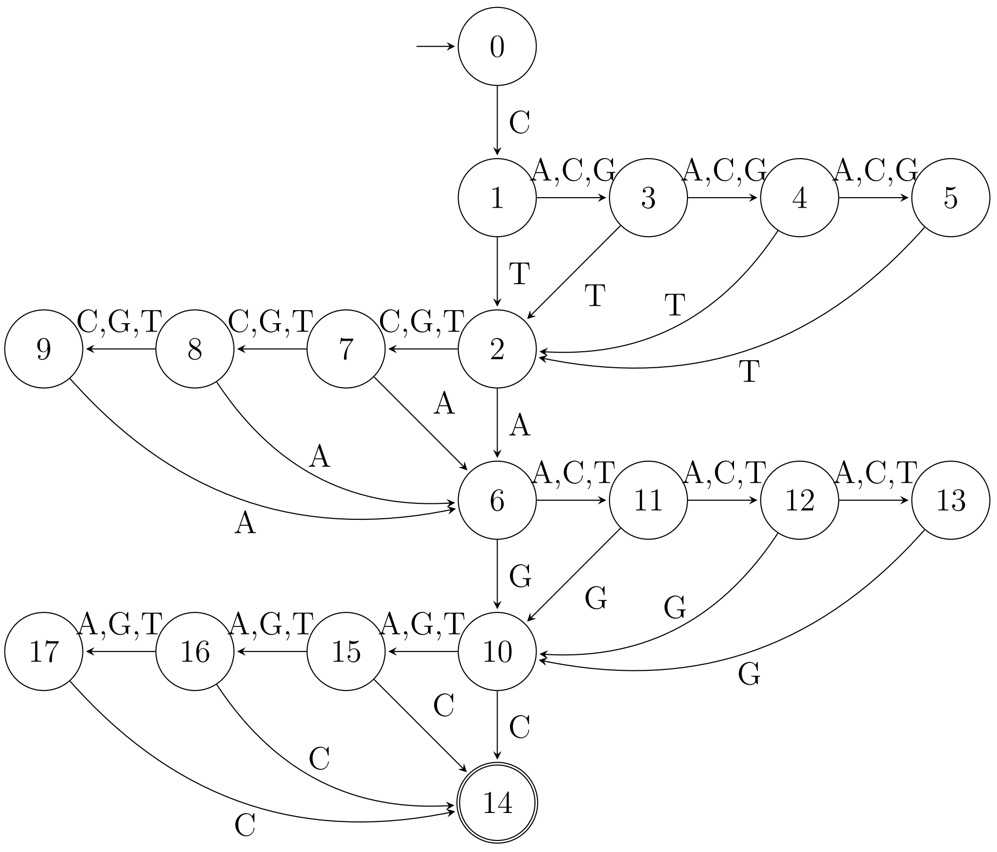


Figure 13: DFA-Gap finite automaton

In this DFA, the number of states needed (N) can be easily calculated as a function of m and k:

$$N = 1 + m + k(m-1)$$

Thus, for a given pattern of length m, the size of the DFA grows linearly with k. If the DFA is built using counting states (per [5]), then the value of k will have no influence on the number of states, which will then be $2m$.

3.5.1 Creation of the DFA

Dividing the DFA-Gap process into two parts, the first part is the creation of the automaton itself. This is outlined in algorithm 1.

Lines 1–6 initialize the elements. The body of the outer for-loop runs from line 8 to line 19, and is primarily concerned with extending the trunk of the DFA. The inner for-loop, lines 13–16, builds one branch connected to the state that was the head of the trunk prior to lines 8–10. After the outer loop completes, lines 21–23 assign A (the set of acceptance states) and q_0 (the starting state) and return a 3-element tuple of these values along with the transition function δ itself. This tuple represents what the next part of the algorithm requires in order to perform the matching process.

Lines 2, 4, 9, and 13 reference a process that was not explicitly defined: `CreateState()`. The implementation of this would be dependent on the approach to the algorithm as a whole. In these examples, it is assumed that this process creates one complete state representation with all transitions initialized to the failure value. This allows the default transition to be a failure unless explicitly updated by the algorithm.

3.5.2 Matching with the DFA

The second part of the process is to use the DFA in the matching process. This is outlined in algorithm 2.

Here, lines 1–3 again are simple initialization. This includes a call to the previous algorithm to obtain the DFA elements that are needed for the process. The outer loop is a standard for-loop that iterates over the range of characters in the target sequence S that can be candidates for the first character in a match. The process of traversing the DFA in search of a match is done in the while-loop. Here, there are only two statements: an advancement of the *state* value, and incrementing the *ch* value. When the δ function finds that the combination of *state* and s_{i+ch} results in a failure state, the while-loop exits.

Algorithm 1: CreateDFA

```
Input  : $P$, the pattern to match ($p_0$ ... $p_{m-1}$)
         $\Sigma$, the alphabet ({ $A, C, G, T$ })
         $k$, the maximum gap allowed between characters of $P$
Output : The finite automaton $M = (q_0, A, \delta)$, where $q_0$ is the starting state, $A$ is
         the set of accepting states ($A \neq \emptyset$), and $\delta$ is the transition function
1  $\delta \leftarrow$ *empty list*
2  CreateState($\delta$,0)
3  $\delta(0, p_0) \leftarrow 1$
4  CreateState($\delta$,1)
5  state $\leftarrow 1$
6  new_state $\leftarrow 1$
7  for $i \leftarrow 1$ to $m - 1$ do
8      new_state $\leftarrow$ new_state $+ 1$
9      CreateState($\delta$,new_state)
10     $\delta$(state, $p_i$) $\leftarrow$ new_state
11     last_state $\leftarrow$ state
12     for $j \leftarrow 1$ to $k$ do
13         CreateState($\delta$,new_state $+ j$)
14         $\delta$(new_state $+ j, p_i$) $\leftarrow$ new_state
15         $\delta$(last_state, $\Sigma - p_i$) $\leftarrow$ new_state $+ j$
16         last_state $\leftarrow$ new_state $+ j$
17     end
18     state $\leftarrow$ new_state
19     new_state $\leftarrow$ new_state $+ k$
20 end
21 $A \leftarrow$ state
22 $q_0 \leftarrow 0$
23 return $(q_0, A, \delta)$
```

When the while-loop exits, if the value of *state* is a member of the set A then a match has been found. Recall that A will have only one element, so this comparison can be reduced to a simple equality comparison. The designated list *matches* collects each found instance as a tuple of the index i within S and the complete matched string with gap characters.

3.5.3 Example

Using a smaller example pattern of "CGAG" and a value of $k = 1$, the DFA shown in figure 14 is built. For demonstration purposes, the sequence "CCGAAGC" will be used to search within.

Following algorithm 2, the following steps take place:

Algorithm 2: FindPatternWithGaps

```
Input  : $P$, the pattern to match ($p_0$ ... $p_{m-1}$)
         $S$, the sequence to search within ($s_0$ ... $s_{n-1}$)
         $\Sigma$, the alphabet ($\{ A, C, G, T \}$)
         $k$, the maximum gap allowed between characters of $P$
Output : A list of tuples $(i, S')$, where $i$ is the starting index of the match and $S'$ is
         the full matched substring
1  matches ← empty list
2  ($q_0$, A, $\delta$) ← CreateDFA($P$,$\Sigma$,$k$)
3  end ← length($S$) - length($P$)
4  for $i$ ← 0 to end do
5      state ← $q_0$
6      ch ← 0
7      while $\delta$(state, $s_{i+ch}$) $\neq$ failure do
8          state ← $\delta$(state, $s_{i+ch}$)
9          ch ← ch + 1
10     end
11     if state $\in A$ then
12         S' ← $S[i .. (i + ch - 1)]$
13         append(matches,($i$, $S'$))
14     end
15 end
16 return matches
```

1. Loop starts at $i = 0$
2. $state = 0$, $ch = 0$
3. $\delta(0, C)$ is read, transitions to $state = 1$, ch increments to 1
4. $\delta(1, C)$ is read, transitions to $state = 3$, ch increments to 2
5. $\delta(3, G)$ is read, failure detected
6. $state$ is not in A, so loop advances and $i = 1$
7. $state$ and ch each reset to 0
8. $\delta(0, C)$ is read, transitions to $state = 1$, ch increments to 1
9. $\delta(1, G)$ is read, transitions to $state = 2$, ch increments to 2
10. $\delta(2, A)$ is read, transitions to $state = 4$, ch increments to 3

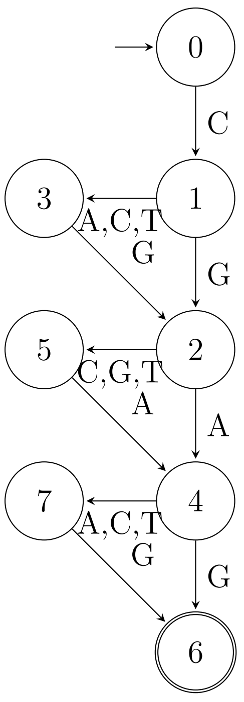


Figure 14: DFA for the pattern CGAG

11. $\delta(4, A)$ is read, transitions to $state = 7$, ch increments to 4
12. $\delta(7, G)$ is read, transitions to $state = 6$, ch increments to 5
13. $\delta(6, C)$ is read, failure detected
14. *state* is in A, so a tuple of (1, "CGAAG") is saved on *matches*

At this point the loop would resume with $i = 2$, but this sufficiently illustrates the process. A total of 9 state transitions were made, resulting in the location of a match 5 characters in size. At the reading of each state transition only one character look-up was required since the δ function initializes each new state to default all transitions to the failure value.

3.6 A Regular Expression Variant

Experimentation with the DFA-Gap algorithm led to some speculation as to whether the performance might be better if the search were performed by means of an actual regular expression in place of a specialized DFA. The general focus of the algorithm remained the same, the only difference being the use of a regular expression engine in place of the automatically-generated automaton.

The requirements of using a regular expression were summarized as:

1. The complexity of generating the regular expression should not exceed the complexity of generating the DFA
2. The regular expression must process the target sequence in a single pass through, as does the DFA-Gap algorithm
3. The regular expression must be capable of identifying both the starting position of each match and either the length or the content of the matching substring
4. The regular expression must find all matching substrings, including overlapping matches

The first requirement could be further expressed as constraining the time-complexity of the set-up to O(m), linear in the length of the pattern. The DFA creation stage of the original algorithm required k additional steps for $m-1$ instances of the main loop. Because k is a constant in this algorithm, the complexity of that part of the original algorithm was only bounded by O($2m$) rather than being polynomial. A complexity of O(m) would, therefore, hint at a simpler variation of the algorithm.

The second requirement might be mistaken for constraining the time-complexity of search to O(n), but this is not the case. As an approximate-matching algorithm, there would necessarily be some back-tracking through the target sequence when it has been determined that a candidate position is not the start of a matching substring. Instead, the goal was to try to keep the complexity close to O($(m+k)n$).

Requirement 3 dictated the need for the chosen regular expression engines to either capture and return matches, or to positively identify the ending-point as well as the starting-point of each match. This was simple to achieve, as match-capturing is a standard feature of regular expression engines.

The last requirement proved the most critical of the set. Because of the second requirement constraining the search process to a single pass, it would be necessary to construct an expression that could match parts of the target string without consuming them and removing them from future consideration. This is covered in greater detail in section 4.5.6, but this requirement ended up disqualifying two different engines (one for C/C++, the other for Rust) from usage during the implementation and testing of the algorithm.

4 Details of the Experiments

To gather the desired measurements the experiments were executed in the chosen languages (C, C++, Rust, Perl, Python), with the algorithm implementations being run under a "harness" application that measured various memory, performance, and energy metrics. Additional tools were used to examine the programs for memory leaks, as well as measure aspects of the source code itself.

4.1 Definitions and Measurements

To begin, some concepts will be introduced and terms defined.

4.1.1 SLOC (Source Lines Of Code)

The *Source Lines Of Code*, or SLOC, measurement attempts to evaluate the conciseness of the source code to a program. It generally distinguishes between physical lines of text, comments, and actual source lines.

As a metric of code quality or developer productivity, SLOC is not without some controversy. In [2] the authors point out that measuring lines of code can be very diverse in its execution, and often not clear in its purpose. Nguyen, et al [25] put forward the basis for an unambiguous standard guide to counting, and describe its use with the support of a configurable counting tool. Here, the purpose of measuring SLOC will be simple and restricted in scope: it will only be used as a comparison of the implementation of identical algorithms in different languages.

For the purpose of evaluating expressiveness, the SLOC measurements were limited to just the count of actual "source" lines as reported by the tool that was eventually chosen for this metric, `sloc`[3].

4.1.2 Language Conciseness Through Compression

In [6], Bergmans, et al describe a technique of measuring the conciseness of programming languages through a process of pre-processing and compressing the source code of a large number of multi-language projects of differing sizes. The higher the compression ratio of the files in a given language, the less concise it is considered to be. The authors hypothesized that this is because a higher compression ratio implies a greater degree of code-redundancy necessary to express the purposes of the program.

Given that none of the experiment source files are of significant length, this metric was applied at the language level, looking at all files for each language in per-language archive files. It was also modified to accomodate the smaller sample size, as will be explained further in a later section.

[3]This and other referenced software tools are summarized in table 25.

4.1.3 Cyclomatic Complexity

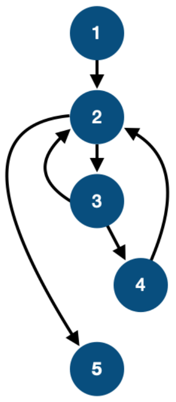
Thomas McCabe introduced the concept of *cyclomatic complexity* in 1976 [23]. In the most simple terms, it measures the number of paths through a program or function. Sometimes referred to as "McCabe Complexity", the value is based on measurement of control structures such as conditional statements, loops and similar means of changing the path of execution through the program or function being measured.

In the following combination of listing 1 and figure 15, a small Python function is accompanied by a directed graph illustrating the cyclomatic complexity of the function.

Listing 1: Python example of complexity

```
1  def count_bits(n):
2    bits = 0
3
4    while n != 0:
5      if n % 2 == 1:
6        bits += 1
7
8      n //= 2
9
10   return bits
```

Figure 15: Cyclomatic graph



The function itself is not very complex: it simply counts the number of "1" bits in the given integer number. The corresponding graph has 5 nodes and 6 edges. McCabe gives his formula for calculating the complexity from a graph representation as:

$$v \;=\; e - n + 2p$$

where v is the cyclomatic complexity, e and n represent the number of edges and nodes in the graph, and p is the number of connected components. In this example, $p = 1$ and thus $v = 3$.

McCabe gives examples of the control graphs for some typical constructs: a *sequence* control, an *if-then-else* control, a *while* control, and an *until* control. Each of the first three are demonstrated in this example: the edge from node 1 to node 2 is a *sequence* control, the edges between nodes 2, 3 and 5 represent a *while* control, and the edges from 3 to 4 and 2 represent an *if-then-else* control.

Thus, the graph nodes correspond to the following lines in the function:

1. The entry-point of the function at line 1

2. The `while`-loop at line 4
3. The `if`-conditional at line 5
4. The "true" branch of the conditional at line 6
5. The exit/return from the function at line 10

Measuring the complexity in a consistent way is important when comparing different languages. A tool called `lizard` was used for all languages except Perl (which was not supported by the tool) and Rust (which exhibited bugs when the data for the Aho-Corasick implementation was reviewed). To gather the metrics for the Perl code a second tool, called `countperl`, was used. For Rust, a tool called `rust-code-analysis-cli` from the Mozilla project was used. This tool provided greater depth and detail into the Rust code than `lizard` had, and did not exhibit the bugs noticed in the data for the Aho-Corasick source code.

It is not clear if the techniques for measuring Perl and Rust were completely identical to the techniques used by `lizard` for the other languages, so this is noted in the results in section 5.3.2.

4.1.4 RAPL (Running Average Power Limit)

The *Running Average Power Limit* (RAPL) measurement system was introduced by Intel into their CPU products starting with the Sandybridge family of processors [19]. The system allows measuring energy over several areas:

Package: The full (socketed) processor package, which may contain multiple cores.

Power-Plane 0: The domain that encompasses the combined cores within the package. This reading will cover all cores within the CPU of the package.

Power-Plane 1: Sometimes referred to as "uncore", this domain generally covers the integrated GPU (if present).

DRAM: The domain for the DRAM memory that the CPU is managing, whose energy usage is separate from the package.

Psys: The domain that covers the entire system-on-chip energy usage. This would include the package and DRAM values, as well as other system-level energy consumption.

A computer system may have more than one package, and the RAPL interface includes methods for determining the number of packages and gathering the energy readings for each package separately. However, it is not possible to measure a package's energy usage at the level of an individual core.

Reading the RAPL data is done through the *model-specific registers* (MSR) interface, as detailed in [16, Chapter 14]. The method involves reading several registers to determine the number of packages the system has, then determining the scaling factors for each of time units (expressed in seconds), power units (Watts), and energy units (Joules). The scaling factors are stored in a single register referred to as `MSR_RAPL_POWER_UNIT`, as groups of bits within the register. Each scaling factor is a 4-bit (5-bit in the case of energy units) value used to compute a fractional floating-point number (where b represents the value of the n-bit factor):

$$S = \frac{1}{2^b}$$

In the case of the energy units factor, the value of b on the test platform was 01110b (14), and $S = 61.04\ \mu\text{J}$.

Obtaining the data during run-time from RAPL required reading from of a series of read-only registers within the MSR and scaling the values obtained by the appropriate value of S for the units. For example, reading the Power-Plane 0 (CPU) energy value uses the `MSR_PP0_ENERGY_STATUS` register. The value obtained from reading this register is 64 bits in width, though only the initial 32 bits hold energy data (the high 32 bits are reserved by Intel). The value read is masked to remove any high bits, then multiplied by the energy units scale factor to produce a value in Joules.

An issue with the RAPL system was encountered during runs of the experiments: because the value of the energy registers is 32 bits in size, it wraps around to 0 when the maximum value is reached. This caused occasional anomalous readings in cases where the register would reset between the initial reading and the final reading, resulting in a negative overall value. The program that processed the output from the experiments was adjusted to recognize such values and adjust them by applying a constant value computed as $C = S \times 2^{32}$, where S is the scaling factor for that value's type (Package, DRAM, etc.).

Adding the appropriate constant for the type of value that was anomalous dealt with the issue and ensured that all iterations of each algorithm and each language would be usable for the analysis of the results.

4.1.5 Summary of Metrics

For each execution of a program comprising an experiment, the following data was gathered:

Total Program Run-time: The complete run-time of the program, as measured by the harness program. Unlike the next metric, this would include program initialization time, the input/output operations of loading the data to be processed, etc. This is measured in floating-point seconds with micro-second resolution.

Algorithm Run-Time: The time spent specifically within running the algorithm itself over the complete set of test data. This is measured solely on the processing of data, and does not include I/O, set-up of the environment, or post-algorithm steps such as freeing of memory. Also measured in micro-second resolution.

Maximum Memory Usage: The largest amount of memory allocated for the running program throughout the course of its execution, in megabytes. This represents the largest size to which the program grew during the run.

Power-Plane 0 (CPU) Energy Usage: The energy consumed by the CPU cores during the execution of the program. Measured over the full lifespan of the program, not just the algorithm itself. Measured in Joules.

DRAM Energy Usage: The energy consumed by the DRAM during the full lifespan of the program. Measured in Joules.

Full (Package) Energy Usage: The energy consumed by the full (socketed) package, which includes the CPU cores' energy but not the DRAM energy. Also measured in Joules.

The Power-Plane 1 (GPU) RAPL values were excluded because the testing machine's package did not have an integrated GPU. Additionally, the Psys values were excluded because they were deemed unnecessary in the context of having the package, CPU and DRAM values.

Independent of the per-program metrics, additional measurements of code expressiveness were made on each of the source files:

Source Lines Of Code: The measured lines of code in the implementation of the program, using a tool (`sloc`) designed to measure these values using consistent standards across the different programming languages.

Cyclomatic Complexity: The measured complexity of the code, including both the average complexity for the file and specific complexities for those functions that directly correspond to each other across the different implementations.

Conciseness Through Compression: Using the same tools as were used in the research in [6] (`cloc` for removing code comments and `xz` for data compression), archives of each language's code were compared to each other.

These processes are explained in more detail in section 5.3.3, along with the results.

4.2 Experiment Harness

The harness that was developed for managing the execution of the experiment programs was based on the code developed by Pereira, et al [26] and made available via their GitHub repository[4]. Their code was adapted and heavily modified to allow for some command-line options controlling features such as the number of iterations each experiment would be run, verbosity of output, etc. It was enhanced to record maximum memory usage and to work with the specific computer selected to be the testing platform.

4.3 Languages

The languages used were chosen for their commonalities as well as their differences:

- Three of the languages (C, C++, Rust) are compiled to machine code and were chosen for performance first and foremost. The remaining two (Perl and Python) are interpreted ("scripting") languages which are highly regarded for speed of development and rapid prototyping, as well as being popular in bioinformatics computing.
- Each language is currently in widespread use across different disciplines of software development.
- The languages showcase differing aspects the approach to memory management, as are detailed below.

Each language section includes a sample of the language, implementing a naive ($O(mn)$) matching algorithm.

4.3.1 C

The C programming language is the oldest and most-established of the chosen languages. Originally designed in the early 1970's by Dennis Ritchie, it remains a very widely-used and influential language since its first appearance in 1972. Since 1989, it has been standardized by both ANSI (the American National Standards Institute) and by the International Organization for Standardization (ISO).

C relies on what is referred to as *manual memory management*, meaning that the programmer is responsible for all allocation and freeing of dynamic memory. This approach

[4] `https://github.com/greensoftwarelab/Energy-Languages`

can often lead to several major classes of bugs when used incorrectly, such as memory safety issues or memory leaks. Multiple pointers to the same region of memory can become “dangling pointers” when one pointer frees the memory without the other pointers being invalidated at the same time. Further attempts to use any of the other pointers can lead to memory corruption or segmentation faults.

Memory management in C is done through a collection of functions in the C standard library, including `malloc`, `calloc`, `realloc`, `reallocarray`, and `free`. While `free` returns allocated memory to the heap, the other functions either allocate memory or change the size of an existing block of allocated memory.

Experiments using the C language were run on three different compiler toolchains: the GNU Compiler Collection (GCC), the LLVM Compiler Infrastructure (LLVM), and the Intel® oneAPI Toolkit. This was done to show the subtle differences between programs generated by these compilers, which are free and commonly-used. Due to the low-level nature of C and the maturity of the compilers, C generally performed well at each algorithm compared to other languages. Listing 2 shows a C implementation of the naive algorithm.

Listing 2: C naive implementation

```c
int match(const char *pattern, const char *string, int **matches) {
  int *saved_matches = NULL;
  int found_matches = 0;
  int m = strlen(pattern);
  int n = strlen(string);

  for (int i = 0; i <= n - m; i++) {
    if (!strncmp((const char *)(string + i), pattern, m)) {
      found_matches++;
      saved_matches = realloc(saved_matches,
                              found_matches * sizeof(int));
      saved_matches[found_matches - 1] = i;
    }
  }

  **matches = saved_matches;
  return found_matches;
}
```

4.3.2 C++

C++ was developed initially as an extension of C, by Bjarne Stroustrup while working at AT&T Bell Labs. It first appeared in 1985 and was initially standardized in 1998. At first envisioned as "C with Classes", the language has been significantly expanded over the years to include many more features while still maintaining low-level memory accessibility. C++ attempts to offer more expressive, concise coding than C, with many of C's memory-management concerns dealt with automatically by class constructors and destructors.

In C++, Stroustrup originated the programming idiom of *resource acquisition is initialization* [31] (RAII). Most dynamic memory is managed via class constructors and destructors, though C++ also supports the `malloc`-based memory management mechanisms inherited from C.

Experiments using the C++ language were also run on the same three compiler toolchains as C: GCC, LLVM, and Intel. Listing 3 shows the naive algorithm implemented in C++.

Listing 3: C++ naive implementation

```
std::vector<int> match(std::string pattern, std::string string) {
  std::vector<int> matches;
  int m = pattern.length();
  int n = string.length();

  for (int i = 0; i <= n - m; i++) {
    if (pattern == string.substr(i, m))
      matches.push_back(i);
  }

  return matches;
}
```

4.3.3 Perl

Perl is an interpreted language developed by Larry Wall while working as a programmer at Unisys. The first version, 1.0, was released on December 18, 1987. The current version as of this writing is 5.36.0, released on May 28, 2022.

Perl's reach grew tremendously with the introduction of the World Wide Web's Common Gateway Interface due to its native support for regular expressions and strong text-processing capabilities. Initially developed as a general-purpose scripting language, Perl borrowed features from languages such as C, Awk, Sed, and the `sh` shell. Perl also offers features generally associated with functional programming, including first-class and higher order functions, lexical closures, garbage collection, and list comprehensions.

Perl is dynamically-typed and multi-paradigm in nature. It supports procedural as well as object-oriented programming styles as well as metaprogramming. Its garbage collection approach to memory management is based on reference counting.

A Perl implementation of the naive algorithm is given in listing 4.

Listing 4: Perl naive implementation

```
sub match {
    my ($pattern, $string) = @_;

    my $m = length $pattern;
    my $n = length $string;
    my @matches = ();

    for my $i (0 .. ($n - $m)) {
        if ($pattern = substr $string, $i, $m) {
            push @matches, $i;
        }
    }

    return \@matches;
}
```

4.3.4 Python

Python is another interpreted language, developed by Guido van Rossum while at Centrum Wiskunde & Informatica in the late 1980's and first released as version 0.9.0 in 1991. Like Perl, it is also dynamically-typed and multi-paradigm in its nature. As a language, Python consistently ranks high in user popularity on such measures as the TIOBE Programming Community Index[5].

Python's memory management is a combination of reference counting and a cycle-detecting garbage collector. A Python implementation of the naive algorithm is given in listing 5.

Listing 5: Python naive implementation

[5]TIOBE Index: `https://www.tiobe.com/tiobe-index/`

```python
def match(pattern, string):
    m = len(pattern)
    n = len(string)
    matches = []

    for i in range(n - m + 1):
        if pattern == string[i:(i + m)]:
            matches.append(i)

    return matches
```

4.3.5 Rust

Rust is the newest of the languages, having first appeared in 2010. Rust offers a promise of expressiveness with greater safety in the areas of memory management and ownership. It is a multi-paradigm, general-purpose language that draws from several previous languages including C++, Haskell, and Standard ML. While often referred to as a systems programming language, its usage is spreading rapidly to other areas including to some scientific programming disciplines [27]. The language began in 2006 as a personal project of Graydon Hoare, an employee of the Mozilla Corporation, with Mozilla beginning to sponsor the work in 2009 and officially announcing the project in 2010 [3]. The first pre-alpha numbered version of the compiler was Rust 0.1, which was released in January of 2012. The current (as of this writing) version of Rust is 1.65.0 and was released in November of 2022.

An area where Rust is distinct from other C-based languages is in the way it manages memory and tracks values on the stack and heap. Rust uses an ownership system [9, Chapter 4], with the ability to specify lifetime information for reference types. There is no automated garbage collection, and resources are managed through the same convention of *resource acquisition is initialization* as in C++, with optional reference counting. Rust's design for memory safety does not permit null pointers, dangling pointers, or data races.

With languages such as C and C++, *data ownership* is handled largely through practice and convention. An instance of a C++ `std::string` owns the buffer allocated for the storage of the string data. Other variables may be created, though, that point to the same buffer or a single character within it. These other interests in the content of the string buffer have their own responsibility for noticing when the original string object is destroyed and the buffer freed. After such point, the outside interests are each responsible for marking their references as no longer valid.

In contrast, Rust integrates the concept of ownership directly into the language itself. Compile-time checks enforce ownership and report violations. When the owner of a value is “dropped” (Rust terminology for freeing) the owned value is dropped as well. While the variables themselves are on the stack, the content is allocated on the heap. Variables own their values, and the complex datatypes (structs, tuples, arrays, and vectors) own their elements.

Listing 6 shows the naive algorithm as implemented in Rust. Note that the subroutine is named slightly differently in this case, as `match` is a keyword in Rust.

Listing 6: Rust naive implementation

```rust
fn match_string(pattern: &str, string: &str) -> Vec<usize> {
    let mut matches: Vec<int> = Vec::new();
    let m = pattern.len();
    let n = string.len();

    for i in 0..=(n - m) {
        if pattern == string[i..(i + m)] {
            matches.push(i);
        }
    }

    matches
}
```

4.4 General Implementation of the Algorithms

Each of the experiment programs providing an algorithm was implemented according to a consistent structure, to better facilitate the direct comparison of the source code across languages. This structure consists of three basic elements:

- An “input” module that encapsulates the loading of sequence, pattern, and answer data
- A “runner” module that provides the controlling loop of the program
- An “algorithm” module that provides the code specific to the algorithm being used as a basis for the experiment

The input and runner modules were written once per language and re-used across all algorithms.

4.4.1 Input modules

The input modules allow the main-loop modules (described next) to further abstract the reading of the external data used in each experiment. Data is separated into three files: the *sequences* file contains lines of randomly-generated target strings, the *patterns* file contains the crafted patterns to search for within the sequences, and the *answers* file provides a representation of the correct number of times each pattern should be found in each sequence. This allows the runner modules to verify the results of each invocation of the algorithms being evaluated. The nature of the data and its creation is further detailed in 4.8.

Each input module defines three routines, one for each of the data files. In most cases, the reading of the pattern files was essentially identical to the sequence files and thus the pattern routine simply calls the sequence routine.

The input modules are the first place in which the distinction in expressiveness and style between the languages becomes apparent. Differences become immediately visible in just the comparison of the C and C++ implementations, where the physical combined length of files (in C and C++, the input modules also required accompanying header files) differs by over 40% in favor of C++. Python measures as being just over 20% of the size of the C code.

4.4.2 Runner modules

Each runner module utilizes the input module to read in the experiment data and loop over it. In the single-pattern algorithms (which includes the approximate-matching algorithm), this is a nesting of two loops: the outer loop iterates over the set of pattern strings and the inner loop iterates over the set of sequence strings. Each iteration of the loop over the sequence strings triggers one execution of the algorithm being evaluated. In the multi-pattern algorithms, this is a single loop over the set of sequence strings, as the complete set of patterns are pre-processed prior to the loop.

The runner records the time according to the system wall-clock when the algorithm pre-processing begins, and the time when all loops and answer validation has completed. Everything that is not input-related or related to reporting of results is recorded in this span of time. At this point, the runner prints three lines to the standard-output stream. The lines identify the language (including compiler variants for C and C++), the algorithm, and report the time spent in the main loop. The runner is also responsible for handling the arguments passed to the program as well as determining the exit-code of the program (to allow the harness program to discern failing runs from successful runs).

4.4.3 Algorithm modules

The algorithm modules are at the heart of the experiments. To maintain consistency, each algorithm module defines a minimum of two functions: an initialization routine and the primary algorithm entry point.

The initialization routine is responsible for any pre-processing necessary for the pattern string, and produces a collection of data elements that represent the pattern in the appropriate internal structure. The exact nature and structure of this representation is language-dependent as well as algorithm-dependent.

The algorithm entry point routine is the means by which each algorithm was applied to the pattern and sequence under consideration. It receives the pattern representation produced by the initialization routine and the sequence representation as parameters, and returns a numerical value indicating how many times the pattern was successfully found within the sequence. In the case of the multi-pattern algorithm implementation, the return value from this routine is a vector of numbers with length equal to the number of patterns.

In addition to these two functions, each algorithm module defines all needed support code for the initialization and entry point. In some cases (such as the C and Rust implementations of the Aho-Corasick algorithm) this included minimal implementations of data structures such as sets and simple queues.

Each algorithm module also provides the language-specific equivalent of a “main” function, that function which is treated as the program entry-point by the operating system. Each “main” function consists of a single call to the runner function provided by the runner module. The call passes the two algorithm-specific functions as pointers (again, in a language-dependent manner) to the runner, followed by the name of the algorithm and a representation of the command-line arguments.

4.5 Algorithm Implementation Details

The different algorithms that were chosen are listed here in the order of their implementation. Each of the algorithms was implemented first in C, as a baseline. These C implementations were then used as templates for the other languages’ implementations. In addition to helping to keep the implementations approximately similar and equal, this provided insight into the differences that the features and expressiveness of the different languages made in the development process itself.

Any inefficiencies observed in the experiments’ implementations are due to this decision.

4.5.1 Knuth, Morris, and Pratt

The C implementation of Knuth-Morris-Pratt was adapted from [12, Chapter 7]. Of note is the discovery that the sample code quietly takes advantage of C's use of a "null" byte at the end of a string to stand in for the sentinel character that the algorithm appends to the pattern string. While this optimization was also applicable to C++ (where strings are represented by instances of the `std::string` class), each of the three remaining languages were required to manually add a sentinel character to each pattern string prior to computing the corresponding "next" table.

This being the first of the algorithms implemented, it is also where the means of passing generic data from the initialization routines to the algorithm routines was developed. The differences in these methods, particularly between the compiled languages, spoke strongly to their relative expressiveness.

In the C implementation it was necessary to use a memory pointer of the type, `void **`. This defines a dynamic list of dynamic pointers, but provides no information about each individual pointer. It is left to the code to properly type-cast the values from this block of memory. If the program is incorrect (such as getting the order of elements wrong) the resulting typed pointers will likely trigger memory faults when dereferenced. This was also the most-flexible of the compiled solutions and required no difference in implementation between the different algorithms.

In C++ it was initially attempted to reuse the `void **` approach that had been used in C. This proved to be extremely difficult under the stricter compiler rules, and the decision was made to use the `std::variant` class instead, which represents a type-safe union in the C++ language. While the same variant type definition could have been used for both the single-pattern and multi-pattern algorithms, it was decided to distinguish them for the sake of readability. The type definitions were made in the file `run.hpp`, the header-file for the runner module of C++. Unpacking a vector of the variant types brought further type-safety in the form of a generically-typed function `get<>` in the standard library that was used to extract the values to new objects of the correct type.

Rust had the language features that seemed the most clear and expressive in implementation: enumeration and pattern-matching. In Rust, an enumeration type (`enum`) can optionally assign data types to the elements of the enumeration. When an enumeration element is instantiated it is provided an instance of that data when applicable. This lead to a system for the Rust programs in which the initialization function for an algorithm created and returned a vector of the enumeration type. The algorithm routines that received these types would extract them from the vector using a pattern-matching[6] construct that ensured the types of each individual element of the vector received from the initialization function. The program would raise a run-time exception if the given vector element was not of the expected type. In contrast, Rust required the least-expressive approach to appending the sentinel value onto the pattern string for the sake of creating the "next" table.

Given the dynamically-typed nature of data in Perl and Python, both of these implementations were simple lists of values returned without any extra effort at encoding. As with the `void **` approach the onus was on the program to unpack the elements in the correct order. Trying to use a list or list-reference value as an ordinary scalar value would have caused a run-time exception in either language.

4.5.2 Boyer and Moore

The C version of the basic algorithm was drawn from [12, Chapter 14]. Like Knuth-Morris-Pratt, it requires a character at the end of the pattern for calculating the suffixes table. The C and C++ implementations used the "null" string-terminating byte for this while the others added it explicitly.

Starting the analysis with Rust in this case, one aspect became very clear in this algorithm's implementation: the need to regularly cast various numerical (particularly integer) types to other types. Because of the larger amount of array-indexing in this algorithm, the number of times an `i32` (signed 32-bit integer) or `u32` (unsigned) had to be cast to or from a `usize` type (an unsigned type used for lengths and indexing) was significant. While this does lead to fewer bugs in the code, at the same time it detracted from the conciseness and readability of the Rust version.

Perl and Python were comparable in their conciseness and logical expression of this algorithm. Python showed an advantage in the form of its native support for iterators as a type and functions such as `range` that return these iterators. Operations like looping backwards through a series of integers was clearer in the Python code than in the corresponding Perl.

The C and C++ implementations are best compared to each other, as this is another case where the C++ Standard Library classes lead to clearer code. Though there was less difference in code-length, various mechanisms of C++ in areas such as argument-passing meant less overhead and less ambiguity about pointers sent and received.

[6]In the computer language sense, not to be confused with pattern-matching in strings.

4.5.3 Bitap

The C version of Bitap was taken from [12, Chapter 5], where it is referred to as Shift-Or. The design of the algorithm given there is clearly drawn from the Shift-Or algorithm illustrated in [4]. The C++ version followed the C very closely with the only changes being the use of the C++ standard classes for strings and vectors. Likewise, the Rust version of this algorithm proved very simple and bore a reasonable resemblance to the C and C++.

Listing 7: Bitap main loop in Rust

```rust
for j in 0..n {
    state = (state << 1) | s_positions[sequence[j] as usize];
    if state < *lim {
        matches += 1;
    }
}
```

The Perl and Python implementations were also very similar to the compiled versions, with the same exception that each language has arrays as first-class data types. The Python version, however, did have one notable difference: due to Python's integers being of arbitrary size, it was not enough to take the bit-complement of 0 to get a "full" bit-field. In Python this resulted in a value of -1, rather than the expected $2^W - 1$ (where W is the word-size used within the algorithm). To force Python to treat these values as unsigned 64-bit, it was necessary to declare a "mask" value equal to $2^W - 1$ and apply this to the result of every logical bit-operation performed. This incurred a significant performance penalty on the Python implementation, giving it the widest performance gap when compared against its Perl counterpart.

4.5.4 Aho and Corasick

For the implementation of this algorithm, no existing code was consulted. Rather, the initial C version was developed directly from the algorithm specification in the paper itself. The C version required the most supporting code, as the algorithm calls for both queue and set data structures. Additionally, the construction of the g and *output* functions required dynamic re-sizing of lists. It was chosen to avoid this added complexity by generously estimating the size needed for those lists and pre-allocating that amount. Constructing f did not have this problem as the size was known at that point from having created the previous arrays. Both queue and set data structures were implemented with just the minimally-required functionality for each. For the queue, this was the operations *create*, *delete*, *expand* (grow), *enqueue*, *dequeue*, and a test for whether the queue is empty. For the set, these were *insert*, *test-for-membership*, and an implementation of a *union* operation. The set's structure was designed in a such a way that using the C `calloc` function was sufficient for both creation and initialization, and `free` was sufficient for deletion and clean-up. That said, these implementations did add to the number of dynamic memory allocations and corresponding releases.

When implementing in C++, the C++ Standard Library was able to provide existing classes for both set and queue implementations, greatly reducing the amount of supporting code in this version. Though C++ vector types can dynamically grow, intermediate experiments found that relying on this for the *g* and *output* function construction introduced a slight performance penalty. As such, it was decided to replicate the C approach of estimating the size of *g* and *output* and pre-allocating. Using C++ features, this also allowed for immediate initialization of the vectors where the C arrays required explicit loops to initialize. In the end, the difference between the two languages' implementations was a factor of almost 2x in SLOC for the C code over the C++ code.

The Perl and Python implementations were both significantly shorter in length than the C or C++, due largely to native support for resizable lists. For the queue data structure, the Python code used the `collections.deque` class that is part of the Python core. Perl used an ordinary list, as the language provides a built-in keyword for removing from the head of a list in O(1) time. Both languages grew the *g* and *output* functions dynamically rather than pre-allocating them. For the implementation of a set data structure, Python provides a `set` type as a native data type. The Perl version initially used the native associative array type, a common technique for emulating sets in Perl. This proved to be a slight performance penalty, so it was later replaced with a simple list of integers that had duplicates removed as needed.

Lastly, the Rust implementation went through several iterations before reaching its final form. The first version closely followed the C++ version: the use of resizable vectors through the native `Vec<>` type, and implementations of set and queue from the standard Rust library (`std::collections::HashSet` and `std::collections::VecDeque`, respectively). This initial version severely under-performed in comparison to the other two compiled languages. With help from the Rust community, it was determined that the two collection classes were incurring large amounts of overhead. To address this, they were replaced with simple implementations adapted from the C code. As with the Perl and Python versions, the `Vec<>` type was allowed to handle the dynamic nature of the *g* and *output* functions. As with C++, some intermediate experiments were performed that pre-allocated these vectors based on estimates of needed space. But unlike with C++, it was found that at best the performance remained the same, and in some runs was noticeably worse. As a result, the pre-allocation was removed.

4.5.5 Approximate Matching by DFA with Gaps

As this algorithm was new material, it was difficult to anticipate what concerns and issues would arise. Some aspects of each implementation were close in design to the Aho-Corasick code due to the fact that both algorithms are centered around a DFA. All implementations were based on the developed algorithms 1 and 2, from section 3.5.

Listing 8: C/C++ DFA-Gap main loop

```
for (int i = 0; i <= end; i++) {
  int state = 0;
  int ch = 0;
  while ((i + ch) < n && dfa[state][sequence[i + ch]] != FAIL)
    state = dfa[state][sequence[i + ch++]];

  if (state == terminal)
    matches++;
}
```

As was the case with some of the previous algorithms, the main differences between the C and C++ implementations were in the use of C++ standard library classes in place of manual memory management; their main loops were identical (listing 8). The availability of a postfix-increment operator in these languages allowed the two statements of lines 8–9 in algorithm 2 to be combined into a single statement. Unlike the Aho-Corasick DFA allocation, this algorithm has a predictable number of states. This made allocation of the DFA much more exact and clean in comparison.

The Rust version of this algorithm is largely identical in structure to the C and C++ versions. In terms of SLOC, it comes in longer than either of those, due primarily to the number of lines used for pattern-matching in the processing of the pattern data structure created by the initialization routine.

The Perl implementation suffered from some readability issues attributable to the ways in which Perl handles slicing arrays that have been passed by reference rather than by value. Perl also features a postfix-increment operator, allowing a savings of one line in the SLOC score. In contrast, the Python implementation once again gained conciseness and readability through its clear syntax around iterators and slightly cleaner syntax around treating strings as arrays and applying slicing operations to them.

4.5.6 Regular Expressions Variant

The performance of the Perl and Python languages on the DFA-Gap algorithm led to some critical thinking about how this algorithm might actually be implemented in those languages in a real application. This, in turn, led to an extra experiment being written for each of these two languages.

Both Perl and Python have native support for extended regular expressions as a part of the language. When this is taken into account, it is found to be preferable to use the implementation of the DFA as a regular expression. The regular expression engines in both Perl and Python are implemented in compiled code and are thus much faster than the hand-coded DFA implementations.

As will be shown in section 5.2.1, this had a very significant effect on the overall performance of the gap algorithm for those two languages.

The key to meeting the four requirements in section 3.6, particularly the fourth requirement, was a mechanism called *positive-lookahead*. This mechanism allows the engine to match the pattern as a sequence ahead of the current point, without actually consuming any of the matched characters. This was instrumental in finding overlapping matches.

Once this had been undertaken in Python and Perl, it was necessary to implement similar experiments for the compiled languages. In the case of C++, a class for regular expressions had been part of the C++ Standard Library since the adoption of the C++11 standard. C, as well, offered regular expression support in the form of the POSIX regular expressions functions included as part of the C standard library. Rust, though, did not have built-in regular expression support.

It was decided to use the `regex` "crate"[7] that was available for Rust, while using the native regular expression support in C and C++. This exposed problems: The C and C++ engines proved to be exceedingly slow, and unable to properly identify the matched substrings. Additionally, Rust's `regex` package did not support the positive-lookahead feature of regular expressions that was necessary to prevent making multiple passes over the target string.

The solution came in the form of the Perl-Compatible Regular Expressions engine. As the name hints, the engine provides support for the various extensions to regular expression syntax introduced by Perl (as well as other languages and dialects). In particular, PCRE supports both positive-lookahead and capturing within the lookahead. Lastly, while written for the C language it was possible to use this engine in both C++ and Rust through wrapper libraries.

As the results will show, the effectiveness of regular expressions varied across the languages.

[7]A "crate" is the term used in the Rust community for a packaged library or extension.

4.6 Initial Observations on Complexity

At this point, some initial observations could be made about the complexity measurements of some of the code. In the code examples given throughout section 4.3, all implementations of the naive algorithm had the same cyclomatic complexity value of 3. All also produced the same directed flow graph, shown in figure 16. In this graph, nodes 1 and 6 represent the entry and exit to the function. Node 2 represents the for-loop, and node 3 represents the if-else conditional block.

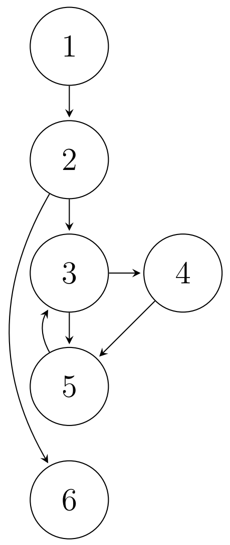


Figure 16: Flow graph of the naive algorithm

Part of the reason for this is that the complexity measurement does not take into account calls to functions. This is by intent, as factoring out content to a separate function is part of the process of managing complexity. In the cases of C and C++, both utilized library functions: `strncmp` in C and `substr` in C++. In contrast, the use of `substr` in Perl is a keyword of the language. Python and Rust skip this entirely by virtue of having array-slicing be a core language feature, along with the ability to compare slices directly. Had the complexity-measuring tools taken the library calls into account, that would have changed the graph for those languages from one connected component to two, and resulted in a higher overall complexity score.

Compare this to the `make_next_table` functions defined in the implementations of Knuth-Morris-Pratt. The complexity tool found the Python version to have a score of 6 while the C and C++ versions both scored 5. The reason for the distinction is that C and C++ allocated the array variable *next_table* prior to the function call and passed it as a parameter. By contrast, in Python *next_table* was created within the function. The loop used to create it is the source of the extra point of complexity. Figures 17 and 18 illustrate these slightly-different flow graphs. In figure 17, states 2 and 3 represent the extra for-loop that introduces the extra point of complexity.

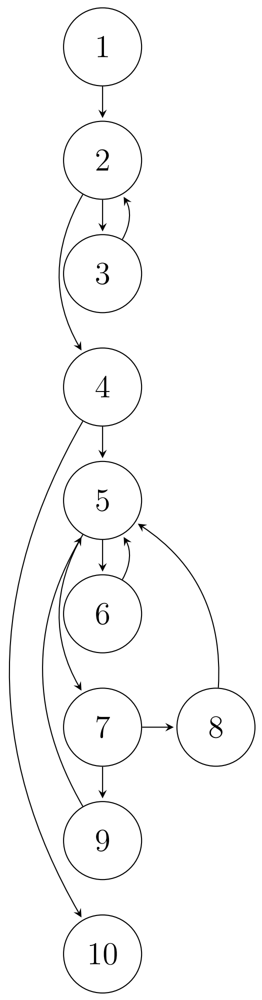


Figure 17: Flow of the Python `make_next_table` function

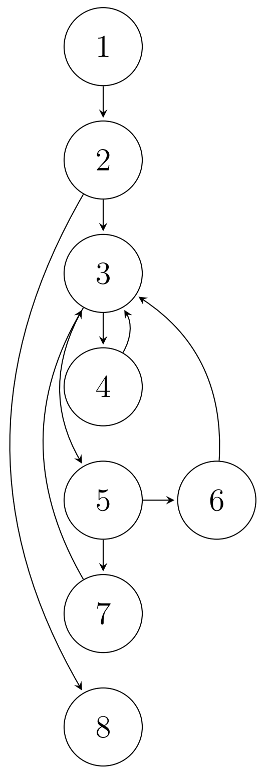


Figure 18: Flow of the C++ `make_next_table` function

The resulting complexity of the various implementations is explored in more depth in section 5.3.2.

4.7 Optimizations

Over the course of the development of the experiments in each language, the need became clear for some basic optimizations. In every case, any optimization was applied consistently across all languages. Some examples of optimizations include:

Data Preprocessing: While the C and C++ languages were able to seamlessly use individual characters from the strings as array indices, Rust and the interpreted languages were not. Based on an assumption that production-targeted code would apply any similar preprocessing, the strings were converted to forms directly usable by the other languages. In the case of Rust, this was a conversion of strings to arrays of unsigned 8-bit integers. In the case of Python, it was a direct mapping of strings (which are already treated as sequences by Python) into the ordinal values of each character. Perl required the most preprocessing, with strings first being converted to arrays of individual characters before being mapped to their ordinal values.

Pattern Preprocessing: To bring down the running times of the interpreted languages' experiments, a mechanism for preprocessing patterns was developed that allowed for each pattern to be processed only once prior to being applied to all sequences. Without this, some instances of the Knuth-Morris-Pratt algorithm (for example) took close to an hour to complete. Once the structure of this was established, it was also applied to the compiled languages as well.

Minimizing Type-Casting: In the case of Rust's strong typing, it was necessary to frequently cast integer values into Rust's `usize` type for use as array indices. Though this would have had little or no effect on run-time, the decision was made in some cases to declare a cast version of the integer value to help in the overall readability of the code.

Compiler Options: For the three compiled languages, consistent choices of compiler options for optimization were applied. For the C and C++ code, the same options were used for both languages. For the Rust code, the `cargo` utility was run with an option that instructed the compiler to build optimized code instead of debug-instrumented code. These were not applicable to the Perl and Python code, as both of those are interpreted languages.

All code-optimization steps were designed to be done inside the timing window of the runner module, and contributed to the reported algorithm run-time.

4.8 Experimental Data

The data used for the experiments was generated in a random fashion. It was specifically created to emulate basic DNA data, while keeping the size of the data manageable from the perspective of the programs that would be run.

4.8.1 Method of Generation

Data generation was handled by a Python script written to allow almost all parameters of the resulting data to be tuned. The script's options included a seed value for the random number generator, which allowed for consistent generation of the data once the parameters were tuned to satisfaction.

Sequence generation was a straightforward process of determining a length from the provided base length and variance values. The standard random number generator provided by Python was then used to draw the sequence of n letters from the set of valid characters.

To ensure that algorithms would be as thoroughly exercised as possible, each candidate pattern was tested before acceptance. The requirement for acceptance was that it match a minimum threshold (0.10%) of the full set of sequences. Generation of the patterns was done by first selecting the pattern length from the specified range, then extracting a sequence of that length from a random location in a candidate sequence. Candidate sequences were chosen by selecting every i^{th} sequence from the full list (starting with the first sequence), with i being the total number of sequences divided by the total number of patterns being generated.

For the approximate-matching algorithm, the same set of patterns was used since they were already known to meet the minimum-matching criteria. For this algorithm, a k value of 0 would lead to the same set of matches as an exact-matching algorithm would find.

4.8.2 Shape of the Data Used

The data set selected for these experiments consisted of the following:

- 100,000 sequence strings of length ranging from 1,008 to 1,040 characters ($1,024 \pm 16$). The number of patterns and their length was chosen to provide sufficient data to extensively exercise each algorithm implementation.

- 100 pattern strings of length ranging from 8 to 10 characters (9 ± 1). The length was adjusted several times through command-line parameters to the data-generation utility until a size was found for which the generated patterns would reliably meet the minimum threshold for sequences matched. The matching percentage over the 100 patterns ranged from 0.10% to 1.62%, with an average of 0.76%.

- A full set of correct counts of each pattern's occurrences in each sequence, as would be found by an exact-matching algorithm. This was established using a backtracking regular expression search for each of the selected patterns over each of the generated sequences. The regular expression was designed to properly include overlapping pattern occurrences.

- An additional set of answer-files were generated for the approximate-matching algorithm, for values of k ranging from 1 to 5. These, too, were generated using crafted regular expressions that would detect overlapping occurrences of the patterns.

A smaller data set was also generated for the purpose of testing and validation of the programs prior to full experiment runs.

The size of the data set was chosen after some preliminary experiments showed this size to be large-enough to rigorously exercise the algorithms while still running in reasonable time overall. This allowed for the experiments to be run repeatedly as the code itself evolved.

4.9 Testing Platform

The experiments were run on a dedicated machine running a version of the Linux operating system. The set of installed software was kept minimal to reduce the chance of background processes influencing the readings of general energy usage during the runs of experiments.

4.9.1 Hardware Specifications

The machine used for the experiments was an Intel-brand NUC7i5BNH i5-7260U, an ultra-compact device referred to by Intel as a "Next Unit of Computing" (NUC). The machine features the following specifications:

Processor	Intel® Core™ i5-7260U Processor
Processor Base Frequency	2.20 GHz
Max Turbo Frequency	3.40 GHz
Memory Type	DDR4-2133 1.2V SO-DIMM
Installed Memory	8 GB
Internal Drive Form Factor	M.2 and 2.5"
Installed Storage	120 GB M.2 SSD

Table 1: Hardware specifications of the test platform

The CPU is dual-core, with hyperthreading cores, for a total of 4 computational cores. This did not affect the experiments as none of the code was written to be multi-threaded.

4.9.2 Operating System and Configuration

The NUC was cleaned of the vendor-installed operating system. Linux was installed on it using the minimal server edition of Ubuntu Linux 22.04.1. All unnecessary packages and software services were either disabled or removed completely.

The development software (see next section) was then installed. This included the Linux version of the "Homebrew" package manager. Homebrew was specifically used to install the latest version of the LLVM Compiler Infrastructure, to allow all experiment code to be compiled directly on the NUC (rather than copying executable files built on a different machine). The Python and Perl interpreters that were used were also provided by Homebrew, as were the majority of the tools used for developing, testing, and evaluating the experiments on the development platform. The Rust language toolchain was installed and managed using the "Rustup" management software. Core software development packages from Ubuntu were also installed at this time.

4.9.3 Compilers and Other Tools

Table 2 lists the primary software tools and packages that were used in the creation and execution of the experiments, with the source from which they were obtained. The double line after the Python details indicates that the remaining tools were used on the development machine only, not on the experiments platform. These tools were used in the testing and evaluation of the experiments code separately from running experiments on the NUC.

The tools used for the experiments themselves do include the GNU Make tool, as it was the driver for automating the running of the full range of experiments on a regular basis.

4.10 Resources

All source code for all experiment programs, as well as the harness utility described and the Python utilities used for data generation and processing of results, is available on the GitHub platform [29].

5 Results and Analysis

During the experiment programs' development, an automation mechanism was developed based on the GNU Make tool. This mechanism allowed for the repeated running of the experiments suite while also providing control over aspects such as the number of repetitions and grouping of the algorithms.

Tool	Version	Source
GNU Make	4.3	Ubuntu 22.04
GCC	11.2.0	Ubuntu 22.04
LLVM	15.0.3	Homebrew
Intel® oneAPI Compiler	2022.2.0	Intel
Rust	1.65.0	Rustup Manager
Perl	5.36.0	Homebrew
Python	3.10.8	Homebrew
PCRE2	10.39	Ubuntu 22.04
Valgrind	3.20.0	Homebrew
`sloc`	0.2.1	Homebrew
`perf`	5.15.53	Ubuntu 22.04
`cloc`	1.94	Homebrew
`xz`	5.2.7	Homebrew
`lizard`	1.17.10	Homebrew
`countperl`	1.0.1	Ubuntu 22.04
`rust-code-analysis-cli`	0.0.23	GitHub

Table 2: Specifications of software tools used

The basis for the mechanism was a series of additional rules added to the already-present "Makefile" files that had been developed for the building of each language's set of programs. A single target-rule, "`experiments`", would recursively descend into each language-specific directory and trigger all algorithms in sequence. Each algorithm ran a specified number of times, and in most cases the initial run was discarded. This was to prevent the possibility of the data input files being in the disk device's cache, skewing the timing and energy readings with regards to later runs. All non-error output from the experiments was captured by the harness program and streamed to a single text file.

The format chosen for the file of results was YAML, due to the ready availability of parsing libraries. YAML had an advantage over other formats such as CSV (comma-separated values) in that it allowed the flexibility of complex nested data were it to be necessary, while also being emitted in a streaming fashion. This greatly reduced the potential for output to be corrupted between algorithm executions.

5.1 Results from the Experiments

The automated suite of experiments was run numerous times over the course of this research. The final run from which the analysis and conclusions here are drawn is preserved in the same repository on the GitHub platform [29] as all the other files related to this research.

The file of raw experiments data is named "`experiments-data-20221111.yml`". The final, full run of experiments took approximately 95 hours and 15 minutes to complete on the NUC device described in section 4.9.

5.1.1 Scope of the Experiments

The final run of the experiments generated a total of 1,190 data-points taken from runs of the 30 programs. The set of programs below includes the regular expression variant of the DFA-Gap algorithm, as described in 4.5.6. Table 3 shows the number of runs on a per-language, per-algorithm basis.

Language	Knuth-Morris-Pratt	Boyer-Moore	Bitap	Aho-Corasick	DFA-Gap	Regexp-Gap
C (GCC)	25	25	25	25	5	5
C (LLVM)	25	25	25	25	5	5
C (Intel)	25	25	25	25	5	5
C++ (GCC)	25	25	25	25	5	5
C++ (LLVM)	25	25	25	25	5	5
C++ (Intel)	25	25	25	25	5	5
Rust	25	25	25	25	5	5
Perl	5	5	5	25	3	3
Python	5	5	5	25	3	3

Table 3: Experiment iterations by language and algorithm

For all numbers 10 and greater, there was an additional "priming" run (as described above) to ensure that disk cache status did not play into the values for full run-time or package-level energy usage. Note also that the DFA-Gap and Regexp-Gap columns are stand-ins for 5 such columns each (for the values of k from 1 to 5). Each value of k was run for the same number of iterations.

5.1.2 Outliers and the Interpreted Languages

While it was known that the interpreted languages (Python and Perl) would be slower than the compiled languages, the reality of the results was surprising: as will be shown in section 5.2, below, the interpreted languages were in some cases more than 150 times slower than the fastest compiled program on the same algorithm.

This discovery required adjusting the automated experiments, to reduce the number of iterations for both interpreted languages. The Knuth-Morris-Pratt, Boyer-Moore, and Bitap algorithms were all reduced to 5 iterations each, for these languages. The Aho-Corasick algorithm ran in a more reasonable length of time and was left at 25 iterations, the same number as were run for the compiled languages.

For the DFA-Gap algorithm, both compiled and interpreted languages had to be reduced in terms of iterations given that approximate-matching algorithms are in general slower than their exact-matching counterparts. The compiled languages ran 5 iterations of this algorithm for values of k ranging from 1 to 5, whereas the interpreted languages were necessarily limited to 3 iterations for the same values of k. When the regular expressions variant of the algorithm was added, it too was run for 5 and 3 iterations over the languages.

5.2 Performance Comparisons

The first of the three measures of the languages' suitability was chosen to be the overall performance. This was chosen for first consideration as this is the metric that most developers notice first: how long did it take the program to complete? Run-time measurement was also the easiest of the metrics to gather, both in terms of total program execution and in wall-clock time specifically spent in the algorithms themselves.

It was in the measuring of performance that it first became clear what a stark difference there was between the interpreted and compiled languages. One can forget the extent of this gap when working with interpreted languages in-depth for long periods of time. The Perl and Python performance numbers were so large in many cases that they had to be omitted from charts to preserve clarity, even when put to logarithmic scaling.

5.2.1 Adjusting for Perl and Python

Early examination of the performance results for Perl and Python (section 4.5.6) had shown that the custom-built DFAs were strongly out-performed by using those languages' native support for regular expressions. Looking at these results separately from the rest of the experiments, the difference was considerable as is shown in figure 19.

Not only were the run-times themselves higher for the DFA implementations, but the growth in run-time as k increases was more pronounced for the DFA implementations than for the regular expression implementations. Note that while Perl's DFA was outperformed by Python's for the first two values of k, at $k = 3$ it was slightly lower and was steadily improving over Python for both $k = 4$ and $k = 5$. However, in the regular expression implementation Python remained the better performer consistently.

This had led to a significant question during development regarding the measurement of performance for the algorithms: should the Perl and Python manually-coded DFA implemen-

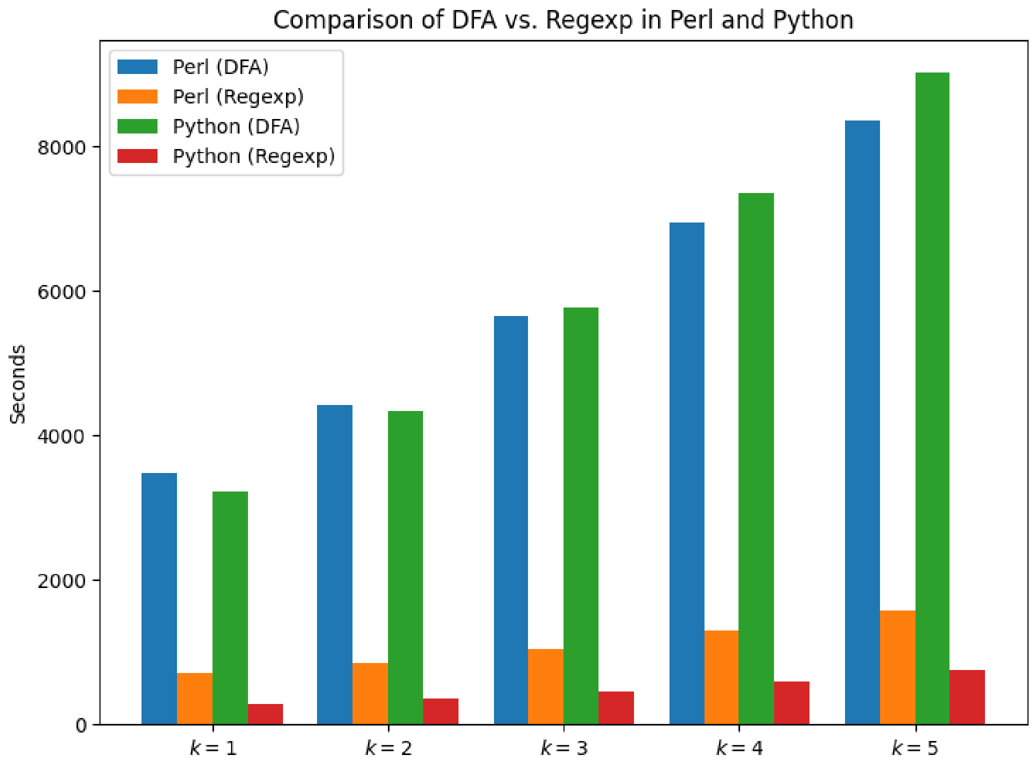


Figure 19: Bar charts of DFA vs. Regexp run-times for Perl and Python

tations be replaced by the regular expression implementations? It is highly unlikely that an experienced programmer in either of these two languages would choose to create the DFA manually when they could instead create a regular expression based on the pattern and a given value of k, in any case. When the performance is so drastically disparate it seems even less likely.

It was decided instead to implement the regular expression version of the experiment for the remaining three languages and include this in the final results and analysis. Figure 20 shows the run-time differences for these languages.

In these charts, the C and C++ values are averaged from the three toolchains. Examination of the complete run-time tables for DFA-Gap (15f) and Regexp-Gap (17f) showed that each toolchains' run-times were extremely close to each other. This is understandable, as the majority of the work in these cases would have been done by the PCRE2 library itself which is identical across the different toolchain executables.

While regular expressions improved the performance of both Perl and Python dramatically, it did the reverse for the compiled languages. While C++ showed to be faster than C when using regular expressions and Rust faster still, as can be seen in figure 20 the use of regular expressions by the compiled languages took more time than the hand-crafted DFAs.

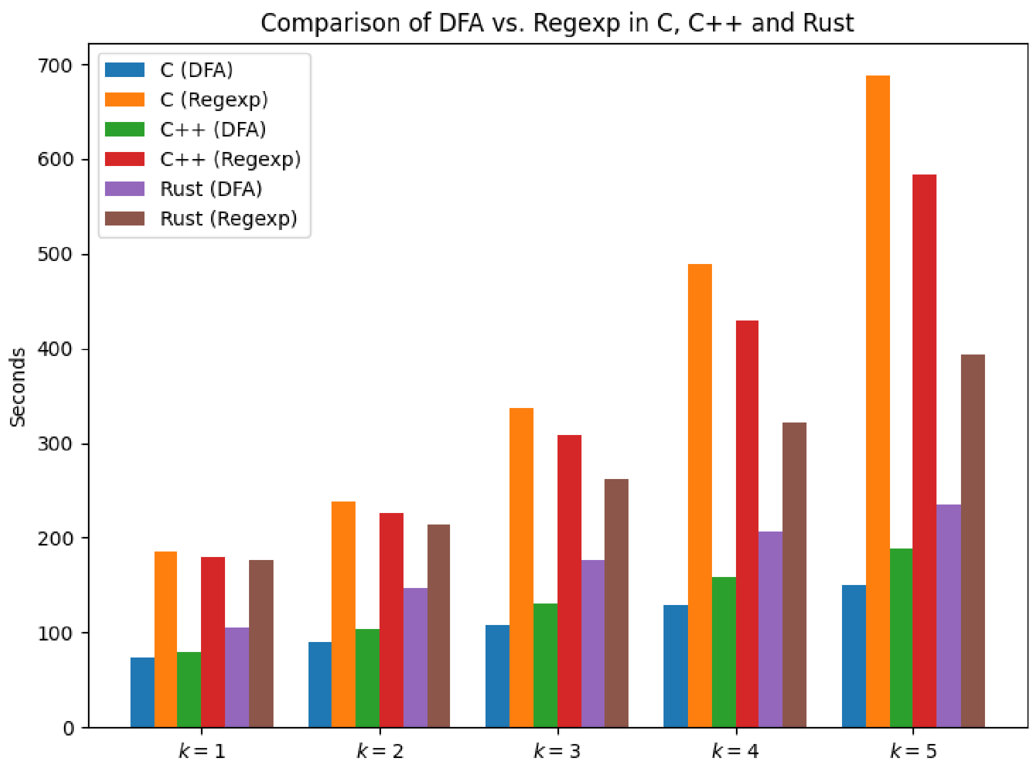


Figure 20: Bar charts of DFA vs. Regexp run-times for the compiled languages

5.2.2 Collected Performance Results

After the decision was made to implement the regular expression variant throughout, processing of the full performance results was done. The collection of sub-tables in table 4 shows the comparative performance of the languages on each of the algorithms. The time measurements are based on the algorithm run-time as opposed to the total run-time. In each table, the fastest language is listed first with the remaining ones following in order of performance. Run times are scaled by the fastest time. This has the result of showing each slower language's time as a percentage over the fastest.

The DFA-Gap and Regexp algorithms are shown for just $k = 3$. However, there was some fluctuation in the ranking of the languages as k went from 1 to 5. For the calculation of the metric of performance, all values of k for both of these algorithms will be used.

The set of charts in figures 21 to 26 illustrate the run-times by language for each of the six algorithms. Note that all of these except figure 26 omit the Perl and Python languages due to their extreme differences in the scale of running-time[8].

Starting with the Knuth-Morris-Pratt (21) and Boyer-Moore (22) algorithms which have

[8]As was mentioned, a logarithmic scale was also tried but the Perl/Python values were still too far from the norm (particularly in the Bitap results).

(a) Knuth-Morris-Pratt

Language	Runtime	Score
C (LLVM)	44.52	1.0000
Rust	50.28	1.1293
C++ (LLVM)	50.65	1.1377
C (Intel)	50.76	1.1402
C++ (Intel)	55.23	1.2406
C (GCC)	58.53	1.3146
C++ (GCC)	59.44	1.3350
Perl	1715.18	38.5237
Python	1847.06	41.4858

(b) Boyer-Moore

Language	Runtime	Score
C (Intel)	20.66	1.0000
C++ (Intel)	21.71	1.0508
C (GCC)	22.03	1.0664
Rust	22.68	1.0977
C++ (LLVM)	23.58	1.1413
C (LLVM)	23.86	1.1549
C++ (GCC)	24.44	1.1831
Perl	969.77	46.9470
Python	1052.63	50.9587

(c) Bitap

Language	Runtime	Score
C (Intel)	7.55	1.0000
C (GCC)	7.60	1.0068
C++ (Intel)	7.91	1.0475
C (LLVM)	8.47	1.1219
Rust	8.57	1.1350
C++ (GCC)	8.77	1.1616
C++ (LLVM)	9.35	1.2381
Perl	1173.06	155.3435
Python	1456.88	192.9283

(d) Aho-Corasick

Language	Runtime	Score
C (LLVM)	1.14	1.0000
C (Intel)	1.20	1.0443
C (GCC)	1.22	1.0644
C++ (GCC)	1.32	1.1519
C++ (LLVM)	1.33	1.1600
Rust	1.35	1.1816
C++ (Intel)	1.37	1.1988
Python	22.55	19.7040
Perl	45.10	39.4041

(e) DFA-Gap (k=3)

Language	Runtime	Score
C (GCC)	100.45	1.0000
C (LLVM)	107.97	1.0749
C (Intel)	116.13	1.1561
C++ (GCC)	123.25	1.2270
C++ (LLVM)	133.10	1.3250
C++ (Intel)	135.04	1.3443
Rust	177.11	1.7632
Perl	5647.35	56.2209
Python	5770.18	57.4437

(f) Regexp-Gap (k=3)

Language	Runtime	Score
Rust	261.48	1.0000
C++ (LLVM)	307.16	1.1747
C++ (GCC)	307.32	1.1753
C++ (Intel)	308.40	1.1794
C (Intel)	336.54	1.2871
C (GCC)	336.67	1.2876
C (LLVM)	337.00	1.2888
Python	454.93	1.7398
Perl	1045.03	3.9966

Table 4: Comparative run-times by algorithm

a passing relation to each other, the red lines indicate the fastest run-time and are meant to show the difference in the other languages' finish-time. In the Knuth-Morris-Pratt results, of interest is the fact that both C and C++ compiled by the LLVM toolchain strongly out-performed the GCC and Intel toolchains. This was the only algorithm in which LLVM controlled the performance for both languages. In Boyer-Moore, the Intel toolchain took the

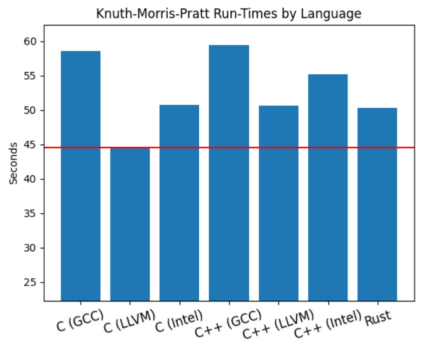


Figure 21: KMP run-times

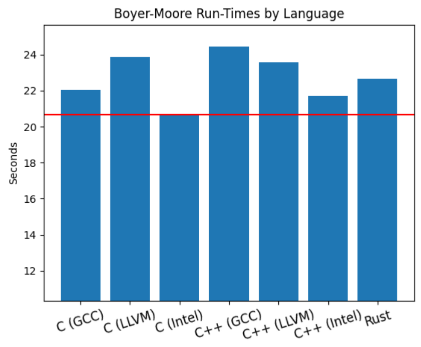


Figure 22: BM run-times

top spot with C, and the second spot with C++. Though this is not the only algorithm for which a C++ experiment placed second, this is the highest that C++ placed throughout all algorithms.

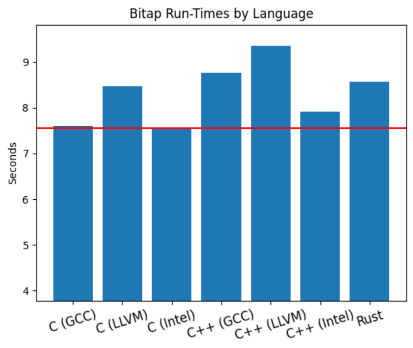


Figure 23: Bitap run-times

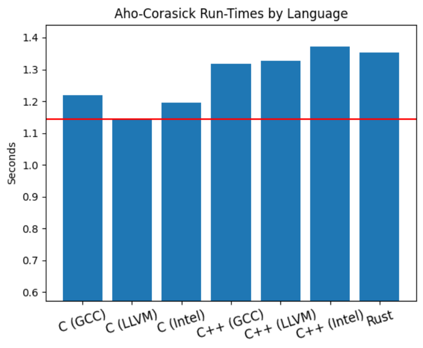


Figure 24: AC run-times

In the Bitap results shown in figure 23, the LLVM toolchain actually produced the slowest times within the C and C++ groups in this case. In the Aho-Corasick results (figure 24) LLVM once again produced the fastest code for the C version of the experiment. The Aho-Corasick run-times also highlight the overall speed of this algorithm, enabled by the ability to test all patterns in a single pass through each target sequence.

For the DFA and regular expression versions of the gap algorithm, figures 25 and 26, different styles of charts were drawn. As both of these experiments ran for five different values of the k parameter, the charts are designed to show all five run-times for each language as a stack.

The DFA experiments follow an almost expected trajectory, with C being faster than C++ and C++ faster than Rust. However, the regular expression experiments show an exactly opposite trend, with Rust being the fastest of the languages. Figure 26 also includes results for the Perl and Python languages, as this experiment gave results for them that could be

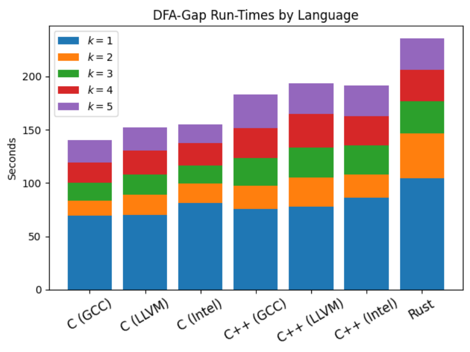


Figure 25: DFA run-times

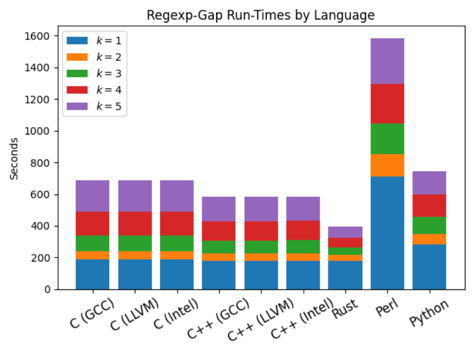


Figure 26: Regexp run-times

comfortably included.

Full tables of run-times scores for all values of k in both DFA-Gap and Regexp-Gap experiments can be found in section A. A short analysis of the relative size of the run-times themselves is done in section B, as those details are less directly relevant to the collected performance calculations.

5.3 Expressiveness Comparisons

Expressiveness was generally the most-challenging of the metrics to measure and evaluate. Even the most concrete, numerical measurements of code such as SLOC and complexity are subject to some debate and dissent. In these aspects of the research, measurement was taken through a range of open-source tools and the results tracked closely with expectation; the interpreted languages showed the strongest scores on the three expressiveness sub-metrics.

However, much of what was measured for this metric was also dependent upon the skill shown in the writing of the programs. Familiarity with the different languages varied, from significant experience in C and Perl to relative newness with regards to Rust. It is not possible to say whether someone more expert in the Rust language, for example, could have implemented the chosen algorithms with more efficiency and better expressiveness metrics.

Additionally, the complexity component of this metric had been cast in some doubt by the inability to use a single tool for all languages involved. This is shown in section 5.6 to have not influenced the final rankings.

In comparing expressiveness, it is useful to look at all three of the chosen comparative measures in an aggregated fashion. First, the individual numbers will be examined. As with the previous two bases, tables will show the numbers comparing the languages. Unlike the previous sections, the numbers shown will be for the combined files of all algorithm modules, as well as runner modules and input modules. For each language the source code was combined in a way that facilitated each of the metrics:

SLOC: Total SLOC values for each language's files were combined. In the case of C and C++, this includes relevant lines from the header files for the runner and input modules.

Complexity: Each file's cyclomatic complexity values were computed on a per-function basis. Each module's function scores were averaged, and all modules' averages for a given language were summed together.

Conciseness: Using a method adapted from Bergmans, et al [6], each source code file for a given language was stripped of comments and then merged into a sort of "archive" using the standard "`cat`" command available on Linux. Each such resulting file was then compressed with the "`xz`" compression utility and the compression ratio recorded.

5.3.1 Source Lines of Code

In table 5, the SLOC totals are shown in three sub-tables: only the algorithm implementations, then the framework totals, and finally the combined totals of all lines.

(a) Algorithm lines

Language	Code	Score
Python	272	1.0000
Perl	376	1.3824
C++	403	1.4816
C	528	1.9412
Rust	543	1.9963

(b) Framework lines

Language	Support	Score
Python	148	1.0000
Perl	211	1.4257
C++	269	1.8176
Rust	272	1.8378
C	353	2.3851

(c) Total of lines

Language	All	Score
Python	420	1.0000
Perl	587	1.3976
C++	672	1.6000
Rust	815	1.9405
C	881	2.0976

Table 5: Comparison of SLOC by language

Here Python is the clear leader, with Perl being 40% larger in table 5c. C++ maintains the third-place ranking across all three sub-tables, and Rust edges out C in both the framework measurement and the total lines measurement. Rust's score in table 5a likely suffered from the necessary mechanism that was used to pass and unpack pattern representation between a given algorithm's initialization function and the algorithm function itself. Compare the listings, below.

Listing 9: C unpacking of pattern data (Aho-Corasick)

```c
    int *pattern_count = (int *)pat_data[0];
    int *goto_fn = (int *)pat_data[1];
    int *failure_fn = (int *)pat_data[2];
    Set *output_fn = (Set *)pat_data[3];
```

Listing 10: Rust unpacking of pattern data (Aho-Corasick)

```rust
    let pattern_count = match &pat_data[0] {
        MultiPatternData::PatternCount(val) => val,
        _ => panic!("Incorrect value at pat_data slot 0"),
    };
    let goto_fn = match &pat_data[1] {
        MultiPatternData::PatternIntVecVec(val) => val,
        _ => panic!("Incorrect value at pat_data slot 1"),
    };
    let failure_fn = match &pat_data[2] {
        MultiPatternData::PatternUsizeVec(val) => val,
        _ => panic!("Incorrect value at pat_data slot 2"),
    };
    let output_fn = match &pat_data[3] {
        MultiPatternData::PatternTypeVec(val) => val,
        _ => panic!("Incorrect value at pat_data slot 3"),
    };
```

In this area, C and C++ each needed only one line per element passed (listing 9), while Rust required four[9] (listing 10). This is a 12-line difference in just the Aho-Corasick implementation, so it can be understood how this could propagate through the other algorithm implementations.

By further comparison: both Python and Perl, having the native ability to pass arrays of differing types, needed only one line to unpack the pattern representations.

5.3.2 Cyclomatic Complexity

Table 6 shows a similar break-down of the cyclomatic complexity measurements: algorithms, framework, and combined total.

Unlike the SLOC measurements, however, there is greater variation in the rankings of the languages. Here the contest between the first and second rankings was between Python and C++. Rust placed last in the measure of the algorithm code, possibly due to the same issue of excess lines around the unpacking of pattern data. Rust did rank third in the framework table 6b where it also had a significantly lower average value than C++ or Python. Even as Rust ended tied for the fourth rank in the total table, its totaled average complexity was still the lowest of the five.

[9]One of which was a closing-brace with a semicolon and might not have been counted by the `sloc` tool.

(a) Algorithms complexity

Language	Total	Avg
Python	76	19.57
C++	81	16.83
Perl	106	26.97
C	114	18.30
Rust	132	17.97

(b) Framework complexity

Language	Total	Avg
C++	43	10.75
Python	47	9.90
Rust	58	5.43
Perl	61	12.90
C	76	19.00

(c) Total complexity

Language	Total	Avg
Python	123	29.47
C++	124	27.58
Perl	167	39.87
C	190	37.30
Rust	190	23.40

Table 6: Comparison of complexity by language

It is important here to note again that it was necessary to use different tools to calculate the complexity of the Perl and Rust code. The `lizard` tool did not support Perl at all, and examination of the Rust results showed some bugs in the handling of Rust. Because of this, it is not possible to say with certainty that the relative comparisons of complexity between the five languages are completely sound.

5.3.3 Conciseness

Conciseness proved to be a difficult concept to quantify. The approach taken in [6] was designed around a significantly larger database of source code, but as the results from that paper were compelling, it led to the adaptation of a variation of that methodology here.

The steps for deriving this measurement were:

1. A clean copy of all source code was made into a series of directories named by language
2. All comments and blank lines were removed by the `cloc` tool
3. Each individual language directory had all text files concatenated into a single file
4. Each of these text files were compressed with `xv` and the compression ratio recorded

A key difference between [6] and here is the limited scope of the data being analyzed; since there is essentially just one "project" being put to scrutiny, the results are sensitive to the fact that some of the files were smaller in size to begin with. Initially the technique in [6] was followed exactly, and involved using the standard `tar` utility to create the archives of each language. But when put into practice it was found that the metadata overhead of a `tar` archive was more than the size of the actual data in some cases.

Further, in their paper, Bergmans and their team had removed most smaller examples of code for each language from the final measurements. This was not an option here given the limited size and number of the samples. For this reason the decision was made to focus the compression on the text content only, by applying concatenation to the stripped versions of the source files and compressing the results of this process.

In table 7, the results are shown.

Language	Ratio	Score
Python	78.50%	1.0000
Perl	80.50%	1.0255
C	80.60%	1.0268
Rust	80.80%	1.0293
C++	81.00%	1.0318

Table 7: Comparison of compressibility by language

The ranking of Python and Perl as the first two is expected. However, the placing of C ahead of both Rust and C++ came as a surprise given the presence of highly-repetitive calls to library routines for the manual memory management. However, C and Rust both likely benefited from the need for additional support code in the Aho-Corasick algorithm which could have brought the compression ratio slightly further down.

5.3.4 Combining the Expressiveness Metrics

At this point the next step was to derive a single measurement of expressiveness from the three measurements taken. For this, it would be needed to have "score" values for cyclomatic complexity that were in line with the scores computed for SLOC and conciseness. These values were computed from table 6c and are shown in table 8.

Language	Total	Score
Python	123	1.0000
C++	124	1.0081
Perl	167	1.3577
C	190	1.5447
Rust	190	1.5447

Table 8: Scoring of the cyclomatic complexity results

Here, the score shows a virtual tie for first position between C++ and Python. Their gap is just one point (less than 1%), while the next gap is 43 (an increase of almost 35%).

Now it would be possible to utilize the three expressiveness metrics to calculate a single value for each language. The following values would be combined into a series of 3-element vectors for each language:

- The SLOC scores from table 5c, the total of lines

- The cyclomatic scores from table 8
- The conciseness scores from table 7

The vectors were normalized into unit vectors. Then the vectors' lengths were calculated and scaled by the lowest value, giving the values in table 9:

Language	SLOC	Complexity	Compression	Score
Python	1.0000	1.0000	1.0000	1.0000
C++	1.6000	1.0081	1.0318	1.1565
Perl	1.3976	1.3577	1.0255	1.2109
Rust	1.9405	1.5447	1.0293	1.4086
C	2.0976	1.5447	1.0268	1.4503

Table 9: Calculated expressiveness score

It was not a surprise that Python ended up ranked as the most-expressive of the five contenders, as it had led the group in all three expressiveness metrics. The ranking of C++ ahead of Perl by just over 5 percentage points was more of a surprise, while the presence of C at the bottom was not surprising.

Figure 27 illustrates the expressiveness values as vectors anchored at the origin. Judging the length of the vectors helps to show the difference between Python in the 1st position and C in the 5th.

Because of the possibly-inconsistent nature of the complexity data, it was decided to derive a second score based on just the SLOC and compression scores. All other calculation steps were the same. This resulted in table 10.

Language	SLOC	Compression	Score
Python	1.0000	1.0000	1.0000
Perl	1.3976	1.0255	1.1406
C++	1.6000	1.0318	1.2159
Rust	1.9405	1.0293	1.3446
C	2.0976	1.0268	1.4070

Table 10: Calculated expressiveness score, 2-axis

Here, the results more closely tracked with expectations: Python and Perl head the list while C++ and Rust take the next two places. As has been mentioned, it is possible that Rust suffered in the measurements due to the data-unpacking mechanism required by the framework.

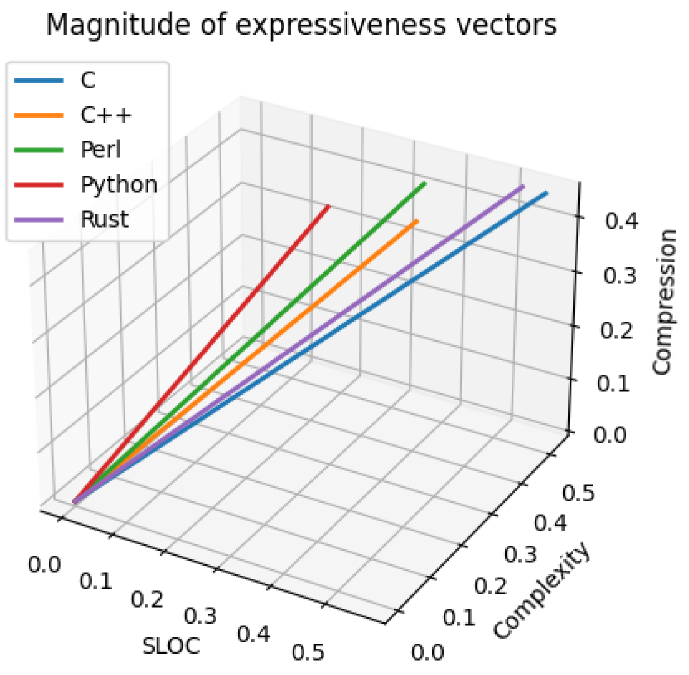


Figure 27: Calculated expressiveness as vectors

5.4 Energy Usage Comparisons

In this section the energy usage results are examined. Table 11 shows the comparative energy usage over time (Joules per second) for each algorithm with the same scaling methodology applied as was used for the performance tables. The tables here use the “Package” and “DRAM” energy values combined together, divided by total run-time.

In regards to energy usage, the only significant barrier to overcome was ensuring that the machine used to run the experiments would be sufficiently isolated so as to have the least amount of interference possible from other running processes. The nature of the RAPL system of measuring power consumption also required that the experiments be run with super-user access levels. This amplified the need for the code to be as safe as possible, even though the testing machine was not made accessible to the general internet.

RAPL posed other challenges to use, such as the previously-mentioned 32-bit limitation in the registers used for tracking the energy usage. But this was overcome, and the RAPL system ended up proving robust-enough to handle measuring programs whose run-times ranged from barely one second to nearly three hours.

The language exhibiting the lowest energy usage is listed first, with the rest ranked behind

(a) Knuth-Morris-Pratt

Language	Energy	Score
Rust	8.29	1.0000
C (GCC)	8.63	1.0403
C (LLVM)	8.92	1.0756
C (Intel)	8.94	1.0786
C++ (GCC)	9.11	1.0983
C++ (LLVM)	9.15	1.1033
C++ (Intel)	9.20	1.1102
Python	9.66	1.1651
Perl	10.74	1.2957

(b) Boyer-Moore

Language	Energy	Score
C++ (GCC)	8.45	1.0000
C++ (LLVM)	8.55	1.0115
Rust	8.73	1.0331
C (GCC)	8.75	1.0358
C++ (Intel)	9.08	1.0747
C (LLVM)	9.22	1.0918
C (Intel)	9.72	1.1501
Python	10.97	1.2985
Perl	11.07	1.3103

(c) Bitap

Language	Energy	Score
C++ (LLVM)	8.95	1.0000
C (Intel)	9.00	1.0049
Rust	9.02	1.0077
C (LLVM)	9.04	1.0101
C++ (Intel)	9.35	1.0440
Python	9.67	1.0799
C++ (GCC)	9.72	1.0854
C (GCC)	9.75	1.0897
Perl	10.98	1.2268

(d) Aho-Corasick

Language	Energy	Score
C++ (LLVM)	9.28	1.0000
C (Intel)	9.56	1.0303
C (GCC)	9.58	1.0332
C (LLVM)	9.69	1.0448
C++ (Intel)	9.71	1.0471
C++ (GCC)	10.03	1.0813
Rust	10.15	1.0939
Perl	10.43	1.1248
Python	10.89	1.1736

(e) DFA-Gap (k=3)

Language	Energy	Score
Rust	7.35	1.0000
C++ (LLVM)	8.53	1.1603
C (LLVM)	8.68	1.1812
C++ (Intel)	8.87	1.2078
C (Intel)	8.99	1.2231
C (GCC)	9.08	1.2352
C++ (GCC)	9.18	1.2497
Python	9.65	1.3140
Perl	10.21	1.3897

(f) Regexp-Gap (k=3)

Language	Energy	Score
C (Intel)	10.15	1.0000
C (GCC)	10.21	1.0055
C (LLVM)	10.25	1.0094
C++ (LLVM)	10.59	1.0428
C++ (GCC)	10.62	1.0460
C++ (Intel)	10.63	1.0473
Perl	10.67	1.0513
Rust	10.72	1.0561
Python	10.78	1.0618

Table 11: Comparative energy usage over time by algorithm

it. The DFA-Gap and Regexp algorithms are again represented by $k = 3$, and again the rankings of the languages varied with k.

These results are further illustrated by figure 28. In this collection of bar-charts, the Rust language (represented by the pink bars) can be seen to be the lowest value in the Knuth-Morris-Pratt and DFA-Gap instances. In fact, Rust scored the lowest energy usage for all five variations of the DFA-Gap algorithm, making it the most-efficient language for 6 of the 14 distinct groups of experiments. Here, Perl and Python are included because the per-second measurements normalize their higher energy usage.

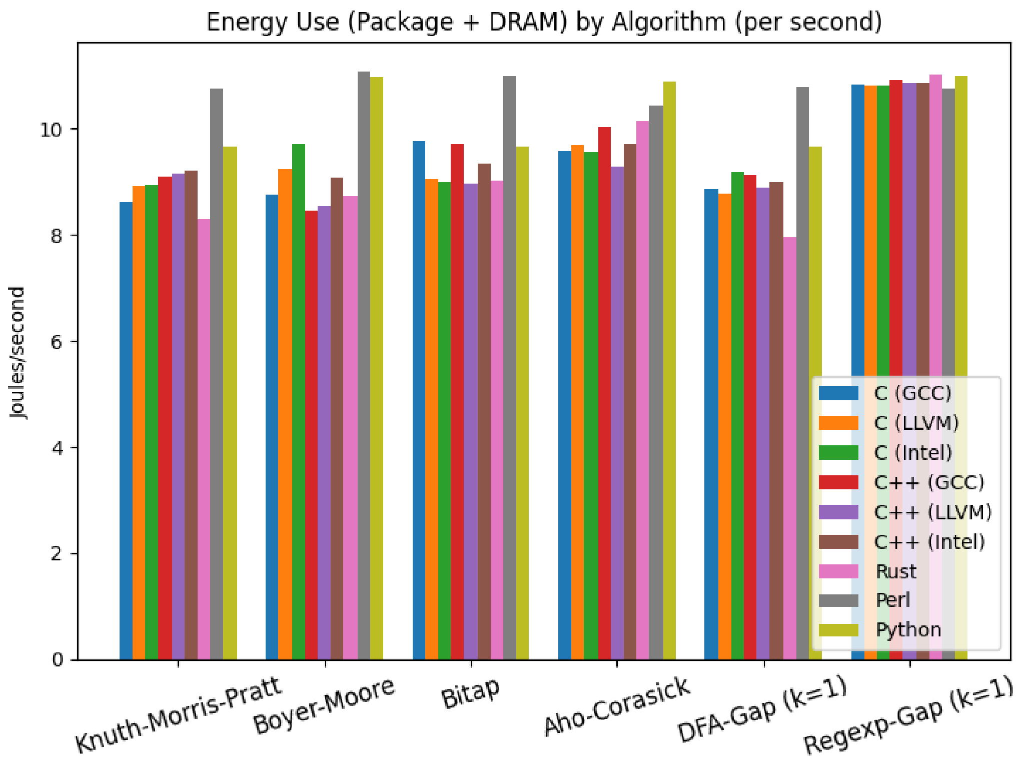


Figure 28: Energy/second, by algorithm

The fact that Rust was the overall best performer in terms of energy usage is underscored by the chart in figure 29. In this chart, the total (mean) energy usage by all experiments in each language is shown. Here, the Perl and Python languages have been removed again due to the fact that their longer run-times lead to energy numbers well out of scale with respect to the compiled languages. The red horizontal line shows the total usage by Rust and illustrates the differences with C and C++ across all three of their toolchains.

As with the run-times score tables, full tables of energy scores for all values of k in both DFA-Gap and Regexp-Gap experiments can be found in section A.

5.5 Combining the Bases

With three distinct measurements available for combination and analysis, the first step was to derive a single set of scores for the run-time basis and the energy-used basis, combining the values from all 14 groups of experiments into single values. For this, values were simply added

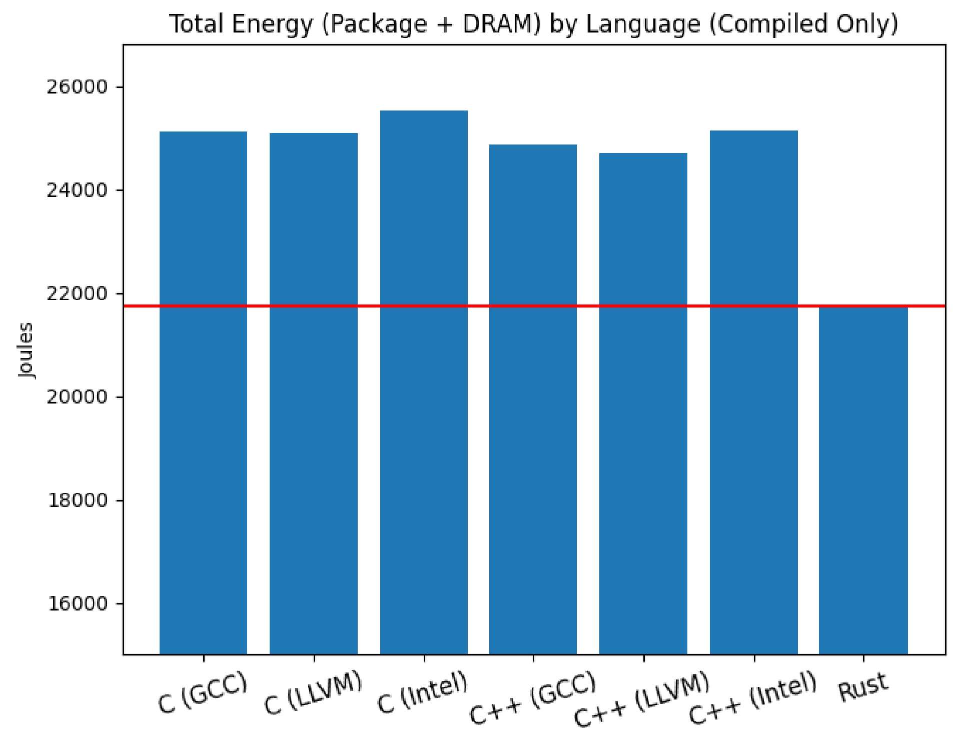


Figure 29: Total energy by language

together then scaled by the minimum value as has been done before. The total run-time table 12a shows these values, derived from the algorithm-only run-time metric. In table 12b, the same is done with the combination of the "Package" and "DRAM" energy metrics.

(a) Scores for total run-time

Language	Runtime	Score
Rust	2321.58	1.0000
C++ (GCC)	2451.74	1.0561
C++ (LLVM)	2482.64	1.0694
C++ (Intel)	2499.37	1.0766
C (GCC)	2541.39	1.0947
C (LLVM)	2566.72	1.1056
C (Intel)	2606.27	1.1226
Python	36503.46	15.7236
Perl	38234.67	16.4693

(b) Scores for total energy usage

Language	Energy	Score
Rust	21756.11	1.0000
C++ (LLVM)	24691.83	1.1349
C++ (GCC)	24878.42	1.1435
C (LLVM)	25074.22	1.1525
C (GCC)	25115.48	1.1544
C++ (Intel)	25130.75	1.1551
C (Intel)	25528.86	1.1734
Python	357076.12	16.4127
Perl	392680.44	18.0492

Table 12: Final scores for totaled run-time and energy, by language

From here came a question in two parts: Should the final scoring be based on the languages' scores for these bases or based on the language rankings? And secondly, should the cyclomatic complexity data be included in the expressiveness basis or not?

While the question regarding the complexity data has already been discussed, it still remained to be seen whether it would affect the final rankings. More than this, however, was the question of whether to simply apply the ranking of each language to determine final placement. Doing so might create a stronger distinction between adjacent languages than the scores themselves would. On the other hand, using the scores would retain the subtlety in the differences between languages.

The decision was made to examine all combinations of these two variables. Table 13 shows the four sub-tables that result from this analysis. As can be seen from comparing 13a with 13b, and 13c with 13d, the cyclomatic complexity values did not affect the respective rankings at all: both pairs of tables have identical placement for their 9 rows. Comparing column-wise (13a with 13c, and 13b with 13d), it can be seen that using the rank in place of scores lead to a slight difference in final ranking. Specifically, Rust and Python each moved up one place under a ranking basis. More significant, though, is how the different approaches affected the changes in scores.

(a) Score by scale, with complexity

Language	Score
C++ (GCC)	1.0000
C++ (LLVM)	1.0001
C++ (Intel)	1.0007
Rust	1.2058
C (GCC)	1.2446
C (LLVM)	1.2447
C (Intel)	1.2454
Python	3.2200
Perl	3.4881

(b) Score by scale, no complexity

Language	Score
C++ (GCC)	1.0000
C++ (LLVM)	1.0001
C++ (Intel)	1.0006
Rust	1.0987
C (GCC)	1.1521
C (LLVM)	1.1523
C (Intel)	1.1529
Python	3.0424
Perl	3.2800

(c) Score by ranks, with complexity

Language	Score
C++ (GCC)	1.0000
C++ (LLVM)	1.0000
Rust	1.4951
C++ (Intel)	1.8150
C (GCC)	2.4132
C (LLVM)	2.4375
Python	2.7547
C (Intel)	2.9406
Perl	3.3166

(d) Score by ranks, no complexity

Language	Score
C++ (GCC)	1.0000
C++ (LLVM)	1.0000
Rust	1.3143
C++ (Intel)	1.6652
C (GCC)	2.1213
C (LLVM)	2.1426
Python	2.4215
C (Intel)	2.5849
Perl	2.7469

Table 13: Final scores for all combined metrics, by language

Comparing the two sub-tables that include complexity values, it can be seen that the difference between the top (GCC C++) and bottom (Perl) is slightly higher in 13a than in 13c. This difference is even more pronounced in the two non-complexity tables. In addition to this, the growth of the score value is smoother when the languages are arranged based on rank. This is illustrated in figure 30.

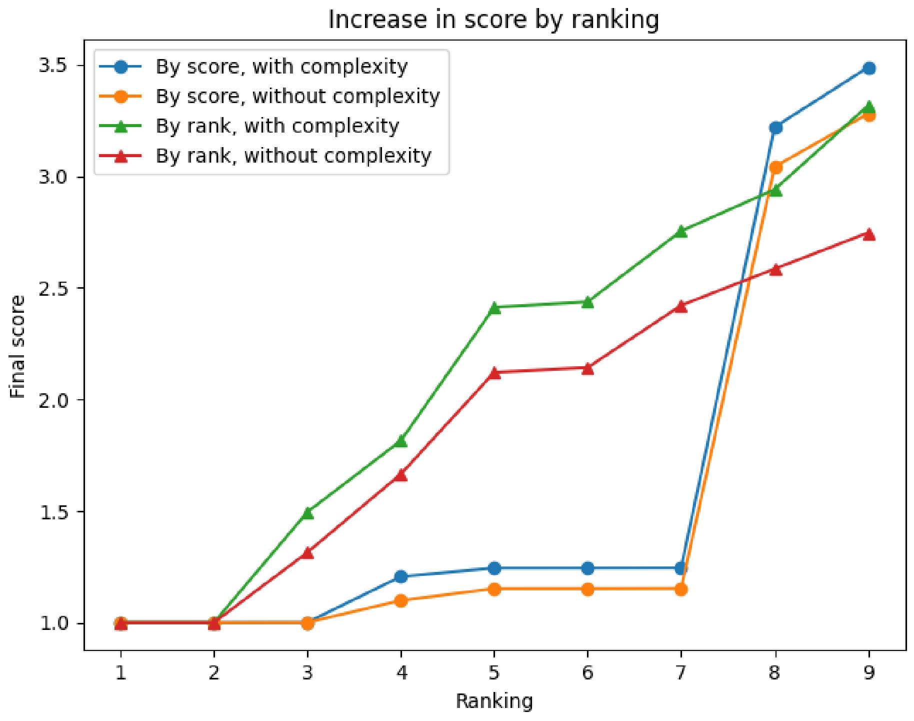


Figure 30: Plot of the rise in final score by ranking

In the lines for the score-based ordering (marked by solid circles on the plot), the final scores remain well under 1.5 until the 8th and 9th positions (which correspond to Python and Perl respectively). This is due to the fact that score-based ordering would necessarily be very reactive to the significant jumps in the final scores for run-time and energy usage. Looking at the lines for ranking-based ordering (solid triangles), these follow a more linear path while the final scores for Perl still ending up in the same general range as for the score-based ordering.

The lines for score-based ordering also show the effect of the top three entries' scores being within a range of 0.07%, resulting in flat lines until the 4th data-point. The ranking-based ordering exhibits this same pattern for the first two data-points as well, due to the entries in 1st and 2nd positions having identical expressiveness rankings and reciprocal rankings for run-time and energy.

Throughout these experiments, the C and C++ languages have been evaluated for three different toolchains each. What would the final results look like, if each of C and C++ were represented by only their best-scoring variants?

In figure 31, the line for the first four rankings is much more straight than had been in figure 30. It is worth noting that the change between Python in 4th place and Perl in 5th is much smaller than for the previous places.

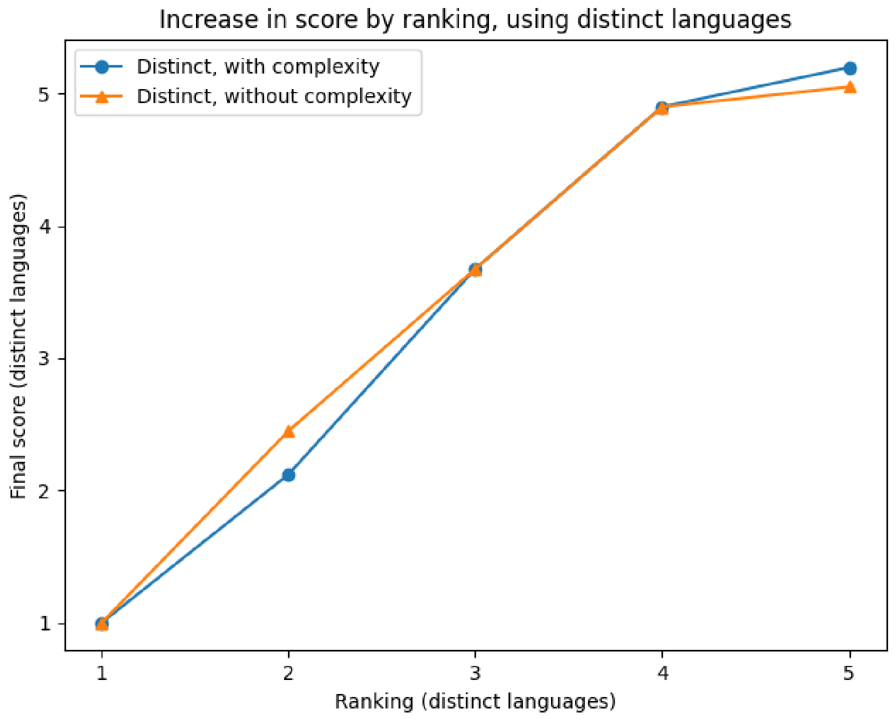


Figure 31: Plot of the rise in final score, using distinct languages

5.6 Final Rankings

In table 14 the five distinct languages are shown with their ordering and score based on ranking, both with and without the complexity metrics. In the end, the measurements taken and evaluated began to resemble the concept of a triathlon.

In performance, Rust outperformed C++ by a 5.6% margin and Rust's performance numbers were steady and reliable across all algorithms. In terms of energy usage, Rust was top-ranked in almost half of the algorithms and variations, also leading C++ in the final energy ranking by 13.5%. The measure of expressiveness is the only area in which Rust was not in the top place, and it was only by virtue of this metric that C++ edged out Rust in the final rankings. They swap places because of the percentage-points of difference

(a) Distinct by ranks, with complexity

Language	Score
C++	1.0000
Rust	1.2247
Python	1.6583
C	1.8930
Perl	2.2174

(b) Distinct by ranks, no complexity

Language	Score
C++	1.0000
Rust	1.0290
Python	1.3933
C	1.5904
Perl	1.7823

Table 14: Final scores for all combined metrics, by distinct language

in expressiveness: 34 percentage-points with complexity and 12 points without. Without complexity, Rust was only 2.9% behind C++ in the final ranking. While Rust came in 4th in all three expressiveness scores, C++ came in 3rd (SLOC), 2nd (complexity), and 5th (conciseness). However, both came in behind Python in general and behind Perl as well when complexity was not considered.

C++, being a more mature language, has the benefit of a more-established standard library that contributed to a better expressiveness score. Rust has the opportunity to improve as the language further evolves. New features and broader utilization will bring in greater clarity and consistency to Rust, which should improve all aspects of expressiveness.

An earlier incarnation of this work had focused on the security benefits of the Rust language. Recently, ZDNet published an article [32] about the National Security Agency's recommendation for developers to consider switching to programming languages that feature greater memory safety. While memory safety was not a focus of this research, it is noteworthy that Rust is considered to be the safest of the group in this regard. All of the experiments written in compiled languages were tested with Valgrind, being refined and debugged until Valgrind reported detecting no memory-related errors[10]. Of the compiled languages, only Rust never exhibited any memory errors.

With this, it becomes more understandable why Perkel [27] found so many in the sciences turning to Rust as their choice of a performance-oriented language. Though the youngest of the languages evaluated here, Rust has quickly grown to showing great potential for a wide range of applications.

Considering the growing demand for power in the world's data centers, as programmers become more focused on providing energy-efficient code, it is clear that a newer language–Rust– is immediately available to meet this need.

[10]Perl and Python were not tested with Valgrind, as they are built on bytecode interpreters.

6 Conclusions

In choosing performance, expressiveness and energy use as the three metrics to measure on, the goal was to provide an environment in which any of the five languages would have the chance to stand out. The results of this research can be considered successful: each of the five languages managed to be at least as high as the 2nd position in at least one metrics table. The methodology itself was demonstrated to evaluate disparate languages in a fair manner, even when some measurements were significantly disproportionate.

In the changing landscape of priorities, where security and power usage become as important as performance, Rust can be recommended for this field of computing not only for its approach to memory safety, but also on the merits of raw performance and lower energy consumption. The methodology developed and demonstrated here showed a difference of nearly 14% between Rust and the next-lowest energy usage score. That level of difference in energy efficiency has real-world implications that cannot be ignored.

Additionally, the novel DFA-Gap algorithm was shown to be effective at approximate-matching while also being simple to implement. When directly compared to the use of an existing regular expression engine, it consistently performed better for the non-interpreted languages.

The DFA-Gap algorithm has shown that it can be applied in cases where an edit distance-based algorithm yields unsatisfactory results. Sequence alignments can be computed with more control over the gaps between nucleotides. With the distinction of its approach to defining and constraining gaps, it can be further developed as an additional tool for researchers to use. Future work can also include greater analysis of and experimentation with the novel algorithm; based on the structure of Aho-Corasick it is very possible that the algorithm can be extended to multiple-pattern matching, as well.

Presented with a larger dataset, how would these metrics change? A greater selection of source code files could lead to more precision in the expressiveness measurements, while more data points would have a similar effect on the scores for energy usage and performance. Where memory safety was only briefly addressed here, additional work and research could add consideration of memory management and memory issues to the metrics. This could take the form of an additional facet of expressiveness, or even become its own metric. It could be possible to refine the methodology in ways that weigh the different metrics as opposed to treating them equally. It might then be further refined in new ways that would allow someone to choose the weights of the various metrics based on their specific needs and goals.

In the end, where energy efficiency is as important as performance, this methodology has shown its ability to clearly evaluate the suitability of a programming language.

References

[1] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333—340, June 1975. URL: `https://doi.org/10.1145/360825.360855`.

[2] Kalev Alpernas, Yotam M. Y. Feldman, and Hila Peleg. The wonderful wizard of loc. *Onward! 2020: Proceedings of the 2020 ACM SIGPLAN International Symposium*, pages 146–156, November 2020. URL: `https://doi.org/10.1145/3426428.3426921`.

[3] Matt Asay. Rust, not Firefox, is Mozilla's greatest industry contribution. `https://www.techrepublic.com/article/rust-not-firefox-is-mozillas-greatest-industry-contribution/`. Published: April 2021.

[4] Ricardo Baeza-Yates and Gaston H. Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10):74–82, October 1992. URL: `https://doi.org/10.1145/135239.135243`.

[5] Michela Becchi and Patrick Crowley. Extending finite automata to efficiently match perl-compatible regular expressions. In *Proceedings of the 2008 ACM CoNEXT Conference*, number 25 in CoNEXT '08, pages 1–12. Association for Computing Machinery, December 2008. URL: `https://doi.org/10.1145/1544012.1544037`.

[6] Lodewijk Bergmans, Xander Schrijen, Edwin Ouwehand, and Magiel Bruntink. Measuring source code conciseness across programming languages using compression. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 47–57. IEEE, 2021. URL: `https://doi.org/10.1109/SCAM52516.2021.00015`.

[7] Donnie Berkholz. Programming languages ranked by expressiveness. `https://redmonk.com/dberkholz/2013/03/25/programming-languages-ranked-by-expressiveness/`. Published: March 2013.

[8] Raju Bhukya and DVLN Somayajulu. Exact multiple pattern matching algorithm using dna sequence and pattern pair. *International Journal of Computer Applications*, 17(8):32–38, March 2011. URL: `https://doi.org/10.5120/2239-2862`.

[9] Jim Blandy, Jason Orendorff, and Leonora F. S. Tindall. *Programming Rust*. O'Reilly Media, Inc., 2 edition, 2021.

[10] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, October 1977. URL: `https://doi.org/10.1145/359842.359859`.

[11] Alex Cabral. The computer science behind dna sequencing. `https://sitn.hms.harvard.edu/flash/2019/the-computer-science-behind-dna-sequencing/`. Published: April 2019.

[12] Christian Charras and Thierry Lecroq. *Handbook of Exact String Matching Algorithms*. College Publications, 2004.

[13] Yangjun Chen and Hoang Hai Nguyen. On the string matching with k differences in dna databases. *Proceedings of the VLDB Endowment*, 14(6):903–915, 2021.

[14] Lok-Lam Cheng, David W Cheung, and Siu-Ming Yiu. Approximate string matching in dna sequences. In *Eighth International Conference on Database Systems for Advanced Applications, 2003.(DASFAA 2003). Proceedings.*, pages 303–310. IEEE, 2003.

[15] Catalin Cimpanu. Chrome: 70`https://www.zdnet.com/article/chrome-70-of-all-security-bugs-are-memory-safety-issues/`. Published: May 2020.

[16] Intel Corp. *Intel® 64 and IA-32 Architectures Software Developer Manual*, volume Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4. Intel, 2022. URL: `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html`.

[17] Clarissa Garcia. The real amount of energy a data center uses. `https://www.akcp.com/blog/the-real-amount-of-energy-a-data-center-use/`. Published: February, 2022.

[18] James M. Heather and Benjamin Chain. The sequence of sequencers: The history of sequencing dna. *Genomics*, 107(1):1–8, January 2016. URL: `https://doi.org/10.1016/j.ygeno.2015.11.003`.

[19] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. Rapl in action: Experiences in using rapl for power measurements. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, 3(2):1–26, June 2018. URL: `https://doi.org/10.1145/3177754`.

[20] Donald E. Knuth, Jr. James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(3):323–350, 1977. URL: `https://doi.org/10.1137/0206024`.

[21] Johannes Köster. Rust-bio: a fast and safe bioinformatics library. *Bioinformatics*, 32(3):444–446, February 2016. URL: `https://doi.org/10.1093/bioinformatics/btv573`.

[22] Eric Masanet and Nuoa Lei. How much energy do data centers really use? `https://energyinnovation.org/2020/03/17/how-much-energy-do-data-centers-really-use/`. Published: March 17, 2020.

[23] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, December 1976. URL: `https://doi.org/10.1109/TSE.1976.233837`.

[24] Peyman Neamatollahi, Montassir Hadi, and Mahmoud Naghibzadeh. Simple and efficient pattern matching algorithms for biological sequences. *IEEE Access*, 8:23838–23846, January 2020. URL: `https://doi.org/10.1109/ACCESS.2020.2969038`.

[25] Vu Nguyen, Sophia Deeds-Rubi, Thomas Tan, and Barry Boehm. A sloc counting standard. In *Cocomo ii forum*, pages 1–16, 2007.

[26] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Energy efficiency across programming languages: How do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2017, pages 256–267. Association for Computing Machinery, 2017.

[27] Jeffrey M. Perkel. Why scientists are turning to rust. *Nature*, 588:185–186, December 2020. URL: `https://doi.org/10.1038/d41586-020-03382-2`.

[28] Pooja Manisha Rahate and M. B. Chandak. Comparative study of string matching algorithms for dna dataset. *International Journal of Computer Sciences and Engineering*, 6(5):1067–1074, May 2018. URL: `https://doi.org/10.26438/ijcse/v6i5.10671074`.

[29] Randy J. Ray. Evaluating languages for bioinformatics: Energy, expressiveness and performance. `https://github.com/rjray/mscs-thesis-project`. Published: November 2022.

[30] Stayam Shandilya. Knuth-morris-pratt algorithm - understanding it my way. `https://i-satyam.blogspot.com/2015/12/knuth-morris-pratt-algorithm.html`. Published: December 10, 2015.

[31] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison Wesley, 1994. URL: `https://www.stroustrup.com/dne.html`.

[32] Liam Tung. Nsa to developers: Think about switching from c and c++ to a memory safe programming language. `https://www.zdnet.com/article/nsa-to-developers-think-about-switching-from-c-and-c-to-a-memory-safe-programming-language/`. Published: Novemmber 11, 2022.

A Gap Algorithm Additional Tables

In this appendix is a collection of tables that provide more information on the experiments that were run using the DFA-Gap algorithm and the regular expression variant.

A.1 DFA-Gap Algorithm Tables

The following two collections of tables show the full range of results for run-times and for energy usage over time, for the DFA-Gap algorithm on all values of k for which it was run.

A.1.1 DFA-Gap Run-times

In table 15 the different rankings by run-time are shown for the DFA-Gap algorithm. The sixth sub-table, table 15f, shows the score resulting from summing all the values from $k = 1$ to $k = 5$ and then scoring them.

This last sub-table acts as an averaging of the results for the different values of k. Interesting in these results are that the different compilers for each of C and C++ appear in the same order of ranking for those languages. Rust's combined performance here was over 69% slower than the fastest, but what is interesting about the performance of Rust is the notable drop in performance for values of k greater than 1. At $k = 1$, Rust was only 50.6% slower than the GCC's C version. But at $k = 3$ it was 76.3%.

Also of interest is how Perl closed the gap on Python as k grew. At $k = 4$ they were within 0.07% of each other and at $k = 5$ Perl had outperformed Python.

A.1.2 DFA-Gap Energy Usage

Here, in table 16, the DFA-Gap energy usage rankings are displayed for the range of k values as well as a combined-values score table.

In a result similar to the run-times tables, one language holds the top spot for all values of k: Rust. This is an interesting observation, as Rust had consistently scored seventh in run-time performance for these same experiments. This shows that Rust was using energy more efficiently overall, despite needing more time to produce the results. This has already been shown in figure 29, in section 5.4.

A.2 Regexp-Gap Algorithm Tables

These tables show the full range of results for run-times and for energy usage over time, for the Regexp-Gap algorithm on all values of k for which it was run.

(a) $k = 1$

Language	Runtime	Score
C (GCC)	69.49	1.0000
C (LLVM)	70.13	1.0092
C++ (GCC)	75.72	1.0896
C++ (LLVM)	77.95	1.1218
C (Intel)	81.65	1.1750
C++ (Intel)	86.37	1.2430
Rust	104.66	1.5061
Python	3213.17	46.2405
Perl	3483.84	50.1357

(b) $k = 2$

Language	Runtime	Score
C (GCC)	83.52	1.0000
C (LLVM)	89.17	1.0677
C++ (GCC)	97.42	1.1665
C (Intel)	99.83	1.1953
C++ (LLVM)	104.92	1.2563
C++ (Intel)	108.28	1.2965
Rust	146.35	1.7523
Python	4335.62	51.9134
Perl	4422.84	52.9577

(c) $k = 3$

Language	Runtime	Score
C (GCC)	100.45	1.0000
C (LLVM)	107.97	1.0749
C (Intel)	116.13	1.1561
C++ (GCC)	123.25	1.2270
C++ (LLVM)	133.10	1.3250
C++ (Intel)	135.04	1.3443
Rust	177.11	1.7632
Perl	5647.35	56.2209
Python	5770.18	57.4437

(d) $k = 4$

Language	Runtime	Score
C (GCC)	119.46	1.0000
C (LLVM)	130.30	1.0907
C (Intel)	137.39	1.1501
C++ (GCC)	151.35	1.2670
C++ (Intel)	162.98	1.3643
C++ (LLVM)	164.51	1.3771
Rust	206.29	1.7268
Perl	6941.50	58.1072
Python	7361.39	61.6221

(e) $k = 5$

Language	Runtime	Score
C (GCC)	140.60	1.0000
C (LLVM)	151.92	1.0805
C (Intel)	155.26	1.1043
C++ (GCC)	183.42	1.3045
C++ (Intel)	191.17	1.3597
C++ (LLVM)	193.67	1.3775
Rust	235.66	1.6761
Perl	8354.58	59.4222
Python	9018.86	64.1469

(f) Combined k

Language	Runtime	Score
C (GCC)	513.51	1.0000
C (LLVM)	549.48	1.0700
C (Intel)	590.26	1.1495
C++ (GCC)	631.15	1.2291
C++ (LLVM)	674.15	1.3128
C++ (Intel)	683.84	1.3317
Rust	870.06	1.6943
Perl	28850.11	56.1820
Python	29699.22	57.8356

Table 15: Comparative run-times of DFA-Gap by value of k

A.2.1 Regexp-Gap Run-times

Table 17 lays out the run-times of the regular expression variant of the gap algorithm. As with the corresponding collection of tables for the DFA-Gap experiments, a single language dominates the top ranking for all values of k. This time, however, the language was Rust. Also of interest is the fact that C++ in all three toolchain variants outperformed all variants

(a) $k = 1$

Language	Energy	Score
Rust	7.94	1.0000
C (LLVM)	8.78	1.1054
C (GCC)	8.87	1.1171
C++ (LLVM)	8.88	1.1189
C++ (Intel)	8.98	1.1316
C++ (GCC)	9.12	1.1483
C (Intel)	9.19	1.1569
Python	9.66	1.2171
Perl	10.78	1.3575

(b) $k = 2$

Language	Energy	Score
Rust	7.69	1.0000
C (LLVM)	8.73	1.1359
C++ (LLVM)	8.81	1.1459
C++ (Intel)	9.01	1.1718
C (GCC)	9.04	1.1760
C (Intel)	9.09	1.1819
C++ (GCC)	9.25	1.2032
Python	9.74	1.2666
Perl	10.78	1.4023

(c) $k = 3$

Language	Energy	Score
Rust	7.35	1.0000
C++ (LLVM)	8.53	1.1603
C (LLVM)	8.68	1.1812
C++ (Intel)	8.87	1.2078
C (Intel)	8.99	1.2231
C (GCC)	9.08	1.2352
C++ (GCC)	9.18	1.2497
Python	9.65	1.3140
Perl	10.21	1.3897

(d) $k = 4$

Language	Energy	Score
Rust	7.13	1.0000
C++ (LLVM)	8.21	1.1515
C (LLVM)	8.54	1.1977
C++ (Intel)	8.67	1.2155
C (Intel)	8.76	1.2287
C++ (GCC)	8.96	1.2565
C (GCC)	9.08	1.2735
Python	9.66	1.3548
Perl	9.69	1.3595

(e) $k = 5$

Language	Energy	Score
Rust	7.08	1.0000
C++ (LLVM)	7.90	1.1150
C (LLVM)	8.38	1.1830
C++ (Intel)	8.45	1.1923
C (Intel)	8.55	1.2066
C++ (GCC)	8.80	1.2421
C (GCC)	9.03	1.2752
Python	9.64	1.3607
Perl	9.72	1.3715

(f) Combined k

Language	Energy	Score
Rust	37.19	1.0000
C++ (LLVM)	42.33	1.1382
C (LLVM)	43.11	1.1592
C++ (Intel)	43.98	1.1826
C (Intel)	44.57	1.1984
C (GCC)	45.10	1.2127
C++ (GCC)	45.31	1.2183
Python	48.36	1.3002
Perl	51.18	1.3762

Table 16: Comparative energy usage by DFA-Gap by value of k

of C.

What makes this of such interest is that fact that all three compiled languages were using the same regular expression engine, PCRE2. As PCRE2 is written in C, one might expect the C versions of this experiment to perform the best. Rust and C++ were both using wrappers around the C interface, yet performed better.

(a) $k = 1$

Language	Runtime	Score
Rust	177.14	1.0000
C++ (Intel)	179.24	1.0119
C++ (GCC)	179.92	1.0157
C++ (LLVM)	180.14	1.0170
C (GCC)	185.43	1.0468
C (Intel)	185.56	1.0475
C (LLVM)	186.05	1.0503
Python	280.86	1.5856
Perl	709.58	4.0058

(b) $k = 2$

Language	Runtime	Score
Rust	213.93	1.0000
C++ (Intel)	225.38	1.0535
C++ (LLVM)	226.29	1.0578
C++ (GCC)	226.63	1.0594
C (Intel)	238.24	1.1136
C (GCC)	238.28	1.1138
C (LLVM)	238.53	1.1150
Python	348.87	1.6308
Perl	850.86	3.9773

(c) $k = 3$

Language	Runtime	Score
Rust	261.48	1.0000
C++ (LLVM)	307.16	1.1747
C++ (GCC)	307.32	1.1753
C++ (Intel)	308.40	1.1794
C (Intel)	336.54	1.2871
C (GCC)	336.67	1.2876
C (LLVM)	337.00	1.2888
Python	454.93	1.7398
Perl	1045.03	3.9966

(d) $k = 4$

Language	Runtime	Score
Rust	322.26	1.0000
C++ (LLVM)	427.24	1.3258
C++ (GCC)	429.11	1.3316
C++ (Intel)	430.85	1.3370
C (Intel)	488.60	1.5162
C (LLVM)	488.88	1.5170
C (GCC)	491.07	1.5238
Python	596.54	1.8511
Perl	1293.45	4.0137

(e) $k = 5$

Language	Runtime	Score
Rust	393.83	1.0000
C++ (LLVM)	582.75	1.4797
C++ (GCC)	583.64	1.4820
C++ (Intel)	585.44	1.4865
C (Intel)	686.90	1.7442
C (GCC)	687.05	1.7445
C (LLVM)	688.79	1.7489
Python	743.90	1.8889
Perl	1582.51	4.0183

(f) Combined k

Language	Runtime	Score
Rust	1368.63	1.0000
C++ (LLVM)	1723.58	1.2593
C++ (GCC)	1726.63	1.2616
C++ (Intel)	1729.31	1.2635
C (Intel)	1935.84	1.4144
C (GCC)	1938.50	1.4164
C (LLVM)	1939.25	1.4169
Python	2425.10	1.7719
Perl	5481.44	4.0050

Table 17: Comparative run-times of Regexp-Gap by value of k

Also worthy of note is the relative closeness of the Perl and Python run-times to the compiled language values. This was less surprising, given that both languages' communities have invested time in the performance of their respective regular expression engines. While Perl was still nearly 4x the run-time of Rust in table 17f, that is a great difference over the 56x difference against GCC's C in table 15f.

A.2.2 Regexp-Gap Energy Usage

Lastly, table 18 presents the energy usage score for the regular expression variant experiments. Of all the energy comparisons, these are the closest in relation to each other. This would indicate that the regular expression engines of Perl and Python, as well as PCRE2, bore the considerable majority of the work in these experiment runs.

(a) $k = 1$

Language	Energy	Score
Perl	10.76	1.0000
C (LLVM)	10.82	1.0056
C (Intel)	10.82	1.0057
C (GCC)	10.84	1.0078
C++ (Intel)	10.86	1.0093
C++ (LLVM)	10.87	1.0102
C++ (GCC)	10.90	1.0134
Python	11.00	1.0228
Rust	11.02	1.0247

(b) $k = 2$

Language	Energy	Score
C (Intel)	10.59	1.0000
C (LLVM)	10.63	1.0039
Perl	10.66	1.0063
C (GCC)	10.67	1.0071
C++ (Intel)	10.75	1.0153
C++ (GCC)	10.78	1.0176
C++ (LLVM)	10.78	1.0178
Rust	10.80	1.0194
Python	10.91	1.0302

(c) $k = 3$

Language	Energy	Score
C (Intel)	10.15	1.0000
C (GCC)	10.21	1.0055
C (LLVM)	10.25	1.0094
C++ (LLVM)	10.59	1.0428
C++ (GCC)	10.62	1.0460
C++ (Intel)	10.63	1.0473
Perl	10.67	1.0513
Rust	10.72	1.0561
Python	10.78	1.0618

(d) $k = 4$

Language	Energy	Score
C (Intel)	9.96	1.0000
C (LLVM)	9.97	1.0012
C (GCC)	10.00	1.0047
C++ (GCC)	10.50	1.0544
C++ (Intel)	10.52	1.0567
C++ (LLVM)	10.52	1.0569
Rust	10.60	1.0647
Perl	10.61	1.0653
Python	10.69	1.0733

(e) $k = 5$

Language	Energy	Score
C (Intel)	9.75	1.0000
C (LLVM)	9.76	1.0012
C (GCC)	9.81	1.0060
C++ (Intel)	10.45	1.0716
C++ (LLVM)	10.46	1.0722
C++ (GCC)	10.46	1.0725
Rust	10.49	1.0751
Perl	10.55	1.0817
Python	10.68	1.0955

(f) Combined k

Language	Energy	Score
C (Intel)	51.27	1.0000
C (LLVM)	51.43	1.0031
C (GCC)	51.53	1.0051
C++ (LLVM)	53.22	1.0379
C++ (Intel)	53.22	1.0379
Perl	53.25	1.0385
C++ (GCC)	53.26	1.0387
Rust	53.63	1.0460
Python	54.07	1.0545

Table 18: Comparative energy usage by Regexp-Gap by value of k

Noteworthy results here include the presence of Perl in the first position for $k = 1$, and never placing lower than fourth. Rust, which had shown strong energy performance for the DFA-Gap experiments, here places last three of the five values of k. However, it is almost always within a 1% difference of the language just ahead of it. This was the case with all of the scoring tables in this grouping; gaps between adjacent scores were rarely more than 1% in difference and sometimes as little as 0.12%.

B Gap Algorithm Additional Graphs

In this appendix the run-time performance of the DFA-Gap algorithm and the regular expression variant will be explored further. The values for Perl and Python are omitted from these graphs due to their out-sized values.

B.1 DFA-Gap Run-time Progression

Figure 25 showed the run-times for the DFA version, with each language "stacking" their times for side-by-side comparison. Here, the increase in run-time as k increases is plotted as a different view of that data.

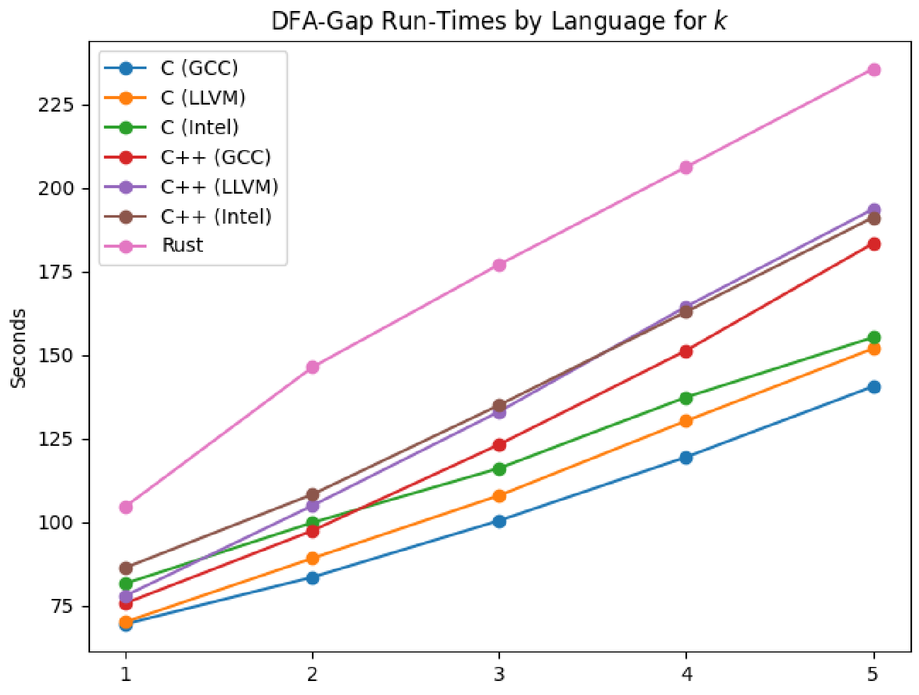


Figure 32: Plots of run-times for DFA-Gap by values of k

The seven plots are all distinguishable from each other, with the GCC C line (blue) being the lowest and the Rust (pink) line the highest. Between these two, the remaining C varieties and all three C++ varieties have very similar slopes (except for Intel C, green, which is increasing more slowly than the others, and might potentially cross the GCC C line around $k = 7$ were the experiments taken that far).

What is most interesting in this set of plots is that each plot itself is functionally linear. The initial impression of this algorithm was that it would most likely show a time-complexity close to $O((m+k)n)$, and indeed in section 3.6 a stated goal of the regular expression variant was to keep within that range. Given that the experiment data was very uniform in both the sequence lengths ("n") and the pattern lengths ("m"), it is hard to extrapolate from this data whether this complexity estimate is accurate.

Further study of this algorithm that includes greater variety in pattern-length and sequence-length would be of great benefit to determining whether the complexity is truly $O((m+k)n)$, closer to the basic $O(mn)$, or something closer to $O(kmn)$.

B.2 Regexp-Gap Run-time Progression

For the regular expression version, figure 26 originally showed the stacked impression of the run-times by language. The following graph shows the same data as above, with each line representing a language and the values growing to the right with increasing k.

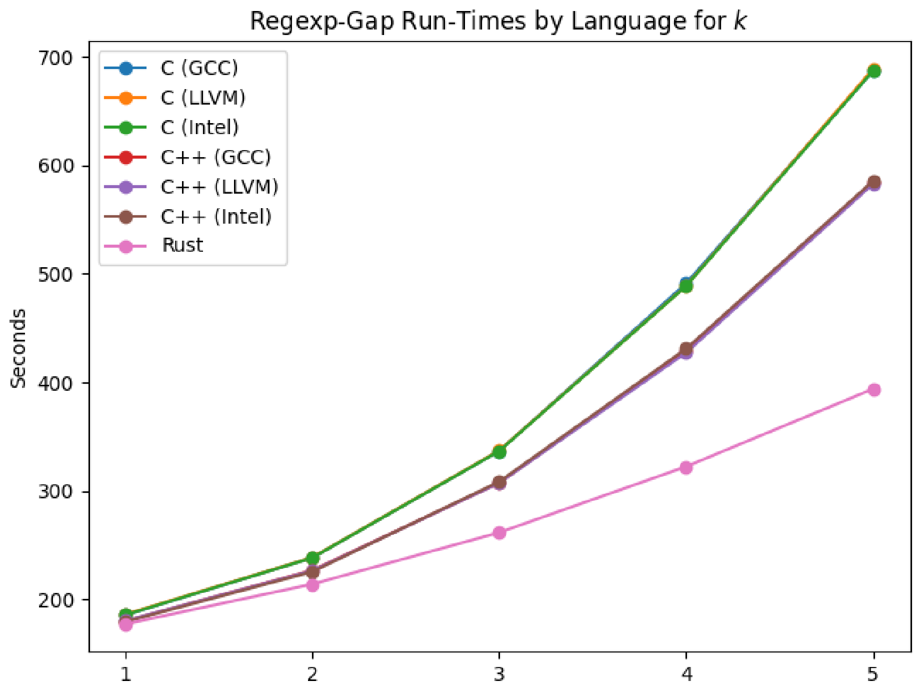


Figure 33: Plots of run-times for Regexp-Gap by values of k

In this set of plots the performance of Rust is clear as the bottom-most (pink) line. What appears to be two additional lines, however, is actually two groups of nearly-identical times. The top-most plot is actually the lines for LLVM C and Intel C almost completely overlapping. And the middle line is the overlapping of GCC C and all three C++ varieties.

This is actually rather expected in terms of results, as all of these experiments were just basic wrappers around the PCRE2 engine. It is the performance of Rust that is the most noteworthy, as it appears at first glance to be almost linear while the C and C++ plots are clearly parabolic. However, the Rust curve does follow a very shallow quadratic polynomial path.

Further study here could also focus on varying lengths of sequences and patterns, but also look into how the value of k influences the curve itself.

C Detailed Results

This appendix provides extended versions of the tables from the collections in table 13 and table 14.

Language	Runtime	Expressiveness	Energy	Unit vector length	Score
C++ (GCC)	1.0561	1.1565	1.1435	0.3079	1.0000
C++ (LLVM)	1.0694	1.1565	1.1349	0.3080	1.0001
C++ (Intel)	1.0766	1.1565	1.1551	0.3081	1.0007
Rust	1.0000	1.4086	1.0000	0.3713	1.2058
C (GCC)	1.0947	1.4503	1.1544	0.3832	1.2446
C (LLVM)	1.1056	1.4503	1.1525	0.3833	1.2447
C (Intel)	1.1226	1.4503	1.1734	0.3835	1.2454
Python	15.7236	1.0000	16.4127	0.9915	3.2200
Perl	16.4693	1.2109	18.0492	1.0741	3.4881

Table 19: Full data for table 13a: final scores, by scale with complexity data

Language	Runtime	Expressiveness	Energy	Unit vector length	Score
C++ (GCC)	1.0561	1.2159	1.1435	0.3261	1.0000
C++ (LLVM)	1.0694	1.2159	1.1349	0.3262	1.0001
C++ (Intel)	1.0766	1.2159	1.1551	0.3263	1.0006
Rust	1.0000	1.3446	1.0000	0.3583	1.0987
C (GCC)	1.0947	1.4070	1.1544	0.3757	1.1521
C (LLVM)	1.1056	1.4070	1.1525	0.3758	1.1523
C (Intel)	1.1226	1.4070	1.1734	0.3760	1.1529
Python	15.7236	1.0000	16.4127	0.9922	3.0424
Perl	16.4693	1.1406	18.0492	1.0697	3.2800

Table 20: Full data for table 13b: final scores, by scale without complexity data

Language	Runtime	Expressiveness	Energy	Unit vector length	Score
C++ (GCC)	2	2	3	4.1231	1.0000
C++ (LLVM)	3	2	2	4.1231	1.0000
Rust	1	6	1	6.1644	1.4951
C++ (Intel)	4	2	6	7.4833	1.8150
C (GCC)	5	7	5	9.9499	2.4132
C (LLVM)	6	7	4	10.0499	2.4375
Python	8	1	8	11.3578	2.7547
C (Intel)	7	7	7	12.1244	2.9406
Perl	9	5	9	13.6748	3.3166

Table 21: Full data for table 13c: final scores, by rank with complexity data

Language	Runtime	Expressiveness	Energy	Unit vector length	Score
C++ (GCC)	2	3	3	4.6904	1.0000
C++ (LLVM)	3	3	2	4.6904	1.0000
Rust	1	6	1	6.1644	1.3143
C++ (Intel)	4	3	6	7.8102	1.6652
C (GCC)	5	7	5	9.9499	2.1213
C (LLVM)	6	7	4	10.0499	2.1426
Python	8	1	8	11.3578	2.4215
C (Intel)	7	7	7	12.1244	2.5849
Perl	9	2	9	12.8841	2.7469

Table 22: Full data for table 13d: final scores, by rank without complexity data

Language	Runtime	Expressiveness	Energy	Vector length	Score
C++	2	2	2	3.4641	1.0000
Rust	1	4	1	4.2426	1.2247
Python	4	1	4	5.7446	1.6583
C	3	5	3	6.5574	1.8930
Perl	5	3	5	7.6811	2.2174

Table 23: Full data for table 14a: distinct languages, by rank with complexity data

Language	Runtime	Expressiveness	Energy	Vector length	Score
C++	2	3	2	4.1231	1.0000
Rust	1	4	1	4.2426	1.0290
Python	4	1	4	5.7446	1.3933
C	3	5	3	6.5574	1.5904
Perl	5	2	5	7.3485	1.7823

Table 24: Full data for table 14b: distinct languages, by rank without complexity data

D Software Sources

This paper has referred to a number of software tools and utilities. Some of these are standard features of a UNIX/Linux operating system, but many must be installed from external sources. This appendix is meant to provide sources for these additional tools.

Application or tool	Location
Homebrew	`https://brew.sh/`
PCRE2	`https://www.pcre.org/`
PCRE2 for Rust	`https://crates.io/crates/pcre2`
Rust	`https://www.rust-lang.org/`
Rustup	`https://rustup.rs/`
Valgrind	`https://valgrind.org/`
YAML	`https://yaml.org/`
`cloc`	`https://github.com/AlDanial/cloc/`
`countperl`	`https://metacpan.org/dist/Perl-Metrics-Simple`
jPCRE2	`https://github.com/jpcre2/jpcre2`
`lizard`	`https://github.com/terryyin/lizard`
`rust-code-analysis-cli`	`https://github.com/mozilla/rust-code-analysis/`
`sloc`	`https://github.com/flosse/sloc`
`xz`	`https://tukaani.org/xz/`

Table 25: Sources for the additional software tools used in this research