Scout Algorithm for Fast Substring Matching

Anand Natrajan[†]
Germantown MD, USA
anand@anandnatrajan.com

Mallige Anand
Germantown MD, USA
mallige@anandnatrajan.com

ABSTRACT

Exact substring matching is a common task in many software applications. Despite the existence of several algorithms for finding whether or not a pattern string is present in a target string, the most common implementation is a naïve, brute force approach. Alternative approaches either do not provide enough of a benefit for the added complexity, or are impractical for modern character sets, e.g., Unicode. We present a new algorithm, Scout, that is straightforward, quick and appropriate for all applications. We also compare the performance characteristics of the Scout algorithm with several others.

[†]Both authors conducted this work independent of any affiliation to any organisation.

https://doi.org/ ID: CACM-20-10-3795

CCS CONCEPTS

• Theory of computation~Design and analysis of algorithms~Data structures design and analysis~Pattern matching

KEYWORDS

scout, substring matching, exact matching, pattern matching

ACM Reference format:

Anand Natrajan and Mallige Anand. 2020. Scout Algorithm for Fast Substring Matching. Submitted to *Communications of ACM*.

1 Introduction

Many software applications require an exact match of a pattern in a target string. Such matching is simpler than and distinct from matching regular expressions, contextual grammars, search engines, fuzzy matches and other related activities. In this paper, we will focus only on exact matches, also called substring searches. We will present an algorithm, dubbed "Scout", which in most cases performs as well as or better than the best alternative substring search algorithms in existence. The algorithm is simple to implement, performs significantly better than the brute force algorithm most commonly employed, requires no preprocessing, has better memory usage characteristics than the best algorithms, and importantly, works for modern character sets, such as Unicode.

Scout performs at least as well as alternative algorithms, as measured by wall-clock time. Assessments of substring matching algorithms often focus on the number of character comparisons as the primary metric of performance. Our research indicates that memory lookups affect performance at least as much as comparisons. We also considered some subjective measures for comparing various algorithms, for example, the suitability to other languages and non-ASCII character sets.

In the rest of this paper, we will present the Scout algorithm, and contrast it briefly with other substring matching algorithms. Of the dozens of algorithms available, we will select some exemplar algorithms for further comparison. We will present the results of performance comparisons for various testbeds. We will briefly examine factors that drive performance, and attempt to tease out the characteristics of the exemplars to explain their performance. In the interests of brevity, we will not present detailed tabular data and charts. We expect to submit implementations of our code to open-source repositories for inclusion in language libraries.

2 Algorithm Description

The central idea in the Scout algorithm is to identify a "scout" character quickly. This character is dispatched to obtain information, in this case, an appropriate location in the target string for a deeper match. Let us denote a pattern string p as containing m characters, and a target string t as containing n characters. The scout is always a character from the pattern string p, and we use it to find an appropriate location in the target string t.

Given p and t, we begin the algorithm by comparing the first characters of each. On a match, we move on to the next characters of each, and so on, sequentially searching, much like the brute force approach. On a mismatch, Scout diverges from brute force. In brute force, we would re-initiate a sequential search starting from the next character of t. In other words we would "slide" the pattern p one character "to the right" along t. In contrast, in Scout, we designate the currently-mismatched pattern character as a "scout". We compare the scout with the character in t immediately past the mismatch. If those characters are mismatched, we advance along t, and compare the next character to the scout. We proceed in this fashion until the first time we encounter a character in t that matches the scout. If we find no such match, p is not contained in t. If we do, we slide p along t such that the last mismatched character in p (which was the scout's original position) lines up with the scout's current match in t. We then initiate a sequential search of p in t starting from this new position.

An additional check, called a "twin" check can occasionally slide *p* along *t* further. Whenever we initiate a per-character sequential search of *p* in *t*, on a match, we additionally check if the current character in *p* is identical to the scout and precedes it. If so, this character, which we call a "twin", can be used to slide *p* further along until the twin aligns with the scout's position. Whether a slide occurred because of a twin or because of the scout alone, we initiate a sequential search of *p* starting from its current alignment with *t*. During this sequential search, we may encounter another mismatch. This mismatched pattern character is the new scout, which continues the algorithm as before.

The Scout algorithm is provably correct. The proof rests on three lemmata, of which two are obvious. First, sequential search obviously either finds a match, or finds the first mismatched pattern character. Second, the scout character obviously finds the first possible target location where a match could occur, by construction. Third, less obviously, sequential search and scout alignment suffice to find a pattern match if one exists. The twin character alignment is not strictly necessary, but is provably correct, and results in better performance. Put together, the Scout algorithm will find a pattern match if one exists.

3 Related Work

The brute force, double-loop algorithm is the most common one used to find a substring. Since the late 1970s through the early 1990s, several alternative algorithms have been proposed, notably, Knuth-Morris-Pratt [2], Karp-Rabin [3] and Boyer-Moore [1], along with dozens of variants. One of the best-performing algorithms is the Sunday Quick Search algorithm [4], a variant of Boyer-Moore. Although the theoretical average- and worst-case performance of each of these algorithms beats brute force, these algorithms have not found their way into common usage. We believe that there are several criteria other than theoretical performance that hinder their adoption. We will examine those criteria shortly.

Most substring search algorithms fall into a few broad classes. One large class contains algorithms that identify properties of the pattern or target so that portions of the target can be skipped over. This class includes Boyer-Moore and several variants. Most of these algorithms require a preprocessing step. Most of them reserve memory proportional to the size of the pattern or the size of the alphabet used in the pattern and target. Brute force may be considered a degenerate member of this class. Brute force skips characters by the minimum possible value, i.e., one. A second class contains algorithms that compute some heuristic that could result in false positives, but not false negatives. On a positive match for the heuristic, these algorithms rely on a sequential search for disambiguation. This class includes Karp-Rabin and some variants. A third class contains algorithms that use characteristics of the language, e.g., character frequencies, to determine how to navigate the target. In our work, we do not undertake any objective comparisons against algorithms in this third class. Our goal is to find the best possible domain-agnostic, generic substring matching algorithm.

The brute force algorithm is deployed widely. It is the algorithm deployed for String `indexOf` in the Java language library for OpenJDK as of version 9.0.1. The C implementation of `strstr` also uses brute force. In other words, the most common libraries for two of the most popular languages for large-scale applications use the brute force approach for any and all substring matching. The Python implementation of the `find` method uses Sunday Quick Search (although it claims to use Boyer-Moore), but makes some accommodations in order to circumvent the most problematic characteristic of that algorithm, namely the need to store a bad character array whose size is dependent on the character set. The python implementation retains a skip value for only the last character, and uses a low-fidelity Bloom filter to check for the presence of a target character in the pattern. These accommodations result in smaller shifts than the classical Boyer-Moore algorithm, but avoid some of its most problematic characteristics.

Exhaustive comparisons across all of forty or so algorithms and variants based on performance, memory, applicability, simplicity, etc. would be a welcome body of work. Our detailed comparisons will use four algorithms alone: Brute force, Karp-Rabin, Sunday Quick Search and Scout. While we acknowledge the risk involved in limiting our focus to a few algorithms alone, we believe our choices are reasonable because they permit us to perform a deep analysis along our lines of inquiry.

We chose brute force simply because it is a baseline as well as an incumbent algorithm. We chose Karp-Rabin to represent the heuristic class of algorithms. The Boyer-Moore class has the largest number of variants, but it is generally accepted that Sunday Quick Search is the fastest among them. We did not choose language-specific algorithms because our goal has always been to identify domain-agnostic algorithms. Finally, we chose our best implementation of Scout.

4 Comparison of Algorithms

When comparing substring search algorithms, most authors focus on performance, understandably so. Performance often translates to number of character comparisons, whether best, worst or average case. Of course, other metrics matter as well, for example memory usage. However, character comparisons dominate most discussions around performance because character comparisons count towards order complexity, and order complexity remains the gold standard in computer science for evaluating the performance of algorithms. We argue for a broader approach for evaluating substring searching algorithms, without diluting the focus on performance.

Our performance numbers indicate that Scout should be a strong contender for a general-purpose substring searching implementation. As a matter of practical performance, we will show that Scout is one of the fastest algorithms available. While it may score poorly on the count of character comparisons, it scores well on memory lookups. Although traditional measures of performance are based on character comparisons, we believe memory lookups should be given more prominence because the former are far less expensive than the latter. Therefore, an algorithm that reduces memory lookups, be it at the expense of more character comparisons is likely to be faster, as shown by Scout.

Several subjective criteria may influence the choice of a general-purpose substring search algorithm over and above practical performance, preprocessing and memory usage.

- *Character set.* Some algorithms, notably Karp-Rabin and Boyer-Moore, assume an ASCII character set consisting of 256 characters, i.e., characters representable within one byte. Modern Unicode character sets can take 1-4 bytes. These algorithms were formulated when Unicode was not prevalent. With Unicode, they either fail outright, or when upgraded, consume prohibitive amounts of memory.
- *Language specificity.* Some algorithms, notably Sunday Maximal Shift [4], rely on letter frequencies in languages. Obviously, letter frequencies vary by language, making the algorithms harder to generalise. Moreover, letter frequencies in specific subdomains will likely differ, e.g., in genome matching.
- *Simplicity.* The brute force approach is simple to implement, requiring less than 20 lines of formatted code, including boiler-plate. The other approaches, including ours, can be 2-4 times larger. Of course, lines of code is a trivial concern, but conceptual complexity can complicate provability, thus hindering broader acceptance.

In our experiments, we have endeavoured to offer every advantage possible to alternative algorithms. We have faithfully transcribed the implementations available publicly. We made variable names and formatting more readable, which does not affect performance. Most importantly, for algorithms that required memory proportional to the alphabet size, we assumed the alphabet size to be 256 characters. In other words, we permitted these algorithms to function for the ASCII character set alone, although Scout works as-is for Unicode. When we refactored the alternative algorithms faithfully to work for Unicode, they either failed outright or resulted in absurdly high wall-clock times.

5 Methodology and Results

We crafted a suite of tests for assessing substring search performance. Our testbeds and tests, written in Java, were crafted to test several aspects:

1. Obviously, our algorithms as well as the alternatives should function correctly. We measured correctness somewhat more stringently than necessary, by checking for the exact position at which a pattern was found in the target, rather than merely a true/false check. We also checked for Unicode patterns and targets.
2. We varied the depth at which the pattern was found in a fixed target. In other words, we crafted a synthetic target of a fixed size (100 characters), and inserted a fixed-size pattern (5 characters) that was guaranteed to exist in the target. We varied whether the pattern was found at the start (0%), the end (100%), the middle (50%) as well as several other intermediate positions. Specifically, our pattern string was "`aabca`", and our target string was "`xx...xaabcaxx...x`", where the prefix of `x` characters was of length p, and the suffix was of length q. Therefore pattern length $m = 5$, and target length $n = p+q+m = 105$. If $p = 0$, the pattern depth is 0%, but if $q = 0$, the pattern depth is 100%. Using this testbed, we tried to ascertain how algorithmic performance varied from best case (0%) through worst-case (100%).
3. We varied the length of the target string. We crafted increasingly long target strings (from 0 through 10,000 characters), and appended a fixed pattern string (5 characters) at the very end to simulate worst-case behaviour. Specifically, our pattern string was "`aabca`", and our target string was "`xx...xaabca`", where the prefix of `x` characters was of length p. Therefore pattern length $m = 5$, and target length $n = p+m$. We varied p in order to craft longer target strings. We expected all algorithms to grow linearly in wall-clock time as length increased. However, we wanted to observe the slopes of those linear curves for various algorithms.
4. We simulated real-life scenarios by choosing a well-known literary passage (Hamlet's famous soliloquy, Act III, Scene I, "To be, or not to be [...] Be all my sins, remember'd."), and searching for substrings deeper and deeper into this target, again starting from 0% through 100%. This test not only simulated real-life human behaviour, but also tested whether all of the algorithms processed punctuation, spaces and mixed cases correctly. By concatenating all of the lines in the soliloquy, we were able to craft a target string of modest-to-large size (1500 characters). We readily concede the cultural monotony in the choice of this text. While it does not include characters from other languages or even accents, by choosing a well-known piece of English text, we gave all algorithms any implicit language-specific advantage they could claim.

For all of the tests, in our Java implementations, we measured four metrics. One, wall-clock time, with timers surrounding repeated method invocations within loops. Two, comparisons, by instrumenting the code with counters. Three, memory accesses, for array characters in the pattern and target string. In short, we counted one memory access for every pair of [and] brackets in the code, but not for accesses to local or global variables. Four, expensive arithmetic operations, such as multiplication, division, modulo and exponentiation (if any). The instrumentation slows down the raw performance of the algorithms, which is a reason to ignore absolute wall-clock times but pay attention to relative times.

We did not measure memory consumption in our tests, primarily so we could focus on practical performance. Of course, memory consumption matters, and several alternative algorithms store large amounts of global or per-pattern or per-target state prior to the actual search. We have penalised all preprocessing by counting them towards the wall-clock time taken to execute the search. Doing so exempts algorithms that require global preprocessing, e.g., constructing a one-time dictionary, and also ignores the memory consumption of these algorithms. We refer interested readers to surveys that have examined the memory consumption characteristics of alternative algorithms. Of our exemplars, only Sunday Quick Search requires any significant memory. This search algorithm, like most members of the Boyer-Moore class, stores a bad-character integer map whose size is proportional to the size of the character set.

Our tests ran on a MacBook Pro with a 2.9 GHz Intel Core i9, with 32GB RAM, running macOS v10.15.7 (Catalina). The Java version was 9.0.1 (build 9.0.1+11). Our performance results are

shown in the charts below. When plotting wall-clock time, the Y-axes are in nanoseconds. For each data point, we averaged the result of one million runs to smooth out any spurious results. The absolute numbers are irrelevant; only the relative performance matters.

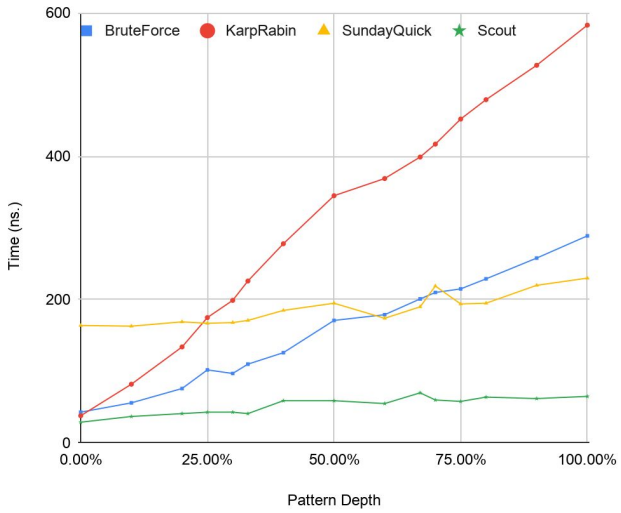

Figure 1: **Wall-clock times (ns.) for different pattern depths in a given target string length (100 characters).**

Scout clearly outperforms all of the other exemplars at most data points in terms of wall-clock time. All of the algorithms display roughly linear performance, but the slopes and intercepts are rather different. For a given target string, as the pattern is situated deeper and deeper (Figure 1), we observe that Scout and Sunday Quick Search have gentler slopes, i.e., the time taken increases slowly as the pattern is found deeper in the target string. However, Sunday Quick Search has a large y-intercept, because of its high preprocessing cost. If the target string is increased in size, with the pattern situated at 100% (Figure 2), all of the algorithms display a linear increase in time. The high preprocessing overhead penalises Sunday Quick Search for small target string lengths, but as the strings elongate, the benefits of that processing become apparent.

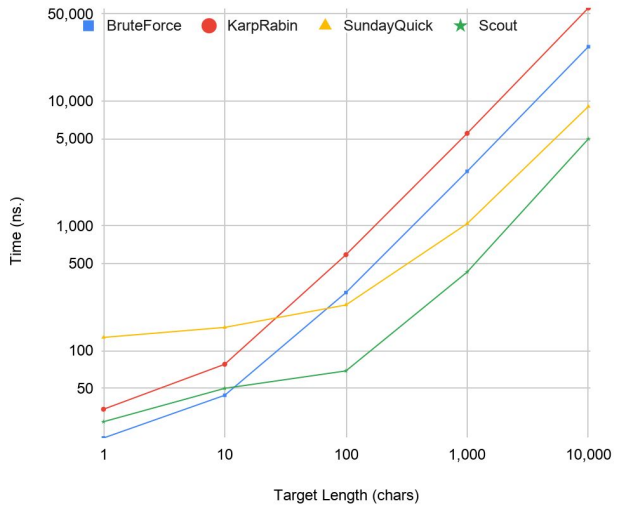

Figure 2: **Wall-clock times (ns.) for different target string lengths with a given pattern depth (100%). Log scales.**

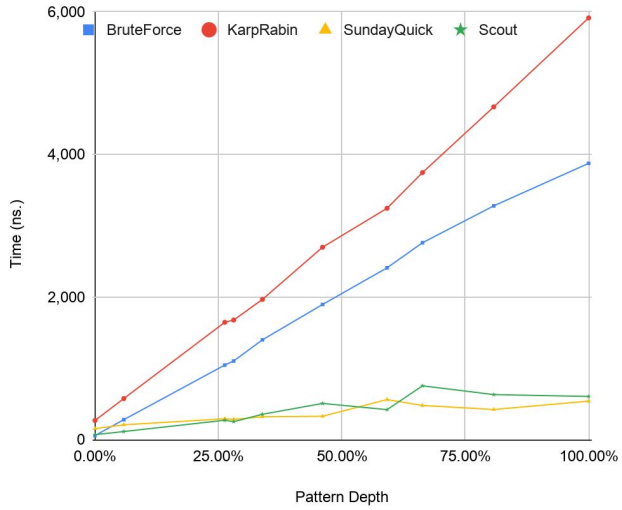

Figure 3: **Wall-clock times (ns.) for different pattern depths in Hamlet's soliloquy (1500 characters).**

6 Discussion of Performance

When comparing alternative algorithms, we checked our implementations of each for correctness. One immediate conundrum was that several of the algorithms, in particular all of the Boyer-Moore variants, tolerated only ASCII characters. When we tested them against Unicode characters, the algorithms failed,e.g., dumped a stack trace. The ASCII-centrism of these algorithms forced us to choose the Hamlet soliloquy as a real-life

testbed. In contrast, Scout runs correctly, as-is, whether the passage is from Hamlet in English, from the Bhagavad Gita in Sanskrit, the Iliad in Greek, or the Tao Te Ching in Chinese.

The memory consumption of alternative algorithms is a significant concern as well. The Boyer-Moore variants require one or more integer arrays with one entry per ASCII character. We could have re-sized this array to have one entry per Unicode character, but the prospect of allocating and initialising an array containing up to 2^{32} entries for every search was daunting. In the end, we left the alternative algorithms as-is in terms of their ASCII-centric memory consumption so as to give them an advantage.

The point of evaluating performance is to assess whether one algorithm is speedier than another, especially at scale. Turning to time taken as a metric of performance, we recognise that while character comparisons certainly contribute towards time taken, they are not as dominant as other factors. Several algorithms undertake preprocessing steps in order to speed up subsequent searching. Technically, the work done in those preprocessing steps does not count towards the number of character comparisons. However, general-purpose substring searching is likely to have low reuse, i.e., it is likely that every time the algorithm is invoked, the pattern and target may change. Therefore, algorithms that preprocess either the pattern or target or both have to undertake those steps for every single invocation. Preprocessing consumes time, whether or not the character comparison counts improve.

Yet other algorithms involve expensive operations, such as modulo arithmetic in Karp-Rabin. On most microcomputers, these operations are far more expensive than character counts. Certainly, some microcomputers may come equipped with separate processing units to speed up arithmetic calculations, but such processing is still likely to be more expensive than character comparisons.

Memory lookups affect performance. On most modern microcomputers, memory lookups can take several more processor cycles than character comparisons. Therefore, reducing the number of memory lookups can reduce time taken to execute an algorithm. Memory lookups play another role in performance when spatial and temporal locality are taken into account. Techniques such as paging and multi-level caching have been developed to reduce the cost of memory lookups. While we do not expect to craft algorithms to account for paging or caching techniques on specific machines, an algorithm that naturally exploits basic paging and caching will perform better than an equivalent algorithm that does not.

We have chosen to normalise all of these factors by comparing wall-clock time. The question of how to account for all of the key factors is challenging. Counting character comparisons alone can seem myopic especially if a low count comes at the expense of too many memory lookups or costly arithmetic operations. Loading up most of the work in a preprocessing step so as to make the search cheap is counter-productive. Deep textual analysis may reduce character comparisons dramatically, but may not lend themselves to practical solutions. By focusing on wall-clock time, we force a normalisation of all of these disparate factors into a common, comparable and practical currency.

Wall-clock time as a common currency is not flawless. Wall-clock times for the same tests run on different machines, operating systems, compilers, loads, etc. will differ. Therefore, only relative wall-clock times lead to meaningful comparisons. More particularly, the wall-clock times have to be compared for the same tests on the same machines, under the same load conditions. Another issue with wall-clock times is that underlying performance trends and patterns are not readily apparent unless several data points are plotted and examined. A pen-and-paper analysis of an algorithm may suggest its performance grows linearly, but a wall-clock time plot may reveal large start-up costs that are not amortised well enough to make the linear growth apparent until extremely large data sets are chosen. Alternatively, another seemingly linear-growth algorithm may be revealed to encounter page thrashing, which causes performance to deteriorate for large data sets.

In the next few charts, we show the results of instrumenting our Java code to count character comparisons as well as memory lookups. We will consider the "real-life" Hamlet testbed for our discussion below.

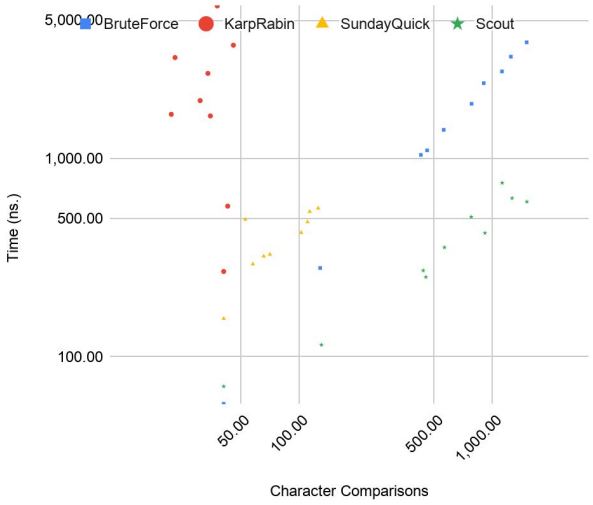

Figure 4: **Wall-clock times (ns) at different counts of character comparisons for exemplar algorithms.**

The scatter-plot in Figure 4 shows that wall-clock time is loosely linearly correlated with the number of character comparisons. We make two observations from this plot. One, when considering a particular algorithm alone, sometimes the correlation is tight (e.g., for Brute force), and sometimes loose (e.g., for Karp-Rabin). Two, when considering all of the algorithms put together, the correlation is extremely weak. In contrast, the scatter-plot in Figure 5 shows a much tighter correlation in both cases between wall-clock time and memory lookups. Of course, character comparisons are correlated to memory lookups; the point of accessing pattern and target characters is to facilitate a comparison. However, whether considering a specific algorithm, or looking at all algorithms,

memory lookups is a much better predictor of wall-clock time performance than character comparison counts.

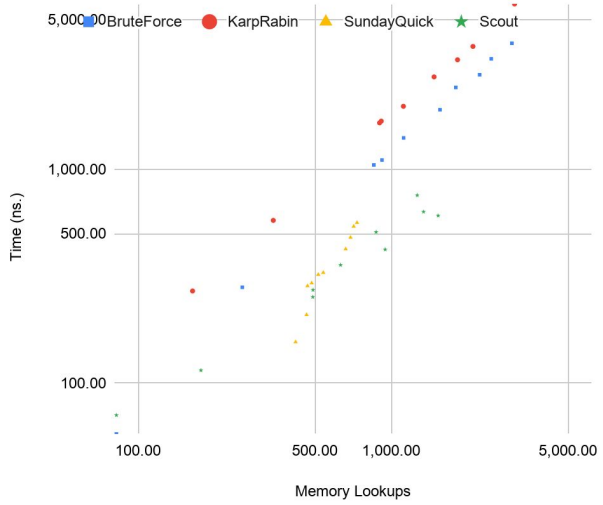

Figure 5: **Wall-clock times (ns) at different counts of memory lookups for exemplar algorithms.**

Another key performance factor is spatial and temporal locality. An algorithm that accesses the same characters in rapid succession, or accesses nearby characters quickly, lends itself to better caching performance. We did not measure the effect of caching on any of the algorithms. However, an inspection of the code for the algorithms reveals that Scout shows good spatial and temporal locality. In contrast, Sunday Quick Search, despite performing fewer memory lookups, shows poor locality. In particular, as with every Boyer-Moore variant, it depends on the bad character array, which is simply another array of integers different from the pattern and the target, and which is indexed and accessed by the ASCII value of the currently-mismatched target character. The access pattern for the bad character array is effectively random, leading to poor locality. Although we were not able to quantify the effects of caching, we encourage further research into how caching can improve the performance of substring matching.

7 Conclusions

Searching for a substring is so routine a task that most languages provide in-built libraries that software developers can reuse. Many of these libraries implement a brute force algorithm to search for a substring despite there being dozens of alternative algorithms. We submit that these library methods should be rewritten to implement better algorithms. We provide an algorithm, Scout, whose performance as measured by wall-clock time ranks among the fastest algorithms. Additionally, Scout requires no preprocessing and has low, constant memory usage. It is language-agnostic and works for any character set.

Scout runs faster than alternative algorithms in most cases. We crafted testbeds so that we could compare Scout against the best alternatives. We picked brute force and the best variants as exemplars for comparison. When performing our comparisons, we endeavoured to give every advantage to each exemplar, so as to make the performance comparisons salient.

The usual metric of performance for substring matches, i.e., character comparisons, is a weak predictor of wall-clock time. Our results show that memory lookups predict performance better. We showed that merely counting memory accesses correlated better with observed wall-clock time than comparison counts. We speculate that caching effects because of spatial and temporal locality explain the remaining memory-related performance.

We encourage popular implementations of substring libraries to use our Scout algorithm.

ACKNOWLEDGEMENTS

We thank several colleagues who critiqued earlier versions of the algorithm, and read longer versions of this paper.

REFERENCES

[1] R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20:762-772, 1977.
[2] D.E. Knuth, J.H. Morris (Jr) and V.R. Pratt. Fast pattern matching in strings, *SIAM Journal on Computing*, 6(1):323-350, 1977.
[3] R.M. Karp and M.O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Res. Dev.*, 31(2):249-260, 1987.
[4] D.M. Sunday. A very fast substring search algorithm, *Communications of the ACM*, 33(8):132-142, 1990.