String Search

Cassidy All

February 20, 2025

1 Introduction

Pattern-matching between two strings is a common computational problem. For example, consider using Cmd+F or Ctrl+F to search for text on a page. Many tasks in computational genomics may be formulated as string searches. For example, trying to establish specific genetic markers for particular traits or diseases requires comparing the DNA of many individuals to look for shared patterns. In the worst case, string search requires individual comparisons between a significant number of elements that, in the biological context, could be in the billions (ex. a reference genome). Thus, we naturally want to reduce the number of comparisons we attempt as much as possible—while still being guaranteed to find relevant matches. We analyze the speed and memory usage of two algorithms commonly used in string search: naïve search and Boyer-Moore search. Naïve search uses two pointers and consists of two steps: (1) iterating through the given text T and (2) iterating through a given pattern P. The crucial insight of the Boyer-Moore algorithm is finding ways to iterate through T more quickly using **only** knowledge of of P, given that, in many cases—especially the biological context—T is too large to pre-process. These algorithms are described in (slightly) more detail in Section 3.2 below. We found that, as pattern size grows, Boyer-Moore quickly becomes faster, even at relatively modest pattern sizes and exhibits a linear relationship with the size of the text. This is consistent with our theoretical analysis. Our empirical results found that Naïve search is competitive when both pattern size and text size are small. Thus, for almost all computational genomics tasks, Boyer-Moore is faster. However, this comes with the trade-off of consuming more memory as pattern size increases, because we have to create the bad character and good suffix tables. We also note in our theoretical analysis worst-case scenarios where Boyer-Moore may perform particularly poorly, such as when a text and pattern are very similar.

2 Results

Naïve search was uniformly faster only when $|P| = 1$, character search, an effectively different problem (Figures 1 & 2). It is already out-competed at $|P| = 3$, even when text size is small (Figures 3 & 4). Although both Naïve search and Boyer-Moore search exhibit a linear relationship between runtime and text size, it scales especially quickly for Naïve search, and very modestly for Boyer-Moore. Boyer-Moore runtime decreased as pattern size increased, although these returns were diminishing (most evident in Figures 2, 4, 6, & 8). At the same time, memory usage increased almost uniformly. The runtime of naïve search actually increased as pattern size grew larger, although memory usage remained constant. These results are largely consistent with our theoretical comparison.

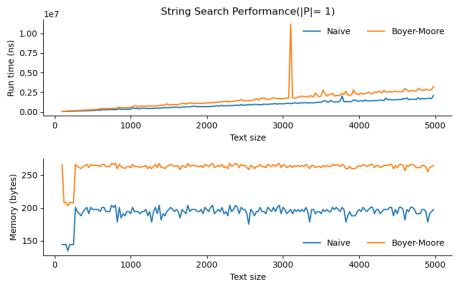

Figure 1: $|P| = 1$, T Large

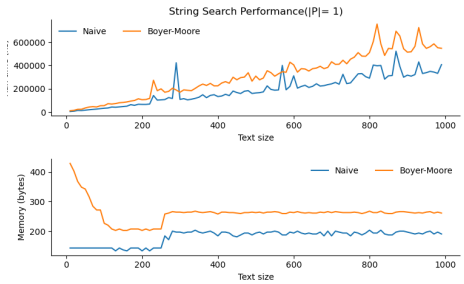

Figure 2: $|P| = 1$, T Small

Figure 3: $|P| = 3$, T Large

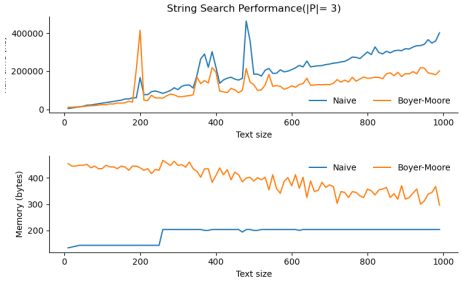

Figure 4: $|P| = 3$, T Small

Figure 5: $|P| = 5$, T Large

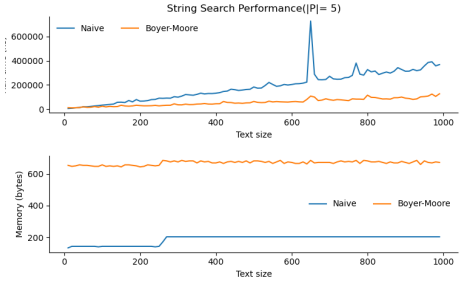

Figure 6: $|P| = 5$, T Small

Figure 7: $|P| = 7$, T Large

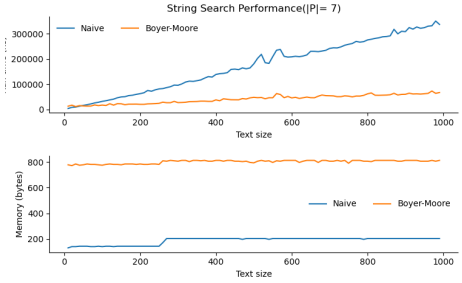

Figure 8: $|P| = 7$, T Small

3 Methods

3.1 Empirical Comparison

We computed the run time (nanoseconds) and memory usage (bytes) for ten simulations of naïve search and an implementation of Boyer-Moore search, and report their average. For each simulation, we randomly generated a string T of a specified length from the alphabet $\Sigma := \{A, C, T, G\}$ and chose a random sub-string P of a given pattern length. We tested 196 values for the length of T in the range [100, 5000] with a step size of 25. We also produced figures for 98 values of T in the range [10, 1000] with a step size of 10 (the "small" figures). We tested 4 pattern sizes in the range [1, 7] with a step size of two. The sub-string P acted as our search target.

3.2 Theoretical Comparison

In naïve search, each character in the reference string T is compared to a pointer iterating through the pattern string P. If a match is found, we note it and continue iterating through T until the number of remaining characters is less than $|P|$. Thus, we have a straightforward analysis for time complexity. In the worst-case, we have a string like "AAAAAA" and a pattern *"AAB"*, and always iterate over the entirety of the pattern. Worst-case time complexity is given by $((n - m + 1) \cdot m)$ where $n = |T|$ and $m = |P|$. When $m \propto n$, this is $\mathcal{O}(n^2)$, usually it is $\mathcal{O}(n \cdot m)$. In the best case, we can quickly rule out matches. For example, given a string *"AAAAAA"* and pattern *"BB"* we will only ever have to do a single comparison for each element of T. Thus, best-case time complexity is given by $(n - m + 1)$, or $\mathcal{O}(n)$. In the ideal case, if $m \propto n$, we may have $\mathcal{O}(1)$, although this is never reasonable in practical applications.

In Boyer-Moore search, we start at alignment $k = m$, and compare P to T, moving right to left. Either we determine a match or a mismatch occurs, at which point we shift k by the maximum value from either the bad character or good suffix table. Thus, our time complexity depends on whether shifts are given by the bad character table or the good suffix table. Considering only the bad character heuristic, the worst case is when the text and pattern are identical. In this case, just as with naïve search, worst case time complexity is given by $((n-m+1) \cdot m)$, which practically evaluates as $\mathcal{O}(n \cdot m)$. But, in the best case, such as when the text and pattern are completely unique, we only make $\frac{n}{m}$ comparisons. Thus,the best-case time complexity is $\mathcal{O}(\frac{n}{m})$. Now, we consider only the good suffix heuristic. In the worst case, when a string and pattern are very similar except for the last character of the pattern, the good suffix table returns a maximum shift of one. So, our worst-case time complexity is the same as for the Boyer-Moore algorithm with the bad character heuristic. But, in the best case, the good suffix heuristic enables larger shifts. So, the best-case time complexity is given by $\mathcal{O}(\frac{n}{2m})$. We note (in Figures 9 & 10 below) that this best-case scenario significantly increases the speed of Boyer-Moore as pattern size increases. Although a theoretically rigorous average-case time complexity would be difficult to develop given the number of factors impacting the speed of the combined Boyer-Moore algorithm, it is reasonable to think that, in most cases, it is around $\mathcal{O}(n)$.

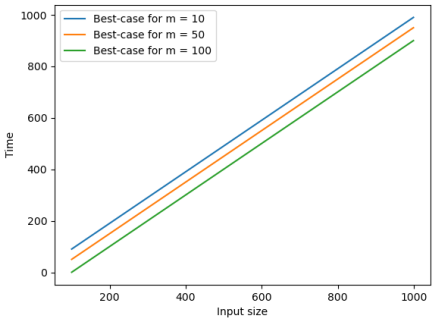

Figure 9: Naïve Search

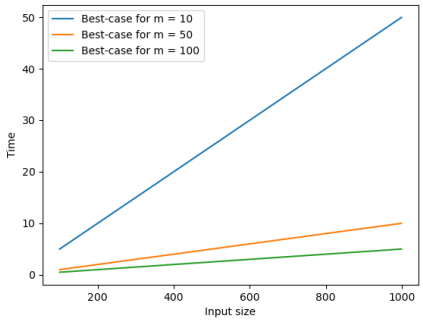

Figure 10: Boyer-Moore Search

3.3 Reproducibility

To replicate these experiments, clone the repository and then run the empirical and simulation experiments as follows:

```
$ clone https://github.com/cu-compg-spring-2025/assignment-4-string-search-cassitude
$ # adjust text_range and pattern_size as needed
$ python src/string_search.py \
        --text_range 100 5000 25 \
        --pattern_size 9 \
        --rounds 10 \
        --out_file search_9.png
# simulation results
$ python src/sim.py
```

4 Conclusion

Our empirical and theoretical analysis tend to agree that, especially given the conditions of computational genomics research (varying pattern sizes, large reference genomes)—Boyer-Moore is almost always a better choice for string search than naïve search. However, we did identify some worst-case scenarios for the time complexity of Boyer-Moore which we did not have the opportunity to test being empirically. That being said, the worst-case time complexity of Boyer-Moore is identical to the worst-case time complexity of naïve search and they have similar worst-case conditions—similar patterns and texts. Future work could explore this empirically.