Space-time Tradeoff for Position Shortest Unique Substring

Gary Hoppenworth

September 2018

Abstract

In this paper I give algorithms for a more fine-grained version of the Range Shortest Unique Substring Problem explored in Abedin et al. 2019. Given a text $T[1..n]$ with an alphabet of size σ and a position $p \in [1, n]$, the shortest unique substring of T covering p, denoted by S_p, is the substring $T[i, j]$ of T such that $i \leq p \leq j$, $T[i..j]$ is unique in T, and $j - i$ is minimized. Let τ be a tradeoff parameter such that $1 \leq \tau \leq n$. I present the following new results. For any text $T[1..n]$ and a position $p \in [1, n]$ chosen at runtime, we can compute S_p using a randomized algorithm in $\mathcal{O}(n\tau \log n)$ time and $\mathcal{O}(n/\tau)$ space. This algorithm is correct with high probability (w.h.p.) and makes use of Rabin-Karp pattern matching an universal hashes.

Introduction

In previous work the shortest unique substring problem covering a specific range of text was studied. In this problem we are given a range $[a, b]$ of the text T, and the task is to find the shortest *unique* substring of $T[a, b]$ that was unique in T Abedin et al. 2019. This problem is of interest in computational biology and information retrieval. In this paper I expand on this line of research by studying the shortest unique substring problem covering a *position* p. Sometimes only a specific position p of the text is of interest. Furthermore, in computational biology in particular, there can be both space and time constraints. The algorithm I present has the added benefit of being a space-time tradeoff, which means that we may specify a parameter τ at run time that allows us to increase and decrease the time complexity and space complexity respectively, or decrease and increase the time and space complexity. Thus the space-time tradeoff algorithm I present allows more flexibility than previous work. This algorithm is randomized; however, it returns the correct value with high probability, and experimental results show that in practice this algorithm is reliable and efficient.

Problem 1 Position Shortest Unique Substring (PSUS)
Given a text $T[1..n]$ with alphabet Σ of size σ and a position $p \in [1, n]$, find

the substring $T[i..j]$ of T where $i \leq p \leq j$, $T[i..j]$ is unique in T, and $j - i$ is minized.

Randomized Algorithm for PSUS

Lemma 1
Given a length ℓ and a position p in T, there exists a randomized algorithm that can determine the existence of a substring with length ℓ that covers position p and is unique in T in $\mathcal{O}(\frac{n\ell}{m})$ time and $\mathcal{O}(m)$ words.

Proof:
There are at most ℓ possible substrings of length ℓ covering p that could be unique. We separate these substrings into $\lceil \frac{\ell}{m} \rceil$ batches of m substrings with adjacent starting positions. There may be a remainder batch that contains fewer than m substrings. For each batch we use the randomized Rabin-Karp rolling hash to hash the m adjacent substrings in $\mathcal{O}(\ell + m)$ time Karp and Rabin 1987. We can then hash the text T determine the uniqueness of the m substrings in T in $\mathcal{O}(n)$ time. Note that this procedure is correct with high probability Carter and Wegman 1979. This yields an $\mathcal{O}(\frac{n\ell}{m})$ time and $\mathcal{O}(m)$ word randomized algorithm that is correct with high probability. □

Theorem 1
There is a randomized algorithm that computes the shortest unique substring (SUS) of a text $T[1..n]$ that covers some query position p chosen at runtime in $\mathcal{O}(\frac{n}{\tau})$ words and $\mathcal{O}(n\tau \log n)$ time.

Proof:
If there is a unique substring in T with length ℓ and $\ell < n$, then there is a unique substring in T with length greater than ℓ. This property lets us use exponential search over the length ℓ of the SUS, starting with $\ell = 1$ and using Lemma 1 as a sub-algorithm, to find the SUS that covers a query position p. If ℓ^* is the length of the shortest unique substring covering p, then there will be $\mathcal{O}(\log \ell^*)$ iterations of the exponential search, resulting in an overall time complexity of $\mathcal{O}(\frac{n\ell^* \log \ell^*}{m})$ and a space complexity of $\mathcal{O}(m)$ words. Setting $m = n/\tau$, this is $\mathcal{O}(\ell^* \tau \log \ell^*) \subseteq \mathcal{O}(n\tau \log n) \subset o(n\tau^2 \log(n/\tau))$. This randomized algorithm is correct w.h.p. □

Implementation and Experimental Evaluation of the Randomized PSUS Algorithm

I implemented this algorithm in Python on my laptop with 7th gen i5 core. I evaluate both its reliability (in terms of producing the correct output), as well as its run time and space usage.

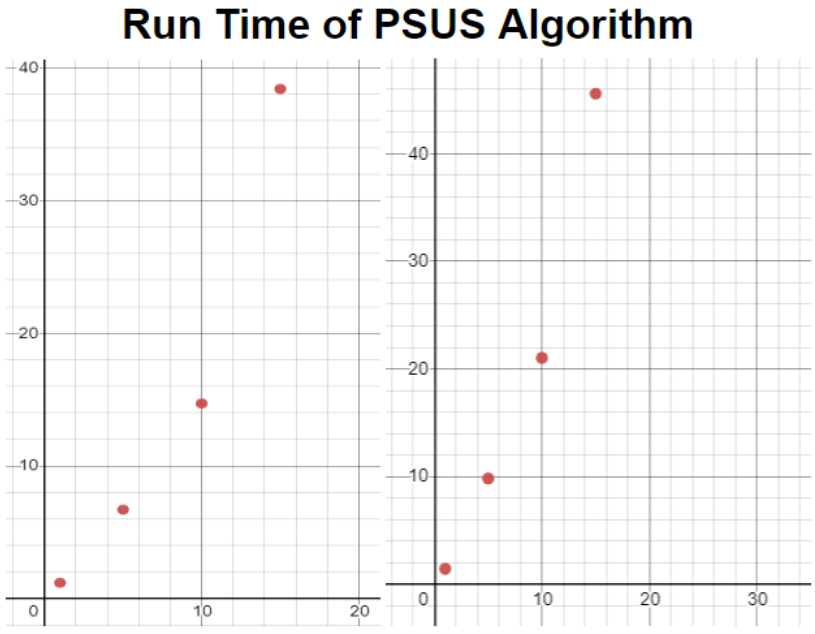
Experimentally-Evaluated Reliability

I evaluated reliability of the algorithm by running the randomized PSUS algorithm on randomly generated texts over a binary alphabet and randomly generated patterns of size $1 - 10000$ and $1 - 1000$ respectively. **Out of** 1000 **pattern and text pairs generated from the above set, I found that all outputs of the randomized algorithm produced a shortest unique substring covering** p**.** This reflects the high reliability of Python's hash function and modern-day universal hash functions in general.

Experimentally-Evaluated Time and Space Complexity

Unfortunately, I was unable to collect useful data for the memory usage of my algorithm because my space-time tradeoff analysis measures additional memory used, beyond the memory taken up by the input. However, it is impossible to measure space used by my python implementation my shortest unique substring algorithm that is not taken by the input text.

However, I was able to measure the time in seconds that my algorithm took to run. For this experimental data I chose $\tau = 1$ and $\tau = \sqrt{n}$. I generated random texts of lengths $1000, 5000, 10000, 15000$ symbols over an alphabet of size 256. Corresponding to each text I generated a random position p.



The experimentally evaluated time complexity for the PSUS algorithm for $\tau = 1$ and $\tau = \sqrt{n}$. The y axis is the time taken (in seconds) on my laptop. THe x axis is the length of the input text (in thousands of symbols).

The left figure shows the experimentally evaluated run time for $\tau = 1$ and the right figure shows the run time for $\tau = \sqrt{n}$. As was expected, the $\tau = \sqrt{n}$ run took longer to complete on average than $\tau = 1$. Both programs appear to run somewhat slower asymptotically than what is expected by our theoretical results. Perhaps this could be improved by optimizing my code.

Conclusion

My randomized algorithm has a comparable time-space complexity to Abedin et al. 2019 (an $\mathcal{O}(n^2 \log n)$ time-space complexity compared to $\mathcal{O}(n \log n)$ space and $\mathcal{O}(\log n)$ query time). Additionally, **my algorithm has the added benefit of giving a space-time tradeoff, which allows for more flexibility in time and space constraints**. This algorithm is randomized, however it produces the correct result provably with high probability (because the Rabin-Karp hash is correct with high probability). Furthermore, my theoretical results are supported by experimental results which indicate that my algorithm is both reliable and efficient.

References

[Abe+19] Paniz Abedin et al. "Range Shortest Unique Substring Queries". In: *International Symposium on String Processing and Information Retrieval*. Springer. 2019, pp. 258–266.

[CW79] J Lawrence Carter and Mark N Wegman. "Universal classes of hash functions". In: *Journal of computer and system sciences* 18.2 (1979), pp. 143–154.

[KR87] Richard M Karp and Michael O Rabin. "Efficient randomized pattern-matching algorithms". In: *IBM journal of research and development* 31.2 (1987), pp. 249–260.