Christopher Payne
Thomas Mathew
Vincent Tsuei
Ashish Goel

Benchmarking String Indexing Structures in Python

Abstract

Indexed string searching is an integral subproblem in many fields (e.g. computational biology, natural language processing). As such, there are a multitude of string indexing structures and libraries available for use on the web. However, there is not much information available as to how the various implementations compare with each other. We present comparative benchmarks of a number of these tools, available for use in Python, across a wide range of testing data, and analyze the performance and shortcomings of each implementation. We also present an improved way of searching in suffix arrays.

Introduction & Related Work

All code for this paper can be found at the GitHub repository at:
https://github.com/vtsuei2/IndexedSearchBenchmarks

Since text is the primary way we store and communicate knowledge, and with the exponential growth in the amount of data we store, we often wish to find specific pieces of information in vast knowledge-bases (e.g. the internet). Exact string searching provides a foundation to find the data we want, and thus, faster or more efficient algorithms can directly benefit many applications.

A significant number of string matching solutions rely on preprocessing the text to be searched into specialized data structures. Since we often want to search the same text repeatedly, this offers faster searching in exchange for some setup time. These structures include Suffix Trees, Suffix Arrays, and the FM-Index.

While each index structure has theoretical time and space bounds, performance on real world tasks is often influenced heavily by constant factors, which are ignored in big-O analysis. In addition, different implementations of the same indexing structure can differ in performance by a large amount. This is especially true when we are comparing ones written for different programming languages.

Though there has been some work on benchmarking various implementations of string indexes in C/C++[1], there is a distinct lack of such work for implementations written in Python. As the popularity of Python in scientific computing grows, we believe that a comparison of Python string indexing libraries will be invaluable in making the correct choice for projects in the future.

Methods

Design

An important consideration for string processing techniques is the size of the alphabet of the string. While the same algorithms apply correctly to both biological data and English text, the complexity of many algorithms and the size of associated structures can depend heavily on the alphabet size. For this reason, we consider three corpuses of text with widely varying alphabet sizes: one biological, one English and one Chinese. The biological corpus has an alphabet ranging over the characters {A, C, G, T}, the English corpus consists of Unicode characters (primarily those also in ASCII), and the Chinese corpus consists of mostly Unicode Hànzì characters.

Since the efficiency and performance of indexing structures also depend greatly on the size of the text to be indexed as well as the size of the query string, we also attempted to perform benchmarking across several order of magnitudes of these parameters.

We benchmarked several implementations of the three popular string indexing structures: Suffix Arrays, Suffix Trees and the FM-Index. For each structure, we attempted to find implementations for the most popular construction algorithms over time. For Suffix Trees, this was Ukkonen and McCreight's construction algorithms. For Suffix Arrays, this was **Karkkaïnen and Sanders'** algorithm and the SA-IS algorithm. We also benchmarked an implementation of the FM-Index, which utilized the aforementioned SA-IS algorithm as a subroutine.

Additionally, for searches, we wished to compare the effect of the "simple" and "super" accelerants, using the LCP (longest common prefix) and LCPLR matrices to improve binary searches on Suffix Arrays for given patterns. This is motivated by the fact that searches on suffix trees are expected to differ in runtime based on their internal representation of nodes, but all construction algorithms for suffix arrays yield identical results, which should take equal time to search.

Implementation

Corpuses

[1] https://code.google.com/p/libdivsufsort/wiki/SACA_Benchmarks

The biological corpus consists of the sequence data of several human chromosomes, appended together. The English (mostly ASCII) corpus consists of text compiled from Project Gutenberg, which is a collection of electronic, public domain books. The Chinese (Unicode) corpus consists of text compiled from Chinese Wikipedia articles.

Selected Libraries

We selected the following implementations to benchmark:

Suffix Trees

1. st_mccreight: A native Python implementation of McCreight's Suffix Tree algorithm
2. st_test_native: A native Python implementation of Ukkonen's Suffix Tree algorithm

Suffix Arrays

1. sa_linsuffarr: A native Python implementation of Karkainnen and Sanders
2. sa_pysuffix: A different native Python implementation of Karkainnen and Sanders
3. sa_sais: A wrapped C, SA-IS driven suffix array with native Python auxiliary structures
4. sa_variants.index_accelerant: Same as 3, but different auxiliary structures
5. sa_variants.simple_accel: Same as 3, but with no auxiliary structures

FM-Index

1. fmindexplusplus: A wrapped C++, SA-IS driven FM Index
2. fm_index: An educational FM-Index implementation, utilizing naive string sorting

Procedure

Each implementation was tested with the following procedure. Let $m \in$ {64K, 128K, 256K, 512K, 1M, 2M, 4M, 8M, 16M, 32M, 64M} and $n \in$ {50, 100, 250, 500, 1000, 2500, 5000} (K = 1000, M=1000000). For each corpus, do the following with every pair (m, n):

1. Select a random substring, T, of length m from the corpus.
2. Build the index structure on T.
3. Next, select 1,000 random substrings from T of length n - these are our queries.
4. For each query string, query the structure 1,000 times with that query string.

We measure the time it takes to build the index structure, the size of the final structure, and the time it takes to execute these 1,000,000 queries. We also check that the index structure returns a correct answer before beginning the time benchmarks.

The above procedure was designed to reduce the chance that random errors could skew our results. We build each structure multiple times over strings of the same length (once for each

different *N*), and query many different strings repeatedly to average out the results. If the testing procedure differed from that outlined above, we have noted so in the results section.

Measurement

The memory usage of each structure was measured using Pympler's[2] *asizeof* function. Since Python's heap is internally managed, we do not have direct access to the it and the memory manager. This makes measuring the memory an object in Python uses a non-trivial task, and designing such a method was beyond the scope of our research. The *asizeof* function only returns a very good estimate of object sizes, but we found it accurate enough for our purposes here.

The timing information was gathered using win32process.GetThreadTimes (part of Python for Windows Extensions[3]), and validated to be reasonable with the standard *timeit* library. It was necessary to use per-thread timing data as simple wall clock measurements can be invalidated by heavy CPU-loads on a system.

Test Bed Specifications

The specifications of the machine used to produce these benchmarks is as follows:

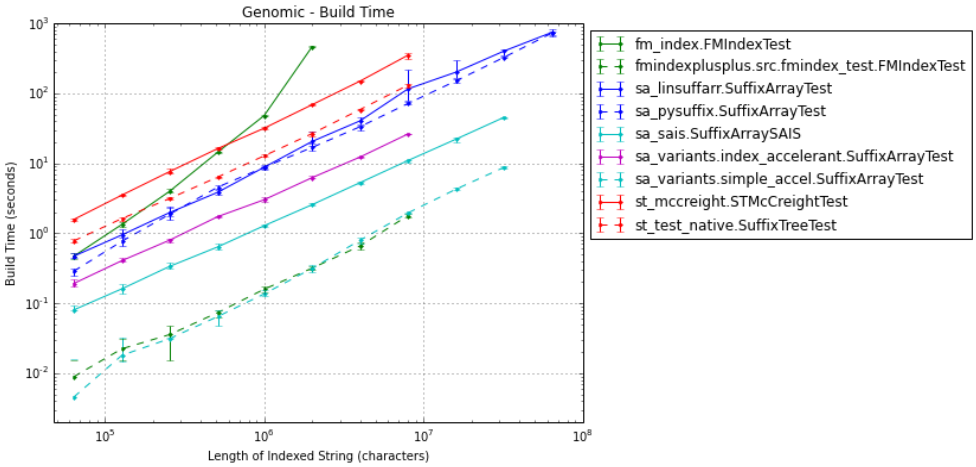
Operating System:	Windows 8.1 Pro (x64)
Processor:	Intel Core i7-4770K @ 3.5 GHz (Overclocked to 3.9 GHz)
RAM:	32 GB
Python Version:	Python 2.7.6 :: Anaconda 2.0.0 (64-bit)

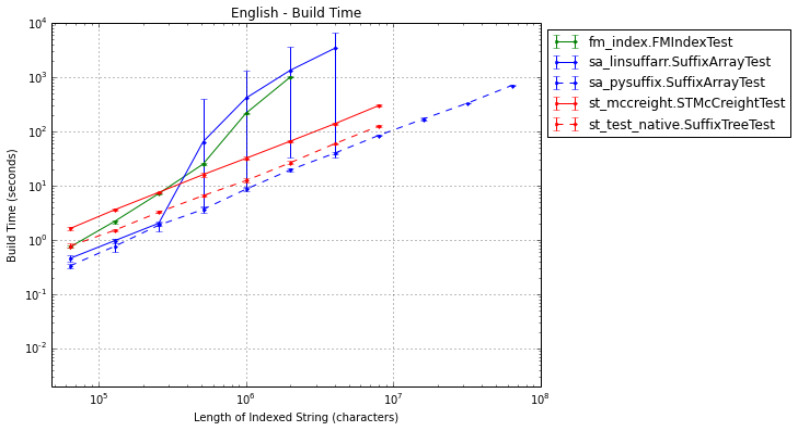
---

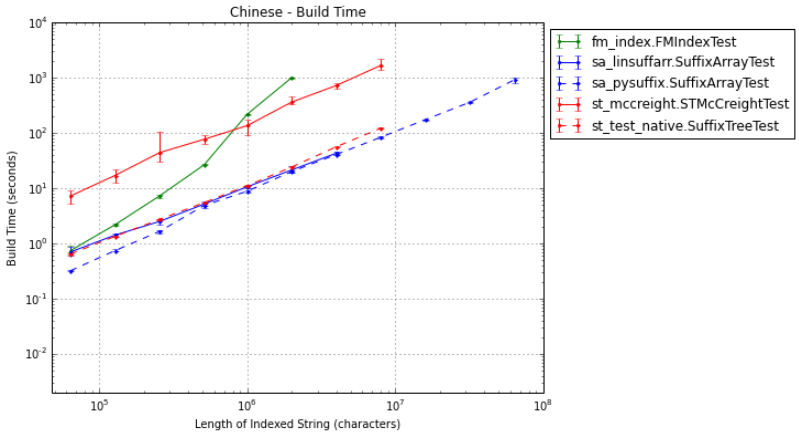
[2] "Pympler - PythonHosted.org." 2014. 10 Dec. 2014 <https://pythonhosted.org/Pympler/>
[3] "Python for Windows Extensions." 2014. 10 Dec. 2014 <http://sourceforge.net/projects/pywin32/>

Results/Evaluation

Build Time







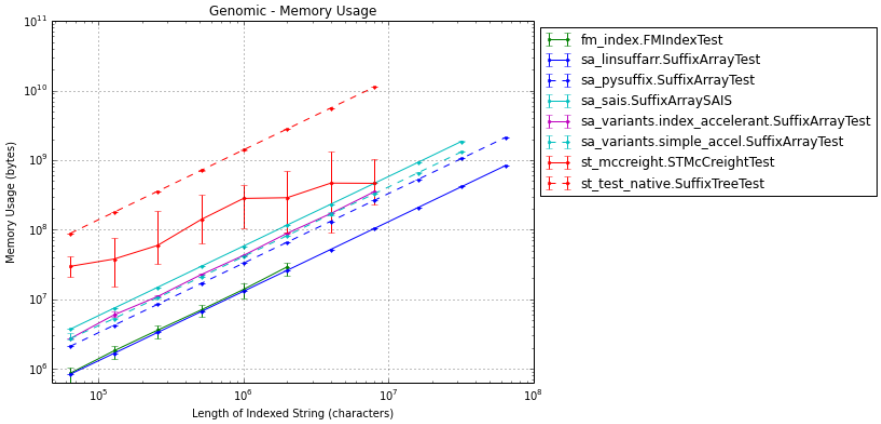
In the above graphs, we examine the build times for each structure for each different corpus. The bars accompanying each point represent the minimum and maximum build times we observed for each structure (each structure is built 7 times, the number of query string lengths). We note that the educational FMIndex implementation (solid green), performed very poorly with exponential growth in build times. The suffix trees take the most time to build, with the <st_test_native> implementation performing the best. The suffix array implementations varied wildly in build time - we note that the C++ variants (purple, cyan, dashed green), performed better than the native python ones (blue).

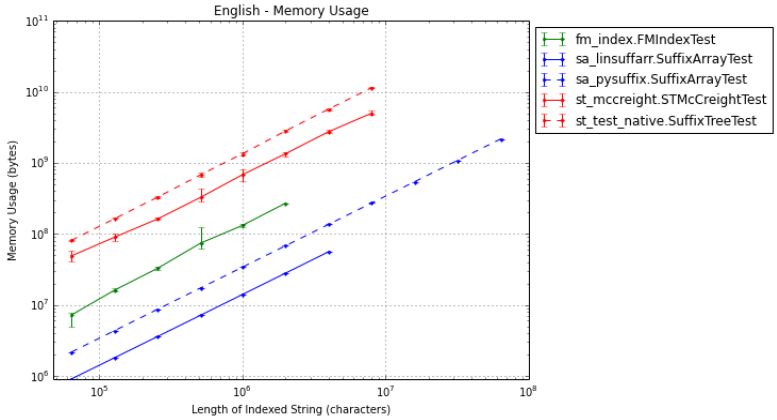
An additional thing to note is the performance <sa_linsuffarr> on English - the build times varied wildly. We suspect an implementation bug of some kind. For Chinese, we note that the construction time for the native suffix arrays was comparable to that of the better suffix tree implementation (st_test_native).

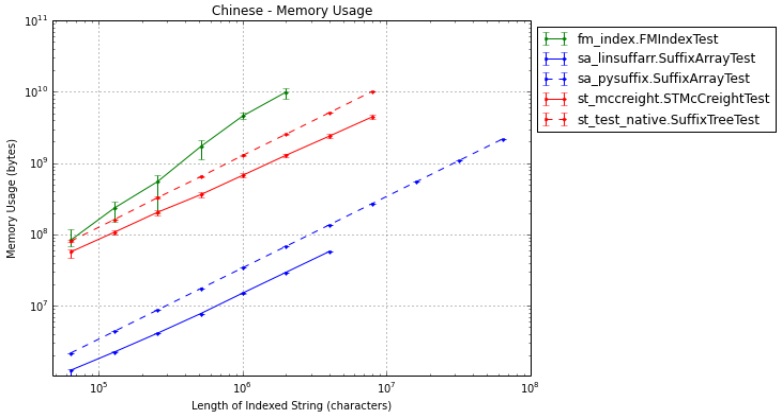
The best performance construction algorithms were the wrapped C SA-IS suffix array <sa_variants.simple_accelerant> and C++ FM-Index construction <fmindexplusplus>. The latter leverages the former as a subroutine. The FM-Index requires little additional time to process the suffix array and construct the FM-Index from it.

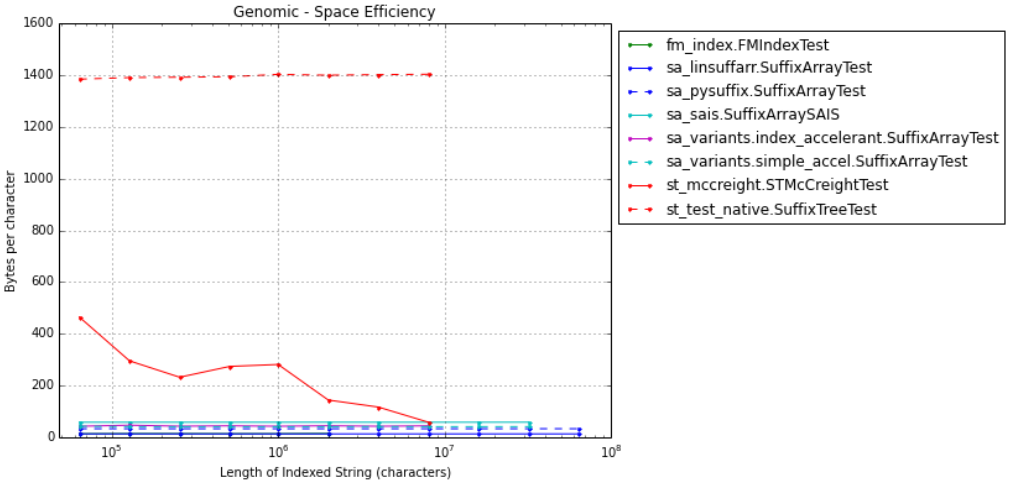
Additionally, <sa_sais> and <sa_variants.index_accelerant> leverage the wrapped C SA-IS construction algorithm, with additional linear processing performed in Python (construction of the LCPLR matrix and a hash table prefix index, respectively). These Python routines dominated performance, unlike in the case of the FM-Index, with the prefix index requiring more time to build.

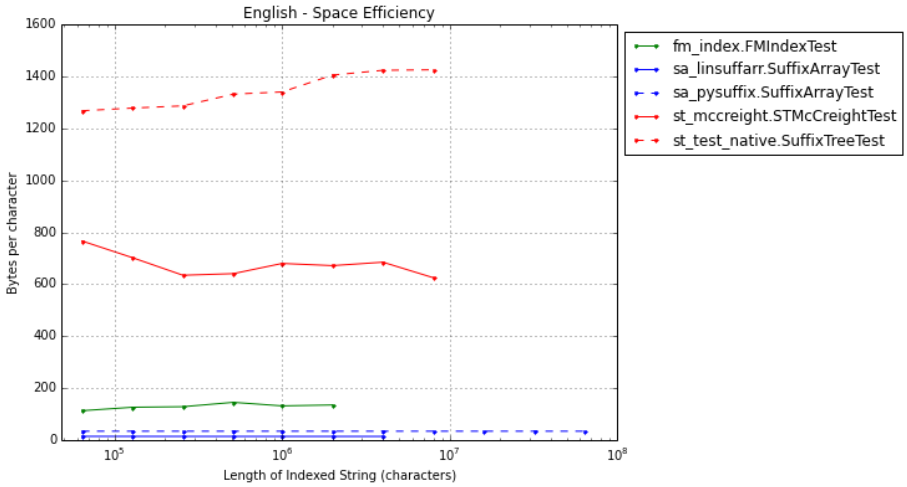
Memory Usage

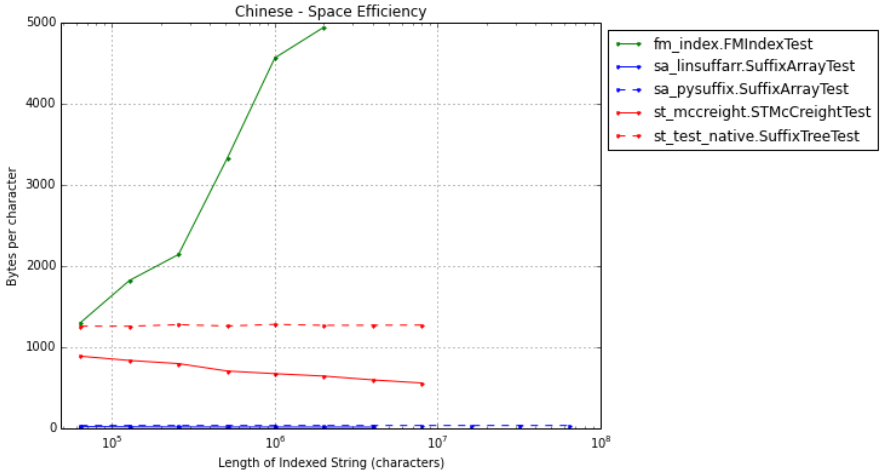





Chinese - Memory Usage
Memory Usage (bytes)
Length of Indexed String (characters)
fm_index.FMIndexTest
sa_linsuffarr.SuffixArrayTest
sa_pysuffix.SuffixArrayTest
st_mccreight.STMcCreightTest
st_test_native.SuffixTreeTest


Genomic - Space Efficiency
Bytes per character
Length of Indexed String (characters)
fm_index.FMIndexTest
sa_linsuffarr.SuffixArrayTest
sa_pysuffix.SuffixArrayTest
sa_sais.SuffixArraySAIS
sa_variants.index_accelerant.SuffixArrayTest
sa_variants.simple_accel.SuffixArrayTest
st_mccreight.STMcCreightTest
st_test_native.SuffixTreeTest


English - Space Efficiency
Bytes per character
Length of Indexed String (characters)
fm_index.FMIndexTest
sa_linsuffarr.SuffixArrayTest
sa_pysuffix.SuffixArrayTest
st_mccreight.STMcCreightTest
st_test_native.SuffixTreeTest

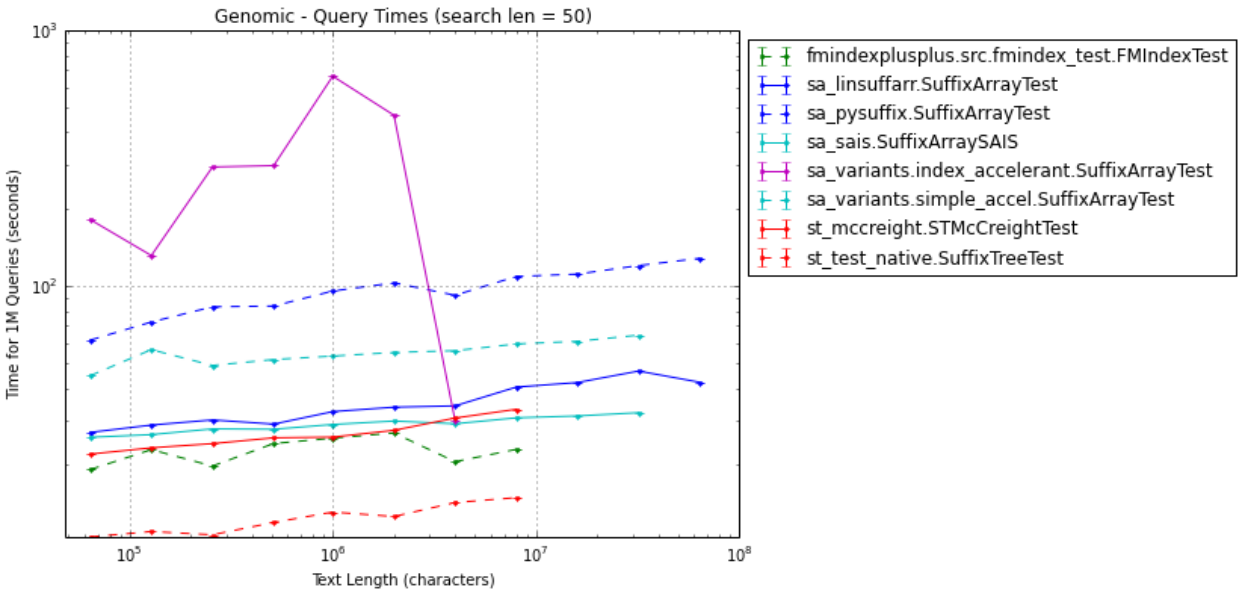

Chinese - Space Efficiency
Bytes per character
Length of Indexed String (characters)
fm_index.FMIndexTest
sa_linsuffarr.SuffixArrayTest
sa_pysuffix.SuffixArrayTest
st_mccreight.STMcCreightTest
st_test_native.SuffixTreeTest

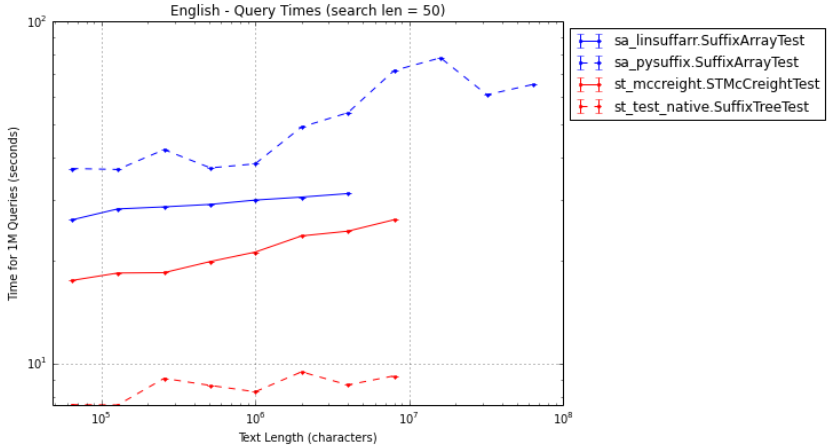
As expected, for genomic data the suffix trees took the most memory to store, followed by the suffix arrays, and then the FM index. We note that for English and Chinese, the FM index gradually starting using more memory compared to the other structures. Indeed, for the Chinese corpus, the FM Index used more memory than suffix trees!

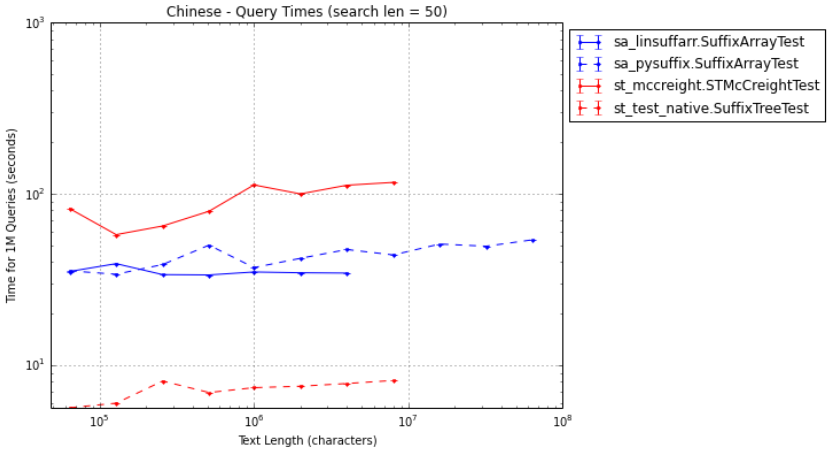
Of the suffix trees, the <st_test_native> implementation was much more memory efficient than the <st_mccreight> implementation. The greatest disparity was present in the Genomic corpus, where asymptotic performance was almost as good as the suffix arrays. We note however, the memory usage for <st_mccreight> varied wildly in our testing with the Genomic corpus.

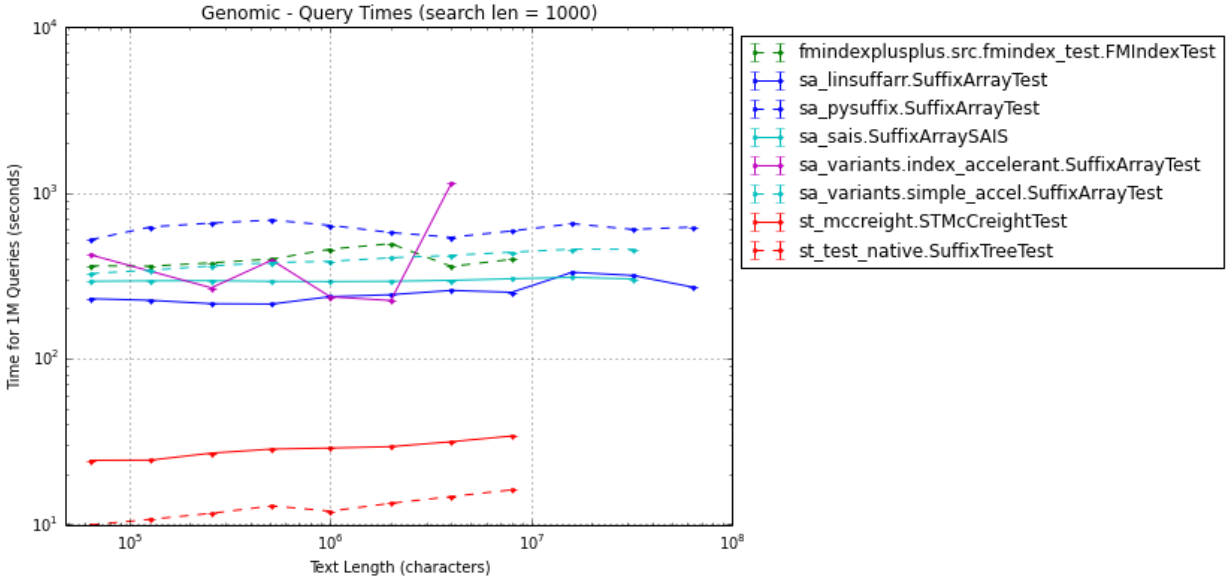
The various suffix arrays were comparable in space efficiency, but <sa_linsuffarr> was ultimately better across the board, beating <sa_pysuffix> by almost a factor of 2 in some cases.

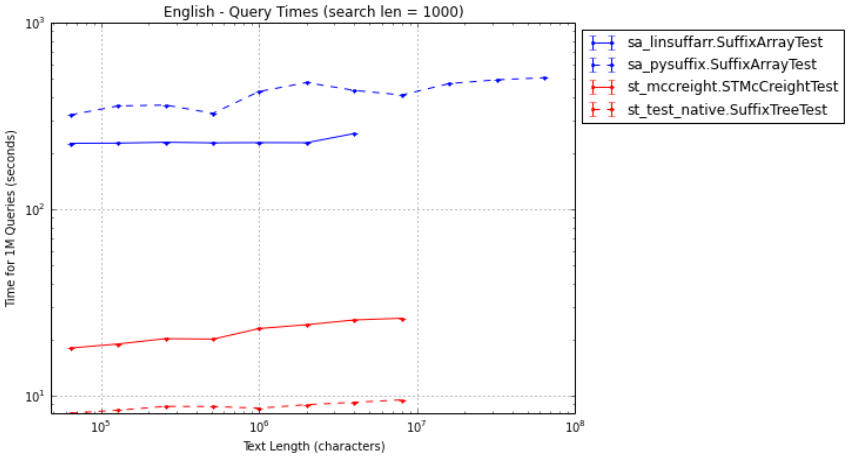
Query Times

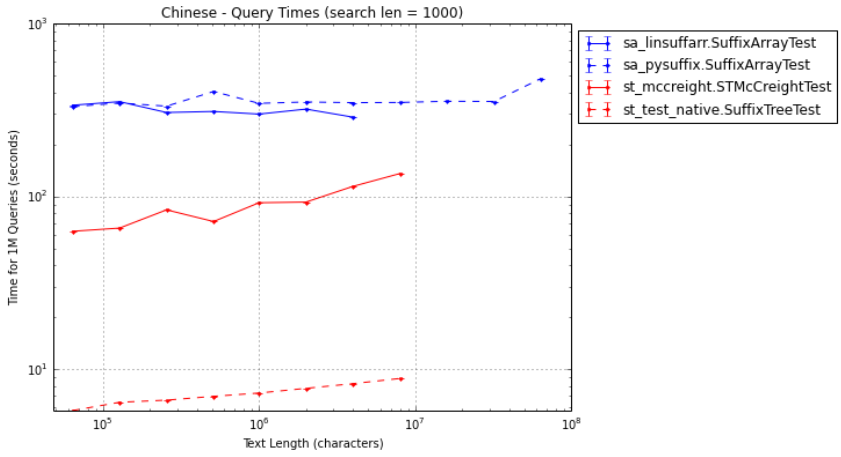

Genomic - Query Times (search len = 50)
Time for 1M Queries (seconds)
Text Length (characters)
fmindexplusplus.src.fmindex_test.FMIndexTest
sa_linsuffarr.SuffixArrayTest
sa_pysuffix.SuffixArrayTest
sa_sais.SuffixArraySAIS
sa_variants.index_accelerant.SuffixArrayTest
sa_variants.simple_accel.SuffixArrayTest
st_mccreight.STMcCreightTest
st_test_native.SuffixTreeTest

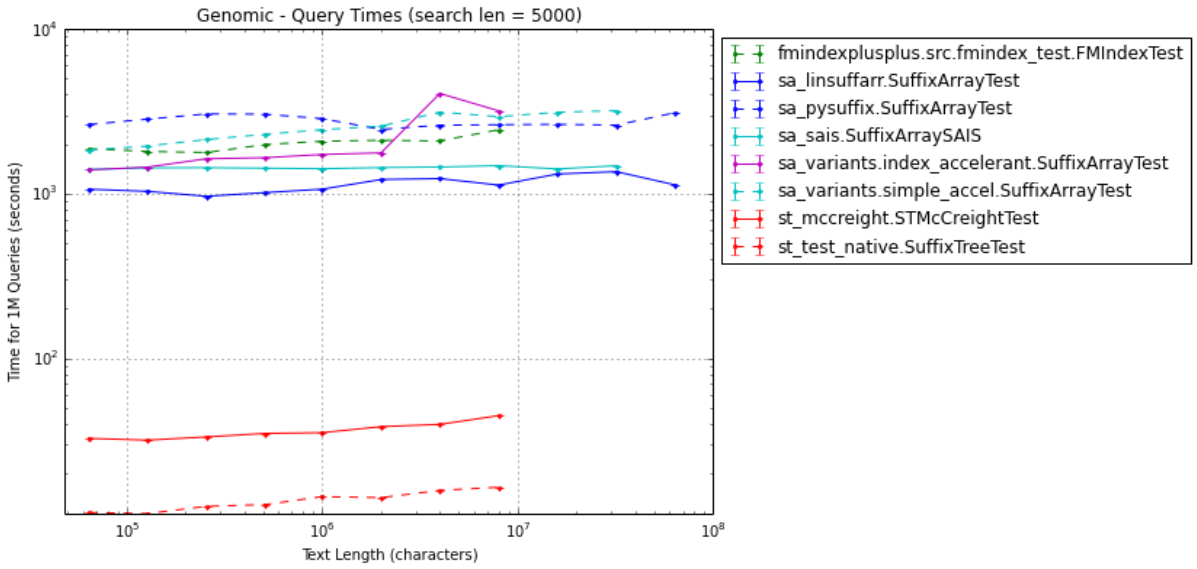

English - Query Times (search len = 50)
Time for 1M Queries (seconds)
Text Length (characters)
sa_linsuffarr.SuffixArrayTest
sa_pysuffix.SuffixArrayTest
st_mccreight.STMcCreightTest
st_test_native.SuffixTreeTest

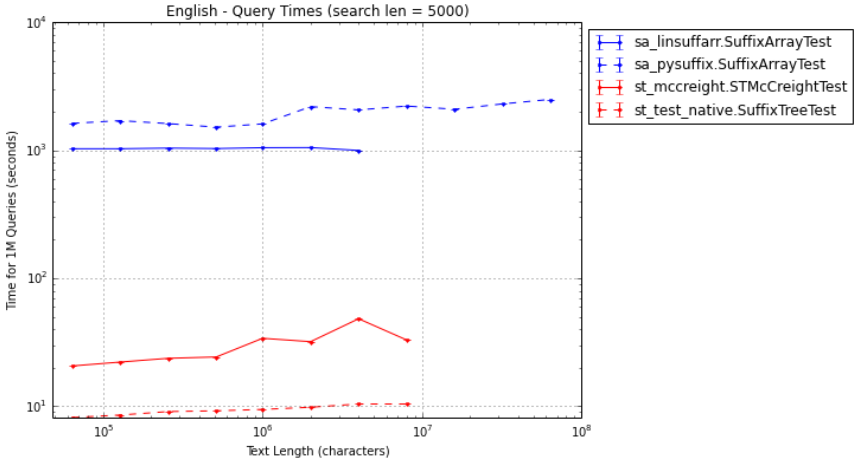

Chinese - Query Times (search len = 50)
Time for 1M Queries (seconds)
Text Length (characters)
sa_linsuffarr.SuffixArrayTest
sa_pysuffix.SuffixArrayTest
st_mccreight.STMcCreightTest
st_test_native.SuffixTreeTest

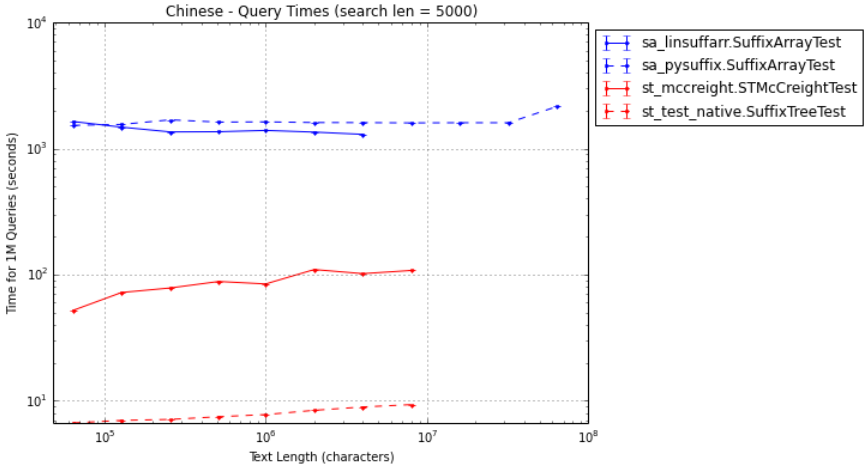

Genomic - Query Times (search len = 1000)
Time for 1M Queries (seconds)
Text Length (characters)
fmindexplusplus.src.fmindex_test.FMIndexTest
sa_linsuffarr.SuffixArrayTest
sa_pysuffix.SuffixArrayTest
sa_sais.SuffixArraySAIS
sa_variants.index_accelerant.SuffixArrayTest
sa_variants.simple_accel.SuffixArrayTest
st_mccreight.STMcCreightTest
st_test_native.SuffixTreeTest


English - Query Times (search len = 1000)
Time for 1M Queries (seconds)
Text Length (characters)
sa_linsuffarr.SuffixArrayTest
sa_pysuffix.SuffixArrayTest
st_mccreight.STMcCreightTest
st_test_native.SuffixTreeTest


Chinese - Query Times (search len = 1000)
Time for 1M Queries (seconds)
Text Length (characters)
sa_linsuffarr.SuffixArrayTest
sa_pysuffix.SuffixArrayTest
st_mccreight.STMcCreightTest
st_test_native.SuffixTreeTest


Genomic - Query Times (search len = 5000)
Time for 1M Queries (seconds)
Text Length (characters)
fmindexplusplus.src.fmindex_test.FMIndexTest
sa_linsuffarr.SuffixArrayTest
sa_pysuffix.SuffixArrayTest
sa_sais.SuffixArraySAIS
sa_variants.index_accelerant.SuffixArrayTest
sa_variants.simple_accel.SuffixArrayTest
st_mccreight.STMcCreightTest
st_test_native.SuffixTreeTest


English - Query Times (search len = 5000)
Time for 1M Queries (seconds)
Text Length (characters)
sa_linsuffarr.SuffixArrayTest
sa_pysuffix.SuffixArrayTest
st_mccreight.STMcCreightTest
st_test_native.SuffixTreeTest


Chinese - Query Times (search len = 5000)
Time for 1M Queries (seconds)
Text Length (characters)
sa_linsuffarr.SuffixArrayTest
sa_pysuffix.SuffixArrayTest
st_mccreight.STMcCreightTest
st_test_native.SuffixTreeTest

The results for query times were mostly as expected. Suffix trees were much faster than suffix arrays or FM indexes, with the disparity increasing as we increased the alphabet size. The <st_test_native> implementation, though much less space efficient, performed far better than <st_mccreight>. The FM-index performed best on Genomic data, but quickly became worse when the corpus had a larger alphabet. The educational FM-index was not included in results as the required number of queries would have taken months (if not years) to complete. Of the various suffix array implementations, the wrapped C version <sa_sais> performed the best on genomic data, though <sa_linsuffarr> was better on the English and Chinese corpuses.

Evaluation

Construction

From the above benchmarks, it is clear that the suffix array implementations are on average, faster than the suffix tree implementations. The SAIS algorithm, one of the most recently developed construction algorithms, was by far the fastest of these suffix array construction implementations.

The Python-wrapped C implementations were both faster and more memory efficient (sometimes by an enormous margin) than their pure Python counterparts. Due to the ease of integrating C libraries with Python and the relative lack of downsides to this approach (assuming correct, memory-safe, bug free implementation in C), this is probably the ideal approach for string indexing libraries.

Search

Suffix Trees are faster for search. Searching a Suffix Tree involves following a path of appropriate pointers down tree until the pattern is exhausted, then performing a traversal of the subtree rooted at the current node (ie. by depth first search) and reporting all leaf nodes.

Suffix Array search is relatively slow in comparison; while the running time of Suffix Tree search depends solely on the length of the pattern and the interior representation of edges, searches on Suffix Arrays are variants of binary search, and depend heavily on the number of entries in the Suffix Array itself, which is equivalent to the size of the text.

Variants of Suffix Array search provided moderate improvement to the overall running time. The naive search algorithm for Suffix Arrays takes $O(|P| * \log|T|)$ time; a binary search is performed with at most $|P|$ character comparisons at each of $\log|T|$ steps. A simple accelerated version of the search does not change this worst case time bound, but in practice yields a significant performance increase. Manber & Myers predicted that this accelerated search has an average performance of approximately $O(|P| + \log |T|)$ steps. It is infrequent that characters will need to be compared multiple times, thus the transformation of the multiplicative factor into an additive one.

Limitations

While we originally intended to benchmark the implementations on much larger input sizes (up to approximately 1G characters), time and memory constraints made this impossible for the largest strings. Indeed, for the suffix tree implementations, we could not test above 8M string sizes, due to the structures running out of memory during the build phase. However, we expect that our results can be used to extrapolate how each library would compare asymptotically.

Because the wrapped C++ implementations were not designed to be run on Windows easily, we were forced to emulate a UNIX environment via Cygwin[4] to benchmark them. Though we ran the benchmarks on the same physical machine, we note that there may be differences if it were run under the normal Windows environment instead. We expect these differences to be small enough that our results are still accurate.

Though we did test some C/C++ implementations that were wrapped for Python use, we did not test implementations in any other language. Given more time, we would have liked to expand the breadth of our benchmarking to include these implementations. In particular, we would have liked to write more Python wrappers for C/C++ implementations. However, we feel Python was still the right choice for this experiment, as it is seeing widespread adoption in scientific fields and the pure C/C++ implementations have already been thoroughly benchmarked.

Additionally, while we suggest a variant of suffix arrays, we developed this idea rather late in the project lifecycle and time constraints limited our ability for testing and optimization. Performance on synthetic data suggests there is ability to improve the average case performance and variance.

Proposed Improvement

We have seen that Suffix Arrays are both much faster to build and much smaller than Suffix Trees for the same texts, with the trade-off that searches in Suffix Arrays are much slower, even when leveraging optimizations such as LCP tracking and comparison.

The decision between which indexing structure to use depends heavily on the available system resources and factors such as the lifespan of the structure (perhaps we can invest more time upfront if we are going to keep the same structure to perform searches on for a very long duration). However, what if we only desire the fast construction property of Suffix Trees, and don't mind investing more memory? Can we achieve a compromise on both space and speed somewhere between these two extremes?

Both Suffix Trees and Suffix Arrays are indexes on strings. We suggest the use of an additional, sparse, secondary index on Suffix Arrays, a meta-index, to allow us to reduce the search space for a given pattern and reduce the time spent on searches. This approach is best for languages and alphabets with approximately uniform distributions and small alphabet sizes, and is thus most suited

[4] https://www.cygwin.com/

for genomic data. The examples which follow are over random strings drawn from {'A', 'C', 'G' and 'T'}.

Related Work

After designing and implementing this method of searching suffix arrays, we were directed to a Bioinformatics paper[5] describing a similar method, but it does not describe significant implementation details, except for noting that a compressed storage method is used.

Design

We describe our index in terms of the alphabet of strings and the length of substrings considered, which we refer to as index depth. Let us refer to the length of these substrings as the depth of the index. For an index of depth 2 over genomic data, we consider all possible 2-mers: AA, AC, AG, AT, CA, CC, CG, CT, … TG, TT. For the first occurrence of each substring in the suffix array, we store the offset in the suffix array where that suffix is stored. That if, if secondary-index(AC) = 98, then no suffix in the suffix array from index 0 … 97 begins with AC; only those indexes 98 … end may begin with AC.

We define the successor of a 2-mer to be the 2-mer which follows it in alphabetical order, i.e. succ(CA) = CG. Then, for a given 2-mer U, start-u = secondary-index(U), we can define V = succ(U), end-u = secondary-index(V). In the rare case where a successor 2-mer does not occur in the suffix array at all, we can let V = succ(succ(U)) and so on; however this is unexpected on typical texts and should not affect the average performance. Then, for any pattern P where P[0:2] = U, suffixes matching (beginning with) P must fall within the suffix array index range [start-u, end-u]. This can be extended to arbitrarily high depth, at the tradeoff of exponentially increasing size.

Storing the secondary-index takes O(|S|^|d|) space, where S is the alphabet and |d| the secondary index depth, or substring length. For an evenly distributed alphabet of 4 characters, d=1 reduces the search space by 1/4th, d=2 reduces it by 1/16th, etc. Consider a suffix array on 64 mega-characters; the suffix array itself consists of 64M start indices, and requires ~25 iterations of binary search to locate a given pattern. Consider an index of depth 10. The index consists of 4^10 = 1.05~M entries and reduces the search space by this same factor: 64M / 1.05M ~= 64. The remaining search space only requires 6 iterations of binary search to locate the pattern. At an overhead cost of 1.5% of the original index, we greatly reduce the time to perform a search.

Implementation

We treat each d-mer as a base 4 (size of genomic alphabet) number. We use numpy to efficiently translate integers to base-4 strings, and use simple iteration to convert between base-4 strings and d-mers. On a 64-bit system, this mapping allows us a maximum d value of 31. In the following passage, assume references to d-mers refer to their integer representation.

---

[5] Tárraga, Joaquín et al. "Acceleration of short and long DNA read mapping without loss of accuracy using suffix array." *Bioinformatics* (2014): btu553.

We store the index as a hash table, where d-mers are keys and the indices of first occurrence are values. Therefore, lookup of the search frame a given pattern may fall into requires two constant time operations, replacing potentially very many steps of binary search (each additional character in the keys partitions the suffix array into 4 regions, approximately 2 binary search steps eliminated per character).

Evaluation

For evaluation purposes, we compare the time for searches for patterns P, |P| = 5,000, for 3 (three) variations of suffix array search (naive, simple accelerant, secondary hash table index) and our fastest suffix tree implementation. We use synthetic, randomly generated texts with equal character (nucleotide) weighting of four different lengths. The results summarized here do not hold any statistical significance, as they are isolated runs used for design purposes. For more accurate performance evaluation, see the mean construction times over many runs in the earlier charts.

entries are build times, measured with time.clock() on a 64-bit Ubuntu 14.10 installation, Python2.7

	4M		**16M**		**64M**		**128M**	
	build	search	build	search	build	search	build	search
simple accelerant	0.857164	11.222416	3.205482	12.044062	13.349374	14.143267	27.456745	16.108828
super accelerant	2.856293	7.006646	11.256786	7.468779	45.40804	9.560148	91.628796	11.987077
secondary index	6.907396	**5.825909**	29.221132	**6.124503**	122.580688	**9.240691**	278.848156	**11.028613**

Shortcomings

Our current implementation leverages only Manber and Myers' simple accelerant as a subroutine after establishing the restricted bounds. Leveraging the guaranteed O(log |T| + |P|) super accelerant algorithm should be possible, with some caveats. The advantage in the super accelerant comes from precomputing the LCP for pairs of indices which may occur during a binary search beginning on the leftmost and rightmost index. As such, our exact indices mapping into the Suffix Array from the sparse meta-index will likely not fall on such a boundary. A solution would be to round the search frame outwards to the smallest valid binary search interval, but depending on the divisors crossed by the search frame, we may only trim the original search frame by a much smaller factor than the simple accelerant improvement, or in the worst case, not at all. Additional analysis is necessary to determine the effectiveness of a secondary index on super accelerant searches.

Furthermore, performance on real world strings differs greatly from the above randomly generated ones. Not only do individual characters not appear uniformly in biological sequences, but sequences of characters occur with different distributions as well. Thus, depending on the pattern searched, there could be large variances in the size of the reduced suffix array range. This is reflected in the earlier graphs, for which the indexed search has highly variant performance, especially on small

texts for which we use a short prefix, which exaggerates the skewed distribution. Furthermore, on synthetic texts, secondary index appears to perform slightly better than the super accelerant. On real world texts, this relation is reversed. However, it still performs better than the simple accelerant for sufficiently large texts and patterns.

Finally, there was an implementation error in the version benchmarked which we were unable to correct under time constraints. This could drastically affects the worst case performance in rare circumstances (eliminating any reduction in the search window whatsoever), and may have contributed to the varied results.

Conclusions

While Python is increasingly adopted as a tool in scientific computing, the fact that it is a memory safe and garbage collected language restricts its performance and inhibits its application to this field. Python is too slow to efficiently handle computations on large amounts of data on its own, with unmanaged languages' implementations performing several times faster than native Python implementations on average. We suggest as a compromise using wrapped C and C++ libraries generated with tools such as SWIG[6]. This allows mixing high performance C/C++ implementations with the utility and flexibility provided by Python. However, many C/C++ implementations lack support for Unicode, due to C's simple handling of strings. Python treats Unicode strings very similarly to ASCII ones, and thus one might consider using a native Python implementation in a scenario where Unicode support is required. However, for the purpose of processing genomic strings, wrapped C/C++ implementations more than suffice.

References

Related Work

1. Suffix Array benchmarks (from libdivsufsort development) <https://code.google.com/p/libdivsufsort/wiki/SACA_Benchmarks>
2. Tárraga, Joaquín et al. "Acceleration of short and long DNA read mapping without loss of accuracy using suffix array." *Bioinformatics* (2014): btu553. <http://bioinformatics.oxfordjournals.org/content/30/23/3396.full.pdf+html> Describes an improvement to suffix arrays using a secondary index similar to our own.

Tools and General References

1. Pympler - PythonHosted <https://pythonhosted.org/Pympler/>
2. Python for Windows Extensions <http://sourceforge.net/projects/pywin32/>

[6] "Simplified Wrapper and Interface Generator." 18 Dec. 2014 <http://www.swig.org/>

3. Cygwin
   <https://www.cygwin.com/>
4. SWIG - Simplified Wrapper and Interface Generator.
   <http://www.swig.org/>

Suffix Trees

1. st_mccreight: A native Python implementation of McCreight's Suffix Tree algorithm
   <http://wizardry-and-studies.blogspot.com/2005/12/mccreights-algorithm-of-building.html>
2. st_test_native: A native Python implementation of Ukkonen's Suffix Tree algorithm
   <https://github.com/kvh/Python-Suffix-Tree/blob/master/suffix_tree.py>

Suffix Arrays

1. sa_linsuffarr: A native Python implementation of Karkainnen and Sanders
   <http://www.gosme.org/suffixArray/linsuffarr-api.htm>
2. sa_pysuffix: A different native Python implementation of Karkainnen and Sanders
   <https://code.google.com/p/pysuffix/>
3. sa_sais: A wrapped C, SA-IS driven suffix array with native Python auxiliary structures (LCPLR)
   SA-IS: <https://github.com/davehughes/sais>
   LCPLR construction: <http://nbviewer.ipython.org/gist/BenLangmead/6783863>
   Super Accelerant:
   <http://www3.cs.stonybrook.edu/~rp/class/549f14/lectures/CSE549-Lec08.pdf>
4. sa_variants.index_accelerant: Same as 3, but different auxiliary structures
5. sa_variants.simple_accel: Same as 3, but with no auxiliary structures
   Simple Accelerant: <http://web.cs.ucdavis.edu/~gusfield/cs224f09/suffarraynotes.pdf>

FM-Index

1. fmindex-plus-plus: A wrapped C++, SA-IS driven FM Index
   <https://code.google.com/p/fmindex-plus-plus/>
2. fm_index: An educational FM-Index implementation, utilizing naive string sorting
   <https://github.com/egonelbre/fm-index>