A Simple Introduction to Compressed Suffix Arrays

David G. Andersen
Carnegie Mellon University

Abstract

This document attempts to provide a simple introduction to a quite fun, and possibly-becoming-useful full-text index structure: The compressed suffix array.

I have tried to write this document so that it's accessible to a reader with a comfortable grasp of advanced undergraduate computer science concepts in both theory and systems. Please let me know if that's not the case.

1 Introduction

As the volume of data available for computing continues its exponential growth, indexing in all forms grows in importance. Indexing allows searching in time proportional to the search string and number of results (and perhaps a logarithmic factor of the data size), instead of being linear in the size of the data. Henceforth, we'll refer to the search string as the *pattern* and the dataset as the *text*, but keep in mind that the text doesn't have to be just text—it could be binary or genome data, etc.

1.1 Traditional Inverted Indexes

The classic approach to indexing, particularly for text, is to use *term indexes*: Break documents into terms (e.g., individual words of english text), and create a list that maps terms to the documents that contain them. This is often called an *inverted index*:

```
dog ->  doc1, doc5, doc8, doc33
cat ->  doc2, doc5, doc7, doc9
toothbrush -> doc99
```

There are several good features about inverted indexes. First, if the text uses the same word more than once (quite common), the index is compressed: It only stores one mapping from word to document, even if the entire document consists of hundreds of repetitions of the word "chicken":

```
chicken -> zongker2000
```

Second, it is easy to perform more complex queries using the inverted index. Using the first example, to find all documents containing `dog` AND `cat`, simply take the two lists and intersect them to find that `doc5` contains our desired canine-feline mixture.

When used for binary data, inverted indexes are often used to index *N-grams*, overlapping repeats of characters. For example, to create an inverted index on this string: `catdog`, a trigram (3-gram) inverted

index would contain entries for `cat, atd, tdo, dog`. This works, but it substantially increases the size of the index.[1]

Drawbacks: The inverted index has several drawbacks, however, mostly related to flexibility of use:

1. It cannot search for substrings (except when used for trigram search);
2. It may not work well for languages in which the terms are not as discrete as they are in English[2]
3. Not as good for long texts with small alphabets, such as DNA sequences, which use a 4-character alphabet of `A T C G`.

1.2 Full-Text Indexes with Suffix Arrays

The *suffix array* data structure can be used as a full-text index, allowing arbitrary substring search in time $O(\log n)$ in the size of the text. The next section discusses suffix arrays in more detail. The major disadvantage of suffix arrays is their size: They require roughly one pointer per character in the text. A pointer needs to be big enough to point to any location in the text, so it requires $\lg N$ bits per pointer. Thus, the size of the suffix array is $O(n \log n)$, whereas the size of the original text is merely $O(n \lg |\Sigma|)$, where Σ is the alphabet.

For example, 4 gigabytes of ASCII text require $2^{32} * 8 = 2^{35} = 32$ gigabits of space. A suffix array for that same text would require 32 bit pointers, and thus $2^{32} * 32 = 2^{37} = 128$ gigabits, an increase of 4x relative to the text size. The increase is even worse for small-alphabet sequences such as genomic data: 4 billion DNA base pairs can be represented using $2^{32} * 2 = 8$ gigabits of space, and so the suffix array would be sixteen times larger than the text.

In the rest of this document, we will explore mechanisms to reduce that overhead from unacceptable to merely somewhat painful.

2 Suffix Arrays: Background

A suffix array is very simple: It lists the positions at which one can find in increasing order, the suffixes of the text. For example, the word `happy` has four suffixes:

```
(0)      happy
(1)       appy
(2)        ppy
(3)         py
(4)          y
```

The "first" substring, in lexicographic order, is `appy`. So we list that first in the suffix array, which, for this text, is:

```
 1, 0, 2, 3, 4
```

[1] For a good example of this, see Russ Cox's post about how he used a trigram index for Google Code Search: http://swtch.com/~rsc/regexp/regexp4.html. He reports that the index was roughly 20% of the size of the corpus, likely because source code contains significant redundancy at the trigram level: you can only type `i++` in so many different ways.

[2] e.g., in Chinese, sentences are written without spaces between the words. The mapping of characters to words is inferred by the reader. It's like reading this sentence: `thequickbrownfox`, except that it's easier in Chinese because the size of the alphabet is much larger, perhaps 3000 characters in everyday use.

To avoid storing the length of the string, many discussions will add an explicit end-of-file marker to the string; it is common to use `$` or `#`. One must also decide where the EOF marker sorts relative to the rest of the alphabet. In this document, we will use the dollar sign: `$`, and we will define it to sort *before* any other character. Our example string thus becomes `happy$`, and its suffix array is `5, 1, 0, 2, 3, 4`.

2.1 Creating suffix arrays

Creating a suffix array is easy. Creating it *efficiently* takes a bit more work. We refer the reader to the Googles and Wikipedia for discovering the best ways to construct one; in practice, most people just use N. Jesper Larsson's *qsufsort*, which works quite well.[3]

For the source code examples in this document, we'll use a Go-like syntax. The examples won't compile directly, but hopefully they'll get the idea across. Of note is the convention that `foo[3:5]` is a *slice* of the array foo, of length 2, containing the elements foo[3] and foo[4]. And `foo[3:]` is a slice beginning at element 3 and containing the rest of the elements in foo.

The core idea to create a suffix array is to first initialize an array of pointers into the text:

```
sa = []int{0, 1, 2, 3, ..., N}
```

And to then sort this list by comparing the text *at that location*:

```
func compare(a, b int) {
  return strcmp(text[a:], text[b:])
}

qsort(sa, compare)
```

At the end of that sort, the `sa` list will be ordered by the text suffixes' relative orders.

Speed of creation There are numerous approaches to speeding up suffix array construction. The naive method described above can potentially take $O(n^2 \log n)$: The call to string compare could potentially take $O(n)$ time, because it's comparing the suffixes *all the way to the end of the string*. That's not very OK. The improvements upon this naive method typically involve two approaches:

1. Recognize that work done on comparing suffixes can be re-used;
2. Use radix sort to eliminate the log n

For example, if asked to sort the suffixes of "happypuppy$", if one had already compared the strings `py$` and `pypuppy$`, then it would be simple to compare the two suffixes `ppy$` and `ppypuppy$`: After recognizing that the first characters are the same, one could reuse the earlier decision about `py$` and `pypuppy$`. This insight forms the basis for the faster mechanisms for suffix array construction. Exploring them is left as an exercise for the reader.

[3] An implementation is available in c++, and the algorithm can also be found in pkg/index/suffixarray/qsufsort.go in the Go language distribution. And probably a lot of other places.

2.2 Using the suffix array as an index

The suffix array stores the suffixes in lexicographically sorted order. Therefore, to search for a pattern using a suffix array, use binary search, comparing based upon a dereference into the text:

```
sa_search(pattern, sa, text, 0, len(text))

func sa_search(pat, sa, text, start, end) int {
  mid = (start+end)/2
  switch strcmp(pat, text[sa[mid]]) {
    case 0:  return mid
    case 1:  return sa_search(pat, sa, text, start, mid)
    case -1: return sa_search(pat, sa, text, mid, end)
  }
}
```

This description isn't entirely complete: It returns *a* match, but there may be many. The full algorithm should find the first and last matching entries, and return the range of sa that covers those. If we're being precise, the runtime of this *forward search* in the suffix array is $O(|P| \log n)$, where $|P|$ is the length of the pattern. Practically speaking, for large text, the lookups in the suffix array require memory fetches, but comparing additional characters in the text is sequential; this extra overhead is negligible for short patterns. However, some of the compression mechanisms we will explore may require (multiple!) memory fetches for each character retrieved from the text, in which case this overhead becomes a concern.

3 The Successor Array: Ψ

All of the approaches for compressing suffix arrays of which the author is aware share a common approach: Instead of using the suffix array directly, they use a *successor array* called Ψ (greek, "Psi"), which can be derived from the suffix array.

Consider our example string and its suffix array:

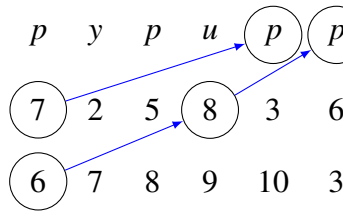
```
            happypuppy$

       SA:  10 1 0 7 2 5 8 3 6 9 4

 Suffixes:  $
            appypuppy$
            happypuppy$
            ppy$
            ppypuppy$
            puppy$
            py$
            pypuppy$
            uppy$
            y$
            ypuppy$
```

The Ψ array points to the location of a suffix's *successor*'s entry in the suffix array. You can think of this as an indirect pointer: If `puppy$` is the suffix in question, we mean `uppy$` as its successor. By an indirect pointer, we mean the index of SA that points to that successor. For example, $SA[3] = 7$, so its Ψ entry points to the location in SA that stores 8: $\Psi[3] = 6$. The general relationship is that:

$$SA[x] = SA[\Psi[x]] - 1$$

In the example given, $SA[3] = 7$, $\Psi[3] = 6$, and $SA[6] = 8$.

Offset	0	1	2	3	4	5	6	7	8	9	10
Text	*h*	*a*	*p*	*p*	*y*	*p*	*u*	*p*	*p*	*y*	$
SA	10	1	0	7	2	5	8	3	6	9	4
Ψ	$	4	1	6	7	8	9	10	3	0	5



The Ψ array has two important properties: It can be compressed, and it can be used to regenerate the original suffix array. These properties, together with some added information, will allow us to create a compressed suffix array that can be accessed (mostly) efficiently. Before proceeding, make sure you understand the relationship between Ψ, the suffix array, and the text. Note that in the example above, $\Psi[0] = \$$. We wrote this because the first suffix array entry points to the *last* position in the text; thus, there is no successor to point to. Observe also that one number does not show up in Ψ. What is it, and why not?

3.1 Regenerating the suffix array from Ψ

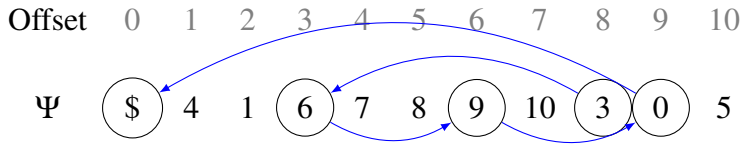
It's helpful to see how the suffix array can be recovered, brute-force, from Ψ. At the end of the previous section, we observed that one number (2) does not show up in Ψ, because $SA[2] = 0$. In other words, that entry in the suffix array points to the start of the text, and thus, no other suffix can have it as a successor.

The missing two immediately tells us that $SA[2] = 0$. Therefore, we have successfully regenerated one entry in the suffix array. We then look at $\Psi[2] = 1$. By construction $\Psi[2] =$ the index of the suffix array entry whose value is 1 more than $SA[2]$. By this, we know that $SA[1] = 1$. Continuing, $\Psi[1] = 4$, and so $SA[4]$ must equal 2. By continuing this process through N steps, we can regenerate the entirety of SA.

Looking up values in the middle. Suppose we wanted to determine the value of $SA[x]$, for some arbitrary x? The following procedure will accomplish this, albeit very, very slowly:

```
func sa_decode(x) int {
  hops_to_end = 0
  for Psi[x] != EOF_MARKER {
    hops_to_end++
    x = Psi[x]
  }

  return N - 1 - hops_to_end
}
```

The procedure merely hops through the successor array until it reaches the last entry (the one with no successor, marked \$). The value of this entry in SA must be $N-1$, the index of the last character in the text. The hop count tells us how far we started from that entry. For example, to determine the value of $SA[8]$:



would take 4 hops ($10-4=6$, which is the correct value of $SA[8]$).

While this procedure works, it requires $O(N)$ time to look up a single entry in SA. Not so helpful, but it forms the basis of something faster.

Looking up values in the middle, rapidly. Recall that the first problem we wish to solve with suffix arrays is that they take $O(n \log n)$ space, instead of the $O(n \log |\Sigma|)$ space required by the text itself. If we wish to construct data structures to augment our lookups from Ψ, we must not use more than that much space.

The simple answer is to store the real suffix array value for $\frac{1}{\log n}$th of the entries in Ψ, so that after doing $\log n$ forward hops, the process will encounter a "marked" node, and check an entry in an array to find out the text position. By taking that position and subtracting the hops to get there, the process can determine the actual SA value.

Quick analysis: Each entry in this array requires $\log n$ bits to store a position in the text. We store $\frac{n}{\log n}$ of them, yielding a total of $\frac{n \log n}{\log n} = n$ bits.

Considerations: One must decide how to store the values for the "marked" nodes, and which nodes to mark. The details can matter here; we'll get in to them later.

3.2 Compressing Ψ

To understand how to compress Ψ, it's helpful to look first at an example using a binary alphabet. We'll use the 16-character string:

```
a b b a a b b a a a b a b b b $
```

The suffix and Ψ arrays for this string are:

```
offset   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
Text     a  b  b  a  a  b  b  a  a  a  b  a  b  b  b  $
SA      15  7  8  3  9  4  0 11 14  6  2 10 13  5  1 12
Psi      $  2  4  5 11 13 14 15  0  1  3  7  8  9 10 12
```

Notice anything interesting about Ψ? The numbers 2 4 5 11 13 14 15 are all in increasing order. The sequence then jumps back, and starts increasing again. If we were to look at the suffixes pointed to by the corresopnding SA entries, we would see:

```
$
aaababbb$
aababbb$
aabbaaababbb$
...
```

All of the entries begin with 'a' (which makes sense; the suffix array entries are sorted lexicographically, so the ones at the start of SA *should* point to suffixes that start with 'a'). $\Psi[8]$ is interesting because it is no longer in the first increasing sequence. $SA[8] = 14$, and the suffix starting at position 14 is `b$` – in other words, this is the first suffix beginning with 'b' instead of 'a'.

Lemma 1: Two consecutive entries at positions x and $x+1$ in Ψ will be increasing if the suffixes they represent begin with the same character.

Explanation: By construction, the suffix array represents lexicographically increasing sequences. That is, $T[SA[x]] < T[SA[x+1]]$. These two suffixes share the same first character; therefore, they must differ in the later parts of the suffix: $T[SA[x]+1] < T[SA[x+1]+1]$.

The Ψ array points to the successor of a suffux: $T[SA[\Psi[x]]] = T[SA[x]+1]$. Because $T[SA[x]]$ and $T[SA[x+1]]$ share the same first character, the relation above holds: Their successors are ordered. Recall that the suffix array entries are in lexicographic order, so if $T[SA[\Psi[x]]] < T[SA[\Psi[x+1]]]$, then the suffix array entry for the first must appear before the second. Therefore, the indexes (the Ψ values) must be ordered, *if* the suffixes share the same first character.

3.3 Ordered sequences are compressible

In the binary example, the Ψ array consists of two increasing sequences of $\frac{N}{2}$ numbers between $0...N$. It should be immediately clear for this example that these sequences require only $O(N)$ bits each to store: Naively, one could store each one as a bitmap, though this approach would not permit random access.

The two most common methods to store increasing sequences of numbers are:

1. *Delta-encoding in blocks*: Operating in reasonably-sized blocks, store each number minus its predecessor ($\Psi[x] - \Psi[x-1]$). With such a dense array, these numbers are small. Then use conventional prefix compression techniques to store them efficiently. For our example, the 'a' array would be: `2, 2, 1, 6, 2, 1, 1`
2. *Quotienting with Elias-Fano*: The second method is a bit more intricate, but provides random access with relatively low overhead. It does, however, require some auxiliary data structures. We explain it next.

3.3.1 Storing dense ordered sequences with Elias-Fano

Given a sequence of increasing numbers $S = (1, 3, 4, 9, 11, 12, ...,)$ between 0 and $N-1$, in which roughly $\frac{N}{2}$ of the total numbers are present, the Elias-Fano representation would store it as thus:

```
bv = a bit vector of length N + N/2

for idx, val := range S { // idx = 0, 1, 2, ... val = 1, 3, 4, 9, ...
  bv[idx+val] = 1
}
```

If the third item (zero-based indexing, so $idx = 2$) in the set is 4, Elias-Fano stores it by setting bit $2+4 = 7$ to true. This produces a unique result because the numbers are in increasing: The next bit set in the array *must* be farther along. Consider items x and $x+1$ in the set. The index $x+1 > x$, obviously, and because they are ordered, $S[x+1] > S[x]$. Therefore, $S[x+1]+x+1 > S[x]+x$. That means that each item has a unique place that it will be set in the bit vector. The bit vector must have N + #items, enough to store the maximum value plus the maximum index into S.

That procedure allows us to *add* values, but how do we retrieve them? From above, if we know the location of the k-th 1, then we know the *value* of the k-th item in the set: $pos - k$. A simple but slow way to retrieve the value, then, would be:

```
func ef_get_slow(bv, k) int {
  ones := 0
  for idx, val := range bv {  // assume it gives us one bit at a time
    if val == 1 {
      ones++
      if ones == k {
        return idx - k
      }
    }
  }
}
```

However, this procedure takes $O(n)$ time. Again, not OK. But as we did with count-to-the-end earlier, we can augment the bit vector with a small amount of auxiliary information to be able to find the position of the k-th 1 bit very rapidly. The function that does this is referred to as `select`. It has a companion function, `rank`, and these two functions are usually provided together:

```
func select(bv, k) -> position of k-th one in bit vector bv
func rank(bv, x) -> number of 1 bits to the left of position x in bv
```

Intuition behind rank & select. The gist of making rank and select fast is that we can store the location of, e.g., every 64th "one" in the bit vector, and then count forward from there. Typical approaches use a two-level hierarchy to reduce the space a little more. The actual implementation of select is a bit more tricky, because the ones may be clumped together or spread out. For more information, see the 15-829 Rank & Select lecture.

Practical implications of rank & select. In practice, the extra data structures for rank & select add about 5-25% space overhead on top of the bit vector. For a large bit vector (tens to hundreds of MB or more), `rank` requires about two memory fetches, and `select` requires about four. The total time is dominated by the memory fetches unless the bit vectors are small enough to fit in cache.

Elias-Fano with rank & select Using these data structures, we can much more quickly perform Elias-Fano:

```
func ef_get(bv, k) int {    // get value of kth item in the sequence
  return select(bv, k) - k;
}
```

Unfortunately, this does add another fancy data structure to the mix. The author is aware of designs for compressed data structures that use both of these approaches (delta-compression or elias-fano-style representation). I'm not sure yet which is actually best in practice.

Suggested exercise: Before moving on, make sure you've got a decent grasp of Elias-Fano. Work through by hand an example bit vector for this sequence:

$$S = (0,3,4,5,8,11)$$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

dga: Ugh, no idea why that zero is showing up as a filled box. tikz leftover? Fix me.

Observe that it required only 17 bits to store the sequence, whereas a naive representation would have used 4 bit numbers for each element of S, and stored $6*4 = 24$ total bits. The space reduction from Elias-Fano is much more with larger sequences where the bits per element would be larger in a naive representation.

3.4 Storing sparse ordered sequences with quotienting

The example above worked well with a binary alphabet, because when we created the Ψ array, it consisted of two dense increasing sequences, one for suffixes starting with 'a' and another for those starting with 'b'. With a larger alphabet (e.g., 8-bit characters), however, Ψ will consist of one increasing sequence per character in the alphabet, or $|\Sigma|$ sequences. Instead of being half full, each will be only $\frac{1}{|\Sigma|}$ full (on average). How can we store *these* sequences more efficiently?

The answer is to use *quotienting*. If we expect only, e.g., $\frac{1}{256}$ of the numbers in a range to be used, as would be the case with an 8 bit alphabet, we can instead explicitly store the low-order bits, and use Elias-fano to represent where in the (now dense) upper bit range a number occurs. This makes sense by example. Consider eight numbers picked randomly in the range 0...1024:

$$(103, 4, 511, 442, 448, 733, 104, 562)$$

Or, in sorted order:

$$(4, 103, 104, 442, 448, 511, 562, 733)$$

We expect *on average* the spacing between the numbers to be roughly $\frac{1024}{8} = 128$, so we can store these numbers as "pairs" that indicate the quotient ($\frac{x}{128}$) and the remainder ($x mod 256$):

$$(\{0,4\}, \{0,103\}, \{0,104\}, \{3,58\}, \{3,64\}, \{3,127\}, \{4,50\}, \{5,93\})$$

We can then encode the quotient parts using Elias-Fano, as before. The sequence is $0,0,0,3,3,3,4,5$, which we can encode simply using $8+8=16$ bits. We then record an array of the remainders, $(4, 103, 104, 58, 64, 127, 58, 50, 93)$ using 7 bits each, for a total storage of 9 bits per element. As before, the savings in this toy example is small, but the representation uses only 9 bits per element *regardless of* n, so as n approaches 2^{32} or larger, the savings becomes substantial.

Asymptotically, using quotienting and Elias-Fano to store the Ψ array requires $n(2+\log|\Sigma|)$ bits. This is just slightly more space than is required to store the original text – two bits more per character.

3.5 Wrap-up of the simple approach

Putting these techniques together, we can compress a suffix array by:

1. Compute the successor array, Ψ (it has n entries).
2. Compute the "hops to the end" helper data structure to help rapidly look up $SA[x]$ using only Ψ (has $\frac{n}{\log n}$ entries, requires $O(n)$ bits)
3. Compress Ψ using delta-encoding or quotiented Elias-Fano (stores Ψ using approx. $n(2+\log|\Sigma|)$ bits).
4. Binary search the suffix array just like normal, but instead of looking up entries in SA, look them up using the compressed Ψ.

This basic idea works reasonably well. When you sum up the auxiliary data structures and the compression of Ψ, the resulting array uses about 4 bits more per character than the original text (a 1.5x increase for 8-bit ASCII text). That compares favorably with the 4x increase for the original suffix array, but this comes at a cost of $O(\log^2 n)$ cost to look up entries: Decoding $SA[x]$ from Ψ takes $\log n$ steps, and the binary search in the suffix array itself requires $\log n$ steps.

3.6 Details: Marking positions in Ψ

Recall that to use the "hop-forward through Ψ" approach, we needed to mark every $\log n$-th hop and store the original SA value for that hop. There are a few basic approaches to doing this:

Mark by position: The simplest approach is to mark items whose $index \% k = 0$, where $k = \log n$. Thus, when hopping through Ψ, as soon as the index value lands on one of these marked nodes, check entry $\frac{index}{k}$ to find its original SA value.

The advantage of this approach is that it is simple, and is effective on "normal" text. The drawback is that it is possible to construct sequences of text for which this approach will perform terribly: The search is not *guaranteed* to go to an index whose value is 0 mod $\log n$ for a long time.

Mark every k hops: Another way is to augment the Ψ array with a bit vector. Starting from the start entry in Ψ, hop through the array and set the bit vector to 1 for every $\log n$-th element. Once that's done, go back through, and compute the `rank()` for each marked item (recall, the `rank` is the number of 1s to the left of the item). The rank tells the location in a densely-packed array of original *SA* entries.

This advantage has guaranteed bounds, but requires an extra bit per element, and in practice, requires one or two more memory fetches per hop. Some work dga: find citation has found that mark-by-position is faster in practice.

3.7 Onward!

From here, there are two good places we can go:

1. We don't need to spend $\log^2 n$ to do the suffix array search using Ψ; it can be done using just $\log n$ steps.
2. There are more advanced techniques that further compress both the text and the index if the text is compressible.

4 Faster Lookups using Compressed Suffix Arrays

To understand how we can speed up searching in a compressed suffix array, let's return to our earlier example:

```
             happypuppy$

        SA:  10 1 0 7 2 5 8 3 6 9 4

  Suffixes:  $
             appypuppy$
             happypuppy$
             ppy$
             ppypuppy$
             puppy$
             py$
             pypuppy$
             uppy$
             y$
             ypuppy$
```

Observe that the middle suffixes, from $SA[3...7]$, all begin with "p". "Duh, Dave," you say, "This is obvious; the suffixes occur in sorted order." Indeed. Which means that if I were to tell you that there are 3 characters $<$ p, (`$, a, h`) and 8 characters $<=$ p (the previous three, plus 5 p's), you could immediately tell me that $T[SA[4]]$ must start with the letter 'p'. *Without having the original text around.*

To be less oblique, one can regenerate the original text from the suffix array and an array termed C that contains the cumulative count of characters in the text, by lexicographic order. For `happypuppy$`,

$$C = \{\$: 0,\ a : 1,\ h : 2,\ p : 3,\ u : 8,\ y : 9,\ EOF : 11\}$$

C is a very small array, only $|\Sigma| \log n$ bits. (Practically speaking, it fits in L2 cache.) Given an index in SA, we can quite easily binary search C to find the starting letter for that suffix. While this is technically $O(\log |\Sigma|)$ time, in practice, it's a few nanoseconds.

Using C for forward search in an uncompressed suffix array. In an uncompressed suffix array, C can be useful as an index to set the starting bounds for binary search. Past that, however, it's not all that great.

Using C for forward search in a compressed suffix array. When using Ψ instead of SA, however, C becomes much more useful. In fact, it is this use that suggests that Ψ is a more poweful representation than the suffix array itself.

Recall that each entry in Ψ points to the suffix array offset of the current string's successor. That's a mouthful for $\Psi[x] = SA^{-1}[T[SA[x]+1]]$. Here, we use SA^{-1} to mean the inverse suffix array, that is, the suffix array entry corresponding to the successor string.

Which means that from x, we can use C to compute $T[SA[x]]$. And from $\Psi[x]$, we can use C to compute $T[SA[x]+1]$. In other words, as we hop forward in Ψ (which we need to do to find the offset in the text anyway), we also get to perform a comparison against the text for free!

This search is still $O(\log^2 n)$, but using C in this way makes it substantially faster: We can stop hopping as soon as we find a character mismatch, *or* once we hit a marked node and can compare sequentially in the text.

Throwing away the text. From the previous, it should be clear that if we have the Ψ array and C, we can throw away the original text. To regenerate the text from a particular search result, we can simply hop forward in Ψ and regenerate the text using C. Practically, however, this starts to introduce pain: One memory fetch per character of the text, and the next hop depends on the result of the current one. Such an implementation would be limited in practice to perhaps 10 million hops per second, or 10MB/sec of text generation.

Backwards search allows $O(|P|\log n)$ string search Instead of searching forward, in which we have to decode the suffix array entry (in $\log n$ time) for each of the $\log n$ steps in binary search, the next cool observation about Ψ is that we can search backwards from the end of the pattern.

Consider again our example:

```
            happypuppy$

       SA:  10 1  0  7  2  5  8  3  6  9  4
      Psi:  $  4  1  6  7  8  9 10  3  0  5

 Suffixes:  $
            appypuppy$
            happypuppy$
            ppy$
            ppypuppy$
            puppy$
            py$
            pypuppy$
            uppy$
            y$
            ypuppy$
```

To search for the pattern 'ppy', begin from the *end* of the pattern. The C array tells us that patterns of the form 'y.*' can be found at positions 9–10 in the suffix array.

C further tells us that patterns of the form 'p.*' can be found between positions 3–7 of the suffix array. So to find a pattern of the form 'py.*', we are actually looking for an entry in the range 3–7 whose Ψ values are between 9–10: Ψ points to the suffix array for the successor, so that's equivalent to saying "find an entry starting with p whose successor is y".

By following back from the end of the search string, we can perform full-text search in time $O(|P|\log n)$ using *only the compressed suffix array and C*. At this point, the compressed suffix array has become a *self-index*, which can actually replace the text. As noted above, however, if you ever need to access the text sequentially, it's probably not a great idea to throw it away yet.

dga: To be continued...

5 Compressing

By the end of the previous section, we had created a index that replaced the original text (in theory), could support full-text pattern search in $O(|P| \log n)$ time, and used $O(n \log |\Sigma|)$ bits – in practice, about 2–4 more bits per character than the original text. At this point, it's worth asking: Can we do better if the original text is compressible? Darn right we can.

5.1 Entropy of a string

To begin, consider a text consisting of 256 characters, the first 255 'a' and the last one 'b':

```
aaaaaaaaaaaaaaaaaaaaaaaaaaa .... aaaaab
```

Intuitively, this string should compress to something small. The 0-th order Shannon entropy, H_0 of this string, refers to the size of it if we compress it based upon individual letter frequencies alone. The more biased the distribution of characters, the lower the entropy. (If 99.9999% of the characters are 'a', it's pretty easy to predict accurately which one comes next!).

$P('a') = \frac{255}{256}$ and $P('b') = \frac{1}{256}$

Therefore:

$$H_0(x) = P('a') \lg \frac{1}{P('a')} + P('b') \lg \frac{1}{P('b')} = 0.03687$$

In other words, a string generated from this distribution has an entropy of 0.03687 bits per character. The entire string of 256 characters contains 9.44 bits of information in total: Far less than its length would indicate.

Basic information theory tells us that the optimal number of bits to assign to these characters is equal to the "self-information" of the symbol – roughly speaking, how surprising it is to see that character. In this case, we should clearly assign fewer bits to an 'a' than to a 'b'. The self-information of a symbol is:

$$-\lg P(sym)$$

Here, 'b' has a self-information of 8 bits, and 'a' has a self-information of 0.005647 bits. If we were to use arithmetic coding to encode this string, then, it would require only about 10 bits. A simpler run-length code might encode this string as a255b0 using approximately 18 bits. In any event, these all compare favorably to the 256 bits required by the original string.

5.2 Representation using Ψ

Here I'll admit that I lied during the earlier analysis: The Ψ array representation we chose already provides some compression. To be precise, it's closer to the 0-th order entropy of the string than it is to the length of the string. To see this, consider again our `aaa..aab` string. When we represent its Ψ array, we need to represent it as two increasing sequences, one for suffixes starting with 'a' and one for suffixes starting with 'b'.

There is only 1 entry in the 'b' array. Rather than taking $O(n)$ bits to represent this array, when we quotient it, we allocate NO bits to the quotient, and $\log 256 = 8$ bits to the remainder. We can therefore store the 'b' array using only 8 bits.

For the 'a' array, there are 255 entries set out of 256 total. We don't need to use quotienting on this array: it's already dense. Therefore, instead of using the Elias-Fano representation (which can represent sequences with duplicates: 0, 0, 1, 2, ...), we can just store a bit array directly, using 256 bits. Note that we're not yet trying to get clever and compress this very obviously compressible bit vector.

Thus, even the basic compressed suffix array we have built thus far would use only $256+8=264$ bits to store *and index* our 256 character binary string. That's decently less than the 512 we would require if the string had a mix of 50% a's and b's.

dga: Continuing...

6 Variations on a theme

6.1 Grossi & Vitter's compressed suffix array

The first breakthrough in compressed suffix arrays came from Grossi & Vitter [**?**][4], which we will refer to as GV-CSA, with a nearly-simultaenous result from Sadakane.

In comparison to the "make $\log n$ hops through Ψ" approach described earlier, which takes $O(\log n)$ work for looking up an SA entry, and $O(\log^2 n)$ work for forward pattern search, the GV-SA achieves $O(\log \log n)$ work for looking up an SA entry. Both this work and the Sadakane approach, however, achieve $O(\log n)$ time for lookup, so it's not clear that the complexity of the GV approach is a good starting point in practice. However, its core insights are worth understanding because they shed further light on the structure of the suffix array and Ψ.

GV-SA begins with the critical insight that suffixes can be represented using the Ψ array; this paper was the first to make this observation. However, GV-SA then uses a divide-and-conquer mapping to map between Ψ and SA instead of the one-level mapping we discussed earlier.

N.B. In the discussion that follows, I have converted the notation used by GV to match the zero-based indexing used elsewhere in this document. The original work used a 1-based approach.

1. Create a bit array called "is-odd". Mark 1 or 0 as appropriate based upon the SA entries. If the *value* of SA[x] is even, set is-even[x]=1.
2. Compute Ψ as before.
3. Split the table into two halves:

Left-half : Retain the odd-valued $SA[x]$ entries.
Right-half : Retain the even-valued $\Psi[x]$ entries.

Observe that at this point, we can directly decode $SA[x]$ entries if their value is odd (1, 3, 5, ...). To decode an $SA[x]$ entry that was even-valued, however, we must go through the Ψ array. As before, $\Psi[x]$ gives an index in SA. We are guaranteed that this new entry will be found in the left-half: The original entry pointed to an even-valued offset in the text, so its successor is an odd-valued offset. Thus, the lookup function can return $SA[\Psi[x]] - 1$.

Now we have the problem of compressing these individual arrays. First, we must get rid of the unused values. This is where the "is-odd" bit array comes into play. First, pack the odd entries densely. We can locate $SA[x]$ in the odd entries by asking "which odd entry does x represent?" This question is `rank[x, is-odd]`. We can similarly locate even entries as `x - rank[x, is-odd]`.

Next, we compress the right-half Ψ as before, using an approach such as quotiented Elias-Fano.

Then, we compress the left half ("odds") by subtracting 1 from each entry and dividing by two. Because all entries are odd, we can regenerate them by takind $Odds[x] * 2 + 1$. This saves one bit per entry in odds, but we're still left with a large array.

Finally – and this is the ingenious next step – we recurse on the left half. The left half now represents a suffix array, which we can split and compress.

Repeating this recursion $\log \log n$ times ensures that lookup only takes $\log \log n$ time. It also means that the lowest level SA that is stored has only $\frac{n}{2^{\log logn}} = \frac{n}{\log n}$ entries, each of size $\log n$ bits, thus requiring only a total of $O(n)$ bits. Plus some space for the auxiliary bit arrays and their rank & select data structures.

[4] http://www.di.unipi.it/~grossi/PAPERS/sicomp05.pdf

6.2 Bibliography pointer

None of the ideas in this document are original; for a good list of pointers to the original papers, see http://homepage3.nifty.com/wpage/links/compressed_suffix_arrays.html.

Organizing a nice bibliography with pointers for each idea is on the to-do list.