Approximate Indexing: Gapped Suffix Arrays

Author: KyungHoon Park
Supervisor: Maxime Crochemore

Submitted in partial fulfillment of the requirements for the degree of
MSc in Advanced Software Engineering

Department of Informatics
School of Natural & Mathematical Sciences
King's College London

August 31, 2013
Table of Contents

1 Introduction .................................................................................................................. 3
  1.1. Overview .............................................................................................................. 3
  1.2. Motivation and Objective .................................................................................... 4
  1.3. Research Questions ............................................................................................. 5
  1.4. Research Aims ...................................................................................................... 5
  1.5. Organization ......................................................................................................... 6

2. Background .................................................................................................................. 7
  2.1. Overview .............................................................................................................. 7
  2.2. Suffix Array .......................................................................................................... 7
  2.3. The Skew Algorithm ............................................................................................ 9
    2.3.1. Differentiating into two lists ........................................................................... 10
    2.3.2. Sorting B1, B2 Array .................................................................................... 11
    2.3.3. Merging B1 and B2 ...................................................................................... 12
  2.4. LCP (Longest Common Prefix) ........................................................................... 13
  2.5. Approximate String Searching ............................................................................ 15

3. Specification and Design ............................................................................................ 17
  3.1. C# ....................................................................................................................... 17
  3.2. Platform Choice .................................................................................................. 17

4. Gapped Suffix Array .................................................................................................... 19
  4.1. Overview .............................................................................................................. 19
  4.2. Constructing the gapped suffix array ................................................................... 20
  4.3. Going beyond gapped suffix array .................................................................... 22

5. Evaluation ................................................................................................................... 28
  5.1. Objective achievement ....................................................................................... 28
  5.2. Further work ....................................................................................................... 28

6. Conclusion .................................................................................................................... 30

7. Bibliography ................................................................................................................ 31

Appendices .................................................................................................................... 32
User manual and screenshots ....................................................................................... 32
Core module source code ............................................................................................. 34
UI Program source code ............................................................................................... 53
1 Introduction

1.1. Overview

According to a research on our life span, if we suppose that the average life span of humans is seventy years, a man spends on average about a year of his life searching for something. This emphasizes the countless times how unconsciously we search for something in our daily lives. The world we live is progressing at an amazing speed growing in digital technology, expanding its capabilities, hence creating even more amounts of data which need to be searched. Likewise, the use of computers and technology is simultaneously speeding making it now near impossible to go without any technological knowledge.

Searching for files and information using computers has the same pattern as the unconscious searching that is continuously developing in our daily lives. Many people try to search certain files saved in a hard drive as well as information on websites quickly by comparing certain search strings created by users. However, aside from search engines bringing results that match the string patterns exactly, the recent technology has developed to a point where the computer not only represent the results of exact string matching patterns, but now represents any related expected search strings to avoid re-typing by users bringing out search results which are not only exact but also match analogically. In order to support the latter circumstance, the algorithm of “Approximate String Matching”, has been the subject of research since 1980. The class of algorithms which point out the need to search and filter information is string matching.

However, as communication speed is growing endlessly, our capability to perform string searching in real-time seems to fall behind. This searching for a string in a text is an important factor that is being researched and analysed on in computer science and its applications, but due to the difficulty in understanding algorithms and implementing the knowledge into practice, there is a need for further research.

Approximate string matching algorithms have long been the subject of research and are still being actively researched on to this date in order to match the speed and efficiency of indexing space. In the field of biology in particular, this is generally used for searching long genome sequences and for computing large amounts of queries. Genome sequences require
to be retrieved at a very fast pace and thus the use of such search technique in comparing is compulsory.

This report will explain what was first introduced by Crochemore on how quickly the Approximate string matching is implemented through the implementation of *Gapped Suffix Array* [1]. This Gapped Suffix Array is used when a pattern P of specific length is used, although the first character and the characters inside a gap do not have to match. The only limitation to gapped suffix array is that in order to complete the indexing, the length of the pattern needs to be determined and set a limit. In addition, there is another limitation that the number of Hamming Distance also needs to be set in advance. If the effects of these limitations are not too big then it is certain that the quickest approximate index algorithm is gapped suffix array, where the standard suffix array is enhanced and tailored to accept searches for patterns up to some mismatches.

### 1.2. Motivation and Objective

Realistically, suffix array algorithm is commonly used where massive amounts of strings are being searched like the DNAs. In the field of biology in particular, this is generally used for searching long genome sequences and for computing large amount of queries. Approximate string matching is much more desired than exact matching of strings in biological molecular sequences as it allows diversity without altering the basic information they carry. As mentioned earlier, genome sequences require to be retrieved at a very fast speed and thus the use of such search technique in comparing is compulsory and it is because of this importance of faster search technique in biology that pattern matching in textual data is being heavily studied on in computer science with many algorithms on strings and sequences devoted to developing this technique.

This is because through the works of indexing, quicker search is made possible. However, in order to carry out approximate matching algorithm, it is not appropriate to use only the index that had been created through suffix array. Let us imagine the length of pattern P is 3 and is constructed with 3 fragments. If for example, there is one mismatch, it is required to search the suffix 3 times with these combinations \((x_0, x_1), (x_1, x_2)\) and \((x_0, x_2)\). However in cases where gap exists in the middle as in the index \((x_0, x_2)\), it is difficult to search with the
predefined suffix index and instead must be carried out by gapped suffix array.

The gapped suffix array is a data structure which supports fast approximate searching. Based on the existing filter technique of approximate searching, the further development can be made on this to introduce gapped suffix array which can be used to improve search speed by avoiding the merging of position lists. The research into gapped suffix array is not so active and so there are not as many references to this. The reason may be due to the limitations of gapped suffix array's search properties. For example, assuming that there is a pattern constructed by 5 characters \(x_0, x_1, x_2, x_3, x_4\) and 2 hamming distances will be used to search the approximate string. In cases like this, it is almost certain that the pattern with 2 gaps in between will be constructed \(x_0, \#, x_2, \#, x_4\). In other words, as the above example, if the gapped suffix array can support the searching of multiple gaps such as those in double gapped algorithm, then it would most definitely be used as the optimal approximate matching algorithm. In this research, the core aim would be to discover such possibilities then following that would be to realistically develop gapped suffix array using the most popular platform and prove if all the algorithms mentioned in this paper can be developed through this. Also the paper states that \(O(n)\) search speed can be provided [1], and just like the construction time is being guaranteed by linear time, such speed will be re-validated. Finally, through understanding of gapped suffix array, there will be further research and development on the possibilities of multiple gap and its limitations.

1.3. Research Questions

Whilst developing gapped suffix array, couple of questions will be considered constantly.

1. Using the developed suffix array, can gapped suffix array be developed in \(O(n)\) time?

2. What are the limitations of gapped suffix array? What countermeasures can be suggested?

1.4. Research Aims

The aim of this research is to gain full understanding of gapped suffix array and through this, be able to realistically develop such algorithm. A step by step guide as to how this aim can
be achieved can be set as below.

1. To fully understand and implement suffix array and LCP.
2. Implement a gapped suffix array from the suffix array in $O(n)$ time.
3. To study and implement the paper gapped suffix array
4. If there are possibilities to develop multiple gapped suffix array, to research into other limitations and restrictions.

1.5. Organization

The paper is organised as follows.

Chapter 2 (Background) introduces the background knowledge required in developing gapped suffix array. First, explaining about the construction of suffix array and about the LCP algorithm required for fast searching. Finally provides information on approximate string matching with examples and explanations.

Chapter 3 (Specification and design) presents the chosen platform for the development of gapped suffix algorithm and the suitable programming language.

Chapter 4 (Gapped Suffix Arrays) shows how realistically gapped suffix array algorithm functions and also looks at the limitations of gapped suffix array and how to and the possibilities of overcoming such limitations.

Chapter 5 (Evaluation) Evaluates the aims of this project related to gapped suffix array, as well as introducing further future work required.

Chapter 6 (Conclusion) ends with comments on the development carried out in this project.
2. Background

2.1. Overview

As explained earlier, the gapped suffix array is also created in the same format as the pre-existing suffix array. The only difference is that it has a different method of constructing indexes. Hence, any understanding of gapped suffix array begins with the understanding of suffix array. Therefore basic understanding of suffix array will be delivered first along with the research into skew algorithm which is the most commonly used method in order to create suffix array at linear time. In addition, in order to help boost the search time in Suffix, LCP algorithm will also be looked at in detail.

2.2. Suffix Array

In 1990, Manber and Myers introduced suffix arrays as an alternative to suffix trees which has since then, been an important fundamental data structure efficiently used in many applications in efficient string processing. Although gapped suffix array is an approximate string matching algorithm, the suffix array is an exact string matching which is a basic step used by many algorithms in computational biology.

Given a text T of length n, the suffix array for T, denoted SA, is an array of integers ranging from 1 to n specifying the lexicographic ordering of the suffixes of the string S. It will be convenient to assume that T[n] = $, where $ is smaller than any other character. That is, SA[j] = i if and only if T[i .. n] is the j-th suffix of T in ascending lexicographical order. We will write Ti := T[i .. n]. The suffix array can be computed by sorting the suffixes, as illustrated in the following example below.

Let us assume the text is T = \textit{abaababbabbb$}, n = 13

\[
\begin{array}{cccccccccccccc}
T & a & b & a & a & b & a & b & b & a & b & b & b & $\\
\hline
i & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13
\end{array}
\]
If the string is sorted into 13 suffixes and sorted alphabetically, it can be sorted as below.

<table>
<thead>
<tr>
<th>i</th>
<th>( SA[i] )</th>
<th>( T[SA[i] \ldots n] )</th>
<th>( ISA[i] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>13</td>
<td>$</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>aabbbabbabbb$</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>ababbbabbabbb$</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>ababbabb$</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>abbabb$</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>abbb$</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>12</td>
<td>b$</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>baabbbabbabbb$</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>babbbabb$</td>
<td>6</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>babbb$</td>
<td>13</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>bb$</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>7</td>
<td>bbabb$</td>
<td>7</td>
</tr>
<tr>
<td>13</td>
<td>10</td>
<td>bbb$</td>
<td>1</td>
</tr>
</tbody>
</table>

Like the above, if the \( SA[i] \) index is sorted alphabetically, the suffix is arranged as shown. The suffix arrangement of the string "abaababbabbb$" therefore becomes \{13, 3, 1, 4, 6, 9, 12, 2, 5, 10, 11, 7, 10\}. The empty string can be considered to be the 13th suffix of "abaababbabbb". However as the empty string $ appears in front of all suffix sorted alphabetically, no specific information is displayed as a result, so even if the empty suffix is left out, that is not a problem.

There is one more concept which must be understood here and that is the arrangement of Inverse Suffix Array ISA. ISA is such that \( ISA[i] = j \) iff \( SA[j] = i \). ISA is valuably used when dealing with the HRANK concept which will be covered later in gapped suffix array.

However, the most difficult part is to create a suffix array using the most effective time and space. First, when the suffix algorithm was first introduced [2], doubling algorithm was also introduced, and in 2003 Skew algorithm by Karkkainen-Sanders was later introduced [3].
Both of these algorithms will be looked into briefly.

One of the algorithms that use the ‘doubling technique’ is the one introduced by Manber-Myers. [2] This suffix array construction algorithm uses bucket sort, where all the suffixes are grouped together according to their first characters. This bucket of suffixes sorted in lexicographic order of first characters is then sorted again up to its second character, and then up to the fourth character, where the number of characters in the suffix string doubles in each step. Here, Manber-Myers suggests the worst-case run-time of the algorithm is $O(n \log n)$, and the average-case run-time to be $O(n)$ [2].

The second algorithm that also uses this ‘doubling technique’ is the one introduced by Karkkainen-Sanders, where unlike the above algorithm, here, the algorithm involves encoding, recursion and merging. How this functions is it encodes two-thirds of suffixes by taking their first three characters and sorting them by recursion. The remaining one-third are then sorted and merged in lexicographic order of the suffixes sorted by recursion. Therefore, on each recursion stage, the size of input reduces to two-thirds. This algorithm functions under linear time and space except for the recursion processes, resulting in the total running time and space of this algorithm to $O(n)$.

If the two algorithms are compared with each other, the latter case is more advanced in its extraordinary performance. Therefore this algorithm will be looked at when creating the suffix array for this project. There is another array that is involved in this progress, known as the LCP array that is used in conjunction with suffix array. LCP array contains the lengths of the longest common prefixes between every pair of consecutive suffixes in sorted order. When a suffix array is coupled with information about LCPs in the suffix array, the string searching speed can be increased to $O(m + \log n)$ time.

### 2.3. The Skew Algorithm

In constructing the suffix array, the section that demands most time is where using this algorithm, a suffix algorithm is constructed in linear time. Hence it can be said that this step requires much deeper understanding and contains many steps. This is where skew algorithm, comes into play. This skew algorithm is a simple efficient algorithm that is easily
adaptable to various computational models, hence plays an important role in the
collection of suffix array. This skew algorithm for suffix array construction has several
steps where those suffixes beginning at positions $i \mod 3 = 0$ are sorted recursively to start
with, then this information is then used for sorting the remaining suffixes. To finish, the two
sorted sequences are then merged.

Each of these steps will be discussed. First, the text that will be used in below example is $T = \text{mississippi}$, $n = 12$.

<table>
<thead>
<tr>
<th>$i$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>m</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td>i</td>
<td>p</td>
<td>p</td>
<td>i</td>
</tr>
</tbody>
</table>

2.3.1. Differentiating into two lists

Firstly, we will look at the stage where the list is separated into two different sequences.
Once the index $i$ has been divided in 3, the remaining 1 and 2 suffixes will be put into a
bucket, which we will call $B_1$. The rest of the list with suffix 0 after dividing the index $i$ in 3
will be put into another bucket, $B_2$. Therefore it is possible to differentiate the list into $i$ and
$T$ as below.

<table>
<thead>
<tr>
<th>$i$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>m</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td>i</td>
<td>p</td>
<td>p</td>
<td>i</td>
</tr>
<tr>
<td>Buckets</td>
<td>$B_1$</td>
<td>$B_2$</td>
<td>$B_2$</td>
<td>$B_1$</td>
<td>$B_2$</td>
<td>$B_2$</td>
<td>$B_1$</td>
<td>$B_1$</td>
<td>$B_2$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Here if the length of $T$ is more than 2, the total number of all suffix in $B_1$ is less than those in
$B_2$. Each of the two Arrays will be holding below values as shown.

$B_1 = \{1, 4, 7, 10, 2, 5, 8\}$
$B_2 = \{0, 3, 6, 9\}$
The remaining step is to sort each bucket in lexicographical order of each of the suffixes and merging the two lists completing the construction of suffix array.

### 2.3.2. Sorting B₁, B₂ Array

Of the Array that has just been created, the B₁ Array must be sorted in lexicographical order of the suffix first, but this sorting is not just a simple sorting like the radix sorting where sorting is performed through the simple sorting algorithm.

The reason why the multiple of 3 is used to divide to create buckets was to compare each of the 3 characters and rank them making it easier to sort. Each of the 3 separated values are as below and where there are no strings at the end, these have been substituted with '0'.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T[i]T[i+1]T[i+2]</td>
<td>iss</td>
<td>iss</td>
<td>ipp</td>
<td>i00</td>
<td>ssi</td>
<td>ssi</td>
<td>ppi</td>
</tr>
</tbody>
</table>

Now these values must be compared and ranked. Here, the sorting algorithm must be used. Radix sorting algorithm was used in this project. Bucket sort is a simple sorting algorithm which is non-comparison based that allocates one storage location for each item to be sorted and data processed, being sorted into its corresponding bucket. Although the bucket sort is great in sorting small data range known in advance, there is a disadvantage that if the range of items to be sorted is very large, it requires unreasonable amount of memory to allocate enough buckets. However, another sorting algorithm that is similar to bucket sorting but does not have the memory problem is the radix sorting. This is when large data is sorted and broken down into several buckets just like bucket sort, but here the items are assigned to each bucket in the radix sort, which only considers a subset of the item key. The radix sort allocates sub-buckets and assigning items into them by different sub range of items’ keys. This process continues until only one item remains in each bucket at which point the items are recollected in order.

<table>
<thead>
<tr>
<th></th>
<th>T[i]T[i+1]T[i+2]</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>T[1]</td>
<td>iss</td>
<td>3</td>
</tr>
<tr>
<td>T[4]</td>
<td>iss</td>
<td>3</td>
</tr>
<tr>
<td>T[7]</td>
<td>ipp</td>
<td>2</td>
</tr>
<tr>
<td>T[10]</td>
<td>i00</td>
<td>1</td>
</tr>
<tr>
<td>T[2]</td>
<td>ssi</td>
<td>5</td>
</tr>
<tr>
<td>T[5]</td>
<td>ssi</td>
<td>5</td>
</tr>
<tr>
<td>T[8]</td>
<td>ppi</td>
<td>4</td>
</tr>
</tbody>
</table>
Realistically, both the T[1]’s triple characters and T[4]’s characters are the same so the next consecutive character must be used to compare. However, due to the suffix array’s recursive characteristic as briefly mentioned above, T[4] follows T[1] and T[7] follows behind T[4]. Therefore, the Rank following that can be used to compare [3]. So if the Full Rank is used to sort, then the SA from B1 can be sorted as below.

<table>
<thead>
<tr>
<th>i</th>
<th>T</th>
<th>Full Rank</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T[1]</td>
<td>3321554</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>T[4]</td>
<td>321554</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>T[7]</td>
<td>21554</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>T[10]</td>
<td>1554</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>T[2]</td>
<td>554</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>T[5]</td>
<td>54</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>T[8]</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

Until now, we have looked at the sorting of B1. If B2 was to be sorted in the same way as B1, it is possible to create SA of B2 as below.

<table>
<thead>
<tr>
<th>i</th>
<th>T</th>
<th>Rank Suffix</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T[0]</td>
<td>1432</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>T[3]</td>
<td>432</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>T[6]</td>
<td>32</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>T[9]</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

2.3.3. Merging B₁ and B₂

Now the sorting of B₁ and B₂ in lexicographical order is complete. Next step is to merge both B₁ and B₂ by comparing against each other to create a SA. Firstly, as the list of 1 is bigger than 2, using B₁ as a standard, we can compare each of the individual cases of B₂ against B₁.
and adding the values together. Here, the comparing standard is that depending on the suffix index of B_i, the comparable value standard differs and can be split into two different cases.

**When the remainder of dividing SA[i] by 3 is 1:** In this case the first character is compared and if the same, the SA value of next character is compared. ISA[SA[i]+1] method is used to search the next SA.

**When the remainder of dividing SA[i] by 3 is 2:** In this case the second string is within B_2 so compares up to the second character. If it is the same up to the second character, the string after the second character must be used to compare, where ISA[SA[i]+1] is used to search for the next SA.

Below shows the comparable values between B_1 and B_2.

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>SA</td>
<td>10</td>
<td>4</td>
<td>7</td>
<td>10</td>
<td>2</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>Comparing values</td>
<td>i0</td>
<td>i4</td>
<td>i5</td>
<td>i6</td>
<td>pp0</td>
<td>ss1</td>
<td>ss2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>SA</td>
<td>0</td>
<td>9</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>Comparing values</td>
<td>m36</td>
<td>p0</td>
<td>s14</td>
<td>s25</td>
</tr>
<tr>
<td></td>
<td>mi6</td>
<td>pi0</td>
<td>si4</td>
<td>si5</td>
</tr>
</tbody>
</table>

If two tables are combined using the Comparing values, below SA can be created

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>SA</td>
<td>10</td>
<td>7</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>9</td>
<td>8</td>
<td>6</td>
<td>3</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

**Theorem 1.** The skew algorithm can be implemented to run in time $O(n)$.

**2.4. LCP (Longest Common Prefix)**
The LCP construction algorithm was introduced by Udi Manber and Gene Myers as $O(n \log n)$ in 1993 when calculated with suffix array, [4] but in 2001 Kasai et al introduced the development of first linear time LCP arrangement under the basis of suffix array. [5] This is a binary search algorithm that finds a particular value quickly in an array of sorted values. This LCP is used to figure out whether to recurse to the left or the right side of the middle point $M$ in a binary search. If LCP array is used, this substitutes the default binary search that is used in suffix array. If LCP algorithm is not used, and binary search used instead, the search time is $O(m \log n)$, but by using LCP this time is developed into $O(m + \log n)$.

Searching using LCP can be carried out in the following methods. The $M$ itself has a prefix of $k$ characters in common with $P$: $\text{lcp}(P,M)=k$. Now there are three possibilities:

- **Case 1:** $k < \text{lcp}(M,M')$, i.e. $P$ has fewer prefix characters in common with $M$ than $M$ has in common with $M'$. This means the $(k+1)$-th character of $M'$ is the same as that of $M$, and since $P$ is lexicographically larger than $M$, it must be lexicographically larger than $M'$, too. So we continue in the right half ($M',...,R$).

- **Case 2:** $k > \text{lcp}(M,M')$, i.e. $P$ has more prefix characters in common with $M$ than $M$ has in common with $M'$. Consequently, if we were to compare $P$ to $M'$, the common prefix would be smaller than $k$, and $M'$ would be lexicographically larger than $P$, so, without actually making the comparison, we continue in the left half ($M,...,M'$).

- **Case 3:** $k = \text{lcp}(M,M')$. So $M$ and $M'$ are both identical with $P$ in the first $k$ characters. To decide whether we continue in the left or right half, it suffices to compare $P$ to $M'$ starting from the $(k+1)$-th character.

The LCP of Suffix table created earlier is as below.
2.5. Approximate String Searching

Approximate String matching involves finding the strings in the collection that are similar to a query string. For example, in a collection of strings $S$, if the given string $r$ is the query string, then the approximate string query will find all strings in the collection $S$ similar to $r$.

Approximate string matching not only perform string matching of a pattern in a text, but is also a type of string matching that allows errors; the one which perform string matching of a pattern in a text where one or both are corrupted. This error model is known as the edit distance which is the minimum number of single-character edit operations needed to transform one corrupted string to another not corrupted through such edit operations - include, insertion, deletion and substitution of simple characters in both strings.

This edit distance is measured in terms of the number of operations required to convert the
string into an exact match. Below are examples of such edit operations.

insertion: cot → coat  
deletion: coat → cot  
substitution: coat → cost

These edit distance operations may be generalized by adding a NULL character wherever a character has been deleted or inserted as below.

insertion: co#t → coat  
deletion: coat → co#t  
substitution: coat → cost

Depending on the requirements of approximate matchers, each edit distance operation impose different constraints. The total number of operations required to convert the pattern into an exact match may be set, in which case some edit distance operations may be limited in its usage. For example, if the total number of operations required to convert to an exact match, below example would not be counted as a single unit of cost as it requires more than one edit distance operation.

If the pattern is coil,  
Foil → one substitution,  
Coils → one insertion,  
Oil → one deletion  
Foil → two substitutions, making this operation not count towards a single unit of cost.
3. Specification and Design

The beginning of this dissertation, we have looked at the key points required in order to understand this project. These points covered at the beginning will be looked at in more depth in this section. This section will look at the platform and the programming language used to develop the demo programme.

3.1. C#

Although this project can be carried out with various languages such as Java and C++, it has been carried out using C# language. C# is a multi-paradigm programming language encompassing strong typing, imperative, declarative, functional, generic, object-oriented (class-based), and component-oriented programming disciplines. It was developed by Microsoft within its .NET initiative and later approved as a standard by Ecma (ECMA-334) and ISO (ISO/IEC 23270:2006). C# is one of the programming languages designed for the Common Language Infrastructure.

Of all the languages, the advantages and disadvantages of C++ and C# were compared but it seemed best to use C# for this project, as by using C++ there seemed no need to handle the memory by itself. C# clears and designates its own memory so can reduce the amount of codes, and has the advantage of having more understanding of the source code compared to C++. This however does not mean that it will use up memory and space as if it thinks nothing of them. The point where C# clears its allocated memory is when it assumes that the computer memory is not enough. It starts to clear out the memory of all the unused and unreferenced items. Moreover, C# supports various up-to-date platforms. Many of the algorithms and codes developed in this project will most definitely require introductions to various platforms and therefore, C# will be the most efficient and useful language for this project.

3.2. Platform Choice

With regards to the selection choice of platforms, the Microsoft .NET platform that supports C# has been chosen. There are many reasons for this choice, but primary reason was because in order to provide a user friendly UI of the demo programme, a windows
environment was required. Microsoft’s .NET platform supports various OS environments.
Firstly, the IDE known as Visual Studio .NET used in development functions based on the .NET Framework used in Windows, also contains the .NET environment that supports Mono Framework making it possible to function under the Linux environment.

.NET Framework platform supports multi language. Not only C# but it is possible to develop the same programme using VB or C++. In addition, even though the DLL engines for this project is developed using C#, it is possible to reference both C++ and VB developed environments. Moreover, just like the Java testing engine, it has the ability to self-test and supports the project, hence the reason why this platform was chosen as the best suitable platform for this project.
4. Gapped Suffix Array

The aim of this section is to describe gapped suffix array. The reason we had to understand the construction and structure of suffix array as introduced in the beginning section is because the construction of gapped suffix array is built straight after the construction of the suffix array. This section will have three main parts (i) the concept of gapped suffix array, (ii) construction of gapped suffix arrays, (iii) approximate string matching using gapped suffix array.

4.1. Overview

Gapped suffix array can be defined as the suffix array constructed based on the gap. As mentioned earlier, even before the index is created, the length of the gap must be already defined. For example, let’s assume that the search is ran against the index - ‘coat’ with the pattern length of 4 strings. Here, if the k-mistakes is defined as 2, the length of the gap will most definitely be same as the length of k-mistake, 2. So if we were to create a gapped suffix array that supports such search criteria as above, it is possible for gapped suffix array to support such changed patterns.

\[ \text{coat} \rightarrow c\#t, \#\#at, co\#\# \]

So, search words such as \#\#at, co\#\# are possible to be searched through already constructed suffix array, whereas those that contain such pattern, c\#\#t can be searched using gapped suffix array. However, as the current gapped suffix does not support multiple gaps, and only supports the gaps that are the same length as the pre-defined length before the index is created, below combination of patterns is not possible to be supported.

\[ \text{coach} \rightarrow c\#a\#h, c\#\#c\# \]

In order for the above patterns to be searched, various combinations of gapped suffix arrays are required. This will be dealt with in more depth in section 4.3, where the gapped suffix array is created after constructing suffix array based on the number of index the suffix array is composed of, starting from the first index to the last index where the cursor finishes. We will define the start cursor of the gap as \( g_0 \) and ending cursor as \( g_1 \). The gapped suffix array
created in this way will be shown as \((g_0, g_1)\)-gSA, often known in short as gSA. The length of the gap can be calculated as \(g_1 - g_0\).

With this in mind, we will now create the gSA based on the string Mississippi used in creating the suffix area in Section 2. Here, \(g_1\) of gSA is 1 and \(g_1\) 2, and the length of the gap is defined as 1.

\[
\begin{array}{cccc}
  i & T[i] & SA & gSA \\
  0 & mississippi & 10 & 10 & i# \\
  1 & ississippi & 7 & 7 & i#pi \\
  2 & ssissippi & 4 & 4 & i#sippi \\
  3 & sissippi & 1 & 1 & i#sissippi \\
  4 & issi & 0 & 0 & m#ssissippi \\
  5 & ssip & 9 & 9 & p# \\
  6 & sip & 8 & 8 & p#i \\
  7 & ipi & 6 & 5 & s#ppi \\
  8 & ppi & 3 & 2 & s#sippi \\
  9 & Pi & 5 & 6 & s#ippi \\
 10 & i & 2 & 3 & s#ississippi \\
\end{array}
\]

As the length of the Gap is 1, the k-mistake of the above table is also assumed to be 1. If the pattern \(pxi\) is used to search, this would require 3 different searches to be made. First, the deformed patterns \(px#\) and \(#xi\) performs the search in suffix array, but the only pattern with a gap in the middle \(pi(p#i)\) will perform the search in gapped suffix array.

### 4.2. Constructing the gapped suffix array

In order to create gapped suffix array, a separate RANK arrays, GRANK and HRANK must be
created. Firstly, GRANK contains the ranks of factors of y with length up to $g_0$. That is, rank created by cutting the characters before the beginning of the gap at position $g_0$. The GRANK created in the predefined gSA would look like below.

<table>
<thead>
<tr>
<th>$i$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T[i]$</td>
<td>m</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td>i</td>
<td>p</td>
<td>p</td>
<td>i</td>
</tr>
<tr>
<td>GRANK</td>
<td>5</td>
<td>1</td>
<td>8</td>
<td>8</td>
<td>1</td>
<td>8</td>
<td>8</td>
<td>1</td>
<td>6</td>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>

Here GRANK is defined as the first factor of RANK. HRANK defines the remainder of the string’s RANK. In the above string $T = \text{mississippi}$, the only 4 factors $m, i, s, p$ exist meaning that there are many cases where the value of the GRANK remains the same. Therefore, an additional RANK table known as HRANK should be created in order to differentiate GRANK but this HRANK contains the RANKs of the suffixes that are at the end of the gap. For example, the structure of the GRANK and HRANK of the fourth suffix \textit{mississippi} is constructed as below.

As we have now already created the suffix array before constructing the gapped suffix, it is possible to easily bring the suffix of where the gap ends. The HRANK can be defined as below.

$$HRANK[r] = ISA[SA[r]+g_i]$$

If the length of the suffix is now as long as the length of the gap, there are no ways to calculate the HRANK here, but we are aware that the empty strings are without the need for any calculation, higher than any other RANK. If we perform the radix sort by combining both GRANK and HRANK created in this way, it is possible to create gSA in linear time.[1]
**Theorem 2.** Given a string $y$ of length $n$ and its suffix sorting SA, the gapped suffix array $(g_0, g_1)$-gSA of $y$ can be computed in linear time $O(n)$.

### 4.3. Going beyond gapped suffix array

In this section we will look at the possibility of additional research on the factors which we defined as restrictions of gapped suffix arrays. [1] First we will define the language used in this section. We will show $k$-mistake as $k$, and the string pattern to be searched as $P$. The length of $P$ will show as $m$ and the gapped suffix array supporting both $m$ and $p$ as $(g_0, g_1)$-gSA($m, k$) with the length of the gap $|g_1 - g_0|$ as $gl$. Here, the value of $gl$ is equal to the value of $k$. For example, the gSA which we used as an example in the earlier section can be defined as $(1,2)$-gSA(3,1) and if we were to interpret this in detail, the length($gl$) of $k$-mistake and the gap is 1 and $m=3$ and the position of the gap is from 1 to 2.

In the case where $(1,2)$-gSA(3,1), we only need to support the below search criteria, so all approximate string matching could be supported.

$$P = cot$$

- (1,2)-gSA(3,1) c#t #ot co#
  - Searching array in the (1,2)-gSA(3,1) in the SA in the SA

Until now we have put $(g_0, g_1)$ in front of the gSA. As the research we want to carry out is the development of approximate string matching that supports values of all cases, the gap at the end of the cursor will always end with $|m - 1|$. Hence, $g_0$ can be defined as $|m - 1 - k|$, and as the value of $g_1$ and $k$ is the same, we will shorten $(g_0, g_1)$-gSA($m, k$) and show as gSA($m, k$).

First, we will look at the simple case where the value $k$ is 1. As an example, if we were to analyse gSA(4,1) and gSA(5,1), we can see that it requires creation of additional gSA as below.
If the condition is that the value of $k$ is 1, we can conclude that we will need creation of $m-3$ additional $gSA$ up to $gSA(m-1, 1)$ $gSA(m-2, 1)$ ... $gSA(3,1)$. The $k=1$ example introduced earlier can be defined as below.

Here, if we were to define the length of the gap as 1, $gSA$ can be created additionally making it possible to support the search performance in all occasions.
**Theorem 3.** If the length of the Gap is 1, the required count of gSA is $|m - 2|$, and it is possible for both construction and search time to be performed in linear time.

We will now look at the example where $k=2$. In this case, at the point where the value of $m - k$ is more than double, multiple gap start to appear. Here, we will show multiple gapped suffix array as $(g_0, g_1)(g_2, g_3)...(g_{n-1}, g_n)$-gSA$(m, k)$.

\[ P = coat \]
\[ gSA(4,2) \]
\[ \Rightarrow \ SA, gSA(3,1), gSA(4,2) \]

\[ P = coast \]
\[ gSA(5,2) \]
\[ \Rightarrow \ SA, gSA(3,1), gSA(4,1), gSA(4,2), gSA(5,2), (1,2)(3,4)-gSA(5,2) \]

\[ P = coasts \]
\[ gSA(6,2) \]
\[ \Rightarrow \ SA, gSA(3,1), gSA(4,1), gSA(4,2), gSA(5,1), gSA(5,2), (1,2)(3,4)-gSA(5,2), gSA(5,2), gSA(6,2), (1,2)(4,5)-gSA(6,2), (2,3)(4,5)-gSA(6,2) \]

In the above, the multiple gapped suffix array $(1,2)(3,4)$-gSA$(5,2)$ appears at gSA$(5,2)$. In this multiple gapped suffix array’s case, it was the gSA that was defined to support the pattern “c#at” from pattern “coast”.

Here, there are two types of approaches that exist in order to support the search of multiple gap. First is to continuously additionally create multiple gapped suffix array as per above method or instead of creating new multiple gapped suffix array, perform a search where the
search is carried out until the first gap of the search pattern, and after that every individual character is compared. Both of these cases will be looked at in more detail. First, let us define the total Count of gSA as \( gC \).

We will first look at the second approach. For example, we can perform the search by combining the searches as below.

\[
\begin{align*}
  &c &\# &a &\# &t \\
  r = gSA[i](3,1), T[r] \\
  &c &\# &a &s &\# &s \\
  r = gSA[i](3,1), T[r] \\
  &c &o &\# &s &\# &s \\
  r = gSA[i](4,1), T[r] \\
\end{align*}
\]

Hence, primarily the first section of the gapped suffix array will be searched under binary search method. If the length of the first fragment is named to be \( fm \), in order to find the first fragment, the search performing speed can be found by using binary search and gLCP making it possible to find within \( \log n + fm \). However, as we will need to compare every individual lists, in worst case, \( O((m - fm)n) \) time is required, meaning the searching time within gSA is \( O((m - fm)n + \log n + fm) \), in the assumption that such multiple index does not exist.
In addition, if the defined table above has been re-organised by removing multiple gapped suffix array, it can show as below.

<table>
<thead>
<tr>
<th>gSA(m, p)</th>
<th>Required gapped suffix arrays</th>
</tr>
</thead>
<tbody>
<tr>
<td>gSA(3, 1)</td>
<td>⇒ SA, gSA(3, 1)</td>
</tr>
<tr>
<td>gSA(4, 1)</td>
<td>⇒ SA, gSA(3, 1), gSA(4, 1)</td>
</tr>
<tr>
<td>gSA(4, 2)</td>
<td>⇒ SA, gSA(3, 1), gSA(4, 2)</td>
</tr>
<tr>
<td>gSA(5, 1)</td>
<td>⇒ SA, gSA(3, 1), gSA(4, 1), gSA(5, 1)</td>
</tr>
<tr>
<td>gSA(5, 2)</td>
<td>⇒ SA, gSA(3, 1), gSA(4, 1), gSA(4, 2), gSA(5, 2)</td>
</tr>
<tr>
<td>gSA(5, 3)</td>
<td>⇒ SA, gSA(3, 1), gSA(4, 1), gSA(4, 2), gSA(5, 3)</td>
</tr>
<tr>
<td>gSA(6, 1)</td>
<td>⇒ SA, gSA(3, 1), gSA(4, 1), gSA(5, 1), gSA(6, 1)</td>
</tr>
<tr>
<td>gSA(6, 2)</td>
<td>⇒ SA, gSA(3, 1), gSA(4, 1), gSA(4, 2), gSA(5, 1), gSA(5, 2), gSA(6, 2), gSA(6, 1)</td>
</tr>
<tr>
<td>gSA(6, 3)</td>
<td>⇒ SA, gSA(3, 1), gSA(4, 1), gSA(4, 2), gSA(5, 1), gSA(5, 2), gSA(5, 3), gSA(6, 3)</td>
</tr>
<tr>
<td>gSA(6, 4)</td>
<td>⇒ SA, gSA(3, 1), gSA(4, 1), gSA(4, 2), gSA(5, 1), gSA(5, 2), gSA(5, 3), gSA(6, 4)</td>
</tr>
<tr>
<td>gSA(7, 1)</td>
<td>⇒ SA, gSA(3, 1), gSA(4, 1), gSA(5, 1), gSA(6, 1), gSA(7, 1)</td>
</tr>
<tr>
<td>gSA(7, 2)</td>
<td>⇒ SA, gSA(3, 1), gSA(4, 1), gSA(4, 2), gSA(5, 1), gSA(5, 2), gSA(6, 1), gSA(6, 2), gSA(7, 2)</td>
</tr>
<tr>
<td>gSA(7, 3)</td>
<td>⇒ SA, gSA(3, 1), gSA(4, 1), gSA(4, 2), gSA(5, 1), gSA(5, 2), gSA(5, 3), gSA(6, 1), gSA(6, 2), gSA(6, 3), gSA(7, 3)</td>
</tr>
<tr>
<td>gSA(7, 4)</td>
<td>⇒ SA, gSA(3, 1), gSA(4, 1), gSA(4, 2), gSA(5, 1), gSA(5, 2), gSA(5, 3), gSA(6, 1), gSA(6, 2), gSA(6, 3), gSA(6, 4), gSA(7, 4)</td>
</tr>
<tr>
<td>gSA(7, 5)</td>
<td>⇒ SA, gSA(3, 1), gSA(4, 1), gSA(4, 2), gSA(5, 1), gSA(5, 2), gSA(5, 3), gSA(6, 1), gSA(6, 2), gSA(6, 3), gSA(6, 4), gSA(7, 5)</td>
</tr>
</tbody>
</table>

It is possible for us to easily find out the count of necessary arrays as they are increasing regularly with the above formula. The formula for calculating $gC$ is as below.

$$gC = \sum_{i=1}^{p-1} \frac{k - i}{1} \text{ if } k - i > 0$$

There are two main approaches to approximate string matching algorithms; Errors in Text approach and Errors in Pattern. The suffix array which we have looked at until now will definitely fall under the latter case and the worst case for this algorithm can be defined as
The search speed of gapped suffix array is then defined as \( O(n) \). [1] However, this only applies to instances where the length of the fixed \( m \) and fixed gap is decided. If the above two approaches are used, then it is most likely that the search speed can be certainly improved. The first approach mentioned above therefore, provides the search speed time of \( O((m + \log n)gC) \) and the second approach \( O(((m - fm)n + \log n + fm)gC) \). However, \( gC \) differs depending on the \( k \) and \( m \) and although we cannot define these values now, even in worst case, if we assume the worst scenario to be \( gC = km \), this is still faster and can reduce the search time, faster than what has theoretically already been proved as the worst case. However, although the search time can be reduced as such, in order to create each of the arrays individually, we will need additional space and construction time.

On the other hand, in the case of first approach, as mentioned earlier, the search speed of each of the gSA is extremely fast. This is because the search speed of each gapped suffix array is supported with \( O(m + \log n) \). However, we must multiply the Count of gSA here. Hence, it becomes \( O((m + \log n)gC) \). But as the gSA that supports multiple gap gets added on, the value of \( gC \) becomes greater than the first approach. Here, there will be a need for further research in the ways \( gC \) can be calculated.
5. Evaluation

5.1. Objective achievement

Section 4 introduced the detail of the gapped suffix array. Before this algorithm is constructed and applied, we first looked at the creation of suffix together with the sorting algorithm. As gapped suffix array is built on the basis of suffix array, suffix algorithm must be understood. Moreover, in this project, suffix algorithm and gapped suffix algorithms are both implemented.

First, it has been established that in the case where gapped suffix array algorithm is used, the $O(n)$ search time can be brought out as a result. [1] The only factor we must consider, is before this algorithm can be used, the fixed pattern length and the fixed gap length must be defined before running the algorithm. However, it is difficult to support approximate string matching of all patterns filtered following the gap structure because the gap length and the position has already been predefined.

As these limitations were already known, the possibilities of multiple gapped suffix array has been researched on in this project. First, although it was possible to deduce through this research that the search speed of multiple gapped suffix array can be faster than what has already been studied as the worst case search speed of $O(m^2 kn/\sigma^{m/(k+1)})$, in order to provide a more secure reasoning and evidence to prove this theory, we can conclude that a much more in-depth analysis and research is required.

In short, through this project, it was possible to have many chances to research on the related algorithms of gapped suffix array and was also possible through various studies to deduct the speed of algorithm as well as learn how to calculate them. On top of this, this project introduced a possible new direction where it triggered a chance to think about further study/research directions about gapped suffix array.

5.2. Further work

The algorithm speed of gapped suffix array has already been confirmed throughout its development. However, as this only supports searching of specific patterns, in order to be
able to support approximate indexing in all situations, there needs to be more research and development into multiple gapped suffix arrays. If there is enough memory of the index, it is possible, as described earlier to support faster searching. Therefore, the task for the future is to study multiple gapped suffix array and figure out how efficient it is to compare to current approximate string matching algorithms.
6. Conclusion

To sum up, this report gives an in-depth explanation of the aims and suggests possibilities of new direction of where the future research of gapped suffix array should involve. First, in the background section we looked at all the necessary background knowledge that was required in understanding this project. In order to understand the gapped suffix array, it is required to have the basic knowledge of approximate string matching and suffix array as well as about LCP. After that, we looked into the realistic development of algorithm that creates gapped suffix array, proving whether or not the re-defined theory of gapped suffix array is being created in $O(n)$ time.[1]

Then the main content of this research looked at the potential possibilities of multiple gapped suffix arrays. Finally, through this research we were able to conclude that it is in fact a field of research with high potential possibilities leaving us to conclude with the fact that further research plans may be required with more accurate evaluation of algorithms in order to fulfill such potentials.

Finally, it has been confirmed through this project that in the case of gapped suffix array, it is constructed of indexing algorithm which has the most respected ability to perform efficient searches. However, if the drawbacks of this algorithm mentioned throughout this essay can be developed and improved on, its value for practical usage will greatly increase.
7. Bibliography


Appendices

User manual and screenshots

When demo program is performed, the initial UI is as below.

The technique is clearly split into suffix array and gapped suffix array. The top right hand side has 3 buttons which are as below.

<table>
<thead>
<tr>
<th>Construct</th>
<th>Suffix Array is created first and based on this suffix array, gapped suffix array is created.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suffix Search</td>
<td>Perform suffix searching using the string written in the Suffix textbox.</td>
</tr>
<tr>
<td>Gapped Suffix Search</td>
<td>Perform approximate string matching using the string put into Gapped Suffix.</td>
</tr>
</tbody>
</table>

Below shows individual search results created from two arrays.
### Suffix Array

<table>
<thead>
<tr>
<th>Index</th>
<th>T</th>
<th>SA</th>
<th>T (GA)</th>
<th>ISA</th>
<th>LCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>mississippi</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>isissippi</td>
<td></td>
<td>ppi</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>isissippi</td>
<td></td>
<td>isissippi</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>issippi</td>
<td>1</td>
<td>isissippi</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>iississippi</td>
<td></td>
<td>iississippi</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>sissippi</td>
<td></td>
<td>pi</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>sippi</td>
<td></td>
<td>ppi</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>ippi</td>
<td></td>
<td>pipi</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>ppi</td>
<td></td>
<td>sississippi</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>pi</td>
<td></td>
<td>sipi</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>i</td>
<td></td>
<td>sipi</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

[Screen2] SA Search
[Screen3] gSA Search

Core module source code

SuffixArray.cs

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace GappedSuffixArray.SuffixArray
{
    public class SuffixArray
    {
        public int[] GenerateSA(char[] strOriginal)
        {
            SkewAlgorithm sa = new SkewAlgorithm();
            return sa.GenerateSA(strOriginal);
        }
    }
}
public int[] GenerateISA(int[] SA)
{
    int[] ISA = new int[SA.Length];
    for (int i = 0; i < SA.Length; i++)
    {
        ISA[SA[i]] = i;
    }
    return ISA;
}

public int[] GenerateLCP(int[] SA, char[] T)
{
    int[] LCP = new int[T.Length];
    for (int i = 1; i < T.Length; i++) //from 1
    {
        int y = 0;
        for (y = 0; y < T.Length - SA[i]; y++)
        {
            if (SA[i] + y == T.Length || SA[i - 1] + y == T.Length)
                break;
            if (T[SA[i] + y] != T[SA[i - 1] + y])
                break;
        }
        LCP[i] = y;
    }
    return LCP;
}

public List<int> SearchUsingLCP(char[] T, int[] SA, int[] LCP, char[] X)
{
    List<int> Result = new List<int>();
    // l = Left
    // d = Right
    // i = Midle (L+R)/2
    int d = 0;
    int f = T.Length - 1;
    int i = (d + f) / 2;
    while (true)
    {
        // ------  1. check if it is correct ------/
        if (IsContain(T, X, SA, i))
        {
            Result.Add(i);
        }
        // LCP
        // i  0
        // ippi 1
        // isppi 1 - Current Cusor
        // ississippi 4
        // mi~ 0
//Smaller
for (int y = i; y > 0; y--)
{
    if (LCP[y] >= X.Length)
    {
        Result.Add(y - 1);
    }
    else
    {
        break;
    }
}
//Greater
for (int y = i + 1; y < T.Length; y++)
{
    if (LCP[y] >= X.Length)
    {
        Result.Add(y);
    }
    else
    {
        break;
    }
} return Result;

//No Match
if (i == d && i == f)
{
    return Result;
}

//------ 2. define the direction to right or to left  ------/

//2.1 Left
if (IsBigger(T, X, SA, i))
{
    //lcp(x,i)==k
    int k = GetLCPLength(T, X, SA, i);
    //Compute LCP
    for (int y = i; y >= d; y--)
    {
        i = y;
        if (LCP[i] <= k)
        {
            break;
        }
    }
    f = i;
    i = (f + d) / 2;
}
//2.2 Right
else
{
    //lcp(x,i)==k
    int k = GetLCPLength(T, X, SA, i);

    //Compute LCP
    for (int y = i + 1; y <= f; y++)
    {
        i = y;
        if (LCP[i] <= k)
        {
            break;
        }
    }
    d = i;
    i = (f + d) / 2;
}

return Result;
}

public int GetLCPLength(char[] T, char[] X, int[] SA, int i)
{
    for (int y = 0; y < X.Length; y++)
    {
        if (SA[i] + y == T.Length)
            return y;
        if (T[SA[i] + y] != X[y])
            return y;
    }
    return X.Length;
}

public bool IsBigger(char[] T, char[] X, int[] SA, int i)
{
    for (int y = 0; y < X.Length; y++)
    {
        if (SA[i] + y == T.Length)
            return false;
        if (T[SA[i] + y] < X[y])
            return false;
        if (T[SA[i] + y] > X[y])
        {
            return true;
        }
    }
    return true;
}

public bool IsSame(char[] T, char[] X, int[] SA, int i)
{
    if (X.Length != T.Length - SA[i])
    {
        return false;
    }
```csharp
for (int y = 0; y < X.Length; y++)
{
    if (SA[i] + y == T.Length)
        return false;
    if (X[y] != T[SA[i] + y])
        return false;
}
return true;
}
public bool IsContain(char[] T, char[] X, int[] SA, int i)
{
    if (X.Length > T.Length - SA[i])
    {
        return false;
    }
    for (int y = 0; y < X.Length; y++)
    {
        if (SA[i] + y == T.Length)
            return false;
        if (X[y] != T[SA[i] + y])
            return false;
    }
    return true;
}
}

GappedSuffixArray.cs

using GappedSuffixArray.SuffixArray.Algorithms;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace GappedSuffixArray.SuffixArray
{
    public class GappedSuffixArray
    {
        //-------------------
        //Step1.
        public int[] GenerateGRANK(char[] GRANK_T)
        {
            int[] GRANK = new int[GRANK_T.Length];
            long[] UnsortedChar = new long[GRANK_T.Length];
            for (int i = 0; i < GRANK_T.Length; i++)
```
```csharp
string temp = ((int)GRANK_T[i]).ToString("0##");
UnsortedChar[i] = Convert.ToInt32(temp);

// Bucket Sort
long[] SortedChar = BucketSort.Sort((long[])UnsortedChar.Clone());

// Sorting
for (int i = 0; i < UnsortedChar.Length; i++)
{
    for (int y = 0; y < SortedChar.Length; y++)
    {
        if (SortedChar[y] == UnsortedChar[i])
        {
            GRANK[i] = y;
            break;
        }
    }
}
return GRANK;

//---------------------------------------------------------------------
//Step2.
public int[] GenerateHRANK(int[] SA, int[] ISA, int g0, int g1)
{
    int[] HRANK = new int[SA.Length];
    for (int i = 0; i < SA.Length; i++)
    {
        if (SA[i] + g1 < SA.Length)
        {
            HRANK[i] = ISA[SA[i] + g1] + 1; //to remove 0
        }
        else
        {
            HRANK[i] = 0;
        }
    }
    return HRANK;
}

//---------------------------------------------------------------------
//Step3.
public int[] GenerateGappedSuffix(int[] GRANK, int[] HRANK, int[] ISA)
{
    int[] gSA = new int[GRANK.Length];
    long[] UnsortedChar = new long[GRANK.Length];
    string strFormat = "0#"; // "00#"
    long count = 10;
```
while (HRANK.LongLength > count)
{
    strForamt = "0" + strForamt;
    count = count * 10;
}

for (int i = 0; i < GRANK.Length; i++)
{
    string temp = GRANK[i].ToString() + HRANK[ISA[i]].ToString(strForamt);
    UnsortedChar[i] = Convert.ToInt64(temp);
}

long[] SortedChar = BucketSort.Sort((long[])UnsortedChar.Clone());

for (int i = 0; i < UnsortedChar.Length; i++)
{
    for (int y = 0; y < SortedChar.Length; y++)
    {
        if (SortedChar[y] == UnsortedChar[i])
        {
            gSA[y] = i;
            break;
        }
    }
}

return gSA;

//Step4. GenerateGappedLCP

public int[] GenerateGappedLCP(char[] T, int[] gSA, int g0)
{
    int[] LCP = new int[T.Length];

    for (int i = 1; i < T.Length; i++) //from 1
    {
        int y = 0;
        for (y = 0; y < T.Length - gSA[i] - g0; y++)
        {
            if (gSA[i] + y == T.Length || gSA[i - 1] + y == T.Length)
                break;
            if (y == 1)
                continue;
            if (T[gSA[i] + y] != T[gSA[i - 1] + y])
            {
                break;
            }
        }
        if (y > 1)
        {
            y -= 1;
        }
        LCP[i] = y;
    }
}
```csharp
return LCP;
}

// Step5. Search gapped suffix array
public List<int> SearchUsingLCP(char[] T, int[] gSA, int[] gLCP, char[] X)
{
    List<int> Result = new List<int>();
    int d = 0;
    int f = T.Length - 1;
    int i = (d + f) / 2;
    while (true)
    {
        if (IsContain(T, X, gSA, i))
        {
            Result.Add(i);
            // LCP
            // i 0
            // ippi 1
            // issppi 1 - Current Cusor
            // ississippi 4
            // mi~~ 0
            //Smaller
            for (int y = i; y > 0; y--)
            {
                if (gLCP[y] >= X.Length)
                {
                    Result.Add(y - 1);
                }
                else
                {
                    break;
                }
            }
            //Greater
            for (int y = i + 1; y < T.Length; y++)
            {
                if (gLCP[y] >= X.Length)
                {
                    Result.Add(y);
                }
                else
                {
                    break;
                }
            }
        }
    }
}
```
return Result;

//No Match
if (i == d && i == f)
{
    return Result;
}

//------  2. define the direction to right or to left ------//

//2.1 Left
if (IsBigger(T, X, gSA, i))
{
    //lcp(x,i)==k
    int k = GetLCPLength(T, X, gSA, i);

    //Compute LCP
    for (int y = i; y >= d; y--)
    {
        i = y;
        if (gLCP[i] <= k)
        {
            break;
        }
    }

    f = i;
    i = (f + d) / 2;
}

//2.2 Right
else
{
    //lcp(x,i)==k
    int k = GetLCPLength(T, X, gSA, i);

    //Compute LCP
    for (int y = i + 1; y <= f; y++)
    {
        i = y;
        if (gLCP[i] <= k)
        {
            break;
        }
    }

    d = i;
    i = (f + d) / 2;
}
return Result;
}

public int GetLCPLength(char[] T, char[] X, int[] SA, int i)
{
    for (int y = 0; y < X.Length; y++)
int Gapped_y = y;
if (y > 0)
    Gapped_y++;

if (SA[i] + Gapped_y == T.Length)
    return y;

if (T[SA[i] + Gapped_y] != X[y])
    return y;
}

return X.Length;
}

public bool IsBigger(char[] T, char[] X, int[] SA, int i) {
    for (int y = 0; y < X.Length; y++) {
        int Gapped_y = y;
        if (y > 0)
            Gapped_y++;

        if (SA[i] + Gapped_y == T.Length)
            return false;

        if (T[SA[i] + Gapped_y] < X[y])
            return false;
    }
    return true;
}

public bool IsSame(char[] T, char[] X, int[] SA, int i) {
    if (X.Length != T.Length - SA[i])
        return false;

    for (int y = 0; y < X.Length; y++) {
        int Gapped_y = y;
        if (y > 0)
            Gapped_y++;

        if (SA[i] + Gapped_y == T.Length)
            return false;

        if (X[y] != T[SA[i] + Gapped_y])
            return false;
    }
    return true;
public bool IsContain(char[] T, char[] X, int[] SA, int i)
{
    if (X.Length > T.Length - SA[i])
    {
        return false;
    }
    for (int y = 0; y < X.Length; y++)
    {
        int Gapped_y = y;
        if (y > 0)
            Gapped_y++;
        if (SA[i] + Gapped_y == T.Length)
            return false;
        if (X[y] != T[SA[i] + Gapped_y])
            return false;
    }
    return true;
}

SuffixRank.cs

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace GappedSuffixArray.SuffixArray
{
    public class SuffixRank
    {
        public char[] Pattern;
        public int Characters;
        public int Rank;
        public int SuffixbyRank;
        public int[] SuffixbyRanks;
        public int Index;
        public int SAIndex;
        public int[] MergeRankCase1;
        public int[] MergeRankCase2;
    }
}
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using GappedSuffixArray.SuffixArray.Algorithms;

namespace GappedSuffixArray.SuffixArray
{
    public class SkewAlgorithm
    {
        public int[] GenerateSA(char[] T)
        {
            //Bucket 1,2,3
            List<SuffixRank> Bucket1 = new List<SuffixRank>();
            List<SuffixRank> Bucket2 = new List<SuffixRank());

            //First Bucket and Last
            int[] ISA = new int[T.Length];
            int[] SA = new int[T.Length];
            string[] T_Three = new string[T.Length];

            //Second Bucket
            int[] ISA2;
            int[] SA2;

            //////////////////////////////////////////////////////////////////////
            /////////
            // Step1. split into two list (T mod 3 ≠ 0 = sort the suffixes starting at
            position i ≠ 0 mod3)
            ///////////////////////////////////////////////////////////////////////////
            //////////////////////////////////////////////////////////////////////
            //1.1. split into two lists each.
            for (int i = 0; i < T.Length; i++)
            {
                //First Bucket
                if (i % 3 == 1)
                {
                    SuffixRank bucket = new SuffixRank();

                    //1. Current Index
                    bucket.Index = i;

                    //2. Character
                    bucket.Pattern = new char[3];
                    int count = 3;
                    if (T.Length - i < 3)
                    {
                        count = T.Length - i;
                    }

                    //3. Character
                    bucket.Pattern[0] = T[i];
                    bucket.Pattern[1] = T[i + 1];

                    Bucket1.Add(bucket);
                }
            }
        }
    }
}
for (int y = 0; y < count; y++)
{
    bucket.Pattern[y] = T[i + y];
}
Bucket1.Add(bucket);
T_Three[i] = new string(bucket.Pattern);
}

for (int i = 0; i < T.Length; i++)
{
    //First Bucket
    if (i % 3 == 2)
    {
        SuffixRank bucket = new SuffixRank();

        //1. Current Index
        bucket.Index = i;

        //2. Character
        bucket.Pattern = new char[3];
        int count = 3;
        if (T.Length - i < 3)
        {
            count = T.Length - i;
        }
        for (int y = 0; y < count; y++)
        {
            bucket.Pattern[y] = T[i + y];
        }
        T_Three[i] = new string(bucket.Pattern);
        Bucket1.Add(bucket);
    }
}

//1.2. sort two lists together using Bucket
Sort(http://en.wikipedia.org/wiki/Bucket_sort)

//1.3. sort above suffix array using bucket sort
Bucket1 = this.SortSuffix(Bucket1, T);

for (int i = 0; i < SA.Length; i++)
{
    SA[i] = -1;
}
//Save a rank to int[] Rank
foreach (SuffixRank sr in Bucket1)
{
    SA[sr.SAIndex] = sr.Index;
    ISA[sr.Index] = sr.SAIndex;
}


Step2. sort the suffixes starting at position i = 0 mod 3.
for (int i = 0; i < T.Length; i++)
{
    //First Bucket
    if (i % 3 == 0)
    {
        SuffixRank bucket = new SuffixRank();

        //1. Current Index
        bucket.Index = i;

        //2. Character
        bucket.Pattern = new char[3];
        int count = 3;
        if (T.Length - i < 3)
        {
            count = T.Length - i;
        }
        for (int y = 0; y < count; y++)
        {
            bucket.Pattern[y] = T[i + y];
        }
        T_Three[i] = new string(bucket.Pattern);
        Bucket2.Add(bucket);
    }
}

Bucket2 = this.SortSuffix(Bucket2, T);
//Save a rank to int[] Rank
ISA2 = new int[Bucket2.Count];
SA2 = new int[Bucket2.Count];
foreach (SuffixRank sr in Bucket2)
{
    SA2[sr.SAIndex] = sr.Index;
    //ISA2[sr.Index] = sr.SAIndex;
}

/////////////////////////////////////////////////////////////////////////////////
// Step3. two list should be merged.
/////////////////////////////////////////////////////////////////////////////////

//3.1. Set Case1,2
// Case1: (First character, T[i+1]'s Rank, T[i+2]'s Rank)
// Case2: (First charactor, Second Character, T[i+2]'s Rank)

int CurrentCursor = 0;
for (int i = 0; i < SA2.Length; i++)
{
    for (int y = CurrentCursor; y < SA.Length; y++)
    {
        if (SA[y] == -1) //Empty space(means last)
        {
            SA[y] = SA2[i];
            CurrentCursor = y + 1;
            break;
        }
        if (SA[y] % 3 == 1) //Case1
        {
if (T[SA[y]] > T[SA2[i]])
{
    // Swap
    int Temp1 = SA[y];
    int Temp2 = 0;
    for (int z = y + 1; z < SA.Length; z++)
    {
        Temp2 = SA[z];
        SA[z] = Temp1;
        Temp1 = Temp2;
    }
    SA[y] = SA2[i];
    CurrentCursor = y + 1;
    break;
}
else if (T[SA[y]] == T[SA2[i]])
{
    if (SA[y] + 1 == SA.Length) // IF it is END (no character)
    {
        continue;
    }
    // Second SA's Rank
    else if (SA2[i] + 1 == SA.Length || ISA[SA[y] + 1] > ISA[SA2[i] + 1])
    {
        // Swap
        int Temp1 = SA[y];
        int Temp2 = 0;
        for (int z = y + 1; z < SA.Length; z++)
        {
            Temp2 = SA[z];
            SA[z] = Temp1;
            Temp1 = Temp2;
        }
        SA[y] = SA2[i];
        CurrentCursor = y + 1;
        break;
    }
}
}
if (SA[y] % 3 == 2) // Case 2
{
    // First Character // Second SA Rank
    if (T[SA[y]] > T[SA2[i]])
    {
        // Swap
        int Temp1 = SA[y];
        int Temp2 = 0;
        for (int z = y + 1; z < SA.Length; z++)
        {
            Temp2 = SA[z];
            SA[z] = Temp1;
            Temp1 = Temp2;
        }
        SA[y] = SA2[i];
        CurrentCursor = y + 1;
        break;
    }
}
else if (T[SA[y]] == T[SA2[i]])
{
    if (SA[y] + 1 == SA.Length) //IF it is END (no character)
    {
        continue;
    }
    else if (SA2[i] + 1 == SA.Length || T[SA[y] + 1] > T[SA2[i] + 1])
    //Second Character
    {
        //-> Swap
        int Temp1 = SA[y];
        int Temp2 = 0;
        for (int z = y + 1; z < SA.Length; z++)
        {
            Temp2 = SA[z];
            SA[z] = Temp1;
            Temp1 = Temp2;
        }
        SA[y] = SA2[i];
        CurrentCursor = y + 1;
        break;
    }
    else if (T[SA[y] + 1] == T[SA2[i] + 1])//Chracter is same
    {
        if (SA[y] + 2 == SA.Length) //IF it is END (no character)
        {
            continue;
        }
        else if (SA2[i] + 2 == SA.Length || ISA[SA[y] + 2] > ISA[SA2[i] + 2]) //Compare Rank
        {
            //-> Swap
            int Temp1 = SA[y];
            int Temp2 = 0;
            for (int z = y + 1; z < SA.Length; z++)
            {
                Temp2 = SA[z];
                SA[z] = Temp1;
                Temp1 = Temp2;
            }
            SA[y] = SA2[i];
            CurrentCursor = y + 1;
            break;
        }
    }
}
}

//return SA
return SA;

private List<SuffixRank> SortSuffix(List<SuffixRank> Bucket, char[] T)
long[] UnsortedChar = new long[Bucket.Count];
int[][] UnsortedRanks = new int[Bucket.Count][];
for (int y = 0; y < Bucket.Count; y++)
{
    string temp = "";
    for (int i = 0; i < Bucket[y].Pattern.Length; i++)
    {
        if (Bucket[y].Pattern[i] == 0)
        {
            temp += "000";
        }
        else
        {
            temp += ((int)Bucket[y].Pattern[i]).ToString("0##");
        }
    }
    Bucket[y].Characters = Convert.ToInt32(temp);
    UnsortedChar[y] = Convert.ToInt32(temp);
}

// Bucket Sort
UnsortedChar = BucketSort.Sort(UnsortedChar);

// Making Ranking
for (int y = 0; y < Bucket.Count; y++)
{
    for (int i = 0; i < UnsortedChar.Length; i++)
    {
        if (Bucket[y].Characters == UnsortedChar[i])
        {
            Bucket[y].Rank = i + 1;
        }
    }
}

// Making Suffix Ranking
for (int y = 0; y < Bucket.Count; y++)
{
    Bucket[y].SuffixbyRanks = new int[Bucket.Count];
    for (int i = 0; i < Bucket.Count - y; i++)
    {
        Bucket[y].SuffixbyRanks[i] = Bucket[y + i].Rank;
    }
    //Attach Zero
    for (int z = Bucket.Count - y; z < Bucket.Count; z++)
    {
        Bucket[y].SuffixbyRanks[z] = 0;
    }
    UnsortedRanks[y] = Bucket[y].SuffixbyRanks;
BucketSort.cs

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace GappedSuffixArray.SuffixArray.Algorithms
{
    class BucketSort
    {
        public static long[] Sort(long[] integers)
        {
            //Verify input
            if (integers == null || integers.Length <= 1)
            return null;

            //Find the maximum and minimum values in the array
            long maxValue = integers[0]; //start with first element
            long minValue = integers[0];

            //Note: start from index 1
            for (long i = 1; i < integers.Length; i++)
            {
                if (integers[i] > maxValue)
                maxValue = integers[i];

                if (integers[i] < minValue)
                minValue = integers[i];
            }
        }
    }
}
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace GappedSuffixArray.SuffixArray.Algorithms
{
    //Worst case: O(KN)
    public class RadixSort
    {
        public static int[][] Sort(int[][] T, long TotalCount)
        {
            Dictionary<int, List<int[]>> Bucket = new Dictionary<int, List<int[]>>();

            //Create a temporary “bucket” to store the values in order
            //each value will be stored in its corresponding index
            //scooting everything over to the left as much as possible (minValue)
            //e.g. 34 => index at 34 - minValue
            LinkedList<long>[] bucket = new LinkedList<long>[maxValue - minValue + 1];

            //Move items to bucket
            for (long i = 0; i < integers.Length; i++)
            {
                if (bucket[integers[i] - minValue] == null)
                    bucket[integers[i] - minValue] = new LinkedList<long>();

                bucket[integers[i] - minValue].AddLast(integers[i]);
            }

            //Move items in the bucket back to the original array in order
            long k = 0; //index for original array
            for (long i = 0; i < bucket.Length; i++)
            {
                if (bucket[i] != null)
                {
                    LinkedListNode<long> node = bucket[i].First; //start add head of linked list

                    while (node != null)
                    {
                        integers[k] = node.Value; //get value of current linked node
                        node = node.Next; //move to next linked node
                        k++;
                    }
                }
            }

            return integers;
        }
    }
}
long ItemLength = T[0].LongLength;
long ItemCount = T.LongLength;

for (int i = -1; i < ItemLength + 1; i++)
{
    Bucket.Add(i, new List<int[]>());
}

for (long i = ItemLength - 1; i >= 0; i--)
{
    for (long y = 0; y < ItemCount; y++)
    {
        Bucket[T[y][i]].Add(T[y]);
    }

    int Count = 0;
    //Fill Again
    for (int y = 0; y < ItemLength + 1; y++)
    {
        foreach (int[] c in Bucket[y])
        {
            T[Count] = c;
            Count++;
        }
        Bucket[y].Clear();
    }
}

return T;

}
public MainWindow()
{
    InitializeComponent();
}

// T
Char[] T;

// SA
int[] SA;

// ISA
int[] ISA;

// LCP
int[] LCP;

// Gapped T
Char[] gT;

// gSA
int[] gSA = null;

// LCP
int[] gLCP;

// GRANK
int[] GRANK;

// GRANK - Character
char[] GRANK_T;

// LCP
int[] HRANK;

// g0, g1
int g0 = 1;
int g1 = 2;

private void btnCommand_Click(object sender, EventArgs e)
{
    GenerateSuffix();
    GenerateGappedSuffix();
}

private void GenerateGappedSuffix()
{
    SuffixArray.GappedSuffixArray gSAEngine = new SuffixArray.GappedSuffixArray();

    //---------------------
    // 0. Step - Length up to g0
    GRANK_T = new Char[T.Length];
    for (int i = 0; i < T.Length; i++)
    {
        GRANK_T[i] = T[i];
    }
1. Step - Get GRANK

GRANK = gSAEngine.GenerateGRANK(GRANK_T);

2. Step - Get HRANK

HRANK = gSAEngine.GenerateHRANK(SA, ISA, g0, g1);

3. Step - Generate gSA

gSA = gSAEngine.GenerateGappedSuffix(GRANK, HRANK, ISA);

4. Step - Generate gLCP

gLCP = gSAEngine.GenerateGappedLCP(T, gSA, g0);

//Binding UI

lvGappedSuffix.Items.Clear();
for (int i = 0; i < T.Length; i++)
{
    //Index
    ListViewItem item = new ListViewItem(i.ToString());

    //T
    StringBuilder strT = new StringBuilder(T.Length - i);
    for (int y = i; y < T.Length; y++)
    {
        strT.Append(T[y].ToString());
    }
    item.SubItems.Add(strT.ToString());

    //gSA
    item.SubItems.Add(gSA[i].ToString());

    //gSA - T
    strT = new StringBuilder(T.Length - gSA[i]);
    for (int y = gSA[i]; y < T.Length; y++)
    {
        if (y == gSA[i] + 1)
            continue;
        strT.Append(T[y].ToString());
    }
    item.SubItems.Add(strT.ToString());

    //GRANK
    item.SubItems.Add(GRANK[i].ToString());

    //HRANK
    item.SubItems.Add(HRANK[i].ToString());

    //LCP
    item.SubItems.Add(gLCP[i].ToString());
    lvGappedSuffix.Items.Add(item);
}
private void GenerateSuffix()
{
    // T
    T = txtPattern.Text.ToCharArray();

    // SA
    SuffixArray.SuffixArray sa = new SuffixArray.SuffixArray();
    SA = sa.GenerateSA(T);

    // ISA
    ISA = sa.GenerateISA(SA);

    // LCP
    LCP = sa.GenerateLCP(SA, T);

    // Binding
    lvSuffix.Items.Clear();
    for (int i = 0; i < T.Length; i++)
    {
        // Index
        ListViewItem item = new ListViewItem(i.ToString());

        // T
        StringBuilder strT = new StringBuilder(T.Length - i);
        for (int y = i; y < T.Length; y++)
        {
            strT.Append(T[y].ToString());
        }
        item.SubItems.Add(strT.ToString());

        // SA
        item.SubItems.Add(SA[i].ToString());

        // SA - T
        strT = new StringBuilder(T.Length - SA[i]);
        for (int y = SA[i]; y < T.Length; y++)
        {
            strT.Append(T[y].ToString());
        }
        item.SubItems.Add(strT.ToString());

        // ISA
        item.SubItems.Add(ISA[i].ToString());

        // LCP
        item.SubItems.Add(LCP[i].ToString());
        lvSuffix.Items.Add(item);
    }
}

// aabaabaabba
private void btnSearch_Click(object sender, EventArgs e)
{
    if (T == null)
    {
        GenerateSuffix();
        GenerateGappedSuffix();
    }
}
if (txtSearch.Text.Trim() == "")
{    
    txtLog.Text = "Please input the search text";
    return;
}

SuffixArray.SuffixArray sa = new SuffixArray.SuffixArray();
List<int> Result = sa.SearchUsingLCP(T, SA, LCP,
txtSearch.Text.ToCharArray());

//For Order
List<int> ResultT = new List<int>();
foreach (int i in Result)
{
    ResultT.Add(SA[i]);
}

if (Result.Count == 0)
{
    txtLog.Text = "Cannot find the match";
}
else
{
    txtLog.Text = "";
    ResultT = ResultT.OrderBy(x => x).ToList();
    foreach (int i in ResultT)
    {
        if (!String.IsNullOrEmpty(txtLog.Text))
        {
            txtLog.Text += "\r\n";
        }
        else
        {
            txtLog.Text += "Found at \r\n";
        }
    txtLog.Text += "T[" + i + "] ";
    }
}

private void txtLog_TextChanged(object sender, EventArgs e)
{
    txtLog.SelectionStart = txtLog.Text.Length;
    txtLog.ScrollToCaret();
}

private void btnGappedSuffix_Click(object sender, EventArgs e)
{
    if (txtGappedSuffix.Text.Length != 3)
    {
        txtLog.Text = "Please input the search text of three length";
        return;
    }
}
if (T == null)
{
    GenerateSuffix();
    GenerateGappedSuffix();
}

string strCase0 = txtGappedSuffix.Text.Substring(0, 2);
string strCase1 = txtGappedSuffix.Text.Substring(1, 2);
string strCase2 = txtGappedSuffix.Text.Substring(0, 1) +
    txtGappedSuffix.Text.Substring(2, 1);
    //string strCase2 = txtGappedSuffix.Text;

////////////////////////////////////////////////////////////////////////////////
// Step1. Suffix Search - g0g1, g1g2
SuffixArray.SuffixArray sa = new SuffixArray.SuffixArray();
List<int> Result = sa.SearchUsingLCP(T, SA, LCP, strCase0.ToCharArray());
List<int> ResultT = new List<int>();
foreach (int i in Result)
{
    if (!ResultT.Contains(SA[i]))
    {
        ResultT.Add(SA[i]);
    }
}
Result = sa.SearchUsingLCP(T, SA, LCP, strCase1.ToCharArray());
foreach (int i in Result)
{
    if (!ResultT.Contains(SA[i]))
    {
        ResultT.Add(SA[i]);
    }
}

////////////////////////////////////////////////////////////////////////////////
///
// Step2. Suffix Search - g1g2
SuffixArray.GappedSuffixArray gSAEngine = new
SuffixArray.GappedSuffixArray();
Result = gSAEngine.SearchUsingLCP(T, gSA, gLCP, strCase2.ToCharArray());
foreach (int i in Result)
{
    if (!ResultT.Contains(gSA[i]))
    {
        ResultT.Add(gSA[i]);
    }
}

txtLog.Text = "";
ResultT = ResultT.OrderBy(x => x).ToList();
foreach (int i in ResultT)
{
if (!String.IsNullOrEmpty(txtLog.Text))
{
    txtLog.Text += "\r\n";
}
else
{
    txtLog.Text += "Found at \r\n";
    txtLog.Text += "T[" + i + "]";
}
if (ResultT.Count == 0)
{
    txtLog.Text = "Cannot find";
}
}