Implementation of SA-IS Suffix Array Construction Algorithm

Submitted by: Anirudh Subramanian
UFID : 94453124

Introduction

Suffix arrays are considered to be fundamental data structures in applications such as data indexing , retrieving , storing and processing. They are particularly used within the field of bioinformatics to index large strings of genome sequences and querying the same. Suffix arrays allow for improved space requirements , easier linear time construction algorithms and improved cache locality. My project was to implement and analyze SAIS which is a linear time suffix array construction algorithm and compare it with other linear time construction algorithms KA and KS as is done by the authors of the paper [1].

Motivation for SAIS

The KS and KA[2, 3] algorithm were published around 2003 and guaranteed a linear running time for suffix array construction. Both algorithms followed a generic framework for their solutions which is as mentioned below:

1. The initial input string is reduced to a smaller string so that the original problem divided into the reduced part and the remaining part.
2. Suffix array of the reduced problem is recursively computed
3. Based on result of previous step suffix array of remaining problem is induced.
4. Finally two arrays are merged as final result.

The KA algorithm and KS algorithm differ in the way they perform the above steps.
In the 1st step the reduction is done by KS algorithm by using triplets such that the starting index is not equal to 0. Thus the problem size is reduced to 2/3rd of the original problem in KS. The KA algorithm uses L and S substrings for problem reduction. So for KA the reduction ratio at max can be ½ by symmetric definition of S and L substrings.

Thus the recurrence relation for KS is expressed as
$T(N) = T(2N/3) + O(N)$

For KA, the recurrence relation can be expressed as
$T(N) = T(N/2) + O(N)$

The better reduction ratio suggests that KA will use less space and be faster which is found true in many performance evaluations.
Although KA runs faster than KS, in practice the design for KA algorithm is far more complicated than KA which samples and sorts the substring with radix sort. KA uses S distance lists which adds to design complexity , execution time and memory used.

Thus SAIS tries to overcome the design complexity of KA while maintaining the ½ maximum reduction ratio. For this SAIS uses LMS substrings to reduce the problem and then does induced sorting of LMS substrings. Below I have described the SAIS algorithm.

SAIS Algorithm description

- suf(S,i) : suffix starting at S[i] and running till the end
- S type suffix : A suffix suf(S, i) is said to be S type if suf(S, i) < suf(S, i + 1)
- L type suffix : A suffix suf(S, i) is said to be L type if suf(S, i) > suf(S, i + 1)
- S[i] will be L type if suf(S, i) is L type

  and S[i] will be S type if suf(S, i) is S type
- S[i] is S type if S[i] < S[i + 1] or S[i] = S[i+ 1] and suf(S, i + 1) is S type
- LMS Character : A character S[i], i in [1, n-1] is called LMS if S[i] is S type and S[i - 1] is L type.
- LMS Substring : An LMS Substring is a (i)substring S[i..j] with both S[i] and S[j] being LMS characters and there is no LMS character in the substring for i != j or (ii) the sentinel itself

- **Ordering**

For the induced sorting phase we need to find a way to order the LMS substrings because we need to name that the end of induced sorting phase to reduce the problem.

The ordering of two LMS substrings are same only if their length type and characters are same. Otherwise character by character comparison is done and if they are lexicographically same, compare their types where S type is of higher priority.

```
SA-IS(S, SA)
     ▷ S is the input string;
     ▷ SA is the output suffix array of S;
     t: array [0..n − 1] of boolean;
     P1, S1: array [0..n1 − 1] of integer; ▷ n1 = ||S1||
     B: array [0..||Σ(S)|| − 1] of integer;
 1   Scan S once to classify all the characters as
     L- or S-type into t;
 2   Scan t once to find all the LMS-substrings in S into P1;
 3   Induced sort all the LMS-substrings using P1 and B;
 4   Name each LMS-substring in S by its bucket
     index to get a new shortened string S1;
 5   if Each character in S1 is unique
 6       then
 7            Directly compute SA1 from S1;
 8       else
 9            SA-IS(S1, SA1); ▷ Fire a recursive call
10   Induce SA from SA1;
11   return
```

- **Induced Sorting Algorithm**

This is a brief overview of the algorithm. For more details please refer to the paper[1]

1. Do the initialization of Suffix Array SA . Set all elements to -1. Do one scan of S and put all LMS substrings into the buckets. Here the buckets are arranged by their start character which will be the correct order in the suffix array. So if a string has ‘BBACDEE$’ then the bucket for A will come first(comprimising of all substrings starting with A) after which will be B and then C, D and then E in the suffix array.

2. Induced Sort all the LMS prefixes of L type. Scan SA from left to right, for each nonnegative item SA[i], if S[SA[i] - 1] is L type put SA[i] - 1 in the respective position at current head and shift current head right.

3. Induced Sort all LMS prefixes using the L type LMS prefixes in 2.

Time Complexity and Space Complexity for SAIS

The time complexity of SAIS is O(n) where n is size of input string. This is true as the recurrence equation of SAIS is T(N) = T(N/2) + O(N)

The worst space complexity is O(NlogN) bits because initially space occupied is NLogN bits and every recursion stage it goes down by half.

Implementation Details

The implementation was done in Java and was compiled and run on Java 1.6.3

- **Data structures used**

1. count data structure to maintain count of the substrings in buckets. Hash table of character to count mapping.
2. bucket data structure which holds start or end of bucket depending on the stage. This is a hash table of character to index mapping.
3. Both these data structures could have been clubbed together but that would have resulted in consumption of extra memory due to java's overhead of an object for storing housekeeping information.
4. A bitvector instead of Boolean array to store the types i.e. L types or S types because it is more space efficient.

- **Modules used**

SAISComputation

This module is the entry point and returns the final suffix array. This does the whole processing as mentioned in the figure in the algorithm section. Initialization of the bit vectors, and the substring pointers, and then induced sorting and recursive call of SAIS. Finally the result array is induced from the sorted array.

induceSort

Does the induced sorting of the array as mentioned above. Follows the three step algorithm to scan and insert the LMS suffixes, then sorts all L type LMS prefixes into their buckets and does the same for all prefixes in the third step.

induceSA

The induce SA algorithm works very similar to the induceSort algorithm. Only the first step differs in a way that it extracts the SA substrings from SA1 instead of putting them in SA.

Performance Evaluation

A python script was written to do the two tests and log the time and memory usage using linux time command and memusage command
I did two sets of tests :

Test 1

- Environment : Experiments run on Dual Core AMD Opteron Processor 2 cores, 16GB RAM
- Dataset: Manzinis Large Dataset [4]
- Wanted to test on general applications and not specific to bioinformatics

Results

Fig 1

Execution Time in seconds				
Data	Size	SAIS	Difference Cover	KA
rfc	111MB	70.54	154.33	117.04
sprot34.dat	105MB	67.424	157.1	119.79
etext99	100MB	63.334	124.1	126.81
chr22.dna	33MB	19.32	33.18	30.57
rctail96	109MB	59.86	198.6	124.71
jdk13c	66MB	31.026	104.72	64.61
howto	38MB	19.877	37.15	37
w3c2	99MB	44.957	148.17	99.09
gcc-3.0.tar	83MB	44.632	96.79	75.5

linux-2.4.5.tar	111MB	57.67	131.25	107.85

Fig 2

		Space in MB			
Data	Size	SAIS	Difference Cover	KA	SAIS/Size
rfc	111	699	1223	976	6.30
sprot34.dat	105	659	1151	937	6.28
etext99	100	633	1106	913	6.33
chr22.dna	33	208	363	289	6.30
rctail96	109	688	1205	1016	6.31
jdk13c	66	401	733	622	6.08
howto	38	218	415	333	5.74
w3c2	99	627	1090	926	6.33
gcc-3.0.tar	83	524	910	726	6.31
linux-2.4.5.tar	111	691	1221	992	6.23
				Mean ratio	6.22

Deductions

The SAIS algorithm has a reduction ratio of ⅓ if the probability of S and T types characters are equal and if this is satisfied it will achieve a very good performance in terms of execution time.Although it will be O(n) the constant will be small. Because with large number of characters we can expect characters to be disordered the probability of S and T type characters will be very similar unless for exceptional cases. Thus we can observe the good execution time in comparison with KA and KS algorithm.

In regards to memory consumption, the worst case memory consumption is O(nlogn) but it consumes very less practice in practice.

It is proved in the paper that if the S and T types has almost similar probabilities and the alphabet size is constant (there can be only constant number of different alphabets) it will have a space consumption of O(n) + O(1) bits which is what is observed.

Test 2

- Environment : Experiments run on AMDV Processor 64 cores, 512GB RAM
- Dataset: Human chromosome and its copies [5]
- Wanted to test human genome which is specific to bioinformatics

Results

Human CHromosome and its copies				
Data	**Size in MB**	**SAIS(Memory)**	**SAIS - Execution Time**	**SAIS/Size**
chr19.fa.masked	**58**	**336**	**18.81**	**5.79**
chr18.fa.masked	**76**	**430**	**34.19**	**5.66**
chr22.dna	**33**	**208**	**19.32**	**6.30**
chr20.fa.masked	**62**	**356**	**22.27**	**5.74**
chr21.fa.masked	**47**	**281**	**17.38**	**5.98**
chr2.fa.masked	**237**	**1487**	**82.3**	**6.27**
chr1.fa.masked	**243**	**1466**	**84**	**6.03**
chr3.fa.masked	**193**	**1168**	**73.2**	**6.05**
chr5.fa.masked	**176**	**1112**	**54.86**	**6.32**
			Mean ratio	**6.02**

Deductions

The ratio taken here is the sais space consumption / file_size . This shows what is the c in the space consumption c*n. From the theoretical results we expect it to be better than the previous tests because there are only 4 alphabets A, C, G, T in this test. Although there is not a great difference the mean ratio has improved.

Thus the SAIS performs linearly with comparable space requirements for suffix array constructions in the field of bioinformatics too. The problem still remains that even for a file size of 2GB the space requirements are too high to run it on a single machine. Thus in practice suffix array has to be run in a distributed system where array is stored in different machines and they work together to find a match. If there is a better O(1) extra space algorithm it would be better suited for very large strings.

References

1. **ieeexplore.ieee.org/iel5/12/4358213/05582081.pdf?arnumber=5582081 (SAIS)**
2. **http://www.cs.cmu.edu/~guyb/realworld/papersS04/KaSa03.pdf (KS)**
3. **http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.83.9781 (KA)**
4. **http://people.unipmn.it/~manzini/lightweight/corpus/ (Manzini Dataset)**
5. **http://hgdownload.cse.ucsc.edu/goldenPath/hg19/bigZips/ (Human chromosome and its copies)**