Loren McGinnis
Asst1
CS265

Paper Critique

"Improving Suffix Array Locality for Fast Pattern Matching on Disk," by R. Sinha et al, clearly presents its purpose and motivation to the reader. The paper is easy to read, and the data problem it attempts to solve is very understandable: to make a suffix array data structure run pattern matching *queries* faster for very large strings (e.g. the human genome). The authors emphasize that the majority of suffix tree/suffix array research has been devoted to quickly *constructing* the data structure, since by comparison querying a suffix array is relatively straightforward. However, with long strings, the implication is that the entire suffix array will not fit into main memory, so steps must be taken to make the disk access as inexpensive as possible. The "straightforward" binary search on an in-memory suffix array does not extend well to disk version because of poor locality. This paper can be seen as a compromise between two paradigms (a binary search, which has poor locality, vs an exhaustive search, which has poor scalability). It is an attempt to find the "sweet spot" that will outperform either extreme.

The solution introduced to address the locality drawbacks involves a hybridization of the on-disk suffix array with a truncated suffix tree, collectively coined the LOF-SA. Since reading data from disk is much faster if it is contiguous (which a binary search is not), reducing the search space to a manageable section of the on-disk suffix array will help mitigate the cost of running a linear search. This is the purpose of a truncated trie/suffix tree: if it can be put in memory (which the authors usual did), random access to the tree nodes during traversal is essentially free, and narrows down the possible range on disk where the suffix array must be searched. The truncation is done by making all nodes with a certain number of descendants the leaf nodes, removing the descendants from the tree and just maintaining a range of indices into the array. The suffix array also holds some number *f* of "fringe" characters from each suffix, a construct intended to reduce the number of times the actual text must be accessed (which is stored separately from the suffix array).

This paper not only provides an interesting solution, but also makes it approachable with clear and intuitive notation. Even for one who has had little exposure to the concepts behind tries, suffix trees, and suffix arrays, the paper is understandable, for it gives clear definitions for each. It also uses these definitions consistently, allowing the reader to follow the where and how each concept is used in the holistic structure. This is important because the solution is not particularly elegant; it increases implementation complexity by introducing layers to the data structure that span across multiple layers of memory. It is commendable that the paper handles this in a comprehensible way.

The paper also does a good job of outlining the experimental methodology the authors employed in testing out the LOF-SA. The datasets used were DNA, Protein Alignment, and English text from the Wall Street Journal. This seems like a good cross-section covering all the types of data where this type of pattern matching is applicable. The tests used file system buffering to read from the disk-resident components of the data structure, which helps with speeding up sequential access, but makes getting reliable results more difficult, since the experiments exerted no control over the buffer behavior (other than labeling some test runs as "cold," when the buffer was flushed beforehand). This is where the paper begins to lose credence, however; as the results of various tests are shown, justification for those results seem vague or arbitrary. Some of the results seem very compelling—for example, Table 6 is a listing of how many "per-query non-sequential" data accesses were made for each implementation, with the LOF-SA far outranking other implementations. But the authors provide no explanation as to how they calculated the metrics for this table.

Another issue is the relevance of all the data as a whole. The largest dataset that the paper does any testing with is two gigabytes. However, in real world applications of suffix arrays, conceivably these data structures are residing on large servers with plenty of main memory to contain the whole suffix array of a two gigabyte dataset. In short, the authors needed to test much larger datasets, large enough to clearly eclipse the size of any potential main memory setup. In fact, it is quite clear that the implementation cannot scale above four gigabytes, since it uses 32-bit based data types.

Despite these facts, the paper does present a compelling argument, even if just conceptually, that disk-based suffix array performance can improve significantly when due attention is given to good locality on disk. The authors have essentially extended the concepts of hierarchical memory to the suffix data structure, and they deserve notoriety for approaching the problem from a different angle than previous work has taken.

Reproducing Results

The results that my experimentation has tried to replicate are from Table 2 in the paper, which catalogs the frequencies of random substrings in large datasets. According to the paper, these results came from selecting a thousand random strings of a certain length *L* from the text, then querying the suffix array to get the number of occurrences of each string. In each cell of Table 2, then, is the summation of the occurrences of the 1000 strings of the given length, taken from a particular corpus.

In lieu of the LOF-SA implementation, which I was unable to obtain*, I used another suffix array implementation mentioned in the paper, TDD, to search the data. Assuming both are implemented correctly, using TDD instead of LOF-SA should not change the results, since the numbers in Table 2 are a function of the dataset, not the method in which it is queried. Additionally, the results are independent of hardware, for the same reason, and since no performance testing is involved. However, I had difficulty getting the TDD implementation to index strings over 200M in size (for the paper, TDD choked over 400M), so I could not repeat every column in Table 2. I was able to obtain a few of the same datasets used in the paper, and used these to produce my results.

I employed several scripts to aid in generating the data. First, I used *tdd.pl* from the TDD distribution to index a corpus, using the default options. In a script of my own, I used a Mersenne Twister randomization to select random query strings from the text. For each string, I used the *simplequery* script, also from TDD, to query the number of occurrences of the string. The following table summarizes my results:

**Ironically, I received an email hours before this assignment was due from one of the authors about getting the code for the LOF-SA.*

L	DNA100M	DNA200M	PROT	DNA100MR	DNA200MR
6	38407552	86625600	242568	24443036	49251344
8	3305453	9257389	199543	1530233	3072976
10	738573	1004432	245824	96401	193032
12	91605	208725	2134761	6831	12920
16	75488	157280	40104	1033	1040
20	35552	86421	19013	1000	1000
24	17189	43728	130344	1000	1000
32	1232	9173	29811	1000	1000
40	3160	7224	400	1000	1000
100	1024	1045	146967	1000	1000
200	1003	1001	1329	1000	1000
400	1001	1000	1145	1000	1000
1000	1001	1000	1004	1000	1000
Avg Query Time	123ms	154ms	133ms	98ms	109ms

The first column is the length L of the strings used to query the suffix array, and in every case, I used 1000 strings. The first three datasets are drawn straight from table 2 of the paper. The data turned out similarly to the paper's. PROT was degenerate in both cases, giving extremely large values for arbitrary string lengths, and the paper explains that there are many long sequences of repetition in the dataset that causes the skew. I additionally added, **DNA100MR** and **DNA200MR** are datasets that I generated with a random distribution, using another tool from TDD, *dnagen.*

When I benchmarked TDD (the last line in the above table), I used the same script, *simplequery,* to search for strings, but also used the UNIX command *time* to get the user time for each query, then averaged it across the runs to get the results in the table. I ran everything on my laptop, with a 1.6 GHz Intel i7 CPU, 4GB of RAM, A 6MB L2 Cache, running Ubuntu 10.4.

Critique Addendum

For the most part my results were similar to that of the papers. As expected, the two real DNA datasets produced a similar number of occurrences as they did in the paper. The PROT dataset gave degenerate results in the paper, and did for me as well. The differences between DNAxM and DNAxMR indicate that the real DNA data has significantly more repetition. As the data show, with increasing string lengths in a purely random distribution, the likelihood of two

substrings in a string of length n is $\frac{n}{|\Sigma|^L}$. In the 100MR dataset, for an alphabet of size 4 and L=16 that probability is 2.32%, and .009% for L=20. Thus, everything over 16 resulted in no repetition.

Interestingly, the type of data also affected the performance of TDD, even when the data size remained constant. The related data can be found in Figures 6 and 7 in the paper. Figure 6 shows that when running on DNA100M and DNA200M, TDD queries took just over 100ms in both cases. My results seemed to behave similarly, being off by only a constant factor of about 10-20%. I only show one value for each dataset, because like in the paper, there was no appreciable difference in TDD run-times based on string length (Figure 7). The differences in run-times between DNAxM and DNAxMR are difficult to explain. While the latter simply generated less occurrences of strings, TDD seemed to be unaffected by that in other cases (for a related reason to the string length constancy in Figure 7 – various string lengths produced occurrences that differed by many orders of magnitude, but that did not affect the run-times).

The differences led me to wonder about what datasets the paper decided to use for benchmarking. Other than Table 2, the only instances where the paper benchmarks the non-DNA datasets are in tables 4, 8 and 9. None of these tables discuss any time-related metrics; all of the time testing is done against DNA data. This indicates that the LOF-SA may be more special purpose than the authors have indicated. DNA seems to be an ideal situation for the hybrid trie-array structure, since the common prefixes are much longer. This makes the trie structure deep, instead of broad. In turn, more of a substring is queried in the trie (which was usually loaded into memory for the paper's benchmarking), and less disk access is required. In a very broad trie (like the WSJ corpus would produce), shorter in memory prefixes would cause more disk access for the remainder of the substring. Thus, while I do not necessarily doubt the results that are shown in the paper, I think that a more comprehensive testing of various data set types would give be a better understanding of the true performance of the LOF-SA, and help answer these questions about how it responds in the worst-case scenario.

If I had the opportunity to run tests against the LOF-SA implementation, I would expect only certain parts of the data to be easily reproducible. The data that describes a holistic view of

performance, simply giving average query times for the implementation or memory usage, seem straightforward. However, the paper did not give clear explanation as to how some of the other data was produced. For example, Table 6 gave a breakdown of how often “non-sequential data accesses” were made, but the authors did not explain how they measured these values. Despite these lingering questions, the paper as a whole gave a fairly convincing testimony as to the benefits of exploiting locality when implementing a suffix tree.