6.851 Advanced Data Structures (Spring '14)

$\boxed{\text{Project Proposal}}$

By Casey O'Brien (cmobrien@mit.edu) and Hayden Metsky (hmetsky@mit.edu)

An Experimental Comparison of Two Recent Data Structures for String Matching with Errors

1 Introduction

In this project we will consider the generalized string matching problem in which, given a pattern p and text t, we search for p in t with error bounds. In particular, we preprocess (or "index") t so that we can quickly look for substrings of t that match p to within some edit distance $k = O(1)$. Solutions to this problem have wide-ranging applications, such as searching error-ridden text or performing a biological sequence analysis.

There have been many theoretical contributions to this problem. In 2004, Cole et al. showed a data structure with worst-case index size $O(n \log^k n)$ and worst-case query time $O(m + \log^k n \log \log n + \#\text{ matches})$, where $n = |t|$ and $m = |p|$ [1]. In 2005, Maaß and Nowak showed a data structure with index size on average $O(n \log^k n)$ w.h.p. and worst-case query time $O(m + \#\text{ matches})$; they also showed a small modification to support worst-case index size $O(n \log^k n)$ and average query time $O(m + \#\text{ matches})$ w.h.p. [2].

It is not clear from the literature how these data structures perform in practice. For example, the exponential dependence on k in the bounds on size may render both structures impractical for many purposes but leave them practical for small k. We therefore will first implement these structures independently, and test and tune their practical performance. Furthermore, these two data structures clearly have similar bounds in theory, but constant factors may cause them to perform very differently in practice, especially when comparing them for different values of n, m, and k. Thus, we will also compare the performance of the two with real data sets.

2 Overview of data structures

Cole et al. presents a way to search for matches with some edit distance by modifying a single suffix tree [1]. Additional subtrees are recursively added to the suffix tree in order to handle errors. By decomposing the tree into centroid paths, they are able to create and store these subtrees efficiently. Using this structure and carefully constructed longest common prefix queries, they achieve query time $O(m + \log^k n \log \log n + \#\text{ matches})$.

While Cole et al. provides worst-case upper bounds, Maaß and Nowak improve on these bounds in the average-case [2]. Their fundamental idea is to store $k + 1$ sets, each consisting of (modified) suffixes of t, that allow a query for prefixes of these suffixes that have different error bounds up to k. In particular they store each of these sets in what they call an "error tree" (a type of trie) in which the ith has height h_i for $0 \le i \le k$. The trees are inductively constructed such that the ith tree has $i + 1$ errors before $h_i + 1$ in its suffixes. Then, loosely speaking, they search for p in each of the error trees. Depending on how the parameters h_i are set, the authors achieve either a worst-case bound for the index size or for the query time (and an average-case bound for the other). Because it is more straightforward to conduct a single pairwise comparison, we will only consider the version of this data structure with worst-case index size $O(n \log^k n)$ and average query time $O(m + \# \text{ matches})$ w.h.p.; note that, in this version, the index size has the same worst-case bound as the data structure provided by Cole et al.

3 Implementation and tests

We will implement both data structures in C. Once the structures have been implemented, we will begin to compare them across a variety of values for n, m, and k. A particularly interesting question is how they perform when p is long, namely $m = \Omega(\log^k n)$, because at this point both structures have equivalent query bounds. We are also interested to see how the structures are able to scale for larger values of k. We will compare both in terms of query time and space (note that they both use space $O(n \log^k n)$). Finally, we plan to test, for each structure, how large n and k can grow before the structure takes up too much memory for our computers to handle.

In performing these tests, we will draw on applications in computational biology. In particular, we will let t be the human genome, which gives our text a fixed alphabet size of 4. If we find that t is too large to store or to search, we can easily reduce n by selecting a segment of it (e.g., a chromosome). Our string patterns will consist of known sequence motifs. Motifs are rarely a fixed sequence of bases; instead, they are typically a pattern in which the base at certain indices could be one of a number (≤ 4) of possibilities. For example, `ATCG` could be an instance of the same motif as `ATCC`, where the fourth index takes on `G` or `C`. Hence, searching for motifs in a large sequence is a natural application of string matching with errors. Of course, in this application it is highly unlikely that we would find any match for most values of m (even $m = \Omega(\log n)$ would be too large). To test and compare the data structures for larger values of m, we can artificially alter the genome to ensure matches.

4 Conclusion

By implementing these data structures we hope to gain a better understanding of how the problem of string matching with errors can be solved in practice, and also to see the limitations of current solutions. We anticipate that neither data structure will completely dominate the other, and thus hope to determine the circumstances under which each structure performs best.

References

[1] Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In *Proc. 36th ACM Symposium on Theory of Computing (STOC)*, pages 91–100, 2004.

[2] Moritz Maaß and Johannes Nowak. Text indexing with errors. In *Proceedings of the 16th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 21–32, 2005.