TEXT INDEXING WITH WILDCARDS: AN EXPERIMENTAL COMPARISON

HAYDEN METSKY (HMETSKY@MIT.EDU) AND CASEY O'BRIEN (CMOBRIEN@MIT.EDU)

1. Introduction

Consider the text indexing problem. Given an input text and query string, we wish to report all the indices of the input text to which the query string matches. In this paper we will consider a generalized version of this problem, in which we allow the query string to contain *wildcards*. A wildcard is a character which can be matched to any other character. We will discuss four possible approaches to this problem. The first two are naive solutions, one of which uses little space but has a slow query time, and the other which has a fast query time but uses a lot of space. Then, we will discuss two approaches presented by Cole et. al in 2004 [1].

While theoretical bounds for each of these solutions have already been established, our goal is to report the performance of each of these solutions in practice. To this end, we implemented each of them in Java. For each solution, we measured the total space and query time for various texts and query patterns, allowing varying numbers of wildcards.

In terms of space, the solutions performed as expected based on their theoretical bounds. Perhaps more interestingly, the actual performances of the query algorithms differed substantially from what was suggested by their theoretical runtimes. In particular, both queries by Cole et. al performed worse than the naive approaches, despite having much better worse case bounds in theory.

1.1. Organization of the Report.

In this section, we will present an overview of each of the solutions as well as their theoretical bounds. In Section 2, we will discuss the details of the implementation of our structures and the methods we used to measure their performance. In Section 3, we present the results of our experimentation, and finally in section 4 we discuss the implications of these results.

1.2. Preliminaries.

Throughout this paper, for ease of notation, we will interchangeably use t to refer to the query text itself and also the length of the query text (and similarly for p). We will use k to denote the maximum number of wildcards allowed in a query. Note that because the building of the structures can depend on k, we require that this parameter be set beforehand and not dynamically with each query. Finally, we will use Σ to denote that size of the alphabet.

1.3. Naive Solution 1: Small Space, Slow Query.

This solution uses only a standard suffix tree S on t. To perform the query, we perform a standard query on the suffix tree, with the addition that each time we come across a wildcard in p, we branch and explore all possible subtrees. The space for this solution is clearly $O(t)$, and the time for a query is $O(\Sigma^k p)$, where Σ is the size of the alphabet. The Σ^k term arises from the fact that we may have to fully explore every subtree of a node each time we search a wildcard.

1.4. Naive Solution 2: Big Space, Fast Query.

1.5. **Cole et al. Solution 1: Centroid Path Decomposition with Slow Query.**

1.6. **Cole et al. Solution 2: Centroid Path Decomposition with Fast Query.**

2. Methodology

3. Results

4. Discussion

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] Erik Demaine. Least Common and Level Ancestors. MIT 6.851 Lectures 15 Notes Spring 2014. http://courses.csail.mit.edu/6.851/spring14/lectures/L15.pdf