A Wildcard Matcher Search Strategy

Claude N. Warren, Jr.

Text licensed under Creative Commons CC-BY-SA-4.0
Code licensed under Apache License 2.0

Motivation

There are times when a single string exemplar must be checked against a number of wildcard patterns to find the best match. This poster presents a strategy for rapidly selecting patterns to check and selecting the best fit among the matches.

An application for such a strategy is the application of Access Control Lists constraints on value to determine if a user has access.

The solution presented here assumes that the more similar the wildcard is to the pattern the higher precedence it has in terms of evaluation.

Background

Let's explore the ACL use case and make the following assumptions:

- The wildcard patterns use asterisks (*) to denote one or more characters.
- The wildcard is implemented as a regular expression replacing the asterisk with the regular expression '.*' to match multiple characters.

The naive algorithm for finding the set of matching patterns for a single target is:

- Create a list of patterns ordered by the number of non-wildcard characters.
- Iterate through the list evaluating each one if a match is found add the pattern to the list of potential solutions. Stop when the number of non-wildcard characters exceeds the number of characters in the length of the target.
- Check each potential solution using a standard measure of text similarity to find the closest match.

Extracting Sequences

This strategy will exploit the work on sequence characterization in bioinformatics[5]. We begin by converting each wildcard pattern set of Hashers[2] that represents each pair of characters in the text by:

Breaking the text into chunks whenever a non character or digit is found.

Remove any chunks that have fewer than 2 characters.

Remove any duplicate chunks.

For each $chunk$

For (int i=0,i<$chunk$.length-2; i++)

token = $chunk$.substring(i,2)

long[] hash = MurmurHash3($token$)

hashes.add(new EnhancedDoubleHasher(hash[0], hash[1]))

return hashes

The hashes can then be used to create a Bloom[1] filter using the Apache Commons Collections® library.

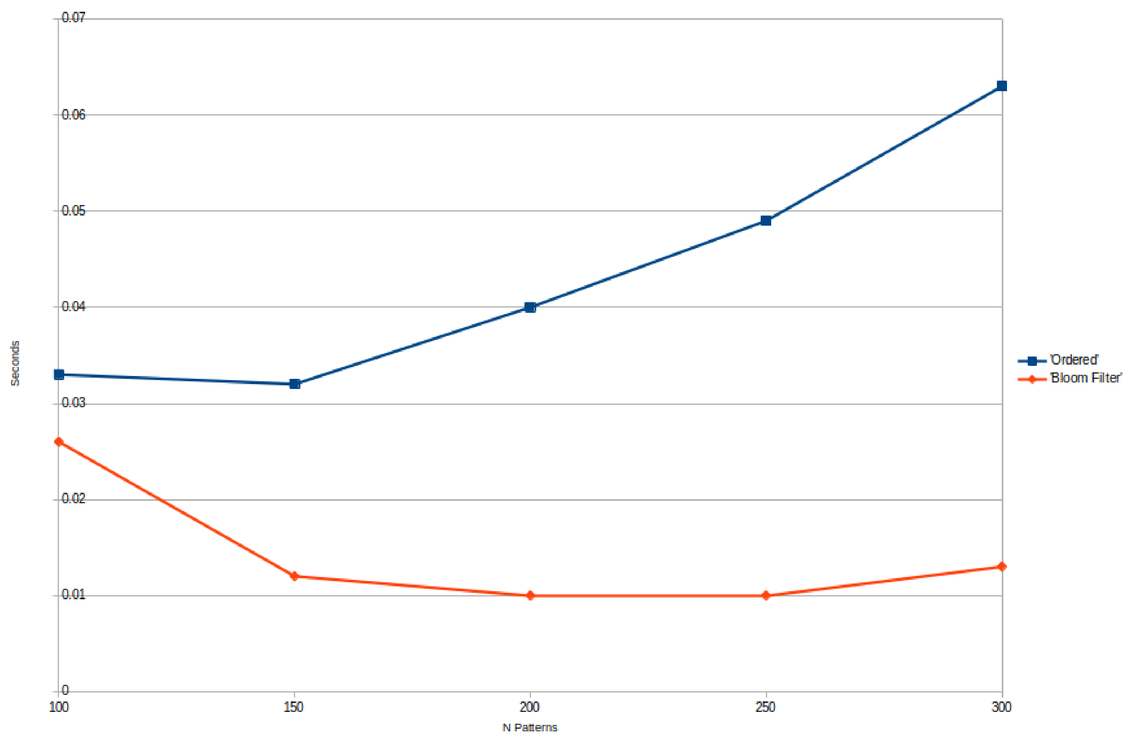
The Bloom filters for the wildcard patterns will have fewer sequences than the patterns they match. This means that we can check if the target Bloom filter contains the wildcard Bloom filter and if so evaluate the pattern for a match.

Selecting The Match

The patterns returned from Extracting Sequences should be compared using a standard measure of similarity like Levenshtein Distance[3] to determine the distance from the target to the pattern. Patterns with smaller distances should be selected over patterns with greater distances. For patterns with the same distances, longer patterns (more specificity) should be selected over shorter patterns.

Finally, the patterns should be tested against the target in the order determined above. The fist matching target is the best fit.

Experimental Results



The graph shows a run of the test code[4] with 100 to 300 patterns. The experiment selected 50 words at random and generates a number of phrases from those words. Two to four words are combined with hyphens '-' to create the number of phrases. The phrases are selected at random, and one or more hyphen separated parts are replaced with asterisks '*' to create the patterns. The table below shows the number of phrases associated with the number of patterns. Each phrase is then tested against the patterns finding the best fit as described in the Motivation section above. The results show that the Bloom filter based search out performs the naive brute force method even for as few as 100 patterns.

Patterns to Phrases Counts

Patterns	Phrases
100	500
150	750
200	1000
250	1250
300	1500

The Bloom filters in the experiment are based on the number of 2-character chunks that were developed for the longest phrase. The maximum bloom filter size was 85 bytes. This means that the solution requires 85 bytes per pattern to implement.

KEY REFERENCES

[1] Burton H Bloom. "Space/time trade-offs in hash coding with allowable errors". In: *Communications of the ACM* 13.7 (1970), pp. 422–426.

[2] Apache Software Foundation. *Hasher (Apache Commons Collections 4.5.0-M1 API)*. May 2024. URL: https://commons.apache.org/proper/commons-collections/apidocs/org/apache/commons/collections4/bloomfilter/Hasher.html.

[3] Apache Software Foundation. *LevenshteinDistance (Apache Commons Text 1.12.0 API)*. May 2024. URL: https://commons.apache.org/proper/commons-text/apidocs/org/apache/commons/text/similarity/LevenshteinDistance.html.

[4] Claude N. Warren Jr. *Claudenw/BloomTextFilter: Example code and Poster for Community over Code EU 20205 Poster Session*. May 2024. URL: https://github.com/Claudenw/BloomTextFilter.

[5] Brad Solomon and Carl Kingsford. "Fast search of thousands of short-read sequencing experiments". In: *Nature biotechnology* 34.3 (2016), pp. 300–302.