CADET: Concurrent fuzzy string searching

1st Conrad Smith
Department of Computer Science
University of Central Florida
Orlando, FL, United States
conradsmith22@knights.ucf.edu

2nd Armin Malekjahani
Department of Computer Science
University of Central Florida
Orlando, FL, United States
arminm@knights.ucf.edu

3rd David Hranicky
Department of Computer Science
University of Central Florida
Orlando, FL, United States
djhranicky@knights.ucf.edu

4th Elbert Tse
Department of Computer Science
University of Central Florida
Orlando, FL, United States
elberttse@knights.ucf.edu

5th Taylor Bean
Department of Computer Science
University of Central Florida
Orlando, FL, United States
TBean@knights.ucf.edu

***Abstract*—Fuzzy string searching describes the process of taking a search term and a corpus (a body of text) and determining which words in the corpus most closely resemble the search term. As more and more applications of fuzzy string searching become utilized, such as autocomplete and IntelliSense, the question of how methods of performing fuzzy string scale and perform come into focus - especially in the light of large corpi. This paper examines common algorithms used to perform fuzzy string searching, proposes a method of parallelizing the algorithms, and evaluates results from benchmark tests to empirically test their performance. Using our benchmarking suite, we saw an average speedup of 4x by utilizing parallelization in our fuzzy string searches. Future work still remains to refine and further increase performance of our approach.**

***Index Terms*—Parallel programming; Parallel processing; Parallel algorithms; Fuzzy string searching; Fuzzy string matching; Edit distance**

I. Introduction

Fuzzy String Searching [8] is the idea of approximately finding a string in a given text (as opposed to searching for the exact string). This idea is used vastly in search engines, autocorrect, IntelliSense, and more. The most common implementations rely on some flavor of edit distance [17], though some alternatives exist in the form of similarity measurement and phonetic algorithms. Depending on the scope and nature of the comparison, these methods may benefit from parallelization. The purpose of this paper is to 1. investigate how we can achieve these potential benefits, and 2. discuss their applicability to existing applications.

II. State of the Art

Modern fuzzy string searching applies to situations where a search term is compared to words in a large body of text or a corpus. These words can be identical but the key for fuzzy string searching is returning words that are similar to a search term. Corpi can take the form of large documents, text files, and databases. Situations such as searching for the word "bee" in a research paper, the name of an animal in a database, or even sequence alignment in bioinformatics are all forms of the fuzzy string searching problem. In a university setting, students are taught algorithms like Knuth-Morris-Pratt to search for patterns in strings or searching in a trie data structure. Databases such as MySQL and PostgreSQL use Soundex [14], a phonetic algorithm, and Trigram matching [1] respectively for their fuzzy text searching algorithms. The auto completion feature in Google's search engine uses RankBrain [22] to suggest or predict search terms, rather than use fuzzy string matching to suggest terms. This shows a trend in using machine learning to predict, rather than algorithmically suggest terms. In the field of bioinformatics, Needleman-Wunsch is used to find an optimal global alignment of genetic sequences, and Smith-Waterman and BLAST for local alignment of sequences.

III. Background

A. The Approximate String Searching Problem

Approximate (or "fuzzy") string searching is a problem that was first identified in literature in the 1960s. One of the first formal specifications of the problem is summarized as follows: given a string s from a set S of all possible strings, find a string t from a corpus T (where T is a subset of S) such that t matches s within some specified margin of error [8].

B. Edit Distance

"Edit distance" most commonly refers to a family of algorithms that aim to quantify the similarity between two strings and is regarded as the standard solution to the fuzzy string searching problem [17].

1) Levenshtein Distance: Levenshtein distance was first proposed as a solution to this problem in 1965 and facilitates a simple set of single-symbol operations to "edit" t, where the "edit distance" between two strings is the minimum number of operations needed to make t match s [13]. These operations include 1. insertion (e.g. insert symbol w into string uv to make uvw), 2. deletion (e.g. delete symbol w from string uwv to make uv), and 3. substitution (e.g. substitute symbol w with symbol x in string uwv to make uxv).

The Damerau variant allows for a fourth operation called transposition where by two characters switch places with each other (e.g. transpose symbols uv to make vu). Optimal String Alignment is a variant of this that restricts each substring to being edited at most once, where in the base Damerau variant there is no such limit.

2) Jaro-Winkler Distance: Jaro-Winkler distance is similar to Levenshtein distance, in that they are both edit distance metrics, but Jaro-Winkler examines matching characters and required transpositions, also giving more weight to words that have similar prefixes [12]. Jaro-Winkler distance d_w is defined as:

$$d_w = 1 - sim_w$$

$$sim_w = sim_j + \ell p(1 - sim_j)$$

$$sim_j = \begin{cases} 0 & m = 0 \\ \frac{1}{3}(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m}) & otherwise \end{cases}$$

Where:

- $|s_i|$ is the length of string s_i.
- m is the number of matching characters.
- t is the number of transpositions.
- sim_j is the Jaro similarity for strings s_1 and s_2.
- ℓ is the length of the common prefix at the start of the string up to 4 characters.
- p is a scaling factor used to adjust the score upwards for having common prefixes. This value be greater than 0.25.
- sim_w is the Jaro-Winkler Similarity for s_1 and s_2.

[25]

A naive approach to parallelizing the algorithm can be done on counting the number of matching characters, where each thread can count over a portion of the string. For large strings, multiple threads can divide the string into substrings and count the matching characters. A major pitfall with this approach is that it does not work well for strings of drastically different length. Furthermore, threads can take a different amount of time to complete the task of counting depending on the portion they are responsible for, leaving some threads waiting before moving on to another string. An better approach to parallelizing the algorithm is ironically not parallelizing the algorithm. Instead, let each thread calculate the Jaro-Winkler distance between a search term and a string from the corpus, and let each thread poll some structure for the next string in a corpus.

3) Needleman-Wunsch/Smith-Waterman: The Needleman-Wunsch algorithm is an edit distance algorithm that was designed for comparing two sequences of DNA to find the best alignment between the two [4]. It works using dynamic programming to break the strings down into comparisons between one letter in each string and calculating a similarity score, then finds the best score by picking the best score when searching from one end to the other.

Smith-Waterman is a variant of Needleman-Wunsch that is focused on a local alignment rather than a global one [20]. The score is calculated slightly differently and the final score does not search over the whole length of the strings, just a portion of it. Local alignment is advantageous in situations where "clumps" of matching characters matter more than the overall strings matching.

C. Similarity Measures

Similarity Measures are ways of comparing two strings and finding how similar they are. The idea is closely related to edit distances as a lower distance implies a higher degree of similarity, but most similarity algorithms don't consider directly editing the strings.

1) Cosine Similarity: Cosine Similarity is a simple measure of similarity between two words that works by finding the cosine of the angle between the two vectors resulting from plotting the words in a specific vector space [18]. The vector space has N dimensions where N is the number of unique letters in both words. A word's vector is constructed by assigning the value 1 to the dimension corresponding to each unique letter in that word.

In figure 1, there are two words "At" and "Am" plotted in a vector space. The blue axis corresponds to "M", the green "A" and the red "T". The vector for "At" is {0, 1, 1} and the vector for "Am" is {1, 1, 0}. From here the cosine of the angle is found using the dot product formula:

$$cos(\theta) = \frac{\vec{A} \cdot \vec{B}}{\|\vec{A}\|\|\vec{B}\|}$$

Where:

$$\vec{A} \cdot \vec{B} = (0 * 1 + 1 * 1 + 1 * 0) = 1$$
$$\|\vec{A}\| = \sqrt{0^2 + 1^2 + 1^2} = \sqrt{2}$$
$$\|\vec{B}\| = \sqrt{1^2 + 1^2 + 0^2} = \sqrt{2}$$

which gives a result of 1/2, with a higher score indicating greater similarity.

2) Jaccard Index: The Jaccard index measures the similarity between two strings by finding the proportion of letters shared between two words [9]. The formula for the Jaccard index is given by

$$J(X, Y) = \frac{|X \cap Y|}{|X \cup Y|}$$

Formally, this finds the cardinality of the intersection of the sets of the two words and divides by the cardinality of the union of the sets of the two words. If word A is "apple" and word B is "orange", the intersection yields the set a, e which has cardinality 2, while their union yields set a, p, l, e, o, r, n, g which has cardinality 8, so their Jaccard index is 0.25 or 25%. The Jaccard distance measures dissimilarity and is computed as 1 - J(X,Y).

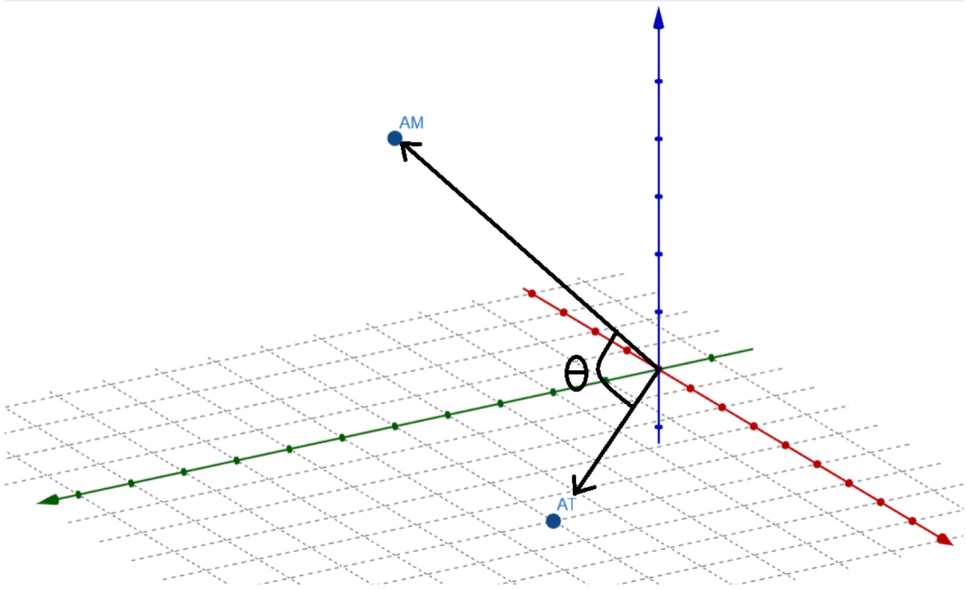

Fig. 1. Graph showing ”At” and ”Am” plotted in a vector space

3) Longest Common Substring: The Longest Common Substring is a similarity measurement function that finds the longest string that both input strings share between them [7]. Given the two strings ”abcd” and ”aebce”, the longest substring between them is ”bc” as it is found within both of the input strings.

4) Longest Common Subsequence: The Longest Common Subsequence (LCS) is a similarity measurement function that searches through words and finds the longest common subsequence, with longer subsequences indicating greater similarity [15]. The difference between this and the longest common substring is that in the substring version, the characters found must be sequential, while in the subsequence version, they do not. Looking at the previous example, the strings ”abcd” and ”aebce” have ”bc” as the longest common substring with a length of 2, where as the longest common subsequence is ”abc” with a length of 3.

5) N-Gram Similarity: N-Gram Similarity extends upon the idea of LCS by comparing differently sized chunks of a word to better encapsulate different contexts within the word [11]. For example, ”ant” and ”and” are more similar than ”the” and ”ate”, yet both have an LCS of 2. Breaking these words down into 2-grams yields {an, nt} and {an, nd}, along with {th, he} and {at, te}. The first pair have one bi-gram in common while the second pair has none, indicating there is more similarity between the first pair than the second.

6) Q-Gram Similarity: Q-Gram similarity or Approximate String Matching works by counting the number of q-grams in common between two words [24], like a less sophisticated version of N-Gram similarity.

7) Sorensen Dice: The Sorensen Dice similarity score is quite close to the Jaccard Index, except rather than divide by the number of elements in the union of the sets, it divides by the sum of the cardinalities of each string's set [21]. Formally, the equation is given by

$$SD(X,Y) = \frac{2 * |X \cap Y|}{|X| + |Y|}$$

D. Phonetic Algorithms

Phonetic Algorithms are those that attempt to encode a string by their pronunciation [3]. The use of phonetic algorithms are especially useful when implementing features like auto-correct, where words are corrected based on how close they sound to other words, not how closely they are written out. Various different algorithms are present to represent words phonetically, including Soundex and Metaphone. It should be stated that these algorithms are, by nature, approximations. In the context of fuzzy string searching, phonetic encoding also uses edit distance algorithms (of which were described above) to get a metric of similarity between two strings. That is, two strings which are phonetically encoded and have a small edit distance are phonetically similar. This approach does have its caveats since phonetic algorithms encode words in an arbitrary encoding, where each character of the encoded string may not hold the same ”weight” in describing the word as a regular English word would.

IV. Build Instructions

To build the code, clone the repository from Github into an empty folder. In a terminal, navigate to the root directory where the src folder is. To compile the code, run the command ”javac -cp ”.:./lib/*” src/Main.java” and to run the code, input

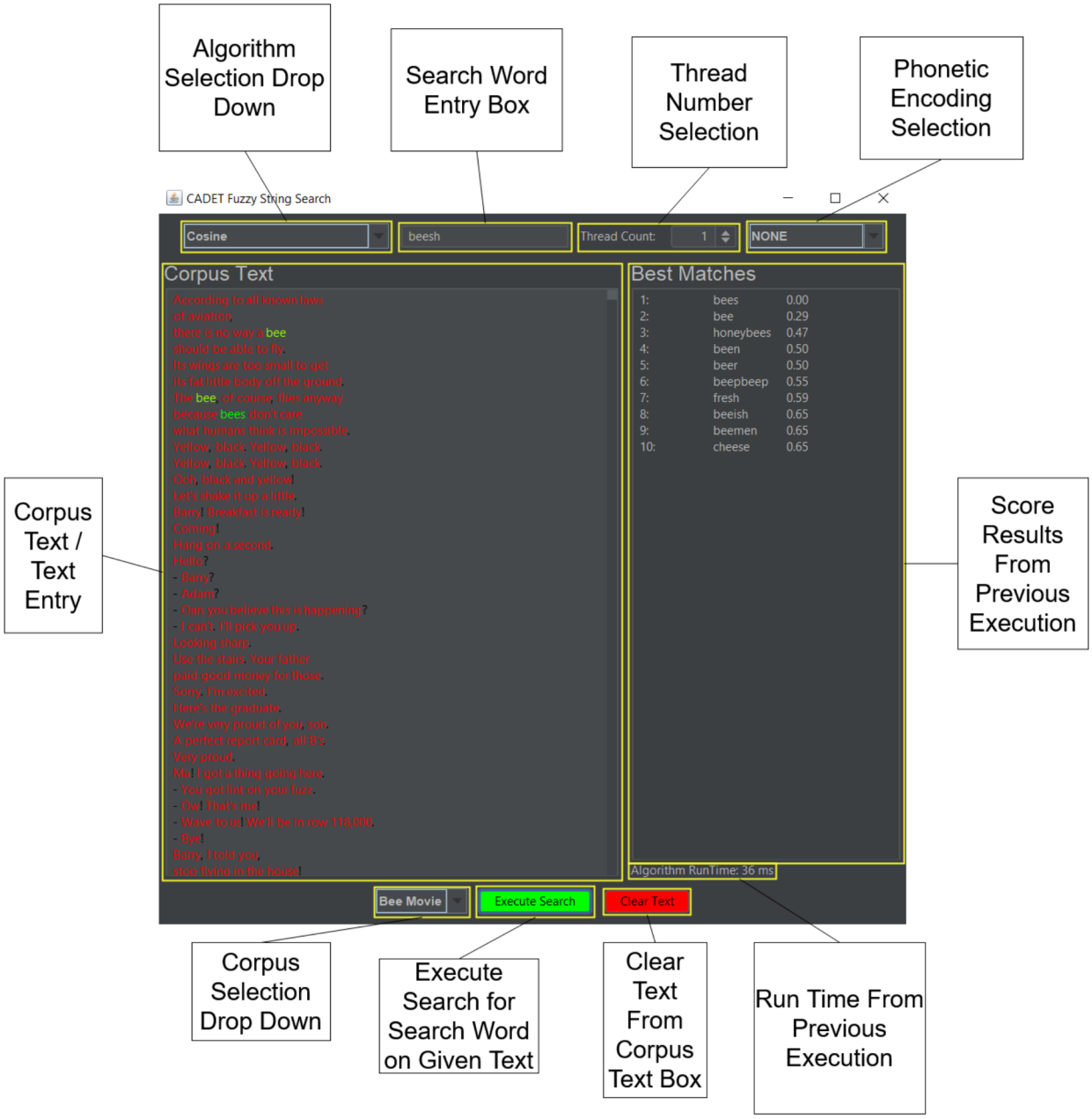

Fig. 2. Example picture of the GUI after a search for "beesh" on the Bee Movie [19] corpus using the Cosine Similarity Algorithm. Theme is Flatlaf [10]

the command "java -cp ".:./lib/*" src/Main". For building on Windows, use a ; instead of a : (.;. instead of .:.).

Alternatively, one can use the helper script "cliRunner.sh". To run, simply call the script in a terminal window (with Windows, use git bash or cygwin) and pass the argument "gui" to open the gui.

V. Approach

A. User Interface

Our implementation sports a User Interface that is interactive and dynamic (see Fig.2). Users can choose a variety of options to tune their choice of corpus, phonetic algorithm, edit distance algorithm and number of threads to utilize. Finally, a word to search for in the corpus is provided. Given these parameters, a view of the corpus visualizes how strongly each word in the corpus relates to the provided search word. Along with this general visualization is a list of the top-ten words that match most strongly with the search word. Not only does this UI make our implementation accessible for others to use, but it also provides a framework to easily iterate and test new algorithms, parameters, and visualizations that we made as we conducted more research and implementations.

B. Corpus Visualization

As mentioned above, the User Interface supports visual representation of the algorithms results across the corpus.

Doing so required the use of the Java Swing JTextPane component which supports advanced text formatting. After receiving the results from the algorithm we compute the necessary statistics, normalize the values, and map the values stored in the ConcurrentSkipListSet to the word allowing for an O(1) look up for any word key found in the corpus. Afterwards, we pass this map to the CorpusTextPanel Class which will iterate through each word in the corpus and find the normalized score value in the provided map. This score value is used to compute the given color for that specific word.

C. Token Provider

"Token Provider" is a helper class that takes in the corpus to search upon and allows threads to concurrently process over it. This class is backed by a ConcurrentLinkedQueue, where each "token" in the corpus is an entry in the data structure. A "token" is simply a space-delimited string from the corpus. By splitting the corpus into these tokens, threads can take a token, perform computations over it, and then move onto another token. Since the ConcurrentLinkedQueue is lock free, threads don't have to acquire locks while they are accessing the data. This is the key feature to our implementation which results in vast performance increases, shown in the Results & Analysis section.

D. Algorithm Runner

"AlgRunner" is the Runnable on which our threads are created. The TokenProvider is passed as a constructor as well as the desired algorithm, phonetic and search word. In whole, this is the starting point of computation execution. While the given TokenProvider contains tokens, threads will perform the given algorithm on a requested token. This result is stored in a ConcurrentSkipListSet as WordScoreEntry which implements comparable and stores both the word and corresponding score.

E. Phonetic Encoding

Apache's Codec [2] package sports numerous phonetic algorithms, all of which we expose in our implementation of fuzzy string searching. An exhaustive list of the provided phonetic utilized algorithms along with Apache's Codec documentation can be found here. It should be noted that not all of these phonetic algorithms are meant to be used in this "general" use case. That is to say, some phonetic algorithms are tailored to specific use cases, such as in identifying similar sounding human names or for working with specific languages. As such, they may provide nonsensical output in relation to expected output. Nonetheless, it is still useful and interesting to see how these algorithms perform amongst the rest.

F. Benchmarking

With an appropriate list of fuzzy-matching algorithms to experiment with (and given that their sequential implementations are not relevant to the paper, a publicly available repository of them was used [23]), as well as a framework for parallelizing them, a benchmarking suite was constructed to empirically evaluate the framework's utility. This suite runs through the entire list of algorithms with a variety of user-specified hyperparameters, the most important ones being which corpus to use, the maximum number of threads to test each algorithm on, the number of "iterations" to run for, and the length of the randomly generated search terms.

An iteration can be considered to be the time it takes for n threads to execute the algorithm on an entire corpus given a random search term. Importantly, this time includes the overhead of spawning and joining threads. The number of threads used starts at 1, and then doubles until the maximum specified thread count is reached. A publically available progressbar library was used in part to gauge program performance [5]. The randomness in the search terms serves two purposes: 1. to eliminate any potential corpus bias (e.g. if one searches the word "bee" in the Bee Movie [19] script, some algorithms may perform slightly better overall, since many words in that corpus include, or are similar to, "bee"), and 2. to mitigate any potential influence that caching may have on algorithm performance (preliminary results suggested that, as the number of iterations of a test increases, the average execution time decreased). While the side-effects of caching would greatly benefit any application that uses these methods, it would compromise the authenticity of the benchmarking results. Finally, each randomly generated word contains solely lowercase alphabetic characters.

VI. Results & Analysis

All results shown were acquired and tabulated by the benchmarking suite. Hardware specifications for the test bench will not be listed, as any number of factors outside of user control may be encountered at both a hardware and software level. However, considerable deviations in results may be encountered depending on CPU, DRAM, and storage device specifications. Relative speedups should, at the very least, remain constant among tests using the same hyperparameters, which will be the focus of these results.

A. Benchmarking Results

The "base bench" involved the following hyperparameters: the "All Words" corpus (a publicly available list of 466,000 lowercase English words, which is the most exhaustive corpus available in the suite) [6], a maximum of 8 threads, an iteration count of 1000 (which took roughly 3 hours to run on the benchmarking machine), and a random search term length of 10 characters. While the suite also records the fastest and slowest execution times for each algorithm and thread combination, only the average execution time (in milliseconds) will be shown.

On the base benchmarking test, each algorithm enjoyed at least 3x speedup going from 1 thread to 8 threads [Table I]. This varied considerably between algorithms, with certain ones like QGram achieving a 5.235x speedup in execution time. The "worst" speedup multiplier was achieved by Jaro-Winkler, coming in at 3.204x. In fact, edit distance algorithms generally experienced less of a speedup overall, garnering an average

Algorithm	1 Thread	2 Threads	4 Threads	8 Threads
Cosine	372.614	228.427	137.957	84.190
Damerau	508.698	294.870	169.729	100.355
Jaccard	425.175	259.180	150.800	92.528
JaroWinkler	181.441	127.241	82.893	56.629
Levenshtein	195.434	132.229	84.121	55.180
LCS	246.338	157.606	97.034	60.158
MetricLCS	246.003	159.468	98.507	61.896
Ngram	276.163	175.807	107.237	66.297
Norm. Levenshtein	192.268	133.008	86.683	58.432
OSA	285.080	178.569	107.642	64.789
Qgram	523.226	299.342	169.767	99.939
RatcliffObershelp	359.848	218.110	130.994	77.513
SorensenDice	503.441	300.687	172.522	99.221

TABLE I
RESULTS OF BASE BENCHMARKING (BENCH 1). ALL NUMBERS SHOWN REPRESENT THE AVERAGE EXECUTION TIME IN MILLISECONDS FOR THAT ALGORITHM AND THREAD COUNT COMBINATION OVER 1000 ITERATIONS.

Algorithm	1 Thread	2 Threads	4 Threads	8 Threads
Cosine	299.771	196.669	121.473	75.959
Damerau	409.338	246.708	144.640	81.931
Jaccard	393.803	239.599	142.024	86.150
JaroWinkler	158.412	118.546	79.760	55.066
Levenshtein	172.146	118.894	76.648	51.346
LCS	211.655	138.765	86.840	56.059
MetricLCS	214.305	143.636	90.248	58.398
Ngram	229.475	154.689	96.554	63.097
Norm. Levenshtein	170.102	122.705	81.276	56.217
OSA	236.160	150.398	93.049	58.758
Qgram	481.202	275.355	156.686	93.697
RatcliffObershelp	209.795	141.401	90.241	59.634
SorensenDice	450.331	268.849	155.649	94.074

TABLE II
RESULTS OF SECOND BENCHMARKING TEST (BENCH 2), WITH A RANDOM SEARCH TERM LENGTH OF 5 INSTEAD OF 10.

Algorithm	1 Thread	2 Threads	4 Threads	8 Threads
Cosine	150.509	85.882	49.119	31.775
Damerau	205.494	111.001	62.234	36.517
Jaccard	175.775	99.472	55.631	36.028
JaroWinkler	88.628	55.345	36.782	27.982
Levenshtein	89.356	55.261	37.380	27.169
LCS	116.012	68.586	43.062	28.903
MetricLCS	120.736	72.253	44.330	28.632
Ngram	107.383	64.426	40.699	28.883
Norm. Levenshtein	87.354	54.357	36.712	27.598
OSA	127.322	75.760	44.389	29.252
Qgram	182.394	103.744	58.123	38.265
RatcliffObershelp	145.972	87.245	52.924	35.374
SorensenDice	187.872	107.141	63.451	39.737

TABLE III
RESULTS OF THIRD BENCHMARKING TEST (BENCH 3), WITH A MUCH SMALLER CORPUS (THE MOBY DICK NOVEL [16]).

Algorithm	Bench 1	Bench 2	Bench 3
Cosine	4.426	3.946	4.737
Damerau	5.069	4.996	5.627
Jaccard	4.595	4.571	4.879
JaroWinkler	3.204	2.877	3.167
Levenshtein	3.542	3.353	3.289
LCS	4.095	3.776	4.014
MetricLCS	3.974	3.670	4.217
Ngram	4.166	3.637	3.718
Norm. Levenshtein	3.290	3.026	3.165
OSA	4.400	4.019	4.353
Qgram	5.235	5.136	4.767
RatcliffObershelp	4.642	3.518	4.127
SorensenDice	5.074	4.787	4.728

TABLE IV
AVERAGE SPEEDUP MULTIPLIER (GOING FROM 1 THREAD TO 8 THREADS) EXPERIENCED BY EACH ALGORITHM ON EACH BENCHMARKING TEST.

Algorithm Family	Bench 1	Bench 2	Bench 3	All
Edit Distance	3.776	3.563	3.812	3.717
Similarity Measure	4.512	4.118	4.393	4.341

TABLE V
AVERAGE SPEEDUP EXPERIENCED BY EACH ALGORITHM FAMILY ON EACH BENCHMARKING TEST.

Bench #	Speedup
1	4.286
2	3.947
3	4.214
All	4.149

TABLE VI
AVERAGE AVERAGE SPEEDUP ACROSS ALL ALGORITHMS FOR EACH BENCHMARKING TEST.

speedup of 3.776x as opposed to the 4.512x achieved by the chosen similarity measure algorithms [Table VI].

This trend was roughly followed across the other two benchmarking tests [Tables II & III]. Speedup was, strictly speaking, slightly less across the board for benchmarking test 2, which saw the length of each random search term lowered to 5 instead of 10. This may suggest that, at higher thread counts, the corpus splitting procedure could not keep up with the speed at which each thread was finishing computing the algorithm on the tokens provided, causing contention for that resource, though this could be studied further with more capable hardware (i.e. a CPU with more cores) and additional iterations. Absolute execution times appeared to be slightly better compared to the base bench, however.

Benchmarking test 3 - which utilized the entirety of Moby Dick as the corpus instead of the significantly larger "All Words" corpus - shared similar results with the base test, which saw the speedup of edit distance algorithms increase negligibly and that of similarity measure algorithms decrease marginally. As expected, absolute times were significantly lower than that of the base bench, with all algorithms roughly seeing a 2.5-3x speed increase for each thread count compared to the base bench.

B. Caching

The potential effects of caching were examined by exclusively running Levenshtein across two corpi ("All Words" and the Bee Movie script), with 125, 250, 500, and 1000 iterations, on 8 threads, using two explicitly defined search terms ("word" and "bee", respectively). The results were roughly the same across all iteration counts, contradicting initial results which suggested that some level of caching was occurring. This could be attributed to a certain method of taking the running average over each iteration, which was later replaced and thus apparently fixed.

VII. Conclusion

Given that the algorithms analyzed achieved an average speedup multiplier of around 4x over 8 threads, we can comfortably say that our corpus splitting procedure and tokenizer have successfully been able to overcome the sequential limitations of these popular fuzzy string matching algorithms. This could prove to be considerably useful in applications that require a simple autocorrect or fuzzy-matching mechanism, and our work demonstrates that these algorithms are easily parallelizable, which can greatly increase the performance of such a mechanism. Additionally, this form of parallelization can be extended to apply to any string-operating function that requires a corpus.

Future work on this subject could be dedicated to further examining any influence caching may have on typical application workflows that may use these algorithms. To reiterate, any caching of results would greatly improve algorithm performance, and should be sought after when building products that use this technology.

References

[1] F.33. $\mathrm{pg}_t rgm, Feb2022$.

[2] Apache. Commons codec, 2022.

[3] Black, P. E. Dictionary of algorithms and data structures. https://xlinux.nist.gov/dads/HTML/phoneticCoding.html.

[4] B.Needleman, S., and D.Wunsch, C. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology 48*, 3 (1970), 443–453. https://www.sciencedirect.com/science/article/abs/pii/0022283670900574?via%3Dihub.

[5] ctongfei. Progressbar, 2022.

[6] dwyl. List of 466k english words, 2022.

[7] Gusfield, D. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997. https://www.cambridge.org/core/books/algorithms-on-strings-trees-and-sequences/F0B095049C7E6EF5356F0A26686C20D3#.

[8] Hall, P. A., and Dowling, G. R. Approximate string matching. *ACM Computing Surveys 12*, 4 (1980), 381–402. https://dl.acm.org/doi/abs/10.1145/356827.356830.

[9] Jaccard, P. The distribution of the flora in the alpine zone. *The new phytolooist 11*, 2 (1912), 37–50. https://nph.onlinelibrary.wiley.com/doi/abs/10.1111/j.1469-8137.1912.tb05611.x.

[10] JFormDesigner. Flatlaf, 2022.

[11] Kondrak, G. N-Gram similarity and distance. *LCNS 3772* (2005), 115–126. http://webdocs.cs.ualberta.ca/~kondrak/papers/spire05.pdf.

[12] kulkarni, S. Jaro winkler vs levenshtein distance. *Medium* (2021). https://srinivas-kulkarni.medium.com/jaro-winkler-vs-levenshtein-distance-2eab21832fd6.

[13] Levenshtein, V. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics - Doklady 10*, 8 (Feb 1966), 707–710. https://nymity.ch/sybilhunting/pdf/Levenshtein1966a.pdf.

[14] LitKnd. Soundex (transact-sql) - sql server, Nov 2021. https://docs.microsoft.com/en-us/sql/t-sql/functions/soundex-transact-sql?view=sql-server-ver15.

[15] Maier, D. The complexity of some problems on subsequences and supersequences. *Journal of the ACM 25*, 2 (1978), 322–336. https://dl.acm.org/doi/10.1145/322063.322075.

[16] Melville, H., and Schaeffer, M. *Moby Dick*. New York, Dodd, Mead and company, 1922. https://www.biodiversitylibrary.org/bibliography/140282.

[17] Navarro, G. A guided tour to approximate string matching. *ACM Computing Surveys 33*, 1 (Mar 2001), 31–88. http://users.csc.calpoly.edu/~dekhtyar/570-Fall2011/papers/navarro-approximate.pdf.

[18] NIST. Cosine distance, cosine similarity, angular cosine distance, angular cosine similarity. *National Institute of Standards and Technology* (2017). https://www.itl.nist.gov/div898/software/dataplot/refman2/auxillar/cosdist.htm.

[19] Seinfeld, J. Bee movie, 2007.

[20] Smith, T. F., and Waterman, M. S. Identification of common molecular subsequences. *Journal of Molecular Biology 147*, 1 (1981), 195–197. https://dornsife.usc.edu/assets/sites/516/docs/papers/msw_papers/msw-042.pdf.

[21] Sorensen, T. A method of establishing groups of equal amplitude in plant sociology based on similarity of species content. https://www.royalacademy.dk/Publications/High/295_S%C3%B8rensen,%20Thorvald.pdf.

[22] Sullivan, D. Faq: All about the google rankbrain algorithm, Mar 2022.

[23] tdebatty. Java string similarity, 2022.

[24] Ukkonen, E. Approximate string-matching with q-grams and maximal matches. *Theoretical Computer Science* (1992), 191–211. https://www.sciencedirect.com/science/article/pii/0304397592901434.

[25] Winkler, W. E. String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage. https://files.eric.ed.gov/fulltext/ED325505.pdf.