Jinning’s Tweet Search System

The search system is done in Java, as I’m more familiar with that. (Note that I have some experience with Python.)

The whole search problem is very similar to “searching words (tokens) in tweets”. I used six different algorithms with different pros and cons: *`Naive`*, *`TokenizedSearch`*, *`SimpleSuffixTree`*, *`MultiPartitionSuffixTree`*, *`SimpleTokenizedSuffixTree`*, *`and MultiPartitionTokenizedSuffixTree`*.

Note Before

A. I did not only store the latest tweet as using that means we can not tune the ranking algorithm.
B. To run the tests, use “run_tests.sh”.

The Highest Performance Solution

The highest performance solution I can think of is to use a `HashMap` and put all tokens in all tweets into the map along with the Tweet indices. -- The given problem size is not very large, so I didn’t partition the HashMaps. However, if we have too many tweets and tokens, we can partition the tokens and put them on different machines, and then use some well-studied reduce algorithms to get the result.

Trading Some Performance for Better Search Result

Hashing everything sounds good, but what if a user:

a) Searches “*BestTaco*” and lots of tweets contain “*TheBestTaco*”?
b) Searches “*apple laptop*” and we return something with “*Have an apple in the laptop bag*” rather than “*The latest Apple laptop is so cool*”?

In these cases, we might want to change our search algorithm a bit and enable it to match substrings of the tokens or even substrings of the tweets. One simple (i.e. implementable in a few hundred lines without using third-party libs) way is to use **suffix tree**, and that’s what I choose.

Suffix tree basically means that we put every suffix of a string into a trie tree, and do matching on the tree. In my implementation, I stored all twitter in a giant suffix tree, and mark the suffix with the tweet indices stored in a set in the tree node.

Performance Concerns

A suffix tree is based on a trie tree, which is known to take a huge amount of memory if the string is not short. Luckily, the tweets are usually short. So the trie tree can be smaller. But still, more than 32GB of memory will be used when we're dealing with a large (> 100,000) number of tweets.

For my suffix tree implementation, a big issue is that my code will use up a huge amount of memory when building the tree, as we will create and link the nodes per character. Also, Java will use tons of memory "by design". I used these tricks to help reduce memory consumption:

Multi Partitioning

As a suffix tree's memory footprint is roughly ***quadratic in the length of the string*** and ***linear in the number of the string***. But in practice, this is only a rough upper bound. If we log the memory consumption of our suffix tree, we can observe that once we've built the tree, the memory footprint will drastically decrease as we will trim the tree down to its most optional form.

If we have more time, we can use some fancy algorithm to build the tree. Like recently a parallel tree-building algorithm with *O(**N**)* time and *O(log²**N**)* span has been proposed.

Optimization 1

As said before, I don't have time to thoroughly read and implement those algorithms. So I just used a simpler approach: Partition the tweets and put them in different suffix trees.

After some experiments, I used *20000* tweets per suffix tree if we have less than *60000* tweets in total; *10000* tweets per tree if we have less than *120000* tweets; and *5000* tweets if we have more than that.

With these settings, I compared the partitioned suffix tree's performance with naive implementation:

Problem size	**Naive Time (ms)**	**Suffix Tree time (ms)**	**Speedup**
Small	20706	70	296x
Medium	95606	658	145x
Large	184869	2736	68x

Algorithm "`MultiPartitionSuffixTree`", "`MultiPartitionTokenizedSuffixTree`" used this optimization.

Optimization 2

Also, there's another way to do it if we don't mind scenario b mentioned above, which is to break tweets into tokens, and put the tokens into the suffix tree.
Algorithm "*`SimpleTokenizedSuffixTree`*", "*`MultiPartitionTokenizedSuffixTree`*" uses this optimization.

The Sorting algorithm

I've implemented a "Recommender" class and an "AlgorithmHandler" interface for sorting the results. Here are the implemented algorithms:

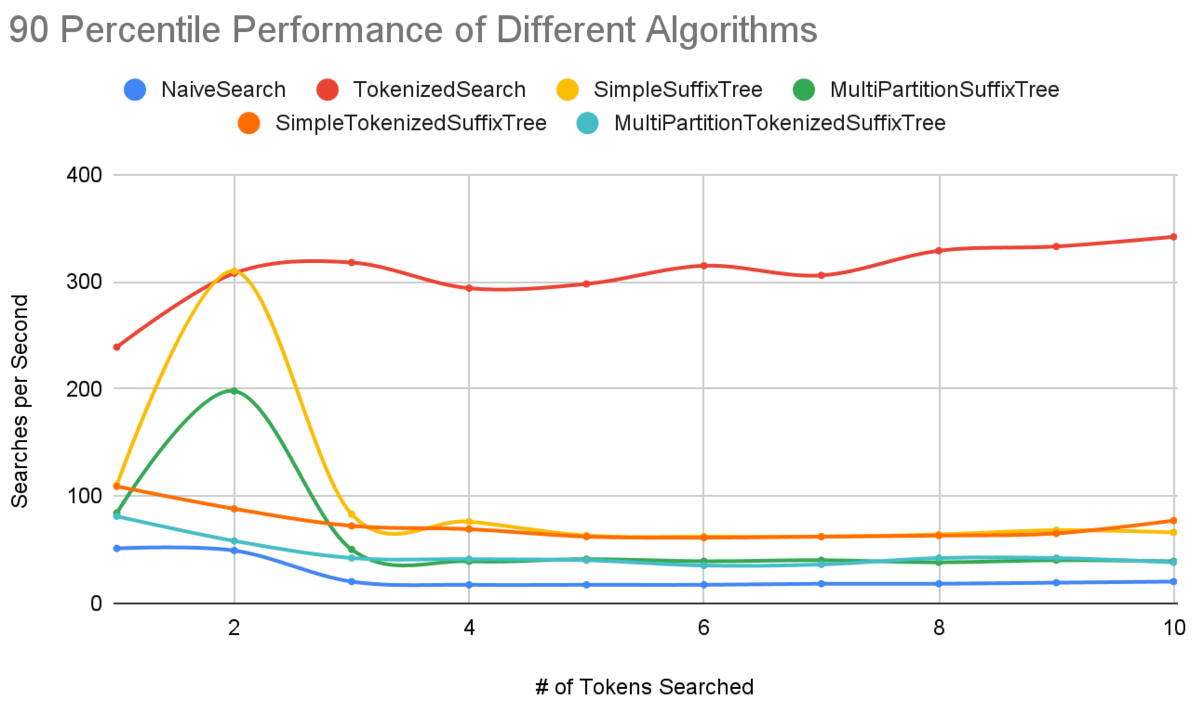
- *`Latest`*
- *`HighestViews`*
- *`HighestLikes`*
- *`HighestRetweets`*
- *`MyAwesomeModel`*

The last algorithm is just a model considering the tweet's all stats, with a strong preference for the latest tweets, calculated as:

$$match\_score = views + likes * 50 + retweets * 100 * deteration\_factor, \text{ where}$$
$$deteration\_factor = (LatestTime + 1) / (LatestTime + 1 - TweetTime)$$

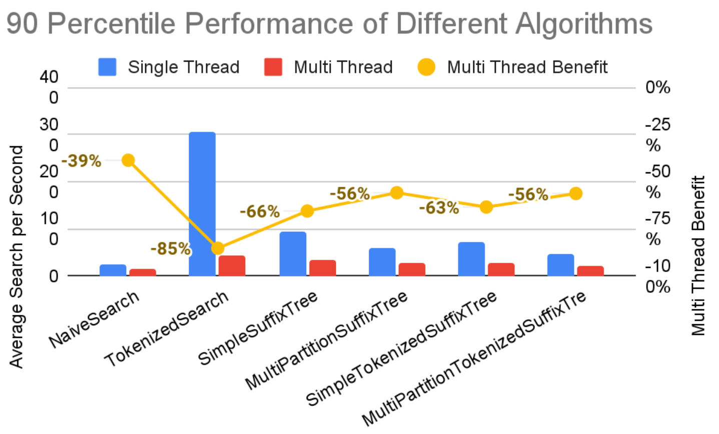
The Overall Performance

See charts for the overall performance:



There's a weird drop for suffix trees when the number of tokens go from 2 to 3. I don't have time to look at this in depth. **If I have more time I plan on using something like a intellij's Java profiler to narrow down why this happens**.

I also tried multi-threading the queries, but it seems that it can't bring any performance benefit on my machine. Probably due to the query is memory limited and multi-threading will just saturate the `CPU -> DRAM` path. If I have more time, I'll do some in-depth profiling using VTune to verify my theory:



Extensions I Would Like To Have

1. Implement the spell corrector.
2. Cache the search results.
3. Separate the control and data in unit tests.
4. Currently, we have bubbles in MultiThreadSearch and MultiThreadReduce.

    ```
    MultiThreadSearch -> Join -> MultiThreadReduce
    ```

    We can make it to:

    ```
    MultiThreadSearch -> MultiThreadReduce
    ```

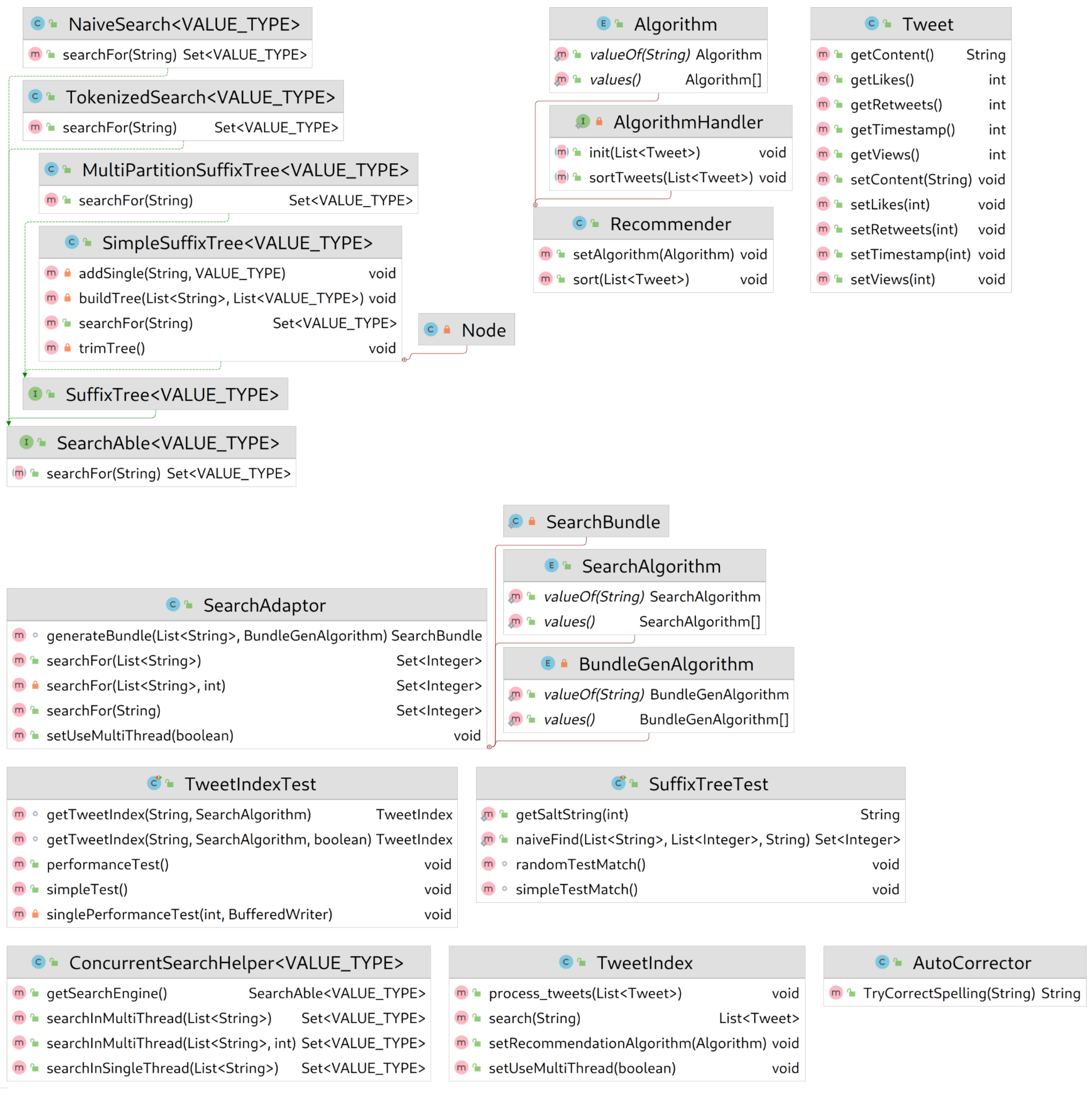
5. Create a `SearchAbleFactory` for cleaner code.

    But it's painful to write Factory with generics in Java, and the code is not likely to live long, so... :)

6. Do in-depth studies for different search algorithms' performance, find optimization opportunities, and use the most optimal performance at different problem sizes.

Appendix

UML Diagram



NaiveSearch<VALUE_TYPE>
searchFor(String) Set<VALUE_TYPE>
TokenizedSearch<VALUE_TYPE>
searchFor(String) Set<VALUE_TYPE>
MultiPartitionSuffixTree<VALUE_TYPE>
searchFor(String) Set<VALUE_TYPE>
SimpleSuffixTree<VALUE_TYPE>
addSingle(String, VALUE_TYPE) void
buildTree(List<String>, List<VALUE_TYPE>) void
searchFor(String) Set<VALUE_TYPE>
trimTree() void
Node
SuffixTree<VALUE_TYPE>
SearchAble<VALUE_TYPE>
searchFor(String) Set<VALUE_TYPE>
Algorithm
valueOf(String) Algorithm
values() Algorithm[]
AlgorithmHandler
init(List<Tweet>) void
sortTweets(List<Tweet>) void
Recommender
setAlgorithm(Algorithm) void
sort(List<Tweet>) void
Tweet
getContent() String
getLikes() int
getRetweets() int
getTimestamp() int
getViews() int
setContent(String) void
setLikes(int) void
setRetweets(int) void
setTimestamp(int) void
setViews(int) void
SearchBundle
SearchAlgorithm
valueOf(String) SearchAlgorithm
values() SearchAlgorithm[]
BundleGenAlgorithm
valueOf(String) BundleGenAlgorithm
values() BundleGenAlgorithm[]
SearchAdaptor
generateBundle(List<String>, BundleGenAlgorithm) SearchBundle
searchFor(List<String>) Set<Integer>
searchFor(List<String>, int) Set<Integer>
searchFor(String) Set<Integer>
setUseMultiThread(boolean) void
TweetIndexTest
getTweetIndex(String, SearchAlgorithm) TweetIndex
getTweetIndex(String, SearchAlgorithm, boolean) TweetIndex
performanceTest() void
simpleTest() void
singlePerformanceTest(int, BufferedWriter) void
SuffixTreeTest
getSaltString(int) String
naiveFind(List<String>, List<Integer>, String) Set<Integer>
randomTestMatch() void
simpleTestMatch() void
ConcurrentSearchHelper<VALUE_TYPE>
getSearchEngine() SearchAble<VALUE_TYPE>
searchInMultiThread(List<String>) Set<VALUE_TYPE>
searchInMultiThread(List<String>, int) Set<VALUE_TYPE>
searchInSingleThread(List<String>) Set<VALUE_TYPE>
TweetIndex
process_tweets(List<Tweet>) void
search(String) List<Tweet>
setRecommendationAlgorithm(Algorithm) void
setUseMultiThread(boolean) void
AutoCorrector
TryCorrectSpelling(String) String