A report on the

Optimizations of an Inverted Search Engine

K23A University Project

By Philip Katis (1115201800069)

Department of Informatics and Telecommunications
National and Kapodistrian University of Athens

Introduction

The following is a report on the performance characteristics and different optimization techniques that were used in the ISE project. The report follows the steps that taken to optimize the application in the order that they were implemented, and it shows how for a 7MB input file we were able to reduce the execution times from over 189 minutes to just under 5 minutes.

The Problem

Our application is tasked with generating answers to a specific document, based on a number of predefined queries. Queries contain a number of keywords (1 to 5), a type and the maximum distance threshold that allows it to be an answer to a document. A query is an answer to a document when, for every keyword it contains there is at least one word in the document within the maximum distance threshold. The distance calculation function is based on the type of the query. The answer to a document is the set of all registered queries that satisfy the above requirements.

The three different type of queries in the data set are the following:

1. **Exact Matching:** As the name suggests, exact matching expects every keyword in the query to be found in the specified document. The maximum distance threshold for this type of query is undefined, and a string matching function is used.
2. **Hamming Distance Matching:** This type of query uses the Hamming Distance metric, to calculate the distance between two strings of equal length. The maximum distance threshold is from 1 to 3 characters, the algorithm used is implemented using a simple linear search.
3. **Levenshtein Distance Matching:** The last type of query uses the Levenshtein Distance, to calculate the distance between two strings. The maximum distance threshold is also from 1 to 3 characters. There are several algorithms to calculate the Levenshtein distance, but only the iterative versions were considered. More details in this to follow.

It is clear that several types of data structures are required for a correct implementation of the solution, not to mention a fast one. One of those data-structures, the BK-Tree was the main point of focus for the previous two assignments. The BK-Tree is a data structure, specifically designed to solve the problem of spell checking, based on a distance metric between to strings. It is used in this assignment for both the Hamming

Distance Matching and the Levenshtein Distance Matching. In the case of Exact Matching, a regular hash table is used.

In addition to the above, there are several other requirements:

- The operating systems and compilers were limited to Linux and gcc 5.4+.
- The application feeding the data to our library could not be modified.

Knowing the details of the problem, it is worth taking a look at the state of the application before the specific task of optimizing it.

Testing Methodology

As with any experiment, a standardized testing methodology is the first part of the process. Here are the sets of hardware, software and testing parameters used to gauge the performance characteristics of the application.

Hardware

Even though the application was developed throughout 3 separate machines, all performance evaluation occurred on a University Linux machine with the following specifications:

Processor

Model	Intel Core i5-6500
Core Count	4
Clock Speed	3.6 GHz
L1 d/i Cache Size	32 KB
L2 Cache Size	256 KB
L3 Cache Size	6144 KB
Other	Support for SSE4.2 and AVX2

Main Memory

Capacity	15 GB

Software

As the hardware, different pieces and versions of software were used to develop the application but all performance evaluation occurred on a University Linux machine with the following software package:

Operating System	Ubuntu 18.04.6 LTS
Compiler	g++ 7.5.0
Memory Error Detector	valgrind 3.13.0
Profiler	callgrind

Testing Parameters

Each test was performed using the above hardware and software combination. The application was compiled using the -O2 optimization flag and debug features were disabled. Additionally, every test used the small_test.txt file provided by the professors, unless stated explicitly. This file contains 7MB of input data.

For timing metrics, 10 runs were conducted and their average value was used. For memory metrics, the valgrind result was used, for allocation count and allocation size.

Initial Implementation

The goal of the initial implementation was correctness with a minor focus on performance. The data structures and algorithms used though, were chosen solely based on their performance characteristics and the application requirements at the time.

Algorithms

There is not much to the algorithms used in the application. The Exact Matching and Hamming Distance Matching algorithms used are simple iterations over strings.

The Levenshtein distance used was the iterative dynamic programming version with a full matrix cache. It was clear that the recursive version was not quick enough. The reason was twofold:

1. The memory and calling cost of recursive functions.
2. The calculation of already calculated substring distances.

The matrix version was also an attractive solution since the length of the strings we were planning to test was known ahead of time. So a single global cache matrix was allocated and reused for each calculation, saving us from expensive memory allocations and frees.

Data Structures

Here are the main data structures used for the first implementation of the search engine.

Query

A query is a collection of keywords, a type and a maximum distance threshold. It is stored separately from keywords, for data duplication purposes, as a query can be part of multiple keywords and vice versa. The keywords in the query are pointers to the keyword table.

Keyword

A keyword is a word in a query. It is stored separately from queries for data duplication purposes, as multiple queries can contain the same keywords.

The keyword structure has a 32-byte string pre-allocated, since that is the maximum allowable length of a keyword. Additionally, the keyword length and hash are also stored in this structure. Furthermore, to reduce the set of queries that a keyword is part of, each keyword stores a linked list of query pointers.

Keyword Table

Incoming queries were parsed, and their keywords were inserted into the hash table, regardless of their type. This ensured that no duplicate keywords were pressed in memory. The hash table uses a dynamically growing array for the buckets and separate chaining to overcome collisions. The chains were implemented using regular single linked-lists. Upon insertion, the hash table would calculate the load factor present. If that value was over 0.85, the hash table would double the bucket count and rehash all the elements. Element removal and downsizing of the hash table were not implemented.

There was not much thought given into the implementation details of the keyword table, due to the high performance nature of hash tables. The keyword hash function implemented utilized the DJB2 algorithm, a simple, widely used and reliable hash

function. An alternative hash function considered was murmur3 but was deemed overkill for the task.

The keyword table is the data structure where keywords are actually stored. Every other data structure stores pointers into the hash table.

Keyword Tree

The keyword tree is the data structure used to find keywords that match a specified input word, within a specified distance threshold. It is a BK-Tree, where each node stores a pointer to a keyword in the keyword table. Moreover, each node can have a maximum of 31 children, since that is the maximum distance between two strings where the maximum length is 31 characters.

Upon creation, the BK-Tree is given a type that specifies the matching function to be used, whether that is the Hamming or the Levenshtein distance. Keyword trees using Hamming Distance only contain words of equal length. So in the application there are 28 keyword trees. One keyword tree stores all the keywords that are part of at least one Levenshtein distance query, and the 27 keyword trees store all the keywords that are part of at least one Hamming distance query, starting with words with length 4 up to length 31.

Node removal/deactivation from the keyword tree was not implemented.

Query Tree

The query tree is the data structure used to find queries based on their ID. It is implemented using a self-balancing AVL-Tree. Upon insertion, the new query is inserted at the correct location, and then each subtree is rotated properly to ensure the tree is balanced.

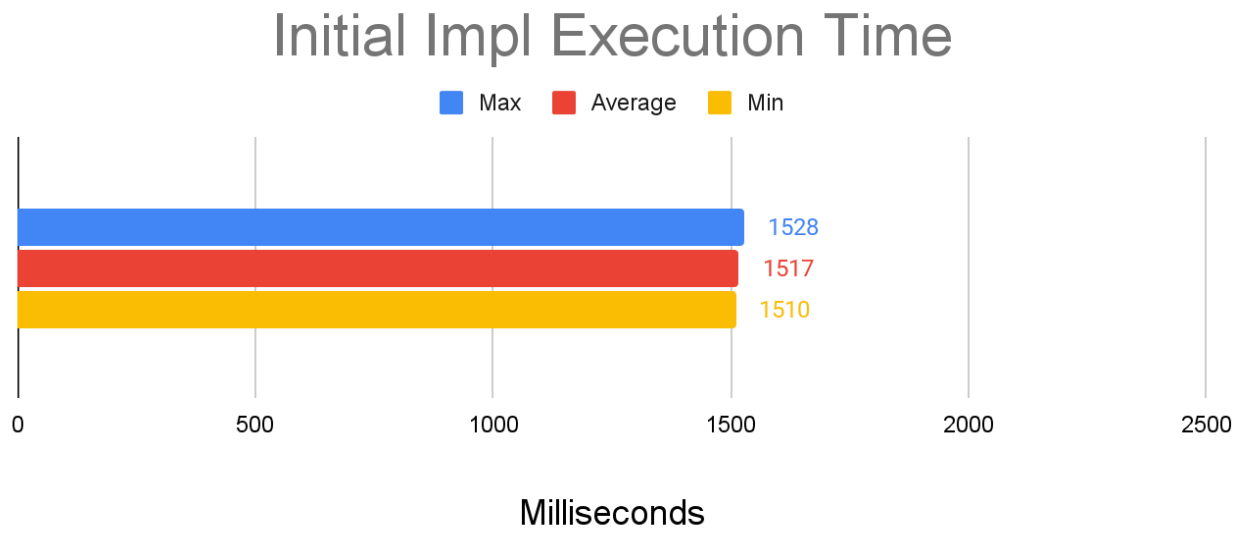
This data structure was picked for its O(log n) average time complexity in all operations. Removal from the query tree was used, even though it required a workaround. Due to the nature of AVL-Trees, during removal the tree needs to remain balanced. This leads to nodes being moved around in memory, changing the memory addressed used to reference these queries in other places, such as the keywords. Thus, an update pass is required everytime we remove a query tree node that has children, to update the keywords referencing it.

Additional Data Structures

A few additional data structures were implemented for utility tasks like function return types and data structure traversals. Those data structures were used regular implementations like lists, dynamic arrays, stacks etc. and are thus not discussed in detail.

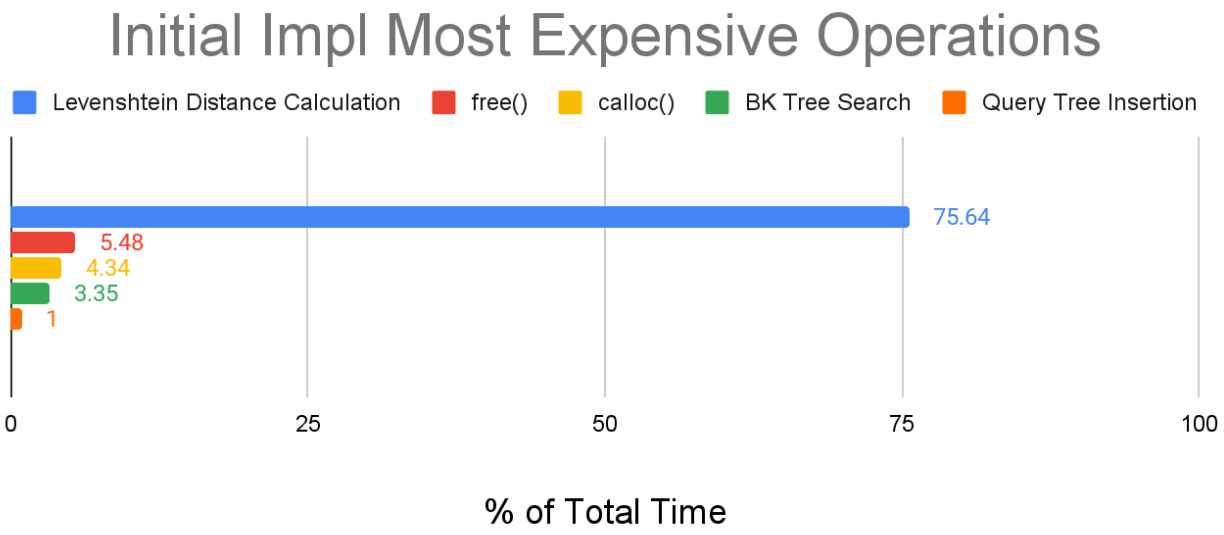
Performance

Since the goal of this assignment was to optimize the performance of the application, it was important to understand the performance characteristics of the initial implementation.



As you can see above, the execution time of the initial implementation averages out to **1517** ms. This time will be the baseline for any subsequent measurements.

With regards to memory usage, there were over **10.5 million** memory allocations totaling over **190MB** of usage over the entire execution time.



Above are the top 5 most expensive operation of the initial implementation. As we expected, calculating the Levenshtein distance is by far the most expensive operation. Coming close to **10 million** invocations of the function this is clearly the point of focus for further optimizations. Additionally it is apparent that memory is not utilized in a very efficient manner. Now let's take a look at some possible solutions to these problems.

Single Threaded Optimizations

The ~76% of the total time being spent in calculating the Levenshtein distance can be easily optimized using multi threading, but at the end of the second assignment it was obvious there were several things than can be optimized further. This list includes TODO items at the end of the second assignment as well as simple and quick optimizations.

1. **Levenshtein distance calculation:** It has been determined that this is the most frequently called function in the entire application, thus it is important to squeeze every bit of performance out of it. The initial version used the global matrix cache. Even for multi threading, this had to transition to a per-thread cache. Another observation is that for each iteration of the loop, only the above row of the matrix and the current row up to the current character was required to calculate the current cost value. So a simple improvement, would be to change to a prefilled max-string-length-sized cache for the previous row (because we know the length of the keywords at compile time) and an accumulator for the current row. This change reduced the number of assembly instructions from **136** to **110**, out of which, **42** are filling the registers with the prefilled cache.

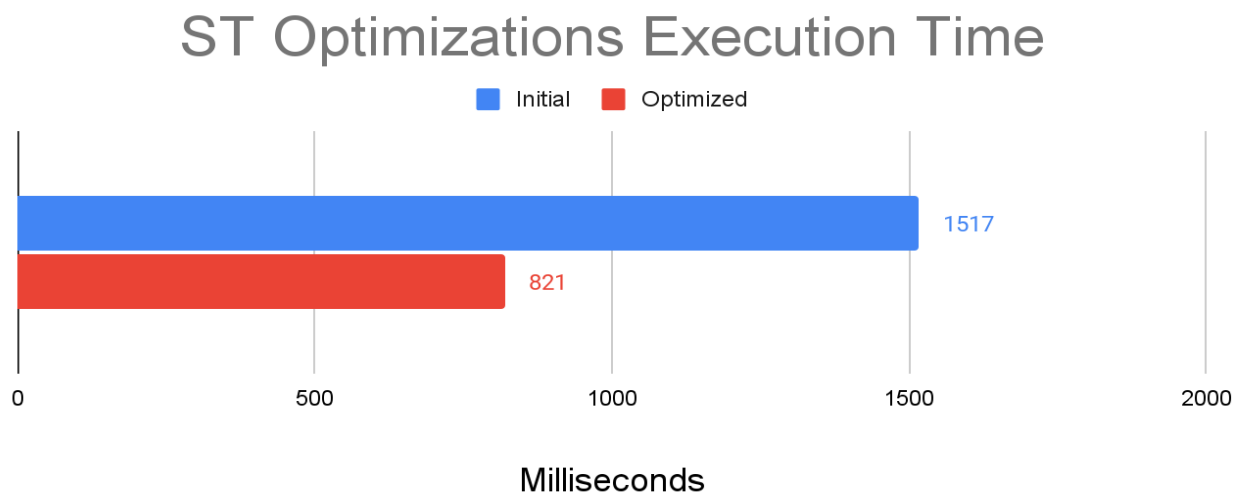
2. **Rethink the way BK-Trees are queried:** In the initial implementation, asking the BK-Tree for matching keywords required a distance threshold parameter. As part of the assignment requirements, that distance threshold cannot exceed 3. So everytime we needed to find the matches to a word from a document, we queried 2 BK-Trees, 3 times each. One for every distance threshold. Apart from calculating the same edit distance of several words and nodes by up to 3 times, we are also loading and unloading tree nodes to/from the cache. If instead of keywords the BK-Tree also returned the distance at which the keyword was found (with the requirement that it was less than 3) we could not only reduce Levenshtein distance calculations but also cache misses by **a factor of 3.**

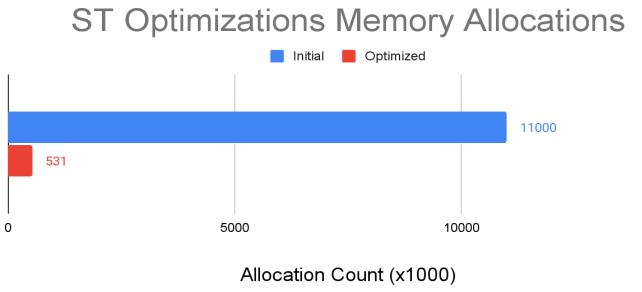
3. **Better memory management during BK-Tree traversal:** Each time a BK-Tree is queried for matching keywords, a node stack is created containing the nodes that need to be searched next, based on their distance to the keyword. A tremendous amount of allocations occur during that time, since the stack is created from scratch and using a linked list implementation. Allocating a global storage space for these nodes, that can grow based on demand and then reused during each query to the BK-Trees makes more sense. Especially for multi threading, where each thread can have its own node storage. This technique can also be used in the above case, where a single match storage buffer is reused by each thread to store the results of every BK-Tree query.

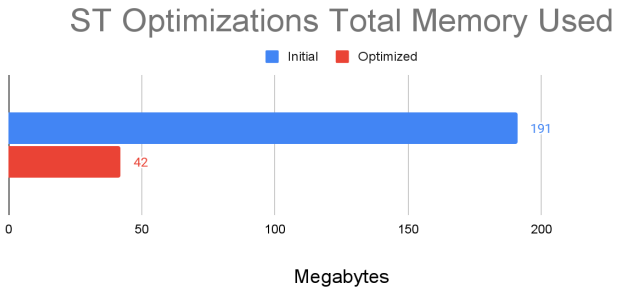
4. **Deactivate BK-Tree nodes that not part of queries:** In the initial implementation there was no deactivation of nodes in the BK-Trees that are no longer needed. This is a fairly quick and easy addition, reducing the amount of keyword matches than need to be processed later down the pipeline.

Performance

The following time metrics utilize the above changes plus some other minor ones and are compared against the initial implementation.







We recorded a **~45% improvement** in the total execution time, a **~95% reduction** in the number of memory allocations and a **~78% reduction** in the total memory allocated. Moreover, the number of Levenshtein distance calculations was reduced by a **factor of ~2.6** down to just 3.8 million invocations. Memory allocations and frees now take up just **~2%** of total time.

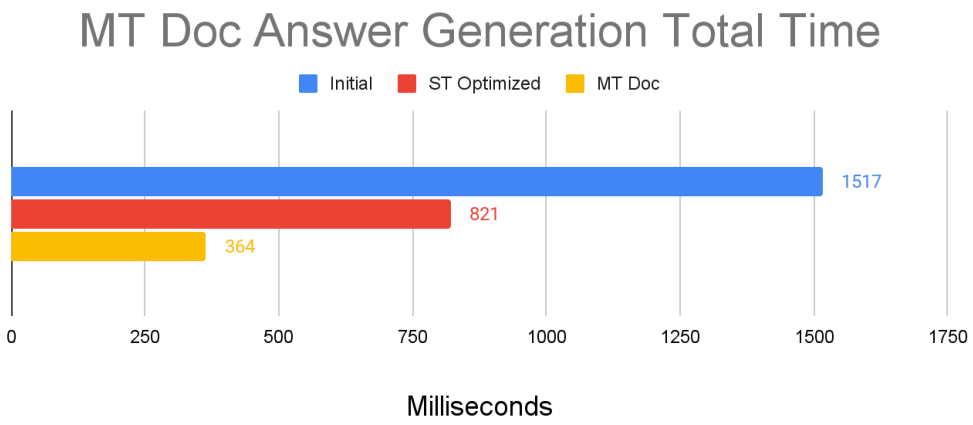
There is still room for improvement but it is time to move to the main topic, multi threading.

Multi-Threaded Document Answer Generation

There are many aspects of this problem that can be multi threaded, but the most essential and easy is document answer generation. This operation needs read only access to all data structures, so if we ensure that there won't be any concurrent write operations, no mutual exclusion or synchronization is required. And fortunately this is the case.

Due to the way the driver application works, every document needs to be answered before any write operations take place (query registration/unregistration). Thus this problem becomes very easy.

We have both a single-producer-multiple-consumer and a multiple-producer-single-consumer problem. Submitting documents to the thread pool and requesting answers from the thread pool respectively. Both of those issues use the same implementation. A fixed size circular buffer. One side pushes data at a write location, blocking if the buffer is full. The other side pops data from a read location, blocking if the buffer is empty. Again taking advantage of the driver application, there will never be excess documents/answers. The rest of the implementation of the work queue is fairly simple so let's look at the performance data.



Above are the total times thus far, with the multi-threaded result utilizing **4 worker threads.** The magnitude of this optimization might not be as drastic with the current data sizes, but there is a **~55% improvement** in performance over the single-threaded optimized version of the code. The hot code path as well as memory consumption and allocations remain the same, as expected.

Next Steps

There are many more optimizations that can be done to this application, but mainly due to lack of time, they were not implemented. Here is a rudimentary list of them:

- **Better Memory Management:** Most of the memory is currently allocated at many chunks, which leads to fragmentation. A better solution for this is for every data structure to handle its own memory blocks, and use a memory arena to quickly allocate memory. If the space in the arena ran out, the application could just allocate more blocks and use them. This also enables a much faster destruction process, as there is no need to traverse the nodes of the data structures to free them.

  Another way to optimize memory would be free lists. Free lists are a way to reduce the number of frees and allocations in a node based data structure. All we had to do is store a list of nodes that were allocated but not used, and whenever a new node is requested, we either pop one from the list or we allocate one from the memory arena.

  Lastly, the way the current driver program is parsing the data is quite slow. A better solution would be to read the file in blocks. This way, instead of overwriting the input buffer with every line, forcing us to copy it to our own buffer for deferred processing, the library could use the original pointers.

- **Algorithm Optimizations:** Not much thought was put into this, but maybe there are ways to improve the algorithm performance using SIMD. The string compare and Hamming distance calculations can be easily vectorized, but the Levenshtein distance calculation would be quite difficult.

- **Different Data Structures:** The data structures currently used are good enough, but there could be better alternatives. For example, the query tree. It is a self balancing AVL tree. Due to the nature of the input data, the tree is forced to rebalance with every insertion, because of the incrementing query IDs. Maybe a better data structure to store the queries would be a hash table. This would alleviate the need for a list of query pointers in every keyword, reducing the memory cost. Removal and insertion would both be much faster.

- **Many more micro optimizations.**