PROJECT REPORT

EFFICIENT AUTO-COMPLETION USING BURST TRIES

Daniel Mitton
B00402291

December 10, 2014

Introduction

Many of today's applications have a need for an auto-complete function: an attempt to discover what the user is looking for before they finish entering a query. There are several data structures commonly used in auto-completion, tries and search trees, to name a few. When selecting a data structure there is typically a trade-off between speed and space.

A tries can easily store information (such as word frequency) within its structure; however, this greatly increases its size. Conversely, splay trees require much less space, but often perform slowly due to redundant character comparisons. When dealing with vocabulary accumulation, the burst trie data structure has been found to have lower space requirements, and to perform quickly. This makes it an ideal data structure for use in auto-completion.

Statement of the Problem

This report explores the possibility of a data structure that can have both a low space overhead and fast, efficient performance when providing text auto-completion. The burst trie has proven to be one of the fastest and most space-efficient data structures for vocabulary accumulation. Therefore the burst trie data structure will be used for auto-completion suggestion retrieval and its performance will be compared to ternary search trees and splay trees in space and speed metrics.

By conducting these experiments I will determine whether the burst trie can maintain a space cost equivalent to splay trees (*O(n)*), while performing at least as fast as ternary search tries (*O(lgn)*). Auto-completion will be based on prefixes, and suggestion of a word will be on the frequency with which it occurs in the sample data.

Survey of Known Results

The burst trie data structure was found to be space efficient and fast when dealing with vocabulary accumulation. The data structure is about 2.5 times faster than splay trees and uses the same amount of memory. Burst tries are also 25% faster than ternary search trees and only use 50% of the memory. The burst trie has proven to perform well in tests using small data sets for vocabulary accumulation. Compared to other data structures which may have faltered, this makes it the ideal structure for vocabulary accumulation, and a promising structure for use in the field of predictive text and auto-completion.

Some attempts have been made to improve upon the burst trie, specifically in the form of HAT-tries. These are cache-conscious data structures. Minimizing the amount of caching required, these are suitable for data that may not be able to fit into RAM all at once. Tests have shown that while the HAT-trie can perform between 10-60 times faster than a regular burst trie, it does require additional space.

Additionally, performance testing was done to determine the best trie for top *k* completion suggestions. Three different types of tries were reviewed (the burst trie was not included), with the score-decomposed trie having the smallest memory footprint, and the second fastest speed. The results indicated that auto-completion suggestions could be returned in about one microsecond per suggestion with these methods.

In the area of fuzzy searching and auto-completion within large text collections, several efficient algorithms were provided to deal with errors on the query side and errors within the text collection. These algorithms were also extended to include queries with multiple words, allowing for fast auto-completion as well as auto-correction results.

Research Highlights

Based on the known results above, I have noted several interesting things:

1. The burst trie has a low space overhead and is efficient for all sizes of standard text data sets as opposed to other data structures. This is valuable knowledge to have as it means the burst trie can be used in a wide range of applications without experiencing significant performance loss due to data set size. This would allow applications which span different media platforms to use the same underlying structure, removing bloated code or varying versions of otherwise similar software.

   For example, devices (smart phones and tablets) generally have less memory than a PC. An application designed for a PC that uses auto-completion may have an underlying data structure with a large memory footprint. Before being able to use the application on a device, a different data structure would need to be implemented for auto-completion on the device. The reduction in memory would sacrifice the speed of the application. The burst trie eliminates this issue as it has a low space cost, and remains comparatively fast.

2. The search performance of the burst trie is a combination of single character comparisons while traversing the access-trie portion, and multiple character comparisons for each record traversed within a container:

   n : The number of characters in the query
   k : The number of access tries traversed
   m: The number of words in the container

   The worst case is that you will have to search all of the words when you reach a container. There will also be (***n***-***k***) string comparisons during traversal of the access tries and there will be ***m*** string comparisons within the container.

   Therefore there will be (n-k) + m total comparisons → O(n + m) (Worst Case)

   In the average case, there will be *lg(**m**)* comparisons within a container.

Therefore, there will be (n-k) + lg(m) total comparisons → O(n + lg(m)) (Average Case)

It is also helpful to note that as you traverse deeper into the burst trie, the containers will become smaller as there are fewer words in the English language that are long. As searching for auto-completion suggestions is very similar to standard searching, it should be expected that retrieving auto-completion suggestions will follow similar bounds.

3. While there are have been attempts to improve the efficiency of the burst trie by accounting for cache misses, the HAT-trie, which is based on the burst trie, still requires additional space overhead, making the burst trie more desirable for auto-completion tasks that are memory sensitive. If all of the data can fit into RAM, then a HAT-trie is not necessarily needed. Future work with auto-completion could involve various testing with HAT-tries.

4. Another interesting result is that there are several efficient algorithms for searching with errors either within the query or within the text collection itself. These could be implemented in future versions of the burst trie to create a more robust structure, accounting for misspellings without a significant hindrance on performance or memory.

Experimental Results

The data used for these experiments came from a compilation of news stories made available by Reuters, Ltd., called Reuters-21578. After processing the data, I created three separate word sets of varying sizes. Word Set 1 (WS1) was a straightforward list of the processed words of Reuters-21578. Word Set 2 (WS2) contained all the values from WS1, but added the reverse spelling of each word. Word Set 3 (WS3) contained all the values from WS2, but I added in strings which moved some characters to the middle of the words. These steps were necessary because there were few freely available text collections and I wanted to have enough sample data for my tests. Figure 1 shows additional information regarding these word sets.

Figure 1 – Word Set Information

Word Set	Size (kB)	Total Words	Unique Words
WS1	12,444	2,137,899	57,149
WS2	19,243	3,214,411	133,600
WS3	26,069	4,214,686	179,459

I created five additional data files to provide words for the auto-complete suggestion algorithms within each data structure. These Suggestion Files (SFs) contained 10, 100, 1,000, 10,000, and 100,000 strings respectively. Ten runs were completed for each suggestion file and the results were averaged. These steps were repeated with each word set.

Several data structures were used for comparing space requirements. Ternary search trees required the most space by a large margin, followed by splay trees and then burst tries. For splay trees and burst tries, there were two available methods of storing data, in Java Strings or character arrays. The results in Figure 2 show that Java Strings tend to take up about 32% more space than character arrays in burst tries, and about 33-34% in splay trees.

Figure 2 – Size of Data Structures in MB when built with Word Sets (MB = 1000^2 bytes)

	WS1	**WS2**	**WS3**
Ternary Search Tree	6.820	18.376	25.624
Splay Tree (Strings)	5.307	12.732	17.285
Splay Tree (Char. Arrays)	3.935	9.526	12.978
Burst Trie (Strings), L = 35	5.177	12.348	16.706
Burst Trie (Char. Arrays), L = 35	3.902	9.352	12.671
Burst Trie (Char. Arrays), L = 50	3.826	9.142	12.384
Burst Trie (Char. Arrays), L = 75	3.772	8.984	12.168
Burst Trie (Char. Arrays), L = 100	3.738	8.917	12.066
Burst Trie (Char. Arrays), L = 150	3.712	8.899	11.991

I also tested several different burst limits, bursting containers when they reached 35, 50, 75, 100, and 150 records in size. The burst trie which used Java Strings is only shown with a burst limit of 35 in Figure 2, as it scaled similarly to burst tries which used character arrays. From these results I observed that as the burst limit increases, the size of the data structure is reduced, although there were diminishing returns between limits of 100 to 150. This is a direct result of there being fewer access tries, which take up more space than the containers, and their corresponding overhead costs. Figure 2 also shows that burst tries can maintain a size that is equal to splay trees.

The same data structures were used in speed testing. Figure 3 (page 5) shows the speed at which the data structures could provide a list of five auto-complete suggestions for each string in the suggestion file. These results were averaged over ten runs for each file, and performed on each of the word sets.

We can clearly see in Figure 3 that the ternary search tree performed the slowest. For all but the smallest tests, this structure took almost twice the time required by the next slowest structure. With large data sets, there could be a large number of nodes between the root node and the starting node for a string. For each subsequent character comparison, the issue of traversing many nodes has a high probability of occurring again. This is the most likely cause for the higher-than-expected space and speed values. Another interesting point in Word Set 1 is that splay trees using character arrays performed significantly slower than splay trees using Java Strings. This could be caused by the words in the suggestion files appearing near the top of the splay tree, coupled with Java Strings being compared faster than char arrays. We can see that it levels out by Word Set 2 and 3.

Figure 3 – Speed Test Completion Times in Seconds

Word Set 1					
	SF10	**SF100**	**SF1,000**	**SF10,000**	**SF100,000**
Ternary Search Tree	0.0019	0.0117	0.1148	0.9197	9.4999
Splay Tree (Strings)	0.0020	0.0064	0.0912	0.7165	6.9859
Splay Tree (Char. Arrays)	0.0018	0.0078	0.1138	1.0026	9.8167
Burst Trie (Strings), L = 35	0.0011	0.0060	0.6770	0.4974	4.8672
Burst Trie (Char. Arrays), L = 35	0.0008	0.0049	0.0531	0.4462	4.3719
Burst Trie (Char. Arrays), L = 50	0.0008	0.0050	0.0521	0.4293	4.2096
Burst Trie (Char. Arrays), L = 75	0.0010	0.0051	0.0498	0.4264	4.1741
Burst Trie (Char. Arrays), L = 100	0.0011	0.0049	0.0490	0.4236	4.1374
Burst Trie (Char. Arrays), L = 150	0.0011	0.0046	0.0504	0.4254	4.1586
Word Set 2					
	SF10	**SF100**	**SF1,000**	**SF10,000**	**SF100,000**
Ternary Search Tree	0.0022	0.0143	0.1574	1.3124	13.0124
Splay Tree (Strings)	0.0025	0.0073	0.0793	0.7075	6.9293
Splay Tree (Char. Arrays)	0.0023	0.0069	0.0821	0.7224	7.0765
Burst Trie (Strings), L = 35	0.0011	0.0064	0.0814	0.6885	6.6790
Burst Trie (Char. Arrays), L = 35	0.0010	0.0059	0.0726	0.6151	5.9519
Burst Trie (Char. Arrays), L = 50	0.0011	0.0058	0.0717	0.5946	5.7802
Burst Trie (Char. Arrays), L = 75	0.0010	0.0056	0.0669	0.5839	5.6738
Burst Trie (Char. Arrays), L = 100	0.0011	0.0054	0.0663	0.5772	5.6203
Burst Trie (Char. Arrays), L = 150	0.0010	0.0054	0.0681	0.5948	5.7687
Word Set 3					
	SF10	**SF100**	**SF1,000**	**SF10,000**	**SF100,000**
Ternary Search Tree	0.0032	0.0187	0.1888	1.6100	16.5460
Splay Tree (Strings)	0.0025	0.0073	0.0945	0.8576	8.5002
Splay Tree (Char. Arrays)	0.0027	0.0077	0.0977	0.8562	8.3041
Burst Trie (Strings), L = 35	0.0012	0.0079	0.0988	0.8124	7.8663
Burst Trie (Char. Arrays), L = 35	0.0011	0.0066	0.0869	0.7362	7.0874
Burst Trie (Char. Arrays), L = 50	0.0010	0.0065	0.0835	0.7193	6.9346
Burst Trie (Char. Arrays), L = 75	0.0011	0.0064	0.0794	0.6928	6.8137
Burst Trie (Char. Arrays), L = 100	0.0011	0.0063	0.0783	0.6844	6.6154
Burst Trie (Char. Arrays), L = 150	0.0013	0.0067	0.0795	0.6833	6.6010

Of the nine data structures, the burst trie performs best in speed. The results show that character arrays outperform Java Strings in speed in this type of data structure. The best burst limit appears to be between L=100 and L=150. As the limit size increase, we can see slight improvements in speed, again with diminishing returns. However, once the limit reaches 150, the data shows that some of the times actually became longer.

Conclusions

Through my testing I observed that the burst trie performs faster than other commonly used data structures in returning auto-completion suggestions. It also uses less space than the other structures, due to its hybrid nature of using concepts from tries as well as trees. I determined that ternary search trees have turned out to be inefficient with large data sets, while the self-adjustment of splay trees allows them to be very competitive in both space requirements and speed.

I showed that the burst limit should be adjusted upward from the suggested limit for vocabulary accumulation, going from 35 to a value between 100 and 150. The data showed consistent gains in speed and memory requirements as the burst limit was increased.

This paper set out to determine whether burst tries could perform as fast as ternary search tries while maintaining space requirements of the splay tree. It can be seen from the data that burst tries outperform ternary search trees by up to 250% in speed over a large number of queries, and about 210% in space requirements. Burst tries were also able to keep pace with splay trees, being slightly faster and more memory-conscious. From the results I conclude that burst tries perform well in both small and large data sets, and should be considered a prime candidate for use in auto-complete functions.

References

Askitis, Nikolas, and Ranjan Sinha. "Engineering Scalable, Cache and Space Efficient Tries for Strings." *The VLDB Journal* 19.5 (2010): 633-60. Web.

Bast, Hannah, and Marjan Celikik. "Efficient Fuzzy Search in Large Text Collections." *ACM Transactions on Information Systems* 31.2 (2013): 1-59. Web.

Heinz, Steffen, and Justin Zobel. "Performance of Data Structures for Small Sets of Strings." *ACSC '02 Proceedings of the Twenty-fifth Australasian Conference on Computer Science* 4 (2002): 87-94. *ACM Digital Library*. Web.

Heinz, Steffen, Justin Zobel, and Hugh E. Williams. "Burst Tries: A Fast, Efficient Data Structure for String Keys." *ACM Transactions on Information Systems* 20.2 (2002): 192-223. Web.

Hsu, Bo-June, and Giuseppe Ottaviano. "Space-Efficient Data Structures for Top-k Completion." *WWW '13 Proceedings of the 22nd International Conference on World Wide Web* (2013): 583-94. *ACM Digital Library*. Web.