Autocomplete System for Embedded Devices

Dwijesh Gohil[1] and Tripta Gupta[2]

[1]Department of Computer Science and Engineering, I.I.T. Delhi, Hauz Khas, New Delhi - 110016, INDIA
[2]S/W Solutions, SRI-Delhi, Noida, INDIA

1 Abstract

Autocomplete is a task of understanding the user's need from a sequence of characters and suggesting relevant queries. While being used extensively in mobile phones, search engines, and application search bars, many times the autocomplete system relies on server for storing and processing huge number of strings. For an embedded device where internet may not be available and with contstrained memory, the autocomplete system must store and process these strings on device itself. In this paper we propose a burst trie based autocomplete system for embedded devices. We propose an efficient sigmoid function based probabilistic delete operation, an encoding scheme for compressing common prefixes and suffixes, a deterministic finite state automaton for decoding the encoded information, an implicit user feedback policy to reward or penalize suggestions. Our experimental results on a subset of widely used wiki dataset[RPC] shows a tradeoff between disk space used and execution time, effectiveness of encoding scheme, and advantages of burst trie over DAWG[KBW+].

2 Introduction

All of us use autocomplete feature every now and then without even noticing it. Let it be a search engine, search boxes in various applications, mobile phone key-board, or television. As the user starts typing the query, autocomplete system tries to understand the user's need and suggests top-k relevant queries. Nowadays, for autocomplete systems, the data(strings and its meta-information) is stored and processed on a server and communicated to the edge nodes. When it comes to the embedded devices where internet may not be available, the autocomplete system needs to store a huge number of strings and meta-information in a very compact form on the device's disk space with limited RAM usage. In this paper, we build upon burst trie[HZW02] data-structure and define our own delete, encoding, and decoding procedure to compactly store the information and experimentally show the memory vs time tradeoff.

Data-structures for storing and processing strings can be divided into two categories. Firstly, lightning fast and memory inefficient ones: trie[Fre60], compressed trie[Mal76] and directed acyclic word graph(DAWG)[Jan00]. Double array implementation of DAWG[YMFA08] and trie[AM92] has been proposed to address the issue of inefficient memory. But such array based implementations do

not support dynamic updates. Secondly, slow but memory efficient ones: balanced binary search tree[ST85a], balanced ternary search tree[ST85b], or a linked list. Burst trie takes best of the two worlds. It merges trie and linked list(in this paper) to store and process strings. In the original paper of burst trie[HZW02], authors already points out the advantages of burst trie over various other data-structures except the DAWG. In this paper we experimentally show the advantages of burst trie over DAWG[KBW$^+$].

[DSMMO15] compares various query ranking schemes(e.g. most popular ranker, time ranker, term freq based ranker, etc.) for an autocomplete system but pays a little or no attention to the underlying data-structure, execution time, and memory. Sujit Pal[Pal] compares in-memory trie, in-memory relational database, and Java TreeSet based autocomplete implementations and finds trie to be the fastest, relational database being the slowest(due to SQL engine overhead). Recenly, neural models are getting attention in autocomplte task, e.g. Microsoft IntelliCode[int] and Tabnine[tab]. Since our focus is running an autocomplete system on an embedded device with low compute power and no internet, we do not explore neural approaches here.

In this paper, we use burst trie in a very specific context of embedded devices with constrained memory. Original paper of burst trie[HZW02] talks in general the burst trie superiority, insert and search operation. We specifically focus burst trie for an autocomplete system with the following major contributions:

- We propose a sigmoid function based probabilistic delete operation for efficiency and minimal memory usage. Delete operation only needs to traverse the burst trie data-structure once and depending upon the access frequency of a string, it will delete a string with probability of delete being inversely proportional to access frequency.
- We propose an encoder which encodes the information stored in the burst trie compactly by compressing the common prefixes and suffixes. We propose a decoder: deterministic finite state automaton to decode the encoded string. We propose a way to incorporate implicit user feedback which rewards or penalizes a suggestion.
- The proposed method is evaluated on a subset of widely known wiki dataset[RPC]. Experimental results shows the tradeoff between the memory and execution time, effectiveness of encoding scheme, and comparison with DAWG[KBW$^+$].

3 Related Work

In autocomplete, the user types some sequence of characters and the system predicts few relevant query that a user wants to ask. Embedded devices like mobile phone or TV may have small memory and may not have stable internet connection. In such cases, the autocomplete system may not rely on a server for computation and storing the data. It must store the data on embedded device itself.

Trie[Fre60] is a widely suggested data-structure for autocomplete systems. It stores the strings in a tree fashion. Every edge corresponds to a character. If the alphabet size is α then every node holds α pointers, one for each character. This way of naive implementation supports all insert, delete and search in $O(k)$ time where k is the maximum length of a string. But as pointed out by authors[Fre60], it ends up using huge storage space especially when the autocomplete system stores multiple grams of a statement. There are many variants to the trie that has been proposed in the literature: Compressed trie[Mal76], Burst trie[HZW02], and HAT trie[AS07].

Compressed tries[Mal76] is similar to the normal trie but it merges the non-branched paths. If there are sequence of nodes in a path with only one child each then it will merge these nodes in

one node. This way it will save some space by not occupying space for null pointers. But for huge number of strings, it is more likely for a node to have two or more branches and the compressed trie[Mal76] will tend to use memory equivalently to that of the trie. In terms of storing the number of pointers, trie[Fre60] takes $O(\alpha nk)$ space. Where α is the alphabet size, n is number of strings and k is the length of the largest string.

Ternary search tree[BS97] is also a string processing data-structure. Every node stores a character and has at most three children. If a character is smaller than the current node character then it follows left child, if larger then it follows right child, if equal then it follows the middle child. So TST[BS97] needs to store only $O(3nk)$ pointers. While being efficient at memory, TSTs for insert, delete, and search, takes on average logarithmic time in $\#strings$ and in worst case linear time in $\#strings$. Even after using the Balanced TST[ST85b], at best, it will take logarithmic time in $\#strings$. Autocomplete systems needs to be fast and dependency of TST operations on $\#strings$ make them unsuitable to be used in such systems.

Self-adjusting Binary Search Tree[ST85a] can also be used. Here every node holds a string rather than a character. It will store only $O(2n)$ pointers. Search, insert, delete will take logarithmic time in $\#strings$. Again making it unsuitable for an autocomplete system.

More efficient double array based[AM92] implementation for trie also exists. It uses two arrays to store a trie structure in a compact form. But it is only suitable for static data where ther are no dynamic updates(insert, delete) happening to the trie structue. An autocomplete system needs to keep inserting strings and deleting irrelevant strings to adapt a user. This makes double array trie[AM92] unsuitable for the task.

Trie[Fre60], BST[ST85a], TST[ST85b] and their variants are nothing but a deterministic acyclic finite state automaton. DFA state minimization technique can be applied to these structures to get a unique smallest possible structure. In autocomplete terminology, it is also known as Directed Acyclic Word Graph(DAWG). One basic approach to construct a DAWG is to first create a trie structure and then do a state minimization. More efficient way involves incremental DAWG construction[Jan00]. DAWG takes much less memory compared to the trie or its variants. Array based DAWG implementation[YMFA08] has also been proposed. Two different double array DAWG construction techniques has been compared recently[FMiA16]. Search time for DAWG is $O(k)$, where k is the length of the largest string. The only disadvantage is that DAWG is only suitable for static data. Where there are no dynamic updates. This is because the way DAWG is constructed. DAWG is constructed once from a list of sorted strings. It does not support inserting a string in random order. This makes it unsuitable for autocomplete task. DAWG with dynamic updates are also proposed in the literature but they end up taking more space.

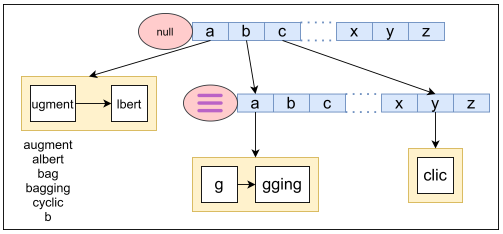

Figure 1: Burst trie example

As pointed out, there exists two worlds: BST[ST85a] (or Linked list) with slow search time - low memory and Trie[Fre60] with efficient search time - more memory. Burst trie[HZW02] combines best of these two worlds. Burst trie is a combination of trie(memory inefficient data-structure) and linked list(memory efficient data-structure). Intermediate nodes in a burst trie are normal trie nodes holding pointers to each characters. Leaf nodes are containers(linked list in this paper). Under certain condition(e.g. $\#elements$ exceeds certain threshold), the container bursts, creates a normal trie nodes and adds children as containers. Authors of burst trie[HZW02] points out many burst heuristics and experiments with many container data-structure(BST, linked list). They provide algorithm for insert and search but not delete. As pointed out by the authors[HZW02], burst trie

takes more memory than BST, less memory than a trie or compressed trie, and is as fast as a trie. Burst trie supports all, insert, delete, prefix search operations. Making it suitable for autocomplete task on embedded devices.

In this paper, we use burst trie for autocomplete task. On top of the original paper[HZW02] we define the delete algorithm, encoding scheme(write burst trie on disk compactly), and decoding scheme(read encoded burst trie from disk) specially designed for storing huge number of strings more compactly on embedded devices.

4 Proposed method

4.1 Burst trie data-structure

Figure-1 shows an example of burst trie for words: "augment", "albert", "bag", "bagging", "cyclic", and "b". Intermediate nodes are normal trie nodes and leaf nodes are containers. As described in the original paper of burst trie[HZW02], container can be BST[ST85a] or Linked list. BST will hold two pointers per node and Linked list will hold one pointer per node. With the motivation of reducing as much memory as possible, our implementation uses a linked list within a container. Figure-2 shows the class diagram of the burst trie implementation. BurstTrie is a class that provides various API calls. BurstTrieComponent is a base class, eRecord, pRecord, Container, and AccessTrie are derived classes. Container holds pointer to the pRecord in a linked list fashion. pRecord is a linked list node. pRecord and eRecord stores access frequency of string in `uint8_t` type. If the string ends up in a container then its meta information will be stored in pRecord, and if it ends up in a normal trie node then its meta information will be stored in eRecord. For example, meta-information about the word "b" will be stored in eRecord as shown in the pink circular node in the figure-1. Access frequency corresponds to the number of times a user has accessed that string. AccessTrie is normal trie node that stores pointers corresponding to each character in alphabet and eRecord.

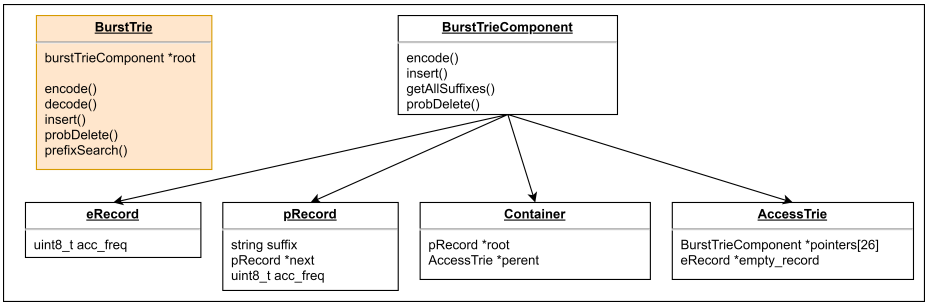

Figure 2: Class diagram for burst trie implementation

4.2 Burst & Delete Heuristics

The container can hold arbitrary number of strings within it. Search within a container is linear for unsorted linked list, logarithmic for sorted linked list. As the container size increases, the search time also increases. It implies that the container size must be bounded. Compared to the trie nodes, container takes much less RAM. This implies that larger the container, less the RAM usage. Considering this tradeoff for container size, our implementation bursts a container if the sum of access frequencies of strings within the container exceeds a threshold and the number of strings

within the container exceeds a threshold. Our implementation only bursts frequently accessed large containers and keeps less frequently accessed containers grow arbitrarily larger.

For autocomplete system on an embedded devices where the data has to be stored in limited disk space, the system may have to delete irrelevant strings. One naive way would be to randomly delete strings. This strategy does not require any sorting but just burst trie traversal. While being efficient, it can delete relevant(more accessed) strings as well. Other way would be to delete all strings that have been accessed less than a threshold times. This strategy will need to traverse the burst trie, hold all strings and corresponding access frequencies in memory, sort them and then delete irrelevant strings. This strategy will delete only the irrelevant strings but will use more RAM.

Probabilistic delete: We propose an another strategy that combines best of the two worlds. It only needs to traverse the burst trie, no sorting is required and is more likely to delete irrelevant strings. We utilize the sigmoid function that projects a real number between 0 and 1. Equation-1 shows the mathematical formulation.

$$P(f) = \frac{1}{1 + e^{g(f)}} \qquad g(f) = \frac{f}{32} - 4 \tag{1}$$

Here f is the access frequency of a string. Since f can at most be 255, $g(f)$ function normalizes it in -4 to 4 range. $P(f)$ represents the probability to delete a string with access frequency f. Larger the access frequency, lower is the probability to delete. This way, the delete operation can be efficiently implemented without any need of sorting and it also preserves the relevanceness up to some extent.

4.3 Encode & Decode Schemes

It is important for an autocomplete system to adapt a user. Every time a user fires a query or visits a webpage, corresponding n-grams will be added to the burst trie. These user specific updates must persist across the sessions. To do so, one naive way would be to traverse the burst trie, and keep writing a word-access freq pair in a file. And read it in the next session and build the burst trie from it. But there also exists an another efficient way to encode the words and access frequencies to save space on embedded devices.

Consider the example in figure-1, suppose all words have zero access frequency then encoded burst trie looks as follows:

$a\#ugment0\_lbert0{-}b{*}0a\#g0\_gging0{-}{-}cy\#clic0{-}{-}$

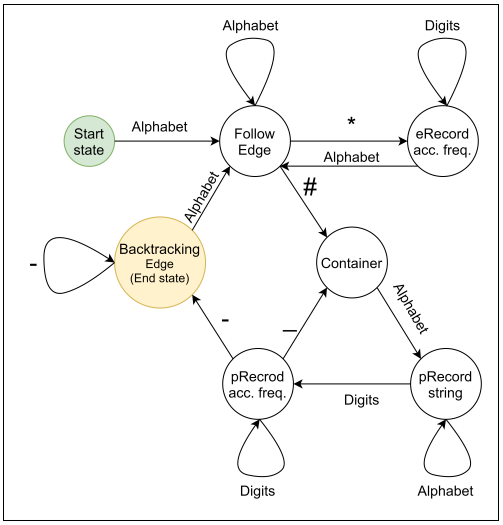

Figure 3: Deterministic finite state automaton to decode encoded burst trie

Traverse the burst trie in depth first search manner and upon following an edge from a trie node, write corresponding edge character, on backtracking an edge write a $-$ character. Every container starts with a # character. Between every container string, write $\_$ character. Every container string is followed by its access frequency. Every non-null empty record is preceeded by $*$ character which is followed by the corresponding access frequency.

This way of encoding will compress the common prefixes since the prefix is written only once for the whole subtree below it. In English language, stop words and suffixes like "ing", "ed", "ic", etc., are very common in a text. So suffixes can also be compressed by replacing them with predefined

characters. In the example(figure-1), "ment" can be replaced with '!', "ing" with '?', and "ic" with '&'. And the final encoded burst trie looks as follows:

$$a\#ug!0\_lbert0 - b * 0a\#g0\_gg?0 - -cy\#cl\&0 - -$$

Figure-3 represents the deterministic finite state automaton(DFSA) to decode the encoded burst trie. It is build in a way similar to how the depth first search would have traversed the burst trie. *Follow Edge* state corresponds to the trie node edge traversal. Depth first search will either stop due to an emtpy record or the container. Therefore *Follow Edge* will either encounter a $*$ character or a # character. *eRecord* state will read the access frequency and will be followed by an alphabet. *Container* state will see the alphabet. *pRecord string* state will read the string and will be followed by the access frequency. *pRecord acc. freq* state will read the access frequency. If there are more strings left in the container then *pRecord acc. freq.* state will be followed by a _ character. If all the container strings have been read then the depth first search would have backtracked the edge. So *pRecord acc. freq.* state will encounter a $-$ character. If there are more children left in a trie node to explore then *Backtracking Edge* state will encounter an alphabet and will continue again.

4.4 Implicit User Feedback

The autocomplete system can retrieve an implicit feedback from the user. Suppose the autocomplete system provides top-k suggestions to the user in the decreasing order of the relevance and the user clicks on i^{th} suggestion. Then it can be assumed that i^{th} suggestion is relevant to the user and has been examined by the user[JSS17]. And all suggestions ranked from 1 to $i-1$ can be penalized by decreasing the access frequency. Clicked suggestion will be rewarded by increasing the access frequency. Access frequency does not overflow. Maximum access frequency can be 255, lowest can be 0.

4.5 Algorithm

Algorithm 1: insert string into the burst trie

```
Input: string: a1 a2 a3 ... an with n characters and root node
Consume character by character and go down from the root node
if string already exists then
    Program flow will encounter pRecord or eRecord;
    Return;
if string does not already exists then
    consumed characters: a1 a2 ... ai;
    remaining characters: ai+1 ... an;
    if flow reaches to a normal trie node then
        if i == n then
            add eRecord if not already exists;
        else
            create a container and insert a pRecord with remaining character string;
    if flow reaches to a container node then
        if no pRecord exists with remaining string then
            insert new pRecord in the end of the linked list;
if burst is required then follow burst algorithm
```

Algorithm 2: Burst container

```
c: Container to burst;
s_i: string stored in the ith record;
j: c is jth child of its parent;
trie_node = new accessTrie();
forall i in 1 to c.numRecords() do
    insert(trie_node, s_i);
c.parent[j] = trie_node;
free container memory;
```

Algorithm 3: Recursive probDelete function for Container class

```
Input: parent node
Output: pointer to the node whose memory has to be freed
forall pRecord p in container do
    if p.needDelete() then
        delete p;
if no pRecord left then
    return this;
else
    return nullptr;
```

Algorithm 4: Recursive probDelete function for accessTrie class

```
Input: parent node
Output: pointer to the node whose memory has to be freed
forall child in trie node do
    if delete_node = child.probDelete(this) then
        delete delete_node;
if empty_record.needDelete() then
    delete empty_record;
if ∃ atleast one non-null child then
    return nullptr;
if empty_record == nullptr and all children are nullptrs then
    return this;
if empty_record != nullptr and all children are nullptrs then
    delete accessTrie node, instead create a container node, store the meta information
      stored in eRecord into pRecord and insert this pRecrod into the container;
    return nullptr;
```

Algorithm 5: prefixSearch

Input: prefix, k
Output: top-k suggestions
start consuming prefix characters and stop when no further traversal is possible;
if *some prefix characters are still left* **then**
    return emtpy_suggestion_list;
else
    suggestions ← collect all possible suggestions by traversing down from the halted node;
    sort(suggestions) ▷ In decreasing order of access frequency;
    return suggestions[:k];

5 Experiments

5.1 Wiki Dataset

Wiki dataset[RPC] contains 2866 text documents. Each document has at least 70 words. Each document belongs to one of the ten categories: art, biology, geography, history, literature, media, music, royalty, sport, or warfare. Total size of all text files is 14.6 MB on disk. Considering the constrained memory on embedded devices, 826 documents(2.7 MB on disk) are randomly chosen as a dataset.

5.2 Compared Methods

Primary focus of autocomplete system on embedded devices are encoding time, decoding time, suggestion time, and disk space. Authors of burst trie[HZW02] already showed the superiority of burst trie over many other data-structures(e.g. red-black tree, binary search tree, ternary search tree, trie, compressed trie, splay tree, and hashing). But authors do not consider the DAWG[KBW$^+$] as any of their baseline. So in this paper, the DAWG[KBW$^+$] and Burst trie has been compared on various factors described above.

5.3 Implementation

For DAWG[KBW$^+$] we use python module. Which internally uses dawgdic C library. dawgdic library uses double array implementation of DAWG similar to [YMFA08]. DAWG code is in python language. Burst trie is implemented from scratch in C++ language. i5, dual core, 7th Gen HP laptop with 8GB RAM is used for the comparison. g++ 9.3.0 and python 3.8.5 is used to run burst trie and DAWG respectively. Burst trie code is compiled with -O3 flag. Unless otherwise specified, for the burst trie implementation, container holds at most 5 strings, all unique 1, 2, 3, 4, and 5-gram strings are used, most common(total 119) suffixes and stop words are replaced with predefined characters.

5.4 Experimental Results

Table-2 compares the time for writing the data-structures on disk averaged over 3 runs, reading and creating the data-structures from disk averaged over 3 runs, suggesting all possible suggestions for a prefix averaged over 10 prefix queries, and disk space used. First column represents the grams used, e.g. 3^{rd} row uses 1-gram(all unique words in the wiki dataset) and 5-gram(all unique 5-grams

in the wiki dataset). To construct k-grams from wiki dataset, first the k-gram of all sentences are generated separately and then only the unique ones are considered for evaluation. Table-1 shows the number of unique strings for various combination of grams. It can be seen from table-2 that for first three tasks: write, read and prediction, DAWG out-performs the burst trie implementation. Burst trie takes significant time to write and read from disk, since it needs to encode and decode a huge string. While being slow, it uses very small disk space compared to the DAWG. Burst trie implementation almost takes less than half the size of what DAWG takes. Which is important for an embedded device where a huge data needs to be stored in small disk space. Prediction time of burst trie(0.01 sec) is also suitable for an offline autocomplete system.

	1, 5	1, 3, 5	1, 2, 3, 4	1, 2, 3, 5	1, 2, 4, 5	1, 3, 4, 5	1, 2, 3, 4, 5
#strings	212439	304483	544730	536322	541137	596268	730644

Table 1: #unique strings from wiki dataset for various combination of grams

Grams	**Write on disk**		**Read from disk**		**Prediction time**		**Disk space**	
	Burst Trie	DAWG	Burst Trie	DAWG	Burst Trie	DAWG	Burst Trie	DAWG
1, 5	1.2 sec	0.06 sec	0.75 sec	0.01 sec	0.003 sec	.001 sec	3.4 MB	14 MB
1, 3, 5	1.3 sec	0.09 sec	0.94 sec	0.01 sec	0.004 sec	.001 sec	3.8 MB	11 MB
1, 2, 3, 4	1.88 sec	0.08 sec	1.44 sec	0.01 sec	0.009 sec	.001 sec	4.9 MB	8.9 MB
1, 2, 3, 5	1.98 sec	0.1 sec	1.5 sec	0.02 sec	0.009 sec	.001 sec	5.6 MB	16 MB
1, 2, 4, 5	2.16 sec	0.08 sec	1.67 sec	0.02 sec	0.008 sec	.001 sec	6.3 MB	15 MB
1, 3, 4, 5	2.6 sec	0.07 sec	1.9 sec	0.02 sec	0.01 sec	.001 sec	7.1 MB	15 MB
1, 2, 3, 4, 5	2.75 sec	0.1 sec	2.14 sec	0.02 sec	0.01 sec	.001 sec	7.6 MB	16 MB

Table 2: Memory and time complexities for Burst Trie and python DAWG[KBW+] module

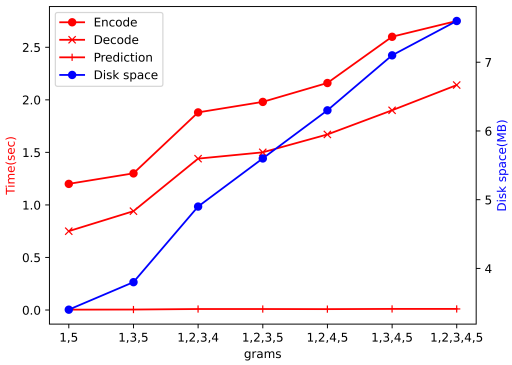

Figure 4: Varying combination of grams

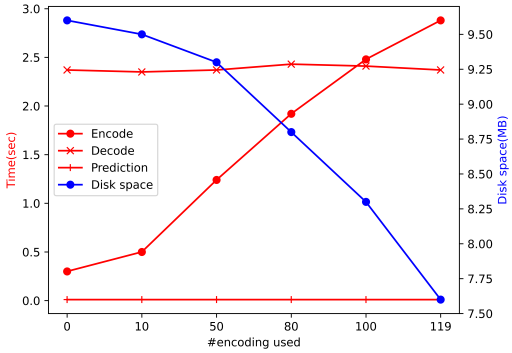

Figure 5: Analysis for varying #encodings

Figure-4, 5, 6 plots encoding time, decoding time, prediction time and disk space against various combination of grams, different container size, and varying #encodings(e.g. how many predefined patterns have been replaced with a predefined character). One point to note is that irrespective of any change, the prediction time always takes ∼0.01 sec. In figure-6, as the container size increases,

the height the burst trie decreases. Hence relatively larger strings are stored in the container. Which increases the overhead of encoding and decoding. As the container size increases from 5, the disk space also increases. This is because as the height of the burst trie decreases, the advantage of writing a common prefix only once per subtree vanishes. In figure-4, as expected, increasing the number of strings in the burst trie will increase encode, decode, and the disk space.

It is important to see if the encoding scheme defined earlier really reduces the disk space or not. In figure-5, the number of patterns(which are replaced with a predefined character) is varied. For example, not replacing any pattern to replacing 119(stop words and common suffixes) patterns. As expected, the encoding time must increase as the number of patterns-to-replace increases. This encoding scheme brings ∼9.5 MB space to ∼7.5 MB space. Which is again an important improvement for embedded devices with constrained memory.

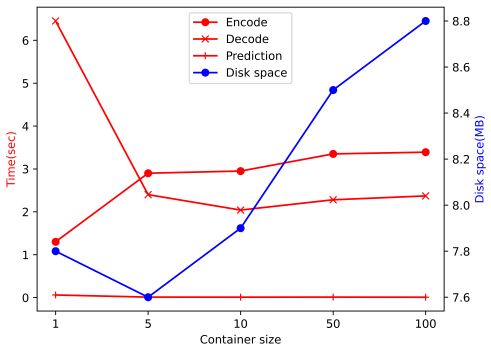

Figure 6: Varying container size

If the disk space usage is a primary aim then using the small container size and as much encoding as possible is preferred. If the encoding and decoding time is a primary focus then usage of small number of encoding is preferred. There is a tradeoff between time and memory. Hence depending upon the application requirements, the parameters must be chosen.

6 Conclusion

In this paper, we proposed a burst trie based autocomplete system designed specifically in context of the low memory, low compute power, and no internet based embedded devices. Our sigmoid function based probabilistic delete function efficiently deletes irrelevant strings from the burst trie. Thanks to the encoding schemes that compresses the common prefixes and suffixes. It enables storing ∼7 lakh strings in ∼7.6 MB disk space. To decode the encoded burst trie, we propose a deterministic finite state automaton. It is also necessary to get a user feedback to adapt and ease the user. We propose an implicit user feedback policy to penalize or reward suggestions. Experimentally, at an expense of encode and decode time(which are executed once at the start and end of the session), our burst trie takes almost half the space of what DAWG[KBW$^+$] takes with almost constant ∼0.01 sec inference time.

References

[AM92] JUN-ICHI AOE and KATSUSHI MORIMOTO. An efficient implementation of trie structures. 22(9), 1992.

[AS07] Nikolas Askitis and Ranjan Sinha. Hat-trie: A cache-conscious trie-based data structure for strings. In *Proceedings of the Thirtieth Australasian Conference on Computer Science - Volume 62*, ACSC '07, page 97–105, AUS, 2007. Australian Computer Society, Inc.

[BS97] Jon L. Bentley and Robert Sedgewick. Fast algorithms for sorting and searching strings. pages 360–369, January 1997. Proceedings of the 1996 8th Annual ACM-

SIAM Symposium on Discrete Algorithms ; Conference date: 05-01-1997 Through 07-01-1997.

[DSMMO15] Giovanni Di Santo, Richard McCreadie, Craig Macdonald, and Iadh Ounis. Comparing approaches for query autocompletion. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '15, page 775–778, New York, NY, USA, 2015. Association for Computing Machinery.

[FMiA16] Masao Fuketa, Kazuhiro Morita, and Jun ichi Aoe. Comparisons of efficient implementations for dawg. *International Journal of Computer Theory and Engineering*, 8(1), February 2016.

[Fre60] Edward Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, September 1960.

[HZW02] Steffen Heinz, Justin Zobel, and Hugh E. Williams. Burst tries: A fast, efficient data structure for string keys. *ACM Trans. Inf. Syst.*, 20(2):192–223, April 2002.

[int] Intellicode.

[Jan00] Jan. Incremental construction of minimal acyclic finite-state automata. *Computational Linguistics*, 26(1), July 2000.

[JSS17] Thorsten Joachims, Adith Swaminathan, and Tobias Schnabel. Unbiased learning-to-rank with biased feedback. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, WSDM '17, page 781–789, New York, NY, USA, 2017. Association for Computing Machinery.

[KBW$^+$] Mikhail Korobov, Dan Blanchard, Jakub Wilk, Alex Moiseenko, Matt Hickford, and Ikuya Yamada. Python dawg module.

[Mal76] Kurt Maly. Compressed tries. *Commun. ACM*, 19(7):409–415, July 1976.

[Pal] Sujit Pal. Three autocomplete implementations compared: In memory trie, relational database, and java sets.

[RPC] N. Rasiwasia, J. Costa Pereira, and E. Coviello. Wiki dataset.

[ST85a] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, July 1985.

[ST85b] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, July 1985.

[tab] Tabnine: Code faster with ai completions.

[YMFA08] Susumu Yata, Kazuhiro Morita, Masao Fuketa, and Jun-ichi Aoe. Fast string matching with space-efficient word graphs. In *2008 International Conference on Innovations in Information Technology*, pages 79–83, 2008.