Todo list

$$N = \text{ total number of trie nodes}$$

$$M = k \cdot l(\text{ length of the dataset})$$

The space for the bitmap with Elias Fano is:

11section*.22), (8) and (9), then the total pessimistic memory usage of sCPT will be: 12section*.23

sCPT : A succinct representation of a Compact Prediction Tree

Rafael Ktistakis, Rajeev Raman
Department of Informatics, University of Leicester, United Kingdom

Abstract

Predicting the next item of a sequence over a finite alphabet has important applications in various domains. *Lossless* approach is a new approach recently introduced by Gueniche et al. [7] which keeps all the training data and yields more accurate predictions.

However, a lossless approach utilises a fairly big amount of memory in order to losslessly keep all the training data. We tested *CPT+* [8] under real life datasets and it utilised 15 to 175 times more space than a compact representation of its training data.This puts some constrains on the size of the training data which will fed a lossless approach like CPT+. Below, we present a *succinct* representation of CPT+ (sCPT) which consistently utilises 1.2 to 3 times the space of compactly represented training data, without any significant performance loss in regards to speed.

write theoretic upper bound for sCPT memory usage

1 Introduction

Let a finite set of items (symbols) $\Sigma = \{e_1, e_2, \ldots, e_\sigma\}$ be called *alphabet*, then a *sequence* S is a list of ordered items $S = \langle i_1, i_2, \ldots, i_k \rangle$ where $i_m \in \Sigma$ for $1 \leq m \leq k$. A multiset of sequences $D = \langle S_1, S_2, \ldots, S_l \rangle$ also called *dataset* is constituted by sequences which share same characteristics. The total sum of all the sequences items defines the length of the dataset M. Given a dataset, a prediction algorithm is *trained* on this dataset. Once the prediction algorithm is trained, it will repeatedly accept a *context* which is another sequence of symbols, and outputs a *prediction*. The context is obtained from an unknown *query sequence* which does not belong to the dataset but shares common characteristics with the dataset. Loosely speaking, the context is derived from a part of the query sequence, and the prediction is about what comes *after* the context in the query. What comes after can take different meanings depending on the application that we target.The process of what the predictor predicts is defined by a prediction task (Appendix C). Each prediction task can be correlated with numerous applications such as, products recommendationt, web page prefetching, compression algorithms etc.

Compact Representation of training data (Γ) will play an important role as a comparison measure of memory utilisation, in this report. It captures the total size of training data that is given as an input to a predictor. It is defined as $\Gamma = \log(\sigma) \cdot M$ bits and constitutes a base for comparison among different predictors and their memory utilisations.

2 CPT and CPT+

In this section we present the structures that CPT predictor uses. Then, we present additional structures and strategies which CPT+ uses to achieve less memory usage than CPT. The predictor of CPT and CPT+ is presented in Section 2.3. Lastly, in Section 2.4, we go through strengths and weaknesses of CPT and CPT+ and we present a memory analysis of both CPT and CPT+.

We sum up by presenting results of experiments with real-life datasets where the weaknesses of CPT and CPT+ appear.

2.1 Data structures of CPT

A trie (*Prediction Tree PT*), an Array of bit-vectors (*Inverted Index II*) and an array of pointers to the last items of sequences inserted to the prediction tree (*Look-up Table LT*) constitute the structures of CPT. Figure 1 shows an example of CPT during multiple sequence insertions. When a sequence is given to CPT, the following steps are followed:

1. Add sequence in the trie (PT)
2. Add a pointer to the end of the inserted sequence in the trie
3. Add a column to the II. The column will define which of the alphabet items the sequence contains.

Later, when we need to locate sequences by using the CPT structure, we apply bitwise-AND operation in II for locating sequences that contain specific items that we search. The II will show us which sequences contain such items. The LT will point at the end of those sequences. The PT can be traversed upwards and hence we can retrieve back a sequence in its initial form. This kind of process is followed by the CPT and CPT+ predictor as we are going to explain later.

Prediction Tree

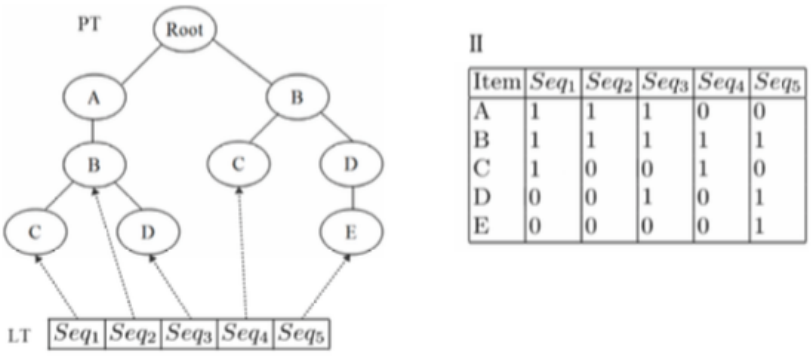
The prediction Tree of the CPT Predictor is constituted by a trie structure (Figure 1). After the build phase where the prediction tree has been built, the main operation that the predictor needs to execute, as a part of a prediction task, is to retrieve back sequences. Which sequences will be retrieved is a matter that is decided by the II. Hence, the prediction tree is important to support instructions that give back the relative sequences. Since the LT points to the end of every sequence, it makes sense to retrieve sequences beginning from root and going up till the root. The retrieved sequence can be reversed in a later time. Therefore, a *getparent()* instruction should be supported in a node of the prediction tree. Currently, the trie offers such functionality which takes $\mathcal{O}(k)$ time for the sequence retrieval to get completed. The space required for the PT is a factor of the total number of nodes. Every node needs 32 bytes of stored information which includes a pointer to the parent node, pointers to all children nodes and the label of the node.

Inverted Index

The II plays an important role in finding which sequences should be retrieved from the prediction tree. As shown in Figure (1) every column of the II shows for a sequence which alphabet items appear within the sequence. The main operation that is needed is a bit-wise AND among different bit-vectors of the II. Depending on the query sequence items, different bit-vectors are chosen from the II to apply a logical bitwise-AND operation. The result is a bit-vector which shows which sequences contain all the query items. According to the above description, it is easy to observe that the II gets quite sparse. This is because usually a sequence does not contain all the alphabet items. A sequence of a real-life dataset has a length as low as couple of items to as high as less than hundred items. However, the alphabet size can be quite large as thousand of distinct items. Thus, II uses space more than the actual needed which affects both memory and performance.

Look-Up Table

The Look-Up Table is the structure that adds the necessary "glue" between the II and the Prediction Tree (figure 1). After that the II calculates which sequences contain the query items then the LT will simply point to those sequences in the PT. The LT is an array with pointers to every end of every sequence in the PT. The pointer for the 1st sequence is at the 0th position of the LT, the pointer for the 2nd sequence at the 1st position etc. As the II, the size of the LT depends on the number of the sequences that the CPT is trained on. For every sequence, the LT uses 64 bits of space. In Section 3 we present a more elegant solution where we can omit LT completely.



II

Item	Seq_1	Seq_2	Seq_3	Seq_4	Seq_5
A	1	1	1	0	0
B	1	1	1	1	1
C	1	0	0	1	0
D	0	0	1	0	1
E	0	0	0	0	1

Figure 1: A Prediction Tree (PT), Inverted Index (II) and Lookup Table (LT)

2.2 Data structures of CPT+

CPT+ keeps the data structure of CPT, however it applies two main new strategies for improving the memory utilisation of CPT [8] (Figure 2). The first one, called *Frequent Subsequence Compression (FSC)*, identifies the frequencies of subsequences that appear within a dataset. Each frequent subsequence can be assigned a unique id. This unique id can replace consecutive nodes of the PT that constitute the same subsequence. Of course, a new structure has to be introduced in order to hold the correlation between a frequent subsequence and an id. A *subsequence Dictionary* takes this role and translates a subsequence to an id (or vice-versa) in $\mathcal{O}(1)$ time.

The second strategy is the *Simple Branches Compression (SBC)*. A *simple branch* is a branch that leads to a single leaf. As a result, each node of a simple branch has between 0 and 1 children. This strategy identifies the simple branches and replaces them with a single node which represents the whole simple branch.

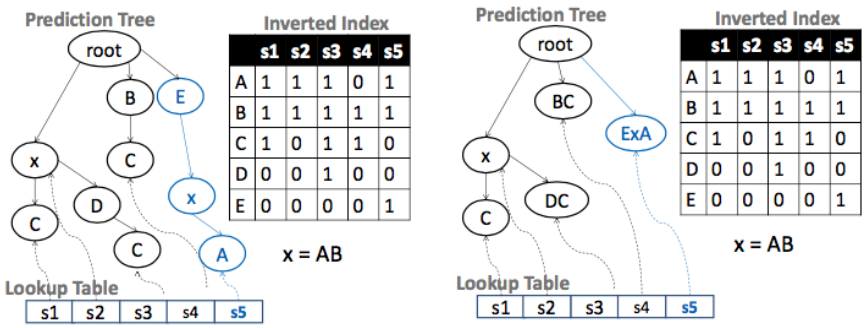


Figure 2: FSC and SBC compression strategies

2.3 CPT and CPT+ Predictor

In order for the CPT to be used for predictions, the aforementioned data structures have to work together providing the necessary information when it is needed. A bitwise AND shows which sequences contain the query items and then these sequences can be located by using the Look-up Table which points to the end of each corresponding sequence in the PT. Then, PT can be traversed backwards (going to parent until root is reached). Having retrieved a requested sequence from the PT and along with the query sequence, a *consequent* can be resulted as a part of the retrieved sequence. The query items are used as a set of items and each one of these items is matched in the retrieved sequence and removed from it. Having removed all these items from the retrieved sequence, a *consequent* is created. Consequent is eventually the part of the retrieved sequence that specifies what comes next after a query sequence. For example, if the sequence $S_1 = \langle a, b, x, z, c, d, e \rangle$ was retrieved from the prediction tree and the query sequence is $S_q = \langle a, b, a, c \rangle$ then the consequent will be $S_c = \langle d, e \rangle$. CPT+ follows the same process for retrieved the matched sequences from PT through LT and II, however as a final step the retrieved sequences has to get *decoded* through the subsequence Dictionary.

CPT uses different techniques [7] to make use of the consequents. Such techniques could facilitate different prediction tasks (see Appendix C). Recursively shrinking the query size (creating sub-queries) and producing multiple consequents is one of them. All the consequents then are inserted into a table that counts the occurrences of the items. For each consequent a different weight is applied to its occurrences. This weight depends on the length of the corresponding sub-query. Finally, the item with the highest count in the table constitutes the prediction. The process for CPT+ is slightly more efficient since it calculates the sub-queries via a *noise* ratio. This noise ratio is simply a measure to filter the items of the query that have a low frequency in the training data. This low frequency is defined by this ratio. Then, the identified "noisy" items can be removed one by one, creating the sub-queries and following the same process as CPT.

2.4 Strengths & Weaknesses

CPT and CPT+ (as lossless approaches) utilise all the available sequences in a trie where they are easily accessible through the II and the LT. Therefore, making predictions can be potentially fast and accurate as described in [7] and [8]. Locating similar sequences can be a relatively easy task that makes CPT and CPT+ work as a good tool for predictors that exploit such sequences. Moreover, CPT can implement all the prediction tasks described in Appendix C, offering a bigger diversity for implementations on several applications. It can be considered undoubtedly as a good state-of-the-art predictor since it is placed among the best 10 best accurate predictors in SPiCe [1] competition.

write about training time weakness

However, even though with the improvement of CPT+, where size of the structures were reduced and speed was enhanced, some performance issues can be still addressed. Bit-vectors are the main element of II which is used for locating CPT+'s sequences. When more and more sequences are being added, the bitwise AND operations are becoming slower and slower, leading to performance issues and overall scalability issues since the vectors are becoming much larger. It is common for real life applications to deal with datasets that contain an enormous number of sequences. Therefore, Bit-vectors can be CPT+'s Achilles' heel regarding its performance. Also, space usage of II in such cases can be wasteful since with *Succinct Data Structures* (see Section A) a better and more memory efficient structure can be proposed (section 3). In addition, CPT+ suffers from memory consistency making it more hard to predict its behaviour in cases large datasets are fed the predictor, since its overall size depends upon the number of sequences.

It is interesting to point out that one of the two compression strategies of CPT+, partially fails to save space from the PT. The SBC strategy depends on simple branches that lead to a single leaf. However, sequences that share same prefixes will be part of the same single branch. LT will point to all those different sequences that share the same single branch. However, replacing a simple branch with a node will result in conflicts with sequences that share the prefix. LT will still point to nodes that should have been freed. If the memory is indeed freed, then LT points to a "bad" memory address otherwise memory saved does not correspond to the numbers of nodes replaced during SBC strategy. Looking at Figure 2, let us assume that LT S_4 sequence points to x. Thus, S_4 is a prefix of S_5. If the branch of sequence S_5 is identified as a simple branch then node x cannot be freed. LT should still point to x and x would point to node E. In this example, the memory saved is zero. Of course there are workarounds for making SBC strategy work on such cases but it is not studied in "CPT+: Decreasing the Time/Space Complexity of the Compact Prediction Tree". The memory evaluation made upon the number of nodes saved and not on the actual memory utilisation as reported by the system. Implementation details for CPT and CPT+ can be found on http://goo.gl/LE4uYO.

In Figure 3 it is presented the memory usage of CPT+ and its data structures as it was reported by Java Virtual Machine (JVM) in respect of each data structure object. For this evaluation, synthetic datasets were used in order to illustrate the linear pattern of the memory increment according to sequence number. Datasets were generated through IBM Quest Dataset Generator [5]. It is an interesting fact that CPT+ utilises almost 40 times the space of Γ. Even the smallest structure like the LT, is around 5 times of Γ. If it is considered that in a lossless approach where all available training data are kept and relatively big quantities of training data might be given, then CPT+ cannot be scalable in applications which depend on *Big Data*.

add figure that shows the memory increase for fixed seq count and variable alphabet

3 Succinct CPT

Succinct CPT copes with all the memory disadvantages of CPT by introducing different succinct data structures for the roles of PT, II and LT. By using succinct data structures and fast operations like *rank/select* (Appendix B), we can achieve a significant memory reduction in performance of CPT+ without relatively sacrificing any speed. The predictor of CPT+ remains the same. This is feasible because the combined succinct data structures offer the same functionalities and instructions as the former CPT+ structures (II, PT, LT). This section will be structured as follows. Firstly, we present a technique for representing Inverted Indices which takes at most $1 \cdot \Gamma$ of space and how this technique can be used for CPT+ predictor functionality. Next, we show how the PT can be represent through a bitmap and how this bitmap can be stored efficiently while it retains the instructions needed for CPT+ predictor. Lastly, we go through the way of omitting the LT structure.

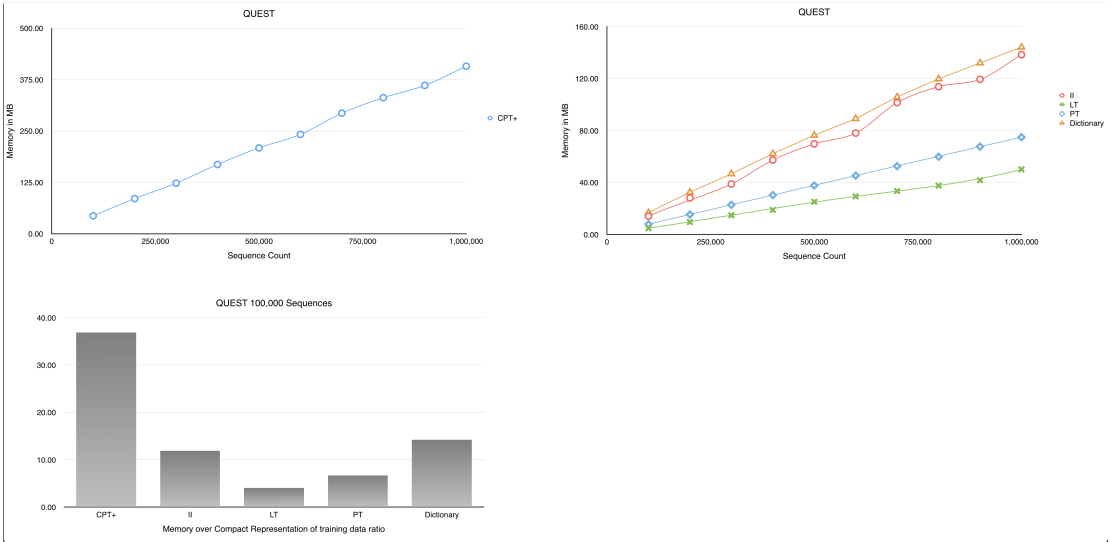


Figure 3: CPT+ memory usage and its structures for datasets of different count number (top); Ratio of CPT+/structures over Γ

3.1 Elias-Fano for Inverted indices

The II as shown in Figure 1 contains which alphabet item is contained in which sequence. However, every bit-vector can be treated as an inverted list of the positions of the set bits. Hence, the bit-vector for item A of the Figure 1 can be written as $\langle 1, 2, 3 \rangle$. The inverted list is for an obvious reason a monotone sequence of numbers. This is important because the compression technique which is mainly being used and described next, is based on *gap encoding*. As it will be proved later, gap encoding will make possible to store the II in space smaller or at least same as the space of the initial dataset. The techniques described below are based on a paper by Elias [3] and a paper which demonstrates similar ideas for representing Inverted Indices [15].

Elias Techinque

It is clear that every bit-vector of the II can be seen as an inverted list of the sequences that the alphabet item is contained. Because the list is monotone can be seen as

$$0 \leq x_0 \leq x_1 \leq \cdots \leq x_{\text{n - 2}} \leq x_{\text{n - 1}} \leq u$$

where $u > 0$ as a known upper bound. That bound is always known because during prediction phase the dataset size is known too. Now, this list of numbers can be split in two bit arrays according to the formulas below:

lower bit array: for every item of the list, the array continuously stores its lower λ bits where $\lambda = max\{0, \lfloor \log(u/n) \rfloor\}$.

upper bit array: then the upper bits are stored in a sequence of unary-coded gaps (see Figure 4)

Space property

Space wise, it can be proven [15] that for every item of a list it is used at most $2 + \lceil \log(u/n) \rceil$ bits. The dataset containing the sequences, from where the II will be created, can be considered

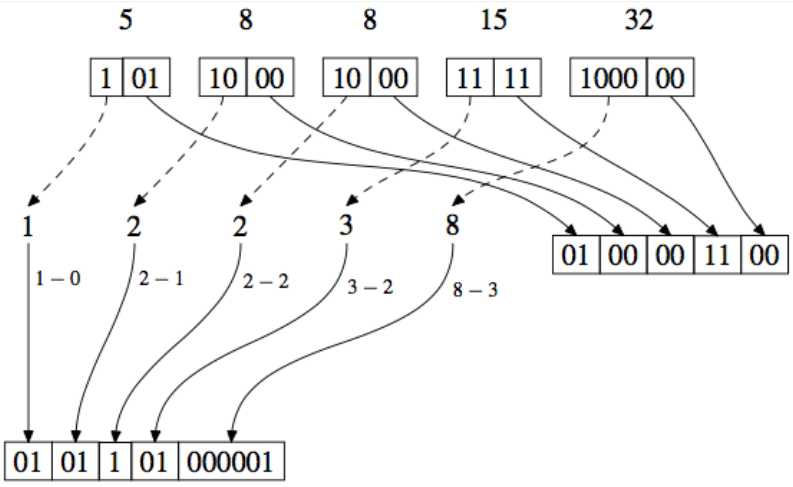


Figure 4: In this example we have a list of numbers 5, 8, 8, 15, 32 with upper bound 36. For the lower bit array $\lambda = \lfloor \log 36/5 \rfloor = 2$. On right it is shown all the λ bits of all elements which are gathered together to create the lower-bets array. On left, the gaps for every number upper bits are calculated and stored sequentially in unary code in the upper-bits array.

as a memoryless source. Then, every symbol appears with a probability of $p_\kappa = \frac{1}{\sigma}$ in the dataset. The bits needed to code a symbol will be $\sum_{\kappa=1}^{\sigma} p_\kappa \cdot \log(\frac{1}{p_\kappa}) = \log(\sigma)$. An II that uses Elias way of recording the set bit positions for every bit-vector, will take a total space:

$$\sum_{\iota=1}^{\sigma} n_\iota \cdot (\log(\frac{l}{n_\iota}) + 2) \tag{1}$$

We set as M to be the total length of the dataset. Then:

$$M \geq l \tag{2}$$

Using (1), we can oversimplify it as:

$$\sum_{\iota=1}^{\sigma} n_\iota \cdot (\log(\frac{M}{n_\iota}) + 2) \tag{3}$$

For any bit-vector of the II:

$$p_\iota = \frac{n_\iota}{M}, \text{ using (2)} \tag{4}$$

Finally, using (3 and (4)), the total memory of an II using Elias technique will be:

$$\sum_{\iota=1}^{\sigma} M \cdot p_\iota \cdot (\log(\frac{1}{p_\iota}) + 2) \tag{5}$$

In last equation (5) and by claiming that the oversimplification at (3) is usually $M > l$, it can be assumed for most cases that the total memory is less than the memory needed to code the whole dataset. So:

$$\sum_{\iota=1}^{\sigma} M \cdot p_\iota \cdot (\log(\frac{1}{p_\iota})) \leq M \cdot log(\sigma) \tag{6}$$

This final statement is important due to the fact that most of the times a usual data structure consumes space that is at least an order of magnitude of the initial input data size. In the above

case, the space which is used by an inverted index with Elias Fano representaions, is usually less or the same as the size of the input data.

Accessing Values

Accessing and retrieving back any random value x_i stored with this way can be done by following the steps on the upper bit array and applying rank/select operations (Appendix B):

1. $[\frac{x_i}{2^\lambda}] = \theta$
2. $\varrho = select_0(\theta)$
3. $\zeta = rank_1(\varrho)$
4. concatenate (the binary representations) of $\varrho - \zeta$ with the value of lower bit array at index ζ

Similarly, we can continue accessing all the values sequentially by keeping track of the number of the 0_s parsed in every position of the upper array. Every time that an 1 is met then we have met a new stored value and therefore we should concatenate our bits. A 0 means that we have to increase our variable that counts zeros since the gap to our next number is being increased.

II in sCPT

CPT predictor uses the II in order to determine which sequences contain specific alphabet items. With an II which uses bit-vectors this can be done by applying a bit-wise AND to the relevant bit-vectors. We have studied different ways of representing an II. Such representations were the *Succinct Multibit Tree* (SMBT) [13], the *RRR-vectors* [12], the *Wavelet Tree* (WT) [14] and the *Word Aligned Hybrid* (WAH) [16]. Elias Fano representation had the best memory performance which can be also proved (see previous section).

In inverted indices that follow the Elias gap encoding way, the same functionality can be achieved by *decoding* every list of values for every alphabet item and them by applying an intersection among the relevant lists. However, intersecting lists that might contain a relative big amount of values is inefficient because every intersection will depend on the length of the largest list. With Elias-Fano II we can take advantage of the fact that we can *jump in* a specific value of a list and complete an intersection in relation to the smallest list. Therefore, the overall speed performance is not affected much as shown in experimental evaluation.

3.2 PT in sCPT

Currently CPT+ uses trie as a structure in order to store the sequences of a dataset and retrieve the sequences back during prediction phase (Figure 1). However as it has already been mentioned that pointers are needed for every child and parent node in order to represent this linking relation among the nodes. This kind of representation can be a waste of space. On the other hand, having pointers to link nodes together, provide us with a fast, a convenient and a rapid traversal among the trie nodes. Below, it is described a way to reduce the space by a factor of 20 times and without sacrificing speed performance in terms of the CPT predictor needs.

put theoretic space (if I can prove it)

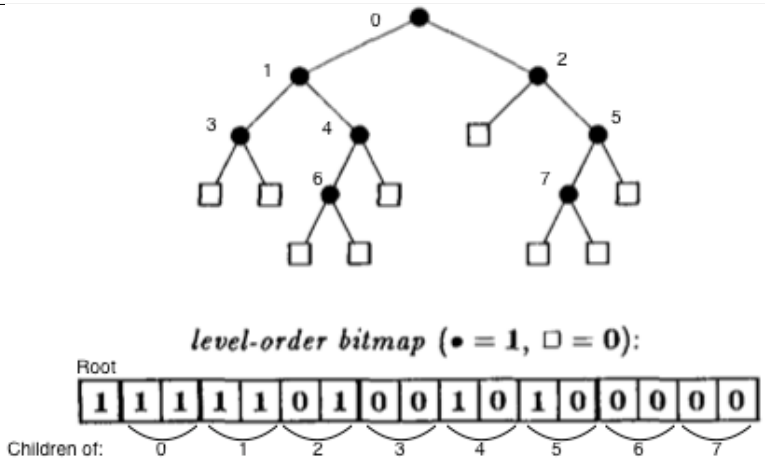


Figure 5: Level-order bitmap representation

Binary Tree as a Level-order Bitmap

A binary tree can be represented as a series of bits, called *bitmap* [9] . Later, it will be shown how relations among nodes can be achieved, same as having a normal binary tree and it can be generalised for any *k-ary* trees. Figure 5 shows a bitmap representation of a binary tree. In order to produce the bitmap of the binary tree, the tree has to be traversed in level order beginning from the root and writing down the first 1 of the bitmap. Then, the next level should be traversed where an 1 is put in the bitmap if there is a node or 0 in case a node does not exist at the appropriate position. As it can be seen from the figure, the number of 1_s written in the bitmap equals to the number of nodes of the binary tree. The overall length of the bitmap is $2 \cdot \eta + 1$ where η is the overall number of nodes of binary tree.

Accessing a child or parent node of the tree can now be achieved by applying rank/select operations on the bitmap itself.

get child: $rank_1(i) \cdot 2 - 1$

get right sibling: $rank_1(i) \cdot 2 + 1$

get parent: $select_1(\frac{i+1}{2})$

Where i is the index of the node in the bitmap. Looking at Figure 5, an example of finding the bitmap index of a parent node located at 9_{th} position would be the node located at 4_{th} position in the bitmap. That is because the parent node of node 6 will be the node 4 (counting in level order). Similarly, it can be checked, at any position, whether there is a right sibling or not. If the remaining of the position index divided by 2 is not zero, then there is a right sibling. Accessing a child and getting a sibling is achieved by using rank operation as described.

K-ary Trees as Level-order Bitmap

The binary tree bitmap can easily be generalised to any K-ary tree as shown in Figure 6. Then accessing child, getting sibling and getting parent are generalised as:

get child: $rank_1(i) \cdot K - 1$

get right sibling: $rank_1(i) \cdot K + 1$

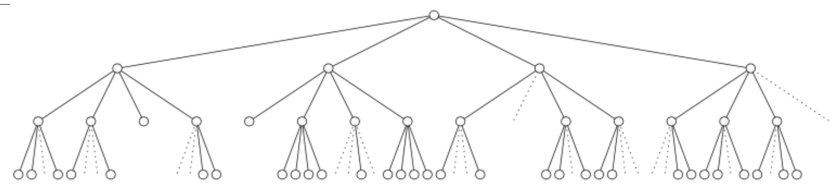


Figure 6: Generalized Jacobson encoding of a 4-ary tree: 1111 1111 1111 1011 1110 1101 1001 0000 0011 0000 1111 0010 1111 1001 1101 1100 0011 1101 1011 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000— 200 bits

get parent: $select_1(\frac{i+1}{K})$

sibling exist if: remaining of $\frac{i}{K}$ is not zero

Storing bitmap

A relative serious drawback of the described approach for representing a tree using a binary bitmap is the fact that the bitmap gets quite long. The overall length is twice as long as the total number of nodes (for a binary tree). Usually, rank/select operations have a space and time overhead that depends upon the type of the bit-vector that they support. Deciding how to store the bitmap could affect both the memory utilisation and the speed performance. According to SDSL-lite library [6], a rank which supports a bit-vector has a 25% of its total length bits overhead and an $\mathcal{O}(1)$ time complexity. These stats vary according to the representation of the bit-vector (such other bit-vector representation is RRR-vectors [12]). CPT+ predictor is mainly depended on accessing parents nodes since LT points to the end of sequences. Elias Fano representation of bit-vectors, as described in section 3.1, is implemented with sd-vectors by SDSL-lite library. Getting the parent node in a bitmap requires only select operation. The select operation on a sd-vector has a space overhead of 64 bits and an $\mathcal{O}(1)$ time complexity. These properties and the way the bitmap is stored (as already explained for inverted indices) make it an ideal option for storing the bitmap of the PT. Accessing the parent node will be achieved quite as fast as in the trie representation of CPT+. At the same time the representation will be succinct which will save a big amount of space.

Proof that a bitmap stored with Elias Fano will take space up to $\Gamma + 2 \cdot M$

$$N = \text{ total number of trie nodes}$$

$$M = k \cdot l(\text{ length of the dataset})$$

The space for the bitmap with Elias Fano is:

$$N \leq M$$

$$\left(\log\left(\frac{(N+1)\cdot\sigma}{(N+1)}\right) + 2\right)\cdot N = (\log\sigma + 2)\cdot N = (\log\sigma + 2)\cdot M = \tag{7}$$
$$= \log\sigma \cdot M + 2\cdot M = \Gamma + 2\cdot M, \text{using definition for } \Gamma$$

3.3 Look-Up Table Omission

In CPT+ Look-Up table is used for pointing in the last node of a sequence within the trie. However shown in Figure 7, the nodes of the trie can be counted in a level-order way and thus an II upon the node count would be easy to be constructed. The 1^{st} sequence will be the 6_{th} column in the inverted index due to the fact that this sequence ends on node 6 in the trie. However, the II length now depends on the total number of trie nodes. The total number of nodes is usually much higher than the total number of sequences (in case that is less due to high sequence repetition then this approach is even better for the II memory utilisation). As a result, the II would contain longer bit-vectors for every alphabet item. It is common to assume that no memory is saved but the opposite happens. On the other hand, sCPT uses bit-vectors coded with Elias Fano technique. It can be proven that the extra overhead per bit-vector will be $log_2(M)$, where M is the ratio of nodes count over sequence number (assuming that node count is higher than sequence number). Also, in most datasets the sequence number is greater than the alphabet size. Thus, this overhead will be less than the 64 bits per sequence which is required by LT.

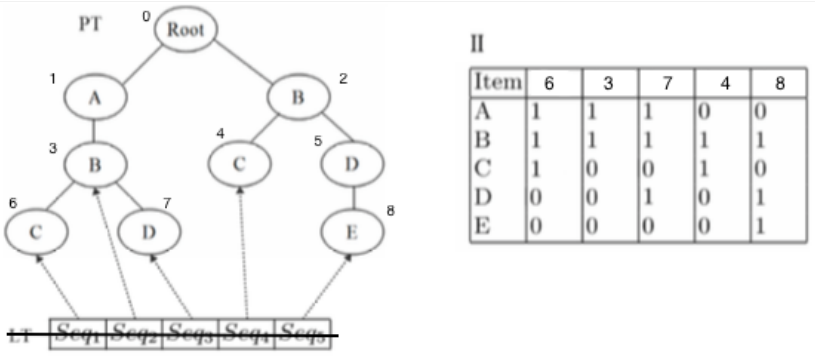
In equation (5), it was proved that an II with Elias Fano represenations takes up to $1 \cdot \Gamma$. However, with the LT omission, we construct II upon the total number of trie nodes (N). If $N \leq l$ then (5) still holds. If $N \geq l$ then the II will take space:

$$\begin{aligned}
&\sum_{\iota=1}^{\sigma} n_\iota \cdot (\log(\frac{N}{n_\iota}) + 2), \text{ where: } N = x \cdot l \\
&\Rightarrow \sum_{\iota=1}^{\sigma} n_\iota \cdot (\log(\frac{x \cdot l}{n_\iota}) + 2) \\
&\Rightarrow \sum_{\iota=1}^{\sigma} n_\iota \cdot (\log(x) + \log(\frac{l}{n_\iota}) + 2) \\
&\Rightarrow \log(x) \cdot \sum_{\iota=1}^{\sigma} n_\iota + \sum_{\iota=1}^{\sigma} n_\iota \cdot (\log(\frac{l}{n_\iota}) + 2) \\
&\Rightarrow \log(x) \cdot M + \Gamma, \text{using equation (5)} \\
&\Rightarrow \Gamma + \log(\frac{N}{l}) \cdot M
\end{aligned} \tag{8}$$

$$\begin{aligned}
&N \leq M \\
&\frac{M}{l} = k, \text{ sequence length}
\end{aligned} \tag{9}$$

Using equations (7), (8) and (9), then the total pessimistic memory usage of sCPT will be:

$$\begin{aligned}
&\Gamma + \log(k) \cdot M + \Gamma + 2 \cdot M = \\
&= 2 \cdot \Gamma + 2 \cdot M + \log(k) \cdot M = \\
&= 2 \cdot \Gamma + M \cdot (2 + \log(k))
\end{aligned} \tag{10}$$



II

Item	6	3	7	4	8
A	1	1	1	0	0
B	1	1	1	1	1
C	1	0	0	1	0
D	0	0	1	0	1
E	0	0	0	0	1

Figure 7: Illustration of how LT can be omitted and have LT take its role

4 Experimental Results and Performance Comparison

For the experimental results a C++ version of CPT and CPT+ implemented. The design and architecture is the same as CPT and CPT+ original implementation in Java 8 (provided at http://goo.gl/LE4uYO). The experiments performed on a Intel Xeon CPU E5-2620 v3 at 2.40GHz, Cache 15MB, 16GB Ram DDR4 at 1866MHz. The datasets that were used are available through SPMF library [5] and constitute real life datasets. The BMS, Kosarak, MSNBC Fifa, Nasa_07 and Nasa_08 datasets consist of webpages logs by users on webpages. SIGN dataset is transcribed from videos, in sign language. BIBLE is constituted by the set of sentences divided into words which represents a sequence.

To evaluate memory, both CPT+ and sCPT were trained with the whole dataset and then the memory calculation performed upon the total memory retained in C++ Heap segment after the training is finished. The speed performance evaluated by feeding the predictor with queries of length 3 items which are generated through the same dataset that trained CPT+ and sCPT respectively. Since the accuracy of the prediction is not currently evaluated and the purpose of this evaluation was to perform a full prediction process (to evaluate speed when predictor does all its steps), a K-Fold [11] validation environment was not set up. It should pointed out that both CPT+ and sCPT use the exact same predictor. What changes is the instruction call to their data structures respectively.

Space Comparison

For the space comparisons the compact representation of training data usage (Γ) will be used as a metric between CPT+ and sCPT. CPT+ utilises $15 \cdot \Gamma$ to $174 \cdot \Gamma$ (Figure 8). sCPT space utilisation lies between $1.2 \cdot \Gamma$ to $4.7 \cdot \Gamma$. The succinct representation of CPT+ can be considered more consistent and predictable since its variance range for memory is much less than CPT+. The inverted indices correspond to the biggest structure in regards of memory consumption among the other structures in CPT+. II in CPT+ takes $0.2 \cdot \Gamma$ to $163 \cdot \Gamma$ while in sCPT $0.5 \cdot \Gamma$ to $2.9 \cdot \Gamma$ (Figure 9). These comparisons illustrate the memory benefits of sCPT over CPT. Therefore, the ability for sCPT to scale is much greater than this of CPT+.

Speed Comparison

Figure 8 depicts the speed performance of sCPT opposite to CPT+. Positive number means that sCPT is slower and negative that it is faster. sCPT through experiments found to be 10% to 60% slower in half of the datasets while its performance considered the same or slightly faster in the

rest of the datasets. The performance differences lie only on the different data structures used in CPT+ and sCPT and not in any variations of the predictor.

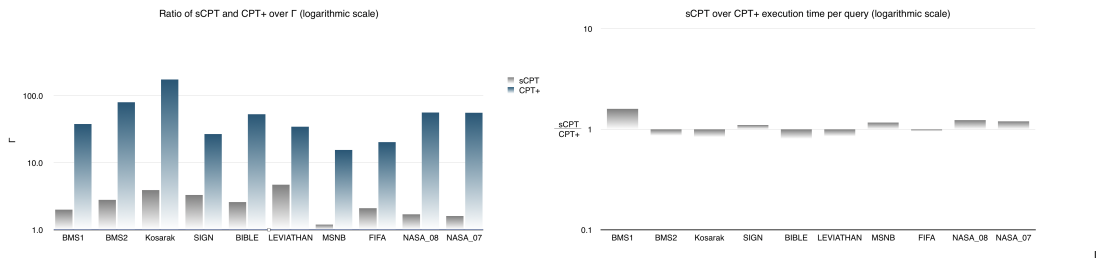


Figure 8: at right: sCPT and CPT+ over Γ, at left: execution time ratio of sCPT over CPT+. The execution time (clock time) measured for 1000 queries, same for both implementations.

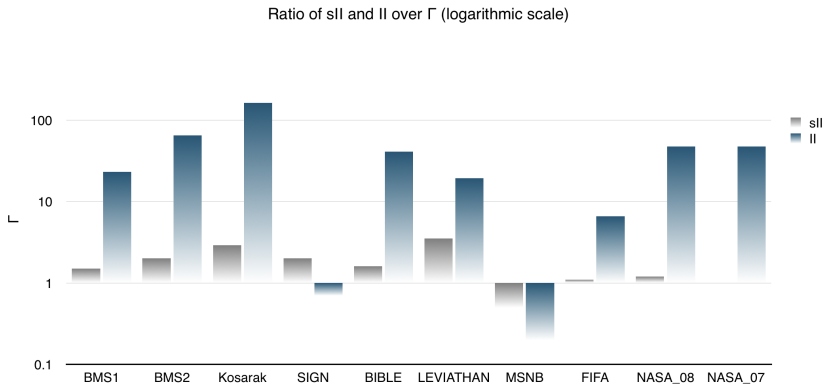


Figure 9: II of sCPT over Γ and II of CPT+ over Γ. Negative number means that the structure takes less space than the compact representation of training data

5 Conclusion

CPT+ is proven to be one of the state-of-the-art predictors which ranked among the best in SPiCe competition. However its memory consumption lack of consistency and predictability which make it difficult for CPT+ to scale up as a lossless approach. sCPT overcomes these weaknesses of CPT+ and offers a consistent and predictable memory utilisation without sacrificing any serious amount of performance. With sCPT the ability of greater scalability is offered. As it was already proven by CPT and CPT+, losslessly keeping all the available training data offers greater accuracy. As a future work for sCPT would be to evaluate the fact that scaling the lossless predictor in a factor where much more training data is available, would improve prediction accuracy even more.

A Succinct Data Stuctures

This kind of structures are high space efficient data structures (introduced by Jacobson 1989) that support rapid and efficient operations, (look Appendix B), like *rank* & *select* [2, 10]. A data structure in order to be considered *succinct*, it should use space that approaches the information-theoretic lower bound of the space that is required to represent the data. One interesting fact is that in contrast with other compressed representations succinct data structures do not sacrifice performance in order to deliver space efficiency when represent and retrieve back any relative data.

A.1 Why Succinct Data Structures

Using *Succinct data structures* can help reducing the space required by a lossless approach and keep most of the data "in memory" (which is stated as a main challenge in Section **??**). This leaves open room to improve an algorithm's scalability. If we consider that compression methods are mostly based in compression and decompression mechanisms, it is almost clear that much time is spent in compressing and decompressing data. Even though, it would have been achieved a space reduction by using a compression method, the overall performance would not have been any good. Even worse, in a scalable level such an algorithm would take huge amounts of time to complete leaving out an important aspect of the algorithm which is scalability. Also, if we consider prediction tools like backward search on FM-Index [4] which their complexity to find a search pattern depends only on the times a rank/select were performed, this leads to algorithms which are more scalable with a robust performance time which is in-dependable of the input training set; aspect crucial for an algorithms performance.

B Functionality of Rank & Select

Let B be a sequence of n items. Then **rank(c)** and **select(c)** can be defined as:

rank(o): Counts the number of items which appear in B from its start to its o-th position [10].

select(o): Finds the o-th occurrence of an element in B [10].

The time complexity of rank/select queries depends on the structure where these two operation are built on. For example, it is possible to execute rank/select in constant $\mathcal{O}(1)$ time for some bit-vectors, like rrr-vectors [12], or another binary rank index [2].

C Prediction Tasks

The prediction algorithm is given a context $\langle q_1, q_2, \ldots, q_j, \rangle$ and the predictor aims to say something about what is in $\langle q_{j+1}, q_{j+2}, \ldots \rangle$. The adoption of which item is predicted after the context and in $\langle q_{j+1}, q_{j+2}, \ldots \rangle$ is only specified by a prediction task:

Right next item: The prediction algorithm gives one prediction for the q_{j+1} coming after the given context.

Top K predictions: The prediction algorithm gives K alternative predictions for the q_{j+1} item of the given context.

Distribution of right next items: For each item of the alphabet Σ, the prediction task gives the item's probability to be the q_{j+1}-th item.

Item in a future window For a given value of w, it is predicted an item that appears somewhere among the q_{j+1}-th, q_{j+2}-th, ..., q_{j+w}-th items. So, w mostly behaves as a *window* value where the predicted item could appear.

References

[1] Borja Balle, Rémi Eyraud, Franco M Luque, Ariadna Quattoni, and Sicco Verwer. "Results of the Sequence PredIction ChallengE (SPiCe): a Competition on Learning the Next Symbol in a Sequence". In: *13th International Conference in Grammatical Inference*. Vol. 57. Proceedings of the 13th International Conference in Grammatical Inference. Delft, Netherlands: JMLR W&CP, Oct. 2016. URL: https://hal.archives-ouvertes.fr/hal-01399429 (cit. on p. 5).

[2] Craig Dillabaugh. *Succinct Data Structures*. 2007 (cit. on p. 15).

[3] Peter Elias. "Efficient Storage and Retrieval by Content and Address of Static Files". In: *J. ACM* 21.2 (Apr. 1974), pp. 246–260. ISSN: 0004-5411. DOI: 10.1145/321812.321820. (Visited on 01/24/2017) (cit. on p. 7).

[4] P. Ferragina and G. Manzini. "Opportunistic data structures with applications". In: *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*. 2000, pp. 390–398. DOI: 10.1109/SFCS.2000.892127 (cit. on p. 15).

[5] P. Fournier-Viger, A. Gomariz, T. Gueniche, A. Soltani, C. Wu., and V. S. Tseng. "SPMF: a Java Open-Source Pattern Mining Library". In: *Journal of Machine Learning Research (JMLR)* 15 (2014), pp. 3389–3393 (cit. on pp. 6, 13).

[6] Simon Gog. *simongog/sdsl-lite*. 2015. URL: https://github.com/simongog/sdsl-lite (cit. on p. 11).

[7] Ted Gueniche, Philippe Fournier-Viger, and Vincent S. Tseng. "Compact Prediction Tree: A Lossless Model for Accurate Sequence Prediction". In: *Advanced Data Mining and Applications* (2013), pp. 177–188. DOI: 10.1007/978-3-642-53917-6_16 (cit. on pp. 2, 5).

[8] Ted Gueniche, Philippe Fournier-Viger, Rajeev Raman, and Vincent S. Tseng. "CPT+: Decreasing the Time/Space Complexity of the Compact Prediction Tree". In: *Lecture Notes in Computer Science* (2015), pp. 625–636. DOI: 10.1007/978-3-319-18032-8_49 (cit. on pp. 2, 4–6).

[9] Guy Jacobson. "Space-efficient static trees and graphs". In: *Foundations of Computer Science, 1989., 30th Annual Symposium on*. IEEE, 1989, pp. 549–554. (Visited on 02/18/2016) (cit. on p. 10).

[10] Guy Jacobson. "Space-efficient Static Trees and Graphs". In: *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*. 1989, pp. 549–554. DOI: 10.1109/SFCS.1989.63533 (cit. on p. 15).

[11] Ron Kohavi. "A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection". In: *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*. IJCAI'95. Montreal, Quebec, Canada: Morgan Kaufmann Publishers Inc., 1995, pp. 1137–1143. ISBN: 1-55860-363-8. URL: http://dl.acm.org/citation.cfm?id=1643031.1643047 (cit. on p. 13).

[12] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. "Succinct Indexable Dictionaries with Applications to Encoding K-ary Trees and Multisets". In: *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '02. San Francisco, California: Society for Industrial and Applied Mathematics, 2002, pp. 233–242. ISBN: 0-89871-513-X. URL: \url{http://dl.acm.org/citation.cfm?id=545381.545411} (cit. on pp. 9, 11, 15).

[13] Yasuo Tabei. "Succinct Multibit Tree: Compact Representation of Multibit Trees by Using Succinct Data Structures in Chemical Fingerprint Searches". In: *Proceedings of the 12th International Conference on Algorithms in Bioinformatics*. WABI'12. Ljubljana, Slovenia: Springer-Verlag, 2012, pp. 201–213. ISBN: 978-3-642-33121-3. DOI: 10.1007/978-3-642-33122-0_16. URL: http://dx.doi.org/10.1007/978-3-642-33122-0_16 (cit. on p. 9).

[14] Yasuo Tabei and Koji Tsuda. "Kernel-based Similarity Search in Massive Graph Databases with Wavelet Trees". In: *Proceedings of the 2011 SIAM International Conference on Data Mining* (2011), pp. 154–163. DOI: 10.1137/1.9781611972818.14 (cit. on p. 9).

[15] Sebastiano Vigna. "Quasi-succinct Indices". In: *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*. WSDM '13. New York, NY, USA: ACM, 2013, pp. 83–92. ISBN: 978-1-4503-1869-3. DOI: 10.1145/2433396.2433409. (Visited on 07/28/2016) (cit. on p. 7).

[16] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. "Optimizing Bitmap Indices with Efficient Compression". In: *ACM Trans. Database Syst.* 31.1 (2006), pp. 1–38. ISSN: 0362-5915. DOI: 10.1145/1132863.1132864. (Visited on 07/27/2016) (cit. on p. 9).