C.CPT

A succinct representation of a Compact Prediction Tree

Rafael Ktistakis
crk15@le.ac.uk
Supervised by: Prof. Rajeev Raman
Department of Computer Science
University of Leicester

January 2017

Contents

List of Figures

List of Tables

1 Introduction

Predicting the next item of a sequence over a finite alphabet has various approaches where some of them contain a relatively extensive literature. The applications which are covered by these various prediction approaches vary and different challenges are addressed for each one. *Lossless* approach is a new approach recently introduced by Gueniche et al. 2013. CPT credited with a fairly good ranking in SPICE (reference) competition opposite to several other *lossy* approaches. However, it suffers from one major problem. Scalability is an issue that currently CPT (and its later improved version) cannot overcome. Its memory utilisations are inconsistent leading to a potentially inability to handle big amount of data. CCPT is an improved version where it successfully handles all the memory issues of CPT while offering up to 90 times less memory utilisation, memory consistency, predictability and retaining the current speed performance.

2 Problem Definition

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 that created under a similar way. 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.

3 Lossless

Lossless sequence prediction is a relatively new approach and therefore appears to have quite a few challenges that someone that studies it has to cope with. It is clear that due to the fact that a whole dataset is utilised, a lossless approach achieves a higher space utilisation over a lossy approach. This fact leads to two results; firstly, it is somewhat intensive for an algorithm to go through all the data and extract the appropriate information needed for predictions, and secondly, keeping all the relative data *in memory*, through the corresponding data structures, appears to be a considerably hard challenge because the represented structure's space utilisation often reaches an order of magnitude of the initial gathered data. As a result, a lossless sequence prediction approach needs sophisticated tools to search for patterns and sequences in *Big Data* when simultaneously the aforementioned challenges are coped. CPT/CPT+ (**C**ompact **P**rediction **T**ree) introduced by Gueniche et al. 2013 and later improved by Gueniche et al. 2015, is currently the only implementations in lossless sequence prediction.

On contrary, a *lossy* approach can be less memory hungry by representing the given information of a dataset through models (like *dependency graphs*). This leads to implementation that sometimes utilises less space and can be potentially faster. However, a lossy approach lacks accuracy especially in hard cases when information in a dataset is not repeating as other parts of informations or a repetition is either generalised or ignored by a model having a partly loss of information that is not reversible. Often such hard cases lead to more complex models that along with Big data is hard to make algorithms that produce predictions with efficiency and performance.

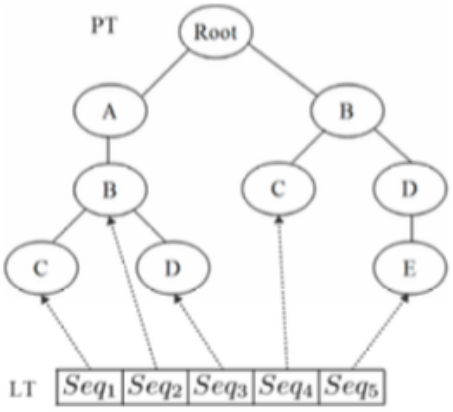
4 Motivation

refer briefly to some SPICE results
Spice led us to the fact that CPT worths to be improved:
1.memory
2. memory consistency
3. accurate memory predictability
4. retaining fairly same speed performance
5. Able to handle efficiently more data, means EVEN better accuracy. Current approach

5 CPT Predictor and its Data Structures

CPT is constituted by 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*). As a lossless approach, CPT utilises the whole dataset provided for building its structures. Each sequence from the dataset is inserted to the Prediction Tree and a pointer to the last item of the sequence is added to the Look-up Table. The Inverted Index has the role of finding which sequence contains which alphabet's items by simply executing a bitwise AND operation. Every bit-vector has the same length with the total number of the dataset's sequences while it notes, using 1/0, whether an alphabet item appears in the sequence or not 1.

CPT can retrieve any searching sequence with specific alphabet items by using the inverted index along with a combination of the two other structures. A bitwise AND shows which sequences contain some given alphabet items and then these sequences can be located by using the Look-up Table which points to the end of each corresponding sequence. Having retrieved a requested sequence and along with the query sequence, a *consequent* can be resulted as a part of the retrieved sequence. The consequent simply constitutes the sequence containing the remaining items after every item of the query (query on this case is used as a set) has been removed from the retrieved 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 $S_c = \langle d, e \rangle$. The produced consequent is then used by a *prediction task* (see Appendix C) in order for a prediction to get calculated.



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)

In Figure 1 it is shown an example of CPT during multiple sequence insertions.

5.1 Strengths & Weaknesses

CPT (as a lossless approach) utilises all the available sequences in a tree where they are accessible through the II and the LT easily and fast by a bitwise AND for each sequence. Therefore, making predictions can be potentially fast and accurate as described in [5] and [6]. Locating similar sequences can be a relatively easy task that makes 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 of several applications.

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 (detailed performance in section 7). 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 6). In addition, CPT suffers from memory consistency making it more hard to predict its behaviour in cases large datasets are utilised.

5.2 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. Regardless of the space usage limitation that it is already mentioned, there is an extra limitation that can be studied in a near future. The fact that the pointers of all the children of a node are being contained in an simple array makes the build phase more and more less efficient for bigger and bigger datasets. That is because for every sequence insertion in the PT, all the children of a node has to be parsed in order to determine whether a new node insertion is required or not. A *Ternary Tree* [1] could solve such a problem during the building phase, but this is not yet studied.

5.3 Inverted Index

During prediction phase the II plays an important role in finding which sequences should be retrieved from the prediction tree. Every bit-vector of the II corresponds to an alphabet item and matches whether this alphabet item appears to a sequence or not. Every index of a bit-vector belongs to a sequence with the 0^{th} index to belong to the 1^{st} sequence of the dataset and the last index to the last sequence of the dataset. 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 the 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 for someone to oversee that the size of the II is increasing along with the number of sequences. In a real-life application, it is usually the case that the alphabet size remains relatively stable. However, the dataset size can be increased as time passes. An example can be the fact of users' weblogs for a specific website that uses a predictor like CPT for caching web-pages. The website has a determined number of pages (constitutes the alphabet), however, the order that a user visits the pages and the amount of the pages that the user visits per session varies. Hence, different transactions are being created (constituting the sequences). Since the application lies on a lossless approach, it means that the bit-vectors of the II would take more and more space. The II is undeniably the less scalable structure to the CPT which takes $\sigma \times l$ bits of space.

5.4 Look-Up Table

The Look-Up Table is the structure that adds the necessary "glue" between the II and the Prediction Tree. After that the II will calculate which sequences contain the query items then the LT will simply point out 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 1^{st} sequence is at the 0^{th} position of the LT, the pointer for the 2^{nd} sequence at the 1^{st} 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. In section 6 it will be presented a more elegant solution where the LT can be omitted completely.

6 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, fast operations like *rank/select* (Appendix B) are offered. Thus, a significant memory reduction is achieved and performance is relatively retained.

6.1 Elias-Fano for Inverted indicies

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 $\langle 1, 2, 3 \rangle$. Since, every position in the bit-vector is correlated with a sequence serial number (1^{st}, 2^{nd}, ..., l^{th} sequence) it is obvious to conclude that the aforementioned inverted list is monotone sequence of numbers. Thist 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 IIs which use such techiques [9].

6.1.1 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 $\lambda = max\{0, \lfloor \log(u/n) \rfloor\}$ lower bits.

upper bit array: then the upper bits are stored in a sequence of unary-coded gaps (see Figure 2)

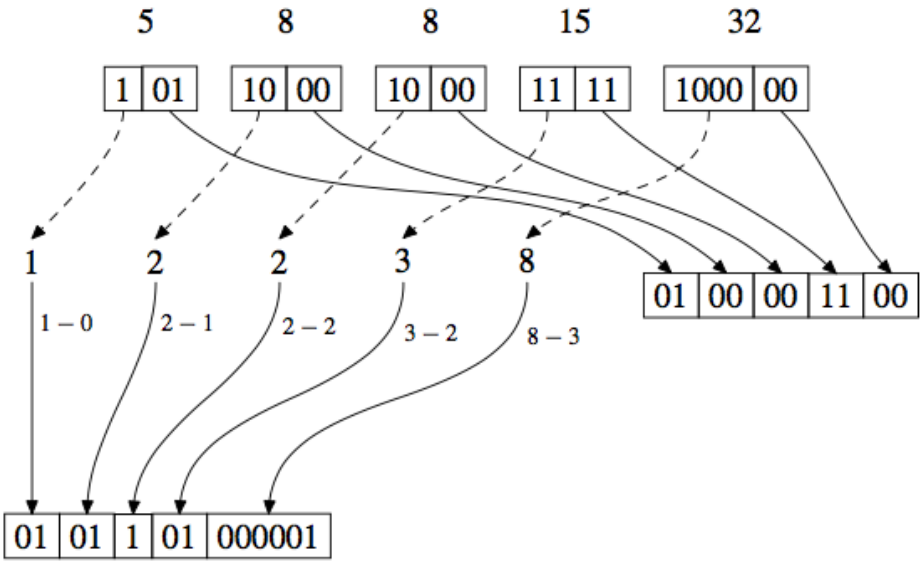


Figure 2: In this example we have a list of numbers 5, 8, 8, 15, 32 with upper bound 36. For the upper bit array $\lambda = \lfloor \log 36/5 \rfloor = 2$. On right it is shown all the λ bits of all elements which are gathered together to create the lower-bets array. On left, the gaps for every number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

6.1.2 Space property

Space wise, it can be proven [9] 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 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 a factor of the initial dataset size. In the above case, the space is usually less or the same. It is not guaranteed but experiments with real life datasets (presented in Section 7) follow the aforementioned behaviour is followed most of the times.

6.1.3 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.

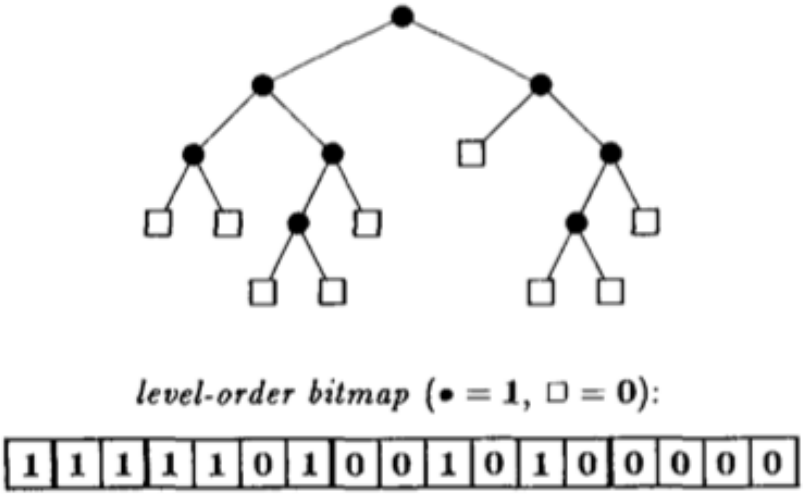


Figure 3: Level-order bitmap ($\cdot = 1, = 0$)

6.1.4 Instructions for CPT predictor

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. In inverted indices that follow the Elias a 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.

6.2 Prediction Tree Encoding and Storing

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 the pointers need to every child node and to parent to represent this linking relation among the nodes, make this kind of representation wasteful of space. On the other hand, having pointer to link nodes together, provide us fast with a convenient and a rapid traversal of the trie. Below, it is described a way to reduce the space by a factor of 20 times (in most cases) and without sacrificing and speed performance in terms of the CPT predictor needs.

6.2.1 Binary tree as a level-order bitmap

REFER TO CPT+ frequent patterns and encoder structure

6.3 Look-Up Table Omission

7 Experimental Results and Performance Comparison

7.1 Space Comparison

7.2 Speed Comparison

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, 7]. 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.0.1 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 3). 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 [7].

select(o) Finds the o-th occurrence of an element in B [7].

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 [8], or another binary rank index [2].

C Prediction Tasks

The adoption of which item is predicted after the context (basically the value of w) is only specified by the prediction task. During this report, there are four different prediction tasks that are mentioned and used; Right next item, Item in a future window, Top K predictions and finally Distribution of right next items.

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] Jon L. Bentley and Robert Sedgewick. “Fast algorithms for sorting and searching strings”. In: *SODA*. Vol. 97. 1997, pp. 360–369. (Visited on 01/19/2017) (cit. on p. 6).

[2] Craig Dillabaugh. *Succinct Data Structures*. 2007 (cit. on p. 10).

[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. 10).

[5] 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. 4, 5).

[6] 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. 4, 5).

[7] 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. 10).

[8] 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 p. 10).

[9] 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).