

大規模データ利用のための圧縮技術

Compression Methods for Handling Very Large Data

岡野原大輔

Daisuke Okanohara

東京大学大学院情報理工学系研究科コンピュータ科学専攻
Department of Computer Science, University of Tokyo

1 はじめに

近年, Web 情報, ゲノム配列, 技術論文集, そして対訳コーパスといった膨大なテキスト情報を利用した情報抽出, 機械学習, 自然言語処理が盛んに研究されている. 大規模なデータを利用することで, 処理結果の精度や性能が向上することが期待されるが, 必要な計算量や作業領域量も大きくなり, 大規模なデータの利用は現実的には難しかった. 本稿は, その問題を解決するための高速な圧縮全文索引, 及び部分復元可能な圧縮法を提案する.

2 圧縮全文索引

圧縮全文索引は従来の全文索引と比較し, 約 1/10 の作業領域量で, わずかな速度低下を伴うが同等の処理を実現するデータ構造である, これを用いることで従来の計算機上でより大きなデータをメモリ上で扱うことが可能となる. 多くの情報抽出, 機械学習, 自然言語処理に必要なタスクは圧縮全文索引が提供する基本的な操作を組み合わせることで実現可能である.

近年提案された圧縮全文索引である圧縮接尾辞配列 (CSA) [5, 8, 4] や FM-index [1] は必要な作業領域が $O(nH_0)$ bit 以下と小さく, さらに検索時に元テキストを必要としない特徴 (*self-index*) を持つため全文索引より大規模なデータを扱うことが可能である. しかし, 実用上の問題点として, Succinct Bit Vector (SBV) と呼ばれるデータ構造の速度, 作業領域の両面で効率の良い符号法の実現が残されていた. SBV は Bit 配列 $B[0..n-1]$, $B[i] \in \{0, 1\}$ が与えられた時, B に対する $rank$, $select$ を報告可能なデータ構造である. ただし, $rank_1(i)$ は, $B[0..i]$ 中の 1 の個数であり, $select_1(i)$ は, $(i+1)$ 番目の 1 の位置である¹. 次章では, SBV を実現する符号法として, Vertical Code と呼ばれる新しい符号化を提案する.

3 Vertical Code

SBV は圧縮全文索引の中で多く用いられる中心的なデータ構造であるが, その速度, 作業領域量の両面で効率的な実装は実現されていない.

本稿は高速な SBV を実現する符号法である Vertical Code (VC) を提案する. VC は 1 が疎な SBV に対して有効なデータ構造である. 圧縮全文索引で利用される SBV は多くの場合 1 が疎な bit vector である. 例えば CSA

¹ $rank_0(B, i)$, $select_0(B, i)$ も同様に定義されるが, $rank_0(B, i)$ は $i - rank_1(B, i)$ として求められるため扱わない. また, $select_0(B, i)$ を使う例は少数であり (例外は *Wavelet Tree* [4]) 本稿では扱わない.

における Ψ の符号化や, サンプリングした SA の位置を保存する場合がそうである (詳細は [5, 8] を参照).

VC は以下の通り実現される. はじめに, Bit 配列 $B[0..n-1]$ を次のようにして, B 中の 1 の間隔を保持した $D[0..m]$ に変換する. 但し $m = rank_1(n-1)$.

$$D[i] \equiv \begin{cases} select_1(i) - select_1(i-1) - 1 & (i \geq 0) \\ select_1(0) & (i = 0) \end{cases}$$

次に, D を一定数 M ($M=8, 16$) ずつのブロック $B_0, B_1, \dots, B_{n/M}$ に分割する. $B_i = D[iM, iM+1, \dots, (i+1)M-1]$ である. また B_i 中の最大の数を 2 進表現した時の最上位 bit の位置を $T[i]$ とする. B_i 中の値は全て $T[i]$ bit を用いて表現できることに注意する.

次にブロック B_i 中の要素をそれぞれ 2 進表現した時の j 桁目の bit を並べたものを $V_i[j]$ とする. VC はこの $V_i[k]$ に加え, 各ブロック先頭の $select$ の値 $S[i] = select_1(iM)$, 及び $T[i]$ を保持する.

VC を用いて, どのように $select_1(i)$, $rank_1(i)$ を求めるかを以下に述べる. $p = i/M$, $q = i \bmod M$ とおく. この時 $select_1(i)$ は, i が所属するブロック B_p の先頭の $select$ の値 $S[p]$ と $D[pM], D[pM+1], \dots, D[pM+q]$ の和として求められる. D の和を求めるには各桁 $V_p[0], \dots, V_p[T[p]]$ 毎に q bit 目までの 1 の数を数えて足し合わせる. この操作は $T[p]$ 回の表引きで行うことができる. $rank_1(i)$ を求めるには $select_1(i)$ の結果を用いて 2 分探索を行う. このことから次の定理が成り立つ.

定理 1 VC を利用した $select_1$ の最悪計算量は $O(\log n)$ ($rank_1$ は $O(\log^2 n)$) であり, 最悪作業領域量は $O(m \log(n/M))$ である.

最悪計算量が必要なデータの例は稀であり, 実際のデータを使った殆どの場合には, 同一ブロック中の値は似た値をとるため $select_1$ の計算量は $O(\log(n/m))$ ($rank_1$ は $O(\log n \log(n/m))$), 作業領域量は $O(m \log(n/m))$ である.

VC の工学的な特徴はブロックサイズ M が 8 の倍数の時, $d[i]$ の値がどのような場合でも $V_i[k]$ は全てバイト単位で表現でき, 上の操作は全てバイト単位の処理のみで行うことが可能な点である. これにより bit 単位の処理を行う場合に比べ大幅な高速化を達成できる. また, 従来の γ 符号や δ 符号等をはじめとした整数符号法が整数を unary 符号と binary 符号を用いて表現しているのに対し, VC は unary 符号を必要としないため復号が高速である. この unary 符号を必要としない点は Recursive

Integer Code[6]と同じである。VCが必要とする $S[i]$ は $T[i]$ は[6]と同様に、再帰的に同じデータ構造を用いて保存し、作業領域量の削減が可能である。

CSAを構築する際、 Ψ の符号化に、VCを使った場合と γ 符号を使った場合では、作業領域量はほぼ同じであったが、VCを使った方が約10倍高速であった。

4 Static PPM

大規模なデータを圧縮して利用する場合、一般的なデータ圧縮は先頭から復元処理を行うため、任意の位置からの復元を行うには非常に時間がかかる。また、データをブロックに分割してそれぞれを圧縮する場合でも全体の情報が使えないため圧縮率が著しく低下する問題があった。本稿で紹介するStatic PPM (SP)[7]は従来のデータ圧縮法と同等の圧縮率を保ちながら任意の位置からの復元を実現する。任意の位置からの部分復元が可能なデータ圧縮法は他にLZ78[9]法、BWT[2]法を利用したものが提案されている。

PPMは1文字符号化する度に各文脈での文字の出現頻度情報を更新し、それを用いて次の各文字の出現確率を予測し、その確率に従って符号化する。そのため部分復元を行うときに各位置での文脈情報を前から順に復号していかなければ再現できない。それに対しSPはデータ全体を通して解析した静的な頻度情報を用いて次の文字の出現確率を予測する。SPは圧縮時にまず文脈ごとの文字の出現頻度を表現する木(予測木)を作成し、それを用いて入力文字列 T を圧縮する。出力サイズは文字列に対する符号に加えて、予測木自体のサイズも含まれる。部分復元を可能とするため、データを長さ w 毎のブロックに分割し(w 文字処理する毎に履歴情報を初期化する)、各ブロックを符号化しているビット列へのポインタを格納する。圧縮と復元はブロック単位で行なう。通常のPPMと異なり、SPは1文字圧縮・復元するたびに予測木を更新する必要が無いため高速である上に、予測木は全てのブロックで共通のものを用いるため、ブロックサイズを小さくしてもあまり圧縮率が落ちない。 $w = 256$ 程度でも文字列全体を1つのブロックとした場合と同様の圧縮率になる。

予測木の形状は、全体の圧縮率が最小になるようにgreedyに決める。具体的な構築手順は以下の通りである。はじめに T を逆順にした T' に対するSuffix Treeを構築する。また、予測木を空に初期化する。次に、根から順番に各節点を、それを予測木に含めることで圧縮率が向上するかどうか(節点を含めたことによって予測木のサイズが増えた分と予測率が向上したことによって T の圧縮後のサイズが小さくなった分を比較する)を調べていき、圧縮率が向上する場合はその節点を予測木に含める。実際には、全てのSuffix Treeは前もって構築せず根から必要な部分だけを順に構築していく。木の形状が決定した後、予測木はBalanced Parenthesis Tree (BPT)[3]で保持される。BPTは節点数 M 、葉数 N の木を $2(M+N)$ bitで保持し、各節点の親、子、兄弟である節点、葉を定数時間で求めることが可能である。

表1 VC, γ 符号を用いたCSAの比較

	Ψ (ns)	作業領域量 (bpc)
VC	0.159	7.0
γ 符号	1.70	6.0

表2 各圧縮手法の比較。部分復元はランダムな位置からの1kB復元操作100回の平均

手法	圧縮時間 (s)	部分復元 (ms/kB)	圧縮率 (bpc)
SP	134.4	0.317	1.723
CSA	290.7	0.626	4.391
gzip-9	12.9	-	2.604
bzip-9	52.2	-	1.754

5 実験結果

Canterbury Corpus中のE.coli(4530KB)に対しCSAを構築した時、 Ψ を保存するのにVCを利用した場合と γ 符号を利用した場合の Ψ の操作速度、使用した作業領域量の結果を表1に示す。VCと γ 符号の作業領域量がほぼ同じながらVCの方が約10倍高速である。

また、表2にXMarkを用いて作成されたXMLファイル(111MB)を圧縮した時のSPと各圧縮手法の比較を示す。SPがbzip並の高い圧縮率を実現しながら、高速な部分復元を実現している。

6 まとめ

本稿では大規模なデータを扱うことを可能とする高速な圧縮索引と部分復元可能なデータ圧縮を提案した。これらの技術を用いることで、従来難しかった大規模なデータを利用したアプリケーションの実行が可能となる。

今後は木やグラフ情報といった1次元のテキスト情報以外のデータも効率的に扱う圧縮法の開発が望まれる。

参考文献

- [1] P. Ferragina, and G. Manzini. Opportunistic data structures with applications. In *Proc. of FOCS*, 2000.
- [2] P. Ferragina and G. Manzini. Indexing compressed texts. *J. ACM*, 52:552–581, 2005.
- [3] R. Geary., N. Rahman., R. Raman., and V. Raman. A simple optimal representation for balanced parentheses. In *Proc. of CPM*, pages 159–172, 2004.
- [4] R. Grossi., A. Gupta., and J. Vitter. High-order entropy-compressed text indexes. In *Proc. of SODA*, pages 841–850, 2003.
- [5] R. Grossi. and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 2005.
- [6] A. Moffat and V. Anh. Binary codes for non-uniform sources. In *Proc. of DCC*, pages 133–142, 2005.
- [7] D. Okanohara. Partially decodable compression with static ppm. In *Proc. of DCC*, 2005.
- [8] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. Algorithms*, 48(2):294–313, 2003.
- [9] K. Sadakane and R. Grossi. Squeezing succinct data structures into entropy bounds. In *Proc. of SODA*, 2006.