平成 23 年度　国立情報学研究所　市民講座　第 5 回
「データを圧縮する―大量のデータを小さく収納するには？―」
講師：定兼　邦彦
（国立情報学研究所　情報学プリンシプル研究系　准教授）

◆　講　義　◆

こんばんは。
定兼（さだかね）と申します。
よろしくお願いします。
今日は「データを圧縮する」というタイトルでお話しします。
配付資料には途中で出すクイズの答えが書いてあるので、クイズを楽しみたい方はあまりご覧にならない方がいいかもしれません。

・スライド 2「データ圧縮とは」

堅い話になってしまいますが、データ圧縮とは何かということをまず言います。
データとは、数値の集まりと言うことができます。
1、2.5、100、3.14 というように、どんなものでも数値で表現できます。
ただ、数値を見ただけでは、それが何を表しているのかという解釈がいろいろあるので、数値だけを見ても特に意味はないのですが、それに意味が加わったものが情報となります。
例えば、データには文字や画像、音声や動画などいろいろあるのですが、数値に対して何らかの意味が付いたものが情報です。
コンピュータの中では、すべてのデータは 0,1 の列で表されます。
この 0,1 をビットと呼びます。
これだけではまだよく分からないと思いますが、後でもう少し例を出します。

次に、圧縮とは、データを表現する 0,1 列、ビット列の長さを短くすることです。
データを小さく、長さを短くすることを圧縮といい、その逆、圧縮されたデータから元のデータを求めることを復元、伸長、復号、解凍といいます。
復元はそのまま元に戻すことです。
伸長は、小さくしているのでそれを伸ばすということです。
復号は復元と同じような意味ですが、なぜ「号」を使うかというと、0,1 の列を符号というので、その符号を元に戻すということで、復号と言います。
また、なぜ解凍というか、不思議に思われるでしょうが、これは、現在使われている圧縮のプログラムを開発した人が北海道に住んでいる日本人だったので、圧縮することをフリーズ（凍らせる）と言って、それを元に戻すことを解凍と言ったので、「解凍」という言葉が使われるようになりました。
多分、この語源を知っている人は少ないのではないかと思います。
解凍するという言葉を普通に使いますが、その理由はそういうことです。
今日は圧縮の話をします。

・スライド3「2種類の圧縮法」

圧縮には2通りあります。
一つは可逆圧縮、もう一つは非可逆圧縮です。

可逆圧縮は、完全に元に戻る圧縮法のことです。
どういうときに使うかというと、文書やプログラムのような、完全に元に戻らないと意味を成さないときに使います。

非可逆圧縮の方は、完全には元に戻りません。
元に戻したときに、人間には区別できない程度の違いがあります。
例えば画像、音声、動画などは、パッと見ても分からない程度の違いは忘れて、その分、圧縮率を高くします。
よく使われるものを挙げると、JPEGは画像、mp3は音声や音楽、MPEGは動画などの圧縮形式です。
その場合はロスがあり、微妙に違うものが復元されます。

今日は、完全に元に戻る圧縮法・可逆圧縮だけを扱います。

・スライド4「本の検索」

圧縮に関連する話として、検索があります。
データ量が多いと、圧縮したいと思うわけです。
また、大量のデータから検索するときに、なるべく速く検索したいと思います。

人が検索するときには、例えば本の中身を検索する場合、普通は索引を使います。
索引は本の最後の方に数ページ載っていますが、キーワードごとに、それが何ページに載っているかということがずらりと書いてあります。
索引に載っていない単語は、当然、索引を使っても見つからないので、1 ページずつ見ていくしかありません。

計算機でも、検索を高速に行おうと思うと、同様のことをします。
索引を作り、計算機がそれを使って、速く検索します。
つまり、本の索引のようなものを計算機が使うわけです。
その索引のことを、データ構造といいます。
索引の見出しを増やせば、当然、データ量（索引のページ数）が増えますが、その代わり、いろいろなものが見つけられるようになります。
また、普通の本の索引だと、単に何ページに載っているという情報しかないのですが、もう少し細かく、何ページの何行目に載っているなどの情報を付け加えると、さらに検索が速くなります。
すべての単語を索引に載せると、本のページ数が索引の分だけ増えてしまい、倍以上になるという問題

があるので、これまでの検索手法ではこのような索引は使われていません。

・スライド5「索引の例」

私の研究でやっている例なのですが、NTCIR の PATENT というデータがあります。
これはNII で公開している、93年から97年の過去5年分の日本語の特許文書のデータで、サイズが113GB、文書数だと約 350 万件あります。
ピンとこないと思いますが、2 層分で映像が 4 時間ほど入る DVD だと、1 枚当たり 10GB 入るので、DVD10 枚分に当たります。
計算機のメモリはバイトで測りますが、せいぜい4GBなので、普通の計算機の約20倍のデータ量があり、非常に多いのです。

日本語の文書は結構冗長なので、圧縮すると小さくなり、この場合は約 7 分の 1 の 15GB ぐらいになります。
ただし、圧縮してしまうと検索がしにくいので、検索するためにデータ構造を付け加えるのですが、そうすると途端にサイズが大きくなってしまいます。
「接尾辞配列」が一つの索引の名前なのですが、これを使うとどんな単語も検索できる代わりに、データ量が非常に多くなってしまうという問題がありました。

私が研究している簡潔データ構造を使うと、圧縮前は 680GB あった索引と元の文書のデータの合計が 21GB ぐらいになり、約 30 分の 1 に圧縮できました。
私は研究で、こういうことをやっています。
ただ、あまり細かい話はできないので、今日は、データ圧縮はどのようにするのかという話を中心に説明します。

・スライド6「簡潔データ構造」

簡潔データ構造とは、データに索引を追加してもサイズが増えず、かつデータも圧縮できるものです。
なおかつ、圧縮されていない索引（データ構造）を用いた場合と、ほぼ同じ速度で検索できるようになっています。

・スライド7「データ圧縮の基本」

では、データ圧縮の基本を説明します。

・スライド8「次のデータは何を表しているでしょう？」

クイズです。
スライドのデータは何を表しているでしょうか。
これを見てすぐ分かる人はいないと思いますが、前回の市民講座に出た方は分かるかもしれません。
これがヒントです。
分かる方はいらっしゃいませんよね？

多分、普通は分かりません。

実はこれは、「国立情報学研究所」を表しています。
ビットの列を 8 ビットごとに区切ると 1 バイトになって、それを二つ組み合わせると一つの漢字を表しているということになるのです。
Shift-JIS コードという日本語の漢字コードで、例えば「8D91」の並びで「国」を表すということが決まっています。
それを見ると「国立情報学研究所」を表していることが分かるのですが、もちろんデータだけ見ても、その数字の並びが「国立情報学研究所」を表していることなど分からないわけです。
これが、データか情報かという違いなのです。

これを見ると、結構冗長のような気がします。
例えば 100 がよく出てくるなどということがあるので、これを何とか圧縮できないかと考えるわけです。

・スライド 9「データ圧縮法」
データ圧縮の一例は、「サクラサク」という電報の文章です。
長い文章を、「サクラサク」のような短いものに置き換えるわけです。
すべてのデータ圧縮は、長いものを短いものに置き換えることだと言えます。
「サクラサク」の例だと、5 文字も必要ありません。
合格か不合格を 1,0 で表せばいいので、本当は 1 ビットで表現できるのですが、そこまですると分かりにくいので、5 文字ぐらい使います。
しかし、送られてくる情報には合格か不合格の 1,0 という 2 通りしかないとあらかじめ分かっていれば、1 ビットで表現できるのですが、そうすると、その 2 通りの文章しか表現できなくなってしまいます。
そういうときに、特定の 2 通りだけでなくてどんな文章でも表現できるようにするにはどうすればいいかと考えます。

・スライド 10「モールス符号（信号）」
昔からある圧縮法の一つとして、モールス信号（モールス符号）があります。
これは 1830 年代に提案された、非常に古いものです。
短点（・）と長点（－）の組み合わせで文字を表しています。
例えば、A なら短点と長点（・－）、B なら長点と短点三つ（－・・・）というように、アルファベットとモールス符号の対応表を使って、文字を符号に変換するわけです。

モールス符号の場合は、長点は短点三つ分の長さとなっています。
SOS だと、S が「・・・」、O が「－－－」なので、このように「・・・－－－・・・」となります。
余談ですが、なぜ SOS というようになったかというと、モールス符号で打ちやすいためです。

アルファベットとモールス符号の対応表を見ると、点が一つだったり、二つだったり、三つだったり、

文字ごとに符号の長さが違うわけです。
なぜかというと、英語で使われる頻度の高い文字に対して短い符号が割り当てられているからです。
例えば英語で一番よく使われるＥに対しては、「･」だけで表現できます。
英語で２番目によく使われるＴは、「－」だけで表現されます。
このように、よく使われるものを短い符号で表現しているこれは、データ圧縮になっています。
例えばほとんど使われないＱ、Ｚ、Ｘ、Ｙなどは四つの符号で表現されています。
あまり使われないものに長い符号を割り当て、よく使われるＥやＴに短い符号を割り当てることで圧縮ができる形になっています。

・スライド 11「次の符号は何を表しているでしょう？」
クイズです。
答えを見ないでください。
「－・・－－・－－－－・・」は何を表しているでしょうか。
答えを求めるには対応表を使い、文字を復元します。
最初の四つ「－・・－」がＸ、次の四つ「－・－－」がＹ、次の四つ「－－・・」がＺを表しているので、これはＸＹＺだろうと思うのですが、同時に「ＤＧＯＤ」を表現しているかもしれないのです。
Ｄは「－・・」、Ｇは「－－・」、Ｏは「－－－」なので、２通り、もしくは何通りもの解釈ができてしまいます。
そうすると、「－・・－－・－－－－・・」を見た人は、元の文字が何だったか、分からなくなってしまいます。

従って、一通りではないことから、こういうものを「一意に復号可能」ではないといいます。
その原因は、一つの文字を表す符号がどこで切れているのか分からないことです。
切れ目が４文字目かもしれないし、３文字目かもしれないと、何通りにも切り方があって、何通りも解釈できてしまうため、これでは符号としては使えないわけです。
そこで、モールス符号ではどうしているかというと、文字の間には短点三つ分の空きを入れています。
例えば、Ｘの符号「－・・－」を打って、短点三つ分の休みを入れ、次の文字の符号を送るようにすることで、符号の区切りが分かるようになっています。

・スライド 12「一意に復元可能な符号」
一意に復元可能な符号には、一体どういう性質があるのかを見ていきます。
一意に復元可能な符号とは、符号を先頭（左）から見ていったときに、１通りにしか復元されない符号のことです。
例えばモールス符号では、一意ではないので空白を入れていました。

一意ではないということを、tree（木）と呼ばれる図で表すことができます。
この図を上下ひっくり返すと、根っこから枝が伸びているように見えるからです。
この図が、実は符号を表しています。

一番上の点から左に下りていくものは短点「・」を表していて、右に下りていくものは長点「ー」を表しています。
一番上の点から左に行くとEと書いてあります。
つまり、短点一つ分がEという文字を表していることを示しています。
右に下りていくとTが書いてあり、長点一つがTを表していることを意味します。
同様に、一番上の点から、左、左といくと、「・・」でIと書いてあります。
つまり、アルファベットとモールス符号の対応表を表しているのが、treeの図なのです。

先ほど、一意ではないと言いましたが、例えば一番上の点から右、左、左（ー・・）と行くとDが書いてありますが、そこからさらに右下に行くとXと書いてあります。
つまり、一番上の点から下りていったとき、途中に文字が書いてあると、これが一意ではないということを表しています。
ですから、一意ではない符号は、木の途中に文字が割り当てられているということになっています。

・スライド13「コンピュータで表現するには？」
モールス符号をコンピュータで表現することを考えます。
モールス符号には短点と長点と空白の3種類が必要でした。
しかし、コンピュータでは0と1しか使えないので、0と1のビットで表現する必要があります。
例えば、短点を0で表現して、長点を11で表現、空白を10で表現してみると、X、Y、Zの間に空白を入れているものは、11001110110111110111100と表現されます。
これは一つの例なので、別にこれに限ったことではありませんが、こんな感じで表現できます。

・スライド14「各文字の符号の最後に、空白を表す10をつけると一意に復号可能になる」
各文字の符号の最後に空白を表す10を付け、先ほどの木の上で表現してみます。
例えばEはもともと短点一つだったのですが、それを、短点を表す0と空白を表す10で、010で表現します。
木の一番上の点から、010と、左、右、左とたどっていったところにEを書きます。
つまり、010がEを表しています。
同様に、短点二つ「・・」のIは0010で表現されます。
木の一番上の点から、0010と、左、左、右、左とたどっていった点が、Iの符号を表しているという感じになります。
Aは01110なので、左、右、右、右、左と行ったところにAと書いてあります。

先ほどの木の絵と違うのは、木の内部の点に文字が書いていないことです。
先ほどは木の内部の点に文字が書かれ、同じ「・」と「ー」で別の文字を表していたりしましたが、文字の最後に空白を表す符号（10）を付け加えると、この木の内部には文字が割り当てられず、必ず、これ以上先がない末端部分に文字が割り当てられるようになります。
そうすると一意に復号可能になります。

従って、この木は一意に復号可能な符号を表現しています。

・スライド 15「どのような符号が良いか」

そこで、どういう符号を使えばいいかを考えます。
例えば ABCABCAABAACAABC という文字列を圧縮することを考えます。
符号 1 として、A を 00、B を 01、C を 10 で表現すると考えます。
そうすると、最初の A が 00 になって、次の B が 01 になり、00011000011000000100001000000110 というビット列が得られます。
その長さは、文字列の中に、A は 8 個、B が 4 個、C が 4 個出てきて、A は 2 ビットで表現するので、2 ビット×8 個。
B は 2 ビット×4 個なので、2×4。
C は 2 ビット×4 個なので 2×4 となって、合計で 32 ビットになります。
つまり、この 0,1 列の長さが 32 あるわけです。

符号 2 では、A を 00、B を 1、C を 01 と符号化します。
そうすると、先ほどとは 0,1 の列が変わってきて、B が 01 だったのが 1 になっています。
このときの符号の長さを計算すると、A が 2 ビット×8、B が 1 ビット×4 個、C が 2 ビット×4 個で、合計 28 ビットとなって、先ほどより少し短くなり、圧縮できました。

さらに別の符号 3 を考えます。
A を 0、B を 10、C を 11 で表現すると、そのときの長さは、1×8+2×4+2×4 で 24 ビットになって、さらに圧縮できました。
このように、符号を変えると圧縮率が変わります。

ちなみに、ここにある符号はすべて一意に復元可能な符号になっています。
例えば、符号 1 では、A が 00、B が 01、C が 10 となっていて、木の内部には文字が割り当てられていません。
符号 2 と符号 3 でも、同様に内部には文字が割り当てられていない。
これらは一意に復元可能ですが、符号によって圧縮率が変わります。

・スライド 16「符号 1,2,3 の中で、符号 3 が一番小さくなっている」

なぜ符号 3 の圧縮率が一番よかったか、一番小さくなっているのかというと、一番多く現れる文字 A の符号が短いからです。
つまり、なるべくよく現れる文字に短い符号を割り当てる方が、圧縮できるわけです。

どういう符号を使うと一番圧縮ができるかを考えます。
今、符号が 3 通りありましたが、もっといい圧縮、符号がないかということを考えます。

A、B、C の符号の長さをそれぞれ x、y、z とします。
すると、先ほどの ABCABCAABAACAABC という文字列を表現する符号の長さは、「L＝8x＋4y＋4z」となります。
8x の 8 が A の数、4y の 4 が B の数、4z の 4 が C の数です。

今は、一意に復号できる符号のみを考えないと意味がないので、そういうものだけ考えます。
一意に復号できる符号のみ考える場合は、x と y と z の符号の長さは、「$1/2^x+1/2^y+1/2^z \leqq 1$」という条件を満たす必要があります。
これはクラフトの不等式というのですが、これがなぜそうなるのかという話は置いておいて、こういう条件を付けて、符号の長さ L が一番短くなるようにと考えると、x＝1、y＝2、z＝2 が解になります。
つまり、先ほどの符号 3 が最適であると言っているのです。

・スライド 17「エントロピー」
少し難しい話が出てきますが、エントロピーというものがあります。
エントロピーは統計力学や熱力学などで出てくる話です。
情報の方で出てくるエントロピーも似ているのですが、特に物理的な意味はないと思います。
この定義は、文字列中の文字 c の出現確率を p(c) と表現すると、文字列のエントロピーは、$p(c) \times \log_2 (1/p(c))$ をすべての c に対して足したものと定義されます。
c が A に入っている（c∈A）と書いていますが、この A とは文字の集合で、例えば A から Z まですべての文字を表現しています。

このエントロピーを先ほどの例で計算すると、文字 A の出現確率は、文字 A が 8 個だったので、8／16 で 1／2。
文字 B と C は 4 回出てくるので、4／16 で、1／4 の確率になります。
これを代入して計算すると、先ほどの文字列（ABCABCAABAACAABC）のエントロピーは 3／2 となります。
このエントロピーは、符号化するのに 1 文字当たり何ビット必要かを表しています。

・スライド 18「どんな符号でも、1 文字あたりの平均ビット数はエントロピーよりも小さくならない」
先ほど何種類か符号を見せましたが、定理として、どんな符号でも 1 文字当たりの平均ビット数はエントロピーよりも小さくならないという結果があります。

符号 3 の 1 文字当たりの平均ビット数を計算すると、符号 3 では 16 個の文字を 24 ビットで符号化していたので、3／2 となって、1 文字当たり 3／2 ビット（1.5 ビット）で表現されています。
実はこの値は、今計算したエントロピーと一致しています。
つまり、符号 3 は最適な符号だと言えます。

エントロピーの式をよく見ると、$1/2 \times \log_2 2/1$、log の底は 2 になっています。
これを計算すると、1／2×1＋1／4×2＋1／4×2＝2／3 となるのですが、この式の最初に出てくる 1／2

がＡの出現確率で、それに掛かる１がＡの符号の長さとなっており、これが１文字当たりの平均の符号長を表しています。

・スライド 19「ハフマン符号（1952 年)」

その平均の符号長が一番短い符号があります。
それがハフマン符号というもので、大体 60 年前に提案されました。
これは、平均符号長が最小の符号であることが証明されており、最も圧縮できる符号ということになります。
実は、符号３はハフマン符号なのです。

作り方について、先ほどは方程式を解いて求めていましたが、実はそんな難しいことをしなくても作れます。
文字列が与えられたら、まず、その中に出てくる各文字の頻度を数えます。
そして、出現頻度が最少の文字（一番出てこない文字）と、２番目に少ない文字に対して、０と１を割り当てます。
先ほどの例だと、Ａが８回、Ｂが４回、Ｃは４回だったので、一番出てこないＢとＣに０と１を割り当てます。
木の一番上から左に行ったところにＢ、右に行ったところにＣを書きます。
つまり、Ｂは０、Ｃは１という符号で表されています。

次に、ＢとＣを合わせて一つの文字にします。
どういうことかというと、ＢとＣを合わせて文字Ｄにするのです。
つまり文字Ｄは、ＢまたはＣであることを表しています。
ＢとＣがそれぞれ４回なので、文字Ｄは８回出てくると見なします。
それを、文字が１種類になるまで繰り返します。

・スライド 20「Ａ：８回、Ｄ：８回なので、Ａに０、Ｄに１を割り当てる」

次に、今、ＢとＣが一つになってＤが８個出てくることになっています。
すると、ＡもＤも８回出てきていることになるので、それぞれに０と１を割り当て、Ａを０、Ｄを１とします。

次に、ＡとＤを合わせて文字Ｅだけにすると、それが 16 回出てきたことになります。
そうすると文字が一つになったので、これで終わりです。

これで符号がどうなっているかを見ます。
Ａが０、Ｄが１だったので、二つに枝分かれしている木の図が描けます。
実は、ＤはＢまたはＣで、Ｂは０、Ｃは１だったので、ＢとＣに関する木を付け加えると、Ｄからさらに枝回れした木ができます。

すると、A が 0、B が 10、C が 11 という符号になって、これが先ほどの符号 3 になっています。
このように作った符号は、平均符号長 L が、H≦L＜H＋1 になっているという性質があります。
つまりハフマン符号は、一意に復元可能な符号の中で最適なものなのです。

・スライド 21「データ圧縮の欠点は？」
次に、圧縮の欠点は何かということを考えます。
パソコンで普通に圧縮を使っている方なら分かると思いますが、一つの欠点は復元が遅いということです。
圧縮されているのを元に戻すのに時間がかかります。

さらに、一部分だけ復元することが難しいという問題があります。
つまり、部分的に復元したいのだけれども、全体を復元するしかないのです。

なぜそうなるかというと、例えば ABCABCAABAACAABC という文字列が圧縮されていたとして、その中の 10 文字目だけを知りたいので、それだけを復元しようと考えます。
A が 00、B が 01、C が 10 という符号 1 の場合は簡単です。
なぜかというと、すべての符号が 2 ビットなので、10 文字目は必ず 20 ビット目にあるからです。
20 ビット目がどこかを人間が見ようと思ったら 1 個ずつ数えなければいけませんが、もしマス目が付いていて 20 ビット目がどこかすぐに分かるとすると、20 ビット目を見て、そこに 00 と書いてあれば、10 文字目が A であることが分かるわけです。
このように、符号 1 の場合は部分的に復元することが簡単です。

では、A が 0、B が 10、C が 11 の符号 3 の場合はどうでしょうか。
この場合、10 文字目の文字を表している部分がどこかということが、実はよく分からないのです。
もちろん先頭から一つずつ見ていけばいいのですが、何ビット目を見ればいいか分からないので、部分的に復元することが難しいということがあります。

・スライド 22「符号 1 のように全ての文字が同じビット数の符号を固定長の符号と呼ぶ」
符号 1 はすべての文字が同じビット数でしたが、そういうものを固定長の符号と呼びます。
文字のアルファベットサイズを σ とすると、σ の log を取ったビット数で表現されます。
この log とは、log の底が 2 の場合で、計算機でよく使う文字の場合は、アルファベットサイズは 256 になっており、つまり一つの文字が 8 ビットで表現されています。
コンピュータで扱うデータは、普通は固定長の符号で表現されているので、何文字目かを取ってくるのは簡単なのですが、これは全く圧縮されていない状態です。

圧縮しようと思うと、どうしても可変長の符号を使う必要があります。
文字によって符号の長さが違うものを可変長といいますが、そうしないと圧縮できないわけです。
よく現れる文字に短い符号を割り当てて、あまり現れない文字に長い符号を割り当てるしか、圧縮する

方法はありません。
すると、必ず可変長の符号になってしまい、圧縮はできるのですが復元が遅くなります。
なぜかというと、先頭から 1 文字ずつ復元していかないと 10 文字目が何ビット目かということが分からないからです。
これが、圧縮によって復号が遅くなる理由です。

・スライド 23「部分復号の高速化」
では、これを何とか速くしたいと考えます。
つまり、部分復号を高速化することを考えます。
今は、ABCABCAABAACAABC という文字列の 10 文字目の復元を考えています。
先頭から見ていったら遅いので、速くしようと思ったら、途中から復元できるようにすればいいのです。
そのために、符号の開始位置を記憶しておきます。
例えば、一つずつ復元していって、1 文字目は 1 ビット目から、6 文字目は 9 ビット目から、11 文字目は 16 ビット目から、16 文字目が 23 ビット目から始まっています。
このことは、一度復元すれば分かるので、この 1、9、16、23 という数字を覚えておきます。

そうすると、10 文字目を復元しようと思ったら、6 文字目が 9 ビット目だということが分かっているので、そこから 6 文字目、7 文字目、8 文字目、9 文字目、10 文字目と復元していけばいいのです。
どの文字を復元する場合でも、最大 5 文字だけ復元すれば大丈夫だと言えます。
11 文字目を復元するときは、その場所が 16 ビット目だと分かっているので、すぐそこから取ればいいし、12 文字目だったら、11 文字目と 12 文字目だけ復元すればいいということになります。
どんなに文字列が長くても、最大 5 文字復元すれば、特定の文字が復元できます。

これで高速にはなるのですが、問題は、開始位置を覚えておかなければいけないので、それを表現するビットも必要なわけです。
そこで、開始位置（1、9、16、23）は何ビットで表現できるかということを考えます。

・スライド 24「開始位置の記憶」
文字列の長さを n とします。
先ほどの例では、n=16 です。
アルファベットサイズは、n より小さいとします。
文字列を圧縮した後のサイズ、ビット数を m とすると、$m \leqq n \times \log \sigma$ になります。
$n \times \log \sigma$ は、圧縮されていない状態のサイズです。
今、σ は n 以下と仮定しているので、$n \log \sigma$ は $n \log n$ よりも小さいということになります。

開始位置を d 文字置きに記憶したとします。
先ほどの例では d=5 です。
そうすると、n／d 個の開始位置を記憶することになります。

先ほどは5個置きだったので、16／5個、つまり3個か4個の開始位置を記憶するわけです。

一つの開始位置は何ビットで表現できるかを考えると、開始位置は、1 から m の整数なので、対数を取って log m ビットで表現できます。

・スライド25「全ての開始位置を記憶するには」

これを計算すると、すべての開始位置を記憶するには、(n log m) ／d ビット必要です。
式変形をすると、大体 (2 n log n) ／d ビット必要だということになります。

今、d=2 log n とすると、(2 n log n) ／d の値は n より小さくなるので、開始位置は n ビット以下で表現できます。
今、n 文字あるので、1 文字当たり 1 ビット以下で開始位置を表現できます。
つまり、開始位置を格納するためのビット列のサイズはそれほど大きくはならないことが分かります。

一つの文字を復号するために必要な時間は d に比例します。
つまり、d 文字を復元しないと一つの文字が復元できないので、必要な時間は d に比例するのです。
従って、d を大きくすると復元が遅くなるのですが、圧縮率がよくなる。
つまり、d を大きくすると、開始位置を記憶するのに必要なビット数を求める式の分母が大きくなるので、サイズが小さくなるのです。
その代わり、復元するのに必要な時間が増えるというトレードオフがあります。

・スライド26「開始位置の圧縮」

今の表現だと、開始位置を格納するときに、d を小さくすると復元は早くなるのですが、サイズが大きくなってしまう問題があります。
そこで、この開始位置を圧縮することを考えます。

先ほどは数字で開始位置を記憶していましたが、それをビット列で表現します。
つまり、ABCABCAABAACAABC という文字列を圧縮した文字列と同じ長さのビット列を使って、5 文字置きの開始位置に 1 を立てます。
つまり、10000001000000100000010 というビット列の 1 が立っているところを見ると、最初の 1 が 1 文字目、次の 1 が 6 文字目、次の 1 が 11 文字目、次の 1 が 16 文字目であることを表しています。
つまり、この 10000・・・というビット列があれば、開始位置を表現できているわけです。

・スライド27「簡潔データ構造」

ここで「簡潔データ構造」が出てきます。

・スライド28「ビットベクトル」

この簡潔データ構造では、開始位置を表しているような 10 の列を何とか圧縮しています。

01 のベクトル（ビットベクトル）に対して、rank と select という操作を考えます。
select（B, i）とは、B（10000001000000100000010）の先頭から i 番目の 1 の場所を返す操作です。
つまり select（B, 1）とやると、1 が返ってきて、先ほどの 1 文字目の開始位置を表しています。
select（B, 2）とやると 9 が返ってくるのですが、それは二つ目の 1 の場所で、6 文字目の開始位置になっているという関係があります。
従って、6 文字目の開始位置を求めようと思ったら、select（B, 2）の操作を行えば、9 ビット目であることが分かります。
以上のように、ビットベクトルに対して rank と select という操作ができると、開始位置が表現できます。
今は rank は使っていませんが。

・スライド 29「Select の求め方」
select を求めるときに、（前のスライドの）B というビット列（10000001000000100000010）を先頭から見ていくと非常に時間がかかってしまうので、それをやってはいけません。
それを速く求めるために、次のようなデータ構造を使います。

・スライド 30「大ブロックの長さ m が $\log^c n$ 以下のとき」
B のビット列に対して、木を作ります。
一番下の左端の箱の中に書いてある 1 は、ビット列の先頭から 3 ビットの中に 1 が何個あるかを表しています。
最初の 3 ビットは 100 なので 1 が 1 個、次は 101 なので 2 個、次が 000 なので 0 個という感じです。

次にその上の一番左に 3 とあるのは、先ほどの箱の 1・2・0 の合計です。
つまり、ビット列の先頭（1 ビット目）から 9 ビット目の中に 1 が 3 個あることを表しています。

このデータ構造を使って、例えば 5 番目の 1 の場所を求めることを考えます。
この場合、この木の一番上から下りていきます。
まず、一番左に行くと、3 と書いてあるので、その下には 1 が 3 個しかないことを表しています。
今、探しているのは 5 番目の 1 は、この 3 の下にはないということが分かります。

次にその右の 4 のところを見ます。
4 と書いているということは、この下に 1 が 4 個あることを表しています。
それより左側には 3 個あったので、探している 5 番目の 1 は、この 4 の下にあることが分かります。
というわけで、この 4 から下がっていきます。

次に、4 の一番左の子供を見ると 1 と書いてあるので、先ほどの 3 と合わせて、4 番目の 1 がここにあることが分かります。

4 の真ん中の子供を見ると 1 と書いてあります。
これは先ほどの 3 と 1 と 1 を足すと、5 番目の 1 がここにあることが分かって、つまりここに 5 番目の 1 があることが分かります。

以上のように、5 番目の 1 を探すときに、ビット列を先頭から見ていくのではなくて、木を上から下に下りていくだけの探索で、5 番目の 1 の場所が分かるという操作になります。
この例は小さくてよく分かりませんが、以上のようにすると、このビット列の先頭から 1 ビットずつ見ていくよりも速いスピードで求めることができます。

・スライド 31「B は約 n ビット（n＋o（n））で表現でき、rank と select は答えを圧縮しないで記憶した場合と同じ時間（定数時間）で計算できる」
どれぐらい速いかというと、select という操作が定数時間、要するに一瞬で求まるということになります。
これは圧縮しないで答えを記憶しておいた場合と同じ時間で求まるということです。
このように、データは圧縮してあるのですが、圧縮していない場合と同じ検索速度のものを簡潔データ構造といいます。

・スライド 32「順序木の簡潔データ構造」
いろいろなデータ構造に対して、簡潔データ構造、圧縮されたデータ構造があります。

・スライド 33「圧縮全文索引ライブラリ」
例えば、私が作って公開している圧縮全文索引ライブラリ（csalib）は、文字列の検索をするためのデータ構造です。
特許文書の 113GB の中から、特定のパターンが何回出てきているとか、よく出てくるパターンがどういうものかということをすぐ求めることができます。
ある程度以上の回数で出てくるパターンを列挙すると、「図である」「することを特徴とする」「することができる」がよく出てくることが、このプログラムを作ると分かります。
特許の文書なので、「特徴とする」などの言葉がよく出てきていることが分かるなど、面白いことができます。

私の研究の話はあまりできませんでしたが、興味のある方は私のホームページを見ていただくと、いろいろ資料があります。
また、プログラムもそこに置いてあるので、使ってみてください。
データ圧縮の話は以上です。
ありがとうございました（拍手）。