Annales UMCS Informatica AI 1 (2003) 61-72

Annales UMCS
Informatica
Lublin-Polonia
Sectio AI

http://www.annales.umcs.lublin.pl/

Efficient modification of LZSS compression algorithm

Paweł Pylak*

Faculty of Mathematics and Science, Catholic University of Lublin,
Al. Racławickie 14, 20-950 Lublin, Poland

Abstract

This paper presents a new method of lossless data compression called LZPP, being an advanced modification of the well-known algorithm LZSS [1]. It introduces improvements of the LZ family algorithms [2, 3], such as the use of a special coding of two and three byte matches, use of an auxiliary entropy coder and new criteria of symbol exclusions. Minimization of the data compression ratio (bpc) has been chosen as the primary goal of the proposed modifications of LZSS algorithm.

1. Analysis of LZSS algorithm and proposed modifications

1.1. Index coding

The first thing which should be considered, when one wants to improve the quality of compression, are indexes generated by every LZ method.

Indexes can be coded with one of the statistical adaptive compression methods, like the arithmetic or the Huffman coding. Confirmation of this is the sample probability distribution of indexes (calculated as the distance from the end of the window) shown in Figure 1, which LZSS method generates during compression of the "xplik" file (merged all files from Calgary Corpus).

Graphs for the most of the files for which experiments were conducted had similar shapes. Therefore it can be assumed that generally indexes have such distribution. Moreover, symbols having such distribution should be compressed well with a statistical coder, because the entropy (see [4]) equals H(S) = 8,954 bits in this case. This means that the maximal static entropy coding would reduce the number of bits necessary to store indexes even by 44%.

Using this fact in the LZPP method, the fast variant of arithmetic coding (RangeCoder [5]) was applied for index coding. The range coder was additionally equipped with a rotating buffer, handling of the escape marker and the exclusions mechanism. Additionally, an independent coding of index bytes

* *E-mail address:* ppprezes@wp.pl

was used for the index processing. This enabled the use of a very large window (even up to 16MB).

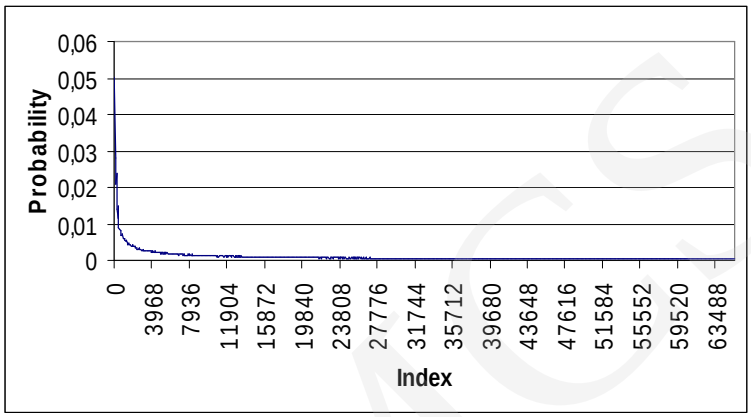

Fig. 1. Probability distribution of indexes generated by LZSS method during compression of file “xplik”

1.2. Lengths of matches coding

While looking at Figure 2 showing the numbers of the matches occurrences for lengths from 3 to 20 in the file “obj”, one can easily state, that it is worth to code lengths using an entropy coding such as the Huffman or the arithmetic coding. It can reduce the average number of bits necessary for storing lengths even by 75%.

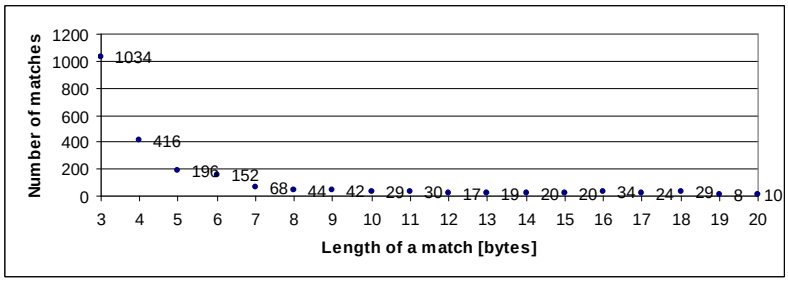

Fig. 2. Distribution of lengths of matches (3-20) in file "obj1"

In LZPP method, similarly to the case of indexes, *RangeCoder* was used to code lengths of matches, also with an independent coding of bytes. This allowed to use easily lengths up to 64KB.

1.3. Flag coding

Flags used for controlling the decompression process can be also specially coded to remove redundancy.

In LZSS method the flag responsible for distinguishing between coding of symbols (0) and substrings (1) usually has probability distribution different from uniform ($p(0) \neq 1/2$ and $p(1) \neq 1/2$). An example of changes of the probability distribution of that flag are shown in Figure 3.

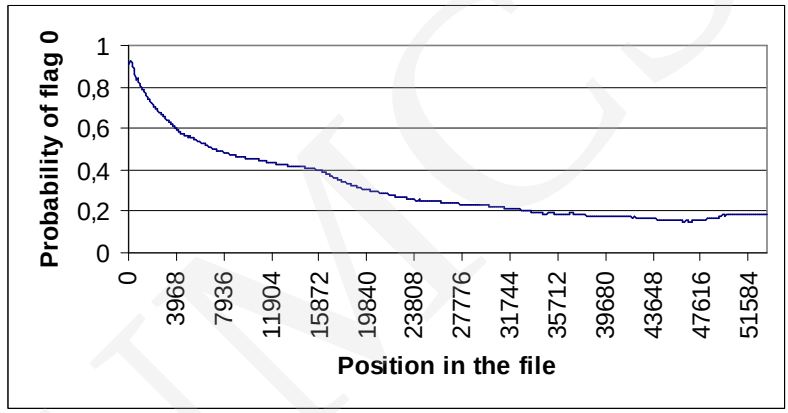

Fig. 3. Changes of probability distribution of symbol coding flag for file "paper1"

It can be easily seen, that the amount of redundant information in this case is considerable. Further studies have shown that the use of *RangeCoder* can improve the compression quality by up to 2%.

1.4. Non-compressed symbols

The following possible improvement is coding (with any entropy coder) all the symbols, which were not compressed by LZSS and were sent to the output unchanged. Using the Huffman coding one can reduce, by approximately half, the number of bits used to store these symbols.

In LZPP method *RangeCoder* was used for the compression of these symbols. Certainly, it has better properties than Huffman coder.

1.5. Special coding of three-byte sequences

Another redundancy present in LZSS algorithm is caused by not considering the frequency of occurrences of particular sequences. In other words, when in the last *n* symbols (there *n* is the size of the window) the string ‘abc’ was present 5 times and the string ‘def’ only once, it is natural to expect, that in the data to come the string ‘abc’ will appear more frequently than ‘def’. Standard LZSS

does not consider these probabilities and returns only the index of a given string. Observe, that even if an entropy coding is used for indexes (as described in section 1.1), the coder does not take into account the content of a match, but only its index. For such a coder all occurrences of the string 'abc' are distinct. Moreover, if the string 'def' occurred later than the last 'abc', the probability of 'def' is greater for such coder, because it has smaller index (see Figure 1 in section 1.1).

Although the above problem concerns sequences of any length, due to the implementation issues and the compression speed, it is reasonable to consider it only for short ones. In LZPP, a special mechanism for coding three-byte sequences was implemented. Three bytes were selected, because such sequences are statistically most often found in input data.

For this purpose, a special structure was created. It holds information about a fixed number of the last three-byte sequences together with the number of occurrences of each one. So, in the case of the three-byte sequence, LZPP generates a specially coded (entropy) tuple *(2,index_3)* in place of the usual triple *(1,length,index)*. There *index_3* is the index of the three-byte sequence in the aforementioned data structure. The occurrence counter of that sequence is used to code its index.

1.6. Symbol exclusions based on the substring criterion

The next step in the elimination of redundancy in LZSS is based on the observation, that after coding two symbols, additional information about the next symbol can be obtained. This symbol cannot be any of the symbols, which, with preceding two, would create one of the three-byte sequences. It is not necessary to consider sequences longer than three, because three-byte sequences are subsequences of the longer ones.

The *exclusions* mechanism of LZPP uses this fact.

1.7. Symbol exclusions based on the criterion of sequence continuity

Another property of LZSS, which can be exploited for optimisation, is the fact that the symbol following the just-coded match in the window cannot appear as the next symbol in the incoming data. Of course, the case when the given match reached the maximal length is an exception from that rule.

Example:

The content of the window is: *xxxabcdyyxxz*. Let the next data be: *abcdxx*. Of course LZPP will encode the symbols *abcd* as a 4-element match. We know that next symbol certainly will not be *y*, because if it were *y*, the encoded match would have length 5.

Similarly to the previously described optimisation, the exclusions mechanism was used.

1.8. Restrictions on the indexes of matches of length 4 and 5

It appears, that when processed matches are short (4 or 5 bytes), the use of large indexes is inefficient. Thus in LZSS algorithm indexes of 4 byte sequences are restricted to 255 and 5 byte to 65535. It allowed encoding of only one least significant byte in the first case and two least significant bytes in the second. If given match does not satisfy a proper criterion (e.g. it is four byte long and has an index greater than 255) it is ignored by the algorithm.

1.9. Contexts of order 1

The already described operations on sequences considered only sequences longer than two bytes. In practice, some relations between the pairs of bytes can also be used for a better compression. Following that observation, a mechanism of contexts of order one, similar to that used in PPM [6], was implemented in LZPP. Also here the context of order one of given symbol (byte) means one byte that precedes the considered one.

This optimisation was conducted in the following way. An array indexed by all contexts was created. It contains objects storing information about the distributions of symbols within the confines of the given context. A new value of flags (3) was added. It denotes encoding of an element in the context of order one. Encoding of one symbol used so far, when no matching substring was found, was preceded by verification of the possibility of encoding given symbol in a context of order 1. This verification is based on checking if the analysed symbol has a nonzero probability of appearing in that context. If it is so, flag 3 is sent and the current symbol is encoded using the distribution specific for the given context. Otherwise, LZPP encodes the current symbol in the standard way, sending flag 0 and the symbol itself. Finally, the probability of a given symbol in the current context of order 1 is updated.

2. Detailed description of LZPP algorithm

2.1. Basic notions and constants

“Escape” flag

A few arithmetic coders used in LZPP were enriched with the mechanism of the “*escape*” flag. The input alphabet is extended with an additional symbol “escape”. This symbol is used for coding symbols whose main frequency counters are equal to zero. Note that the value of the “escape” flag’s counter is always greater than zero. When processing a symbol with the frequency counter equal to zero, the “escape” flag is coded first. After that, the symbol itself is coded, also by the arithmetic coder. Before the last coding all the symbols with frequency counters different from zero are removed from the alphabet (the mechanism of exclusions). Only the symbols with main frequency counters

equal to zero are considered. Therefore an additional counter is held for each symbol. It stores the number of codings of a given symbol by the „escape" flag, increased by one. Only last 4096 symbols encoded by the „escape" flag are used to determine the value of this counter.

Settings of coders responsible for coding of indexes, lengths, flags, symbols and three-byte sequences

In the LZPP method, individual bytes of indexes (2 complete and 5 bits of the third – 21 bits altogether) are coded independently. Because of that, also the tables of the symbol counters for individual bytes of indexes are stored independently. Each of these counters is computed according to the formula $C_{b,w} := 2 + F_{b,w}$, where $F_{b,w}$ is the number of occurrences of a given value w in the corresponding byte b of index among last 4096 indexes encoded. Escape flags are not used for index encoding.

Lengths of substrings are coded as two independent bytes too. Counters for all values of both bytes depend only on the last 4096 lengths encoded. Each counter is equal to the number of the occurrences of individual value of a given byte. Escape flags are used for the length encoding.

In LZPP, four different flags are used. The alphabet of the coder used for flag coding consists of four symbols – numbers 0, 1, 2 and 3. The value of each of these symbols can be expressed as $C_f := 1 + F_f$, where F_f is the number of the occurrences of a given flag f among the 256 last encoded flags. Escape flags are not used for the flag encoding.

The symbol coding in a context of order 0 is implemented by the coder, which takes into account only the 1024 last encoded symbols in that context to determine values of counters. Each of the symbol frequency counters is equal to the number of occurrences of the adequate symbol increased by one.

While coding symbols in a context of order 1, only the last 256 symbols appearing in a given context, encoded in that context or in the context of order 0, are taken into account. Here each frequency counter is equal to the number of occurrences of the adequate symbol in a given context. In both cases, escape flags are not used for encoding.

In coding of three-byte sequences, the values of frequency counters of individual sequences, necessary for arithmetic coding, are computed from the last `HT_LEN` = 512 compressed data bytes.

Buffers and hash table

In the LZPP method, similarly to LZSS, one buffer consisting of a dictionary buffer and a coding buffer is used. The size of the dictionary buffer is set to `BUFF_SIZE` = 2MB, while the size of coding buffer is arbitrary, but not less

than 65539 bytes (that value is equal to the maximal length of a substring of `MAX_DL` = 65539 bytes).

In the compression algorithm, for finding substrings of four (= `MIN_DL`) and more bytes, a hash table of 65536 elements is used. This table consists of unsorted lists of absolute indexes of sequences.

The function

$$f_{CRC32}(x_0,x_1,x_2,x_3) = \mathrm{CRC32}(x_0,x_1,x_2,x_3) \bmod 65536$$

is used for computing the hash of a substring. Here x_0, x_1, x_2, x_3 are the first four bytes of a given substring. Usually, each four-byte substring from the dictionary buffer has its entry in the hash table. The only exception is the case when one of the index lists is overflowed. This happens when it already contains `MAX_ITEMS` = 2048 elements. In this case, before adding a new index to such a list, the oldest index is removed.

2.2. Algorithm outline

Because of the volume limitations, we present here only the compression algorithm of LZPP.

1. Initialization:
   1.1. Setting of the main constants: `MIN_DL` = 4, `MAX_DL` = 65539, `BUFF_SIZE` = 2^{21}, `MAX_ITEMS` = 2048, `HT_LEN` = 512,
   1.2. Initialization of all necessary coders, with details presented in the previous section,
   1.3. Initialization of other necessary variables and structures, including the hash table and the variable `w` describing the position of the byte being compressed relative to the beginning of data,
2. While there are bytes in the coding buffer:
   2.1. Find the longest substring having length in the range [`MIN_DL`; `MAX_DL`] and equal to substring at the beginning of the coding buffer – like in the standard LZSS.
      2.1.1. If such substring is found, denote by `maxn` its absolute position in the input data, by `maxl` its length, and by `dn` the distance from the beginning of the substring to the last byte in the dictionary buffer – it will be the value coded as the substring index. If (`maxl` = 4 and `dn` < 256) or (`maxl` = 5 and `dn` < 65536) or `maxl` ≥ 6, then LZPP starts to encode the substring:
         2.1.1.1. The flag 1 is coded,
         2.1.1.2. Both bytes of length of the substring decreased by `MIN_DL`, that is the number `maxl–MIN_DL` (which is in range 0-65535), are coded. The mechanism of exclusions

of impossible symbols is used, when there is less than 64KB of data left,

2.1.1.3. If (`maxl` ≥ 5 and `w` > 256), then the byte containing bits 8-15 of `dn` is coded. If `w` < 65536 then during the process of encoding the impossible values are excluded (eg. when `w` = 40000, then encoded byte cannot have any of the values 157-255).

2.1.1.4. If (`maxl` ≥ 6 and `w` > 65536), then the most significant byte of `dn` is coded. The mechanism of exclusions of impossible values is also used,

2.1.1.5. Next, the least significant byte of `dn` is encoded. The mechanism of exclusions of impossible values is used. However, now it consists of two steps:

(a) LZPP excludes all values, which together with the rest of bytes in dn would give the index of the substring beginning from the one of the symbols excluded in this moment (see criteria of symbol exclusions discussed in paragraphs 1.6 and 1.7),

(b) If `w` < 256, then LZPP excludes additionally all the values from the range [`w`; 255],

2.1.1.6. All necessary variables and structures are updated, including the hash table, buffer, variable `w` and three-byte sequences coder,

2.1.1.7. Go back to point 2.

2.1.2. If there is no appropriate substring, it is checked if the first three bytes of coding buffer can be encoded as a three-byte sequence (that is, if the counter of that sequence is greater than zero):

2.1.2.1. If it is so, than:

(a) The flag 2 is encoded,

(b) Given substring is encoded by the three-byte coder with exclusions of sequences beginning with one of the excluded at the moment symbols,

(c) All necessary variables and structures are updated, like in point 2.1.1.6.

(d) Go back to point 2.

2.1.2.2. Else, LZPP encodes a single symbol. First, if `w` > 0, it is checked whether the given symbol has nonzero value of the counter in the current context of order 1:

(a) If it is so, then:

(i) The flag 3 is encoded,

(ii) After removing all currently excluded symbols from the alphabet, the given symbol is encoded by the arithmetic coder using symbol counters appropriate for a given context,
(iii) All necessary variables and structures are updated, like in 2.1.1.6.
(iv) Go back to point 2.

(b) Else:
(i) The flag 0 is encoded,
(ii) After removing from the alphabet all currently excluded symbols and these symbols, whose counters are nonzero in the current context of order 1, the given symbol is encoded by the arithmetic coder using symbol counters appropriate for the context of order 0,
(iii) The counter of the given symbol in the current context of order 1 is updated, as well as all necessary variables and structures, like in 2.1.1.6.
(iv) Go back to point 2.

3. Algorithm LZPP finishes.

3. Results and conclusions

One can implement many other optimisations different from those described in this paper, but the results achieved by LZPP are quite satisfactory.

Table 1 shows, that LZPP achieves good results on Calgary Corpus [7]. The compression ratio of LZPP is better by about 30% than that of LZSS and by about 10% than WinZip's (and other similar, like pkzip, zlib, etc.). However, it is worse by about 1,5%-2% compared with the commercial WinRar. Unfortunately, too big amount of remaining redundancy places all LZ family methods far behind the algorithms based on PPMZ [8], including RK.

Table 1. Comparison of results achieved by LZPP algorithm with results of other algorithms (Calgary Corpus).

File	Original size	LZSS		LZPP		WinZIP 8.0		WinRAR 2.9		PPMZ2 0.8		RK 1.04	
		bytes	bpc	bytes	bpc	bytes	bpc	bytes	bpc	bytes	bpc	bytes	bpc
bib	111261	39660	2,8517	32659	2,3483	34878	2,5078	32759	2,3555	23892	1,7179	24292	1,7467
book1	768771	355980	3,7044	278018	2,8931	312257	3,2494	275058	2,8623	210942	2,1951	207420	2,1585
book2	610856	232189	3,0408	185356	2,4275	206134	2,6996	179801	2,3547	140708	1,8428	139548	1,8276
geo	102400	91180	7,1234	61815	4,8293	68392	5,3431	61522	4,8064	58603	4,5784	47632	3,7213
news	377109	166683	3,5360	125811	2,6690	144377	3,0628	125619	2,6649	103945	2,2051	105364	2,2352
obj1	21504	12772	4,7515	10054	3,7403	10297	3,8307	9815	3,6514	9856	3,6667	9452	3,5164
obj2	246814	101521	3,2906	75750	2,4553	81064	2,6275	73135	2,3705	69144	2,2412	61712	2,0003
paper1	53161	22164	3,3354	18026	2,7127	18518	2,7867	18074	2,7199	14699	2,2120	15036	2,2627
paper2	82199	34659	3,3732	28436	2,7675	29642	2,8849	28502	2,7740	22447	2,1846	22676	2,2069
paper3	46526	21893	3,7644	17515	3,0116	18049	3,1035	17752	3,0524	14327	2,4635	14640	2,5173
paper4	13286	7035	4,2360	5394	3,2479	5509	3,3172	5499	3,3112	4634	2,7903	4788	2,8830
paper5	11954	6351	4,2503	4919	3,2920	4970	3,3261	4957	3,3174	4336	2,9018	4480	2,9982
paper6	38105	16370	3,4368	12996	2,7285	13188	2,7688	13063	2,7425	10927	2,2941	11180	2,3472
pic	513216	91381	1,4244	50126	0,7814	52359	0,8162	49046	0,7645	48278	0,7526	30708	0,4787
progc	39611	16365	3,3051	13082	2,6421	13237	2,6734	13120	2,6498	11174	2,2567	11404	2,3032
progl	71646	20431	2,2813	15929	1,7786	16140	1,8022	15739	1,7574	12960	1,4471	13280	1,4828
progp	49379	14227	2,3049	10857	1,7590	11162	1,8084	10749	1,7415	8943	1,4489	9256	1,4996
trans	93695	24287	2,0737	18030	1,5395	18838	1,6085	17916	1,5297	14225	1,2146	14592	1,2459
Sum	3251493	1275148	3,1374	964773	2,3737	1059011	2,6056	952126	2,3426	784040	1,9291	747460	1,8391
xplik	3251493	1258250	3,0958	958400	2,3581	1080523	2,6585	935940	2,3028	804028	1,9782	782320	1,9248

The tests conducted on Canterbury Corpus [9] give the results different from the above. They are shown in table 2.

Table 2. Comparison of the results achieved by the LZPP algorithm with the results of other algorithms (Canterbury Corpus)

File	Original size	LZSS		LZPP		WinZIP 8.0		WinRAR 2.9		PPMZ2 0.8		RK 1.04	
		bytes	bpc	bytes	bpc	bytes	bpc	bytes	bpc	bytes	bpc	bytes	bpc
alice29.txt	152089	61447	3,2322	51015	2,6834	54161	2,8489	50954	2,6802	39131	2,0583	38928	2,0476
asyoulik.txt	125179	57329	3,6638	46388	2,9646	48798	3,1186	46702	2,9847	36139	2,3096	36264	2,3176
cp.html	24603	9629	3,1310	7781	2,5301	7955	2,5867	7901	2,5691	6642	2,1597	6660	2,1656
fields.c	11150	3729	2,6755	3125	2,2422	3109	2,2307	3084	2,2127	2643	1,8963	2688	1,9286
grammar.lsp	3721	1516	3,2593	1256	2,7003	1216	2,6144	1237	2,6595	1071	2,3026	1176	2,5284
kennedy.xls	1029744	317389	2,4658	86624	0,6730	209703	1,6292	119785	0,9306	176724	1,3730	23924	0,1859
lcet10.txt	426754	162440	3,0451	128129	2,4019	144400	2,7069	126098	2,3639	95875	1,7973	94368	1,7690
plrabn12.txt	481861	224948	3,7347	175874	2,9199	194246	3,2249	174952	2,9046	132281	2,1962	130932	2,1738
ptt5	513216	92912	1,4483	50126	0,7814	52359	0,8162	49046	0,7645	48287	0,7527	30704	0,4786
sum	38240	16656	3,4845	12091	2,5295	12750	2,6674	11687	2,4450	12135	2,5387	10832	2,2661
xargs.1	4227	2130	4,0312	1742	3,2969	1730	3,2742	1743	3,2988	1506	2,8502	1624	3,0736
Sum	2810784	950125	2,7042	564151	1,6057	730427	2,0789	593189	1,6883	552434	1,5723	378100	1,0761

On Canterbury Corpus the LZPP algorithm is better by 68% than the simple LZSS, about 30% better than WinZip and 5% better than WinRar. Thanks to good compression of the file 'kennedy.xls' LZPP achieved the result only slightly worse than PPMZ2. However, due to a very clever combination of PPMZ and LZ the RK algorithm is practically beyond any of the remaining methods.

References

[1] Storer J.A., Szymański T.G., *Data compression via textual substitution*, Journal of the ACM, 29 (1982) 928.
[2] Ziv J., Lempel A., *A Universal Algorithm for Sequential Data Compression*, IEEE Transactions on Information Theory, 23 (1977) 337.
[3] Ziv J., Lempel A., *Compression of Individual Sequences via Variable-Rate Coding*, IEEE Transactions on Information Theory, 24 (1978) 530.
[4] Shannon C.E., *A mathematical theory of communication. Bell System Technical Journal*, 27 (1948) 379 and 623.
[5] Campos A.S.E., *Range coder*, http://www.arturocampos.com, Barcelona, (1999).
[6] Cleary J.G., Witten I.H., *Data compression using adaptive coding and partial string matching*, IEEE Transactions on Communications, 32 (1984) 396.
[7] Bell T.C., Cleary J.G., Witten I.H., *Text Compression*, Prentice Hall, Englewood Cliffs, NJ, (1990).
[8] Bloom Ch., *Solving the Problems of Context Modelling*, http://www.cbloom.com/papers/ppmz.zip, (1998).
[9] Arnold R., Bell T.C., *A corpus for the evaluation of lossless compression algorithms*, IEEE Data Compression Conference (DCC), (1997).
[10] Pylak P., *Metody optymalizacji algorytmów bezstratnej kompresji danych*, Praca magisterska, Katolicki Uniwersytet Lubelski, Lublin (2002), in Polish.