Department of Computer Science
Faculty of Science
Pavol Jozef Šafárik University

Peter Vojtáš (Ed.)

ITAT 2004 Information Technologies - Applications and Theory

Workshop on Theory and Practice
of Information Technologies, Proceedings
Slovakia, September 2004

Department of Computer Science
Faculty of Science
Pavol Jozef Šafárik University

Peter Vojtáš (Ed.)

ITAT 2004
Information Technologies - Applications and Theory

Workshop on Theory and Practice
of Information Technologies, Proceedings
Slovakia, September 2004

Preface

The 4th workshop **ITAT'04 – Information Technologies – Applications and Theory** (`http://ics.upjs.sk/itat`) was held in chalet Popradské Pleso (`http://www.horskyhotel.sk/Majo/indexen.html`) located 1494 meter above the sea level in High Tatra, Slovakia, in September 2004.

ITAT workshop is a place of meeting of people working in informatics from former Czechoslovakia (official languages are Czech and Slovak, although other languages are not forbidden). The workshop was founded together with the Institute of Computer Science at Faculty of Science, P.J. Šafárik University as one of its corner stones.

Emphasis is on exchange of information between participants, rather than make it highly selective. Workshop offers a first possibility for a student to make a public presentation. On the other side it offers a student possibility to discuss with the "elders". A big space is devoted to informal discussions (the place is chosen at least 1000 meter above sea level in a location not directly accessible by public transport).

This workshop coordinates its activities with workshops with similar working attitude in the Czech Republic CLA, Dateso, and MIS.

Thematically workshop ranges from Foundations of informatics, security, through data and semantic web to software engineering. Papers are divided into following classes:

- original scientific papers (fulfilling comparable international criteria at workshops);
- didactics of informatics;
- work in progress;
- extended abstracts and
- student papers (here we offer to students a first acquaintance with writing an own paper, refereeing process and presentation – full validation of original ideas is not the top requirement here).

All papers were refereed by at least two independent referees. There were 25 submissions.

The workshop was organized by Institute of Informatics of University of P.J. Šafárik in Košice and Institute of Computer Science of Academy of Sciences of the Czech Republic, Prague.

Partial financial support has to be acknowledged from the slovak grant VEGA 1/0385/03 "Intelligent search, retrieval and processing of information", the slovak IT project "Tools for knowledge acquisition and organization from heterogeneous information sources" and by the project 1ET100300419 of the Program Information Society (of the Thematic Program II of the National Research Program of the Czech Republic) "Intelligent Models, Algorithms, Methods and Tools for the Semantic Web Realization".

Peter Vojtáš

Program Committee

Gabriela Andrejková, *University of P.J. Šafárik, Košice, SK*
Mária Bieliková, *Slovak University of Technology in Bratislava, SK*
Viliam Geffert, *University of P.J. Šafárik, Košice, SK*
Karel Ježek , *The University of West Bohemia, Plzeň, CZ*
Jozef Jirásek, *University of P.J. Šafárik, Košice, SK*
Jana Kohoutková, *Masaryk University, Brno, CZ*
Ján Paralič, *Technical University in Košice, SK*
Martin Plátek, *Charles University, Prague, CZ*
Jaroslav Pokorný, *Charles University, Prague, CZ*
Karel Richta, *Czech Technical University, Prague, CZ*
Václav Snášel, *Technical University VŠB in Ostrava, CZ*
Vojtěch Svátek, *University of Economics in Prague, CZ*
Július Štuller, *Institute of Computer Science, AS CR, Prague, CZ*
Peter Vojtáš, (chair), *University of P.J. Šafárik, Košice, SK*

Organizing Committee

Hanka Bílková, *Institute of Computer Science, AS CR, Prague, CZ*
Peter Gurský, *University of P.J. Šafárik, Košice, SK*
Stanislav Krajči, *University of P.J. Šafárik, Košice, SK*
Rastislav Lencses, *University of P.J. Šafárik, Košice, SK*
Katarína Mičková, *University of P.J. Šafárik, Košice, SK*

Organization

ITAT 2004 Information Technologies – Applications and Theory was organized by

University of P.J. Šafárik, Košice, SK
Institute of Computer Science, AS CR, Prague, CZ

Table of Contents

Original Scientific Papers

Didactics of Informatics

Work in Progress

Extended Abstract

Student Papers

PVM algorithms for some problems in bioinformatics

Hongmei He, Xuan Liu, Matthew Newton, and Ondrej Sýkora

Department of Computer Science, Loughborough University, Loughborough, Leicestershire LE11 3TU, The United Kingdom
H.He, X.Liu, M.C.Newton, O.Sykora@lboro.ac.uk

Abstract. We design and analyze implementation aspects of a PVM version of the well known Smith-Waterman algorithm, and then we consider other problems important for bioinformatics, such as finding longest common substring, finding repeated substrings and finding palindromes.

Key words: Smith-Waterman algorithm, PVM algorithms, the longest common string, the longest repeated string, the longest Palindrome

1 Introduction

String database searching is one of the most important and challenging tasks in bioinformatics. It is necessary to find the best match between two given DNA or protein strings. In the match, we have a penalty for opening gaps or extending gaps for each of the strings. The best match is the one with the minimum sum of such penalties. Pairwise comparison provides computer tools to directly compare two strings. They are the starting points for all kinds of string analysis. These tools can be very useful in string analysis, cloning projects, PCR analysis, and many more.

The developers of hardware and software database searching and handling are facing a great challenge from the genetic-string information that rises quickly to large amounts. Although the computing resources have increased exponentially for decades, the genetic string information maybe has extended beyond the growth speed of the computing power. If the above facts keep on going, it will be necessary to use more expensive supercomputers to search existing databases.

Cluster computing is a relatively new field of research in parallel computing. A cluster computer typically exists as a set of PCs or workstations interconnected by a switch or a fast ethernet network. In a certain sense a cluster is just a parallel computer with a, possibly, slower interconnection network. Clusters offer incredible computing power at a fraction of the cost of parallel supercomputers. In comparison, their communication power is modest, and no dedicated software is provided. For communication one mostly relies on the concepts of PVM [12] or the MPI library of communication routines [10], which provides a reasonably efficient set of primitives.

2 Smith-Waterman algorithm

When looking for strings in a database similar to a given query string, the search programs compute an alignment score for every string in the database. This score represents the degree of similarity between the query and database string. The score is calculated from the alignment of the two strings, and is based on a substitution score matrix and a gap-penalty function. A dynamic programming algorithm computing the optimal local-alignment score was first described by Waterman and Smith [17]. Later Gotoh [4] reduced the complexity of the algorithm. Both versions have been implemented many times.

Database searches using the algorithm are unfortunately quite slow on ordinary computers, so many heuristics have been developed, such as FASTA and BLAST. These methods have greatly reduced the running time, however, at the expense of sensitivity. As a result, a distantly related string may not be found in a search by using these heuristic algorithms. To get faster, but optimal solutions, one should use parallel computing. For example the authors of [18] used a 64 processor system to align 324 protein strings in 13 hours instead of single processor machine running 29 days to execute the same work.

One of the first parallel implementations of the Smith-Waterman algorithm or Gotoh's version of it is due to Sitting et al.[15] and by [6]. In [7] a cluster implementation of Gotoh's version[4] of the Smith-Waterman algorithm was designed and run on a cluster of workstations using the PVM paradigm. They claimed to achieve similar performance to a massively parallel computer Intel iPSC/860 hypercube. Special parallel hardware to implement the Smith-Waterman algorithm was developed by more researchers e.g. [3], [11], [8], [13]. Systolic algorithms to implement the Smith-Waterman algorithm were also created [14].

Our PVM implementation of the Smith-Waterman Algorithm has been run on a cluster of 20 Sun ULTRAsparc 5 workstations running Debian GNU/Linux. They are connected with 100Mbit Ethernet using Cisco 2950 switches. We tested it for different sizes of strings, different relative sizes and for different numbers of workers.

We used PVM, because it is standard and it frees the algorithm designer from load balancing, resource control, fault tolerance and other problems with parallel software, and it is still quite popular as well.

2.1 PVM Smith-Waterman algorithm

The PVM Smith-Waterman algorithm is shown in Algorithm 1, in which y is the pattern array, x, the text array, $y[u, v]$, the substring of y from index u to v, p, the number of workers in the cluster, and g and t are the pattern and text lengths respectively.

2.2 Test results and analysis

In the first experiment, we used patterns and text of six different lengths, from 0.5 K to 16 K. We measured the running time for each pair of text and

Algorithm 1 Parallel Smith-Waterman Algorithm

MASTER :

```
1: Send (y, all); k= 0;
2: while (k < t) do
3:    Send (x[k, k + (g/p) –1],all);
4:    k = k + g/p
5: end while
```

Worker(r) :

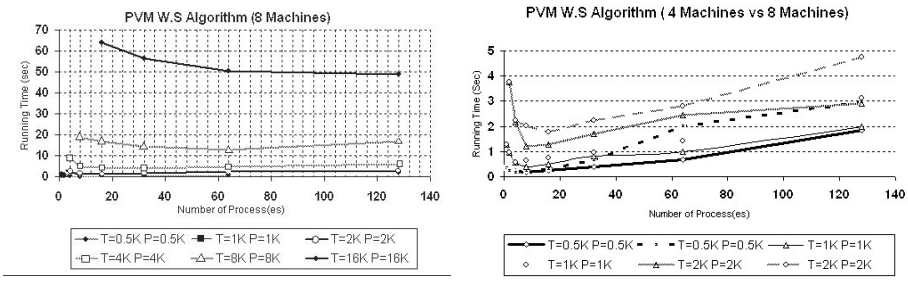
```
1: while (k < t) do
2:    if data from worker(r-1) was received then
3:       WS(y[r(g/p),(r+1)(g/p)–1],x[k, k+(g/p)–1]}
4:       Send (Return data of WS(y[r(g/p),(r+1)(g/p)–1],x[k, k+(g/p)–1]},
5:       worker(r + 1));
6:    end if
7:    if r = p then
8:       worker(r) returns Best
9:    end if
10: end while
```

Smith-Waterman Algorithm :

```
1: WS(string1,string2)
2: {Define f[i, j] as maximal similarity score, d as the cost of deletion and sim[i, j]
   as the similarity of the i-th character of the pattern and the j−th character of
   the text}
3: Best = 0; f[0, 0] = 0;
4: for (0 < i ≤ string1_length) do
5:    for (0 < j ≤ string2_length) do
6:       {f[i, j]=max{f[i−1, j]−d, f[i, j−1]−d,f[i-1, j-1]+sim[i, j]};
7:       Best = max ( f[i, j],Best)
8:    end for
9: end for
```

a pattern. We also tested different numbers of workers (see Fig. 1(a)). In another experiment, we used the same texts and patterns and worker numbers, but ran on only 8 machines. In Fig. 1(b) both cases are compared.

The running time of Gotoh's version of the Smith-Waterman algorithm is $O(pattern\_length \times text\_length)$, where the pattern length is different from the text length. In our experiments the length of text and of pattern were equal. The total parallel running time is composed of computation time and communication time. In our parallel Smith-Waterman algorithm, $2p - 1$ (where p is the number of workers) periods of computation of size $(text\_length/2)^2$ on every active worker are followed by communication of at most p messages, each one being $text\_length/p$ in length, which were sent and received by workers across PVM. From our experiments followed, that our parallel Smith-Waterman algorithm's running time is $O((text\_length)^2/p)$. Test results show that the number of workers plays a role, if the total length of input becomes larger. Also if the



(a) (b)

Fig. 1.

total length of input exceeds a limit, it matters if we use more virtual processors than the real number of processors.

3 PVM algorithm for finding repeated strings

An important and very usual activity in bioinformatics is detection of repeated patterns in strings.

A repeated substring is one having at least two matching occurrences at distinct positions within the string, with the possibility that such occurrences may overlap. A repeated substring may be said to be maximal if the match of a pair of its instances cannot be extended further in either direction: Given string y, with length $n > 0$, identify and locate the longest substring $|x|$ occurring at one or more distinct string overlapping positions in y.

There are many sequential methods for finding repeated substrings [16]. Our PVM algorithms are efficient, especially in the case of finding all repeated substrings.

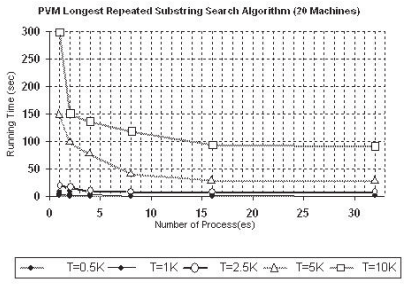


Fig. 2.

First we discuss a general problem of finding a non-empty substring x of a string y which repeats in a non-overlapping position in the string y, i.e. the substring x of length m repeats in y, so that there are two positions $k < l$ in the string y such that $x[i] = y(k + i - 1) = y(l + i - 1)$ for $1 \leq i \leq m$ and $k + m < l$. A special subproblem of this problem is to find the longest tandem substring in a string. When looking for the longest tandem substrings we want to find concatenated two longest repeated substrings. It means the first element of the second longest substring is next to the the final element of the first longest substring.

Algorithm 2 Parallel Longest Repeated Substring Algorithm

MASTER :

1: {x is the text array, p is the number of workers in the cluster, g is the text lengths.}
2: Initialisation
3: Send the x to all workers.
4: Receive the Longest Repeated Substring,maxRepeatedStr from all workers.
5: Compare each result from workers and output the Longest Repeated Substring.

Worker(r**)** :

1: define maxRepeatedStr with three elements, the length of string,$maxRepeatedStr.len$,the first start position in x, $maxRepeatedStr.x1$, and the second start position in x, $maxRepeatedStr.x2$
2: Initialize maxRepeatedStr with the element, len=0;
3: step=$g/(2 \times p) + 1$;
4: frontPart=$WorkerID \times step$;
5: rearPart=$(2(p-1) - WorkerID) \times step$;
6: **for** $(k = frontpart; k < frontPart + step; k++)$ **do**
7: { substring=searchSubstr(x, g, k) }
8: if(substring.len>maxRepeatedStr.len) maxRepeatedStr=substring;
9: **end for**
10: **if** $(frontPart != rearPart)$ **then**
11: **for** (k=rearPart; k<rearPart+step & k<g; k++) **do**
12: substring=searchSubstr(x, g, k)
13: if(substring.len>maxRepeatedStr.len) maxRepeatedStr=substring;
14: **end for**
15: **end if**
16: send the maxRepeatedStr to Master.

Search Longest Repeated Substring :

1: SearchSubstr(X,g, k)
2: {
3: define maxZeroString.
4: Initialise $maxZeroString$ with the element $len = 0$;
5: $i = 0$
6: **while** $(i < g - k)$ **do**
7: $val[i] = X[i] - X[i + k]$
8: i++;
9: **end while**
10: set variable, len to record the length of current 0 string
11: $i = 0$
12: **while** $(i < g - k)$ **do**
13: i=the start position of the next 0 string in array,val
14: $counter = thelengthofthe0string$
15: i=i+counter
16: if $(k < counter)$ $len = k$; (overlap)
17: else $len = counter$; ($k \geq counter$, no overlap)
18: if (len>maxZeroString.len) {
19: maxZeroString.len=len;
20: maxZeroString.x1 = $i-$ counter;
21: maxZeroString.x2 = $i + k-$ counter;
22: }
23: **end while**
24: Return maxZeroString;
25: }

What follows is the PVM algorithm for both non-overlapping and overlapping cases.(see Algorithm 2) Fig. 2 shows test results for searching longest repeated substring.

3.1 Analysis

Work (we count the number of comparisons executed) of this PVM algorithm is m^2 and the running time is $O(m^2/p)$, where p is the number of workers and m is the string size. The running time is almost entirely composed of computation running time as communication is solely used to start the computation and to combine the partial results of workers. The same holds for all other parallel algorithms in the rest of the article. The parallel algorithm decreases the running time for searching longest repeated substring, but if we used too many workers to find the longest repeat string in small string, the running time would increase.

3.2 Discussion of the overlapping case

Fig. 3 shows the procedure of searching longest substring.

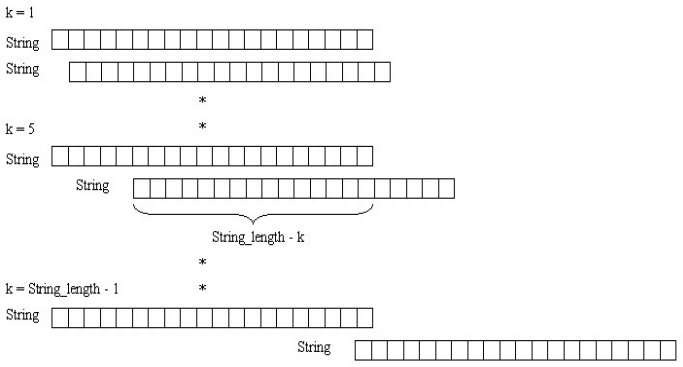


Fig. 3.

We use k for the number of times the string is shifted. We calculate the difference of the common part of the two strings, count the size of the longest matching substring, and get the corresponding start positions of the longest repeating substrings. We use a variable, *counter*, to record the length of the matching substring, and we consider overlapping of repeated substrings too.

In the non-overlapping case, for example, using a string, "AAAAAAA", we can say the longest repeating substring is "AAA", whose first start position is 0, and second start position is 3. In another example, a string: "AGTCAGTCA", the longest repeating substring is "AGTC" and rather than "AGTCA". However, in the PVM algorithm, if there are overlapping substrings in the string, the

value of *counter* could be more than the real value of length. Fig. 6 describes the procedure of finding an overlapping substring. For the example (Fig. 4) of "AAAAAAA", we have $k = 1, counter = 6$, and obviously, it is not correct to record the value of *counter*. For the example (Fig. 5) of "AGTCAGTC", we have $k = 4, counter = 5$ and we also cannot record the current value of *counter*. Actually, k is the real length of repeating substrings instead of *counter*. So we just need to record the value of k as it is largest, when $k < counter$.

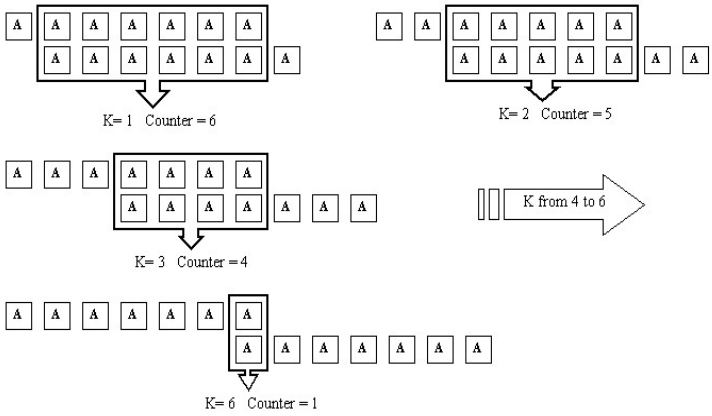


Fig. 4.

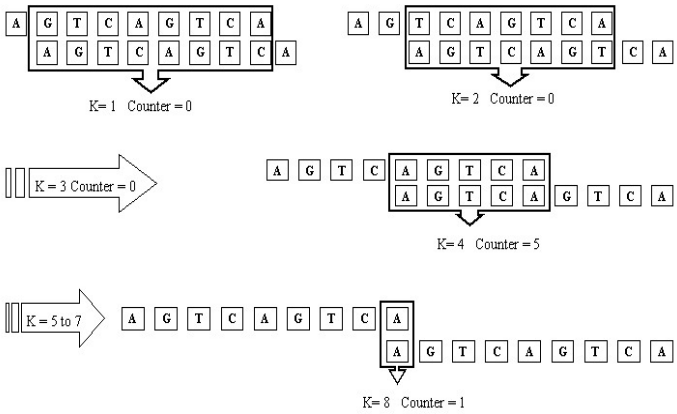


Fig. 5.

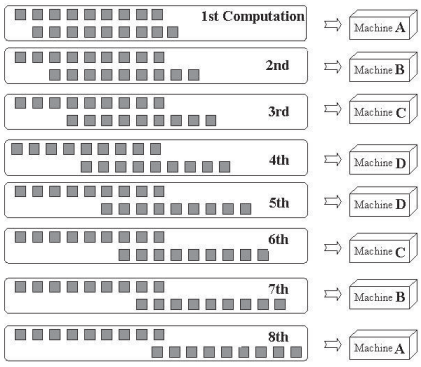


Fig. 6.

3.3 Load balancing—onion peeling principle

To keep the workload balanced equally over processes, we suggest using the so called onion peeling principle which ensures almost equal distribution of workload. In Fig. 6 we show an example with 4 workers and computation of 8 blocks of different size. The size of neighbouring blocks differs by 1. Computation of the 8 blocks will be distributed over these 4 workers as follows. Like peeling an onion skin, the first worker computes the outside layer, which are the first and the eighth blocks. The second worker will execute the computation of the next layer, which are the second and the seventh blocks, and so on.

4 PVM algorithm for the longest common substring

The problem of finding the longest common substring is an important problem too. For example, in DNA sequencing, shotguns[2] are used, and one should find pairs of shotguns with the longest common prefix from one shotgun, matching the suffix of another shotgun.

4.1 Longest common substring algorithm

A longest common substring of two strings is a substring common to both, having maximal length, i.e. it is at least as long as any other common substring of the strings. Given two strings x and y, with lengths $|x| = m, |y| = n$, where $0 < m <= n$, find $lcs(x, y)$, where $lcs(x, y)$ is a longest common substring of x and y. We could also be interested in f longest common substrings, where f is a constant.

There are many sequential methods to solve this problem (see e.g. [16], [1]). We suggest a relatively simple parallel algorithm with a load balancing idea described in Algorithm 3. The ID of a worker is denoted “WorkerID”.

Algorithm 3 Parallel Longest Common Substring Algorithm

MASTER :

{y is the pattern array, x is the text array,p is the number of workers in the cluster, g and t are the pattern and text lengths respectively}
Initialization :
Send the y to all workers.
Send the x to all workers.
Receive the Longest Common Substring from all workers.
Compare each result from workers and output the Longest Common Substring.

Worker(r) :

define maxCommonString with three elements, the length of string,$maxCommonString.len$, the start position in y, $maxCommonString.y$, and the start position in x, $maxCommonString.x$
Initialize maxCommonString with the element, len=0;
Step=$g + t/(2p) + 1$;
$FrontPart = WorkerID \times step$;
According to the distribution of machines, each worker in the cluster includes two parts, front part and rear part.
$RearPart = (p + WorkerID) \times step$;
for (frontpart < k< frontPart + step) **do**
  substring=LCS(y, x, g, t, k)
  if ($substring.len > maxCommonString.len$)
  $maxCommonString = substring$;
end for
for (k=rearPart; k<rearPart+Step & k<g+t; k++) **do**
  substring=LCS(y, x, g, t, k)
  if ($substring.len > maxCommonString.len$)
  $maxCommonString = substring$;
end for
send the maxCommonString to Master.

Search Longest Common Substring :

LCS(Y,X,g,t,k)
define maxZeroString.
Initialise $maxZeroString$ with the element $len = 0$;
if (g-k-1>0){Start1=g-k-1; Start2=0}
else { Start1=0; Start2=k-g+1;}
if (g-Start1)<(t-Start2)
$L = g - Start1$
else $L = t - Start2$
set m=0; maxLen=0;
set i=Start1;j=Start2;
while (m < L) **do**
  $val[m++] = Y[i++] - X[j++]$
end while
set $i = 0$; set maxlen=0;
while ($i < L$) **do**
  i=the start position of next 0 string in array, val;
  counter=the length of the 0 string
  i=i+counter;
  if ($counter > maxZeroString.len$)
  {$maxZeroString.len = counter$;
  $maxZeroString.y = Start1 + i - counter$;
  $maxZeroString.x = Start2 + i - counter$; }
end while
return maxZeroString;

4.2 Test results and analysis

We used 20 machines in the cluster to run the PVM algorithm, in which different length text strings and pattern strings were applied. We also tested different numbers of workers (see Fig. 7).

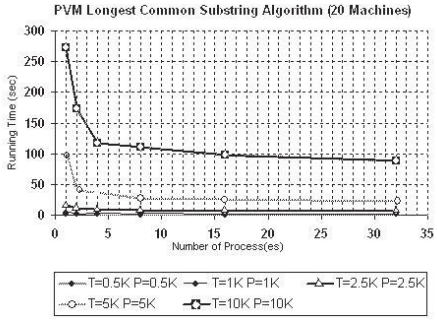


Fig. 7.

Work of this PVM algorithm (we count the number of comparisons to be executed) is $\min^2(|x|, |y|) = m^2$ and the running time is $O(\min^2(|x|, |y|)/p) = O(m^2/p)$, where p is the number of processors.

4.3 Load balancing

In parallel longest common substring algorithm, we use a kind of load balancing implementation method, which is similar to the onion peeling principle. As Fig. 8 shows, the computation between two strings contains two parts, which will be carried out when the upper string slides from the right end of the lower string to the centre, and go on to slide from the centre to the left end. The longest computing time will happen when the upper string is on the central position of the lower string. We arrange these calculations so each worker works in tandem, which means any PVM worker will be given some easy computations (like the 1st and 2nd) and some of hard computations (such as the 6th and 7th). Other PVM workers may be arranged the 3rd, 4th, 8th and 9th computations to keep the load balanced, and so on.

5 PVM algorithm for the longest palindrome substring

A string x of length n such that $x(i) = x(n - i + 1), 1 \leq i \leq n$, is called a palindrome. We design a PVM algorithm to find the longest palindrome, and it can be easily and efficiently applied for finding f longest palindromes, or for finding all palindromes of at least some constant length (see Algorithm 4).

Algorithm 4 Parallel Longest Palindrome Substring Algorithm

MASTER :

1: ***Initialisation*** {X is the text array, p is the number of workers, g is the text lengths}
2: Send X to all workers.
3: Receive the Longest Palindrome Substring from all workers.
4: Compare each result from workers and output the Longest Palindrome Substring.

Worker(r) :

1: define maxPalindrome with two elements, the length of string,$maxPalindrome.len$,the start position in x, $maxPalindrome.x$
2: Initialize maxPalindrome with the element, len=0;
3: step=$g/p + 1$;
4: FrontPart=WorkerID $\times$ step;
5: RearPart=(p+WorkerID)$\times$step
6: **for** (frontpart<k<frontPart + step) **do**
7: substring=searchPLD(x, g, k)
8: if(substring.len>maxPalindrome.len) maxPalindrome=substring;
9: **end for**
10: **for** (k=rearPart; k<rearPart+step & k<2$\times g$; k++) **do**
11: substring=searchPLD(x, g, k)
12: if(substring.len>maxPalindrome.len) maxPalindrome=substring;
13: **end for**
14: send the maxPalindrome to Master.

Search Longest Palindrome :

1: SearchPLD(X,g, k)
2: {
3: define $maxZeroString$.
4: Initialise $maxZeroString$ with the element $len = 0$;
5: if (g-k-1)>0 {$start1 = g - k - 1$; $start2 = (g - 1)$}
6: else {$start1 = 0$;$start2 = 2 \times (g - k - 1)$}
7: if ($i > 0$) $L = StringLength - i$
8: else $L = j + 1$
9: set $m = 0$; $i = start1$; $j = start2$;
10: **while** ($m < L$) **do**
11: val$[m++] = X[i++] - X[j--]$
12: **end while**
13: **while** ($i < L$) **do**
14: i=the start position of the next 0 string in array,val
15: $counter = thelengthofthe0string$
16: $i = i + counter$
17: if ($counter > maxZeroString.len$) {
18: $maxZeroString.len = counter$;
19: $maxZeroString.x = start1 + i - counter$;
20: }
21: **end while**
22: Return $maxZeroString$;
23: }

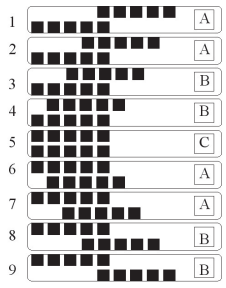


Fig. 8.

5.1 Test results and analysis

In our experiment, we ran our PVM algorithm on a 20 machine cluster. We tested different string sizes from 0.5kb to 10kb. We also used different numbers of workers (see Fig. 9).

This PVM algorithm does m^2 comparisons and the running time is $O(m^2/p)$, where p is the number of processors and m is the string size. From the Fig. 9 we can see that the best running time would be achieved if we applied more workers. But, if the longest palindromes are "tiny", the quickest running time is achieved with "less" workers. This is due to the fact that there is a "larger" number of palindromes and reporting them to the master increases the transmission time.

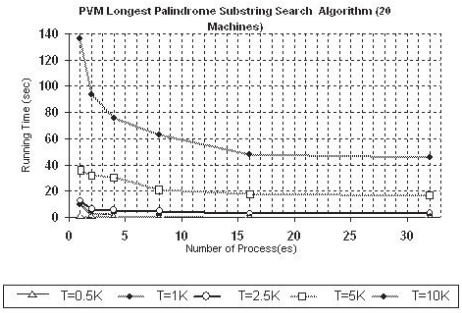


Fig. 9.

6 Conclusions

In this article we designed and tested PVM algorithms to compute some important problems in bio-informatics.

We designed and analyzed implementation aspects of a PVM version of the well known Smith-Waterman algorithm and PVM algorithms for finding the

longest common substring, finding repeated substrings and finding palindromes. We used the so called onion peeling principle to evenly distribute the workload.

We observed some interesting facts, such as the increase in transmission time, causing slowdown, if a larger number of workers was used. However, the PVM algorithms still perform better than their sequential counterparts.

References

1. Crochemore M., Rytter W., Text Algorithms: Oxford Univerity Press 1994.
2. Czabarka E., Konjevod G., Marathe M.V., Percus A.G., Torney D.C.: Algorithms for Optimizing Production DNA Sequencing. In: Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '00), 2000, 399–408.
3. Dale D., Grate L., Rice E., Hughey R.: The ucsc Kestrel General Purpose Parallel Processor. In: Proc. Int. Conf. Parallel and Distributed Processing Techniques and Applications, 1999, 1243–1249.
4. Gotoh O.: An Improved Algorithm for Matching Biological Sequences. Journal of Molecular Biology, **162**, 1982, 705–708.
5. Grice J.A., Hughey R., Speck D.: Reduced Space Sequence Alignment. Comput. Appl. Biosci., **13**, 1997, 45–53.
6. Guan X., Mural R.J., Mann R., Uberbacher E.C.: On Parallel Search of DNA Sequence Databases. In: Proc. 5th SIAM Conference on Parallel Processing for Scientific Computing, SIAM, 1991, 332–337.
7. Guan X., Mural R.J., Uberbacher E.C.: Sequence Comparison on a Cluster of Workstation Using the PVM System. In: Proc. 9th Intl. Parallel Processing Symposium, IEEE Computer Society, 1995, 190–195.
8. Hughey R.: Parallel Hardware for Sequence Comparison and Alignment. Comput. Appl. Biosci., **12**, 1996, 473–479.
9. Martins W.S., del Cuvillo J.B., Useche F.J., Theobald K.B., Gao G.R.: A Multithreaded Parallel Implementation of a Dynamic Programming Algorithm for Sequence Comparison. In: Pacific Symposium on Biocomputing, 2001.
10. `http://www.mcs.anl.gov/mpi/`
11. `http://www.paracel.com/gm/brochure.pdf`
12. `http://www.epm.ornl.gov/pvm/`
13. Rogens T., Seeberg E.: Six-Fold Speed-up of Smith-Waterman Sequence Database Searches Using Parallel Processing on Common Microprocessors. Bioinformatics, **16**, 2000, 699–706.
14. Schmidt B., Schroder H., Schimmler M.: Massively Parallel Solutions for Molecular Sequence Analysis. In: Proceedings of IPDPS'2002.
15. Sitting D., Foulser D.F., Carriero N., McCorkle G., Miller P.L.: A Parallel Computing Approach to Genetic Sequence Comparison: The Master-Worker Paradigm with Interworker Communication. Computers and Biomedical Research, **24**, 1991, 152–169.
16. Stephen G.A.: String Searching Algorithms. Lectures Notes Series on Computing, **3** World Scientific, 1994.
17. Waterman M.S., Smith T.F.: RNA Secondary Structure: A Complete Mathematical Analysis. Mathematical Bioscience, **42**, 1978, 257–266.
18. Yap T.K., Frieder O., Martino R.L.: Parallel Computation in Biological Sequence Analysis. IEEE Transactions on Parallel and Distributed Systems, **9**, 1998, 283–294.

New circular drawing algorithms*

Hongmei He and Ondrej Sýkora

Department of Computer Science, Loughborough University
Loughborough, Leicestershire LE11 3TU, The United Kingdom
H.He, O.Sykora@lboro.ac.uk

Abstract. In the circular (other alternate concepts are outerplanar, convex and one-page) drawing one places vertices of a $n-$vertex $m-$edge connected graph G along a circle, and the edges are drawn as straight lines. The smallest possible number of crossings in such a drawing of the graph G is called circular (outerplanar,convex, or one-page) crossing number of the graph G. This paper addresses heuristic algorithms to find an ordering of vertices to minimise the number of crossings in the corresponding circular drawing of the graph. New algorithms to find low crossing circular drawings are presented, and compared with algorithm of Mäkinen [4], CIRCULAR+ algorithm of Six and Tollis [5] and algorithm of Baur and Brandes [1]. We get better or comparable results to the other algorithms.

1 Introduction

A number of data presentation problems involve drawings of graphs on a two-dimensional surface. Examples include circuit schemas, algorithm animations, software engineering, VLSI pin arrangements, and many others. The primary requirement of graph drawing algorithms is that the output graph should be readable, that is, it should be easy to understand and follow. There are many optimisation goals for variation from one application to another and from one human to another, such as minimising crossings, minimising area, minimising bends (in orthogonal drawings), maximising display of symmetries, etc. In the circular [5] (outerplanar [2], convex [7], or one-page [6]) drawing, one places vertices of a $n-$vertex $m-$edge connected graph $G = (V, E)$ along a circle, and the edges are drawn as straight lines. The common task of circular algorithms is to find an ordering $f : V \rightarrow \{0, 1, ...n - 1\}$ of the vertices, minimising the number of edge crossings. Minimum possible number of crossings of any ordering is called circular [5](outerplanar [2], convex [7], or one-page [6]) crossing number of the graph G. We will follow the notation in [6] i.e. one-page crossing number to avoid possible misunderstandings: ν_1(G). Recently, a lot of research was done for circular graph drawing. In this paper, new algorithms to find low crossing circular drawings are presented. We compare performance of our algorithm with the well known algorithms, such as Mäkinen's algorithm [4] and

* This research was supported by the EPSRC grant GR/S76694/01 and by VEGA grant No. 2/3164/23

CIRCULAR/CIRCULAR+ of Six and Tollis [5] and with Baur and Brandes' algorithm [1] on

- our suite of Random Connected Graphs (RCG) with different densities;
- Rome Graphs, which are from the test suite of GDToolkit and were utilised in [1]. We use the undirected graph sets to test our and other previously published algorithms.
  - RND_BUP: this graph set contains about 200 graphs generated randomly. Each graph in the set is biconnected, undirected and planar.
  - ALF_CU: this graph set contains about 10,000 connected and undirected graphs

We get better or comparable results to the other algorithms.

2 Previously published algorithms

There are two basic ways to minimise the number of crossings.

- Minimising the circular length of the graph. The problem was proved to be NP-hard [4]. The circular length is defined as follows:

$$\sum_{(u,v)\in E} \min(|f(u) - f(v)|, n - |f(u) - f(v)|)$$

  The circular length of the graph shown in the Fig. 1 a) is 53 and the number of edge crossings is 37. See another drawing in Fig. 1 b) where the circular length is reduced to 39 and number of crossings to 10.

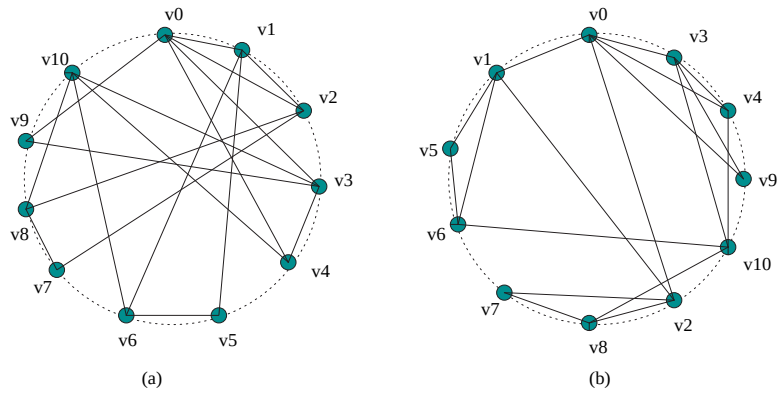


Fig. 1. Circular drawings of a graph. Drawing b) was produced by algorithm of Mäkinen.

– Maximising the number of edges appearing on the circumference of the embedding circle.

The general problem of placing vertices such that the number of edge crossings is minimum, is the well know NP-hard crossing number problem as well as the more restricted problem of Circular Drawing [3].

2.1 Algorithm of Mäkinen

Mäkinen [4] proposed an algorithm attempting to minimise the number of crossings by minimising the circular length of the graph. Algorithm of Mäkinen consists of two steps:

– Two vertices with the highest degrees are placed at positions 0 and n-1, which correspond to the first position of the right and left halves in the drawing.
– The other vertices are processed as follows: left and right connectivity arrays for all the vertices not yet placed are maintained, where the connectivity is the number of adjacent vertices already placed on the left or right half in the drawing. The vertex with the highest value of (*right connectivity-left connectivity*) will be placed on the right half. The vertex with the lowest (*right connectivity-left connectivity*) will be placed on the left half.

The running time of the Mäkinen's algorithm is $O(nm)$.

2.2 Algorithms CIRCULAR and CIRCULAR+

CIRCULAR [5] algorithm reduces the number of edge crossings by maximising the number of edges appearing on the circumference of the embedding circle. The algorithm first removes one edge from every triangle subgraph in the graph, then places the vertices, which are in the longest path of a Depth-First-Search tree along the embedding circle, and finally builds the corresponding vertex ordering. There is additional phase applied after CIRCULAR and the number of crossings is further reduced by local optimisation of the placement of every vertex. The running time of the algorithm CIRCULAR is $O(mn)$ while the running time of CIRCULAR+ is $O(m^2)$, where $m \geq n - 1$. Fig. 2 shows a drawing produced by the CIRCULAR algorithm for the graph from Fig. 1 a).

2.3 Algorithm of Baur and Brandes (BB)

While we were preparing this article a paper of Baur and Brandes [1] was accepted for presentation at WG'04 conference. Therefore we included comparison of our algorithms to their best too. The best algorithm in the Baur and Brandes' article [1] works as follows:

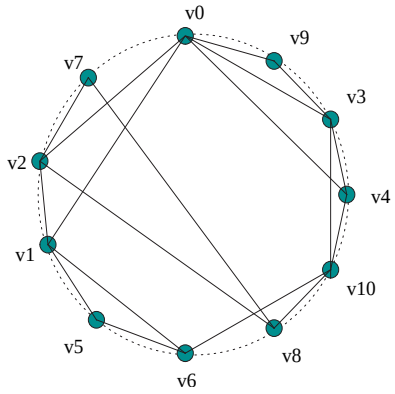


Fig. 2. A drawing of the graph from the Fig. 1 a) produced by CIRCULAR algorithm.

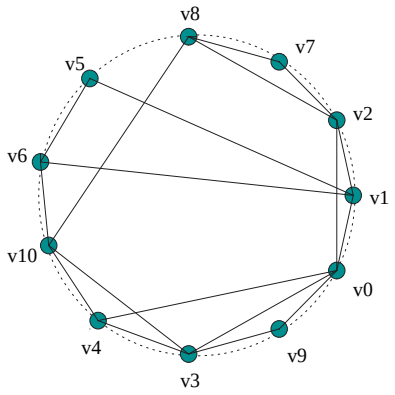


Fig. 3. A drawing of the graph from Fig. 1 a) produced by BB-algorithm.

– Greedy phase: at each step a vertex with the largest number of already placed neighbours is selected, where ties are broken in favour of vertices with fewer unplaced neighbours, and then appended to the end that yields fewer crossings of edges being closed with open edges.
– Sifting phase: Every vertex is moved along a fixed ordering of all other vertices. The vertex is then placed in its (locally) optimal position.

We will use BB to denote the greedy phase. The running time of the algorithm BB with sifting is $O(n^2m)$.

3 Our algorithms

3.1 Algorithm: adjacent vertex with smallest degree first (AVSDF)

Depth First Search starts with a vertex as the root, and travels as far as possible along a path emanating from v. Then backtracks towards v until it finds a first vertex u from which there is an edge to a vertex w not yet visited. Minimising the circular length of a graph "fuzzily" means making every edge of graph (geometrically a chord of the circle) as short as possible. To achieve this, we change the DFS algorithm so that we first place the vertex with the smallest degree, and then visit the adjacencies of current vertices, which have not been visited yet, such that the smallest degree vertex has highest priority for visiting. Since DFS generates a spanning tree of a graph, our algorithm produces optimal (zero crossing) drawing for any tree. A description of the algorithm AVSDF follows.

Algorithm 1 Adjacent Vertex with Smallest Degree First

```
1: Initialise an array order[n], and a stack, S.
2: Get the vertex with the smallest degree from the given graph, and push it into S.
3: while (S is not empty) do
4:    Pop a vertex v, from S
5:    if (v is not in order) then
6:       Append the vertex v into order.
7:       Get all adjacent vertices of v, and push those vertices, which are not in order
         into S with descending degree towards the top of the stack (the vertex with
         smallest degree is at top of S).
8:    end if
9: end while
```

The running time of the AVSDF algorithm without counting of crossings is $O(m)$. With counting of crossings its running time increases to $O(n^2)$.
Fig. 4 presents the circular drawing of the graph from Fig. 1 a) produced by AVSDF algorithm.

3.2 Preprocessing phase

For any graph, vertices forming a branch (mutually connected vertices of degree two except one vertex, which is of degree one) do not produce crossing on a circular layout.
We remove these vertices first and then deal with the remaining subgraph using heuristics and finally reinsert the previously removed vertices. For a sparse graphs this preprocessing reduces substantially running time. In comparison of heuristics we first use preprocessing and then apply heuristics AVSDF, BB, and CIRCULAR to the subgraph.

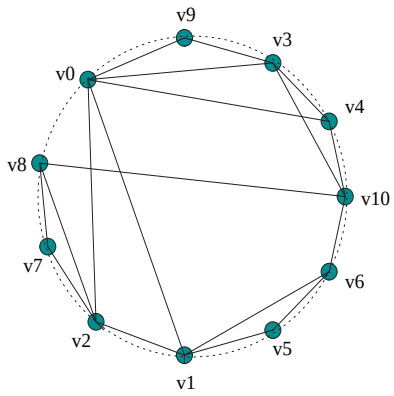


Fig. 4. A drawing of the graph from Fig. 1 a) produced by AVSDF-algorithm.

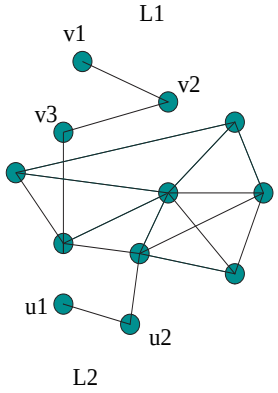


Fig. 5. Example of branches, branch $L1$: $v1, v2, v3$, branch $L2$:$u1, u2$.

3.3 Postprocessing phase–adjusting

One can further improve the algorithm AVSDF. Taking vertices in the order with descending number of crossings on their incident edges (we start with the vertex whose incident edges create the highest number of crossings), we find its best position among the current one and the ones close to adjacent vertices. We call the procedure adjusting. Its running time is $O(nm)$.
In Fig. 6, the vertex, v, whose incident edges create the largest number of crossings, is adjusted first. In this drawing v has three possible positions to try.
In our experiments we combine AVSDF as well as BB with adjusting and sifting.

Algorithm 2 Local adjusting

1: For every vertex calculate the crossings on edges incident to them.
2: Sort the vertices according to descending number of crossings. Let variable, *currentV*, point to the vertex whose incident edges have the largest number of crossings.
3: **for** (all vertices) **do**
4: Get the positions of adjacent vertices of *currentV* into *pList* array.
5: Try all these positions and calculate the crossing number to find the best location for *currentV*.
6: change the pointer,*currentV* to the next vertex
7: **end for**

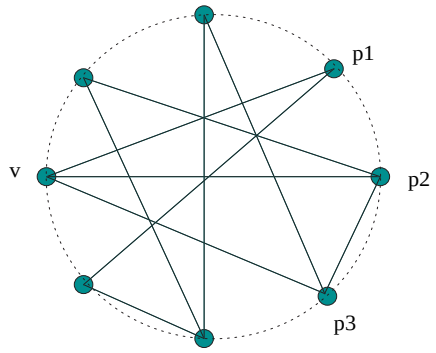


Fig. 6. Three possible positions, $p1, p2, p3$, of vertex, v, to adjust.

3.4 Counting of edge crossings

Edges (i, j) and (k, l) intersect if and only if the positions of i, j alternate with the positions of k, l. This induces the following formula to calculate the number of crossings.
Given the adjacency matrix adjMatrix$[n][n]$ of the graph G, by traversing all edges of the upper triangle of the adjacency matrix, for every edge e, we count the number of edges crossing with e :

$$cr(G) = \sum_{i=0}^{n-3} \sum_{j=i+2}^{n-2} \left(\text{adjMatrix}[i,j] \sum_{k=i+1}^{j-1} \sum_{l=j+1}^{n-1} \text{adjMatrix}[k,l] \right).$$

3.5 Generation of the suite Random Connected Graphs

To test our algorithms and compare them with the others we created a suite Random Connected Graphs with different densities. A complete undirected graph on n vertices, has $n(n-1)/2$ edges. We define density d as the ratio of the number of edges in the graph and the number of edges in the complete graph $(n(n-1)/2)$. The algorithm for random connected graph generation is described as follows:

Algorithm 3 Generation of a random connected graph

{Initialization} :
Create two vertex sets: $uSet[n], vSet[n]$ to store all vertices;
Calculate the number of edges, $eNum = density \times n(n-1)/2$
get a vertex $u \in vSet$ to be stored in $uSet$, and remove it from $vSet$.
{Create a random tree with $n-1$ edges} :
$count = 0$;
while $(count < vNum - 1)$ **do**
  Get a vertex u, from $uSet$ randomly
  Get a vertex v, from $vSet$ randomly
  Set the corresponding element in adjMatrix to 1
  Put v into $uSet$, and remove it from $vSet$
  $count++$
end while
{Add remaining edges} :
while $(count < eNum)$ **do**
  generate an edge e, which is not in the current graph
  $count++$
end while

4 Experiments with algorithms

4.1 Test Suite

- Random Connected Graphs.
  We used three densities 1%, 3%, and 5%. For each density, 12 groups of graphs with different number of vertices were tested; and for every group 10 different graphs were generated and average running time and average number of crossings were calculated.
- Rome graphs
  Rome graphs are several sets of graphs from GDToolkit. Here we use two sets of graphs, RND_BUP and ALF_CU. RND_BUP is a set of random biconnected undirected planar graphs. ALF_CU is a set of connected undirected graphs. For these graphs we first apply preprocessing phase as described in subsection 3.2 and then on the resulting graph run heuristics.

4.2 Comparison of AVSDF, algorithm of Mäkinen, CIRCULAR, and BB

In this subsection we present the results of comparison the algorithms AVSDF, Mäkinen's, CIRCULAR and BB without any postprocessing (adjusting, sifting or local optimisation) improvements.

We carried out experiments on the test suite Random Connected Graphs with densities 1%, 3% and 5%. AVSDF and BB algorithms produced lower crossing drawings than the other two ones. For density 1% (maximal number of vertices in a graph was 260), for density 3% and $n < 120$, and for density 5% and $n < 50$ AVSDF produced better results than BB.

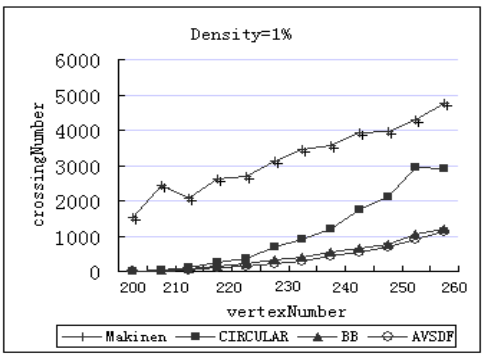


Fig. 7. Test results for density 1%.

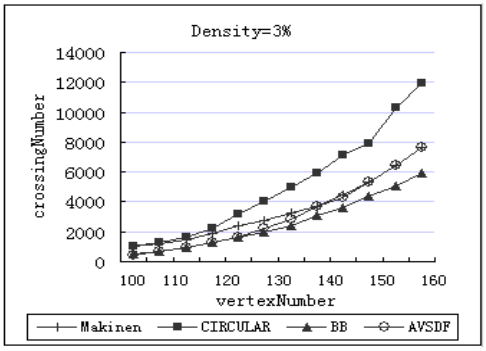


Fig. 8. Test results for density 3%.

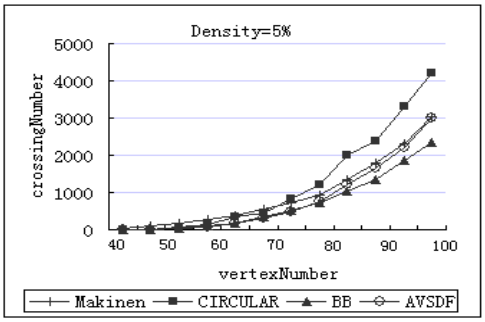


Fig. 9. Test results for density 5%.

4.3 Comparison of AVSDF and BB with different postprocessing

We combined the algorithm AVSDF and BB with adjusting and sifting to get the combinations: AVSDF with adjusting, AVSDF with sifting, AVSDF with adjusting and sifting, AVSDF with sifting and adjusting, BB with adjusting, BB with adjusting and sifting, BB with sifting and adjusting and compared them to the algorithm BB with sifting which was the best algorithm produced in [1]. We make adjusting and/or sifting round only once in every combination. The results are in Table 1, where each unit has three values. The first (second, third) value is the percentage of graphs where the corresponding combination of algorithms achieved smaller (same, greater) number of crossings than BB + sifting. We also calculate the total percentage for all graphs.
Table 2 displays how many times a combination of algorithms produces the best result.

Graphs	Avsdf +sift	Avsdf +adjust	Avsdf +adjust +sift	Avsdf +sift +adjust	BB +adjust	BB +adjust +sift	BB +sift +adjust
RCG (5%)	32,9,59	30,10,60	44,9,47	43, 9,48	20,17,63	59,24,17	67,33,0
RCG (3%)	17,0,83	19,0,81	35,1,64	32,0,68	39,1,60	93,0,7	100,0,0
RCG (1%)	56, 0, 44	52,1,47	55,0,45	56,0,44	5,5,90	63,24,13	63,37,0
RND_BUP	43,17,41	39,17,44	45,16,39	44,17,39	6,46,48	29,66,5	21,79,0
ALF_CU	38,24,38	33,21,46	40,24,36	41,24,35	10,30,60	31,62,7	25,75,0
TOTAL	38,13, 49	36,11, 53	43,13, 44	43,13, 44	14,23,63	49,45, 6	47,53, 0

Tab. 1. Density given in brackets.

Graphs	Avsdf +adjust	Avsdf +adjust +sift	Avsdf +sift	Avsdf +sift +adjust	BB +adjust	BB +adjust +sift	BB +sift	BB +sift +adjust
RCG (5%)	19	39	23	36	14	54	22	48
RCG (3%)	0	16	2	15	0	57	0	34
RCG (1%)	21	51	54	62	0	30	7	33
RND_BUP	67	95	86	92	48	91	71	81
ALF_CU	93	145	129	146	70	153	121	144
TOTAL	200	346	294	351	132	385	221	340

Tab. 2. How many times a combination of algorithms produces the best result.

4.4 Comparison of AVSDF with adjusting and BB with sifting done more than once

The test was similar to the one above, but the adjusting and sifting phases were run more than once - until no improvement was achieved. This number was surprisingly low! Generally it was about 5 and always less than 8. AVSDF with

adjusting produced for graphs from RND_BUP in 35% better, in 18% equal and in 47% worse results than BB with sifting. For ALF_CU graphs, AVSDF with adjusting produced in 29% better, in 22% equal and in 49% worse results than BB with sifting. In case of RCG suite with density 1% (3%, 5%) the percentages were 47, 3, 50 (21, 0,79; 30,10,60).

4.5 Running time

In this subsection we compare the average running time of AVSDF, BB, Mäkinen and CIRCULAR on RCG with different densities,5%,3%,1%, RND_BUP, and ALF_CU of Rome Graphs. AVSDF has absolute superiority for all graphs (see table 3).
This can be explained so that the average degree of a random connected graph with density d is $adeg = d(n-1)$, and this is the number of positions which has to be tried in adjusting postprocessing. Especially for sparse graphs, ajusting takes much less running time than sifting, which tries n positions. The high running time of sifting might make it impossible to use for larger graphs and multiple runs.

Graphs	AVSDF (ms)	Mäkinen(ms)	CIRCULAR(ms)	BB(ms)
RCG (density=5%)	2	16	17	14
RCG (density=3%)	5	24	38	80
RCG (density=1%)	13	56	130	295
RND_BUP	1	1	2	5
ALF_CU	1	1	3	3

Tab. 3. Average running time of AVSDF, Mäkinen, CIRCULAR and BB on different graphs.

5 Conclusion

In this work we designed a new algorithm, AVSDF+, to produce circular drawings, and carried out comparisons with previously published algorithms. AVSDF and BB [1] algorithms without postprocessing (adjusting, sifting or local optimisation) produce better results than the well known algorithm of Mäkinen [4] and CIRCULAR of Six and Tollis [5]. For lower densities AVSDF produces better results than BB.
We combined the algorithm AVSDF with postprocessing adjusting and sifting to get the combinations: AVSDF with adjusting, AVSDF with sifting, AVSDF with adjusting and sifting, AVSDF with sifting and adjusting and compared them with the algorithm BB with sifting which was the best algorithm designed in [1]. The results of experiments show that that AVSDF combined with one or two operations of postprocessing produces approximately same results as BB combined with same operations.

We carried out similar tests on the adjusting and sifting phases run more than once (until no improvement was achieved) for AVSDF with adjusting and BB with sifting. The results show that BB with sifting is slightly better than AVSDF with adjusting but from the the running time point of view the AVSDF with adjusting algorithm is much faster than the algorithm BB with sifting. Another interesting fact was that the number of rounds was low - less than 8 for all tested graphs.

References

1. Baur M., Brandes U.: Crossing Reduction in Circular Layouts. Accepted to WG'2004.
2. Kainen P.C.: The Book Thickness of a Graph II. Congressus Numerantium, **71**, 1990, 121–132.
3. Masuda S., Kashiwabara T., Nakajima K., Fujisawa T.: On the NP-Completeness of a Computer Network Layout Problem. In: Proc. IEEE Intl. Symposium on Circuits and Systems 1987, IEEE Computer Society Press, Los Alamitos, 1987, 292–295.
4. Mäkinen E.: On Circular Layouts. International Journal of Computer Mathematics, **24**, 1988, 29–37.
5. Six J.M., Tollis I.G.: Circular Drawings of Biconnected Graphs. In: ALENEX'99, LNCS, **1619**, Springer, 1999, 57–73.
6. Shahrokhi F., Sýkora O., Székely L.A., Vrťo I.: The Book Crossing Number of a Graph. Journal of Graph Theory, **21**, 1996, 413–424.
7. Shahrokhi F., Sýkora O., Székely L.A., Vrťo I.: Towards the Theory of Convex Crossing Numbers. In: Towards a Theory of Geometric Graphs, J. Pach (ed.), Contemporary Mathematics, **342** American Mathematical Society, Providence 2003, 249–258.

Evolutionary tree genetic programming *

William H. Hsu[1] and Ján Antolík[2]

[1] The Department of Computing and Information Sciences, Kansas State University, 234 Nichols Hall Manhattan, KS 66506-2302, bhsu@cis.ksu.edu
[2] The Department of Computer Sciences, Charles University, Malostranské nám. 25, 118 00 Praha 1, Czech Republic, jan.antolik@matfyz.cz

Abstract. We introduce an extension of a genetic programming (GP) algorithm we call Evolutionary Tree Genetic Programming (ETGP). The biological motivation behind this work is the observation that the natural evolution follows a tree like pattern. We want to simulate similar behavior in artificial evolutionary systems such as GP. In this article we provide multiple reasons why we believe simulation of this phenomenon can be beneficial for GP systems. We present various empirical results from test runs. The performance of the ETGP algorithm is compared to the performance of GP system. Code size and variance are reduced by a robust but insignificant percentage in both problems, but no significant speedup is found.

1 Introduction

Evolutionary algorithms are one of the more promising techniques in the field of computer science in general and artificial intelligence in particular. Genetic programming is a subfield of evolutionary algorithms that is in the center of our interest. A lot of work has been dedicated in the past towards modifying the basic algorithm of genetic programming in order to increase its efficiency. This article is a compiled version of a master thesis [3] that represents an attempt to contribute to this effort.

2 Genetic programming

The most general definition of GP is probably the simple statement that genetic programing is genetic algorithms applied to evolution of computer programs. The author of genetic programming is John Koza. His most important idea was [7] to code the program as a tree. Each program thus consists of number of nodes and terminals composed into a tree. As we will further see, this simple idea gives us

* The author was partially supported by the Grant Agency of the Czech Republic, Grant-No. 201/04/2102.

[3] This article represents a summary of ideas introduced in full detail in a master thesis [1] written under supervision of Dr. William Hsu at Computing and Information Sciences department of Kansas State University.

very powerful and at the same time natural representation of computer programs, especially suitable for GP systems. From this point, when we will mention the term "GP", we will refer to this variation of the more general notion of GP.

2.1 GP trees

Every individual in the GP is associated with an tree consisting of set of functions F and set of terminals T. Each function has some number of parameters (functions with zero parameters are terminals). Each node of the GP tree holds a function or a terminal. The number of subtrees that branch out from the given node corresponds with the arity of its corresponding function. Thus terminals are the leaves of the tree. The interpretation of such a tree is that each subtree is a subprogram that feeds its output data to its parent node. Each node N associated with a function F with arity a first evaluates each of its a subtrees, then feeds the vector of values $\boldsymbol{x}$ that the subtrees have produced to the function F. The output value of the tree rooted in the node N is then the value of the expression $F(\boldsymbol{x})$.

2.2 GP operators

Now that we have described in detail how to build programs we can have a look at the operators in GP. The power of the tree-like representation introduced by Koza is the fact that the operators are defined very naturally. Let us at first define the crossover operator. The goal of the crossover operator is to take two programs and produce two new that are recombination of their parents. This can be accomplished very easily by selecting two random nodes, each in one of the parents and swap the subtrees rooted in this two nodes. The mutation operator works in similar manner. It finds a random node in the tree and replaces the subtree rooted in this node with a new randomly generated one. Figure 1 depicts GP crossover and mutation operators in action.

3 Previous work

The ETGP is an example of a method that adds species formation control into the basic GA. As a consequence of the speciation, a kind of restricted mating mechanism is also incorporated into the algorithm, because mating of individuals from different subpopulations is prohibited. There is a number of modifications of the basic GA respectively GP algorithm, that are related to ETGP in a way that is interesting for us. Generally, we can classify these variations into four different classes: the methods using the speciation itself, the methods exploring the possibilities of niche formation, methods incorporating various forms of restricted mating and island model based systems. We would like to give a very brief introduction into these techniques.

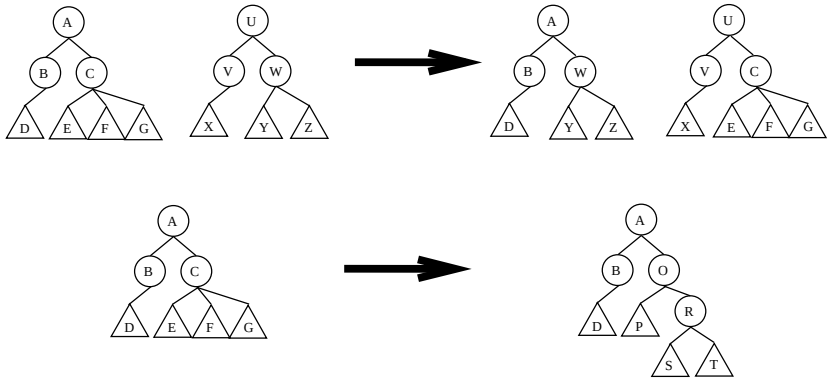


Fig. 1. The upper picture shows the crossover operator in action; the lower picture, the mutation operator in action.

3.1 Niche and speciation

Many fitness landscapes contain multiple local optima. This raises the problem of premature convergence in a suboptimal local optima, that can be fought by maintaining sufficient diversity in the population. The niche formation and speciation represent techniques inspired by this effort. 'Intuitively we may view a niche as an organism's job or role in an environment, and we can think of species as a class of organisms with common characteristics' [4]. In a general way we can think of the niche techniques as a softer version of speciation. For a deeper formal motivation behind niche and speciation techniques see [4].

3.2 Restricted mating

Generally, the idea of restricted mating is motivated by a following empirical observation. Crossover of two individuals that have large distance in their genotype in many problems usually leads to some significantly weaker individuals, that are in the literature referred to as *lethals*. There have been several attempts to eliminate these potentially wasteful crossovers by introducing various rules that restricted mating of arbitrary individuals. Again restricted mating techniques do not impose explicit speciation. One of the first examples of techniques, belonging to this class, can be found in Hollstein's [5] work.

3.3 Island model techniques

The island model (or multi-deme) [10] techniques incorporate multiple subpopulations into the basic GP algorithm which relates them to the ETGP. However, the subpopulations in the island models are not necessarily considered as separate species. They are rather generally considered as isolated concurrently evolving populations. An important feature of most of the island model systems,

that distinguishes them from the ETGP algorithm, is that the subpopulations occasionally exchange members in a process called migration. Originally the island model systems were motivated by parallel subpopulation evaluation, unlike the ETGP algorithm that is primarily looking for improved performance in undistributed configuration. However as several works have shown, island model systems are also capable of finding better individuals than the single population GAs (see for instance [3]). These results have often been attributed to increased diversity due to the migration of individuals among subpopulations. The ETGP algorithm represents an alternative approach to the introduction of diversity into the population, by separating clusters of individuals into isolated subpopulations and thus forcing evolution to follow multiple paths.

4 Motivation

An interesting observation of natural evolution is that it proceeds in a tree-like pattern. Scientists believe that at the beginning of the life on the planet Earth only single extremely simple life form existed. The members of this life form gradually evolved until a point when two distinct species have been segregated. During the millennia of terrestrial evolution the same process was repeated in all the once-existing species, leading to creation of a structure that is known as the *Tree of Life*. This work represents an attempt to recreate this process, although in very simplified manner. It is motivated by believe that it can have some positive influence on the performance of a GP system. Several other existing EA systems were already motivated by some biological foundations of species formation [2] such as adaptive landscape and shifting balance theory or a more recent theory of punctuated equilibria.

Naturally we did not stop only at formulation of the biological motivations. We tried to identify few reasons why the tree pattern of natural evolution can be beneficial to artificial genetic search. In the rest of this section, we will discuss the results of our attempts to answer the previous question.

One argument that stands behind speciation techniques that use genetic distance as the species formation criterion is the problem of lethals. We have already briefly discussed the theory of lethals in previous section, therefore let us proceed to the second observation that is related specifically in GP.

In the genetic algorithms, the crossover operator removes a fragment of genetic material of an individual and inserts it into another individual at the same position. This means that, with respect to the most common interpretations of the individuals genome, the semantic meaning of the exchanged fragment of genetic material is preserved. This is, however, not entirely the case in GP, because the subtree removed by crossover operator from the first individual is inserted in a random position in the tree of the second individual, which means that the context of the subprogram represented by the subtree is changed. Although it is true that the local semantics of the given subtree are preserved, because it still represents the same subprogram, its overall contribution to the output of the whole program is changed, because it depends on the programming context in

which it resides. From this observation one can conclude that the GP method to certain degree suffers from a sort of deceptive representation.

5 Evolutionary tree genetic programming

5.1 Basic algorithm

Our system includes a population P that is separated into number of subpopulations S_i. The ETGP algorithm starts with a single large subpopulation. After each cf generations, where cf is parameter that we will refer to as the *clustering frequency*, we divide each subpopulation into a new set of subpopulations. An important aspect of the system is that the subpopulations sizes are not constant throughout the run of the algorithm. The sizes of the subpopulations are constrained to be proportional to the average fitness of the individuals they contain.

The division of subpopulation S proceeds in two steps. At first we determine whether the given subpopulation S is large enough to undertake division. In this way we can prevent fragmentation of the population into large number of very small subpopulations. If the result of this decision is positive, we cluster the subpopulation S according to a metric we will discuss in the next section. We then insert each of these clusters into the new population as new subpopulation instead of the original subpopulation S.

The decision whether the given subpopulation is large enough to undertake division is controlled by a parameter we refer to as *branch factor* β. This decision process can be expressed in the following way:

$$\text{if } |S| \begin{cases} \leq |P| \cdot \beta \text{ do nothing} \\ > |P| \cdot \beta \text{ proceed with clustering} \end{cases}$$

where S is the subpopulation that we intend to divide and $|X|$ is the size of the (sub)population X.

5.2 Metric

The distance between each two members of the processed subpopulation has to be computed so we can supply this information to the clustering algorithm. Therefore, we have introduced a metric that computes the distance between any two GP trees. A simple requirement, that we would like our metric to satisfy comes from the observation that the closer a node is to the root of a GP tree, the more significant it is. Therefore, we want the distance between GP trees that differ in nodes close to the root to be larger than the distance between GP trees that differ only in the deeper nodes. We will refer to this requirement as the *fading property*. When developing the metric, we also faced the following problem. There are operators with different arity and there are also operators in which the sequence of their arguments does not affect the result, such as addition operation. Therefore, if we try to recursively define the distance between two trees to be some function of their roots and some function of the distances

between all their direct subtrees, it is not clear how to pair all the subtrees to compute these distances. One can argue that one possibility would be to use the pairing that would produce the smallest distance, however that would make the metric exponential in computation time. Therefore, we have employed an alternative method, that only finds the pair of subtrees that has the lowest distance (which has quadratic running time in the number of nodes) and then use this number to compute the resulting distance. Because of the scope of this article we will not investigate this topic in greater detail.

5.3 Clustering method

In this section we will discuss the clustering method that we have used in our technique. Nowadays, there are numerous clustering methods available for use. The most common of them are for example: *K-Means Clustering, Hierarchical Agglomerative Clustering (HAC), Self Organizing Maps [6] and Bayesian Clustering (Autoclass) [9].* We have decided to use HAC clustering, because it is the only method from the above list that does not require in its basic form a vectorial representation of the input entities.

We use the minimum variance as the agglomerative clustering criterion. There are some specific modifications that we have made to the original HAC algorithm. The HAC algorithm usually builds a full hierarchy of clusters, which means that we have all the clusterings ranging from n clusters down to one cluster, where n is the number of entities to be clustered. Our goal, however, is to find only two clusters. We would like both these two clusters to be as large as possible, because we do not want the subpopulation to be divided into one tiny one and another that is almost identical with the original one. To put it in more general terms, we are looking for the two of the most common code evolution trends in the given population. Because of the specific way how the HAC works, we cannot simply use the two clusters that HAC formed in the next-to-last step. It is very common that these two clusters significantly differ in size. Therefore, we have employed alternative strategy. Without going deeper with our description let us present the final algorithm:

```
find clusters A and B such that following requirement holds:
```

$$|\mathtt{A}| \geq |\mathtt{B}| \wedge \forall_{\mathtt{x} \in \mathtt{S}} : (|\mathtt{x}| \leq |\mathtt{B}| \vee \mathtt{x} \equiv \mathtt{A})$$

```
if (|A| + |B| ≥ α Σ_{x∈S} |x|) ∧ (|A|/|B| > β)
then
    let U := A and V := B
    For each cluster C ∈ S except cluster A and B do
        if dist(A,C) ≤ dist(B,C) then U := U ∪ C else V := V ∪ C
    return (U,V)
else
    continue with the next level of clustering
```

where S refers to the current set of clusters, parameter α was set to 0.7, and the parameter β was set to 0.4 in all the experiments.

6 Results

First we would like to very briefly discuss the methodology we have used in our experiments. We have conducted a large number of different tests. In order to make the situation more comprehensible, we have divided them into three series. Because of the scope of this article only the first two will be discussed. All the tests were repeated 100 times and the averages over these 100 runs are reported. In the very beginning we have generated 100 random numbers and we used these numbers throughout all the tests as the seeds for the random number generator in the 100 repetitions, thus ensuring that each test will have the same starting populations. Everything was implemented as an extension of *Evolutionary Computation in Java (ECJ)* package by Luke [8]. As benchmark problems the **11-multiplexer** and the **Artificial Ant (Santa Fe Trail)** problems were selected.

6.1 First experiment series

The goal of the first series of experiments was to find the optimal values of the branch factor and clustering frequency of the ETGP method. We have conducted two experiments. In both of them, one of these parameters was changed while the other was fixed to a predetermined value. Therefore, the interrelationship of these two parameters was not explored.

In the first experiment we have examined how the algorithm behaved when we have changed the branch factor β parameter. The range of this parameter was from zero to 0.4, and we have taken measurements with step of 0.05. For a more detailed description of branch factor, see section 4.1.

The second parameter we examined was clustering frequency. As the name of this parameter implies, this parameter determines how often will be the subpopulations clustered and new possibly larger number of populations created. The range in which this parameter was examined was [0 - 10].

Unfortunately, the variation of these two parameters did not produce a one point optimums or one sided trends. Following table summarizes the best values for each of these parameters in both problems:

	Clustering frequency	Branch factor
Artificial Ant	3, 4	0.2, 0.25
11-multiplexer	7, 10	0.2, 0.3, 0.4
Selected value	7	0.2

Tab. 1. Best performance values.

With respect to this table we have decided that a reasonable compromise will be to use a value of 0.2 as the Branch factor and 7 as a value of the Clustering Frequency in all subsequent experiments.

6.2 Second experiment series

The second series of experiments will finally provide us with a comparison of the ETGP algorithm with the basic version of the GP system. At first we will present graphs showing the performance of standard GP system on the two benchmark problems. Reader can see these graphs in Figure 2.

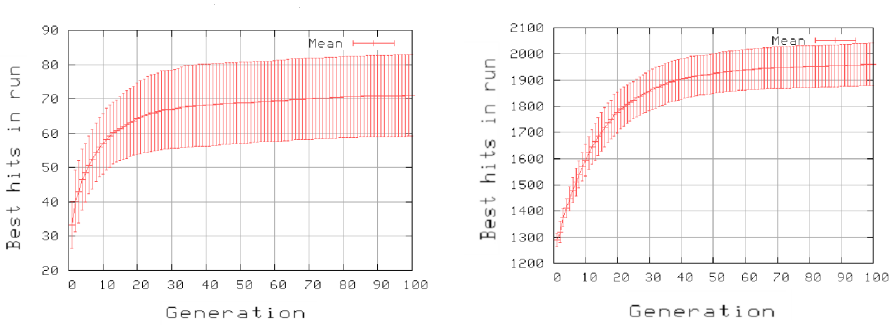


Fig. 2. Performance of the standard GP algorithm. The left plot depicts results from the Artificial Ant problem; the right, results from the 11-multiplexer problem.

In the second series of experiments, we also switch on two additional features of the ETGP algorithm that are not present in standard GP systems. First of these is mutation. This may be at first glance surprising, however for example in [7] empirical experiments showed that usually it is not beneficial to use mutation in GP systems. Since we loose variability in ETGP, however, we would like to explore the possibility to reduce this negative effect by the means of mutation. The second extension explored in the second series of experiments is the depth growth control. We did not mention this modification of ETGP algorithm so far. The reason is, that the motivation for introducing this parameter follows from constructive argument that did not fit into our motivation section because the scope of this article. Anyways, let us briefly describe what the depth control modification does. There are two parameter related to this extension. The first one *Starting depth S* defines what is the maximal allowed depth of any GP tree in the population in the beginning of the evolution. The second one called *End depth E* defines the maximal depth of GP trees in the population in the end of evolution. The actual maximal allowed depth m in generation g can be computed from these two parameters in following way $m = S + (E - S)\frac{g}{numGen}$ where *numGen* stands for the maximal number of generation in the given run. One can see that the previous expression is nothing else but a simple linear interpolation.

We will explore also the relationship between these two parameters; therefore, we have run tests with a variety of combinations of these two parameters. The *mutation ratio* is explored in the range of [0.05 - 0.4] with step 0.05. For the depth

growth control the *Start depth* parameter is varied in the range [3 - 17] with step of 2. The *End depth* parameter is kept at value 17 which is the default setting for maximal depth of a tree in ECJ library. The results of these experiments for the two benchmark problems can be seen on Figure 3.

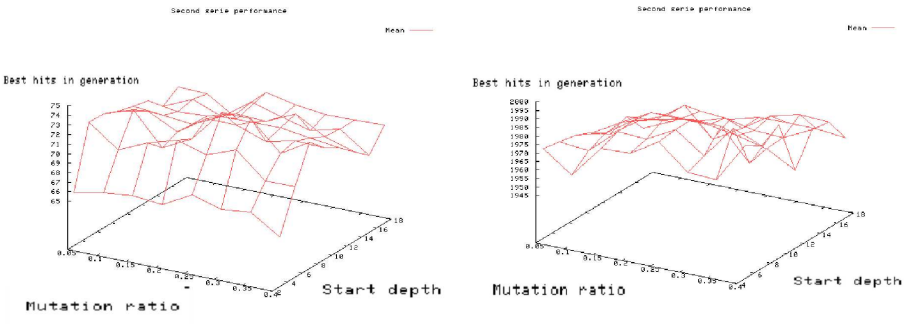


Fig. 3. Performance of the ETGP depending on the *mutation ratio* and *Start depth* parameter. The left plot depicts results from the Artificial Ant problem; the right, results from the 11-multiplexer problem.

Unfortunately the differences in the results are not too large and also no completely clear trends are present. The three dimensional graphs are also not very good at viewing small differences in the results and reveal the somehow hidden trends. Therefore, let us present some numerical values that we hope will clarify the results from these experiments. First of all let us state that there seems to be a clear trend for both benchmark problems that the best results are obtained with the *Start depth* parameter set to 7. The picture is not so clear with the *mutation ratio* however still some trend can be observed. The best values are generated when the *mutation ratio* is set between 0.25 and 0.35. This is a little bit surprising because for standard GA systems this is very large value for this parameter. The best result for the Artificial Ant problem were generated when the *Start depth* parameter was set to 7 and the *mutation ratio* had the value of 0.35. The best hits in run averaged over the 100 repeated runs reached the value of 74.45 compared to 71.01 in the case of standard GP system. In the case of the 11-multiplexer problem the best values were reached when the *Start depth* parameter was set to 7 and the *mutation ratio* had the value of 0.25 however almost the same value was also reached when the *mutation ratio* was set to 0.3 (the difference was only 0.05). For the 11-multiplexer problem the best performance of ETGP system was 1996.06 hits compared to 1960.5 generated by the standard GP system. The 95% confidence interval of the mean performance of ETGP for the 11-multiplexer case was [1982.9,2009.2] and for ant problem [71.95,76.95]. The 95% confidence interval of the mean performance of standard GP for the 11-multiplexer case was [1944.46,1976.5] and for ant

problem [68.65,73.37]. The p-value of the t-test testing the null hypothesis that the mean performance of ETGP system is greater than the mean of the GP system was 0.024033 for Artificial Ant problem and 0.000308 for 11-multiplexer problem. As a consequence of the low p-values we can with high confidence say that a real speedup, although very small was observed. Reader can examine these two best cases compared to the standard GP system in Figure 4.

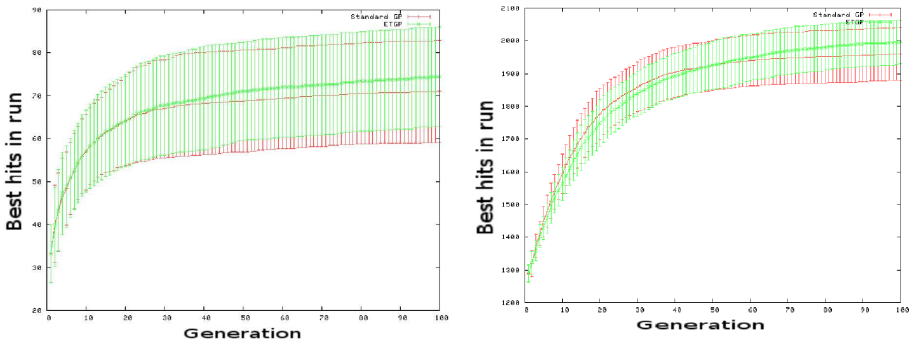


Fig. 4. The comparison of the standard GP system performance and the ETGP with the best parameter setting determined in the previous experiment. The left plot depicts results from the Artificial Ant problem; the right, results from the 11-multiplexer problem.

7 Conclusions

The focus of this work was to explore the possibility to simulate evolution of species as we can observe it in nature. We have extended the basic GP algorithm by incorporating a dynamic formation of species. The goal was to build a system in which the species evolution form a evolutionary tree just as in natural evolution. As the mechanism of separation of new species we have employed a clustering technique that divides the given subpopulation into two new one, according to the genetic similarity of the individuals.

The rest of our work then focused upon exploring various properties of the newly created system. Obviously the most interesting was the performance comparison with the standard GP algorithm. Unfortunately, as the previous section indicates no significant speedup of the evolution is observed. Because of the scope of this article, we had to skip the third experiment series, that provided us with some insights into the dynamics of species formation in ETGP system. Anyways, the overall conclusion following from these experiments is that we were not completely successful in simulating the natural properties of 'tree-like' evolution because we do not usually observe a longer coexistence of significantly different species; therefore, the evolution in our system is practically still following single

path of evolution as in the standard GP systems with single population. There are two possible solvents from this situation. The one negative may be that for the fitness functions and codings of standard problems solved in present GA community such as our benchmark problems, formation of significantly different species is unlikely, because of the nature of the fitness landscape of these problems. Therefore, the speedup that we can expect from ETGP for these classes of problems will be insignificant. The second possibility is that the formation of species is possible but we will have to develop more sophisticated mechanisms to achieve these properties. The ETGP algorithm itself has indeed many open ends that can be filled with different algorithms, such as the clustering method or metrics measuring GP tree distance. The scope of our work did not allowed exploration of wide variety of these possibilities. Let us also note that even though we did not observe desired behavior of the system, it still reached comparable results in non-distributed configuration when compared with related algorithm, particularly island model system.

Let us finish this article by discussing some future work. First of all we would like to examine other possibilities for clustering method. For example, the HAC algorithm can use various criterions determining which sub-clusters are to be joined together. As we have stated in the section 4.2, the design of metric over GP trees is also far from trivial and we had to employe several compromises at this point. Therefore, an exploration of this issue can also be interesting. To close the list of possible directions of future research we can consider the modification that would combine the ETGP algorithm with some niching technique.

References

1. Antolík J.: Evolutionary Tree Genetic Programming. Master's thesis, Computing and Information Scienses Department, Kansas State University, April 2004.
2. Baluja S.: A Massively Distributed Parallel Genetic Algorithm, 1992.
3. Fern'andez F., Tomassini M., Vanneschi L.: An Empirical Study of Multipopulation Genetic Programming. Genetic Programming and Evolvable Machines, **2**, 2003, 1469–1476.
4. Goldberg D.A.: Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley Publishing Company, Inc., 1989.
5. Hollstein R.B.: Artificial Genetic Adaptation in Computer Control Systems. PhD thesis, University of Michigan, 1971.
6. Kohonen T.: Self-Organizing Maps. Springer-Verlag Heidelberg, 1995.
7. Koza J.R.: Genetic Programming: On the Programming of Computers by the Means of Natural Selection. The MIT Press, 1992.
8. Luke S.: Evolutionary Computation in Java. `http://cs.gmu.edu/~eclab/projects/ecj/`, 2001.
9. Self M., Stutz J., Taylor W., Freeman D., Cheeseman P., Kelly J.: Autoclass: A Bayesian Classification System. In Proceedings of the Fifth International Conference on Machine Learning, 1988, 54–64.
10. Pettey C.B., Leuze M.R., Grefenstette J.J.: A Parallel Genetic Algorithm. In Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application, Lawrence Erlbaum Associates, Inc., 1987, 155–161.

Sequential monotonicity for restarting automata*

T. Jurdziński[1,2] and F. Otto[1]

[1] Fachbereich Mathematik/Informatik, Universität Kassel, D-34109 Kassel
{tju,otto}@theory.informatik.uni-kassel.de
[2] On leave from the Institute of Computer Science, Wrocław University, Poland

Abstract. As already 2-monotone R-automata accept NP-complete languages, we introduce a restricted variant of j-monotonicity for restarting automata, called *sequential j-monotonicity*. For restarting automata without auxiliary symbols, this restriction still yields infinite hierarchies. However, for restarting automata with auxiliary symbols, all degrees of sequential monotonicity collapse to the first level.

1 Introduction

Analysis by reduction is a technique used in linguistics to analyse sentences of natural languages. It consists of a stepwise simplification of a given sentence so that the (in)correctness of the sentence is not affected. Restarting automata were introduced by Jancar et. al. as a theoretical model for the analysis by reduction [5]. They can do a bottom-up syntactic analysis for natural and formal languages with sufficient generality and *'explicative power'* (cf. [15]). The notions developed during the study of restarting automata give a rich taxonomy of constraints for various models of analysers [12]. Already several programs used in Czech and German (corpus) linguistics are based on the idea of restarting automata [11], [15].

A restarting automaton, RLWW-automaton for short, is a device M that consists of a finite state control, a flexible tape containing a word delimited by the sentinels ¢ and \$, and a read/write window of a fixed size. This window moves along the tape by performing move-right and move-left instructions until the control decides (nondeterministically) that the content of the window should be rewritten by some shorter string, thereby shortening the tape. In general, the new string may contain some auxiliary (that is, non-input) symbols. After a rewrite, M can continue to move its window until it either halts and accepts, or halts and rejects, or restarts, that is, it places its window over the left end of the tape, reenters the initial state, and continues with the computation. Thus, each computation of M can be described through a sequence of cycles.

In addition to this general model, various restricted versions of the restarting automaton have been considered. First of all there is the RRWW-automaton, which does not use any move-left instructions, and there is the RWW-automaton, which is an RRWW-automaton that makes a restart immediately after a rewrite

* This research was supported by a grant from the Deutsche Forschungsgemeinschaft.

operation. Then there is the RLW-automaton, which only uses the letters of the input alphabet in its rewrite operations. A further restriction leads to the RL-automaton, where each rewrite operation is actually just a delete operation, that is, during each rewrite operation some letters from the content of the read/write window are simply deleted. Obviously the restrictions on the restart operation and the restrictions on the rewrite operation can be combined leading to the R(R)W-automaton and the R(R)-automaton.

Also a *monotonicity* property was introduced for the various types of restarting automata which is based on the idea that from one cycle to the next in a computation, the actual place where a rewrite operation is performed must not increase its distance from the *right* end of the tape. Monotone restarting automata essentially model bottom-up one-pass parsers. It was shown that the monotone RWW-, RRWW-, and RLWW-automata characterize the class CFL of context-free languages, and that the monotone version of the deterministic R(R)(W)(W)-automaton characterizes the class DCFL of deterministic context-free languages [6]. Thus, monotone restarting automata do not have sufficient expressive power to capture all issues of the analysis by reduction of natural languages. On the other hand, general RLWW-automata even accept some NP-complete languages [7], [10], which means that they cannot be implemented efficiently.

Therefore, the notion of *j-monotonicity* ($j \geq 1$) was introduced in [14], [16] as a generalization of the notion of monotonicity. It models the generalization from bottom-up one-pass parsers to bottom-up multi-pass parsers, and it allows to measure the level of non-monotonicity of a language. Further, this new notion seems to be much better suited to the real task of modelling the analysis by reduction. A restarting automaton is called *j-monotone* for an integer $j \geq 1$ if, for each of its computations, the corresponding sequence of cycles can be partitioned into at most j subsequences that are each monotone. It is shown in [16] that the expressive power of the j-monotone RRW-automaton increases with the value of the parameter j.

Unfortunately, it turned out that already 2-monotone R-automata accept NP-complete languages [9]. This fact means in particular that one should not expect that there exist efficient general algorithms for recognizing languages defined by j-monotone restarting automata. Therefore a different generalization of the notion of monotonicity is needed to capture the phenomena of natural languages.

Here we introduce such an alternative generalization of monotonicity, called *sequential j-monotonicity*. It differs from the 'classical' notion of j-monotonicity in that the $(i+1)$-st monotone subsequence only starts after the last cycle of the i-th monotone subsequence. In other words, it is not allowed that the j monotone subsequences interleave. This notion seems to be much more appropriate than j-monotonicity to model the generalization from one-pass parsers to multi-pass parsers.

After giving the necessary definitions in Section 2, we study the expressive power of sequentially j-monotone automata without auxiliary symbols. We show

in Section 3 that there exist strict infinite hierarchies with respect to the level of sequential monotonicity for all variants of nondeterministic restarting automata without auxiliary symbols. Further, we investigate restarting automata with auxiliary symbols in Section 4. We show that all degrees of sequential monotonicity collapse to the first level in this case, implying that such automata accept only context-free languages. On the one hand this fact ensures the existence of polynomial time recognition algorithms for all languages defined by sequentially j-monotone automata. On the other hand this result is somewhat 'frustrating,' as it shows that the expressive power of sequential j-monotonicity is rather limited. However, this result can be seen as another example of the 'expressibility' of the family of languages defined by context-free grammars, as sequential j-monotone restarting automata are in some aspect a much less restricted machine model than pushdown automata. In fact, sequential j-monotonicity may be expressed intuitively as a possibility to 'reuse' the pushdown store j times.

In Section 5 we consider sequential j-monotonicity for deterministic restarting automata. For automata that are not allowed to perform move-left transitions, all levels of sequential j-monotonicity collapse to the first level, which is an immediate consequence of the corresponding result for the 'classical' notion of j-monotonicity. On the other hand, we obtain an infinite hierarchy for deterministic restarting automata without auxiliary symbols that are allowed to perform move-left transitions.

For the details concerning the notions introduced we refer to [12], [13], and for complete proofs we refer to the technical report [8].

2 Definitions and notation

Let $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ be an RLWW-automaton. A *configuration* of M is a string $\alpha q \beta$, where $q \in Q$, and either $\alpha = \varepsilon$ and $\beta \in \{¢\} \cdot \Gamma^* \cdot \{\$\}$ or $\alpha \in \{¢\} \cdot \Gamma^*$ and $\beta \in \Gamma^* \cdot \{\$\}$; here q represents the current state, $\alpha\beta$ is the current content of the tape, and it is understood that the read/write window contains the first k symbols of β or all of β when $|\beta| \leq k$. A *restarting configuration* is of the form $q_0 ¢ w \$$, where $w \in \Gamma^*$; if $w \in \Sigma^*$, then $q_0 ¢ w \$$ is an *initial configuration*.

Any finite computation of M consists of certain phases. A phase, called a *cycle*, starts in a restarting configuration, the head moves along the tape performing MVR, MVL, and Rewrite operations until a Restart operation is performed and thus a new restarting configuration is reached. The execution of a cycle that starts from the restarting configuration $q_0 ¢ w \$$ and ends with the restarting configuration $q_0 ¢ w' \$$ is denoted as $w \vdash_M^c w'$. If no further Restart operation is performed, any finite computation necessarily finishes in a halting configuration – such a phase is called a *tail*. We require that M performs *exactly one* Rewrite operation during any cycle – thus each new phase starts on a shorter word than the previous one. During a tail at most one Rewrite operation may be executed.

An input *word* $w \in \Sigma^*$ *is accepted by* M, if there is a computation which, starting with the initial configuration $q_0 ¢ w \$$, finishes by executing an Accept instruction. By $L(M)$ we denote the language consisting of all words accepted by M; we say that M *accepts the language* $L(M)$.

Each cycle C contains a unique configuration $¢xqy\$$ in which a Rewrite instruction is applied. Then $|y\$|$ is the *right distance* of C, denoted by $D_r(C)$. We say that a *sequence of cycles* $S = (C_1, C_2, \cdots, C_n)$ is *monotone* (or *right-monotone*) if $D_r(C_1) \geq D_r(C_2) \geq \ldots \geq D_r(C_n)$. A *computation* is *monotone* if the corresponding sequence of cycles is monotone. Observe that the tail of the computation does not play any role here. An RLWW-automaton M is called *monotone* if all its computations that start with an initial configuration are monotone. The prefix mon- will be used to denote the corresponding classes of restarting automata.

Let j be a positive integer. A sequence of cycles $(C_1, C_2, \ldots, C_n)$ of the computation of a restarting automaton is called *sequentially j-monotone* if there exist indices $0 = p_0 < p_1 < \cdots < p_j = n$ such that, for each $i = 1, \ldots, j$, the subsequence $(C_{p_{i-1}+1}, C_{p_{i-1}+2}, \ldots, C_{p_i})$ is monotone. Observe that here the j monotone subsequences follow sequentially one after the other, while for the general definition of j-monotonicity it is allowed that the j subsequences are interleaved [16]. A restarting automaton is called *sequentially j-monotone* if all its computations that start with an initial configuration are sequentially j-monotone. We will use the prefix j-s-mon- to denote the classes of sequentially j-monotone restarting automata.

3 Restarting automata without auxiliary symbols

In this section we show that for nondeterministic restarting automata without auxiliary symbols, sequential $(j+1)$-monotonicity is more expressive than sequential j-monotonicity. For proving this result we present a family of example languages L_j $(j \geq 1)$. For defining these languages we need a function $\vartheta : (\bigcup_{p \geq 1} \mathbb{N}_+^{2p}) \to \{0, 1\}$ that is defined inductively as follows:

$$\text{For } n, m \in \mathbb{N}_+,\ \vartheta(n,m) := \left\{ \begin{array}{l} 1 \text{ if } n = m \\ 0 \text{ if } n \neq m \end{array} \right\}, \text{ and for } n, m, r_1, \ldots, r_{2p} \in \mathbb{N}_+,$$

$$\vartheta(n, m, r_1, \ldots, r_{2p}) := \begin{cases} 1 & \text{if } n = m \text{ and } \vartheta(r_1, \ldots, r_{2p}) = 1 \\ & \text{or } n > 2m \text{ and } \vartheta(r_1, \ldots, r_{2p}) = 0, \\ 0 & \text{otherwise.} \end{cases}$$

Then, for each integer $j \geq 1$, the language L_j on $\Sigma := \{a, b\}$ is defined as

$$L_j := \{a^{n_1} b^{m_1} \ldots a^{n_j} b^{m_j} \mid \vartheta(n_1, m_1, \ldots, n_j, m_j) = 1\}.$$

The following observations on ϑ will be useful.

Lemma 1. *Let $n_1, \ldots, n_j, m_1, \ldots, m_j \in \mathbb{N}_+$.*

(a) *If $n_i \neq m_i$ and $n_i \leq 2m_i$, then $\vartheta(n_i, m_i, \ldots, n_j, m_j) = 0$.*

(b) *If $\vartheta(n_i, m_i, \ldots, n_j, m_j) = 0$ for some i satisfying $0 < i \leq j$, and if $n_l \leq 2m_l$ for each $l < i$, then $\vartheta(n_1, m_1, \ldots, n_j, m_j) = 0$.*

Proposition 1. *For each $j \geq 1$, $L_j \in \mathcal{L}(j\text{-s-mon-RR})$.*

Proof. We show that L_j is accepted by an j-s-mon-RR-automaton M that works as follows. If the word on the tape does not belong to $(a^+b^+)^j$, it is rejected. For a word of the form $(a^+b^+)^j$, we call the factors from a^+b^+ *blocks*. Let $a^{n_1}b^{m_1}\dots a^{n_j}b^{m_j}$ be the given input. In each cycle M chooses nondeterministically a block in which a rewrite step is to be executed. Assume that the i-th block is chosen. Then M decides nondeterministically which of the conditions $n_i = m_i$ or $n_i > 2m_i$ should be checked for that block. In the former case it removes a factor ab from that block, in the latter case it removes a factor a^2b from that block, provided that $n_i \geq 3$ and $m_i \geq 2$ hold. After executing the rewrite step M checks the correctness of the choices made. They are correct if

- $i = j$, and the rewrite step $ab \to \varepsilon$ was executed, or
- for each $i < l \leq j$, the l-th block has the form $a^{n_l}b^{m_l}$ such that $n_l, m_l \in \mathbb{N}_+$ satisfy $m_l = 1$ or $n_l \leq 2$, $\vartheta(n_{i+1}, m_{i+1}, \dots, n_j, m_j) = 1$ and the rewrite step $ab \to \varepsilon$ was chosen, or $\vartheta(n_{i+1}, m_{i+1}, \dots, n_j, m_j) = 0$ and the rewrite step $a^2b \to \varepsilon$ was chosen.

Observe that M is able to verify the condition $n_l > 2m_l$ or $n_l = m_l$ for each block $a^{n_l}b^{m_l}$ satisfying $m_l = 1$ or $n_l \leq 2$. Hence, it can determine the value of $\vartheta(n_{i+1}, m_{i+1}, \dots, n_j, m_j)$ in its finite control.

On recognizing that an incorrect choice was made, M rejects immediately. Otherwise, the tape content is reduced to satisfying $m_i = 1$ or $n_i \leq 2$ for each $i \in \{1, \dots, j\}$. Then M accepts if and only if $\vartheta(n_1, m_1, \dots, n_j, m_j) = 1$ holds. It follows easily that $L(M) = L_j$ holds. As M processes the input block by block from right to left, each computation of M consists of (at most) j monotone sequences. Hence, M is sequentially j-monotone. □

On the other hand, we have the following negative result.

Proposition 2. *For each $j \geq 2$, $L_j \notin \mathcal{L}((j-1)$-s-mon-RLW$)$.*

Proof. Let $M = (Q, \Sigma, \Sigma, ¢, \$, q_0, k, \delta)$ be an RLW-automaton for L_j, let p be the constant for M from the Pumping Lemma (see, e.g., [14]), and let $r := p!$. We say that the i-th block of a word of the form $a^{n_1}b^{m_1}\dots a^{n_j}b^{m_j}$ is *short* if $n_i < 3r$ or $m_i < 3r$. From the Pumping Lemma we immediately obtain the following claim.

Claim 1. Let $a^{n_1}b^{m_1}\dots a^{n_j}b^{m_j} \vdash_M^c a^{n'_1}b^{m'_1}\dots a^{n'_j}b^{m'_j}$ be a cycle of a computation of the RLW-automaton M. Then, for each $\alpha_1, \dots, \alpha_j, \beta_1, \dots, \beta_j \in \mathbb{N}$, the cycle

$$a^{n_1+\alpha_1 r}b^{m_1+\beta_1 r}\dots a^{n_j+\alpha_j r}b^{m_j+\beta_j r} \vdash_M^c a^{n'_1+\alpha_1 r}b^{m'_1+\beta_1 r}\dots a^{n'_j+\alpha_j r}b^{m'_j+\beta_j r}$$

is part of a computation of M, provided that $\alpha_i = \beta_i = 0$ for each i for which the i-th block in $a^{n_1}b^{m_1}\dots a^{n_j}b^{m_j}$ is short. □

A word $(a^nb^n)^j$ is called a *basic input* if n is a multiple of r and $n \geq 3r$. Note that each basic input belongs to L_j, and that none of its blocks is short.

Claim 2. Let $a^n b^n \dots a^n b^n \vdash_M^{c^*} a^{n'_1} b^{m'_1} \dots a^{n'_j} b^{m'_j}$ be an initial segment of an accepting computation of M on the basic input $w = (a^n b^n)^j \in L_j$, and let $i \in \{1, \dots, j\}$. If the i-th block is not short in any of the configurations during the above computation, then

(a) during this computation no rewrite step is applied to any of the first $i-1$ blocks, that is, $n'_t = m'_t = n$ for each $t < i$;
(b) $n'_i = m'_i \geq 3r$, and each rewrite step that is applied during this computation to the i-th block is of the form $a^s b^s \to a^{s-t} b^{s-t}$ for some $k > s \geq t > 0$.

Proof of Claim 2. This claim is also proved by contradiction. So assume that it is not true for some $n \geq 3r$ that is a multiple of r and some $i \in \{1, \dots, j\}$, and let $a^{n'_1} b^{m'_1} \dots a^{n'_j} b^{m'_j}$ be the first configuration that contradicts this claim, that is,

- $n'_l \neq n$ or $m'_l \neq n$ for some $l < i$, or
- $n'_i \neq m'_i$, or
- some rewrite step that changes some of the first i blocks is *not* of the form $a^s b^s \to a^{s-t} b^{s-t}$ for any $k > s \geq t > 0$.

Observe that the third condition implies that also one of the first two conditions is satisfied. Hence, we concentrate only on the first two conditions.

According to our choice of $a^{n'_1} b^{m'_1} \dots a^{n'_j} b^{m'_j}$, the computation considered ends with a cycle of the form $a^{n''_1} b^{m''_1} \dots a^{n''_j} b^{m''_j} \vdash_M^c a^{n'_1} b^{m'_1} \dots a^{n'_j} b^{m'_j}$, where $n''_t = m''_t = n$ for all $t \in \{1, \dots, i-1\}$ and $n''_i = m''_i \geq 3r$. By analysing this cycle in detail, we obtain the intended contradiction. □

It remains to derive the statement of Proposition 2 from the two claims above. Let $w = (a^n b^n)^j \in L_j$ be a basic input. By Claim 1, M cannot accept as long as the first block is not short. On the other hand, Claim 2 implies that the first rewrite step that changes the i-th block is possible only after a configuration has been reached in which the $(i+1)$-st block is short. Thus, each accepting computation first shortens the j-th block, then the $(j-1)$-st block, and so on. On the other hand, the first rewrite step that changes the i-th block is executed in the center of the block (see Claim 2). Hence, its right distance is larger than the right distance of the previous rewrite step applied to the blocks $(i+1), \dots, j$. Thus, this rewrite step initiates a new monotone sequence, that is, we obtain at least j sequentially monotone sequences. □

In order to obtain corresponding results for R(W)-automata, a more involved variant $\tilde{L}_j$ of the language L_j ($j \geq 1$) is needed (see [8] for the details). For these languages the following results can be established.

Proposition 3. *For each $j \geq 2$, $\tilde{L}_j \in \mathcal{L}(j\text{-s-mon-R}) \setminus \mathcal{L}((j-1)\text{-s-mon-RLW})$.*

As a consequence of these propositions, we obtain the following theorem.

Theorem 1. *For each $j \in \mathbb{N}_+$ and each $X \in \{\mathsf{R}, \mathsf{RW}, \mathsf{RR}, \mathsf{RRW}, \mathsf{RL}, \mathsf{RLW}\}$,*

(a) $\mathcal{L}(j\text{-s-mon-X}) \subsetneq \mathcal{L}((j+1)\text{-s-mon-X})$,
(b) $\mathcal{L}((j+1)\text{-s-mon-R}) \setminus \mathcal{L}(j\text{-s-mon-RLW}) \neq \emptyset$.

4 Restarting automata with auxiliary symbols

It is known that the classes of languages $\mathcal{L}(\mathsf{mon\text{-}R(R)WW})$ and $\mathcal{L}(\mathsf{mon\text{-}RLWW})$ coincide with the class CFL of context-free languages [6]. Here we will show that, for all $j \geq 2$, this characterization extends to the classes of languages that are accepted by sequentially j-monotone restarting automata with auxiliary symbols. For establishing this result we will use the following technical result on pushdown automata.

Let $P = (Q, \Sigma, \Gamma, \delta, q_0, \#, F)$ be a pushdown automaton (PDA) with input alphabet Σ and stack alphabet $\Gamma \cup \{\#\}$, where $\#$ denotes the bottom marker of the pushdown store, Q is the set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, and δ is the transition relation. For a word $u \in \Sigma^*$,

$$\mathsf{SC}_P(u) := \{\, w \in \Gamma^* \mid \exists\, q \in F : (q_0, u, \#) \vdash_P^* (q, \varepsilon, \#w) \,\}$$

is the *language of final stack contents* that P can generate on input u. For a language $L \subseteq \Sigma^*$, $\mathsf{SC}_P(L) := \bigcup_{u \in L} \mathsf{SC}_P(u)$. It is well-known that the language $\mathsf{SC}_P(L)$ is regular for each regular language L [4].

Lemma 2. *Let P be a pushdown automaton with input alphabet Σ and pushdown alphabet $\Gamma \cup \{\#\}$, and let S be a subset of $\mathsf{SC}_P(\Sigma^*)$. If S is context-free, then so is the language $\mathsf{SC}_P^{-1}(S) := \{\, u \in \Sigma^* \mid \mathsf{SC}_P(u) \cap S \neq \emptyset \,\}$.*

Proof. Let $\bar{\Gamma}$ be a new alphabet in one-to-one correspondence to Γ such that Γ and $\bar{\Gamma}$ are disjoint. Further, let $\bar{\#}$ be another new symbol, and let R be the finite string-rewriting system $R := \{\, b\bar{b} \to \varepsilon \mid b \in \Gamma \cup \{\#\} \,\}$ on $\Delta := \Gamma \cup \bar{\Gamma} \cup \{\#, \bar{\#}\}$. Then R is a special system that is confluent (see, e.g., [2]). For $w \in \Delta^*$, $\Delta_R^*(w)$ is the set of all descendants of w with respect to the reduction relation induced by R, that is, $\Delta_R^*(w) := \{\, z \in \Delta^* \mid w \to_R^* z \,\}$, and $\Delta_R^*(T) := \bigcup_{w \in T} \Delta_R^*(w)$ for each subset $T \subseteq \Delta^*$. It is well-known that the set $\Delta_R^*(T)$ is regular, whenever T is a regular language.

In [3] it is shown that a finite-state transducer B with input alphabet Σ and output alphabet Δ can be constructed from the PDA P such that the following equality holds for each word $u \in \Sigma^*$:

$$\Delta_R^*(B(u)) \cap \bar{\#}\# \cdot \Gamma^* = \bar{\#}\# \cdot \mathsf{SC}_P(u).$$

Now let S be a subset of $\mathsf{SC}_P(\Sigma^*)$. Then a word $u \in \Sigma^*$ belongs to the set $\mathsf{SC}_P^{-1}(S)$ if and only if $\Delta_R^*(B(u)) \cap \bar{\#}\# \cdot S \neq \emptyset$ holds.

Let $\nabla_R^*(w)$ denote the set of ancestors of w with respect to the reduction relation induced by R, that is, $\nabla_R^*(w) := \{\, x \in \Delta^* \mid x \to_R^* w \,\}$, and for $T \subseteq \Delta^*$, $\nabla_R^*(T) := \bigcup_{w \in T} \nabla_R^*(w)$. Then we obtain the following equality:

$$\mathsf{SC}_P^{-1}(S) = B^{-1}(\nabla_R^*(\bar{\#}\# \cdot S)).$$

Now if S is a context-free language, then so is the language $\bar{\#}\# \cdot S$. From the form of the rules of R we see immediately that then $\nabla_R^*(\bar{\#}\# \cdot S)$ is context-free,

too, which implies that $B^{-1}(\nabla_R^*(\bar{\#}\# \cdot S))$ is context-free, as the class of context-free languages is closed under (inverse) finite transductions [1]. Thus, we see that $\mathsf{SC}_P^{-1}(S)$ is context-free. □

Let $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ be an RRWW-automaton. With M we associate the language

$$L_{\mathrm{mon}}(M) := \{\, w \in \Gamma^* \mid M \text{ accepts } w \text{ by a monotone computation} \,\}$$

and the mapping $f_M : \Gamma^* \to \mathfrak{P}(\Gamma^*)$ that is defined as follows:

$$f_M(w) := \{\, y \in \Gamma^* \mid \text{There exists a monotone sequence of cycles of } M \text{ that starts from the restarting configuration } q_0¢w\$ \text{ and ends with the restarting configuration } q_0¢y\$ \,\}.$$

Observe that the language $f_M(\Gamma^*)$ is simply the set Γ^*, as from each restarting configuration a monotone computation of M originates, which, however, may consist of a single cycle only, or which may even consist of no cycle at all.

Lemma 3. *Let $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ be an* RRWW*-automaton. Then the following statements hold:*

(a) *The language $L_{\mathrm{mon}}(M)$ is context-free.*

(b) *There exists a pushdown automaton P_M such that, for each word $w \in \Gamma^*$, $\mathsf{SC}_{P_M}(w) = f_M(w)$ holds.*

Proof. (a) In [6] it is shown that the language $L(M)$ that is accepted by a monotone RRWW-automaton M is necessarily context-free by presenting a simulation of M by a PDA P'_M. Even if the RRWW-automaton M is not monotone, then, given a word $w \in \Gamma^*$, this PDA simulates the monotone computations of M that originate from the restarting configuration $q_0¢w\$$. Thus, this PDA accepts the language $L_{\mathrm{mon}}(M)$, which means that this language is indeed context-free.

(b) The PDA P'_M above simulates the monotone initial parts of all computations of M. We modify P'_M in such a way that, each time it starts the simulation of a cycle of M, it may decide (nondeterministically) to abort the simulation process. In this case it pushes the remaining suffix of the input onto the pushdown store, and halts and accepts. Then the resulting PDA P_M satisfies the requirement that, for each word $w \in \Gamma^*$, $\mathsf{SC}_{P_M}(w) = f_M(w)$ holds. □

Let M be an RRWW-automaton with tape alphabet Γ. If we denote by $L_{\mathrm{mon}}^{(j)}(M)$ the language

$$L_{\mathrm{mon}}^{(j)}(M) := \{\, y \in \Gamma^* \mid M \text{ accepts } y \text{ by a sequentially } j\text{-mon. computation} \,\},$$

where $j \geq 1$, then we see that, for each word $w \in \Gamma^*$ and each number $j \geq 2$,

$$w \in L_{\mathrm{mon}}^{(j)}(M) \text{ if and only if } f_M(w) \cap L_{\mathrm{mon}}^{(j-1)}(M) \neq \emptyset.$$

Now the main result of this section is an immediate consequence of the following technical result, which can be proved by induction on j using the results above.

Theorem 2. *$L^{(j)}_{\mathrm{mon}}(M)$ is a context-free language for each* RRWW*-automaton M and each number $j \geq 1$.*

For each RLWW-automaton, there exists an RRWW-automaton that accepts the same language and that executes exactly the same cycles ([17] Proposition 1). If an RRWW-automaton $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ is sequentially j-monotone, then $L(M) = L^{(j)}_{\mathrm{mon}}(M) \cap \Sigma^*$. By the result above this yields that $L(M)$ is context-free. As each context-free language is accepted by a monotone RWW-automaton [6], we obtain the following characterization.

Corollary 1. *$\mathcal{L}(j$-s-mon-R(R)WW$) = \mathcal{L}(j$-s-mon-RLWW$) =$ CFL *for all $j \geq 1$.*

5 Deterministic restarting automata

It is known that all levels of j-monotonicity for deterministic R(R)(W)(W)-automata collapse to DCFL [17]. As each sequentially j-monotone computation is also j-monotone, we obtain the following result.

Corollary 2. *For each* X $\in \{$R, RR, RW, RRW, RWW, RRWW$\}$ *and for each $j \in \mathbb{N}_+$, $\mathcal{L}($*det-j-s-mon-X$) =$* DCFL.

Monotone deterministic RL-automata recognize some languages that are not in DCFL [14]. In fact, using the example languages L_j from Section 3 it can be shown that also the hierarchies with respect to the level of sequential monotonicity do not collapse for deterministic RL(W)-automata.

Proposition 4. *$L_j \in \mathcal{L}($*det-j-s-mon-RL$) \setminus \mathcal{L}($*det-$(j-1)$-s-mon-RLW$)$ $(j \geq 2)$.*

This proposition implies the following hierarchies.

Theorem 3. *For each $j \in \mathbb{N}_+$ and each* X $\in \{$RL, RLW$\}$,*

$$\mathcal{L}(\text{det-}j\text{-s-mon-X}) \subsetneq \mathcal{L}(\text{det-}(j+1)\text{-s-mon-X}).$$

6 Concluding remarks

We have seen that sequential j-monotonicity yields interesting extensions of those language classes that are defined by monotone restarting automata without auxiliary symbols without loosing the efficiency of the solution to the membership problem. On the other hand, it turned out that all degrees of sequential monotonicity collapse to the first level for restarting automata with auxiliary symbols. This fact may be of interest in its own right as it gives a much less restricted machine model for CFL as the pushdown automaton. On the other hand, this result says that the expressibility of sequentially j-monotone restarting automata is not sufficient to cover many important aspects of the analysis by reduction. Thus, other generalizations of monotonicity are needed.

An interesting open problem is whether sequential j-monotonicity gives an infinite hierarchy for deterministic RLWW-automata.

References

1. Berstel J.: Transductions and Context-Free Languages. Teubner, Stuttgart, 1979.
2. Book R.V., Otto F.: String-Rewriting Systems. Springer, New York, 1993.
3. Cremanns R., Otto F.: The Language of Final Stack Contents of a Pushdown Automaton is Effectively Regular. Mathematische Schriften Kassel 11/93, Universität Kassel, 1993.
4. Greibach S.: A Note on Pushdown Store Automata and Regular Systems. *Proc. American Math. Soc.*, **18**, 1967, 263–268.
5. Jančar P., Mráz F., Plátek M., Vogel J.: Restarting Automata. In: H. Reichel (ed.), FCT'95, Proc., Springer, Berlin, *LNCS* **965**, 1995, 283–292.
6. Jančar P., Mráz F., Plátek M., Vogel J.: On Monotonic Automata with a Restart Operation. *J. Autom. Lang. and Comb.*, **4**, 1999, 287–311.
7. Jurdziński T., Loryś K., Niemann G., Otto F.: Some Results on RRW- and RRWW-Automata and Their Relationship to the Class of Growing Context-Sensitive Languages. Mathematische Schriften Kassel 14/01, Universität Kassel, 2001. Also: *J. Autom. Lang. and Comb.*, to appear.
8. Jurdziński T., Otto F.: Sequential Monotonicity for Restarting Automata. Mathematische Schriften Kassel 10/04, Universität Kassel, 2004.
9. Jurdziński T., Otto F., Mráz F., Plátek M.: On the Complexity of 2-Monotone Restarting Automata. In: C.S. Calude, E. Calude and M.J. Dinneen (eds.), Developments in Language Theory, Proc. DLT, Springer, Berlin, *LNCS* **3340**, 2004, 237–248.
10. Niemann G., Otto F.: Further Results on Restarting Automata. In: M. Ito and T. Imaoka (eds.), Words, Languages and Combinatorics III, Proc., World Scientific, Singapore, 2003, 352–369.
11. Oliva K., Květoň P., Ondruška R.: The Computational Complexity of Rule-Based Part-of-Speech Tagging. In: V. Matousek and P. Mautner (eds.), TSD 2003, Proc., Springer, Berlin, *LNCS*, **2807**, 2003, 82–89. .
12. Otto F.: Restarting Automata and Their Relations to the Chomsky Hierarchy. In: Z. Esik and Z. Fülöp (eds.), Developments in Language Theory, Proc., Springer, Berlin, *LNCS*, **2710**, 2003, 55–74.
13. Otto F.: Restarting Automata – Notes for a Course at the 3-rd International PhD School in Formal Languages and Applications. Mathematische Schriften Kassel 6/04, Universität Kassel, 2004.
14. Plátek M.: Two-Way Restarting Automata and j-Monotonicity. In: L. Pacholski and P. Ružička (eds.), Theory and Practice of Informatics, Proc. SOFSEM 2001, Springer, Berlin, *LNCS*, **2234**, 2001, 316–325.
15. Plátek M., Lopatková M., Oliva K.: Restarting Automata: Motivations and Applications. In: M. Holzer (ed.), Workshop 'Petrinetze' and 13. Theorietag 'Formale Sprachen und Automaten', Proc., Institut für Informatik, Technische Universität München, 2003, 90–96.
16. Plátek M., Mráz F.: Degrees of (Non)monotonicity of RRW-Automata. In: J. Dassow and D. Wotschke (eds.), Preproceedings of the 3rd Workshop on Descriptional Complexity of Automata, Grammars and Related Structures, Fakultät für Informatik, Universität Magdeburg, Report No. 16, 2001, 159–165.
17. Plátek M., Otto F., Mráz F., Jurdziński T.: Restarting Automata and Variants of j-Monotonicity. Mathematische Schriften Kassel 9/03, Universität Kassel, 2003.

Binární faktorová analýza genetickým algoritmem

Aleš Keprt and Václav Snášel

Katedra informatiky, FEI, VŠB - Technická Univerzita Ostrava
17. listopadu 15, 708 33 Ostrava-Poruba, Česká republika
Ales@Keprt.cz, Vaclav.Snasel@vsb.cz

Abstrakt BFA je analýza binárních dat na principu redukce dimenze binárního prostoru, umožňuje nám tak najít v binárních datech skryté vztahy. Řešení BFA je však algoritmicky velmi obtížně zvládnutelný problém. Jedním z možných řešení může být použití genetického algoritmu (GA), což je v současnosti také zřejmě nejlepší známá metoda.

Klíčová slova: Genetický algoritmus, analýza binárních dat, information retrieval

1 Úvod

Binární data jsou jedním ze základních kamenů počítačů a v počátcích výpočetní techniky se i informace zcela odlišné povahy násilně kódovaly a uchovávaly v čisté binární podobě, obvykle především z důvodů technických omezení. Ačkoliv na fyzické úrovni jsou binární data stále klíčovým prvkem, na logické úrovni postupem času jasně převládlo v uchování i zpracování informací jejich přirozené lidské chápání, v jeho čistě nebinární formě. Dnes je již stále běžnější i používání fuzzy technologií, které umožňují pracovat až s neurčitostí v podobě vágnosti, která je z našeho pohledu jakýmsi protipólem nativně binárního přístupu k informacím.

Datová analýza a získávání důležitých, leč skrytých informací z datových zdrojů není téma vůbec nové a bylo obsahem statistiky již dlouho před tím, než se zformovala informatika jako samostatný obor. I v dnešní době je však datová analýza stále velmi aktuálním tématem, z hlediska informatiky zejména s ohledem na rychlý rozvoj internetu, nebo obecně jako důsledek vysokého důrazu na ekonomickou stránku života a hospodářský úspěch společnosti jako celku, i jejích dílčích částí.

Na tomto místě je zajímavé si všimnout, že s velkým rozvojem různých druhů datové analýzy se potichu vytratil zájem nebo alespoň soustředění na data binární povahy. Ačkoliv je zřejmé, že celá řada veličin skutečného světa má matematicky řečeno reálnou, dokonce spojitou povahu, i binární data se vyskytují. I když je lze často analyzovat i běžnými metodami založenými na lineární algebře, aproximaci nebo hledání extrémů funkcí, výsledky běžných metod v případě binárních dat nejsou vůbec uspokojivé. Binární data mají totiž natolik specifickou

povahu, že je pro ně vhodné mít také specifické analytické nástroje. Příkladem takového nástroje mohou být konceptuální svazy. Ty jsou typickým příkladem metody určené specificky pro binární data a umožňují jejich analýzu z hlediska hierarchie. Zároveň dokládají, jak opožděně se objevuje seriózní zájem o zkoumání binárních dat.

Naším cílem je pak zkoumání binární faktorové analýzy (BFA) – další specificky binární metody, jejíž účel a smysl však leží v jiném typu analýzy než u dnes již rozšířených konceptuálních svazů. Jedná se o nelineární analýzu čistě binárních dat, kde z principu věci není možno použít ani znalosti lineární algebry, ani matematické (funkcionální) analýzy. Experimentálně bylo zjištěno, že běžné nebinární metody, pouze doplněné o následné převedení výsledků do binární podoby, dávají velmi neuspokojivé výsledky. Proto bylo postupně navrženo a realizováno několik nových metod, které pracují s booleovou aritmetikou na binární bázi. Byly to postupně neuronové sítě, kombinatorické hledání řešení a převod problému BFA na problém stavění konceptuálních svazů.

Obsahem tohoto textu je popis řešení BFA s využitím modifikovaného genetického algoritmu (GA). Nejprve je uveden pro srovnání stručný úvod ke klasické faktorové analýze. Následuje definice problému BFA a popis GA-BFA metody. V závěru jsou uvedeny výsledky srovnání několika metod BFA v aplikaci na analýzu textových dokumentů.

2 Klasická faktorová analýza

BFA vychází z klasické faktorové analýzy (FA)[1]. Ta vychází z předpokladu, že jevy, které lze pozorovat (a zaznamenat do databáze), jsou jen důsledkem skrytých **faktorů**, tedy jevů stojících v pozadí. Každá měřitelná a zaznamenatelná veličina (označována jako **proměnná**) je pak ve vyjádření FA lineární kombinací faktorů. Toto pojetí a základní impulz k rozvoji FA dala psychologie. Ta se totiž snaží na základě pozorování vnějšího chování jedinců určit, jakého jsou charakteru, jaké mají duševní poruchy apod. Na příkladu psychologie je idea FA jasná. Matematicky jde o snahu vyjádřit datovou matici X pomocí součinu dvou (mnohem) menších matic $\mathsf{F} \cdot \mathsf{A}$.

$$\mathsf{X}_{[n\times p]} \approx \mathsf{F}_{[n\times m]} \cdot \mathsf{A}_{[m\times p]}$$

Sloupce matice X reprezentují p **proměnných** (variables), řádky reprezentují n **případů** (cases). Matice F, nazývaná **factor scores**, vyjadřuje stejná fakta jako matice X, ovšem pomocí faktorů, kterých má být mnohem méně než proměnných: $m << p$. Matice A vyjadřuje vztah mezi proměnnými a faktory. FA tedy provádí redukci dimenze vektorového prostoru (z p na m), opírá se přitom o poznatky statistiky a lineární algebry. Předpokladem je normální rozložení. Začínáme výpočtem korelační matice, dál existuje několik numerických metod,

[1] Slovo "klasická" zde slouží jen k označení originální nebinární verze faktorové analýzy.

které lze použít. Celkově je výpočet natolik náročný, že v minulosti nebyl bez použití počítačů prakticky realizovatelný.

Úspěch faktorové analýzy posuzujeme dle rozdílu součinu $\mathsf{F}\cdot\mathsf{A}$ oproti původní matici X. Protože obecně nelze rozložit jakoukoliv matici na součin dvou menších matic, ve výše uvedeném vzorci je použit symbol $\approx$ jako vyjádření přibližné rovnosti.

3 Binární faktorová analýza

BFA má symbolicky stejné zadání jako klasická FA. Máme binární datovou matici X a naším cílem je vyjádřit ji pomocí součinu dvou menších binárních matic $\mathsf{F}\cdot\mathsf{A}$.

$$\mathsf{X} \approx \mathsf{F} \odot \mathsf{A}$$

Řádky F a A musejí být nenulové. $\odot$ je binární součin matic – odpovídá klasickému součinu, ale s použitím booleovské aritmetiky. Ta je přirozeným nástrojem pro práci s binárními hodnotami, od klasického násobení binárních matic se liší jen při sčítání, kde platí: $1 + 1 = 1$. Buňky binárních matic nazýváme **bity**.

Nástroje klasické FA tentokrát nelze použít, a to hned z několika důvodů. Především zde máme diskrétní prostor (na rozdíl od lineárního=vektorového prostoru), nelze vůbec hovořit o normálním rozložení a nemá ani smysl sestavovat korelační matici. Tím tedy padají všechny klasické metody řešení FA.

Úspěch BFA měříme pomocí chybové funkce **discrepancy**, značíme d, která je definována jako počet rozdílných bitů mezi součinem $\mathsf{F}\odot\mathsf{A}$ a původními daty X.

$$\hat{\mathsf{X}} = \mathsf{F} \odot \mathsf{A}$$

$$d = \sum_{i=1}^{n} \sum_{j=1}^{p} |\hat{x}_{ij} - x_{ij}|$$

Cílem BFA samozřejmě je, aby hodnota d byla co nejmenší, v ideálním případě $d = 0$. Stejně jako u klasické faktorové analýzy, ani zde nelze počet faktorů m nijak rozumně spočítat, musí tedy být součástí zadání. Případně lze provést výpočet pro několik hodnot m a dle výsledku vybrat nejvhodnější variantu.

4 Výpočet binární faktorové analýzy

Ačkoliv zadání uvedené v předchozí kapitole je jednoduché a snadno pochopitelné, žádná spolehlivá obecně použitelná metoda řešení BFA zatím není známa. Výzkum v posledních letech přinesl několik metod, které nabízejí alespoň částečně uspokojivé řešení. Jedná se o různé varianty a modifikace těchto tří základních postupů:

Neuronová síť Hopfieldova typu (viz [4])
Bylo dokázáno, že modifikovaná Hopfieldova síť může být použita k vyhledávání binárních faktorů způsobem podobným práci lidského mozku. Učením jsou parametry sítě nastaveny tak, aby během fáze vybavování odpovídaly atraktory (stavy, do kterých síť směřuje) binárním faktorům ve smyslu BFA. Problém je určení počátečního nastavení sítě ve fázi vybavování, aby atraktorem opravdu byl faktor a abychom postupně dokázali najít všechny faktory (ne jen jeden). Další problém pak je, že síť funguje pouze při malé informační zátěži.

Slepé hledání (zkoušení všech možností – viz [4], [5], [6])
Jelikož řešení BFA je ukryto ve správném nastavení bitů matic F a A, je nasnadě, že pro velmi malé rozměry matic lze jít cestou vyzkoušení všech bitových kombinací. Tento postup má sice vysokou složitost řádu $O(2^n)$, vede však přímo k nalezení nejlepšího možného řešení.
V poslední době byla publikována celá řada algoritmů, jak efektivně omezit toto prohledávání bez nebezpeční zhoršení kvality výsledku. V praxi je tak možno tímto algoritmem nalézt řešení až pro zhruba 100 bitů (při složitosti 2^{100} by výpočet samozřejmě trval miliardy let).

Převod na problém konceptuálních svazů (viz [6])
Řešení BFA lze získat také převodem na problém sestavení konceptuálního svazu. Celý postup je částečně obdobou slepého hledání, ovšem prohledávají se jen formální koncepty, kterých je v průměrném případě mnohem méně než bitů v maticích, takže složitost v průměrném případě je výrazně redukována. Tato metoda ve všech testech obstála mnohem lépe než obě výše uvedené metody.

Tento článek popisuje zcela nový (čtvrtý) přístup k řešení BFA, s pomocí genetického algoritmu (GA). Ten funguje podobně jako Hopfieldova síť na bázi minimalizace chybové funkce, genetickým algoritmům je však obecně připisována schopnost nacházet globální minimum (ne jen lokální). Použitelnost GA je však podmíněna nalezením vhodné datové reprezentace (kódování genů) a genetických operátorů.

5 Genetický algoritmus pro výpočet BFA

5.1 Co je to genetický algoritmus

Genetický algoritmus je počítačová simulace, ve které jsou jedinci populace abstraktních reprezentací kandidátních řešení optimalizačního problému stochasticky vybíráni, rekombinováni, mutováni a potom odstraněni nebo ponecháni v populaci podle jejich kvality neboli vhodnosti (**fitness**), viz [3]. Tento princip simuluje chování přírody ve smyslu boje o přežití. Zákonitost přírodního výběru a přežití nejsilnějších jedinců se osvědčuje při řešení algoritmicky obtížně zvládnutelných problémů, mezi které patří i BFA. V případě BFA je nasnadě, že jedinci v populaci budou matice F a A (buď v páru, nebo jen jedna z nich – podle konkrétního návrhu algoritmu, viz níže). Tito jedinci budou pak „bojovat“ o přežití. Výsledkem by mělo být řešení BFA.

5.2 Standardní genetický algoritmus

Nelze očekávat, že by existoval jeden univerzální GA, kterým by bylo možno přímo řešit všechny problémy. Speciálně u BFA narážíme na fakt, že se pohybujeme v diskrétním prostředí, které znesnadňuje použití jakýchkoliv obvyklých postupů (jak již bylo zmíněno výše). Většina konkrétních GA vychází z původního Hollandova [1] a Goldbergova [2] algoritmu, obvykle označovaného SGA (Simple G. A.).

SGA reprezentuje jedince jako bitové řetězce (konstantní délky) = „kus paměti“. Každý jedinec představuje jednu možnou variantu řešení daného problému.

Vhodnost jedinců = „kvalita řešení“ je dána fitness funkcí η, která každému jedinci přiřadí nezáporné reálné číslo – hodnotu vhodnosti (nula je nejhorší).

Noví jedinci jsou vytvářeni vždy ve vlnách – generacích. Délka života je vždy jedna generace. K vytváření nových jedinců používá SGA tři operátory:

Mutace - náhodná změna bitu v řetězci.
Křížení - Dva bitové řetězce AB a CD se v náhodném místě rozdělí a spojí se do kříže (odtud pojem křížení) v nové jedince AD a CB.
Selekce - Pravděpodobnost, že jedinec bude vybrán k reprodukci = „vytvoření potomků“ je tím vyšší, čím vyšší je jeho fitness hodnota.

Na tomto místě záměrně vynecháváme hlubší detaily SGA, neboť pro námi navržené řešení BFA nejsou podstatné. Obecně platí, že úspěch SGA závisí na vlastnostech fitness funkce (spojitost a „tvar“ funkce), vhodně zvolené počáteční generaci (v našem případě vždy náhodně generovaná), nastavení pravděpodobností mutace a křížení a případně na nutnosti výpočtu fitness funkce převodem z funkce jiných vlastností (to je i případ BFA – naše d má jiné vlastnosti než má mít fitness funkce).

5.3 Aplikace SGA pro řešení BFA

Datová reprezentace jedinců je v případě BFA velmi snadná – můžeme totiž přímo vzít binární matice. Jedinec je tedy tvořen maticemi F a A, uloženými v paměti po řádcích zleva doprava (běžné ukládání matic v paměti).

Mutaci lze realizovat buď inverzí jednotlivých bitů, nebo inverzí celých řádků (práce se sloupci je implementačně obtížná, proto ji vynecháváme).

Křížení lze realizovat rovněž buď na úrovni bitů, kdy se matice kříží v libovolném bodě, nebo na úrovni řádků, kdy zůstává zachována celistvost řádků matic.

Discrepancy d má opačný průběh než fitness funkce η (menší hodnoty jsou lepší), je nutno použít nějakou formu převodu. Jelikož d je diskrétní funkce shora omezená celkovým počtem bitů v maticích, lze použít tento jednoduchý převodní vzorec ($n \times m$ a $m \times p$ jsou rozměry matic F a A, i označuje konkrétního jedince z populace):

$$\eta(i) = m \cdot (n + p) - d(i)$$

Výpočet začíná s náhodnou populací. V každém kroku algoritmu je vytvořena celá nová generace takto: Do pomocného pole se nakopíruje každý jedinec i tolikrát, jakou hodnotu má jeho $\eta(i)$. Potom se náhodně vybírají páry jedinců z tohoto pole a zkříží se. Každý bit všech jedinců nové generace je pak s pravděpodobností p_m mutován. Zde popsaný postup se v literatuře vyskytuje v různých nuancích. Vzhledem k tomu, že cílem této práce je popis nového lepšího algoritmu, některé detaily SGA zde nebyly podrobně rozepsány.

5.4 Výsledky SGA

Při řešení BFA vykazoval algoritmus SGA poměrně slabých výsledků. Hodnota d byla ve všech testech více než dvojnásobná ve srovnání s výsledky dosaženými metodou formálních konceptů (viz [6]). Populace, i když vycházely z náhodného počátečního nastavení genů, obvykle postupně konvergují do stavu, kdy všichni jedinci jsou identičtí ve smyslu binární rovnosti. Jakmile nastane tato situace (stav úplné konvergence), vývoj jde dále kupředu jen pomocí mutací. Odtud přímo plyne fakt, že funguje-li mutační operátor na úrovni celých řádků matic, je schopnost hledat nová řešení silně omezena. To se potvrdilo i experimentálně.

Hlavním závěrem však je, že SGA dokáže řešit problém BFA, avšak podstatně hůře než výše zmíněná metoda formálních konceptů, je tedy ve své podstatě zbytečný.

6 Genetický algoritmus GABFA

6.1 Princip GABFA

Zjištění, že klasický genetický algoritmus SGA neřeší BFA příliš dobře, nemusí automaticky znamenat, že problém BFA geneticky řešit nelze. Tato kapitola představuje modifikovaný genetický algoritmus, nazvaný GABFA (zkratka z „Genetický Algoritmus pro Binární Faktorovou Analýzu“), který vykazuje diametrálně odlišné (mnohem lepší) výsledky. Tento algoritmus vznikl postupným zkoušením různých modifikací SGA a analýzou chování systému při aplikaci na umělá i reálná data.

Princip fungování GABFA se od SGA v mnoha ohledech liší. Stejný zůstal jen konceptuální pohled na problém: Řešení BFA je hledáno principem GA, tedy pomocí postupného genetického vývoje jedinců v populaci. Každý jedinec žije pouze jednu generaci, reprodukce opět funguje na bázi operátorů selekce, křížení a mutace.

6.2 Datová reprezentace, využití znalostí

Využití našich znalostí může pomoci k rychlejšímu nalezení řešení (tj. během menšího počtu generací). Ukázalo se však, že přílišné zasahování do pseudonáhodného chování GA spíše škodí, než pomáhá, proto musíme postupovat velmi opatrně.

Využijeme především algoritmus pseudo-dělení binárních matic. Každý jedinec pak obsahuje jen matici A, zbytek dopočítáme jako $\mathsf{F} = \mathsf{X}/\mathsf{A}$. Dále je vhodné aplikovat veškeré známé metody předzpracování (preprocessing) na matici X (viz [4], [5], [6]).

6.3 Parametry a parametrizace algoritmu

Algoritmus je možno parametrizovat takto:

Velikost populace $|pop|$ - stačí i poměrně malé hodnoty, např. 200 jedinců.
Pravděpodobnost mutace p_m - obvykle v intervalu $[0.001 - 0.1]$. Příliš velká hodnota devastuje populaci (viz Černobyl 1986). Oproti živým organizmům je zde však velmi silně aplikován výběr silnějších jedinců, takže devastující dopad mutací je v GABFA omezen.
Pravděpodobnost křížení (crossover) p_c - v intervalu $[0.1 - 0.5]$. Viz níže.

Nastavení těchto tří číselných hodnot ovlivňuje chování algoritmu. Požadujeme-li větší „jistotu“, je možné provést výpočet s několika různými nastaveními parametrů a vybrat potom nejlepší řešení ze všech. Není to však nutné.

Za „parametr“ nepovažujeme způsob převodu d na η. Jelikož GABFA má menší nároky na vlastnosti funkce η, stačí pouhá změna znaménka $\eta = -d$.

6.4 Inicializace (výchozí populace)

První populace je vytvořena náhodně. To je velmi rychlé, je proto možno vytvořit na začátku více jedinců, například dvojnásobek, a umožnit tak rychlejší rozběhnutí GA. Praxe však ukázala, že vytváření velkých populací je zbytečné, neboť GABFA je natolik robustní, že dosáhne řešení z „téměř“ libovolné počáteční populace.

6.5 Krok výpočtu (každá jedna další generace)

Krokem výpočtu rozumíme vytvoření další generace z generace stávající. Na vstupu očekáváme libovolnou populaci o velikosti $\geq |pop|$ a výstupem je opět populace o velikosti $\geq |pop|$. Ta je potom použita jako vstup v dalším kroku.

1. Všem jedincům spočítáme fitness hodnotu. Zapamatujeme si nejlepší řešení. Najdeme-li jedince, jehož $\eta = 0$, výpočet končí (máme nejlepší řešení).
2. Seřadíme jedince sestupně podle jejich fitness hodnot.
3. Selekce - Zmenšíme populaci na velikost $|pop|$. (Vezmeme $|pop|$ nejlepších.)
4. Křížení - Všichni jedinci se křížení zúčastní, bez ohledu na jejich fitness hodnoty. Pro každého jedince i provedeme následující postup:
   (a) Náhodně vybereme partnera j, $j \neq i$.
   (b) Každý řádek z i s pravděpodobností p_c nahradíme příslušným řádkem z j.
   (c) Přidáme nově vzniklého jedince do populace pro další generaci.

5. Postup křížení popsaný v bodě 4 opakujeme třikrát.
6. Mutace - Procházíme celou novou populaci a každý bit s pravděpodobností p_m změníme na opačnou hodnotu.

Takto modifikovaný GA velice dobře řeší problém BFA (důkaz experimentálně – podrobněji předvedeno níže). Přitom je důležité přesně dodržet popsaný algoritmus. V některé jeho body mohou svádět k zdánlivě banálním změnám z důvodu snadnější implementace, může to však mít fatální dopad na fungování algoritmu jako celku. Některé možnosti úprav algoritmu jsou popsány níže.

Hledané **řešení BFA** je to, které jsme si zapamatovali v bodě 1 (nebereme jej tedy z poslední generace). Důvodem je nepravděpodobná, leč možná „ztráta vítěze“, kdy populace konverguje do silného atraktoru někde blízko globálního minima, nebo kdy se díky mutaci z některé generace vytratí všichni jedinci reprezentující nejlepší řešení.

7 Testy

Podobně jako u jiných metod z oblasti soft computingu, ani funkčnost genetických algoritmů nelze formálně dokázat. Proto důkazy provádíme experimentálně. GABFA jsme testovali na datech použitých v dříve publikovaných pracích o BFA (např. [4], [5], [6], [8]), pro snazší srovnání s jinými známými metodami.

7.1 Datová sada p3

Datová sada p3 je použita jako zástupce nejjednodušších a nejmenších dat. Je to uměle vytvořená řídká matice o rozměru 100×100, kterou lze beze zbytku rozložit na součin dvou matic pomocí 5 faktorů. Obě matice tedy mají celkem 500 bitů, díky řídkosti datové matice X a častému opakování řádků je lze zredukovat na 50 bitů.

Výsledky jsou v tabulce 1. Slepé hledání najde řešení, avšak jen v případě, že předem víme, kolik jedniček má být na každém řádku A. Formální koncepty i GABFA najdou řešení bez dalších omezení a to ve zlomku sekundy. Výpočet pomocí GABFA jsme opakovali $20\times$. Všechna opakování vyšla překvapivě podobně – výsledek byl vždy nalezen a to dokonce vždy během 17 ± 5 generací, což trvalo jen zlomek sekundy.

Metoda	Počet jedniček	Čas (m:s)	Discrepancy (chyba)	Poznámky
Slepé hledání	2-4	61:36	0	375 kombinací na řádku A
Slepé hledání	3	0:12	0	120 kombinací na řádku A
Formální koncepty	1-10	0:00	0	Použito 8 z 10 konceptů
GABFA	jakýkoliv	0:00	0	Průměrně 17 generací

Tab. 1. Výsledky testů na datové sadě p3 (řídká matice 100×100).

7.2 Datová sada p2

Datová sada p2 je opět uměle vytvořená matice rozměrů 100×100. Tentokrát však již nejde o řídkou matici. Řešení pomocí slepého hledání není prakticky proveditelné, podrobnou analýzu a zdůvodnění je možno nalézt v [6].

Matici lze rozložit opět beze zbytku na součin matic s vnitřním rozměrem 5 (tedy pomocí 5 faktorů). Formální koncepty i GABFA dokáží najít správné řešení, výsledky ukazuje tabulka 2.

Metoda	Počet jedniček	Čas (m:s)	Discrepancy (chyba)	Poznámky
Slepé hledání - 2 faktory	6	11:44	743	Výpočet pro pouze 2 faktory
Slepé hledání - 5 faktorů	6	10^9 let	0	Pouze odhad
Formální koncepty	1-10	0:07	0	Použito 80 ze 111 konceptů
GABFA	libovolný	0:01	0	Průměrně 114 generací

Tab. 2. Výsledky testů na datové sadě p2 (běžná matice 100×100).

Tabulka ukazuje, že slepé hledání nelze použít pro více než 2 faktory. Chyba výsledku je tedy pochopitelně hodně velká. Podrobnější analýzou algoritmu lze zjistit, že výpočet všech 5 faktorů by trval přibližně 10^9 let (viz také [6]), což je daleko mimo rozumných mezí. Formální koncepty dokáží vypočítat přesné řešení během 7 sekund. GABFA provede výpočet obvykle za zhruba 1 sekundu (provedeno 10 pokusů, rozptyl $\pm$ několik desetin sekundy) – nová metoda, přestože používá náhodná čísla, je tedy nejen rychlá, ale i překvapivě robustní a nevykazuje výchylky od průměrné doby výpočtu.

7.3 Berryho datová sada

Představitelem reálných dat je Berryho datová sada obsahující výskyty slov v textových dokumentech (velmi vhodná data pro BFA, byla použita např. v článku [8]). Binární faktory při této aplikaci BFA tedy reprezentují množiny slov, které vykazují výskyt v podobných dokumentech. BFA lze potom použít jako nástroj pro posuzování podobnosti obsahu textových dokumentů pro úlohy typu: „Zaujal mě tento dokument a chci najít další jemu podobné.“, což je alternativou k obvyklému vyhledávání podle obsažených slov (styl Google). Výsledky ukazuje tabulka 3.

Tato data již nelze beze zbytku rozložit na malý počet faktorů (obecně může být počet faktorů i větší než počet proměnných!), proto jsme hledali vždy 4 faktory a sledovali discrepancy výsledku u jednotlivých metod (menší je lepší). Slepé hledání je vyloženě nepoužitelně pomalé, najde však zaručeně nejlepší řešení, proto může posloužit jako referenční. Zajímavé je, že metoda formálních konceptů ani při 4 faktorech s využitím gradientního zpřesnění nenajde řešení srovnatelné se 3 faktory nalezenými slepým hledáním (chyba 110 ku 102). Naopak řešení GABFA je opět nalezeno velmi rychle a přitom má prakticky poloviční discrepancy než ostatní metody.

Metoda	Počet jedniček	Čas (m:s)	Discrepancy (chyba)	Poznámky
Slepé hledání - 3 faktory	3	0:39	102	Výpočet pro pouze 3 faktory
Slepé hledání - 4 faktory	3	127:00	68	68 je globální minimum d
Formální koncepty	2-15	0:00	126	Použito 19 ze 37 konceptů
F.k. + gradient	2-15	0:00	110	
GABFA	libovolný	0:01	68	68 je globální minimum d

Tab. 3. Výsledky testů na Berryho datové sadě (reálná data 18×14, 4 faktory).

8 Závěr

Všechny testy ukázaly, že genetický algoritmus GABFA je doposud nejlepší metoda pro binární faktorovou analýzu. Co do rychlosti je srovnatelný s doposud nejrychlejším algoritmem (formální koncepty) a co do přesnosti výpočtu je dokonce ještě lepší. GABFA přitom pracuje velmi obecně a nevyžaduje žádné speciální vlastnosti datové sady, kterou analyzujeme. Za jeho nedostatek můžeme považovat jen fakt, že jeho funkčnost není nijak formálně dokázána (totéž však platí o algoritmu formálních konceptů, neurosíťový algoritmus pro změnu má důkazy, nebyl však doposud implementován v použitelné verzi).

Reference

1. Holland J.H.: Adaptation in Natural and Artificial Systems. Ann Arbor, University of Michigan Press, 1975. ISBN 0-262-58111-6.
2. Goldberg D.E.: Genetic Algorithms in Search, Optimization & Machine Learning. Addison-Wesley, 1989. ISBN 0-201-15767-5.
3. Hondroudakis A., Malard J., Wilson G.V.: An Introduction to Genetic Algorithms Using RPL2. 1996.
4. Húsek D., Frolov A.A., Keprt A., Řezanková H., Snášel V.: O jednom neuronovém přístupu k redukci dimenze. In: Znalosti 2004, Brno, VŠB Technická Univerzita, Ostrava, 2004, 327–337. ISBN 80-248-0456-5.
5. Keprt A.: Binární faktorová analýza a komprese obrazu pomocí neuronových sítí. In: Wofex 2003. VŠB TU, Ostrava, 2003, ISBN 80-248-0106-x.
6. Keprt A.: Using Blind Search and Formal Concepts for Binary Factor Analysis. In: Dateso 2004. Desná-Černá říčka. VŠB Technická Univerzita, Ostrava, 2004.
7. Olej V.: Comparison Of Distributed Genetic Algorithms and Evolution Strategies. In: Mendel '97. VUT, Brno, 1997. ISBN 80-214-0884-7.
8. Řezanková H., Húsek D., Frolov A.A.: Using Standard Statistical Procedures for Boolean Factorization. In: SIS 2003. Neapol, Itálie, 2003.

Comparison of kernel based regularization networks and RBF networks

Petra Kudová*

Institute of Computer Science, Academy of Sciences of the Czech Republic
18207 Prague, Czech Republic
petra@cs.cas.cz

Abstract. The problem of *learning from examples* is a subject of great interest at present. We discuss two approaches to this problem – the Radial Basis Function Networks (RBF networks) and the Regularization networks (RN). The RBF networks represent a model of artificial neural networks with both neuro-physiological and mathematical motivation. The RNs are derived from the regularization theory and have very good theoretical background.
Performance of both approaches is demonstrated on experiments, including both benchmark and real-life learning tasks. We claim that the performance of RN and RBF network is comparable in terms of generalization error. The RN approach usually leads to solutions with higher model complexity (high number of base units). In this situations, the RBF networks can be used as a 'cheaper' alternative.

1 Introduction

The problem of *learning from examples* (also called *supervised learning*) is a subject of great interest. Systems with the ability to autonomously learn a given task, would be very useful in many real life applications, namely those involving prediction, classification, control, etc.

The problem can be formulated as follows. We are given a set of examples $\{(\boldsymbol{x}_i, y_i) \in R^d \times R\}_{i=1}^N$ that was obtained by random sampling of some real function f, generally in presence of noise. To this set we refer as *a training set*. Our goal is to recover the function f from data, or find the best estimate of it. It is not necessary that the function exactly interpolates all the given data points, but we need a function with good generalisation. That is a function that gives relevant outputs also for the data not included in the training set.

The learning problem can be handled by artificial neural networks. There is a good supply of network architectures and corresponding supervised learning algorithms (see [1]). In this case the model, that is a particular type of neural network, is chosen in advance and its parameters are tuned during learning so as to fit the given data. In section 2 we will describe one type of neural network – an RBF network.

* This work was supported by GA ČR grant 201/02/0428.

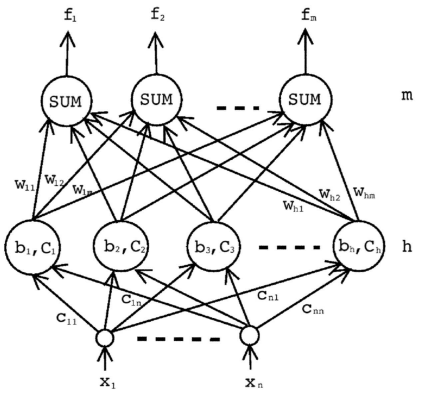
The supervised learning of neural networks is also studied as a function approximation problem. Given the data set, we are looking for the function that approximate the unknown function f. It is usually done by *Empirical Risk Minimization*, i.e. minimizing the functional $H[f] = \frac{1}{N}\sum_{i=1}^{N}(f(\boldsymbol{x}_i) - y_i)^2$ over a chosen *hypothesis space*, i.e. over a set of functions represented by a chosen type of neural network. In section 3 we will study the problem of learning from examples as a function approximation problem and show how regularization network (RN) is derived from regularization theory.

Both approaches (RBF network and RN) suffer from the presence of additional parameters that has to be set in advance. Methods for estimation of these parameters are disscussed in section 4.

In section 5 the performance of RBF network and RN is compared on experiments, including both benchmark and real learning tasks.

2 RBF neural networks

An RBF neural network (RBF network) represents a relatively new model of neural network. On the contrary to classical models (multilayer perceptrons, etc.) it is a network with local units which was motivated by the presence of many local response units in human brain. Other motivation came from numerical mathematics, radial basis functions (RBF) were first introduced in the solution of real multivariate problems [2].



$$y(\boldsymbol{x}) = \varphi\left(\frac{\| \boldsymbol{x} - \boldsymbol{c} \|_C}{b}\right) \quad (1)$$

$$f_s(\boldsymbol{x}) = \sum_{j=1}^{h} w_{js}\varphi\left(\frac{\| \boldsymbol{x} - \boldsymbol{c}_j \|_{C_j}}{b_j}\right) \quad (2)$$

Fig. 1. a) RBF network architecture b) RBF network function.

An RBF network is a standard feed-forward neural network with one hidden layer of RBF units and linear output layer (fig. 1). By an RBF unit we mean a neuron with n real inputs and one real output, realising a radial basis function (1), usually Gaussian. Instead of the Euclidean norm we use the *weighted norm* $\|\cdot\|_C$, where $\|\boldsymbol{x}\|_C^2 = (C\boldsymbol{x})^T(C\boldsymbol{x}) = \boldsymbol{x}^T C^T C\boldsymbol{x}$.

The network computes a function $\boldsymbol{f} = (f_1, \ldots, f_m)$ as linear combination of outputs of the hidden layer (see (2)).

The goal of RBF network learning is to find the parameters (i.e. centers $\boldsymbol{c}$, widths b, norm matrices C and weights w) so as the network function approximates the function given by the training set $\{(\boldsymbol{x}_i, \boldsymbol{y}_i) \in R^n \times R^m\}_{i=1}^N$.

There is a variety of algorithms for RBF network learning, in our past work we studied their behaviour and possibilities of their combinations [3], [4].

The two most significant algorithms, *Three step learning* and *Gradient learning*, are sketched in Algorithm 2.1 and Algorithm 2.2. See [3] for details.

Input: Data set $\{\boldsymbol{x}_i, \boldsymbol{y}_i\}_{i=1}^N$ **Output:** $\{\boldsymbol{c}_i, b_i, C_i, w_{ij}\}_{i=1..h}^{j=1..m}$

1. Set the centers $\boldsymbol{c}_i$ by a k-means clustering.
2. Set the widths b_i and matrices C_i.
3. Set the weights w_{ij} by solving $\Phi W = D$.

$$D_{ij} = \sum_{t=1}^{N} y_{tj} e^{-\left(\frac{\|\boldsymbol{x}_t - \boldsymbol{c}_i\|_{C_i}}{b_i}\right)^2}, \Phi_{qr} = \sum_{t=1}^{N} e^{-\left(\frac{\|\boldsymbol{x}_t - \boldsymbol{c}_q\|_{C_q}}{b_q}\right)^2} e^{-\left(\frac{\|\boldsymbol{x}_t - \boldsymbol{c}_r\|_{C_r}}{b_r}\right)^2}$$

Algorithm 2.1

Input: Data set $\{\boldsymbol{x}_i, \boldsymbol{y}_i\}_{i=1}^N$ **Output:** $\{\boldsymbol{c}_i, b_i, C_i, w_{ij}\}_{i=1..h}^{j=1..m}$

1. Put the small part of data aside as an evaluation set ES, keep the rest as a training set TS .
2. $\forall j$ $\boldsymbol{c}_j(i) \leftarrow$ random sample from TS_1, $\forall j$ $b_j(i), \Sigma_j^{-1}(i) \leftarrow$ small random value, $i \leftarrow 0$
3. $\forall j, p(i)$ in $\boldsymbol{c}_j(i), b_j(i), \Sigma_j^{-1}(i)$:
   $\Delta p(i) \leftarrow -\epsilon \frac{\delta E_1}{\delta p} + \alpha \Delta p(i-1), \qquad p(i) \leftarrow p(i) + \Delta p(i)$
4. $E_1 \leftarrow \sum_{\boldsymbol{x} \in TS_1} (f(\boldsymbol{x}) - y_i)^2$, $E_2 \leftarrow \sum_{\boldsymbol{x} \in TS_2} (f(\boldsymbol{x}) - y_i)^2$
5. If E_1 and E_2 are decreasing, $i \leftarrow i + 1$, go to 3, else STOP. If E_2 started to increase, STOP.

Algorithm 2.2

3 Approximation via regularization network

In this section we will study the problem of learning from examples by means of regularization theory.

We are given a set of examples $\{(\boldsymbol{x}_i, y_i) \in R^d \times R\}_{i=1}^N$ obtained by random sampling of some real function f and we would like to find this function.

Since this problem is ill-posed, we have to add some a priori knowledge about the function. We usually assume that the function is *smooth*, in the sense that two similar inputs corresponds to two similar outputs and the function does not

oscillate too much. This is the main idea of the regularization theory, where the solution is found by minimizing the functional (3) containing both the data and smoothness information.

$$H[f] = \frac{1}{N}\sum_{i=1}^{N}(f(\boldsymbol{x}_i) - y_i)^2 + \gamma\Phi[f], \tag{3}$$

where Φ is called a *stabilizer* and $\gamma > 0$ is *the regularization parameter* controlling the trade off between the closeness to data and the smoothness of the solution. The regularization scheme (3) was first introduced by Tikhonov [5] and therefore it is called a Tikhonov regularization.

The regularization approach has good theoretical background, it was shown that for a wide class of stabilizers the solution has a form of feed-forward neural network with one hidden layer, called *regularization network*, and that different types of stabilizers lead to different types of regularization networks [6], [7].

Poggio and Smale in [7] proposed a learning algorithm 3.1 derived from the regularization scheme (3). They choose the hypothesis space as a Reproducing Kernel Hilbert Space (RKHS) $\mathcal{H}_K$ defined by an explicitly chosen, symmetric, positive-definite kernel function $K_{\boldsymbol{x}}(\boldsymbol{x}') = K(\boldsymbol{x}, \boldsymbol{x}')$. The stabiliser is defined by means of norm in $\mathcal{H}_K$, so the problem is formulated as follows:

$$\min_{f\in\mathcal{H}_K} H[f], \text{ where } H[f] = \frac{1}{N}\sum_{i=1}^{N}(y_i - f(\boldsymbol{x}_i))^2 + \gamma||f||_K^2. \tag{4}$$

The solution of minimisation (4) is unique and has the form

$$f(\boldsymbol{x}) = \sum_{i=1}^{N} c_i K_{\boldsymbol{x}_i}(\boldsymbol{x}), \qquad (N\gamma I + K)\boldsymbol{c} = \boldsymbol{y}, \tag{5}$$

where I is the identity matrix, K is the matrix $K_{i,j} = K(\boldsymbol{x_i}, \boldsymbol{x_j})$, and $\boldsymbol{y} = (y_1, \ldots, y_N)$.

The most commonly used kernel function is Gaussian $K(\boldsymbol{x}, \boldsymbol{x}') = e^{-\left(\frac{\|\boldsymbol{x}-\boldsymbol{x}'\|}{b}\right)^2}$.

Input: Data set $\{\boldsymbol{x}_i, y_i\}_{i=1}^N \subseteq X \times Y$ **Output:** Function f.

1. Choose a symmetric, positive-definite function $K_{\boldsymbol{x}}(\boldsymbol{x}')$, continuous on $X \times X$.
2. Create $f : X \to Y$ as $f(\boldsymbol{x}) = \sum_{i=1}^{N} c_i K_{\boldsymbol{x}_i}(\boldsymbol{x})$ and compute $\boldsymbol{c} = (c_1, \ldots, c_N)$ by solving

$$(N\gamma I + K)\boldsymbol{c} = \boldsymbol{y}, \tag{6}$$

where I is the identity matrix, K is the matrix $K_{i,j} = K(\boldsymbol{x_i}, \boldsymbol{x_j})$, and $\boldsymbol{y} = (y_1, \ldots, y_N)$, $\gamma > 0$ is real number.

Algorithm 3.1

The power of the algorithm (2.1) is in its simplicity and effectiveness. On the other hand, it has also some drawbacks. First of all, the size of the model (that is a number of kernel functions) corresponds to the size of the training set and so the tasks with huge data sets lead to solutions of implausible size.

Then there are the parameters γ and in case of Gaussian kernel also width b, which are supposed to be fixed. Let us describe how they influence the solution. Once they are fixed, the algorithm reduces to the problem of solving linear system of equations (6).

Since the system has N variables, N equations, K is positive-definite and $(N\gamma I + K)$ is strictly positive, it is well-posed, i.e. is has a unique solution and the solution exists. But we would also like it to be well-conditioned, i.e. insensitive to small perturbations of the data. In other words, we would like the condition number of the matrix $(N\gamma I + K)$ to be small, which is fulfilled if $N\gamma$ is large. Note that we are not entirely free to choose γ, because with too large γ we loose the closeness to data. See figure 3b.

The second parameter b determines the width of the Gaussians. Suppose that the distances between the data points are high or the widths are small, than the matrix K has 1s on diagonal and small numbers everywhere else and therefore is well-conditioned. If the widths are too large, all elements of the matrix K are close to 1 and its condition number tends to be high.

The real performance of the algorithm depends significantly on the choice of parameters γ and b. The optimal choice of these parameters depends on a particular data set.

4 Model selection

By model selection we mean the selection of network architecture and its parameters. We use Gaussian function as a kernel function for RN and as a basis function for RBF network, so the selection reduces to estimation of regularization parameter γ and width b in case of RN, and size of hidden layer h in case of RBF network.

We estimate the parameters γ and b of RN by grid search and crossvalidation. During the grid search the pairs (γ, b) are tried and the one with the lowest cross-validation error is picked. To reduce the time requirements of the search we practice a *grid refining*, i.e. start with a coarse grid and then use a finer grid only in the region with the lowest cross-validation error.

Still the grid search and the cross-validation are quite time consuming, since they require iterative re-runs of the algorithm. On the other hand they are suitable for parallelization since both the pairs (γ,b) and particular runs of cross-validation are independent.

In case of RBF networks, it is usually sufficient to try several network sizes.

5 Experiments

We tested the described methods on three experiments. All of them were run on nodes of Linux cluster with AMD Athlon(tm) XP 2100+ processors.

RN with Gaussian kernel function and RBF networks with Gaussian units were used. Linear systems were solved using the LAPACK library [8].

We always use two disjunct data sets, a training set for training of the network and a testing set for evaluating an error of results. In all experiments we use the normalized error (7):

$$E_{ts} = 100\frac{1}{N}\sum_{i=1}^{N}||\boldsymbol{y}_i - f(\boldsymbol{x}_i)||^2 \qquad \begin{array}{l} N \text{ number of examples in } \{(\boldsymbol{x}_i, \boldsymbol{y}_i)_{i=1}^N\} \\ f \text{ network output} \end{array} \quad (7)$$

In order to compare an accuracy of RN and RBF networks, we have selected three benchmark problems from the Proben1 database, the *Cancer* and the *Glass* classification tasks, and the *Heart* approximation problem. Moreover, each of the Proben1 data sets is available in three different ordering defining different data partitions for training and testing. These are referred to, eg. as glass1, glass2, and glass3 in the original report [9].

Table 2 and graphs in figure 2 compare the performance of RN, RBF and in addition MLP (multilayer perceptron) on Proben1 tasks. RBF networks were trained by the gradient algorithm 2.2 (5000 iterations), RN by the algorithm 3.1. The results for MLP are taken from [9]. We can see that the RN and the RBF network achieved almost the same rate of accuracy.

Table 3 compares the classification acurracy (the percentage of correctly classified samples) of RN, RBF network and support vector machine (SVM) [10]. The classification accuracy for MLP is not included in [9], so the comparison with MLP is not possible in this case.

The time requirements are following: 9 seconds (1 second, 22 seconds) for one run of the algorithm 3.1 on *cancer* (*glass, hearta* respectively) data set, 100 iterations of the algorithm 2.2 took 14 seconds, 9 seconds, 74 seconds, respectively. Figure 3 shows the value of the resulting error with respect to γ and b.

Task	Type	n	m	Train. set size	Test set size
Cancer	Class.	9	2	525	174
Glass	Class.	9	6	161	54
Heart	Approx.	35	1	690	230

Tab. 1. Overview of data sets from Proben1 database.

As an example of a practical task we have chosen the prediction of flow rate on the river Ploucnice. We have two data sets named *ploucnice1* and *ploucnice2*, for the prediction based on information from previous one and two days, respectively.

	RN			RBF			MLP		
	E_{ts}	d	γ	E_{ts}	std	arch	E_{ts}	std	arch
cancer1	1.60	1.0	0.0002	1.64	0.16	20	1.60	0.41	4+2
cancer2	2.99	1.4	0.0002	2.89	0.07	20	3.40	0.33	8+4
cancer3	2.76	1.3	0.0005	2.74	0.20	20	2.57	0.24	16+8
cancer	2.45			2.42			2.52		
glass1	6.75	0.3	0.0008	6.59	0.32	15	9.75	0.41	16+8
glass2	7.28	0.3	0.0014	7.85	0.43	15	10.27	0.40	16+8
glass3	6.48	0.2	0.0017	6.95	0.26	15	10.91	0.48	16+8
glass	6.84			7.13			10.31		
hearta1	4.44	1.9	0.0008	4.84	0.25	30	4.76	1.14	32+0
hearta2	4.32	1.9	0.0012	4.66	0.08	30	4.52	1.10	16+0
hearta3	4.45	1.9	0.0008	4.54	0.06	30	4.81	0.87	32+0
hearta	4.40			4.68			4.70		

Tab. 2. Comparison of Regularization Network (RN), RBF network (RBF) and multilayer perceptron (MLP).

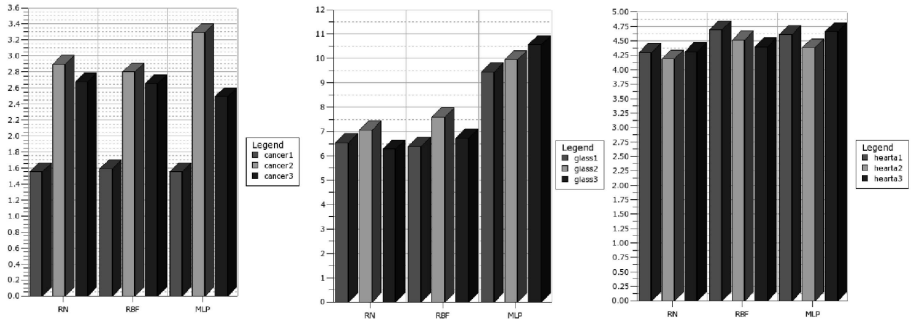


Fig. 2. Comparison of RN, RBF, MLP: test set error.

	RN	RBF	SVM
cancer1	98.85%	98.74%	97.12%
cancer2	95.40%	96.84%	96.55%
cancer3	95.98%	96.95%	95.97%
glass1	75.00%	72.45%	73.58%
glass2	73.07%	64.53%	66.03%
glass3	76.92%	72.26%	79.24%

Tab. 3. Comparison of classification accuracy of RN, RBF and support vector machines (SVM).

Table 4 shows the resulting error of a RN and an RBF network with 15 units. The parameters γ and b of the algorithm 3.1 were estimated by grid search and cross-validation. The RBF network was trained by the algorithm 2.1 and

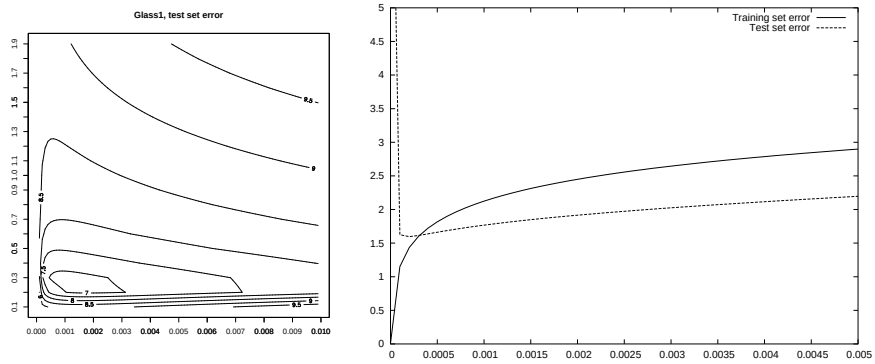


Fig. 3. a) The dependence of the error function (computed on test set) on parameters γ a b. b) The relation between γ and training and testing error.

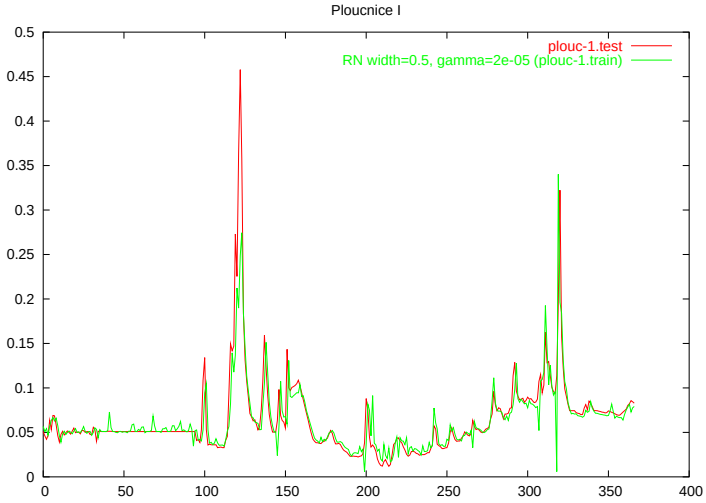


Fig. 4. Prediction of the flow rate on the river Ploucnice by RN.

the computation was run 50 times. Mean errors and its standard deviation are listed. In figure 4 and 5 you can see the prediction by both the RN and the RBF network. Time of one run of the algorithm 2.1 (RBF) was approximately 28 seconds, one run of the algorithm 3.1 (RN) lasted 55 seconds.

6 Conclusion

In this work we discussed two approaches to the learning task – regularization network and RBF network. We demonstrated their behaviour and performance on experiments, including both benchmark and real data sets. We showed that

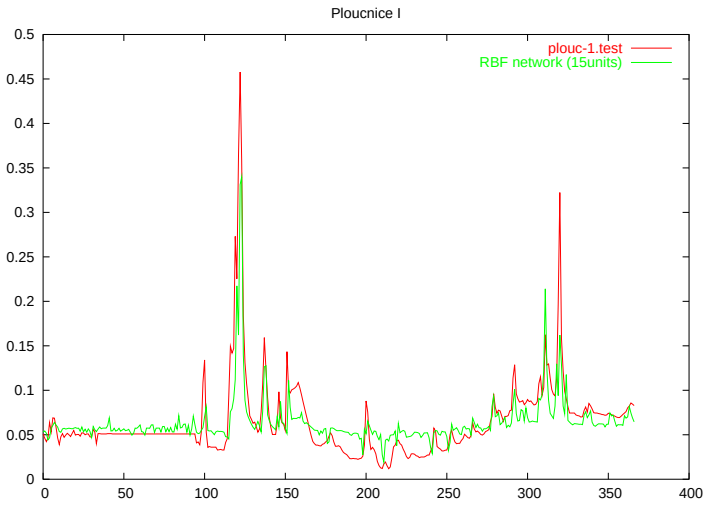


Fig. 5. Prediction of the flow rate on the river Ploucnice by RBF.

		ploucnice1	ploucnice2
RBF	E_{ts}	0.246	0.452
	std	0.15	0.12
	h	15	15
RN	E_{ts}	0.056	0.121
	γ	1.48e-05	1.3e-05
	d	0.5	1.8

Tab. 4. Comparison of the errors on the tasks ploucnice1 and ploucnice2.

the models are comparable, so the RBF network can be used as an alternative to RN in situations where the lower model complexity is desirable.

Both algorithms for RN and RBF networks suffer from the presence of extra parameters that have to be set explicitly. They are the regularization parameter γ and the width b in case of RN, and the number of hidden units h in case of RBF. However, while the estimation of parameters of RN by cross validation is quite time consuming, it is usually sufficient to try several values of h in the case of RBF network.

In our future work we will concentrate on the improvement of described algorithms, especially the ways of estimating the additional parameters.

References

1. Haykin S.: Neural Networks: A Comprehensive Foundation. 2nd edn. Tom Robins, 1999.
2. Powel M.: Radial Basis Functions for Multivariable Interpolation: A review. In: IMA Conference on Algorithms for the Approximation of Functions and Data, RMCS, Shrivenham, England, 1985, 143–167.
3. Neruda R., Kudová P.: Hybrid Learning of RBF Networks. Neural Networks World, **12**, 2002, 573–585.
4. Neruda R., Kudová P.: Learning Methods for RBF Neural Networks. Future Generations of Computer Systems, 2004, in press.
5. Tikhonov A., Arsenin V.: Solutions of Ill-Posed Problems. W.H. Winston, Washington, D.C, 1977.
6. Girosi F., Jones M., Poggio T.: Regularization Theory and Neural Networks Architectures. Neural Computation, **7**, No.2, 1995, 219–269.
7. Poggio T., Smale S.: The Mathematics of Learning: Dealing with Data. Notices of the AMS, **50**, No.5, 2003, 537–544.
8. LAPACK: Linear Algebra Package (`http://www.netlib.org/lapack/`).
9. Prechelt L.: Proben1 – A Set of Benchmarks and Benchmarking Rules for Neural Network Training Algorithms. Technical Report 21/94, Universitaet Karlsruhe, 1994.
10. Chih-Chung C., Chi-Jen L.: Libsvm: A Library for Support Vector Machines, 2002. `http://www.csie.ntu.edu.tw/~cjlin/libsvm/`.

Distribuované indexovanie textu podporované relačnou databázou

Rastislav Lencses

Ústav informatiky, Prírodovedná fakulta, Univerzita P. J. Šafárika v Košiciach
Jesenná 5, 040 01 Košice, Slovenská republika
lencses@science.upjs.sk

Abstrakt V príspevku uvádzame distribuovaný algoritmus generujúci invertovaný zoznam uložený v databáze. Tento invertovaný zoznam potom slúži systému na získavanie informácií (Information Retrieval System) pre potreby dopytovania. Efektívnosť algoritmu bola testovaná na kolekcii slovenských a anglických dokumentov.

1 Úvod

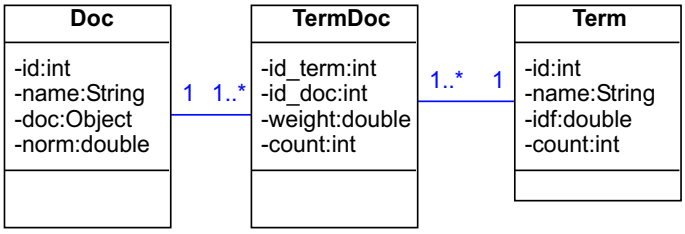
Oblasť získavania informácií (Information Retrieval) sa zaoberá uchovávaním informácií a prístupom ku nim. Algoritmy používané v tejto oblasti sú podložené teoretickými modelmi. My budeme používať vektorový model, ktorý sa javí z viacerých hľadísk najlepší [1]. Pre IRS (Information Retrieval System; systém schopný uchovať a získať informácie) sú informácie zvyčajne uložené vo forme dokumentov. Pri využití relačného databázového servera ako úložišťa dát pre systém poskytujúci vyhľadávanie informácií treba riešiť dve výrazne oddelené úlohy : indexovanie dokumentov a vyhľadávanie s využitím jazyka SQL. IRS používa vo väčšine prípadov kvôli zrýchleniu vyhľadávania invertovaný zoznam [1]. Ide o zoznam termov, pričom pre každý z nich je vytvorený zoznam jeho výskytov v dokumentoch. Pomocou tohto invertovaného zoznamu môžeme rýchlo a ľahko nájsť množinu dokumentov, ktoré obsahujú hľadaný term.

Typické IRS používajú pre vytvorenie a správu invertovaného zoznamu svoje vlastné (proprietárne) mechanizmy. Výhodou tohto prístupu je možnosť plne sa prispôsobiť požiadavkám IRS, a mať možnosť ovplyvniť algoritmy týkajúce sa invertovaného zoznamu, prípadne nahradiť ich novšími a efektívnejšími. V prípade použitia relačnej databázy ako dátové úložište pre invertovaný zoznam zase získame odladený a odskúšaný komponent nášho systému, často bez zníženia funkčnosti a efektívnosti, so zníženým časom vývoja a menšou zložitosťou celého systému. Ďalšia z motivácií vývoja IRS ako aplikácia relačnej databázy je dostupnosť kvalitných databázových strojov. Už koncom 80-tych rokov, v čase vzniku prvých komerčne dostupných relačných databáz[1], vznikli prvé práce, ktoré sa zaoberali aspektom využitia databáz pre systém získavania informácií [7], [4]. V týchto počiatkoch však slabé dosahované výsledky hovorili

[1] v roku 1978 bola na trh uvedená databáza Oracle

proti tomuto využitiu. Neskôr sa táto situácia zmenila. V práci [3] sa autori zaoberajú porovnávaním dvoch systémov pre získavanie informácií, ktoré sa líšia len úložišťom. Jeden používa vlastný mechanizmus na správu invertovaného zoznamu - variant B-stromu, druhý systém používa objektovú databázu. Výsledkom bolo zlepšenie výkonnosti, najmä vďaka inteligentnej správe vyrovnávacej pamäti (cache). Podobné práce [5], [2] taktiež dokumentujú zlepšenie výkonnosti pre systém využívajúci relačnú databázu oproti proprietárnemu riešeniu založenom na špecifickom systéme vo vlastnej réžii. Treba však poznamenať, že proprietárne riešenia (úložište navrhnuté vo vlastnej réžii bez pomoci databázového stroja) sa samozrejme vyrovnajú výkonnosti databázových strojov, keďže používajú podobné (či totožné) algoritmy.

Základ IRS podporovaného relačnou DB spočíva vo vytvorení troch relačných tabuliek, nad ktorými sa pomocou SQL dá dopytovať (obr. 1).



Obr. 1. Diagram tried reprezentujúci databázové relácie pre IRS.

Tabuľka Doc obsahuje zoznam dokumentov, tabuľka Term obsahuje zoznam jedinečných termov zo všetkých dokumentov a tabuľka TermDoc reprezentuje výskyty termov v dokumentoch. Dôležitosť výskytu termu v dokumente je reprezentovaná váhou (weight), dôležitosť samotného termu sa dá podľa vektorového modelu určiť pomocou inverznej frekvencie dokumentu (idf). V práci [8] sme navrhli, implementovali a testovali algoritmus, ktorý je schopný efektívne indexovať veľké kolekcie dokumentov. Indexovanie spočívalo vo vytvorení invertovaného zoznamu, ktorý bol potom vložený do databázy.

V tomto príspevku sa budeme zaoberať zrýchlením nášho algoritmu pomocou distribuovania. Distribuovaný výpočet je typicky riešený viacerými samostatnými počítačmi spojenými sieťou, nevyžaduje rýchlu komunikáciu (resp. nevadí, ak je pomalšia), dekomponované podúlohy nemusia bežať paralelne, pamäť nie je zdieľaná.

V literatúre [10] sa uvádzajú štyri známe architektúry distribuovania výpočtov: SISD, SIMD, MISD, MIMD. Tieto sa líšia tokom inštrukcií a dát. Tok inštrukcií (Instruction) programu môžu byť jednoduchý (Simple), čiže rovnaké inštrukcie pre každý procesor/počítač, alebo rôzne (Multiple) pre každý procesor/počítač. Analogicky dáta (Data), ktoré treba spracovávať, môžu byť totožné (Simple) alebo rôzne (Multiple).

V tomto príspevku budeme využívať distribuovanú architektúru typu SIMD (simple instruction, multiple data), ktorá bude obsahovať samostatné počítače spojené sieťou. Pri analýze časovej zložitosti distribuovaných algoritmov sa do úvahy berie aj počet výpočtových jednotiek. Ideálny distribuovaný algoritmus zrýchli výpočet toľkokrát, koľko výpočtových jednotiek je k dispozícii. Vtedy hovoríme o lineárnom zrýchlení. Kvalitu distribuovaného algoritmu teda vieme merať pomocou nasledovne. Označme $S_K(n)$ ako čas potrebný na riešenie problému K pomocou najlepšieho známeho sekvenčného algoritmu na jednej výpočtovej jednotke (n je veľkosť vstupných dát). $T_K(n,q)$ označme ako čas distribuovaného algoritmu potrebný na riešenie problému K pomocou q počítačov. Zrýchlenie $A_K(n,q)$ pre q počítačov definujeme ako

$$0 < A_K(n,q) = \frac{S_K(n)}{T_K(n,q)} \leq q$$

Platí, že toto zrýchlenie je nanajvýš rovné q (vtedy to je optimálne). Malo by byť väčšie než 1, aby malo distribuovanie výpočtu K význam. Ak by bolo menšie, je vhodnejšie použiť sekvenčný algoritmus. Nakoniec efektivita $E_K(n,q)$ je zrýchlenie prepočítané na jednu výpočtovú jednotku.

$$0 < E_K(n,q) = \frac{A_K(n,q)}{q)} \leq 1$$

I keď v literatúre existujú prístupy ku distribuovaniu indexovania textových dokumentov, omnoho častejšie sa vyskytuje paralelizácia pomocou počítača s viacerými procesormi. Uveďme teraz popis distribuovaného algoritmu na indexovanie dokumentov.

Algoritmus v práci [9] vytvorí pre celú kolekciu najprv hash tabuľku termov (ktorá bude na pevnom disku). Potom bude znova spracovávať dokumenty a evidovať si výskyty termov v dokumentoch. Tieto výskyty bude zapisovať do alokovanej pamäťovej oblasti, kde ich bude triediť (aby pre každý term získal dokumenty, v ktorých sa vyskytuje). Po zaplnení tejto pamäťovej oblasti zapíše jej obsah komprimovane na pevný disk. Posledná fáza bude viaccestné zlučovanie zoznamov výskytov, ktoré boli zapísané na pevný disk, do jedného zoznamu. Algoritmus vyžaduje dvojnásobné parsovanie dokumentov raz pre získanie zoznamu termov, druhý krát pre získanie výskytov termov. Časová zložitosť je O(n). Autori tiež vykonali priamočiare distribuovanie tohto algoritmu. Každý počítač vytvorí nad svojou časťou kolekcie zoznam termov a výskytov termov. Tieto zoznamy sa budú hierarchicky zlučovať. Dosiahnutá efektivita sa blížila jednej (čo je optimálne).

2 Distribuovanie indexovania

Výpočet indexu môžeme urýchliť distribuovaním indexovania pomocou nášho algoritmu [8] medzi viaceré počítače. Hlavná myšlienka spočíva v rozdelení výpočtu na viac fáz, z ktorých niektoré môžu byť distribuované. Väčšinu času zaberú výpočty, zasielanie dát medzi počítačmi bude trvať zanedbateľný čas

(počas jednotlivých fáz sa zašle spojite dohromady najviac niekoľkonásobok veľkosti kolekcie). Prvá fáza (A) sa týka spracovania (parsovania) dokumentov. Množinu dokumentov, ktorú treba indexovať, rozdelíme rovnomerne (vzhľadom na veľkosť alebo počet dokumentov) medzi q počítačov a každý z nich vytvorí dve tabuľky $TermDoc_i$ a Doc_i (kde $i = 1 \dots q$). Každý počítač si tiež vytvorí tabuľku $Term_i$. V tej však nebude identifikátor (primárny kľúč) *id*. Pri spájaní tabuliek vytvorených rôznymi počítačmi by sme totiž mali problém s tým, že rovnaké termy identifikované rôznymi počítačmi by mali rôzne identifikačné čísla (stĺpec *id* v tabuľke $Term$). Preto teda tabuľky $Term_i$ zo všetkých počítačov bude treba neskôr centrálne spojiť vo fáze (B). Táto fáza bude vykonávaná centrálne na jednom počítači a ostatné počítače budú vtedy nečinné. Získanie globálnej tabuľky termov (spolu s globálnymi váhami) potom bude nasledovné:

```
INSERT into Term (name, df, idf)                              (1)
select name, sum(df), log(n/sum(df))
from
select name, df from Term1 union all
select name, df from Term2 union all ...
group by name
```

kde n je počet dokumentov. Primárny kľúč *id* tejto tabuľky $Term$ bude typu auto_increment, čiže jeho hodnota sa nastaví automaticky. Tabuľku $Term$ obsahujúce termy celej kolekcie spolu s ich globálnymi váhami zašleme všetkým počítačom, ktoré ju použijú na výpočet váh termov svojej tabuľky $TermDoc_i$ (s využitím lokálnej váhy, ktorú získali počas parsovania). Výpočet váh termov - fáza (C) prebehne na každom počítači pomocou nasledovného SQL príkazu:

```
INSERT into TermDoc1 (id_term, id_doc, weight, count)       (2)
SELECT TD.id_term, TD.id_doc, TD.count*Term1.idf, TD.count
from Term1 left join Temp on TD.name=Term.name
```

Tabuľka $Temp$ vznikla načítaním poľa výskytov termov zo súboru do databázy (pomocou rýchleho LOAD príkazu dostupného vo väčšine databázových strojov). V rámci tejto fázy tiež prebehne výpočet noriem dokumentov (všetky údaje má každý počítač k dispozícii). Posledná fáza (D) spočíva v zlúčení tabuliek $TermDoc_i$ a Doc_i na centrálny počítač. Budú sa tu prenášať parciálne tabuľky dokumentov, ktorých celková veľkosť bude približne rovná veľkosti kolekcie. Ďalej sa budú prenášať tabuľky obsahujúce výskyty termov v dokumentoch ($TermDoc$). Ich celková veľkosť sa pohybuje medzi dvoma extrémnymi prípadmi. Buď je lineárne závislá od veľkosti tabuľky $Term$ (ak každý dokument obsahuje len jeden jedinečný term, ktorý sa v ňom vyskytuje opakovane). Alebo je veľkosť lineárne závislá od veľkosti kolekcie (ak sa v žiadnom dokumente neopakujú termy). V praxi sa blížime samozrejme ku druhému extrému, čiže veľkosť tabuľky $TermDoc$ je zvyčajne lineárne závislá od veľkosti kolekcie (v závislosti od použitých databázových typov je veľkosť medzi 60 až 90 percentami veľkosti kolekcie).

Pre úplnosť ešte diskutujme o veľkosti tabuľky $Term$, ktorá sa prenáša vo fáze (B). Veľkosť tabuľky $Term$ je exponenciálna (s exponentom menším než 1, typická hodnota je medzi 0,4 a 0,6) voči veľkosti kolekcie. Vyplýva to z Heapsovho zákona [6], počet jedinečných termov m je exponenciálny voči počtu všetkých (opakujúcich sa) termov. Fázu (D) nemusíme vykonať, ak chceme mať dopytovanie distribuované. Vtedy sa totiž otázka bude zasielať viacerým počítačom súčasne a vrátené odpovede po jednoduchom utriedení podľa ohodnotenia budú predstavovať celkovú odpoveď. Výhoda tohto riešenia je urýchlenie vypočítania odpovede. Čím je kolekcia na jednom počítači menšia, tým je možné odpoveď vypočítať rýchlejšie.

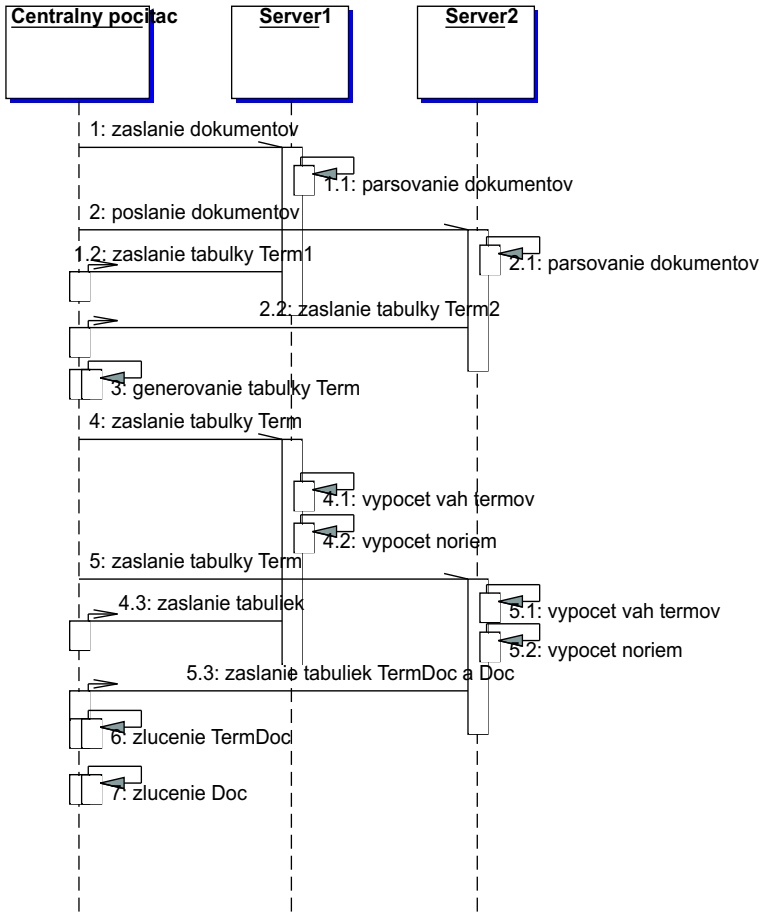
Dokumenty sa vtedy nebudú vyskytovať na viacerých počítačoch súčasne; taktiež vypočítaná váha je konzistentná naprieč celou kolekciou, takže broker (počítač, ktorý distribuuje otázku) nebude tieto váhy prepočítať, taktiež nebude agregovať ohodnotenia jedného dokumentu (to musí, ak by sa vyskytoval na viacerých počítačoch súčasne). Ak však chceme mať dopytovanie zabezpečené centrálne pomocou jedného počítača, musíme vykonať fázu (D). Táto fáza sa musí vykonať centrálne, čo nám zníži efektívnosť využitia počítačov (okrem centrálneho sú všetky v tejto fáze nečinné). Algoritmus si teda môžeme skrátene popísať nasledovne.

Algoritmus Index(D, M) pozostáva z niekoľkých fáz:

- A: Kolekcia dokumentov bude rozdelená na q častí a po častiach bude zaslaná počítačom, ktoré budú spracovávať dokumenty a vytvárať tabuľky $TermDoc_i$ a Doc_i. Na obr. 2 tomu zodpovedajú asynchrónne správy 1. a 2. Po vytvorení týchto tabuliek (správy 1.1 a 2.1) budú parciálne tabuľky $Term_i$ zaslané späť riadiacemu počítaču, ktorý bude v stave čakania (1.2 a 2.2)
- B: Riadiaci počítač vykoná zlúčenie tabuliek $Term_i$ a vytvorí tabuľku $Term$ (krok 3)
- C: Tabuľka $Term$ je zaslaná asynchrónne na všetky počítače, ktoré vypočítajú váhy termov v dokumentoch a ich normy, potom zašlú tabuľky $TermDoc_i$ a Doc_i (kroky 4.x a 5.x)
- D: Riadiaci počítač zlúči parciálne tabuľky $TermDoc_i$ a Doc_i (krok 6. a7.)

3 Výsledky testov

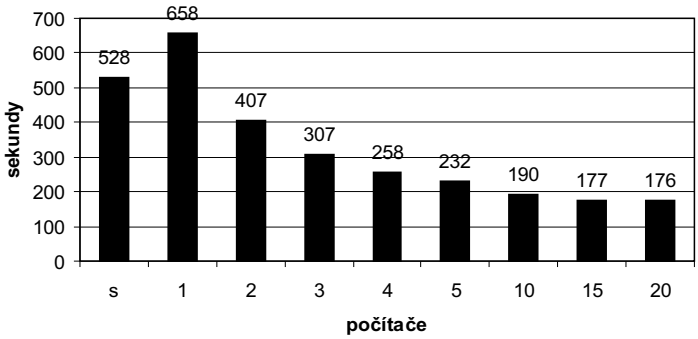
Testy sme vykonali na dvoch kolekciách. Kolekcia novinových článkov denníka SME má okolo 25 500 dokumentov, veľkosť 53 MB. Druhá kolekcia obsahovala anglické články novín Los Angeles Times, 130000 dokumentov, veľkosť 390 MB. Počítačová konfigurácia obsahovala procesor 1,2 GHz a 512 MB operačnej pamäte, ako databáza bola použitá MySQL. Celkový prehľad výsledkov distribuovania na kolekcii SME a jeho porovnania voči sekvenčnému algoritmu je na obr. 3. Vidno, že modifikácia distribuovaného algoritmu spôsobila jeho neefektívnosť v prípade, že ho vykonávame len na jednom počítači. Ak však použijeme viac počítačov, dosiahneme zrýchlenie, ktoré však časom bude klesať. Tento pokles zapríčiňujú najmä fázy (B) a (D) distribuovaného indexovania.



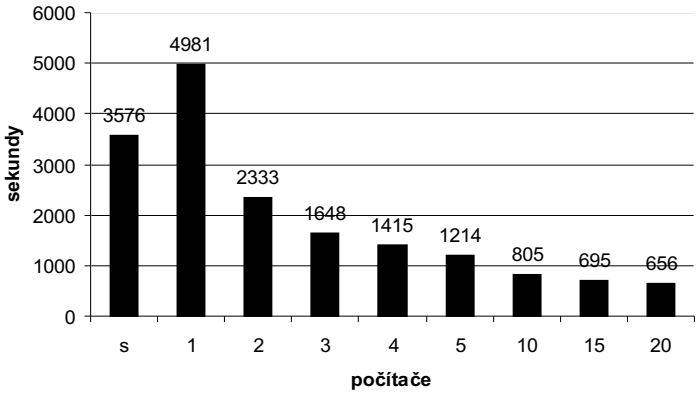
Obr. 2. Algoritmus distribuovania výpočtu indexu.

Pre kolekciu LATimes je badateľné lepšie zrýchlenie - na väčších dátach sa lepšie prejaví distribuovanie fáz (A) a (C), viď obr. 4.

Prvá fáza indexovania - parsovanie dokumentov - vykonávaná pomocou sekvenčného algoritmu trvá pre našu kolekciu SME približne 70% celkového času indexovania (pre kolekciu LATimes približne 60%). V tejto fáze teda bude najviac viditeľný prínos uplatnenia distribuovania. Parsovanie dokumentov má lineárnu časovú zložitosť, nie je tam penalizácia za použitie viacerých počítačov a teda distribuovaním sa zrýchli optimálne; efektivita jedného počítača teda bude rovná jednej. Potvrdzuje to aj obr. 5. Okrem parsovania sa však vo fáze A vykonávajú aj ďalšie dve činnosti: zápis výskytov termov do databázy a generovanie tabuliek termov. Čas zápisu výskytu termov sa síce po pridaní q počítačov sa skráti q krát (je teda lineárne), viď obr. 5, zrýchlenie oproti sekvenčnému algoritmu však nie je optimálne (keďže miesto identifikátora termu sa zapisuje do tabuľky



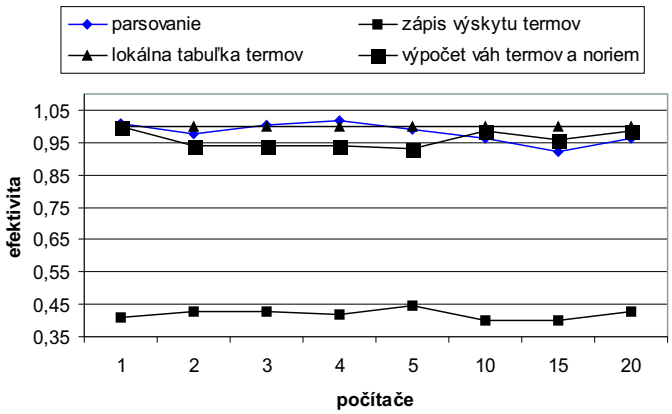
Obr. 3. Porovnanie rýchlosti distribuovania pre rôzny počet počítačov pre kolekciu SME; s znamená sekvenčný algoritmus.



Obr. 4. Porovnanie rýchlosti distribuovania pre rôzny počet počítačov pre kolekciu LATimes; s znamená sekvenčný algoritmus.

jeho názov a ten je zvyčajne dlhší než číslo; taktiež vytváranie databázového indexu pre reťazce zaberie viac času). Generovanie tabuľky termov (či už lokálne alebo globálne) sa pri sekvenčnom algoritme nevyskytuje vôbec (teda zrýchlenie je rovná nule). Na druhej strane však distribuovaný algoritmus nezapisuje termy do databázy (namiesto toho práve generuje tabuľku termov). Efektivita distribuovaného zápisu termov do databázy, výpočtu lokálnej tabuľky termov a parsovania sú zobrazené na obr. 5.

Vo fáze (B) ide o globálne generovanie termov. V prípade zmeny počtu počítačov sa zmení tvar príkazu (1), ktorý sa zväčšuje s narastajúcim počtom počítačov (pre každý počítač v príkaze pribudne jedna tabuľka). Ide tu o spájanie tabuliek, pričom ich celková veľkosť je konštantná. Čím viac tabuliek (hoci



Obr. 5. Efektivita činností fázy A a C v závislosti od počtu počítačov.

súčet ich veľkostí je konštantný) máme spojiť, tým bude formálne zložitejší SQL príkaz na ich spojenie. Syntaktická analýza tohto príkazu, inicializačné práce pre každú tabuľku v príkaze a ďalšie práce spôsobia, že časová zložitosť teda bude lineárna (čo sme experimentálne overili) a bude tvaru:

$$t_G(q,n) = {t^1}_G(q,n) + k_G.q$$

kde ${t^1}_G(q,n)$ zodpovedá vykonanie fázy (B) na jednom počítači, $k_G.q$ zodpovedá časovej penalizácii zapríčinenej s väčším množstvom tabuliek. Hodnota k_G na našej počítačovej konfigurácii je okolo 0,5s na počítač (alebo tabuľku). Globálne generovanie termov nezodpovedá žiadnej fáze v sekvenčnom algoritme a teda zrýchlenie tu bude rovné nule (navyše táto fáza bude vykonávaná centrálne). Fáza (C) - výpočet váh a noriem - má optimálne lineárne zrýchlenie (obr. 5). Časová zložitosť fázy (D) bude analogická ku fáze (B) (taktiež ide o spájanie tabuliek, ktoré bude časovo penalizované počtom tabuliek) a tiež je vykonávaná centrálne (čiže nemá význam hovoriť o zrýchlení).

$$t_J(q,n) = {t^1}_J(q,n) + k_J.q$$

Celkový čas pre danú kolekciu veľkosti n a počet počítačov q teda bude možné vyjadriť nasledovne:

$$t(q,n) = t_A + \frac{{t^1}_B(n)}{q} + \frac{{t^1}_C(n)}{q} + \frac{{t^1}_D(n)}{q} + {t^1}_G(q,n) + k_G.q + \\ + \frac{{t^1}_{EF}(n)}{q} + {t^1}_J(q,n) + k_J.q$$

Konštanta t_A označuje čas načítania morfologického slovníka ispellu, ${t^1}_C(n)$ je čas, ktorý zaberie parsovanie kolekcie na jednom počítači, ${t^1}_D(n)$ je čas, ktorý

zaberie zápis výskytov termov na jednom počítači, ${t^1}_C(n)$ je čas, ktorý je nutný na vytvorenie lokálnej tabuľky termov na jednom počítači, Výraz ${t^1}_G(q,n)+k_G.q$ označuje čas, ktorý treba na vytvorenie centrálnej tabuľky termov, ${t^1}_{EF}(n)$ je čas, ktorý je nutný na výpočet váh a noriem na jednom počítači. Nakoniec k_J je konštanta, ktorá súvisí so zlúčením lokálnych tabuliek $TermDoc$ a Doc. Zo vzťahu (5) vidno, že pre určité množstvo počítačov sa už neoplatí pridať ďalší. Po zistení konštánt a premenných závislých od n je možné riešiť kvadratickú nerovnicu (6) s neznámou q pre nejaké zadané malé ε. Po vyriešení budeme mať k dispozícii maximálny počet počítačov, ktoré má zmysel použiť pri distribuovaní algoritmu.

$$t(q,n) \geq t(q+1,n) + \varepsilon$$

Pre testované kolekcie a napríklad $\varepsilon = 20s$ máme nasledovné optimálne počty počítačov: pre kolekciu SME nemá význam použiť viac než 10 počítačov, pre kolekciu LATimes 22 počítačov. Tieto čísla sú závislé od veľkosti kolekcie, ďalej sa tu predpokladá homogenita počítačov (ak by niektorý pracoval rýchlejšie než iný, bol by nutné upraviť algoritmus, aby rýchlejší počítač dostal viac dát). Taktiež je tu závislosť na nastavení použitej databázy (najmä cache pamäte používané pri vytváraní indexov a pre join operácie). Tieto konštanty (napr. k_G, k_J) je nutné najprv vopred zistiť (pri výpočtoch s malou časťou kolekcie).

4 Záver

V príspevku sme prezentovali distribuovaný algoritmus indexovania textu, ktorý využíva SQL príkazy. Tento algoritmus sme otestovali na reálnych kolekciach dokumentov. Uviedli sme aj odhad zrýchlenia a optimálny počet počítačov vhodný pre výpočet. Tieto odhady však boli zväčša založené na experimentoch. V budúcnosti chceme vykonať testy na väčších kolekciach dokumentov a presne (teoreticky) vypočítať zrýchlenie tohto algoritmu.

Literatúra

1. Baeza-Yates R., Ribeiro-Neto B.: Modern Information Retrieval. Addison Wesley (1999).
2. Blair D.C.: An Extended Relational Document Retrieval Model. Inf. Process & Management, **24**, 1988.
3. Brown E., Callan J.P., Croft W.B., Moss J.E.B.: Supporting Full-Text Information Retrieval with a Persistent Object Store. In 4th EDBT Conference, 2004.
4. Crawford R.: The Relational Model in Information Retireval. Journal of the American Society for Information Science, 1981, 51–64.
5. Grossman D.A., Driscoll J.R.: Structuring Text within a Relation System. In 3rd Intl. Conference on Datatabase and Expert System Applications, 1992.
6. Heaps J.: Information Retrieval – Computational and Theoretical Aspects. Academic Press, 1978.
7. Macleod I.: A Relational Approach to Modular Information Retrieval Systems Design. In Proceedings of the ASIS Annual Meeting, 1978, 83–85.

8. Lencses R.: Indexovanie pre plnotextový stroj využívajúci relačný databázový model. Proc. Workshop ITAT 03, Sliezsky dom, Slovensko, September 2003.
9. Ribeiro-Neto B., Moura E.S., Neubert M.S., Ziviani N.: Efficient Distributed Algorithms to Build Inverted Files. In 22th ACM Conf. on R&D in Information Retrieval, 1999.
10. Tvrdlík P.: Parallel Systems and Algorithms. ČVUT, 1997.

Analýza efektivity provádění dotazu ve vícerozměrných datových strukturách

Pavel Moravec, Michal Kolovrat, Michal Krátký, and Václav Snášel

Katedra informatiky, FEI, VŠB – Technická Univerzita Ostrava
17. listopadu 15, 708 33 Ostrava–Poruba, CZ
{pavel.moravec,michal.kolovrat,michal.kratky,vaclav.snasel}@vsb.cz

Abstrakt V příspěvku se zabýváme porovnáním různých vícerozměrných datových struktur z hlediska jejich efektivity a schopností indexovat vícerozměrná data. Zatím totiž nebylo nikde komplexně otestováno, kterou vícerozměrnou datovou strukturu lze nejvýhodněji použít pro konkrétní typ úlohy a konkrétní datovou sadu. Pro tuto analýzu a testování jsou v příspěvku použity datové struktury *BUB-strom*, *pyramidový strom* a *R-strom*. Budeme zkoumat efektivitu datové struktury vzhledem k použité datové sadě pro různé dimenze. Výsledky testů mají dokázat, že efektivita použití konkrétní datové struktury se nezhoršuje jen s rostoucí dimenzí, ale převážně závisí na použité datové sadě. V tomto článku také ukážeme, že pro obecná data není žádná ze zmíněných datových struktur odolná vůči prokletí dimenzionality, i přes případné tvrzení jejích autorů.

Klíčová slova: R-strom, BUB-strom, pyramidový strom, porovnání, dotazování, vícerozměrné datové struktury

1 Úvod

Při práci s multimediálními daty často musíme řešit problém, jak je ukládat a rychle v nich vyhledávat. Multimediální data proto většinou reprezentujeme jako vektory ve vysocedimenzionálním prostoru. V případě textů jde o vektory *termů* v dokumentech, u obrázků ukládáme buď celý vektor vzniklý z matice obrázku seřazením řádků za sebe, nebo *vektor příznaků* obrázku. Obdobně můžeme postupovat pro zvukové soubory, časové řady či semistrukturovaná data, např. XML [7].

Vzhledem k velké dimenzi vektorů je datová matice velká a sekvenční prohledání časově náročné, zejména není-li možno umístit celou matici v operační paměti. V případě nestrukturovaných textů jsou sice vektory dosti řídké a lze použít řádkové (*CCS*) či sloupcové (*CRS*) komprese *matice termů v dokumentech*, ale v tomto případě nastávají problémy s nestejnou délkou záznamu pro jednotlivé vektory a se stránkováním této matice na disku.

Již dlouhou dobu jsou hledány efektivní implementace *indexů*, které by umožnily vyloučit předem část nerelevantních dat a urychlit tak vyhledávání. Většinu z nich můžeme zařadit do jedné či více následujících kategorií:

1. Indexy vycházející z B^+-stromu, které uchovávají ve svých uzlech jedinou hodnotu, indikující určitou vlastnost. Jako příklad lze uvést *pyramidové stromy* [1]; *NB-stromy* [5], počítající Euklidovskou normu vektorů; metody iMinMax a iDistance [12], a mnohé další.
2. Stromové indexy vycházející z hierarchického dělení prostoru na regiony, které uchovávají v mezilehlých uzlech informace o vnořených regionech a v listových uzlech jednotlivé vektory či reference na ně, zejména *R-stromy* [6] a z nich odvozené struktury a *UB-* a *BUB-stromy* [9], [4], ukládající v uzlech adresu bodu na *Z-křivce*.
3. Indexy vycházející z aproximace určitého intervalu hodnot několikabitovou hodnotou; při prohledání se nejprve zkontroluje aproximace a teprve při kladném výsledku jsou prohledávány odpovídající vektory. Do této oblasti patří mj. *VA-files* [3], *A-stromy* [10] a *VQ-index* [11].

Každá z těchto struktur je vhodná pro určitou distribuci dat a na jiné zcela selhává. Všechny však ve větší či menší míře trpí takzvaným „*prokletím dimenzionality*“ – se vzrůstající dimenzí datových vektorů jejich efektivita klesá a od určité dimenze je často efektivnější projít a porovnat všechny vektory v kolekci.

V tomto příspěvku srovnáme chování některých z těchto struktur z první a druhé skupiny, jmenovitě pyramidových stromů, R-stromů a BUB-stromů nad nejrůznějšími distribucemi dat oproti *sekvenčnímu průchodu celou kolekcí*.

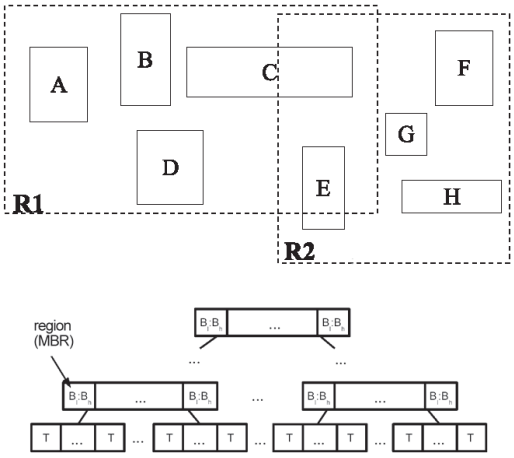
V druhé kapitole stručně popíšeme použité datové struktury, vybrané datové sady a předpokládané výsledky, ve třetí metodiku testování a ve čtvrté pak experimentální výsledky testů. V závěru zhodnotíme dosažené výsledky.

2 Indexování vícerozměrných dat

2.1 Vícerozměrné datové struktury

R-strom a R*-strom - jedná se o vícerozměrnou datovou strukturu, kde dochází k dělení datového prostoru na oblasti ohraničené geometrickými útvary (nejčastěji hyperkvádry). Tímto dělením lze zúžit prohledávaný prostor a zmenšit tak počet testovaných bodů.

Klasický R-strom je rozšířením známého B-stromu za účelem podpory prostorových dat. Má stejné vlastnosti jako B-strom, je vyvážený a garantuje efektivní využití paměťových stránek (min. 50%). V listových uzlech R-stromu jsou již uloženy prostorové objekty. Mohou to být body nebo další útvary. Tyto objekty jsou ohraničeny MBR (Minimal Bounding Rectange). Vždy musí platit, že objekty z jiných datových stránek nepatří do tohoto MBR. Do vnitřních uzlů se ukládají MBR ohraničující všechny potomky, nazývané regiony. Regiony celého datového prostoru se mohou i nemusí vzájemně překrývat (podregiony se již však překrývat nesmí). Region může také pokrývat část datové stránky, patřící do jiného regionu. Grafické znázornění regionů a R-stromu je na obrázku 1. Datové stránky jsou označeny A-H, regiony R1 a R2. S chováním regionů souvisí dva pojmy: Pokrytí úrovně stromu (celková plocha regionů na dané úrovni R-stromu) a přesah na úrovni stromu (součet všech ploch, které současně pokrývají alespoň



Obr. 1. Regiony a obecná struktura R-stromu.

2 regiony na stejné úrovni stromu). Pro efektivní zpracování dotazu je důležité, aby oba tyto ukazatele byly co nejmenší.

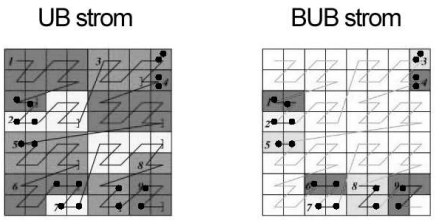
Z analýzy R-stromu vychází R*-strom. Byly zde rozpoznány další faktory, ovlivňující výkon. Okraj datové stránky (součet délek všech jejích stran) musí být co nejmenší a je nutné lepší využití kapacity stránek (zlepšením se zmenší výška stromu). Při různém pořadí vkládání bodů dostáváme různé stromy. Na tomto principu je založena myšlenka opětovného vkládání z podtečených datových stránek. Tímto postupem se zlepší výkon stromu při zpracování dotazů.

UB-strom a BUB-strom - jsou často používané struktury pro indexování vícerozměrných dat. Využívají Z-uspořádání pro transformaci n-rozměrných bodů na bitový řetězec.

Důležitým pojmem je zde Z-region, což je prostor pokrývající část Z-křivky specifikované intervalem $[\alpha, \beta]$. Body patřící do jednoho regionu jsou uloženy ve stejné diskové stránce. Za adresu celého Z-regionu volíme Z-adresu posledního bodu v intervalu (β). (α) je adresa (β) předcházejícího Z-regionu. Adresy regionů jsou ukládány do vnitřních uzlů B-stromu. V listových uzlech jsou uloženy jednotlivé body.

Rozdíl mezi UB a BUB stromem spočívá v tom, že UB strom obsahuje celý prostor se všemi regiony. BUB strom obsahuje pouze ty regiony, ve kterých se nachází n-rozměrné body, viz obrázek 2.

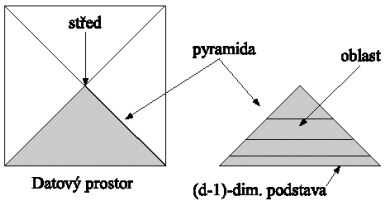
Pyramidový strom - základní myšlenka této struktury je opět (stejně jako u předchozích) založena na převodu n-rozměrného bodu na jednu hodnotu tzv. pyramidovou, která slouží jako adresa původního bodu. Tyto hodnoty se pak vkládají do klasického B^+-stromu (jedná se o binární strom, kde všechny cesty



Obr. 2. Rozdíl mezi Z-regiony.

od kořene k listům mají stejnou délku a optimální rozdělení stromu je zachováno pomocí automatické reorganizace položek stromu). Převod n-rozměrného prostoru se skládá ze dvou kroků. Nejprve se prostor rozdělí na pyramidy, a pak se jednotlivé pyramidy rozdělí na oblasti. V prvním kroku dělení se prostor rozdělí na 2n pyramid (podstava pyramidy je tvořena (n-1)-rozměrným útvarem ležícím na povrchu datového prostoru a vrchol pyramidy je ve středu). Střed je klasický geometrický střed normalizovaného prostoru (všechny souřadnice jsou převedeny na rozsahu 0-1). Souřadnice středu tedy jsou (0,5; 0,5; ..; 0,5). Pro každý bod existuje algoritmus, kterým lze určit, ke které pyramidě daný bod náleží. Konkrétně, všechny body ležící uvnitř jedné pyramidy mají jednu hodnotu souřadnice vzdálenou více od středu než druhou. Vzdálenost od středu se bere pouze v jedné souřadnici a musí být maximální pro všechny body pyramidy. Následně je třeba jednotlivé pyramidy očíslovat a dále rozdělit na oblasti (tvořící listové uzly stromu). Oblasti tvoří řezy vedené rovnoměrně s podstavou. Počet řezů a velikost se odvíjí podle počtu bodů. Schéma rozdělení prostoru na pyramidy a rozdělení pyramid na oblasti je na obrázku 3.

Pyramidová hodnota lze určit jako součet čísla pyramidy a maximální vzdálenost bodu (v jedné dimenzi) od středu. Strom konstruujeme tak, že pyramidovou hodnotu použijeme jako klíč a bod vkládáme do B^+-stromu.



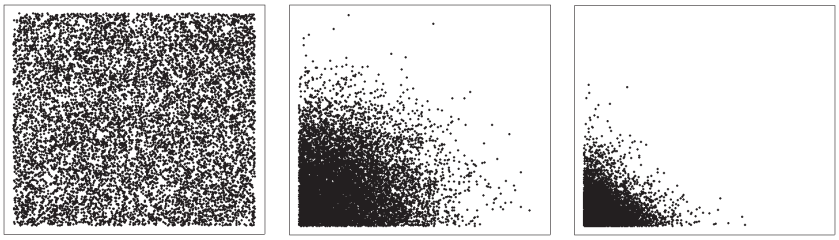
Obr. 3. Dělení prostoru na pyramidy a oblasti.

2.2 Datové sady

Jelikož se snažíme nalézt silné a slabé stránky jednotlivých datových struktur a zhodnotit jejich odolnost proti „prokletí dimenzionality“, musíme použít celé řady různých distribucí dat pro jejich zhodnocení.

Datová struktura, jež se jeví na první pohled jako výhodná, použijeme-li v jejích testech dat s rovnoměrným rozložením, může být na jiných datech zcela nepoužitelná. Příkladem může být NB-strom, který indexuje vícerozměrná data na základě Euklidovské metriky. Pokusíme-li se do něj ukládat normalizované vektory vah termů v dokumentech, jsme předem odsouzeni k nezdaru, protože Euklidovská metrika všech vektorů bude totožná.

V našich testech se proto zaměříme na uměle vytvořené datové sady, a to zejména na sady obsahující náhodná data s různým rozložením (rovnoměrným, Gaussovským a exponenciálním) (obrázek 4) a sady obsahující shluky (kulové a čtvercové) (viz obrázek 5 a)–b)). Zejména rovnoměrně rozložená data a data, obsahující shluky, jsou často využívána autory zejména v případě aproximačních struktur (VA-files) a jednohodnotových charakteristik (iMinMax), kde je na nich dosaženo velmi dobrých výsledků.

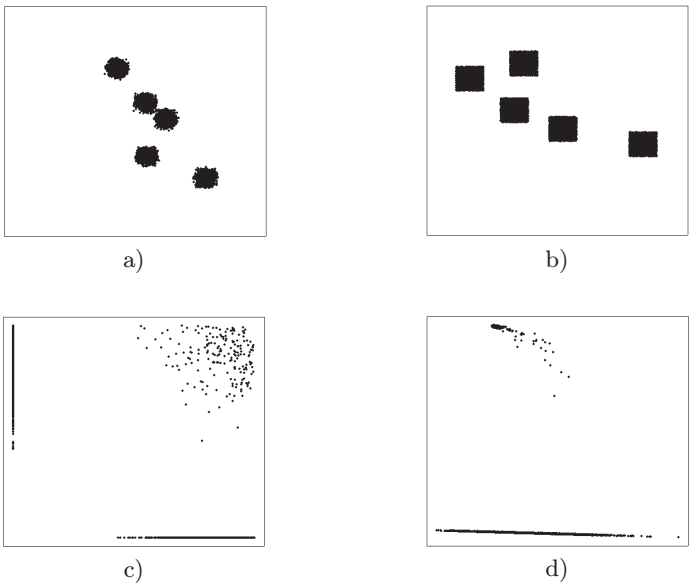


Obr. 4. Rozložení dat: a) rovnoměrné, b) Gaussovo, c) exponenciální.

Dále využijeme sadu s daty blízko stěn a hran hyperkvádru – na obrázku 5 c) – která simuluje chování typické pro vektorový model textových dokumentů, kde je vektor dokumentu velice řídký.

Na závěr provedeme testy na skutečných datech, získaných redukcí dimenze kolekce textových dokumentů pomocí *indexování latentní sémantiky*[1] (*LSI*). Rozptyl hodnot složek vektoru ve vyšších dimenzích výrazně klesá, každá další dimenze je o něco méně významná než předchozí, ale snižuje chybu aproximace. První dvě dimenze složek získaných výpočtem LSI nad kolekcí textových dokumentů jsou zobrazeny na obrázku 5 d).

[1] Bližší informace o této metodě lze nalézt v např. v [2].



Obr. 5. Data: a) kulové a b) čtvercové shluky; c) poblíž stěn hyperkvádrů, d) získaná jako výsledek LSI na kolekci textů.

2.3 Předpokládané chování datových struktur

Na základě znalosti postupu tvorby datových struktur a jejich popisu autory je možno předpokládat, že na rovnoměrně rozložených datech a shlucích se budou tyto struktury chovat efektivněji než sekvenční průchod kolekcí (a porovnání všech hodnot).

Horší situace nastane, budou-li data v prostoru rozložena nerovnoměrně a nebudou-li tvořit vhodně tvarované shluky. Zejména v případě aproximačních struktur s lineárním rastrem, struktur založených na dělení prostoru s možnosti překryvu regionů a struktur založených na výpočtu jediné hodnoty (pyramidový strom, NB-strom, atd.) může dojít k situaci, kdy bude prohledán celý index.

Protože každá hierarchická datová struktura nutně potřebuje doplňkové údaje pro tvorbu hierarchie, bude výsledný index větší, než původní matice vektorů. Proto může v některých případech nastat situace, kdy bude při prohledávání indexu načteno více stránek z disku (či proběhne více porovnání), než při sekvenčním průchodu celou kolekcí.

3 Popis metodiky testování

Pro účely testování byly vytvořeny datové sady, obsahující náhodná data a shluky. Každá z těchto sad obsahovala 100000 bodů, přičemž nejvyšší dimenze prostoru byla 30.

V prvním testu byly vytvořeny sady s rovnoměrným, Gaussovským a exponenciálním náhodným rozložením. Pro druhý test bylo vygenerováno 50 malých a 10 velkých kulových a čtvercových shluků. Ve třetím testu byla vytvořena sada dat rozložených u stěn a hran prostoru, čímž bylo simulováno chování textových dokumentů[2].

Mimo uměle vytvořených sad bylo také spočítáno LSI na 10000 dokumentech z kolekce TREC (Los Angeles Times 01/89 a 01/90) a výsledná redukovaná matice konceptů v dokumentech byla použita ve čtvrtém testu.

Všechny datové sady byly zaindexovány (velikost uzlu ve stromu byla 2 kiB) a byl měřen průměrný počet diskových přístupů (DAC). Pro účely testů bylo provedeno 1000 různých *rozsahových dotazů* v rozmezí 0,1 až 0,5 velikosti hrany hyperkvádru, obsahujícího data kolekce. Hyperkvádr dotazu byl určen tak, aby výsledek dotazu obsahoval alespoň jeden existující bod a nedocházelo k prohledávání prázdného prostoru. Výsledný počet přečtených 1 kiB stránek byl poté vztažen k počtu diskových přístupů při sekvenčním prohledávání kolekce.

Testy probíhaly na počítači s procesorem Intel Celeron 2,4 GHz, čipovou sadou SiS-650, 512 MiB DDR RAM a 40 GiB diskem. Pro testy bylo využito vlastní implementace datových struktur ve frameworku ATOM. Při prohledávání BUB-stromu bylo vyžito vylepšené metody DRU (bližší informace jsou ve článku [8]).

4 Výsledky testů

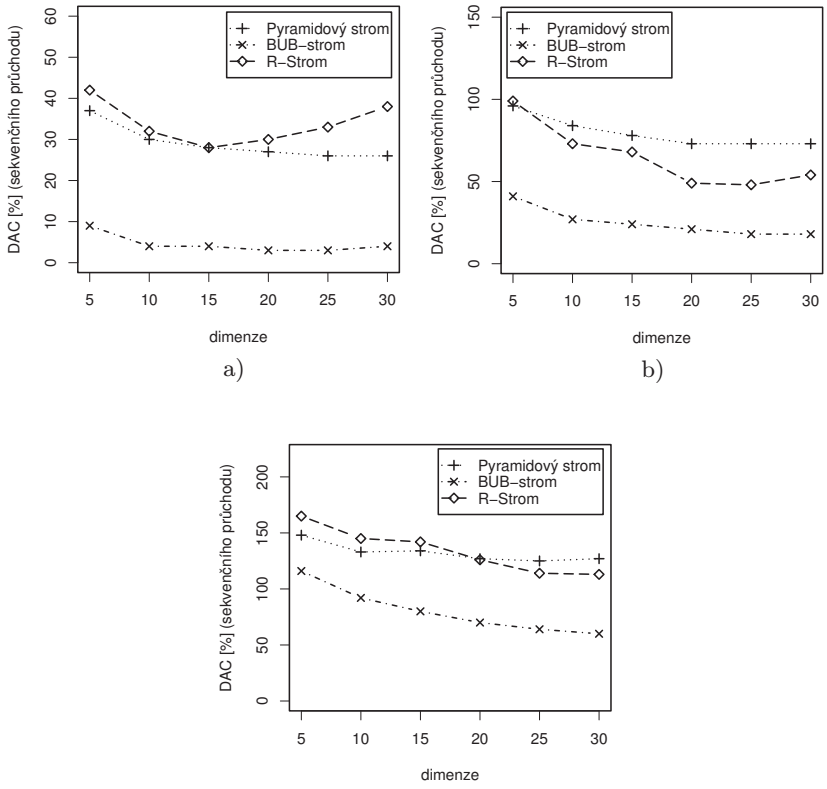
Výsledky prvního testu nad náhodnými daty jsou shrnuty v obrázku 6.

Jak je z obrázku vidět, všechny tři struktury přinášejí na rovnoměrně rozložených datech výrazné zlepšení oproti sekvenčnímu přístupu (100%). Při nerovnoměrném rozložení se ale chování zhoršuje – zatímco u Gaussova rozložení se počet načtených stránek u R-stromu a pyramidového stromu blíží chování sekvenčního průchodu (zejména pro malé dimenze, kde je ve stránkách obsaženo velké množství hodnot), u exponenciálního rozložení jediný BUB-strom překonává sekvenční průchod, zatímco ostatní struktury jsou o 20 - 60% horší díky nutnosti čtení stránek mezilehlých uzlů.

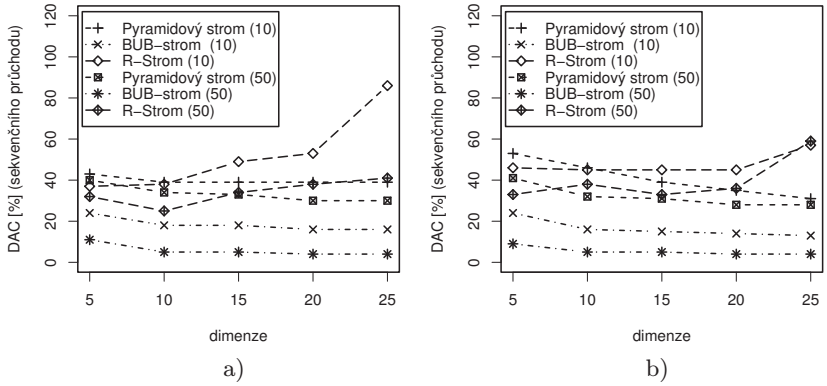
Ve druhém experimentu byla testovány kolekce, obsahující shluky dat. Výsledky jsou shrnuty v obrázku 7, čísla v závorkách udávají počty shluků v kolekci. Efektivita je opět lepší než u sekvenčního průchodu, ovšem s rostoucí dimenzí si lze v případě R-stromu povšimnout prudkého nárůstu počtu porovnávaných stránek, který je způsoben již zmíněným „prokletím dimenzionality“.

Ve třetím experimentu byla testována data u stěn a hran hyperkvádru. Výsledky jsou shrnuty v obrázku 8 a). Zatímco BUB-stomy a R-stromy si s touto kolekcí poradí velmi dobře, pyramidový strom dosahuje nejen špatných výsledků, ale jeho efektivita se z rostoucí dimenzí zhoršuje – je proto zřejmé, že i přes tvrzení autorů není pyramidový strom na obecné kolekci odolný proti „prokletí dimenzionality“.

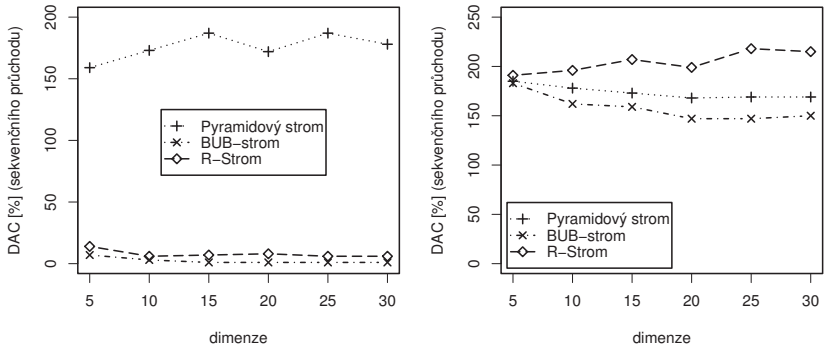
[2] Je ale nutno připomenout, že u textů je poměr nulových složek vektorů výrazně vyšší a dimenze se pohybuje i v řádech statisíců.



Obr. 6. Rozsahové dotazy: a) rovnoměrné, b) Gaussovo a c) exponenciální rozložení.



Obr. 7. Rozsahové dotazy: a) kulové a b) čtvercové shluky dat.



Obr. 8. Rozsahové dotazy: a) data u stěn a hran prostoru; b) textová data po LSI.

V posledním testu byla použita kolekce dat, získaných po LSI. Jak je vidět z obrázku 8 b), zatímco na vygenerovaných datech BUB-stromy vždy překonávaly sekvenční průchod, v tomto případě byl výsledek také výrazně horší.

Horší výsledky některých testů u dimenzí 5 a 10 (grafy 6 a 7) jsou zčásti zaviněny příliš vysokým počtem položek v uzlech. Počet diskových stránek přiřazených uzlu nebyl ale z důvodů jednotného testovacího prostředí měněn.

5 Závěr a výhled do budoucna

Na základě testů bylo ukázáno, že žádná ze struktur není nad obecnými daty ve všech případech lepší než sekvenční průchod, a že v některých případech se „prokletí dimenzionality“ a s ním spojený problém (nutnost projít celou datovou strukturu) uplatní. Jak se ukazuje v případě BUB-stromu, je nutno testovat tyto struktury i nad reálnými daty.

Některým případům, kdy je daná datová struktura neefektivní, se lze vyhnout změnou algoritmu pro výpočet adresy, podle níž se indexuje a změnou metody štěpení stránek pro určité sady dat. Problémem zůstává, jakým způsobem pak vyhodnotit, do které kategorie data spadají a určit příslušný způsob štěpení automaticky. Jednou z alternativ je použití histogramu vzdáleností vzorku dat – při testech se ale bohužel ukázalo, že i dvě velmi odlišné datové sady (rovnoměrné rozložení a Gaussovo rozložení) mají ve vysokých dimenzích histogramy Euklidovských vzdáleností prakticky totožné, přesto je na nich dosahováno velmi odlišných výsledků. Při použití jiných metrik zůstává otázka použití histogramů stále otevřená.

V budoucnu bychom rádi rozšířili testy i na další vícerozměrné datové struktury (A-strom, iMinMax, atd.) a doplnili výsledky testů o počty porovnání bodů a míry selektivity dotazů.

Reference

1. Berchtold S., Böhm C., Kriegel H.-P.: The Pyramid-Technique: Towards Breaking the Curse of Dimensionality. In Proc. Int. Conf. on Management of Data, ACM SIGMOD, 1998, 142–153.
2. Berry M., Browne M.: Understanding Search Engines, Mathematical Modeling and Text Retrieval. Siam, 1999.
3. Blott S., Weber R.: A Simple Vectorapproximation File for Similarity Search in High-Dimensional Vector Spaces, 1997.
4. Fenk R.: The BUB-Tree. In Proceedings of 28rd VLDB International Conference on VLDB, 2002.
5. Fonseca M.J., Jorge J.A.: Indexing High-Dimensional Data for Content-Based Retrieval in Large Databases. In Proceedings of the 8th International Conference on Database Systems for Advanced Applications (DASFAA 2003), Kyoto, Japan, 2003, 267–274.
6. Guttman A.: R-Trees: A Dynamic Index Structure for Spatial Searching. In Proceedings of ACM SIGMOD 1984, Annual Meeting, Boston, USA, ACM Press, June 1984, 47–57.
7. Krátký M., Pokorný J., Snášel V.: Implementation of XPath Axes in the Multi-Dimensional Approach to Indexing XML Data. In Accepted at International Workshop DataX, Int'l Conference on EDBT, Heraklion - Crete, Greece, LNCS, Springer-Verlag, 2004.
8. Krátký M., Skopal T.: Benchmarking the UB-Tree. In Proceedings of DATESO 2003, 2003, 83–94.
9. Ramsak F., Markl V., Fenk R., Zirkel M., Elhardt K., Bayer R.: Integrating the UB-tree into a Database System Kernel. In Proceedings of 26th VLDB International Conference, Morgan Kaufmann, 2000.
10. Sakurai Y., Yoshikawa M., Uemura S., Kojima H.: The A-tree: An Index Structure for High-Dimensional Spaces Using Relative Approximation. In Proc. of the 26th International Conference on Very Large Data Bases (VLDB), Cairo, September 2000, 516–526.
11. Tuncel E., Ferhatosmanoglu H., Rose K.: Vq-Index: An Index Structure for Similarity Searching in Multimedia Databases. In Proceedings of the tenth ACM international conference on Multimedia, 2002, 543–552.
12. Yu C.: High-Dimensional Indexing. Springer–Verlag, LNCS 2341, 2002.

Sequential and parallel genetic algorithms for k-planar graph drawing*

Matthew C. Newton and Ondrej Sýkora

Department of Computer Science, Loughborough University, Loughborough, Leicestershire LE11 3TU United Kingdom
{m.c.newton,o.sykora}@lboro.ac.uk

Abstract. We propose sequential and parallel genetic algorithms for finding low crossing drawings if a graph is drawn in two or more planes. These are the first practical algorithms to solve the problem. The algorithms are actually designed for a stronger model, with fixed vertices and rectilinear edges. Comparison of the algorithms for the case of two planes shows the superiority of the parallel island genetic algorithm.

1 Introduction

We call a graph G *biplanar* [2], if one can write $G = G_1 \cup G_2$, where G_1 and G_2 are planar graphs. Let $\mathrm{cr}(G)$ denote the standard crossing number of the graph G, i.e. the minimum number of crossings of its edges over all possible drawings of G in the plane, under the usual rules for drawings and crossing numbers [12], [14]. Motivated by printed circuit boards, Owens [9] introduced the *biplanar crossing number* of a graph G, that we denote by $\mathrm{cr}_2(G)$. By definition $\mathrm{cr}_2(G) = \min\{\mathrm{cr}(G_1) + \mathrm{cr}(G_2)\}$, where the minimum is taken over all unions $G = G_1 \cup G_2$. A *biplanar drawing* of a graph G is defined as the drawings of two sub-graphs, G_1 and G_2, of G, on two disjoint planes under the usual rules for drawings and crossing numbers, such that $G_1 \cup G_2 = G$. Note that one can always realize $\mathrm{cr}_2(G)$ by drawing the edges of G_1 and G_2 on two different sides of the same plane, while identical vertices of G_1 and G_2 are placed in identical locations on the plane on the two sides. Owens described a biplanar drawing of the complete graph K_n with $\mathrm{cr}_2(K_n) \leq 7n^4/1536 + O(n^3)$. One can define $\mathrm{cr}_k(G)$ similarly for any $k \geq 2$, making G a union of k sub-graphs. So for $k \geq 2$, the k-planar crossing number is defined as

$$\mathrm{cr}_k(G) = \min\{\mathrm{cr}(G_1) + \mathrm{cr}(G_2) + \cdots + \mathrm{cr}(G_k)\},$$

where the minimum is taken over all edge disjoint sub-graphs $G_i = (V, E_i)$, $i = 1, 2, ..., k$, so that $E = E_1 \cup E_2 \cup ... \cup E_k$. Determining $\mathrm{cr}_k(G)$ would have application to the design of multilayer VLSI circuits [1], but perhaps the case

* This research was supported in part by the EPSRC grant GR/R37395/01 and by the VEGA grant No. 02/3164/23.

$k = 2$ is the most interesting, and even this simplest case has not been explored in depth.

Some exact results for biplanar crossing number are in [4] and for $k-$planar crossing numbers in [11]. In [11] an algorithm is also designed for k-planar drawing of general graphs, which is the only previously existing algorithm for k-planar drawings we know of. The $k-$planar crossing number problem is related to the *thickness* and *book crossing number problems*. The *thickness* $\Theta(G)$ of G is the minimum number of planar graphs whose union is G. By definition, $\mathrm{cr}_k(G) = 0$ if and only if $\Theta(G) \leq k$, i.e. G is *$k-$planar*. The nature of the crossing number and the biplanar crossing number problems seems different, since testing whether $\mathrm{cr}(G) = 0$ can be done in linear time, while testing biplanarity is an NP-complete problem [6]. Surveys on biplanar graphs and the thickness problem can be found in [2], [7].

In a similar way that the k-planar crossing number was defined, one can define the *$k-$planar fixed vertices rectilinear crossing number* $\overline{\mathrm{cr}^*}_k(G)$, where vertices are placed in the same places in all planes and edges are drawn as straight line segments. Note, that this definition is a different from the definition of the rectilinear $k-$planar crossing number $\overline{\mathrm{cr}}_k(G)$, introduced in [11]. Obviously $\mathrm{cr}_k(G) \leq \overline{\mathrm{cr}}_k(G) \leq \overline{\mathrm{cr}^*}_k(G)$.

For the practical purposes of this article we can simplify our representation of the problem and work with planar fixed-vertices rectilinear drawings where the edges are coloured by k colours (one colour corresponds to one plane).

Genetic algorithms (GAs) are randomised optimisation heuristics that are based loosely on the paradigm of natural selection. While the exact mechanisms behind natural evolution are not very well known, some aspects have been studied in considerable depth. It is believed that chromosomes are the information carriers and that the evolution process works at the chromosome level through reproduction. The reproduction can be made by either combining chromosomes from the parents to produce offspring, a process called crossover, or by a random change occurring in the chromosome pattern, termed mutation.

A GA first creates an initial population of solutions. The solutions are then evaluated, using an application-specific criteria of fitness, to characterise them from most fit to least fit. A subset of the population is selected, using criteria that tend to favour the most fit solutions. This subset is then used to produce a new generation of offspring solutions. Finally, a number of solutions in this new generation are subjected to random mutations. The processes of selection, crossover and mutation are then repeated. A drawback of GAs is that the optima of these problems are generally unknown and it is therefore difficult to assess their performance. Another drawback is that GAs need a simple fitness function with a reasonably fast evaluation, but this is often not possible. GAs are usually slow, especially because the fitness function evaluation takes a long time. They are, however, quite popular among the graph drawing community (see e.g. [15]).

In this paper, we focus on the design of genetic algorithms for producing low crossing $k-$planar fixed-vertices rectilinear drawings, although the experimental results use only $k = 2$. As we already mentioned, the only algorithm for the

problem is in [11]. It is based on finding special acyclic decompositions of a graph and although it is interesting theoretically, from a practical point of view it is difficult to be implemented (we do not know about any implementation of the algorithm). For this reason we could not compare our algorithm with any other results.

On the other hand, it is obvious that some of the graphs we use, meshes and cycles for example, can be drawn with no crossings. The algorithms can be tested in a limited way with these graphs.

2 Sequential genetic algorithm

We start with a random population of solutions—genomes (i.e. placement of vertices in a plane and distribution of edges over k planes) $ind_1, \ldots, ind_{popsize}$. The fitness function is defined as the number of crossings of the drawing of the graph represented by positions of vertices and distribution of edges over k planes, where a smaller fitness value is better. Let $\overline{\mathrm{cr}^*}_k(ind_i)$ denote the number of same colour crossings in the individual ind_i.

Our crossover operation is as follows: two parental individuals, ind_1 and ind_2, are chosen randomly depending on their fitness value, where a lower value has a higher probability of being chosen. A rectangular area of random size of the individual ind_1 is chosen. The same rectangular area is taken from ind_2. Two new individuals, ind'_1 and ind'_2, are created such that ind'_1 contains the rectangular area from ind_2 and the edges incident with this area have the same colour as in ind_2. The rest vertices of ind_1 are in the same positions and the rest edges have the same colour as they had in ind_1. The individual ind'_2 contains the rectangular area from ind_1 with the incident edges of the same colour as in ind_1 with the rest being the positions of other vertices of ind_2 and rest edges with the same colour as it was in ind_2. The crossover operation is shown in Algorithm 1.

Algorithm 1 [Crossover operation]

Select two individuals, $parent_1$ and $parent_2$
Get *rectangular area* randomly
Copy vertices and edges in *rectangular area* from $parent_1$ to $child_2$
Copy vertices and edges in *rectangular area* from $parent_2$ to $child_1$
Complete $child_1$ and $child_2$ with the rest of the original individuals, $parent_1$ and $parent_2$

After creating *popsize* children, the step of mutation is executed. The mutation, which is either the swap of two randomly picked vertices of an individual or the altering of the colour of an edge, is done with some probability on each individual in the new population. Finally, if the best individual from the previous generation was better than the best individual in the new population, it is preserved by being copied over a randomly chosen individual in the new population. Mutation is shown in Algorithm 2.

Algorithm 2 [mutation operation]

```
for (all permutations) do
  Randomly produce a number, p,
  if (p<prob) then
    Produce two different random positive integers r1, r2 ≤ |V|;
    Exchange the vertices r1 and r2 of the individual.
  end if
end for
```

The algorithm terminates when there has been no decrease in the best crossing number for five seconds, or a drawing has been found with zero crossings.

3 Parallel genetic algorithms

Parallel computing has been a valuable tool for improving running time and enlarging feasible sizes of problems and it has become a key economic and strategic issue. Strong efforts are put into developing standards for parallel programming environments, such as PVM (Parallel Virtual Machine) [10], MPI (Message Passing Interface) [8], BSP (Bulk Synchronous Parallel) [3] and HARNESS [5]. Among these, one of the most common is currently MPI.

In this section, we only decribe the differences between the sequential algorithm and two parallel genetic algorithms. Algorithm 3 shows the general scheme of the parallel genetic algorithms used.

Algorithm 3 Basic Parallel Genetic Algorithm

```
Start up N parallel tasks
repeat
  Process one iteration on each task
  Organise transfers of data between tasks
until root declares we have finished
```

Based on Algorithm 3, we introduce two parallel algorithms, described in the following sections. The first uses a new 'mesh' model of computation, and the second uses the standard distributed computing 'island' model.

3.1 'Mesh' model parallel algorithm

There are two distinct components to the k-planar fixed vertices rectilinear drawing problem. The first is where to place each vertex; the second, on what plane to place each edge. Each individual in the standard genetic algorithm contains all information representing one solution. The idea with the algorithm presented here is to divide the algorithm up into two separate, but closely linked, parts, each attempting to solve the vertex and edge components of the problem.

Let us use N computers in our parallel genetic algorithm, where $N = 2 \times popsize$ and $popsize$ is the size of the population on each processor. We divide the processors into two groups: $N/2$ processors use the genetic algorithm to find only the positions of the vertices, while the other $N/2$ processors find only colours for the edges.

In total, there are $popsize^2$ individuals in the population. It is possible to imagine these arranged in a square, as shown in Figure 1. Processors A, B, C, and D deal with vertices; the individuals in processor A's population, for example, are 1, 2, 3 and 4 in the diagram. Processors a, b, c, and d find edge colours. Processor a, for example would have individuals 1, 5, 9, and 13 in its population.

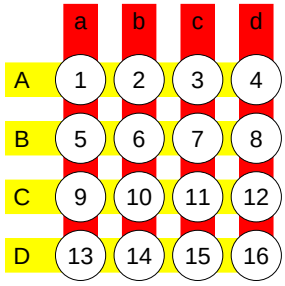


Fig. 1. Mesh processor arrangement (eight processors).

As processors a, b, c, and d need to know the vertex positions to calculate fitness values, they are sent these values from processors A, B, C, and D. Similarly, A, B, C, and D need to know edge colours, which they are sent from a, b, c, and d. This synchronization happens after each iteration in the genetic algorithm.

Processor A, therefore, only alters the position of the vertices in its individuals, and is given the edge colours from processors a, b, c, and d.

The algorithm terminates when there has been no change in the best overall number of crossings for five seconds.

3.2 'Island' model parallel algorithm

The island model is a commonly used system for parallelization, especially on distributed parallel computers. Each processor in the system is viewed as an island, on which work is done. Boats sometimes travel between islands, however, and therefore work is combined and knowledge transferred.

In this particular algorithm, each processor runs the sequential algorithm. After each iteration there is a small possibility that two processors will share some data; this happens by two processors swapping one individual with each other. The individuals to be sent are chosen according to the same probability as when parents are chosen for crossover. The implementation used here has a 1% chance that two of the available processors, chosen at random, will swap individuals after each iteration. One of the processors, the *root*, co-ordinates this swapping as well as performing its genetic algorithm.

The root task also has the responsibility of deciding when to finish the algorithm. As with the other algorithms, this is when the crossing number over all processors has not decreased in the previous five seconds.

Figures 2 and 3 show the drawing of a 5×5 mesh before and after drawing. In this case the sequential algorithm achieved 0 crossings, the optimum. The two planes are shown with thick grey lines and thin black lines.

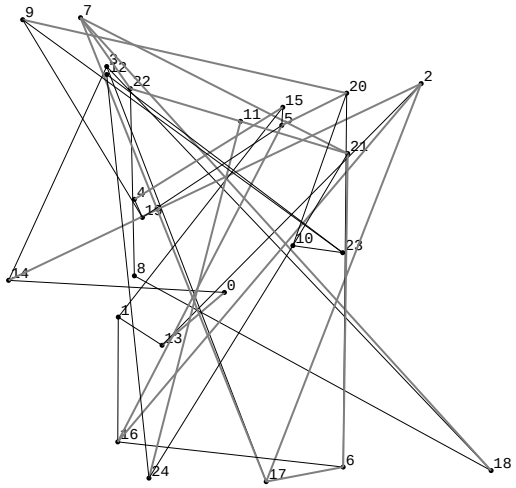


Fig. 2. Drawing of 5×5 mesh before processing, 85 crossings.

4 Test results

The programs were written in C, and used MPI as the parallel communications library. The experiments were run on an SGI Altix multiprocessor machine, each computing node being an Intel Itanium running at 1.3Ghz. The entire system has 30Gb memory.

Both parallel algorithms used eight processors, and the island model had a population size of eight (the mesh model had to use the fixed size of $(\frac{8}{2})^2 = 16$ individuals in total, as in Figure 1). The sequential algorithm was run on the same system, and also used a population size of eight. For these experiments we use $k = 2$. Any experiment that took more than three minutes running time were stopped.

We used graphs that include a selection of randomly generated graphs, graphs generated from images, graphs from the US National Institute of Standards and Technology, and some graphs with structural symmetries, such as meshes, cycles and hypercubes. Each program was run on every graph three times, and an average taken. All graphs had up to 200 vertices.

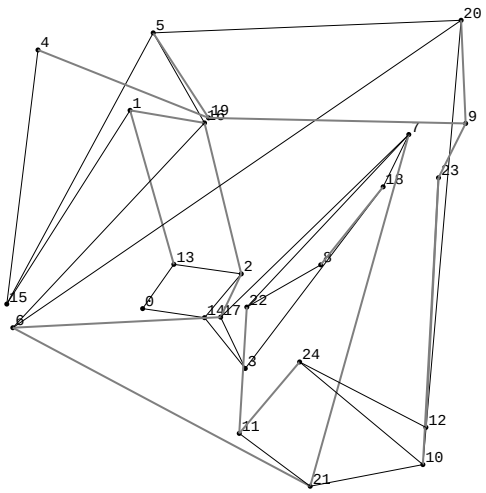


Fig. 3. Drawing of 5×5 mesh after processing, 0 crossings.

The experiment results show that the parallel island genetic algorithm provides better results than the parallel mesh genetic algorithm. In the experiments we observed an interesting phenomenon: the sequential algorithm running time was comparable with the parallel island algorithm running time, but the drawings produced by the parallel island algorithm were much better, i.e. they contained fewer crossings. One explanation could be that the parallel algorithm spent the same running time more efficiently—whilst one processor could run a long time without improving the best individual, the parallel algorithm improved it more often.

5 Conclusions

In this paper we designed a sequential genetic algorithm—in fact the first practical algorithm—for producing low crossing k-planar drawings (although it is for a stronger model with fixed vertices and rectilinear edges). We also designed two different parallel genetic algorithms for the same problem and compared them for the case of $k = 2$. Comparison of all three algorithms shows the superiority (from the point of view of the quality of the produced drawing) of the parallel island genetic algorithm.

References

1. Aggarwal, A., Klawe, M., Shor, P., Multi-layer grid embeddings for VLSI, *Algorithmica* **6**, (1991), 129–151.

2. Beineke, L. W., Biplanar graphs: a survey, *Computers and Mathematics with Applications* **34** (1997), 1–8.
3. Bulk Synchronous Parallel, `http://www.bsp-worldwide.org/implmnts/oxtool/`, 10 September 2004
4. Czabarka, E., Sýkora, O., Székely, L.A., Vrťo, I., Biplanar crossing numbers I: A survey of results and problems, in: *Finite and Infinite Combinatorics*, (T. Fleiner, G. O. H. Katona, eds.), Bolyai Society Mathematical Studies, Akadémia Kiadó, Budapest (accepted in 2002), to appear.
5. Dongarra, J., Geist, A., Kohl, J.A., Papadopoulos, P.M., Sunderam, V. HARNESS: Heterogeneous Adaptable Reconfigurable Networked Systems, *Oak Ridge National Laboratory*, `http://www.csm.ornl.gov/harness/`
6. Mansfield, A., Determining the thickness of graphs is NP-hard, *Mathematical Proceedings of the Cambridge Philosophical Society* **9** (1983), 9–23.
7. Mutzel, P., Odenthal, T., Scharbrodt, M., The thickness of graphs: a survey, *Graphs and Combinatorics* **14** (1998), 59–73.
8. Message Passing Interface, `http://www-unix.mcs.anl.gov/mpi/`, 29 June 2004
9. Owens, A., On the biplanar crossing number, *IEEE Transactions on Circuit Theory* **18** (1971), 277–280.
10. Parallel Virtual Machine, `http://www.csm.ornl.gov/pvm/`, 29 June 2004
11. Shahrokhi, F., Sýkora, O., Székely, L.A., Vrťo, I., Bounds and methods for k-planar crossing numbers, in: Proc. *11th Intl. Symposium on Graph Drawing, Lecture Notes in Computer Science* **2912**, Springer, 2003, 37–46.
12. Shahrokhi, F., Sýkora, O., Székely, L. A., Vrťo, I., Crossing numbers: bounds and applications, in: *Intuitive Geometry*, Bolyai Society Mathematical Studies **6**, (I. Bárány and K. Böröczky, eds.), Akadémia Kiadó, Budapest, 1997, 179–206.
13. Shahrokhi, F., Sýkora, O., Székely, L. A., Vrťo, I., The book crossing number of graphs, *J. Graph Theory* **21** (1996), 413–424.
14. Székely, L. A., A successful concept for measuring non-planarity of graphs: the crossing number, to appear in *Discrete Math.*
15. Utech, J., Branke, J., Schmeck, H., Eades, P., An evolutionary algorithm for drawing directed graphs, in Proc. *of the Int. Conf. on Imaging Science, Systems and Technology,* CSREA Press, 1998, 154–160.

Generalized linear list automata*

Martin Plátek[1], Petr Jančar[2], and Jörg Vogel[3]

[1] Charles University, Department of Computer Science
Malostranské nám. 25, 118 00 Praha 1, Czech Republic
platek@ksi.ms.mff.cuni.cz
[2] Technical University of Ostrava, Department of Computer Science
17. listopadu 15, 708 33 Ostrava, Czech Republic
Petr.Jancar@vsb.cz
[3] Friedrich Schiller University, Computer Science Institute,
07740 Jena, Germany
vogel@informatik.uni-jena.de

Abstract. We follow some previous studies of list automata and restarting automata and introduce a generalized and refined model – a two-way generalized linear list automaton (GLLA) and its subclasses defined by sets of allowed operations. Motivation for the model comes from (computational) linguistics, mainly from the need to express syntactic constraints. We also present several subclasses of GLL-automata, providing some variants and extensions of the Chomsky hierarchy. Our technical results include comparing the expressive power of automata having only move-to-the-right and delete-to-the-left operations with the class of (D)CFL ((deterministic) context-free languages); in particular we show an infinite hierarchy inside DCFL, defined by the increasing size of the read/write lookahead window.

1 Introduction

In order to enhance theoretical basis for development of syntactic analyzers and searchers, we follow some previous studies of list and restarting automata (see [1], [7], [3], [6]) and introduce a generalized model – a *two-way generalized linear list automaton with a look-ahead window* (GLLA). We argue that the new model is more flexible for modelling the syntactic analysis of natural languages (compare [8], [9], [10], [5]), and for developing several software tools for linguistic modelling.

A GLLA M is a device with a finite state control and a read/write window of fixed size. This window moves (in both directions) on a tape (i.e., list of items) containing a word delimited by sentinels. M can decide (in general nondeterministically) to replace the contents of the window: it can delete some items from the list (moving the head in one or the other direction), insert some items into the list and/or rewrite some items.

* The work is supported by the Grant Agency of the Czech Republic, Grant-No. 201/02/1456, and Grant-No. 201/04/2102.

We summarize results for several subclasses of GLLA obtained by reducing the set of operations, and we also provide some variants and extensions of the Chomsky hierarchy of languages.

Our technical results also include comparing the expressive power of automata having only move-to-the-right and delete-to-the-left operations with the class of (D)CFL ((deterministic) context-free languages); in particular we show an infinite hierarchy inside DCFL, defined by the increasing size of the read/write lookahead window.

2 Definitions

We start by giving a definition of generalized linear-list automata.

A *generalized linear-list automaton with a lookahead window*, GLLA for short, is a one-tape machine given by a tuple $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$, where Q is a finite set of *states*, Σ is a finite *input alphabet*, Γ is a finite *working alphabet* containing Σ; $¢, \$ \notin \Gamma$ are the markers (*sentinels*) for the left and right border of the work space, respectively, $q_0 \in Q$ is the *initial state*, $k \geq 1$ is the size of the *read/write* (look-ahead) *window*, and δ is a finite set of *instructions* of the type

$$(q, u) \longrightarrow (q', z)$$

where $q, q' \in Q$ and $u \in \mathcal{PC}^{\leq k}$; by $\mathcal{PC}^{\leq k}$ we mean the set $(¢ \cdot \Gamma^{k-1}) \cup \Gamma^k \cup (\Gamma^{\leq k-1} \cdot \$) \cup (¢ \cdot \Gamma^{\leq k-2} \cdot \$)$ of (all) *possible contents* of the read/write window of M. The parameter z in the instructions is one of the special symbols

$$M_r, M_l, W(v), d_R(v), d_L(v), I(v), \mathsf{Accept}, \mathsf{Reject}$$

where the indicated symbols are accompanied by a word v satisfying the constraints described below.

A *configuration* of M is a string $\alpha q \beta$ where $q \in Q$, and either $\alpha = \lambda$ (λ denoting the empty word) and $\beta \in ¢ \cdot \Gamma^* \cdot \$$ or $\alpha \in ¢ \cdot \Gamma^*$ and $\beta \in \Gamma^* \cdot \$$; here q represents the current state, $\alpha\beta$ is the current contents of the tape, and it is understood that the head scans the first k symbols of β or all of β when $|\beta| < k$; the list item containing the first symbol of β is viewed as the *position* of the head. An *initial configuration* is of the form $q_0 ¢ w \$$, where $w \in \Sigma^*$.

A *computation* (i.e., a sequence of configurations) is generated by the instruction set δ, as explained further:

1. A *move-right step* $(q, u) \longrightarrow (q', M_r)$ assumes $u \neq \$$. If M is in state q and is scanning u in its read/write window then this step causes M to shift the (position of the) read/write window by one to the right and to enter state q'.
2. A *move-left step* $(q, u) \longrightarrow (q', M_l)$ assumes $u \notin ¢ \cdot \Gamma^*$. It causes M to shift the (position of the) read/write window by one to the left and to enter state q'.
3. A *rewrite step* $(q, u) \longrightarrow (q', W(v))$ assumes that $|v| = |u|$ and the sentinels are at the same positions in u and v (if at all). It causes M to replace the contents u of the read/write window by the string v, and to enter state q'. The head does not change its position.

4. A *delete-right step* $(q, u) \longrightarrow (q', d_R(v))$ assumes that v is a proper subsequence of u, containing all sentinels in u. It causes M to replace u with v (by deleting excessive items-symbols) and to enter state q'; the new position of the head is on the first item of v (the contents of the window is thus 'completed' from the right; the position distance to the right sentinel decreases).
5. A *delete-left step* $(q, u) \longrightarrow (q', d_L(v))$ assumes that v is a proper subsequence of u, containing all sentinels in u. It causes M to replace u with v (by deleting excessive items-symbols), to enter state q' and to shift the head position by $|u| - |v|$ items to the left – but to the left sentinel ¢ at most. (Thus the contents of the window is 'completed' from the left; the distance to the left sentinel decreases if the head position was not at ¢ already.)
6. An *insert step* $(q, u) \longrightarrow (q', I(v))$ assumes that u is a proper subsequence of v (keeping the obvious sentinel constraints). It causes M to replace the scanned u by v (by inserting the relevant items), and to enter state q'. The head position is shifted by $|v| - |u|$ to the right (the distance to the left sentinel increases).
7. An *accept step* $(q, u) \longrightarrow (q', \mathsf{Accept})$ causes M to halt and accept. The state q' can provide a type of accepting.
8. A *reject step* $(q, u) \longrightarrow (q', \mathsf{Reject})$ causes M to halt and reject. The state q' can provide a type of rejecting.

We consider only finite computations, which end either by accepting or by rejecting.

In general, automaton M is *nondeterministic*, that is, there can be two or more instructions with the same left-hand side (q, u), and thus there can be more than one computation for an input word. If this is not the case, the automaton is *deterministic*.

An input *word* $w \in \Sigma^*$ *is accepted by* M, if there is a computation which, starting with the initial configuration $q_0 ¢ w \$$, finishes by executing an Accept instruction. By $L(M)$ we denote the language consisting of all input words accepted by M; we say that M *recognizes (accepts) the language* $L(M)$.

We work further with finite computations, which end by accepting or rejecting operations.

Now we define some subclasses of GLLA that are relevant for our further investigations. We will characterize the individual classes of GLLA by (restrictions of) the set of operations, which they will use. I.e. $\{M_r, d_L\}$-automaton means a GLLA which uses M_r, d_L-operations only.

Notation. For brevity, the prefix det- will be used to denote the property of being deterministic. For any class A of automata, $\mathcal{L}(\mathsf{A})$ will denote the class of languages recognizable by the automata from A. Let k be a natural number. $\mathcal{L}(\mathsf{k\text{-}A})$ will denote the class of languages recognizable by the automata from A which uses the lookahead window of the size k.

By $\subset$ we denote the proper subset relation. Further we sometimes use regular expressions instead of the corresponding regular languages. Throughout the paper, λ denotes the empty word and $\mathbb{N}_+$ denotes the set of all positive integers.

Without loss of generality we will consider only automata with finite computations.

3 Characterizations of the Chomsky hierarchy

At first we repeat the observations from [1] in our terms.

Proposition 1. – $\mathcal{L}(1\text{-det-}\{M_r\})$ *coincides with the class of* regular languages.
– $\mathcal{L}(1\text{-}\{M_r, W, d_L\})$ *coincides with* CFL *(the class of context-free languages).*
– $\mathcal{L}(1\text{-}\{M_r, M_l, W\})$ *coincides with* CSL *(the class of context-sensitive languages).*
– $\mathcal{L}(1\text{-det-}\{M_r, M_l, W, I\})$ *coincides with the class* RE *(of recursively enumerable languages).*

Proposition 2. – $\mathcal{L}(1\text{-det-}\{M_r, W, d_L\})$ *coincides with* DCFL *(the class of deterministic context-free languages).*
– $\mathcal{L}(1\text{-det-}\{M_r, M_l, W\})$ *of languages coincides with* DCSL *(the class of deterministic context-sensitive languages).*

The previous characterizations use 'maximally constrained' versions of GLLA. We can easily extend these characterizations to 'minimally constrained' GLLA (mainly using techniques from [1]):

Corollary 1. – *The following classes of languages coincide with the class of* regular languages*:* $\mathcal{L}(\{M_r, M_l\})$, $\mathcal{L}(\{M_r, W, d_R, I\})$.
– *The following class of languages coincides with* CFL *(the class of context-free languages):* $\mathcal{L}(\{M_r, W, d_L, d_R, I\})$.
– *The following class of languages coincides with* CSL *(the class of context-sensitive languages):* $\mathcal{L}(\{M_r, M_l, W, d_R, d_L\})$ *(inserting is not allowed).*
– *The class* $\mathcal{L}(\{M_r, M_l, W, I, d_R, d_L\}\})$ *coincides with the class* RE *(of recursively enumerable languages).*

Corollary 2. – *The following class of languages coincides with* DCFL *(the class of deterministic context-free languages):* $\mathcal{L}(\text{det-}\{M_r, W, d_L, d_R, I\})$.
– *The following class of languages coincides with* DCSL *(the class of deterministic context-sensitive languages):* $\mathcal{L}(\text{det-}\{M_r, M_l, W, d_R, d_L\})$ *(inserting is not allowed).*

Note. Considering all constraints being between a 'minimal' and a 'maximal' one, we could obtain several other characterizations of (deterministic and nondeterministic variant of) the Chomsky hierarchy. The robustness of these characterizations is due to the unlimited size of the lookahead-windows.

4 An infinite hierarchy inside DCFL

We first recall some results from [7] and [2] (where (D)CFL denotes the class of (deterministic) context-free languages):

Theorem 1. *$\mathcal{L}$(1-det-$\{M_r, d_L\}$) is a proper subset of* DCFL, *and $\mathcal{L}$(1-$\{M_r, d_L\}$) is a proper subset of* CFL.

Proof sketch. Any (deterministic) 1-$\{M_r, d_L\}$-automaton M can be simulated by a (deterministic) pushdown automaton in a straightforward way (the part of the list of M to the left of the head can be stored in the push-down). On the other hand, the language

$$L_1 = \{\, d^{n_1} a_1 d^{n_1} d^{n_2} a_2 d^{n_2} \dots d^{n_p} a_p d^{n_p} c a_p a_{p-1} \dots a_1 \mid$$

$$p \geq 0,\ a_j \in \{a, b\},\ n_j \geq 0 \text{ for } j = 1, 2, \dots, p \,\}$$

is (obviously) in DCFL but is not recognizable by any 1-$\{M_r, d_L\}$-automaton. For details see [2].

We now show a (nontrivial) generalization of the result in the deterministic case. Namely we show that increasing the size of the lookahead window increases the recognition power of det-$\{M_r, d_L\}$-automata; thus we get an infinite hierarchy inside DCFL.

Theorem 2. *For each $i = 1, 2, 3, \dots$, $\mathcal{L}$(i-det-$\{M_r, d_L\}$) is a proper subset of $\mathcal{L}$($(i+1)$-det-$\{M_r, d_L\}$) and a proper subset of* DCFL.

Proof. Any i-det-$\{M_r, d_L\}$-automaton can be simulated by a deterministic pushdown automaton in a straightforward way – similarly as in the case $i = 1$ mentioned in Theorem 1 (the contents of the lookahead window can be stored in the finite control unit).

So the theorem will be proven when we define, for each i, language L_i which is recognized by an $(i+1)$-det-$\{M_r, d_L\}$ automaton but not recognizable by any i-det-$\{M_r, d_L\}$ automaton.

Language L_1 from the proof sketch of Theorem 1 is a basis of the desired hierarchy; generally we define (using notation ^{R} for the reverse image):

$$L_i = \{\, d^{n_1} w_1 d^{n_1} d^{n_2} w_2 d^{n_2} \dots d^{n_p} w_p d^{n_p} c (w_p)^R (w_{p-1})^R \dots (w_1)^R \mid$$

$$p \geq 0,\ n_j \geq 0,\ w_j \in \{a_1, a_2, \dots, a_{2^i}\}^i \text{ for } j = 1, 2, \dots, p \,\}$$

We particularly note that all w_j's have length i and contain symbols from an alphabet with cardinality 2^i (in the case $i = 1$, we have words of length 1 over a 2-symbol alphabet).

We first sketch that each L_i is recognizable by an $(i+1)$-det-$\{M_r, d_L\}$ automaton. Since its window is longer than i (i.e. the length of w_j), such an automaton can first move to w_1, then stepwise delete d^{n_1} from both left-hand and right-hand sides of w_1, then (after encountering ¢) move to w_2, delete d^{n_2} on both

sides, etc. Finally it cuts around c, thus verifying that the remaining words (from $\{a_1, a_2, \ldots, a_{2^i}\}^*$) are the reverse image of each other. The automaton can easily recognize (and reject) when the input word is not of the form required by L_i.

Now we assume (for the sake of contradiction) that an i-det-$\{M_r, d_L\}$ automaton M recognizes L_i (for some fixed i). For technical convenience, we can suppose that M always deletes the whole input word before accepting.

By the *first phase* we mean the part of the computation from the beginning till the first configuration when c appears in the window.

Suppose now that for any number r there is an accepting computation of M such that at least r occurrences of d remain from an original sequence d^{n_j}. For large r (where 'large' could be specified by a function of size of M), it is clear that the d's near the middle of such a segment d^r are visited a bounded number of times during the whole computation (the bound depending just on the size of M): this is easily deducible from the fact that M is deterministic and has to behave "regularly" inside a (long) d-segment.

But then there are two items in this segment d^r which have the same histories: the *history* of an list item can be defined as a sequence of triples (q, u, m), where q is a state, u the window content, and m the position of the relevant item in the window – the triples are given in the order as they appear in the computation.

It can be easily verified that the original d-segment between two such items (with the same histories) can be pumped without affecting acceptance – which is obviously a contradiction (with the assumption $L(M) = L_i$).

So we know that there is a constant r_0 such that the length of each d-segment remaining after the first phase of an accepting computation is lesser than r_0.

Let us now consider the computation of M on a word

$$d^{n_1} w_1 d^{n_1} d^{n_2} w_2 d^{n_2} \ldots d^{n_p} w_p d^{n_p} c (w_p)^R (w_{p-1})^R \ldots (w_1)^R \qquad (1)$$

where $r_0 << n_1 << n_2 << \ldots << n_p$ (where $<<$ means 'much lesser than'). M will shorten all d-segments to the length lesser than r_0 in the first phase. Recalling that M is deterministic, and its computation must not allow the 'pumping' as mentioned above, it is a routine to verify that M has to 'cut around' w_1; and thus w_1 must be shortened (since it is of the same length as the window). Moreover, (the principle part of) this 'cutting around' has to be performed by a computation cycle – and this cycle necessarily shortens both d-segments on the left-hand side and the right-hand side of w_1 by the same length (otherwise we could easily construct a word outside L_i which would be accepted by M).

On a word (1), such a cycle can only finish by encountering ¢; so the left-hand d^{n_1} has been shortened to the length which is lesser than the number of the symbols deleted from w_1. Then 'cutting around' w_2 necessarily follows, where the relevant cycle finishes by encountering the first non-d symbol on the left. This 'cutting around' w_j's continues, and the first phase finishes with possibly leaving the final d-segment of length lesser than r_0.

We now observe that for fixed p and $n_1, n_2, \ldots, n_p$ we have

$$(2^i)^{ip} = 2^{i^2 p}$$

words of the form (1). The first phases of the computations of M on these words finish with some prefixes uc, whose total number can be bound by

$$\big((2^i)^{i-1} + (2^i)^{i-2} \cdot 2 + (2^i)^{i-3} \cdot 3 + \ldots + (2^i)^1 \cdot (i-1) \big)^p \cdot r_0$$

which is surely less then

$$2^{(i^2-i+1)p} \cdot r_0 = \frac{2^{(i^2p)}}{2^{(i-1)p}} \cdot r_0$$

For sufficiently large p, $2^{(i-1)p}/r_0$ exceeds the number of states of M, and we can then derive the existence of two words of the form (1), with different sequences $w_1 w_2 \ldots w_p$, such that in both cases the first phase finishes in the same state and with the same prefix uc left on the list. But this can be immediately shown to contradict the assumption $L(M) = L_i$ (we can switch the suffixes behind c between those two words). □

We strongly conjecture that the union of the above infinite hierarchy is a proper subset of DCFL. E.g., the language

$$L = \{ a^n bwcw^R u \mid u, w \in \{a, b\}^*, |u| = 2n \}$$

is obviously in DCFL but it seems not recognizable by any det-$\{M_r, d_L\}$-automaton (but we have not found a precise proof for this).

We can note that the above language L can be recognized by a nondeterministic $\{M_r, d_L\}$-automaton; the question if DCFL $\subseteq \mathcal{L}(\{M_r, d_L\})$ is more unclear.

5 Further remarks, future work

The original aim of introducing GLL-automata was to increase the power of list automata and restarting automata for implementation of constraints of syntactic nature, aiming at development of individual modules of syntactic analyzers (mainly of natural languages), or at searching syntactic phenomena in a syntactically un-annotated corpus (or text).

We briefly recall the mentioned *restarting automata*. The most general so far studied type of restarting automaton is the (two-way) restarting automaton, RLWW-automaton for short. It is a device M with a finite-state control and a read/write window of a fixed size. This window moves along a tape containing a word delimited by sentinels executing move-right and move-left instructions until its control decides (nondeterministically) that the content of the window should be rewritten by some shorter string. After a rewrite, M can continue to move its window until it either halts and accepts, or halts and rejects, or restarts, that is, it moves its window to the leftmost position, and reenters the initial state, in this way reaching a 'restarting configuration.' Each part of a computation of M between an initial configuration or a restarting configuration and the next restarting configuration is called a 'cycle.' Thus, each computation of M can be described through a sequence of cycles. In fact, M cannot only be considered as

a device for accepting a language, but it can also be interpreted as a 'rewriting system,' as each cycle replaces a factor of its initial tape content by a shorter factor, in this way performing a rewrite of the tape content.

Also various restricted versions of the restarting automaton have been considered. The RRWW-automata, which can only move their window from left to right along the tape, and the RWW-automata, which are in addition required to perform a restart step immediately after executing a rewrite operation are of particular interest. Then there is the RRW-automaton, which only uses the letters of the input alphabet in its rewrite operations. A further restriction leads to the RR-automaton, where each rewrite operation simply deletes some letters from the content of the read/write window. Obviously the restriction on the restart operation and the restrictions on the rewrite operation can be combined leading to the RW-automaton and the R-automaton.

Further a *monotonicity* property was introduced for RLWW-automata which is based on the idea that from one cycle to the next in a computation, the actual place where a rewrite is performed does not increase its distance from the *right* end of the tape. The monotone restarting automata essentially model bottom-up one-pass parsers. As it turned out the monotone RLWW- and RWW-automata characterize the class CFL of context-free languages, while the monotone deterministic RRWW- and RWW-automata as well as several restricted versions thereof all characterize the class DCFL of deterministic context-free languages [3]. Also a generalization of the notion of monotonicity was introduced, which models the generalization from bottom-up one-pass parsers to bottom-up multi-pass parsers [11]. For an integer $j \geq 1$, a restarting automaton is called *j-monotone* if, for each of its computations, the corresponding sequence of cycles can be partitioned into at most j subsequences such that each of these subsequences is monotone. It is shown in [11] that by increasing the value of the parameter j, the expressive power of the (nondeterministic) restarting automata without auxiliary symbols is increased.

The previous types of constraints can be further refined and combined, using the features and constraints introduced here for GLL-automata. We thus hope to obtain much richer tools for expressing syntactical constraints, which is our motivating target.

References

1. Chytil M.P., Plátek M., Vogel J.: A Note on the Chomsky Hierarchy. Bulletin of the EATCS 27, 1985.
2. Jančar P., Mráz F., Plátek M.: Forgetting Automata and Context-Free Languages. Acta Informatica **33**, Springer-Verlag, 1996, 409–420.
3. Jančar P., Mráz F., Plátek M., Vogel J.: On Monotonic Automata with a Restart Operation. J. Autom. Lang. Comb. **4**, 1999, 287–311.
4. Lopatková M., Plátek M., Kuboň V.: Analysis by Dependency Reduction for Natural Languages. Accepted for ITAT 2004.

5. Plátek M., Lopatková M., Oliva K.: Restarting Automata: Motivations and Applications. In: Proceedings of the Workshop "Petrinetze" and 13. Theorientag 'Formale Sprachen und Automaten', Holzer M. (ed.), Institut für Informatik, 'Technische Universität München, 2003, 90–96.
6. Otto F.: Restarting Automata and Their Relations to the Chomsky Hierarchy. In: Z. Esik, Z. Fülöp (eds.): Developments in Language Theory, Proceedings of DLT'2003, LNCS 2710, Springer, Berlin, 2003.
7. Plátek M., Vogel J.: Deterministic List Automata and Erasing Graphs. The Prague Bulletin of Mathematical Linguistics **45**, 1986.
8. Plátek M., Tichá B.: List Automata as Dependency Parsers. PBML 50, 1988, 71–87.
9. Oliva K., Plátek M.: List Automata with Syntactically Structured Output. COLING'88, Budapest, Hungary, **2**, 1988, 498–500.
10. Plátek M.: A Note on Parsers. The Prague Bulletin of Mathematical Linguistics **54**, 1990, 25–36.
11. Plátek M.: Two-Way Restarting Automata and j-Monotonicity. In: L. Pacholski, P. Ružička (eds.), SOFSEM'01, Proc., LNCS 2234, Springer, Berlin, 2001, 316–325.
12. Plátek M., Otto F., Mráz F., Jurdziński T.: Restarting Automata and Variants of j-Monotonicity. Mathematische Schriften Kassel, no. 9/03, 2003.
13. Sgall P.: Generativní popis jazyka a česká deklinace. Academia, Praha, 1967 (in Czech).
14. Sgall P., Hajičová E., Panevová J.: The Meaning of the Sentence in Its Semantic and Pragmatic Aspects. J. Mey (ed.), Dordrecht, Reidel and Prague, Academia, 1986.
15. Duchier D., Debusmann R.: Topological Dependency Trees: A Constraint-based Account of Linear Precedence. In: 39th Annual Meeting of the Association for Computational Linguistics, ACL 2001, Morgan Kaufmann Publ. Toulouse, France, 2001, 180–187.

Formal model of meta–information acquisition from information resources

Vojtěch Svátek[1] and Václav Snášel[2]

[1] Department of Information and Knowledge Engineering
University of Economics Prague, W. Churchill Sq. 4, 130 67 Praha 3, Czech Republic
svatek@vse.cz
[2] Department of Computer Science, Technical University of Ostrava
17. listopadu 15, 708 33 Ostrava-Poruba, Czech Republic
vaclav.snasel@vsb.cz

Abstract. An outline of formal model describing the acquisition of 'meta–information' on information resources is being proposed, which should enable to compare the quality of different analysis procedures. It is illustrated an example from website analysis.

1 Introduction

While the amounts of information resources available on the internet as well as in company intranets are exponentially growing, their retrieval is still predominantly based on brute-force methods such as full-text indexing. It is however clear that the presence of reliable *meta-information* can bring significant added value to resource discovery. Since it is not feasible to assign meta-information to each resource by hand (as in the traditional library indexing), various knowledge-based, statistical, algebraic and other methods have recently been designed for automatic acquisition of meta-information from the full content of the resources. Examples are *text classification* or *information extraction*, which are frequently applied to either plain text (e.g. articles) or websites. The research on (often quite similar) meta-information acquisition methods is actually fragmented among many different communities, such as that of information retrieval, (semi-structured) databases, automated semantic annotation (for the semantic web), library subject categorisation and the like. It is highly desirable to find a model that would enable to capture (at least partially) the essence of different approaches, to *compare* different methods and tools, and to determine their *complementarity* and/or *supplementarity* with respect to prototype tasks.

Currently, to the knowledge of the authors, all existing models of meta-information acquisition belong to two extreme categories:

1. Models restricted to a *single type* of meta-information. Typical approaches are *classification* of information resource to a particular, predefined category (e.g. subject topic), and *retrieval* of resource that fulfils a predefined role with respect to the given resource (e.g. the bibliography page within a personal website).

2. *Open* models enabling to specify an almost arbitrary structure of meta-information. A typical example is the apparatus of *semantic web ontology languages* such as OWL [4].

It seems highly desirable to fill in the blank space between the two extremes. Our recent work on *problem-solving models* for deductive web mining, in particular, the TODD framework [8], successfully unified apparently dissimilar information resource analysis methods. It however addresses the problem of (mostly) *vertical co-operation* between different analysis tools, while we need *horizontal alignment* of such tools.

In this paper, we propose a model that associates a resource with a collection of properties that belong to three types (closed-domain properties, object properties and content properties) distinguished by the range of their values. The evaluation of 'correct' value assignment differs according to property type, the results can however be aggregated into a single table. We hypothesise that such a model is particularly useful when multiple complementary/supplemantary procedures can be applied on different data structures related to same underlying entities. This is typical for websites (hence the choice of illustrative example), which are known to exhibit a high degree of redundancy in presentation, but the model can presumably be generalised to other types of resources, too.

In section 2 we explain the model itself. Section 3 presents an in-breadth example demonstrating the whole approach. Section 4 discusses some limitations of the approach and compares it with the web-ontology approach. Finally, section 5 wraps up the paper.

2 Resources, properties, values and meta–information

In this section we formally define the important notions and explain their role in the whole model.

Definition 1. *Let* $\mathcal{R}$ *be the* universe of information resources *(further only resources). Let* $R_t \subseteq \mathcal{R}$ *be a* set of resources *of same* type t*.*

An information resource can be any unique entity that carries some information content. In the case of web, it can be for example a physical web page, a hyperlink, or a whole website. 'Physical web page', 'hyperlink', and 'website' can be considered as types of resources.

Definition 2. *Let* $P = P_{CD} \cup P_O \cup P_{Cn}$ *be a set of* properties *relevant for a set of resources R;* P_{CD}*,* P_O *and* P_{Cn} *are pairwise disjoint.* P_{CD} *will be called* closed–domain properties*,* P_O object properties *and* P_{Cn} content properties*.*

Every p from $P_{CD} \cup P_{Cn}$ *has an associated* value set*,* V_p*. Every p from* P_O *has an associated resource type* t_p*.*

We assume that a fixed set of properties can characterise the resources (of a given type) from different aspects, e.g. to indicate the *author* of a given page,

the *direction* ('upward', 'downward' etc.) of a given hyperlink, or the *startup page* of a website. The first is an example of a content property, the second of a closed–domain property, and the third of an object property. While the value set of closed–domain properties is assumed as enumerated, the value set of content properties is derived from some standard open data type. For simplicity, let us assume that content properties have character strings for values—this is the typical type of target information in information extraction. The model can thus be understood as unifying, from a very particular viewpoint, the paradigms of *information retrieval* (values of object properties refer to retrieved resources), *text classification* (values of closed-domain properties can be understood as classes of resources) and *information extraction* (values of closed-domain properties correspond to semantically labelled text extracted from the content of resources).

Note that the model is not limited to meta–information in the usual sense, i.e. 'information about the resource itself', but extend it to any information that could be acquired from the resource and is 'somehow' related to it. Typically, it is the case of information about the entity that 'owns' and/or 'created' the resource, e.g. about the company a website is devoted to. This is not a conceptual problem, since such 'second–order' meta–information can be directly captured by *composed* properties, e.g. 'location–of–company–owning–the–site'.

Definition 3. *For each* $r \in R, p \in P$, *the* reference value *of* p *for* r *will be denoted as* $Ref(r,p)$.

- *If* $p \in P_{CD} \cup P_{Cn}$ *then* $Ref(r,p) \in V_p$.
- *If* $p \in P_O$ *then* $Ref(r,p) = r_k \in R_t$, *where* t *is the resource type associated with* p.

Note that we do not introduce the reference value in the sense of 'ontologically true' value, but just as the value to which other values would be compared. The reference value could be even 'N/A' (i.e., 'not available') if the real value cannot be derived from the resource content in principle.

Now, we will introduce the notion of *meta–information set*, which corresponds to a (possibly partially) filled 'template' over the set of properties.

Definition 4. *A* meta–information set *of resource* $r \in R$, $\mathcal{M}_r$, *is a set* $\{(p_1, v_1), (p_2, v_2), \ldots, (p_k, v_k)\}$, *where* $\{(p_1, p_2, \ldots, p_k)\}$ *are all properties relevant for* R. *A* reference meta–information set *of resource* $r \in R$, $\mathcal{M}_r^{Ref}$, *is a set* $\{(p_1, Ref(r,p_1)), (p_2, Ref(r,p_2)), \ldots, (p_n, Ref(r,p_n))\}$, *where* $\{(p_1, p_2, \ldots, p_n)\}$ *are all properties relevant for* R. *Every resource has* exactly one *reference meta–information set.*

For simplicity, we assume that a meta–information set always covers *all* properties, some of them may however have the value 'N/A'.

Definition 5. *In a given* context, *every* p *from* P_{Cn} *has an associated* value–acceptability relation $Acc_p \subseteq V_p \times V_p$.

The value–acceptability relation may, depending on the nature of the property, amount e.g. to:

- strict equality: $Acc_p(v, Ref(r,p))$ iff $v = Ref(r,p)$ (e.g for a company registration code)
- term permutation (e.g. for keyword lists)
- term superset with length restriction (e.g. for person names—if the page author is 'John Smith', we could possibly accept the value 'Dr. John Smith', sometimes even 'page written by John Smith' but not a long sentence).

The acceptability relation may not be fixed for the given property: it may vary according to 'context'. The notion of context may, in practice, correspond e.g. to an *evaluation session* for different meta–information acquisition procedures. We may thus bias the evaluation toward 'strictness' or 'sloppiness', and obtain coarser or finer distinctions of the procedures' quality. Alternatively, the context may be the *syntactical standard* for true meta–information.

Definition 6. *A meta–information set of resource $r \in R$, $\mathcal{M}_r$, is* correct *for r in property p if it contains a pair (p, v) such that:*

1. *if $p \in P_{CD} \cup P_O$ then $v = Ref(r,p)$*
2. *if $p \in P_{Cn}$ then $Acc_p(v, Ref(r,p))$.*

Note: We chose to require strict identity for properties from $P_{CD} \cup P_O$. In principle, we could introduce some 'acceptability relation' even for these. For example, relaxed criteria for object properties might require, in a certain context, merely sub/super–object (part–of) relation to hold (instead of identity), and similarly, for closed–domain properties, the sub/superclass relationship in a hierarchy could fulfil such role.

Definition 7. *Given two meta–information sets $\mathcal{M}_{r_i}$, $\mathcal{M}'_{r_i}$ of the same resource r_i, $\mathcal{M}_{r_i}$ is* superior *(or equal) to $\mathcal{M}'_{r_i}$ with respect to r_i, $\mathcal{M}_{r_i} \geq \mathcal{M}'_{r_i}$ if $\mathcal{M}_{r_i}$ is correct for r_i in every property in which $\mathcal{M}'_{r_i}$ is correct for r_i.*

This enables to compare the performance of two meta–information acquisition procedures on the same resource (e.g. web document).

3 Example

Let us demonstrate our scheme on a 'toy' example: a hypothetical web page of a small toy producer (Fig. 1).

Let us consider this page as our 'current resource' on which the procedures will be evaluated. Let the *reference meta-information set* wrt. this resource be:

$\mathcal{M}^{Ref}$ = { (p_1 = *page_type*, 'Main information page')
(p_2 = *page_author*, "John Black")
(p_3 = *name_of_company_referenced_by_page*, "BlackWood Ltd.")
(p_4 = *domain_of_competence_of_company_referenced_by_page*, "toys")
(p_5 = *pricelist_location*, '`/html/body/ul`')
(p_6 = *contact_address_location*, 'N/A') }

```
<html>
<head>
<meta author="John Black">
<meta description="Production and sale of wooden and textile toys">
</head>
<body>
<h1>BlackWood Ltd.</h1>
<p>An opportunity for lovers of original hand-carved and hand-knitted
toys. BlackWood specialises in toys for children aged over 6 years,
and in collectioners' items.</p>
<p>In our shop you can find:</p>
<ul>
<li>wooden animals from 4,-
<li>knitted dolls from 8,-
<li>toy furniture collection, special offer for just 120,-
</ul>
<hr>
<it>Page maintained by J. Black, last modification Feb 28, 2002.</it>
</body>
</html>
```

Fig. 1. Example resource: page of a toy producer.

p_1 is a closed–domain property, p_2, p_3 and p_4 are content properties, and p_5 and p_6 are object properties. The acceptability relations for content properties will be defined as follows:

- $Acc_2(x, y)$ holds (for sequences of terms) if x contains the last term from y and no more than three other terms.
- $Acc_3(x, y)$ holds (for sequences of terms) if x contains the first term from y and no more than one other term.
- $Acc_4(x, y)$ holds (for sequences of terms) if x contains all terms from y and less than $|y|$ other terms.

We will consider four *meta–information acquisition procedures* Pr_1, Pr_2, Pr_3 and Pr_4. For instructiveness, we will assume that[3]

- Pr_1 is a Naive Bayes page categoriser operating on unigram representation.
- Pr_2 is an extractor of META tag content, equipped with heuristics mapping META attributes on target meta–information properties.
- Pr_3 is a linguistic, parser–based information extractor equipped with a database of domain–neutral lexical indicators.
- Pr_3 is an HTML–and–punctuation–based information extractor equipped with a database of domain–neutral lexical indicators.

[3] These hypothetical procedures roughly correspond to some of the tools developed within the *Rainbow* project [7].

The meta–information sets produced by the procedures will be ($\mathcal{M}(k)$ denoting the output of the $k-$th procedure, for the given resource):

$\mathcal{M}(1) = \{$ (p_1 = *page_type*, 'Main information page')
(p_2 = *page_author*, 'N/A')
(p_3 = *name_of_company...*, 'N/A')
(p_4 = *domain_of_competence...*, 'N/A')
(p_5 = *pricelist_location*, 'N/A')
(p_6 = *contact_address_location*, 'N/A') }

(the keyword–based categoriser is clearly designed for classification only, and does not care about structural patterns; assignment to 'Main information page' might be due to the presence of several 'promotion' keywords such as 'opportunity', 'lovers' or 'original')

$\mathcal{M}(2) = \{$ (p_1 = *page_type*, 'N/A')
(p_2 = *page_author*, "John Black")
(p_3 = *name_of_company...*, 'N/A')
(p_4 = *domain_of_competence...*,
"Production and sale of wooden and metallic toys")
(p_5 = *pricelist_location*, 'N/A')
(p_6 = *contact_address_location*, 'N/A') }

(assuming that the meta–attribute 'description' is tentatively mapped on the property *domain_of_competence...*)

$\mathcal{M}(3) = \{$ (p_1 = *page_type*, 'N/A')
(p_2 = *page_author*, "J. Black")
(p_3 = *name_of_company...*, "BlackWood")
(p_4 = *domain_of_competence...*,
"toys for children aged over 6 years, collectioners' items")
(p_5 = *pricelist_location*, '`/html/body/ul`')
(p_6 = *contact_address_location*, 'N/A') }

(the value of *page_author* is identified with the subject that 'maintains' the object 'page'; the value of *name_of_company...* is identified with the subject that 'specialises' in *something*—which is, in turn, identified with the value of *domain_of_competence...*; finally, the *pricelist_location* is identified with the HTML element immediately following that with the phrase '...you can find')

$\mathcal{M}(4) = \{$ (p_1 = *page_type*, 'Main information page')
(p_2 = *page_author*, "Page maintained by J. Black")
(p_3 = *name_of_company...*, "BlackWood Ltd.")
(p_4 = *domain_of_competence...*, 'N/A')
(p_5 = *pricelist_location*, '`/html/body/ul`')
(p_6 = *contact_address_location*, 'N/A') }

(the *page_type* is recognised by presence of free–text paragraphs as well as a list—a mixture presumably typical for 'Main information page'; the value of *page_author* is identified with the slanted text in the page footer; the value of *name_of_company...* is identified with the text in the topmost header, containing

the pattern 'Ltd.'; finally, the *pricelist_location* is identified with the unordered HTML list containing estimated 'price–patterns' in each item).

Taking into account the acceptability relations for p_2, p_3 and p_4, the 'correctness scores' of the meta–information sets for the individual properties are as follows:

Property	Pr_1	Pr_2	Pr_3	Pr_4
page_type	1	0	0	1
page_author	0	1	1	0
name_of_company...	0	0	1	1
domain_of_competence...	0	0	0	0
pricelist...	0	0	1	1
contact...	0	0	0	0

From the table we can deduce that e.g.:

- $\mathcal{M}(4)$ is superior to $\mathcal{M}(1)$
- $\mathcal{M}(3)$ is superior to $\mathcal{M}(2)$

It is however clear that *complementarity* of procedures is more important than *superiority*. We can easily see that:

- no combination of procedures can correctly acquire all meta–information
- if we removed the 'inaccessible' $p4$ and 'inavailable' $p6$, the minimal combinations of procedures needed for correct acquisition would be $\{Pr_1, Pr_3\}$, $\{Pr_2, Pr_4\}$ and $\{Pr_3, Pr_4\}$.

Furthermore, it is more reasonable to remove the *closed–world assumption*, since ignorance should not be treated the same as error. The original table will then look as follows:

Property	Pr_1	Pr_2	Pr_3	Pr_4
page_type	1	?	?	1
page_author	?	1	1	0
name_of_company...	?	?	1	1
domain_of_competence...	?	0	0	?
pricelist...	?	?	1	1
contact...	?	?	?	?

We could also construct the table for *pairs* of procedures. Here, in the case where both procedures return a value, we can either demand, for a correct result, that *both* are correct, or just that *at least one* is correct.

With the first interpretation, the table looks as follows:

Property	Pr_1,Pr_2	Pr_1,Pr_3	Pr_1,Pr_4	Pr_2,Pr_3	Pr_2,Pr_4	Pr_3,Pr_4
page_type	1	1	1	?	1	1
page_author	1	1	0	1	0	0
name_of_company...	?	1	1	1	1	1
domain_of_competence...	0	0	?	0	0	0
pricelist...	?	1	1	1	1	1
contact...	?	?	?	?	?	?

We can see that, in the first (more natural?) interpretation, the combinations involving Pr_4 now become inferior, since its incorrect claim of knowing the value or *page_author* invalidates the correct suggestion of Pr_2 or Pr_3, respectively. The combination $\{Pr_1, Pr_3\}$ then appears as clear 'winner'.

With the second interpretation, the table looks as follows (changes in boldface):

Property	Pr_1,Pr_2	Pr_1,Pr_3	Pr_1,Pr_4	Pr_2,Pr_3	Pr_2,Pr_4	Pr_3,Pr_4
page_type	1	1	1	?	1	1
page_author	1	1	0	1	**1**	**1**
name_of_company...	?	1	1	1	1	1
domain_of_competence...	0	0	?	0	0	0
pricelist...	?	1	1	1	1	1
contact...	?	?	?	?	?	?

The three candidate combinations would then become 'equally correct' again.

4 Discussion

The present model strikes for certain balance between simplicity and coverage. However, due to stress on the former, many important aspects of real-world meta-information acquisition tasks remain uncovered. Among the most important limitations (and topics for future research) are probably the following:

- The model does not treat different methods of combining the results of multiple procedures for the same property (such as voting)
- No uncertainty is allowed in the results of the procedures.
- The properties are single-valued.
- The model is static, it does not anticipate possible change of property values over time.
- The model does not introduce homomorphism (similarity, deformation) for the results of multiple procedures, it is main problem for modelling meta-information

It might also be interesting to compare the typology of properties in our model with the inventory of *web ontology languages* such as OWL [4]. In OWL, content properties would correspond to *datatype properties with embedded data type*, while closed–domain properties would correspond to (object/datatype) *properties with enumerated class / data type*, respectively, since enumeration is possible, in principle, for both types of properties in OWL. Unlike web ontology languages, we however treat *class membership* simply as (closed–domain) properties, to keep the model general enough. Mapping (of semantically relevant elements) from OWL to our model seems to be rather straightforward: a structured ontology could be flattened to our model by replacing pairs of chained properties with new, composed, properties. Having done such mapping, we could then e.g. 'tap' on existing tools producing meta-information in the form of RDF triples [6].

In Kalfoglou [2] was used *CHU space* for Ontology Mapping. When we want model structure of meta-information (infomorphisms and homomorphisms) by Chu spaces see [5].

Chu space is a matrix over a set Σ. It is formalized in as follows:

Definition 8. *A Chu space $A = (A, r, X)$ over a set Σ, called the alphabet, consist of a set of points constituting the carrier A, a set X of states constituting cocarrier, and a function $r : A \times X \to \Sigma$ constituting the matrix.*

In our case it is matrix with rows corresponding to resources and columns corresponding to properties. Chu space entries are drowning from $\{0, ?, 1\}$. Where 0 indicates resources has not properties, ? indicates activity in progress and 1 indicates finished activity. Formally, carrier A is set of resources, cocarrier X is set of properties and alphabet is $\Sigma = \{0, ?, 1\}$.

5 Conclusions

We presented a model of meta-information acquisition from information resources, and illustrated it on an example from website analysis. Although the web is probably the most characteristic area of application for this kind of model, it could probably be applied to other types of resources.

Future work will address some of the limitations discussed in section 4. Since the model is currently merely descriptive, we examine some existing algebraic formalisms that could extend it with more rigorous semantics. We would also like to elaborate on the problem of transformation/mapping from ontology languages suggested in section 4; an adequate method could possibly be adapted from state-of-the-art research on ontology transformation [3] and alignment [1]. Finally, we plan to implement a simple software tool and test it on the top of a collection of *Rainbow* web services [7].

Acknowledgments

The research is partially supported by grant no. 201/03/1318 of the Grant Agency of the Czech Republic, “Intelligent analysis of the WWW content and structure”.

References

1. Doan A., Halevy A., Noy N.: Proceedings of the Semantic Integration Workshop Collocated with the Second International Semantic Web Conference (ISWC-03), online at http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS//Vol-82/.
2. Kalfoglou Y., Schorlemmer W.M.: IF-Map: An Ontology-Mapping Method Based on Information-Flow Theory. J. Data Semantics I 2003, 98–127.
3. Omelayenko B., Klein M.C.A. (Eds.): Knowledge Transformation for the Semantic Web. Frontiers in Artificial Intelligence and Applications **95**, IOS Press 2003.

4. OWL Web Ontology Language Reference, W3C Recommendation, 10 February 2004, http://www.w3.org/TR/owl-ref/.
5. Pratt V.R.: "Chu Spaces". Course notes for the School in Category Theory and Applications. Coimbra, Portugal, 1999.
6. Resource Description Framework (RDF), W3C, http://www.w3.org/RDF/.
7. Svátek V., Kosek J., Labský M., Bráza J., Kavalec M., Vacura M., Vávra V., Snášel V.: Rainbow - Multiway Semantic Analysis of Websites. In: 2nd International DEXA Workshop on Web Semantics (WebS03), Prague, IEEE 2003.
8. Svátek V., Labský M., Vacura M.: Knowledge Modelling for Deductive Web Mining. In: 14th International Conference on Knowledge Engineering and Knowledge Management (EKAW 2004), Whittlebury Hall, Northamptonshire, UK. Springer Verlag, LNCS, 2004.

Filtrace webových stránek Suffix Tree frázemi*

Roman Tesař and Karel Ježek

Katedra informatiky a výpočetní techniky, Západočeská univerzita,
Univerzitní 22, 30614 Plzeň
{romant, jezek_ka}@kiv.zcu.cz
http://www.kiv.zcu.cz/

Abstrakt Internet je v současné době běžným informačním médiem. Využívá ho stále více lidí nejen k práci, ale i k trávení volného času a zábavě. Není však žádný dohled nad jeho obsahem, jako je tomu u jiných médií, například televize nebo rozhlasu. Kdokoli zde může nalézt návody na výrobu trhavin nebo drog. Libovolné extremistické skupiny zde mohou prezentovat své názory a přesvědčení. Volně přístupné komukoliv jsou i pornografické stránky. Cílem našeho projektu se proto stalo vytvoření filtračního systému, který by umožňoval tento závadný obsah odhalit a ochránit tak uživatele od jeho vlivu. Vhodné použití by nalezl především v institucích jako jsou školy a univerzity nebo v bezpečnostních složkách. Důležitým krokem při jeho vývoji bylo nalezení účinného klasifikátoru textu, který by ke svému natrénování využíval pokud možno jen množinu závadných dat. Toho jsme dosáhli pomocí ohodnocených častých frází nalezených algoritmem Suffix Tree (ST-fráze). V článku popisujeme nejen způsob jejich získání, ale i možnosti dalšího zvyšování úspěšnosti klasifikace pomocí těchto ST-frází. Kromě samotné filtrace by měl vyvíjený systém umožňovat na základě již zachycených WWW stránek vyhledávat na Internetu další stránky se závadným obsahem, což je cíl, ke kterému bychom se rádi v konečné fázi vývoje systému dostali a na kterém již v současné době pracujeme.

Klíčová slova: klasifikace, filtrace textu, Internet, WWW stránky, ST-fráze, Suffix Tree, míra závadnosti, ohodnocení dokumentu, váha fráze

1 Úvod

Na Internetu je možné v současné době nalézt obrovské množství informací všeho druhu. To na jedné straně usnadňuje a zrychluje naší práci, na druhé straně to poskytuje možnost typ informací, které jsou v rozporu se zákonem, zneužít. Proto se stalo cílem jedné z výzkumných skupin[1] ze Západočeské univerzity v Plzni navržení systému pro real-time filtraci WWW stránek pomocí klasifikace jejich textového obsahu.

* Příspěvek vznikl za částečné podpory výzkumného záměru MSM 235200005 a ME494.

[1] http://www.kiv.zcu.cz/research/groups/text/index.php

V úvodní fázi návrhu jsme si kladli otázku, co lze vlastně považovat za závadný obsah a rozhodli jsme se zaměřit především na rasismus, toxikomanii, pornografii a terorismus. Pro jednotlivá témata bylo dále nutné stanovit, kdy je uživatelem přistupovaná stránka již závadná. Zde jsme zvolili princip přičítání negativního hodnocení přistupované stránky na základě slov a větných frází. Při překročení určité hranice, kterou si uživatel může sám nastavit, je stránka považována za závadnou a není zobrazena v internetovém prohlížeči.

Využití takového systému je poměrně široké, od filtrace internetových stránek se závadným obsahem (drogy, rasismus, pornografie, terorismus) ve školách až po vyhledávání závadných stránek na Internetu za účelem jejich zmapování a případného odstranění. Stejně jako existující obdobné systémy bude i náš systém vycházet z myšlenky již jednou prozkoumané WWW stránky ukládat do centrální databáze a tu potom prohledávat. To je samozřejmě (za předpokladu vhodně řešené struktury databáze a rychlého připojení) časově nejméně náročné řešení. Průběžně vytvářený seznam závadných stránek by mohl být případně (v rámci zákonů České republiky) k dispozici pro výzkumné účely.

Typickým zástupcem takového systému je například Proventia Web Filter[2] využívající databázi společnosti Cobion (viz [11]), která je největší svého druhu na světě. V současné době obsahuje asi 2.6 miliard WWW stránek. Každá stránka obsažená v této databázi je analyzována a zařazena do jedné z mnoha definovaných tématických kategorií. Pokud tento webový filtr uživatelem přistupovanou stránku nenalezne v centrální databázi, dojde na základě analýzy obsahu k jejímu klasifikování a uložení. Analýza obsahu stránky neprobíhá často přímo na stroji uživatele, ale adresa přistupované stránky je poslána na výkonný server, který klasifikaci provede a postará se i o uložení adresy do své databáze pod přiřazenou kategorii. Během procesu klasifikace prochází stránka většinou několika moduly, přičemž každý je zaměřen na určitý typ obsahu. Proventia Web Filter disponuje například modulem pro klasifikaci textu, který využívá Bayesův klasifikátor v kombinaci s dalšími algoritmy. Jiným modulem je OCR (Optical Character Recognition), který je schopen v obrázku identifikovat písmo různého typu, velikosti i barvy nezávisle na jeho natočení. Propracovanost tohoto webového filtru dokazuje přítomnost modulu pro identifikaci obličejů, který dokáže v obrázku vysoké kvality rozpoznat i jednotlivé osoby a modulu pro rozpoznání obnažených lidských těl na základě identifikace barevných tónů lidské kůže v obrázku.

Vytvořit komplexní systém detailně analyzující takové množství obsahových složek samozřejmě vyžaduje mnoho času a prostředků. Proto je většina aplikací realizující tuto funkčnost dostupných jen v komerční sféře. Nicméně jak bude ukázáno dále, i pouhou klasifikací textu lze dosáhnout vysoké úspěšnosti při rozpoznávání obsahu WWW stránky. Navíc u systémů popsaného typu lze sice zvolit, jaké kategorie jsou pro uživatele závadné a nemají být zobrazovány, ale není možné měnit úroveň jejich závadnosti. Ta je pevně určena pro všechny uživatele zařazením stránek do kategorií v databázi konkrétního filtračního systému.

[2] http://www.iss.net/products_services/webfilter/

Námi vyvíjené řešení bude samozřejmě volně dostupné a mělo by přesně takovéto chybějící nastavení poskytovat. Na jeho základě by potom bylo možné při praktickém použití ověřit, jaká úroveň nastavení je optimální pro dosažení maximální úspěšnosti filtrace a zároveň celkové použitelnosti systému v závislosti na požadavcích konkrétního uživatele.

2 Struktura systému

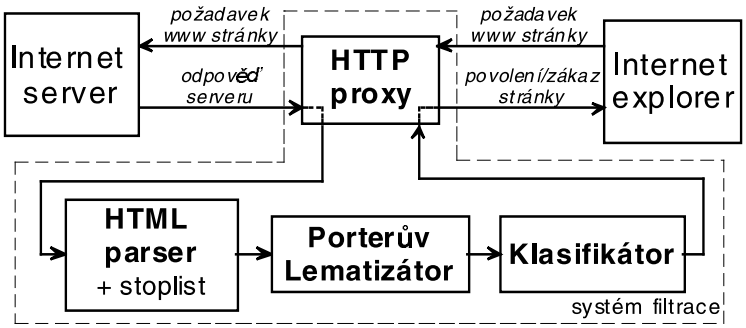
Nejčastěji používaným prostředkem pro zobrazování obsahu Internetu jsou internetové prohlížeče, které většinu informací přenášejí pomocí HTTP protokolu. Protože chceme systém pro filtraci a zachytávání závadných stránek udělat co nejvíce použitelný a zároveň jednoduchý, rozhodli jsme se ho po zvážení několika variant implementovat jako HTTP proxy server, který v sobě zahrnuje několik modulů určených ke zpracování HTML kódu a k následnému získání maximálního množství textové informace z uživatelem přistupované stránky. Takto zvolené řešení umožňuje široké použití, protože internetové prohlížeče (Internet Explorer, Netscape, Mozilla,) ve své konfiguraci nastavení jednotlivých typů proxy serverů (http, secure, ftp,) podporují. Implementace v jazyce Java navíc zaručuje přenositelnost na mnoho platforem, přičemž umístění může být buď na centrálním počítači, který bude veškerou komunikaci filtrovat, nebo je možné systém spustit přímo na lokálních strojích, u nichž požadujeme, aby byl jejich internetový provoz filtrován. Pokud by bylo potřeba k některým internetovým serverům přistupovat přímo a ne přes proxy server, je možné jejich IP adresy v konfiguračním souboru definovat. Stejným způsobem lze definovat i servery, na které má být přístup permanentně odepřen.

2.1 HTTP proxy server

Nejprve bylo uvažováno o využití již existujícího proxy serveru, konkrétně o produktu SQUID (viz [13]), který poskytuje velké množství nastavitelných funkcí včetně nadefinování regulárních výrazů pro filtraci zobrazovaného obsahu. Je však vyvíjen pouze pro platformu Unix a neposkytuje možnost dostat se ke kompletnímu kódu HTML stránky, což pro naše účely není příliš vhodné.

Proto jsme pro snadnější manipulaci a začleňování nových funkčních modulů implementovali vlastní HTTP proxy server v jazyce Java. Důvodem je kromě jiného množství volně dostupných knihoven, velká podpora práce se sítí a možnost objektového přístupu při vytváření aplikací v tomto programovacím jazyce. Navržená architektura systému je znázorněna na Obrázku 1.

Po spuštění HTTP proxy serveru je monitorován provoz na zvoleném portu (obvykle port 3128). Pokud se internetový prohlížeč na tomto portu připojí k proxy serveru s požadavkem na libovolnou stránku (viz Obrázek 1 *požadavek WWW stránky*), je nejprve zjištěno IP adresa serveru, na kterém se požadovaná stránka nachází a následně je porovnána se seznamem zakázaných serverů, které je možné definovat v konfiguračním souboru. K serveru nalezenému v tomto



Obr. 1. Základní bloková architektura systému.

seznamu je přístup ihned odmítnut. Následně je procházen seznam adres povolených serverů, ke kterým je okamžitě povolen přístup bez jakéhokoli filtrování. Pokud není adresa v těchto seznamech nalezena, je serveru v síti Internet předán požadavek prohlížeče. Jeho odezva je zachycena proxy serverem (viz Obrázek 1 *odpověď serveru*) a je poslána ke zpracování do HTML parseru.

2.2 HTML parser

Vytvářený systém je v úvodní fázi vývoje a je prozatím zaměřen především na zpracování WWW stránek s obsahem typu html/text a text/plain. Po předání odezvy serveru HTML parseru se tedy nejprve z úvodní HTTP hlavičky otestuje atribut content-type, zda je obsah typu html/text nebo text/plain. Pokud ne, je poslán prohlížeči (viz Obrázek 1 *povolení stránky*) a ihned zobrazen.V opačném případě je obsah dále zpracován. Aby bylo možné dosáhnout při klasifikaci stránky co největší přesnosti, je nutné získat maximum informací, které přistupovaná stránka obsahuje. Jednoduchým odstraněním HTML tagů a ponecháním jen textu mezi nimi bychom se ve většině případů připravili o mnoho cenných údajů, které mohu být pro klasifikaci velmi důležité. Jazyk Java umožňuje pomocí třídy ParserCallback nejen elegantně rozpoznávat jednotlivé HTML tagy a případný text který uvozují, ale také zajišťuje přístup k jejich jednotlivým atributům, pokud jsou definovány. Veškerý získaný text je ihned převeden na malá písmena a rozdělen na jednotlivá slova pomocí standardní třídy StringTokenizer. Pokud slovo není obsaženo ve stoplistu, který obsahuje 390 nejčastějších anglických slov převzatých z WAIS, je zahrnuto do výsledného textového obsahu zpracovávané stránky. Při dělení slov uvedenou třídou je velmi důležité definovat v jejím konstruktoru oddělovače jednotlivých slov. Po prozkoumání obsahu všech testovacích kolekcí, které jsme měli k dispozici a několika iteracích, při kterých jsme srovnávali úspěšnost klasifikace, byly kromě bílých znaků použity ještě následující oddělovače slov:

. , : ; ? ! ” _ - = / \’ ( ) { } ˜ |* @ # $ % ˆ & + [ ] <> ©

Protože není pravidlem, že by se na WWW stránce vyskytovaly věty ukončené tečkou tak jako v klasických článcích, je potřeba stanovit, které úseky budou brány jako logicky související. Optimálním řešením se po prozkoumání vybraného vzorku zdrojových kódů závadných i nezávadných stránek jevilo ponechat jako oddělovače konce řádků. Struktura výsledného textu zbaveného HTML tagů potom poměrně věrně korespondovala se strukturou textu HTML stránky zobrazené v internetovém prohlížeči. Jak bude uvedeno dále, tuto vlastnost zatím neuvažujeme a pracujeme s textem jednotlivých HTML stránek (dokumentů) jako s jedním dlouhým řetězcem (větou), ale v budoucnu chceme tohoto členění stránky na jednotlivé související úseky využít při trénování klasifikátoru.

2.3 Porterův lematizátor

Činnost systému jsme zatím testovali jen na anglických kolekcích dokumentů, proto jsme použili ověřenou Java implementaci Porterova lematizačního algoritmu (viz [14]) pro anglický jazyk. Jeho detailní popis je možné nalézt v [1]. Naším cílem je samozřejmě v budoucnu rozšířit filtraci i na další, především evropské jazyky. Pro mnoho z nich je již na adrese [4] Porterův algoritmus k dispozici. Ke každému uvedenému jazyku je zde také možné nalézt seznam slov s lematizovanými ekvivalenty a seznam stop slov.

2.4 Klasifikátor

Ke klasifikaci dokumentů jsme se rozhodli použít frází získaných pomocí Suffix Tree algoritmu (viz [8]) dále označovaných jako ST-fráze. Jedná se o slovo nebo o posloupnost po sobě následujících slov, které se vyskytují v množině trénovacích dokumentů. Intuitivně by tedy měly ST-fráze vystihovat okruhy představující jednotlivé klasifikační třídy lépe než jednotlivá slova a to i v případě, že si budou tématicky velmi podobné. Vycházíme přitom z předpokladu, že tématicky blízké dokumenty budou často obsahovat stejná nebo podobná slova a delší slovní spojení lépe vystihnou zaměření klasifikovaného dokumentu. Jak bude uvedeno dále, ke klasifikaci pomocí ST-frází postačuje mít k dispozici jen data závadné třídy, není nutné vytvářet kolekci nezávadných dokumentů. Tento fakt přináší samozřejmě snížení celkové složitosti, protože postačuje vytvořit ST-fráze jen pro třídu zastupující úzký tématický okruh, který chceme rozpoznávat pro účely filtrování, což je oproti jiným klasifikačním metodám výrazné zjednodušení.

3 ST filtrace

3.1 Získání ohodnocených Suffix Tree frází

K získání ST-frází jsme implementovali algoritmus postupně vytvářející modifikovanou Suffix Tree strukturu z trénovací kolekce dokumentů. Vycházeli jsme přitom z algoritmu STC (Suffix Tree Clustering) popsaném v [8] a [9], který má lineární složitost. Při hledání ST-frází je text jednotlivých internetových stránek

zpracovaný HTML parserem považován za souvislou větu a pracujeme s ním jako s jedním dlouhým řetězcem. V budoucnu plánujeme text jednotlivých dokumentů ještě dále členit na související úseky (viz závěr podkapitoly 2.2).

Postup vytváření Suffix Tree struktury je podobný jako v [8]. Maximální délka hledaných ST-frází, kterou je nutné zvolit, určuje hloubku výsledného stromu a představuje v podstatě také velikost pomyslného okénka postupně klouzajícího zpracovávaným textem. Definujeme-li tedy maximální délku ST-fráze jako m, bude se celý postup pro získání ohodnocených ST-frází skládat z následujících kroků:

1. Vytvoříme kořenový uzel stromu a prázdný seznam pro uložení množiny slov z dokumentů trénovacího korpusu;
2. Ze vstupního dokumentu načteme prvních m slov $w_1 w_m$ (případně méně, pokud jich není dostatek);
3. Zjistíme, zda se již načtená slova vyskytují v seznamu slov a pokud ne, přidáme je;
4. Jednotlivá vstupní slova w_i, kde $i = 1m$, zařadíme do Suffix Tree struktury (formou odkazu do seznamu slov) tak, aby bylo slovo w_i umístěno v hloubce i a cesta od kořene stromu do tohoto uzlu vedla přes uzly odpovídající slovům $w_1 w_{i-1}$. Pokud se uzel představující zpracovávané slovo na takovéto pozici již vyskytuje, inkrementujeme pro tento uzel čítač výskytů.
5. Ze vstupu vyřadíme první slovo a pokud není vstup prázdný, opakujeme uvedený postup od kroku 2;
6. Načteme další dokument a celý postup opakujeme od kroku 2.

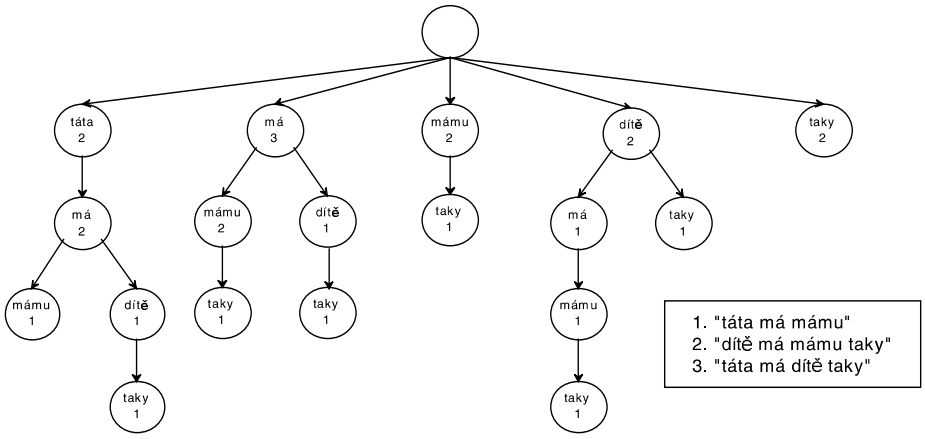
Pokud bychom tedy chtěli získat ST-fráze ze tří souborů (trénovacího korpusu), přičemž každý by obsahoval jeden z následujících řetězců

"táta má mámu"
"dítě má mámu taky"
"táta má dítě taky"

a definujeme-li maximální hloubku vytvářeného stromu (délku ST-fráze) $m = 4$, bude mít výsledná Suffix Tree struktura tvar uvedený na Obrázku 2.

Každý uzel představuje jedno slovo z množiny trénovacích dokumentů. Neobsahuje však přímo řetězec reprezentující slovo, ale odkaz do seznamu slov, který je vytvářen současně se Suffix Tree strukturou, čímž je možné dosáhnout výrazného snížení celkových paměťových nároků. Jednotlivá slova se totiž v úrovních stromu větších než 1 opakují. Uzly dále obsahují informaci, v kolika různých dokumentech se jimi reprezentované slovo objevilo a v jaké hloubce stromu se nalézají.

V našem případě není potřeba v uzlu uchovávat seznam dokumentů, ve kterých se slovo vyskytlo, protože všechny naše dokumenty, ve kterých hledáme ST-fráze, patří vždy do jedné třídy (jednoho tématického okruhu) a navíc my již dále nebudeme vytvářet shluky jako tomu bylo v případě [8] a [9].



Obr. 2. Příklad Suffix Tree struktury.

ST-fráze	četnost výskytu	ohodnocení [%]	hloubka stromu (délka ST-fráze)
má	3	100	*1*
táta	2	67	
mámu	2	67	
dítě	2	67	
taky	2	67	
má mámu	2	67	*2*
táta má	2	67	
má dítě	1	33	
mámu taky	1	33	
dítě má	1	33	
dítě taky	1	33	
má mámu taky	1	33	*3*
má dítě taky	1	33	
táta má mámu	1	33	
táta má dítě	1	33	
dítě má mámu	1	33	
táta má dítě taky	1	33	*4*
dítě má mámu taky	1	33	

Tab. 1. ST-fráze získané ze struktury na Obrázku 2.

Jednotlivé fráze jsou po vytvoření Suffix Tree struktury získány procházením uzlů stromu od kořene k listům. Celkový počet výskytů dané fráze se vydělí celkovým počtem všech zpracovaných dokumentů trénovací kolekce a vynásobí hodnotou 100, čímž získáme její výsledné ohodnocení v procentech (viz Tabulka 1). Nalezené ST-fráze charakterizují oblast, kterou svým obsahem pokrývají dokumenty obsažené v korpusu sloužícím jako vstup pro Suffix Tree algoritmus. Pro klasifikaci do více tříd je samozřejmě potřeba získat množiny ST-frází charakteristických pro jednotlivé třídy. K tomu je nutné vytvořit korpusy pokrývající odpovídající tématické oblasti.

3.2 Klasifikace pomocí Suffix Tree frází

Počet nalezených a ohodnocených ST-frází různé délky získaných z celé kolekce trénovacích WWW stránek bude většinou velmi velký, protože Suffix Tree struktura se s přidáváním nových slov poměrně rychle rozrůstá. I když trénujeme na vhodných dokumentech s použitím stoplistu a lematizace, nemusí samozřejmě všechny nalezené fráze dostatečně vystihovat téma, které bychom chtěli rozpoznávat a filtrovat. Proto ke klasifikaci vybereme jen fráze s nejvyšším ohodnocením. Jejich počet se samozřejmě bude lišit v závislosti na tématickém rozsahu trénovací kolekce dokumentů a s tím spojené četnosti výskytů jednotlivých slov. ST-fráze délky 1 (tedy jednotlivá slova) budou mít samozřejmě většinou největší ohodnocení, proto je vhodné vybírat určitý počet frází jednotlivých délek. Delší fráze mají totiž větší tématickou vypovídací hodnotu a je vhodné jim při klasifikaci přiřadit větší váhu.

Samotná klasifikace probíhá podobně jako u metody Itemsets (viz [5] a [6]), ze které uvedený postup vychází. V testovaném dokumentu hledáme vybrané ST-fráze a v případě jejich výskytu přičítáme jejich ohodnocení (vynásobené jejich vahou) k celkovému ohodnocení klasifikovaného dokumentu.

Oproti metodě Itemsets zde však netestujeme výskyt K-itemsetů jednotlivých klasifikačních tříd a v závěru nezařazujeme dokument vždy do třídy (případně do tříd) s nejvyšším ohodnocením. Tento způsob totiž předpokládá trénování na množině dokumentů vystihujících všechny třídy do kterých chceme klasifikovat (tedy závadné i nezávadné).

V našem případě máme pouze závadné třídy charakterizované vybranými ST-frázemi. Proto po otestování výskytu jednotlivých frází dostáváme jedinou hodnotu vyjadřující ohodnocení (míru závadnosti) testovaného dokumentu (případně dostaneme hodnoty ohodnocení pro jednotlivé závadné třídy). Na základě vhodně nastavených prahových hodnot lze potom rozhodnout, zda je daný dokument nezávadný nebo závadný a do kterých závadných oblastí případně patří.

Výhody tohoto přístupu jsou zřejmé. Není potřeba vytvářet trénovací kolekci nezávadných dat, pouze závadných. Kompletní kolekci plně vystihující množinu nezávadných textů není samozřejmě možné vytvořit vzhledem k příliš širokému spektru témat. Dále lze již v průběhu ohodnocování ukončit klasifikaci dokumentu, pokud bude pro danou závadnou třídu překročena nastavená prahová hodnota. Způsob, jakým lze tuto prahovou hodnotu určit, společně se srovnáním úspěšnosti klasifikace ST-frází s jinými metodami, uvedeme v následujícím odstavci.

3.3 Testování a experimenty

Při testování jsme se prozatím zaměřili na rozpoznání pornografického materiálu především z důvodu jeho snadné dostupnosti. Pro natrénování klasifikátoru jsme použili 1600 WWW stránek s touto tématikou v anglickém jazyce, které byly staženy z různých internetových serverů, upraveny HTML parserem a následně lematizovány.

	celkem dokumentů				
	závadných	nezávadných	celkem různých slov	minimum slov v dokumentu	maximum slov v dokumentu
reuters & porn sites	400	400	22501	76	10633
7 sectors[3]	0	4581	36653	18	7615
news20[3]	0	16330	104785	45	22568
webkb-data[3]	0	8273	63675	31	30853
Internet porn sites[4]	18323	0	75042	186	14226

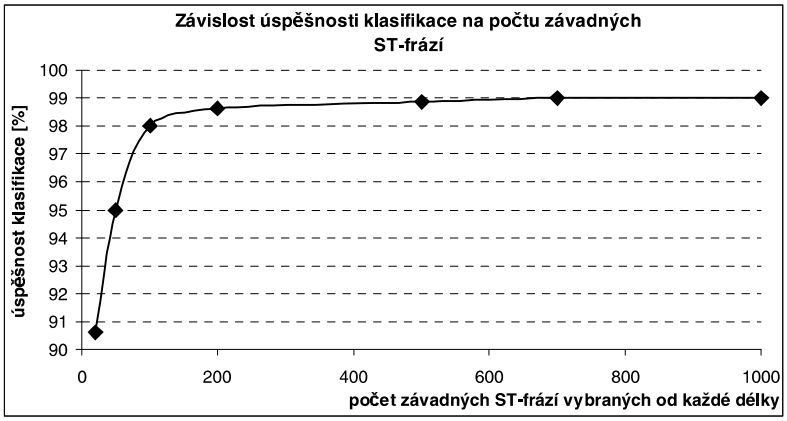
[3] Dostupné z www-2.cs.cmu.edu/afs/cs.cmu.edu/project/theo-11/www/wwkb/index.html

[4] Kolekci jsme vytvořili stažením odkazů z několika relevantních webových rozcestníků

Tab. 2. Charakteristiky testovacích kolekcí.

	Itemsets	Naive Bayes	NBCI	TFIDF	C4.5	ST-fráze	ST-fráze s průnikem	ST-fráze + průnik + váhy (3/4 - 8 - 16)
reuters & porn sites	99,75	97,88	96,75	99,25	99,38	99,05	**100**	99,75
7 sectors	99,39	94,56	96,86	88,73	99,19	98,64	**99,5**	99,41
news20	99,82	**99,96**	98,71	90,92	97,89	99,06	99,31	99,45
webkb-data	**99,14**	97,04	96,97	92,28	**99,14**	97,6	98,97	99,1
Internet porn sites	97,23	96,88	96,92	98,19	95,8	98,3	97,8	**98,5**

Tab. 3. Výsledky různých metod klasifikace na testovacích kolekcích [%].



Obr. 3. Závislost úspěšnosti klasifikace na počtu závadných ST-frází.

Celková velikost tohoto trénovacího korpusu byla po provedených úpravách 20 MB, počet různých slov byl 25387 a maximální počet slov v dokumentu 16850. Pomocí Suffix Tree algoritmu jsme z něj získali velké množství ohodnocených ST-frází délky 1, 2 a 3. Pro klasifikaci jsme vybrali vždy 1000 frází s nejvyšším

ohodnocením od každé délky. Tento počet jsme zvolili na základě charakteristiky, kterou zobrazuje Obrázek 3 a která byla vytvořena z první testovací kolekce v Tabulce 2.

Důležitou otázkou bylo, jak vhodně stanovit práh závadnosti dokumentu. Jako nejlepší řešení se ukázalo stanovit tento práh empiricky za pomoci vybraného vzorku závadných i nezávadných dat. K tomu nám posloužil korpus vytvořený ze 400 článků kolekce Reuters Volume 1 a 400 zpracovaných závadných WWW stránek (viz první kolekce v Tabulce 2). Následně byl každý dokument tohoto korpusu ohodnocen vyhledáním závadných ST-frází. Pokud označíme maximální dosažené ohodnocení nezávadného dokumentu jako MAX_{nez} a minimální dosažené ohodnocení závadného dokumentu jako MIN_{zav}, můžeme zvolit vhodnou prahovou hodnotu podle vzorce

$$PRAH_{zav} = \frac{MAX_{nez} + MIN_{zav}}{2} \tag{1}$$

Jak je ze vzorce (1) patrné, stoprocentní úspěšnosti klasifikace lze dosáhnout pouze v případě, pokud $MAX_{nez}<MIN_{zav}$. Tento způsob nastavení hodnoty prahu se poměrně osvědčil, což jsme ověřili na testovacích kolekcích, jejichž charakteristika je uvedena v Tabulce 2.

Ačkoli se jedná o nejjednodušší variantu klasifikace pomocí ST-frází, lze tímto postupem dosáhnout vysoké úspěšnosti, což je patrné ze sloupce ST-fráze v Tabulce 3, která obsahuje výsledky různých klasifikačních metod na testovacích kolekcích. Výše uvedený postup předpokládá relevantní korpus dostatečně vystihující tématické zaměření v budoucnu klasifikovaných dokumentů, který poslouží k nastavení pomyslné hranice (prahu) mezi závadným a nezávadným dokumentem. Hlavním cílem je zde určit, jaká míra závadnosti je pro uživatele ještě přijatelná. Tu lze stanovit i bez nezávadných dat pouhým odhadem a průběžnou korekcí při praktickém používání.

Úspěšnost klasifikace je možné navíc zvýšit následujícím postupem. Z množiny nezávadných dokumentů vytvoříme ST-fráze stejné maximální délky jako závadné ST-fráze, které již máme k dispozici a vybereme z nich určitý počet s nejvyšším ohodnocením (opět pro každou délku). Provedeme průnik množin závadných a nezávadných ST-frází, přičemž společné fráze z množiny závadných odstraníme.

Jedná se tedy v podstatě o začištění, kdy jsou dodatečně upřesněné nezávadné fráze z množiny závadných frází odstraněny, aby nezpůsobovaly chyby při klasifikaci. Jak je patrné z hodnot sloupce ST-fráze s průnikem v Tabulce 3, vedl tento postup ve většině případů ke zvýšení úspěšnosti klasifikace, především samozřejmě u kolekcí s nezávadnými dokumenty.

Jak již bylo uvedeno, nelze vytvořit kompletní korpus vystihující množinu všech nezávadných textů, avšak dle našich dosavadních poznatků použití libovolné kolekce nezávadných dokumentů vede ke zlepšení výsledků klasifikace. Samozřejmě pokud použijeme k začištění závadných ST-frází nezávadnou kolekci, která vystihuje tématickou oblast, ve které se bude pohybovat budoucí uživatel, dosáhneme při klasifikaci nejlepších výsledků.

Vhodné je také upřednostnit delší ST-fráze, protože lépe vystihují tématické zaměření klasifikovaného dokumentu. To je možné realizovat vhodným přiřazením vah. Vyzkoušeli jsme mnoho variant, přičemž nejlépe se osvědčilo nastavení

3/4	pro ST-fráze délky 1
8	pro ST-fráze délky 2
16	pro ST-fráze délky 3

zastoupené sloupcem ST-fráze + průnik + váhy 3/4-8-16 v Tabulce 3. Úspěšnost klasifikace se oproti nejjednodušší variantě zvýšila u všech testovacích kolekcí, oproti verzi s průnikem u třech kolekcí. Na rozdíl od metody Itemsets tedy využití delších ST-frází přináší, zejména při vhodném nastavení vah, zvýšení celkové úspěšnosti klasifikace.

Ačkoli se nastavení váhy pro ST-fráze délky 3 může zdát poněkud vysoké, vzhledem k jejich nízkému ohodnocení a empiricky nastavenému prahu závadnosti by se v dokumentu musely průměrně vyskytovat 3 až 4 závadné fráze této délky, aby byl považován za závadný.

Při klasifikaci jsme vždy brali v úvahu vícenásobný počet výskytů závadné ST-fráze v dokumentu. Tento přístup se osvědčil více než pouhé uvažování výskytu, protože v tomto případě nejsou tolik patrné rozdíly v ohodnocení závadných a nezávadných dokumentů. Důsledkem je samozřejmě nižší úspěšnost klasifikace, která byla při testování vždy horší o 10 až 20% oproti výsledkům uvedeným v Tabulce 3.

Časová i paměťová složitost metody Suffix Tree při získávání častých frází je lineární (viz [8]), přičemž výsledný čas a použitá paměť je dána počtem slov v trénovací kolekci dokumentů a nastavenou hloubkou vytvářeného stromu (délkou ST-frází). Konkrétní hodnoty se samozřejmě budou lišit v závislosti na efektivnosti konkrétní implementace algoritmu a použitém programovacím jazyce.

4 Plánovaná rozšíření systému

V současné době již pracujeme na rozšíření, které umožní na základě prozkoumání odkazů obsažených v již zachycených závadných WWW stránkách vyhledávat a klasifikovat další závadné stránky. Předpokládáme totiž, že mnoho odkazů obsažených v závadné stránce bude odkazovat na jinou stránku s podobným obsahem. Postupně by tak docházelo k vytváření databáze stránek s různým závadným zaměřením. Z vytvořené databáze plánujeme získat adresy serverů, na kterých se nachází největší množství závadného materiálu. Jejich adresy a zaměření potom plánujeme zveřejnit na internetu. Mohou posloužit například správcům sítě pro zakázání přístupu nebo institucím státní správy pro jejich zrušení, pokud by byl jejich obsah v rozporu se zákonem. Prioritou je rozšířit náš systém kromě anglického jazyka o podporu dalších, především evropských jazyků. To znamená v první řadě získat relevantní závadné korpusy, které ovšem nejsou snadno dostupné. Proto bychom v tomto směru samozřejmě uvítali jakoukoli pomoc.

Reference

1. Porter M.F.: An Algorithm for Suffix Stripping. Program, **14**, (3), 1980, 130–137.
2. Koch, Traugott and Ardö, Anders: Automatic Classification of Full-Text HTML-Documents from one Specific Subject Area. (EU Project DESIRE II D3.6a WP2), 2000. `http://www.lub.lu.se/desire/DESIRE36a-WP2.html`.
3. Jiawei Han, Micheline Kamber: Data Mining: Concepts and Techniques. Morgan Kaufmann Publishers, August 2000, 550 pages.
4. `http://snowball.tartarus.org/`.
5. Hynek J., Jezek K.: Document Classification Using Itemsets. Proc. 34th Int. Conf. MOSIS 2000, ISM 2000, 97–102.
6. Hynek J., Jezek K., Rohlik O.: Short Document Categorization – Itemsets Method. PKDD 4-th European Conference on Principles and Practice of Knowledge Discovery in Databases, Workshop Machine Learning and Textual Information Access, Lyon, France, Sept. 2000, 14–19.
7. Zendulka J., Kotásek P.: AprioriItemset – A New Algorithm for Discovering Frequent Itemsets. In: 33rd Spring International Conference Modelling and Simulation of Systems MOSIS'99 Proceedings ISM'99 Information Systems Modelling, MARQ Ostrava, Rožnov pod Radhoštěm, 1999, 49–56.
8. Zamir O., Etzioni O.: Web Document Clustering: A Feasibility Demonstration. In Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '98, Melbourne, Australia, Aug. 24-28), 1998, W.B. Croft, A. Moffat, C.J. van Rijsbergen, R. Wilkinson, J. Zobel, Chairs. ACM Press, New York, NY, 46–54.
9. Grolmus P., Hynek J., Ježek K.: User Profile Identification Based on Text Mining. Proc. of 6th Int. Conf. ISIM'03, MARQ Ostrava, 109–118.
10. Kučera M., Ježek K., Hynek J.: Text Categorization Using NBCI Method (in Czech), ZNALOSTI 2003, Proc., VŠB Technická univerzita Ostrava, 33–42.
11. `http://www.cobion.com/`.
12. `http://www.internetfilterreview.com/`.
13. `http://www.squid-cache.org`.
14. `http://tartarus.org/ martin/PorterStemmer/index.html`.

Klasifikace multilinguálních korpusů s využitím tezauru EuroWordNet*

Michal Toman and Karel Ježek

Katedra informatiky a výpočetní techniky, Západočeská univerzita,
Univerzitní 22, 30614 Plzeň
{mtoman, jezek_ka}@kiv.zcu.cz
http://www.kiv.zcu.cz/

Abstrakt Klasifikace textů je jednou z aktuálních oblastí výzkumu. Tento článek popisuje výsledky experimentů klasifikace textových korpusů s využitím tezauru EuroWordNet (EWN), porovnává různé přístupy a vyvozuje závěry pro možná vylepšení klasifikační úlohy. Součástí práce je popis algoritmů a metod použitých v navrženém klasifikačním systému. Cílem experimentů bylo ověřit vliv multilinguality textových korpusů na kvalitu klasifikace a navrhnout vhodné využití vícejazykového tezauru za účelem zlepšení, či zobecnění možností klasifikace multilinguálních textových korpusů. V článku je diskutována problematika lemmatizace a indexace s respektováním vícejazyčného prostředí. Popsáno je též následné navázání lemmat na tezaurus EWN. Testy byly vykonány na korpusech Reuters a ČTK a poukazují na fakt, že při použití vhodných klasifikačních algoritmů lze provádět klasifikaci dokumentů zcela nezávislou na jazyku. Dále je ve článku prokázáno, že výsledky jsou velmi závislé na volbě klasifikačního algoritmu. Jako perspektivní klasifikační metody modifikovatelné pro vícejazykové prostředí se ukazují např. algoritmy NBCI, Itemsets, TF/IDF.

Klíčová slova: klasifikace, kolekce dokumentů, tezaurus, EuroWordNet, mutilinguální korpus, lemmatizace, přirozený jazyk, analýza dokumentů, Bayesův teorém

1 Multilinguální korpusy

V současné době se častěji objevuje nutnost uchovat a počítačově zpracovávat dokumenty, které jsou uloženy v jedné knihovně, ale jsou napsány v různých jazycích. Dříve se tento aspekt spíše zanedbával. Mnohé systémy pro zpracování textů předpokládají jednojazyčné prostředí a svou funkci tomu mají uzpůsobenou. Možnost uložení vícejazyčných dokumentů bud vůbec neřeší, nebo pouze okrajově. Považujeme-li Internet, konkrétně webové stránky, za velký elektronický archiv, je zřejmé, že obsažené informace jsou obecně v různých jazycích.

* Příspěvek vznikl za částečné podpory výzkumného záměru MSM 235200005.

S postupující integrací jednotlivých států a rozšiřováním Evropské unie se dostává respektování vícejazyčnosti do popředí zájmu. Typickým příkladem aplikace multinguálního systému může být prohledávání webových stránek, vědeckých článků, zákonů, předpisů a podobně. Lze také rozšířit stávající vyhledávací systémy tak, aby lépe umožňovaly vyhledávání ve vícejazykovém prostředí. Předpokládáme, že vytvářený systém by našel uplatnění v rozsáhlejších digitálních knihovnách, kde se vyskytují dokumenty v různých jazycích, případně ve státní správě, která bude stále častěji přicházet do styku s cizojazyčnými dokumenty. V neposlední řadě může být zakomponovaný jako součást námi řešeného systému pro podporu vědeckých pracovníků.

Stávající řešení dokumentografických systémů, na výjimky reprezentované pokusnými systémy (MILK, AutIndex [5], ...), nemají problematiku multilinguálního prostředí jako svůj primární cíl. Ve většině systémů se provede rozpoznáním jazyka a následně oddělené zpracování dokumentů, což nemusí dostačovat.

Za předpokladu, že uživatel zná několik jazyků, je vhodné umožnit jedním dotazem vyhledat všechny relevantní dokumenty. S použitím navrhovaného modelu se může například provádět kategorizace dokumentů i v případě, že jsou stávající třídy definovány pouze pro jediný jazyk. Problém nastává v případě indexování vícejazyčného korpusu, kdy ekvivalentní překlady téhož slova mají v rozdílných jazycích různé indexy (např. slovo strom je indexováno jinak než anglický překlad tree). Výhodnější je indexovat ekvivalentní překlady jedním indexem.

Rozhodli jsme se vytvořit model vícejazykového systému, který by poskytoval stejně kvalitní výstupy jako stávající systémy, ale s dokonalým respektováním vícejazyčného prostředí. Dlouhodobějším cílem je vytvoření systému, který bude poskytovat jedním dotazem výsledek, jenž není závislý na jazyce dotazu, ani jazyce vyhledaných dokumentů.

Jako pokusnou úlohu na otestování našeho přístupu jsme zvolili klasifikaci dokumentů. V tuto chvíli je testovací kolekce složena z českých dokumentů (tiskové zprávy ČTK) a anglických dokumentů (tiskové zprávy Reuters). Do budoucna se počítá s rozšířením i na další evropské jazyky. Kolekce vznikla výběrem shodných tříd z obou tiskových agentur dovolujícím vyzkoušet klasifikaci dokumentů. Cílem bylo ověřit vhodnost použití tezauru EuroWordNet (EWN) jako jádra zpracování vícejazyčných korpusů.

2 Tezaurus EWN

Jádro navrhovaného systému je aplikace tezauru EuroWordNet (EWN [6]) jako referenčního slovníku pro provázání slov jednotlivých jazyků. EWN je tezaurus, který sobě odpovídajícím skupinám synonym (synsetům) přiřazuje shodné indexy. To následně umožňuje jednotnou indexaci pro různé jazyky.

EWN je vícejazyčná databáze pro některé evropské jazyky (angličtina, čeština, dánština, italština, španělština, němčina, francouzština, estonština). Je strukturovaná podobně jako původní Wordnet vytvořený Princetonskou univerzitou. EWN obsahuje množiny synonym a vzájemné vztahy mezi nimi. Jednotlivé

množiny synonym (synsety) jsou navíc spojené pomocí tzv. ILI (inter-lingual-index) tak, že shodný synset jednoho jazyka má tentýž index v jiném jazyce. Díky tomu bylo možné považovat tento index za ekvivalentní indexu, jenž je použitý při klasické indexaci jednojazyčného korpusu.

Určitou nevýhodou tezauru EWN je jeho přílišná jemnost. Pro téměř totožný výraz je často definován rozdílný index. V takovém případě může docházet k nekorektní indexaci a výsledky klasifikační metody mohou být velice špatné. Jedná se o problém, který je řešitelný např. shlukováním podobných synsetů. Řešení jsou diskutována dále. Další nevýhoda EWN souvisí s nestejnou rozpracovaností jednotlivých wordnetů, takže některé synsety nemají v určitých jazycích odpovídající ekvivalent.

3 Moduly systému

Experiment využití EWN pro klasifikaci vícejazyčných dokumentů byl rozdělen do několika fází. Nejdříve bylo nutné vytvořit kvalitní lemmatizační slovník. Stávající algoritmická a slovníková lemmatizace se musela upravit, aby bylo možné lemmata propojit s tezaurem EWN. Dosud námi používané metody převáděly slova na lemmata, která nebyla totožná se základními tvary slov. Následně nebylo možné vytvořit korektní vazbu na EWN. Pro splnění této podmínky bylo nutné vytvořit nové lemmatizační slovníky.

Lemmatizovaná slova jsou následně namapována na synsety EWN a pomocí jednoduché transformace je zjištěn index, který je zpracován při klasifikaci [obr. 1]. Funkce jednotlivých komponent jsou rozebrány dále.

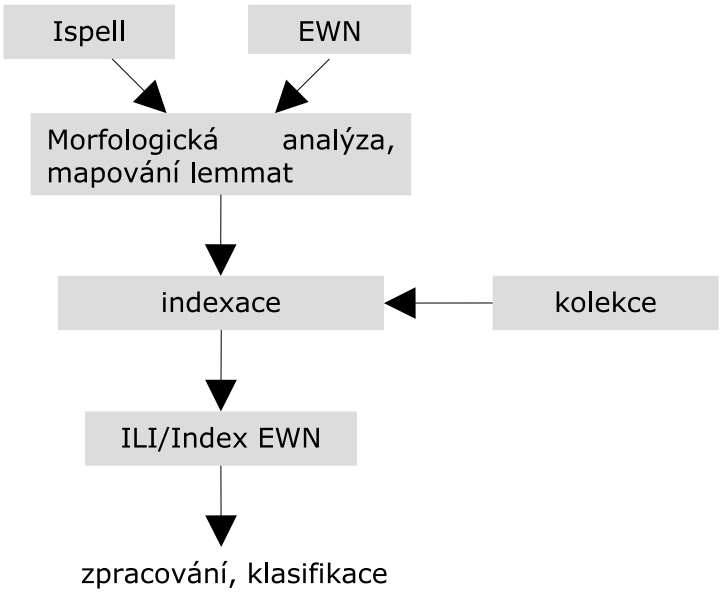
3.1 Lemmatizační modul

K vytvoření lemmatizačního slovníku jsme zvolili extrakci tvarů slov z programu Ispell [7]. Lemmatizačním slovníkem rozumíme alfabeticky uspořádanou množinu slov a jim odpovídající lemmata. Ispell je interaktivní program pro kontrolu pravopisu, který podporuje většinu evropských jazyků. Primárním účelem programu je procházet texty, kontrolovat pravopis a případně navrhovat opravy nerozpoznaných slov.

Základní myšlenkou bylo vzít kmeny slov, které jsou uloženy ve slovníku Ispellu a z těch pomocí Ispellu odvodit všechny existující tvary. Kmen slova byl považován za základní tvar, měl by se tedy vyskytovat v tezauru. Tento přístup fungoval dokonale u anglického jazyka, ovšem selhával u češtiny, která disponuje daleko větší flexí.

Základní problém spočíval v tom, že kmen slova se nemusí shodovat se základním tvarem. Příkladem může být slovo lano, jehož kořen uvedený v Ispellu je lan a přípony mohou být například -o -em -ech apod. Tedy základní tvar slova (lano) se neshoduje s kořenem (lan).

Při řešení výše uvedeného problému jsme vycházeli z předpokladu, že základní tvar slova je v množině všech možných tvarů slova, které jsme získali z Ispellu. Proto jsme vzali slovník Ispellem vygenerovaných slov, vytvořili jsme



Obr. 1. Návaznost komponent klasifikačního systému.

podmnožiny tvarů slov odpovídající jednomu kmeni, resp. základnímu tvaru slova, a pro každou množinu jsme hledali odpovídající lemma v EWN.

Algoritmus lze popsat:

- Pro každou množinu kmene slova proveď:
  - Pro každý tvar z množiny proveď:
    - Hledej odpovídající tvar v EWN
    - Pokud se tvar vyskytuje, tvar je základní pro všechna slova z aktuální množiny
    - Pokud se zde tvar nevyskytuje, pokračuj dalším tvarem

V případě, že se v množině tvarů nenalezne ani jeden tvar slova, který lze navázat na EWN, tak se prohlásí kmen slova za základní tvar. Tato možnost nenastává příliš často.

Dalšího vylepšení bylo dosaženo využitím morfologického analyzátoru. Jako ukázka výsledku morfologického analyzátoru mohou sloužit slova být a je, která se typicky indexují různými indexy, protože nemají shodný kmen slova. Po aplikaci analyzátoru jsou slova převedena do korektního základního tvaru, tedy být. Podobná vlastnost je důležitá také při stupňování přídavných jmen.

Nevýhodou modelu je velikost takto vytvořeného slovníku. V případě uchování slov v seznamu dvojic slovo, základní tvar dosáhne velikost téměř 100 MB pro češtinu. Tu lze však považovat za určitý extrém, jelikož podobnou flexi má jen málo jazyků. Pro angličtinu je velikost slovníku pouze 3 MB.

Do lemmatizátoru vstupují slova a výstupem jsou indexy obsažené v EWN (ILR indexy). Každý index se skládá z označení (např. eng20 znamenající slovo

zařazené anglickým týmem, EWN verze 2.0), vlastního unikátního čísla (např. 06900919) a jednopísmenné zkratky označující slovní druh (v – sloveso, n – podstatné jméno, a – přídavné jméno, apod.).
Příklad výstupu lemmatizátoru a indexace do EWN:

Původní tvar:
„Vlna chladu si vyžádala 100 mrtvých.“

Výstup po lemmatizaci a indexaci:
„eng20–06900919–n eng20–04448750–n eng20–02526983–v eng20–13048967–n eng20–00100393–a“

Jednodušší je situace u anglického jazyka, kde lze použít pro lemmatizaci i algoritmickou metodu – např. Porterův algoritmus. Také vytvoření slovníků s použitím Ispellu poskytuje korektní výsledky i s „naivním“ přístupem, jenž se u jazyků se složitějším tvaroslovím nedá uplatnit.

Velkou výhodou takto vytvářených lemmatizačních slovníků je využití již ověřených částí (Ispell, EWN), které jsou navíc dostupné již téměř ve všech evropských jazycích. Obecně lze předpokládat kvalitu takto vytvářených slovníků mezi kvalitou anglického (velmi jednoduché tvarosloví) a českého (složité tvarosloví). Přesto považujeme za vhodné provést pro každý nově přidávaný jazyk určité úpravy algoritmu lemmatizace takové, aby respektovaly specifika flexe daného jazyka. Příkladem může být němčina, kde by bylo nutné věnovat zvýšenou pozornost slovům s odlučitelnými předponami.

3.2 Mapování lemmat na synsety EWN

Získané slovníky je nutné namapovat na synsety EWN. Cílem je vyhledat k jednotlivým základním tvarům odpovídající slova v EWN a odvozeným tvarům slova přiřadit index EWN. V jazyce se ovšem vyskytují víceznačná slova, která jsou stejně zapsána, ale mají jiný význam, tudíž jsou zahrnuta v několika synsetech. K rozhodnutí správného významu je nutné využít disambiguaci. Ve slovnících není dostatečná znalost souvislosti daného slova s nějakým významem – jedná se o oddělená slova bez kontextu. Úloze disambiguace se nelze vyhnout a bude se provádět při zpracování textového korpusu. Výsledkem mapování jsou slovníky, kdy každému tvaru slova odpovídá jeden index EWN.

3.3 Morfologická analýza

Do fáze vytváření lemmatizačního slovníku byl zakomponován morfologický analyzátor vytvořený Janem Hajičem [8]. Jedná se o univerzální nástroj pro morfologickou analýzu textu. Uplatnil se především při zpracování stupňovaných přídavných jmen a nepravidelných sloves.

4 Použité klasifikační metody

Využití EWN pro zpracování multilinguálních kolekcí jsme se rozhodli ověřit na případě klasifikační úlohy, se kterou máme zkušenosti a vytvořené nástroje

pro testování přesnosti a úplnosti klasifikace. Jako testovací korpus jsme zvolili české a anglické texty, konkrétně tiskové zprávy ČTK a Reuters, které jsme používali na testování i dříve. Testy byly prováděny na korpusu 82000 českých a 25000 anglických dokumentů. Zprávy spadaly do jedné z 5 tříd, které byly v obou korpusech obsahově podobné. Jednalo se o počasí (4 %), sport (30 %), politiku (58 %), zemědělství (3 %) a zdravotnictví (5 %). Cílem bylo ověřit, zda a jak moc multilinguální prostředí ovlivní výsledky klasifikace. Volili jsme různý stupeň zapracování EWN do klasifikační úlohy. Nejdříve byl uvažován referenční stav, kdy EWN není použit vůbec, dále bylo použito EWN na jednojazyčný korpus a nakonec se testovala křížová klasifikace (trénovací korpus český, testování na anglických datech).

4.1 NBCI

Tato metoda byla vytvořena naší katedrou a jedná se o kombinaci metody Naive Bayes s Itemsety [9]. Z každého dokumentu jsou vybrány charakteristické itemsety a ty jsou dále zpracovávány metodou Naive Bayes. Výhodou je vyšší rychlost a v případě multilinguálního korpusu netrpí metoda problémem malého průniku termů v různých jazycích jako prosté použití Naive Bayes.

4.2 TFIDF

Jedná se o metodu založenou na sledování frekvence termů v dokumentech (viz [10]). Každému termu je přiřazena váha, která je tím vyšší, čím se daný term vyskytuje v dokumentu více a naopak nižší, čím se term vyskytuje častěji v jiných dokumentech.

Pro klasifikaci dokumentů je použita kosinová míra:

$$Sim(Q, D_i) = \frac{\sum_{k=1}^{n} (q_k \cdot w_{ik})}{\sqrt{\left(\sum_{k=1}^{n} {w_{ik}}^2 \cdot \sum_{k=1}^{n} {q_k}^2\right)}} \tag{1}$$

kde Q je vektor vah q_k termů nově zařazovaného dokumentu a D_i je vektor vah w_{ik} náležejících termům ve třídě C_i.

4.3 Itemsety

Pro hledání častých skupin termů je využita modifikace Apriori algoritmu, která slouží k nalezení položek, které se v dokumentech jedné třídy často vyskytují. Itemset je vybrán jako charakteristický pro určitou třídu, pokud váhový faktor překročil zvolenou mez. V klasifikační fázi je dokument zařazen do třídy za předpokladu, že C_j (množina itemsetů reprezentující třídu T_j) obsahuje itemsety s dostatečnou váhou asociace dokumentu D s třídou T_j podle vzorce (2).

$$W_{Tj}^{D} = \sum_{i=1}^{|C_j|} wf_{|\Pi_i|} \times w_{\Pi_i} \tag{2}$$

kde $(\Pi_i \in C_j) \wedge (\Pi_i \subseteq D)$ pro $j = 1, 2, \ldots, L$.

Váha asociace je zde určena součtem součinů vah w_{Π_j} s váhovými faktory $wf_{|\Pi_i|}$ pro všechny itemsety dané třídy, jež se vyskytují v právě klasifikovaném dokumentu. Více je metoda diskutována v [1] a [2].

4.4 Naive Bayes

Metoda je založena na Bayesově teorému [3]. Klasifikace je prováděna na základě následujícího vzorce:

$$v_{NB} = \arg\max P(v_j) \prod_i P(a_i|v_j) \tag{3}$$

kde a_i jsou termy nově zařazovaného dokumentu a v_j jsou slova zkoumané třídy, do které se snažíme dokument zařadit.

Jak bude vidět z výsledků testů, je přesnost metody velice nízká. Tak nízká přesnost je způsobena malým překrytím množin lemmat obou korpusů. V takovém případě není splněn předpoklad, že testovací a trénovaní data obsahují statisticky shodná data. Metoda Naive Bayes poskytuje v případě křížového testu systematicky chybný výsledek. Tento problém lze eliminovat shlukováním podobných synsetů, což je postup popsaný v kapitole 6.

5 Výsledky testů

Analýzou textů jsme dospěli k názoru, že obě části korpusu se liší ve své celkové skladbě. Zatímco ČTK je poměrně obecný zdroj informací, Reuters je zaměřený na burzovní zprávy a například v kategorii počasí není výjimkou rozsáhlé hodnocení ekonomických dopadů živelné katastrofy, což v českém korpusu není možné najít. Tato vlastnost právě způsobila špatné výsledky u klasifikační metody Naive Bayes.

Testy byly prováděny pomocí dříve vytvořených klasifikačních metod a bylo použito několik druhů předzpracování vstupních dat. Nejdříve jsme testovali referenční případ (viz tab. 1, řádka 1, 2, 3, 4; sloupec monolinguální), kdy jsou zprávy v korpusech klasifikovány odděleně (jako dva jednojazyčné korpusy).

Dalším krokem bylo aplikování lemmatizace založené na využití EWN na jednojazyčný korpus (viz tab. 1, řádka 1, 2, 3, 4; sloupec multilinguální), čímž jsme ověřili korektní fungování lemmatizátoru a indexace. Jak je vidět z tabulky, jsou výsledky srovnatelné s referenčními.

Ve třetím kroku jsme provedli klasifikaci vícejazyčné kolekce. Nejdříve byl uvažován případ, kdy v testovacích i trénovacích datech jsou oba jazyky zastoupeny v náhodném poměru (viz tab. 1, řádka 5, 6, 10). To simuluje situaci, kdy jsou například v digitální knihovně již zatříděné texty různých jazyků a probíhá

automatická klasifikace nově příchozích dokumentů. Jinak řečeno, ekvivalentní překlady mají v různých jazycích jiné indexy. Pro případ klasifikační úlohy je takový přistup možný a poskytuje smysluplné výsledky. Pro klasifikační úlohu bylo dosaženo výsledků srovnatelných s referenčními. Test jsme opakovali na korpusech lemmatizovaných klasickou metodou (slovníkovou) i pomocí metody s využitím EWN.

Posledním testem byla křížová klasifikace (viz tab. 1, řádka 7, 8, 9), kdy v archivu předpokládáme zaklasifikovaná data jednoho jazyka a snažíme se zatřídit data jiného jazyka, tj. trénovací jazyk klasifikátoru je jiný než testovací. V tomto případě velmi záleží na velikosti shody obou korpusů a některé metody jsou na tento faktor velice citlivé (Naive Bayes).

Typickou přesnost a úplnost klasifikace lze odhadovat a úplnost klasifikace lze odhadovat mezi 80 – 95 %.

			monolinguální		multilinguální	
	metoda	data	přesnost [%]	úplnost [%]	přesnost [%]	úplnost [%]
1	NBCI	cz	90.53	93.83	91.28	93.41
2	NB	cz	95.36	95.57	92.44	93.15
3	NBCI	eng	95.11	95.47	96.04	96.20
4	NB	eng	96.85	96.91	94.79	95.17
5	NBCI	cz+eng	86.75	92.06	86.05	89.52
6	NB	cz+eng	95.25	95.46	92.04	92.83
7	NBCI křížově	cz+eng	–	–	80.93	89.42
8	NB křížově	cz+eng	–	–	3.42	3.42
9	Itemsets kříž.	cz+eng	–	–	73.78	81.49
10	Itemsets	cz+eng	75.76	81.91	78.65	84.90
11	TFIDF	cz+eng	93.37	93.37	92.79	92.79

Tab. 1. Výsledky klasifikačních testů.

6 Možnosti zdokonalení

V této části zmíníme dvě možnosti, které na základě zatím předběžných experimentů slibují zlepšit výsledky klasifikace. První možností je začlenění disambiguace (zjednoznačnění) slov do procesu klasifikace, druhou je úprava seskupení synsetů v EWN.

6.1 Disambiguace

Poměrně samostatnou úlohou, kterou je nutné řešit při indexaci, je disambiguace [4]. Po provedení lemmatizace mají některá slova shodné základní tvary, ale jejich význam je odlišný. Jedním z příkladů může být slovo kohout, kde není bez znalosti kontextu zřejmé, zda se jedná o kohout – uzávěr, nebo kohout – živočich. Pro rozhodnutí významu je nutná znalost kontextu, ve kterém se slovo nachází. Pro disambiguaci existuje několik algoritmů.

Jako výchozí metodu jsme zvolili disambiguaci s učitelem, konkrétně Bayesovskou disambiguaci. Jedná se o poměrně jednoduchou metodu, která poskytuje kvalitní výsledky s přesností přes 90 %. Jistou nevýhodou metody je veliká závislost na trénovacím korpusu. Pro pokusy s disambiguátorem jsme pro trénování zvolili Brownův korpus, který je volně k dispozici na stránkách EWN. Jedná se o označkovaný korpus anglických textů, kterým jsou již přiřazeny indexy synsetů EWN. Daný korpus jsme použili pro natrénování bayesovského disambiguátoru a testovali jsme přesnost na tomtéž korpusu i na vybraných případech vět z jiných kolekcí. V bayesově disambiguátoru jsme provedli několik vylepšení – použití kontextového okénka a jeho modifikace, využití syntaktických vztahů mezi slovy a použití váhové funkce pro zvýhodnění blízkých slov víceznačnému slovu. Vylepšení zvýšila přesnost disambiguace od 1 do 3 %.

V případě použití disambiguace na věty z jiného než trénovacího korpusu se kvalita významně zhorší, což je způsobeno nedostatečným rozsahem trénovacího korpusu a tím, že neobsahuje veškerá víceznačná slova. Přesnost disambiguace se pak pohybovala pouze okolo 50 %.

Do budoucna uvažujeme o použití paralelních korpusů pro disambiguaci s využití tezauru EWN. Vycházíme z předpokladu, že slova, která jsou víceznačná v jednom jazyce, jsou jednoznačná v jiném. Srovnáním kolokací v různých jazycích můžeme víceznačná slova disambiguovat.

6.2 Shlukování synsetů v EWN

Při testování klasifikátoru a návrhu indexátoru jsme zjistili, že jemnost členění jednotlivých významů slov v EWN je příliš veliká. Tudíž disambiguace pracuje s omezeným trénovacím korpusem na velkém počtu tříd, což snižuje její přesnost.

Druhý problém související s jemností EWN spočívá v navazování jednotlivých jazyků na jednotný index ILR. Přílišná jemnost vede k situaci, kdy se slovo podobného významu převádí na index, který nemá v jiném jazyku přesný ekvivalent, což činí problémy zejména při křížové klasifikaci. Vlastnost se projevila především při klasifikaci metodou Naive Bayes, kde byl velice malý průnik shodných lemmat jedné třídy v české a anglické části korpusu.

Pro eliminaci tohoto nedostatku je nutné vybrané synsety seskupit do větších celků a těmto shlukům přiřadit nové indexy. Tím bude množství klasifikačních tříd menší při zachování významu jednotlivých slov. Shluky lze vytvořit například na základě podobnosti kontextu, ve kterém se slova nacházejí.

Jinou možností je modifikovat klasifikační metody tak, že bude pro jedno slovo definováno více indexů s váhou. Tzn. slovo bude reprezentováno množinou slov (indexů). Váha slova s příbuzným významem se bude snižovat se vzdáleností od slova, které odpovídá lemmatu. Vzdálenosti lze určit průchodem v grafu EWN přes slova nadřazeného významu.

7 Závěr

Použitím tezauru EWN na klasifikační úloze bylo dokázalo, že se jedná o použitelnou metodu, která poskytuje slibné výsledky. Přestože je přesnost klasi-

fikace vícejazyčných korpusů pochopitelně o něco nižší než při klasifikaci jednojazyčných korpusů, lze považovat metodu za kvalitní s přesností pohybující se okolo 90 %. Po zahrnutí navrhovaných vylepšení lze očekávat další zvýšení přesnosti. V následujících měsících hodláme zahrnout navrženou klasifikaci do systému vyhledávání v multilinguálních korpusech.

Reference

1. Hynek J., Ježek K.: Automatic Dokument Classification Using Itemsets Metod, Its Modification and Evalution. Sborník konference Datacon 2001 Brno, M. Bieliková (Ed.).
2. Hynek J., Jezek K.: Dokument Classification Using Itemsets. Sbornik konference MOSIS 2000.
3. Manning C.D., Hinrich S.: Foundations of Statistical Natural Language Processing. The MIT Press, 2000.
4. Brown P.F., Della Pietra S.A., Della Pietra V.J., Mercer R.L.: Word-Sense Disambiguation Using Statistical Methods. Berkeley, 1991.
5. Maas D., Ripplinger B.: The AutIndex System. `http://www.iai.uni-sb.de/docs/autindex2.pdf`.
6. EuroWordNet: `http://www.illc.uva.nl/EuroWordNet/`.
7. Ispell: `http://fmg-www.cs.ucla.edu/fmg-members/geoff/ispell.html`.
8. Hajic J.: Morfologický analyzátor: `http://quest.ms.mff.cuni.cz/pdt/Morphology_and_Tagging/Morphology/index.html`.
9. Kučera M., Ježek K., Hynek J.: Kategorizace textů metodou NBCI. Znalosti 2004.
10. Joachims T.: A Probabilistic Analysis of the Rocchio Algorithm with TFIDF for Text Categorization. International Conference on Machine Learning (ICML), 1997.

Jiné programování*

Tomáš Holan

KSVI MFF UK Praha
Tomas.Holan@mff.cuni.cz

Abstrakt V prvním ročníku studia informatiky na MFF UK Praha všichni studenti absolvují úvodní kurs programování. V tomto kursu se seznámí se základními pojmy, algoritmy, datovými strukturami, technikami a se způsobem, jak analyzovat úlohu shora. Poslední přednáška tohoto kursu potom bývá věnována tomu, jak řešit úlohy, které jsou pro takový způsob programování příliš těžké.
Tento text shrnuje obsah této poslední přednášky. Postupně jsou předkládány problémy, které jsou pro obvyklý postup příliš složité a jsou demonstrovány alternativní způsoby jejich řešení.

1 „Klasické“ programování

Jako klasické programování chápeme takové způsoby programování, jaké jsou vyučovány v základních kursech na vysokých školách a jaké dobře vystihuje originální název knihy [**?**]: „Algorithms + Data Structures = Programs“.

Tento způsob je charakteristický snahou pochopit problém, rozdělit jeho řešení na menší části podle postupu řešení nebo podle různých případů hodnot vstupních dat a tyto části potom řešit pomocí určité standardní výbavy algoritmů (vyhledávání, třídění…), meta-algoritmů (rekurse, dynamické programování) a datových struktur. Některé úlohy jsou ovšem na tento postup příliš obtížné, veliké, neuchopitelné.

V následujících kapitolách si ukážeme několik takových úloh spolu s návody, jak je řešit „jinak“.

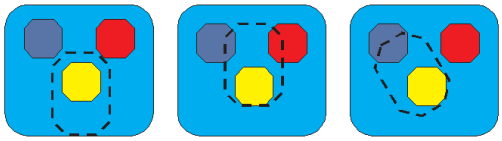
2 Problém na úvod: Vodovodní baterie

Páková vodovodní baterie obsahuje dvě hladké keramické destičky. První destička je pevná a jsou v ní tři otvory napojené na přítok studené vody, přítok teplé vody a odtok. Druhá destička je pohyblivá a je v ní prohlubeň.

Zvedáním páky se pohyblivá destička posouvá nad pevnou destičkou a jejími otvory, takže voda může prohlubní protékat z otvorů napojených na přítok studené a teplé vody do otvoru napojeného na odtok. Větší zvednutí páky přitom způsobuje větší posuutí a tím větší průtok vody.

Otáčením páky se mění směr, kterým se bude pohyblivá destička posouvat a tím lze ovlivňovat, zda a jakou měrou bude prohlubeň při posunu destičky

* Tato práce byla podporována grantem GA ČR číslo 201/02/1456.



Obr. 1. Páková vodovodní baterie.

zasahovat nad oba přítoky a nad odtok a tedy jaká bude výsledná teplota tekoucí vody.

Přirozený požadavek na pákovou baterii je, aby nastavená teplota vody, zůstala zachována bez ohledu na to, jak moc vody protéká. Tedy aby poměr ploch částí otvorů připojených na přítoky, zakrytých prohlubní v pohyblivé destičce, i při různé úrovni posunutí prohlubně zůstával stejný.

Problém:
Jaký tvar by měly mít otvory a prohlubeň?

Tento problém uvádíme jako příklad problému, na který nevystačíme s „klasickým“ programováním. Jeho řešení zatím odložíme.

3 Úloha 1: Nakreslení grafu

Zadání:
Napište program, který nakreslí zadaný graf. Graf je neorientovaný, bude zadán jako seznam vrcholů a hran, hrany budou kresleny jako úsečky a nakreslení má být hezké.

V zadání se uvádí, že hrany budou kresleny jako úsečky, jde tedy pouze o určení polohy jednotlivých vrcholů.

Zadání je poměrně neurčité, není těžké představit si nakreslení grafu, které hezké není — kde se budou hrany zbytečně křížit, budou procházet vrcholy, některé vrcholy budou příliš blízko u sebe — a naopak určitá nakreslení určitých grafů, která hezká budou. Krom těchto extrémů však zůstává určitá neurčitá oblast.

Kdybychom tuto úlohu řešili „klasicky“, můžeme se napřed omezit na řešení pro souvislé komponenty grafu a potom testovat, zda se nejedná o některý zvláštní případ, který umíme hezky nakreslit – kružnice, strom, cesta... Takové řešení ovšem bude pracné a stále bude zůstávat mnoho grafů, pro které žádné řešení mít nebudeme. Zkusme proto tuto úlohu řešit jinak.

Rada 1.: Máte-li úlohu, kterou nechcete nebo neumíte řešit ...
... řešte jinou úlohu!

Představme si, že vrcholy grafu odpovídají bodům, které se odpuzují silou klesající se čtvercem vzdálenosti. Naopak hrany působí jako guma nebo pružiny a přitahují příslušné vrcholy silou rostoucí se čtvercem vzdálenosti.

Program, který má hledat nakreslení grafu (souvislé komponenty, tento krok z klasického řešení si ponecháme) bude pracovat tak, že na začátku body rozmístí náhodně a potom opakovaně v každém kroku vždy pro jeden bod vypočte vektory přitažlivých a odpudivých sil a polohu bodu upraví ve směru daném součtem těchto sil.

Polohy bodů, ve kterých se tento pohyb ustálí, nebo které se alespoň už nebudou příliš měnit, použijeme jako výsledné polohy vrcholů grafu.

4 Úloha 2: Hádání obrázku

Tato úloha je umělá, slouží jen k tomu, abychom mohli předvést další techniku řešení úloh. Později uvidíme, že stejnou technikou lze řešit i úlohy opravdovější.

Máme napsat program, který uhádne obrázek uložený v jiné části programu (a zadaný uživatelem při startu programu). Takovýto typ úloh je poměrně častý — nevíme přesně, co máme dělat a jediné, co se dozvídáme je, že to, co děláme, není správně. Pokud se nám ale dostává informace, *jak moc* to není správně, máme napůl vyhráno. Můžeme totiž použít druhou radu:

Rada 2.: Máte-li úlohu, kterou nechcete nebo neumíte řešit...
...zapojte přirozený výběr!

Technika, kterou chceme použít, se nazývá *genetický algoritmus* a lze ji vysvětlit opravdu jednoduše — máme-li hledat řešení a máme k disposici *ohodnocovací funkci*, která každý předložený pokus o řešení ohodnotí, potom budeme postupovat tak, že si vytvoříme určitou množinu řešení, neboli *populaci*, ze začátku vytvořenou náhodně. Každý prvek populace necháme ohodnotit ohodnocovací funkcí a potom ze současné populace vytvoříme populaci novou, *novou generaci*.

Přitom prvky nové generace budou *některé prvky staré populace*, výběr se provádí náhodně, ale s přihlédnutím k ohodnocení jednotlivých prvků, takže ti jedinci populace, kteří mají lepší ohodnocení, mají vyšší pravděpodobnost, že se objeví i v další generaci — a dále také některé *nové prvky* vzniklé pomocí operací *křížení* a *mutace*.

Křížení znamená, že nového jedince složíme ze dvou jedinců staré populace — a opět s přihlédnutím k ohodnocení, takže úspěšnější jedinci mají větší šanci vyvést potomky.

Mutace označuje náhodně vniklou chybu, s určitou pravděpodobností pozměníme některý údaj nebo údaje v popisu jedince. K čemu je mutace dobrá, uvidíme za chvíli.

Ze stávající populace tedy vzniká populace nová, z ní zase nová a při vytváření nových populací mají vždy větši šanci přežít jedinci nebo jejich části (křížení), kteří dosahovali lepšího ohodnocení. Postupně by se tedy v populaci měli vyskytovat jedinci, kteří se budou více a více přibližovat hledanému řešení.

Náš program, který řeší úlohu hádání obrázku, používá zjednodušenou versi genetického algoritmu — krok vypadá tak, že z populace jen odstraníme prvek s nejhorším hodnocením a nahradíme ho novým prvkem vytvořeným zkřížením dvou nejlepších jedinců a použitím mutace.

Tento zjednodušený způsob je snazší naprogramovat (i když ani střídání celých populací není těžké naprogramovat) a navíc máme zaručeno, že nemůžeme ztratit již nalezené řešení s vysokým ohodnocením (protože ztrácíme vždy jen nejhorší řešení). Za toto zjednodušení platíme „ztrátou výkonu“, protože vlastně v každé nové generaci je jen jeden nový prvek, zatímco u vytváření celé nové populace by mohlo být nových jedinců více a tím by byla i větší šance rychleji najít lepší řešení.

Ohodnocovací funkce je jednoduchá, jedinec-obrázek je bod po bodu porovnán s (tajným) cílovým obrázkem, za každý uhádnutý bod je hodnocení zvýšeno, za každý neuhádnutý bod je hodnocení sníženo (což je zbytečné, mohlo by zůstat stejné).

Křížení probíhá tak, že každý bod obrázku nového jedince je náhodně vybrán od jednoho nebo od druhého rodiče.

Mutace potom znamená náhodnou změnu náhodně vybraného bodu.

Na pravděpodobnosti mutace záleží, jak rychle (a zda vůbec) genetický algoritmus najde řešení.

Kdyby mutace neexistovala (kdyby její pravděpodobnost byla rovna nule), nemohla by se v populaci objevit žádná nová informace.

Naopak pokud je pravděpodobnost mutace příliš vysoká, zmutovaní jedinci mutací ztrácejí informaci, kterou získali od svých předků.

Někdy se ovšem může vyplatit pravděpodobnost mutace měnit v průběhu výpočtu, například na začátku nastavit pravděpodobnost vyšší a po tom, co už byla dosažena určitá kvalita řešení, mutaci snížit.

Genetické algoritmy tedy představují metodu, jak hledat řešení, pokud dokážeme každý pokus o řešení popsat pomocí dat (gen) a pokud máme k dispozici ohodnocovací funkci. Jejich úspěšnost je různá, záleží zejména na tom, jak spojitý je vztah mezi genem a ohodnocením.

5 Úloha 2': Broučci

Genetický algoritmus si ukážeme ještě na dalších příkladech. V tomto příkladu nemáme jasné zadání, pokud bychom ho přece chtěli zformulovat, znělo by asi takto:

Zadání:
Mějme dán svět, ve kterém žijí broučci. Broučci se umějí pohybovat (otočit se doprava nebo doleva nebo udělat krok), vidí (před sebe a do stran) a jejich potravou jsou (ostatní) broučci. Najděte algoritmus, kterým se má brouček řídit, aby nezahynul hlady ani nebyl sežrán.

Když brouček nemá žádnou paměť, lze jeho algoritmus popsat pomocí funkce

$$(CoJeVlevo, CoJeVepredu, CoJeVpravo) \rightarrow Tah$$

Tuto funkci můžeme nahradit tabulkou.

Populace broučků je v programu viditelná, brouček je znázorněn šipkou otočenou ve směru pohledu broučka, barva určuje, kolik broučků již brouček snědl.

Vývoj neprobíhá po kolech, ale jednotliví broučci se střídají v tazích a vždy, když počet broučků klesne pod určitou mez, jsou vytvořeni noví broučci z nejúspěšnějších členů populace (rodiče se nemusejí setkat).

Pokaždé, když je vytvářen nový brouček, program vypisuje výkon (počet sežraných broučků) a algoritmus jeho rodiče — nejúspěšnějšího (žijícího) broučka.

```
Pred sebou:      NIC    BROUK
Napravo:         N B    N B
Nalevo N:        < >    ^ >
Nalevo B:        < >    ^ X
```

Obr. 2. Příklad algoritmu, který dokázal sežrat 22 broučků.

Zobrazený algoritmus bychom slovy mohli popsat třeba takto:

- je-li před tebou brouček a napravo nic, jdi dopředu (žer)
- je-li napravo brouček, otoč se doprava s výjimkou případu, že je brouček i před tebou i nalevo, pak nedělej nic
- není-li brouček před tebou ani napravo, otoč se doleva.

6 Úloha 2": Broučci s pamětí

Broučci v minulém příkladu nemají žádnou pamět, jejich algoritmy tedy představují přímočaré rozhodování na základě toho, co vidí. Jak by se změnilo chování a hlavně schopnosti broučků, kdyby si mohli pamatovat?

Paměť broučkům přidáme tak, že krom informace, co právě vidí, budou mít informaci o tom, v jakém jsou stavu. Stav bude proměnná z rozsahu jedna až maximální povolený počet stavů.

Algoritmus broučka bude tedy opět možné vyjádřit funkcí a tabulkou, pouze funkci přibude jeden parametr a tabulce další rozměr a výsledkem funkce a obsahem políčka v tabulce bude nejenom akce, kterou má brouček provést, ale také údaj, do kterého stavu má přejít.

Schopnosti broučků budeme měřit tak, že budou sbírat potravu v bludišti, a budeme vyhodnocovat, kolik potravy se kterému broučkovi podařilo nalézt.

Každé políčko bludiště může obsahovat potravu, neprůchodnou zeď nebo nic — tedy prázdný prostor, kterým lze projít.

Abychom ověřili všestrannost algoritmu, který brouček má, budeme každého broučka testovat na deseti bludištích, která se liší množstvím potravy, zdí a prázdných políček. Z každého bludiště určíme, kolik procent potravy brouček našel a výsledky (jako procenta nalezené potravy) za jednotlivá bludiště budeme skládat s tím, že nejhorší výsledek se bude započítávat s pětinásobnou vahou. Bludiště jsou vygenerována náhodně, ale pro všechny broučky stejná.

Hvězdičky označují potravu, písmena M překážky a mezery volný prostor, čísla nad každým bludištěm udávají počet hvězdiček. Bludiště je na toroidu, takže nejpravější sloupec sousedí s nejlevějším sloupcem a první řádek sousedí s posledním řádkem.

Souhrnné výsledky broučka použijeme jako ohodnocovací funkci pro genetický výběr v populaci broučků.

Naše zkoumání skončilo u dvou resp. tří stavů, protože už dva stavy stačily k tomu, aby brouček dokázal sebrat všechny dostupné hvězdičky kromě dvou, při třech stavech potom všechny hvězdičky až na jednu.

7 Vsuvka: Historka o magnetofonu

Na úvod k další radě uvedeme nejdříve jednu historku. Představme si, že máme magnetofon s počítadlem zaznamenávajícím otáčky navíjecího kotouče.

Zadání:
Napište program, který dokáže vzájemně převádět stav počitadla magnetofonu a odehraný čas.

Historka z názvu kapitoly spočívá v tom, že jsme změřili průměr navíjecího kotouče, tloušťku pásky, vyjádřili délku navíjené pásky... — a dostali cosi nepoužitelně nepřesného. Tím tento pokus skončil.

Po čase nás napadlo nestarat se o věci, jako je navíjecí kotouč nebo tloušťka pásky a prostě zkusit nějak aproximovat funkci určující vztah mezi otáčkami a časem.

Lineární ten vztah není, tak jsme zkusili kvadratickou funkci.

Kvadratickou funkci můžeme vyjádřit jako

$$y = Ax^2 + Bx + C$$

a potřebujeme tedy jednom určit hodnoty A, B a C.

K určení těchto hodnot a k určení celé kvadratické funkce stačí znát hodnotu funkce ve třech bodech. Nulovému času odpovídají nulové otáčky, zjistili jsme ještě stav počítadla po převinutí celé kazety (45 minut) a uprostřed. Vypočtené

koeficienty určily funkci — která, na rozdíl od původního výpočtu, dávala velice přesné výsledky! A to přesto, že tento výpočet neměl nic společného s tím, co vztah mezi počítadlem a časem vytváří, žádná tloušťka pásky, žádký průměr navíjecího kotouče... Pomohlo podívat se na problém jako na černou skřínku, nemyslet na to, co je uvnitř a jen sledovat jeho chování.

Rada 3.: Máte-li úlohu, kterou nechcete nebo neumíte řešit...
...zkuste se na ni podívat z odstupu!

8 Úloha 3: Rozpoznávání písmen

Tato úloha souvisí se známou úlohou rozpoznávání písma (OCR). Předpokládáme, že už byly nalezeny oblasti textu, odděleny řádky i jednotlivé znaky a nyní je třeba rozpoznávat písmena.

Zadání:
Napište program, který bude rozpoznávat písmena zadaná jako bitová mapa.

Předpokládejme, že máme dán rastrový obrázek, pro jednoduchost jsou jeho body pouze černé nebo bílé. Naším úkolem je pro daný obrázek určit, jaký znak je na něm zobrazen.

Tato úloha bývá řešena mnoha „klasickými“ způsoby, ukažme si jedno jiné řešení a před ním další radu:

Rada 4.: Máte-li úlohu, kterou nechcete nebo neumíte řešit...
...omezte se na řešení v určitém tvaru!

Naše, poněkud naivní, řešení bude založeno na úvaze, že je-li písmen 26, je to méně než 2^5 a tedy pro rozlišení jednotlivých písmen si stačí z celého rastru všímat pouze pěti bodů.

Algoritmus rozpoznání potom bude triviální, barvy oněch pěti bodů mohou tvořit celkem 32 kombinací a nahlédnutím do tabulky pro každou kombinaci zjistíme, o jaké písmeno se jedná.

Zbývá nám jiná úloha, jak najít správných pět bodů.

Neřešíme tedy už úlohu rozpoznávání znaků, řešíme úlohu, jak nalézt pětici bodů na obdélníku daných rozměrů, která má tu vlastnost, že pro každé z dvaceti šesti písmen dává jinou kombinaci černé a bílé barvy.

Způsob, jak takovou pětici nalézt už známe — hledáme vlastně deset čísel $x_1, y_1, x_2, y_2 \ldots x_5, y_5$, pro která platí

$$F(x_1, y_1, x_2, y_2 \ldots x_5, y_5) = 0\,,$$

kde funkce F je funkce, která pro jakoukoliv pětici bodů, neboli desetici svých parametrů, spočte počet chyb–kolizí, případů, kdy dvě různá písmena dala stejnou kombinaci barev.

Funkci F je snadné spočítat i naprogramovat a protože v ní máme ohodnocovací funkci, můžeme naše body najít pomocí genetického algoritmu.

Poznámka:
Genetický algoritmus není jediná možnost a nijak to neznamená, že úlohu rozpoznávání znaků řešíme pomocí genetického algoritmu! Genetický algoritmus použijeme pouze jednou, v době tvorby našeho rozpoznávacího programu, aby nám nalezl vhodné koeficienty; ostatně kdybychom měli dost trpělivosti, dokázali bychom je zřejmě najít i prostým mechanickým prohledáním všech pětic bodů.

Tato úloha je samozřejmě zjednodušená, předpokládáme jen jeden typ písma, nepředpokládáme chyby, hledání bezkolizní pětice také může trvat dlouho... ale nic nám nebrání místo pětice hledat třeba desetici bodů a navíc si třeba do ohodnocovací funkce předepsat, že počítá kolize i při změně jedné hodnoty — tj. že požadujeme, aby se kombinace v bodech lišila ne alespoň v jedné, ale třeba ve třech hodnotách. Tím získáme odolnost vůči chybě — i když někde bude místo černé bílá nebo místo bílé černá, naše body budou stále správně určovat písmeno.

Zrovna tak nemusíme vyhodnocovat na dvaceti šesti písmenech, ale můžeme mít testovací data, kde budou písmena různých znakových sad nebo různě zmrzačená, zašuměná atd. Pokud algoritmus nebude konvergovat, můžeme zvyšovat počet bodů.

9 Úloha 4: Obchodování s akciemi

Ukažme si ještě jeden příklad na hledání řešení v určitém tvaru. Bude se týkat obchodování s akciemi, konkrétně rozhodování, zda v určité situaci akcie určitého druhu koupit nebo prodat nebo držet.

Možná jsme už někde slyšeli „Akcie klesají, musím je prodat!“ a jindy zase „Akcie klesají, je čas nakupovat“, ze stejných podmínek různé závěry. A možná bychom chtěli znát „ta správná“ pravidla.

Zadání:
Napište program, který nalezne pravidla přinášející při obchodování s akciemi největší zhodnocení. Vstupní informace pravidel jsou kurs a objem obchodování dnes a ve třech okamžicích v minulosti – před 2, 20 a 100 obchodními dny.

Pravidla budeme hledat geneticky (rada číslo 2). Prvkem populace bude obchodník, který má určitou soustavu pravidel, jak obchodovat. Ohodnocení této soustavy pravidel-genů proběhne tak, že obchodníkovi dáme určité množství peněz, necháme ho, podle jeho pravidel, obchodovat na skutečných údajích z pražské bursy a uvidíme, jak se mu podaří danou výchozí částku zhodnotit.

Omezíme se na pravidla v určitém tvaru (rada číslo 4 a zadání úlohy).

Mohli bychom ohodnocovat samostatná pravidla, ale místo toho bude mít každý obchodník pravidel více a každému pravidlu přiřadíme určitou váhu.

Mohli bychom vybírat pravidlo s největší váhou ale tím bychom ztráceli výhodu toho, že máme více pravidel. Místo toho vyhodnotíme všechna pravidla a u těch, která mají splněné předpoklady, připočteme jejich váhu k váze akce, kterou doporučují. Tedy jakési hlasování, kde různá pravidla mají různě silný hlas, ale i na tom, kdo má hlas nejslabší, může někdy záviset výsledné rozhodnutí.

Rada 5.: Máte-li úlohu, kterou nechcete nebo neumíte řešit...
...a potřebujete-li vybírat mezi více možnostmi...
...přihlédněte k různým hlediskům...
...přiřaďte jim váhy...
...a nechte je hlasovat!

(Správné váhy můžete najít geneticky.)

Typ pravidla potom bude:

```
type
   TGen = record
            Hodnota: real;
            case integer of
            20: (A2:array[0..2] of { nakup, prodej, drzet }
                      array[1..MaxDozadu] of
                      record
                         prahK, vahaK,
                         prah0, vaha0: real;
                      end);
            21: (DNK: array[1..DelkaGenu] of real)
          end;
```

Pro každou z akcí a pro každý den v minulosti, se kterým můžeme srovnávat, máme dvě podmínky, jednu pro objem a jednu pro kurs, přičítající svou váhu k váze této akce, pokud poměr kursu resp. objemu tehdy a dnes překročil daný práh.

Pro práh větší než 1.00, například 1.10 to přitom znamená, že poměr je *větší* než hodnota prahu, pro práh menší než 1.00, například 0.95 to znamená, že poměr je *menší* než hodnota prahu. Tím se zbavujeme nutnosti udávat znaménko nerovnosti.

V deklaraci je vidět, že se zároveň na celou genetickou informaci můžeme dívat jako na pole hodnot typu real, což nám usnadní křížení a mutaci.

Například z údajů o obchodování s akciemi Komerční banky za období od 5.4.1994 do 19.6.2002 byla nalezena pravidla zhodnocující počáteční vklad více než sedminásobně.

Pozor:
Není zajímavé, že použijeme-li data za osm let obchodování s jedním druhem akcií, dokážeme najít množinu nákupů a prodejů, která počáteční investici několikrát znásobí. Zajímavé je, že dokážeme najít pravidla, při jejichž dodržení lze počáteční investici takto zhodnotit!

Pozor 2:
Přenositelnost takto naučených pravidel na jiný cenný papír nebo i na stejný cenný papír v jiném čase... je velice špatná.

10 Úloha 5: Vodovodní baterie

Už víme, jak hledat odpověď na otázku položenou v úvodu.

Gen bude tvořen dvěma maticemi (obrázky, bitovými mapami) popisujícími obě destičky. Šedá barva znamená nic, červená přítok teplé, modrá přítok studené, žlutá odtok, černá prohlubeň. (Pokud vidíte obrázek bez barev... černou poznáte, šedá je ten odstín šedé, který je v levé i pravé polovině obrázku, modrá a červená se vyskytují pouze v levé polovině obrázku a to symetricky, a žlutá je ta zbývající barva v levé polovině obrázku.) Hledat budeme geneticky, vyhodnocovací funkce hodnotí odchylku vypočtených průtoků od křivky požadovaných průtoků při různých pootočeních a posunech.

Obr. 3. Výsledné tvary a průtoky.

Do ohodnocovací funkce jsme zapomněli přidat podmínku, že otvory mají být souvislé... tuto úpravu ponecháváme čtenáři.

11 Závěr

Uvedli jsme několik technik, jak postupovat při řešení úloh příliš obtížných pro standardní způsoby řešení. Jednotlivé techniky jsou presentovány na řešení sedmi úloh. Plný text, obrázky a zdrojové texty lze najít na adrese

`http://ksvi.mff.cuni.cz/~holan/jinak`

Reference

1. Wirth N.: Algoritmy a štruktúry údajov. ALFA, Bratislava 1988.

Opomíjené aspekty profilu absolventů informatických oborů*

Jaroslav Král, Michal Žemlička, and Alena Koubková

Univerzita Karlova, Matematicko-fyzikální fakulta, Katedra softwarového inženýrství
Malostranské nám. 25, 118 00 Praha 1
{kral,zemlicka,koubkova}@ksi.ms.mff.cuni.cz

Abstrakt Současné trendy v softwaru, jako je orientace na služby, agilní formy vývoje, využívání customizovaných systémů, programování webových stránek a outsourcing směřují k výrazné redukci podílu klasického programování na vývoji softwaru. Roste naopak rozsah činností, při kterých je nutná úzká spolupráce s neinformatiky. Tento trend vyvolává stále urgentnější potřebu vyššího porozumění znalostních oborů uživatelů, užší spolupráce s uživateli a stále širšího používání hotových systémů. Potřeba klasického programování klesá, zatímco specifikace požadavků a také podpora provozu systémů jsou přesunutelné do zahraničí v daleko menší míře, neboť závisí na kultuře uživatelů, jejich jazyce a místních kontaktech a zvycích. Mnozí absolventi informatických oborů však přeceňují svoji schopnost dobře programovat a neradi spolupracují s odborníky jiných oborů. Symptomem tohoto přístupu je kromě jiného neochota akceptovat přístupy experimentálních věd, jednat se zákazníky a používat matematickou statistiku jako důležitý nástroj při své práci a při spolupráci s uživateli. Důležitým prostředkem změny je správná integrace matematické statistiky do znalostního profilu informatiků. Je to obtížné, neboť u mnohých studentů je odpor k experimentálním vědám a introverze výrazný osobnostní rys a důvodem, proč přicházejí studovat programování. Současná kurikula na tento problém nedostatečně reagují. Součástí kurikula by proto mělo být dostatečné množství výuky experimentálních oborů a komunikačních dovedností a měla by se více používat statistika jako nástroj poznávání světa včetně informatiky. Matematická statistika a teorie pravděpodobnosti jsou v informatice silně podceňovány i jako zdroj poznatků důležitých pro informatiku, jako je výzkum a analýza algoritmů, plánování a řízení softwarových prací a specifikace požadavků.

1 Úvod

Současné trendy v softwaru, jako je orientace na služby nebo outsourcing směřují k výrazné redukci podílu klasického programování na vývoji softwaru. Roste naopak rozsah činností, při kterých je nutná úzká spolupráce s neinformatiky (uživateli). Ta je také nutná u některých moderních forem vývoje softwaru, jako jsou

* Tato práce byla podporována grantem Grantové agentury ČR č. 201/03/0911 a projektem "Informační společnost" číslo 1ET100300517.

agilní formy vývoje nebo návrh a implementace rozhraní služeb v servisně orientovaných architekturách softwarových systémů. Klíčovým požadavkem je schopnost komunikovat s uživateli a tedy chápat jejich obor znalostí a jejich filosofii řešení problémů. To vyžaduje používat jazyk a pojmy, kterým uživatel rozumí (tedy spíše odborný jazyk uživatele), chápat potřeby a cíle uživatelů, v jistém smyslu být schopen myslet jako oni. Vyžaduje to také schopnost být schopen komunikovat s lidmi. Znalostní obory různých uživatelů mohou být různé. Jednou z podstatných podmínek je proto akceptace přístupů experimentálních věd a schopnost používat matematickou statistiku podobně, jak ji používají experimentální vědy. Statistika je důležitá při řízení projektů, měření softwaru (sběr a analýza softwarových metrik), ve funkcích systému požadovaných uživateli (především ve funkcích podporujících činnost managementu), analýze algoritmů a datových struktur a pro měření kvality dat a kvality a efektivnosti softwarového systému.

Matematická statistika by měla být daleko více integrována do výuky informatiky počínaje úvodními přednáškami. Jinak hrozí, že studenti ještě více omezí svou schopnost komunikace s lidmi, zvláště s těmi, pro které je informatika pomocný obor. Pak ovšem budou studenti informatiky znevýhodnění v kariérním růstu a postupně i na celosvětovém trhu práce. Spolu se statistikou je samozřejmě nutné cvičit komunikační schopnosti včetně základů odborného stylu a používání přirozeného jazyka a seznámit se s některými experimentálními vědami, hlavně s tím, jak experimentální vědy poznávají svět.

Pro ilustraci tohoto problému uveďme několik výroků:

- "Já o ty nafoukané informatiky nestojím. Snáze doučím strojaře programovat, než informatika spolupracovat s uživateli." (vedoucí střední softwarové firmy, Bochum)
- "Já nemohu ty arogantní programátory pustit k uživatelům. Hned je svou nafoukaností a neschopností se vyjádřit naštvou a ohrozí tím celý projekt. S uživateli ale musí někdo spolupracovat. A tak sháním jinde." (vedoucí softwarové firmy, Brno)
- "Jsem s informatiky z místní univerzity docela spokojen. To, že umí programovat, je cenné, ne však rozhodující. Programovat je dokážeme, pokud mají slušné základy, snadno naučit. Více nám ale pomáhá, že umí aplikovat ve své práci neinformatické znalosti (např. dovednosti související s uměleckou grafikou), které jsou mimo naše znalosti. Zčásti to platí i pro moderní informatické technologie, např. grafické databáze." (vedoucí jiné softwarové firmy, Brno)
- v Austrálii je dnes průměrná nezaměstnanost 5,5%. Nezaměstnanost informatiků je 11%, ale programátorů plných 18% (z vystoupení Prof. J. Voříška na konferenci DATAKON 2004)

V USA již nejsou informatické profese zárukou dobrého zaměstnání. Účastníci konference IRMA2004 byli v úvodní přednášce seznámení s plánem státu Luisiana na vybudování společného pracoviště průmyslu, státu a universit s cílem omezit outsourcing vývoje softwaru do Indie. Jiné řešení nabízí [1].

Podle mnoha průzkumů, viz např. [2] nebo www.standishgroup.com, více než 80% nákladů na nápravu chyb, ke kterým došlo při vývoji softwaru, jde na vrub chyb, ke kterým došlo při specifikaci požadavků. Chyby ve specifikacích vznikají prakticky výhradně jako důsledek problémů ve spoluprácí mezi uživateli a vývojáři (neporozumění, problémy mezilidské komunikace a spolupráce). Problémy ve spolupráci mezi vývojáři a uživateli a nedostatky v oblasti managementu softwarových i uživatelských firem jsou rozhodující pro úspěch nebo neúspěch softwarových projektů. Pokud chceme úspěšně realizovat softwarový projekt, musíme se postarat o dobrý management projektu jak se strany naší softwarové firmy, tak i o manažerskou podporu se strany managementu zákazníka. Musíme umět najít správné lidi u uživatele pro specifikaci požadavků, navázat s nimi spolupráci a chápat jejich požadavky a správně je formulovat. Je proto žádoucí, aby naši studenti dobře rozuměli i problematice managementu. Potřebují to často i při specifikaci požadavků na funkce systému, které jsou stále častěji určeny k podpoře managementu.

Všimněme si, že naše klíčová zjištění vychází ze statistické analýzy dat o softwarových projektech. Statistika je klíčovým nástrojem efektivního vývoje softwarových systémů. Uvidíme, že může přispět i k prevenci některých nevýhodných vlastností a postojů informatiků. Klíčem k úspěchu jsou schopnost komunikovat a používat adekvátně přirozený jazyk, schopnost spolupracovat s zákazníky a používat statistické nástroje.

Blízko ke klasickému programování má vývoj domácích stránek. To však také vyžaduje schopnost spolupracovat s uživateli a znalosti z oblasti psychologie a estetiky. Navíc je dosti pravděpodobné, že si mnohé stránky koncoví uživatelé naprogramují sami (viz End User Programming, `www.cs.uml.edu/~hgoodell/EndUser/`, nebo CACM 9/2004).

Absolventi informatiky se zatím na trhu práce dobře uplatňují. Nemusí to tak ale být stále. Možné hrozby se mohou uplatnit v poměrně blízké budoucnosti. Vědomí této skutečnosti zatím neproniklo do studijních plánů výuky informatiků na universitách i technikách, včetně plánů mezinárodně uznávaných jako jsou doporučení ACM nebo EUCIP, dokonce i CC-2004. Například problematika specifikace požadavků a statistické metody hodnocení kvality jsou zmiňovány dosti okrajově a to ještě s orientací na výhradně objektový pohled.

2 Servisní orientace a potřeba opustit kyberprostor

Těžiště zájmu softwarového inženýrství se v posledních letech přesunuje k problematice vývoje rozlehlých softwarových systémů a k podpoře činnosti managementu. To je spojeno s rostoucím využíváním existujících aplikací a produktů třetích stran, užší spoluprací s uživateli (včetně jejich managementů) a vyšším stupněm kvality analýzy dat a kvality samotných dat.

Tyto trendy vedly k přechodu k *servisně orientovaným architekturám* (SOA), což jsou virtuální peer-to-peer (p2p) sítě autonomních komponent, které se chovají jako služby v systémech hromadné obsluhy. Z toho důvodu se takové softwarové komponenty nazývají službami. O architektuře takových systémů říkáme, že

je servisně orientovaná. Důvodů k použití principů servisní orientace je mnoho. Historicky nejstaršími servisně orientovanými systémy byly systémy podporující spolupráci reálných procesů, které měly samy charakter služeb, jako jsou systémy řízení výrob [3]. Klíčové principy servisní orientace lze vysledovat v operačních systémech používajících symetrický multiprocessing a leccos dokonce v kobolských systémech z šedesátých let.

Servisní orientace se stává rozhodujícím paradigmatem současného softwarového inženýrství z dalších důvodů, které se staly aktuálními v posledních letech. Moderní hardware umožnil konstrukci snadno použitelného systému globální spolupráce aplikací (middlewaru). Ukázalo se, že opravdu velké systémy nelze koncipovat jako logické monolity, neboť je nutné z důvodů organizačních, ekonomických, politických a především softwarově inženýrských používat existující aplikace, aplikace třetích stran a nové aplikace vyvíjené autonomně. Řešením je servisní orientace. Servisní orientace umožňuje, aby software měl klíčové softwarově inženýrské přednosti jako modifikovatelnost, používání dávkových aplikací, různé formy ladění a různé formy datové podpory, inkrementální vývoj atd. [4], [5].

Stabilita řešení a další uživatelsky důležité vlastnosti servisně orientovaných softwarových systémů silně závisí na hranicích a rozhraní služeb. A také na tom, zda se komunikující služby navzájem znají, jako je tomu v případě řízení výrob, *e*-governmentu a mnoha dalších situacích, a zda nikoliv, tj. zda musí před začátkem komunikace partnera nejdříve vyhledat, jako je tomu v *e*-komerci.

Hranice služeb a rozhraní služeb v SOA [4], [6] by měly být uživatelsky orientovány. To znamená, že jejich rozhraní musí být podobné rozhraním služeb reálného světa. To vyžaduje úzkou spolupráci informatiků a uživatelů. Stále větší úsilí je věnováno podpoře manažerských činností. To vše vyvolává potřebu užší spolupráce s uživateli na všech stupních hierarchie a rostoucí potřebu rozumět problematice analýzy a kvality dat pro potřeby managementu uživatele, pro hodnocení vlastností produktu, jako je efektivnost a spolehlivost, a pro kontrolu a optimalizaci procesu vývoje, provozu a údržby systému. Úzká spolupráce s uživateli je podmínkou uplatnění moderních trendů ve vývoji softwaru, jako je extrémní programování [7] nebo agilní formy vývoje [8]. Studijní profily to ale dostatečně nezohledňují.

Shrneme-li výše uvedenou diskusi, závisí schopnost spolupracovat s uživateli na třech hlavních podmínkách:

1. na schopnosti porozumět potřebám uživatelů;
2. na schopnosti s uživateli spolupracovat, což kromě porozumění potřebám vyžaduje jistou úroveň schopností diskutovat a vyjadřovat se;
3. na ochotě se s uživateli vůbec bavit a překračovat hranice informatiky[1].

Hlavní principy orientace na služby jsou známy velmi dlouho. Je také již od dob COBOLu známo, že jsou servisně orientované systémy neobyčejně stabilní. Mohou být používány prakticky bez údržby po celá desetiletí [4], [9]. Totéž je známo

[1] Je třeba, aby studenti informatiky získali respekt k dalším (neinformatickým) oborům i proto, že odtud mohou získat mnohé užitečné poznatky a stimuly.

o SOA systémech řízení výrob. Přesto se orientace na služby prosazuje velmi pomalu, neboť pro mnohé výrobce softwaru není příjemné změnit své systémy tak, že by mohlo dojít ke snížení otrocké závislosti uživatelů na produktech daného výrobce. Je tomu tak hlavně proto, že uživatelé jsou dnes stále ještě přesvědčeni, že nejlepší je informační systém dodaný jediným výrobcem na klíč jako řešení použitelné po dlouhou dobu beze změn.

Informatici se na druhé straně musí se naučit uplatňovat novou filosofii přístupu k věci. To ovlivňuje nejen principy řešení, včetně specifikace požadavků, ale také marketing a metody spolupráce s uživateli. To, že jsou principy servisní orientace známy po dlouhou dobu, a přesto se obtížně prosazují, naznačuje, o jak obtížnou změnu se jedná.

Informatici si musí např. zvyknout neprogramovat od začátku, myslet v termínech služeb, používat i dávková řešení atd. Informatici se nejraději pohybují ve virtuálním světě uvnitř počítače (kyberprostoru), kde se nemusí starat o uživatele a reálný svět. Moderní trendy v softwaru je nutí tento prostor opustit.

3 Psychologické bariéry

Mnozí absolventi informatických oborů přeceňují svoji schopnost dobře programovat. Převažuje u nich podceňování neprogramátorských a nepočítačových oborů. Často se domnívají se, že programování je věcí intelektuální soutěže. Neradi komunikují s odborníky nepočítačových oborů. Tento přístup nazveme hackerským syndromem, protože bývá velmi silně vyvinut u hackerů. Samozřejmě zde míníme hackery v původním smyslu, kteří se nesnažili o zisk kriminální cestou. Symptomem takového přístupu je výše zmiňovaná neochota akceptovat postupy experimentálních věd a používat matematickou statistiku jako důležitý nástroj při své práci. Informatika je obor, kde mohou skutečně schopní jedinci prokázat svoje intelektuální kvality – ať již v poněkud zvrhlé formě tvorby virů, nebo v chvályhodné formě v oblasti volně dostupného softwaru, jako je mnoho aplikací pro Linux i Linux samotný.

V informatice pravidla, jak se bude softwarový produkt chovat, často, byť mnohdy jen zdánlivě, určuje informatik. Může tedy přitom zůstat ve svém světě, v kyberprostoru. To do značné míry diskvalifikuje informatiky pro kvalifikovanější práce v raných stádiích životního cyklu softwaru, při řízení vývoje softwaru a při prodeji softwaru. Zároveň je odsuzuje k činnostem, které se dají poměrně snadno přesunout do Asie. Schopnost programování se poměrně rychle snižuje s věkem. Lidé orientovaní na kyberprostor nebývají vhodní pro práci v týmech. Při práci v týmech pak často představují hrozbu známou jako antipattern Corncob, viz www.antipatterns.com/briefing/.

4 Výcvik praktických dovedností

Pro výcvik potřebných dovedností a především pro prevenci nevhodných postojů, je důležité, aby se již v úvodních přednáškách z informatiky a při programátorských činnostech posluchači seznamovali s problematikou celého životního cyklu

softwaru včetně jeho statistických aspektů. To je v podmínkách školy obtížné, zvláště při výuce dovedností potřebných pro specifikaci požadavků. Praktické zkušenosti s komunikací s uživateli je na škole velmi obtížné získat Částečné řešení tohoto problému je požadavek, aby studenti se brzy naučili zpřesňovat rámcové zadání.

U bakalářských prací by se měly podporovat týmové projekty. Prostor pro týmové projekty nezávislé na bakalářských pracích se po zavedení bakářského stupně studia zmenšil. S tím je spojen problém individuálního zadání a individuálního hodnocení prací studentů. Řešením by mohlo být využití principů servisně orientovaných systémů, kde lze jednotlivé části vyvíjet autonomně a dosáhnout odpovědnosti jednotlivých pracovníků za jejich kvalitu.

Dalším opatřením, které se např. na MFF UK v Praze a na FI MU v Brně celkem osvědčilo, by mohlo být zařazení přednášek o odborném stylu a důraz na kvalitu uživatelské dokumentace a specifikace požadavků v informatických přednáškách a výcvik dovedností jak komunikovat a jak prezentovat. Dokumentace by měla mít minimální rozsah nutný k tomu, aby se dal systém nezávisle znovu vyvinout, a hlavně, aby se dal používat bez přítomnosti autora (viz agilní formy vývoje). To by mělo být pravidlem při psaní bakalářských a diplomových prací a pro dokumentaci systémů vyvinutých v jejich rámci. Je to oto důležitější, že moderní doba ohrožuje vývoj dovedností mezilidské komunikace. Méně se čte, méně se diskutuje, hrají se počítačové hry a mnoho času pohltí televize a video.

Velmi brzy by měli být studenti seznamováni s“metodikami experimentálních věd, jako je fyzika či sociologie, a statistickými metodami zjišťování poznatků o světě. Tento problém je urgentnější na universitách než na technikách. Na technikách zase hrozí problém memorování, který je v některých případech nutný, jindy brání přemýšlení. Již v bakalářském studiu by měli být studenti seznamováni s moderními metodami řízení prací, v případě nutnosti i na úkor specializovaných informatických poznatků a možná i klasických matematických poznatků.

Integrace zásad matematické statistiky a experimentálních věd je důležitá pro informatiku samotnou, pro prevenci hackerského syndromu, a také pro kariérní možnosti informatiků (aby například obstáli v soutěži s ekonomy či experimentálními fyziky, z nichž mnozí úspěšně pracují na informatických místech).

5 Integrace matematické statistiky do studia

Matematická statistika se uplatňuje při návrhu a analýze softwarových nástrojů a při řízení softwarových projektů. Je stále důležitější ve funkcích požadovaných uživateli a při specifikaci požadavků, například při řešení problému, zda kvalita dat umožňuje splnit určitý požadavek. Je rovněž důležitá při obsluze softwarových systémů a pro porozumění neinformatickým oborům. Je velmi pravděpodobné, že sama o sobě může být užitečná při prevenci nevhodných postojů.

Bohužel je dnes matematická statistika často považována mezi studenty informatiky za něco nepodstatného, obtížně pochopitelného, málo použitelného a dokonce protivného. To je do značné míry způsobeno i tím, že matematické

statistice a teorii pravděpodobnosti je v bakalářském studiu obvykle věnována jediná jednosemestrální (někdy dvousemestrální) přednáška. Ve statistických přednáškách se málo používají příklady z informatiky. A naopak ani v informatických přednáškách se matematická statistika, bohužel, nepoužívá.

V rámci současných studijních plánů se studenti učí mnoho úzce zaměřených a rychle zastarávajících informatických znalostí a dovedností, jako je velmi detailní znalost stavby operačních systémů nebo překladačů. To je samozřejmě důležitá součást profesních znalostí. Jde však o znalosti, které jsou většinou v praxi použitelné (na rozdíl od matematické statistiky) jen nepřímo. Málokdo bude například programovat kompilátor. To však neznamená, že by o tom neměl student nic vědět. Jde ale o míru a rozsah znalostí, což je věcí optimalizace znalostního portfolia.

Podíváme-li se na nabídku přednášek našich vysokých škol ale také doporučení EUCIP 2004 nebo CC 2001 od ACM, lze se jen obtížně zbavit pochybností, zda je profil studia optimální. Předběžný návrh CC-2004 (www.acm.org/education/Overview_Draft_11-22-04.pdf) představuje jistý pokrok, ale např. význam statistiky nebyl ani zde podle dostupných informací rozpoznán.

Matematická statistika je bohužel často přednášena pouze jako suchá matematická disciplína (definice – věta – důkaz), a nikoli i jako nástroj poznávání světa. Studenti informatiky se jen neradi smiřují s faktem, že svět není černobílý a něco platí jen někdy, a přesto s tím musíme počítat, a naopak, je-li něco velmi málo pravděpodobné, lze to zanedbat. To je vážný profesní nedostatek, protože to znamená, že naši studenti nejsou schopni kvalifikovaně řešit mnoho důležitých problémů.

6 Jak překonat odpor informatiků ke statistice

Jak jsme uvedli, většina informatiků – především ti, kteří nade vše milují počítače a berou tvorbu softwaru jako intelektuální hru – statistiku neznají a nejsou přesvědčeni že je vůbec potřebná. To je obvykle spojeno s nechutí spolupracovat s uživateli a chápat jejich potřeby. To je značná nevýhoda, protože v SOA (a také v agilních formách vývoje) roste význam stálé spolupráce s uživateli [4], [6].

Z výše uvedené diskuse vyplývá, že je reálná naděje, že pokud se informatici smíří s matematickou statistikou jako nástrojem analýzy dat, budou i snáze schopni komunikovat s uživateli a budou schopni lépe rozumět jejich potřebám. Toto přesvědčení je založeno na dlouhodobých zkušenostech autorů z výuky informatiky, statistiky i z praxe vývoje softwarových systémů. Statistika je důležitou součástí hodnocení kvality dat. Současně je ovšem nutné řešit nedostatek komunikativnosti informatiků [10]. Jako bonus lze získat znalosti potřebné ke zlepšení softwarových procesů u softwarových firem.

Problém je i v tom, že v informatické literatuře existuje málo dobrých příkladů aplikací statistické analýzy dat. Matematická statistika se učí příliš pozdě a spíše jako abstraktní matematická disciplína. Pravděpodobně by bylo lépe slevit trochu z exaktnosti a učit pravděpodobnost a statistiku spíše jako kuchařku, aby se její poznatky mohly více využívat v informatických předmětech. Jednodu-

ché aplikace matematické statistiky v informatice vyžadují pouze středoškolské znalosti, nevyžadují proto absolvování přednášky ze statistiky, a lze je tedy zahrnout do úvodních přednášek z programování. Exaktní prezentace statistiky by měla být součástí magisterského studia. Je důležité do této přednášky zahrnout příklady ukazující, že moderní statistické nástroje mají mnohem větší vypovídací schopnost než hrubé grafické intuitivní postupy, které je nutné používat v bakalářském studiu. Nedostatečnou integrací statistiky do studijních plánů trpí (viz výše) i nejnovější návrhy profilů studia včetně nejnovějšího pracovního návrhu CC-2004.

Pedagogové na informatických katedrách to mohou považovat za potvrzení toho, že lze při výuce informatiky vystačit s metodami orientovanými na kyberprostor a že se nemusí starat o prevenci hackerského syndromu. Problém je i na straně statistiků, kteří se nesnaží založit výuku pravděpodobnosti a statistiky pro informatiky na příkladech z jejich oboru. Ani v tomto případě však nejsou informatici zcela bez viny, neboť neposkytují statistikům vhodné příklady. Některé takové příklady jsou uvedeny níže.

7 Kdy jaké statistické poznatky

Statistické znalosti lze z hlediska jejich použití v informatice rozdělit do několika skupin.

1. Poznatky, které mohou být součástí specifikace požadavků na funkce softwaru. Příkladem jsou podklady pro rozhodnutí managementu. Tyto znalosti lze často uplatnit i při řízení vývoje softwaru.
2. Výsledky použitelné při vývoji systému. Příkladem je rozhodnutí, kdy končit testování a obecně vyhodnocování metrik kvality softwaru.
3. Hodnocení kvality softwaru a kvality dat.
4. Statistická a pravděpodobnostní analýza algoritmů a datových struktur.
5. Měření výkonu.

Je důležité, aby se statistické metody analýzy informatických dat objevovaly jako příklady v přednáškách z matematické statistiky a také v příslušných informatických přednáškách, pokud možno již v úvodních informatických kursech. Uveďme některé příklady.

Účinnost testování Je známo, že testování částí odhalí 24% chyb, integrační testování 24% chyb a testování systému 36% chyb. Jaká je účinnost provedení všech tří testů? Zmínit se o předpokladu nezávislosti a také o tom, že se spíše odhalí ty chyby, které mají větší pravděpodobnost, že způsobí chybnou práci systému. Takže výsledek není tak hrozivý, jak vypadá, sláva to ale není.
Je-li účinnost vyhledávání chyb při testování prováděné jeden týden x, jaká bude účinnost testování prováděné n týdnů? Předpokládáme při tom, že jsme schopni účinně testovat po celou dobu n týdnů. Jednoduchým závěrem je, že nelze smysluplně požadovat, aby software neměl žádné chyby.

Kdy ukončit testování (a také inspekce) Kriterium pro ukončení testování lze založit na vyhodnocování, zda frekvence selhání (incidentů, tj. případů chybné práce systému) při testování klesla pod určenou mez. To lze vyhodnocovat graficky již při testování prvých složitějších programů. I to umožňuje ukázat, proč u složitějších programů nelze dosáhnout "nulové chybovosti". V pozdějších přednáškách lze aplikovat jednodušší metody a základní principy teorie spolehlivosti včetně odhadu průměrné doby mezi poruchami.

Scrambling Při přenosu dat po vodiči je pro některá zařízení důležité, aby se počet nul a jedniček ve každé delší zprávě Z přibližně rovnal. Scrambling tento problém řeší tak, že odchozí zprávu superponuje pomocí operace **xor** s náhodnou posloupností bitů N (vzorkem bílého šumu, bity jsou nezávislé a obě jejich hodnoty stejně pravděpodobné), čímž získáme zprávu Z'. Na příjmu se provede superpozice N **xor** Z'.

Hodnocení kvality a změn kvality částí softwaru a kvality pracovníků Příkladem je hledání komponent s významně vyšším počtem problémů na 1000 řádků programu. Lze použít histogramy (např. v Excelu) v časných přednáškách a statistické metody založené na použití t-testu, případně jiné statistické postupy v pozdějších přednáškách. Histogramy pro hodnocení výkonu poskytují některé databázové systémy. Podobnými technikami lze hodnotit kvalitu práce programátorů či efekty použití vývojových metod a nástrojů v softwarovém inženýrství. Pro složitější softwarové produkty platí poznatky teorie spolehlivosti (např. známá vanová křivka). Při řízení prací na softwaru bývá nutné rozhodnout, zda je třeba systém přepsat (je-li již ve stoupající větvi vanové křivky). Hrubá metoda je sledování, zda frekvence selhání vybočuje z hranice intervalu spolehlivosti určeného z dat naměřených ze dna vanové křivky. V přednášce o matematické statistice je možné ukázat, oč přesnější je v tomto případě použití t-testu než použití hrubé grafické metody. Podobné postupy lze použít při sledování frekvence komunikace s různými adresami. Ty se výrazně liší pro ty co pracují a ty, co si hrají. Dají se tak indikovat počítačové závislosti lidí na kyberprostoru nebo detekovat spamy.

Aplikace poznatků z teorie systémů hromadné obsluhy Poznatky a výsledky teorie hromadné obsluhy lze použít ve správě sítí, jsou důležité při vývoji operačních systémů, při rozhodování zda a jak zvýšit kapacitu (např. přidáním zdrojů v komunikačních sítích či klonováním služeb v SOA). Nejjednodušší aplikace, při níž stačí středoškolské znalosti, je volba kapacity serveru jako hranice konfidenčního intervalu součtu náhodných veličin. Bez serveru musíme na každém klientu volit kapacitu $M + 3\sqrt{D}$, kde M je průměrné zatížení klienta (většinou velmi malé) a D je rozptyl zatížení klienta (obvykle velký). Server pro n klientů by měl mít (pokud jsou požadavky klientů nezávislé) kapacitu $nM_1 + 3 \cdot \sqrt{n \cdot D}$, kde $M_1 < M$, a nikoliv $nM + 3n \cdot \sqrt{D}$. Podobná úloha je řešena při používání statistických multiplexorů, kdy se musí rozhodovat, kolik telefonních hovorů může současně přenášet linka určité kapacity. Zde je dobré uvést, že při kapacitní rezervě menší než 1/3 dochází v systémech hromadné obsluhy k nepříjemnému zvětšování front a tedy i doby odezvy. Očekáváme-li proto větší variace zátěže

(především směrem k růstu), bývá výhodné volit alespoň 50%-ní rezervu. Na problém konfidenčního intervalu součtu nezávislých náhodných veličin vede i úloha řízení projektu. Je-li nestranný odhad M_i doby řešení i-té etapy projektu (zahrnujícího n lineárně propojených etap) a odhad doby řešení v "nejhorším případě" $M_i + 3S_i$ (hranice konfidenčního intervalu), pak je očekávaná doba řešení (za určitých opatření při vedení projektu – viz [11]) ΣM_i a s velkou pravděpodobností ne větší než $\Sigma M_i + 3\sqrt{n\Sigma S_i^2})$. Při tom nelze použít metodu kritické cesty. To je snad nejhezčí příklad jak formulace požadavků a cest k jejich splnění může zásadně záviset na kvalitě dat a dalších organizačních opatřeních. To by mělo být probíráno i v přednášce ze softwarového inženýrství či řízení projektů.

Základní poznatky z teorie organizace a marketingu se hodí jako podpůrné znalosti při specifikaci požadavků na informační systémy a jsou užitečné i při volbě obchodních strategií softwarových firem. Nejjednodušší aplikací je závislost mezi optimální velikostí zakázek a kapacitou firmy, případně pravděpodobností krachu firmy z nedostatku zakázek. Tento problém vede k analýze rozptylu náhodného součtu náhodných veličin a měl by proto být zmíněn v softwarovém inženýrství a v pokročilejší přednášce z matematické statistiky. Zde lze pravděpodobně využít výsledky teorie marketingu a managementu.

Výběrová šetření Základní poznatky teorie a praxe výběrových šetření lze použít pro průzkum spokojenosti uživatelů se široce používaným softwarovým systémem. Podobné postupy lze někdy užít i při identifikaci skupin uživatelů během specifikace požadavků.

Analýza rizik Při diskusi analýzy a řízení rizik se používá jednoduchý model založený na výpočtu očekávané ztráty jednotlivých rizik. To je dobrý příklad konceptuálního modelování v informatice; ve statistice lze tento problém zařadit do širšího rámce pojistné matematiky.

Vliv práce na počítači na oči Je obecným přesvědčením, že práce u počítače ničí oči. Bohužel se to nepodařilo jednoznačně prokázat (mj. jako pracovní úraz u pracovně právních sporů v USA). Důvodem může být značný rozptyl dat a existence takových faktorů, jako je koukání na televizi. Zde by se mohlo zmínit, jak zlepšit statistickou analýzu pomocí jemnějších metod, než je t-test.

Problémy regresní analýzy Má-li náhodná veličina Y rozložení $Y = aX + b + Z$, kde X je nezávislá proměnná a Z je náhodná veličina s rozptylem D, dostaneme jako odhad lineární regrese Y přímku $y = rx + d$, kde $r < a$ a r je tím menší, čím větší je rozptyl D. Poněvadž mají data softwarových metrik velký rozptyl, dostaneme běžnou analýzou regrese nesprávný výsledek, že jedna řádka velkých programů se snáze napíše, jedná-li se o velké programy. Podobný výsledek dostaneme, spojíme-li data z různých zdrojů, kdy data jednotlivého zdroje se dají popsat vztahem $Y_i = a_i \cdot X_i + Z$, kde Z má malý rozptyl.

Optimalizace datových replikací Většina manažerských použití dat včetně OLAP je založena na statistické analýze. Pokud je takové použití výhradní, není nutné replikovat všechna data, ale stačí jen jistá část dat. Jaká to má být

část závisí na základních statistických charakteristikách těchto dat. U velkých souborů dat mohou být úspory dramatické.

Sledování výkonu Mnohé systémy poskytují informace o době provádění jednotlivých operací. Na příkladech lze provést statistickou analýzu a provést porovnání jejích předpovědí se skutečností (a také s kvalitou předpovědí hrubým odhadem). Podobné nástroje lze použít i při testu, zda nějaká část systému má významně větší počet chyb nebo při optimalizaci vyhledávačů a databázových dotazů.

8 Pravděpodobnostní a statistická analýza algoritmů a datových struktur

Analýza efektivnosti algoritmů a datových struktur (AADS) vyžaduje poměrně rozvinutý aparát, a proto může být předmětem přednášek navazujících na kurs matematické statistiky nebo presentována v magisterském studiu. Na druhé straně nevyvolává AADS u informatiků takový odpor, jako postupy uvedené výše. Cenou je, že to prakticky neovlivňuje jejich postoje. Je to tím, že metoda a technika analýzy algoritmů má leccos společného s klasickou matematikou a není při tom třeba tápat v hromadách dat a hledat v nich zákonitosti. Informatik zde může mít stále pocit, že je to on, nikoliv příroda, kdo určuje pravidla hry, a že tedy může zůstat v milovaném kyberprostoru.

AADS má značné praktické dopady a je i vděčným polem výzkumu. Z hlediska využití matematické statistiky v běžné práci informatiků má ale jen marginální význam, protože většinou nevyžaduje analýzu experimentálních dat. Pro úplnost uveďme konkrétnější oblasti AADS, viz [12]. U analýzy časové a prostorové složitosti algoritmů je vhodné ukázat závislost formulace cílů řešení na statistických vlastnostech dat (např. při řešení problému obchodního cestujícího, při rozvrhování či při řízení projektů metodou kritického řetězce). To jsou případy, kdy volba cíle aplikace silně závisí na statistických vlastnostech dat.

Vděčným tématem jsou algoritmy používané v databázích, především při indexování nebo hašování. Zde je vhodné ukázat praktické důsledky analýzy nejhoršího a průměrného případu. Zajímavé jsou otevřené problémy týkající se algoritmů třídění.

Významné aplikace lze nalézt v kryptografii (náhodné generování klíčů, kódování pomocí náhodných řetězců, pravděpodobnostní analýzy používaných postupů), v počítačové grafice (filtrace šumu) a v analýze systémů majících charakter systémů hromadné obsluhy s aplikacemi na komunikační systémy, řídící systémy a výše zmíněné servisně orientované softwarové systémy. Existují i aplikace v dokumentografických systémech a také ve vyhledávačích jako je Google. Zde je důležité s využitím statistických metod stanovit v hy indexových termů a míry podobnosti mezi dotazem a dokumentem charakterizovaným určitými termy. Používají se poměrně pokročilé metody jako Bayesovská analýza a Bayesovské sítě a také shluková analýza. Podobné metody se používají při hodnocení úspěšnosti vyhledávání. Všimněme si, že v tomto případě musí informatik do jisté míry svůj kyberprostor do jisté míry opustit.

Roste význam statistiky v dolování dat (role kvality dat, vylučování okrajových hodnot, použití statisticky relevantních metod, správná interpretace výsledků, atd.). Tyto problémy jsou stále urgentnější, o čemž svědčí bonmot informatického folkloru tvrdící, že hlavní způsob použití datových skladů a data miningu je proces "Hnůj tam, hnůj ven". Podrobnější informace lze nalézt v [12]. Stimulující může být použití simulací, především pro ilustraci průběhu procesů v systémech hromadné obsluhy. Problémy kvality dat jsou předmětem standardizace a dokonce i federálního zákona USA. Zájem studentů o statistické postupy mohou zvýšit příklady překvapivého použití bílého šumu pro zlepšení zobrazení rastrového mikroskopu nebo použití bílého šumu v metodě zvané scrambling umožňující podstatné zlepšení charakteristik komunikačních kanálů (viz výše).

Instruktivní mohou být i jiná použití pravděpodobnosti v komunikacích. Na příkladě protokolu TCP/IP lze ilustrovat, že někdy je lépe implementovat systém, který s velmi malou pravděpodobností může selhat, než systém, u kterého k tomu teoreticky nemůže dojít. Takový systém však může v praxi vykazovat více chyb v důsledku toho, že vyžaduje složitější (a tudíž k chybám náchylnější) implementaci.

Pro určité algoritmy s ne zcela známým chováním je výhodné provést měření efektivnosti, a pak provést statistickou analýzu získaných dat. Vzorem mohou být publikace hodnotící efektivnost algoritmů třídění. To je dovednost, která vyžaduje jistou zběhlost v používání matematické statistiky a která je velmi často potřebná při psaní diplomové práce z informatiky. Lze použít vhodné statistické programy (GPSS, StatGraphics, Mathematica, atd.). Jejich používání by mělo být zmíněno v přednáškách.

8.1 Analýza dat pro uživatele i pro hodnocení softwaru

Analýza dat, tj. zjišťování neznámých zákonitostí z víceméně náhodných dat je jádrem aplikací matematické statistiky a hlavním nástrojem experimentálních věd. Je to oblast, kterou mají informatici, zejména ti z univerzit, zvláště neradi. Analýza dat má v informatice kromě výše uvedených příkladů dvě hlavní oblasti aplikace – data mining a řízení procesů vývoje softwaru s využitím softwarových metrik. Filtrace a výběr dat v data miningu je věcí jak informatiků (rozhodují nároky časové i prostorové), tak statistiků (otázka optimálního rozsahu výběru – ne vždy je výhodné využívat všechna dostupná data, problém okrajových dat i otázka vhodné formulace zadání statistické analýzy z hlediska efektivnosti řešení, věcná správnost použité metody, správná interpretace výsledků).

Zvláštní kapitolou je použití softwarových metrik (kvantitativních charakteristik softwarových produktů). Zde je třeba konstatovat, že kultura prezentace analýz, které jsou jádrem klíčových odhadů pracnosti a doby řešení softwarového projektu, je v informatické literatuře velmi nízká. Na tuto skutečnost by měli být studenti upozorněni. V informatice je bohužel špatným zvykem nezveřejňovat data (zvláště v tom vyniká společnost Standish Group), na jejichž základě se dospělo k různým experimentálním zákonitostem a také se nezabývat statistickou kvalitou dat, jako je dostatečný rozsah výběru, směrodatná odchylka, nezávislost, atd.

Jako příklad uveďme doporučení CMM [13] nebo novější CMMI [14], podle kterého se má postupně dosáhnout opakovatelnosti a dokonce zlepšení kvality (optimalizace) procesů vývoje softwaru. Poněvadž jsou však atributy procesů vývoje softwaru náhodné veličiny, je kvalita řešení CMM závislá na rozsahu výběru a kvalitě dat. To má vážné důsledky pro uzavírání smluv (odhady nákladů a doby řešení). O tom se v metodice CMM pro tzv. repeated level obsahující stovky doporučení prakticky nemluví.

9 Závěr

Absolventi s orientací na programování se po graduování a mnohdy i před ním velmi dobře uplatňují v praxi a vydělávají často mnohem více než jejich profesoři a mnohé jiné profese. Můžeme se proto ptát, proč na jejich studiu něco měnit, zvláště když jsou většinou spokojeni s tím, že nebudou nikdy analytici, natož manažeři.

Domníváme se, že je nebezpečné se smiřovat s tím, že naši absolventi budou mít omezené možnosti kariérního růstu, a že se dobrovolně omezí na kariéru spíše řemeslníka, byť velmi kvalifikovaného, než inženýra nebo manažera. Pro někoho to je pro jeho typ schopnosti samozřejmě optimální životní perspektiva, se kterou může být spokojen. Asi by to ale neměl být jediný cíl studia magistrů na universitě a asi ani bakalářů. Pravděpodobně totiž změní názor, až se budou muset pohybovat v nižších patrech hierarchie firmy. A hrozí jim i nezaměstnanost. Na trhu práce bude pro informatiky zaměřené na návrh a kódování brzy asi velmi těsno, protože takové lidi produkuje stále více škol. K tomu už dochází ve Spojených státech, kde už není profese informatik zárukou dobrého zaměstnání, nemluvě o Austrálii (viz výše).

K redukci potřeby programovat asi dojde i v důsledku outsourcingu a globalizace používání některých typů aplikací (ty, které mohou být na celém světě stejné), snazšího vývoje nových a používání existujících aplikací a produktů třetích stran a využívání předností servisní orientace.

Vzhledem k tomu, že dnešní studenti půjdou do penze po čtyřiceti a možná i padesáti odpracovaných letech, mělo by být naší povinností zlepšit jejich možnosti v soutěži o zaměstnání v oborech mimo informatiku. Mnohé z toho, o čem se domníváme, že je dnes přínosem pro informatiku, se dá použít velmi dobře i mimo informatiku (statistika, psychologie, management, výtvarné cítění a poznatky psychologie používané při návrhu domovských stránek). Mnohá rizika jsou aktuální v perspektivě osmi až deseti let, to jest celkem krátce po tom, co dostudují studenti, kteří byli letos u přijímacích zkoušek.

Je žádoucí v přednáškách upozornit na základní problémy kvality dat z hlediska statistiky (rozsah souboru, rozptyl, stálost statistických vlastností, nezávislost) a důsledky těchto vlastností pro běžné statistické úkoly, jako jsou odhady důležitých veličin, například doba řešení, spotřeba práce, střední doba mezi poruchami. Je důležité upozornit, že v softwaru nejsou dodržovány osvědčené postupy a dobré zvyky experimentálních věd, jako je zveřejňování směrodatných

odchylek, poskytování zdrojových dat k nezávislému prověření publikovaných výsledků a jiné údaje o kvalitě dat (aktuálnost, relevance, atp.).

V mnoha slavných softwarově inženýrských doporučeních jsou problémy kvality dat zmiňovány jen okrajově. Klasickými příklady jsou metodiky COCOMO [15] a do jisté míry i Function Points [16]. Platí to i pro metodiky CMM/CMMI, které doporučují stovky opatření na zlepšení softwarových procesů, nezmiňují se ale dostatečně o potřebě kvalitních dat. Tento problém je opomíjen i v nejnovějších doporučeních znalostních profilů informatických profesionálů, jako jsou[17]-[21]. Matematická statistika se spolu s komunikačními schopnostmi a schopností a ochotou spolupracovat s uživateli stává problematickým místem profesních znalostí informatiků, především tehdy, vyvíjejí-li servisně orientované systémy.

U mnoha studentů informatiky je vážnou překážkou profesního uplatnění neochota či neschopnost spolupracovat s uživateli. To může být oslabeno včasnou integrací metod matematické statistiky do informatických přednášek. Přeceňují se čistě informatické a některé matematické znalosti a nedoceňuje se potřeba znalostí a dovedností důležitých pro spolupráci s uživateli a pro řízení prací při vývoji softwaru. Změna nebude snadná, vzhledem k zavedené pedagogické praxi a struktuře studia. Nepromyšlené a metodicky neověřené změny mohou zhoršit to, co umíme, bez jakýchkoliv přínosů. Vážnou překážkou je setrvačnost plynoucí ze zavedené struktury studia, která má tendenci zesilovat svoje základní rysy, tedy orientaci na návrh a programování.

Reference

1. Matloff, N.: Globalization and the american worker. Communications of the ACM **47** (2004) 27–29
2. Boehm, B.W.: Software Engineering Economics. Prentice Hall (1981)
3. Král, J., Demner, J.: Towards reliable real time software. In: Proceedings of IFIP Conference Construction of Quality Software, North Holland (1979) 1–12
4. Král, J.: Orientace na služby - klíčové paradigma současného softwaru. In: DATAKON 04. (2004) Zvaná přednáška.
5. Král, J., Žemlička, M.: Architecture and modeling of service-oriented systems (2005) To appear.
6. Král, J., Žemlička, M.: What literacy for software developers. In Carbonara, D., ed.: Technology Literacy Uses in Learning Environments, Heshey, PA, USA, Idea Group Publishing (2004) To appear.
7. Beck, K.: Extreme Programming Explained: Embrace Change. Addison Wesley, Boston (1999)
8. Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R.C., Mellor, S., Schwaber, K., Sutherland, J., Thomas, D.: Agile programming manifesto (2001) http://www.agilemanifesto.org/.
9. Sneed, H. Position statement at panel discussion at CSMR 2002, Budapest, Hungary (2002)
10. Král, J., Töpfer, P.: Education of software experts for a changing world. In Pudlowski, Z., ed.: 2nd Global Congress an Engineering Education, UICEE (2000) 267–271

11. Goldratt, E.M.: Critical Chain. North River Press, Great Barrington, MA (1997)
12. Koubková, A., Král, J.: Pravděpodobnost a matematická statistika v informatických oborech (2004) Referát na konferenci Robust 2004, Třešť, červen 2004.
13. SEI Institute: Capability Maturity Model, home page (2005) A methodology of the SEI institute, http://www.sei.cmu.edu/cmm/cmm.html, as visited 10th January 2005.
14. Chrissis, M.B., Konrad, M., Shrum, A.: CMMI: Guidelines for Process Integration and Product Improvement. Addison Wesley, Boston (2003)
15. COCOMO: COCOMO II (1995) http://sunset.usc.edu/research/COCOMOII/.
16. Treble, S., Douglas, N.: Sizing and Estimating Software in Practice. McGraw-Hill, London (1995)
17. EUCIP: EUCIP business analyst (2004)
18. EUCIP: EUCIP information systems analyst (2004)
19. EUCIP: EUCIP software developer (2004)
20. Gorgone, J.T., Davis, G.B., Valacich, J.S., Topi, H., Feinstein, D.L., Herbert E. Longenecker, J.: IS 2002: Model curriculum and guidelines for undergarduate degree programs and information systems (2002)
21. ACM: Computing curricula 2004 (2004) draft.

Závislostní redukční analýza přirozených jazyků*

Markéta Lopatková[1], Martin Plátek[2], and Vladislav Kuboň[3]

[1] CKL MFF UK, Praha
lopatkova@ckl.mff.cuni.cz
[2] KTIML MFF UK, Praha
platek@ksi.ms.mff.cuni.cz
[3] ÚFAL MFF UK, Praha
vk@ufal.mff.cuni.cz

Abstrakt V tomto článku vyložíme podstatu závislostní redukční analýzy (DAR, dependency analysis by reduction) a její souvislost s pojmy závislost a závislostní strom. Výklad budeme ilustrovat příklady z češtiny, což je jazyk s (výrazně) volným slovosledem. Tento výklad shrnuje základní rysy vývojových postupů závislostní syntaxe. Bude využit jako podklad pro ověřování (a vysvětlování) adekvátnosti formálních a počítačových modelů těchto postupů.

1 Slovo úvodem k redukční analýze

Větná stavba angličtiny a dalších jazyků s pevným slovosledem se formálně popisuje frázovými gramatikami. Popisy větné stavby latiny, italštiny, němčiny, slovanských jazyků, arabštiny a dalších jazyků bývají častěji a adekvátněji založeny na postupech, které se souhrnně nazývají závislostními. Oba principy lze vykládat pomocí pozorování postupného zjednodušování jednotlivých vět jazyka, pomocí tzv. redukční analýzy. Principy frázové redukční analýzy a závislostní redukční analýzy se však podstatně liší. Frázovou redukční analýzu (jazyků s pevným slovosledem) lze přímo modelovat analýzou zdola pomocí frázových (Chomského) gramatik. Na základě tohoto textu by čtenář měl nahlédnout, že závislostní redukční analýzu jazyků s volným slovosledem je nutné modelovat podstatně jinak.

Abychom přiblížili význam předkládaného textu, naznačíme, v čem se syntaktická analýza (větný rozbor) vět přirozeného jazyka liší od analýzy vět (textů) umělého (např. programovacího) jazyka. U umělých jazyků určujeme stavbu vstupní věty, přičemž obvykle máme k dispozici formálně zapsanou bezkontextovou gramatiku příslušného jazyka (obvykle vzniká gramatika zároveň s jazykem), navíc běžně požadujeme gramatiku takovou, která zaručuje jednoznačnost rozboru každé věty jazyka.[4]

* Tento článek je výsledkem výzkumu, který probíhá za podpory grantu MŠMT LN00A063 a grantu GA ČR No. 201/02/1456

[4] Ve formálních jazycích se obvykle mluví o ‚symbolech abecedy' a ‚slovech', my budeme nadále používat u formálních i přirozených jazyků termínů ‚slovo (ze slovníku)' a ‚věta'.

U přirozených jazyků je situace odlišná – při větném rozboru také hledáme stavbu vstupní věty. Ta však nebývá určena jednoznačně. Místo ucelené formální gramatiky máme k disposici dovednost větného rozboru – učíme se jej ve školách, je popisován (často pomocí implicitních pravidel) v mluvnicích daného jazyka.

Mluvnice češtiny předpokládají, že člověk rozumí smyslu rozebírané věty dříve, než začne větný rozbor provádět (citujme zde Učebnici větného rozboru [10]: Správný rozbor větný není možný bez přesného porozumění větě, ... "). Automatická syntaktická analýza (podle formální gramatiky) porozumění větě nepředpokládá, ani je nemá k disposici. Je naopak jednou z prvních fází při počítačovém modelování smyslu věty.

Jaký je vztah větného rozboru k redukční analýze? Jednoduše řečeno, větný rozbor češtiny je založen na elementárnější schopnosti provádět redukční analýzu, tj. postupně zjednodušovat zkoumané věty. Cestu od větného rozboru k počítačové syntaktické analýze vybraného jazyka lze rozdělit na následující kroky:

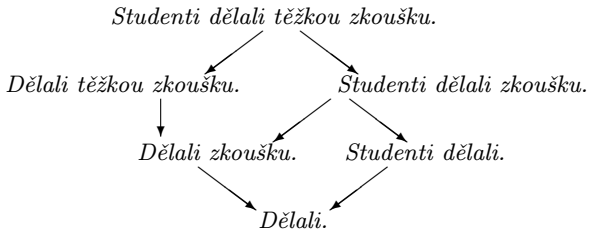
(a) vyhledání a popis pravidel redukční analýzy jazyka,
(b) transformace pravidel redukční analýzy do pravidel a omezení závislostního formalismu,
(c) sestrojení (vybrání) vhodného analyzátoru pro daný soubor závislostních pravidel a omezení.

V tomto článku se budeme věnovat pozorováním patřícím bodu (a) a částečně bodu (b), se zaměřením na závislostní analýzu češtiny.

Redukční analýzou se zde zabýváme především proto, abychom získali jasnou představu o tom, jak ji formálně a počítačově modelovat. K tomuto účelu jsou zavedeny a studovány restartovací automaty (viz např. [3]).

Následující příklad ilustruje zjednodušeně metodiku závislostní redukční analýzy.

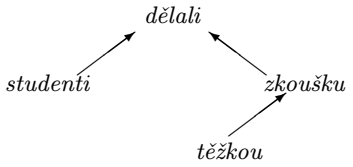
Příklad 1. Větu *‚Studenti dělali těžkou zkoušku.‘* lze při zachování syntaktické správnosti zjednodušit dvěma způsoby (viz též schéma na obrázku 1) – vypuštěním slova *studenti*, nebo vypuštěním slova *těžkou* (ale už ne vypuštěním slova *zkoušku* – věta *‚*Studenti dělali těžkou.‘* není správně utvořená). Ve druhém kroku můžeme vypustit slovo *těžkou* (v první větvi analýzy), nebo slovo *studenti*, či slovo *zkoušku* (ve druhé větvi). V posledním kroku lze vypustit slovo *zkoušku* (v první větvi), nebo slovo *studenti*.



Obr. 1. Schéma DAR pro větu *‚Studenti dělali těžkou zkoušku.‘*

Schéma DAR úzce souvisí se závislostním stromem (na obrázku 2 je závislostní strom pro větu *‚Studenti dělali těžkou zkoušku.‘*).

(i) Skutečnost, že nějaké slovo lze z věty vypustit tak, že získáme větu jednodušší, znamená, že závisí na některém slově ze zkrácené věty (rozvíjí ho).
(ii) Dvě slova lze postupně vypustit v libovolném pořadí, právě když jsou vzájemně nezávislá.
(iii) Navíc se ukazuje, že některá slovní spojení (např. spojení předložky s podstatným jménem; viz např. příklad 4 dále) je nutno vypouštět v jednom kroku – i v tomto případě je někdy vhodné určovat závislosti.



Obr. 2. Závislostní strom věty *‚Studenti dělali těžkou zkoušku.‘*

Předchozí příklad ilustruje postup, jak pomocí DAR získat ve větě informaci o závislostech (vztazích mezi rozvíjenými a rozvíjejícími členy věty). Můžeme si všimnout, že vezmeme-li věty *‚Těžkou zkoušku studenti dělali.‘* či *‚Těžkou dělali studenti zkoušku.‘*, nebo jejich další permutace, získáme zcela analogická schémata redukce jako pro větu původní, tedy vypuštěná slova (slovní spojení) v jednotlivých redukčních krocích budou stejná. Závislostní redukční analýza nám tímto způsobem umožňuje pozorovat, do jaké míry jsou závislosti invariantní vůči slovosledu.

Následující kapitoly se podrobněji zabývají závislostní redukční analýzou a určováním závislostí na základě redukční analýzy.

2 Závislostní redukční analýza

Závislostní redukční analýza (DAR) spočívá v postupném zjednodušování věty – každý krok DAR je reprezentován právě jednou **operací redukce**, která může být realizována dvěma způsoby:

(i) vypuštěním alespoň jednoho slova vstupní věty, nebo
(ii) nahrazením (obecně nesouvislého) podřetězce věty kratším podřetězcem.

Možnost aplikovat určitou redukci je podmíněna zachováním některých (alespoň prvního) z následujících **principů DAR**:

(a) zachování správnosti stavby věty (syntaktická správnost);
(b) zachování lemmatu (slovníkového hesla) a vybrané morfologické značky (soubor morfologických kategorií, které charakterizují daný výskyt slova);

(c) zachování významu původních slov ve větě (reprezentován např. valenčním rámcem,[5] či vhodným ekvivalentem v jiném jazyce);
(d) zachování významové samostatnosti věty (významově samostatná věta je taková věta, která vyslovena samostatně nevyvolává nutně další otázky).[6]

V závislosti na konkrétním úkolu (např. kontrola gramatické správnosti) lze tyto požadavky na DAR uvolňovat; principy, které nejsou uvolněny, potom označujeme jako **platné principy DAR** (např. v příkladu 1 byl uvolněn požadavek zachování významové samostatnosti věty – původní věta byla redukována až na jednočlennou větu *‚Dělali.‘*, což už není samostatná věta).

Pokud lze v určitém kroku DAR aplikovat určitou redukci při zachování platných principů, mluvíme o **přípustné redukci**. Pomocí všech přípustných redukcí se lze dostat ke všem **přípustným zjednodušením** zpracovávané věty. Původní, nezjednodušenou větu budeme (z technických důvodů) považovat též za přípustné zjednodušení věty.

Schématem DAR (redukčním schématem) věty jazyka nazveme orientovaný graf, jehož uzly reprezentují všechna přípustná zjednodušení dané věty (včetně původní věty) a jehož hrany odpovídají všem přípustným redukcím, které lze aplikovat vždy na počáteční uzel hrany a jejichž výsledkem je přípustné zjednodušení věty v koncovém uzlu hrany.

Příklad 2. Redukční schéma věty *‚Studenti dělali těžkou zkoušku.‘* z příkladu 1 ilustruje redukce typu (i) – v každém kroku DAR je vypuštěno jedno slovo vstupní věty, přičemž možnost větvení zachycuje nedeterministickou povahu DAR. Redukce typu (ii) je ilustrována na možném zjednodušování věty *‚Kursem prošlo patnáct studentů.‘*, jejíž redukční schéma je na obrázku 3. První krok první větve schématu se realizuje vypuštěním slova *kursem* (podle bodu (i), opět je uvolněn požadavek (d) na zachování významové úplnosti věty), v prvním kroku druhé větve je řetězec *prošlo patnáct studentů* nahrazen kratším řetězcem *prošli studenti* (podle bodu (ii)). Dále redukce pokračuje obdobně.

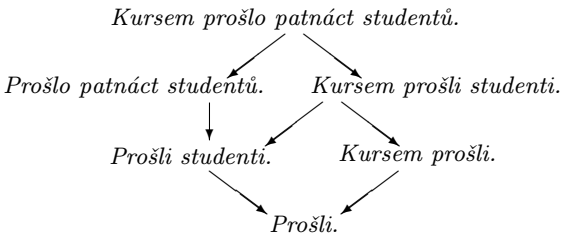
3 Redukční struktura a závislostní strom

Schéma DAR umožňuje zavést a klasifikovat různé typy vztahů. Na základě takto zavedených vztahů definujeme redukční strukturu věty. Podíváme se na její vztah k závislostnímu stromu.

Mějme jazyk L, větu $v \in L$, $v = v_1 v_2 \dots v_m$, kde $v_1, v_2, \dots, v_m$ jsou slova, a schéma DAR věty v. Řekneme, že slova v_i, $i \in N, N \subseteq \{1, 2, \dots m\}$ tvoří **redukční komponentu**, pokud jsou všechna v_i vypouštěna vždy najednou (tj. v redukčním schématu jsou všechna v_i vypouštěna vždy v jednom kroku,

[5] Valenční rámec, který popisuje syntakticko-sémantické vlastnosti slova, odpovídá jednomu jeho významu, více viz např. [6].

[6] V lingvistické terminologii to znamená, že významově samostatná věta se skládá ze slovesa, všech jeho sémanticky ‚povinných‘ doplnění a (rekurzivně) jejich ‚povinných‘ doplnění, přičemž ‚povinná‘ doplnění jsou ta doplnění, která musí mluvčí i posluchač znát pro dobré porozumění větě, více viz [7].

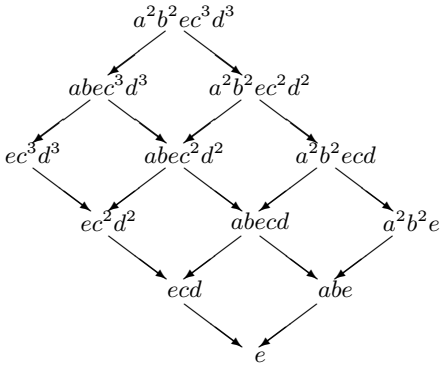


Obr. 3. Schéma DAR pro větu ‚*Kursem prošlo patnáct studentů.*‘

kterému odpovídá jedna hrana schématu). Řekneme, že slovo v_i je **(redukčně) závislé** (příp. **v redukci závislé**) na slově v_j, pokud je ve všech větvích DAR slovo v_i vypuštěno dříve než v_j; slovo v_j budeme nazývat slovem **(v redukci) řídícím**.
Řekneme, že slova v_i a v_j jsou **redukčně nezávislá**, pokud jsou vypouštěna v libovolném pořadí (tj. existuje větev DAR, ve které je slovo v_i vypuštěno před slovem v_j, a existuje větev DAR, ve které je slovo v_j vypuštěno před slovem v_i).

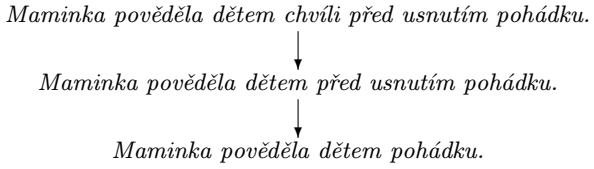
Příklad 3. Mějme formální jazyk $L = \{a^i b^i e c^n d^n | i, n \geq 0\}$. Na obrázku 4 je schéma redukční analýzy věty $a^2b^2ec^3d^3$. Přijměme pro jednoduchost výkladu předpoklad, že vypouštět v jednom kroku se smějí pouze dvojice nejbližších slov, pokud je více možností. Potom v této větě a větách redukovaných tvoří sousední slova a a b redukční komponentu, podobně sousední slova c a d tvoří také redukční komponentu; slova a, b, c a d jsou redukčně závislá na slově e; slova a a b jsou redukčně nezávislá na slovech c a d. Jazyk L je příklad jazyka s pevným slovosledem. Jazyk s volným slovosledem a ‚stejnými‘ redukcemi a (ne)závislostmi získáme tak, že budeme uvažovat množinu všech permutací vět z L.

Na základě zavedených vztahů redukční závislosti a redukční komponenty definujeme redukční strukturu věty – ilustrujeme ji na následujícím příkladu.



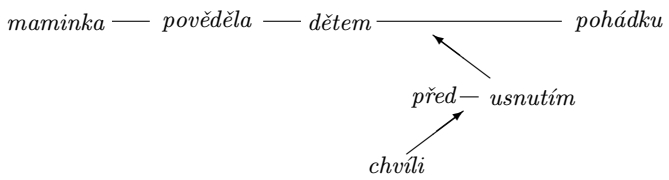
Obr. 4. Schéma DAR pro větu $a^2b^2ec^3d^3$.

Příklad 4. Redukční schéma věty *‚Maminka pověděla dětem chvíli před usnutím pohádku.‘*, které zachovává všechny principy DAR z předchozí sekce (tj. včetně principu (d) zachování významové samostatnosti věty), je na obrázku 5 – předložková skupina *před usnutím* musí být redukována v jednom kroku; sloveso *povědět* má tři ‚povinná‘ doplnění, která odpovídají podmětu a tzv. přímému a nepřímému předmětu, podstatné jméno *maminka* nemá povinná doplnění, proto má významově samostatná věta tvar *‚Maminka pověděla dětem pohádku.‘*.



Obr. 5. Schéma DAR pro větu *‚Maminka pověděla dětem chvíli před usnutím pohádku.‘* při uplatnění principu zachování významové samostatnosti věty.

Redukční strukturu budeme zachycovat diagramem, ve kterém uzly reprezentují jednotlivá slova věty, horizontální hrany spojují slova tvořící redukční komponentu (hranou jsou spojeny vždy dvě sousední slova redukční komponenty).[7] Šikmé hrany odpovídají redukčním závislostem; považujeme je za orientované od závislého slova (resp. od celé redukční komponenty) ke slovu řídícímu (resp. opět k celé redukční komponentě, pokud je pro dané slovo / komponentu řídící). Lineární uspořádání uzlů (zleva doprava) zachycuje slovosled (pořadí slov ve větě). Na obrázku 6 je redukční struktura reprezentující větu *‚Maminka pověděla dětem chvíli před usnutím pohádku.‘*



Obr. 6. Redukční struktura pro větu *‚Maminka pověděla dětem chvíli před usnutím pohádku.‘*

Stavba (české) věty se tradičně zachycuje závislostním stromem. To je průhledné a korektní pro věty, které nejsou komplikovány koordinacemi, elipsami a některými okrajovými jevy. **Závislostní strom** je struktura, která je konečným stromem ve smyslu teorie grafů, má kořen, do kterého směřují všechny cesty

[7] Za sousední slova zde považujeme vždy dvojici slov z jedné redukční komponenty, mezi kterými neleží slovo ze stejné komponenty (taková slova se nemusí vyskytovat na sousedních slovosledných pozicích ve větě, mezi nimi může být slovo, které do této komponenty nepatří).

a jeho uzly jsou úplně (lineárně zleva doprava) uspořádány. Uzly (jistým, ne vždy stejným způsobem) reprezentují výskyty slov ve větě, hrany reprezentují vztah mezi rozvíjeným a rozvíjejícím slovem (jednotkou) ve větě.

Zbývá popsat, jak přecházet od redukční struktury k závislostnímu stromu. Redukční závislosti nepředstavují problém, příslušné hrany charakterizují vztah mezi slovem rozvíjejícím a slovem, které je rozvíjeno, pořadí slov ve větě je zachováno. U redukční komponenty musíme určit, které slovo z dané komponenty bude považováno za slovo řídící a které slovo / která slova za závislá. K tomu je potřeba uvést další pravidla pro jednotlivé lingvistické jevy, které přiblížíme v následující sekci.

4 Redukční vztahy v přirozeném jazyce

Formální typologie závislostí zavedená v předchozí sekci odpovídá tradiční lingvistické klasifikaci – zde se pokusíme tuto souvislost detailněji popsat.

4.1 Krátce k lingvistickým pojmům

Přibližme ve stručnosti pojmy (české) lingvistiky, které budeme dále používat. Více lze nalézt například v [10], [9], [7] a [5].

Stavba věty zachycuje vztahy mezi jednotlivými větnými členy, a to zejména (i) vztah **podřízenosti** (anglicky **subordination**),[8] tj. vztah rozvíjený větný člen – větný člen jej rozvíjející, a (ii) **slovoslednou pozici** ve větě (viz [10]). Jako větné členy přitom označujeme výrazy (jednotlivá slova i skupiny slov), které mají jedinou syntaktickou funkci (tj. vyjadřují např. podmět, predikát, předmět, přívlastek či příslovečné určení).

Kritérium pro rozlišení větných členů ve dvojici **člen rozvíjený** – **člen** jej **rozvíjející** (anglicky **modified** – **modifying member**) je založeno na pojmu tzv. endocentrické konstrukce,[9] viz [9]: pokud lze jedno ze dvou slov vypustit, aniž se změní distribuční vlastnosti celého páru (tj. aniž se změní schopnost vyskytovat se ve stejném syntaktickém okolí), je toto slovo považováno za rozvíjející. Na základě **principu analogie** (na úrovni slovních druhů) se určí směr rozvíjení i pro tzv. exocentrické konstrukce[10] – ilustrujme takový postup v případě slovesa a jeho aktoru (což je větný člen v činné větě typicky odpovídající

[8] Termínem ‚podřízenost' v tomto článku označujeme jazykový vztah, zatímco termín ‚závislost' vyhrazujeme pro formální struktury, pomocí nichž modelujeme jazykové vztahy, tedy např. podřízenost. Obdobně např. Kunze v [4] rozlišuje ‚Unterordnung' pro jazykové vztahy a ‚Abhängigkeit' pro jejich formální zachycení.

[9] Endocentrická konstrukce je konstrukce sestávající alespoň ze dvou slov, z nichž jediné je ‚povinné' a tvoří ‚hlavu' konstrukce, ostatní slova jsou ‚nepovinná' a rozvíjejí tuto ‚hlavu', např. spojení *malý stůl* je endocentrická konstrukce, podstatné jméno *stůl* tvoří ‚hlavu', přídavné jméno *malý* je rozvíjí.

[10] Exocentrická konstrukce je konstrukce, která nemá ‚hlavu', která by mohla (syntakticky) zastupovat celou konstrukci (jde tedy např. o předložkové skupiny); z argumentace v [9] lze vyvodit, že za exocentrické konstrukce lze považovat i např. konstrukce sestávající ze slovesa a jeho ‚povinných' doplnění.

subjektu): protože existují slovesa, která nevyžadují aktor (např. *prší*), je vhodné považovat aktor vždy za větný člen rozvíjející sloveso. Obdobně je i ‚povinný' objekt (např. u slovesa *potkat*) považován za větný člen rozvíjející sloveso (neboť existují slovesa, která povinný objekt nemají, např. sloveso *zemřít*).

Jako **doplnění slovesa**, resp. **podstatného jména, přídavného jména** či **příslovce** budeme označovat větné členy, které rozvíjejí dané sloveso, resp. podstatné jméno, přídavné jméno či příslovce. Přitom rozlišujeme vnitřní a volná doplnění. **Vnitřní doplnění** (v anglické literatuře obvykle **inner participants** či **arguments**) odpovídají podmětu a předmětu/ům daného slovesa (např. *studenti dělali zkoušku, kursem prošli studenti, maminka pověděla dětem pohádku, král zemřel, Petr četl o neštěstí, rodiče čekali na děti*), resp. jistým typům přívlastků u podstatného jména (např. *začátek přednášky*) a jistým rozvitím u přídavných jmen a příslovcí (např. *závislý na počasí, kolmo na základnu*). **Volná doplnění** (anglicky obvykle **adjuncts**) odpovídají příslovečným určením (času, místa, způsobu, ..., např. *jde včas, jde domů, jde pomalu*), resp. jistým typů přívlastku (např. *malý stůl*), či jistým rozvitím přídavných jmen a příslovcí (např. *trochu krátký, velmi rychle, zcela jistě*).

Všechna doplnění dělíme na **obligatorní**, tj. povinně přítomná ve významové reprezentaci věty (která však nemusí být vyjádřena v konkrétní (povrchové) realizaci věty, posluchači / čtenáři můžou být známa z předchozího kontextu), a na doplnění ‚nepovinná', **fakultativní**. Pro rozlišení obligatorních a fakultativních doplnění slouží **dialogový test** popsaný v [7].

4.2 Modelování jazykových jevů pomocí redukční struktury

Vraťme se zpět k formální typologii redukčních vztahů a podívejme se, jakým způsobem souvisejí s různými typy vztahů jazykových.

Redukční závislosti dovolují přímo modelovat fakultativní volná doplnění – jde vesměs o endocentrické konstrukce, u nichž lze celou dvojici nahradit rozvíjeným slovem, ‚hlavou' konstrukce (a to beze ztráty významové samostatnosti, princip (d) DAR). Takto jsou tedy zachyceny vztahy typu *malý stůl, jde pomalu, jde domů, jde včas, trochu krátký, velmi rychle, zcela jistě*.

Slovo (v redukci) řídící odpovídá rozvíjenému slovu ve větě (tj. ‚hlavě' endocentrické konstrukce), slovo v redukci závislé odpovídá slovu, které toto řídící slovo rozvíjí (viz obrázek 7).

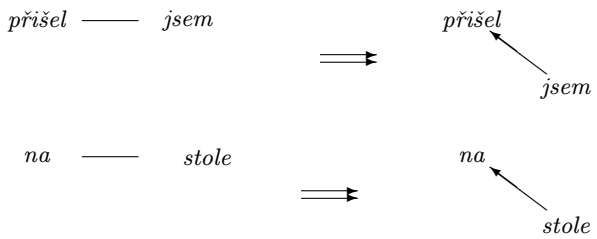
Obr. 7. Redukční závislosti modelují volná doplnění.

Zbývá určit řídící a závislý člen zpracovávané redukční závislosti v případech, kdy je rozvíjený nebo rozvíjející člen této závislosti tvořen celou redukční komponentou.

(i) Pokud je rozvíjející člen tvořen redukční komponentou, potom jako závislý člen určujeme řídící slovo této komponenty (určení řídícího slova u redukční komponenty viz níže, zbylé členy komponenty budou tvořit podstrom s kořenem v příslušném řídícím slově).
(ii) Pokud je rozvíjený člen tvořen redukční komponentou, potom v obecném případě dochází k (významovým) nejednoznačnostem, které jsou pro češtinu dosti typické (zajímavé příklady jsou uvedeny v [2]).

Redukční komponenty dovolují modelovat složitější vztahy mezi výskyty slov. Jsou to jednak
(a) vztahy morfologicko-syntaktické, jednak
(b) vztahy syntakticko-sémantické.

ad (a) Redukční komponenty popisují tzv. **formémy**, což jsou jednotky odpovídající jednomu větnému členu – jsou to zejména předložkové skupiny (*před usnutím, na stole, vzhledem k Pavlovi*) nebo složené slovesné tvary (*přišel jsem, bude obědvat, je vytištěn, tiskne se*). (Jde tedy o exocentrické konstrukce.)



Obr. 8. Možný převod formémů na závislostní podstrom (podle úzu PDT).

Formémy se v tradiční lingvistice znázorňují jako jeden uzel diagramu, resp. závislostního stromu znázorňujícího větnou stavbu, viz např. [10], resp. [9].[11] Pro mnohé prakticky orientované úkoly (např. kontrola gramatické správnosti, budování syntakticky anotovaného korpusu) je ovšem vhodné reprezentovat každé slovo věty vlastním uzlem (tedy nejen slova plnovýznamová); aby byl zachován zavedený datový typ závislostního stromu, je potřeba určit dodatečná pravidla, na jejichž základě se i redukční komponenty převedou na podstromy, tj. je nutné určit, které slovo formému se bude určovat jako řídící a které/á jako druhotně závislé/á (vybrané řídící slovo budeme pro jednoduchost označovat jako ‚hlava‘,

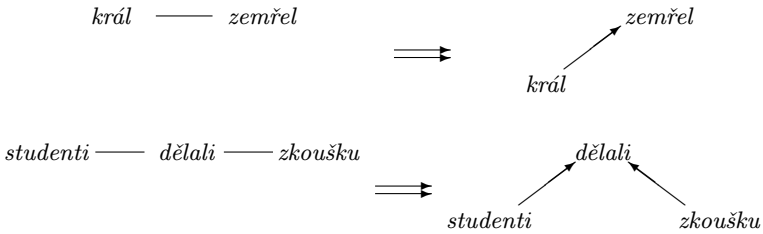
[11] V těchto pojetích jsou reprezentována samostatným uzlem pouze slova plnovýznamová (zejména významová slovesa, podstatná jména, přídavná jména a příslovce).

stejně jako u endocentrických konstrukcí). Taková pravidla jsou obvykle technického charakteru a mohou se v jednotlivých projektech lišit (na obrázku 8 je řešení přijaté v Pražském závislostním korpusu, PDT).

ad (b) Druhým typem vztahů, které jsou modelovány redukčními komponentami, jsou syntakticko-sémantické vztahy. Jsou to zejména **valenční vztahy** – vztahy slovesa, resp. podstatného jména, přídavného jména či příslovce a jeho povinných valenčních doplnění. Jde tedy o konstrukce typu *studenti dělali zkoušku, kursem prošli studenti, maminka pověděla dětem pohádku, král zemřel, Petr četl (o neštěstí), rodiče čekali (na děti), (na přednášku) přišli studenti, začátek přednášky, závislý (na počasí), kolmo (na základnu)*[12]. Tyto konstrukce (bývají charakterizovány jako exocentrické konstrukce) nelze beze ztráty významové samostatnosti, princip DAR (d), nahradit jedním slovem, ‚hlavou' konstrukce.

Valenční vztahy tradiční lingvistika zachycuje pomocí závislostního stromu (viz [9], kde se vychází z [10]). Teoretickým kritériem pro určení rozvíjeného a rozvíjejícího větného členu se stal princip analogie, který zde byl krátce popsán v předchozích odstavcích. Na základě tohoto kritéria stanovujeme také pravidla pro určení řídícího slova při převodu redukční struktury na závislostní strom: sloveso určujeme jako řídící větný člen, slovesná doplnění jako jeho slova závislá; obdobně pro podstatná jména, přídavná jména a příslovce a jejich doplnění.

Poznamenejme, že princip analogie zde lze jednoduše nahradit uvolněním požadavku (d) na zachování významové samostatnosti během DAR.



Obr. 9. Převod valenčních vztahů na závislostní podstrom.

5 Závěrečné poznámky

DAR dovoluje formulovat vztah základních syntaktických jevů: závislosti a slovosledu. To je nepostradatelné zejména pro modelování skladby jazyků s bohatou flexí a volným slovosledem, kde závislost a slovosled souvisejí velmi volně a u jednotlivých národních jazyků dosti odlišně (srovnejme s angličtinou, kde závislosti určuje (hlavně) velmi striktní slovosled).

Ukázali jsme, že závislosti odvozujeme ze dvou různých, nepřekrývajících se, jednoduše pozorovatelných, jazykově nezávislých jevů: z redukčních závislostí

[12] Závorkou zde pracovně vyznačujeme formémy, viz bod (a).

a redukčních komponent. Poukázali jsme na to, že lingvistická taxonomie jazykových jevů (česká) tomuto rozkladu (rozdělení) odpovídá. Přiblížili jsme tak lingvistický a informatický (množinový) pohled na danou problematiku.

Reference

1. Hajič J.: Building a Syntactically Annotated Corpus: The Prague Dependency Treebank. In: Issues of Valency and Meaning. Studies in Honour of Jarmila Panevová E. Hajičová (ed.), Karolinum, CU Press, Prague, 1998, 106–132.
2. Holan T., Kuboň V., Oliva K., Plátek M.: On Complexity of Word Order. In: Les grammaires de dépendance - Traitement automatique des langues (TAL), **41**, No. 1 S. Kahane (q.ed.), 2000, 273–300.
3. Jančar P., Mráz F., Plátek M., Vogel J.: On Monotonic Automata with a Restart Operation. Journal of Automata, Languages and Combinatorics, **4**, 4, 1999, 287–311.
4. Kunze J.: Abhängigkeitsgrammatik. Volume XII of Studia Grammatica, Akademie Verlag, Berlin, 1975.
5. Lopatková M.: Valency in the Prague Dependency Treebank: Building the Valency Lexicon. In: PBML 79-80, 2003, 37–59.
6. Lopatková M., Žabokrtský Z., Skwarska K., Benešová V.: Tektogramaticky anotovaný valenční slovník českých sloves. UFAL/CKL, TR-2002-15, 2001.
7. Panevová J.: Formy a funkce ve stavbě české věty. Academia, Praha, 1980.
8. Plátek M., Lopatková M., Oliva K.: Restarting Automata: Motivations and Applications. In: Proceedings of the Workshop „Petrinetze“ Holzer M. (ed.), Technische Universität Mnchen, 2003, 90–96.
9. Sgall P., Hajičová E., Panevová J.: 1986. The Meaning of the Sentence in Its Semantic and Pragmatic Aspects J. Mey (ed.), Prague, Academia, 1986.
10. Šmilauer V.: Učebnice větného rozboru. Skripta FF UK, SPN, Praha, 1958.

Algoritmy pro copánkové grupy a jejich užití v kryptografii

Eliška Ochodková[1], Václav Snášel[1], and Jan Šútora[2]

[1] Katedra infromatiky FEI, VŠB - TU Ostrava
tř. 17. listopadu 15, 708 33 Ostrava - Poruba
{Vaclav.Snasel, Eliska.Ochodkova}@vsb.cz
[2] Katedra informatiky PřF, UP Olomouc
Tomkova 40, 779 00 Olomouc
jan.sutora@upol.cz

Abstrakt Copánková grupa, je neabelovská grupa nekonečného řádu, přirozeně vycházející z geometrického copánku. Ukážeme si její základní vlastnosti a použití v kryptografii. Dále si ukážeme, jak implementovat operace s copánky pro počítačové řešení úloh.

Klíčová slova: Copánková grupa, konjugační problém, pozitivní copánek

1 Úvod

Asymetrický kryptosystém založený na copánkové grupě byl představen na konferenci CRYPTO 2000, viz [9]. Tento kryptosystém používá jednocestnou funkci se zadními vrátky založenou na NP problému nazvaném Obecný konjugační vyhledávací problém (Generalized Conjugacy Search Problem).

Další existující asymetrické kryptosystémy jsou založeny na jiných číselně teoretických problémech, viz tabulka 1.

Výpočetní problém	Asymetrický kryptosystém
Prvočíselná faktorizace čísla $n = pq$ (nalézt velká prvočísla p, q)	RSA (1978), Rabin, ...
Problém diskrétního logaritmu (nalézt x tak, aby $g^x = y$, kde g a y jsou prvky grupy G)	ElGamal (1985), ECC, ...
Problém batohu (nalézt prvky superrostoucí posloupnosti tak, aby jejich součet byl roven kladnému číslu k)	Knapsack (1978), ...
Aplikace třídy opravných kódů tzv. Goppa kódy	McEliece (1978)
Slovní problém pro copánkovou grupu	Copánkové kryptosystémy (1985)

Tab. 1. Asymetrické kryptosystémy a korespondující výpočetní problémy.

Vzhledem k existenci a pozdějšímu praktickému používání kvantových počítačů, a z toho plynoucí nezabezpečenosti asymetrických kryptosystémů (např. RSA), se pro sestavení asymetrických kryptosystémů jeví být výhodnými copánkové grupy nebo zobecnění EL-Gamal šifrování (diskrétní logaritmus) pro neabelovské grupy, viz [12], [13]. Pro praktické používání copánkových grup v kryptosystémech je třeba dalšího výzkumu, např. různých reprezentací prvků grupy, náhodného šifrování (náhodných copánků) apod., viz [9], [13], [14].

Text je rozdělen do následujících kapitol. Druhá kapitola obsahuje rychlý přehled o copech (geometrický pohled, algebraický pohled, definice pojmů), třetí kapitola zahrnuje popis aplikace copánků v asymetrické kryptografii, popis výměny společného šifrovacího klíče a kapitola čtvrtá se věnuje algoritmům pro práci s copánky.

2 Copánkové grupy

Teorie copánkových grup byla představena E. Artinem v roce 1926, jako možné řešení uzlového problému, tj. problému zda se dva uzly sobě rovnají. V roce 1923 J. W. Alexander ukázal, že každý uzel lze reprezentovat uzavřeným copánkem (Alexanderova věta). Její důkaz je konstruktivní. Ukažme si jej na příkladě, na obrázku 1 je zobrazen otevřený copánek (geometrický copánek), který lze získat z copánku uzavřeného (tzv. uzlu).

Obr. 1. Uzel reprezentovaný otevřeným copánkem.

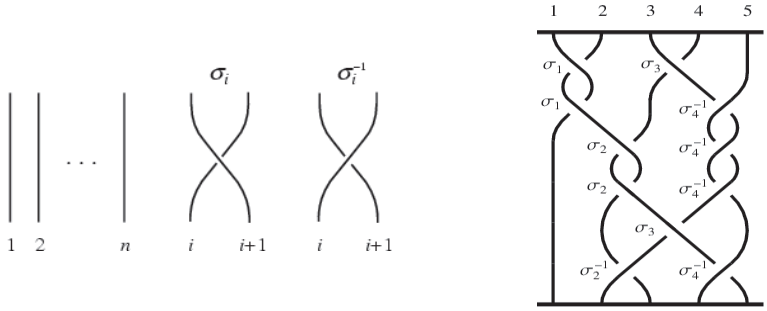
Na otevřené copánky ze dvou uzlů E. Artin aplikoval copánkové operace, viz (2). Podrobnosti viz [11], [15]. Poznamenejme, že v roce 1936 A. A. Markov řešil uzlový problém s užitím operací, viz (2) a tzv. Markovových operací I a II druhu. Markovovo řešení je propojení teorie uzlů s teorií copánků.

2.1 Otevřený (geometrický) copánek

Pro algebraický popis copánku vyjděme z jeho geometrické představy (viz obr. 2 a viz [11]), ve které si označíme místa křížení v jeho regulárním diagramu takto:

- pokud se kříží i-té vlákno *pod* $i+1$ vláknem, toto křížení se označí σ_i, pro $i = 1, 2, \ldots, n-1$,
- pokud se kříží i-té vlákno *nad* $i+1$ vláknem, toto křížení se označí σ_i^{-1}, pro $i = 1, 2, \ldots, n-1$.

Copánek může být popsán zřetězením operátorů σ_i a σ_i^{-1}, tak, že čteme jeho křížení shora dolů. Ověření, že množina geometrických copánků s operací zřetězení tvoří grupu, lze nalézt v [7], [15].



Obr. 2. Křížení vláken copánku. Copánek a jeho copánkové slovo $\sigma_1\sigma_3\sigma_1\sigma_4^{-1}\sigma_2\sigma_4^{-1}$ $\sigma_2\sigma_4^{-1}\sigma_3\sigma_2^{-1}\sigma_4^{-1}$.

Geometrické vyjádření míst křížení můžeme chápat jako prvky zdrojové abecedy S, jejíž mohutnost je $|S| = 2(n-1)$ a algebraické křížení můžeme chápat jako kódová slova, sestavená z kódové abecedy $A = \{\sigma_1, \sigma_2, \ldots, \sigma_{n-1}, \sigma_1^{-1}, \sigma_2^{-1}, \ldots, \sigma_{n-1}^{-1}\}$, jejíž mohutnost je $|A| = 2(n-1)$, tj. získali jsme kódovací schéma. Pokud aplikujeme poznatky z teorie kódování zjistíme, že kód

$$C = \{\sigma_1, \sigma_2, \ldots, \sigma_{n-1}, \sigma_1^{-1}, \sigma_2^{-1}, \ldots, \sigma_{n-1}^{-1}\} \tag{1}$$

je kódem blokovým (délka bloku je 1), je jednoznačně dekódovatelný a platí $|S| = |A|$ (McMillanova nerovnost). Užití kódu (1) ukazuje např. obr. 2.

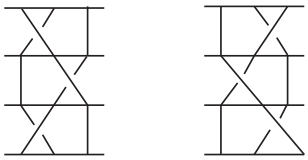
Definition 1. *(Algebraická definice copánkové grupy) Pro $n \geq 2$ má n-*copánková grupa B_n *následující prezentaci:*

$$B_n = \left\langle \sigma_1, \sigma_2, \sigma_3, \ldots, \sigma_{n-2}, \sigma_{n-1} \middle| \begin{matrix} \sigma_i\sigma_j = \sigma_j\sigma_i \quad pro\ |i-j| \geq 2 \\ \sigma_i\sigma_{i+1}\sigma_i = \sigma_{i+1}\sigma_i\sigma_{i+1}\ pro\ |i-j| = 1 \end{matrix} \right\rangle \tag{2}$$

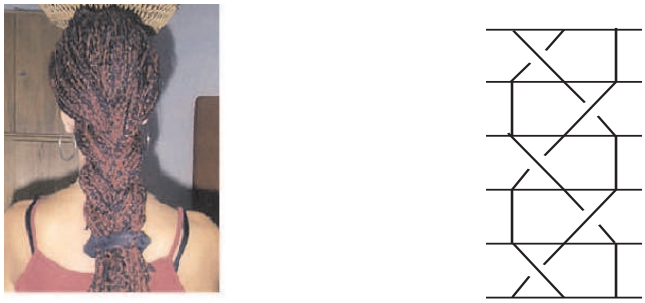
kde n je index copánku *(počet vláken copánku).*

Remark 1.

- Násobením $a * b$ dvou copánků a a b vznikne takový copánek, kde copánek a je umístěn nad copánek b (na jeho vrchol).
- Identita I je copánek, který se skládá z n přímých vláken.
- Inverzní copánek k copánku a je copánek, který vznikne z copánku a jeho zrcadlením podél vodorovné osy.
- Copánky jsou ekvivalentní (viz Příklad 1), jestliže jeden copánek přejde v druhý posloupností deformací, nebo podle isotopie mají stejný copánkový diagram, viz [11], [15].



Obr. 3. Copánkové operace $\sigma_i\sigma_{i+1}\sigma_i = \sigma_{i+1}\sigma_i\sigma_{i+1}$, tvorba ekvivalentních tříd.



Obr. 4. Copánek sestavený z vlasů a copánkový diagram pro prvek grupy B_3, $\sigma_1^{-1}\sigma_2\sigma_1^{-1}\sigma_2\sigma_1^{-1}$.

- Symboly σ_i se nazývají Artinovy generátory, užití generátorů viz [11].
- Grupa (2) je popsána s užitím kombinatorické teorie grup (množina generátorů a relací), obě množiny jsou konečné, proto má B_n konečnou prezentaci.
- Geometrický význam relátorů v (2) viz obr. 3.
- Příklad copánkové grupy s indexem $n = 3$, $B_3 =< \sigma_1, \sigma_2 | \sigma_1^{-1}\sigma_2\sigma_1^{-1} = \sigma_2\sigma_1^{-1}\sigma_2$ demonstrující výše uvedené je na obr. 4.

Example 1. Ukažte, že platí $\sigma_1\sigma_2\sigma_1\sigma_3\sigma_2 = \sigma_2\sigma_3\sigma_1\sigma_2\sigma_3$. Aplikací (2) na podtržené členy získáme: $\underline{\sigma_1\sigma_2\sigma_1}\sigma_3\sigma_2 = \sigma_2\sigma_1\underline{\sigma_2\sigma_3\sigma_2} = \sigma_2\underline{\sigma_1\sigma_3}\sigma_2\sigma_3 = \sigma_2\sigma_3\sigma_1\sigma_2\sigma_3$.

Užitím kódu (1) lze nakreslit regulární digramy jednotlivých pozitivních copánků.

Definition 2. Pozitivní copánek *je prvek B_n, který je napsán v kladných mocninách generátorů $\sigma_i, i = 1, 2, \ldots, n-1$, tj. bez použití inverzních operátorů σ_i^{-1}.*

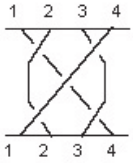
Remark 2. B_n^+ je množina pozitivních copánků v B_n. B_n^+ je monoid. Garside v [6] dokázal, že přirozený homomorfismus $f : B_n^+ \longrightarrow B_n$ je injekce. Tak jsou dva pozitivní copánky ekvivalentní v B_n právě tehdy když jsou ekvivalentní v B_n^+.

Mějme grupu n-permutací Σ_n, vnoření symetrické grupy Σ_n s n znaky do Σ_n^+ je indukováno zobrazením $\pi(i, i+1) = \sigma_i$, kde $(i, i+1)$ reprezentuje transpozici i na $i+1$ a $i+1$ na i. Nechť $\pi \in \Sigma_n$ je permutace $\pi : \{1, \ldots, n-1\} \longrightarrow$

$\{1, \ldots, n-1\}$, kterou jednoduše značme $\pi = (\pi(0), \pi(1), \ldots, \pi(n-1))$. Permutace $\pi = (3, 1, 2, 4)$ může být reprezentována součinem cyklů $(1, 2)(2, 3)$, který je mapován na $\sigma_1\sigma_1$.

Definition 3. *Pozitivní copánek se nazývá* pozitivně permutační *copánek, jestliže může být nakreslen jako geometrický copánek, ve kterém se každá dvojice vláken kříží nejvíce jedenkrát.*

Remark 3. Σ_n^+ je množina pozitivně permutačních copánků.

Copánkový diagram	Copánkové slovo (pozitivně permutační)	Přiřazená permutace
	$\sigma_1\sigma_3\sigma_2\sigma_3\sigma_1$	Vyjádříme ve tvaru cyklů: $(1, 4)$

Tab. 2. copánek a příslušná permutace.

Theorem 1. *Jestliže copánky $A_1, A_2 \in \Sigma_n^+$ indukují stejnou permutaci na jejich vláknech, pak $A_1 = A_2$. Pro každé $\pi \in \Sigma_n$ existuje jeden copánek $A_\pi \in \Sigma_n$ který je určen touto permutací.*
Důkaz. Důkaz viz [4]. □

Definition 4. *Nechť $a \in G$, kde $< G, *, 1 >$ je grupa. Prvek $b^{-1} * a * b$, pro nějaké $b \in G$ se nazývá* odvozený, konjugovaný *od a.*

3 Copánková kryptografie

Využití copánkových grup v kryptografii je vcelku novým přístupem, v širší povědomí tato možnost vešla až po zveřejnění článku [9], viz také [3].

Slovní konjugační problém (Word Problem) je klasický problém pocházející z Artinovy práce. Obtížnost jeho a dalších problémů (např. konjugačního vyhledávacího problému) je východiskem pro mnoho copánkovových kryptosystémů:

- Word Problem. Mějme dva copánky $\alpha, \beta \in B_n$. Určete, zda $\alpha = \beta$.
- Conjugacy Decision Problem. Mějme dva copánky $\alpha, \beta \in B_n$. Určete, zda α konjuguje s β.
- Conjugacy Search Problem. Mějme dva copánky $\alpha, \beta \in B_n$, β je konjugovaný s α. Nalezněte γ, takové, že $\gamma^{-1}\alpha\gamma = \beta$.
- Generalized Conjugacy Search Problem. Mějme dva copánky $\alpha, \beta \in B_n$, takové, že $\delta^{-1}\alpha\delta = \beta$ pro nějaké $\delta \in B_n$. Nalezněte γ takové, že $\gamma^{-1}\alpha\gamma = \beta$.

Copánková verze algoritmu Diffie-Hellman Jedním z asymetrických kryptografických algoritmů je algoritmus pro ustavení (výměnu) tajných klíčů a je založen na tzv. *Zobecněném konjugačním vyhledávacím problému* (Generalized Conjugacy Search Problem). Tento algoritmus je copánkovou verzí klasického algoritmu Diffie-Hellman, založeného na problému diskrétního logaritmu, viz [10].

Předpokládejme dvě podgrupy LB_l a RB_r grupy. Podgrupa LB_l (resp. RB_r) je podgrupu grupy B_{l+r}, která obsahuje copánky spletené z levých l vláken (resp. pravých r vláken) mezi $l+r$ vlákny. Důležitá vlastnost je komutativita, tedy platí $\forall a \in LB_l$ a $\forall b \in RB_r$ platí $ab = ba$.

- Veřejné hodnoty jsou vhodná dvojice celých čísel (l, r) a dostatečně komplikovaný $(l + r)$-copánek $x \in B_{l+r}$.
- Alice vybere náhodný tajný parametr - copánek $a \in LB_l$ a Bobovi pošle $y_1 = axa^{-1}$.
- Bob vybere analogicky copánek $b \in RB_r$ a pošle $y_2 = bxb^{-1}$ Alici.
- Alice vypočte sdílený tajný klíč $K_{AB} = ay_2a^{-1}$.
- Bob vypočte sdílený tajný klíč $K_{AB} = by_1b^{-1}$.
- Jelikož $a \in LB_l$ a $b \in RB_r$, $ab = ba$. Proto $ay_2a^{-1} = a(bxb^{-1})a^{-1} = b(axa^{-1})b^{-1} = by_1b^{-1}$.

4 Algoritmy pro copánkovou grupy

V této sekci si ukážeme základní algoritmy (viz [8], [5]) pro práci s copánkovou grupou, které lze užít např. pro copánkovou kryptografii. Algoritmy jsou implementovány v jazyce PERL. Copánky jsou reprezentovány prostřednictvím permutací. Sestavme tedy následující kódovací schéma:

$$\begin{array}{cccccc} \sigma_1 & \sigma_2 & \sigma_3 & \dots & \sigma_{n-2} & \sigma_{n-2} \\ (1,2) & (2,3) & (3,4) & \dots & (n-2,n-1) & (n-1,n) \end{array}$$

Zdrojová abeceda $S = \{\sigma_1, \sigma_2, \sigma_3, \dots, \sigma_{n-1}\}, |S| = n - 1$, kódová abeceda $A = \{1, 2, 3, \dots, n\}$. Kód $C_1 = \{(1,2), (2,3), (3,4), \dots, (n-1,n)\}$. Tento kód je blokový (tj. jednoznačně dekódovatelný) s délkou bloku 2. Poznamenejme, že E. Artin je nazval elementární transpozice. Z pohledu teorie grup se jedná o generátory Σ_n.

4.1 Přiřazení permutační tabulky pozitivnímu copánku

Algoritmus je podobný jako při např. sčítání m celých čísel.
Vstup: pozitivně permutační copánek (řetez generátorů σ_i), jeho indexy jsou prvky pole *copanek*.
Výstup: permutační tabulka.

```
... for i = 0 to delka−1 do {
  k=copanek[i];
  j=k+1;
  puvk=permutace[k];
```

```
  puvj=permutace[j];
  permutace[j]=puvk;
  permutace[k]=puvj;
}...
```

Example 2. Vstup: $\sigma_1\sigma_2\sigma_2\sigma_3\sigma_2\sigma_2\sigma_1\sigma_2\sigma_2\sigma_3$. Výstup: identická permutace.
O správnosti se lze přesvědčit, nakreslením příslušného permutačního diagramu. Časová složitost algoritmu je $O(n)$, kde n je copánkový index.

4.2 Permutační tabulce se přiřadí pozitivní copánek

Pozitivní copánek je reprezentován permutační tabulkou π (viz 4.1). Cílem je převedení permutační tabulky do tvaru, který je roven identickému prvku v Σ_n, $\pi w = 1$ (w je hledaný pozitivní copánek). Toho docílíme, tak, že budeme postupně skládat permutaci $\pi\sigma_i$ pro $i = 1, 2, 3, \ldots, n-1$ (čímž dojde k výměně hodnot $\pi(i)$ a $\pi(i+1)$ pro i a $i+1$. Tj. nejdříve zjistíme, pro který prvek i platí, že $\pi(i) = 1$, pak se provede $\pi(i-1, i) = \pi\sigma_{i-1}$, tím se obdrží, permutace π' kde platí $\pi'(i-1) = 1$, permutaci $\pi'(i-2, i-1) = \pi'\sigma_{i-2}$, tak to postupujeme tak dlouho, až se získá permutace π" kde platí π"$(1) = 1$. Hodnoty v permutaci se tedy posunou o jednu hodnotu doprava, mezi prvky 1 až i. Podobně pro prvky $2 \ldots n$.
Vstup: pole obsahující permutační tabulku, pole permutace.
Výstup: pozitivní copánek ve tvaru zřetězení generátorů σ_i.

```
$min=1; $pozice=1; $i=1;
$copanek_index=pocet_prvku_v_poli_permutaci; ...

if ($permutace[$i] == $min) {
  # zjisteni prvku v poli, ktery je roven postupne 1,2,..., n
  $k=$i-1;
  for ($j=$k;$j>=$pozice;$j--)
  { # cyklicky posun prvku v poli doprava o jednu pozici
    $permutace[$j+1]=$permutace[$j];
    # vlozeni indexu j generatoru j do pole
  }
  $min+=1;   # zvetseni hledane hodnoty
  $pozice+=1; # zvetseni hodnoty pozice od které prvky se posunou doprava
  $i=$pozice;
}...
```

Example 3. Vstup: $(6, 2, 1, 7, 5, 3, 4)$ - permutace je vyjádřena v jednořádkovém zápisu. Výstup: $\sigma_2\sigma_1\sigma_2\sigma_5\sigma_4\sigma_3\sigma_6\sigma_5\sigma_4\sigma_6\sigma_5$.
Časová složitost je $O(n^2)$. Aplikováním relací (2) lze získat k danému pozitivnímu copánku ekvivalentní copánky např. $\sigma_1\sigma_2\sigma_1\sigma_5\sigma_4\sigma_3\sigma_6\sigma_5\sigma_4\sigma_6\sigma_5$, $\sigma_2\sigma_1\sigma_5\sigma_2\sigma_4\sigma_3\sigma_6$ $\sigma_5\sigma_4\sigma_6\sigma_5$ atd.

4.3 Porovnání dvou pozitivních copánků

Pro ekvivalenci dvou copánku se použije kritéria uvedeného v poznámce 1.

Oba pozitivní copánky si vyjádříme ve tvaru permutačních tabulek π_1, π_2 (podle 4.1). Takto získané permutační tabulky (prvky S_n) porovnáme, tj. $\pi_1(i) = \pi_2(i)$ pro $i = 1, 2, \ldots, n$.
Vstup: dva pozitivní copánky.
Výstup: jsou/nejsou ekvivalentní.

```
#Ziskani permutacnich tabulek viz 4.1

$i=1; $shoda=1; while(($i<=$copankovy_index)&&($shoda == 1)) {
  if ($permutace0[$i] != $permutace1[$i])
  {
    $shoda =0;
  }
  $i++;
}...
```

Example 4. Vstup: $\sigma_1\sigma_2\sigma_1\sigma_3\sigma_2$ a $\sigma_2\sigma_3\sigma_1\sigma_2\sigma_3$. Výstup: copánky jsou ekvivalentní.
Časová složitost je $O(n)$.

4.4 Výpočet hodnoty vnitřního automorfismu $(f(a))^u = Da^uD^{-1}$

Copánky D a a si vyjádříme užitím permutačních tabulek (viz 4.1). Pro určení a^u je vhodné si vyjádřit permutační tabulku pro copánek a užitím cyklů. Takto se časově efektivně provede k-krát složit příslušnou permutační tabulku. Pokud si označíme $(f(a))^u = b$, jedná se o relaci konjugace a^u s D.

1. Vyjádření permutační tabulky s užitím cyklů
Postup jak sestavit cykly z dané permutační tabulky je znám z lineární algebry, viz [1].
Vstup: copánek a.
Výstup: cykly příslušející copánku a.

Algoritmem z 4.1 se copánku a přiřadí permutační tabulka, pro implemntaci užito asociativni pole permutace.

```
... $i=1; # identifikace zkoumaneho prvku v permutaci $j=-1; $m=0;
while($n > 0) {
  if ( exists ($permutace{$i}))
  {
  ...
    { #i-ty prvek je nesamodruzny
      $zacatek=$i;
      $k=$permutace{$i};
      $j+=1;
      $cyklus[$j][$m]=$zacatek; # prvni prvek v j-tem cyklu
```

```
delete($permutace{$i});
$n−=1;
#print $n." ".$zacatek."\n";
while($zacatek != $permutace{$k})
{ # cyklus není uzavren
  $m+=1;
  $cyklus[$j][$m]=$k; # dalsi prvek v j−tem cyklu
  $hodn=$permutace{$k};
  delete($permutace{$k});
  $k=$hodn;
  $n−=1;

  #print $n." ".$cyklus[$m]."\n";
}
$m+=1;
$cyklus[$j][$m]=$k;  # posledni prvek v j−tem cyklu
push(@pole_delek,$m); # vlozeni poctu prvku v j−tem cyklu
...
```

Example 5. Vstup: $(8, 12, 1, 2, 14, 11, 6, 15, 10, 9, 13, 5, 7, 3, 4)$ - permutace je vyjádřena jednořádkovou notací. Výstup: cyklus $1 = (1, 8, 15, 4, 2, 12, 5, 14, 3)$, Cyklus $2 = (6, 11, 13, 7)$, cyklus $3 = (9, 10)$.

Časová složitost je $O(n)$.

2. Hodnota copánku a^k.
Vstup: Vyjádření permutace užitím cyklů viz 4.4.1; hodnota mocniny u
Výstup: Permutace odpovídající u-té mocnině prvku a.

```
#Vystup z algoritmu v 4.4.1 − cykly dane permutaci

for($i=0;$i<=$j;$i++) { #pruchod jednotlivymi cykly v permutaci
  $delka=$pole_delek[$i]+1;
  for($k=0;$k<$delka;$k++)
  {
    $prvek=$cyklus[$i][$k];   # k−ty prvek v i−tem cyklu
    print $prvek." ";
    $novy_index=($k+$mocnina)\%$delka
    $nova_hodn=$cyklus[$i][$novy_index]; ;#urceni obrazu k−teho prvku v i−
        tem cyklu
    $vysl_permutace[$prvek]=$nova_hodn; # nova hodnota pro i−ty prvek v
    }
 print "\n";
}
```

Example 6. Vstup: Cykly z výstupu viz 4.4.1, mocnina $u = 15$. Výstup: $(5, 8, 12, 1, 4, 13, 11, 14, 9, 10, 7, 15, 6, 2, 3)$ - jednořádková notace permutace.
Časová složitost je $O(n)$.

4.5 Výpočet konjugace $b = Da^uD^{-1}$

copánky a^u a D si vyjádříme ve tvaru jim příslušejících permutací, algoritmus viz 4.1. V tomto tvaru se hodnota konjungovaného prvku b určí tak, že prvky v permutační tabulce příslušející copánku a^u se nahradí jejich obrazy z permutační tabulky, která přísluší copánku, kterým konjungujeme, viz [7]. Poznamenejme, že permutační tabulku příslušející copánku a^u si vyjádříme ve tvaru cyklů - algoritmus 4.4.1.
Vstup: pozitivní copánky a^u a D.
Výstup: copánek, který je s a^u konjungovaný, pomocí copánku D

```
#Algoritmus 4.1 vyjadri copanky v permutacnich tabulkach
#Algoritmus 4.4.1 vyjadri copanek a^u ve tvaru disjunktnich cyklu

... for($i=0;$i<=$j;$i++) {
  $delka=$pole_delek[$i]+1;
  for($k=0;$k<$delka;$k++)
  {
    $prvek=$cyklus[$i][$k];
    $cyklus[$i][$k]=$cim_konjugace{$prvek}; #konjugace copankem D
  }
}
```

Algoritmus 4.2 přiřadí permutační tabulce konjungovaného prvku odpovídající pozitivní copánek.

Example 7. Vstup: $u = 1$ (mocnina), copánek $a = \sigma_2\ \sigma_1\ \sigma_3\ \sigma_2\ \sigma_{13}\ \sigma_{12}\ \sigma_{11}\ \sigma_{10}\ \sigma_9\ \sigma_8\ \sigma_7\ \sigma_6\ \sigma_5\ \sigma_4\ \sigma_3\ \sigma_{14}\ \sigma_{13}\ \sigma_{12}\ \sigma_{11}\ \sigma_{10}\ \sigma_9\ \sigma_8\ \sigma_7\ \sigma_6\ \sigma_5\ \sigma_4\ \sigma_{13}\ \sigma_{12}\ \sigma_{11}\ \sigma_{10}\ \sigma_9\ \sigma_8\ \sigma_7\ \sigma_6\ \sigma_5\ \sigma_9\ \sigma_8\ \sigma_7\ \sigma_6\ \sigma_{14}\ \sigma_{13}\ \sigma_{12}\ \sigma_{11}\ \sigma_{10}\ \sigma_9\ \sigma_8\ \sigma_7\ \sigma_{13}\ \sigma_{12}\ \sigma_{11}\ \sigma_{10}\ \sigma_9\ \sigma_{13}\ \sigma_{12}\ \sigma_{11}\ \sigma_{10}\ \sigma_{12}\ \sigma_{11}\ \sigma_{14}\ \sigma_{13},$

Copánek $D = \sigma_{14}\ \sigma_{13}\ \sigma_{12}\ \sigma_{11}\ \sigma_{10}\ \sigma_9\ \sigma_8\ \sigma_7\ \sigma_6\ \sigma_5\ \sigma_4\ \sigma_3\ \sigma_2\ \sigma_1\ \sigma_{14}\ \sigma_{13}\ \sigma_{12}\ \sigma_{11}\ \sigma_{109}\ \sigma_8\ \sigma_7\ \sigma_6\ \sigma_5\ \sigma_4\ \sigma_3\ \sigma_2\ \sigma_{14}\ \sigma_{13}\ \sigma_{12}\ \sigma_{11}\ \sigma_{10}\ \sigma_9\ \sigma_8\ \sigma_7\ \sigma_6\ \sigma_5\ \sigma_4\ \sigma_{31}\ \sigma_4\ \sigma_{13}\ \sigma_{12}\ \sigma_{11}\ \sigma_{10}\ \sigma_9\ \sigma_8\ \sigma_7\ \sigma_6\ \sigma_5\ \sigma_4\ \sigma_{14}\ \sigma_{13}\ \sigma_{12}\ \sigma_{11}\ \sigma_{10}\ \sigma_9\ \sigma_8\ \sigma_7\ \sigma_6\ \sigma_5\ \sigma_{14}\ \sigma_{13}\ \sigma_{12}\ \sigma_{11}\ \sigma_{10}\ \sigma_9\ \sigma_8\ \sigma_7\ \sigma_6\ \sigma_{14}\ \sigma_{13}\ \sigma_{12}\ \sigma_{11}\ \sigma_{10}\ \sigma_9\ \sigma_8\ \sigma_7\ \sigma_{14}\ \sigma_{13}\ \sigma_{12}\ \sigma_{11}\ \sigma_{10}\ \sigma_9\ \sigma_8\ \sigma_{14}\ \sigma_{13}\ \sigma_{12}\ \sigma_{11}\ \sigma_{10}\ \sigma_9\ \sigma_{14}\ \sigma_{13}\ \sigma_{12}\ \sigma_{11}\ \sigma_{10}\ \sigma_{14}\ \sigma_{13}\ \sigma_{12}\ \sigma_{11}\ \sigma_{14}\ \sigma_{13}\ \sigma_{12}\ \sigma_{14}\ \sigma_{13}\ \sigma_{14}$

Výstup: Konjungovaný copánek: $\sigma_7\ \sigma_6\ \sigma_5\ \sigma_4\ \sigma_3\ \sigma_2\ \sigma_1\ \sigma_{10}\ \sigma_9\ \sigma_8\ \sigma_7\ \sigma_6\ \sigma_5\ \sigma_4\ \sigma_3\ \sigma_2\ \sigma_6\ \sigma_5\ \sigma_4\ \sigma_3\ \sigma_{13}\ \sigma_{12}\ \sigma_{11}\ \sigma_{10}\ \sigma_9\ \sigma_8\ \sigma_7\ \sigma_6\ \sigma_5\ \sigma_4\ \sigma_{11}\ \sigma_{10}\ \sigma_9\ \sigma_8\ \sigma_7\ \sigma_6\ \sigma_5\ \sigma_{10}\ \sigma_9\ \sigma_8\ \sigma_7\ \sigma_6\ \sigma_{10}\ \sigma_9\ \sigma_8\ \sigma_7\ \sigma_{14}\ \sigma_{13}\ \sigma_{12}\ \sigma_{11}\ \sigma_{10}\ \sigma_9\ \sigma_8\ \sigma_{10}\ \sigma_9\ \sigma_{12}\ \sigma_{11}\ \sigma_{10}\ \sigma_{12}\ \sigma_{11}$

Časová složitost je $O(n)$.

Cílem implementace algoritmů bylo provést experimenty s grupami, které mají copánkový index 500 a více. V dostupných zdrojích, např. [2], totiž byly prováděny experimenty s grupami s menším copánkovým indexem. V tabulce 3 jsou uvedeny výsledky pro algoritmus 4.1.

Copánkový index	Počet křížení	Doba [s]	Pozn.
500	124750	1	Základní copánek
1000	499500	5	Základní copánek
2000	1999000	20	Základní copánek
500	22500	1	
1500	202500	2	
2000	977500	10	
2000	990000	10	
2500	1440000	15	

Tab. 3. Výsledky experimentů (PERL, WEB, IIS 5.0, MS Windows 2000 SP 4, RAM 130MB, Intel Pentium II MMX, 330MHz (pro algoritmus 4.1)).

5 Závěr

V tomto článku jsme podali stručný úvod do teorie copánkových grup. Dále jsme ukázali jejich využití v kryptografii na konkrétním algoritmu pro ustavení klíčů. Nakonec jsme předvedli základní operace s copánky prostřednictvím permutací na grupách s copánkovým indexem větším než 500. Naše další práce se bude týkat experimentů s copánkovými grupami s ještě větším indexem, dále implementace dalších algoritmů a rovněž implementace copánkových kryptosystémů založených na „velkých“ copánkových grupách. Kryptografické systémy vyžadují množinu generátorů určité velikosti, proto chceme také provést porovnání copánkových kryptosystémů s ostatními vzhledem k této množině.

Reference

1. Bečvář J.: Lineární algebra. MatFyzPress, Praha, 2000.
2. Campagna M.J.: Algorithms in Braid Group. Cryptology ePrint Archive, 2003 `http://www.iacr.org/`.
3. Dehornoy P.: Braid-Based Cryptography. 2003. `http://www.tcs.hut.fi/~helger/crypto/link/public/braid/`.
4. Elrifai E.A., Morton H.R.: Algorithms for Positive Braids. Quart J.Math. Oxford **45**, 1994, 479–497.
5. Franco N., González-Meneses J.: Conjugacy Problem for Braids Groups and Garside Group, 2002. `http://www.tcs.hut.fi/~helger/crypto/link/public/braid/`.
6. Garside F.A.: The Braid Group and Other Groups. Quart. J. Math., Oxford, **20**, 78, 1969, 235-254.
7. Humphreys J.F.: A Course in Group Theory. Oxford Univerity Press, 2004.
8. Cha J.Ch., Ko K.H., Lee S.J., Han J.W., Cheon J.H.: An Efficient Implementation of Braid Groups, 2001. `http://www.math.snu.ac.kr/~jhcheon/Published/2001_Asiacrypt/braid-impl.pdf`.
9. Ko K.H., Lee S.J., Cheon J.H., Han J.W., Kang J., Park C.: New Public Cryptosystem Using Braid Group, 2000. `http://www.tcs.hut.fi/~helger/crypto/link/public/braid/`.
10. Menezes A., van Oorschot P., Vanstone S.: Handbook of Applied Cryptography, 2001. `http://cacr.math.uwaterloo.ca/hac/`.

11. Murasugi K., Kurpita B.I.: A Study of Braids. Kluwer Academic press, 1999.
12. Prenneel B., et al.: Research Agenda for the Future of Cryptology, 2003. http://www.stork.eu.org/documents/RUB-D52_1.pdf.
13. Prennel B., et al.: New Trends in Cryptology, 2003. http://www.stork.eu.org/documents/ENS-D41_4.pdf.
14. Procházka L.: Úvod do studia reprezentací grup. Karolinum, Praha, 1999.
15. Stilwell J.: Classical Topology and Combinatorial Group Theory. Springer, 1995.

Maintaining consistency in disk file systems

Mikuláš Patočka

Charles University, Faculty of Mathematics and Physics, Prague, Czech Republic

Abstract. With growing sizes of disks, checking the whole filesystem becomes too slow and it is inappropriate to do it after every system crash. Methods have been developed to maintain disk consistency across crashes, the most widely used is journaling, others are phase tree or soft updates. This paper describes new method for maintaining data consistency — crash counts — and shows using this method in real filesystem implementation.

1 Introduction

During several last years disk sizes have grown several order of magnitudes. Even though disk access speed has improved too, time to scan the whole disk became much longer [1]. After a crash, operating systems used to scan all filesystem metadata and repaired the filesystem. Today it is unacceptable, because it may cause downtime of several hours. Another problem with repairing the filesystem is that it may violate access security — after recovery from a crash users could read unallocated parts of the disk with possible confidential information belonging to other users. There exist methods to keep the filesystem consistent after a crash. This paper presents new method of maintaining filesystem consistency across crashes and describes a filesystem implementation using it.

2 Existing methods

2.1 Journaling

Journaling [2] is the most common method for maintaining consistency of disk after a crash. A part of a filesystem is reserved as a journal (some filesystems can have journal on an external device). Transaction is a set of modifications that transfer the filesystem from one consistent state to another. For example, when creating a file, a transaction consists of marking the new inode dirty in inode bitmap, writing the new inode and updating the directory. Transaction is first written to the journal and no data are written to their normal disk positions. When the transaction is completely in journal, disk cache is flushed, special commit mark is written to journal, disk cache is flushed again and the data can be written to normal positions.

If system crashes before the commit mark was written, the journal is ignored on next mount. If system crashes after the commit mark is written, operations

in journal are replayed — written to disk positions where they should be. This causes that either the whole transaction is ignored or written to disk. Another less common variant of journaling operates so that old data before transaction are written to journal and transaction is rolled-back after crash (NTFS is supposed to use both methods).

Journaling is used in many filesystems — JFS [3] (IBM's filesystem for AIX, OS/2 and Linux), XFS [4] (SGI's filesystem for Irix and Linux), Ext3 [5] (Linux filesystem compatible with Ext2), ReiserFS [6] (Hans Reiser's filesystem for Linux) NTFS [7] (Windows NT/2000/XP/2003 filesystem).

However, journaling causes a lot of design problems in VFS layer and filesystem driver.

Journal must not be filled up, so before starting transaction, system must compute the maximum amount of journal space that this transaction might consume and wait if there is not sufficient space.

Buffers of uncommitted transactions must be pinned to the buffer cache and not written to the disk. This complicates the design of the buffer cache and may cause deadlock if the free memory goes low, because the memory can't be reclaimed from the buffers.

For performance reasons it is not acceptable to write the file content to journal. This complicates synchronization between data and metadata. For example imagine a case when a directory is created, deleted and a file is allocated at the same place. If special care was not taken, replaying the journal after crash would overwrite the file with previous (already deleted) directory and corrupt it.

There are numerous bugs in journaled filesystems, for example:
`http://www.uwsg.iu.edu/hypermail/linux/kernel/0310.3/0887.html`
`http://www.uwsg.indiana.edu/hypermail/linux/kernel/0409.1/1123.html`
`http://marc.theaimsgroup.com/?l=linux-xfs&r=1&b=200405&w=2`

2.2 Phase tree

Phase tree [8] is another method that maintains filesystem consistency across crashes. The whole filesystem is viewed as a tree of pointers. Superblock points to bitmap directory, bitmap directory points to bitmaps, superblock points to root inode, inode points to blocks containing directory content, directory entries point to another inodes and so on. All updates are written only to unallocated places in the filesystem. When the superblock is written with new pointers, filesystem atomically transfers from one consistent state to another.

Phase tree is used in Linux filesystem tux2. It is probably slower because it has to write more blocks (the whole pathway from root) than normal filesystems.

2.3 Soft updates

Soft updates [9], unlike journaling and phase tree, do not preserve filesystem consistency. Instead, they cause that the filesystem can be corrupted only in minor non-severe ways. When filesystem driver marks dirty buffers, it specifies order in which the buffer must be written to disk. For example, when a file is

created, driver specifies, that the first should be written an inode, then inode bitmap and finally directory entry. Kernel bufdaemon thread writes buffers in this order and issues disk cache flush operations correctly. If system crashes at any time, it may cause lost inode but not pointer pointing to unallocated inode.

Soft updates are implemented in FreeBSD. FreeBSD 5, because of filesystem snapshot feature, allows checking the filesystem while it is mounted — snapshot is taken, that static snapshot is checked, and fsck instructs the kernel to free lost blocks and inodes. With this feature FreeBSD 5 allows immediate recovery after crash, except that performance is degraded for the time of fsck running.

3 Crash counts

3.1 Overview

In this paper I present new method for maintaining filesystem consistency — crash counts.

Crash counts use different approach to keep the filesystem consistent. Disk contains preallocated part — a crash count table. Data structures contain two additional values. These values together with the crash count table determine, if the structure is valid. When writing new files and directories they are written in such way that they will be considered invalid according to crash count table if the system crashes. Some structures (bitmaps) have two versions, when they are written the version that is currently considered invalid is being written.

When new crash count table is written, the new files and bitmaps are atomically made valid. This keeps the filesystem consistent in any case. If crash happens before crash count table is written, new file records will be treated as invalid and old bitmaps will be used. If crash happens after writing crash count tables, new files will be treated as valid and new bitmaps with files' place marked as allocated will be used.

Crash counts solve a lot of problems existing in journaling filesystems — filesystem with crash counts needs no special ordering of disk writes and can take more advantage of on-disk write cache. This improves write performance. Crash counts filesystem can access disk buffers just like ordinary non-journaling filesystem and let the operating system kernel flush them. No buffers are pinned to the memory. Because the filesystem can write and free all its dynamically allocated data structures at any time it is deadlock-free even under extreme conditions.

4 Description of crash counts

Each structure on disk that needs to be kept consistent with other structures in case of crash contains additional values — crash count (cc) and transaction count (txc). Disk contains table where there for each crash count exists a transaction count. The table is initialized to all zeros on filesystem creation. That table is also loaded in memory (`fs->cct`, fs is structure describing each mounted filesystem).

(in my implementation crash count is 16 bits and transaction count is 32 bits, resulting in 256kB table size). Disk also contains current crash count (`fs->cc`). `fs->txc` is just a shortcut for `fs->cct[fs->cc]`.

On mount, filesystem loads `fs->cct` and `fs->cc` from disk to memory. It increases crash count on disk (but leaves old `fs->cc` in memory) and increases `fs->cct[fs->cc]` in memory (but leaves old value on disk).

On sync, filesystem writes all dirty buffers and pages then flushes disk cache, writes its `fs->cct` to disk, flushes disk cache again and again increases (in memory only) `fs->cct[fs->cc]`.

On unmount, filesystem is synced, crash count on disk is decreased and disk cache is flushed.

Structures that form lists (for example directory entries) with (cc, txc) pair on disk are considered valid if `fs->cct[cc] - txc >= 0`. If this condition is not valid the structure is containing invalid data and is skipped. New structures are created with `cc = fs->cc` and `txc = fs->txc`. When deleting the structure, it is set with `cc = fs->cc` and `txc = fs->txc ^ 0x80000000`.

There are no constraints on buffer cache write operations — kernel can write any buffers with any order. All buffers must only be written before sync operation.

If system crashes after the structure was created but before sync operation, system boots with crash count increased by one. It will never again modify `fs->txc[cc]` and that value will be one-less than txc of that pointer — so condition `fs->cct[cc] - txc >= 0` will be always false and the structure will never ever be considered valid. It will be eventually overwritten with another structure. If the system crashes after the structure was created and sync finished the new `fs->cct` will be already written to disk and the structure will be always considered valid.

If system crashes after the structure was deleted but before sync operation, `fs->cct[cc] - txc` will be equal `0x7fffffff` and the structure will be considered valid. If system crashes after sync `fs->cct[cc] - txc` will be `0x80000000` and the structure will be considered invalid.

Other structures (in my implementation that are free-space allocation pages and directory hash tables) aren't exactly lists. For these structures, the filesystem needs to hold two versions of structures and (cc, txc) pair. The first version of the structure is valid if `fs->cct[cc] - txc >= 0` and the second version is valid otherwise. When writing to this structure filesystem checks if `cc == fs->cc && ((txc & 0x7fffffff) == fs->txc)` — if true write is allowed. If not true, the valid version is copied into the other section and (cc, txc) is set so that `txc = ( (fs->cct[cc] - txc) & 0x80000000) | fs->txc` and `cc = fs->cc`.

If the system crashes before sync the old version of the structure will be still considered valid. If the system crashes after sync the new version will be considered valid. Copying the whole structure might be slow but it is done only once between syncs.

New blocks are allocated if they are free in both bitmaps (but written only to the current bitmap). If there's no free space available the filesystem must

perform sync operation to actually free the blocks of possibly previously deleted files. If new block can't be allocated again after sync the filesystem is considered full and `ENOSPC` is returned.

File's blocks are described in a different way. I could use pointer with (cc, txc) pairs or doubling structures but I decided to abandon them to save space. Each file contains runs — two directly in the file structure (fnode), if more is required special allocation structures (anodes) are allocated. The runs are not protected with any crash counts. Instead fnode contains two file-size values size0 and size1 and (cc, txc) pair to describe which one is valid (as described in previous paragraphs). Runs are only considered valid as long as they allocate blocks less or equal than currently valid file size.

When extending files more sectors are added to the last run or new runs are created and the invalid version of size is extended. If the system crashes before sync the new sectors in runs are considered invalid because they exceed the file size. If the system crashes after sync the new size will be considered valid and file will be properly allocated.

The trick happens when the file was truncated and extended before sync. In this case runs cannot be updated with pointers to newly allocated space because in case of crash the old runs are required to exist. In such situation already free space of old runs is "resurrected". The space is already marked as allocated in currently invalid (in case of crash valid) bitmaps. The resurrecting procedure marks it valid even in current bitmaps and the filesystem continues to use this space as new file storage. Thus the block allocator has three functions (allocate blocks, free blocks, resurrect blocks) instead of usual two present on common filesystems. Another possibility to resolve this situation would be to force sync at this position but for a better performance I decided to do resurrecting instead.

Problems happen if crash count overflows 65536 — in this case the whole filesystem must be walked crash count on all structures set to 0 and global crash count reset. However I don't think that this limitation happens in practice — no system will crash 65536 times. When the 31-bit transaction count overflows, crash count is automatically increased during filesystem operation and transaction count is reset to zero, so the filesystem is not limited to 2^{31} transactions. The crash count will overflow after 2^{47} transactions but again I don't think that it's practical limit.

5 SPADFS filesystem

5.1 Filesystem structure

In this section I will describe the structure of new SPADFS filesystem using crash counts. SPADFS was developed from a scratch. Its design goals were data consistency across crashes, high performance and taking space efficiency.

Superblock is located at sector 16384 leaving 8MB of unused data for kernel (currently kernel has 1MB but I decided to reserve more space in case the kernel grows or in case of porting the system to RISC architectures with much bigger code size). Superblock contains general information and pointers to txblock,

two apage index pointers, crash count table (cct) and root fnode. Txblock contains general filesystem information that is changed during filesystem activity. The rationale for splitting information between superblock and txblock is that if txblock is written badly the filesystem could still be salvaged. If superblock information were lost with it there is little possibility to recover any data. Txblock contains crash count (cc) and (a_cc, a_txc) pair which determines what version of apage index is valid.

Data is allocated using apages. Apage index contains pointers to individual apages and range of disk blocks that they cover. Apage contains double-linked list of struct aentry. aentry is a pair of (start, len). List is sorted. There is additional freelist of free aentries. Search is done with average $O(\sqrt{n})$ complexity — filesystem takes randomly $\sqrt{n}$ aentries, selects the one with highest but less-than desired value and uses sequential search from this. If apage is filled up with aentries it is split into two apages and apage index is updated. If the apage covers too small space it is converted to a bitmap. Bitmaps are more efficient for describing very fragmented free space but they are inefficient for large runs of free or allocated space — in this case lists of blocks are more efficient. I decided not to use trees for free space allocation — they are theoretically faster but their practical implementation is too complex. Lists of structures offer $O(\sqrt{n})$ complexity which is acceptable. Because each apage can contain up to 1023 entries (its size is 32768 it has two versions and size of one aentry is 16) $\sqrt{n}$ can be at most 32.

Free space is divided into three zones — metadata zone, small files zone and large files zone. Blocks are first allocated in the zone where they belong, if it is full they are allocated in other zones. Zones are only specified in superblock one apage can span multiple zones. The purpose of metadata zone is that filesystem can do extensive readahead on it and it greatly improves scanning directory structure. Allocating small and large files in different zones reduces fragmentation.

Each fnode describes a file or directory. Fnode contains two sizes and (cc, txc) pair describing which one is valid. The same pair (with flags in fnode header) is used to describe if the fnode is valid at all or if it's just a free space in fnode block. Fnode contains two direct data runs and an optional pointer to anode containing more runs.

Anodes contain array of struct extent. Each anode contains at least one direct extent and some indirect extents pointing to other anodes. (The rationale for the requirement of at least one direct extent was that allocating one file block should allocate at most two disk blocks to make accounting of lazy-allocations easy). The root anode contains 20 direct extents, one one-level indirect extent (containing pointer to anode of full 31 direct extents), one two-level indirect extent (containing 1 direct extent and 30 pointers to one-level indirect anodes) and so on up to 10-level indirect extent. The root anode also contains a copy of the first two direct extents from fnode and file name so that when the fnode is damaged the file can still be recovered from the anode. I decided to not use balanced trees for anodes because operations on trees are complex (they are simple in memory-only implementations but in real tree-based filesystems like

ReiserFS, XFS or JFS code for tree operations is very large and prone to bugs) — instead they are like unix indirect blocks but unlike indirect blocks they contain sector runs not individual sectors. Filesystem doesn't support sparse files because they are used rarely and they would only complicate the design.

Directories are stored as a linear list of fnodes (struct fnode_block). If the list is too long it is split into multiple fnode blocks and extensive hashing is used to locate them. struct dnode is allocated that contains pointers to fnode blocks. On search filename is hashed, appropriate place in dnode is found according to hash and that one fnode block is searched. If one dnode is not enough there can be multiple nested levels. Finally, if allocating the dnode is not possible (for example because of too many hash collisions or if the filesystem is too fragmented so that it doesn't contain more continuous blocks) fnode blocks can form a chain.

5.2 Tunable parameters

Filesystem has various parameters for tuning performance. Some are set when creating the filesystem, some can be set on mounting.
Parameters when creating the filesystem:

`-block-size <number>` (default 512)
: Specifies size of minimum IO unit. Filesystem won't issue any IO operations less than this value. Specifying larger block-size won't increase speed and will only waste more space. Use this parameter if your physical device can't do IO operations at 512-byte granularity.

`-page-size <number>` (default 32768)
: Page size of target operating system. Buffer cache can't load more continuous data that one page. Filesystem won't create structures that cross page size boundary. Filesystem can't be mounted on operating system with lower page size.

`-fnode-size <number>` (default 8192)
: Largest size of fnode block. Fnode block could be as large as page size but searching such block sequentially would be slow. So this parameter limits its size. If the fnode block would overflow, it is split and hash is used to access both halves.

`-cluster-size <number>` (default 32768)

`-cluster-threshold <number>` (default 131072)
: Files larger or equal than cluster-threshold are aligned to cluster-size and allocated in big-files zone of the filesystem. This prevents fragmentation of files as it's almost guaranteed (unless you fill up the whole filesystem with small files) that minimum fragment size is cluster-size. If you'll use the filesystem for storing real-time multimedia data you can increase both values to get some sort of guaranteed throughput.

`-zone-size <number>` (default 1/64 of the filesystem size)
: Reserves this amount of bytes for metadata zone. If the zone is full metadata are allocated in small-file-zone and then in big-file-zone. If file zones are full files are allocated in metadata zone. If you are going to store lots of small files set larger zone, if you are going to store lots of large files set small zone.

Parameters when mounting the filesystem:

`/RO`
: Read-only mount.

`/DONT_MAKE_CHECKSUMS`
: Do not create checksums on metadata structures.

`/DONT_CHECK_CHECKSUMS`
: Do not check validity of checksums on metadata structures. Checksums are used to protect against buggy ATA DMA controllers that damage the data. Each structure has a flag if checksum is valid and checksum itself. Specifying these flags can slightly increase performance (not too much because checksums are checked only the first time the buffer is loaded into memory).

5.3 Performance considerations

The filesystem was designed for maximal performance:

Metadata are stored in a special zone where large readahead is possible. The minimum buffer size the cache operates on is 32768 bytes, few of these are read ahead at a time. As a result — it took only 1 second to search all files on SATA-150 disk on filesystem with 3000 directories and 85000 files with cold cache (and 0.6 seconds with hot cache).

Fnodes are stored directly in directories rather than through external pointers (like in Ext2 and similar Unix filesystems). When user types "ls -al" there is no need to seek to inode for each listed file.

Lookup in the directory with hashing is faster than b-trees used in most common filesystems. File can be found in directory with 4M files with only three disk head seeks. (dnode has 2048 entries, fnode block can contain 6 or 7 fnodes and there are 16 fnode blocks depending on setting -fnode-size — so for two levels of dnodes you get 4M files).

Files' allocation is described in extents rather than lists of blocks like in ext2.

Files are almost never fragmented because the whole file is allocated when the filesystem is synced (or when the file is evicted from cache) not in time of write syscall. The only fragmented files are those that grow in time — for example logs (however, even they are fragmented no more than on Linux/Ext2). A test showed that copying the directory with source codes (with hot cache) was twice as fast as on Linux/Ext2.

The filesystem has only one thread syncing data to disks — the question is if it's an advantage or an disadvantage. I think that data write process is rather disk-bound than CPU-bound, so introducing more threads is not necessary. The thread does all operations except commit asynchronously. However, if usage shows the need for more threads, it is possible to add them with some trivial locking to prevent more threads allocate blocks in the same apage or write the same fnode simultaneously. Note that writes and reads to the cache are not blocked by the thread — any process can read, write or extend a file while its blocks are being allocated and written. The only operation blocked is truncate — file can't be truncated while it's being written.

5.4 Benchmarks

The following benchmarks are only temporary. SPADFS filesystem is work in progress and I still have to optimize it in few cases and better tune readahead and flushing dirty data. The kernel with SPADFS has also some internal debugging checks enabled that cause slow down. However, the benchmarks serve as a proof that this development seems promising.

For benchmarks I used repository with source code for various programs. It contained 1832 directories in a subtree and 48900 files. The total size of all files in repository was 814775546 bytes. I used two computers with identical motherboard, CPU Pentium 4 3GHz and 512MB memory. I extracted repository on SPADFS filesystem with default parameters (512B blocks, 32kB pages, 32kB clusters, 128kB cluster threshold) and on EXT2 filesystem with 4k blocks and filetype and sparse-super features. On SPADFS it took 831017kB space (kB is 1024 bytes), on EXT2 it took 932152kB. Extracting time for uncompressed tar file on SPADFS was 44s, for EXT2 46s (but note that on EXT2 it was extracted from different filesystem while on SPADFS it was extracted from itself).

Cache was invalidated before each test. Scanning time for a filename (`find . -name qwerty`) was 1s on SPADFS and 8s on EXT2. Reading all files (`find . -type f|xargs fgrep qawsedrf`) took 43s on SPADFS and 1:03s on EXT2. Deleting the tree took 18s on EXT2 and 1s on SPADFS.

In scanning and reading benchmark you can see that big readahead of metadata on SPADFS greatly improves performance. Deleting on SPADFS is so fast because the delete operation is delayed. The delete is actually only file scan while the files are marked in cache as to-be deleted. When it's the time to write them they are actually deleted.

6 Conclusion

This work outlines the crash counts as a replacement for journaling. Crash counts are implemented for new filesystem, however they could be added to most existing filesystems with some modifications of their structures. Additionally this work shows work in progress on new SPADFS filesystem focusing on consistency after crash, high performance and reasonable space consumption. Benchmarks look rather good, suggesting that the filesystem could outperform other common filesystems.

References

1. Rik van Riel: Wanted: Fundamental CS Research
   `http://www.surriel.com/research_wanted/`
2. Juan I. Santos Florido: Journal File Systems
   `http://www.linuxgazette.com/issue55/florido.html`
3. IBM: Open source JFS project Web site
   `http://oss.software.ibm.com/developerworks/opensource/jfs/`

4. SGI: Developer Central Open Source — Linux XFS
   http://oss.sgi.com/projects/xfs/
5. Stephen C. Tweedie: Journaling the ext2fs Filesystem
   http://www.osdever.net/documents/journal-design.pdf
6. Hans Reiser: ReiserFS 4
   http://www.namesys.com/v4/v4.html
7. Linux NTFS project: NTFS documentation
   http://linux-ntfs.sourceforge.net/ntfs/
8. Daniel Phillips: Explanation of Phase Tree
   http://seclists.org/lists/linux-kernel/2000/Jul/2863.html
9. Kirk McKusick: Soft Updates, Snapshots, and Back-ground Fsck
   http://www.mckusick.com/softdep/

Problém softwarových agentů a sémantický web*

(Rozšířený abstrakt)

Jiří Wiedermann

Ústav informatiky AV ČR
Pod Vodárenskou věží 2, 182 07 Praha 8, Česká republika
jiri.wiedermann@cs.cas.cz

Softwarový agent, ať už přijmeme jeho jakoukoliv rozumnou definici, je prapodivná entita: totiž, je (resp. měla by být) inteligentní, ale nemá tělo. Přebývá v internetu. Může taková věc vůbec existovat? Pro zastánce tzv. sémantického webu nepochybně ano. Ve svých představách si malují agenta, schopného najít pro nás dovolenou dle našich specifikací; jiní uvažují o agentech, schopných ve vzájemné spolupráci vybrat pro nemocnou babičku nemocnici odpovídající jejímu pojištění a kriteriím dětí o kvalitě takového zařízení, zařídit převoz babičky do nemocnice, a jaksi mimochodem zařídit změnu letenky z Ameriky do Evropy a další potřebné změny v našem denní agendě (viz např. [1]). Vyznavači tzv. vtělené (nebo také nové) inteligence v tom však vidí problém: věří (vědí?) totiž, že nutnou ingrediencí inteligence je nejen mozek, ale celé tělo, ve fyzickém smyslu. Dle vyznavačů této víry teprve tělo ve spolupráci se smysly a mozkem umožňuje poznání (sémantiky) světa, umožňuje imitační učení, pohyb agenta, jeho interakci s jinými agenty, atd. [2]. Jak se může softwarový agent obejít bez těla, existovat ve světě, ve kterém agenti nevnímají jeden druhého, který nejde ohmatat, kde se nedá rozhlédnout, pozorovat a napodobovat jeden druhého, ve světě, který je pouze stínem reálného světa, kterého sémantiku agent již nemůže rekonstruovat? Právě uvedené provokativní otázky – a zajisté by bylo možné generovat i další z podobného soudku – jsou motivovány představou softwarového agenta viděného očima lidí, kteří patřičně instruovaného agenta "vyšlou" na jeho misi do světa informací. Je taková představa reálná? Pokusíme se odpovědět z pohledu současných znalostí z oblasti nové umělé inteligence a interaktivních evolučních výpočtů, tj. zhruba v rámci teorie popsané v pracích [2], [3]. Na tomto základě přijdeme k následujícím závěrům:

- Scénář, ve kterém nepočítáme s průběžnými konzultacemi agenta s námi a instruujeme agenta "jednou provždy", je nerealistický, protože není možné předvídat všechny situace, ve kterých se agent může ocitnout (internet je proměnlivý), a tento nemůže předjímat naše rozhodnutí v podobných situacích. Proto je třeba uvažovat interaktivní režim práce agenta. Formálně to vede na tzv. evoluční interaktivní výpočty jejichž výpočetní síla je větší než turingovská (viz např. [3], [5]).
- Aby agent "fungoval" podobně jako lidský agent, musí oplývat podobnou inteligencí; tuto ale nemůže získat pouze ze světa formálních informací. Takový agent musí existovat na rozhraní dvou světů: našeho "lidského" (aby

* Práce byla vytvořena s částečnou podporou grantu č. 1ET100300419 NPV "Informační společnost"

si s námi rozuměl) a světa dat (aby se v internetu vyznal). Nestačí, jak se někteří autoři představují, postupná "evoluce" (rozuměj učení) agenta zcela "uvnitř" (sémantického) webu – tento svět je v porovnání s naším světem příliš jednoduchý, málo strukturovaný, postrádající kvalitu vnímatelnou lidskými smysly. V takovém světě nemůže agent jakkoliv zevrubným zkoumáním odhalit sémantiku "objektů" (rozuměj informací) v něm se nacházejících (podobnost se Searlovou Čínskou komnatou – viz. např. [2]). Kromě toho, není jasné, jak by mohlo fungovat imitační učení, které je základem pro řešení složitějších kognitivních úloh. Tuto problematiku budeme podrobněji diskutovat tím způsobem, že budeme hledat analogie a rozdíly mezi živými a softwarovými agenty.
– Jediná alespoň teoreticky možná schůdná cesta se jeví být cesta umělého vtěleného agenta, který se – podobně jako lidé – nejprve naučí žít současně v našem světe a současně "ovládat" pravidla, pomocí kterých jsou data organizovaná v internetu (znalost pouze jednoho z obou světů nestačí). Teprve poté může být "exkarnován" (tj. "vyňat" těla) a jako software vnořen do světa internetu. To kromě jiného vede k etickým otázkám, které také budou diskutovány v příspěvku. Výsledkem bude umělý myslící agent, který bude "integrován" do webovského prostředí. Alternativně lze na takového agenta nahlížet jako na umělou myslící entitu, pro kterou je web jakousi "protetickou" pomůckou, která umocňuje jeho intelektuální schopnosti. V takovém případě budeme moci hovořit i o "moudrém webu". Úvahy o možných principech konstrukce takového agenta lze nalézt např. v pracích [4], [6].

Závěrem lze konstatovat, že vize inteligentních softwarových agentů v prostředí sémantického webu je reálná, ale z naší analýzy vyplývá, že se jedná o složitější úkol, než by se mohlo zdát na první pohled.

Reference

1. Berners-Lee T., Hendler J., Lassila O.: The Semantic Web. Scientific American, May 2001.
2. Pfeifer R., Scheier Ch.: Understanding Intelligence. The MIT Press, Cambridge, Massachusetts, London, England, 1999, 697 s.
3. van Leeuwen J., Wiedermann J.: The Turing Machine Paradigm in Contemporary Computing. In: Mathematics Unlimited – 2001 and Beyond, Engquist B., Schmid W. (eds), Berlin, Springer 2001, 1139–1155.
4. Wiedermann J.: Towards Algorithmic Explanation of Mind Evolution and Functioning. In: Mathematical Foundation of Computer Science 1998, Proceedings, Brim L., Gruska J., Zlatuška J. (eds), LNCS, Vol. **1450**, Berlin, Springer 1998, 152–166.
5. Wiedermann J., van Leeuwen J.: The Emergent Computational Potential of Evolving Artificial Living Systems. Ai Communications, IOS Press, Vol. **15**, No. 4, 2002, 205–216.
6. Wiedermann J.: Mirror Neurons, Embodied Cognitive Agents and Imitation Learning. Computing and Informatics, Vol. **22**, 2003, 545–559.

Využití SIMD při implementaci neuronových sítí

Aleš Keprt, Martin Zlý

Katedra informatiky, FEI, VŠB - Technická Univerzita Ostrava
17. listopadu 15, 708 33, Ostrava-Poruba, Česká Republika
Ales@Keprt.cz, Martin.Zly.fei@vsb.cz

Abstrakt Vektorové instrukce SIMD (single instruction multiple data) umožňují několikanásobné zvýšení rychlosti programů díky paralelnímu zpracování většího množství dat současně. Tento příspěvek se zabývá možností využití těchto instrukcí při implementaci neuronových sítí - vícevrstvého perceptronu (MLP) a Hopfieldovy sítě. V obou případech se experimentálně potvrdilo, že nasazením vektorových instrukcí se výpočetní časy zkrátily. Vlastnosti vektorově implementovaných neuronových sítí se nesnažíme podchytit teoreticky, nýbrž experimentálně pomocí testování.

Keywords: neuronové sítě, SIMD, MMX, SSE, vícevrstvý perceptron, Hopfieldova síť

1 Úvod

Vektorové instrukce SIMD (single instruction multiple data), umožňující několikanásobné zvýšení rychlosti programů díky paralelnímu zpracování většího množství dat současně, jsou díky vysoké využitelnosti v multimediálních aplikacích dnes součástí instrukční sady všech běžných mikroprocesorů. Ve srovnání s jinými typy paralelizmu jsou poměrně snadno hardwarově implementovatelné a díky existenci jednotného trhu procesorů (Intel+AMD), kde se SIMD instrukce vyskytují v podobě MMX a SSE, jsou používány mnohým softwarem, včetně operačního systému Windows.

Princip SIMD znamená provádění jednotného kódu současně nad větším množstvím dat, je tedy vhodný jen pro určité typy aplikací. Naším cílem bylo využít SIMD instrukce při implementaci neuronových sítí za účelem zrychlení některých časově náročných výpočtů. Jako referenční model neuronové sítě jsme vzali vícevrstvý perceptron, v jehož aktivačním algoritmu lze vysledovat námi hledané opakující se výpočty velmi snadno. Poté, co jsme dosáhli úspěchu u tohoto typu sítě, zkoušeli jsme paralelizovat i další typy neuronových sítí. Všechny uvedené vlastnosti vektorově implementovaných neuronových sítí se nesnažíme podchytit teoreticky, nýbrž experimentálně pomocí testování.

2 Technologie SIMD

2.1 Úvod do SIMD

Neustálá snaha o zvyšování výkonu vedla nejprve k návrhu výlučně vektorových, nebo paralelních počítačů. Migrace těchto technologií do osobních počítačů byla

pak jen otázkou času. Vektorové zpracování dat je na osobních počítačích vhodné zejména pro multimediální aplikace, tedy výpočty v oblasti obrazu a zvuku.

Zejména zpracování obrazu, ať už v jakémkoliv smyslu (3D grafika, komprese, dekomprese, video, hry, editace bitmapových i vektorových obrázků, analýza obrazu, apod.) je výpočetně velmi náročné. Tento problém na jedné straně postupně odeznívá díky neustálému zrychlování výkonu mikroprocesorů, na druhé straně se však objevují další úlohy, které jsou na samotné zpracování obrazu navazovány a zabírají tak jistou část výkonu pro sebe (umělá inteligence, různé druhy matematických výpočtů apod.). Kromě samotného zvyšování frekvence a propustnosti datových sběrnic lze za nejpřínosnější technologii v této oblasti považovat instrukční sadu typu SIMD - „Single Instruction Multiple Data“.

SIMD stojí na principu vykonávání jednotného kódu nad větší množinou dat. Například namísto sečtení vektorů pomocí příkazu for

```
int a[8],b[8],c[8];
for(i=0; i<8; i++) {
  c[i] = a[i] + b[i];
}
```

můžeme s využitím SIMD stejně snadno sečíst celé vektory

```
int a[8],b[8],c[8];
c = a + b;
```

Ukázka kódu v pseudo-C jazyku samozřejmě zakrývá jeden významný fakt: Kód `c=a+b` je skutečně proveden jako jeden příkaz. Odtud také pochází označení „vektorové instrukce“. Datové položky, se kterými SIMD instrukce pracují, jsou vektory obsahující vždy daný počet menších datových buněk (na příkladu je to 8 proměnných typu int v jednom vektoru). Tato technologie se zdá býti dobře pochopitelnou programátory i poměrně snadno realizovatelnou výrobci mikroprocesorů.

Sčítání dvou vektorů (neboli polí) uvedené výše je jen knižní ukázkou použití SIMD. Praktické použití snadno nalezneme při již zmíněném zpracování obrázků, neboť ty se skládají z velkého množství pixelů, které jsou v paměti za sebou a tvoří tedy datový vektor. Operaci poloprůhledného překrytí dvou obrazů bychom tedy mohli symbolicky naznačit takto:

```
int image1[],image2[],image3[];
image3 = image1/2 + image2/2;
```

Samozřejmě nemůžeme očekávat, že sloučení dvou obrazů proběhne v jedné jednotce času bez ohledu na jejich velikost (dostali bychom tím nedeterministický výpočetní stroj). Skutečná implementace SIMD instrukcí obvykle zpracovává vektory jisté pevné délky. Například při délce vektoru 8 (dle prvního příkladu výše) se zpracování poloprůhlednosti dvou obrazů zrychlí 8×. SIMD instrukce totiž umožní provést dělení dvěma a sečtení dvou hodnot paralelně pro 8 takových dvojic.

V praxi máme tolik implementací SIMD instrukčních množin, kolik je typů procesorů, rozdíly mezi nimi však nejsou velké. Na mikroprocesorech Intel typu IA-32 (x86) se SIMD instrukce objevují pod obchodním označením MMX a SSE a umožňují pracovat s vektory o délce 8 resp. 16 bajtů. Také procesory AMD postupně přejímají instrukční sadu Intelu, avšak s jistým zpožděním. Navíc AMD nabízí vlastní vektorové instrukce 3D Now!.

2.2 MMX

Technologie MMX byla poprvé uvedena v mikroprocesoru Pentium MMX, později pak Pentium II a všech pozdějších modelech, včetně pozdějších mikroprocesorů AMD. Využívá mj. toho, že zdánlivě 32bitový procesor Pentium MMX má 64bitové jádro, které nelze běžnými instrukcemi plně využít. MMX nepřináší žádné nové 64bitové registry, namísto toho pouze rozšiřuje možnosti použití stávajících osmi 64bitových registrů FPU[1] jednotky o celočíselné vektorové výpočty. Nevýhodou tohoto řešení je nemožnost současné práce FPU a MMX jednotky. MMX umožňuje paralelní zpracování dat v podobě 8×8 bitů, 4×16 bitů, 2×32 bitů nebo 1×64 bitů, viz tabulka 1.

Datový typ	**MMX Registr**							
1×64 bitů	qword (64 bitů)							
2×32 bitů	dword 1				dword 0			
4×16 bitů	word 3		word 2		word 1		word 0	
8× 8 bitů	bajt 7	bajt 6	bajt 5	bajt 4	bajt 3	bajt 2	bajt 1	bajt 0

Tab. 1. Možná datová mapování MMX registrů.

MMX instrukce umožňují s těmito vektory provádět nejen běžné aritmetické operace (po složkách), ale i některé speciální výpočetní operace, které v základní instrukční sadě nenajdeme. Tyto speciální výpočetní instrukce vznikly na přímou „objednávku“ reálných aplikací a umožňují efektivně zrychlit některé multimediální algoritmy i více než desetinásobně (přestože vektor obsahuje jen 4 prvky). Důležitá je také schopnost efektivně balit (pack) a rozbalovat (unpack) data do/z vektorů.

2.3 SSE

Procesor Intel Pentium III přinesl dvě další sady SIMD instrukcí. Bylo to MMX+, rozšiřující možnosti celočíselných výpočtů pomocí MMX, a nová sada SSE pro výpočty v plovoucí řádové čárce. Všechny tyto instrukce opět najdeme i na novějších modelech AMD (a dalších).

SSE přináší také 8 nových 128bitových registrů XMM0-XMM7, což přináší několik výhod: máme dvojnásobnou velikost vektorů (Pentium III má již 128bitové jádro), můžeme používat FPU i SIMD instrukce současně (neboť SSE neblokuje FPU registry) a pokud je to výhodnější, můžeme používat MMX i SSE

[1] FPU = Floating Point Unit, jednotka pro výpočty v plovoucí řádové čárce.

současně (mají samostatné registry). SSE však neumožňuje provádět celočíselné výpočty nad 128bitovými vektory, zná totiž jen jediný datový typ a to je 4bajtový single (float v C/C++). S registry lze také pracovat vcelku (to je však vhodné jen pro kopírování paměti apod.). Viz tabulka 2.

Datový typ	**XMM Registr**			
1× 128 bitů	m128 (16bajtů, tj. 128 bitů)			
4× 32 bitů	single/float 3	single/float 2	single/float 1	single/float 0

Tab. 2. Možná datová mapování XMM registrů v SSE.

Zatímco MMX bylo pro svou celočíselnou povahu obtížně nasaditelné v obecných výpočetních aplikacích, SSE má již podstatně širší možnosti použití (například pro simulaci neuronových sítí).

2.4 SSE2 a další SIMD rozšíření

Na mikroprocesorech typu IA-32 (x86) najdeme krom tří zmíněných (MMX, MMX+ a SSE) ještě několik dalších SIMD instrukčních sad.

Zásadním krokem vpřed je sada SSE2, která v podstatě slučuje MMX a SSE a přidává řadu dalších speciálních vektorových instrukcí pro zrychlení specifických multimediálních algoritmů. SSE2 je přitom obecně použitelná sada, neboť umožňuje na 128bitových XMM registrech provádět velmi rychle výpočty nad různými datovými typy. Zajímavá pro nás je zejména podpora 64bitových double čísel v plovoucí řádové čárce. Nevýhodou SSE2 pak je především to, že jej podporuje jen procesor Pentium 4.

Další SIMD instrukce najdeme na běžných počítačích pod názvy 3D Now! a 3D Now!+ (AMD, VIA), SSE3 (Intel) a EMMX (VIA). 3D Now!(+) je velmi známá sada a je také podporována přímo knihovnami Windows, avšak bez podpory Intelu je její využití mimo operační systém poměrně malé. SSE3 a EMMX se prakticky nepoužívají.

3 Použití SIMD

3.1 Vývojové nástroje

Existuje samozřejmě mnoho možností, jak vyvíjet software využívající SIMD. Nejjednodušší je situace v assembleru, kde je možno používat tyto instrukce bez omezení. Bohužel, vyššími programovacími jazyky tyto instrukce přímo podporovány nejsou (alespoň jsme o žádném takovém neslyšeli). Základním vývojovým nástrojem se tak stává inline assembler, což je rozšíření umožňující do vyššího jazyka (C++, Fortran, Pascal, ...) vkládat části podprogramů v assembleru. Výhodou tohoto řešení je dobrá čitelnost a přehlednost kódu.

Intel nabízí také knihovnu, která obsahuje intrinsic funkce[2] pro C++. Tato knihovna pak umožňuje psát v C++ plně efektivní SIMD kód bez použití assembleru. Tento kód je sice poněkud hůře čitelný, avšak umožňuje nám použít paradigma vyššího jazyka, což obvykle vede k lepšímu výsledku, neboť ne každý dnes už zná assembler. Intel C++ Compiler také provádí optimalizaci SIMD kódu - dělá za nás alokaci registrů apod. (Poznámka: Intel samozřejmě nenabízí tuto knihovnu pro instrukční sady procesorů AMD nebo VIA.)

3.2 Složený datový formát

Pro práci s vektory obsahuje MMX i SSE řadu instrukcí pro zabalení a rozbalení hodnot do vektoru. Tyto instrukce umožňují provádět různé přesuny a výměny pořadí mezi složkami vektorů, změnu datových formátů i přenos dat mezi MMX/SSE a běžným nevektorovým uložením v registrech mikroprocesoru nebo v paměti. Díky schopnosti snadno a velmi rychle přeskládávat data ve velkých blocích, instrukce MMX se hodí i pro některé úlohy datové konverze (změna formátu obrázku apod.).

3.3 Podmíněné vykonávání

Podmíněné vykonávání kódu je ve vyšších strukturovaných jazycích obvykle řešeno větvením příkazem `if` nebo podobným. Při paralelním zpracování není takový způsob možný, neboť každá z paralelně zpracovávaných hodnot může vést k jinému výsledku vyhodnocení podmínky a vyžadovat tak vykonání jiné větve kódu, což by odporovalo principu SIMD (tj. „Single Instruction…“).

MMX obsahuje vektorové porovnávací instrukce, které (přesně ve stylu assembleru vlastním) výsledek testování podmínky ukládají ve formě příznaků zpět do MMX registrů tak, aby byla zachována struktura vektoru. Větvení kódu pak můžeme nahradit aritmeticko-logickými operacemi, které mohou běžet paralelně.

Uveďme pro příklad výpočet maxima ze dvou hodnot `c = max(a,b);`. To můžeme provést jednoduchým algoritmem:

```
c = a;
if(b > a) c = b;
```

Tento kód však nemůže běžet paralelně. Potřebujeme-li tento výpočet provést uvnitř jiného MMX kódu, musíme jej provést sériově a navíc je zde nutnost datových konverzí (rozbalit vektory, otestovat podmínku, provést vnitřní kód každé větve sériově, zapamatovat si mezivýsledky, na závěr opět zabalit mezivýsledky do vektoru).

Pomocí vektorových porovnávacích instrukcí lze kód přepsat do následujícího tvaru:

[2] Intrinsic funkce = vlastní funkce. Obvykle se nepřekládá, značí vestavěné funkce vyššího programovacího jazyka, při jejichž použití nedochází k „volání“, ale přímému vykonání kódu. Hodí se jen pro malé funkce přímo podporované mikroprocesorem.

```
x = (b > a);
c = (b AND x) OR (a AND NOT x);
```

Tento způsob provádění podmíněných výpočtů může běžet paralelně a je v praxi výrazně rychlejší.

3.4 Uspořádání dat v paměti

Pro efektivní využití výhod SIMD je vhodné uzpůsobit tomu také datové struktury. Pokud například máme 10000 bodů v prostoru, jejich souřadnice můžeme v paměti uchovávat v poli třírozměrných vektorů.

```
struct Point {
  float x,y,z;
};
Point data[10000];
```

Tento způsob uložení dat v paměti je běžný a ve vyšších programovacích jazycích se s ním dobře pracuje. Nazýváme jej obvykle AOS (Array Of Structures - pole struktur).

Potřebujeme-li počítat například skalární součiny nad těmito souřadnicemi, musíme složitě vybírat jednotlivé složky vektorů z našeho pole. Tento problém můžeme eliminovat tak, že použijeme obrácený způsob uložení dat v paměti, označovaný SOA (Structure Of Arrays - struktura polí).

```
struct {
  float x[10000];
  float y[10000];
  float z[10000];
} data;
```

Tento způsob organizace dat v paměti je funkčně identický s AOS, avšak je v praxi méně obvyklý[3].

Nyní již můžeme načítat z paměti rovnou celé SIMD vektory (128 bitů, tedy 4 float hodnoty v případě SSE), to přináší výraznou úsporu času.

SSE navíc vyžaduje data zarovnaná na fyzickou hranici 16 bajtů, aby bylo možno jednu XMM buňku načíst/uložit v jednom cyklu. Toto omezení sice pro základní sadu instrukcí procesorů x86 neplatí, protože případné kolize řeší jednotka řízení přístupu do paměti přímo uvnitř mikroprocesoru, ale nezarovnané čtení je 2× a zápis dokonce 4× pomalejší než zarovnané čtení/zápis.

4 Implementace neuronových sítí s využitím SIMD

Implementace neuronových sítí většinou vyžaduje práci s desetinnými čísly, proto se jako první nabízí využití SSE. Prvním krokem v implementaci jakékoliv neuronové sítě je vždy vytvoření vhodného modelu uložení dat v paměti (viz kap. 3.4). Jelikož přínos SIMD je pro různé typy neuronových sítí často velmi podobný, zvolili jsme si jako referenční model vícevrstvý perceptron.

[3] z pochopitelných důvodů, které zde není třeba diskutovat

4.1 Vícevrstvý perceptron

Vícevrstvý perceptron, bez ohledu na způsob učení, pracuje po vrstvách tak, že na vstup každého neuronu jsou přivedeny výstupy všech neuronů předešlé vrstvy. Vnitřní potenciál v neuronu j ve vrstvě l je sumou výstupních signálů y neuronů předchozí vrstvy $l-1$ násobených příslušnými váhovými koeficienty w. Symbolicky (notace dle Oravce [2])

$$v_j^{(l)} = \sum_{i=0}^{p} w_{ji}^{(l)} y_i^{(l-1)}$$

Při použití SOA (viz kap.3.4) lze tento výpočet realizovat velmi efektivně pomocí MMX či SSE. Hodnoty $w_{ji}^{(l)}$ a y_i uložíme do dvou polí, aby hodnota i byla adresovacím indexem těchto polí. Zajímavé je, že tento způsob uložení dat v paměti tentokrát ani neodporuje přirozenému objektově orientovanému návrhu. Násobení pak můžeme provádět po čtveřicích hodnot a výpočet výše uvedeného vzorce probíhá v těchto krocích:

1. Vynuluj XMM2.
2. Načti do registru XMM0 čtveřici hodnot z pole w.
3. Načti do registru XMM1 čtveřici hodnot z pole y.
4. Vynásob XMM0*XMM1, výsledek přičti k XMM2.
5. Opakuj kroky 2 – 4 pro všechny čtveřice.
6. Sečti všechny složky XMM2 → výsledek.

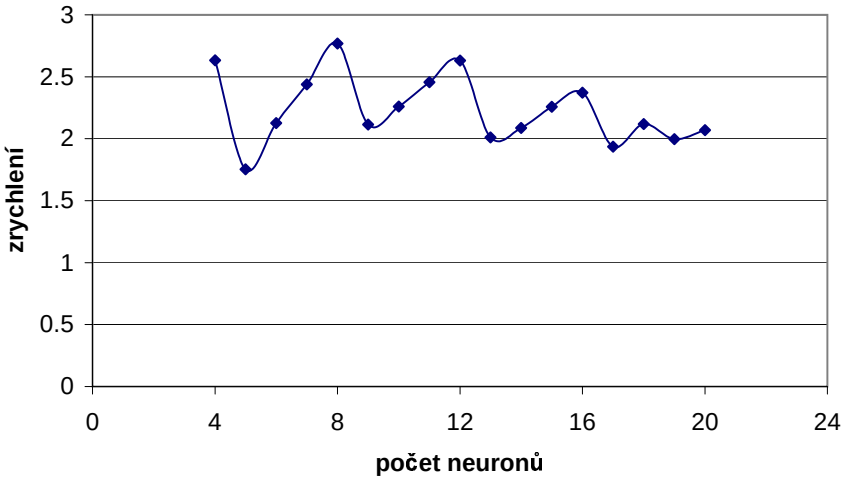
Poznámka: V kroku 4 uvedeného algoritmu chceme provádět sumarizaci dílčích součinů. To však v SSE nelze provést přímo, proto zde jen sumarizujeme vektory po složkách. V posledním kroku algoritmu je třeba sečíst všechny čtyři složky XMM2, což je pomalé (5 instrukcí), provádíme to však jen jednou na konci výpočtu. V případě použití MMX+ můžeme využít k součtu celočíselných složek vektoru instrukci pro výpočet aritmetického průměru následovanou bitovým posunem doleva. Celočíselná povaha MMX se však pro tento výpočet nehodí – celá čísla sice lze považovat za desetinná čísla s pevnou řádovou čárkou, avšak výpočty jsou pak dosti nepřesné (32 bitů je málo).

Testy

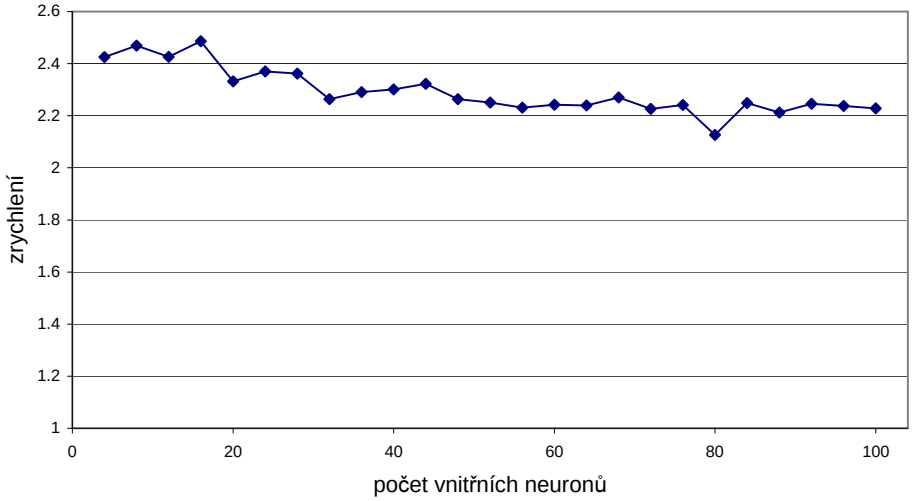
Algoritmus je vhodné vyzkoušet v praxi a změřit jeho přínos přímo v podobě úspory výpočetního času. Pro tyto testy není podstatné, na co je neuronová síť využita, ale poměr zrychlení paralelního výpočtu vůči sériovému. Výsledky ukazuje obrázek 1.

Na obrázku je vidět, že zrychlení se pohybuje okolo 2×, což je velmi dobrá hodnota (SSE verze je tedy v průměru 2× rychlejší). Největšího zrychlení však dosáhneme, když je počet neuronů násobkem čtyř a je tak zaplněn celý XMM vektor.

Se zvětšujícím se počtem neuronů se postupně eliminuje zpomalení vzniklé necelým využitím posledního vektoru (když počet neuronů není násobkem čtyř). Proto jsme se dál omezili jen na počty neuronů v násobcích čtyř. Výsledek měření je na obrázku 2. Z grafu je vidět, že zrychlení je přibližně 2.3×.



Obr. 1. Zrychlení výpočtu vícevrstvé sítě (konfigurace sítě 64-X-8).



Obr. 2. Zrychlení výpočtu vícevrstvé sítě pro větší počty vnitřních neuronů (konfigurace 64-X-8, X je násobkem čtyř).

4.2 Hopfieldova síť

Dalším zajímavým typem neuronové sítě je síť Hopfieldova. Klasickou binární Hopfieldovu síť jsme také zvolili pro otestování možností celočíselných MMX instrukcí.

počet neuronů	poměr zrychlení
12×12	1.60
48×48	1.60
100×100	1.62
400×400	1.56
1000×1000	1.54

Tab. 3. Poměr zrychlení u Hopfieldovy sítě.

Paralelní výpočet zde použijeme ke stanovení výstupu jednotlivých neuronů. Výstup neuronu je stanoven jako binární výsledek porovnání vnitřního potenciálu s danou prahovou hodnotou (obdoba funkce signum).

Tato operace je v SIMD poměrně složitě realizovatelná, neboť musíme dokázat pomocí logických operací nahradit ne zcela triviální větvení (podrobnosti jsou nad rámec tohoto textu). Algoritmus jsme implementovali dle poznatků uvedených v kap. 3.3. Výsledky měření ukazuje tabulka 3: Dosáhli jsme zrychlení přibližně 1.6×, přitom u velmi velkých sítí poměr zrychlení lehce klesá. I tento komplikovaně paralelizovatelný výpočet se tedy při použití SIMD instrukcí zrychlí.

4.3 Další typy neuronových sítí

Obecně lze říci, že prakticky každý typ neuronové sítě může být implementován pomocí SIMD. Během výpočtu totiž vždy opakujeme tytéž nebo jen velmi málo odlišné operace pro každý neuron nebo pro určité skupiny neuronů stejného typu.

5 Další poznámky

Pro úspěch SIMD implementace je důležitá především velikost implementovaných sítí. Při malém počtu neuronů je zrychlení poměrně nevýrazné a teprve při větších počtech neuronů si můžeme být jisti významným zrychlením.

Požadavek, aby počty spolu počítaných neuronů byly násobkem čtyř, není třeba klást. Ukázalo se totiž, že nepoužité buňky ve vektorech nezpůsobují žádné výpočetní problémy, stačí je na začátku nastavit na nulu (nebo jedničku, dle typu algoritmu). V tomto ohledu je nejvhodnější vždy alokovat paměť pro neurony v počtu násobku čtyř. V samotném výpočtu se pak na to, že skutečný počet neuronů je menší, již nemusíme ohlížet. (Tento postup je běžný při práci s grafikou, zde jej pouze zobecníme na složitější objekty.)

Instrukce SSE vyžadují, aby vektory byly v paměti fyzicky zarovnány na hranice 128 bitů (což je fyzická šířka datové sběrnice, velikost slova mikroprocesoru Pentium III). Praxe ukázala, že tento požadavek programování nijak nekomplikuje. Překladače C++ s podporou SSE dokáží toto pravidlo hlídat automaticky.

6 Shrnutí

Představili jsme zde myšlenku, jak vhodně využít SIMD instrukce běžných mikroprocesorů Intel/AMD k výraznému zrychlení výpočtu umělých neuronových sítí. Tyto vektorové verze algoritmů dávají identické výsledky s algoritmy původními v až několikanásobně kratším čase.

Námi zvolený referenční model vykazuje přibližně trojnásobné zrychlení při použití SSE instrukcí. Také další typy neuronových sítí, které jsme v rámci naší práce zkoušeli implementovat, vykazovaly zrychlení, zhruba dvojnásobné. Pro implementaci neuronových sítí diskrétního typu je vhodné použít MMX, pro všechny spojité sítě je vhodnější SSE. Přestože MMX může být bez problémů použito i pro neceločíselné výpočty[4], použití SSE se ukazuje jako lepší varianta.

Výpočty v oblasti grafiky a geometrie někdy dosahují při použití SSE deseti i vícenásobného zrychlení. Podobných výsledků jsme u neuronových sítí nedosáhli. Důvodem je skutečnost, že SSE obsahuje řadu velmi specifických instrukcí, které jsou šity na míru grafickým výpočtům, zatímco pro neuronové sítě použijeme jen některé obecné SSE instrukce.

Do budoucna může být zajímavé zkusit využít rozšířených možností SSE2 a SSE3. Tyto nové instrukční sady by mohly přinést ještě až znovu dvojnásobné zrychlení oproti současnému SIMD kódu, může to však být za cenu snížení výpočetní přesnosti (větší numerické chyby). Limitující je také skutečnosti, že tyto nové instrukce najdeme zatím jen na nejnovějších mikroprocesorech Pentium 4.

Reference

1. *Intel Architecture Software Developer's Manual Volume I-III.* Intel, 2003. `http://www.intel.com/`
2. M.Oravec: *Neuronové siete pre extrakciu príznakov, kompresiu a rozpoznávanie obrazu.* Habilitační práce, Katedra telekomunikácií FEI STU, Bratislava, 2001.
3. S.Tommesani: Quexal Visual Dev Tool for MMX/iSSE. `http://www.tommesani.com/`
4. E.Volná: *Neuronové sítě a genetické algoritmy.* Ostravská univerzita v Ostravě, Ostrava, 1998.
5. I.Vondák: *Umělá inteligence a neuronové sítě.* Vysoká škola báňská - Technická univerzita Ostrava, Ostrava, 2002.
6. I.Zelinka: *Umělá inteligence I. Neuronové sítě a genetické algoritmy.* Vysoké učení technické v Brně, Brno, 1998.

[4] Pevná řádová čárka neznamená, že není možno počítat s desetinnými čísly.