Erschienen in: Proceedings, 2002 IEEE International Conference on Multimedia and Expo : August 26 - 29, 2002, Swiss Federal Institute of Technology, Lausanne, Switzerland; vol. 1. - Piscataway, NJ : IEEE Operations Center, 2002. - S. 801-804. - ISBN 0-7803-7304-9

FAST LIST VITERBI DECODING AND APPLICATION FOR SOURCE-CHANNEL CODING OF IMAGES

Martin Röder, Raouf Hamzaoui

University of Konstanz, Department of Computer and Information Science
78457 Konstanz
martin.roeder@uni-konstanz.de, hamzaoui@fmi.uni-konstanz.de

ABSTRACT

A list Viterbi algorithm (LVA) finds the n best paths in a trellis. We propose a new implementation of the tree-trellis LVA. Instead of storing all paths in a single sorted list, we show that it is more efficient to use several lists, where all paths of the same list have the same metric. For an integer metric, both the time and space complexity of our implementation are linear in n. Experimental results show that our implementation is much faster than all previous LVAs. This allows us to consider a large number of paths in acceptable time, which significantly improves the performance of a popular progressive source-channel coding system that protects embedded data with a concatenation of an outer error detecting code and an inner error correcting convolutional code.

1. INTRODUCTION

Sherwood and Zeger [1] devised a powerful systems for protecting images against errors in a memoryless noisy channel. The system uses a concatenation of an outer cyclic redundancy-check (CRC) code for error detection and an inner rate-compatible punctured convolutional (RCPC) code for error correction. The image is compressed with an embedded wavelet coder [2]. Then the output bitstream is divided into consecutive packets of fi ed length. To each packet, a CRC code is appended, and the resulting blocks are encoded with equal error protection by an RCPC coder. The receiver uses an LVA to fin the most likely (best) decoding solution (path) for a received packet. This path is checked by computing the CRC checksum bits. If an error is detected, the LVA is used again to fin the next best path, which is also checked for errors. This process is repeated until the CRC test is passed, or a maximum number of paths is reached, in which case, called incomplete decoding, the decoding is stopped, and the image is reconstructed using only the correctly decoded packets.

The performance of this system depends on the maximum number of candidate paths for one packet. Increasing the maximum number of candidates decreases the probability of incomplete decoding and improves the average reconstruction fidelit . However, the number of candidates is limited by the time complexity of the LVA. On the other hand, the probability that the CRC test fails to detect a packet error increases monotonically with the number of candidates. This problem can be handled by increasing the length of the CRC codeword. But then, the RCPC code rate has to be increased as well to keep the transmission rate fi ed.

The major contribution of this paper is an efficien implementation of the tree-trellis LVA [3]. Instead of storing all n paths in a single sorted list, we show that it is advantageous to use several lists, where all paths of the same list have the same metric. In particular, we prove that for the Hamming metric H the worst-case time complexity of our implementation is $O(l(2^m + n) + H(p_n(\emptyset)))$, where l is the length of the original packet, m is the memory order of the convolutional coder, and $p_n(\emptyset)$ is the nth path of the zero codeword. Since $H(p_n(\emptyset)) \leq lr$, where r is the rate of the mother code, the time complexity of our implementation is linear in n. This compares favorably with the original algorithm [3] whose average complexity is $O(l(2^m + n^2 + n))$ and with other state-of-the-art LVAs [4, 5]. We also extend this result to arbitrary integer metrics, which makes our algorithm useful in soft decision decoding. Experimental results for the binary symmetric channel (BSC) showed that our algorithm was also the fastest in practice. Finally, we determined for both the original search depth of 100 paths [1] and our proposed search depth of 10,000 paths the optimal combination of the CRC codeword length and the RCPC code rate and showed that the higher search depth significantl improves the rate-distortion performance of the system at several bit error rates.

2. TERMINOLOGY AND PREVIOUS WORK

A binary rate $1/q$ convolutional encoder outputs q bits for each input bit. This group of q output bits, called *output frame* of the encoder, is transmitted over the channel and becomes an input frame of the decoder or simply a *code frame*. The encoder has a memory order of m, which leads to 2^m possible *states* $z = 0, \ldots, 2^m - 1$. When an input bit is read by the encoder, it is shifted into the memory. This leads to a *state transition* $(z_1 \to z_2)$. The output frame associated to state transition $(z_1 \to z_2)$ is denoted by $r(z_1 \to z_2)$. Let M be a bit metric. Then the *transition metric* between the output frame $r(z_1 \to z_2) = (y_1, \ldots, y_q)$ and the code frame $c = (c_1, \ldots, c_q)$ is $M(r(z_1 \to z_2), c) = \sum_{i=1}^{q} M(y_i, c_i)$. Let l be the number of received code frames. For $t = 0, 1, \ldots, l$, we denote by $z(t)$ the state at stage t. A *path* $p = (z(0) \to z(1)), \ldots, (z(l-1) \to z(l))$ is a sequence of state transitions starting and ending at a valid state. We suppose that both the starting state and the fina state are the zero state. Let $c(t)$ be the code frame received at stage t. Then, $M(p) = \sum_{t=1}^{l} M(r(z(t-1) \to z(t)), c(t))$ is the *path metric* of path p. An LVA find the n paths with the lowest metric.

The best previous LVAs are the parallel LVA (PLVA) [4], the serial LVA (SLVA) [4], which was later improved in [5], and the tree trellis algorithm (TTA) [3]. Although the time complexity of the PLVA is linear in n, the algorithm is not suitable for most applications because on one hand, the multiplicative constant in its $O(n)$ time complexity is large and on the other hand, it determines the n best paths simultaneously. The other algorithms

fin the best paths successively. The TTA consists of a forward pass and n backward passes. The forward pass stores the two best path metrics for each state $z_j(t)$. The lower path metric is denoted by $M_1(z_j(t))$ and the higher path metric by $M_2(z_j(t))$. Furthermore, a backtrace pointer $v(z_j(t))$ to the predecessor state of the path with the lower metric must be stored for each state. A backward pass uses a stack of paths which stores for each element S: $S.m$, the path metric, S_p, the path this path branches from, $S.t$, the stage at which this path branches from $S.p$, $S.i$, the state at which this path branches from $S.p$, and $S.m_p$, the partial path metric from $S.t$ to the end of the trellis. The stack has to be sorted by ascending values $S.m$. Furthermore, the decoded data of all already found paths must be stored to reconstruct the common partial path from the branching point to the end of the trellis. At the beginning of the firs backward pass, the stack is initialized with a single element consisting of $S.m = M_1(z_0(l))$ (the metric of the best path), $S.p = 0$, $S.t = l$, $S.i = 0$, and $S.m_p = 0$. For each backward pass, the following steps are done:

1. The firs entry is read and deleted from the stack. The backward pass starts at stage $S.t$ and state $S.i$, thus we set $t = S.t$ and $i = S.i$. The new path and the path $S.p$ share the common path segment from $S.t$ to the end of the trellis, therefore this path segment can simply be copied from the stored path $S.p$. The running partial path metric m is set to $m = S.m_p$.
2. In the firs backward pass, we follow the backtrace pointer, $j = v(z_i(t))$. Otherwise, the new path branches from $S.p$ at stage $S.t$ and state $S.i$. Therefore we have to take the predecessor state the backtrace pointer does not point to for the firs state transition, thus we set j such that the state transition $(z_j \to z_i)$ is possible, but $j \neq v(z_i(t))$.
3. m increases by the metric of the state transition, so we set $m = m + M_1(z_i(t)) - M_1(z_j(t-1))$. The input bit to the encoder associated with state transition $(z_j \to z_i)$ is prepended to the already decoded data of the new path and then the state transition is finishe by setting $t = t - 1$ and $i = j$.
4. At the new state, the locally second best path must be inserted into the stack. The metric of this path is the sum of $M_2(z_i(t))$, which was stored during the forward pass, and m. Therefore, with k being the number of the current path (which is also the number of the current backward pass), we insert a new element S with $S.m = M_2(z_i(t)) + m$, $S.p = k$, $S.t = t$, $S.i = i$ and $S.m_p = m$ at the correct position into the sorted stack.
5. All following state transitions follow the backtrace pointers; thus we set $j = v(z_i(t))$.
6. We repeat Steps 3 to 5 until stage 0 is reached.

Because the stack is sorted with the path metric, the next best path is at the top of the stack after each backward pass. To reduce memory requirements and avoid unnecessary stack insertions, the stack size can be limited to $n-1$ elements because all elements inserted behind element $n-1$ never reach the top of the stack.

The time complexity of the forward pass is $O(2^m l)$. In a backward pass, the most time consuming part is the sorting of the stack because we need to insert one element for each visited state. All other steps can be done in constant time. The stack is commonly implemented as a linked list. To insert a new element in a list of n elements, $n/2$ comparisons are needed in average. If we assume that a new path branches from an existing path at half of the trellis in average, we have to visit $l/2$ states in each backward pass, which leads to $l/2$ stack insertions and $l/2$ state transitions per backward pass. Thus, for n paths, the average time complexity is $O(l(n^2+n))$. On the other hand, the space complexity of the algorithm is $O(2^m l + nl + n)$ [6].

3. IMPROVEMENTS TO THE TTA

To reduce the time complexity of the TTA, we propose to maintain the sorted stack with a red-black-tree, which guarantees a logarithmic time complexity for insertion and search operations. This reduces the time complexity of the n backward passes to $O(l(n + \log_2((n-1)!)))$. However, a better approach is to replace the sorted list by multiple lists such that all paths in one list have the same metric. Those lists do not have to be sorted, and an insertion operation is done by simply appending the new stack entry into the list corresponding to its metric, which is a constant time operation. The problem is now to determine the number of lists needed. The following proposition gives an upper bound on this number.

Proposition 1 *Suppose that M is an integer metric. Let w be the received packet, n be the maximum number of candidate paths for w, and $M(p_j(w))$ be the metric of the jth path of w. Then at most $M(p_n(w)) - M(p_1(w)) + 1$ lists are needed.*

Proof. Since the TTA is an LVA, it find paths $p_j(w)$ such that $M(p_1(w)) \leq M(p_j(w)) \leq M(p_n(w))$. We need at most one list for each of the $M(p_n(w)) - M(p_1(w)) + 1$ integer values between $M(p_1(w))$ and $M(p_n(w))$. ∎

The problem is to fin $M(p_n(w)) - M(p_1(w))$ or at least an upper bound for it that is independent of the received packet w.

Proposition 2 *Let v be a valid codeword and w be an arbitrary packet of the same length. Let H be the Hamming metric. Then for all $j \geq 1$*

$$H(p_j(w)) - H(p_1(w)) \leq H(p_j(v)) - H(p_1(v)). \qquad (1)$$

Proof. We give here the main idea of the proof. For the full proof see [6]. Consider a valid codeword v. If one bit of v is changed, then $H(p_j(v))$ either increases or decreases by 1 for all paths p_j. $H(p_1(v))$ must increase by 1, because it cannot be negative. Thus $H(p_j(v)) - H(p_1(v))$ either decreases by 2 or remains unchanged. Any packet w can be generated from a valid codeword by changing a number of bits and increasing the Hamming metric between the packet and the original valid codeword by 1 at each bit change. When generating a packet this way, $H(p_j(v)) - H(p_1(v))$ cannot increase at any bit change. ∎

A packet consisting only of zero bits is always a valid codeword and is called a zero codeword $\emptyset$. The Hamming metric of the best path of a valid codeword is always 0. Thus, we can use $H(p_n(\emptyset)) + 1$ as an upper bound for the number of lists, where $\emptyset$ has the same length as the packets to decode. This motivates the following multiple-list tree trellis algorithm (mL-TTA).

List allocation: Set $m_{\max} = H(p_n(\emptyset))$ and allocate $m_{\max} + 1$ lists $L_0, \ldots, L_{m_{\max}}$.

Forward pass: the same as in the original TTA.

Stack initialization: The element for the best path is initialized with the same values as in the original TTA and inserted as the firs element of L_0. The metric of this path, $m_{\min}$, is the smallest path metric, i.e., $m_{\min} = H(p_1(w)) = H_1(z_0(l))$. We also need two pointers f_s and f_e to the firs and last non-empty list. These are initialized as $f_s = 0$ and $f_e = m_{\max}$.

Backward pass: The backward pass is the same as in the original TTA, but with modifie operations on the path stack.

Stack insertion: If $S.m > m_{\max}$, no insertion is done. Otherwise, the new element is appended to the list $L_{S.m-m_{\min}}$. If the stack contains more than n paths now, the last element from the stack can be removed. This is the last element from the last non-empty list indicated by f_e. Since the stack was altered, we must firs update f_e by setting $f_e = \max j$ where $j \leq f_e$ and L_j is not empty. Then we remove the last element from L_{f_e}. We must also update $m_{\max}$ by $m_{\max} = f_e + m_{\min}$ because there are already n paths in the stack, and paths inserted after the n-th path would never be decoded.

Reading and removing the top of stack element: The top of stack element is the firs element of the firs non-empty list denoted by f_s. Since the stack was altered, f_s must be updated firs by $f_s = \min j$ where $j \geq f_s$ and L_j is not empty. Then the firs element of L_{f_s} is read and removed from L_{f_s}.

To insert a path into the stack, we append it to the list indicated by its metric, which is a constant time operation. If the stack consists of more than n elements after insertion, the last non-empty list must be found to remove the last element from it. For this, only lists L_j with $j \leq f_e$ must be checked since no element could have been inserted in lists with $j > f_e$ because their metric would be greater than $m_{\max}$. Therefore, during *all* n backward passes needed to fin n paths, the array of lists will be completely traversed at most *once*. Finding the firs non-empty list for removing the top of stack element is similar. No path can be inserted in lists L_j with $j < f_s$ because such a path would have a lower metric than the current one, breaking the path order condition for LVAs. Furthermore, the firs non-empty list cannot be behind the last non-empty list. Thus, for both search operations, the array of lists will at most be completely traversed once during all n backward passes.

This leads to a worst-case time complexity of $O(nl + H(p_n(\emptyset)))$ for n backward passes and an overall space complexity of $O(2^m l + nl + n + H(p_n(\emptyset)))$. The worst-case time complexity of the mL-TTA is much lower than the average time complexity of the single list TTA because $H(p_n(\emptyset))$ increases very slowly. Note that $H(p_n(\emptyset))$ is bounded by rl.

To compute $H(p_n(\emptyset))$, we start with an approximation, e.g., $m = 1$. Then we set $m_{\max} = m$ and allocate m lists which gives all paths $p_j(\emptyset)$ with $H(p_j(\emptyset)) \leq m$. We determine $p_1(\emptyset), p_2(\emptyset), \ldots$ until we either fin $p_n(\emptyset)$ or the path stack is empty, in which case, we double m and allocate lists and search paths again until we fin $p_n(\emptyset)$. In this way, $\lceil \log_2 H(p_n(\emptyset)) + 1 \rceil$ executions of the mL-TTA give $H(p_n(\emptyset))$.

The Hamming metric is useful with hard decision decoding. In practice, it is often better to use soft decision decoding.

Proposition 3 *Let $M(x, y)$ be an integer symbol metric, A be the channel alphabet, $x_1, x_2 \in A$ and $y_1, y_2 \in \{0, 1\}$. If there exists $d > 0$ with*

$$|M(x_1, y_1) - M(x_2, y_2)| \leq d \tag{2}$$

for all x_1, x_2, y_1, y_2, then for all packets w and all $j \geq 1$

$$M(p_j(w)) - M(p_1(w)) \leq d(H(p_j(\emptyset)) - H(p_1(\emptyset))) \tag{3}$$

Proof. The Hamming metric of all paths changes by 1 if one bit of the packet is changed. With a metric satisfying (2), the metric of all paths can at most change by d, so all path metrics can at most be d times as high as with the Hamming metric. ∎

To use soft decision decoding with the mL-TTA, we fin the lowest d that satisfie (2) and multiply it with the number of needed lists. We also must use integer values for the symbol metric. These integer values can be calculated from a floatin point metric by scaling and rounding. Thus, also for arbitrary integer metrics, the time complexity of our algorithm remains linear in n.

4. APPLICATION TO IMAGE TRANSMISSION

By increasing the path search depth in the source-channel coding system of [1], we decrease the probability of incomplete decoding, but we increase the probability that the CRC fails to detect an error. If we f x the search depth and the bit error rate (BER) of the BSC, a rate-based optimal performance of the system can be obtained by maximizing the expected number of correctly received source bits over all CRC and RCPC code rate pairs. This can be done as follows. We send a large number N_P of random packets through the channel and decode each received packet, distinguishing between correct decoding, undetected error, and incomplete decoding. Suppose that the target transmission rate is b, which corresponds to $N = \frac{rbP}{L+c+m}$ packets sent, where P is the number of pixels in the image, c is the CRC length, and m is the memory order. We divide the N_P random packets into n_B blocks of N successive packets. In each block $i = 1, \ldots, n_B$, we count the number N_{r_i} of received packets before the firs incomplete decoding. We estimate the expected number of received packets N_{e_i} in block i by N_{r_i} if all N_{r_i} packets were correctly decoded and by zero if at least one error was undetected, which is a reasonable assumption because an undetected error can completely damage the image reconstruction. The expected number of correctly received source bits is simply $E_N = \frac{\sum_{i=1}^{n_B} N_{e_i}}{n_B} L$, where L is the number of source bits per packet.

5. EXPERIMENTAL RESULTS

Figure 1 compares the time complexity of the PLVA [4], the SLVA [4], the improved SLVA (SLVA2) [5], the TTA with a single list as a stack (L-TTA) [3], the TTA with a red-black tree as a stack (T-TTA), and the mL-TTA. The figur shows the CPU time of each algorithm as a function of the number of paths for a random packet of length 200 bits that was transmitted over a BSC with bit error rate (BER) 0.1 and encoded with the rate 1/4 convolutional code (0177,0127,0155,0171) (octal) of [7]. The time was measured on an AMD Athlon 1000 MHz processor with a main memory size of 256 Mbytes. The CPU time of the PLVA and the SLVA increased very fast, and only SLVA2 and the TTA variants required less than 5 ms for the firs 200 paths. The L-TTA was faster than the T-TTA when the number of paths was low because inserting into a balanced binary tree is more complex than inserting into a list, and the time for search operations in the list is not dominant for small lists. However, the situation changed when we increased the number of paths. The mL-TTA was always the fastest algorithm.

To maximize the expected number of correctly decoded source bits E_N, we considered RCPC codes rates $\{8/9, \ldots, 8/32\}$ and CRC lengths 16, 24, and 32 bits. We used the rate 1/3 memory order 6 mother code (0133,0165,0171) for rates $8/9, \ldots, 8/24$ and the rate 1/4 memory order 6 mother code (0177,0127,0155,0171) for the remaining rates. Both mother codes are optimum distance

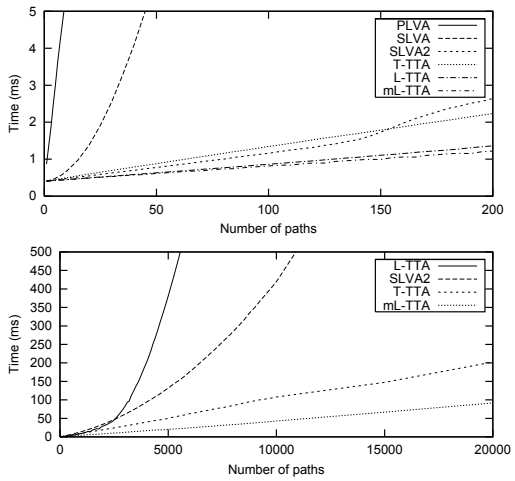

Fig. 1. Time complexity of LVAs.

spectrum codes from [7], and the RCPC codes are low rate optimized with a puncturing period of 8 [7]. The CRC generator was $(x^{16}+x^{14}+x^{12}+x^{11}+x^8+x^5+x^4+x^2+1)$ from [1] for a 16 bit CRC, $(x^{24}+x^{23}+x^{18}+x^{17}+x^{14}+x^{11}+x^{10}+x^7+x^6+x^5+x^4+x^3+x+1)$ for a 24 bit CRC, and $(x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1)$ for a 32 bit CRC. The 16 bit CRC generator was optimized for the selected packet length of 200 source bits, the other CRC generators are standard. Table 1 shows the results of the optimization for the search depth of 100 paths proposed in [1] and our proposed depth of 10,000 paths. The transmission rate was 1.0 bpp. The test image was the standard 512 × 512 Lena. For both search depths, the best CRC length was 16 bits. For BER=0.001, the same code rate was selected because it is already the highest possible code rate for the selected puncturing period and, therefore, only a small increase of E_N is noticeable. For the higher BER values, the optimal code rate for 10,000 paths is higher than that for 100 paths. Thus, we can send more source bits for the same number of packets.

The source-channel coding system with the mL-TTA and 10,000 paths needed 0.40 ms, 0.44 ms, and 0.45 ms CPU time to decode a packet at BER 0.001, 0.01, and 0.1, respectively. This was less than 25 % more than the time required with the L-TTA and 100 paths. Table 2 shows the optimization results with the L-TTA and a search depth of 900 paths, which corresponds to the complexity of the system with the mL-TTA and 10,000 paths. We conclude that even when both LVAs were allowed the same CPU time, the mL-TTA yielded a better performance.

Finally, Figure 2 shows the rate-distortion curves for BER=0.1 at various transmission rates. The peak-signal-to-noise ratio (PSNR) was computed from the corresponding E_N. At 1.0 bpp, the search depth of 10,000 paths led to a gain of 0.5 dB compared to the original search depth of 100 paths and to a gain of 0.2 dB compared to a search depth of 900 paths.

BER	Code Rate		CRC length		E_N	
	n_1	n_2	n_1	n_2	n_1	n_2
0.001	8/9	8/9	16	16	208,807	209,723
0.01	8/11	8/10	16	16	162,705	177,221
0.1	8/26	8/23	16	16	71,116	80,304

Table 1. Optimization results for $n_1 = 100$ and $n_2 = 10,000$ paths at 1.0 bpp.

BER	Code rate	CRC length	E_N
0.001	8/9	16	209,684
0.01	8/11	16	170,643
0.1	8/24	16	76,777

Table 2. Optimization results for 900 paths at 1.0 bpp.

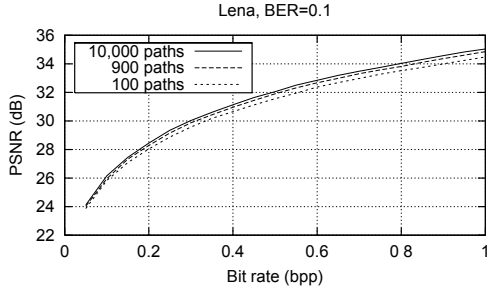

Fig. 2. Rate-distortion curves for the three search depths.

6. CONCLUSION

We showed that the time complexity of the TTA [3] can be made linear in the number of candidates by using multiple lists instead of a single sorted list. In contrast to the previous LVAs [3, 4], our new implementation can consider a large number of candidates in reasonable time. This significantl improves the PSNR performance of the system of [1] because a higher code rate can be used without severely increasing the probability of incomplete decoding.

Our results can be further improved in many ways. For example, we may allow more CRC lengths than the three used in this paper. We also may try to determine an optimal number of candidate paths. Increasing the number of paths does not necessarily improve the performance of the system because we may reach a point where the frequency of undetected errors becomes too high.

7. REFERENCES

[1] P. G. Sherwood, K. Zeger, *Progressive image coding for noisy channels*, *IEEE Sig. Proc. Letters* 4(7) 189–191, 1997.

[2] A. Said, W. A. Pearlman, *A new fast and efficient image codec based on set partitioning in hierarchical trees*, *IEEE Trans. Circuits Sys. Vid. Tech.* vol. 6, pp. 243–250, 1996.

[3] F. K. Soong, E.-F. Huang, *A tree-trellis based fast search for finding the N best sentence hypotheses in continuous speech recognition*, *Proc. ICASSP'91*, vol. 1, pp. 705–708.

[4] N. Seshadri, C.-E. W. Sundberg, *List Viterbi decoding algorithms with applications, IEEE Trans. Comm.* vol. 42, pp. 313–323, 1994.

[5] C. Nill, C.-E. W. Sundberg, *List and soft symbol output Viterbi algorithms: extensions and comparisons*, *IEEE Trans. Comm.* vol. 43, pp. 277–287, 1995.

[6] M. Röder, R. Hamzaoui, *Fast list Viterbi decoding and application for source-channel coding of images*, Konstanzer Schriften in Mathematik und Informatik, University of Konstanz, to appear 2002.

[7] P. Frenger, P. Orten, T. Ottosson, A. Svensson, *Multi-rate convolutional codes*, Technical Report 21, Chalmers University of Technology, Göteborg, 1998.