CondorCKD – Implementing an Algebraic Knowledge Discovery System in a Functional Programming Language

Jens Fisseler[1], Gabriele Kern-Isberner[2],
Andreas Koch[1], Christian Müller[1], Christoph Beierle[1]

[1]Department of Computer Science, Knowledge Based Systems,
FernUniversität in Hagen, 58084 Hagen, Germany
[2]Department of Computer Science, Information Engineering,
Universität Dortmund, 44221 Dortmund, Germany

Abstract

We introduce CondorCKD, an implementation of a novel data mining algorithm using the lazy functional programming language Haskell. While functional programming languages are often considered to be applicable to "toy problems" only, we try to give prove that these languages can of course be used to tackle demanding real-world programming tasks, and that programmers can benefit from the advantages of functional languages without having to pay too high a price with regard to running time and memory consumption. We describe our experiences gained developing with Haskell, including implementation of a graphical user interface and a novel approach to compute the cycles of an undirected graph.

1 Introduction

Data mining algorithms put high demands on their implementation languages when it comes to speed and the ability to handle large volumes of data. Because of this, data mining software has mainly been written in imperative languages such as C and C++, but also Java has become quite popular [22]. To the best of our knowledge, there are very few examples of data mining algorithms implemented in functional programming languages ([4] describes one such example), although logical languages like Prolog are very popular in the subarea of *Inductive Logic Programming* [15].

When evaluating several programming language for implementing a novel data mining algorithm, we were looking for a language which would make it easy to transfer the mathematical, high-level specification of the algorithm into program

code, and that would allow us to quickly implement a working prototype which we could further refine easily. As functional programming languages have a reputation to offer a higher productivity than imperative ones, we evaluated several functional languages and finally chose Haskell, for several reasons. Its clean syntax seemed well suited to express our mathematical specification, it offered a comprehensive standard library and good development tool support. We also hoped to turn our prototype into a usable product, without having to recode the algorithm in another language because of severe performance penalties.

In the rest of this paper we will discuss if Haskell has lived up to our expectations. We give a short overview of our algorithm in Section 2, before further elaborating on our choice of Haskell in Section 3. Two important parts of our implementation, a novel algorithm for enumerating the cycles of an undirected graph, and a graphical user interface, are discussed in Sections 4 and 5. Our general experience in using Haskell for implementing quite a demanding algorithm are stated in Section 6, and in Section 7 we conclude.

2 Knowledge discovery by reversing inductive knowledge representation

In a very general sense, the aim of knowledge discovery is to reveal *structures of knowledge* which can be seen as *structural relationships*, being represented by rules, often also called conditionals in this paper. There are two key ideas underlying the approach we used for our implementation: First, knowledge discovery is understood as a process which is inverse to inductive knowledge representation. So the relevance of discovered information is judged with respect to the chosen representation method. Second, the link between structural and numerical knowledge is established by an algebraic theory of conditionals, which considers conditionals as agents acting on possible worlds. By applying this theory, we develop an algorithm that computes sets of probabilistic rules from distributions. The inductive representation method used here is based on information theory, so that the discovered rules can be considered as being most informative in a strict, formal sense. This approach is described in detail in [10]; we will give a brief overview in this section, also presenting a small running example that will help illustrating both the method and the implementation. The results shown are found with the help of CONDOR, but the example is simple enough to be calculated "by hand". Nevertheless, it may well serve to show how the algorithm works, in particular, how missing information is dealt with.

Example 1 Suppose in our universe are *animals* (A), *fish* (B), *aquatic beings* (C), *objects with gills* (D) and *objects with scales* (E). The following table may

reflect our observations:

object	*freq.*	*prob.*	*object*	*freq.*	*prob.*
$abcde$	59	0.5463	$a\overline{b}cde$	11	0.1019
$abcd\overline{e}$	21	0.1944	$a\overline{b}cd\overline{e}$	9	0.0833
$abc\overline{d}e$	6	0.0556	$abc\overline{d}\overline{e}$	2	0.0185

We are interested in any relationship between these objects, e.g., to what extent can we expect an animal that is an aquatic being with gills to be a fish? This relationship is expressed by the *conditional* $(b|acd)$, which is read as "b, under the condition a and c and d". If P is the probability distribution given by the table above and $x \in [0,1]$ is a probability value, P satisfies the *probabilistic conditional* $(b|acd)[x]$, written as $P \models (b|acd)[x]$ iff for the conditional probability, it holds that $P(b|acd) = x$. In our example, it is easily calculated that $P \models (b|acd)[0.8]$. ■

Basically, probabilistic rules are considered not merely as statistical relationships but as chunks of knowledge, i.e. as plausible cognitive links the strength of which is expressed by a probability value and which can be used for inductive reasoning. Using probabilistic rules to build up knowledge bases, and applying an appropriate inductive reasoning mechanism to such bases, we may obtain a full probabilistic description of our world, i.e. a probability distribution. This process of completing partial knowledge is called inductive knowledge representation here. The task a knowledge discovery procedure has to accomplish can be seen as being inverse to this process: It takes a frequency distribution as description of the world and aims at extracting relevant information that somehow represents the world knowledge, see Figure 1.

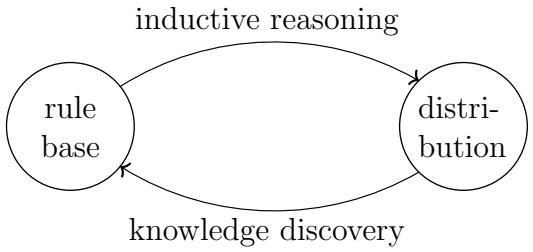

Figure 1: Knowledge discovery as inverse to inductive reasoning.

Our method exploits this interrelationship by using the *principle of maximum entropy* (*ME-principle*) as a vehicle to represent incomplete probabilistic knowledge inductively, taking the discovered rules as a basis for ME-generating the distribution under investigation. The entropy $H(P)$ of a probability distribution P is defined as $H(P) = -\sum_\omega P(\omega) \log P(\omega)$ and measures the amount of indeterminateness inherent to P. Let $\mathcal{R}^*$ be a knowledge base consisting of a set of probabilistic rules. Applying the principle of maximum entropy then

means to select the unique distribution $P^* = ME(\mathcal{R}^*)$ that maximizes $H(P)$ subject to the condition that P satisfies $\mathcal{R}^*$, i.e. $P \models \mathcal{R}^*$. In this way, the ME-method ensures that no further information is added, so that the knowledge $\mathcal{R}^*$ is represented most faithfully. $ME(\mathcal{R}^*)$ is called the *ME-representation of* $\mathcal{R}^*$. The ME-principle provides a most convenient and theoretically sound method to represent incomplete probabilistic knowledge (cf. [7, 18, 8]).

Our method is a *bottom-up approach*, starting with conditionals with long premises, and shortening these premises to make the conditionals most expressive but without losing information, in accordance with the information inherent to the data. The method is based on statistical information but not on probabilities close to 1; actually, it mostly uses only structural information obtained from the data and it is able to disentangle highly complex interactions between conditionals. We are going to discover not single, isolated rules but a set of rules, thus taking into regard the collective effects of several conditionals. Moreover, zero probabilities computed from data are interpreted as missing information, not as certain knowledge. To achieve these aims, we make use of an algebraic, group theoretical approach to conditional relationships that takes kernels of homomorphisms as structural invariants of probability distributions (for further details, cf. [9, 2]). The procedure to discover appropriate ME-generator rules from a distribution P can be sketched as follows:

- Start with a set $\mathcal{B}$ of simple association rules the length of which is considered to be large enough to capture all relevant dependencies. Ideally, $\mathcal{B}$ would consist of rules whose antecedents have maximal length (i.e. $\#(variables) - 1$).
- Search for numerical relationships in P by investigating which products of probabilities match, in order to calculate the kernel $ker\, P$.
- Compute the corresponding conditional structures with respect to $\mathcal{B}$, yielding equations of elements in the associated group $\mathcal{F}_{\mathcal{B}}$.
- Solve these equations by forming appropriate factor groups of $\mathcal{F}_{\mathcal{B}}$.
- Building these factor groups corresponds to eliminating and joining the basic conditionals in $\mathcal{B}$ to make their information more concise, in accordance with the numerical structure of P. Actually, the antecedents of the conditionals in $\mathcal{B}$ are shortened so as to comply with the numerical relationships in P.

An overview of the algorithm in pseudocode is given in Figure 2, its data flow is illustrated in Figure 3; it has been implemented as a component of the CONDOR system (for an overview, cf. [3])

Ideally, the set $\mathcal{B}$ of basic rules that the algorithm starts with would consist of rules whose antecedents have maximal length (i.e. $\#(variables) - 1$). This, however, would not be tractable and hence may not really serve as a starting point in our algorithm. There is another problem which one usually encounters

Algorithm CKD
(Conditional Knowledge Discovery)

Input A frequency/probability distribution P,
obtained from statistical data,
only listing entries with positive probabilities,
together with information on
variables and appertaining values

Output A set of probabilistic conditionals

Begin
% Initialization
Compute the *basic tree of conjunctions*
Compute the list NC of *null-conjunctions*
Compute the set $\mathcal{S}_0$ of *basic rules*
Compute $ker\, P$
Compute $ker\, g$
Set $\mathcal{K} := ker\, g$
Set $\mathcal{S} := \mathcal{S}_0$
% Main loop
While equations are in $\mathcal{K}$ **Do**
Choose $gp \in \mathcal{K}$
Modify (and compactify) $\mathcal{S}$
Modify (and reduce) $\mathcal{K}$
Calculate the probabilities of the conditionals in $\mathcal{S}$
Return $\mathcal{S}$ and appertaining probabilities
End.

Figure 2: The CKD algorithm [10].

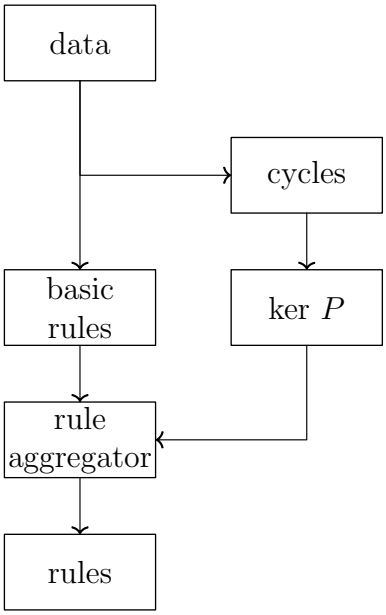

Figure 3: Dataflow of the CondorCKD algorithm.

in data mining problems: The frequency distributions calculated from data are mostly not positive – just to the contrary, they would be sparse, full of zeros, with only scattered clusters of non-zero probabilities. This overload of zeros is also a problem with respect to knowledge representation, since a zero in such a frequency distribution often merely means that such a combination has not been *recorded*. The strict probabilistic interpretation of zero probabilities, however, is that such a combination does not *exist* which seems not to be adequate.

Both of these problems – the exponential complexity of the ideal conditional starter set and the sparse and mostly incomplete knowledge provided by statistical data – can be solved in our framework in the following way: The zero values in frequency distributions are taken to be unknown, but equal probabilities, that is, they are treated as non-knowledge without structure. More exactly, let P be the frequency distribution computed from the set of data under consideration. Then, for each two worlds ω_1, ω_2 not occurring in the database and thus being assigned a zero probability, we have $P(\omega_1) = P(\omega_2)$ and hence $\frac{\omega_1}{\omega_2} \in \mathit{ker}\, P$. In this way, all these so-called *null-worlds* contribute to $\mathit{ker}\, P$, and their structure may be theoretically exploited to shrink the starting set of conditionals in advance.

In order to represent missing information in a most concise way, *null-conjunctions* (i.e. elementary conjunctions with frequency 0) have to be calculated as disjunctions of null-worlds. To this end, the *basic tree of conjunctions* is built up. Its nodes are labelled by the names of variables, and the outgoing edges are labelled by the corresponding values, or literals, respectively. The labels of paths going from the root to nodes define elementary conjunctions. So, the leaves of the tree either correspond to complete conjunctions occurring in the database, or to null-conjunctions. These null-conjunctions are collected and aggregated to

define a set NC of most concise conjunctions of probability 0.

Now we are able to set up a set $\mathcal{S}_0$ of *basic rules* also with the aid of tree-like structures. First, it is important to observe that conditionals may be separately dealt with according to the literal occurring in their consequents. So $\mathcal{S}_0$ consists of sets $\mathcal{S}_0.v$ of conditionals with consequent v, for each value v of each variable $V \in \mathcal{V}$. Basically, the full trees contain all basic single-elementary conditionals from $\mathcal{B}$, but the trees are pruned with the help of the set NC of null-conjunctions.

Next, the numerical relationships in P have to be explored to set up *ker* P. We only use complete conjunctions with non-zero probabilities for this purpose. So again, we avoid to use missing information. Usually, numerical relationships between probabilities stemming from ME-learning association rules can be found between neighboring complete conjunctions (i.e. complete conjunctions that differ in exactly one literal). We construct a *neighbor graph* from P, the vertices of which are the non-null-worlds, labeled by their frequencies or probabilities, and with edges connecting any two neighbors. Then any such relationship corresponds to a simple cycle of even length (i.e. involving an even number of vertices) in the neighbor graph, such that the cross-product built from the frequencies associated with the vertices, with alternating exponents $+1$ and -1 according to the order of vertices in the cycle, amounts to (a number close to) 1. Therefore, the search for numerical relationships holding in P amounts to searching for such cycles in the neighbor graph. Finally, as the last step of the initialization, the kernel of a structure homomorphism, *ker* g, has to be computed from *ker* P with respect to the set $\mathcal{S}_0$ of conditionals.

In the main loop of the algorithm *CKD*, the sets $\mathcal{K}$ of group elements and $\mathcal{S}$ of conditionals are subject to change. In the beginning, $\mathcal{K} = ker\ g$ and $\mathcal{S} = \mathcal{S}_0$; in the end, $\mathcal{S}$ will contain the discovered conditional relationships. Note that no probabilities are used in this main loop – only structural information (derived from numerical information) is processed. It is only afterwards, that the probabilities of the conditionals in the final set $\mathcal{S}$ are computed from P, and the probabilistic conditionals are returned.

Example 2 We continue Example 1. First, the set NC of *null-conjunctions* has to be calculated from the data; here, we find $NC = \{\overline{a}, \overline{c}, \overline{b}\,\overline{d}\}$ – no object matching any one of these partial descriptions occurs in the data base. These null-conjunctions are crucial to set up a starting set $\mathcal{B}$ of basic rules of feasible size:

$$\begin{array}{rlcllcl}
\mathcal{B} = \{ & \phi_{b,1} & = & (b|acde) & \phi_{d,1} & = & (d|abce) \\
 & \phi_{b,2} & = & (b|acd\overline{e}) & \phi_{d,2} & = & (d|abc\overline{e}) \\
 & \phi_{b,3} & = & (b|\overline{d}) & \phi_{d,3} & = & (d|\overline{b}) \\
 & \phi_{e,1} & = & (e|abcd) & \phi_{a,1} & = & (a|\top) \\
 & \phi_{e,2} & = & (e|abc\overline{d}) & & & \\
 & \phi_{e,3} & = & (e|a\overline{b}cd) & \phi_{c,1} & = & (c|\top)\ \}
\end{array}$$

So, the missing information reflected by the set NC null-conjunctions helped to shrink the starting set $\mathcal{B}$ of rules from $5 \cdot 2^4 = 80$ basic single-elementary rules to

only 11 conditionals. The next step is to analyze numerical relationships. In this example, we find two numerical relationships between neighboring worlds that are nearly equal:

$$P(a\bar{b}cde) \approx P(a\bar{b}cd\bar{e}) \quad \text{and} \quad P(\frac{abcde}{abcd\bar{e}}) \approx P(\frac{abc\bar{d}e}{abc\bar{d}\bar{e}})$$

These relationships are translated into the algebraic group equations and help modifying the set of rules. For instance, $\phi_{b,1}$ and $\phi_{b,2}$ are joined to yield $(b|acd)$, and $\phi_{e,3}$ is eliminated. As a final output, the CKD algorithm returns the following set of conditionals:

Conclusion	*Premise*	*Prob.*
A=YES		1.0
B=YES	D={NO}	1.0
B=YES	A={YES}, C={YES}, D={YES}	0.8
C=YES		1.0
D=YES	B={NO}	1.0
D=YES	A={YES}, B={YES}, C={YES}	0.91
E=YES	A={YES}, B={YES}, C={YES}	0.74

Note that our system actually has generated the LaTeX-code for the table given above as output. The only modification necessary was to adapt the table width to the column width. The following table shows the same set of conditionals using the mathematical notation used hitherto:

cond.	*prob.*	*cond.*	*prob.*
$(a\|\top)$	1	$(c\|\top)$	1
$(b\|\bar{d})$	1	$(d\|\bar{b})$	1
$(b\|acd)$	0.8	$(d\|abc)$	0.91
$(e\|abc)$	0.74		

All objects in our universe are aquatic animals which are fish or have gills. Aquatic animals with gills are mostly fish (with a probability of 0.8), aquatic fish usually have gills (with a probability of 0.91) and scales (with a probability of 0.74). ∎

3 Using Haskell for data mining

When beginning with the implementation of the CondorCKD algorithm, we had to choose a suitable programming language. As we had an abstract, high-level description of the algorithms and their corresponding data structures, we were looking for a programming language that would make it easy to transfer these algorithms from their mathematical description to an executable form. We also wanted to be able to quickly implement a prototype so to review the results of the algorithms and further refine them. In order to do this, it should be possible for people with less a background in programming to look at the code and get a rough idea of what it would do. This led us to favor functional programming languages over imperative ones.

In his famous paper [6], Hughes points out that functional programming languages offer two potential advantages over conventional ones: *higher-order functions* and *lazy evaluation*. Higher-order functions help in modularizing programs by allowing the programmer to separate recurring patterns of computation into a general – thereby reusable – higher-order function and a small specialized function, thus increasing programmer efficiency. Lazy evaluation is regarded even more important to modular programming, as it allows the programmer to combine smaller functions to increasingly larger ones, but computing only what is really needed, thus enabling the program to process potentially infinite data structures.

While higher-order functions and lazy evaluation are two advantages shared by other functional programming languages, our final decision for Haskell was based on several other important factors:

- Clean and concise syntax
- Strong typing and good support for user defined data-types
- A comprehensive standard library
- Availability of good compilers and additional development tools

Figure 4 very briefly illustrates the Condor system that is completely implemented in Haskell. It takes data in the form of CSV or ARFF files (these are formats for exchanging tabular data widely used in data mining systems, see e.g. [22]) as input. The parser of the CondorCKD component reads these input files and makes the data available to functions generating the internal representations of probability distributions, including meta data about the involved variables. Using a logic representation component, probabilistic rules are extracted from the probability distribution and are presented both in a simple text format (ready for further processing) as well as in a polished LaTeX version. The complete user interaction is supported by a graphical user interface which is also fully implemented in Haskell.

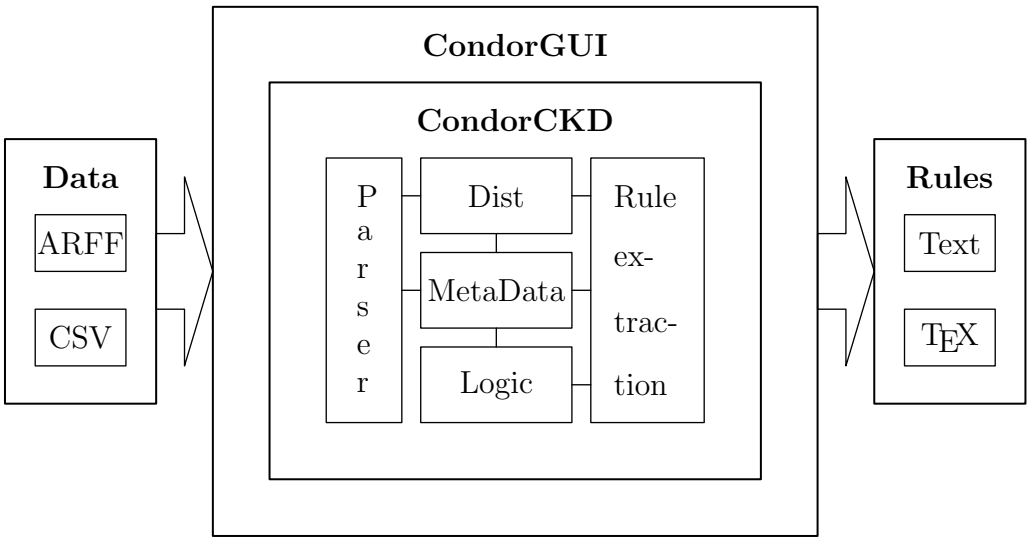

Figure 4: Overview of the CONDOR system.

In the following, we will present two components of the CONDOR system in more detail. The first one deals with finding particular cycles in labelled undirected graphs which play a central role at the heart of our CKD algorithm. The other component is the graphical user interface for which we used the wxHaskell library. After presenting our functional programming approach towards these two components, we will give a general overview of our experiences in using Haskell for implementing quite a complex data mining algorithm.

4 Enumerating all cycles of an undirected graph – a functional programming challenge

As described in Section 2, one of the most important parts of the CONDORCKD algorithm is the computation of the simple cycles of the neighbor-graph. These depict numerical relationships in the input data, which are used for aggregating the basic rules.

4.1 Cycles in an undirected graph

Recall that, for an undirected graph $G = (V, E)$ with edges $E \subseteq \{\{u, v\} \mid u, v \in V, u \neq v\}$, a *simple cycle* (of length k) is a sequence $\langle v_0, v_1, \ldots, v_k, v_0 \rangle$ of pairwise distinct vertices $v_i \in V$. We are interested in computing *all* simple cycles with even length up to a certain maximum length $k_{\max} \in \{2, 4, 6, \ldots\}$. This cycle length restriction is necessary, as the number of simple cycles can be exponential in the number of vertices.

The enumeration of all simple cycles can be done separately for every biconnected component of an undirected graph, as two biconnected components are

connected only by a single vertex, a so called *cut vertex*. Because of this, every path between two vertices contained in different biconnected components must include at least one cut vertex, and a cycle between these two vertices would contain these cut vertices twice, hence cannot be a simple cycle.

There are several approaches to compute the simple cycles of an undirected graph [14], of which the *vector space* approach and *search-based algorithms* are the most important ones.

Vector space algorithms are based on the fact that all cycles of an undirected graph can be expressed as a composition of *fundamental cycles*. Every spanning tree (V, T) of an undirected graph $G = (V, E)$ induces a set of fundamental cycles, as every *back edge* $e \in E \setminus T$ creates exactly one fundamental cycle when added to T. Since every spanning tree of G contains $|V| - 1$ edges, there are $|E| - |V| + 1$ fundamental cycles $\{S_1, S_2, \ldots, S_{|E|-|V|+1}\}$. Such a *fundamental cycle set* is a basis of a vector space, and every cycle of G can be written as $(\cdots(S_{i_1}\Delta S_{i_2})\Delta\cdots)\Delta S_{i_n}$ for some $S_{i_1}, \ldots, S_{i_n} \in \{S_1, S_2, \ldots, S_{|E|-|V|+1}\}$, where Δ depicts the symmetric difference operation between sets of edges. Whereas every cycle of an undirected graph can be represented this way, generally only a small fraction of the $2^{|E|-|V|+1}-|E|-|V|$ possible combinations are cycles, the rest being disjoint unions of cycles. Although several vector space algorithms have been developed [14], very little has been done regarding pruning these unnecessary computations, let alone incorporating cycle length restrictions.

Search-based algorithms use a depth-first search during which edges are added to a path until a cycle is produced. Careful pruning of the search space is necessary to ensure that every cycle is generated exactly once and that very little unnecessary work is done. To this end, we compute a spanning tree (V, T) using a graph-exploration search (DFS or BFS), during which a unique number $d(v)$ is assigned to every vertex $v \in V$ upon the first visit of this v. Simultaneously, a set of possible cycle starting vertices is calculated. Because every back edge (the edges creating a fundamental cycle, see above) is an edge of at least one simple cycle, and as every cycle must end in a back edge, the vertices with entering back edges are possible cycle starting vertices. To compute all cycles, a (limited) depth-first search is started in every cycle starting vertex s, and we build a path $\langle v_0 = s, v_1, v_2, \ldots, v_k\rangle$ with $d(v_i) > d(s), 1 \leqslant i \leqslant k$. As the vertices are totally ordered by $d(\cdot)$, we can assume that every cycle is rooted at its lowest numbered vertex, and we only have to consider path-extending vertices $v_{k+1} \notin \{s, v_1, \ldots, v_k\}$ with $d(v_{k+1}) > d(s)$. This way, every cycle is only found twice, and by imposing a direction on the back edges, it can be ensured that every cycles is found only once.

4.2 A combined approach

Although several depth-first search algorithms for computing all the cycles of a graph are described in the literature [20, 19], their imperative description – which uses destructive updates for marking vertices not available for extending the current path – makes their implementation in a lazy functional language quite

a challenge. Though we did not use the monadic-state based approach described in [11], we have paid attention to use data structures supporting fast insertion, lookup and deletion (like *IntMap*s) for storing the available vertices. Making functions as strict as possible was also important in obtaining a fast algorithm. But more important than this was a new idea of further reducing the search space of the depth-first search by using results based on the vector space approach.

The key to our new algorithm is the fact that every fundamental cycle has one special edge that is not a member of any other fundamental cycle – the back edge closing a path in the corresponding spanning tree, thus yielding this fundamental cycle. Assume a fixed ordering of the elements of a given fundamental cycle set $\{S_1, S_2, \ldots, S_{|E|-|V|+1}\}$. Recall that every cycle c can be written as $c = (\cdots(S_{i_1} \Delta S_{i_2}) \Delta \cdots) \Delta S_{i_n}$, w.l.o.g. $i_1 < i_2 < \cdots < i_n$. At least one edge must have been removed from every S_i, otherwise one or more of the S_i would be part of c as a whole, which means c would be a disjoint union of cycles. Thus, in order to compute all cycles containing the back edge of S_i, $1 \leqslant i \leqslant |E| - |V| + 1$, we can restrict the graph to be searched to the subgraph induced by $S_1 \cup \cdots \cup S_i$.

Our cycle enumeration can be described by three steps:

1. Compute a set of fundamental cycles $\{S_1, S_2, \ldots, S_{|E|-|V|+1}\}$.

2. For every S_i, compute a subgraph G_i. G_i is the union of a subset of the fundamental cycles $S_1, \ldots, S_i$, and is defined by the cycles of the equivalence class of S_i with respect to the transitive closure of the relation $R^i_\cap$,

$$R^i_\cap := \{(S_r, S_t) \mid S_r, S_t \in \{S_1, \ldots, S_i\} \wedge S_r \cap S_t \neq \emptyset\}.$$

3. Conduct a DFS-based cycle enumeration in each of these subgraphs G_i, starting at one of the vertices incident to the back edge of S_i.

Functional pseudo-code for an algorithm computing a set of fundamental cycles is given in Figure 5. We use a breadth-first search, as the resulting fundamental cycles tend to be short, i.e. will have common edges with only a few other fundamental cycles. Because of this, the subgraphs described in step 2 above tend to be small (this is just a heuristic found useful in practice).

After computing a set of fundamental cycles, we generate the subgraphs that are subsequently searched for simple cycles. As the subgraphs pertaining to higher numbered fundamental cycles are supergraphs of previously computed subgraphs, we can save some computations by storing the already generated subgraphs in an appropriate data structure. This set contains edge-disjoint unions of fundamental cycles and is initially empty. If we want to compute all cycles containing the back edge of fundamental cycle S_i, $1 \leqslant i \leqslant |E| - |V| + 1$, we look at all the subgraphs already processed and check whether they have at least one edge in common with the fundamental cycle S_i. All these subgraphs are removed from this set and are conjoined, together with S_i, yielding a new subgraph G_i. This new subgraph contains all cycles possibly including the back edges of the

```
generateFundamentalCycles :: Graph
                          → Set.Set Vertex
                          → [FundamentalCycle]
generateFundamentalCycles graph availableVerts
  | Set.null availableVerts
      = []
  | otherwise
      = fcsInComp ++ (generateFundamentalCycles
                      graph availableVerts'')
  where
    (v, availVerts') = Set.deleteFindMin availVerts
    parents = Map.empty
    depth = Map.singleton v 0
    queue = Queue.listToQueue (map (λ u → (u, v))
                                   (Graph.edges graph v))
    (fcsInComp, availVerts'') = BFS graph availVerts'
                                queue parents depth
```

Figure 5: Fundamental cycle generation.

fundamental cycles contained in G_i and is subsequently processed with a depth-limited DFS-based cycle enumeration algorithm, but the search is started only once in the vertex with the incoming back edge of S_i. Afterwards, the newly computed cycles are added to the set of all fundamental cycles, the subgraph G_i is added to the set of the already processed subgraphs and we continue with fundamental cycle S_{i+1}.

Preliminary results comparing the running time of our algorithm (FCs + DFS) and the running time of a simple search-based algorithm (DFS) are promising (see Figure 6).

We have compared both algorithms on three graphs, using different maximum cycle lengths. The Haskell code was compiled using GHC 6.4.1 with optimization turned on and code generation via C. The executables were running on an AMD Athlon64-3200+ in 32bit-mode with 1GB of RAM, using Linux.

Although the results are encouraging, there is room for further improvement. Our DFS-based search algorithm does not use the approach described in [11], i.e. does not have the same asymptotical run-time behaviour as its imperative counterparts. Using a monadic-state based approach in conjunction with unboxed arrays might further improve the running time. Other constraints imposed on the cycles might be exploited to further reduce the algorithm's run-time, but this is specific to our problem at hand.

graph	*max. length*	*#cycles*	*FCs + DFS*	*DFS*
A	10	2827	0:00:01	0:00:02
	12	16699	0:00:05	0:00:13
	14	119734	0:00:45	0:08:13
	16	890204	0:05:48	7:23:54
B	10	2929	0:00:04	0:00:06
	12	23021	0:00:42	0:01:28
	14	222459	0:09:11	0:42:09
C	6	2927	0:00:10	0:01:26
	8	18695	0:00:36	0:02:13
	10	268097	0:13:51	0:30:27

Figure 6: Runtime comparison.

5 Power to the user – a GUI for knowledge discovery

From the start, CONDOR has been developed with the end user in mind. Though these end users will probably be researchers or other people doing knowledge discovery – i.e. users who probably have above average computer literacy skills – we wanted to offer them a comfortable user interface to make working with CONDOR as convenient as possible. To this end, we have developed a graphical user interface, using the wxHaskell library [12]. In this section we will give a short overview of the GUI; a description of our experience in using wxHaskell is given in Section 6.4.

All user actions are available through the menu, see Figure 7. As the main action when working with CONDOR will be to compute the rules modelling a previously loaded distribution, we put the corresponding menu item at the top of the "Investigate distribution" menu. Selecting "Calculate rules..." will open a new dialog, see Figure 8. In this dialog, the user can adjust several parameters concerning the CONDORCKD algorithm, and must specify the output file into which the computed rules will be written. There are several other parameters available as well, but as these will only be of concern to more advanced users or developers, we have put these into another notebook tab labelled "expert options". This way, occasional users won't be distracted by unnecessary detail. One of these advanced options is the choice of the output format of the rules. These are written to plain text files by default, which is fine most of the time, as it allows for quick inspection. But on certain occasions, a more elaborate presentation is necessary. Because of this, CONDOR offers the option of exporting the rules using LaTeX-code. Figure 9 shows an excerpt from a report on rules calculated from the "Postoperative Patient" dataset available from the UCI Machine Learning Repository [16]. The screenshots given in Figure 7, 8 and 10 were taken during a data mining session analyzing this dataset.

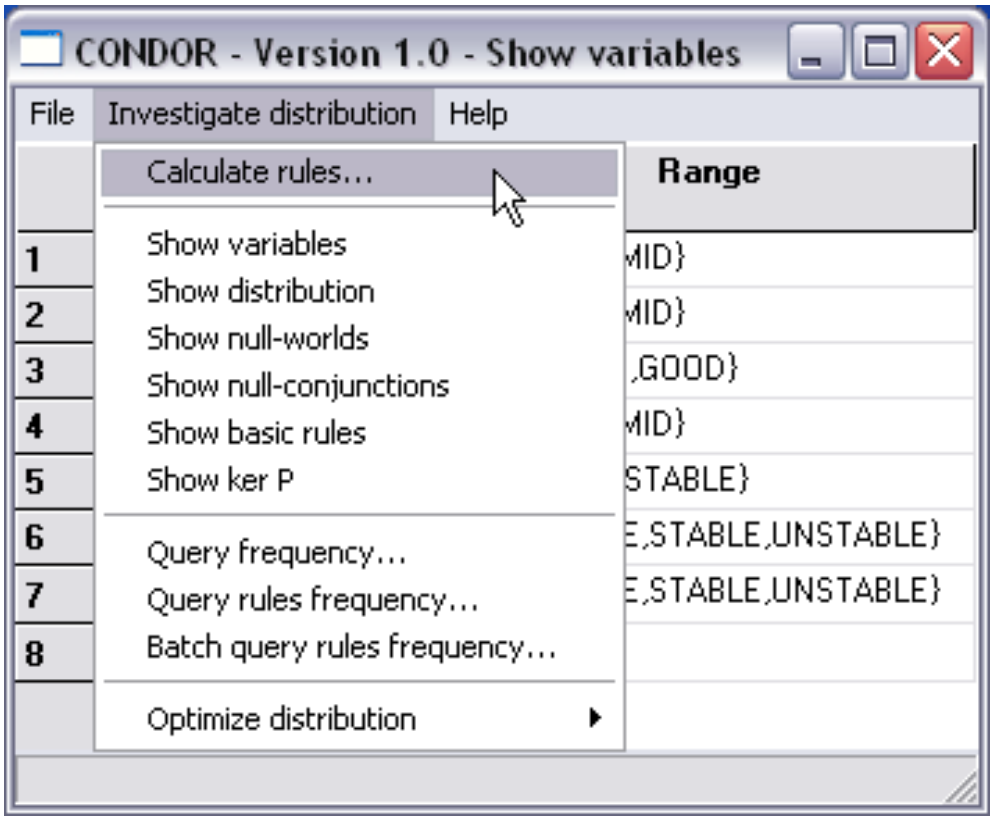

Figure 7: The "Investigate distribution" menu.

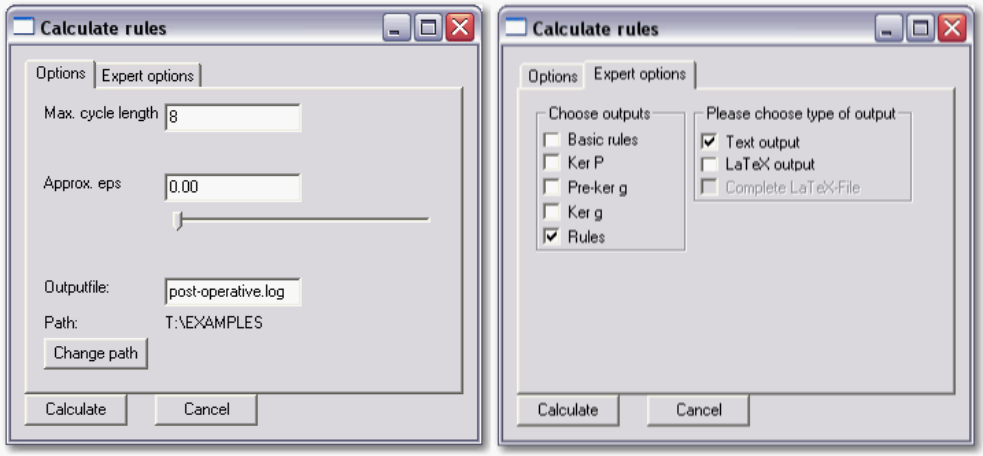

Figure 8: The "Calculate rules" dialog.

Condor Report

Meta data

Variable	*Literals*
LCORE	$\{HIGH, LOW, MID\}$
LSURF	$\{HIGH, LOW, MID\}$
LO2	$\{EXCELLENT, GOOD\}$
...	

Rules

Conclusion	*Premise*	*Prob.*
LCORE=HIGH	LSURF={HIGH}, LO2={EXCELLENT}, LBP={HIGH}	0.75
LCORE=HIGH	LSURF={HIGH}, LO2={EXCELLENT}, LBP={MID}	0.5
LCORE=HIGH	LSURF={HIGH}, LO2={GOOD}, LBP={HIGH}	0.5
LCORE=HIGH	LSURF={HIGH}, LO2={GOOD}, LBP={LOW}	0.0
...		

Figure 9: Excerpt from a report file.

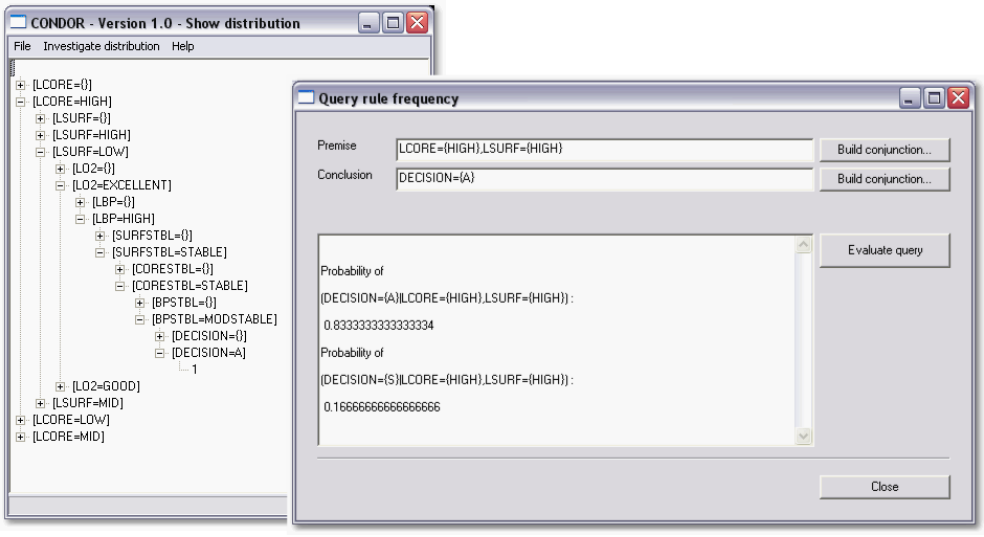

Figure 10: GUI components for investigating a distribution.

Another important part of working with CONDOR is querying the probability distribution. This incorporates the calculation of (conditional) probabilities, which can be used to verify prior assumptions about the relationships present in the probability distribution, but can also be utilized to validate rules. Sometimes it may also be helpful to look at the internal data structure used to represent the probability distribution. Both tasks are supported by the GUI, see Figure 10.

There is even more information available through the GUI, but most of it is only important to advanced users and developers, as its comprehension requires a good understanding of the CONDORCKD algorithm. Therefore, we have only briefly introduced the main parts of the GUI.

Though there may be many opportunities for improving our user interface, it has nevertheless been an enormous advantage being able to provide a GUI. Today's computer users are in general more familiar with – and more comfortable using – a graphical user interface than a text-based or even a command line interface. A graphical interface also offers better ways of user interaction and facilities for displaying information. Because of this, a GUI better supports the user during the interactive and iterative knowledge discovery process.

6 Lessons learned

After introducing the CONDORCKD algorithm and reviewing some important parts of our implementation, we now want to give an overview of our general experience in using a functional programming language for implementation of a knowledge discovery algorithm. One thing to note is that some of the programmers involved in the implementation of the CONDOR system only had very little previous experience with Haskell, consisting mostly of an introductory university course in functional programming.

6.1 Clean and concise syntax

As stated in Section 3, the description of the CONDORCKD algorithm was given in pseudocode with a strong mathematical flavor. This was very convenient during the implementation, as the specification was already decomposed into many intertwined functions and utilized set-based syntax for describing collections of data with certain properties and constraints. These mathematical concepts were easily and rapidly coded in Haskell, whereas Haskell's concise syntax allowed us to stay very close to the original specification.

The often mentioned brevity of functional programs applies to CONDORCKD, too. Our functions generally consist of only a few lines, few have more than a dozen. In our view, this brevity results from the use of higher-order functions (like *map*, *filter* and *fold*) in combination with Haskell's automatic memory management. When manipulating data in imperative or object-oriented languages, a good part of the code deals with traversing the data and managing the memory needed for its storage, whereas in functional languages the programmer can focus

on the data manipulation itself. The memory management is done implicitly, and the traversal is done by a higher-order function, accepting the data manipulating code (in terms of an anonymous function) as an argument.

Shorter, more concise functions are not only easier to comprehend, they are also tested more easily, and can then be combined to build more complex functions. Figure 11 shows one of our more complex functions, in fact it shows the core function of the CondorCKD algorithm, the modification of $ker\, g$, i.e. the elimination and aggregation of conditionals based on certain numerical properties found in the data. Although the function depicted in Figure 11 is quite lengthy, it's inner workings are easily understood nonetheless, thanks to the "compositional glue" offered by functional programming languages and because of the use of *guards* and *pattern matching*. Both allow to structure conditional expressions more concise than with nested *if-then*-statements, and thus make it easier to verify whether all different conditions have been considered. Pattern matching is even more important than are guards, because the compiler aids the programmer in ensuring to meet every boundary condition, what cannot be done for guards, but see Section 6.2 for a more elaborate discussion.

6.2 Strong typing

Haskell's type system is often emphasized as one of the language's most important features in helping the user writing correct programs, and we can only support this claim.

When writing a new module function, we have made a habit of writing its type signature first, then implementing its body. This way, the compiler or interpreter would complain about type mismatches, which easily occur when writing new functions not used by or not using other module functions. Writing a type signature is hence an additional tool in forcing the programmer to think about the function once again instead of quickly hacking it down. Additionally, type signatures act as some sort of documentation, supporting the programmer when no other documentation has yet been written or is outdated.

It could be argued that other languages like C++ or Java also feature an expressive type system, but Haskell's type system, in cooperation with several other aspects, offers additional benefits. Whereas a C++ or Java compiler will only issue a warning or an error when the arguments given to a function or its return value don't match its declaration, a Haskell compiler can infer the type of a function based on its arguments. By comparing this type to the type signature of the function, it is often quite easy to find the bug resulting in the type mismatch.

Algebraic data types are another important part of Haskell's type system. In conjunction with *pattern matching*, algebraic types allow the processing of data based on its structural properties. Because pattern matching is the only way to extract data from algebraic types, incomplete patterns will cause the compiler to issue a warning, thus forcing the programmer to rethink the boundary conditions of his data types. Of course, this warning will be given only when using the appropriate compiler flag, but activating all possible warnings and striving for a

```
modifyKergElem :: Set Conditional
                  → (ConditionalProduct,
                      [ConditionalProduct])
                  → Maybe (Set Conditional,
                                [ConditionalProduct])
modifyKergElem conds (eqCondProd, restCondProds)
  | isEquationTypeA lhs rhs
      = Just (filterConditionalProduct rhs
                conds restCondProds)
  | isEquationTypeB lhs rhs
      = Just (conjoinConditionalProduct (fst $ head lhs) rhs
                conds restCondProds)
  | isEquationTypeA rhs lhs
      = Just (filterConditionalProduct lhs
                conds restCondProds)
  | isEquationTypeB rhs lhs
      = Just (conjoinConditionalProduct (fst $ head rhs) lhs
                conds restCondProds)
  | otherwise
      = Nothing
  where
    (lhs, rhs) = splitConditionalProduct eqCondProd
    isEquationTypeA :: ConditionalProduct
                       →ConditionalProduct
                       →Bool
    isEquationTypeA [] _
        = True
    isEquationTypeA (_ : _) _
        = False
    isEquationTypeB :: ConditionalProduct
                       →ConditionalProduct
                       →Bool
    isEquationTypeB [] _
        = False
    isEquationTypeB (_ : []) eqRhs
        = all ((== 1) . snd) eqRhs
    isEquationTypeB (_ : _) _
        = False
```

Figure 11: The "heart" of the CondorCKD algorithm.

```
incrComplConj :: DTree
              → MetaData.Variable
              → [MetaData.Literal]
              → DTree
incrComplConj NullNode _ [] = ...
incrComplConj NullNode varIdx (lit : lits) = ...
incrComplConj leaf@(Leaf {}) _ [] = ...
incrComplConj Leaf {} _ (_ : _) = error ...
incrComplConj Node {} _ [] = error ...
incrComplConj node@(Node {}) varIdx (lit : lits) = ...
```

Figure 12: Function for incrementing the frequency of an elementary conjunction.

clean compile is a good habit anyway. Otherwise warnings pertaining to possibly serious errors will drown in unimportant messages.

Figure 12 shows an example of pattern matching applied to the algebraic data type *DTree*. The pattern matching is actually done on the first and third parameter and covers all boundary conditions, whereas two boundary conditions (4th and 5th line of the function definition) should not occur if everything works correctly, thus these result in an *error*.

Taking into account every possible boundary condition is quite difficult in languages like C++ or Java, as these don't support algebraic types, which thus have to be modelled using inheritance or certain member fields of a structured data type. It is then up to the programmer to take care of the boundary conditions, with the possibility – and high probability – of corresponding errors.

6.3 A comprehensive standard library

As Okasaki stated in [17], "functional programmers [...] too often reach for lists when an ADT would be more appropriate". Although list processing is omnipresent in functional programming, C++'s *Standard Template Library* and the *Java Collections Framework* demonstrate that a well designed library of container classes and accompanying functions increase programmer productivity by allowing him to focus on the real important task.

We extensively used the *Haskell Hierarchical Libraries* while implementing the CondorCKD algorithm, especially the collection types *Map* and *Set*. They allow for similar processing as lists, as they also offer the probably most often used (list) manipulation functions *map*, *filter* and *fold*. This made it easy to change functions processing lists to utilize a more appropriate collection type. The use of higher-order functions was a great benefit over the iterator-based interface used in imperative languages. Because of this, we could focus on how to work with the data instead of having to worry about how to employ iterators to shuffle the data around or even memory-management issues.

PARSEC [13], a parser combinator library, is another example where Haskell's clean and concise syntax, in collaboration with a well designed programming interface, support the programmer in getting his work done quickly and elegantly. Figure 13 shows a small EBNF grammar (the rules for *variable-identifier* and *literal* have been left out as these are little more than alphanumeric strings), and Figure 14 depicts the Haskell code for the *query* rule. Though the real code is slightly more complicated, using qualified names and additional PARSEC functions for improved error messages, this clearly shows that using appropriate abstractions, parsers can be implemented easily and concisely. The code clearly resembles the EBNF structure for the corresponding rule, a definite advantage over tools requiering the usage of another language to specify the parser grammar.

```
query                = "(" variable-literal-list
                         [ "|" variable-literal-list ]  ")";
variable-literal-list = variable-literals { "," variable-literals }*;
variable-literals    = variable-identifier "={" literal-list "}";
literal-list         = [ literal { "," literal }* ];
```

Figure 13: A small EBNF grammar.

```
query :: Parser ([(String, [String])], [(String, [String])])
query = do
          char '('
          spaces
          concl ← variableLiteralsList
          try ( do
                  char ')'
                  return (concl, [])
            <|> ( do
                    char '|'
                    spaces
                    prem ← variableLiteralsList
                    char ')'
                    return (concl, prem))
```

Figure 14: Haskell code implementing a EBNF rule.

6.4 Implementing a graphical user interface

Having a suitable GUI-library available is another important aspect of whether a language will be used by software developers, because nowadays most end-user software has to offer a graphical user interface, otherwise most users simply won't accept it.

As described in Section 5, we have built a GUI using the wxHaskell library [12] in order to make the usage of CONDOR more comfortable. We have chosen wxHaskell over other GUI libraries (like gtk2hs), because wxHaskell seemed to be one of the most matured mid-level graphical user interface libraries available, and as it is a Haskell wrapper for the platform-independent wxWidgets library, we expected the resulting code to be portable as well.

Our experiences show that wxHaskell in general provides all the functionality needed to realize the GUI of the CONDOR system, but might need some further polishing at some of its parts. A specific observation we have made concerns the compatibility with respect to different platforms like Windows and Linux. Whereas in general, this compatibility was achieved by using the cross-platform library wxHaskell, some differences surfaced, e.g. when using a combo-box (a widget that allows the selection of one of a set of predefined items, but can also support free-form text entry). The reason for the slightly varying behaviour on different platforms might be found in the behaviour of the wxWidgets underlying the wxHaskell library, or even some of the platform-specific libraries wrapped by wxWidgets itself.

6.5 A suitable development environment

To support the development of medium- to large-scale software, complementary tools are necessary in addition to a suitable compiler or interpreter and a standard library: debuggers and profilers, documentation tools, and editing support or even integrated development environments. Fortunately, all these kind of tools are available for Haskell, although not all have been used to the same extent during the implementation of the CONDORCKD algorithm.

The most important part of a development environment is of course the language implementation itself, and there are quite a few available for Haskell. Initial development was done with Hugs, because its interactive interpreter made the incremental development very convenient. But as the project's code size grew and the algorithms needed to be tested on real-world data, better run-time performance was required and we switched to GHC, which has an interactive environment, too, but also the ability to generate executables.

Debugging was mainly done by interactive testing of functions and excessive printing of intermediate data whenever possible. This can only be considered unwise in retrospect, but the existence of Hood [5] just escaped us. We're planning to use Hood during further development of CONDORCKD.

Profilers are another invaluable development tool, because only their usage permits the exact location of performance bottlenecks, even more so with lazy

evaluation [21]. Fortunately, GHC includes support for space and time profiling, and this has helped increasing our algorithms' performance on several occasions, although *how* to improve performance was not always obvious, see Section 6.6.

Source code editing was mainly done with Emacs and its Haskell mode, which offered enough support for our needs. But as with debugging, other IDEs were not tried and might offer additional benefits.

6.6 Battling with laziness and fighting excessive memory consumption

One language feature that nearly every Haskell programmer has to deal with when developing larger applications is *lazy evaluation*. While it allows for elegant solutions to a lot of problems, and can even make programs more efficient by evaluating only what is needed, our experience has shown that when dealing with really large data sets, laziness will delay many computations, and the program will take up huge amounts of heap space. Fortunately, there are several remedies.

Functions can be made strict by using "*seq*" to force evaluation of their arguments. As it is not always obvious which functions should be made strict, time profiling is needed to identify possible candidates. To assess whether the important functions are as strict as possible, inspection of the Core code generated by GHC is necessary. Thus, making a function strict is possible, but quite laborious, and it can make the resulting more difficult to read, loosing some of the conciseness emphasized in Section 6.1. Hopefully this will be remedied with the introduction of the so-called *bang patterns* proposed for the upcoming Haskell' standard [1].

Using strictness flags ("!") to force the evaluation of the arguments of a data constructor is another way to remove some laziness, and in combination with the "UNPACK" pragma this can even help in reducing excessive memory consumption.

Other means in reducing memory consumption are of course using appropriate data types. For example, CondorCKD uses a lot of small objects for representing so called *conjunctions* (think of these as simple logical formulas representing the premises of rules). Initially, these conjunctions where implemented using nested lists of *Int*s. In retrospect, this can only be considered naive, because of Haskell's lazy evaluation, every list element occupies three words of memory. As the flexibility offered by lists was not needed for representing conjunctions, these have replaced by unboxed arrays of *Int*s, leading to some serious reduction in memory requirements.

7 Conclusion

We have introduced our implementation of CondorCKD, a novel algorithm for knowledge discovery based on the principle of maximum entropy. We have used Haskell for its implementation, whereas our choice was based on our expectation

to be able to quickly implement the algorithm based on its abstract, high-level description. Haskell has lived up to our expectations, as a prototype was implemented quite rapidly, though optimizing its run-time behaviour and memory consumption was a lengthier – and sometimes problematic – process, though this is the case with every optimization. Nonetheless, the resulting code displays the brevity ascribed to functional programs, as the whole *documented* codebase involves little more than 9000 lines of code, including a GUI. An implementation written in C++ or Java can be expected to be several times larger, and despite being optimized, the Haskell code still closely resembles its abstract description, a definite advantage over imperative languages.

Having several well designed libraries available during development, including several collection types, a parser library and a GUI-library, was also beneficial. Haskell's development tools, especially GHC with its profiling abilities, were of great use in optimizing the algorithms.

Currently, we're planning to further refine and enhance our algorithm and its implementation. This includes work to further reduce the memory consumption, but also trying to utilize external data storage, because when analyzing large and complex data sets, the intermediate data structures computed by our algorithm, esp. the cycles (see Section 4), will definitely be too large to fit into even todays computer's main memory.

We will also develop an inference algorithm utilizing the discovered rules to draw information-optimal conclusions, see [8]. This will also be done in Haskell, because for us, Haskell was a very good choice.

Acknowledgements

The research reported here was partly supported by the DFG – Deutsche Forschungsgemeinschaft (grant BE 1700/5-3).

References

[1] Bang pattern proposal. `http://hackage.haskell.org/trac/haskell-prime/ticket/76` (06.01.2006).

[2] C. Beierle and G. Kern-Isberner. An alternative view of knowledge discovery. In *Proceedings Hawaii International Conference on System Sciences, HICSS-36*. IEEE Computer Society, 2003.

[3] C. Beierle and G. Kern-Isberner. Modelling conditional knowledge discovery and belief revision by abstract state machines. In E. Boerger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003 – Advances in Theory and Applications, Proceedings 10th International Workshop, ASM´2003*, pages 186–203. Springer, LNCS 2589, 2003.

[4] Amanda Clare and Ross D. King. Data mining the yeast genome in a lazy functional language. In Verónica Dahl and Philip Wadler, editors, *Practical Aspects of Declarative Languages, 5th International Symposium, PADL 2003, Proceedings*, volume 2562 of *Lecture Notes in Computer Science*, pages 19–26. Springer, 2003.

[5] Andy Gill. Debugging haskell by observing intermediate data structures. *Electronic Notes in Theoretical Computer Science*, 41(1), 2000.

[6] John Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.

[7] E.T. Jaynes. *Papers on Probability, Statistics and Statistical Physics*. D. Reidel Publishing Company, Dordrecht, Holland, 1983.

[8] G. Kern-Isberner. Characterizing the principle of minimum cross-entropy within a conditional-logical framework. *Artificial Intelligence*, 98:169–208, 1998.

[9] G. Kern-Isberner. Solving the inverse representation problem. In *Proceedings 14th European Conference on Artificial Intelligence, ECAI'2000*, pages 581–585, Berlin, 2000. IOS Press.

[10] G. Kern-Isberner and J. Fisseler. Knowledge discovery by reversing inductive knowledge representation. In *Proceedings of the Ninth International Conference on the Principles of Knowledge Representation and Reasoning, KR-2004*. AAAI Press, 2004.

[11] David J. King and John Launchbury. Structuring depth-first search algorithms in haskell. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)*, pages 344–354. ACM Press, 1995.

[12] Daan Leijen. wxHaskell – a portable and concise GUI library for Haskell. In *ACM SIGPLAN Haskell Workshop (HW'04)*. ACM Press, 2004.

[13] Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

[14] Prabhaker Mateti and Narsing Deo. On algorithms for enumerating all circuits of a graph. *SIAM Journal on Computing*, 5(1):90–99, 1976.

[15] Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19/20:629–679, 1994.

[16] D. J. Newman, S. Hettich, C. L. Blake, and C. J. Merz. UCI repository of machine learning databases, 1998.

[17] Chris Okasaki. An overview of edison. *Electronic Notes in Theoretical Computer Science*, 2000(1):60–73, 2000.

[18] J.B. Paris. *The uncertain reasoner's companion – A mathematical perspective*. Cambridge University Press, 1994.

[19] R. C. Read and R. E. Tarjan. Bounds on backtrack algorithms for listing cycles, paths, and spanning trees. *Networks*, 5(3):237–252, 1975.

[20] Edward M. Reingold, Jurg Nievergelt, and Narsingh Deo, editors. *Combinatorial Algorithms: Theory and Practice*, chapter Graph Algorithms. Prentice-Hall, Englewood Cliffs, 1977.

[21] Colin Runciman and David Wakeling. Heap profiling of lazy functional programs. *Journal of Functional Programming*, 3(2):217–245, 1993.

[22] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2005.