Declarative Merging of and Reasoning about Decision Diagrams[⋆]

Thomas Eiter, Thomas Krennwallner, and Christoph Redl

Institut für Informationssysteme, Technische Universität Wien
Favoritenstraße 9-11, A-1040 Vienna, Austria
{eiter,tkren,redl}@kr.tuwien.ac.at

Abstract. Decision diagrams (DDs) are a popular means for decision making, e.g., in clinical guidelines. Some applications require to integrate *multiple* related yet different diagrams into a single one, for which algorithms have been developed. However, existing merging tools are monolithic, application-tailored programs with no clear interface to the actual merging procedures, which makes their reuse hard if not impossible. We present a general, declarative framework for merging and manipulating decision diagram tasks based on a belief set merging framework. Its modular architecture hides details of the merging algorithm and supports pre- and user-defined merging operators, which can be flexibly arranged in merging plans to express complex merging tasks. Changing and restructuring merging tasks becomes easy, and relieves the user from (repetitive) manual integration to focus on experimenting with different merging strategies, which is vital for applications, as discussed for an example from DNA classification. Our framework supports also reasoning over DDs using answer set programming (ASP), which allows to drive the merging process and select results based on the application needs.

1 Introduction

Many medical decisions are based on Decision Diagrams (DDs), which support medical doctors and health care personnel in decision making like to determine the best medication or intervention for a patient depending on her medical history and examinations. Such diagrams are also used for diagnosis as part of computer supported decision systems, cf. [6]. A very common use case frequently found in clinical protocols is to quantify the degree of severity depending on the patient's condition, see eg. [14]; the suggested treatment depends then on the stage. Clearly, DDs are relevant beyond clinical practice and have become popular in economy (e.g. in liquidity rating), psychology (e.g. in tests for personality disorders), and basic life sciences; for example, [12]—which we will consider in more detail below—uses decision trees to classify given DNA sequences into protein coding and non-coding ones. More applications are listed in [2].

Multiple DDs often exist for the same issue, due to various reasons: different institutes working on similar projects, different views of correct decisions, statistical impreciseness, or simply human errors. Making a choice between several DDs, especially from authoritative sources, is notoriously difficult and ignoring expertise captured

[⋆] This research has been supported by the Austrian Science Fund (FWF) project P20840 and P20841, and by the Vienna Science and Technology Fund (WWTF) project ICT 08-020.

in such diagrams is a waste of resources. This requires to integrate multiple diagrams into a single one that should be concise and coherent. Several integration algorithms have been developed [7, 10], and implemented tools exist [13]. However, they are monolithic programs tailored for a particular application, without clear interface between the integration components and other programs parts; reusing the merging procedures is hard if not impossible. Moreover, existing approaches usually produce *one* output diagram, but merging can often be done in various ways, hence it is interesting to develop merging strategies that produce *multiple* diagrams. A diagram can then be selected according to the application needs. This naturally calls for *reasoning* about DDs.

In this paper we present a general, declarative approach which supports semi-automatic integration of multiple DDs into a single one, as well as reasoning about DDs. Both features can also be combined to include user-defined constraints or—more generally—rules that influence the further integration process. To this end, we encode decision diagrams to *belief sets* and transform the *integration of DDs into a belief set merging problem in the generic framework* of [11], which provides merging plans of abstract merging operators to accomplish complex belief merging tasks. *With the MELD system*, which implements the framework via answer set programming (ASP) [3], *we can exploit a rich infrastructure to realize a powerful declarative tool*. It facilitates a range of different DD integration algorithms, allows to formulate complex, operator-based integration tasks in a modular, flexible manner, and offers on top the possibility to use ASP for reasoning about DDs. Specifically, this can be exploited to compute properties of diagrams (like height or number of variables) that are used for filtering results.

We proceed as follows. After fixing a formal model of decision diagrams, we map DDs to belief sets and integration of DDs into a belief set merging in Sec. 3. Reasoning over DDs and support for it in our tool, the *DDM system*, is discussed in Sec. 4. Finally, we consider application for *DNA classification* similar as in [12], with the aim to stress flexibility and user friendliness, and capabilities beyond those of other systems (Sec. 6).

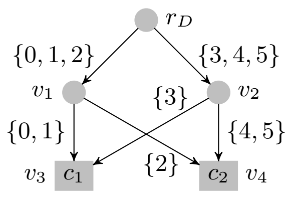
2 Decision Diagrams

We define a *classification function* $c\colon \mathcal{D} \to \mathcal{C}$ as a mapping of some domain $\mathcal{D}$ to a set of class labels $\mathcal{C}$. To represent such a function one can use, e.g., lookup tables, production rules, or decision diagrams. We focus on the latter as they have turned out to be very useful in practice not only because they are comprehensible and easy to explain.

Abstractly, we can define a decision diagram as follows.

Definition 1. *A* decision diagram (DD) *over $\mathcal{D}$ and $\mathcal{C}$ is a labelled rooted directed acyclic graph $D = \langle V, E, \ell_{\mathcal{C}}, \ell_E \rangle$, where V is the set of nodes with unique root node $r_D \in V$ and $E \subseteq V \times V$ is a nonempty set of directed edges. The labelling function $\ell_{\mathcal{C}}$ maps leaf nodes in D to elements from $\mathcal{C}$, and $\ell_E\colon E \to 2^{\mathcal{D}}$ assigns each edge a subset of domain $\mathcal{D}$. We call D a* decision tree *if every node has at most one incoming edge.*

Classifying an element $d \in \mathcal{D}$ is intuitively done by starting at node r_D and following an outgoing edge e iff $d \in \ell_E(e)$. This step is repeated until a leaf node v is reached. Then $\ell_{\mathcal{C}}(v)$ is the class label assigned to d. To guarantee that the classification by some DD D is deterministic and the result is unique, ℓ_E is required to satisfy the following two conditions. In this case we say that D is *valid*:



(a) Graphical representation

$\mathit{root}(r_D)$; $\mathit{inner}(r_D)$; $\mathit{inner}(v_1)$; $\mathit{inner}(v_2)$;
$\mathit{leaf}(v_3, c_1)$; $\mathit{leaf}(v_4, c_2)$;
$\mathit{cond}(r_D, v_1, z, <, 3)$; $\mathit{else}(r_D, v_2)$;
$\mathit{cond}(v_1, v_3, z, <, 2)$; $\mathit{else}(v_1, v_4)$;
$\mathit{cond}(v_2, v_3, z, <, 4)$; $\mathit{else}(v_2, v_4)$

(b) Formal encoding

Fig. 1: A valid decision diagram D

(a) for non-leaf nodes v of D, $\bigcup_{(v,w)\in E} \ell_E(v,w) = \mathit{In}(v)$, where $\mathit{In}(r_D) = \mathcal{D}$ and $\mathit{In}(u) = \bigcup_{(p,u)\in E} \ell_E(p,u)$ for $u \neq r_D$, and
(b) for a node v of D and any successors u, w of v, $\ell_E(v,u) \cap \ell_E(v,w) \neq \emptyset \Rightarrow u = w$.

Condition (a) states that, if a node is reached for element d, there must be an outgoing edge for d, i.e., computation can always continue, while (b) forces branching at internal nodes to be deterministic. In the following we consider valid decision diagrams only.

Example 1. Fig. 1a shows a valid decision diagram D over $\mathcal{D} = \{0,1,2,3,4,5\}$ and $\mathcal{C} = \{c_1, c_2\}$. The edges are marked with ℓ_E, and its leaves with $\ell_\mathcal{C}$. It represents the classification function c s.t. $c(d) = c_1$ if $d \in \{0,1,3\}$ and $c(d) = c_2$ if $d \in \{2,4,5\}$.

For practical purposes it is convenient to realize edge labels by *queries* over some query language. A query $Q(z)$ with free variable z is then a shortcut for all domain elements which satisfy it. E.g., if V is the set of positive integers and we want to distinguish between prime and non-prime numbers, we may label the according edge with $Q(z) = z > 1 \wedge \forall m, n\, (m > 1 \wedge n > 1 \supset z \neq m \cdot n)$ instead of $\{2,3,5,7,11,\dots\}$. In practice, simple queries of form $X \circ Y$ often suffice, where X and Y are constants or attribute values of the domain, and $\circ$ is a comparison operator. For example, if the domain consists of patient records with attributes such as blood values, the query $Q(z) = z.\mathit{TSH} > 4.5 mU/l$ is true for all patients z that have a Thyroid-Stimulating Hormone level larger than 4.5 milli-units per liter. More complex queries involving logical connectives can easily be rewritten to this form by introducing intermediate nodes.

For the further development of our framework we assume that the query language of a DD is fixed. When developing our encoding, we will assume that queries are of the form $X \circ Y$, but it can easily be extended to more complex query languages.

3 Merging of Decision Diagrams

Some applications require to work with multiple DDs. The reasons for this are manyfold: statistical fluctuations, different expert opinions on correct decisions, or simply human errors. For an example merging, see Fig. 4, which is discussed in Section 6. We will see in Sec. 4 that merging operators (as introduced here) may produce multiple output diagrams. Picking one of them over another is not easy to justify, and sometimes it is shearly impossible to have any preference among a variety of diagrams.

Belief Merging. Redl *et al.* [11] developed the ASP-based MELD system for belief set merging tasks.[1] MELD can integrate multiple collections of belief sets using merging operators that are hierarchically arranged in trees, called *merging plans*. They are evaluated bottom-up, and the result is the one at the root.

More in detail, a *belief* $(\neg)p(c_1, \ldots, c_n)$ is a literal (atom or negated atom) where p is a predicate symbol of arity n from a set of predicate symbols Σ_P, and the c_i are constants from a set Σ_C of constant symbols. A *belief set* is any set B of beliefs (wrt. Σ). A *collection of belief sets* $\mathcal{B}$ is any set of belief sets (wrt. Σ); $\mathbb{B}_\Sigma$ denotes the set of all collections of belief sets. A *belief set merging operator* is a function $Op\colon (\mathbb{B}_\Sigma)^k \times \mathcal{A}_1 \times \cdots \times \mathcal{A}_m \to \mathbb{B}_\Sigma$ that assigns each tuple $\hat{\mathcal{B}} = (\mathcal{B}_1, \ldots, \mathcal{B}_k)$ of collections of belief sets $\mathcal{B}_i$ and arguments $A_1, \ldots, A_m$ from domains $\mathcal{A}_1, \ldots, \mathcal{A}_m$ a result collection of belief sets $Op(\hat{\mathcal{B}}, A_1, \ldots, A_m)$; we allow $k = 1$ (in abuse of terminology) to enable also transformations of collections of belief sets. A *merging plan* is any expression built using the operators over *belief bases*, which comprise facts and (optionally) logical rules; each belief base has an associated collection of belief set (its semantics), used for evaluation.

Example 2. We define operator $\circ_\cup^{2,0}$ for consistently integrating two collections $\mathcal{B}_1$ and $\mathcal{B}_2$ of belief sets

$$\circ_\cup^{2,0}(\mathcal{B}_1, \mathcal{B}_2) = \{B_1 \cup B_2 \mid B_1 \in \mathcal{B}_1, B_2 \in \mathcal{B}_2, \nexists A \text{ s.t. } \{A, \neg A\} \subseteq (B_1 \cup B_2)\} \quad .$$

The operator computes the pairwise union of two belief sets B_1 and B_2 from $\mathcal{B}_1$ and $\mathcal{B}_2$, respectively, where classically inconsistent pairs are skipped. Assume $\mathcal{B}_1 = \{\{a, b, c\}, \{\neg a, c\}\}$ and $\mathcal{B}_2 = \{\{\neg a, d\}, \{c, d\}\}$, we get that $\circ_\cup^{2,0}(\mathcal{B}_1, \mathcal{B}_2) = \{\{a, b, c, d\}, \{\neg a, c, d\}\}$. Let $\mathcal{B}_3 = \{\{\neg d, e\}, \{d, e\}\}$, then $\circ_\cup^{2,0}(\circ_\cup^{2,0}(\mathcal{B}_1, \mathcal{B}_2), \mathcal{B}_3)$ is a merging plan which evaluates to $\{\{a, b, c, d, e\}, \{\neg a, c, d, e\}\}$.

We instantiate the framework for decision diagram merging. We obtain an implementation, based on MELD, called the *DDM system*.[1] The basic idea is to

- *encode DDs as belief sets*, described by *belief bases*;
- define *merging operators* for MELD (implemented in C++), tailored to the integration and manipulation of encoded diagrams; and based on them
- declare *merging plans* to integrate and manipulate (convert, optimize, etc.) the encoded DDs.

Encoding. Let $D = \langle V, E, \ell_\mathcal{C}, \ell_E \rangle$ be a decision diagram over domain $\mathcal{D}$ and class labels $\mathcal{C}$. We assume that $\mathcal{D}$ is a set of tuples $(a_1, \ldots, a_n) \in \mathcal{D}_1 \times \cdots \times \mathcal{D}_n$, where $\mathcal{D}_i$, $1 \leq i \leq n$, is the domain of attribute a_i. Informally, $\mathcal{D}$ consists of composed objects, which are described by n attribute values. Then we use the following atoms to encode D:

- $root(r_D)$ for the root node r_D of D;
- $inner(v)$ for inner nodes $v \in V$;
- $leaf(v, c)$ for leaf nodes $v \in V$ with assigned class label $c = \ell_\mathcal{C}(v)$;
- $cond(v, w, a, Op, b)$ for an edge $(v, w) \in E$ with some condition $Q(\mathbf{z}) = a\ Op\ b$ such that $Q(\mathbf{z})$ holds iff $\mathbf{z} \in \ell_E((v, w))$, where a and b are constants or named attributes of $\mathbf{z}$ and Op is a comparison operator;

[1] `http://www.kr.tuwien.ac.at/research/systems/dlvhex/meld.html`

– $else(v,w)$ for an *else-edge* $(v,w) \in E$. It is optional but unique for v and encodes the set of domain elements $\mathcal{D} \setminus \bigcup_{(v,u) \in E, u \neq w} \ell_E((v,u))$.

Formally, the query language has expressions a *Op* b and $else$ for optional else-edges, where $else(v,w)$ is viewed as the conjunction of the negated expressions on all other out-edges of v. Thus, a tuple $t \in \mathcal{D}$ belongs to $\ell_E((v,w))$ iff no condition of some other out-edge of v is true for t. An example encoding that corresponds to the diagram in Fig. 1a is shown in Fig. 1b. This basic encoding can be easily extended to provide more features and support enriched decision diagrams. For instance, we could allow leaf nodes to store additional information besides the class label (we will use this below).

Decision Diagram Merging using Merging Plans. Let $\daleth_{\mathcal{D},\mathcal{C}}$ denote the set of all decision diagrams over domain $\mathcal{D}$ and classes $\mathcal{C}$. We then define (recall that 2^X is the powerset of a set X):

Definition 2. *An n-ary DD merging operator is a function*

$$\circ^n \colon (2^{\daleth_{\mathcal{D},\mathcal{C}}})^n \times \mathcal{A}_1 \times \cdots \times \mathcal{A}_m \to 2^{\daleth_{\mathcal{D},\mathcal{C}}}$$

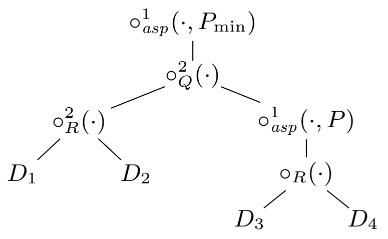
which maps each tuple $\Delta = (\Delta_1, \ldots, \Delta_n)$ of sets of DDs Δ_i (over $\mathcal{D}$ and $\mathcal{C}$) to a set of DDs $\circ^n(\Delta, A_1, \ldots, A_m)$, where $A_i \in \mathcal{A}_i$ are additional arguments from domains $\mathcal{A}_i$ for all $1 \leq i \leq m$.

In our examples, the domains of additional arguments will usually be either the natural numbers or the set of all ASP programs. Def. 2 allows for arranging operators hierarchically in so-called *merging plans* (for an example see Fig. 2a). The merging plan can be evaluated bottom-up, and the final result is the output of the topmost operator. Concrete operators $\circ^n(\Delta, A_1, \ldots, A_m)$ are given e.g. in [7, 10] and in Sec. 6. Allowing operators of arity $n = 1$ enables decision diagram transformations. It is often convenient to transform diagrams into a special form (e.g. trees) prior to integration; this may simplify the implementation of the actual merging operators ($n \geq 2$) enormously.

Each such operator produces an output decision diagram which behaves as if the input classifiers were consulted independently and the results were combined as described below for some predefined operators:

– *majority voting*: the majority of the input diagrams $D_1, \ldots, D_n$ decides;
– *user preference*: wrong decisions may be of different severity. In medical screening tests, e.g., one usually prefers false positives to false negatives: additional tests may refute the former, while the latter let the disease proceed. Thus a natural decision rule could be: "If the input classifiers vote differently, classify as X rather than Y";
– *average*: interpreting decision diagrams as decision boundaries in an n-dimensional feature space, it is natural to compute the (possibly weighted) "average boundary."

Technically, merging plans are declaratively specified in a user-friendly language and may be automatically evaluated by our prototype implementation. The set of predefined operators can be extended by custom ones by implementing an operator-API in C++. For more technical details we refer to the online documentation.



(a) Decision Diagram Merging Plan

$$
\begin{array}{l}
cnt(I,C) \leftarrow LC = \#count\{L : leaf_{in}(I,L,C)\}, \\
\qquad IC = \#count\{N : inner_{in}(I,N)\}, \\
\qquad root_{in}(I,R), C = LC + IC \\
c(I) \leftarrow root_{in}(I,R), \text{not } \neg c(I) \\
\neg c(I) \vee \neg c(J) \leftarrow root_{in}(I,R), root_{in}(J,S), I \neq J \\
leaf(L,C) \leftarrow c(I), leaf_{in}(I,L,C) \\
\vdots \\
else(N_1,N_2) \leftarrow c(I), else_{in}(I,N_1,N_2) \\
\bot \leftarrow M = \#min\{NC : cnt(I,NC)\}, \\
\qquad c(I), cnt(I,C), C > M
\end{array}
$$

(b) $P_{\min}$

Fig. 2: Node count minimization

4 Reasoning about Decision Diagrams

The second major benefit of our formal representation is the *possibility of reasoning about DDs*. Properties of a diagram (e.g., height, variable occurrences, redundancy, etc.) can be computed automatically from an encoding as in Sec. 3. This is particularly interesting when merging and reasoning operators are combined in merging plans. According to Def. 2, merging operators output *sets of* DDs. Hence, when a merging operator encounters a choice point, it may simply produce alternative diagrams. Such choice points can e.g. be leaves with (almost) uniform distributions, i.e., the best classification is not obvious. One can then select the most appropriate diagram by reasoning over the alternatives, resorting e.g. to properties as above. For example, take the redundancy measure defined as the number of indistinguishable nodes. Computationally optimal representations require an appropriate choice for the diagram with minimum redundancy. Another possibility is to prefer DDs with minimum height or minumum number of nodes. This is reasonable if the decision diagram is intended for being used by humans such as medical doctors applying classifiers published in medical guidelines.

To reason about DDs, answer set programs are well-suited for several reasons: (1) transitive closures allow to reason over paths in diagrams, (2) the multi-model semantics allows for producing multiple diagrams (one per answer set), and (3) constraints are useful to rule out inappropriate diagrams, or to account for a cost. Technically, we realize reasoning over diagrams (i.e. "applying" programs to diagrams) by instantiating Def. 2 as a special operator $\circ_{asp}(\Delta, P)$ which can be used as any other operator in the merging plan. The input is a set $\Delta \in 2^{\urcorner_{\mathcal{D},\mathcal{C}}}$ of DDs and an answer set program P. The operator $\circ_{asp}$ encodes all input diagrams in Δ, adds them as facts to the user-defined program P, and returns as result its answer sets. They are expected to contain the encoded output diagrams (one per answer set). ASP is well-suited for this purpose because of its multi model semantics which allows for producing multiple alternative results.

We slightly modify our encoding from Sec. 3: to handle *multiple* diagrams within one set of input facts, we add a diagram index I as first argument to all predicates $p \in \{root, leaf, inner, cond, else\}$ and call them p_{in}; to distinguish between program input diagrams and result diagrams, p_{in} are used for the input, and p denote output predicates.

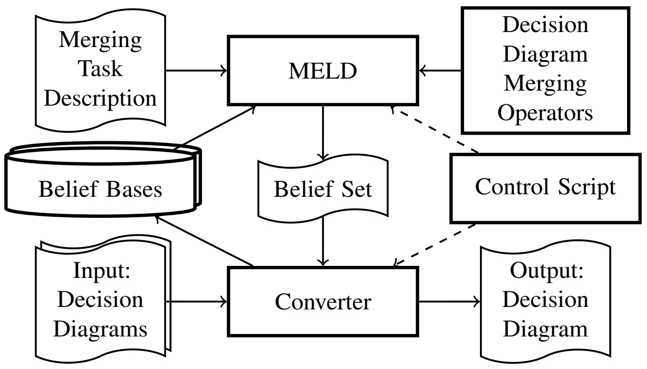


Fig. 3: DDM Architecture (data flow $\rightarrow$, control flow $\dashrightarrow$)

Use Case: Node Count Minimization. The merging plan in Fig. 2a shows four input DDs $D_1, \ldots, D_4$. First, we merge D_1 and D_2, as well as D_3 and D_4 using operator $\circ^2_R$ whose result is subsequently fed into a user-defined program P (potentially, any program can be used that is using the encoding as described above). The result of this and the merge of D_1, D_2 is passed to the next binary merging operator $\circ^2_Q$, which is eventually filtered by $P_{\min}$ as shown in Fig. 2b. Program $P_{\min}$ is intended to select among arbitrarily many input DDs the one with the minimal number of nodes. Let $V(D)$ denote the set of nodes in diagram D. We have the following result.

Proposition 1. *For a set Δ of input decision diagrams, we get that $\circ_{asp}(\Delta, P_{\min})$ yields a set of decision diagrams $\Delta_{\min} \subseteq \Delta$ such that for every $D \in \Delta_{\min}$ there is no $D' \in \Delta$ with $|V(D)| > |V(D')|$.*

Intuitively, $P_{\min}$ filters diagrams with minimal node number. It computes in cnt for each diagram (identified by its root) the total number of nodes (rule 1). Then it selects non-deterministically exactly one input diagram at a time (rule 2–3) and copies it to the answer set (rules 4–); to minimize the node count, answer sets representing non-minimal diagrams are eliminated in the last integrity constraint. In practice, the selection criteria might be more involved. In Sec. 6 we will (abstractly) propose a program P_{sel} which tests the input diagrams over some test set and selects the diagram with the best behavior.

5 Prototype Implementation of the DDM System

The architecture of our prototype (see Fig. 3) consists of the following parts.[2]

MELD. This is the underlying belief set merging system. It is implemented on top of the logic programming reasoner DLVHEX, which evaluates HEX programs; for details we refer to Redl et al. ([11]). Our DDM system extends MELD by two major components: a *converter* between different forms of decision diagram representation, and a *suite of*

[2] `http://www.kr.tuwien.ac.at/research/systems/dlvhex/ddm.html`

decision diagram merging operators, which are plugins for the MELD system. Further components are the Merging Task Description and the Control Script.

Converter. MELD expects decision diagrams in the belief set encoding from above; the Converter transforms human-readable input and output formats of machine learning tools (which realize hierarchical structures) into corresponding belief bases (sets of facts in HEX format). Our implementation, graphconverter, currently supports (1) a graph-based input format, (2) the output format of the machine learning tool RapidMiner (`http://rapid-i.com`), (3) a representation as logic program or as answer set. graphconverter takes two arguments that specify the input resp. output format; it reads from standard input and writes to standard output.

Merging Operators. At the core of our DDM system is a suite of merging operators which interpret their input as encoded decision diagrams. We provide some predefined operators, which are mostly considered to serve as examples for demonstrating the possibilities. Users may use them directly, refine them to make them suitable for a certain application, or develop completely new operators as plugins (in C++).

Merging Task Description. The merging plan, the decision diagrams and the merging operators used, is stored as a *merging task description* in an `.mt` file, say `task.mt`, formulated in MELD's *merging task language (MTL)*. The merging operators are tailored to decision diagrams only, i.e., they assume that belief sets associated with belief bases encode decision diagrams (otherwise an error is raised). The merging task can be initiated by invoking the command

```
$ dlvhex --merging task.mt > res.as
```

storing the output in file `res.as`. (Note that `dlvhex --merging` invokes MELD.) The result is another diagram represented by the facts in the belief set.

Control Script. A simple control script, as used in the examples of the system, manages the workflow of executing a merging task. It converts the input diagrams, stored in files `diagN.X` (the filename extension `X` tells the input diagram type) to belief bases in files `diagN.hex`. It then calls MELD as above, and finally converts the obtained diagram (represented by a set of facts given by `res.as`) into the input format `X`; e.g., for `dot` files it calls

```
$ graphconverter as dot < res.as > res.dot
```

Further details on system usage input and MTL format description is given at the accompanying homepage.[2]

6 Example: DNA Classification

A central task in (semi-)automatic generation of protein databases is to recognize genes in DNA sequences. Recall that DNA molecules are composed of the four bases (A)denine, (G)uanine, (C)ytosine and (T)hymine which are lined up in vast strings. In reproduction, only a minor part of the total DNA will be transcribed as most of it is *junk DNA* not encoding proteins. To construct protein databases like SWISSPROT (`http://expasy.`

org/sprot/) requires to automatically classify sequenced DNA into (protein) *coding* and *non-coding* (junk DNA) samples.

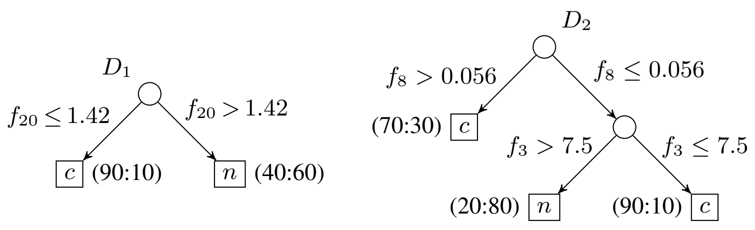
To this end, one usually computes *numeric features* for a set of annotated training sequences. They incorporate knowledge from molecular biology which allows to discriminate—with some level of uncertainty—between the two classes. For instance, it is known that the predominant bases at the first codon position in coding sequences are purines (A and G), while in non-coding sequences the distribution is rather random. Hence, a useful feature is the relative frequency of A and G on the first codon position with respect to the number of codons in a sequence. For details we refer to [8]. Feature vectors can then be used to train a classifier using machine learning techniques.

We instantiate Def. 1 with $\mathcal{C} = \{c, n\}$ ($\underline{c}$oding and $\underline{n}$on-coding) and use the query language in Sec. 3 for the edge labelling. We extend our basic DD model by an additional frequency distribution $\ell_F(v) = (c_v, n_v)$ at each node v, which tells the number of coding (c_v) and non-coding (n_v) samples (in fact, below we will use it only at leaf nodes). E.g., if in leaf l we have that 70 out of 100 training samples were coding, the classification is $\ell_{\mathcal{C}}(l) = c$ with frequency distribution $\ell_F(l) = (70, 30)$.

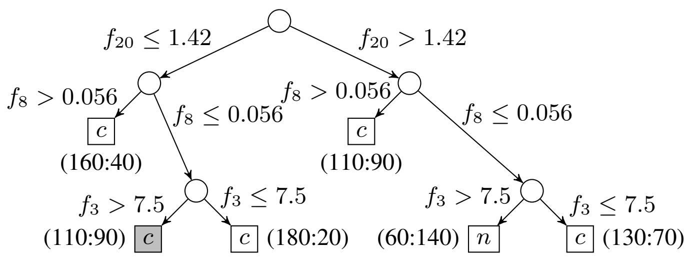
MORGAN Merging Operator. The MORGAN system [13] trains *multiple* decision trees $D_1, \ldots, D_n$ and merges them afterwards (see Fig. 4). The class of a yet unseen sample s is then determined as follows. First s is classified by each tree D_i, ending in leaf l_i. Then all frequency distributions $\ell_F(l_i)$ are summed up by component-wise vector addition, and the class with the largest count is the final classification. For instance, if we have two classifiers which yield the distributions $\ell_F(l_1) = (90, 10)$ and $\ell_F(l_2) = (20, 80)$, they are added to $(110, 90)$, and consequently the final classification is c (since $110 > 90$). The implementation of the binary case is straightforward as follows: Each leaf node of diagram D_1 is replaced by a copy of the diagram D_2. Then the frequency distribution of each new leaf node is recomputed, and the class label is set according to the highest component. More than two diagrams are integrated by iteration.

We implemented MORGAN's merging strategy as operator $\circ_M$ for our DDM system. It takes as input two singleton collections of belief sets (general diagrams must be converted to trees, for which our system provides operators). The output will be another classifier behaving as the suggested procedure, but represented by a new decision tree. Fig. 4c shows the diagram after application of another operator $\circ_{simp}$ from our system, which simplifies the input diagram by eliminating unnecesary branches and reusing equivalent subdiagrams.

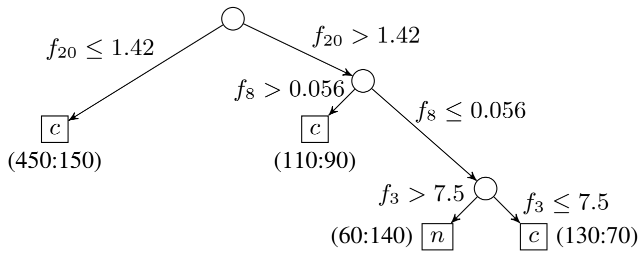
Experiments with Decision Diagram Merging. Concerning the accuracy increase, our most impressive results were achieved with three different decision trees, trained by the open-source machine learning tool RapidMiner. The variations concerned both the selected algorithm and the training samples. The training sets of only ten sequences per tree was drawn randomly from a set of $4,000$ sequences ($2,000$ coding and $2,000$ non-coding) from [5]. The intuition was to obtain trees that involve at most two attributes (with the greatest variance between coding and non-coding sequences). These attributes depend on the training set and the selected learning technique. The performance of these trees was tested with $2,000$ test instances ($1,000$ coding and $1,000$ non-coding) outside the training set. As expected, the results were very poor due to the very small training set. Table 1 shows the overall performance which is about 50%, or in other words, as



(a) Two decision diagrams D_1 and D_2



(b) Merged decision diagram $D_1 \circ_M D_2$



(c) Simplified decision diagram $\circ_{simp}(D_1 \circ_M D_2)$

Fig. 4: Classifiers for coding (c) and non-coding (n) DNA sequences, based on features f_i

good as random classification. An interesting observation is that the first classifier tends towards *non-coding* and the second towards *coding*; the third is slightly better balanced, i.e., ratio of false positives and false negatives is smaller.

The merged tree, produced by the described merging procedure, performs surprisingly good. The overall accuracy was 65.25%, which is much better than any of the source classifiers. Recall that we used only very few (ten) training examples to train the individual decision trees; in total we had 30 samples. In experiments we found that about $1,000$–$2,000$ training examples are needed to reach this accuracy with a single-set decision tree. Furthermore, such trees had depth $\approx$7, which is much larger than height 3 of the merged tree. This accuracy cannot be enhanced much by using more training samples or source classifiers. Empirical results for different algorithms show that $\approx 75\%$ is the best one can expect, which seems to be a limit of the statistical features [15].

Our findings in experiments with DNA data from the Human Genome Project largely confirm those of [13]. Compared to training single diagrams, the merging approach

Table 1: DNA Classification Results (Data from Human Genome Project)

	Input 1		Input 2		Input 3		Merged	
	TC	TN	TC	TN	TC	TN	TC	TN
PC	175	214	854	877	262	346	565	260
PN	825	786	146	123	738	654	435	740
A	48.05%		48.85%		45.80%		65.25%	

PC/PN: predicted coding/non-coding, TC/TN: true coding/non-coding, A: accuracy

– often yields a simpler diagram structure (in particular height);
– often gains the same accuracy with a smaller (overall) training set; and
– can use parallel training (also with different methods).

We stress that qualitative improvements (as targeted in machine learning) was *not* the primary goal of our research, and thus we omit detailed statistical results here. Instead, our contribution is the *methodology and tool support for flexible DD merging*: many experiments and trials are needed to obtain the above results, and this is only reasonably possible with a tool allowing to quickly restructure the modular merging plans. This makes our system more powerful than, e.g., MORGAN, which uses a hard-coded merging *procedure*. We can switch from MORGAN's merging strategy to majority voting by changing one line in the merging plan; more than two input diagrams can be merged hierarchically, possible using different operators. Furthermore, different from MORGAN operatores can be reused for other applications.

Extending the Scenario. In addition, a major advantage when using our declarative approach is the *possibility to reason about DDs* between the merging steps. While $\circ_M$ returns exactly one output diagram for two input diagrams [13], the following variant $\circ_{M'}$ seems reasonable (for space reasons, we omit formal details): when merging two leaf nodes l_1 and l_2 with frequency distributions (c_1, n_1) and (c_2, n_2), the merged node l gets $\ell_F(l) = (c_1 + c_2, n_1 + n_2)$. While $\circ_M$ classifies l as coding (c) if $c_1 + c_2 > n_1 + n_2$, and non-coding (n) otherwise, it makes sense to produce *both* alternatives if $\left|\frac{c_1+c_2-(n_1+n_2)}{c_1+c_2+n_1+n_2}\right| < \epsilon$ for some threshold $\epsilon > 0$, i.e., the numbers of coding and non-coding samples are almost equal.

This strategy intends to avoid overfitting by estimating and minimizing the generalization error of the classifier, which is known as the *model selection problem* in machine learning (for more information, see [1]). Moreover, it may even lead to different diagram structures after simplification. Applying $\circ_{simp}$ to the diagram in Fig. 4b gives the diagram in Fig. 4c. In contrast, when the label of the shaded leaf node is switched from c to n, the left and the right subtree of the root become equivalent. Therefore $\circ_{simp}$ will eliminate the unnecessary branching at the root, and D_2 is reproduced.

A declarative choice program P_{sel} can then select, one of the alternatives. This program may prefer the diagram which performs best over some test set, or it prefers diagrams with a simpler structure or lower number of nodes. Our contribution in this regard is *tool support for convenient generation and selection of best merges* by declarative merging plans, e.g., $\circ_{asp}(\circ_{simp}(\circ'_{M}(\{D_1\},\{D_2\})), P_{sel})$, where $\circ'_{M}$ (or any other operator) constructs candidates, and $\circ_{asp}$ makes the final selection. In the DNA application, this strategy led to sensible differences in the resulting diagrams (yet not to a significant increase in best precision).

7 Related Work and Conclusion

The integration of several classifiers is known in machine learning as *ensemble learning*, for which well-working methods are available; see e.g. [4, 9] for an overview. However, these approaches *train* new classifiers using an existing one and training samples. In decision diagram merging, we *directly integrate them* without using training samples at all. This strategy was also discussed in [13] where an algorithm and a tool for integrating decision diagrams for DNA classification was developed. We have discussed this scenario (and extensions). Their system, however, is monolithic, hard-coded, and tailored to this application. Our DDM system, instead, is more general and can be used for different tasks as well. Its modular architecture simplifies the exchange, reusability and modification of merging strategies enormously. This is especially useful for experimenting with different strategies and evaluating their outcomes empirically.

The real strength of our system becomes visible when combining merging capabilities with *declarative reasoning* about decision diagrams between the merging steps. User-defined ASP programs may be used on a meta-level to constrain the further integration process. This allows merging operators to produce *multiple* alternative results. The ASP program can in turn select one which is appropriate for the application in mind. In particular, the high expressivity of HEX programs and the possibility to access other software from them offers support to declare involved criteria.

Concerning complexity issues, both the time complexity of merging and the size of the integrated diagram depends on the merging operators in use. The analysis of concrete operators remains for future work.

References

1. Arlot, S., Celisse, A.: A survey of cross-validation procedures for model selection. Statist. Surv. 4, 40–79 (2010)
2. Bahar, R., Frohm, E., Gaona, C., Hachtel, G., Macii, E., Pardo, A., Somenzi, F.: Algebraic decision diagrams and their applications. In: ICCAD'93. pp. 188–191. IEEE (1993)
3. Brewka, G., Eiter, T., Truszczyński, M.: Answer set programming at a glance. Commun. ACM (2011), to appear
4. Dietterich, T.G.: Ensemble methods in machine learning. In: Intl. Workshop Multiple Classifier Systems. pp. 1–15. Springer (2000)
5. Fickett, J.W., Tung, C.S.: Assessment of protein coding measures. Nucleic Acids Res. 20(24), 6441–6450 (1992), `http://fruitfly.org/sequence/human-datasets.html`
6. Mair, J., Smidt, J., Lechleitner, P., Dienstl, F., Puschendorf, B.: A decision tree for the early diagnosis of acute myocardial infarction in nontraumatic chest pain patients at hospital admission. Chest 108(6), 1502–1509 (1995)
7. Naylor, B., Amanatides, J., Thibault, W.: Merging BSP trees yields polyhedral set operations. In: SIGGRAPH'90. pp. 115–124. ACM (1990)

8. Peng, H., Long, F., Ding, C.: Feature selection based on mutual information: Criteria of max-dependency, max-relevance, and min-redundancy. IEEE Trans. Pattern Anal. Mach. Intell. 27(8), 1226–1238 (2005)
9. Polikar, R.: Ensemble based systems in decision making. IEEE Circuits Syst. Mag. 6(3), 21–45 (2006)
10. Quinlan, J.R.: Simplifying decision trees. Int. J. Hum.-Comput. St. 51(2), 497 – 510 (1999)
11. Redl, C., Eiter, T., Krennwallner, T.: Declarative belief set merging using merging plans. In: PADL'11. pp. 99–114. Springer (2011)
12. Salzberg, S.: Locating protein coding regions in human DNA using a decision tree algorithm. J. Comput. Biol. 2(3), 473–485 (1995)
13. Salzberg, S., Delcher, A.L., Fasman, K.H., Henderson, J.: A decision tree system for finding genes in DNA. J. Comput. Biol. 5(4), 667–680 (1998)
14. Sobin, L., Gospodarowicz, M., Wittekind, C. (eds.): TNM classification of malign tumors. Wiley-Blackwell, 7 edn. (2009), `http://www.uicc.org`
15. Sree, P.K., Babu, I.R.: Identification of protein coding regions in genomic DNA using unsupervised FMACA based pattern classifier. Int. J. Comput. Sci. Network Secur. 8(1), 305–309 (2008)