Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS

Matej Vitásek

Inference of XML Integrity Constraints

Department of Software Engineering

Supervisor of the master thesis:	RNDr. Irena Mlýnková Ph.D.
Study programme:	Informatika
Specialization:	ISS

Praha, 2011

I would like to thank everyone participating in the jInfer project for creating a solid foundation for this work, namely Irena Mlýnková, Michal Klempa, Mário Mikula, Robert Smetana and Michal Švirec. The thanks goes again to Ms. Mlýnková for supervising this work, as well as everyone reviewing it, in no particular order Lýdia Švagerková, Michal Švagerka, Zuzana Masárová, Michal Kesely and others^{HB}.

Název práce: Odvozování integritních omezení v XML

Autor: Matej Vitásek

Katedra: Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Irena Mlýnková Ph.D.

Abstrakt: Tato práce navazuje na dřívější pokusy odvodit (inferovat) schéma existujících XML dokumentů. Jelikož je odvozování *struktury* již relativně dobře popsáno, soustředíme se na integritní omezení. Několik jich popisujeme, pozornost pak soustředíme na ID/IDREF/IDREFS atributy z DTD. Na bázi článku od Barbosa a Menelzon (2003) stavíme heuristický přístup k problému hledání optimální sady ID atributů, jeho funkčnost a vhodnost pak ověřujeme na škále experimentů.

Klíčová slova: XML, ID atributy, odvozování

Title: Inference of XML Integrity Constraints

Author: Matej Vitásek

Department: Department of Software Engineering

Supervisor: RNDr. Irena Mlýnková Ph.D.

Abstract: In this work we expand upon the previous efforts to extract (infer) schema information from existing XML documents. We find the inference of *structure* to be sufficiently researched and focus further on *integrity constraints*. After briefly introducing some of them we turn our attention to ID/IDREF/IDREFS attributes in DTD. Building on the research by Barbosa and Menelzon (2003) we introduce a heuristic approach to the problem of finding optimal ID set. The approach is evaluated and tuned in a wide range of experiments.

Keywords: XML, ID attributes, inference

Contents

Preface

Along with technologies such as SQL/noSQL[1] databases, proprietary binary file formats, plain-text configuration files and JSON[2], XML is one of the leading formats for storing structured data. However, even though languages such as DTD and XML Schema[3] to describe XML structure exist for a long time, most of the documents use outdated or no schema at all [VMP08]. To tackle this problem one may employ reverse-engineering techniques to infer the schema from existing documents, such as those described in [Aho96, BNV08, Vyh09]. In particular, [KMS+11a] introduces the jInfer schema inference framework, dealing primarily with the structural parts of the schema: how all the elements, attributes and text data are to be organized in an XML document conforming to that schema. Inference of this kind of structural information was greatly improved in [Kle11].

But the schema is not the only constraint that can be imposed on an XML document. Any textual or numerical value featured in the document may be subject to type constraints, such as the requirement to conform to a specific regular expression. Furthermore, the concept of *keys* and *foreign keys*, well known from the relational database world, applies to schemas as well and will be the topic of this work. One could go even further and try to find even more sophisticated relations in the data, such as *functional dependencies* researched in [Š11].

From all the constraints that can be applied to an XML document by means of its schema, this work will focus on keys and foreign keys. Most important concepts in this field are introduced in [BDF+01] and formalized in the notions of `ID`/`IDREF`/`IDREFS` attributes in DTD and XSD and `xs:key`/`xs:keyref` structures in XSD (both in [BPM+08]).

[1]noSQL: collection of non-relational database technologies, `http://nosql-database.org/`.
[2]JSON: JavaScript Object Notation, lightweight data format, `http://www.json.org/`.
[3]DTD and XML Schema: 2 most prominent XML schema languages, [BPM+08]

The scope of this work is finally limited to the inference of `ID`/`IDREF`/`IDREFS` from existing XML documents. `ID` attributes were chosen over `xs:key` because the preliminary research found that while it is possible to find real-life XML data with schemas containing `xs:key` structure, schemas with `ID` attributes are much more common and it is much easier to obtain large data sets for experiments.

Structure of the thesis

The thesis will be structured as follows.

In Chapter 1 we introduce a few notions required throughout the work, such as XML tree, `ID` attributes, ID sets, linear programming and the mixed integer problem.

Then in Chapter 2, we review approaches to ID attribute search from previous articles on this topic and formulate the problem of finding the optimal ID set.

This will lead us to the NP-complete problem of Maximum Independent Set, where we will inspect the approaches to solving it in Chapter 3.

We will discuss a closely related Mixed Integer Problem and show that by solving MIP we can solve the Maximum Independent Set and thus the original problem of optimal ID set.

Afterwards, we will show how to use an external MIP solver and demonstrate that this can take too much time. Next we show how to use a heuristic approach to find good solutions much faster.

An extension to jInfer for finding `ID` attributes using MIP solver and a combination of heuristics will be presented and experimentally evaluated in Chapter 4.

Conventions

As usual, source code excerpts, class, field and method names shall be written in fixed-width font, such as `getHeuristic()`. Names of specific heuristics will be written like *`Mutation`*. Name of test data sets will be written like *`OVA1`*.

Pseudocode examples such as the one in Listing 1 will always be presented in a functional way, with inputs and outputs of the function clearly marked in the beginning.

Algorithm 1 Example Algorithm

```
Input: $I$ input data
Input: $n$ maximum number of iterations
Output: results found
  for $i = 1 \to n$ do
    // try to find a solution
    $attempt \leftarrow$ calculate possible solution from $I$
    if $attempt$ is a valid solution then
      return $attempt$
    end if
    return "solution not found"
  end for
```

There is a list of abbreviations following the bibliography in Listing 5.

Please note that throughout this work we will be explicitly ignoring the $\mathcal{O}()$ complexities of algorithms we use. This is because the algorithms we use are by principle strongly stochastic and their performance often depends on behavior of external tools, which we regarded as black boxes and mostly ignored their inner workings.

1. Definitions

1.1 XML Tree

We shall use the representation introduced in [BM03], where an XML file is represented by a labeled tree consisting of nodes for elements, attributes and simple text data. Parent nodes are connected to child nodes with the edges. This tree shall be called an *XML tree*. For a given node v of an XML tree we define $label(v)$ (name of the node in in the document, only for elements and attributes), $id(v)$ (unique identifier across the document) and $value(v)$ (text content, only for attributes and simple text data) in the same way as the cited article does.

Without a loss of generality we ignore the actual ordering of nodes in the tree.

Example This example introduces an XML file fragment that will be used for demonstration throughout this work. An XML tree representing it is in Figure 1.1(a), where each node is annotated with a triple $label(v) : id(v) : value(v)$.

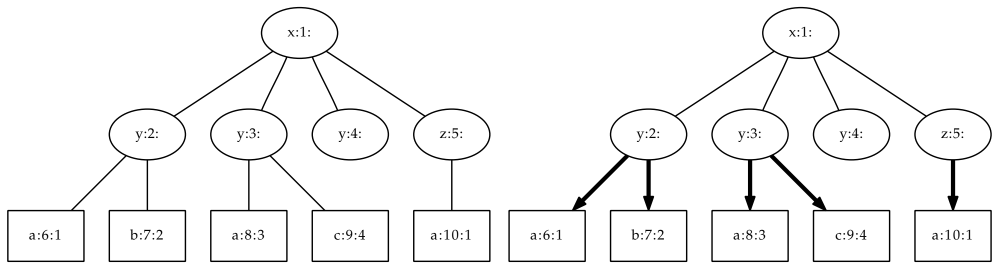
```
<x>
  <y a="1" b="2"/>
  <y a="3" c="4"/>
  <y/>
  <z a="1"/>
</x>
```

Furthermore, we denote $\mathcal{I}$ the set of all ids and $\mathcal{V}$ the set of all values in the document. We will need two more definitions from the article.

Definition 1.1 (Node equality). v_1 and v_2 are *node equal*, written $v_1 =_n v_2$ iff $id(v_1) = id(v_2)$.

Definition 1.2 (Value equality). v_1 and v_2 are *value equal*, written $v_1 =_v v_2$ iff $value(v_1) = value(v_2)$.

Figure 1.1: Example XML Tree



(a) XML tree (b) Attribute Mappings

1.2 `ID`, `IDREF`, `IDREFS` Attributes

According to [BPM+08], an XML attribute may have the type `ID`, `IDREF` or `IDREFS` (among others). The following constraints are related to these types.

Validity constraint: `ID`

Values of type `ID` MUST match the Name production. A name MUST NOT appear more than once in an XML document as a value of this type; i.e., `ID` values MUST uniquely identify the elements which bear them.

Validity constraint: One `ID` **per Element Type**

An element type MUST NOT have more than one `ID` attribute specified.

Validity constraint: `ID` **Attribute Default**

An `ID` attribute MUST have a declared default of `#IMPLIED`[1] or `#REQUIRED`[2].

Validity constraint: `IDREF`

Values of type `IDREF` MUST match the Name production, and values of type `IDREFS` MUST match Names; each Name MUST match the value of an `ID` attribute on some element in the XML document; i.e. `IDREF` values MUST match the value of some `ID` attribute.

[1]`#IMPLIED` means that the attribute has a specified default value.
[2]`#REQUIRED` means that the attribute cannot be empty.

1.3 Attribute Mappings

Now we return to [BM03] to define the notion of an *attribute mapping* (or AM for short). We will use a different definition (without introducing keys from [BDF$^+$01]) that will however give us the same result.

Definition 1.3 (Σ^E, Σ^A, Σ)**.** Σ^E is the set of all element labels, Σ^A is the set of all attribute labels. $\Sigma = \Sigma^E \cup \Sigma^A$ is their union and effectively the set of all labels in the document.

Definition 1.4 (Attribute mapping)**.** For $x \in \Sigma^E$ and $y \in \Sigma^A$ we define the *attribute mapping* of y over x, denoted M_x^y, the $\mathcal{I} \times \mathcal{I}$ relation defined by

$$M_x^y = \{(z, w) : label(z) = x, label(w) = y, parent(w) = z\}$$

Thus the relation M_x^y contains edges in the XML tree connecting element nodes labeled x and attribute nodes labeled y.

We can use projection to retrieve all the unique *ids* of either elements or attributes from the relation, with notation $\pi_E(M_x^y)$ and $\pi_A(M_x^y)$.

Definition 1.5 (Type of the attribute mapping)**.** Attribute mapping M_x^y is of the *type* $\tau(M_x^y) = x$.

Example The XML tree from Figure 1.1(b) has the following non-empty AMs drawn in bold lines: $M_y^a = \{(2, 6), (3, 8)\}$, $M_y^b = \{(2, 7)\}$, $M_y^c = \{(3, 9)\}$ and $M_z^a = \{(5, 10)\}$.

The following example equations hold.

$$\begin{aligned} \pi_E(M_y^a) &= \{2, 3\} \\ \pi_A(M_z^a) &= \{10\} \\ \tau(M_y^c) &= y \end{aligned}$$

Definition 1.6 (Image of the attribute mapping)**.** *Image* ι of the attribute mapping M_x^y is defined as $\iota(M_x^y) = \{z : z = value(w), w \in \pi_A(M_x^y)\}$

So the image of an AM is a set of all the values of all the attribute nodes contained in the mapping.

Example Again referring to the XML tree from Figure 1.1, we get the following AM images.

$$\begin{aligned}\iota(M_y^a) &= \{1,3\}\\ \iota(M_y^b) &= \{2\}\\ \iota(M_y^c) &= \{4\}\\ \iota(M_z^a) &= \{1\}\end{aligned}$$

Attribute Mapping Model An attribute mapping model is a data structure containing the information about all the AMs in a document, together with their images. We shall use this notion later in experimental part of this work.

Definition 1.7 ($name()$)**.** Given an attribute mapping $m = M_x^y$, $name(m)$ shall be defined as the string $x - y$.

1.4 ID Set

Based on the requirements for an `ID` attribute from Section 1.2 we will define ID set with the help of the following definition.

Definition 1.8 (Candidate attribute mapping)**.** An attribute mapping m is a *candidate attribute mapping* if it is an injective function, that is,

$$|m| = |\pi_E(m)| = |\pi_A(m)| = |\iota(m)|$$

Example In our example all the attribute mappings are candidate AMs.

Now we can proceed to define an ID set.

Definition 1.9 (ID set)**.** A set of candidate attribute mappings $I = \{m_1, \dots m_n\}$ is an *ID set* iff

$$\bigcap_{m_i \in I} \tau(m_i) = \emptyset \wedge \bigcap_{m_i \in I} \iota(m_i) = \emptyset$$

That is, an ID set has images without repeating values and all the types are unique (an element cannot have more than one `ID` attribute).

Example Returning to our example, the following are all the possible ID sets: $\{M_y^a\}, \{M_y^b\}, \{M_y^b, M_z^a\}, \{M_y^c\}, \{M_y^c, M_z^a\}$. Note that once we select an AM of type y we can never add any other with the same type. Note also that $\{M_y^c, M_z^a\}$ is not an ID set, because $\iota(M_y^c) \cap \iota(M_z^a) \neq \emptyset$.

`IDREF` and `IDREFS` Condition

Given an ID set I, the requirements from Section 1.2 give us the following condition for an attribute mapping m to be marked `IDREF`.

$$\iota(m) \subseteq \bigcup_{m_i \in I} \iota(m_i)$$

Furthermore if m contains multivalued attributes, it is to be marked `IDREFS`.

1.5 Attribute Mapping Weight

This definition of weight for AMs or AM sets comes from [BM03] again.

Let $M = \{m_1, \ldots m_i\}$ be the set of all non-empty AMs in the document.

1.5.1 Support

Definition 1.10 (Support). *Support* of an attribute mapping m is defined as follows.

$$\phi(m) = \frac{|m|}{\sum_{p \in M} |p|}$$

> The support of attribute mapping M_x^y is the fraction of edges in the XML tree that connect x elements to y attributes.

Example Support of M_y^a in our example is $2/(2+1+1+1) = 0.4$. Support of every other mapping is $1/(2+1+1+1) = 0.2$.

1.5.2 Coverage

Definition 1.11 (Coverage). *Coverage* of an attribute mapping m is defined as follows.

$$\chi(m) = \left(\sum_{p \in M, p \neq m} |\iota(m) \cap \iota(p)| \right) / \sum_{p \in M} |\iota(p)|$$

> The coverage of an attribute mapping measures how much of the image of that mapping occurs elsewhere, as a fraction of all mappings images in the document.

Example Coverage of M_y^a in our example is $(0 + 0 + 1)/(2 + 1 + 1 + 1) = 0.2$. Coverage of M_z^a is 0.2 as well, all the other mappings have coverage 0.

Weight of an attribute mapping is then defined as a linear combination of its support and coverage.

Definition 1.12 (Weight). For $\alpha, \beta \geq 0$ as relative priorities of support and coverage we define the AM *weight* as follows.

$$weight(m) = \alpha.\phi(m) + \beta.\chi(m)$$

For a set of AMs (which may or may not be an ID set) $S = \{m_1, \dots m_i\}$ we define the weight of this set as the sum of the weights of its AMs.

$$weight(S) = \sum_{m \in S} weight(m)$$

Note that this definition of weight is quite arbitrary and all the algorithms mentioned later could easily work with AM weight defined in any other way, even for example defined interactively by the user.

1.6 Independent Set

We shall need the notion of an indepentent set (IS) of vertices in a graph and its weighted variant.

Definition 1.13 (Independent set)**.** Given an undirected graph $G = (V, E)$, a set of vertices $I \subseteq V$ is an *independent set*, iff the following holds.

$$\forall v_1, v_2 \in I, v_1 \neq v_2 : (v_1, v_2) \notin E$$

Definition 1.14 (Maximum weighted independent set)**.** Given an undirected graph $G = (V, E)$ and a weight function $w : V \to \mathbb{R}$, an independent set I_{max} is the *maximum weighted independent set*, iff the following is satisfied.

$$\forall I' \subseteq V, I' \text{is an independent set} : \sum_{v \in I'} w(v) \leqslant \sum_{v \in I_{max}} w(v)$$

It is well known that finding the maximum weighted IS is an NP-hard optimization problem, [JR86].

1.7 Linear Programming

The problem of *linear programming* is optimization of a linear function under a set of linear constraints. The formulation is usually called a *linear program*.

It can be written in the following form (minimization version is possible too).

$$\begin{aligned} \max_{x} z &= \mathbf{c}^{\mathrm{T}}\mathbf{x} \\ s.t.\ A\mathbf{x} &\leqslant \mathbf{b} \\ \mathbf{x} &\geqslant 0 \end{aligned}$$

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix}, \mathbf{c} = \begin{bmatrix} c_1 \\ c_2 \\ \dots \\ c_n \end{bmatrix}, \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \dots \\ b_m \end{bmatrix}$$

Where $\mathbf{x}$ is the vector of variables (to be found by the optimization), $\mathbf{b}$ is the vector and A its accompanying matrix of constraints and $\mathbf{c}$ is the vector of coefficients for the objective function. $\mathbf{x}$ and $\mathbf{c}$ have length n, $\mathbf{b}$ has length m and A has dimensions $m \times n$. Furthermore,

$$\max_x f(x)$$

means maximize the value of $f(x)$ by changing the value of x.

Another way to write this formulation is in the following.

$$\begin{aligned}
\max_x z = \sum_{i=1}^{n} c_i x_i & \\
s.t. & \\
a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &\leqslant b_1 \\
a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &\leqslant b_2 \\
\ldots & \\
a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n &\leqslant b_m \\
x_i \geqslant 0, i = 1, \ldots n &
\end{aligned}$$

Solving a linear program is usually possible in polynomial time using the *simplex algorithm* described for example in [Dan98].

1.8 Mixed Integer Problem

Definition 1.15 (Mixed integer problem)**.** MIP, or *mixed integer problem*, is an instance of linear programming in which some or all variables are limited to integral or boolean (0, 1) values.

Solving MIP in general is NP-hard.

2. Related Work

According to the article [BM03, Chapter 4], the problem of finding an ID set with weight more than some K (K-IDSet) is in NP. Furthermore, the independent set (IS) problem can be reduced to K-IDSet, meaning K-IDSet is NP-hard and thus NP-complete.

The transformation from IS problem formulation to K-IDSet problem formulation is as follows.

> Let $G = (V, E)$ be a simple connected graph with vertex set $V = \{v_1, \ldots, v_n\}$, and edge set $E = \{e_1, \ldots, e_m\}$. We define the attribute mappings as follows. Let $\mathcal{I} = V \cup E$, and define $value(x) = x, x \in \mathcal{I}$. For each vertex $v_i \in V$, we create a mapping $m_i = \{(v_i, e_j) : e_j \in E \text{ is incident on } v_i\}$, and define $\tau(m_i) = v_i$; let $C = \{m_1, \ldots, m_n\}$ be set of all such mappings. It is clear that G has an independent set of size K iff C has an ID set of size K. Also, C can be constructed in time polynomial on $n + m$.

The article continues by proving that the problem of finding maximum weighted IS can be reduced to the problem of finding an ID set with maximum weight (Max-IDSet). This again means that Max-IDSet is NP-complete and, furthermore, unless $\mathsf{P} = \mathsf{NP}$, Max-IDSet has no constant factor approximation algorithm.

The difference in transformation from maximum weighted IS to Max-IDSet is as follows.

> [...] with the added restriction that $w(m_i) = w(v_i), v_i \in V$.

Note that the transformation works in both ways: it is equivalently possible to create a maximum weighted IS instance for a given Max-IDSet instance.

The article further suggests a heuristic approach described in Section 3.4.1, which was incorporated into the framework proposed by this work.

To the best of our knowledge, there are no other articles dealing with this problem.

Fajt in [Faj10] lists several algorithms to find XML keys (see [BDF$^+$01]) in existing XML documents and introduces some new ones. XML keys found this way, being quite different from `ID` attributes, can under some circumstances (they have to be simple) be translated to an equivalent `ID` attribute definition. The process is described in [Vli02, Ch. 9, s. 3]. This opens a new line of possible research: finding XML keys using an algorithm modified to look only for *useful* keys and then converting them to `ID` attributes.

In our work we find the `ID` attributes directly. Even though we can convert them to XML keys by the process mentioned above, we are unable to find more complex keys this way.

Maximum weigthed IS is a well researched topic with a lot of known direct or approximation algorithms, see e.g. [JR86] or [FGK09].

3. MIP Approach

In this chapter we introduce a new approach to finding maximum ID sets. First, we transform the problem formulation to maximum weighted IS problem formulation. Then we transform this into a MIP formulation, and demonstrate how this can be solved using a solver such as GLPK [glp]. We will continue by applying heuristical approaches to improve the performance of the process.

3.1 ID Set to IS Formulation

Given $C = \{m_1, \ldots, m_n\}$ a set of all AMs in a document, we construct a graph $G = (V, E)$ as follows. For each AM $m_i \in C$ we create a vertex $v_{name(m_i)}$. Two vertices $v_{name(m_i)}$ and $v_{name(m_j)}$ shall be connected by an edge iff they cannot share the same ID set, either because they have the same type ($\tau(m_i) = \tau(m_j)$), or their images intersect ($\iota(m_i) \cap \iota(m_j) \neq \emptyset$). Weight of a vertex $v_{name(m_i)}$ is the weight of the attribute mapping: $w(v_{name(m_i)}) = weight(m_i)$.

Now finding the maximum weighted IS in G finds the maximum (optimal) ID set in the original document.

3.2 IS to MIP Formulation

Given a graph $G = (V, E)$ with a weight function $w : V \to \mathbb{R}$, we introduce a binary variable x_i for each vertex $v_i \in V$ and an inequality constraint $x_i + x_j \leq 1$ for each edge $e = (v_i, v_j) \in E$. Furthermore we introduce an objective function in form $\sum_{x_i} x_i w(v_i)$.

It is obvious that the objective function and all the constraints consitute a MIP instance, and that solving it finds the maximum weigthed IS in G.

3.3 Finding ID Sets With GLPK

By chaining these two translations we can create a MIP formulation for a given set of AMs from a document. Solving this MIP instance will give us the optimal ID set for this document.

GLPK is a multi-platform, multi-purpose solver well suited for this task. It uses the Simplex method to solve LP problems and *Branch & Bound* for MIP.

Branch & Bound This is an optimization method where the search space is systematically divided into smaller sub-spaces. This is the *branching* component, and a so-called *search tree* is built this way. Then, the sub-problems are recursively solved and whole branches of the search tree are discarded when it becomes obvious that the solution does not lie there. This is the *bounding* component. See [LD60] for detailed description.

An advantage of using *Branch & Bound* is that while traversing the search tree it finds intermittent, sub-optimal solutions. It is thus possible to limit the total search time and instead of the optimum take the best solution found so far.

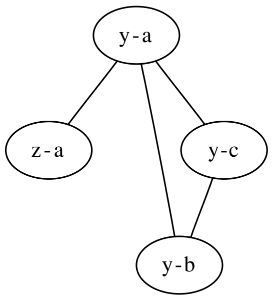
We will now demonstrate the full process of finding the optimal ID set of an example XML file using GLPK.

Example

Consider again our XML file fragment.

```
<x>
  <y a="1" b="2"/>
  <y a="3" c="4"/>
  <y/>
  <z a="1"/>
</x>
```

Figure 3.1: IS Representation Graph



Recall that attribute mappings in this example are $C = \{M_y^a, M_y^b, M_y^c, M_z^a\}$. Corresponding vertices in the IS formulation will be $V = \{v_{y-a}, v_{y-b}, v_{y-c}, v_{z-a}\}$. Edges in the IS formulation will be the following.

$$(v_{y-a}, v_{y-b})$$
$$(v_{y-a}, v_{y-c})$$
$$(v_{y-b}, v_{y-c})$$
$$(v_{y-a}, v_{z-a})$$

The first three edges are due to the type collision (y), the last one is due to $\iota(M_y^a) \cap \iota(M_z^a) = \{1\}$. The graph G constructed in this way is in Figure 3.1.

The next step is the MIP formulation. We do not need to translate from the IS formulation, as the translation from ID set formulation is straightforward, too. For each AM m there will be one binary variable $x_{name(m)}$. Objective function coefficients in vector **c** will be weights of respective mappings. For each pair of AMs m_1, m_2 that cannot share the same ID set there shall be a row in matrix A representing the inequality $x_{name(m_1)} + x_{name(m_2)} \leqslant 1$. **b** will be a vector of ones with corresponding length.

$$\mathbf{x} = \begin{bmatrix} x_{y-a} \\ x_{y-b} \\ x_{y-c} \\ x_{z-a} \end{bmatrix}, \mathbf{c} = \begin{bmatrix} weight(M_y^a) \\ weight(M_y^b) \\ weight(M_y^c) \\ weight(M_z^a) \end{bmatrix} = \begin{bmatrix} 0.2 \\ 0.2 \\ 0.6 \\ 0.4 \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}, A = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix}$$

The problem now is, recall, to solve the following.

$$\begin{aligned} \max_x z &= \mathbf{c}^{\mathrm{T}}\mathbf{x} \\ s.t.\ A\mathbf{x} &\leqslant \mathbf{b} \end{aligned}$$

In GLPK *MathProg* language (see [Mak]), this translates to the following formulation.

```
set AMs;
param Weight {i in AMs};
var x {i in AMs} binary;
maximize z: sum {i in AMs} x[i] * Weight[i];
s.t. c1: x['y-a'] + x['y-b'] <= 1;
s.t. c2: x['y-a'] + x['y-c'] <= 1;
s.t. c3: x['y-b'] + x['y-c'] <= 1;
s.t. c4: x['y-a'] + x['z-a'] <= 1;
data;
set AMs := y-a y-b y-c z-a;
param Weight :=
y-a 0.6
y-b 0.2
y-c 0.2
z-a 0.4;
end;
```

We can use this as an input for the GLPK solver, and we get the solution.

```
...
Problem:    glpk_input
Rows:       5
Columns:    4 (4 integer, 4 binary)
Non-zeros:  12
Status:     INTEGER OPTIMAL
Objective:  z = 0.6 (MAXimum)

...

   No. Column name       Activity     Lower bound   Upper bound
------ ------------    ------------- ------------- -------------
     1 x[y-a]       *              1             0             1
     2 x[y-b]       *              0             0             1
     3 x[y-c]       *              0             0             1
```

```
    4 x[z-a]      *            0            0            1

...
```

This output tells us that the solution is $x_{y-a} = 1$, $x_{y-b} = 0$, $x_{y-c} = 0$ and $x_{z-a} = 0$. This means that the optimal ID set with maximum weight contains only the M_y^a attribute mapping.

It is obvious that this approach works and for any possible input we can let GLPK find the optimal solution. However, sometimes it takes too long to find the optimum (see e.g. Section 4.3.2), hence, the aim of this work is to improve this process.

3.4 Heuristics

The definition of *heuristic* or *heuristic algorithm* varies from one source to another. We shall be using it roughly in the following sense.

Definition 3.1 (Heuristic)**.** A *heuristic* is an approach to problem solving based on prior experience, educated guess or common knowledge.

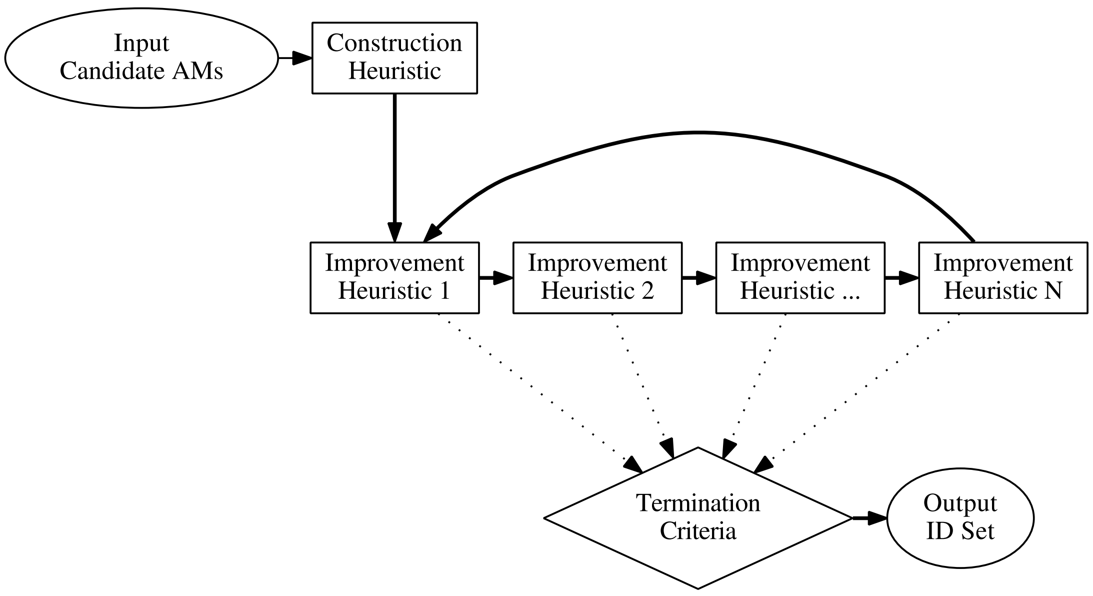
Definition 3.2 (Heuristic Algorithm)**.** A *heuristic algorithm* is one that, in a reasonably short time, generates a good, maybe even optimal solution to an optimization problem. However, it won't provide any formal guarantee about its quality.

This definion of heuristic algorithm coming from [DC10] is rather vague, however, it will be sufficient for us.

An example of a heuristic is the commonly used approach of *trial and error*: after a failed attempt, change some parameter and try again. We will see many more heuristics later in this chapter.

While a heuristic algorithm can be seen as a tool designed to solve one specific problem, the notion of a *metaheuristic* reminds of a recipy to solve a whole family of problems. We shall be using the metaheuristic presented in Figure 3.2

Figure 3.2: Metaheuristic schema



to find optimal ID sets in this work. But before we start describing its structure and components, we need to introduce some more notions.

Definition 3.3 (Solution Space, Solution Quality)**.** Solution space in general is the set of all permissible solutions (not violating any constraints). In the specific case of a MIP formulation it is the set of all $\mathbf{x}$ subject to $A\mathbf{x} \leqslant \mathbf{b}$. Every solution in the solution space has its *quality*, in case of MIP for solution $\mathbf{x}$ it is the value of the objective function in $\mathbf{x}$.

Definition 3.4 (Solution Neighborhood)**.** Neighborhood of a solution $\mathbf{x}$ in the solution space are all the other solutions close to $\mathbf{x}$ according to some metric.

The precise definition of the neighborhood is always adjusted according to specific needs. However, the neighborhood should be defined so that is *continuous* (with respect to quality) to be useful. The exact reasons for this requirement are sketched in Section 3.4.2.

Definition 3.5 (Solution Pool, Incumbent Solution)**.** *Solution pool* (sometimes called pool of feasible solutions or feasible pool) is a set of solutions of different qualities that were found in one of the stages of the metaheuristic. The solution(s) with the highest quality is (are) called the *incumbent* solution(s).

A quick reminder of what we are trying to solve using our metaheuristic: given a list of AMs with weights, find a non-conflicting subset maximizing the sum of weights in the subset. We will now describe its structure, please refer again to Figure 3.2.

First we take the list of candidate AMs and ask a *construction heuristic* (see Section 3.4.1) to provide us with a pool of solutions. Then, in a loop, we use this pool as input for *improvement heuristics* (see Section 3.4.2) and in turns ask them to improve it. All the time we check whether *termination criteria* are met, and if so, we terminate the metaheuristic. The incumbent solution from the last pool is then declared the Output ID set.

The notion of metaheuristic covers a wide range of topics in the field of heuristics, such as Tabu Search (see [GL97]), Ant Colony Optimization (see [DS04]) or Genetic Algorithms (see e.g. [Gol89]), to name a few.

We will now introduce the heurisitics we have implemented to use in our metaheuristic for finding optimal ID sets.

3.4.1 Constructions Heuristics

When we start to solve a problem using a metaheuristic approach, at first we have no solutions at all. The purpose of a construction heuristic (CH) is then to provide us with at least some solution. This may or may not be already the optimum, in the latter case it will be improved on later using improvement heuristics (IH). Some IHs can profit from a pool of several sub-optimal solutions, and some CHs can produce this pool from them.

`FIDAX`

The first construction heuristic is the algorithm described in [BM03] (we shall call it *`FIDAX`* from now on). It can trivially be used to give us one feasible solution - it is, after all, a deterministic heuristic algorithm.

The algorithm works in two steps. First, all candidate AMs are grouped according to their types, and for each type the AM with the highest weight is selected. Second, all these AMs (now called C') are traversed in order of their decreasing size. For each AM m, a set S of all conflicting AMs from C' is found and weights of both m and S are calculated. Then the weights are compared and either m or S is removed from C'.

The pseudocode of this CH (taken from the original article with trivial modifications without changing the logic) is in Listing 2.

Algorithm 2 *FIDAX* CH

```
Input: $C$ list of candidate AMs
Output: a feasible solution
  $C' \leftarrow C$ sorted by decreasing size
  Compute the weight $w(m)$ of each $m$ in $C$
  for each $t$ in $\Sigma^E$ do
    Let $m$ be a highest-weight mapping of type $t$ in $C'$
    Remove from $C'$ all mappings of type $t$ except $m$
  end for
  for each $m$ in $C'$ do
    $S \leftarrow$ all mappings in $C'$ whose images intersect $\iota(m)$
    if $w(m) > \sum_{p \in S} w(p)$ then
      remove all $p \in S$ from $C'$
    else
      remove $m$ from $C'$
    end if
  end for
  return $C'$
```

`Random`

One of the most natural heuristics when dealing with the ID set problem can be described as follows: select from candidate AMs at random, if possible (addition would not violate the ID set condition) add them to the solution. This is obviously a hungry heuristic.

The advantages of this trivial heuristic are simplicity, speed and ease with which it can create a pool of variable solutions, almost for free. As we will see later in the experiments (Section 4.3.3), it performs surprisingly well.

See the Listing 3 for its pseudocode.

Algorithm 3 *Random* CH

```
Input: $N$ required size of pool
Input: $C$ list of candidate AMs
Output: pool of $N$ feasible solutions
  $r \leftarrow$ empty pool
  for $i = 1 \to N$ do
    // create 1 solution
    $s \leftarrow$ empty solution
    while $s$ is a feasible ID set do
      $a \leftarrow$ pick at random from $C \backslash S$
      $s \leftarrow s \cup a$
    end while
    $r \leftarrow r \cup s$
  end for
  return $r$
```

Fuzzy

Fuzzy is an improvement over the *Random* CH: the next AM to be added is selected based on *weighted* instead of uniform random. The weight used here is the usual weight of an AM as defined in Section 1.5. Because of the randomness involved in the choice, we can again easily create a pool of solutions this way.

This is again a hungry heuristic, the Listing 4 contains its pseudocode.

Algorithm 4 *Fuzzy* CH

```
Input: $N$ required size of pool
Input: $C$ list of candidate AMs
Output: pool of $N$ feasible solutions
  $r \leftarrow$ empty pool
  for $i = 1 \rightarrow N$ do
    // create 1 solution
    $s \leftarrow$ empty solution
    $C' \leftarrow C$
    while $C'$ not empty do
      $a \leftarrow$ pick at weighted random from $C'$
      if $s \cup a$ is a feasible ID set then
        $s \leftarrow s \cup a$
        $C' \leftarrow C' \backslash a$
      end if
      for each $c \in C'$ do
        if $s \cup c$ is not a feasible ID set then
          // if $c$ cannot be possibly added anymore
          $C' \leftarrow C' \backslash c$
        end if
      end for
    end while
    $r \leftarrow r + s$
  end for
  return $r$
```

`Incremental`

This trivial heuristics sorts all candidate AMs by their decreasing weights (see Section 1.5) and then tries to iteratively add them to solution, if possible. This way it can create only one solution, and again, this is a hungry heuristic.

See Listing 5 for its pseudocode.

Algorithm 5 *`Incremental`* CH

```
Input: $C$ list of candidate AMs
Output: a feasible solution
  $C' \leftarrow$ sort $C$ by decreasing weight
  $s \leftarrow$ empty solution
  for each $c \in C'$ do
    if $s \cup c$ is a feasible ID set then
      $s \leftarrow s + c$
    end if
  end for
  return $s$
```

`Removal`

This is basically a reversal of the idea from the *`Incremental`* heuristic - start with a solution containing all the candidate AMs. This probably does not satisfy the ID set condition. Therefore, sort them by increasing size and start removing them from the solution, until it satisfies the ID set condition. Again, this is a hungry heuristic returning only one solution.

See Listing 6 for its pseudocode.

Truncated Branch & Bound - `Glpk`

This construction heuristic will be called *`Glpk`* from now on. It is basically a time-constrained run of GLPK. Recall that GLPK uses a the *Branch & Bound* algorithm that produces feasible solutions even before the optimum is found.

Algorithm 6 `Removal` CH

```
Input: $C$ list of candidate AMs
Output: a feasible solution
  $C' \leftarrow$ sort $C$ by increasing weight
  $s \leftarrow C'$
  for each $c \in s$ do
    if $s$ is a feasible ID set then
      return $s$
    end if
    $s \leftarrow s \backslash c$
  end for
```

Limiting the run time gives us the best solution found so far, which means this is a construction heuristic.

To be able to create a pool of solutions, the GLPK input in MathProg langauge is always randomly shuffled by changing the order in which variables and constraints appear. This is causes the solver to explore the search tree in various orders, producing different solution in each of the time-constrained runs.

3.4.2 Improvement Heuristics

Improvement heuristics in general start with a solution pool, attempt to improve one or more solutions in it and then return this improved pool in the end. We will need two notions to describe their behavior.

Intensification is the attempt to move the solution towards the nearby local optimum in the solution space.

Diversification is the attempt to move the solution away (escape) from from the local optimum, to be able to explore more of the solution space when the metaheuristic starts stagnating.

A metaheuristic needs to combine intensification and diversification tendencies to explore the solution space and at the same time arrive at a local optimum. Recall the requirement for the solution space to be continuous in terms of quality: this guarantees that as we approach a solution $\mathtt{x}$, the quality of solutions we encounter approaches the quality of $\mathtt{x}$.

`Identity`

This ultimately trivial improvement heuristics does nothing. It simply returns the feasible pool unchanged. For the sake of completeness, see its Listing 7.

Algorithm 7 *`Identity`* IH

```
Input: $FP$ pool of feasible solutions
Output: the same pool of feasible solutions
  return $FP$
```

`Remove Worst`

This trivial IH tries to improve the solution pool by removing the worst solution (i.e. the one with the lowest quality). This is interesting in cooperation with other improvement heuristics that increase the solution pool size, to keep it from growing by pruning inferior solutions.

See Listing 8 for details.

Algorithm 8 *`Remove Worst`* IH

```
Input: $FP$ pool of feasible solutions
Output: pool of feasible solutions
  $s_{min} \leftarrow$ solution with the lowest weight $\in FP$
  return $FP \backslash s_{min}$
```

`Random Remove`

This is again a rather trivial diversification improvement heuristic. By removing a random subset of specified size from each solution in the pool, it provides the variability needed to escape from local optima in the solution space.

The number of AMs to remove from each solution is specified as fraction from $(0, 1)$ of the solution size (number of AMs in solution). For example, *Random Remove* with $fraction = 0.1$ would remove 1 random AM from a solution containing 10 AMs and 2 from a solution containing 17 AMs (due to rounding).

This heuristic returns a pool of solutions of the same size as it got on its input.

See Listing 9 for pseudocode.

Algorithm 9 *Random Remove* IH

```
Input: $FP$ pool of feasible solutions
Input: $k \in (0, 1)$ fraction of AMs to remove from each $s \in FP$
Output: pool of feasible solutions
  for each $s \in FP$ do
    $K \leftarrow k * |s|$
    remove $K$ random AMs from $s$
  end for
  return $FP$
```

`Hungry`

This simple improvement heuristic assumes that the solutions in the pool are not "complete", i.e. there are AMs that could be added to them without violating the ID set condition.

Hungry tries to improve each solution in the feasible pool in the following way. It orders all candidate AMs not present in the solution by decreasing weight. Afterwards, it iteratively tries to extend the solution with these AMs, taking care not to violate the ID set condition. The resulting solution (whether any AMs were added or not) is then returned to the pool. This is then intensification, and Listing 10 captures the process.

The following three IHs: *Mutation*, *Crossover* and *Local Branching* are

Algorithm 10 *Hungry* IH

```
Input: $FP$ pool of feasible solutions
Input: $C$ list of candidate AMs
Output: pool of feasible solutions
  for each $s \in FP$ do
    // improve a single solution
    $C' \leftarrow C \backslash s$
    $C' \leftarrow C'$ sorted by decreasing weight
    for each $c \in C'$ do
      if $s \cup c$ is a feasible ID set then
        $s \leftarrow s \cup c$
      end if
    end for
  end for
  return $FP$
```

inspired by [DC10].

Mutation

Mutation is based on the following idea. We assume that an incumbent solution may already contain some AMs belonging to the optimal solution. We will take a random guess and fix some of these AMs, i.e. we add new constraints to the MIP formulation fixing values of the respective variables to 1.

This new formulation contains less free variables and should be easier to solve, probably even to optimum. We run GLPK again using this constrained formulation, enforcing again a time limit. Solution found this way is a feasible solution of the original problem, however the optimum is not necessarily the same as in unconstrained formulation. It is an intensification approach: we limit the search to the neighborhood of an already found solution.

Mutation changes the MIP formulation in following way. For every AM

AM_F fixed to appear in the solution a following constraint is added to GLPK input:

$$s.t.f_{index} : x['name(AM_F)'] = 1;$$

$index$ is a unique integer to number all the constraints.

Additionaly, every other mapping AM_i colliding with AM_F ($\iff \iota(AM_F) \cap \iota(AM_i) \neq \emptyset$) will cause the following constraint to be added:

$$s.t.f_{index} : x['name(AM_i)'] = 0;$$

And the original constraint in form:

$$s.t.c_{index} : x['name(AM_F)'] + x['name(AM_i)'] <= 1;$$

will not be included.

Listing 11 captures the process of randomly selecting a specified fraction of AMs of the incumbent solution to fix, then running GLPK again. *`Mutation`* requires pool of at least one solution as input, and adds the improved solution to the result pool.

Algorithm 11 *`Mutation`* IH

```
Input: $FP$ pool of feasible solutions
Input: $k$ fraction of AMs to fix
Output: pool of feasible solutions
  $incumbent \leftarrow$ incumbent solution in $FP$
  $K \leftarrow k * |incumbent|$
  fix $K$ random AMs from $incumbent$ in GLPK problem formulation
  $improved \leftarrow$ run GLPK
  return $FP \cup improved$
```

`Crossover`

This improvement heuristic expands on the idea of *`Mutation`*. But, instead of randomly selecting AMs in the incumbent solution, it looks for commonalities among the solutions in the pool. This is based on the hope that if more solutions agree on the same AMs, those are probably included in the optimal solution too.

`Crossover` takes a parameter - fraction of solutions in the pool among which to look for commonalities. AMs found in every one of them are fixed in the modified MIP formulation the same way as in `Mutation`. This again amounts to an intensification tendency.

Listing 12 captures the process. `Crossover` requires at least one solution in the pool, but to work properly, more are needed. Solutions are picked at random from the pool, common AMs found and fixed. GLPK is run again (with a time constraint) and the improved solution is added to the result pool.

Algorithm 12 `Crossover` IH

```
Input: $FP$ pool of feasible solutions
Input: $k$ fraction of solutions among which to look for commonalities
Output: pool of feasible solutions
  $K \leftarrow k * |FP|$
  $FP' \leftarrow K$ random solutions $\in FP$
  $am \leftarrow$ AMs found in all solutions $\in FP'$
  fix $am$ in GLPK problem formulation
  $improved \leftarrow$ run GLPK
  return $FP \cup improved$
```

Local Branching

`Local Branching` is another intensification heuristic. This time the neighborhood being searched is defined by edit distance.

The incumbent solution is represented by a vector $\mathrm{x}_{INCUMBENT}$ of ones and zeroes. Based on it, a new constraint will be added to the MIP formulation. For every other solution x_i the edit distance, i.e. number of positions in which $\mathrm{x}_{INCUMBENT}$ and x_i differ, will have to be lower than some threshold K. This will be represented in MathProg as follows.

$$s.t. LB : sum\{i\, in\, INCUMBENT\}(1-x[i]) + sum\{i\, in\, REMAINING\}x[i] \leq K;$$

Where $INCUMBENT$ is the set of names of AMs in the incumbent solution, $REMAINING$ is the set of all AMs not included in the incumbent solution and K is the maximum edit distance allowed. K is determined as a fraction of the count of all AMs, provided as parameter k.

See Listing 13 for pseudocode. The heuristic requires a pool containing at least one solution, solves the modified MIP formulation using GLPK limited to some time again and adds the improved solution to the result pool.

Algorithm 13 `Local Branching` IH

```
Input: $FP$ pool of feasible solutions
Input: $k$ fraction of the total AM count to determine max edit distance
Output: pool of feasible solutions
  $K \leftarrow k * |\text{total AM count}|$
  $incumbent \leftarrow$ incumbent solution in $FP$
  add max edit distance requirement to GLPK problem formulation
  $improved \leftarrow$ run GLPK
  return $FP \cup improved$
```

Genetic algorithms It is worth noting that by combining `Mutation`, `Crossover` and `RemoveWorst` we get a very simple genetic algorithm.

3.5 IDREF

Once an ID set is found, regardless of how exactly, it is easy to find the IDREF set, i.e. the attribute mappings that can be declared as IDREF. This algorithm is adopted from [BM03].

First of all, from the set of all the attribute mappings in the model remove all the AMs contained in the ID set. This is because the specification of DTD/XSD does not allow an attribute to be `ID` and `IDREF` (`IDREFS`) at the same time. Let us denominate these mappings as *IDREF candidates* (obviously different from *candidate AMs*).

Second, find the image of the ID set as the union of images of all the AMs in this ID set.

$$\iota(ID) = \bigcup_{m \in ID} \iota(m)$$

Now the IDREF set contains all the AMs whose images are a subset of the ID set image.

$$\iota(c) \subset \iota(ID) \Rightarrow c \in IDREF$$

This can be easily determined in a loop over the list of candidates. The process is captured in Listing 14.

Algorithm 14 IDREF Search

```
Input: $AMs$ list of all AMs
Input: $ID$ ID set as a list of AMs
Output: $IDREF$ set as a list of AMs
  $IDREF \leftarrow \emptyset$
  $candidates \leftarrow AMs \backslash ID$
  $\iota(ID) \leftarrow \bigcup_{m \in ID} \iota(m)$
  for each $c \in candidates$ do
    if $\iota(c) \subset \iota(ID)$ then
      $IDREF \leftarrow IDREF \cup c$
    end if
  end for
  return $IDREF$
```

4. Experiments

At this point of the thesis the reader should be already familiar with the notions we have introduced: the problem of finding the optimal ID set (with respect to a given *weight*), that it is directly related to the NP-complete problem of finding the maximal weighted independent set, that this can be solved using the MIP approach, and that there are several possibilities how to optimize the work of the solver by employing various heuristics.

We have implemented these ideas and incorporated them in the jInfer framework (see Appendix A). But before we describe the experiments themselves, we should try to formulate our aim.

First of all, we describe how the whole system and its components behave. We want to see the changes introduced by modifying several parameters, while keeping the others fixed. They probably will not be orthogonal, we might at least isolate some of the parameters that are less important to the overall behavior.

Second, we evaluate the system performance in terms of the speed of finding good heuristic results. We find tweaks to make the whole process as fast as reasonably possible.

And in the end, we formulate general recommendations regarding the problem of finding ID sets.

4.1 Experimental Data

To conduct out experiments, we are using XML documents of three categories:

- Realistic
- Realistic with artificial (converted) attributes
- Artificial

In case of realistic data we want to see the performance in cases taken from the real world. The problem with realistic data is that sometimes, interesting values (that might or might not contain IDs) are stored as simple text nodes instead of attributes. We will try to convert some of these values to attributes (e.g. using a smart XSL transformation), let our heuristics find the ID sets, and then translate them back to XML keys (see Section 4.1.2 for details). And finally, we create completely artificial data to create inputs that will put our heuristics in stress. This is because the realistic data often prove to be too simple to solve - the list of candidate AMs is usually too short to be hard to be solved to optimality.

Definition 4.1 (Data set)**.** One or more XML files sharing the same schema (even if only an implicit schema) shall be referred to as a *data set*. In the scope of this work this will always mean a *single* XML file. However, this definition of a data *set* covers also the extension to more XML files as described in [BM03].

To understand our test data sets we discuss their origin and *graph representation*. As mentioned earlier in Chapter 3, the problem of finding the optimal ID set is in fact the problem of finding the maximum weighted independent set in a graph. Therefore it is interesting to actually see the graphs of these data sets and understand some related metrics.

The former will be achieved with the help of the GraphViz tool ([gra]), where we will draw the graphs so that all the vertices represent the *candidate AMs*, and the edges represent pairs of AMs that have nonempty intersection of their images (and thus cannot be in the same ID set together). Thus solving the maximal weighted IS on these graphs will be equivalent to solving our problem of optimal ID set.

The latter will come in form of tables containing information regarding the data sets, such their size, known optimum for $\alpha = \beta = 1$ (found by running the `Glpk` heuristic without a time limit) and the numbers of vertices and edges in aforementioned graphs.

Table 4.1: List of realistic test data files

Name	Size [kb]	$\lvert V\rvert$	$\lvert E\rvert$	Optimum
`OVA1`	4.5	29	43	0.45588235294117635
`OVA2`	11.9	23	36	0.1634615384615385
`OVA3`	237.6	31	47	0.25537156151635415
`XMA-c`	1 807.7	1	0	0.7546666666666667
`XMA-p`	13 748.3	1	0	0.2019306150568969
`XMD`	1 743.0	17	15	0.09786094165493507

4.1.1 Realistic Data

From 3 different sources we collected 6 different data sets, called `OVA1 - OVA3`, `XMA-c`, `XMA-p` and `XMD`. Their summary is in Table 4.1, their graph representations can be seen in Figure 4.1.

To interpret the data: `OVA*` sets have quite interesting and challenging graphs, but they are relatively small. We can consider them to be the "typical" representants.

On the other hand, the `XMA-*` sets are relatively huge, but trivial: their only candidate AM will just get picked and the heuristic will end. Therefore we will see the performance of the other components of the whole system, such as loading the data sets into memory representations.

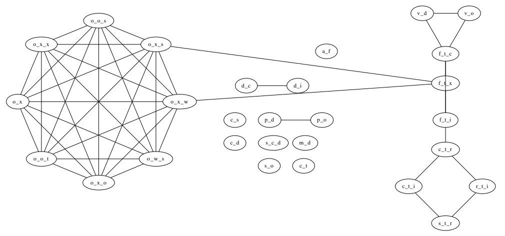
Finally, the `XMD` set is relatively big and, at the same time, has non-trivial graph representation. In this case we should see a performance more balanced between processing and finding the ID set.

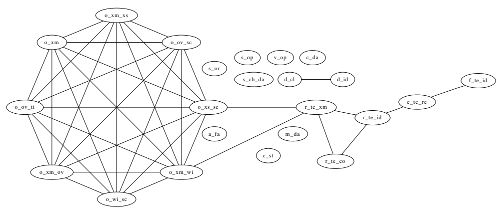
4.1.2 Realistic Data With Artificial Attributes

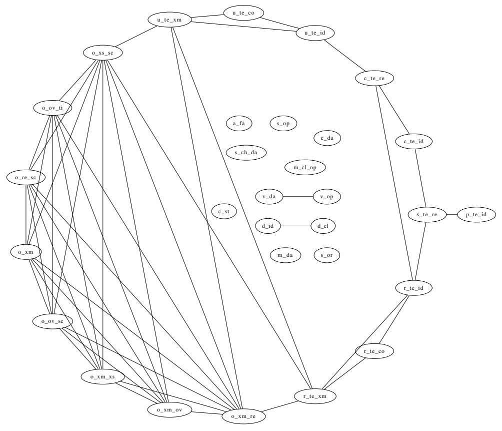
We used 2 data sets to convert, `MSH` and `NTH`. None of these sets had any attributes before the conversion. Their summary is in Table 4.2, their graphs are in Figure 4.2.

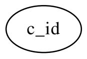
To address the conversion: in case of `MSH` we found 2 elements with values resembling a key of the records contained in the file, and converted them to be

Figure 4.1: Realistic data

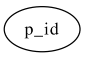

(a) *OVA1*


(b) *OVA2*

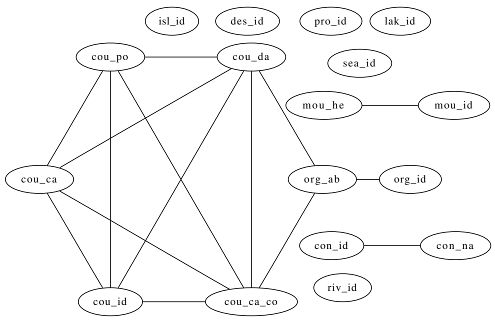

(c) *OVA3*



(d) *XMA-c*



(e) *XMA-p*

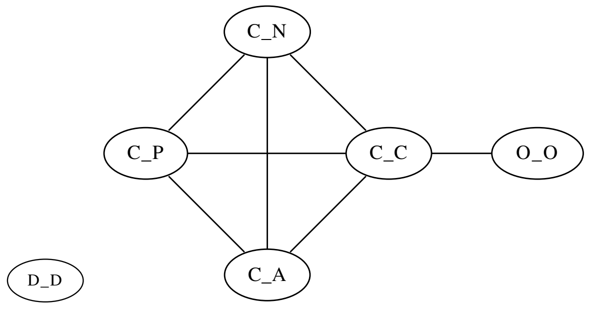


(f) *XMD*

Table 4.2: List of realistic test data files with converted attributes

Name	Size [kb]	$\|V\|$	$\|E\|$	Optimum
MSH	3 100.5	1	0	0.5416472778036296
NTH	2 523.5	5	7	0.057918595422124436

Figure 4.2: Realistic data with converted attributes



(a) *MSH* (b) *NTH*

attributes of these records using a simple XSL transformation. In the case of *NTH* we converted all the values in sub-elements of the record elements to be the attributes of the records.

This approach is useful, because as mentioned in Chapter 2, `ID` attributes are a special case of XML keys. We can use this approach to find XML keys: convert some "suspicious" data into attributes, find the optimal ID set and then create XML key based on this ID set.

In case of *MSH* we created 2 attributes, of which only one constituted a candidate AM. This is then the case similar to *XMA-** sets: quite large data, yet only one trivial `ID` attribute to be found.

In case of *NTH* we introduced 8 attributes. Out of them 5 proved to be candidate AMs, with 7 edges constraining them. This means we have a relatively large set with considerably simple work to be done by the heuristics.

4.1.3 Artificial Data

As soon as we started experimenting with the data coming from the real world, it was obvious that they are not complex enough. After we built the model, we got the most complex graphs of 31 vertices and 47 edges (see Table 4.1). We approach this problem from the other side: in the end, we will be solving the equivalent of IS problem on a graph created from XML data. We will create the XML data to contain a more complex graph with a specific number of vertices and edges.

This data will represent very complex real world schemas with a lot of various attributes with similar values.

Our artificial test data will look similar to the following excerpt from an XML file:

```
<graph>
  <vertex0 attr="-2968876296119015800"/>
  <vertex1 attr="1729745997570096518"/>
  <vertex2 attr="-9020549659620928934"/>
  ...
  <vertex99 attr="-7545982394508643394"/>

  <vertex82 attr="0"/><vertex21 attr="0"/>
  <vertex64 attr="1"/><vertex21 attr="1"/>
  <vertex44 attr="2"/><vertex2 attr="2"/>
  ...
  <vertex96 attr="99"/><vertex40 attr="99"/>
</graph>
```

Our aim is to create a graph with approximately v vertices and e edges. First, we introduce v elements with names `vertex0` - `vertex{V-1}`. To constitute an AM, they need an attribute `attr`, but with large enough random values, so that they do not conflict with others. Second, for each of the e edges we choose two `vertex*` elements at random, and give them the same value of their `attr`. This will ensure they cannot share the same ID set, thus effectively creating the edge in the graph representation.

This way we can generate data more complex than anything found in the real world. It will however serve us to measure the performance of our algorithms in extreme cases.

The respective pseudocode for this is provided in Algorithm 15.

Algorithm 15 Random XML Data Creation

```
Input: $v$ requested number of vertices
Input: $e$ requested number of edges
Output: XML file content
  print <graph>
  for $i = 1 \to |V|$ do
    $R \leftarrow RANDOM$
    print <vertex$i$ attr="$R$">
  end for
  for $i = 1 \to |E|$ do
    $v1 \leftarrow RANDOM(|V|)$
    $v2 \leftarrow RANDOM(|V|)$
    print <vertex$v1$ attr="$i$"> <vertex$v2$ attr="$i$">
  end for
  print </graph>
  return
```

With this process it is possible to create as much data as needed, with any combination of v and e requested.

There is one characteristic that can describe random graphs like this, and that is the *density*. This can be defined in various ways, we will use two different interpretations. The first is $\frac{|E|}{|V|}$, that is, how many edges are there for one vertex (multiplied by 2 we would get the average degree of the vertices).

The second, perhaps more interesting is $\frac{|E|}{E_{max}}$, where $E_{max} = \frac{|V|.(|V|-1)}{2}$. This is the density as the fraction of edges that are to all edges that could be in a complete graph with $|V|$ vertices.

We have created 3 sets to be used in experiments along with the realistic and converted sets, called *100-100*, *100-200* and *100-1000*. Note that the name is always in the form $v - e$.

All of the experimental data sets mentioned so far, realistic, converted and artificial alike will be referred to as *official test data (sets)*.

Table 4.3: List of artificial test data files

Name	Size [kb]	$\|V\|$	$\|E\|$	$\frac{\|E\|}{\|V\|}$	$\frac{\|E\|}{E_{max}}$	Optimum
100-100	8.4	99	95	0.95	0.02	0.836666666666667
100-200	13.0	96	174	1.81	0.04	0.726000000000000
100-1000	49.5	93	754	8.11	0.16	0.380952380952381

Also, we will need data of comparably similar characteristics but varying size to study the effects of size on the run times of experiments. For this reason we created 11 more sets, from *0-0* as the trivial one to *100-500* as the largest one. These will be referred to as *sized test data (sets)*. We wanted to keep the same density among these sets, so we picked the $\frac{|E|}{E_{max}}$ density interpretation for this.

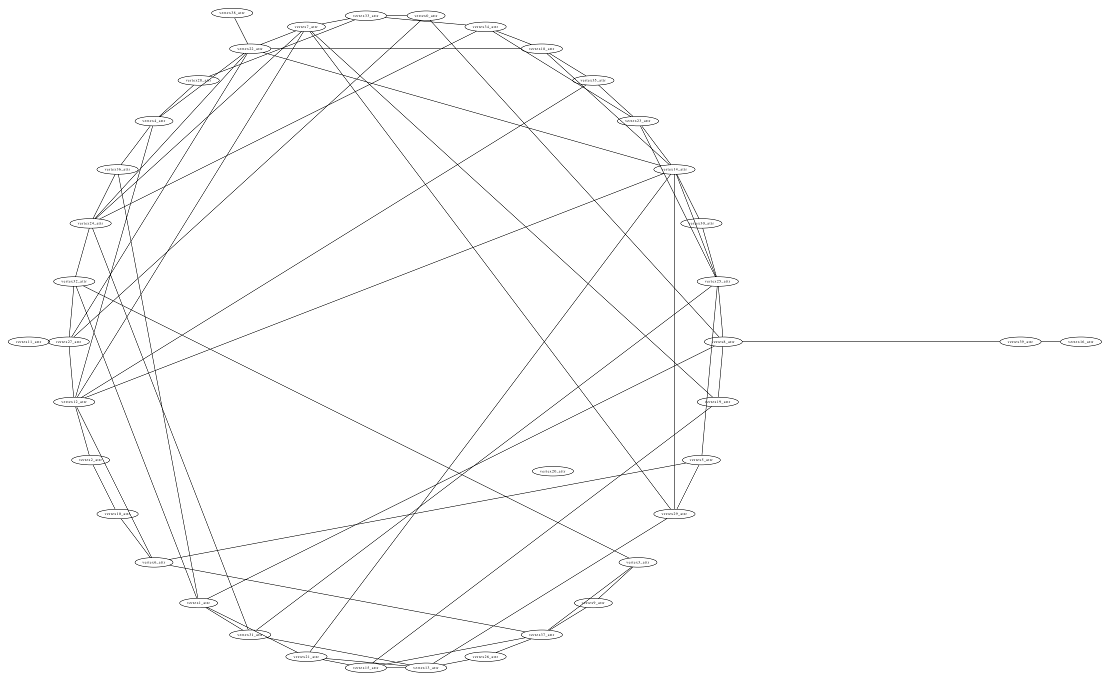
A summary is provided in Table 4.3 and Table 4.4; these tables contain 2 new columns: values of density in both interpretations we introduced. Some of the graph representations can be seen in Figure 4.3.

While studying the tables it becomes obvious that the actual numbers $|V|$ and $|E|$ do not match to the v and e in the names of the sets. This is because of the way the random generation algorithm works: it might pick the same edge twice, which will automatically render it unsuitable for the ID set. Because of the so-called Birthday paradox (see e.g. [McK66]), this will happen more with higher e.

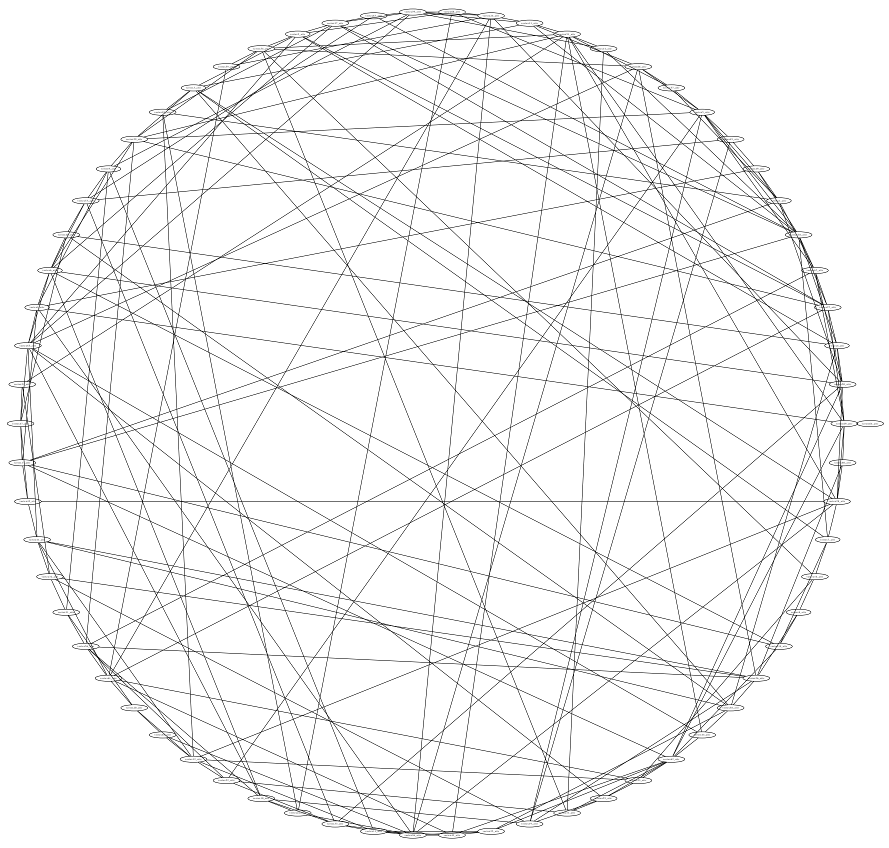
To interpret Tables 4.3 and 4.4: we get 3 sets of different sizes and densities in the first one. The $|V|$ and $|E|$ numbers are orders of magnitude higher than in any realistic (or converted) data set we are using.

In the second table we aimed for the $\frac{|E|}{E_{max}}$ density of $0.1 = 10\%$, and we can see that this was indeed achieved. There is an interesting observation to be made here: the optimum is steadily decreasing with the increasing overall graph size. This intuitively suggests that the maximum quality theoretically achievable is related to the $\frac{|E|}{|V|}$ density, not to the one we fixed. Exploration of this phenomenon is beyond the scope of this thesis.

Figure 4.3: Artificial data



(a) *48-80*



(b) *70-245*

Table 4.4: List of "sized" artificial test data files

Name	Size [kb]	$\lvert V\rvert$	$\lvert E\rvert$	$\frac{\lvert E\rvert}{\lvert V\rvert}$	$\frac{\lvert E\rvert}{E_{max}}$	Optimum
0-0	0.2	0	0	-	-	0.0
10-5	0.6	10	5	0.50	0.11	0.8500000000000002
20-20	1.7	18	13	0.72	0.08	0.7166666666666669
30-45	3.1	29	43	1.48	0.11	0.7083333333333334
40-80	5.1	39	72	1.85	0.10	0.6950000000000002
50-125	7.5	48	111	2.31	0.10	0.6566666666666666
60-180	10.4	58	157	2.71	0.09	0.6214285714285716
70-245	13.8	67	205	3.06	0.09	0.5982142857142856
80-320	17.6	76	261	3.43	0.09	0.5791666666666667
90-405	21.9	86	352	4.09	0.10	0.528888888888889
100-500	26.7	91	388	4.26	0.09	0.4981818181818182

Note that all the data sets we used in experiments can be found on the DVD enclosed with this work.

4.2 Experimental Setup

As was mentioned before, we will use an extension to the jInfer framework called *IDSetSearch*. Please see Appendices A and B for more detailed information on these two pieces of software.

We now have to introduce a few notions before moving forward to the description of our experiments. In the following text, words *experiment* and *experimental* in various phrases (*experiment X*, *experimental Y*) will be used interchangeably.

Experiment parameters are the following ones.

- All the parameters in all the heuristics.
- The specific way in which the heuristics are chained.
- Parameters α and β in the weight (quality) measurement.
- Initial pool size.
- The termination criteria.
- The input XML file.
- Known optimum for this file and α, β.

An *experiment instance*, or *experiment configuration*, is one specific setting of all experiment parameters.

And finally, one or more experiment configurations, regardless whether their parameters differ, constitute an *experiment set*.

4.2.1 Grammar and Model Creation

This section will briefly describe the process by which an input data set is processed to obtain the AM model as described in Section 1.3.

An input data set is a single XML file on the filesystem; however, there is a straightforward extension to multiple files conforming to the same schema. The first step in this process is to use jInfer's module *BasicIGG* module (see [KMS+11b] for details) to obtain a list of rules - an *initial grammar* (IG). Please see [KMS+11a] for detailed specification of IG format.

The second step is to convert the grammar into the AM model. This is done by a linear scan and retrieving a so-called *flat* representation. This consists of a list of tuples in the following format.

(element name, attribute name, attribute value)

There is a tuple for every attribute node with a value found in the initial grammar. Note that the information about the context in which the element was originally found is lost - but this is not a problem with regard to the definition of XML `ID` attributes. Furthermore, tuples in flat representation do not need to be unique.

The model now has to be able to return the list of all attribute mappings and their respective images. This is achieved by simply grouping the flat representation by the pair *(element name, attribute name)* and aggregating all attribute values for each such pair. Another responsibility of the model is to return the list of *types* - that is simply the list of unique *element name*s.

Example

Recall the following XML file fragment from Chapter 1.

```
<x>
  <y a="1" b="2"/>
  <y a="3" c="4"/>
  <y/>
  <z a="1"/>
</x>
```

Its IG representation is the following set of IG rules.

$$\begin{array}{lcl} x & \rightarrow & y, y, y, z \\ y & \rightarrow & @a, @b \\ y & \rightarrow & @a, @c \\ y & \rightarrow & empty\_concatenation \\ z & \rightarrow & @a \end{array}$$

The flat representation will consist of the following set of tuples.

$$(y, a, 1)$$
$$(y, b, 2)$$
$$(y, a, 3)$$
$$(y, c, 4)$$
$$(z, a, 1)$$

Attribute mappings in this model will be (y, a), (y, b), (y, c) and (z, a). Their images will be $(1, 3)$, (2), (4) and (1), respectively. The list of types in this model will be (y, z).

4.2.2 Hardware and Software

We will use the following configuration when conducting our experiments.

```
Intel Core 2 Duo processor @ 2.33 GHz
4 GB DDR2 RAM
Windows 7 SP1 64bit
Java SE Runtime Environment (build 1.6.0_26-b03)
Java HotSpot 32-Bit Client VM (build 20.1-b02)
GLPK version 4.45 (Cygwin)
GLPK version 4.34 (native)
```

4.2.3 Methodology

We will attempt to protect our experiment from the influence of the environment as much as reasonably possible. First of all, NetBeans running the experiments is the only relevant program running in the system while the experiments are performed. Unfortunately, NetBeans itself is quite a large environment, and we would most certainly get more reliable results if we could run our experiments outside of it. This improvement is left for the future work.

Also, every experimental configuration is run 50 times so that the effects of any events adversely affecting our results (e.g. OS deciding to run some house cleaning) will be averaged out. Whenever possible, we will use boxplots instead of a simple average (or average and variance) to present results of these multiple runs.

4.2.4 Measuring the Time

Whenever it is necessary to measure the duration of an operation, we will use the `System.nanoTime()` built-in function. The result cannot be interpreted in an absolute manner, but by subtracting the time at the start from the time at the end, we can get a reasonably reliable measurement.

4.2.5 Obtaining the Results

Every run of an experiment produces a trace such as the one presented and commented on in Appendix C. We can get all the information relevant to that experiment run from this trace alone. An experimental set will produce a number of these traces and store them in plain text files in a folder. Parsing these files to aggregate and collate them might be a tedious task even using tools like `sed` and `grep`, so some of the experiment sets directly output tabular data in format recognized by GnuPlot [gnu], which we use to plot charts found in this work.

4.2.6 Reading Boxplots

To present a set of measurements obtained by iteratively running an experiment we shall prominently use the *boxplot* chart. Because we use boxplots produced by GnuPlot, let us quote its manual [Kel] for the exact definition.

> Quartile boundaries are determined such that 1/4 of the points have a value equal or less than the first quartile boundary, 1/2 of the points have a value equal or less than the second quartile (median)

> value, etc. A box is drawn around the region between the first and third quartiles, with a horizontal line at the median value. Whiskers extend from the box to user-specified limits. Points that lie outside these limits are drawn individually.

The "user-specified limits" of whiskers are set to default value, let us quote from the manual again.

> By default the whiskers extend from the ends of the box to the most distant point whose y value lies within 1.5 times the interquartile range.

4.3 Experimental Results

We will now present the experiments we performed along with their results and conclusions. Each experiment will be introduced with a table summarizing the most important parameters of the experiment, such as data used, number of iterations and CH pool size, α and β parameters and the actual heuristics used.

4.3.1 Grammar and Model Generation

The first experiment set will try to establish how long it takes to extract the IG from the input XML file and to create the AM model from this IG. For now, we will not be running or measuring any heuristics.

Input data	all official and sized test data sets
Iterations	50
Pool size	not applicable
α, β	not applicable
CH	not applicable
IHs	not applicable

The experimental set will contain $50 * (11 + 11) = 1100$ configurations: 50 iterations for 11 test data sets plus 11 sized test data sets. There will be no CHs or IHs. We will be gathering the timing data for IG extraction and model generation in GnuPlot format.

The results are captured in Table 4.5. We are presenting the average grammar extraction (GE) times and their standard deviation, the same for model creation (MC) and total (sum of these two, Tot) times. For many data sets the average time is less than 10 ms: this is not enough to be precise and we do not calculate the standard deviation in these cases.

We can see from the results that for most data sets their model can easily be created under around one second, only in case of the biggest set `XMA-p` (13

Table 4.5: Grammar Extraction and Model Creation Times

Data set	GE avg [ms]	GE stdev	MC avg [ms]	MC stdev	Tot avg [ms]	Tot stdev
OVA1	< 10	-	< 10	-	< 10	-
OVA2	< 10	-	< 10	-	< 10	-
OVA3	42.94	19.8509	60.92	27.0848	103.86	31.6911
XMA-c	140.32	33.2618	90.24	45.8803	230.56	56.2633
XMA-p	7518.82	922.8882	10135.46	502.8997	17654.28	1353.8794
XMD	979.18	307.1760	563.04	341.4697	1542.22	134.6883
MSH	570.24	167.1119	225.48	90.6775	795.72	161.8340
NTH	328.36	118.3766	1074.9	155.5604	1403.26	137.8695
100-100	< 10	-	< 10	-	< 10	-
100-200	< 10	-	< 10	-	< 10	-
100-1000	18.34	10.2372	18.84	1.0373	37.18	9.9338
0-0	< 10	-	< 10	-	< 10	-
10-5	< 10	-	< 10	-	< 10	-
20-20	< 10	-	< 10	-	< 10	-
30-45	< 10	-	< 10	-	< 10	-
40-80	< 10	-	< 10	-	< 10	-
50-125	< 10	-	< 10	-	< 10	-
60-180	< 10	-	< 10	-	< 10	-
70-245	< 10	-	< 10	-	< 10	-
80-320	< 10	-	< 10	-	12.48	8.3574
90-405	< 10	-	< 10	-	15.88	10.3778
100-500	< 10	-	< 10	-	18.74	8.8889

MB) this takes some 17 seconds. We can conclude that grammar and model creation times are not a bottleneck for now. Heuristics run times will be order of magnitude higher.

GLPK Interface Timing

A related problem is how long it takes to create an input for GLPK and then parse its results. We will use the same test data sets as in the previous case, but now we will gather times needed to communicate with GLPK.

The results are captured in Table 4.6. For each data set there are the times of (GLPK) input creation (IC) - average and standard deviation, then the same for output parsing (OP) and total (Tot).

Interestingly enough, in most cases the times to create an input for GLPK and then to parse its output are very similar. Also, for sized test data sets it is interesting to note that even though the $|V|$ and $|E|$ numbers are increasing, the times remain almost the same. This is probably due to the fact that IC and OP times include the I/O when writing to a file for GLPK or reading the file it produced, and these times are probably the most relevant.

4.3.2 GLPK: Native vs. Cygwin

In this experiment we will try to remove one of the variables out of the equation: that is the effect of different versions of GLPK on the overall results. The rationale is this: on Windows systems, the two most accessible ways to install GLPK are via a binary distribution or via Cygwin as one of its packages.

If we find out which of these Cygwin version is better, we will be using it exclusively knowing this should not affect any other aspect of our experiments. We might also find that there is no relevant difference, which would be and interesting finding, too.

Apart from comparing different versions, we shall see how the pure GLPK approach behaves. The first part of this experiment will be limiting the run

Table 4.6: GLPK Interface Times

Data set	IC avg [ms]	IC stdev	OP avg [ms]	OP stdev	Tot avg [ms]	Tot stdev
`OVA1`	36.46	66.8517	49.8	114.0687	86.26	150.1044
`OVA2`	39.52	75.8210	48.8	102.4484	88.32	154.9596
`OVA3`	34.1	74.1838	38.62	89.3772	72.72	134.7295
`XMA-c`	40.88	88.6632	33.84	65.8636	74.72	127.7338
`XMA-p`	36.54	70.7436	49.24	101.2412	85.78	145.2092
`XMD`	37.98	69.2719	32.88	70.2173	70.86	114.6692
`MSH`	40.42	91.9885	36.52	72.1018	76.94	138.6198
`NTH`	36.02	66.3403	38.06	88.8244	74.08	128.9974
`100-100`	46.5	103.3929	46.92	89.7049	93.42	158.7267
`100-200`	42.34	96.1204	38.22	90.0284	80.56	152.6534
`100-1000`	32.92	64.4534	42.1	89.4546	75.02	127.8541
`0-0`	46.8	123.5183	46.92	102.2601	93.72	181.5228
`10-5`	40.06	75.7370	40.1	72.4851	80.16	126.7135
`20-20`	33.72	70.7263	34.1	66.2781	67.82	116.3783
`30-45`	38.26	71.7549	45.94	110.1284	84.2	155.7594
`40-80`	37.06	67.0024	49.26	106.3185	86.32	144.9918
`50-125`	50.44	101.9162	84.76	364.7350	135.2	378.7835
`60-180`	38.38	89.3379	42.54	94.3742	80.92	149.6049
`70-245`	41.5	93.2951	40.3	93.4858	81.8	149.6797
`80-320`	51.92	121.9812	47.98	96.0904	99.9	171.4617
`90-405`	40.5	91.5373	36.46	88.5099	76.96	144.2890
`100-500`	37.82	85.7571	43.4	90.3257	81.22	141.9103

time, thus making it an instance of `Truncated branch & bound`. In this case we will see the dependency between the run time and the quality achieved in it. In the second part we will let GLPK run until the optimum is found. We shall see the dependency between input size and run time needed to achieve the optimum.

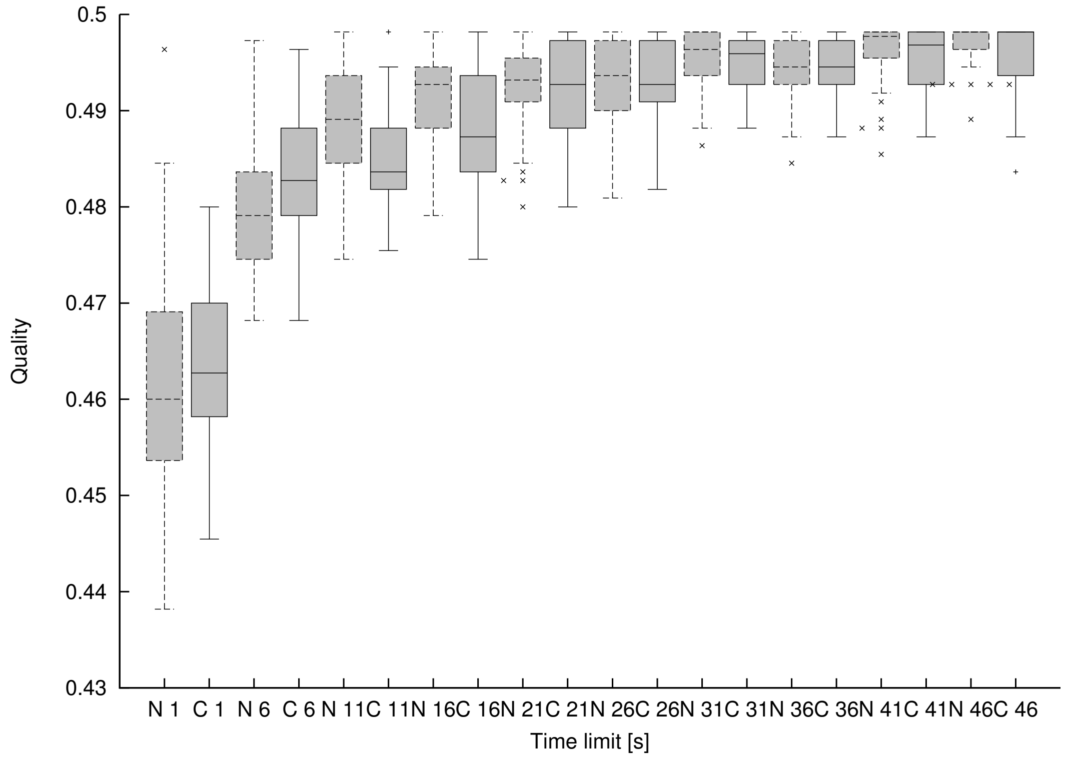
Input data	`100-500`
Iterations	50
Pool size	1
α, β	$1, 1$
CH	`Glpk`
IHs	$\emptyset$

Our experimental set will contain 500 experimental configurations for each of these two GLPK versions. Every configuration will use `Glpk` CH set to a time limit from 1 to 46 seconds with increments of 5, meaning 10 settings * 50 iterations = 500 configurations in total (see Algorithm 16). There will be no improvement heuristic. The only data we gather in the GnuPlot file are the final qualities (weights). The data set used is `100-500` as the biggest one in sized test data.

Algorithm 16 GLPK: Native vs. Cygwin Set Generation 1

```
Output: experimental set $ES$
  $ES \leftarrow \emptyset$
  for $i = 1 \rightarrow 50$ do
    for $time = 1 \rightarrow 46$ step 5 do
      $ES \leftarrow ES \cup CH = \texttt{Glpk}(limit = time), IH = \emptyset$
    end for
  end for
  return $ES$
```

Figure 4.4: Time vs. Quality



The results of this test are captured in Figure 4.4. They should be interpreted as follows: for each time limit from 1 to 46 seconds there are two boxplots next to each other, the left, dashed one is the native GLPK, the right, solid one is the Cygwin GLPK. This is reflected in the tics on the X (time) axis, meaning that the axis cannot be interpreted in the usual way.

We can see from the graph that even though for smaller times (1 and 6 seconds, respectively) the Cygwin GLPK is reaching better qualities with smaller variance, starting from 11 seconds the native GLPK is at least as good or better for every following time. The results are inconclusive though, it is necessary to wait for confirmation from the second part of this experiment.

Input data	all sized test data sets
Iterations	50
Pool size	1
α, β	$1, 1$
CH	*Glpk*
IHs	$\emptyset$

The other way to compare the performance of these two GLPK versions is to see how long it takes them to find the optimum for a set of data of increasing size. This experimental set will contain 550 configurations for each version. Every configuration will let *Glpk* CH run for unlimited time, until it finds the optimum. This will be repeated in 50 iterations for each of the 11 files from the sized test data set (see Algorithm 17). There will again be no IH, the only data we will collect are the times of the CH run in each case.

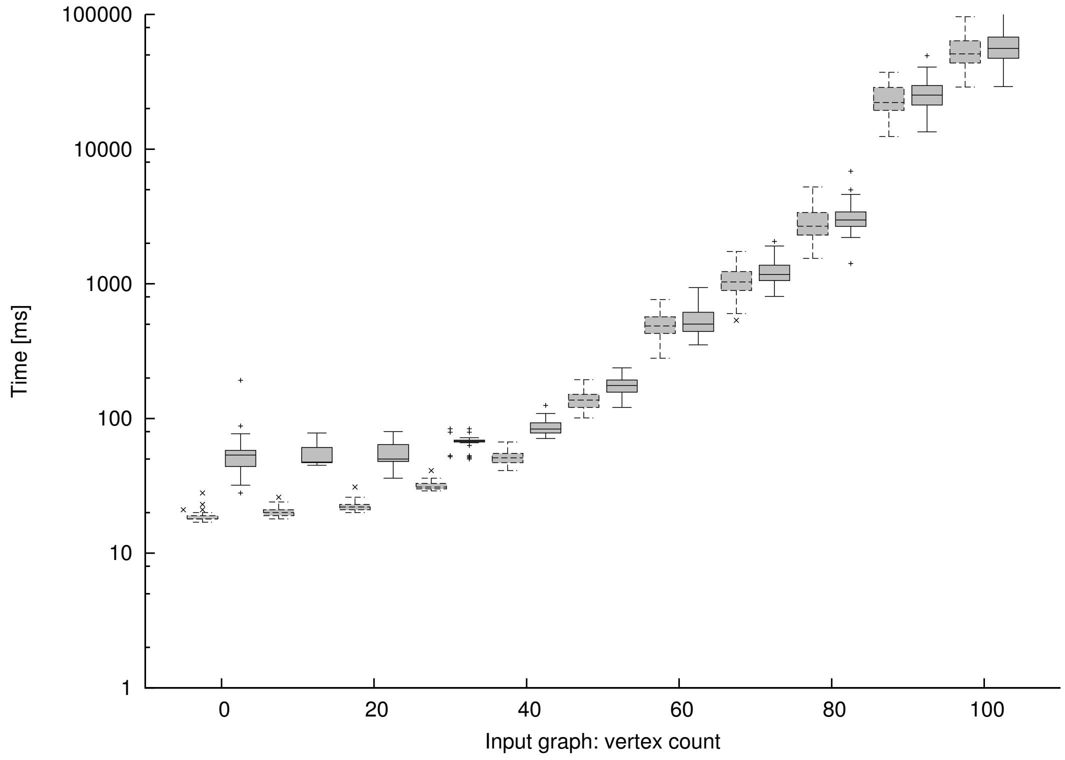
Algorithm 17 GLPK: Native vs. Cygwin Set Generation 2

```
Output: experimental set $ES$
  $ES \leftarrow \emptyset$
  for $i = 1 \rightarrow 50$ do
    for $file \in$ sized test data do
      $ES \leftarrow ES \cup \{file, CH = \mathit{Glpk}(no\,limit), IH = \emptyset\}$
    end for
  end for
  return $ES$
```

The results are captured in Figure 4.5; please take a note that the Y axis is in log scale. As with the previous case, the X axis cannot be interpreted in the usual way. For each data set there are two boxplots next to each other: the left one is the native GLPK, the right one is the Cygwin GLPK.

From these results it becomes clear that the native GLPK has in general shorter running times for each and every input data set than its Cygwin counterpart. This becomes less extreme with the increasing input size, which leads

Figure 4.5: Time Until Optimum



us to suspicion that the core parts of computation in both cases are equally powerful. Regardless of that, we shall be using the **native** GLPK for the following experiments.

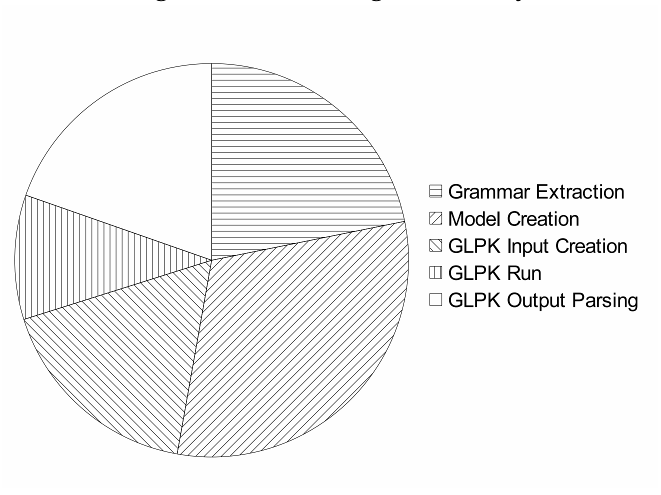
To conclude the first timing experiments we introduce a summary pie chart in Figure 4.6. This shows the typical distribution of times needed to find the optimum for the *OVA3* data set.

These experiments proved that for bigger data sets the times to reach the optimum might become too long. We shall attempt to find heuristics to reach the optimum faster in the following experiments.

4.3.3 Random **vs.** Fuzzy **vs.** FIDAX

Our investigation into various CHs will start by comparing *FIDAX* from the original article [BM03] to 2 of our trivial randomized hungry heuristics, *Random* and *Fuzzy* .

Figure 4.6: Timing Summary



Input data set	all official test data sets
Iterations	50
Pool size	10
α, β	1, 1
CH	*Random*, *Fuzzy*, *FIDAX*
IHs	$\emptyset$

The experimental set will contain 1650 configurations in total: 3 different CHs * 11 official test data sets * 50 iterations. There will be no improvement heuristics. The pool size will be set to 10, even though *FIDAX* cannot profit from this. Listing for this can be found in Algorithm 18.

We will be gathering the running time of the CH itself and quality of the best solution found for GnuPlot.

Results can be found in Figure 4.7 - qualities achieved and Figure 4.8 - times spent. The Y (time) axis in the latter figure is again in log scale. For each data set there are 3 boxplots next to each other. The first, leftmost, represents *Random*, second *Fuzzy* and finally the third, rightmost is *FIDAX*.

We can draw the following conclusions: *Fuzzy* consistently finds the best solution, but it is by far the slowest of these CHs. The trivial *Random* is better than *FIDAX* in artificial as well as some real data.

Algorithm 18 ***Random*** vs. ***Fuzzy*** vs. ***FIDAX*** Set Generation

```
Output: experimental set $ES$
  $ES \leftarrow \emptyset$
  for $i = 1 \to 50$ do
    for $file \in$ official test data do
      $ES \leftarrow ES \cup \{file, CH = \mathtt{Random}(pool = 10), IH = \emptyset\} \cup \{file, CH = \mathtt{Fuzzy}(pool = 10), IH = \emptyset\} \cup \{file, CH = \mathtt{FIDAX}, IH = \emptyset\}$
    end for
  end for
  return $ES$
```

Figure 4.7: Random vs. Fuzzy vs. FIDAX - Quality

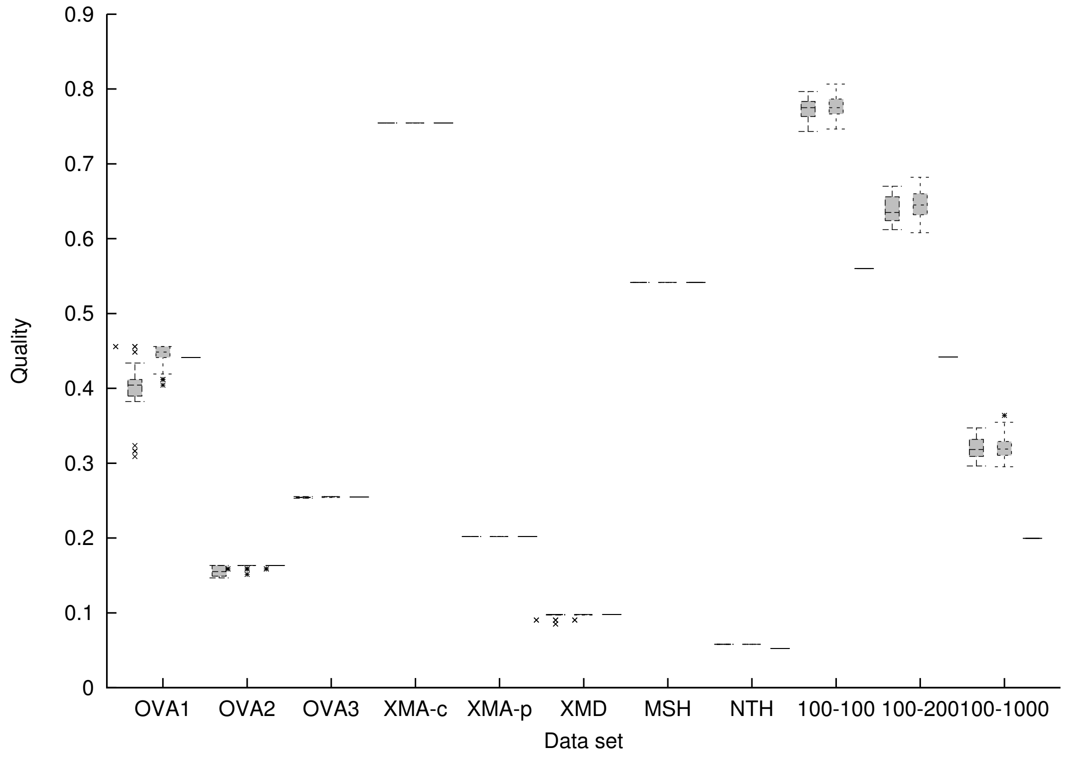
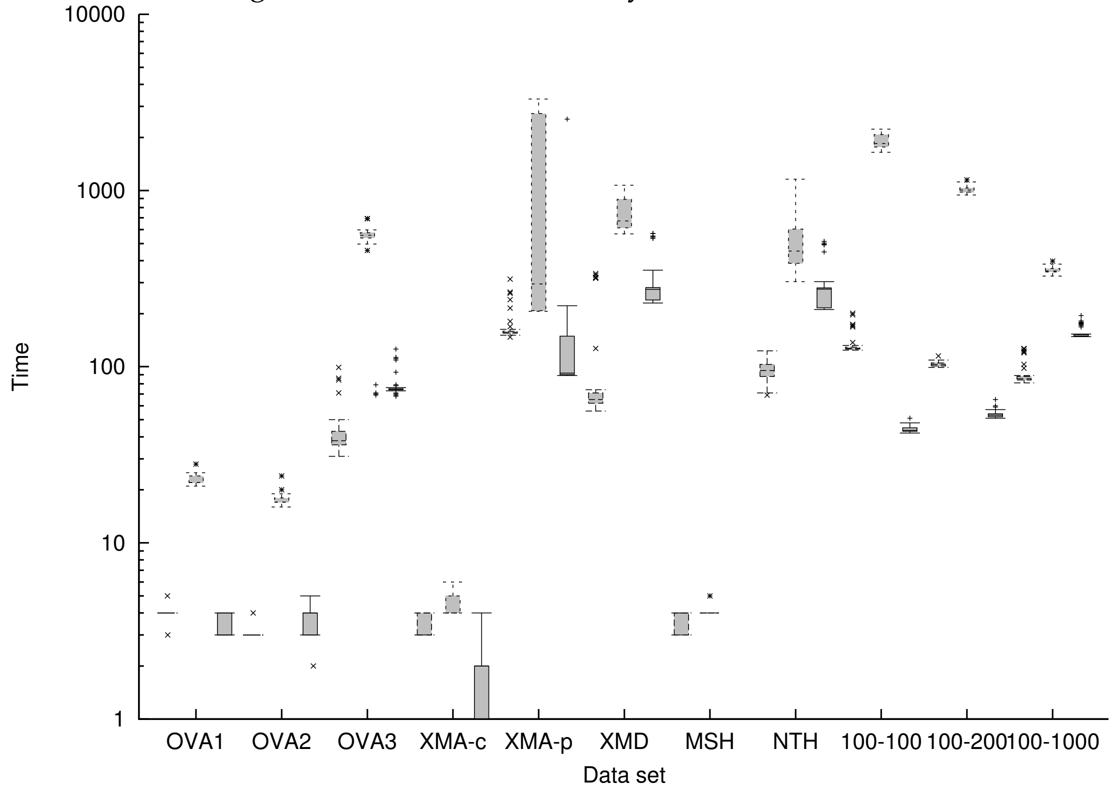


Figure 4.8: Random vs. Fuzzy vs. FIDAX - Time



Improving `FIDAX` with `Hungry`

Now we shall try to answer a minor question, whether it is possible to improve *`FIDAX`* by using *`Hungry`* as IH. This short experiment answers that question.

Input data set	all official test data sets
Iterations	1
Pool size	1
α, β	$1, 1$
CH	*`FIDAX`*
IHs	*`Hungry`* or $\emptyset$

We need a pool size of one and only a single iteration - both *`FIDAX`* and *`Hungry`* are deterministic. We will try all official data sets, first with empty IH, second with *`Hungry`* as IH. We will gather the qualities in each case and see whether there is any improvement.

Table 4.7: Results of adding *Hungry* after *FIDAX*

Data set	Quality - *FIDAX*	Quality - *FIDAX* + *Hungry*
OVA1	0.4411764705882353	0.4411764705882353
OVA2	0.16346153846153846	0.16346153846153846
OVA3	0.25482414123443264	**0.2553715615163541**
XMA-c	0.7546666666666666	0.7546666666666666
XMA-p	0.2019306150568969	0.2019306150568969
XMD	0.09786094165493509	0.09786094165493509
MSH	0.5416472778036296	0.5416472778036296
NTH	0.05259709474828076	**0.057918595422124436**
100-100	0.56	**0.6766666666666669**
100-200	0.44200000000000017	**0.5980000000000003**
100-1000	0.19952380952380955	**0.29619047619047617**

The experimental results are summarized in Table 4.7 and are quite surprising. As trivial a heuristic *Hungry* is, it is still able to improve the ID set found by *FIDAX* by as much as almost 50% (the last row, *100-1000*).

Table 4.8 lists the `ID` attributes found in both cases for this most extreme input, *100-1000*. Note that the content of each cell means "attribute `attr` in element `vertexXY` should be marked as `ID` attribute".

4.3.4 Best Standalone CH

We shall now try to find the best standalone CH, that is the CH that finds on average the best solutions when run without any IHs. We need to set a time limit for *Glpk* to make it an instance of *Truncated Branch & Bound*, and we shall use 1 second. This is the smallest time limit possible for GLPK and it is still a reasonably short time, fair to other CHs.

Table 4.8: ID Sets in *FIDAX* Versus *FIDAX* + *Hungry*

FIDAX	*FIDAX* + *Hungry*
	vertex5
	vertex26
vertex30	vertex30
vertex31	vertex31
vertex32	vertex32
vertex34	vertex34
vertex35	vertex35
vertex36	vertex36
vertex37	vertex37
vertex39	vertex39
	vertex60
	vertex69
	vertex70
vertex74	vertex74
vertex75	vertex75
vertex80	vertex80

Input data	all official test data sets
Iterations	50
Pool size	10
α, β	1, 1
CH	various
IHs	$\emptyset$

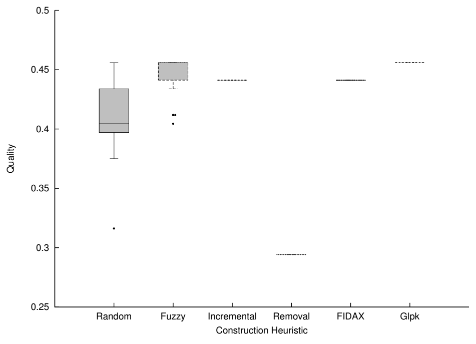
We will use all the official data sets, set the pool size to 10 where applicable, α and β to 1. This experiment will consist of 50 iterations * 11 data sets * 6 CHs = 3300 experimental configurations. See the Algorithm 19 for details. This time we are not interested in run times, only in qualities which we shall gather in a format for GnuPlot.

Algorithm 19 Best Standalone CH Set Generation

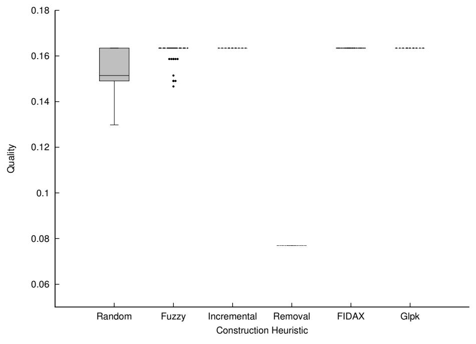
```
Output: experimental set $ES$
  $ES \leftarrow \emptyset$
  for $file \in$ official test data do
    for $i = 1 \to 50$ do
      $ES \leftarrow ES \cup \{file, CH = \texttt{Random}, IH = \emptyset\}$
      $ES \leftarrow ES \cup \{file, CH = \texttt{Fuzzy}, IH = \emptyset\}$
      $ES \leftarrow ES \cup \{file, CH = \texttt{Incremental}, IH = \emptyset\}$
      $ES \leftarrow ES \cup \{file, CH = \texttt{Removal}, IH = \emptyset\}$
      $ES \leftarrow ES \cup \{file, CH = \texttt{FIDAX}, IH = \emptyset\}$
      $ES \leftarrow ES \cup \{file, CH = \texttt{Glpk}(limit = 1), IH = \emptyset\}$
    end for
  end for
  return $ES$
```

For data sets *XMA-c*, *XMA-p*, *MSH* and *NTH* every CH found the optimum every time. Graphs representing the results for remaining data sets can be found in Figure 4.9.

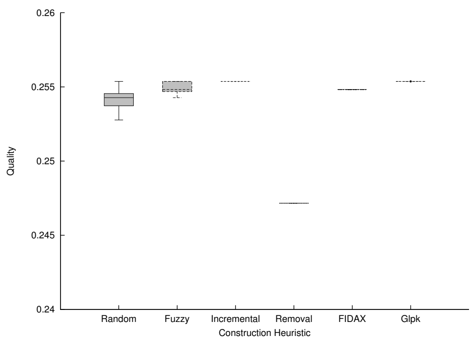
Figure 4.9: Best Standalone CH



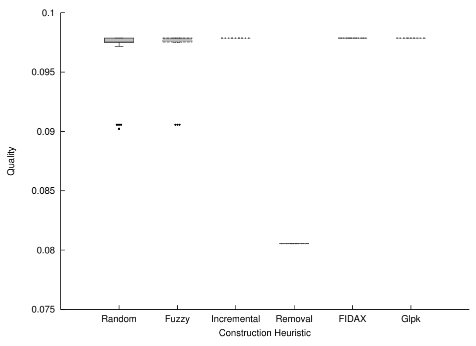
(a) `OVA1`



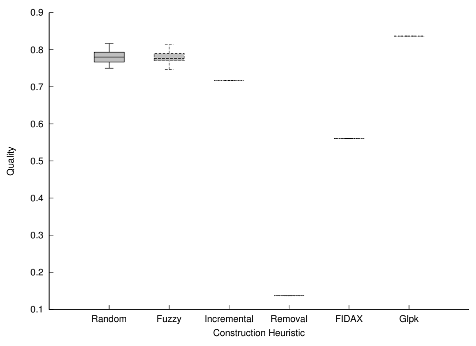
(b) `OVA2`



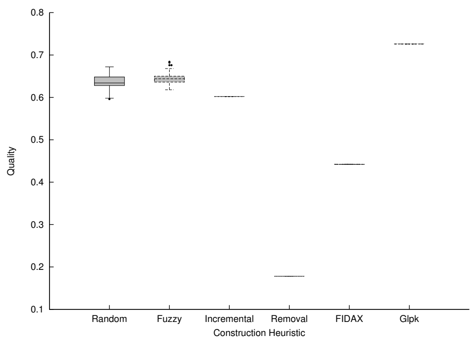
(c) `OVA3`



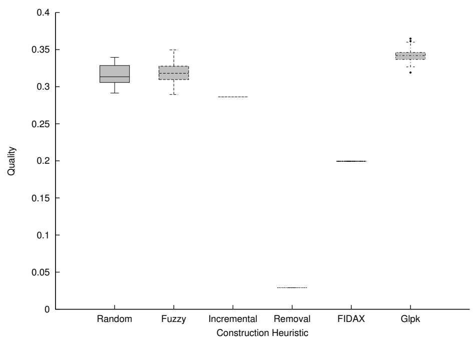
(d) `XMD`



(e) `100-100`



(f) `100-200`



(g) `100-1000`

We can see that *Glpk* wins (or is among the best) in every single case. We will start from there and try to build upon this result.

4.3.5 Best IH for `Glpk`

The next logical step is to try to add one IH after the best CH we have found, *Glpk*. We will investigate all IHs except for *RandomRemove* and *RemoveWorst*, which cannot help us at this time.

We should note that the combination *best CH - best IH* found this way does not necessarily need to be the best one overall, because we find it using a hungry approach.

Input data	*80-30, 90-405, 100-500, 100-100, 100-200, 100-1000*
Iterations	50
Pool size	10
α, β	1, 1
CH	*Glpk*
IHs	*Crossover, Hungry, Local Branching, Mutation*

This experimental set will contain 6 data sets * 50 iterations * 4 IHs = 1200 experimental configurations. Note that we are using only the most challenging data sets, as the combination of *Glpk* as CH and any other IH is already an overkill for easier data sets.

The results are listed in Table 4.9. We shall denote *improvement* the absolute increase in quality after running *Glpk* and after running the IH. The table now lists for each data set and each IH the average improvement as well as the standard deviation of the improvement. Bold number represents the best IH for that specific data set. *Mutation* proves to be the best IH for 3 out of 6 data sets.

Table 4.9: Best IH for *Glpk*

Data set	*Hungry* improv - avg	*Hungry* improv - stdev	*Crossover* improv - avg	*Crossover* improv - stdev
80-320	0.00017	0.00118	0.00017	0.00118
90-405	0.00502	0.00618	0.00033	0.00165
100-500	0.00664	0.00667	0.00016	0.00081
100-100	0.00000	0.00000	0.00000	0.00000
100-200	0.00000	0.00000	0.00000	0.00000
100-1000	0.01630	0.01294	0.00180	0.00506

Data set	*LB* improv - avg	*LB* improv - stdev	*Mutation* improv - avg	*Mutation* improv - stdev
80-320	**0.00072**	0.00223	0.00064	0.00218
90-405	0.00698	0.00616	**0.00851**	0.00659
100-500	0.00796	0.00797	**0.00964**	0.00804
100-100	0.00000	0.00000	0.00000	0.00000
100-200	0.00000	0.00000	0.00000	0.00000
100-1000	0.01710	0.01188	**0.02337**	0.01558

Algorithm 20 Best IH for `Glpk` Set Generation

```
Output: experimental set $ES$
  $ES \leftarrow \emptyset$
  for $file \in \{$80-30, 90-405, 100-500, 100-100, 100-200, 100-1000$\}$ do
    for $i = 1 \rightarrow 50$ do
      $ES \leftarrow ES \cup \{file, CH = \texttt{Glpk}(limit = 1), IH = \texttt{Crossover}(fraction = 0.1, limit = 1)\}$
      $ES \leftarrow ES \cup \{file, CH = \texttt{Glpk}(limit = 1), IH = \texttt{Hungry}\}$
      $ES \leftarrow ES \cup \{file, CH = \texttt{Glpk}(limit = 1), IH = \texttt{Local Branching}(fraction = 0.1, limit = 1)\}$
      $ES \leftarrow ES \cup \{file, CH = \texttt{Glpk}(limit = 1), IH = \texttt{Mutation}(fraction = 0.1, limit = 1)\}$
    end for
  end for
  return $ES$
```

`Random` as CH

As we mentioned before, we chose the combination `Glpk` and `Mutation` in a hungry manner. We will now try to take a step back and attempt to replace `Glpk` with `Random`, hoping to get similar qualities in much shorter time (a reminder: `Glpk` always takes 1 second).

Input data	`80-30`, `90-405`, `100-500`, `100-100`, `100-200`, `100-1000`
Iterations	50
Pool size	10
α, β	1, 1
CH	`Random` or `Glpk`
IHs	`Mutation`

Setup used will be almost identical to that from the previous experiment. Experimental set will consist of 6 data sets * 50 iterations * 2 CHs = 600 experimental configurations, see Algorithm 21. We shall collect the eventual quality after running both the CH and the IH in format suited for GnuPlot.

Algorithm 21 *Random* as CH Set Generation

```
Output: experimental set $ES$
  $ES \leftarrow \emptyset$
  for $file \in \{$80-30, 90-405, 100-500, 100-100, 100-200, 100-1000$\}$ do
    for $i = 1 \rightarrow 50$ do
      $ES \leftarrow ES \cup \{file, CH = $ Random$, IH = $ Mutation$(fraction = 0.1, limit = 1)\}$
      $ES \leftarrow ES \cup \{file, CH = $ Glpk$(limit = 1), IH = $ Mutation$(fraction = 0.1, limit = 1)\}$
    end for
  end for
  return $ES$
```

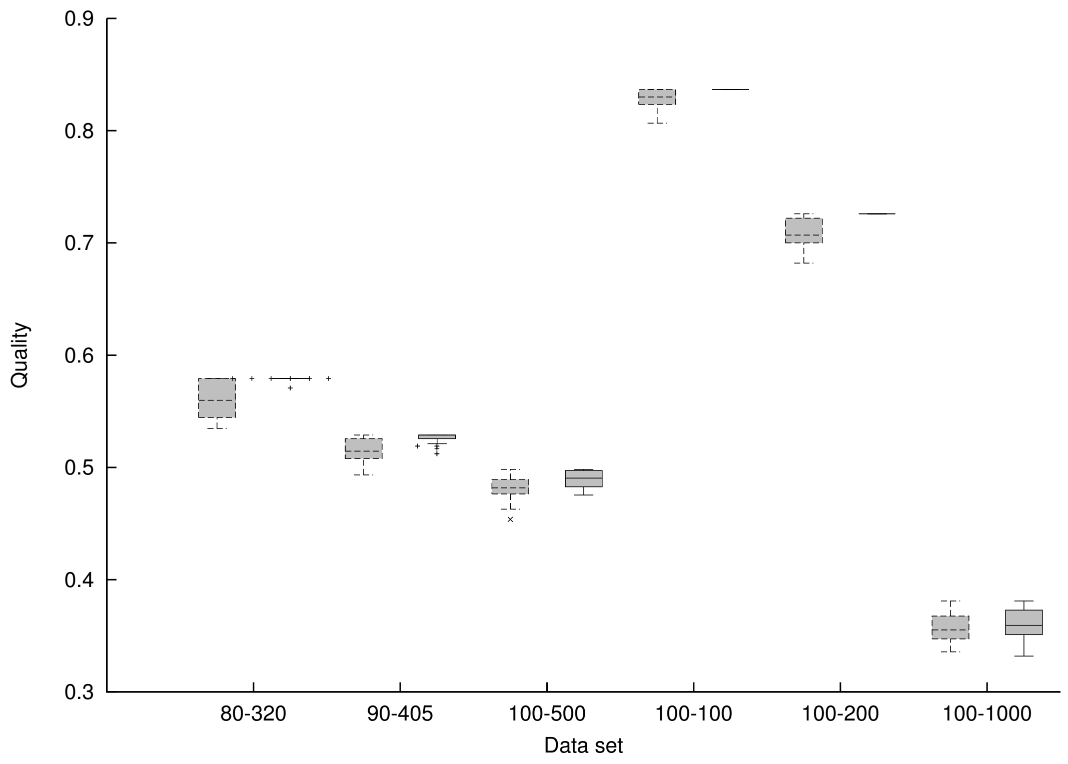
Results are summarized in Figure 4.10. Again, for each data set there are two boxplots representing *Random* (left one) and *Glpk* (right one). The combination *Glpk* + *Mutation* always finds the optimum for the simpler data sets, thus the collapsed boxplots. Moreover, it achieves higher quality in each data set. On the other hand, combination *Random* + *Mutation* has much shorter running times and in the biggest (and hardest) data set *100-1000* has almost comparable results. This makes it a reasonable choice for big inputs where short time is more important than optimal quality.

4.3.6 Various α, β

After finding the best combination of a CH and IH we turn our attention to parameters. The first ones are the α and β from the definition of our weight function (Section 1.5). A short reminder: the weight is defined as follows.

$$weight(m) = \alpha.\phi(m) + \beta.\chi(m)$$

Figure 4.10: CH for *Mutation*



Where $\phi(m)$ is support the attribute mapping m and $\chi(m)$ is its coverage.

It is thus obvious that only the *ratio* between α and β matters, not their actual values. This means that investigating effects of these parameters is in fact a 1-dimensional problem. However, for the sake of simplicity we will use 25 combinations of various α and β and normalize them only during evaluation.

We do not expect any changes in performance of heuristics and we will limit the inquiry to different ID sets produced under different settings.

Input data	realistic + converted official test data sets
Iterations	1
Pool size	1
α, β	$\{0.1, 0.25, 0.5, 0.75, 1\} \times \{0.1, 0.25, 0.5, 0.75, 1\}$
CH	*Glpk*
IHs	$\emptyset$

This experimental set will contain 5 different α settings * 5 β settings * 8 data sets = 200 experimental configurations. We are not using the artificial data sets, because due to the way they are generated (attribute values are random numbers), they cannot possibly create different optimal ID sets. The pseudocode capturing this is provided in Algorithm 22. We will use *Glpk* constrained to 1 second (thus making it an instance of *Truncated Branch & Bound*) and no IHs. Pool size as well as iteration count will be 1. We are noting the actual ID set found by the run of the heuristic.

Algorithm 22 Various Values of α and β Set Generation

```
Output: experimental set $ES$
  $ES \leftarrow \emptyset$
  for $\alpha \in \{0.1, 0.25, 0.5, 0.75, 1\}$ do
    for $\beta \in \{0.1, 0.25, 0.5, 0.75, 1\}$ do
      for $file \in$ realistic of converted official test data do
        $ES \leftarrow ES \cup \{file, CH = \texttt{Glpk}(limit = 1, alpha = \alpha, beta = \beta), IH = \emptyset\}$
      end for
    end for
  end for
  return $ES$
```

The following data sets have the same optimal ID sets regardless of the setting of α and β: *MSH*, *NTH*, *XMA-c*, *XMA-p*. The *OVA** data sets showed various dependencies on α and β; we shall now describe one representative example.

Results for OVA1

The 2 different ID sets found for various α and β in *OVA1* are listed in Table 4.10 (note that the actual names had to be anonymized for reasons discussed in Section 4.1.1). The differing attribute mapping is marked.

Table 4.11 summarizes the dependency of the ID set found on various values of α, β. We then define the $\alpha - ratio$ as $\frac{\alpha}{\alpha+\beta}$ and summarize the findings in a linear manner, sorted by increasing $\alpha - ratio$ in Table 4.12. Note that the

Table 4.10: Different ID Sets Found for *`OVA1`*

ID set **1**: element@attribute	ID set **2**: element@attribute
`aff@fa`	`aff@fa`
`com@ty`	`com@ty`
`cre@da`	`cre@da`
`cri@te`	`cri@te`
`cve@st`	`cve@st`
`* def@id`	`* def@cl`
`fil@co`	`fil@co`
`mod@da`	`mod@da`
`ova@xs`	`ova@xs`
`pat@op`	`pat@op`
`sof@op`	`sof@op`
`sta@da`	`sta@da`
`sub@or`	`sub@or`
`sbt@te`	`sbt@te`

Table 4.11: Effect of α, β on ID Set Found for `OVA1`

$\alpha \backslash \beta$	0.1	0.25	0.5	0.75	1
0.1	1	2	2	1	1
0.25	2	2	2	1	2
0.5	2	1	1	1	2
0.75	1	2	1	2	1
1	1	2	1	1	2

$\alpha - ratio$s are not unique due to the way we constructed the experimental configurations here.

Interestingly enough, there is no clear separation between the two ID sets depending on the $\alpha - ratio$ to be found. The existence of the two sets might be due to the fact that `Glpk` randomizes the order in which AMs are presented to the external GLPK solver. However, this question is beyond the scope of this work, and shall be left for future work.

4.3.7 Ignoring Text Data

When considering data sets such as `XMA-p`, we notice that they contain a lot of simple text nodes that do not contribute to our search, but possibly slow it down. Precisely for this reason the `BasicIGG` module in jInfer contains an option to turn off processing of such nodes. (It also allows to ignore the content of attributes, but this would be devastating to our cause.) Ignoring the content of text nodes means internally that these are created, but their actual string content is skipped and not saved in the memory structures. This means that the whole data model occupies less space on the heap, which can possibly lead to better performance.

We shall now investigate this matter by taking the biggest data set `XMA-p` containing a lot of text data.

Table 4.12: Effect of $\alpha - ratio$ on ID Set Found for `OVA1`

$\alpha - ratio$	ID set	$\alpha - ratio$	ID set
0,091	1	0,500	2
0,118	1	0,500	2
0,167	2	0,571	1
0,200	2	0,600	1
0,250	1	0,667	1
0,286	2	0,667	1
0,333	2	0,714	2
0,333	2	0,750	2
0,400	1	0,800	2
0,429	1	0,833	2
0,500	1	0,882	1
0,500	2	0,909	1
0,500	1		

Input data	XMA-p
Iterations	50
Pool size	1
α, β	1, 1
CH	Glpk
IHs	not applicable

Our experimental set will contain 50 iterations * 2 = 100 experimental configurations as described in Algorithm 23. We will be using Glpk limited to 1 second with no additional IH and pool size set to 1. After the first 50 iterations we will turn on the option to ignore the simple text node data and run the same 50 iterations again. We will be collecting the grammar extraction (GE) and model creation (MC) times as in the experiment in Section 4.3.1.

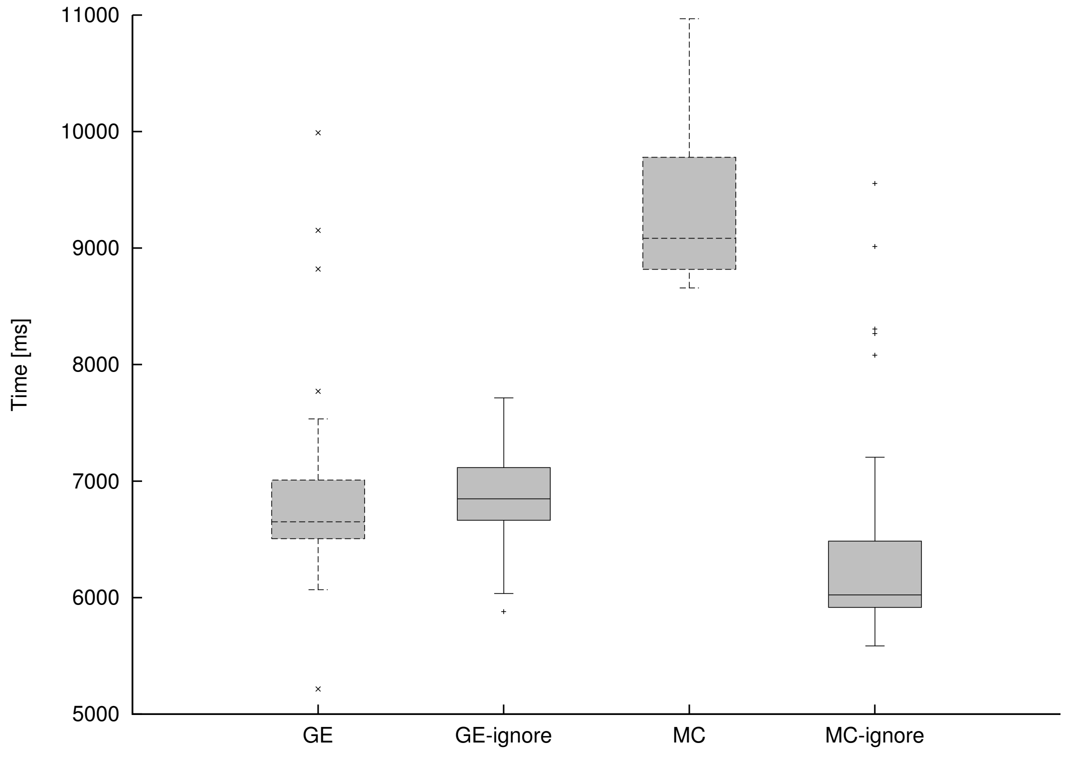
Algorithm 23 Ignoring Text Data Set Generation

```
Output: experimental set $ES$
  $ES \leftarrow \emptyset$
  for $i \in 1 \to 50$ do
    $ES \leftarrow ES \cup \{\texttt{XMA-p}, CH = \texttt{Glpk}(limit = 1), IH = \emptyset\}$
  end for
  set "ignore text data"
  for $i \in 1 \to 50$ do
    $ES \leftarrow ES \cup \{\texttt{XMA-p}, CH = \texttt{Glpk}(limit = 1), IH = \emptyset\}$
  end for
  return $ES$
```

The results are summarized in Figure 4.11. Boxplots drawn in dashed lines represent the original case, not ignoring the text data. Solid lines represent the case where we ignore the text data.

Interestingly, the grammar extraction times tend to be shorter in the case when text data is not ignored, although this is inconclusive. However, there

Figure 4.11: Ignoring Text Data



is a clear improvement of about 50 % in case of model creation times. The conclusion then is to ignore the simple text node content whenever possible when finding `ID` attributes.

4.3.8 Chaining the IHs

In this section we will describe the most interesting experimental area, that is chaining more than one improvement heuristics and running them in a loop. Unfortunately, the sheer number of possible combinations in which IHs can be ordered (as well as the number of ways to set their parameters) prohibits us from investigating this in depth.

We shall then employ a higher-level heuristic: we will choose 3 strategies (lists of IHs, or metaheuristics), assess their performance to find the best one and then tune its parameters. This approach is by no means exhaustive, it is just a probe in the problem space.

The 3 strategies we assess will be constructed from the following instances of improvement heuristics:

- `RR` is `RandomRemove` with $fraction$ set to 0.1.
- `MUT` is `Mutation` with $fraction$ set to 0.1 and time limit set to 1 second.
- `CX` is `Crossover` with $fraction$ set to 0.1 and time limit set to 1 second.
- `LB` is `Local Branching` with $fraction$ set to 0.1 and time limit set to 1 second.
- `RW` is `RemoveWorst`.
- `H` is `Hungry`.

The strategies themselves shall be the following:

- ***Strategy 1.*** `RR` $\rightarrow$ `MUT` $\rightarrow$ `RR` $\rightarrow$ `CX` $\rightarrow$ `RW` $\rightarrow \dots$
- ***Strategy 2.*** `CX` $\rightarrow$ `RW` $\rightarrow$ `MUT` $\rightarrow \dots$
- ***Strategy 3.*** `CX` $\rightarrow$ `RR` $\rightarrow$ `MUT` $\rightarrow$ `RW` $\rightarrow$ `LB` $\rightarrow$ `RW` $\rightarrow \dots$

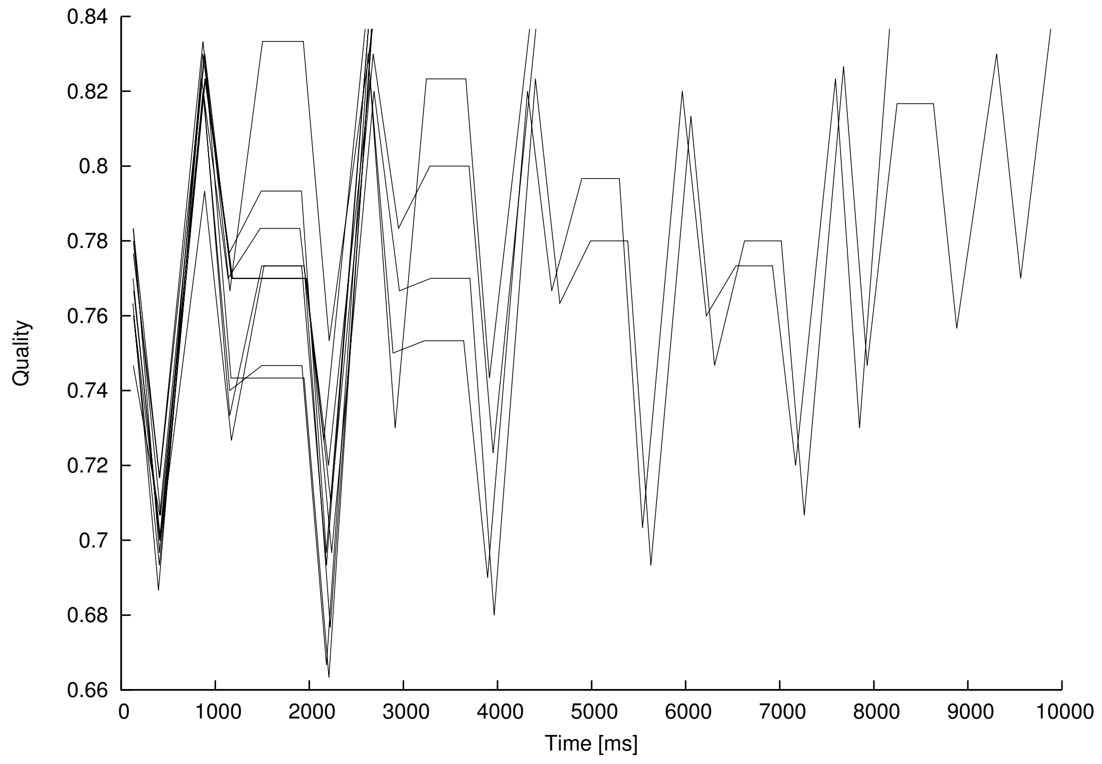
Input data	all official test data sets
Iterations	20
Pool size	10
α, β	$1, 1$
CH	`Random`
IHs	various

The experimental set will consist of 3 strategies * 11 data sets * 20 iterations = 660 experimental configurations. Their construction is formalized in the Algorithm 24. The construction heuristic will be `Random` with pool size 10. All the fractions are set to 0.1 for the time being. The termination criterion is set to limit the total runtime to 10 seconds and (potentialy) infinite iterations.

Algorithm 24 Chaining IHs Set Generation

```
Output: experimental set $ES$
  $ES \leftarrow \emptyset$
  $\texttt{MUT} \leftarrow \texttt{Mutation}(fraction = 0.1, limit = 1)$
  $\texttt{CX} \leftarrow \texttt{Crossover}(fraction = 0.1, limit = 1)$
  $\texttt{LB} \leftarrow \texttt{Local Branching}(fraction = 0.1, limit = 1)$
  $\texttt{RR} \leftarrow \texttt{RandomRemove}(fraction = 0.1)$
  $\texttt{H} \leftarrow \texttt{Hungry}$
  $\texttt{RW} \leftarrow \texttt{RemoveWorst}$
  $IHs \leftarrow \emptyset$
  $IHs \leftarrow IHs \cup (\texttt{RR}, \texttt{MUT}, \texttt{RR}, \texttt{CX}, \texttt{RW})$
  $IHs \leftarrow IHs \cup (\texttt{CX}, \texttt{RW}, \texttt{MUT})$
  $IHs \leftarrow IHs \cup (\texttt{CX}, \texttt{RR}, \texttt{MUT}, \texttt{RW}, \texttt{LB}, \texttt{RW}, \texttt{H})$
  for $ih \in IHs$ do
    for $file \in$ official test data do
      for $i = 1 \rightarrow 20$ do
        $ES \leftarrow ES \cup \{file, CH = \texttt{Random}, IH = ih, limit = 10seconds\}$
      end for
    end for
  end for
  return $ES$
```

Figure 4.12: Chained IHs - `100-100` Results for Strategy 1



Log traces like the one in Appendix C will be gathered - also, after each iteration, the time taken so far and the quality of incumbent solution is noted.

Resulting traces for each data set can be summarized in graphs like the one for `100-100` in Figure 4.12. This one deserves more explanation than usual.

X and Y axes represent the time and quality, as usual. Each line represents one run of the strategy (metaheuristic) in the following way: the Nth break in the line (i.e. the Nth data point) is the partial result after the Nth step of the strategy. Its X position denotes the absolute time in which this step finished, and its Y position represents the incumbent solution quality after this step. Every time a line "disappears" before reaching 10 seconds it means that this metaheuristic run found the optimum before the 10 second mark. There is an obvious repetitive regularity in the shape of each line, which corresponds to the fact that there is a finite number of IHs in this strategy (5 of them in Strategy 1) which repeat over time. The obvious similarity between different lines corresponds to the fact that each run is from the same strategy, and over time, they do the same steps.

We can see effects of different IHs from this graph:

- Every $(1+5k)^{\text{th}}$ and $(3+5k)^{\text{th}}$ step is a *RandomRemove*, and each time this happens there is a rather sharp drop in quality
- Every $(2+5k)^{\text{th}}$ step is a *Mutation*, and there is a consistent increase in quality each time.
- Every $(4+5k)^{\text{th}}$ step is a *Crossover*, and each time it happens there is a consistent increase, yet smaller than with *Mutation*.
- Every $5k^{\text{th}}$ step is a *RemoveWorst*, and as expected, this removes the worst solution not touching the best ones that decide the incumbent quality. The line thus stays flat every time it happens.

In this particular example there is only 1 run out of 10 that does not finish (find optimum) under the 10 second mark.

There are two more graphs like this in Figures 4.13 and 4.14 for comparison, capturing Strategy 2 and Strategy 3 respectively working on the same data set, *100-100*. Describing them in detail is beyond the scope of this work.

It is now necessary to assess which of the strategies perform the best. We shall take a look at the different data sets. Easily we can discard *MSH*, *NTH*, *XMA-c*, *XMA-p*, because the optimum is found in the very first step. Let us now introduce a metric for assessment of a strategy: namely how many times of the 20 runs it found the optimum. The respective results are summarized in Table 4.13.

Each cell contains the number of times the strategy found optimum in the data set, out of 20 runs. The strategies that performed best on that data set are highlited. We can see that Strategies 1 and 3 are very similar in performance. We shall nonetheless choose Strategy **1** as the winner for its simplicity. Now we can tune its parameters.

Figure 4.13: Chained IHs - *100-100* Results for Strategy 2

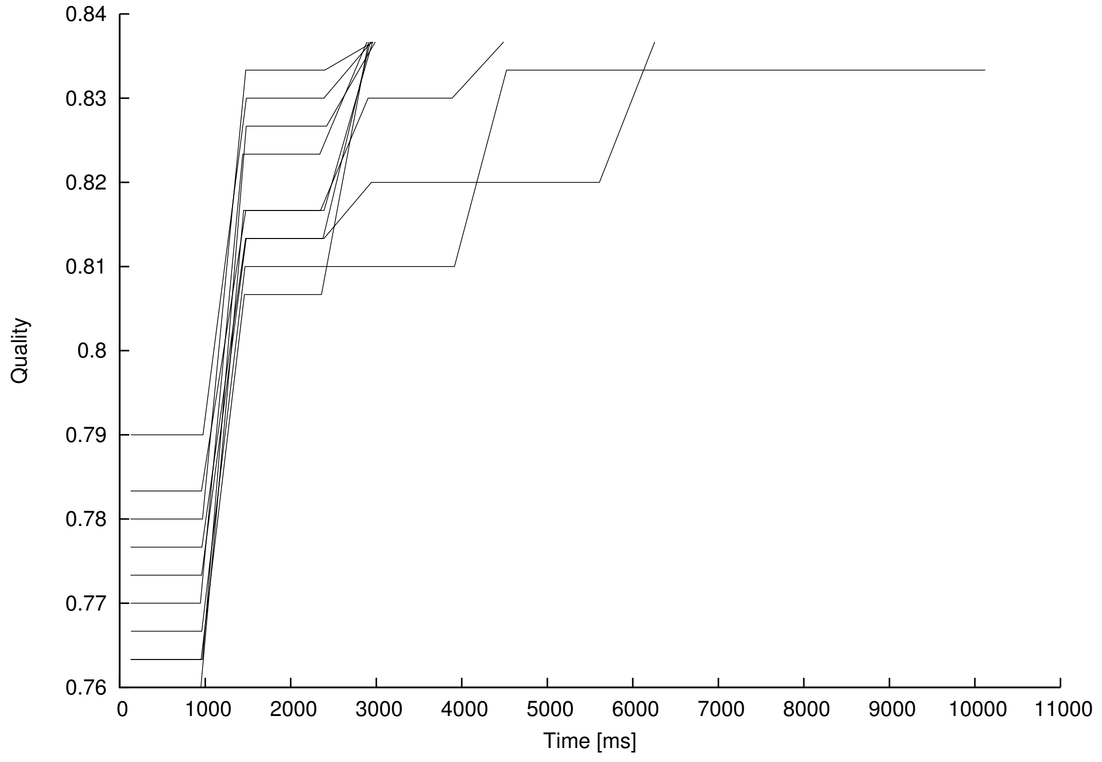


Figure 4.14: Chained IHs - *100-100* Results for Strategy 3

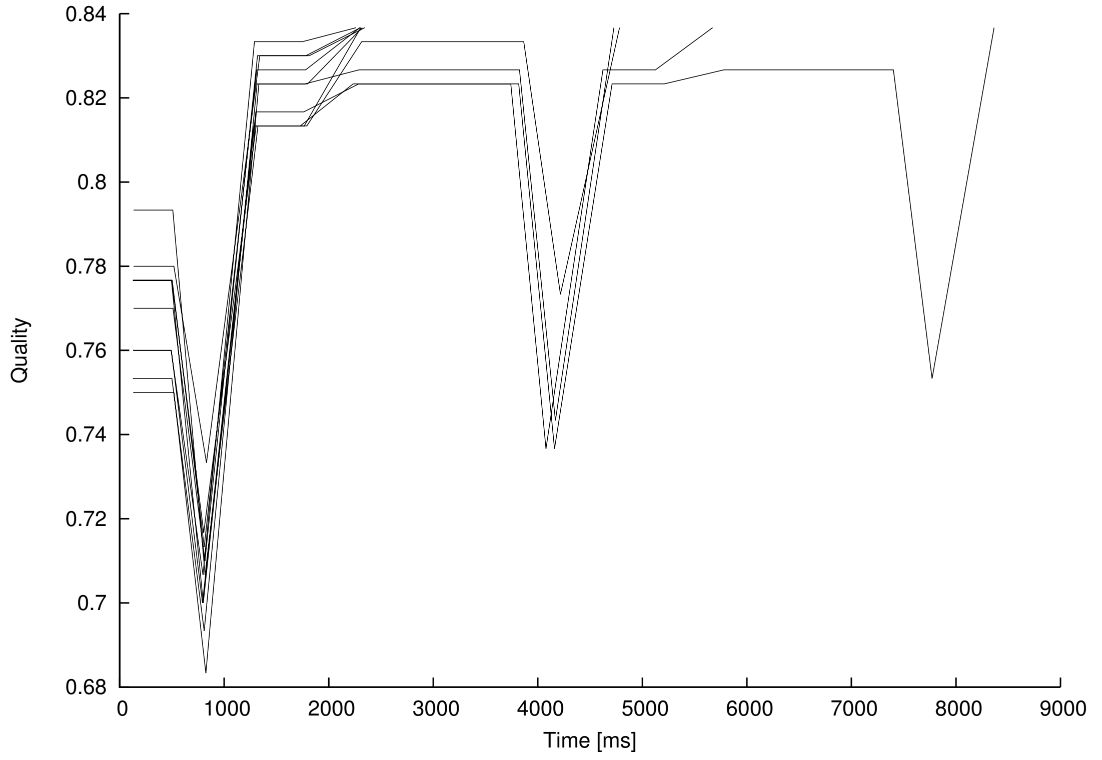


Table 4.13: Performance of Various IH Chains

Dataset	Strategy 1	Strategy 2	Strategy 3
`100-100`	**20**	19	**20**
`100-200`	**19**	18	17
`100-1000`	4	1	**5**
`OVA1`	**20**	**20**	**20**
`OVA2`	**19**	13	18
`OVA3`	17	18	**20**

Tuning Strategy 1

A short reminder: Strategy 1 consists of `Random` as the CH and the following IHs: `RR` $\to$ `MUT` $\to$ `RR` $\to$ `CX` $\to$ `RW` $\to \ldots$

The parameters we can tune in this strategy are the fractions in `RandomRemove` (possibly 2 of them, as there are 2 instances in use), `Mutation` and `Crossover`. We shall not tune the time limits in `Mutation` and `Crossover` and leave them set to 1 second. This presents us with a 3-dimensional space of parameters, where we want to find a combination best suited for our test data sets. We will sample this space by taking a total of 45 configurations of the aforementioned fractions.

Input data	all sized test data sets
Iterations	25
Pool size	10
α, β	1, 1
CH	`Random`
IHs	`RR` $\to$ `MUT` $\to$ `RR` $\to$ `CX` $\to$ `RW` $\to \ldots$

This experimental set will consist of 45 fraction combinations * 11 data sets * 25 iterations = 12375 experimental configurations. CH will be `Random` with pool size of 10. IHs will be the ones from Strategy 1, with their fractions set to one of the 45 combinations produced in the following way.

- `RandomRemove` fraction will be from $\{0, 0.05, 0.1, 0.2, 0.5\}$
- `Mutatio` fraction will be from $\{0.05, 0.1, 0.2\}$
- `Crossover` fraction will be from $\{0.05, 0.1, 0.2\}$

The total limit will remain at 10 seconds. The process of creating the configurations is captured in Algorithm 25. We will be gathering the following information for each run: what were the parameters, how long did the run take and whether it found the optimum.

Algorithm 25 Chained IHs - Improving Strategy 1 Set Generation

```
Output: experimental set $ES$
  $ES \leftarrow \emptyset$
  $RW \leftarrow$ RemoveWorst
  for $rrFraction \in \{0, 0.05, 0.1, 0.2, 0.5\}$ do
    for $mutFraction \in \{0.05, 0.1, 0.2\}$ do
      for $cxFraction \in \{0.05, 0.1, 0.2\}$ do
        $RR \leftarrow$ RandomRemoval $(fraction = rrFraction)$
        $MUT \leftarrow$ Mutation $(fraction = mutFraction, limit = 1)$
        $CX \leftarrow$ Crossover $(fraction = cxFraction, limit = 1)$
        for $file \in$ sized test data do
          for $i = 1 \rightarrow 50$ do
            $ES \leftarrow ES \cup \{file, CH =$ Random $, IH =$ (RR, MUT, RR, CX, RW) $, limit = 10seconds\}$
          end for
        end for
      end for
    end for
  end for
  return $ES$
```

After averaging the data, we get a large result table; an excerpt of it is in

Table 4.14: Performance of Strategy 1 Depending on Parameters - Excerpt

RR	*MUT*	*CX*		*60-180*	*70-245*	*80-320*	*90-405*	*100-500*
...	...	...	...	...	...	...	...	...
0.5	0.1	0.1	AT	1049.72	1584.72	2349.2	7408.28	9414.28
0.5	0.1	0.1	SR	1	1	1	0.52	0.28
0.5	0.1	0.1	AQ	*opt*	*opt*	*opt*	0.52533	0.49196
0.5	0.1	0.2	AT	763.44	1343.08	2599.88	9448.12	10269.76
0.5	0.1	0.2	SR	1	1	1	0.24	0
0.5	0.1	0.2	AQ	*opt*	*opt*	*opt*	0.52213	0.48462
0.5	0.2	0.05	AT	1438.84	1608.32	2647.4	4954.04	7784.88
0.5	0.2	0.05	SR	1	1	1	0.8	0.4
0.5	0.2	0.05	AQ	*opt*	*opt*	*opt*	0.52693	0.49647
0.5	0.2	0.1	AT	1333.12	1741.08	2506.84	4720.32	8150.6
0.5	0.2	0.1	SR	1	1	1	**0.84**	**0.56**
0.5	0.2	0.1	AQ	*opt*	*opt*	*opt*	0.52733	**0.49651**
0.5	0.2	0.2	AT	922.16	1353.76	2394.48	4633.12	7424.8
0.5	0.2	0.2	SR	1	1	1	**0.84**	0.44
0.5	0.2	0.2	AQ	*opt*	*opt*	*opt*	**0.52804**	0.49495

Table 4.14. Only the results for the biggest data sets and a few combinations of RR, MUT and CX fractions are presented.

In the left part of the table are the fraction values. In the right part are the averaged running times (AT), success ratios (ratio of runs that found the optimum, SR) and average qualities (AQ) for each data set. The highest success ratios and qualities are highlited.

It is now necessary to pick one fraction combination as the best one, and it is $(RR = 0.5, MUT = 0.2, CX = 0.1)$. Using this combination for all the data sets from *10-20* up to *80-320* the optimum was always found and for *90-405* and *100-500* the success ratios were the highest.

Now to interpret the fractions in the best combination. `RandomRemove` fraction of 0.5 means that a randomly chosen half of all AMs from every ID set in the pool will be discarded. This amounts to a very strong diversification tendency and keeps the strategy from stalling in local optima. `Mutation` fraction of 0.2 means around 1/5th of AMs in the incumbent solution will be fixed for the next GLPK optimization. `Crossover` fraction of 0.1 means that around 10% of ID sets in the pool (randomly chosen) will be scanned for common AMs.

`RandomRemove` and `Mutation` fractions in the best combination are at the upper bound of the range we chose for them. As a future work option it is possible to start moving these fractions even more in their preferred way.

Final Comparison

Finally we shall compare the performance of Strategy 1 with tuned parameters (fractions) to the approach we started the experiments with: using the `Glpk` CH with no time limit to find the optimum. We will compare them on the biggest of sized test data: `100-500`.

We already have the running times for pure `Glpk` on `100-500` from the "Time Until Optimum" experiment in Section 4.3.2. The last experiment to find the performance of tuned Strategy 1 without a time limit will have the following parameters.

Input data	`100-500`
Iterations	50
Pool size	10
α, β	$1, 1$
CH	`Random`
IHs	`RR` $\rightarrow$ `MUT` $\rightarrow$ `RR` $\rightarrow$ `CX` $\rightarrow$ `RW` $\rightarrow \ldots$

The last experimental set will consist of 50 iterations = 50 experimental configurations. As with the previous one, `Random` will be the CH, IHs will be from Strategy 1, however this time there will be no time limit. The process of generating the experimental set is in Algorithm 26.

Algorithm 26 Chained IHs - Tuned Strategy 1 Performance Set Generation

```
Output: experimental set $ES$
  $ES \leftarrow \emptyset$
  $RW \leftarrow \texttt{RemoveWorst}$
  $RR \leftarrow \texttt{RandomRemoval}(fraction = 0.5)$
  $MUT \leftarrow \texttt{Mutation}(fraction = 0.2, limit = 1)$
  $CX \leftarrow \texttt{Crossover}(fraction = 0.1, limit = 1)$
  for $i = 1 \to 50$ do
    $ES \leftarrow ES \cup \{\texttt{100-500}, CH = \texttt{Random}, IH = (\texttt{RR}, \texttt{MUT}, \texttt{RR}, \texttt{CX}, \texttt{RW})\}$
  end for
  return $ES$
```

Results are summarized in Figure 4.15. Both boxplots represent run times until the optimum is found. It is clear that Strategy 1 is an improvement, achieving on average almost 4x shorter times than pure *Glpk* and finding the optimum under 10 seconds in more than a half of the cases.

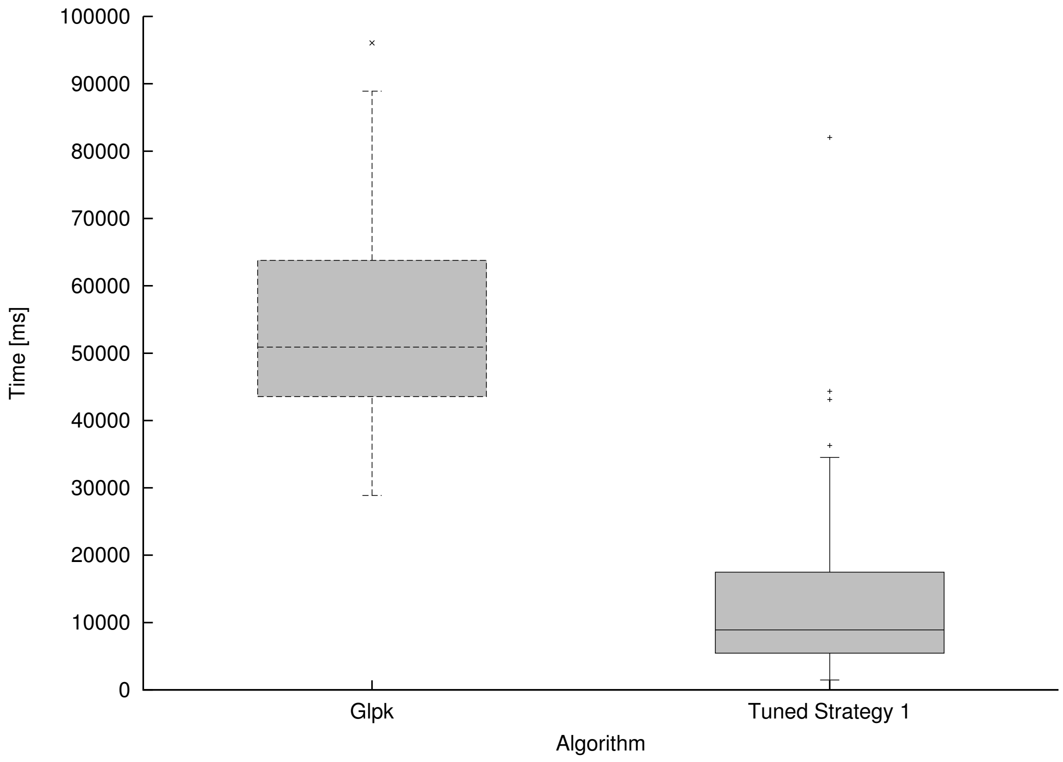
4.4 The "Best" Algorithm

After answering a lot of questions related to the overall system behavior, parameter effects and various heuristic combinations we can now summarize our results and draw conclusions.

The first fact is that if we have the time available, it is best to just let the GLPK run. It will find the optimum eventually, even though this might take minutes or hours to complete. For many purposes, this is just fine - we need to infer something about the schema, we do it only once, so it does not matter how long it takes.

Secondly, if we do not have enough time, or have to work in a dynamic environment, we should employ a metaheuristic with a series of improvement heuristics, more specifically Strategy 1. In all our realistic data sets the opti-

Figure 4.15: Chained IHs - Pure *Glpk* vs. Tuned Strategy 1



mum was found almost instantly, and the most complex and bigges artificial data sets took only around 1/4 of the time to finish, compared to `Glpk`.

Furthermore, it is always good to ignore the simple text data nodes, as it will improve the total run time.

5. Future work

A straightforward extension granting the ability to handle more than one input XML file has already been suggested in [BM03]. However, it was not implemented in this work either, so it remains an obvious first choice of future work.

It is possible (and easy) to add more construction and improvement heuristics, as well as more metaheuristics in which the existing IHs are chained. A starting point is in Section B.1.

As it was mentioned in Section 3.4.2, the combination of *`Crossover`*, *`Mutation`* and *`RemoveWorst`* can be seen as a sort of genetic programming. However some modifications would still be necessary to make it a real instance of genetic algorithm metaheuristic.

Likewise it is possible to create an Ant Colony Optimization metaheuristic solving the same problem. It would be interesting to see all these metaheuristics compared to each other in a set of comprehensive experiments.

The approach used in this work was strictly single-threaded, however there are in principle no limitations to extending this to a parallel, multi-threaded environment. For example, creating a pool of initial solutions in *`Glpk`* construction heuristic can be improved by running several instances of GLPK solver in parallel - as GLPK on its own uses only a single thread.

From the point of view of a user - researcher, the current implementation of the experimental framework leaves a lot to be desired. As jInfer already contains support for interchangeable and configurable modules, it is possible to create GUI for experiment and experimental set configuration on the fly.

jInfer as well as the *`IDSetSearch`* module are open source projects, meaning that anyone wishing to build upon this work can do so easily.

Conclusion

From all the integrity constraints in XML we chose the `ID/IDREF/IDREFS` attributes and decided to improve upon the search for them. We discussed the approach from [BM03] and the equivalence of ID set search and maximum weighted independent set. Based on this article we introduced the MIP approach and demonstrated how to find the optimal ID set using external GLPK solver in the environment of jInfer framework.

However, this approach took too long for some inputs, so we introduced a whole range of construction as well as improvement heuristics. We combined these algorithms to create a metaheuristic and performed a number of experiments to understand its behavior. Finally we selected a promising metaheuristic strategy and tuned its parameters to find very good ID sets while maintaining low running times.

To the best of our knowledge, at the time of writing this work is our approach to finding `ID` attributes the best one known.

Bibliography

[Aho96] H. Ahonen. *Generating Grammars for Structured Documents Using Grammatical Inference Methods*. PhD thesis, Department of Computer Science, University of Helsinki, Series of Publications A, Report A-1996-4, 1996.

[BDF$^+$01] Peter Buneman, Susan Davidson, Wenfei Fan, Carmem Hara, and Wang-Chiew Tan. Keys for XML. In *Proceedings of the 10th international conference on World Wide Web*, WWW '01, pages 201–210, New York, NY, USA, 2001. ACM.

[BM03] Denilson Barbosa and Alberto Mendelzon. Finding ID Attributes in XML Documents. In Zohra Bellahsene, Akmal Chaudhri, Erhard Rahm, Michael Rys, and Rainer Unland, editors, *Database and XML Technologies*, volume 2824 of *Lecture Notes in Computer Science*, pages 180–194. Springer Berlin / Heidelberg, 2003. 10.1007/978-3-540-39429-7_12.

[BNV08] Geert Jan Bex, Frank Neven, and Stijn Vansummeren. Schemascope: a System for Inferring and Cleaning XML Schemas. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1259–1262, New York, NY, USA, 2008. ACM.

[BPM$^+$08] Tim Bray, Jean Paoli, Eve Maler, François Yergeau, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C recommendation, W3C, November 2008. `http://www.w3.org/TR/2008/REC-xml-20081126/`.

[Dan98] G.B. Dantzig. *Linear Programming and Extensions*. Landmarks in Physics and Mathematics. Princeton University Press, 1998.

[DC10] G. Di Caro. Heuristics Lecture Notes. 2010.

[DS04] M. Dorigo and T. Stützle. *Ant Colony Optimization*. Bradford Books. MIT Press, 2004.

[Faj10] Stanislav Fajt. Mining XML Integrity Constraints. Master's thesis, Charles University in Prague, 2010.

[FGK09] Fedor V. Fomin, Fabrizio Grandoni, and Dieter Kratsch. A Measure & Conquer Approach for the Analysis of Exact Algorithms. *J. ACM*, 56:25:1–25:32, August 2009.

[GL97] Fred Glover and Manuel Laguna. *Tabu Search*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[glp] GNU Linear Programming Kit. `http://www.gnu.org/s/glpk/`.

[gnu] Gnuplot, an Interactive Plotting Program. `http://www.gnuplot.info/`.

[Gol89] D.E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Artificial Intelligence. Addison-Wesley Pub. Co., 1989.

[gra] Graph Visualization Software. `http://www.graphviz.org/`.

[JR86] J.M and Robson. Algorithms for Maximum Independent Sets. *Journal of Algorithms*, 7(3):425 – 440, 1986.

[Kel] Williams Kelley. Gnuplot Manual. `http://www.gnuplot.info/`.

[Kle11] Michal Klempa. Optimization and Refinement of XML Schema Inference Approaches. Master's thesis, Charles University in Prague, 2011.

[KMS$^+$11a] Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec, and Matej Vitásek. jInfer Architecture. `http://jinfer.sourceforge.net/modules/architecture.pdf`, 2011.

[KMS$^+$11b] Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec, and Matej Vitásek. jInfer BasicIGG Module Description. `http://jinfer.sourceforge.net/modules/basicigg.pdf`, 2011.

[KMS$^+$11c] Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec, and Matej Vitásek. jInfer: Java Framework for XML Schema Inference. `http://jinfer.sourceforge.net`, 2011.

[LD60] A. H. Land and A. G Doig. An Automatic Method of Solving Discrete Programming Problems. *Econometrica*, 28(3):497–520, 1960.

[Mak] Andrew Makhorin. GNU Mathprog. `http://lpsolve.sourceforge.net/5.5/MathProg.htm`.

[McK66] E. H. McKinney. Generalized Birthday Problem. *The American Mathematical Monthly*, 73(4):pp. 385–387, April 1966.

[Nor] Theodore Norvell. A Short Introduction to Regular Expressions and Context-Free Grammars. `http://www.engr.mun.ca/~theo/Courses/fm/pub/context-free.pdf`.

[Vli02] E.V. Vlist. *XML Schema*. O'Reilly Series. O'Reilly, 2002.

[VMP08] Ondřej Vošta, Irena Mlýnková, and Jaroslav Pokorný. Even an Ant Can Create An XSD. In *DASFAA'08: Proceedings of the 13th international conference on Database systems for advanced applications*, pages 35–50, Berlin, Heidelberg, 2008. Springer-Verlag.

[Š11] Michal Švirec. Efficient Detection of XML Integrity Constraints. Master's thesis, Charles University in Prague, 2011.

[Vyh09] Julie Vyhnanovská. Automatic Construction of an XML Schema for a Given Set of XML Documents. Master's thesis, Charles University in Prague, 2009.

List of Figures

List of Algorithms

List of Tables

List of Abbreviations

AM	Attribute Mapping
CH	Construction Heuristic
CSV	Comma Separated Values
GLPK	GNU Linear Programming Kit
IG	Initial Grammar
IG	Initial Grammar
IH	Improvement Heuristic
IS	Independent Set
ISS	ID Set Search
MIP	Mixed Integer Problem

A. jInfer

This appendix describes shortly yet comprehensively **jInfer** - the Java framework for XML schema inference, in which the algorithms described in this work were implemented. Please see project web [KMS$^+$11c] for complete information, documentation and download options.

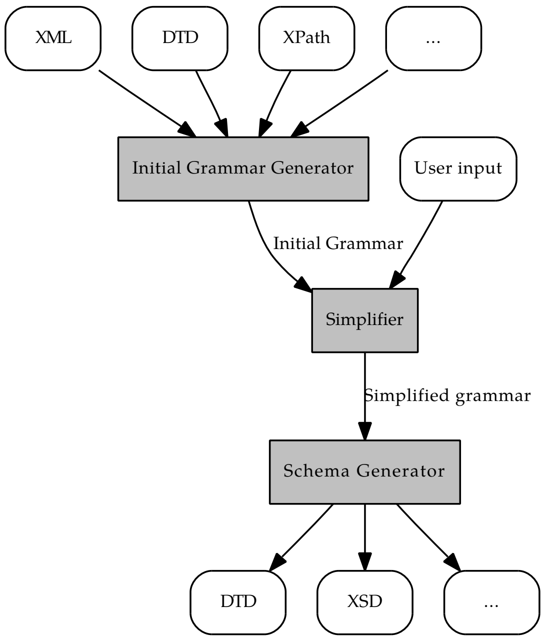
jInfer was developed between 2009 and 2011 at the Charles University in Prague as a Software Project by team consisting of Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec and Matej Vitásek. The main idea was to create a structure in which all aspects of XML schema inference can be easily implemented and evaluated. The goal was achieved: the SW project was successfuly defended when jInfer was inferring DTD and XSD schemas based on XML documents, old DTD and XSD schemas and XPath queries. Since then, Michal Klempa has successfuly defended his own thesis improving on the grammar simplification process (see below), Michal Švirec has extended the framework with capabilities to detect and repair functional dependencies violation (see [Š11]) and defended his thesis as well. This thesis is the third one based on this framework, and Mário Mikula's is on its way, too.

To the best of our knowledge, at the time of writing this thesis is jInfer the only public, open source and actually working solution for XML schema inference-related tasks.

At heart of jInfer inference process is a modular system provided by NetBeans Platform allowing to define services (interfaces), implement them in any number of ways and then let the user choose which implementation to use. Most importantly, the whole process consists of 3 consecutive steps (see A.1), responsibility of 3 different services - interchangeable modules.

The responsibility of the first module, the `Initial Grammar Generator`, is to parse all input files (documents, schemas and queries) and create a so-called *initial grammar* (IG). This is the representation in which will the structure live until it is used to create the final product - the schema. As the name suggests, IG is a grammar - an *extended context-free grammar*, to be more precise (see [Nor]).

Figure A.1: Inference process in jInfer



As such, its left hand side is an element, its right hand side is a regular expression representing its content model. IG is used to create the AM model used in this thesis, too. jInfer contains one such module, the *BasicIGG*, which is described in detail in [KMS+11b].

After leaving the *Initial Grammar Generator*, the IG needs to be made more general, shortened, *simplified*. This is the responsibility of an aptly named module, the *Simplifier*. To get the full idea about how this can be done it would be probably best to read Michal Klempa's thesis [Kle11], which describes this in great detail. Whatever happens, there is simplified grammar on the exit of *Simplifier*, ready to be processed by...

The last module, *Schema Generator* takes the simplified grammar and creates the resulting schema from it. This process is not too interesting, but anyone wishing to find out all about it is invited to read the documentation to the two *Schema Generator*s bundled with jInfer - the BasicDTD and BasicXSD modules.

B. `IDSetSearch`

This appendix will shortly describe the *`IDSetSearch`* jInfer module. As the name suggests, its main purpose is to find ID and IDREF sets and provide attribute statistics in general for grammars originating from any stage of XML schema inference. Virtually every piece of code that was added to jInfer in the course of creating this thesis is contained in this module.

From jInfer's point of view, this module resides in codebase `cz.cuni.mff.ksi.jinfer.iss` and is a service provider for `cz.cuni.mff.ksi.jinfer.base.interfaces.IDSetSearch` interface. Invoking the `showIDSetPanel()` method displays a fully-featured window containing all the relevant attribute statistics as well as possibility to find the ID and IDREF sets for a specified grammar.

Most important packages in *`IDSetSearch`* are the following.

- `objects`, containing classes for attribute mappings and the AM model.
- `heuristics.construction`, containing all the CHs hidden behind the `ConstructionHeuristic` interface, with sub-packages `fidax` containing the whole implementation of FIDAX heuristic (Section 3.4.1). and `glpk` containing the whole interface the external GLPK solver (Section 3.3).
- `heuristics.improvement`, containing all the IH hidden behind the `ImprovementHeuristic` interface.
- `experiments`, containing everything related to experimenting.

`Experiment` is a class representing a single experiment with specified input data (encapsulated in `TestData` interface), settings (encapsulated in `ExperimentParameters`) and a metaheuristic as defined in Section 3.4. Its method `run()` will launch the metaheuristic, first executing the construction heuristic and then running the specified improvement heuristics in a loop until termination criteria defined in an implementation of `TerminationCriterion` are met. The quality of a single ID set is measured by an instance of `Quality-Measurement`. After the experiment finishes, it invokes the `notifyFinished()` method.

However, experiments are almost never run alone. For the purpose of running a whole experimental set there is the `ExperimentSet` interface and its abstract implementation `AbstractExperimentSet`. Its descendants need only to provide a list of `ExperimentParameters` and looping as well as data collection will be handled for them.

B.1 How to Create a New Heuristic

Decide whether it should be a CH or IH and create a class implementing `ConstructionHeuristic` or `ImprovementHeuristic`, respectively. In each case implement all the `get*Name()` methods inherited from `NamedModule` and then the most important `start()` method.

In this method use the provided `Experiment` instance (and `List<IdSet> feasiblePool` in case of IH) to create a pool of feasible solutions and in the end return it by invoking the `finished()` method of the provided `Heuristic-Callback` parameter.

B.2 How to Create a New Experimental Set

Subclass the `AbstractExperimentSet` class, override `getName()` to provide the name of this set and finally override `getExperiments()` to return the list of `ExperimentParameters` that will constitute this sit.

It is possible to override any of the following methods: `notifyStart()`, `notifyFinished()` and `notifyFinishedAll()`. They will be invoked before running the first experiment, after each experiment run and after all experiments finished, respectively. Note that `notifyFinished()` already can output some information regarding the currently finished experiment to a file, but it can be safely overriden without a need to call `super.notifyFinished()`.

C. Experimental Trace

Following is a trace logged from a sample experiment run. It shows all the relevant information related to this instance, any and every piece of information we might be interested in.

To save space, 2-column layout is used. Commentary on the particulars follows right after its end.

```
CPU info
  Intel(R) Core(TM)2 Quad CPU Q9550 @ 2.83GHz
  Cores: 4
  Clock speed: 2983 MHz
Memory info
  Size: 8192 MB
OS info
  Name: Windows 7
  Version: 6.1
  Architecture: amd64
Java info
  Version: 1.6.0_26
  VM: Java HotSpot(TM) 64-Bit Server VM
GLPK info
  GLPSOL: GLPK LP/MIP Solver 4.34

Configuration:
File name: graph.xml (101599 b)
  Graph representation: 82 vertices, 1101 edges
alpha: 1.0, beta: 1.0

Results:
Total time spent: 7754 ms
Final quality: 0.19951219512195123 (10 AMs)
Highest quality: 0.23463414634146343 (12 AMs)
Construction phase:
  Algorithm: Random
    Time taken: 248 ms / Time since start: 248 ms
    Pool size: 10
    Quality: 0.19975609756097568 (11 AMs)
Improvement phase:
  pass #1:
  Algorithm: RandomRemove, ratio = 0.2
    Time taken: 0 ms / Time since start: 841 ms
    Pool size: 10
    Quality: 0.15878048780487808 (9 AMs)
  pass #2:
  Algorithm: Mutation, ratio = 0.1, limit = 1 s
    Time taken: 1512 ms / Time since start: 2710 ms
    Pool size: 11
    Quality: 0.21975609756097558 (11 AMs)

  <... 7 more passes removed ...>

  pass #10:
  Algorithm: Remove Worst
    Time taken: 80 ms / Time since start: 7676 ms
    Pool size: 12
    Quality: 0.19951219512195123 (10 AMs)
Termination reason: Maximum iterations exceeded.

Time,Quality,AMs
248,0.19975609756097568,11
841,0.15878048780487808,9
2710,0.21975609756097558,11
2927,0.1890243902439024,9
4421,0.23463414634146343,12
4703,0.23463414634146343,12
4896,0.1960975609756098,10
5793,0.23463414634146337,12
5972,0.19951219512195123,10
7433,0.19951219512195123,10
7676,0.19951219512195123,10

ID
Element,Attribute,Weight
vertex0,attr,0.024146341463414635
vertex2,attr,0.01975609756097561
vertex33,attr,0.016829268292682928
vertex34,attr,0.02219512195121951
```

```
vertex4,attr,0.022682926829268292
vertex41,attr,0.014878048780487804
vertex7,attr,0.02170731707317073
vertex70,attr,0.018780487804878048
vertex76,attr,0.01780487804878049
vertex8,attr,0.02170731707317073
vertex80,attr,0.01780487804878049
vertex97,attr,0.016341463414634147

IDREF
Element,Attribute
```

The first section deals with system information. Please note that some of these characteristics cannot be easily obtained programmatically and are thus stored in the source code as constants.
To obtain GLPK information, the program parses the first line of standard output produced by running `glpsol -v`. It tries to guess whether it's the Cygwin version by looking at the path to the binary.

The second section states the input file along with its size and graph representation (Section 4.1). Alpha and beta parameters for this instance belong here too.

```
Configuration:
File name: graph.xml (101599 b)
  Graph representation: 82 vertices, 1101 edges
alpha: 1.0, beta: 1.0
```

Results section opens stating the most important information first: how long did the experiment run and what was the highest and final quality (these two are potentially different). Numbers of attribute mappings in the best and final solution respectively are stated as well.

```
Total time spent: 7754 ms
Final quality: 0.19951219512195123 (10 AMs)
Highest quality: 0.23463414634146343 (12 AMs)
```

Construction phase results go next. Among reported information are the full identification of the heuristic (possibly along with its parameters), time taken, size of the pool created and the quality of the incumbent solution (again, with the number of its AMs).

```
Algorithm: Random
```

```
  Time taken: 248 ms / Time since start: 248 ms
  Pool size: 10
  Quality: 0.19975609756097568 (11 AMs)
```

Now for each of the improvement phases there is one section in output log. Information presented here has the same structure as with the construction phase. Please note that the `Pool size` is always measured *after* the improvement run.

```
Algorithm: Mutation, ratio = 0.1, limit = 1 s
  Time taken: 1512 ms / Time since start: 2710 ms
  Pool size: 11
  Quality: 0.21975609756097558 (11 AMs)
```

After the last improvement phase, the reason why the metaheuristic terminated is stated. Possible causes are exceeding the maximum time available, maximum iterations or reaching the known optimum for this file and alpha / beta settings.

To be able to reconstruct the progress of the metaheuristic, the next section contains CSV formatted data for each iteration. Each row contains the time in milliseconds, quality of the incumbent solution and the number of its AMs.

```
Time,Quality,AMs
...
841,0.15878048780487808,9
2710,0.21975609756097558,11
...
```

And finally, it is important to know what is the ID/IDREF set recommended by this experiment run. Thus the log is concluded by a CSV formatted list of element - attribute name pairs to be included in the ID and IDREF set, respectively.

```
Element,Attribute,Weight
vertex0,attr,0.024146341463414635
...
```

Note that in this example trace there were no `IDREF` AMs found. □