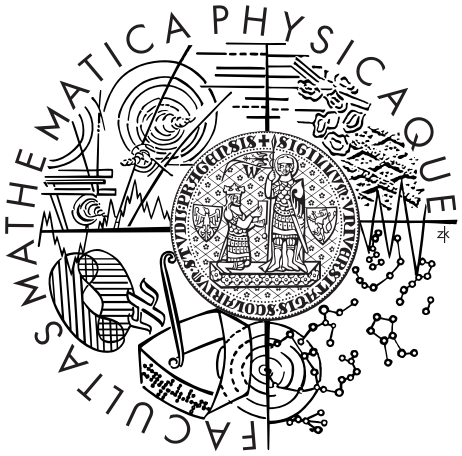
Charles University in Prague

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

[Sample: Here you may thank whoever you wish (the supervisor of the thesis, the consultant, the person who lent the software, literature, etc.)]

TODO thank jInfer team, Mlynkova, anyone helping create this, colleagues at work...

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature

Název práce: Odvozování integritních omezení v XML

Autor: Matej Vitásek

Katedra: Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Irena Mlýnková Ph.D., Katedra softwarového inženýrství

Abstrakt: [abstract of 80-200 words in Czech, but not a copy of the assignment of the master thesis]

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., Department of Software Engineering

Abstract: [abstract of 80-200 words in English, but not a copy of the assignment of the master thesis]

Keywords: XML, ID attributes, inference

Contents

Preface

Along with technologies like JSON, SQL/noSQL databases and bla, XML is one of the leading formats for storing structured data. However, even though languages such as DTD and XML Schema to describe XML structure already exist since a long time, most of the documents use outdated or no schema at all (link Vosta's Even ant can create...). To tackle this problem one may employ reverse-engineering techniques to infer the schema from existing documents, such as those described in A, B, C, jInfer. But the schema is not the only constraint that can be imposed on an XML document: the concept of *keys* and *foreign keys*, well known from the relational database world, applies here as well. One could go even further and try to find even more sophisticated relations in the data, such as *functional dependencies* (link Sviro).

This work will be building upon the achievements of jInfer schema inference framework (TODO link Anti's improvements in schema inference), expanding its possibilities in the field of search for *key*- and *foreign key*-like structures in existing XML documents.

TODO Integrity constraints can be keys (with ID attributes as a "sub-group"), FKs, functional dependencies (quote Sviro), etc. We will focus on the first kind - ID and IDREF attributes.

TODO argument: test data with DTD (and thus possibly ID/IDREF) is more common, we can have better test sets.

0.1 Structure of the thesis

The thesis will be structured as follows.

First, we will introduce a few notions required throughout the work, such as XML tree, ID attributes, ID sets and keys for XML.

Secondly, we will review approaches to ID attribute and XML keys search from previous articles on this topic.

This will lead us to the NP-complete problem of maximal independent set

(IS), where we will inspect approaches for solving it.

We will discuss a closely related Mixed Integer Problem (MIP) and prove their "equality".

Afterwards, we will show how to use an external MIP solver and various heuristics to tackle this problem.

An extension to jInfer for finding ID attributes using MIP solver and a combination of heuristics will be presented and experimentally evaluated in the closing chapters.

0.2 Conventions

TODO list conventions - how we write code related stuff, module names, heu names, etc

TODO perhaps explain the way we will be writing pseudocode?

TODO mention we have a list of abbreviations

1. Definitions

- XML tree - from FIDAX or wherever
- Element, Attribute
- ID, IDREF, IDREFS attribute according to specification
- XSD keys (compare them to ID attributes)
- Key according to Keys for XML [**?**]
- Attribute mapping
- ID set
- Weight: support, coverage

2. Research

According to the article [**?**], ...

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

TODO talk about Fajt - but that's different, that's keys TODO add citation for Fajt

2.1 Independent Set

TODO define IS + MIS rigorously

TODO other approaches to max IS

3. MIP Approach

TODO rigorously define MIP - optimization problem, some of the variables integral

TODO FIDAX is cool, but their heuristic might not be the best. It is a IS problem, let's try a multi-purpose solver. Use GLPK.

3.1 Finding ID sets with GLPK

TODO how GLPK works (branch & bound), how to construct a problem, how output looks, how to interpret it, limit time-> limit quality, ...

-> wow, GLPK works. However, it takes too long to get to optimum (link experiments), so let's try other heuristics.

3.2 Heuristics

TODO what is heuristics (link wise books), what is metaheuristics (we will be using them)

TODO mention things like Taboo search and Genetic Algorithms (we can emulate them with Crossover/Mutation) (we won't be using them)

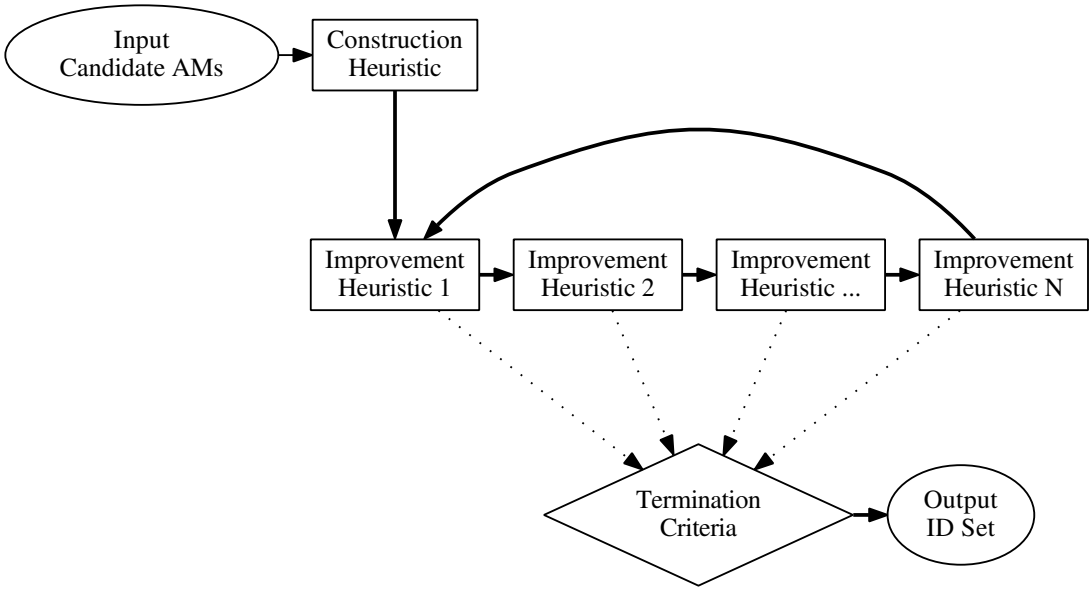
TODO we will be working with heuristics striving to do the following: input is a list of AMs, they have their weight, we try to find a non-conflicting subset which maximizes the weight

TODO we will be using a pool - what is a pool

3.2.1 Constructions Heuristics

TODO construction heuristics are heus that provide us with at least some solution.

Figure 3.1: Metaheuristic schema



`FIDAX`

It should be obvious by now that the algorithm described in [**?**] (we shall call it *FIDAX* from now on) can trivially be used as a construction heuristic that will give us one feasible solution.

The pseudocode of this CH (taken from the original article with trivial modifications without changing the logic) is in Listing 1.

`Random`

One of the most natural heuristics when dealing with the IS 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 2 for its pseudocode.

Algorithm 1 *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 Σ^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'

Algorithm 2 *Random* CH

Input: N required size of pool

Input: C list of candidate AMs

Output: pool of N feasible solutions

```
r ← empty pool
for i = 1 → N do
    // create 1 solution
    s ← empty solution
    while s is a feasible ID set do
        a ← pick at random from C\S
        s ← s ∪ a
    end while
    r ← r ∪ s
end for
return r
```

`Fuzzy`

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

Again, this is a hungry heuristic, the Listing 3 contains its pseudocode.

`Incremental`

This trivial heuristics sorts all candidate AMs by their decreasing weights (TODO link definition of weight) 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 4.

Algorithm 3 *Fuzzy* CH

Input: N required size of pool

Input: C list of candidate AMs

Output: pool of N feasible solutions

```
r ← empty pool
for i = 1 → N do
    // create 1 solution
    s ← empty solution
    C′ ← C
    while C′ not empty do
        a ← pick at weighted random from C′
        if s ∪ a is a feasible ID set then
            s ← s ∪ a
            C′ ← C′\a
        end if
        for each c ∈ C′ do
            if s ∪ c is not a feasible ID set then
                // if c cannot be possibly added anymore
                C′ ← C′\c
            end if
        end for
    end while
    r ← r + s
end for
return r
```

Algorithm 4 *Incremental* CH

Input: C list of candidate AMs

Output: a feasible solution

```
C' ← sort C by decreasing weight
s ← empty solution
for each c ∈ C' do
  if s ∪ c is a feasible ID set then
    s ← 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, order 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 5.

Algorithm 5 *Removal* CH

Input: C list of candidate AMs

Output: a feasible solution

```
C' ← sort C by increasing weight
s ← C'
for each c ∈ s do
  if s is a feasible ID set then
    return s
  end if
  s ← s\c
end for
```

Truncated Branch & Bound

This CH will be called *Glpk* from now on, for it is basically a time-constrained run of GLPK.

TODO if we limit GLPK's runtime, we get this

TODO we shuffle AMs - we get different runs - pool is possible

3.2.2 Improvement Heuristics

TODO what they are, that they need a pool sometimes, their input and output is a pool, ...

TODO mention intensification, diversification

`Identity`

This ultimately trivial improvement heuristics does nothing. It simply returns the feasible pool unchanged. For the sake of completeness, see its listing 6.

Algorithm 6 *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 might be interesting in cooperation with other improvement heuristics that increase the solution pool size, to keep it from growing by pruning inferior solutions.

See listing 7.

`Random Remove`

This is again a rather trivial IH, something which is usually referred to as a *perturbation* function. By removing a random subset of specified size from each

Algorithm 7 *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 $FP \backslash s_{min}$

solution in the pool, it provides variability needed to escape from local optima.

The number of AMs to remove from each solution is specified as ratio (from the interval $(0, 1)$) of the solution size. For example, *Random Remove* with $ratio = 0.1$ 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 8.

Algorithm 8 *Random Remove* IH

Input: FP pool of feasible solutions

Input: $k \in (0, 1)$ ratio 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. Listing 9 captures this process.

Algorithm 9 *Hungry* IH

Input: FP pool of feasible solutions

Input: C list of candidate AMs

Output: pool of feasible solutions

```
for each s ∈ FP do
  // improve a single solution
  C' ← C\s
  C' ← C' sorted by decreasing weight
  for each c ∈ C' do
    if s ∪ c is a feasible ID set then
      s ← s ∪ c
    end if
  end for
end for
return FP
```

`Mutation`

TODO explain how this translates to GLPK input

See listing 10.

`Crossover`

TODO explain how this translates to GLPK input

See listing 11.

`Local Branching`

TODO explain how this translates to GLPK input

See listing 12.

Algorithm 10 *Mutation* IH

Input: FP pool of feasible solutions

Input: k ratio of AMs to fix

Output: pool of feasible solutions

$incumbent \leftarrow$ best solution in FP // *best = highest weight*

$K \leftarrow k * |incumbent|$

fix K random AMs from $incumbent$ in GLPK problem formulation

$improved \leftarrow$ run GLPK

return $FP \cup improved$

Algorithm 11 *Crossover* IH

Input: FP pool of feasible solutions

Input: k ratio of solutions to look for commonalities in

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 $FP \cup improved$

Algorithm 12 *Local Branching* IH

Input: FP pool of feasible solutions

Input: k ratio of total AM count to bound the Hamming distance to

Output: pool of feasible solutions

$K \leftarrow k * |$total AM count$|$

$incumbent \leftarrow$ best solution in FP // *best = highest weight*

add max Hamming distance requirement to GLPK problem formulation

$improved \leftarrow$ run GLPK

return $FP \cup improved$

3.3 IDREF

TODO describe the (rather trivial) algorithm of finding IDREF attributes once we have the ID set

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 some *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 we can try to do more than just let the solver work: by employing various heuristics.

Now we shall attempt to implement these ideas. But before we describe the experiments themselves, we should try to formulate our aim.

First of all, we will try to get understanding of how the whole system and its components behave. We would like to see the changes introduced by modifying some of the key 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. In the end we should have at some intuition about the system.

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

And in the end, we should be able to formulate some kind of general recommendations regarding the problem of finding ID sets.

This chapter will be structured in the following way: first we will discuss the experimental data we used, then the methodology used in conducting the experiments, followed by the actual list of experiments with their full description and results, and in the end we shall draw some final conclusions.

4.1 Experimental Data

The first section will discuss the XML data we will be using to conduct out experiments. We are using XML documents of three categories:

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

A short reasoning for this choice: realistic - 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 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 easy to solve - the list of candidate AMs is usually too short to be hard to be solved to optimality.

To understand these data sets we will discuss their origin and *graph representation*. As mentioned earlier, 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 GraphViz tool, 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.

Table 4.1: List of realistic test data files

Name	Size [kb]	$\|V\|$	$\|E\|$	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

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.

4.1.1 Realistic data

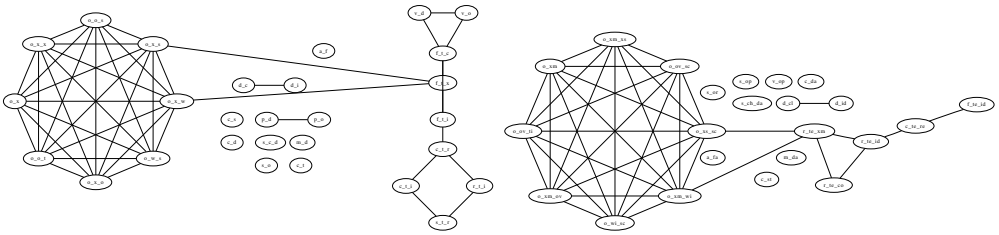
From 3 different sources we collected 6 different data sets, called *OVA1* - *OVA3*, *XMA-c*, *XMA-p* and *XMD*. Their summary is the Table 4.1, their graph representations can be seen in Figure 4.1. Because the legal status of disclosing these data sets is unclear, we will refrain from identifying them beyond these artificial identifiers. Neither will they be included on the DVD distributed with this thesis.

To interpret the data: *OVA** sets have quite interesting and challenging graphs, but 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 that the heuristic will end. We will see the performance of the other components of the whole system, such as loading the data sets into memory representations.

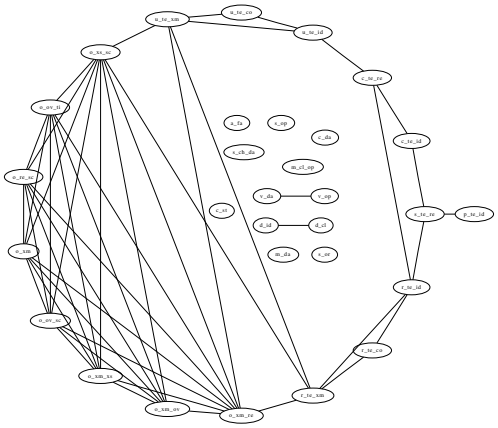
Finally, the *XMD* set is relatively big and at the same time has non-trivial graph representation. In this case we should see a performance more balanced between processing and finding the ID set.

Figure 4.1: Realistic data

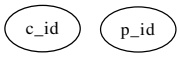


(a) *OVA1*

(b) *OVA2*

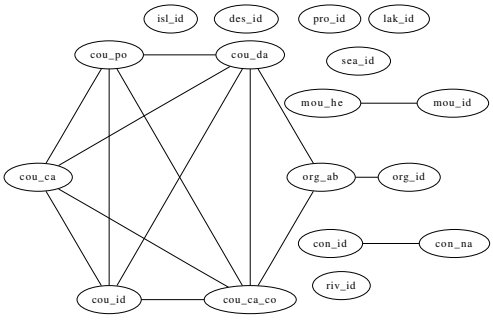


(c) *OVA3*



(d) *XMA-c*

(e) *XMA-p*

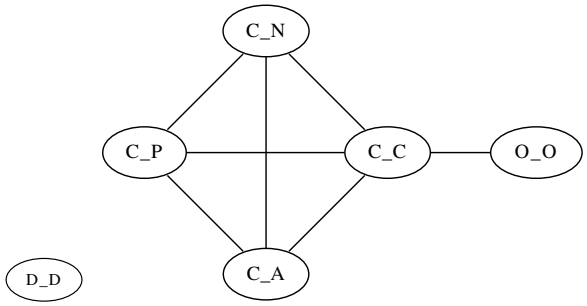


(f) *XMD*

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*

4.1.2 Realistic data with artificial attributes

We used 2 data sets to convert, *MSH* and *NTH*. Unfortunately, the same problem with disclosure as in the previous case applies here. None of these sets had any attributes before the conversion. Their summary is the Table 4.2, their graphs are in Figure 4.2.

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

Again interpretation: in the 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 the case of *NTH* we introduced more attributes, 8 to be precise. 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 build the model, we get the most complex graphs of 31 vertices and 47 edges (see Table 4.1). Our solution is to approach the 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.

Consider 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 around 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 these values will not conflict with any 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.

Algorithm 13 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 → |V| do
    R ← RANDOM
    print <vertexi attr="R">
end for
for i = 1 → |E| do
    v1 ← RANDOM(|V|)
    v2 ← RANDOM(|V|)
    print <vertexv1 attr="i"> <vertexv2 attr="i">
end for
print </graph>
return
```

The pseudocode for this is in Listing 13.

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 ratio of edges that are to all edges that could be in a complete graph with $|V|$ vertices.

We have created 3 sets to use in experiments along with the realistic and converted sets, these are 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

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

artificial alike will be referred to as *official test data (sets)*.

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. From now on, these will be referred to as *sized test data (sets)*. We wanted to keep density the same among these sets, and we picked the $\frac{|E|}{E_{max}}$ density interpretation for this.

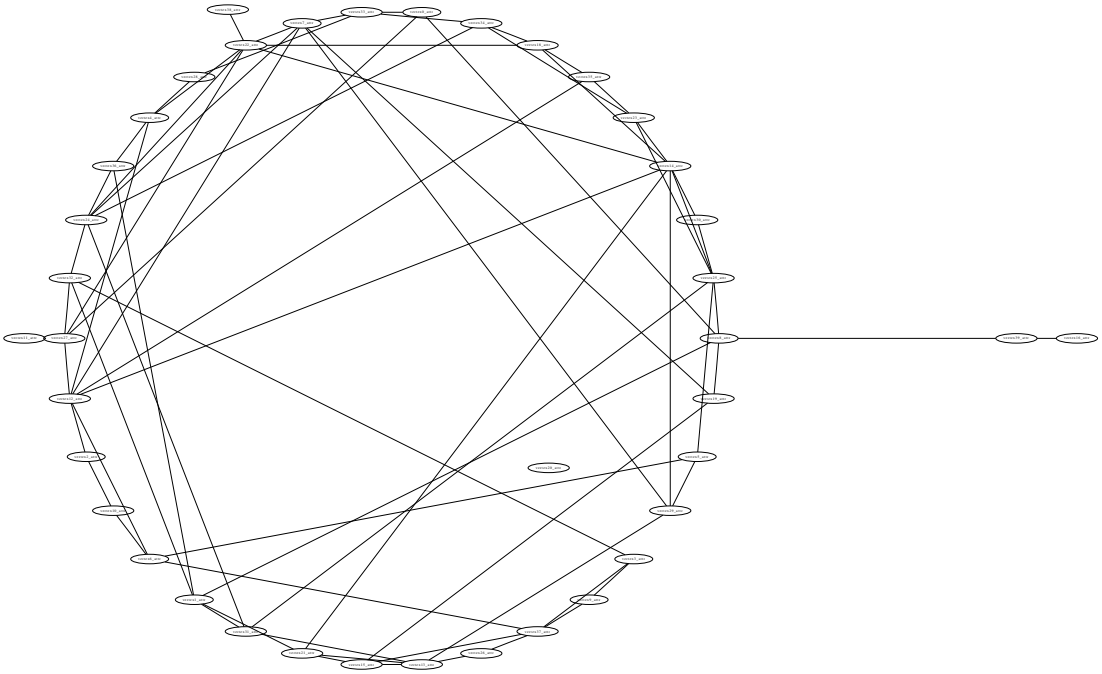
The summary is the Table 4.3 and Table 4.4, these tables contain 2 new columns: values of density in both interpretations we introduced. Some of the graph representations can be seen in Figure 4.3.

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 how 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, this will happen more with higher e.

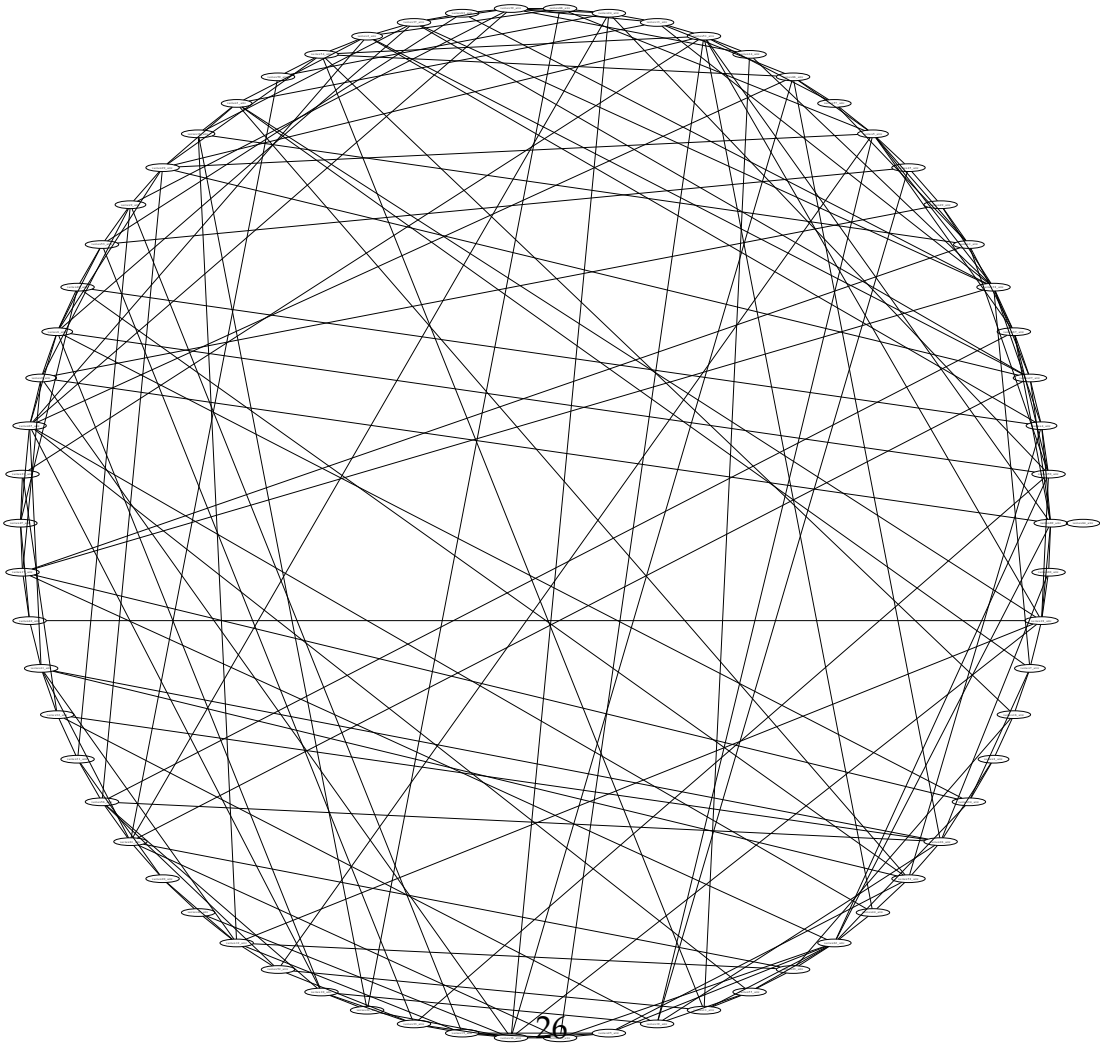
To interpret Tables 4.3 and 4.4: we get 3 sets of different sizes and densities in the first one. The $|V|$ and $|E|$ numbers are orders of magnitude higher than in any realistic (or converted) data set we are using.

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 maximal quality theoretically achievable has to do with the $\frac{|E|}{|V|}$ density, not with the one we fixed. Exploration of this phenomenon is among the possibilities of future work 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]	$\|V\|$	$\|E\|$	$\frac{\|E\|}{\|V\|}$	$\frac{\|E\|}{E_{max}}$	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

On a final note, the artificial data we used in experiments can be found on the DVD enclosed with this work.

4.2 Experimental Setup

As was mentioned before, we will be using 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 define a few notions before moving forward to the description of our experiments.

Definition 4.1 (Experiment parameters)**.** *Experiment parameters* are the following.

- 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.

Definition 4.2 (Experiment(al) configuration)**.** An *experiment instance*, or *configuration*, is one specific setting of all (experiment) parameters.

Definition 4.3 (Experiment(al) set)**.** One or more experiment configurations, regardless whether their parameters differ, constitute an *experimental 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 TODO link.

An input data set is essentialy 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 [**?**] for details) to obtain a list of rules - an *initial grammar* (IG). Please see [**?**] for detailed specification of IG format.

The second step is to convert the grammar into the AM model. This is done by scanning it in a linear fashion 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

This example will present the whole process, from an input XML file to the model and its AMs and types.

Consider the following XML file fragment.

```
<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.

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

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 be using the following machine 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 shield 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 with a life of its own, 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 in hope 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 always be using 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. We then have to divide it by one million to obtain the number of milliseconds passed.

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 [**?**], 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 ([**?**]) 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 again from the manual.

> 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

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 how long does it take 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.

Results are 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 don't 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 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.

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

GLPK Interface Timing

A related problem is how long it takes to create 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.

Results are 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 $|E|$ counts are increasing, the times remain almost the same. This 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 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 time we will let GLPK run until optimum is found. We shall see the dependency between input size and run time needed to achieve the

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

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 version. 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 Listing 14). 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.

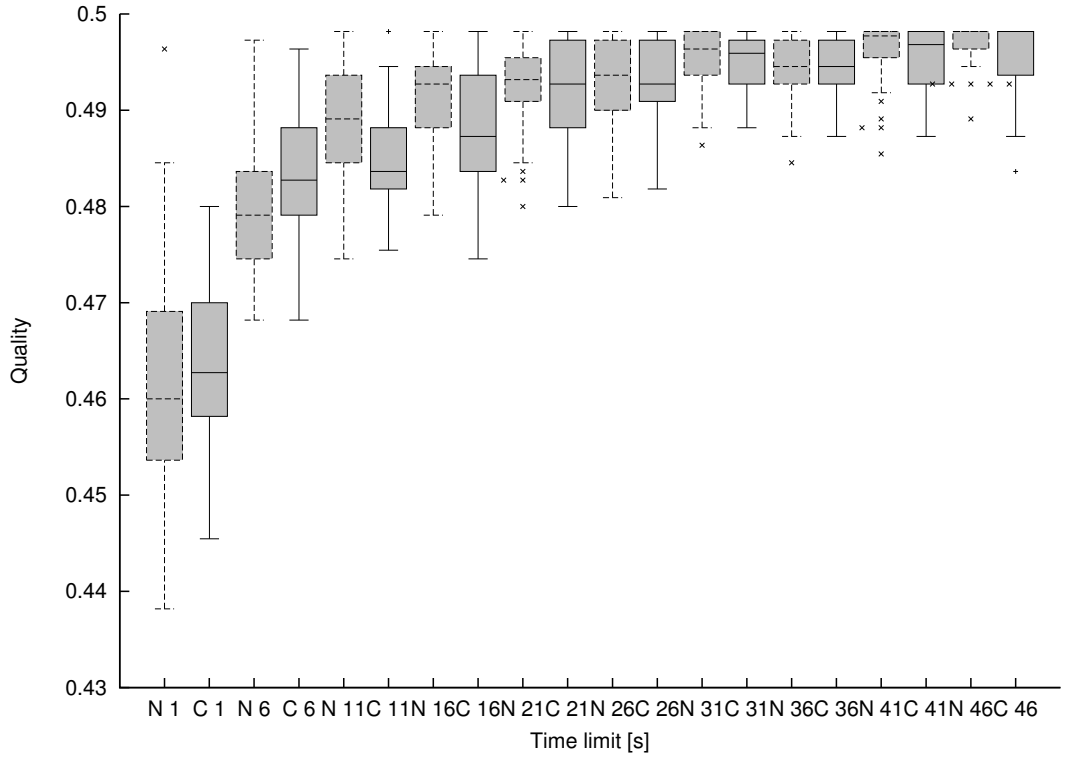
Algorithm 14 GLPK: Native vs. Cygwin Set Generation 1

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

Results are 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.

Figure 4.4: Time vs. Quality



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 Listing 15). There will again be no IH, the only data we will collect are the times of the CH run in each case.

Algorithm 15 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**

    $ES \leftarrow ES \cup \{file, CH = \texttt{Glpk}(no\,limit), IH = \emptyset\}$

  end for

end for

return ES

Results are 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 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 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.

Figure 4.5: Time Until Optimum

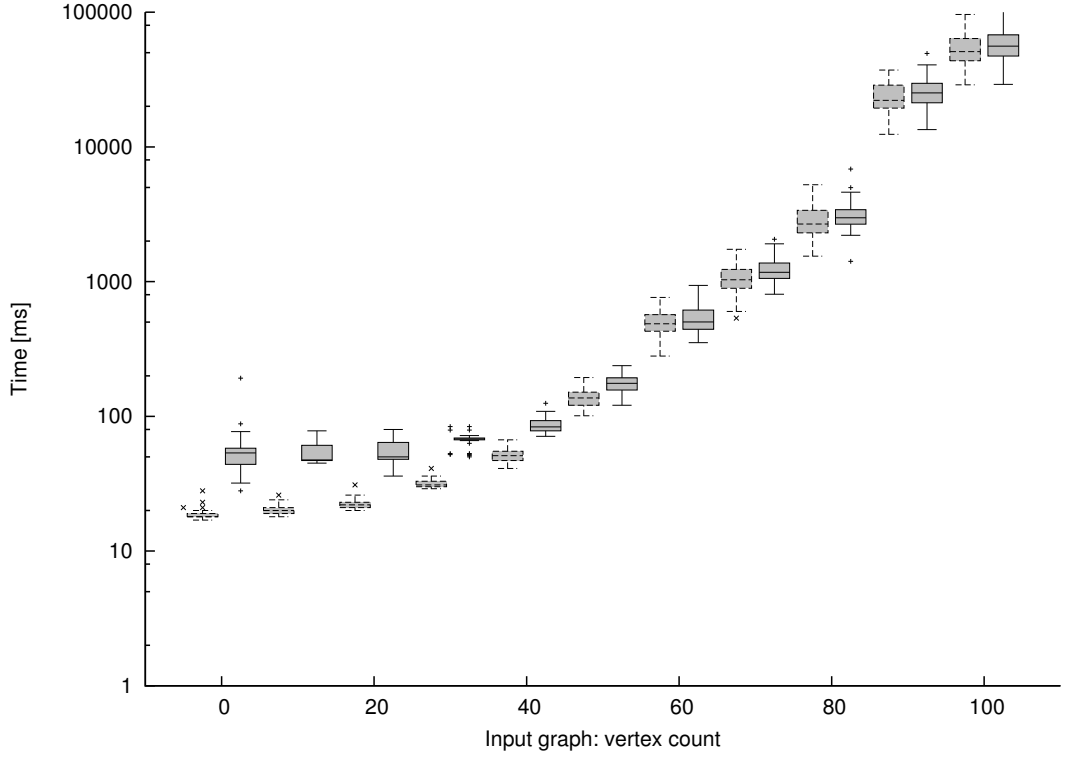
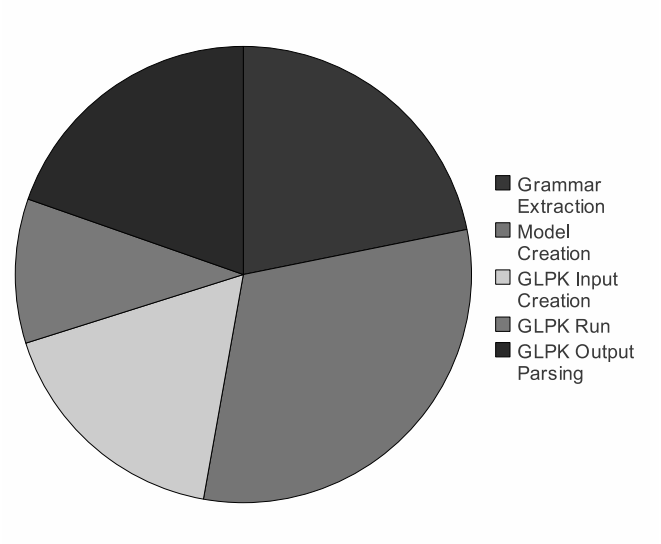


Figure 4.6: Timing Summary



4.3.3 `Random` vs. `Fuzzy` vs. `FIDAX`

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

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 not profit from this. Listing for this can be found in 16.

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

Algorithm 16 *Random* vs. *Fuzzy* vs. *FIDAX* Set Generation

Output: experimental set ES

$ES \leftarrow \emptyset$
for $i = 1 \rightarrow 50$ **do**
  for $file \in$ official test data **do**
    $ES \leftarrow ES \cup \{file, CH = \textit{Random}(pool = 10), IH = \emptyset\} \cup \{file, CH = \textit{Fuzzy}(pool = 10), IH = \emptyset\} \cup \{file, CH = \textit{FIDAX}, IH = \emptyset\}$
  end for
end for
return ES

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

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

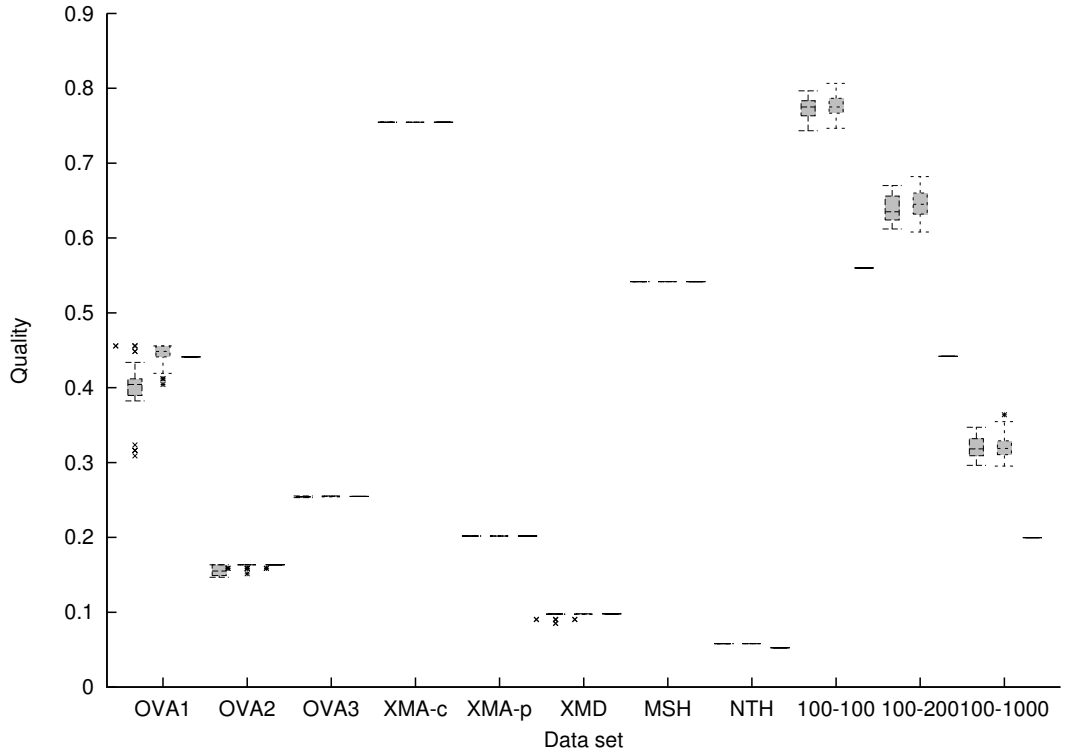
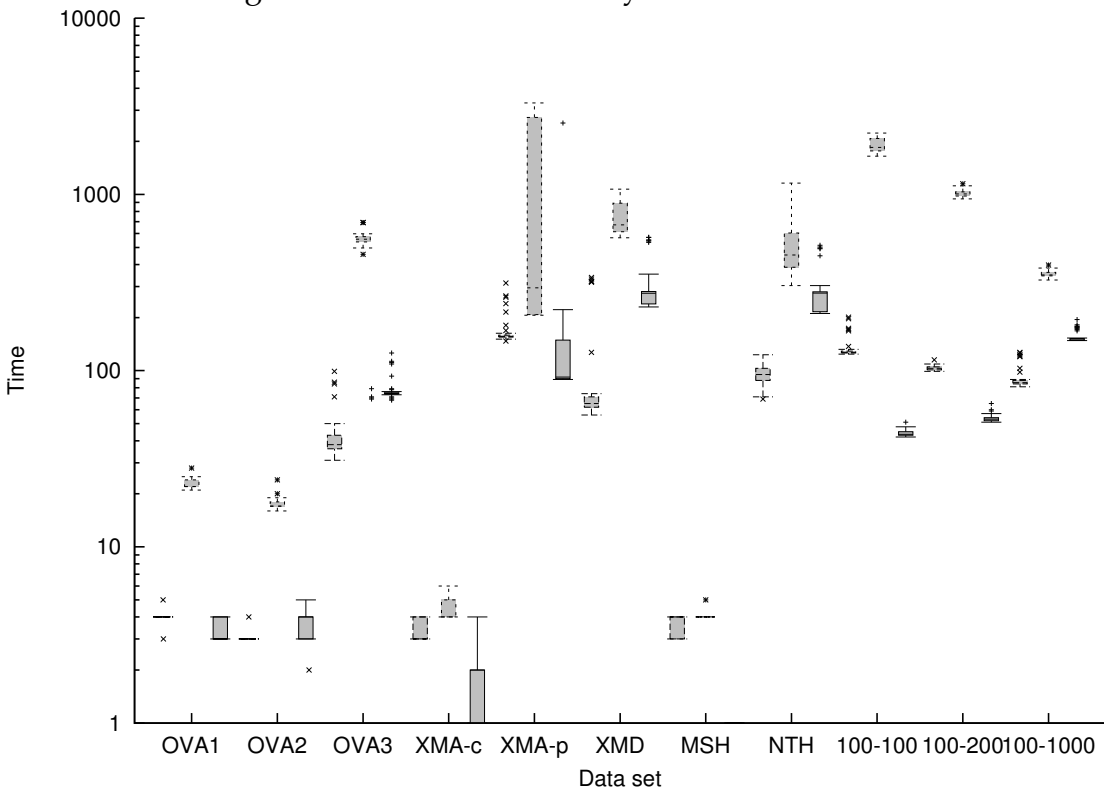


Figure 4.8: Random vs. Fuzzy vs. FIDAX - Time



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's by far the slowest of these CHs. The trivial *Random* is better than *FIDAX* in artificial as well as some real data.

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.

The experimental results are summarized in the 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".

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**

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.

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

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

= 3300 experimental configurations. See the Listing 17 for details. This time we are not interested in run times, only in qualities which we shall gather in a format for GnuPlot.

Algorithm 17 Best Standalone CH Set Generation

Output: experimental set ES

$ES \leftarrow \emptyset$
for $file \in$ official test data **do**
  for $i = 1 \rightarrow 50$ **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.

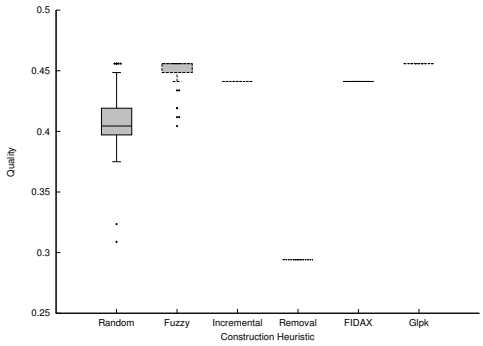
We can see that *Glpk* wins in every single case. We will start from there and try to build upon this result.

4.3.5 Best IH for `Glpk`

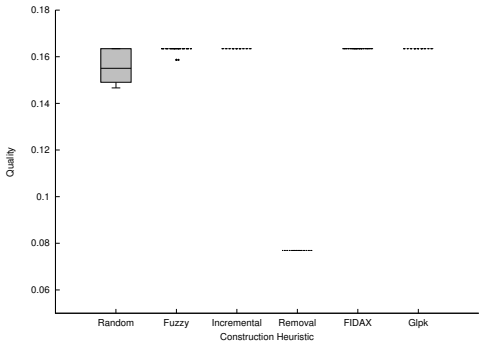
The next logical step will be to try to add one IH after the best CH we have found, *Glpk*. We will investigate all IHs except for *RandomRemove* and *RemoveWorst*, which cannot help us at this time.

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

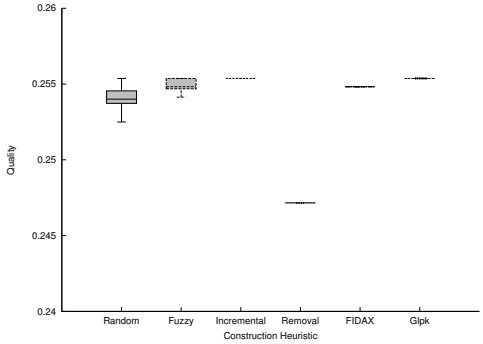
Figure 4.9: Best Standalone CH



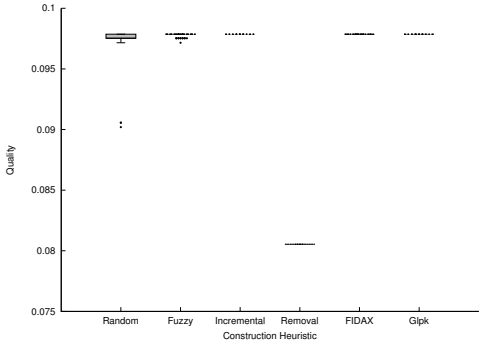
(a) *OVA1*



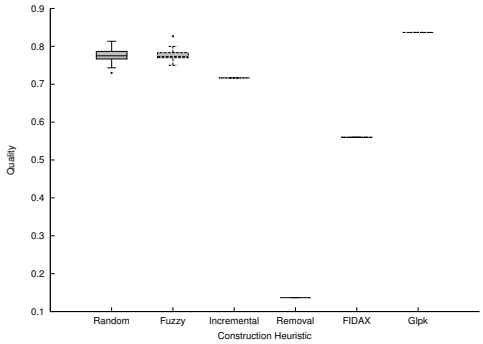
(b) *OVA2*



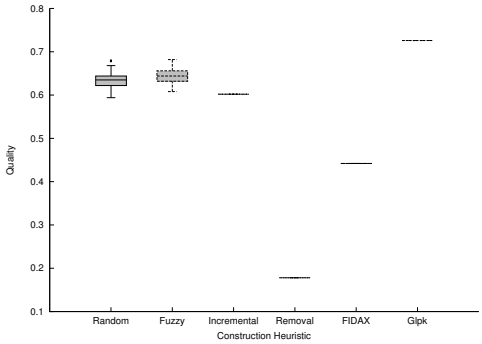
(c) *OVA3*



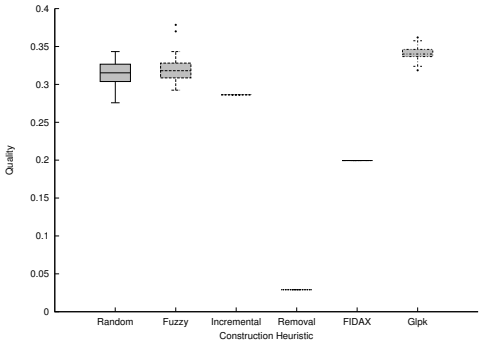
(d) *XMD*



(e) *100-100*



(f) *100-200*



(g) *100-1000*

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.

Algorithm 18 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**

$ES \leftarrow ES \cup \{file, CH = \textit{Glpk}(limit = 1), IH = \textit{Crossover}(ratio = 0.1, limit = 1)\}$

$ES \leftarrow ES \cup \{file, CH = \textit{Glpk}(limit = 1), IH = \textit{Hungry}\}$

$ES \leftarrow ES \cup \{file, CH = \textit{Glpk}(limit = 1), IH = \textit{Local Branching}(ratio = 0.1, limit = 1)\}$

$ES \leftarrow ES \cup \{file, CH = \textit{Glpk}(limit = 1), IH = \textit{Mutation}(ratio = 0.1, limit = 1)\}$

end for

end for

return ES

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

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

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.

`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 Listing 19. We shall collect the eventual quality after running both the CH and the IH in format suited for GnuPlot.

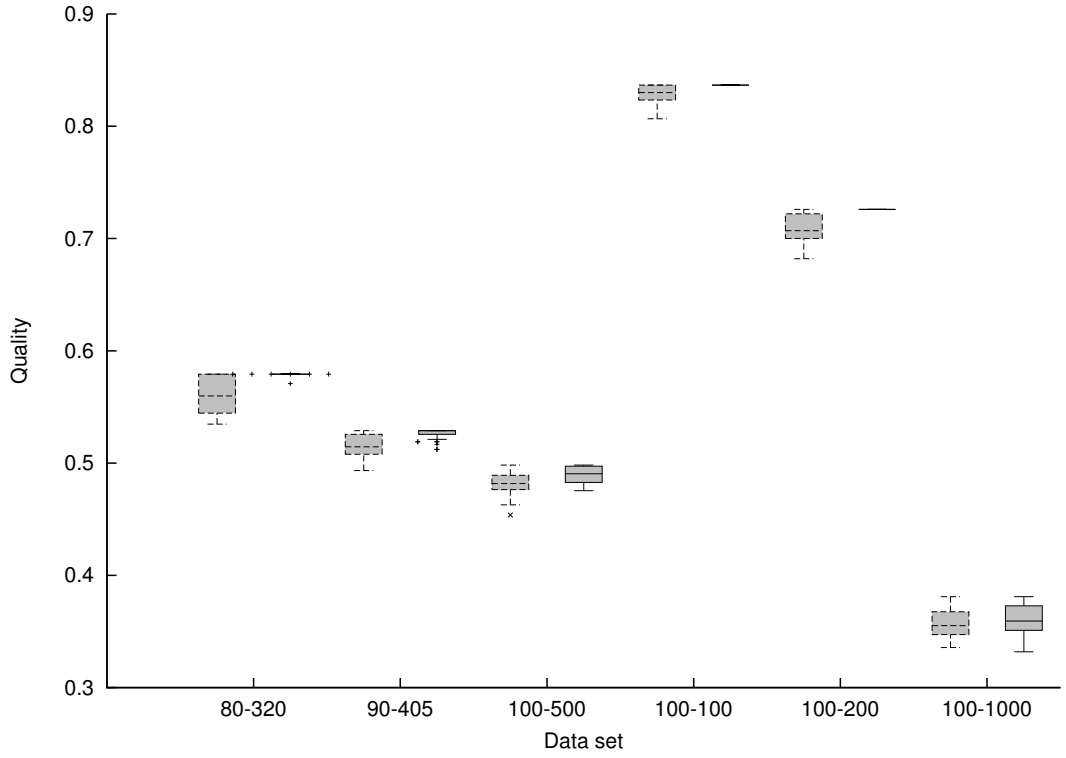
Algorithm 19 *Random* as CH Set Generation

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*$(ratio = 0.1, limit = 1)\}$
    $ES \leftarrow ES \cup \{file, CH =$ *Glpk*$(limit = 1), IH =$ *Mutation*$(ratio = 0.1, limit = 1)\}$
  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

Figure 4.10: CH for *Mutation*



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 some of the parameters. The first ones are the α and β from the definition of our weight function. A short reminder: the weight is defined as TODO copy formula. 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 simplicity's sake we will use 25 combinations of various α and β and normalize them only during evaluation.

It is worthy noting that 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 the way they are generated (attribute values are random numbers) they cannot possibly create different optimal ID sets. The pseudocode capturing this is in Listing 20. 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 20 Various Values of α and β Set Generation

Output: experimental set ES

```
ES ← ∅
for α ∈ {0.1, 0.25, 0.5, 0.75, 1} do
  for β ∈ {0.1, 0.25, 0.5, 0.75, 1} do
    for file ∈ realistic of converted official test data do
      ES ← ES ∪ {file, CH = Glpk(limit = 1, alpha = α, beta = β), IH = ∅}
    end for
  end for
end for
return ES
```

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.

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

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 highlited.

Table 4.11 summarizes the dependency of the ID set found on various values of α, β. We than 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 $\alpha - ratio$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 very 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 outside of the scope of

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

$\alpha \setminus \beta$	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

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		

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.

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 Listing 21. 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.

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.

Algorithm 21 Ignoring Text Data Set Generation

Output: experimental set ES

$ES \leftarrow \emptyset$

for $i \in 1 \rightarrow 50$ **do**

$ES \leftarrow ES \cup \{$*XMA-p*$, CH =$ *Glpk*$(limit = 1), IH = \emptyset\}$

end for

set "ignore text data"

for $i \in 1 \rightarrow 50$ **do**

$ES \leftarrow ES \cup \{$*XMA-p*$, CH =$ *Glpk*$(limit = 1), IH = \emptyset\}$

end for

return ES

Interestingly, the grammar extraction times tend to be shorter in the case when text data is not ignored, although this is inconclusive. However, there is a clear improvement of about 50 % in the 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

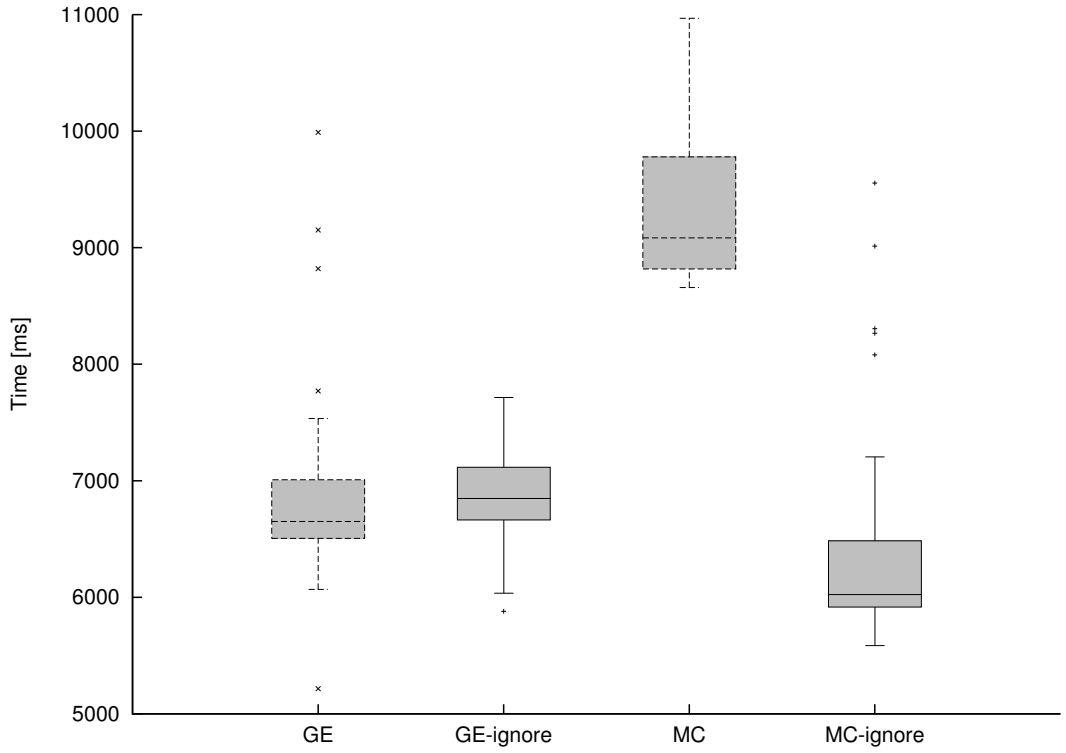
In this section we will describe the most interesting experimental area, that is chaining more than one improvement heuristics and running them in a loop. Unfortunately, the sheer number of possible combinations in which IHs can be ordered (as well as the number of ways to set their parameters) prohibits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

We shall then employ a higher-level heuristic: we will choose 3 scenarios (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 scenarios we will be assessing will be constructed from the following instances of improvement heuristics:

- *RR* is *RandomRemove* with $ratio$ set to 0.1.

Figure 4.11: Ignoring Text Data



- *MUT* is *Mutation* with $ratio$ set to 0.1 and time limit set to 1 second.
- *CX* is *Crossover* with $ratio$ set to 0.1 and time limit set to 1 second.
- *LB* is *LocalBranching* with $ratio$ set to 0.1 and time limit set to 1 second.
- *RW* is *RemoveWorst*.
- *H* is *Hungry*.

The scenarios themselves shall be the following:

- ***Scenario 1***. $RR \rightarrow MUT \rightarrow RR \rightarrow CX \rightarrow RW \rightarrow \ldots$
- ***Scenario 2***. $CX \rightarrow RW \rightarrow MUT \rightarrow \ldots$
- ***Scenario 3***. $CX \rightarrow RR \rightarrow MUT \rightarrow RW \rightarrow LB \rightarrow RW \rightarrow \ldots$

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 scenarios * 11 data sets * 20 iterations = 660 experimental configurations. Their construction is formalized in the Listing 22. The construction heuristic will be *Random* with pool size 10. All the ratios 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.

Data gathered will be traces like the one in Appendix C - 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 scenario (metaheuristic) in the following way: the N$^{\text{th}}$ break in the line (i.e. the N$^{\text{th}}$ data point) is the partial result after the N$^{\text{th}}$ step of the scenario. 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, this corresponds to the fact that there is a finite number of IHs in this scenario (5 of them in Scenario 1) which repeat over time. The obvious similarity between different lines corresponds to the fact that each run is from the same scenario, 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

Algorithm 22 Chaining IHs Set Generation

Output: experimental set ES

$ES \leftarrow \emptyset$

$\texttt{MUT} \leftarrow \texttt{Mutation}\,(ratio = 0.1, limit = 1)$

$\texttt{CX} \leftarrow \texttt{Crossover}\,(ratio = 0.1, limit = 1)$

$\texttt{LB} \leftarrow \texttt{LocalBranching}\,(ratio = 0.1, limit = 1)$

$\texttt{RR} \leftarrow \texttt{RandomRemove}\,(ratio = 0.1)$

$\texttt{H} \leftarrow \texttt{Hungry}$

$\texttt{RW} \leftarrow \texttt{RemoveWorst}$

$IHs \leftarrow \emptyset$

$IHs \leftarrow IHs \cup (\texttt{RR}, \texttt{MUT}, \texttt{RR}, \texttt{CX}, \texttt{RW})$

$IHs \leftarrow IHs \cup (\texttt{CX}, \texttt{RW}, \texttt{MUT})$

$IHs \leftarrow IHs \cup (\texttt{CX}, \texttt{RR}, \texttt{MUT}, \texttt{RW}, \texttt{LB}, \texttt{RW}, \texttt{H})$

for $ih \in IHs$ **do**

  for $file \in$ official test data **do**

    for $i = 1 \rightarrow 20$ **do**

      $ES \leftarrow ES \cup \{file, CH = \texttt{Random}, IH = ih, limit = 10seconds\}$

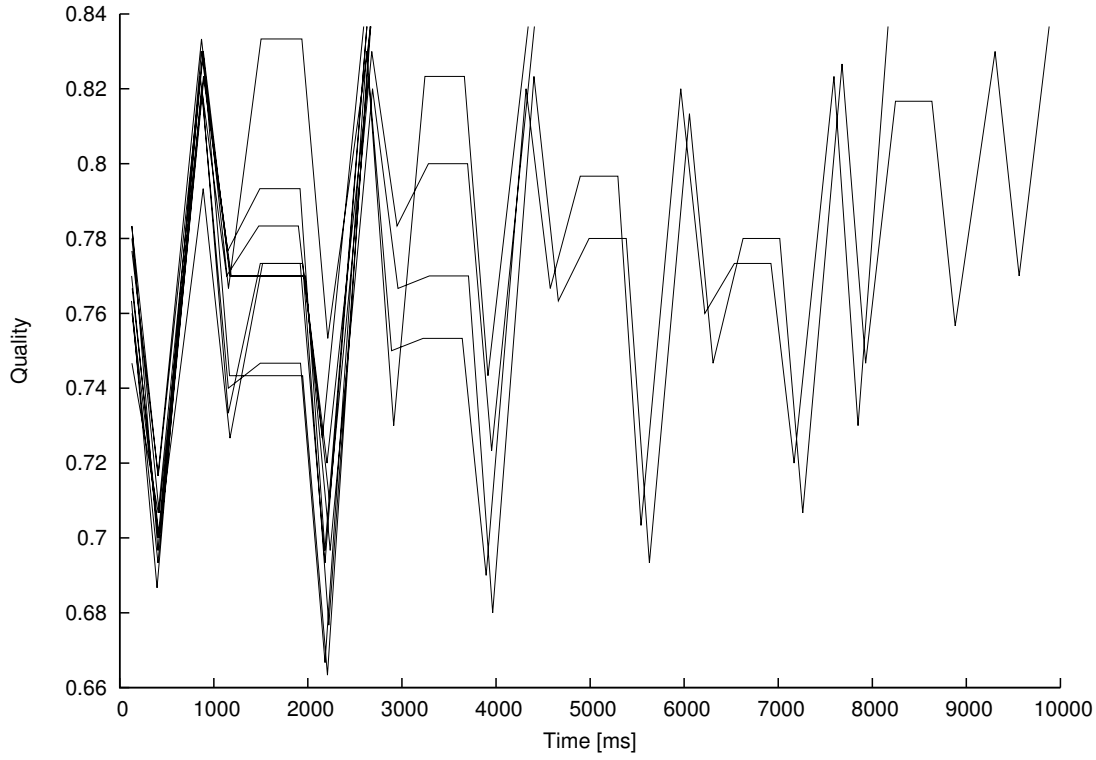
    end for

  end for

end for

return ES

Figure 4.12: Chained IHs - *100-100* Results for Scenario 1



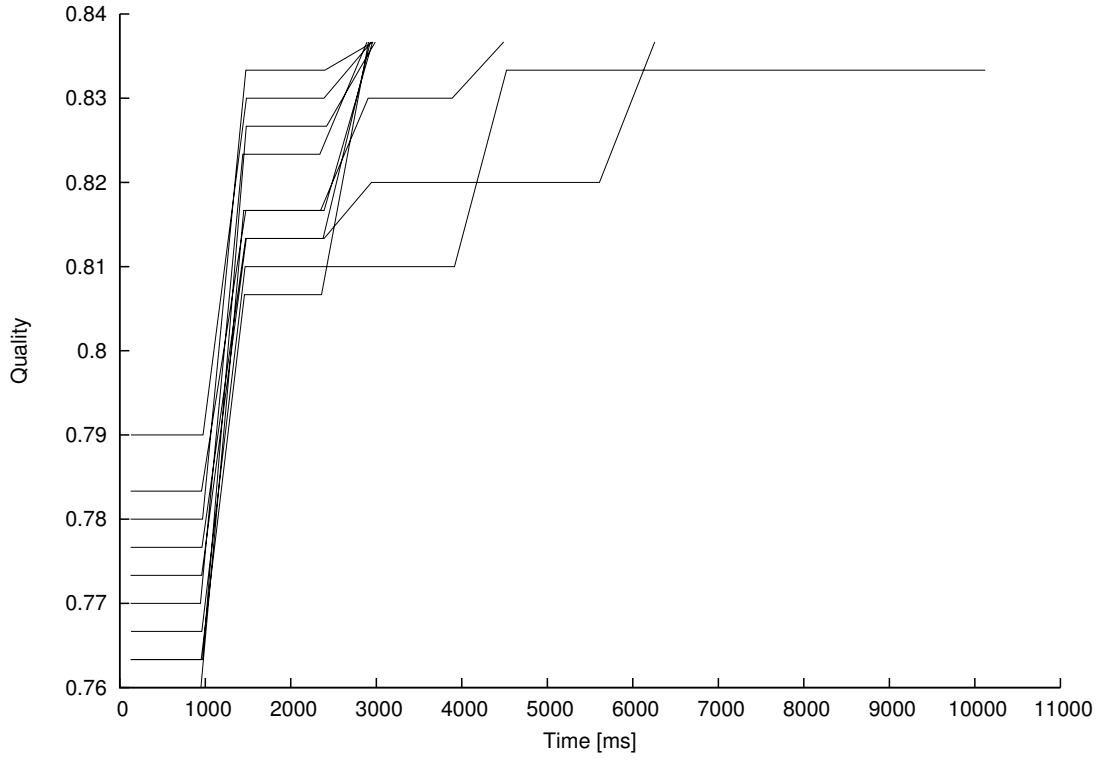
happens there is a rather sharp drop in quality

- 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 Scenario 2 and Scenario 3 respectively working on the same data set, *100-100*. Describing them in detail is unfortunately outside of the scope of this

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



work.

It is now necessary to assess which of the scenarios 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 scenario: namely, how many times of the 20 runs did it find the optimum. Results of this are summarized in Table 4.13.

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

Improving Scenario 1

A short reminder: Scenario 1 consists of *Random* as the CH and the following IHs: $RR \rightarrow MUT \rightarrow RR \rightarrow CX \rightarrow RW \rightarrow \ldots$

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

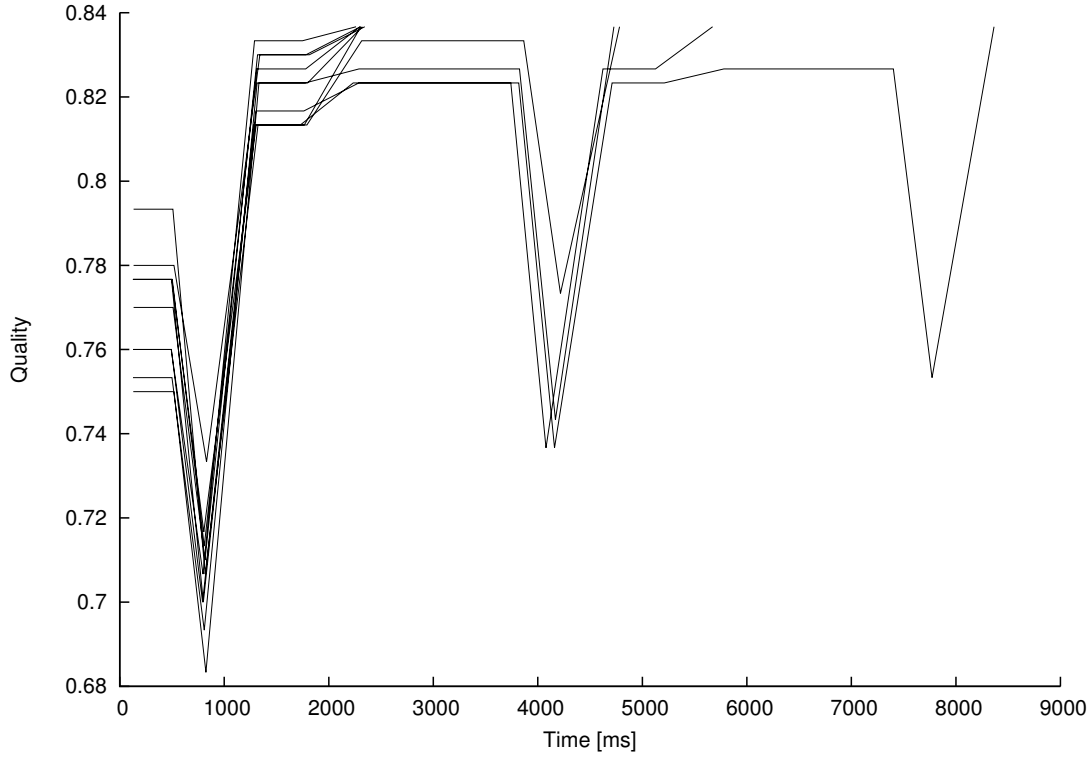


Table 4.13: Performance of Various IH Chains

Dataset	Scenario 1	Scenario 2	Scenario 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**

The parameters we can tune in this scenario are the ratios 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 ratios.

Input data	*100-100*, *100-200*, *100-1000*, *OVA1*, *OVA2*, *OVA3*
Iterations	50
Pool size	10
α, β	1, 1
CH	*Random*
IHs	$RR \rightarrow MUT \rightarrow RR \rightarrow CX \rightarrow RW \rightarrow \ldots$

This experimental set will consist of 45 ratio combinations * 6 data sets * 50 iterations = 13500 experimental configurations. CH will be *Random* with pool size of 10. IHs will be the ones from Scenario 1, with their ratios set to one of the 45 combinations produced in the following way.

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

The process of creating the configurations is captured in Listing 23. We will be gathering the following information for each run: what were the parameters, how long did the run take and whether it found optimum.

After averaging the data we get a large result table, an excerpt from which is in Table 4.14. In the left part are the ratio values, in the right part averaged running times for each data set. Highlited are the shortest times for each set. Only rows containing at least one such shortest time are presented.

Algorithm 23 Chained IHs - Improving Scenario 1 Set Generation

Output: experimental set ES

$ES \leftarrow \emptyset$

$RW \leftarrow$ `RemoveWorst`

for $rrRatio \in \{0, 0.05, 0.1, 0.2, 0.5\}$ **do**

for $mutRatio \in \{0.05, 0.1, 0.2\}$ **do**

for $cxRatio \in \{0.05, 0.1, 0.2\}$ **do**

$RR \leftarrow$ `RandomRemoval` $(ratio = rrRatio)$

$MUT \leftarrow$ `Mutation` $(ratio = mutRatio, limit = 1)$

$CX \leftarrow$ `Crossover` $(ratio = cxRatio, limit = 1)$

for $file \in \{$`100-100`, `100-200`, `100-1000`, `OVA1`, `OVA2`, `OVA3`$\}$ **do**

for $i = 1 \rightarrow 50$ **do**

$ES \leftarrow ES \cup \{file, CH =$ `Random`$, IH = ($`RR`, `MUT`, `RR`, `CX`, `RW`$)\}$

end for

end for

end for

end for

end for

return ES

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

RR	*MUT*	*CX*	*100-100*	*100-1000*	*100-200*	*OVA1*	*OVA2*	*OVA3*
0.2	0.2	0.2	2749.58	**8633.76**	3029.82	110.82	98.68	2413.86
0.5	0.05	0.05	1031.24	10324.46	1371.08	**71.88**	63.86	1282.94
0.5	0.05	0.2	**919.74**	10978.58	1322.02	74.46	62.42	**1217.00**
0.5	0.1	0.2	1002.20	11030.64	**1288.06**	85.22	80.96	1480.90
0.5	0.2	0.1	1674.78	9779.82	2007.42	108.14	**58.56**	1827.46

It is now necessary to pick one ratio combination as the best one. It is $(RR = 0.5, MUT = 0.05, CX = 0.2)$ for the following reason: it is the best one for sets *100-100* and *OVA3* and second best for *100-200*, *OVA1* and *OVA2*. Only the *100-1000* does not profit from these settings.

Now to interpret ratios in the best combination. *RandomRemove* ratio 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 scenario from stalling in local optima. *Mutation* ratio of 0.05 means only around 5% of AMs in the incumbent solution will be fixed for the next GLPK optimization. *Crossover* ratio of 0.2 means that around $1/5^{\text{th}}$ of ID sets in the pool (randomly chosen) will be scanned for common AMs.

All of the ratios in the best combination are at one end of the range we chose from them. As a future work option it is possible to start moving these ratios even more in their preferred way, possibly removing *Mutation* in the process.

TODO compare this to pure GLPK run - for example for *100-200*, we got from TODO seconds to on average TODO seconds (when the 10 second limit is lifted).

TODO this will need 2 boxplots again...

4.4 The "Best" Algorithm

After asking and answering a lot of questions related to the overall system behavior, parameter effects and various heuristic combinations is now the time to 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 doesn't matter how long it takes.

Secondly, if we don't have enough time, or have to work in a dynamic environment, we should employ a metaheuristic with a series of improvement

heuristics, more specifically Scenario 1. In under 10 seconds we will have very good results, often the optimum.

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

5. Future work

A straightforward extension granting the ability to handle more than one input XML file has already been suggested in [**?**]. However, it wasn't implemented in this work either, so it still 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 TODO link appendix - how to write...

As it was mentioned in TODO link, 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 to perform the computation.

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

TODO conclusion: From all integrity constraints in XML we chose the ID attributes and decided to improve the search for them. We discussed the approach chosen in FIDAX. Based on this article we introduced the MIP approach and demonstrated how to find the optimal (w.r.t. weight) ID set.

However, this approach takes too long for some inputs, so we introduced a whole spectrum of construction as well as improvement heuristics. By combining these algorithms and tuning their parameters we were able to find very good solutions while maintaining low running times.

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.

[gnu] Gnuplot, an interactive plotting program. http://www.gnuplot.info/.

[HMU01] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Addison-Wesley, 2001.

[HW07] Yo-Sub Han and Derick Wood. Obtaining shorter regular expressions from finite-state automata. *Theor. Comput. Sci.*, 370(1-3):110–120, 2007.

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

[KMS+11a] Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec, and Matej Vitásek. *jInfer Architecture*. 2011.

[KMS+11b] Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec, and Matej Vitásek. *jInfer Base Module Description*. 2011.

[KMS+11c] Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec, and Matej Vitásek. *jInfer BasicIGG Module Description*. 2011.

[KMS+11d] Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec, and Matej Vitásek. jinfer: Java framework for xml schema inference. `http://jinfer.sourceforge.net`, 2011.

[KMS+11e] Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec, and Matej Vitásek. *jInfer TwoStep simplifier design and implementation*. 2011.

[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`.

[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.

[Vyh] Julie Vyhnanovská. Automatic construction of an xml schema for a given set of xml documents.

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
IH	Improvement Heuristic
IS	Independent Set
MIP	Mixed Integer Problem

A. jInfer

This appendix will try to describe shortly yet comprehensively **jInfer** - the Java framework for XML schema inference. Please see project web [**?**] for complete information, documentation and download options.

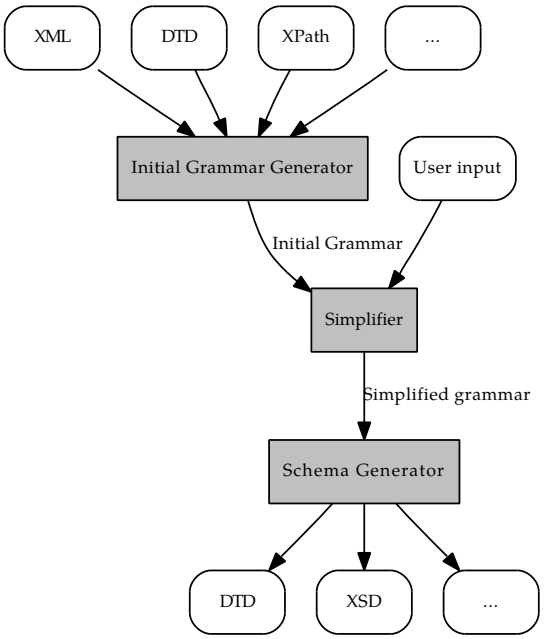
jInfer was developed between 2009 and 2011 at Charles University in Prague as a Software Project by team consisting of Michal Klempa, Mario Mikula, Robert Smetana, Michal Svirec and Matej Vitasek. The main idea was to create a structure in which all aspects of XML schema inference can be easily implemented 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 Svirec has extended the framework with capabilities to detect and repair functional dependencies violation and defended his thesis as well. This thesis is the third based on this framework, and Mario 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, TODO nomenclature). 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 [**?**]). As such, its left hand side is an element, its right hand

Figure A.1: Inference process in jInfer



side is a regular expression representing its content model. (TODO picture?) 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 [**?**].

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 (TODO link), 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 the object representation of attribute mappings and AM model.
- `heuristics.construction`, containing all the CHs hidden behind the `ConstructionHeuristic` interface, with sub-packages `fidax` containing the whole implementation of FIDAX heuristic and `glpk` containing the whole interface to an external GLPK solver.
- `heuristics.improvement`, containing all the IH hidden behind the `ImprovementHeuristic` interface.
- `experiments`, containing everything related to experimenting with these heuristics.

`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 TODO link. 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 `QualityMeasurement`. 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 `HeuristicCallback` 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 optionaly 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 contains logic to 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.
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 - the reason why we do all this! 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.