Università di Pisa
Dipartimento di Informatica

Technical Report: TR-09-18

Searching the Best (Formulation, Solver, Configuration) for Structured Problems

Antonio Frangioni Luis Perez Sanchez

October 15, 2009

address: Largo B. Pontecorvo 3, 56127 Pisa, Italy. tel: +39 050 2212700 fax: +39 050 2212726

Searching the Best (Formulation, Solver, Configuration) for Structured Problems

Antonio Frangioni * Luis Perez Sanchez †

October 15, 2009

Abstract

Complex, hierarchical, multi-scale industrial and natural systems generate increasingly large mathematical models. I-DARE is a structure-aware modeling-reformulating-solving environment based on Declarative Programming, that allows the construction of complex structured models. The main aim of the system is to produce models that can be *automatically and algorithmically* reformulated to search for the "best" formulation, intended as the one for which the most efficient solution approach is available. This requires exploration of a high-dimensional space comprising all (structured) reformulations of a given instance, all available solvers for (each part of) the formulation, and all possible configurations of the relevant algorithmic parameters for each solver. A fundamental pre-requisite for this exploration is the ability to predict the efficiency of a given (set of) algorithm(s), considering their configuration(s), for a given instance; this is, however, a vastly nontrivial task. This article describes how the I-DARE system organizes the information on the instance at hand in order to make the search in the (formulation, solver, configuration) space possible with several different exploration techniques. In particular, we propose a way to combine general machine learning mechanisms and ad-hoc methods, where available, in order to effectively compute the "objective function" of the search, i.e., the prediction of the effectiveness of a point in the space. We also discuss how this mechanism can take upon itself part of the exploration, the one in the sub-space of configurations, thus simplifying the task to the rest of the system by reducing the dimensionality of the search space it has to traverse.

Keywords: *Mathematical Models, Machine Learning, Automatic Algorithm Selection, Reformulation*

1 Introduction

Mathematical Modeling is commonly used for countless many industrial applications: transportation (constrained shortest paths [40], vehicle routing [7], traveling salesman problem [1], etc.), location (plant location [20, 9], location on networks [25], etc.), scheduling [14], complex industrial systems [33], networks [6], bio-informatics [26], chemical engineering [3, 31, 32], medical equipment configuration [30]. Mathematical Modeling is also used in physics [27], statistics [19], data mining [21, 17], mathematics [24, 5], artificial intelligence [22, 10] and many other fields.

However, some of the most striking discoveries of science and mathematics in the last century revealed that just creating a mathematical model does not mean being able to solve it; conversely, "most" mathematical models are very difficult (if at all possible) to solve algorithmically. This has spurned an enormous body of work on *structures* which make the mathematical models tractable.

*Dipartimento di Informatica, Università di Pisa, Polo Universitario della Spezia, Via dei Colli 90, 19121 La Spezia, Italy. E-mail: frangio@di.unipi.it

†Dipartimento di Informatica, Università di Pisa, Largo B. Pontecorvo 3, 56127 Pisa, Italy. E-mail: perez@di.unipi.it

1.1 Modeling and Structures

Each application field has its own concepts about structure. In this article we will consider mathematical models of quite general systems. In particular, while many mathematical models are described by analytical constraints, and therefore our modeling system must easily accommodate them, our underlying concept of structure does not require that this is the case; rather, we only require that *structures* in the model have associated *solvers* capable of tackling them, and that algorithmic rules are available to map inputs and outputs between different structures. Our interest lies in particular to mathematical models where one (or more) objective function(s) is(are) defined, that is, in *mathematical optimization* models.

When a model of a practical industrial/scientific application is built, often times a choice is made *a priori* (and possibly unintentionally) about *which structure* of the model is the most prominent from the algorithmic viewpoint. This is done by choosing which of the several main classes of optimization problems, and the corresponding modeling tools, the problem is molded in. That is, the modeler often times arbitrarily decides first that the model will be a Linear Program (LP) [13], a Mixed Integer Linear Program (MILP) [36], a Constraint Program (CP) [34, 2], and so on. This decision is mostly driven by the previous expertise of the modeler, by the set of tools (bag of tricks) he has available, and by his understanding (or lack thereof) of the intricate relationships between the choices made during the modeling phase and the effectiveness/availability of the corresponding solution procedures. Indeed, it is possible to identify two opposite extreme behaviors in the modeling effort:

- *the algorithmic unconscious user* will write down the model using the mathematical terms he find most appropriate from his knowledge of the nature of the problem, not caring much about the implications that this has on the effectiveness of the solution procedures;
- *the algorithmic conscious user* will try to squeeze the reality into the most algorithmic-friendly class of models he possibly can, not caring much about the fact that the introduced simplifications may make the answers of the model too inaccurate to be useful.

Of course, most proficient modelers will try to carefully balance themselves between these two extremes, by one hand making informed choices about the problem classes that may have a chance to actually result in a solvable model, and on the other hand to carefully check ex-post that the obtained results actually have a meaning. However, it is clear that this process heavily relies on the fact that appropriate knowledge is available to the modeler, which may simply not be true. The continuous improvements of solution methods have created an enormous wealth of results about different problem classes and the conditions under which some algorithms are more or less effective in solving them.

Even when a problem class has (more or less arbitrarily) been selected, general-purpose solvers may exhibit poor performance on many problems since they are mainly focused on representing the general structures, often ignoring underlying forms of sub-structure. Indeed, practical problems most often exhibit *several structures* simultaneously, and it is not *a fortiori* clear which of the structures is computationally more relevant, i.e., offers more help for developing efficient solution techniques for that particular problem. An enormous literature is available about which algorithms are best for solving specific sub-classes of problems; specialized solvers are better when a specific structure is there. It is the user who has to discover the structure, realize that a solver for that structure is available, and write the model in the appropriate "language" of the specialized software. Often, modelers lack both knowledge and resources to perform these complex tasks, and therefore the wealth of available knowledge about specialized algorithms for specific structures lies unused gathering dust in the pages of the scientific journals and/or in prototypical software codes which, despite holding great promises, are too specialized to be known outside a small circle of interested specialists, while actual users cannot solve their problem efficiently enough.

To make matters worse, the applicability of specialized approaches crucially depends on the realization that the structure is there, which in turn depends on having chosen "the right" formulation that reveals it. Arguably, the "right" form of a mathematical model is the one which is appropriate for the best possible algorithm. However, the required structures are typically not "naturally" present in the mathematical models, and must be purposely created by such weird tricks of the trade such as creating apparently unnecessary copies of variables and/or relations, replacing an exact compact nonlinear formulation with a much larger approximate linear one, and many similar others. These ultimately allow the application of specifically tailored sophisticated methods, such as (just to name a few) preconditioning techniques in linear algebra

[41], effective domain reduction techniques in CP [2], specialized row- and column-generation algorithms in MILP [15], appropriate selection and breeding procedures in evolutionary programs [37], and effective learning rules in swarm-intelligence approaches [16]. In other words, one needs a *proper reformulation* of the problem, where structures inside the model are transformed into other "equivalent" structures that are better suited to algorithmic approach.

Finding reformulations is a costly and painstaking process, up to now firmly in the hands of very specialized experts with little to no support from modeling tools. There have been efforts in trying to define automatic reformulation techniques [39, 4, 28, 29], most of them dealing with particular and restricted cases, or without defining proper algorithmic approaches, or defined mainly at an algebraic level.

Once a formulation is chosen, one also needs to determine which solver will be applied and how to configure it. It is well known that the configuration process may be difficult, and it is crucial for the solver's performance. Indeed, solvers that may be either extremely very fast or extremely slow, depending on the configuration, are usually deemed unsuitable for general use. This is only made worse if, as the recent advances dictate, details about the (parallel) hardware architecture need also be comprised into the configuration process.

Because modeling, reformulating and solving a problem is such a complex task, a system capable of streamlining these operations, much like Integrated Development Environments do for computer programming, would clearly be very useful. While a few commercial solutions exist for this, they are typically strongly tied to a specific solver, and *a fortiori* to a specific problem class. This sharply contrasts with the need of experimenting with different formulations, and therefore problem classes. Even more damning is usually the experience of trying to integrate different solution methods, each one adapted to one of the many different structures that a complex problem may simultaneously exhibit. Efforts to ease these problems have been done in the OSI [35] project, but again the decision of whether to use a solver or not and how to configure it, has to be made by the user, which implies a high level of expertise.

The I-DARE system aims of moving forward by performing on behalf of the user three decision processes: formulation selection, solver selection and configuration selection. It is based on the concept of *structure*, intended as whatever characteristics of the problem that can be exploited for algorithmic purposes. This requires a framework where solution techniques are able to communicate between each other in order to solve a problem by means of a general solver interface which permits to plug different existing or new solvers. The user is expected to provide a model picking from the set of available structures; the system will then compute possible reformulation of the models, using available reformulation rules, and figure out—transparently to the user—which formulation allows for the best algorithmic approach, taking into account the issue of configuration, which comprises such delicate choices such as the balancing between solution time and accuracy and the choice of the most cost/effective architecture. Such a framework may favor the creation of competitive solvers concentrated in particular forms of structure (or *combinations* of structures), profoundly reshaping the way in which solution software is developed, tested and deployed.

A fundamental prerequisite for such a system is that all the needed information is available to properly define the search space(s). Even when it is done, the problem of designing the proper algorithms to search for the "best" (formulation, solver, configuration) has to be solved. In this article we will focus on the definition of an unified way of handling search spaces, necessary to make the decisions. Furthermore, we will propose an interface to accommodate all possible decision algorithm. Finally we will present a general machine learning control mechanism that proposes a framework for the implementation of potential decision algorithm based con machine learning techniques.

2 I-DARE – Overview

I-DARE is an extensible environment to model, reformulate and solve structured problems [18]. The fundamental concept behind its design is that of *structure*, intended in its most general sense as "set of characteristics relevant for algorithmic purposes". It is divided in two main parts: the modeling part and the solving part (see Figure 1). The idea is to separate the modeling part as much as possible from the solving mechanism, giving the user the possibility to just focus on the models without worrying about how to solve them. The modeling part is in charge of constructing the models and reformulating them. This is the role of the I-DARE(`lib`) and the I-DARE(`im`) packages [18]; I-DARE(`lib`) defines an extensible library that stores

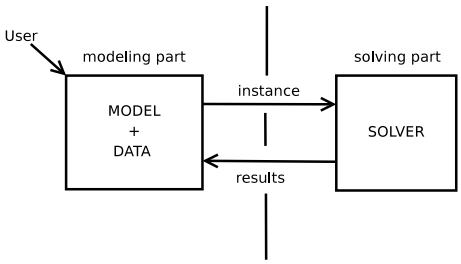

Figure 1: I-DARE, first cut

in a hierarchy all the structures that will be available to construct the models, while I-DARE(`im`) provides all the basic constructs that allow to construct structured models picking from elements of I-DARE(`lib`).

Once the model is done, we need to link it with the problem data;this is the task of the I-DARE(`ei`) package, that wraps the formulation with an *Extended Model* (EM) to provide the means to interact with the data (such as the size of a certain dimension or the value of a constant). I-DARE(`ei`) is built in a plug-in fashion, enabling the extension of the set of data formats that can be manipulated.

Finally, the package I-DARE(`t`) defines a set of atomic reformulation rules [18] that can be used to transform the original EM into equivalent ones (see Figure 2). The reformulation system will is based on the general concept that we can reformulate a structure A into a structure B if and only if there is a mapping from the arguments of A into the arguments of B and another mapping from the answer of B into the answer of A. These mappings must also transform the data linked to the formulation (see again Figure 2). It is important to remark that these mappings need not be purely algebraic; every (sensible, i.e., not too much computationally demanding) algorithmic mapping is allowed. After we do the selection over the possibles

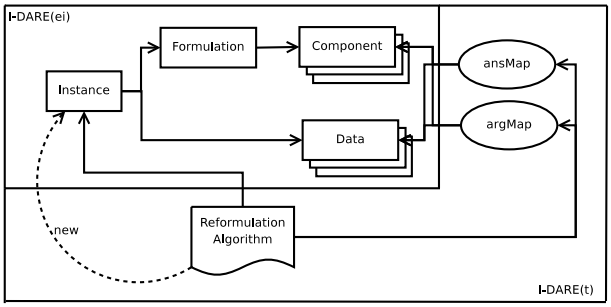

Figure 2: I-DARE(`ei`) and I-DARE(`t`)

reformulations, we can finally generate the enhanced instance to be passed to the solving part.

Package I-DARE(`solve`) makes up the solving part by defining an extensible solver library unified by a common interface. Each solver will register to a structure in I-DARE(`lib`), thus providing a configuration template (see §3.3). I-DARE will generate a *solving tree* for the EM, which defines which solvers will be applied and with which configuration; this is fed to I-DARE(`solve`) which coordinates the solution process, and once a solution is available (if ever) passes it back to the modeling part (see Figure 3).

The functions in I-DARE go well beyond those in "static" algebraic languages; this crucially requires the use of technologies with higher expressive and deductive power, such as Declarative Programming. Indeed, while a few parts (mainly I-DARE(`solve`)) are implemented in in C++ for the sake of efficiency, most of I-DARE is implemented Frame Logic's implementation $\mathcal{F}$LORA-*2* [23, 42].

3 Search Spaces

$\mathcal{F}$LORA-*2* queries provide I-DARE modules with an effective way to obtain structured data about the current instance. This data can be used to characterize and explore the search space, that is composed by three

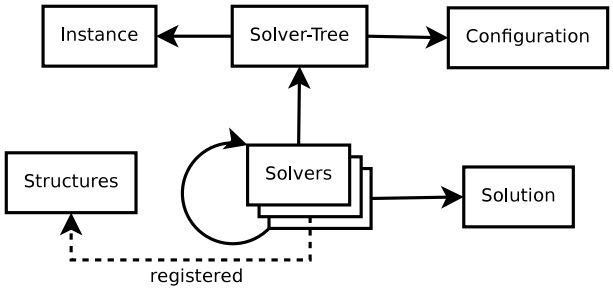

Figure 3: I-DARE(`solve`)

main sub-spaces:

- Formulation + Instance = Extended Model;
- Solvers;
- Configurations.

For each sub-space, an extensible set of predefined queries and methods are available to consult the data. These queries provide any control mechanisms (cf. §4) with the information it needs for effectively guiding the search throughout the whole space.

3.1 Extended Model

A large number of queries are available to retrieve information about variables, constants, dimensions, and components of an Extended Model (EM); how these components are related between each other within a formulation; and, finally, how variables are shared between components. A representative set of queries allows to retrieve:

- `?F[variables -> ?L]` – the list of variables of formulation `?F`,
- `?F[constants -> ?L]` – the list of constants of formulation `?F`,
- `?F[indexes -> ?L]` – the list of indexes of formulation `?F`,
- `?F[components -> ?L]` – the list of components of formulation `?F`,
- `?F[parent(?C) -> ?P]` – the parent of component `?C` in formulation `?F`,
- `?F[ancestors(?C) -> ?L]` – the list of `?C`'s ancestors until root in formulation `?F`,
- `?F[classof(?C) -> ?CC]` – the class of component `?C`,
- `?F[variables(?C) -> ?L]` – the list of variables in component `?C` within formulation `?F`,
- `?F[var_context(?V) -> ?L]` – the list of components where variable `?V` is used, within formulation `?F`,
- `?EM[dim_size(?D) -> ?S]` – the cardinality of dimension `?D` in extended model `?EM`,
- `?EM[const_val(?C) -> ?V` – the value of constant `?C` in extended model `?EM`.

However, using $\mathcal{F}$LORA-*2* query power and the way I-DARE organizes the data, one can define arbitrarily complex queries. For instance, the query

Listing 1: More complex query

```
?L = collectset{?C | ?F[components -> ?LC],
                     member(?C,?LC),
                     ?C : d_LeafProblem_C,
                     ?F[parent(?C) -> ?P],
                     ?P[subs -> ?subs],
                     length(?subs, ?N),
                     ?N > 2}.
```

leaves in `?L` the list of all component in a formulation `?F` that are leaf problems and have as parent a block with more than two children.

The above queries allow to obtain a complete description of any "static" EM; however, the main goal of I-DARE is to allow reformulating the models. This is obtained by applying atomic reformulation rules (ARR) [18] to specific components inside the formulation. Each ARR is an instance of the class

Listing 2: ARR class definition

```
d_ARR [
    A => d_Component_C,
    B => d_Component_C
].
```

where `A` and `B` are the component classes between which the ARR is defined. Several $\mathcal{F}$LORA-*2* queries and methods are available to consult the *ARR database*, such as

- `usable_ARRs(?C, ?L)` – retrieves in `?L` the list of all ARRs such as `?X : ARR.A`,
- `apply_ARR(?EM, ?C, ?ARR) -> ?nEM` – applies an `?ARR` to a component `?C` inside an extended model `?EM`, efficiently creating a new extended model `?nEM` using Transaction Logic Backtrackable Updates present in $\mathcal{F}$LORA-*2*.

Since the queries set is extensible, new queries can be implemented to support all kind of moves in the formulations space, such as *breeding* in population-based heuristics, *basin-hopping* in local search, and many others.

3.2 Solvers

I-DARE(`solve`) defines a general interface for solver plug-ins. Each solver, besides implementing the abstract methods of the class `d_solver`, must register itself to one structure in I-DARE(`lib`) and define its configuration template (see §3.3). I-DARE automatically generates a $\mathcal{F}$LORA-*2* file containing solver and configuration data. Each solver adds to the file the following object

Listing 3: Solver exported to the $\mathcal{F}$LORA-*2* file

```
solverK : d_solver [
    structure -> ?SN,
    confType  -> ?CTN
].
```

where

- `?SN` is the structure class in I-DARE(`lib`) to which the solver is registered,
- `?CTN` is the configuration template (cf. §3.3) associated to this solver.

Each structure class in I-DARE(`lib`) may have more than one solver registered. This database defines the solver's sub-search space, which can be consulted using the defined instances of `d_solver` to retrieve registration and/or configuration information.

3.3 Configuration Templates

As previously mentioned, each solver must define how it must be configured. For this purpose I-DARE defines *Configuration Templates*. A CT is a hierarchical structure defining the relevant algorithmic parameters and the possible range of their values. Hence, CTs can be easily used to describe *single configurations* by simply forcing each parameter to have a single-valued domain.

Two descriptions of CTs are available: the "external" and the "internal" one. The external one is in terms of an XML file that specifies parameters and their domains. CTs currently support four base parameter types: *integer*, *double*, *choice* and *vector*. The integer type defines bounds of the parameter and a default value. For instance,

Listing 4: Example of Integer parameter type

```
1    <INT name = "param1" bounds = "0:1,4,8:10" def_val = "0"/>
```

defines an integer parameter that can take values 0, 1, 4 and from 8 to 10, with default value set to 0. The attribute `bounds` will be a list composed by integer numbers or pairs of the form `l:u`. Each pair `l:u` specifies a lower and upper bound of a subset in the domain of the parameter. The elements of the `bounds` list must be disjoint.

The double type defines also bounds and default value. It also includes a step that specifies the increment that will be used to iterate between the bounds. For instance,

Listing 5: Example of Double parameter type

```
1    <DOUBLE name = "param2" bounds = "0:0.01:1,4:6.5" def_val = "0"/>
```

defines a double parameter that can take values between 0 and 1, with increment (step) set to 0.01 and between 4 and 6.5 also with step 0.01, and default value set to 0. Note that in this case there is a new kind of element in the `bounds` list, `l:s:u`, that represents a lower bound, step and upper bound. When there is an interval without step, in the `bounds` list, the minimum step (present in the list) will be taken; but if there is no step defined at all, a system predefined step will be assumed.

The vector type defines a parameter that may contain a multi-dimensional vector. It specifies the dimension bounds, the type of each vector's element and a default value. For instance,

Listing 6: Example of Integer parameter type

```
1     <VECTOR name="param3" dims = "2|3">
2        <INT name="internal1" bounds="-1:100" def_val = "3"/>
3      <DEF_VAL dims="2|2">
4        <V val = "-1"/>
5        <V val = "2"/>
6        <V val = "34"/>
7        <V val = "4"/>
8      </DEF_VAL>
9    </VECTOR>
```

defines a vector parameter with two dimensions, the first one bounded to 2 and the second one to 3. It also specifies that each vector's element must be of integer type (with the stated bounds and default value). Note that a vector type's default value is defined using a vector tag that sets the dimensions (respecting the bounds) and each element. The elements are represented in a linear form, even if it has more than one dimension; of course, the total amount of elements must be equal to $\Pi_{d \in \text{dimension}}\, d$.

Finally, the choice type defines a nominal parameter that may be used to describe, for instance, a method to be used inside the solver. This type specifies all the nominal values it may take. Each nominal value is a configuration template as well. For instance,

Listing 7: Example of Choice parameter type

```
1    <CHOICE name="param4" def_val = "choice1">
2        <E name = "choice1"/>
3      <E name = "choice2">
```

```
4           <DOUBLE name="subparam1" bounds="0.01:0.005:0.5" def_val = "0.05"/>
5       </E>
6     </CHOICE>
```

defines a domain of two choices. The second choice specifies a sub-parameter of type double; within the tag *E*, a whole CT may appear. This allow us to create *hierarchical configurations*, where the set of parameters in the configuration is not always the same (although of course the total set of parameters which may appear in any configuration pertaining to a CT is fixed). This is useful because solvers may support more than one different algorithm, and some parameters may not have a meaning for some of them. For instance, Linear Program solvers may employ either simplex approaches or interior-point ones; while some algorithmic parameters are typically common to both approaches, there are others that only make sense for one of them.

When a solver is exported to the $\mathcal{F}$LORA-*2* file, it exports also its CT. Therefore, the "internal" representation of the CT in $\mathcal{F}$LORA-*2* is automatically constructed as an instance of the classes: `configuration_T`, `confInt_T`, `confDouble_T`, `confVector_T` and `confChoice_T`,

```
configuration_T.

confNumber_T [
  bounds  => _list,
  def_val => _number
].

confInt_T :: confNumber_T.

confDouble_T :: confNumber_T.
```

```

confVector_T [
  dims => _list,
  type => confNumber_T,
  def_val => [_list, _list]
].

confChoice_T [
  def_val => _string
].
```

For instance, the $\mathcal{F}$LORA-*2* file corresponding to the parameter types exposed in Listings 4, 5, 6 and 7 would look like

```
param1 : confInt_T [
  bounds
-> [(0,1), 4, (8,10)],
  def_val -> 0
].

param2: confDouble_T [
  bounds
-> [(0,0.01,1), (4,6.5)]
  def_val -> 0
].

internal1: confInt_T [
  bounds  -> [(-1,100)],
  def_val -> 3
].

param3: confVector_T [
  type -> internal1,
  dims -> [2 ,3],
  def_val -> [[2,2],[-1,2,34,4]]
].
```

```
choice1 : configuration_T.

subparam1 : confDouble_T [
  bounds
-> [(0.01,0.005,0.5)],
  def_val -> 0.05
].

subparam1 : choice2.
choice2
: configuration_T.

choice1 : param4.
choice2 : param4.
param4  : confChoice_T [
  def_val -> choice1
]

param1 : confTemplate1.
param2 : confTemplate1.
param3 : confTemplate1.
param4 : confTemplate1.
confTemplate1 : configuration_T.
```

Note that when a parameter type is meant to have sub-types it is expressed with the : relation (*has_a* relation). This way one may easily consult the configuration database; for instance, `?X:confTemplate1` will retrieve all the sub-types in `confTemplate1`. More in general, the powerful $\mathcal{F}$LORA-*2* queries make it very easy to deal with CTs, by implementing operations like expanding all possible configurations represented by

a template, constructing the union or the intersection of two CTs, and so on. This is very useful for the different uses of CTs described later on.

4 I-DARE(`control`): Controlling the Search in the (Formulation, Solver, Configuration) Space

All the I-DARE components described so far are conceived for providing the basic blocks for the most delicate and innovative feature of the system: given a structured instance, to automatically select the "best" combination in the space of the possible (re)formulations, solvers and configurations. That is, one must select one particular (re)formulation among all the possible ones obtainable by the atomic reformulation rules, select an appropriate solver—among the possibly several available ones—for each node in the formulation tree, and select an appropriate configuration—among the possibly very many choices—for each of the solvers.

This is clearly a very complex process, for which several different techniques may be used. In principle, of course, it requires the solution of an appropriate "meta" optimization problem in a suitably defined space. However, the problem is made particularly difficult by the fact that even predicting the performances of a given (set of) algorithm(s) and configuration(s) on a given formulation and instance is far from being a trivial task.

In the design of the I-DARE system, we have chosen to provide a rather general and abstract setting for performing the search, so as to allow different search mechanisms to be compared and contrasted. The whole search is controlled by the I-DARE(`control`) module, which may be any control mechanism conforming to the simple interface

```
1 d_control [
2     process(d_InstanceWrapper) -> [d_InstanceWrapper, _term]
3 ].
```

The interface declares a method that, given an extended model, returns the selected "best" reformulation of the model along with the solver tree and the correspondent configuration. Of course the initial and final model may be the same, in which case I-DARE(`control`) "only" selects the best solver and configuration for the given instance. This is already a rather difficult problem in itself, for which little is known in practice; indeed, it is important to remark that *even predicting the running time of a given algorithmic approach on a given data input* is problematic. While there is a huge literature about the theoretical complexity and practical performances of the countless many different algorithms for each of the many possible structures the I-DARE aims at eventually capture, very little is available in terms of methods capable of taking this kind of decision in a general setting. This seems to essentially require the use of *Machine Learning* (ML) techniques (e.g. [8]), which may be the only approach capable of automatically devising suitable approximations of the function which estimates the efficiency (and, possibly, the effectiveness) of an algorithmic approach when applied to the solution of a given instance. As we shall see, the use of ML tools, besides being necessary to evaluate the "objective function" of the search, provides a natural way for actually performing a part of the search, in particular that in the subspace of algorithms and configurations. Remarkably, the use of ML tools for the selection of algorithm parameters have been recently advocated in [12], although in a much more limited context, with promising initial results.

Thus, while the I-DARE system does not specify the exact strategy used by the I-DARE(`control`) module to search the (Formulation, Solver, Configuration) space, it must provide any actual implementation with enough information to effectively drive the process. While I-DARE(`control`) has full access to all the characteristics of the instance (cf. §3.1), the previous discussion highlights the need for further mechanisms that allow an efficient comparison between different points of the space. These are described in the next sections.

4.1 Objective function computation

The fundamental mechanism needed for driving the search is an effective and efficient way for evaluating the quality of a (Formulation, Solvers, Configuration) choice; we will consider this the "objective function" of the search, and denote it by ψ. At first reading, one may imagine that ψ measures the running time required by the solver, with the specified configuration, to solve the corresponding instance; however, different cases

are also possible. For instance, since many problems are "hard", it may well be impossible to solve them to proven optimality in a reasonable amount of time. In this case, the user would typically set a desired target accuracy, and a maximum time limit. Hence, ψ should now account for the running time it takes to the solver obtain a solution with the prescribed optimality, if that can be done within the time limit, and a weighted sum of time limit and final objective function gap otherwise. In this way, the fastest solver capable of attaining the desired accuracy within the limit is selected, if there is any, and the solver providing the most accurate solution at the end of the allotted time is selected otherwise. Alternatively, accuracy of the solution may be treated as a parameter (cf. "Fixed Features" below).

In general, one should not expect that an arithmetic or algorithmic description of ψ be available for all possible formulations, solvers and configurations, although this may indeed happen in some cases. Therefore, we propose the application of ML techniques to approximate such function based on known observations.

4.1.1 Features

As usual in ML, one critical point is the definition of the set of *features* that represent each *data point* in the learning set of the method. It is well-known that the complexity (and practical performances) of several optimization algorithms can be shown to depend in somewhat predictable ways from some well-understood characteristics of the instances: for Linear Programs, for instance, some of the main features are the number of variables and constraints together with the density of the constraint matrix. However, the relevant set of features should be expected to be very different for different problem classes, and even for different algorithms for the same problem class; again in the LP case, degeneracy of the vertices of the polyhedron (that can usually be estimated by some properties of the RHS of the constraints) strongly affects simplex approaches but is next to irrelevant for interior-point ones. Therefore, defining a unique set of features for a problem does not seem reasonable: each solver should be able to specify a different set of features. On the other hand, the responsibility of defining the right set of features cannot be demanded to a general mechanism, so each solver will be required to define them.

Thus, we define a layer over the existing I-DARE solver interface, which is called *Solver Wrapper* (SW), that will provide the list of relevant features to parametrize ψ, i.e., a dictionary `[name=val, ....]`, where `name` is the nominal representing the feature and `val` is the value this feature takes. Of course, a SW must ensure that its feature list always contains the same set of names. All SWs must inherit from the following interface

Listing 8: "Solver Wrapper Interface"

```
1 d_solverWrapper [
2     solver         => d_solver,
3     retrieve(?EM, ?CT) => [_list, CT],
4     [internal]
5 ].
```

Given the current EM, the `retrieve()` method returns the feature list and a list of possible configurations, represented by a CT. The meaning of the method is somewhat different according to the value of the optional property `internal`:

- When `internal` is not present, the evaluation of ψ is demanded to the general mechanism described later on. In this case, the SW "only" has the responsibility to extract from `EM`, that is of course of known type, the features set. The second return value in `retrieve()` is a CT that is intended to describe *all possible configurations* (compatible with the fixed choices, see below) of the solver for this particular instance type.
- When `internal` is present instead, the evaluation of ψ *for the given solver* is done inside the wrapper. In this case, there will be only one (or few) features, consisting in the (estimated) value(s) of ψ (or, maybe, in the description of the function relating running time with accuracy) for the instance `EM`. Actually, the very concept of ψ requires that of a *configuration* attached to the solver, since this choice impacts on the performances. In fact, in this case second return value in `retrieve()` is meant to contain the *single configuration* that produces the estimated value of ψ. It is intended in this case that *the SW will choose the (estimated) best configuration*, if more than one is available.

Actually, since the SW can implement `retrieve()` in any (sensible) way, there may be intermediate scenarios between these two extreme ones. For instance, the SW may internally compute some sophisticated performance figures, out of which predicting the actual running time may be much easier, and/or return a configuration template containing only a subset of the possible configurations, discarding those that are estimated to be unlikely to prove efficient. This general mechanism allows on one side to use a general ML mechanism (described below) for the case where nothing relevant is known about predicting the performances of a solver, and on the other side to exploit specialized techniques when they are available. Note that the SW may well use, internally, a specialized ML approach to select the best configuration, should one be available (cf. e.g. [12]).

4.1.2 The Generic Machine Learning Sub-system

When a SW does not compute its ψ value internally, the Generic Machine Learning Sub-system (GMLS) can be invoked to try to estimate it.

In general, a SW will produce a features list and a CT. The feature list is dependent only on the specific instance `EM`, and not on the configuration, whereas the CT is independent from `EM`. Therefore, this information actually corresponds to *several* values of ψ, one for each configuration in the template (although there may be only one, e.g. when `internal` is present). In ML parlance, the SW (implicitly) produces several *data points*, each one formed by the unique feature set of `EM` and *one* among the different configurations from the template; in other words, the actual features set of the ML is a *pair* (features of the instance, configuration of the algorithm).

This information can be used with any of the several possible ML approaches to try to estimate ψ; clearly, different approaches may turn out to be more effective for different algorithms. In order not to tie-in the I-DARE system to any specific ML technology, I-DARE(`control`) defines a general interface to ML algorithms, described by the following class

Listing 9: "Machine Learning Interface"

```
1 d_machineLearning [
2     evaluate(_list) => _list,
3     => train(_list, _list)
4 ].
```

where

- `train()` trains the ML using a set of data points—that is, (features, configuration) pairs—and a list of known ψ-values (one for each point);
- `evaluate()` computes ψ for the specified data point.

Each concrete class inheriting from `d_machineLearning` will define an actual ML technique (Neural Networks, Support Vector Machine, Decision Tree, ...); the GMLS sub-system will associate each SW with one (possibly the "most appropriate", cf. §4.2.2) concrete ML in charge of computing ψ for the corresponding solver. Note that for nested structures (formulations that contain other structured problems as sub-blocks, cf. [18]), the SW has the possibility to access the SWs of the sub-blocks and therefore it can (but it does not necessarily need to) exploit their computation of the ψ values for the sub-blocks as inputs for its own computation (either with ML techniques, or with any other mean) of the ψ values for the entire block. This allows to nicely decompose the (difficult) task of prediction ψ for a complex algorithm into the (hopefully, easier) tasks of predicting ψ for each component and then predicting how the individual performances affect the global one.

4.1.3 Machine Learning as a Search Mechanism

Clearly, the above ML approach provides one way to automatize the search in the configuration space. Provided that the configurations are "few", one may simply list them all and compute ψ for each; then, the configuration with the best value is retained as the selected one. Provided that the possible solvers for a given structure are not too many either (which looks a reasonable assumption), an effective ML approach to

computing ψ would provide all the tools for performing the search in the (Solver, Configuration) sub-space, leaving "only" the (re-)formulations space to be explored.

In general, however, the set of configurations may be rather large. One might thus devise ML approaches capable of working with "meta" data points, i.e., pairs (features of the instance, configurations *template*). These approaches might for instance still rely on standard ML techniques at their core, but coupled with smart sampling techniques that avoid to compute all possible data points, somewhat in the spirit of *active learning* techniques [38]. More in general, one may devise ML approaches aimed not just at predicting ψ for a given configuration, but rather at predicting *the configuration which produces the best value of ψ within a given CT*. Some very preliminary steps along this line have already been done e.g. in [12].

4.1.4 Fixed Features

The `retrieve()` method of the SW has a second parameter `?CT` (that may conceivably be empty), whose use has not been discussed so far. That is intended to be a *partial* CT, whose use is to constraining the possible configurations to be generated by SW. This allows the caller of a SW to instruct it (in particular, in the case where ψ is computed internally) to avoid considering some configurations that are not feasible, or not "interesting".

There are at least three important cases that may require such a mechanism:

- handling of *accuracy in the solution*, in terms of either constraint satisfaction or of quality of the obtained solution;
- handling of *maximum resource usage* (typically, CPU time) in the solver;
- handling of the *architecture*, i.e., the fact that the same solver may be executed on different parallel hardware (say with a different number of cores, and/or with the presence of specialized hardware such as GPU accelerators).

These aspects may reasonably be considered included in the configuration of a solver. However, depending on the actual form of ψ, they may not be freely chosen by the SW in quest for the smallest ψ value. In fact, accuracy of the overall solution and/or the maximum total allotted running time will typically be set by the final user depending on her needs. In turn, a block of the formulation that has some sub-blocks may want to explore their accuracy/time frontier to seek for the most appropriate setting, e.g. settling to (slightly) less accurate solutions in change for a (consistently) reduced running time; this is, for instance, the setting that is most often chosen for *separation algorithms* in Mixed-Integer Programs when—as it often happens—they require the solution of a hard subproblem. However, in other cases a "master" problem may require solutions of its subproblems with a higher degree of accuracy from the one of the solutions it is expected to provide. For maximum resource usage, it is clear that, in most cases, subproblems of a more complex formulation will have to be solved in much less time than the maximum one allotted for the whole problem. Finally, a SW may want to explore the possibility to allocate its subproblems to different computational nodes to exploit their complementary strengths (see e.g. [11] for one example); on the other hand, some solvers may not be available (or be known to scale very badly) on some architectures, or the target architecture may be severely limited by the user due to price or availability concerns.

All this cases can be handled with the general mechanism of externally constraining the set of available configurations. Note that if the SW is not able to generate at least one configuration that satisfies the constraints imposed by `?CT`, it will *fail* by returning an empty CT, thus signaling that it cannot be used under that set of conditions; basically, this amounts at producing an infinite value of ψ. This way, inner solvers may "constrain" their outer solvers to avoid some specific configuration parameters.

Note that I-DARE does not, in general, enforces that the parameters set by the partial CT in `retrieve()` be meaningful for the SW. For instance, some solvers may only be capable of providing exact solutions to their problem, and therefore the accuracy setting may not be meaningful for them. Also, parameters are always dealt with at the syntactic level, and therefore some discipline will be needed in the construction of the I-DARE library to ensure that at least some main parameters (e.g. accuracy, running time and architecture) be uniformly recognized by all solver. Note that, however, checks can be easily put in place so as to ensure syntactic compatibility between CTs, so that at least warnings can be ensued.

4.2 Training and Meta-Learning

4.2.1 Training

The fundamental assumption under any ML approach is that the machine be fed with an appropriate set of *samples*, i.e., data points with the associated value(s) of the function(s) to be learn. This is known as *training*. The GMLS sub-system will therefore have to execute a learning process before that the ML be ready for actual use in the search. The learning process consists in solving the instances in the *training database* with all available algorithms and all available configurations, thereby producing the data to be fed to the `train` method of `d_machineLearning`. This process can clearly be very time-consuming, and it will have to be (partially) repeated each time either new instances are added to the training database, or solvers are updated/added. Luckily, the learning process can be easily deployed in a parallel environment to take advantage of its high level of inherent parallelism.

4.2.2 Meta Learning

It is obvious that the effectiveness of the prediction of ψ, upon which all the search process ultimately rely, can be very significantly affected by the choice of the concrete ML in charge of computing ψ for any specific SW, together with its possible several *learning parameters* (topology of the Neural Network, parameters of the Support Vector Machine, ...) [8]. Choosing the "most appropriate" ML is therefore, itself, a difficult (yet fundamental) task. Thus, GMLS will also have to implement a *meta learning* process, whereby the results of the same learning phase for a given solver are fed into different ML, and the "best" machine is selected as the one which minimizes some appropriate discrepancy measure between the actual values and the predictions. This can be done with the usual procedures, akin to k-fold validation, whereby the set of available data is (randomly, in several different ways) subdivided into a *training set*, that is actually fed to the ML, and a *testing set* upon which predictions of the ML are computed and contrasted with the (known) true results. Since all ML share the same interface, this process can be automatized and regularly repeated e.g. whenever the testing database significantly changes; again, while very time-consuming the process is also inherently very parallel. Furthermore, the computationally heavy part—actually executing the solvers on the given instances for the selected configurations—need to be done only once; provided that the results are properly stored, they can re-used by all MLs, and over and over again during subsequent meta-learning phases.

4.3 The overall search process

The GMLS sub-system thus defined provides a sound basis for implementing any general search procedure in the (formulation, solver, configuration) space; actually, it may also directly take care of the selection of the latter two components (solver and configuration), leaving to I-DARE(`control`) "only" the task of appropriately traversing the (re)formulations space using the available ARRs to reformulate parts of the whole structured model. Ultimately, I-DARE(`control`) has the responsibility of providing the end user with the ((re)formulation, solver, configuration) that is going to be used to actually solve her problem within the allotted time, accuracy and/or monetary budget constraints. This of course requires implementing a search over the formulation space, for which several different approaches are possible, from complete enumeration to (more likely) heuristic searches such as any variant of local search (with taboo or simulated annealing) or population-based searches such as genetic algorithms. It is also possible to apply ML as a search tool, analogously to what it is done for configurations. Figure 4 shows a diagram that outlines the overall search process in I-DARE(`control`), highlighting the fundamental role of GMLS.

It worth mentioning, that each time a final EM is selected and actually solved, all solution data is sent back to I-DARE so as to be added to the testing database. This way, the testing database is automatically enriched from real problems, strengthening the observation set and therefore allowing the GMLS to perform a better approximation of ψ in the future. This may be the source of a positive feedback loop, whereby good performances of the system attract more users, who provide more data which in turn ultimately leads to even increased performances.

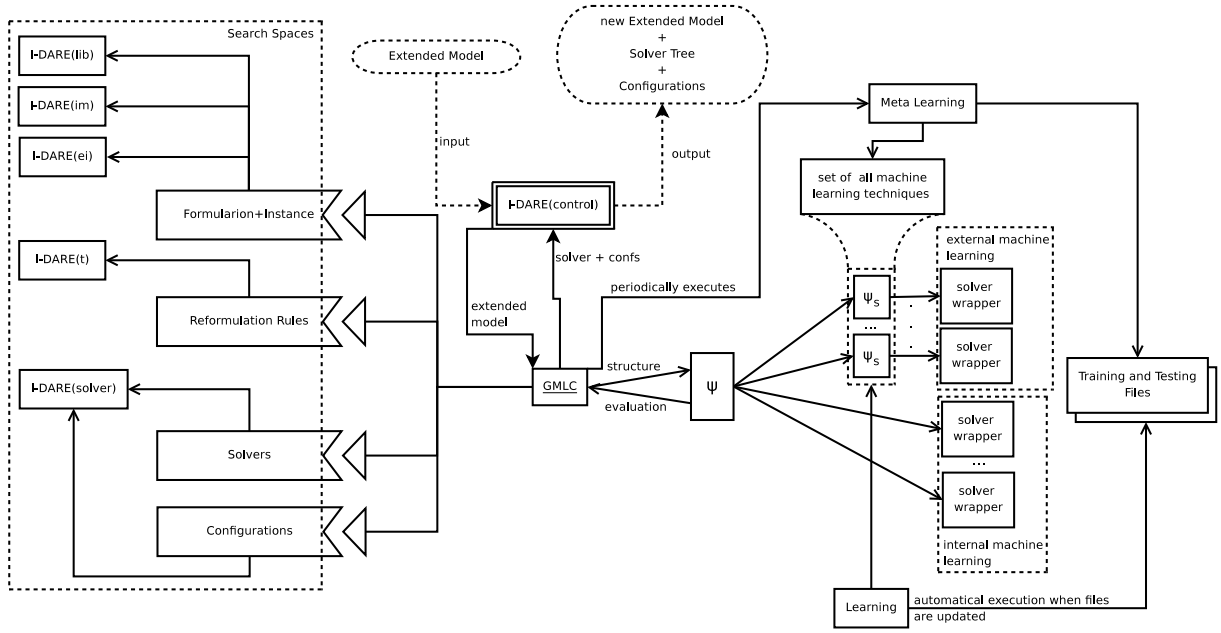

Figure 4: GMLS diagram

5 Conclusions

The development of mathematical models of reality, which often take the form of decision or optimization problems, is arguably the single most important way in which humanity improves its understanding and control over the physical world. As such, it is fundamental for the continuous improvement of science and technology that better and better mathematical models be available to researchers of all fields. However, it is one of the most striking discoveries of science and mathematics in the last century that just creating a mathematical model does not mean being able to solve it; conversely, "most" mathematical models are very difficult (if at all possible) to solve algorithmically. This has spurned an enormous body of work on *structures* (a word with different meanings in different fields) which make the mathematical models (less) algorithmically (un)solvable. The combination of these advances in algorithmic technology and computer hardware has propelled giant strides in the solution of many classes of mathematical models; however, being able to apply the right set of tools to the increasingly sophisticated models of complex, multi-scale hierarchical industrial and physical systems that are constantly being developed is becoming more and more of an issue. It is not unreasonable to be concerned about the growing burden of the task to developing, testing and deploying solution methods for models of ever-increasing complexity in a cost-effective way.

The objective of the I-DARE project is to harness the vast body of knowledge that has been developed over the years about which combinations of (re)formulations and algorithms are best for many classes of optimization problems, and make it available to non experts. This involves the conception of a software system for automatically performing this task on behalf of, and transparently to, the user. This is clearly a very complex task; it requires defining an appropriate general concept of *structured formulation* [18] that be algorithmically treatable with appropriate tools, developing a large library of pre-defined structures which makes it easy for users to model their problem, link each of them with the appropriate solvers, and then be able to effectively search the huge space of possible formulations and solvers (with their many possible algorithmic parameters, comprising such delicate choices such as accuracy, resource limits and architecture) to identify the best option.

This paper is focussed on the last task. We describe the set of architectural choices in the I-DARE system that have been designed to make an effective search possible, while avoiding to tie-in the system to specific search strategies that may not ultimately prove effective enough (such as complete enumeration). In particular, we discuss the fundamental role of the General Machine Learning Sub-system (GMLS), which

allows to integrate general-purpose ML approaches with specialized methods for the (vastly) nontrivial task of computing the "objective function(s)" of the search. This task is "naturally" extended to that of selecting the best algorithmic configuration of the available solvers, thereby providing (whatever actual implementation of) the I-DARE(`control`) sub-system with a powerful tool to streamline the search. This requires a sophisticated ML (meta) process that is continuously running and keeps modifying the assessment of each reformulation with respect to given algorithms, so that it is kept synchronized with latest performance data given by practical problem solution runs. Although use of ML techniques to select algorithmic parameters have very recently been advocated elsewhere, the scale of our proposal is, to the best of our knowledge, unheard of.

The outcome of this sophisticated process may well be a very significant improvement of the efficiency experienced by the "average" (non expert) user in the solution of her models, thereby significantly contributing to the overall scientific and technological progress. Furthermore, it has the possibility to substantially broadening the audience of the very many specialized solvers (and of their underlying theory) that have been developed over the last forty years for problems with specific structure. Indeed, insofar as such a system would greatly facilitate the fair comparison of solution algorithms and effective dissemination of the corresponding results, it might conceivably contribute to organizing, rationalizing and ultimately stimulating the research in solution algorithms for many classes of mathematical models. Actually, the possibility to taking into account monetary concerns during the search could lead to a substantial change in how mathematical software packages are evaluated, possibly forming the basis of a fair and extremely competitive "marketplace" for people supplying problems to be solved and people supplying solution algorithms to be used. Such a marketplace could dramatically improve adoption of best-of-class approaches, possibly rewarding their authors in different ways, and it would allow the developers of very specialized approaches for very specific forms of structure to reach an audience that they would never be able to serve in the current system. This may radically change, for the better, the marketplace for mathematical software, while providing customers with much greater value. Therefore, while very significant theoretical and practical challenges still need to be overcome before this vision can become reality, we believe that the research on automatic reformulation and algorithm-selection techniques is worth to be undertaken.

References

[1] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. The Traveling Salesman: a Computational Study. *Princeton University Press, Princeton*, 2007.

[2] K. R. Apt. *Principles of Constraint Programming.* Cambridge University Press, 2003.

[3] C. Audet, J. Brimberg, P. Hansen, S. Le Digabel, and N. Mladenović. Pooling problem: Alternate formulations and solution methods. *Princeton University Press, Princeton*, 50(6):761–776, 2004.

[4] C. Audet, P. Hansen, B. Jaumard, and G. Savard. Links between linear bilevel and mixed 0-1 programming problems. *Journal of Optimization Theory and Applications*, 93(2):273–300, 1997.

[5] C. Audet, P. Hansen, F. Messine, and S. Perron. The minimum diameter octagon with unit-length sides: Vincze's wife's octagon is suboptimal. *Journal of Combinatorial Theory A*, 108(1):63–75, 2004.

[6] M. Ball, T. Magnanti, C. Monma, and G. Nemhauser. Network Routing, volume 8 of Handbooks in Operations Research and Management Science. *North-Holland, Amsterdam*, 1995.

[7] C. Barnhart and G. Laporte. Transportation, volume 14 of Handbooks in Operations Research and Management Science. *North-Holland, Amsterdam*, 2007.

[8] C.M. Bishop. *Pattern recognition and machine learning.* Springer, New York, 2006.

[9] J. Brimberg, P. Hansen, N. Mladenović, and E. Taillard. Improvement and comparison of heuristics for solving the uncapacitated multisource weber problem. *Operations Research*, 48(3):444–460, 2000.

[10] Gilles Caporossi and Pierre Hansen. Variable neighborhood search for extremal graphs: 1. the autographix system. *Discrete Mathematics*, 212(1-2):29–44, 2000.

[11] P. Cappanera and A. Frangioni. Symmetric and Asymmetric Parallelization of a Cost-Decomposition Algorithm for Multi-Commodity Flow Problems. *INFORMS Journal on Computing*, 15(4):369–384, 2003.

[12] A. Cassioli, D. Di Lorenzo, M. Locatelli, F. Schoen, and M. Sciandrone. Machine Learning for Global Optimization. Technical Report 2360, Optimization Online, 2009.

[13] G. B Dantzig. Linear programming and extensions. *Princeton, NJ: Princeton University Press*, 1963.

[14] T. Davidović, L. Liberti, N. Maculan, and N. Mladenović. Towards the optimal solution of the multiprocessor scheduling problem with communication delays. In *MISTA Proceedings*, 2007.

[15] Guy Desaulniers, Jacques Desrosiers, and Marius M. Solomon, editors. *Column generation*. Springer, 2005.

[16] M. Dorigo. *Optimization, Learning and Natural Algorithms*. PhD thesis, Politecnico di Milano, Italy, 1992.

[17] O. du Merle, P. Hansen, B. Jaumard, and N. Mladenović. An interior point algorithm for minimum sum-of-squares clustering. *SIAM Journal Scientific Computing*, 21(4):1485–1505, 2000.

[18] A. Frangioni and L. Perez Sanchez. Artificial intelligence techniques for automatic reformulation of complex problems: the i-dare project. Technical report, TR-0913, Dipartimento di Informatica, Università di Pisa, 2009.

[19] E. Gourdin, P. Hansen, and B. Jaumard. Finding maximum likelihood estimators for the three-parameter weibull distribution. *Journal of Global Optimization*, 5(4):373–397, 1994.

[20] P. Hansen, J. Brimberg, N. Mladenović, and D. Urosević. Primal-dual variable neighbourhood search for the simple plant location problem. *INFORMS Journal on Computing*, 19(4):552–564, 2007.

[21] P. Hansen and B. Jaumard. Cluster analysis and mathematical programming. *INFORMS Journal on Computing*, 79:191–215, 1997.

[22] B. Jaumard, P. Hansen, and M. Poggi de Aragão. *Column generation methods for probabilistic logic*, pages 313–331. IPCO, University of Waterloo Press, 1990.

[23] Michael Kifer, Georg Lausen, and James Wu. Logical fundations of object-oriented and frame-based languages. Technical report, Department of Computer Science, SUNY at Stony Brook, NY, 1994.

[24] S. Kucherenko, P. Belotti, L. Liberti, and N. Maculan. New formulations for the kissing number problem. *Discrete Applied Mathematics*, 155(14):1837–1841, 2007.

[25] M. Labbé, D. Peeters, and J.-F. Thisse. *Location on networks*. Network Routing, volume 8 of Handbooks in Operations Research and Management Science. North-Holland, Amsterdam, 1995.

[26] C. Lavor, L. Liberti, and N. Maculan. *Molecular distance geometry problem*. Encyclopedia of Optimization, Springer, New York, 2 edition, 2009.

[27] C. Lavor, L. Liberti, N. Maculan, and M.A. Chaer Nascimento. Solving Hartree-Fock systems with global optimization metohds. *Europhysics Letters*, 5(77):50006p1–50006p5, 2007.

[28] L. Liberti. Reformulations in mathematical programming: Definitions and systematics. *RAIRO-RO*, 43(1):55–86, 2009.

[29] L. Liberti, S. Cafieri, and F. Tarissan. Reformulations in mathematical programming: a computational approach. In A. Abraham, A.-E. Hassanien, P. Siarry, and A. Engelbrecht, editors, *Foundations of Computational Intelligence Vol. 3*, number 203 in Studies in Computational Intelligence, pages 153–234. Springer, Berlin, 2009.

[30] L. Liberti, N. Maculan, and Y. Zhang. Optimal configuration of gamma ray machine radiosurgery units: the sphere covering subproblem. *Optimization Letters*, 3:109–121, 2009.

[31] L. Liberti and C.C. Pantelides. An exact reformulation algorithm for large nonconvex NLPs involving bilinear terms. *Journal of Global Optimization*, 36:161–189, 2006.

[32] C. Maranas and C. Floudas. Global optimization in generalized geometric programming. *Computers and Chemical Engineering*, 21(4):351–369, 1997.

[33] F. Marinelli, O. de Weck, D. Krob, and L. Liberti. A general framework for combined module- and scale- based product platform design. Technical report, LIX, Ecole Polytechnique, 2007.

[34] K. Marriott and P. J. Stuckey. *Programming with constraints: an introduction*. MIT Press, 1998.

[35] Ted Ralphs Matthew Saltzman, Lszlo Ladnyi. *The COIN-OR Open Solver Interface: Technology Overview*, May 2004.

[36] George L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley & Sons, New York, 1988.

[37] Kimmo Nieminen and István Maros. Genetic algorithm for finding a good first integer solution for milp. Technical report, Department of Computing, Imperial College, 2003.

[38] B. Settles. Active Learning Literature Survey. Computer Sciences Technical Report 1648, University of Wisconsin–Madison, 2009.

[39] H. Sherali. *Personal communication*. 2007.

[40] H. Sherali, K. Ozbay, and S. Subramanian. The time-dependent shortest pair of disjoint paths problem: Complexity, models and algorithms. *Networks*, (4):259–272, 1998.

[41] Jonathan Richard Shewchuk. An introduction to the conjugate gradient method without the agonizing pain, 1994.

[42] Guizhen Yang, Michael Kifer, Hui Wan, and Chang Zhao. *Flora-2: User's Manual*.