Local Search over Relational Databases

Toni Mancini[1], Pierre Flener[2], and Justin Pearson[2]

[1] Dipartimento di Informatica, Sapienza Università di Roma, Italy
`tmancini@di.uniroma1.it`
[2] Department of Information Technology, Uppsala University, Sweden
`Pierre.Flener,Justin.Pearson@it.uu.se`

Abstract. Solving combinatorial problems is increasingly crucial in business applications, in order to cope with hard problems of practical relevance. However, the approach of exploiting, in production scenarios, current constraint or mathematical programming solvers has severe limitations, which demand new methods: data is usually stored in potentially large relational databases, and maintaining the problem in central memory, as required by current solvers, could be expensive, challenging, or even impossible, due to the size of the data and the possibly unacceptable loss of data integrity. We present a declarative language based on SQL for modelling combinatorial problems, and novel techniques for local search algorithms explicitly designed to work directly on relational databases, also addressing the different cost model of querying data in the new framework. We also discuss and experiment with a solver implementation that, working on top of any relational DBMS, exploits such algorithms in a way transparent to the user, making a step forward to the seamless integration of combinatorial problem solving into business environments.

1 Introduction

Solving combinatorial problems is increasingly crucial in many business scenarios, in order to cope with hard problems of practical relevance, like scheduling, resource and employee allocation, security, financial and enterprise asset management. However, severe limitations may arise when exploiting, in business applications, traditional solvers based on constraint programming (CP), mathematical programming (MP), SAT, or answer set programming (ASP). In fact, data usually resides on centralised information systems, in form of possibly large and distributed relational databases (DB) with complex integrity constraints, serving *multiple* concurrent applications. Although current solvers often have means to load input from external sources via DB management systems (DBMSs), the approach of processing data outside the information system may be problematic from different, orthogonal standpoints.

First, problem instances are potentially large, hence computing an optimal solution to the, usually NP-hard, desired combinatorial problems may require huge computational time, with *any* solver. During this time, data needs to be locked, and this may be unaffordable in dynamic business settings, where the other applications connected to the same data sources need to work as usual. Also, in some scenarios, the size of the portion of the data relevant to the problem may be too large for it to be easily represented in central memory, which is a typical requirement for current solvers. An example is given by applications in network security, where to perform, e.g., intrusion detection or attack impact assessment, relevant combinatorial problems typically state constraints over most of the content of several and potentially very large event logs.

Second, in several scenarios, the structure of both the data and the problem is very articulated and complex: modern DBMSs provide the user with great flexibility, thanks to libraries of functions for handling various data-types (numbers, timestamps, text, large unstructured objects, XML-trees) and support for the definition of custom functions and new types. Hence, designing an encoding of the combinatorial problem in the constructs offered by current constraint solvers needs great engineering efforts and significant preprocessing time, given the potential size of the data and the complexity of its structure (or lack of structure, in some cases). Also, programmers in business scenarios, like DB administrators (DBAs), typically have no previous knowledge of CP/MP/SAT/ASP, and external specialists are usually needed for the problem to be modelled and solved.

All this may result in the overall cost for exploiting such technologies to become uneconomical. Although large companies may decide to invest in tackling their business-critical problems using specialised or ad-hoc technologies, we believe that the issues above have hindered the more widespread adoption of general-purpose off-the-shelf solvers by industry, especially for what concerns small or medium-sized enterprises or the resolution of non-critical problems.

In our previous work [5], in the aim of closing the gap between combinatorial problem modelling and current DB methodologies, we proposed an alternative approach, showing how standard query languages well-known to the average DBA, like relational algebra (RA) [1] and SQL, could be straightforwardly extended by means of *non-determinism* in order to get seamless integration between the two worlds. This led to powerful languages (called NP-ALG and CONSQL) to model combinatorial problems in NP and optimisation versions thereof, in the spirit of existential second-order logic [13] and first-order model-expansion [11]: the use of these languages, as well as the relational structure of the data, are as intuitive to the average DBA as, e.g., the well-known CP language OPL [15] and its data structures are to the average CP/MP programmer.

The contributions of this paper are as follows. We extend our approach presenting a new version of our SQL-based modelling language, and investigating the applicability of *local search* (LS) [9] as a practical means to solve combinatorial problems over relational DBs. In particular, we re-think classical LS techniques, with an explicit focus on the relational query cost model and on the features already present in modern DBMSs to manage efficiently relational data, potentially large and stored in external memory. To this end, we show how the efficient join optimisation techniques of current DBMSs allow us to perform a *joint* (i.e., collective) exploration of the neighbourhood that largely improves the time needed by evaluating neighbours one by one, as typically done in LS solvers. Although joint neighbourhood exploration is not a completely new idea in LS, see, e.g., work about search over very large-scale neighbourhoods [3], rather than relying on structural properties of the neighbourhood, we exploit the different query cost model and primitives. Further, given that the size of the neighbourhood may become impractical in presence of large instances stored in external memory, we propose, at each iteration, to limit search to a reasonably small set of candidate neighbours, by characterising and eliminating many non-improving moves. Such a dynamic neighbourhood generation is obtained by exploiting the *reasons* why problem constraints are violated, rather than just counting these reasons.

The result is on one hand a declarative modelling framework where modelling a combinatorial problem is reduced to the definition of a *second-order view* of the existing DB (in a way fully compliant with current DB design methodologies), and on the other hand a portfolio of solving techniques based on LS, relying entirely on the DBMS data representation and query mechanisms. This allows the approach to fit the traditional multi-layer data representation paradigm of DBMSs (which distinguishes among internal, logic, and external layers), and to scale transparently toward large data sets in external and potentially distributed storage devices, typical of many business applications (since data consistency and memory management issues could be totally delegated to the DBMS), while *potentially preserving the efficiency of classical solvers*, in case the data is small enough to be entirely cached in main memory and custom DBMS RAM storage engines optimised for the tasks required by problem solving are implemented.

The structure of the paper is as follows. After recalling NP-Alg (Section 2), in Section 3 we propose a much simplified version of ConSQL having just four new keywords w.r.t. standard sql. In Section 4 we propose two novel methods to boost scalability of LS on NP-Alg/ConSQL specifications in the relational query cost model: *joint incremental neighbourhood exploration* and *violation-directed neighbourhood design*. In Section 5 we briefly describe a new implementation of ConSQL based on these ideas and on a portfolio of LS algorithms, with full support to the user for declaratively choosing the search strategy to follow, and experimentally evaluate it in terms of performance and scalability. Finally, Section 6 discusses related work and draws conclusions.

2 The Language NP-Alg

A relational schema $\mathbf{R}$ is a finite set of relational symbols of arbitrary arities. A finite DB $\mathcal{D}$ over schema $\mathbf{R}$ is a set of finite extensions (i.e., finite sets of tuples) for relations in $\mathbf{R}$. It is convenient to assume that the set of constants occurring in $\mathcal{D}$ is also stored in a unary relation $DOM_{\mathcal{D}}$.

An NP-Alg [5] expression over schema $\mathbf{R}$ is a pair $\langle \mathbf{S}, \mathit{fail} \rangle$, where $\mathbf{S} = \{S_1, \ldots, S_n\}$ is a set of new relations (called *guessed* relations) of arbitrary finite arities ($\mathbf{R} \cap \mathbf{S} = \emptyset$) and *fail* is an expression of plain RA on the new DB vocabulary $\mathbf{R} \cup \mathbf{S}$. Evaluating $\langle \mathbf{S}, \mathit{fail} \rangle$ on finite DB $\mathcal{D}$ over $\mathbf{R}$ amounts to non-deterministically populating extensions of relations in $\mathbf{S}$ with constants in $DOM_{\mathcal{D}}$ in such a way that expression *fail* evaluates to the empty relation $\emptyset$. The *answer* to the decision problem is "yes" iff such an extension $\overline{\mathbf{S}}$ for $\mathbf{S}$ exists. In this case, $\overline{\mathbf{S}}$ encodes a *solution* to the problem instance. Intuitively, *fail* encodes *constraints* that extensions of $\mathbf{S}$ need to satisfy. These are represented as a set of tuples called *violation set*, hence the goal is to guess an extension $\overline{\mathbf{S}}$ that leads to an empty violation set, i.e., that makes $\mathit{fail} = \emptyset$. NP-Alg can express all (and only) the queries that specify decision problems belonging to NP. We observe that a dual approach with constraints defined with universal quantification (as is typical in, e.g., CP) can however be followed (and suitable syntactic sugar be added by exploiting De Morgan's laws), although this would let the language deviate from the typical DB paradigm (where queries are existentially quantified formulas). This alternative modelling paradigm (intuitively, we encode as $\nexists \mathbf{x}.\neg c(\mathbf{x})$ a constraint that a CP practitioner would write as $\forall \mathbf{x}.c(\mathbf{x})$) does not introduce any blow-up in the size of the problem model, since a constraint $\neg c(\mathbf{x})$ is not represented extensionally, but intensionally by a relational algebra (RA) –hence first-order– formula. In Section 3 we show by example how natural the modelling activity is for the average DB programmer, while in Section 4 we describe how precious the information carried out by the tuples in the violation set is in order to drive search.

We start from the basic version of RA [1] with operators $\{\sigma, \pi, \times, -, \cup, \cap\}$ for selection, projection, Cartesian product, set difference, union, and intersection, respectively, and use its standard positional notation (the unnamed perspective of [1]) for relation attributes ($\$1, \$2, \ldots,$ or $R.\$i$, $i \geq 1$, to denote the i-th column of R in queries involving multiple relations). Also, given a tuple τ, we denote its i-th component by $\tau[\$i]$. However, to ease the notation of the forthcoming queries, we extend the language with two new constructs: *conditional* if-then-else *term expressions* and the *replacement operator* Ψ_ξ. Term expression (if *cond* then $term_T$ else $term_F$) evaluates to $term_T$ or to $term_F$ depending on the truth value of condition *cond*. As for Ψ_ξ, when applied to a relation R, returns a new relation with each tuple τ of R changed into $\xi(\tau)$, with ξ being a function over the schema of R. Both extensions have direct counterparts in SQL.

NP-ALG is expressive enough to capture bounded integers and arithmetic, typed guessed relations, guessed functions, and guessed permutations, but syntactic sugar can be added for these features: e.g., we could force relations to be *typed* and also to model *functions*, as the following example shows.

Example 1 (Graph k-colouring). Let $\mathcal{D}$ be a DB with relations encoding a graph, in terms of sets of nodes (N, unary) and edges (E, binary, as set of pairs of nodes), and a set of k available colours (in a unary relation K with k tuples). The graph k-colouring problem, which is NP-complete for $k \geq 3$, can be expressed in (sugared) NP-ALG by the following expression:

1. Guessed relation: $Col : N \rightarrow K$, a total function.
2. $fail = \sigma_{\$1 \neq \$3 \wedge \$2 = \$4}\,(Col \times Col) \cap E$.

Guessed function Col is a relation having two *typed* columns, with entries in column 1 ranging over N and those in column 2 ranging over K. Being forced to be a total function, Col is such that any node in N occurs *exactly once* in column 1 (this is in fact very similar to a direct CSP encoding where, for each node in N, we have a CSP variable with set K as domain). The problem is satisfiable iff there exists an extension for Col such that no two different nodes ($\$1 \neq \$3$) assigned to the same colour ($\$2 = \$4$) share an edge in E. Such an extension is a solution to the problem instance.

We finally observe that forcing *all* guessed relations to be *typed total functions* between two *unary* DB relations does not affect the expressive power of the language. This is because: (i) tuples of non-unary relations can be denoted by new constants that act as *keys*; and (ii) arbitrary guessed relations can be encoded by characteristic functions. Also, *fail* can be rewritten in *disjunctive form*, as a *union* of sub-expressions. Since this variation underlies our practical language described next, from now on we rely on the following new definition:

Student(id, name, ...)	Set of students with id and other attributes
Teacher(id, name, ...)	Set of teachers
Course(id, teacher, num_lect)	Set of courses to be scheduled, with teacher and number of lectures planned
Period(id, day, week, start, end)	Set of time-periods (in which single lectures are allocated), plus info on day, start and finish times
Room(id, capacity)	Available rooms with their capacity
Enrolled(student, course)	Info on enrolment of students in the various courses
Equip(id, name)	Equipment available for teaching (e.g., VGA projector)
EquipAvail(equip, room)	Presence of equipment in the various rooms
EquipNeeded(course, equip)	Equipment needed for the various courses
TeacherUnav(teacher, period)	Unavailability constraints for teachers: lectures of their courses cannot be scheduled in these periods

Fig. 1: DB schema for the university timetabling problem.

Definition 1 (NP-Alg expression). *An* NP-Alg expression $\langle \mathbf{S}, \mathit{fail} \rangle$ *over a relational schema* $\mathbf{R}$ *is defined as:*

1. *A set* $\mathbf{S} = \{S_1, \ldots, S_n\}$ *of new binary relations, with any* $S_i : D_i \rightarrow C_i$ *forced to be a total function with domain* D_i *and co-domain* C_i*, with both these relations being unary and belonging to* $\mathbf{R}$*. We call them* guessed functions.
2. *An RA expression* $\mathit{fail} = \bigcup_{i=1}^{k} \mathit{fail}_i$ *on the vocabulary* $\mathbf{R} \cup \mathbf{S}$.

3 Modelling as Querying

We present a much simplified version of ConSQL, a non-deterministic extension of SQL, with its optimisation-free subset having the same expressive power as NP-Alg. The language, the initial version of which we proposed in [5], is a super-set of SQL: users can hence safely exploit the rich set of language features of SQL during problem modelling. The new language reduces the number of keywords added to standard SQL from 11 to just 4, in the aim of greatly easing the modelling task for the average DBA. For space reasons, we omit a formal description of the language, and introduce it by an example.

Example 2 (University timetabling). Assume a university wants to solve the following *course timetabling* problem (a variant of the ones presented in [6], in order to show the flexibility of the language), namely finding a schedule for the lectures of a set of courses, in a set of classrooms, by minimising the number of students enrolled in courses with overlapping lectures, and by satisfying constraints about lecture allocation in time and rooms (con1, con2, con6), room equipment (con3), conflicts (con7), and course lengths (con5). Also, teacher unavailability constraints (con4) make timetables non-periodic. Data reside in a relational DB with the tables listed in Figure 1 (primary keys are underlined; foreign keys are omitted). A specification for this problem is given in Figure 2. The few new keywords are all capitalised.

As can be seen, a problem specification is defined in terms of the new construct create SPECIFICATION, embedding the definition of a set of views (via the standard SQL create view construct), an optional objective function (via the new constructs MINIMIZE and MAXIMIZE applied to an arbitrary aggregate SQL query), and a set of constraints (via the standard SQL check keyword applied to an arbitrary SQL Boolean expression). Some of the views are guessed, in that they

```
create SPECIFICATION Timetabling (
  // A guessed view (see new construct CHOOSE), encoding a 'guessed' timetable, as an
  // assignment of courses to room/period pairs (with some pair possibly unassigned, i.e., null)
  create view TT as
    select p.id as p, r.id as r, CHOOSE(select id as c from Course) is null from Period p, Room r
  // A view defined in terms of TT (dependent guessed view)
  create view ConflCourses as select c1.id as c1, c2.id as c2 from Course c1, Course c2, TT t1, TT t2
    where c1.id < c2.id and t1.c = c1.id and t2.c = c2.id and t1.p = t2.p
  // Objective function, minimise the number of students enrolled in conflicting courses
  MINIMIZE select count(*) from Student s, ConflCourses cc, Enrolled e1, Enrolled e2
    where e1.student = s and e1.course = cc.c1 and e2.student = s.id and e2.course = cc.c2
  // Constraints:
  // con1. No two lectures of the same course on the same day
  check "con1" (not exists ( select * from TT t1, TT t2, Period p1, Period p2
    where t1.c = t2.c and t1.c is not null and t1.p = p1.id and t2.p = p2.id and p1.day = p2.day and t1.p != t2.p ))
  // con2. Capacity constraint, in terms of ordinary SQL view
  create view Audience as select e.course as c, count(*) as nb_stud from Enrolled e group by e.course
  check "con2" ( not exists ( select * from TT t, Room r, Audience a
    where t.r=r.id and t.c=a.course and r.capacity<a.nb_stud ))
  // con3. Equipment
  check "con3" ( not exists ( select * from TT t, EquipNeeded en where t.c=en.course and
    en.equip not in (select equip from EquipAvail ea where ea.room=t.r) ))
  // con4. Teachers unavailability
  check "con4" ( not exists ( select * from TT tt, Course c, TeacherUnav tu
    where tt.c = c.id and c.teacher = tu.teacher and tt.p = tu.period ))
  // con5. Right number of lectures for each course, in terms of helper view over TT
  create view CourseLen as select t.c, count(*) as nb_lect from TT t where t.r is not null group by t.c
  check "con5" ( not exists ( select * from CourseLen cl, Course c where cl.course=c.id and cl.nb_lect<>c.nb_lect ))
  // con6. At most 3 lectures of the same course per week
  check "con6" (3>=all( select count(*) from TT t, Period p where t.p = p.id group by t.c, p.week ))
  // con7. Courses taught by the same teacher not in the same period
  check "con7" ( not exists ( select * from TT tt1, TT tt2, Course c1, Course c2
    where c1.id < c2.id and c1.teacher = c2.teacher and tt1.p = tt2.p ))
);
```

Fig. 2: ConSQL specification for the university timetabling problem.

have special columns, called *guessed column sets*, defined by the new construct CHOOSE, which is the main enabler of the non-determinism added to SQL.

Guessed column sets play the same role as guessed functions in NP-Alg (Definition 1): the solver is asked to populate *non-deterministically* the guessed column sets of views, choosing tuples from the query argument of CHOOSE, in such a way that all *constraints* are satisfied and the optional *objective function* takes an optimal value. In the example, our goal is to assign courses to room/period pairs (guessed view TT), with some pairs possibly having no assigned course (is null),[3] in such a way that the number of students enrolled in courses with conflicting lectures is minimised and that all the constraints are satisfied. Figure 3(b) (on page 9) shows guessed view TT when the DB holds periods p1, p2, p3, and rooms r1, r2 and r3 (plus possibly others): the pure SQL part of the view definition (columns 'p' and 'r' storing all period/room pairs) is extended to the right with a guessed column set of one column, 'c', having the schema of the argument query: its values are non-deterministically picked from the set of ids of courses. Some of them can be assigned to null, allowing us to model partial functions. Values in a guessed column set may be forced to be all different by the standard SQL distinct modifier, hence modelling a particular case of the all-different global constraint.

[3] is null is a standard SQL keyword used in table definitions, stating that values of some field *can be* (rather than 'must be', as the name suggests) the special value null.

4 Local search to solve NP-Alg queries on relational DBs

Local search (LS) [9] has proved to be an extremely promising approach to solve combinatorial problems over potentially large data sets (typical of many business scenarios). Also, its intrinsic flexibility may be exploited when building systems that need to cope with dynamic and concurrent settings. Below, we restate the main notions of LS in terms of an NP-Alg expression $\langle \mathbf{S}, \mathit{fail} \rangle$, and then present novel techniques to take advantage of the relational query cost model, and to scale seamlessly toward large data sets stored in external memory.

Definition 2 (States, search space, costs, solutions). *A* state *is an extension* $\overline{\mathbf{S}}$ *for guessed functions* $\mathbf{S}$. *The* search space *is the set of all possible states. The* cost *of a state* $\overline{\mathbf{S}}$ *is* $\Sigma_{i=1}^{k} |\mathit{fail}_i|$, *i.e., the total number of tuples returned by the sub-expressions* fail_i *when evaluated on* $\overline{\mathbf{S}}$. *A* solution *is a state where* $\mathit{fail} = \emptyset$, *hence having cost 0.*

Most LS algorithms have, at their core, a *greedy inner loop*: after a possibly random initialisation, they iteratively evaluate and perform small changes (*moves*) from the current state to a *neighbour* state, in order to reduce its cost. Since the greedy loop terminates if no moves lead to a cost reduction (i.e., in the states called *local minima*), these algorithms are enhanced with more sophisticated (not purely greedy) techniques (e.g., simulated annealing or tabu-search), to continue search toward better states. Although, in general, the universe of possible moves is chosen in advance by the programmer from the structure of the problem, the simple structure of the relational model and of guessed functions in NP-Alg suggests a very natural definition for *atomic* moves, while more complex moves could be defined by composition of these atomic moves.

Definition 3 (Move). *A* move *is a pair* $\langle S, \delta \rangle$ *where* $S \in \mathbf{S}$ *(with* $S : D \rightarrow C$*) and* $\delta = \langle d, c \rangle$, *with* $d \in D$ *and* $c \in C$.

Given any state $\overline{\mathbf{S}}$ of $\mathbf{S}$ and the corresponding extension $\overline{S}$ of guessed function S, a move $\langle S, \delta \rangle$ with $\delta = \langle d, c \rangle$ changes $\overline{S}$ by replacing with c the co-domain value assigned to domain value d. We denote the extension of S after performing move $\langle S, \delta \rangle$ as $S \oplus \delta$, and the neighbour state reached with such a move as $\overline{\mathbf{S}} \oplus \langle S, \delta \rangle$. A move $\langle S, \delta \rangle$ executed on state $\overline{\mathbf{S}}$ is *improving*, *neutral*, or *worsening* iff the cost of state $\overline{\mathbf{S}} \oplus \langle S, \delta \rangle$ is, respectively, less than, equal to, or greater than the cost of $\overline{\mathbf{S}}$. Also, move δ executed on $\overline{\mathbf{S}}$ is improving, neutral, or worsening w.r.t. constraint i iff the *cost share* of state $\overline{\mathbf{S}} \oplus \langle S, \delta \rangle$ due to constraint i (i.e., $|\mathit{fail}_i|$) is, respectively, less than, equal to, or greater than that of $\overline{\mathbf{S}}$.

Greedy algorithms follow different strategies for selecting an improving move at each step: for example, *gradient descent* chooses an improving move randomly, while *steepest descent* chooses the move that *maximally* reduces the cost of the current state. Since steepest descent needs to consider *all* possible moves in order to choose the best one, it could be very inefficient on large-scale neighbourhoods. A third very successful algorithm is *min-conflicts*, which randomly selects one guessed function $S : D \rightarrow C$ and one of its domain values $d \in D$, and then chooses the best improving move among all those involving S and d.

The strength of classical LS techniques typically arises from their ability to exploit the similarities between the current state and each of its neighbours, by evaluating the cost variation of neighbours incrementally. In the next paragraph we show not only that this is possible also in our framework, but also argue that the relational model allows a second, orthogonal approach to increase efficiency, namely the exploitation of the similarities *among* the neighbours of a given state. This is done by evaluating the entire neighbourhood jointly.

Joint Incremental Neighbourhood Evaluation (JINE). Consider an NP-Alg expression $\langle \mathbf{S}, \mathit{fail} \rangle$, where $\mathit{fail} = \bigcup_{i=1}^{k} \mathit{fail}_i$, and all fail_i are *conjunctive* queries, i.e., of the form $\sigma_\phi(S_{i_1} \times \cdots \times S_{i_s} \times R_{i_1} \times \cdots \times R_{i_r})$, with all $S_{i_j} \in \mathbf{S}$ and $R_{i_p} \in \mathbf{R}$ (plus occurrences of $\cap$, which can be rewritten in terms of σ and $\times$). In ConSQL this is equivalent to saying that constraints can be expressed as not exists select-from-where queries, as is often the case (see, e.g., the graph k-colouring constraint and most of those in the university timetabling specification, while the others can be rewritten in conjunctive form by standard means). Let us now focus on a single constraint i, and on a distinguished guessed function S (binary relation encoding a function $S : D \to C$ with D and C unary relations in $\mathbf{R}$) occurring $m \geq 1$ times in the expression fail_i. To emphasise the m occurrences of S, we can w.l.o.g. rewrite fail_i as follows (after also changing references to columns in the selection condition ϕ accordingly):

$$\mathit{fail}_i = \sigma_\phi(S^{(1)} \times \cdots \times S^{(m)} \times \mathbf{T}), \tag{1}$$

with $S^{(1)}, \ldots, S^{(m)}$ being the m occurrences of S, and $\mathbf{T}$ the Cartesian product of all the relations other than S in the constraint expression. Let us also assume that, in a given state $\overline{\mathbf{S}}$, fail_i evaluates to a possibly empty set of tuples V_i, which represents the *violation set* of constraint i.

We now show that, given an arbitrary set of moves over S, encoded as tuples $\delta = \langle d, c \rangle$ in a relation $M_S \subseteq D \times C$, we can compute incrementally the violation set of constraint i in *all* the new states $\{\overline{\mathbf{S}} \oplus \langle S, \delta \rangle \mid \delta \in M_S\}$ (and as a consequence the exact cost variation for constraint i upon *each* move in M_S) by running just the following two queries:

$$V^-_{i,S} = \sigma_{\chi'}\big(M_S \times V_i\big), \qquad V^+_{i,S} = \Psi_\xi\Big(\sigma_{\chi'' \wedge \phi'}\big(M_S \times (S^{(1)} \times \cdots \times S^{(m)} \times \mathbf{T})\big)\Big),$$

where $\chi' = \bigvee_{j=1}^{m}(M_S[\$1] = V_i[\$(2j-1)])$, $\chi'' = \bigvee_{j=1}^{m}(M_S[\$1] = S^{(j)}[\$1])$, and ϕ' is obtained from ϕ by replacing any term of the form $S^{(j)}.\$2$ (with $j \in [1..m]$) into the conditional term expression (if $S^{(j)}.\$1 = M_S.\$1$ then $M_S.\$2$ else $S^{(j)}.\$2$).

As for the function of the replacement operator, here ξ, when applied to a tuple $\tau = \langle d, c, d_1, c_1, \ldots, d_m, c_m, \mathbf{t} \rangle$, produces tuple $\tau' = \langle d, c, d_1, c'_1, \ldots, d_m, c'_m, \mathbf{t} \rangle$ such that, for all $j \in [1..m]$, we have that $c'_j = c$ if $d_j = d$, and $c'_j = c_j$ otherwise.

The following result holds (proofs are omitted for space reasons):

Theorem 1. *In the state $\overline{\mathbf{S}} \oplus \langle S, \delta \rangle$ reached after performing any move $\delta \in M_S$ from state $\overline{\mathbf{S}}$, the expression fail_i for constraint i evaluates to:*

$$\Big(V_i - \underset{-M_S}{\pi}\big(\sigma_{M_S=\delta}(V^-_{i,S})\big)\Big) \;\cup\; \underset{-M_S}{\pi}\big(\sigma_{M_S=\delta}(V^+_{i,S})\big),$$

with π_{-M_S} denoting the projection operator that filters out columns of M_S in $V^-_{i,S}/V^+_{i,S}$. Also, the two arguments of $\cup$ have no tuples in common.

Audience

c	nb_stud
c1	27
c2	26
c3	37
c4	35
c5	48
c6	43
c7	67

(a)

TT

p	r	c
p1	r1	c1
p1	r2	–
p1	r3	c3
p2	r1	–
p2	r2	c6
p2	r3	c5
p3	r1	–
p3	r2	c7
p3	r3	c6
...	...	...

(b)

V_{con2}

t.p	t.r	t.c	r.id	r.capacity	a.c	a.nb_stud
p2	r2	c6	r2	40	c6	43
p3	r2	c7	r2	40	c7	67
...	...	...	...	...	...	...

(c)

Fig. 3: (a) View Audience. (b) A portion of the extension of guessed view TT in current state ('-' means null). (c) Portion of V_{con2} due to the given portion of TT.

M_{TT}

p	r	c
p2	r2	c1
p2	r2	c5
p2	r2	c7
p2	r2	–
p3	r2	c1
p3	r2	c5
p3	r2	c6
p3	r2	–

(a)

$V^{-}_{\mathsf{con2,TT}}$

p	r	c	t.p	t.r	t.c	r.id	r.cap	a.c	a.nb_stud
p2	r2	c1	p2	r2	c6	r2	40	c6	43
p2	r2	c5	p2	r2	c6	r2	40	c6	43
p2	r2	c7	p2	r2	c6	r2	40	c6	43
p2	r2	–	p2	r2	c6	r2	40	c6	43
p3	r2	c1	p3	r2	c7	r2	40	c7	67
p3	r2	c6	p3	r2	c7	r2	40	c7	67
p3	r2	–	p3	r2	c7	r2	40	c7	67

(b)

$V^{+}_{\mathsf{con2,TT}}$

p	r	c	t.p	t.r	t.c	r.id	r.cap	a.c	a.nb_stud
p2	r2	c5	p2	r2	c5	r2	40	c5	48
p2	r2	c7	p2	r2	c7	r2	40	c7	67
p3	r2	c5	p3	r2	c5	r2	40	c5	48
p3	r2	c6	p3	r2	c6	r2	40	c6	43

(c)

Fig. 4: A set M_{TT} of moves over TT (a) and result of queries $V^{-}_{\mathsf{con2,TT}}$ (b) and $V^{+}_{\mathsf{con2,TT}}$ (c) (only tuples deriving from the portion of TT shown in Figure 3(b) are given).

Theorem 1 tells us that, given any set of moves M_S over a guessed function S occurring in constraint i, the queries $V^{-}_{i,S}$ and $V^{+}_{i,S}$ correctly compute the changes in the violation set of constraint i in *all* (joint evaluation) the states reached by performing each of these moves, given the violation set in the current state (incremental evaluation). Queries $V^{-}_{i,S}$ and $V^{+}_{i,S}$ return sets of tuples, each representing a move $\delta = \langle d, c \rangle$ in M_S concatenated with a tuple that will be removed from, respectively added to V_i, in case δ is executed. Given that the two arguments of $\cup$ have no tuples in common we can compute with *one* additional query the exact cost of *all* moves in M_S over *all* constraints, by grouping and counting tuples therein. If all violation sets V_i are materialised (i.e., stored in tables), we can *incrementally maintain* them by deleting and inserting tuples returned by these queries.

Example 3 (University timetabling, continued). Consider a generic state where the extension for TT is partially given in Figure 3(b) and the capacity constraint con2 has the violation set V_{con2} of Figure 3(c). The problem instance refers to (among possibly others) periods p1–p3, rooms r1, r2, r3 with capacities of 30, 40, 50 students, respectively, and courses c1–c7. Figure 4 shows a set of moves M_{TT} over TT (part (a)) and the results of computing queries $V^{-}_{\mathsf{con2,TT}}$ (part (b)) and $V^{+}_{\mathsf{con2,TT}}$ (part (c)) from set M_{TT}. As an example, move (p2, r2, c1) (that is, assigning course c1 to room r2 in period p2) will reduce the cost of con2 by 1 (since it occurs once in $V^{-}_{\mathsf{con2,TT}}$, see Figure 4(b), and does not occur in $V^{+}_{\mathsf{con2,TT}}$, see Figure 4(c)). Analogously, move (p2, r2, c5) is neutral w.r.t. con2, because it occurs once in the results of both queries.

Violation-Directed Neighbourhood Design (VND). Classical LS algorithms use problem constraints only to evaluate the quality of a move, i.e., the variation in cost of the associated neighbour state. In our framework, given the definition of *move* (cf. Definition 3), the neighbourhood to explore would be the set of all states reachable from the current one by changing in *all* possible ways the co-domain value associated to *any* domain value of *any* guessed table. The necessary join operations may be computationally very expensive, despite the usual exploitation by the DBMS of clever query optimisation techniques. The straightforward approach of considering, for each $S \in \mathbf{S}$ (with $S : D \to C$), the set $M_S = D \times C$ quickly becomes impractical. Consider the university timetabling problem of Example 2. By Definition 3, at any step, a steepest descent algorithm should consider $|\mathsf{Period}| \cdot |\mathsf{Room}| \cdot |\mathsf{Course}|$ possible moves (each period/room pair can be reassigned to one of the remaining courses plus null), to find and choose the one that maximally reduces the cost of the next state. Even in not-so-large instances, e.g., 960 periods (8 periods/day, 5 days/week for 6 months; recall that timetables are not periodic here), 30 rooms, and 40 courses, the size of M_{TT}, which is to be used in join operations in order to compute, for each constraint i, queries $V^{-}_{i,\mathsf{TT}}$ and $V^{+}_{i,\mathsf{TT}}$ is over 10^6.

To overcome these difficulties, the concept of constraint-directed neighbourhoods has been proposed [2]. The idea is to focus on the constraints also to isolate *subsets* of the moves that possess some useful properties, e.g. those that are *improving* w.r.t. one or more constraints. The modelling paradigm of NP-Alg allows us to improve these methods by considering the content of the violation sets (not only their size, i.e., the current cost of the associated constraints): each tuple in a violation set (such a tuple being called *violation* in what follows) can be interpreted as a *reason why* the associated constraint is violated in the current state. This knowledge can be exploited to avoid wasting efforts in considering many non-improving moves during the greedy inner loop of any LS algorithm.

As an example, Figure 3 shows a state for the university timetabling problem. Tuples in the violation set of con2, V_{con2} (Figure 3(c)), besides giving information on the current cost of con2 (their overall number), give also reasons about why this constraint is not satisfied: e.g., the first tuple shows that the current timetable has the problem that course c6 (43 students) is given in period p2 in a room, r2, which is too small (40 seats); analogously for the other tuples. Hence, to reduce the cost of con2 it makes no sense to consider moves that act on period/room pairs occurring in no violations. Also, it makes no sense to consider e.g. a move that reassigns p2/r2 to another course with more than 40 students.

In general, consider an NP-Alg expression $\langle \mathbf{S}, \mathit{fail} \rangle$, where $\mathit{fail} = \bigcup_{i=1}^{k} \mathit{fail}_i$, and all fail_i are of the form (1). In order to improve the cost of the current state, greedy algorithms iteratively choose (according to different strategies) an improving move δ. However, for δ to be improving, it has to be improving for at least one constraint i, by making at least one tuple in violation set V_i disappear in the neighbour state. Now consider any constraint i and any guessed function S occurring in it, and assume that in the current state $\overline{\mathbf{S}}$ fail_i evaluates to a non-empty violation set V_i. We now define the set of moves over S that are *promising* w.r.t. constraint i.

Definition 4 (Promising moves). *The set of* promising moves *over guessed function $S : D \to C$ w.r.t. constraint i of the form* (1) *is defined by:*

$$\mathit{Prom}_i^S = \underset{D,C}{\pi} (\sigma_{\chi \wedge \neg \phi'} (D \times C \times V_i)) \tag{2}$$

where $\chi = \bigvee_{j=1}^{m}(D.\$1 = V_i.\$(2j-1))$ *and* ϕ' *is as in the definition of* $V_{i,S}^{+}$.

The set of moves that are promising w.r.t. a constraint is a correct over-approximation of the set of moves that are improving for that constraint:

Theorem 2. *In any state* $\overline{\mathbf{S}}$*, for any constraint* i*, and for any guessed function* $S \in \mathbf{S}$*, all moves over* S *that are improving w.r.t. constraint* i *belong to* $Prom_i^S$*.*

This result shows that the concept of promising moves is an important building block for the *seamless application of all LS algorithms*, since it can be used to drive automatically their inner greedy loops in populating the desired move sets M_S *before* running JINE. As an example, to run steepest descent, it is enough to explore the moves that are promising for at least one constraint (hence populating, for each $S \in \mathbf{S}$, the set M_S with the union of the $Prom_i^S$ over all constraints i over S). On the other hand, if the size of the data is too large to make steepest descent practical, simpler algorithms may be applied. As an example, we could bias the search toward improving as much as possible a most violated constraint by feeding JINE with sets of moves containing only those that are promising for it (one set M_S for each S mentioned in the selected constraint).

A few comments on the meaning of query (2) are in order. A generic tuple $\lambda \in V_i$ has the form $\langle d_1, c_1, \ldots, d_m, c_m, \mathbf{t}\rangle$, with $d_1, \ldots, d_m \in D$, $c_1, \ldots, c_m \in C$, and $\mathbf{t}$ a tuple over schema $\mathbf{T}$. $Prom_i^S$ computes moves over S (hence pairs $\langle d, c\rangle \in D \times C$, witness the use of the projection operator π) such that: (i) The domain value of the move, d, is equal to at least one among $d_1, \ldots, d_m$ of at least one tuple $\lambda \in V_i$ (selection condition χ); and (ii) For such a λ, the tuple λ', built by replacing in λ any c_j with the new value c iff $d_j = d$, does not satisfy condition ϕ (hence is not part of the violation set in the state reached after performing the move). Since in RA we cannot modify tuples when evaluating a condition, in (2) we take the alternative approach of changing the condition ϕ into ϕ'. Requirement (i) guarantees that synthesised moves involve only domain values that appear in (i.e., are responsible for) at least one tuple λ of the violation set V_i, hence eliminate at least λ. Requirement (ii) additionally ensures that, after performing each of such moves $\langle d, c\rangle$ over S, the newly introduced tuple in S (i.e., $\langle d, c\rangle$) does not introduce a new tuple λ' in the violation set by satisfying ϕ again with tuple $\mathbf{t}$.

Example 4 (University timetabling, continued). Consider the state shown in Figure 3. The query for $Prom_{\mathsf{con2}}^{\mathsf{TT}}$ (once generalised to handle domain and co-domain relations of arity greater than 1) returns the set of moves that eliminate at least one tuple in the violation set V_{con2} (Figure 3(c)) without introducing a new one by matching again the same tuples of table Room and view Audience (given in Figure 3(a)). Figure 5(a) shows part of the result of $Prom_{\mathsf{con2}}^{\mathsf{TT}}$ containing the moves synthesised from the first two violations. Any move over TT that is not in $Prom_{\mathsf{con2}}^{\mathsf{TT}}$ is guaranteed to be non-improving for con2.

If we do not use VND, then a total of $|\mathsf{Period}| \cdot |\mathsf{Room}| \cdot |\mathsf{Course}|$ moves need to be considered for evaluation at each iteration. With VND instead, at most $|\mathsf{Course}|$ moves for each violation of con2 need to be assessed (usually less, given ϕ'). Hence, *the more progressed the greedy search*, and the closer we are to a solution, *the smaller the neighbourhood* that needs to be explored.

VND can be extended to handle additional algorithms. Consider, e.g., *min-conflicts*: at any iteration this algorithm randomly selects one guessed function $S : D \rightarrow C$ and one of its domain values $d \in D$, and then chooses the best improving move among all those involving S and d. Given that for a move to be improving, it needs to delete at least one tuple in the violation set of some constraint, we can design a violation-directed (and generalised) version of min-conflicts in such a way that it first selects a random violation $\lambda \in \bigcup_{i=1}^{k} V_i$, and then considers as promising all moves that would make λ disappear in the next state. Only such moves would be used to populate M_S, which would then be evaluated with JINE.

VND+JINE can be further generalised to deal with *compound moves* of bounded size, in order to, e.g., generate swap moves to keep the values in distinct guessed column sets all-different, or to capture the core ideas of well-known techniques like, e.g., variable neighbourhood search [12, 4], by, e.g., defining layers of different neighbourhoods of increasing size and complexity, in order to bypass local optima, and limited discrepancy search [8], where assignments different from those of the initial one may be regarded as neighbours. Other aspects, like supporting the non-greedy phase of the search or constraint weights and priorities, may also be taken into account by making small changes to the queries above.

Before concluding this section, some comments on the complexity of VND and JINE are in order. For each guessed function S and constraint i: (i) $Prom_i^S$ can be computed with two joins, the first of which, given condition χ, does not lead to an explosion of the number of tuples (any tuple in V_i can match at most m tuples of D, with m being usually very small: e.g., at most 2 in the constraints of the university timetabling problem); (ii) $V_{i,S}^{-}$ involves a single join; and (iii) $V_{i,S}^{+}$ involves one more join than the expression of constraint i. Condition χ'' avoids the combinatorial explosion of the number of tuples due to the added join.

5 Implementation and Experiments

We implemented a solver based on the ideas above. The system, written in Java, takes as input a problem specification in ConSQL and uses standard SQL commands for choosing, evaluating, and performing moves. The modelling language has been extended with a declarative search strategy definition (SSD) language. Here (see Figure 5(b)), search strategies are given as LS algorithms (and parameters) such as gradient/steepest descent, tabu search, min-conflicts, simulated annealing, and their composition (via random restarts or batches) [16, 7].

The system *interacts transparently* with *any* DBMS. This implementation choice greatly increases robustness, generality, and portability: the solver is immediately usable by the average DBA on any information system (either centralised or distributed) with a standard SQL API and on data of virtually any size (since memory management issues are entirely delegated to the DBMS). On the other hand, this generality certainly introduces a bottleneck for performance, which can however be solved (at least when data is small enough to be cached in main memory) by implementing custom RAM storage engines specific to a DBMS that exploit differentiable data structures (see, e.g., [16]) optimised for the queries required by VND+JINE.

In order to measure the effectiveness of the techniques above on the scalability of LS when applied to relational data, we experimentally evaluated the

$Prom^{\mathsf{TT}}_{\mathsf{con2}}$

p	r	c
p2	r2	c1
p2	r2	...
p2	r2	c5
p2	r2	c7
p2	r2	–
p3	r2	c1
p3	r2	...
p3	r2	c6
p3	r2	–
...	...	...

(a)

```
SOLVE WITH SEQUENCE OF
   STEEPEST DESCENT
      STOP AFTER 5 IDLE ITERATIONS,
   SIMULATED ANNEALING WITH
      TEMPERATURE BETWEEN 100 AND 10
      COOL BY 0.9 EVERY 2 ITERATIONS
      STOP AFTER 10 IDLE ITERATIONS,
   TABU SEARCH WITH
      TABU TENURE BETWEEN 5 AND 10 ITERATIONS
      STOP AFTER 10 IDLE ITERATIONS
   5 TIMES
   RESTART 5 TIMES
```

(b)

Fig. 5: (a) Promising moves over TT computed starting from the displayed portion of violation set V_{con2} (Figure 3(c)). (b) An example of application of the search strategy definition language of ConSQL: the current problem is solved with a bunch of LS solvers applied in sequence 5 times, and allowing 5 random restarts.

performance gain of the current system when VND+JINE are enabled. Given the targeted novel scenario of having data modelled *independently* and stored *outside* the solving engine, and being queried by non-experts in combinatorial problem solving using *standard* DBMS APIs, our purpose is not and cannot be to compete with state-of-the-art LS solvers like the one of Comet [16]. Rather, our experiments have been addressed at seeking answers to the following questions: what is the impact of VND+JINE on: (*i*) the reduction of the size of the neighbourhood to explore; (*ii*) the overall performance gain of the greedy part of the search.

Given our objectives, it is sufficient to limit our experimentation to a simple centralised client-server architecture (both DBMS and LS solver on the same host, a computer with an Athlon64 X2 Dual Core 3800+ CPU, 4GB RAM, using MySQL DBMS v. 5.0.75, with no particular optimisations) and to focus on single greedy runs, until a local minimum is reached. Also, we can focus on *relative* (rather then absolute) times and can omit the numbers of moves performed, since VND+JINE do not affect the sequence of moves executed by the LS algorithm.

We experimented with two problems: graph colouring (a compact specification with only one guessed column set and one constraint, which gives clean information about the impact of our techniques on a per-constraint basis) and university timetabling (much more articulated and complex). Again, given the current objectives we limit our attention to instances that could be handled in a reasonable time by the currently deployed system: 17 graph coloring instances with up to 561 nodes and 6656 edges (from `mat.gsia.cmu.edu/COLOR/instances.html`) and all 21 `compX` instances of the 2007 Int'l Timetabling Competition (`www.cs.qub.ac.uk/itc2007`) for university timetabling, having up to 131 courses, 20 rooms, and 45 periods. Experiments on both problems involved two greedy algorithms: steepest descent (which requires at each step to explore the entire neighbourhood) and the violation-directed version of min-conflicts mentioned at the end of Section 4.

Results are reported in Figure 6(a). Instances have been solved running both algorithms with and without VND+JINE, from the same random seed. As expected, enabling VND+JINE considerably boosts performance: *impressive speed-ups* (of orders of magnitude) were experienced on *all* instances, especially when the entire neighbourhood needs to be evaluated (steepest descent), hence undoubtedly establishing the ability of JINE to exploit the powerful join optimisation techniques implemented in modern DBMSs. Furthermore, the experiments show that VND+JINE bring advantages also when complete knowledge on the neighbourhood is not needed (min-conflicts), although speed-ups are unsurprisingly lower. Finally, we observe (see Figure 6(b)) the *strong* expected *reductions of the size of the neighbourhood actually explored*, thanks to the action of VND, even when it is applied in the worst-case scenario (steepest descent, when the union of promising moves over all guessed functions and w.r.t. all constraints needs to be considered). In particular, VND filters out often more than 30% of the moves at the beginning of search, and *constantly more than 80%* (with very few exceptions) when close to a local minimum. This in turn reinforces JINE, by strongly reducing the size of M_S move sets involved in join operations, and thus its overall efficiency.

6 Conclusions and Future Work

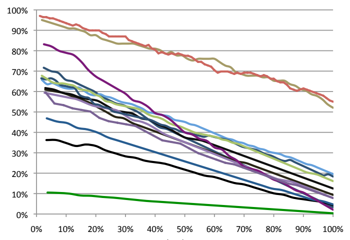
Although some attempts to provide a smoother integration between DBs and NP-hard problem solving have already been carried out, especially in the contexts of deductive database systems (see, e.g., constraint databases [10], which however focus on representing implicitly and querying a possibly *infinite* set of tuples, typically representing spatial or temporal data) and ASP (e.g., DLV$^{\mathrm{DB}}$ [14], a version of DLV which uses a DBMS and its query mechanism to mimic in external memory the computation of answer sets of a disjunctive logic program), to the best of our knowledge ConSQL is the first attempt to provide the *average DB programmer* with effective means to access combinatorial problem modelling and solving techniques when developing business applications, without the intervention of specialised programmers. Our framework appropriately behaves in dynamic and concurrent settings, seamlessly reacting to changes in the source data, thanks to the flexibility of LS coupled with the transactional and change-interception mechanisms, e.g., triggers, well supported by DBMSs. In fact, should a concurrent application make some (small) modifications to the data being processed, such change-interception mechanisms may be transparently used to synchronise the materialised data behind VND+JINE, just by considering the changes as exogenous moves that took place. This makes our approach fully respect data access policies enforced in information systems with concurrent applications: a solution to the combinatorial problem is represented as a *view* of the data, which is *dynamically kept up-to-date* w.r.t. the underlying (evolving) DB relations. Our approach fits the usual development life-cycle of commercial DBMSs, in that the implementation of custom RAM cache and storage engines optimised for the queries required by VND+JINE would allow a more efficient and lower level interaction with the DBMS.

This paper is of course only a step toward achieving full convergence between information management systems and efficient combinatorial problem solving. Thanks to some industrial interest about ConSQL, we are currently working in several directions, such as the extension of VND+JINE to constraints defined by queries with aggregates and groupings. In particular, this extension requires, in order to keep incremental evaluation possible and efficient, the maintenance of additional information, which can be considered the DB counterpart of differentiable data structures exploited in other LS solvers (e.g., [16]). Support for aggregates would also allow us to handle in more clever ways any presence of

Problem / Algorithm	Steepest-descent	Min-conflicts
Graph colouring	8x..665x (avg: 205x)	0.9x..1.6x (avg: 1.4x)
University timetabling	>15x$^{(*)}$ (avg: n/a)	3x..21x (avg: 8x)

(*) Evaluation of all instances except one without VND+JINE starved for more than 12 hours at the first iteration.

(a)



(b)

Fig. 6: (a) ConSQL speed-ups (in the number of iterations in 1 hour) when running steepest descent and min-conflicts with VND+JINE. (b) Behaviour of the ratio of the size of the neighbourhood generated by VND w.r.t. the complete neighbourhood, as a function of the state of run (0%=start, 100%=local minimum reached) for graph colouring (one line per instance whose greedy runs terminated in the time-limit).

objective functions, whose evaluation is now not handled incrementally, and to introduce more sophisticated inference mechanisms to handle distinct guessed column sets (a sort of the all-different constraint) and support for other global constraints, which proved to be precious in CP and LS [16].

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison Wesley, 1995.
[2] M. Ågren, P. Flener, and J. Pearson. Revisiting constraint-directed search. *Information and Computation*, 207(3):438–457, 2009.
[3] R. Ahuja, Ö. Ergün, J. Orlin, and A. Punnen. A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics*, 123:75–102, 2002.
[4] A. Andrew, J. Levine, and D. Long. Constraint directed variable neighbourhood search. In *Proc. of LCSC 2007 (in conj. with CP 2007)*, 2007.
[5] M. Cadoli and T. Mancini. Combining Relational Algebra, SQL, Constraint Modelling, and Local Search. *Theory and Practice of Logic Progr.*, 7(1–2):37–65, 2007.
[6] F. De Cesco, L. Di Gaspero, and A. Schaerf. Benchmarking curriculum-based course timetabling: Formulations, data formats, instances, validation, and results. In *Proc. of PATAT 2008*, 2008.
[7] L. Di Gaspero and A. Schaerf. EASYLOCAL++: An object-oriented framework for flexible design of local search algorithms. *Software – Practice and Experience*, 33(8):733–765, 2003.
[8] W. D. Harvey and M. L. Ginsberg. Limited discrepancy search. In *Proc. of IJCAI 2001*, pages 607–615, 2001. Morgan Kaufmann.
[9] H. H. Hoos and T. Stützle. *Stochastic Local Search, Foundations and Applications.* Elsevier/Morgan Kaufmann, 2004.
[10] G. M. Kuper, L. Libkin, and J. Paredaens, ed. *Constraint Databases.* Springer, 2000.
[11] D. Mitchell and E. Ternovska. A framework for representing and solving NP search problems. In *Proc. of AAAI 2005*, pages 430–435, 2005. AAAI Press/MIT Press.
[12] N. Mladenovic and P. Hansen. Variable neighborhood search. *Computers and Operations Research*, 24(11):1097–1100, 1997.
[13] C. H. Papadimitriou. *Computational Complexity.* Addison Wesley, 1994.
[14] G. Terracina, N. Leone, V. Lio, and C. Panetta. Experimenting with recursive queries in database and logic programming systems. *Theory and Practice of Logic Programming*, 8(2):129–165, 2008.
[15] P. Van Hentenryck. *The OPL Optimization Programming Language.* MIT Press, 1999.
[16] P. Van Hentenryck and L. Michel. *Constraint-Based Local Search.* MIT Press, 2005.